Information Systems 105 (2022) 101953

Contents lists available at ScienceDirect

Information Systems

journal homepage: www.elsevier.com/locate/is

EQUALITY: Quality-aware intensive analytics on the edge

Anna-Valentini Michailidou [a], Anastasios Gounaris [a,*], Moysis Symeonides [b], Demetris Trihinas [c]

[a] Aristotle University of Thessaloniki, Greece
[b] University of Cyprus, Cyprus
[c] University of Nicosia, Cyprus

ARTICLE INFO

Article history:
Received 24 June 2021
Received in revised form 13 October 2021
Accepted 10 November 2021
Available online 26 November 2021
Recommended by Lukasz Golab

Keywords:
Fog computing
Optimization
Sensors
Data quality

ABSTRACT

Our work is motivated by the fact that there is an increasing need to perform complex analytics jobs over streaming data as close to the edge devices as possible and, in parallel, it is important that data quality is considered as an optimization objective along with performance metrics. In this work, we develop a solution that trades latency for an increased fraction of incoming data, for which data quality-related measurements and operations are performed, in jobs running over geo-distributed heterogeneous and constrained resources. Our solution is hybrid: on the one hand, we perform search heuristics over locally optimal partial solutions to yield an enhanced global solution regarding task allocations; on the other hand, we employ a spring relaxation algorithm to avoid unnecessarily increased degree of partitioned parallelism. Through thorough experiments, we show that we can improve upon state-of-the-art solutions in terms of our objective function that combines latency and extent of quality checks by up to 2.56X. Moreover, we implement our solution within Apache Storm, and we perform experiments in an emulated setting. The results show that we can reduce the latency in 86.9% of the cases examined, while latency is up to 8 times lower compared to the built-in Storm scheduler, with the average latency reduction being 52.5%.

1. Introduction

In the recent years, there has been an increasing interest in the Internet of Things (IoT) paradigm, where data are being gathered from sensors and connected devices in order to provide services in a diverse set of domains like smart cities, smart homes and connected vehicles [1,2]. A common characteristic of these application domains is that they rely upon geo-distributed devices that produce a large amount of streaming data to be analyzed in order to produce useful information. According to the International Data Corporation (IDC) [3], the amount of such data being created will increase from 33 ZB in 2018 to 175 ZB by 2025. Moreover, most of these data require real time analysis. Further, the analysis is usually in the form of a complex job that consists of multiple tasks (or steps) that run sequentially or in parallel. In most cases, the data generation devices are simple sensors or wearables that either have very little or no computation capacity at all. Due to that, the need to assign the analysis job to external devices arises. A common approach is to gather these data in a central location, typically in the cloud, to analyze them. However, this can cause large delays due to the cost of transferring data through the network [4–6]. A more recent and promising solution is to analyze data closer to where they are produced, in line with the edge and fog computing vision.

Fog and edge computing differ in that the former assumes that the processing nodes may be close to the data sources (contrary to the cloud nodes) but not necessarily at the same site as the data sources, while the latter assumes that, apart from the cloud nodes, only the elements at the data sources locations are capable of processing the data on the fly [7]. In our setting, complex analysis job tasks can be assigned to the edge devices, to the fog devices, which are more powerful machines close to the sensors or to the remote even more powerful cloud nodes. However, in this work we use the terms fog and edge computing nodes or devices, interchangeably. Each device is characterized by its computation and storage capacities. These devices communicate via links in order to execute jobs in co-operation. The challenge is to devise novel techniques to assign streaming analytics job tasks to the devices available in a way that optimizes several Quality of Service (QoS) metrics, including latency, WAN consumption and data quality.

In addition, the quality of the data is an important aspect in IoT scenarios. Low quality can lead to less useful results. The

* Corresponding author.
E-mail addresses: annavalen@csd.auth.gr (A.-V. Michailidou), gounaria@csd.auth.gr (A. Gounaris), symeonidis.moysis@cs.ucy.ac.cy (M. Symeonides), trihinas.d@unic.ac.cy (D. Trihinas).

https://doi.org/10.1016/j.is.2021.101953
0306-4379/

quality of data can be categorized into multiple dimensions. Some examples of these are *completeness, timeliness* and *accuracy*. Some of the most common factors that lead to decreased data quality include the heterogeneity of data sources, missing and dirty data due to network malfunctions or security constraints [8]. Consider the scenario where a user wants to analyze the output of a sensor every 5 min. If, for some reason, the sensor malfunctions or restarts, the output will be inaccurate or misleading and the user should ignore it. Thus there is a need to "rate" the quality of the data and/or to detect outliers and missing values that could lead to misleading results. However, this data quality rating can be very time consuming especially when dealing with streaming big data that need to be processed under tight latency constraints.

The main novelty of our work is that it considers data quality as a first-class citizen and optimizes for both performance and quality; more specifically, we solve a problem that trades latency for an increased fraction of incoming data, for which data quality measurements are performed. Apart from this, our work is tailored to data-intensive analytics, in which each job step is partitioned into multiple disjoint tasks, e.g., as is common in Apache Spark Streaming, Storm and Flink. In addition, resource constraints both in terms of the computational and memory constraints and in the eligibility to perform a task are considered. Although some of these aspects have been examined in isolation, e.g., [9–11], to the best of our knowledge, this is the first work that considers data quality, partitioned parallelism and constrained resources across heterogeneous geo-distributed devices in combination.

As a motivation example about the trade-off between latency and performance of quality checks, we conducted an experiment on a single machine running Apache Storm.[1] In this experiment, a continuous stream is received that consists of tuples with 10 numeric attributes. The quality check is a completeness one, i.e., it verifies that all attributes are complete and we can configure the percentage of incoming tuples that are checked. Fig. 1 shows the execution and process latency (the latter also includes the acknowledgment time) of the check on each tuple of the stream. As shown in the figure, the check can become a serious overhead or even a bottleneck in case of small inter-arrival times and/or when few resources are available. The quality check in this example, while not being very compute-intensive, still incurs an overhead implying that more advanced quality checking techniques, such as streaming outlier detection [12], will result in much higher overheads per incoming tuple and/or more resources consumed. Another example is data imputation, e.g. using the techniques in [13], where missing data may be filled before they are processed by operators downstream. Quality checks need not refer exclusively to quality related measurements but they can cover arbitrary quality-related operators. Overall, our work is motivated by the fact that it is important to design algorithms that optimize the level at which data quality checks can be performed considering the overhead they impose. To this end, we limit the fraction of the data for which data quality checks are performed in a systematic manner and regardless of what exactly these checks are. Our aim is any quality checks to be as extended as possible without contributing to the latency, for which the dominant factor should remain the communication cost; we enforce computation cost to be dominated by communication cost implicitly, through setting appropriate resource constraints on the devices.

More specifically, we consider a bi-objective problem that targets both latency and quality, while the decision variables are the workload allocation to each device available. We refer to a setting, where it makes sense quality-related operators to be applied before proceeding to further data analysis (e.g., as in the data imputation example). Since the considered problem is intractable, we follow a hybrid approach, which comprises two pillars. Firstly, we solve a relaxed linear programming formula and further enhance it using local search heuristics propagating their impact to the complete execution plan. Secondly, to avoid unnecessarily and inefficiently employing multiple remote resources, we employ a spring relaxation algorithm that is characterized by limited degree of partitioned parallelism. Apart from this algorithmic contribution, we extend Apache Storm in order to support our results, which entails that we define the exact portion of the workload that a bolt instance will receive on each distinct site. We thoroughly evaluate our solution by both deploying a real-world prototype on a fog computing emulator and performing extensive simulations. The results show that we can yield improvements regarding our objective function that combines latency and extent of quality checks up to 2.56X compared to state-of-the-art solutions. In addition, our framework, in 86,9% of the cases, yields a latency drop of at up to 8X when compared to the current (or default) Storm scheduler. Further experiments show how we deviate from the optimal solution in very small analytics jobs, for which the computation of the exact solution is feasible. Our solution along with the accompanying system is termed as *EQUALITY*.

The rest of the paper is organized as follows: First we provide the problem statement in Section 2. In Section 3 we present our proposed solutions for the operator placement problem. In the next section, we discuss the Storm prototype implementation issues. We evaluate our solution in Section 5. We continue with some more generic discussion in Section 6. In Section 7, we present the related work. Finally, we conclude in Section 8.

Table 1
Notations used in the paper.

Symbol	Meaning
G_{op}	Graph representing the overall analytics job
V_{op}	Vertices of G_{op} (operators or tasks)
E_{op}	Edges of G_{op} (data shuffling)
ED	Edge devices
s_i	Selectivity of $i \in V_{op}$
$RCPU_i$	CPU requirement of $i \in V_{op}$
$RRAM_i$	RAM requirement of $i \in V_{op}$
$CCPU_u$	CPU capacity of $u \in ED$
$CRAM_u$	RAM capacity of $u \in ED$
$available_{i,u}$	Availability of $u \in ED$ for $i \in V_{op}$
$comCost_{u,v}$	Communication cost between $u, v \in ED$
$ED_i \subseteq ED$	Subset of edge devices $i \in V_{op}$ can be assigned to
$x_{i,u}$	Fraction of $i \in V_{op}$ assigned on $u \in ED$
$RCPU_{DQ}$	CPU requirement of data quality check
$RRAM_{DQ}$	RAM requirement of data quality check
$DQ_{fraction}$	Fraction of tuples checked for data quality
$x_{DQ,u}$	Fraction of data quality check assigned on $u \in ED$
m	Number of (logical) data sources in G_{op}
α	Network congestion factor
β	Weight denoting the importance of data quality
$enabledLinks_{i,j}$	Number of devices that participate in $(i,j) \in E_{op}$

2. Notation and problem statement

In this section, we describe the system model and present the problem formulation. Informally, the problem we target is as follows: *how to efficiently place the tasks (i.e., the operator partitions) to the edge and cloud devices to minimize the latency and maximize the level at which data quality checks are performed.*

2.1. System model

Table 1 presents the main notation used. Specifically, we represent a streaming analytics job as a Directed Acyclic Graph (DAG) $G_{op} = (V_{op}, E_{op})$ where each node in V_{op} represents an

[1] https://storm.apache.org/.

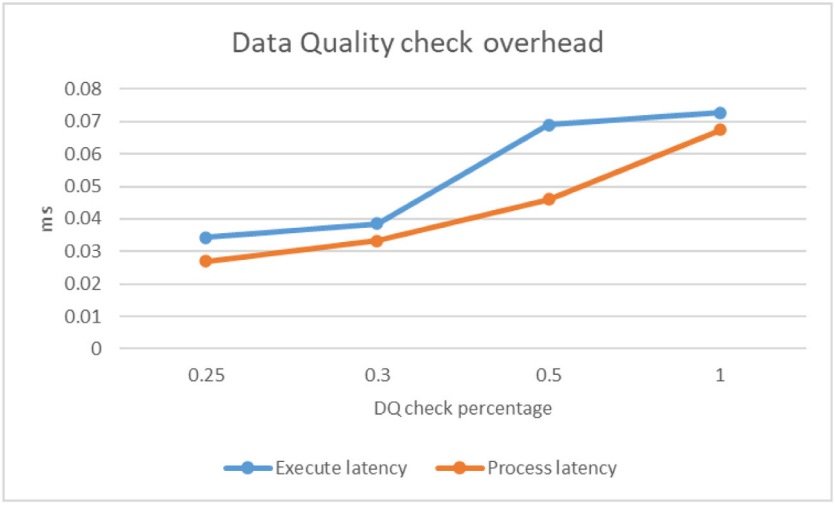

Fig. 1. Overhead incurred by a data quality check on Apache Storm in the latency per incoming tuple.

operator and the edges in E_{op} represent the data flow between operators. An operator represents a set of analysis job steps at a fine level of granularity that can run on the same device (typically in a pipelined manner). Each operator can be physically partitioned into multiple instances, where each instance is responsible for a disjoint data partition, similarly to the partitioned parallelism paradigm in databases [14] or the way Spark RDDs are processed in parallel [15]. Edges represent data re-distribution among the partitioned operator instances. For example, according to the above description, in Apache Spark, operators correspond to Spark job stages while edges are placed when data shuffling takes place. The data sources are the operators in the G_{op} without incoming edges, and we assume that they produce data in batches periodically.

Each operator takes as input multiple tuples (i.e., records) in batches that need to be analyzed and also has a certain selectivity s_i according to its functionality. For example, a transformation operator with selectivity $s_i = 1$ outputs a tuple for each tuple it takes as input, while a filtering operator with $s_i = 0.5$ outputs a tuple for every two input tuples on average. If operator i is a sink node in the graph, then, there is no selectivity, whereas, if it is a source node, its selectivity is equal to 1. Also, each operator has its own CPU and RAM requirements (e.g., when a windowed state needs to be maintained).

Each edge device belonging to ED is characterized by its RAM capacity in MB as well as by its CPU capacity in GHz. For example, an edge device with a CPU capacity of 1.9 GHz can perform 1.9×10^9 clock cycles per second. Each pair of edge devices has a different communication cost, $comCost_{u,v}$ $(u, v \in ED)$, which relates to the connected devices network speed; the latter is also dependent on the physical distance and link capacity/bandwidth between the devices.

Each operator can be assigned to multiple edge devices that run in parallel to benefit from partitioned parallelism, as is common in frameworks such as MapReduce, Spark, Flink and Storm. That means that each device u is assigned a fraction $x_{i,u}$ of tuples of an operator i to analyze. However, due to privacy, security and other reasons, there exists a subset of edge devices where an operator can be assigned to (denoted by the flag $available_{i,u}$ for operator i regarding device u). We define as $EDi \subset ED$ the subset of edge devices operator i can be assigned to. In addition to the operators that perform analysis, there is also a special operator that runs data quality checks and its activation is optional and configurable. Such an operator can potentially contain multiple quality check techniques and algorithms.

The placement problem we deal with in this work is the assignment of fractions $x_{i,u}$ of operators i in G_{op} to edge devices u in ED so that performance and the fraction of input data for which a data quality check is performed are maximized. As will be detailed next in our cost model, the performance objective is the latency in time units, where the communication cost dominates, in line with the most common reasoning in intensive analytics even in a cluster setting at a single geographical place (Section 2 in [16]). For simplicity, we assume that a data quality check can be performed only on each data source; in other words, the source devices are eligible to perform the quality checks locally so that these quality checks precede any further analysis. We will relax this assumption later explaining that allowing data quality checks to be placed on downstream devices has negligible impact on our solution. The variable $DQ_{fraction}$ denotes the percentage of the input data on which the quality check is applied and is equal to $\frac{1}{m}\sum_{u \epsilon ED_i} x_{DQ,u} x_{source,u}$ where here, ED_i are the devices that produce data. Please note that we treat the quality check as a black box and do not deal with the actual implementation as this is out of the scope of this work; instead we aim at judiciously defining the percentage of data these checks will be applied to.

The data quality check also has its RAM and CPU requirements as described previously; these requirements combined with the device capacities ensure that, if met, the device resources are not saturated and the latency bottleneck remains the communication cost. E.g., in the example of Fig. 1, the capacities should be set in such a manner that the incurred latency is an order of magnitude lower than the communication costs. Overall, the decision variables are $x_{i,u}$ and $x_{DQ,u}$. The problem is formalized next.

2.2. Cost model and problem formulation

We formulate the operator placement problem as a non-linear programming one, of the form shown below.

$$min\, F(x, DQ_{fraction}) \tag{1}$$

$$s.t. \sum_{u \epsilon ED_i} x_{i,u} = 1, \forall i \epsilon V_{op} \tag{2}$$

$$\frac{1}{m}\sum_{u \epsilon ED_i} x_{DQ,u} x_{source,u} = DQ_{fraction} \tag{3}$$

$$x_{i,u}, x_{DQ,u}, DQ_{fraction} \in [0, 1] \tag{4}$$

$$\sum_{i \epsilon V_{op}} (RCPU_i * x_{i,u}) + RCPU_{DQ} * x_{DQ,u} * x_{source,u} \leq CCPU_u, \forall u \epsilon ED_i \tag{5}$$

$$\sum_{i \epsilon V_{op}} (RRAM_i * x_{i,u}) + RRAM_{DQ} * x_{DQ,u} * x_{source,u} \leq CRAM_u, \forall u \epsilon ED_i \quad (6)$$

Eq. (2) defines that the sum of the operator's fractions, which are divided across the devices must be equal to 1, so that there is no data loss or data replication. Eq. (3) refers to the data quality checks and does not impose its fractions to sum to 1. Eq. (4) states that an operator and data quality check can be divided into fractions, each of which can be assigned to the same or to different devices and also bounds the $DQ_{fraction}$ variable, not allowing negative values or values greater than 1. Eqs. (5), (6) ensure that the RAM and CPU requirements of an operator i and data quality check are fulfilled and the capacities of the edge devices are not exceeded.

As already stated, a QoS metric we try to minimize is the average latency. This latency is equal to the latency of the critical path (i.e., the slowest path) with regards to a single input data batch and consists of the average communication latency between the operators in the critical path; according to the way it is formulated, the latency defines the higher frequency at which the end results can be updated.[2] As our focus is on geo-distributed realms, we make the realistic assumption that the execution latency of each operator is negligible and the communication cost dominates. The total latency of a tuple is the time to flow from its source node downstream to a sink node.

The communication latency between two nodes is expressed by $max\{x_{i,u} * s_i * \sum_{v \in ED_j} comCost_{u,v} * x_{j,v}\}$, $u \in ED_i$ across all instances of operator i. This is equal to the slowest data transfer that comes from a single device and refers to the cost of a batch of input data. However, this modeling is over-simplistic because it does not take into account the overhead of an operator instance maintaining multiple remote connections. In order to take into account such an overhead, we introduce a parameter, notated as α multiplied by the number of enabled links. A link between device u and device v for two operators i, j is enabled when $x_{i,u} \neq 0$, $x_{j,v} \neq 0$ and $u \neq v$. Thus, $enabledLinks_{i,j}$ denotes the number of devices that exchange data between two operators over the network. In our experiments, we set α as a function of average communication costs and based on the results of real runs, we can claim that such an approach is effective.

The total latency of the topology is equal to the latency of the critical path, that is the slowest path of the graph that leads from a source node to a sink one (not including). We denote as *path* any path from a source to an operator just upstream a sink operator (j is the operator just after i).

$$Latency = max_{path \in G_{op}} \{ \sum_{i \in V_{op} \in path} max_{u \in ED_i} \{x_{i,u} * s_i * \sum_{v \in ED_j} (comCost_{u,v} * x_{j,v}) + \alpha * enabledLinks_{i,j}\}\} \quad (7)$$

$$F = \frac{Latency}{1 + \beta * (DQ_{fraction})}, \ \beta \geq 0 \quad (8)$$

The latency represents the average time a tuple needs to come out of a sink node, beginning from a source one. The latency is increased with more data quality checks but the relationship is not proportional. This is due to the fact that the more the quality checks, the less an edge device can be assigned tasks of downstream operators, thus inducing higher communication cost. We explain in our experiments that this difference, i.e., employing more devices for downstream operators because source devices are more occupied with data quality checks, has a significant impact on the final assignment. The configuration parameter β is a weight denoting the importance of data quality and its exact value depends on the application. The higher its value, the more beneficial (in terms of minimizing F) becomes to increase $DQ_{fraction}$. Setting β to 0 essentially removes DQ from the optimization criteria.

The problem formulation above is NP-hard. We omit details here, given also that in [9,17], even a simpler formulation without data quality checks and partitioned parallelism is shown to be NP-hard. Further, from the task placement point of view, the configurable quality check introduces a new variable type that corresponds to partial assignment of an operator to devices.

[2] In order for the critical path's latency to be representative of the whole graph's latency, we assume there is no waiting time between the tasks of the operators. This is achieved through setting the operator requirements at such levels, so that they sustain the input data production rate. A natural consequence of this is that each operator can start its execution as soon as it receives its input batch of tuples from its parent nodes in the graph.

Table 2
Communication cost between edge devices in GBps.

Device	0	1	2
0	0	1.5	2
1	1.5	0	1
2	2	1	0

Table 3
Fraction of operator assigned to each edge device.

Operator/device	0	1	2
0	0.8	0.2	0.0
1	0.7	0.0	0.3
2	0.3	0.4	0.3

2.3. Example and motivation

Lets assume we have a simple linear DAG with 3 nodes (operators) and 3 devices. The selectivity of each node is the following: $s_0 = 1, s_1 = 1.5$ (s_2 does not have any impact). The communication costs between devices are presented in Table 2. For simplicity we set α equal to 0.

We assign the operators as shown in Table 3. The transfer time for each link in the graph depends on the selectivity of that operator, the communication cost of the devices and the assigned fractions. Each device sends data to every other device based on the fraction of the current and the succeeding operator. For example, for operator 0, device 0 will take $0.8 \times 1 \times 0 \times 0.7$ sec/unit to send data to device 0 (itself), $0.8 \times 1 \times 1.5 \times 0$ sec/unit to device 1 and $0.8 \times 1 \times 2 \times 0.3$ sec/unit to device 2. We sum these cost elements to derive the total cost of communication between operator 0 and operator 1 for the first device with id 0; this sum is 0.48. For device 1, we have $0.2 \times 1 \times (1.5 \times 0.7 + 0 \times 0 + 1 \times 0.3) = 0.27$ sec/unit. For device 2, the communication cost is 0. Therefore, the latency due to the link $0 \rightarrow 1$ is $max\{0.48, 0.27, 0\} = 0.48$. Similarly, we can derive that the cost between $1 \rightarrow 2$ is $max\{1.26, 0, 0.45\} = 1.26$ sec/unit. Therefore, the overall latency is 1.74 sec/unit.

If we assume that the $DQ_{fraction}$ is 0.5 then with $\beta = 1$, the objective function F becomes 1.16. Further assume that we examine another scenario, where $DQ_{fraction}$ is 1 at the expense of moving the complete fraction $x_{2,0}$ to device 2, i.e., the last operator runs 40% on device 1 and 60% on device 2. Then, the latency cost of $1 \rightarrow 2$ becomes $max\{1.89, 0, 0.18\} = 1.89$ and the complete latency is 2.37. The new value of F is 1.185, i.e., despite the important increase in latency, the new plan is not better in terms of F. However, if we give even more weight to the data quality through increasing the β value, e.g., setting $\beta = 2$, then the initial and modified allocation yield F values of 0.87 and 0.79,

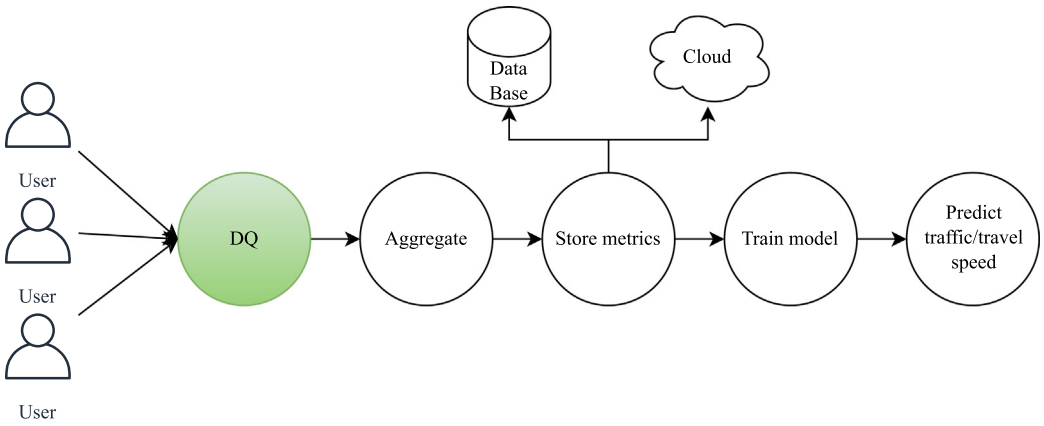

Fig. 2. Example DAG with DQ node.

respectively, i.e., the trade-off in the modification has become beneficial.

A real-world example where DQ measurement is crucial is in a live navigation system that takes as input data from GPS trackers or other road sensors and predicts the future traffic and travel time. This data includes information like speed, road segment, time and direction. Then, the data are aggregated per road segment for each time interval (e.g. 5 min) in order to compute statistics like the average speed. These statistics can be stored in databases or the cloud. The data are then used to train models that make the predictions. It is important to find the quality of the data, as faulty data can lead to misleading predictions. Fig. 2 shows the DAG of the above example, where the first node represents the data quality measurement. If we want to check the quality of an input tuple we configure the first node appropriately; otherwise we disable it. This loosely-coupled approach ensures that our DQ measurement technique can be easily added to any application DAG and also, the removal of DQ-related checks does not affect the application functionality.

Algorithm 1 Initial assignment of data processing and quality checks fractions using an LP solver

```
1: procedure DQASSIGNMENT
2:     sort operators in topological order
3:     minimize Eq. (7) for each edge
4:     for each source source do
5:         for each device dev do
6:             if x_{source,dev} ≠ 0 then
7:                 a ← (CCPU_dev − usedCPU_dev) / (RCPU_DQ * x_{source,dev})
8:                 b ← (CRAM_dev − usedRAM_dev) / (RRAM_DQ * x_{source,dev})
9:                 x_{DQ,dev} ← min{a, b, 1}
10:                DQ_fraction ← DQ_fraction + (1/m) * x_{DQ,dev} * x_{source,dev}
11:            end if
12:        end for
13:    end for
14: end procedure
```

3. Solution

Since the placement problem described in Section 2.2 is an NP-hard one as proved in [17], we tackle it in two different ways that, due to their low computational complexity, are then combined to form a final hybrid solution:

1. We solve a relaxed version of the Non-Linear Programming Formula. More specifically we solve a Linear Programming (LP) formula for each operator considering the placement of its parent nodes and then further optimize it heuristically.
2. The previous solution may employ numerous devices unnecessarily and easily fall into local optima. Thus, we additionally use a spring relaxation algorithm that produces a solution with low or no intra-operator parallelism.

These two techniques are not seen as competing to each other. After running both techniques, we choose the best one. Next we present details about each of the solutions described above.

3.1. Optimization per edge

Let us first assume that the sources are fixed and we will relax this assumption shortly. The solution starts by finding an assignment of operators to devices solving the latency formula in Section 2.2 for each node of the graph independently, i.e., in Eq. (7) the latency is examined for a single edge rather than a complete path. To place the tasks of an operator, we take into consideration the placement of its parents in the graph. Thus, the operators are examined in topological order and the problem of finding the task assignments of the next operator downstream is transformed to a simpler linear-programming (LP) one. This approach is a local one but can still prevent data from being moved from a device to another one where the communication cost is high. Instead, data are favored to remain locally or moved to devices where it is faster to send them.

Then, we insert DQ-related tasks. Equivalently, the solution up to this point has assigned values to $x_{i,u}$ variables assuming that all $x_{DQ,u}$ variables are set to 0. However, the capacity of the source devices may not be filled. Thus we calculate the maximum DQ fraction that can be assigned to each device that takes on a fraction of the sources. This DQ fraction is found by using all the available capacity of the devices and considering the CPU and RAM DQ check requirements, as shown in Algorithm 1. This yields a complete initial solution.

This initial solution can be deemed as the state-of-the-art approach to optimizing the placement per node while accounting for partitioned parallelism [11]. Nervertheless, although the operator assignment is optimal for each node in terms of latency, it is not the optimal solution for the whole graph even if there is no DQ optimization criterion. To improve the placement found, we repeatedly run a local search algorithm, inspired by our previous work [18]. We have implemented two flavors. The first one, called *latOpt*, is described in Algorithm 2. It iterates through a loop where for each edge of the graph (pair of nodes), it first detects its bottleneck device, i.e., the one that takes the longest to transfer

Algorithm 2 latOpt: Moving bottleneck processing tasks

```
for each operator pair (op1, op2) do
    slowest ← findSlowestDevice(op1)
    if (x_{op1,slowest} ≠ 1) then
        removeFraction(op1, slowest, ζ)
        divideFractionToActiveDevices(op1, ζ)
        tempF ← applyChangesDownstream()
        if (tempF < currentF) then
            keepTheChanges()
        end if
    end if
end for
```

Algorithm 3 qualOpt: Moving processing tasks to make space for more DQ ones

```
for each source s do
    for each device dev do
        if (x_{s,dev} ≠ 0) and ((usedCPU_dev ≥ CPU_threshold or usedRAM_dev ≥ RAM_threshold)) then
            for each operator op not in sources do
                if (x_{op,dev} ≠ 1 and x_{op,dev} ≠ 0) then
                    removeFraction(op, dev, ζ)
                    divideFractionToActiveDevices(op, ζ)
                    tempF ← applyChangesDownstream()
                    if (tempF < F) then
                        keepTheChanges()
                    end if
                end if
            end for
        end if
    end for
end for
```

its data and thus determines the communication latency. Then, the solution removes a fraction ζ of the tasks from that device and divides it to the other available devices already employed considering the RAM and CPU capacities; if the capacities are not exceeded, the amount of new workload that other enabled devices receive is equal. If, due to resource constraints, a device cannot accept further workload, this excess workload is kept on the initial bottleneck machine. Then, the algorithm finds a new placement for the nodes downstream of the operator at the start of the edge, using the same LP technique employed for the initial assignment. By doing so, the technique propagates the changes downstream to the graph and inspects their effect to the total communication latency, DQ fraction and F. This new re-assignment is accepted only if it improves the F metric. In the experiments, ζ is set to 30%.

The second flavor, termed *qualOpt*, is described in Algorithm 3 and tries to optimize the DQ part of the F parameter. Its rationale is to remove workload from the devices that are also sources to allow for more DQ checks. Specifically, the algorithm tries to free resources from the devices that have reached a threshold (set to 90% in our algorithm) of their CPU or RAM capacity. This is achieved by removing a fraction of the analysis tasks these devices take on (from all the operators expect the sources that are fixed). After each operator's re-configuration, the changes are transferred to the whole graph as previously, and the remainder capacity is filled with DQ tasks to the largest possible extent.

Both Algorithms 2 and 3 can run multiple times in any order. In the experiments, the number of iterations of Algorithms 2 and 3 over the initial placement is 10 and Algorithm 3 takes as input the output solution of Algorithm 2.

Algorithm 4 Spring Relaxation Algorithm

```
for each operator op do
    while ||N⃗|| > N_threshold do
        N⃗ ← 0⃗
        for each parent p do
            N⃗ ← N⃗ + (Position⃗_op − Position⃗_p) ∗ s_p
        end for
        for each child c do
            N⃗ ← N⃗ + (Position⃗_op − Position⃗_c) ∗ s_op
        end for
        Position⃗_op ← Position⃗_op + N⃗ ∗ δ
    end while
end for
```

Finally, we relax our assumption regarding fixed sources. If there is flexibility in choosing between alternative sources, then we need to add an initial step to choose among these alternatives. To this end, simple heuristics can be employed. Examples include using all or a predefined number of source devices and distribute the source tasks either uniformly or inversely proportionally to the average communication cost from the sources to the devices that are eligible for the children operators.

3.2. Plan optimization with limited partitioned parallelism

Complementarily to the previous solution, we employ another technique, which is more conservative in the number of devices it employs in the sense that it uses, as its starting point, an execution plan without partitioned parallelism and even when subsequently the degree of partitioned parallelism increases, it stays limited. The rationale is to decrease communication cost through decreasing the number of employed devices. This solution is inspired by the work of Pietzuch et al. [19] and, in its core, it leverages spring relaxation. We start by mapping the devices to a three-dimensional space where the euclidean distance between two devices is equal to the inverse of their communication cost; two devices will be close in this space if they have low communication cost. The number of dimensions is configurable and can be altered. The mapping of the devices is decided using the Vivaldi algorithm [20].[3] Then the operators, except the sources that are fixed, are randomly mapped to three-dimensional points and are given as input to the spring relaxation algorithm described in Algorithm 4.

We start, as previously, with the sources being fixed, as typically occurs in practice and as can be enforced via the availability information. Then, for the rest of the operators, the algorithm finds the force $\vec{N}$ its parents and children create on it. Each parent/child tries to "pull" this operator closer like they are connected with a stretched spring that is then released. The force, denoted as $\vec{N}$, is affected by the distance of the operators as well as the operator's selectivity. After that, the position of the operator is changed accordingly to the total force $\vec{N}$ and a constant δ. This is repeated until the magnitude of the force is less than a defined threshold $N_{threshold}$. Similarly to [19], we set δ equal to 0.1 and the $N_{threshold}$ equal to 1.

After the algorithm comes up with the new positions for the operators we map them to the closest devices using the recursive procedure described in Algorithm 5. The algorithm takes as input (i) the operator, the placement of which is performed, (ii) a number k indicating that the kth closest device to the initial position will be examined during this procedure call, and (iii) a

[3] A python implementation is available at https://github.com/pekko/vivaldi.

Algorithm 5 Operator placement to devices

```
1: procedure PLACEMENT(k, f, op)
2:     c ← findKthClosestDevice(op, n)
3:     if (RCPU_op ≤ (CCPU_c − usedCPU_c) and RRAM_op ≤ (CRAM_c − usedRAM_c) and available_op,c == 1) then
4:         assign(op, c, f)
5:         return()
6:     else if available_op,c == 0 then
7:         Placement(k + 1, f, op)
8:     else
9:         ramDiff ← (RRAM_op*f − (CRAM_c − usedRAM_c)) / (RRAM_op*f)
10:        cpuDiff ← (RCPU_op*f − (CCPU_c − usedCPU_c)) / (RCPU_op*f)
11:        if (ramDiff > cpuDiff) then
12:            assign(op, c, f − ramDiff * f)
13:            Placement(k + 1, ramDiff * f, op)
14:        else
15:            assign(op, c, f − cpuDiff * f)
16:            Placement(k + 1, cpuDiff * f, op)
17:        end if
18:    end if
19: end procedure
```

fraction f of tasks, which is set to 1 when the procedure is first called for a given operator. If the requirements of the operator can be handled by the kth closest device and if the device is available, we place the operator to that device and the procedure ends. Otherwise, if the algorithm continues due to the CPU or the RAM capacities/requirements, we calculate the largest fraction that can be assigned to that device and we call the same algorithm for the same operator but with an updated fraction and $k + 1$ as parameters. If the kth closest device is unavailable, the fraction for the new call does not change. The procedure is recursive and runs until all the tasks of an operator are assigned to a device. Operators are examined in a topological order.

To further optimize this solution, we apply a heuristic on top of it. This heuristic bears many similarities with the one in Algorithm 2. Specifically, the algorithm finds the bottleneck operator and removes a fraction of tasks from the slowest device (in many cases this device holds 100% of the tasks) and assigns them to its closest device in the three-dimensional space. As previously, if that device cannot handle the tasks, these are assigned to the next closest one. In case no better solution comes from this algorithm, we keep the initial solution.

4. Proof-of-concept implementation in apache storm

Our solution is, in principle, compatible with current stream processing engines, such as Spark, Flink and Storm. As a proof-of-concept implementation of EQUALITY, and to showcase the practicality and performance gains of our solution in real world edge computing scenarios, we use Apache Storm 2.1.0. Storm topologies consist of two components; spouts and bolts with the former representing the source nodes and the latter encompassing the other operators. The parallelization in Storm is achieved by dividing the components in tasks. We create a custom scheduler by implementing the `IScheduler` interface to assign tasks (which are mapped to executors) to specific worker nodes. There are two alternatives. The first one is to create a very large number of equal tasks for each operator, at the order of hundreds, and assign them to devices proportionally to the placement decisions. This incurs a very high overhead. The second alternative that we have chosen is to create as many tasks as the enabled devices, and assign tuples to them through using the so-called *direct grouping* built-in functionality.[4] For example, if our algorithms suggest an operator workload assignment of {20%, 70%, 10%} over three devices respectively, we create three tasks; the first one receives 20% of the tuples, the second one 70% and so on. We achieve this by sending each tuple to a task with probability equal to the fraction our algorithms decided; a tuple has 20% probability to end up in the first task etc.

In summary, Nimbus (aka the Storm master) receives the compiled (jar) file that implements the new scheduler. Then, in the topology source code, each producer node, either a spout or a bolt, emits output tuples to specific tasks proportionally to the outcome of the optimization algorithms. Then, due to the new scheduler, each task goes to a predefined worker node, where each worker node runs on a distinct device. To achieve this, we restrict each topology node to correspond to a single executor on each distinct device.

To evaluate the efficacy of our algorithms, while also ensuring result reproducibility, we use the open-source Fogify emulator [21]. Fogify eases the modeling of complex fog topologies consisting of heterogeneous resources, network capabilities and QoS criteria, while also handling the deployment and runtime monitoring of the topology over an emulated geo-distributed setting.

We provide our algorithms implemented in Python, as well as our custom scheduler and Storm topologies in a github repository at https://github.com/annavalentina/Equality.

5. Evaluation

The evaluation includes two parts, namely simulations and runs in an emulated geo-distributed setting over real hardware using Fogify [21]. The first part thoroughly investigates the merits of our algorithmic solutions and includes sensitivity analysis. The second part provides evidence that the simulated results can indeed model real runs and that our solutions improve upon the default Apache Storm scheduler.

5.1. Simulation setting and main results

Here, we compare our solutions against the baseline that distributes evenly the workload across the available devices and we also discuss how our complete solutions improve upon simply resorting to a LP solver applied edge-by-edge (as in Algorithm 1), which is the rationale in [11,23]; i.e., the comparison against the simple LP solution is essentially the comparison against the state-of-the-art. Finally, we show the benefits from the hybrid nature of our solution.

The DAGs used cover a broad range of scenarios and are shown in Fig. 3 (taken from [22]) and in Fig. 4 (taken from [9]). The former have always 10 non-source operators. The DAGs in Fig. 4 consist of three types: (i) Replicated, (ii) Sequential and (iii) Diamond. For each of these three DAG types, n is the number of the operators overall. Initially, n=5. As explained in [9,22], these DAG types cover a particularly broad range of real-world complex analytics jobs.

The number of devices is equal to 5 times the number of operators of the running topology. The communication cost of the devices is set between 0.1 and 10 covering a range of network infrastructures speeds that may differ by up to two orders of magnitude; such heterogeneity is common in fog environments. The selectivity of the operators is in the range of [0.5,2] following the real-world evidence in [23]. The sources always have selectivity equal to 1. The CPU capacities and operator/DQ requirements are between 1 GHz and 10 GHz and the RAM

[4] https://storm.apache.org/releases/current/Concepts.html.

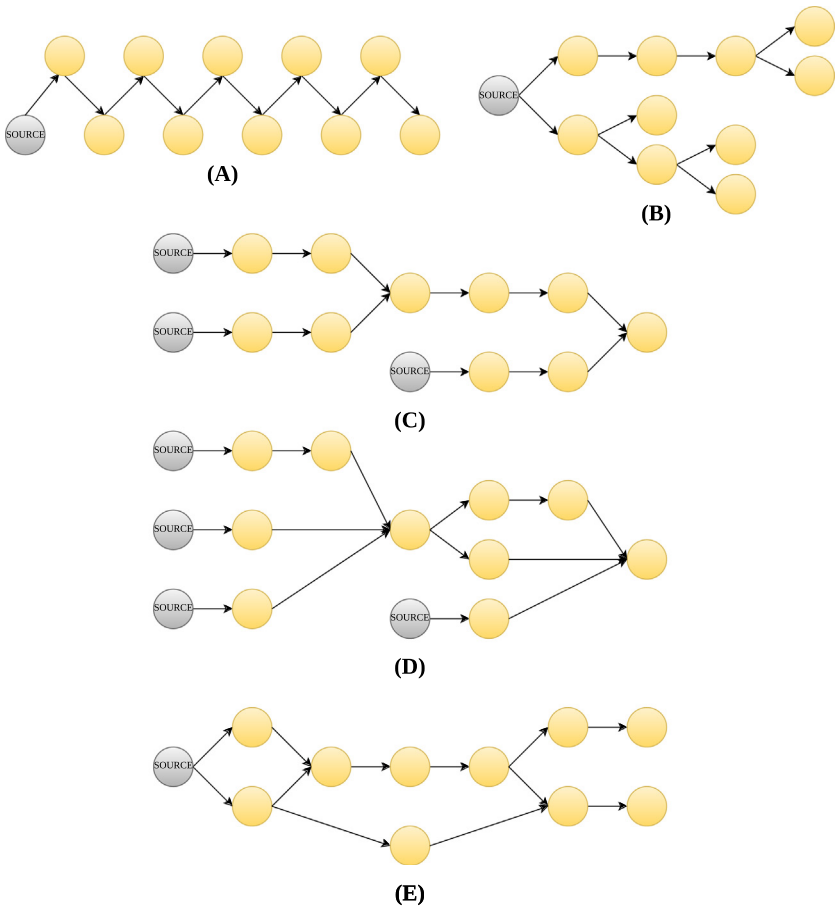

Fig. 3. DAGs used in experiments taken from [22].

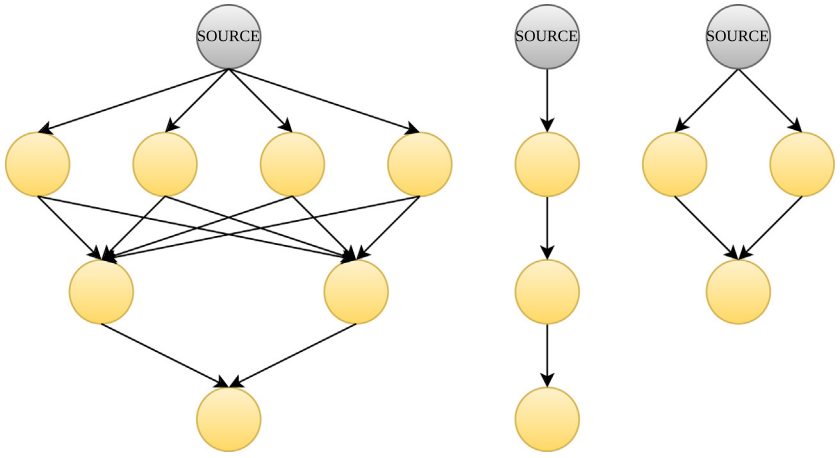

Fig. 4. Replicated, Sequential and Diamond DAGs used in experiments taken from [9].

capacities/requirements range between 100MB and 1000MB uniformly; again, such a heterogeneity is anticipated in fog settings. For each operator, approximately 85% of the devices are available. Also, 1/3 of the devices take on source nodes. The α parameter is set to $\{\overline{comCost}/10\}$ or $\{\overline{comCost}/5\}$. Each experiment is executed 50 times.

Tables 4 and 5 present detailed results for the two values of α examined and $\beta = 1$. The results are normalized so that 1 is the value of the solution that uniformly distributed the workload to all available devices. Both mean and median values are provided. The minimum (for latency and F) and maximum (for DQ) average values are in bold. The main observations are summarized as follows:

(i) The improvements in F over the uniform solution are more important when α is greater, i.e., there is a higher penalty when all devices are employed that is not outweighed by the smaller amount of data each device receives. When $\alpha = \overline{comCost}/5$, the decrease in F can reach 22 times, while it is less than 16.4 times for $\alpha = \overline{comCost}/10$.

(ii) There is no significant difference between mean and median values.

Table 4
Normalized average and median F, DQ and latency values for $\alpha = \overline{comCost}/10$.

DAG	LP avg	Spring avg	latOpt avg	qualOpt avg	LP med	Spring med	latOpt med	qualOpt med
				Normalized F values				
A	0.089	0.238	**0.061**	**0.061**	0.084	0.241	0.060	0.059
B	0.093	0.226	**0.071**	**0.071**	0.084	0.226	0.064	0.064
C	0.106	0.200	0.075	**0.074**	0.101	0.200	0.063	0.063
D	0.088	0.176	0.066	**0.065**	0.084	0.172	0.063	0.062
E	0.121	0.248	0.089	**0.088**	0.115	0.239	0.085	0.085
Replicated	0.296	0.454	0.259	**0.258**	0.270	0.435	0.227	0.227
Diamond	0.348	0.441	0.308	**0.307**	0.337	0.436	0.268	0.268
Sequential	0.213	0.411	0.177	**0.176**	0.178	0.407	0.163	0.163
				Normalized DQ values				
A	**1.043**	0.862	1.040	**1.043**	1.004	0.858	1.011	1.018
B	1.103	0.842	1.104	**1.113**	1.060	0.822	1.072	1.093
C	**1.046**	0.879	1.039	1.040	1.020	0.877	1.012	1.017
D	**1.065**	0.910	1.055	1.056	1.068	0.902	1.054	1.054
E	1.063	0.870	1.082	**1.087**	1.051	0.867	1.074	1.076
Replicated	1.164	0.943	1.165	**1.181**	1.056	0.893	1.056	1.085
Diamond	**1.169**	0.936	1.149	1.153	1.128	0.958	1.109	1.114
Sequential	**1.122**	0.963	1.118	1.119	1.080	0.951	1.095	1.095
				Normalized latency values				
A	0.090	0.221	**0.061**	**0.061**	0.084	0.218	0.060	0.060
B	0.097	0.208	**0.075**	**0.075**	0.091	0.207	0.064	0.063
C	0.108	0.188	**0.076**	**0.076**	0.106	0.185	0.066	0.066
D	0.090	0.169	**0.067**	**0.067**	0.085	0.165	0.064	0.064
E	0.124	0.232	**0.092**	**0.092**	0.118	0.232	0.088	0.088
Replicated	0.306	0.432	**0.267**	**0.267**	0.265	0.430	0.241	0.241
Diamond	0.367	0.422	**0.322**	0.323	0.345	0.411	0.290	0.290
Sequential	0.220	0.401	**0.182**	**0.182**	0.200	0.396	0.160	0.160

Table 5
Normalized average and median F, DQ and latency values for $\alpha = \overline{comCost}/5$.

DAG	LP avg	Spring avg	latOpt avg	qualOpt avg	LP med	Spring med	latOpt med	qualOpt med
				Normalized F values				
A	0.059	0.152	**0.045**	**0.045**	0.055	0.148	0.044	0.043
B	0.062	0.135	0.050	**0.049**	0.061	0.138	0.045	0.045
C	0.053	0.124	**0.044**	**0.044**	0.048	0.123	0.043	0.043
D	0.058	0.113	**0.048**	**0.048**	0.056	0.113	0.044	0.044
E	0.070	0.154	0.060	**0.059**	0.067	0.147	0.057	0.056
Replicated	0.149	0.298	0.142	**0.141**	0.143	0.294	0.127	0.127
Diamond	0.214	0.291	**0.199**	**0.199**	0.200	0.279	0.171	0.171
Sequential	0.142	0.293	0.121	**0.120**	0.141	0.287	0.117	0.117
				Normalized DQ values				
A	1.045	0.863	1.045	**1.055**	1.047	0.884	1.034	1.038
B	1.138	0.942	1.137	**1.148**	1.084	0.904	1.084	1.100
C	1.044	0.894	1.048	**1.050**	1.050	0.887	1.059	1.060
D	**1.080**	0.915	1.069	1.070	1.082	0.914	1.064	1.067
E	1.054	0.869	1.051	**1.067**	1.038	0.871	1.033	1.054
Replicated	1.555	1.041	1.550	**1.564**	1.115	0.946	1.146	1.150
Diamond	**1.212**	0.982	1.196	1.204	1.123	0.936	1.133	1.133
Sequential	**1.177**	0.968	1.151	1.171	1.093	0.968	1.079	1.092
				Normalized latency values				
A	0.059	0.142	**0.046**	**0.046**	0.056	0.141	0.045	0.045
B	0.066	0.130	**0.052**	**0.052**	0.063	0.132	0.049	0.049
C	0.054	0.118	**0.045**	**0.045**	0.050	0.117	0.045	0.045
D	0.060	0.108	**0.049**	**0.049**	0.058	0.107	0.045	0.045
E	0.072	0.143	**0.061**	**0.061**	0.068	0.141	0.058	0.057
Replicated	0.157	0.286	**0.150**	**0.150**	0.151	0.274	0.136	0.136
Diamond	0.225	0.283	**0.209**	**0.209**	0.218	0.275	0.190	0.192
Sequential	0.149	0.284	**0.126**	0.127	0.148	0.282	0.120	0.121

(iii) *qualOpt* is slightly better than *latOpt* on average. In all cases, a better solution in terms of F, which is our main metric, is found by either one or both our proposals compared to simple LP (i.e., the state-of-the-art); the highest average difference is 36% for DAG type E. Moreover, both proposals, on average, significantly improve upon the spring relaxation technique that is conservative in the number of devices employed for each operator.

(iv) DQ values are almost always higher than 1, meaning that the uniform workload allocation performs less DQ checks. In other words, our algorithms manage to increase DQ checks and decrease latency at the same time.

(v) There is a significant difference in the behavior between the DAGs in Fig. 3 and those in Fig. 4. The benefits for the latter are lower. This is mainly due to the lower number of edges in their critical paths; Replicated and Diamond have only 3 and 2 edges, respectively, regardless the number of operators, whereas Sequential has 4 edges when n=5, which is much lower than the number of edges in the DAGs of Fig. 3. This corresponds to less opportunities for optimization.

Next, we turn our attention to two important questions that have arisen from the third observation above. The first one is *"Can our complete solutions yield significant benefits compared to the state-of-the-art, represented by the simple LP-based Algorithm 1?"* To answer this question, we examine all runs individually, as shown in Fig. 5 for the DAGs in Fig. 4, where runs are sorted by their improvement factor. In more than half of the runs there is an

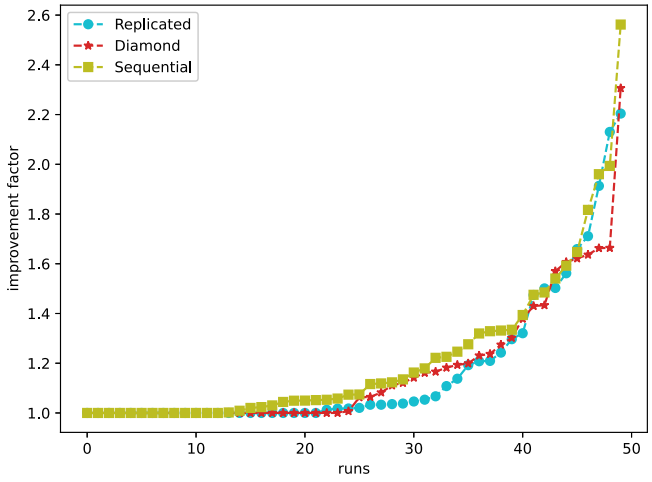

Fig. 5. Improvement factor of our complete solution over simple LP ($\alpha = \overline{comCost}/10, \beta = 1$) (Y axis - 1 equals to no improvement).

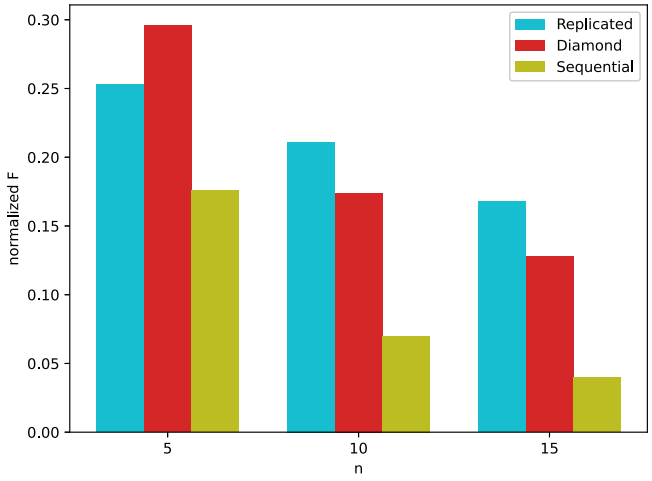

Fig. 6. Mean normalized F for the DAGs in Fig. 4 for varying n ($\alpha = \overline{comCost}/10, \beta = 1$).

improvement, whereas in the Sequential DAG type this improvement occurs in more than 2/3 of the runs. The improvements for this DAG type can be up to 2.56X. For the other two types, the maximum improvement factor observed is 2.2X and 2.31X, respectively.

The second question is: *"Are there any benefits from the spring relaxation technique?"* On average, spring relaxation is inferior, but in fact, dominates the other solutions in 5.9% of the cases. When it dominates, the mean improvement factor over *latOpt* and *qualOpt* is 40.1%, while the highest improvement observed is 3.64X. Therefore, the spring relaxation component does play an important role in the efficiency of the hybrid solution.

5.1.1. Scalability and sensitivity analysis

In the next set of experiments, we perform further scalability and sensitivity analysis tests. First, we examine the behavior with increasing number of operators. Please note that when referring to F values, we provide the minimum F value our algorithms achieved in each run. This is due to our solution being hybrid, meaning that we run *latOpt*, *qualOpt* and *Spring Relaxation* and we keep the best outcome among the three intermediate allocations. The results in Fig. 6 show that our solution yields higher relative performance with increasing DAG size, i.e., the more complex the analytics job the higher the benefits from adopting our proposal. This is also an outcome of the critical path's length as discussed previously.

Table 6
Mean normalized F values for different β values for $\alpha = \overline{comCost}/10$, $n = 5$.

DAG	0.5	1	1.5	2	3
A	0.061	0.061	0.061	0.061	**0.059**
B	**0.069**	0.071	0.074	0.079	0.081
C	0.082	0.088	0.076	0.078	**0.071**
D	0.083	0.087	0.091	0.086	**0.079**
E	0.084	0.088	**0.082**	**0.082**	0.090
Replicated	0.237	0.253	0.241	0.241	**0.234**
Diamond	0.283	0.296	**0.278**	0.290	0.302
Sequential	**0.160**	0.176	0.163	0.163	0.176

Table 7
Mean normalized F values for different threshold values in qualOpt for $\alpha = \overline{comCost}/10$, $n = 5$, $\beta = 1$.

DAG	0.75	0.8	0.85	0.9	0.95
Replicated	0.244	0.240	0.231	0.253	**0.213**
Diamond	**0.270**	0.271	0.301	0.296	0.282
Sequential	0.182	0.179	**0.164**	0.176	0.166

Second, we assess the impact of β, varying it from 0.5 to 3 as shown in Table 6. No clear conclusion can be drawn, but the behavior of our proposal is rather insensitive to β. This indicates that our algorithms can find a trade-off between DQ check and latency while keeping the overall objective function F minimized.

Next, we examine the impact of the threshold in the *qualOpt* algorithm. In most of our experiments the threshold of each device is set to 90% of the total RAM and CPU capacity. However, we also examine the values 0.75, 0.8, 0.85 and 0.95. From the results, presented in Table 7 we conclude that there is not a clear winner but a value between 0.85 and 0.95 is preferable because this range includes either the minimum value or a value very close to the minimum.

Our last experiment in this section is motivated by the fact that finding the communication cost between devices in a real edge computing environment can be a challenge and the observed values may differ from the ones that are fed to our optimization module. Thus, in the next experiment we assess the behavior of our solutions when there is inaccuracy in the initial communication cost metadata. More specifically, we run our algorithms and find an optimized task placement and its corresponding F and latency values. Then, we perturb the communication costs by randomly increasing/decreasing them in the ranges (i) $\pm$10% and (ii) $\pm$50%. Finally, we calculate the real F and latency values due to allocations based on the perturbed data. Fig. 7 presents the difference between the estimated values calculated using the original communication costs and the values calculated using the perturbed metadata. In the figure, we show 50 iterations using the Diamond DAG. We have repeated this experiment for the Sequential and the Replicated DAGs and the results were similar (no detailed results are presented). The average relative error when the actual communication costs differ in the range $\pm$10% is 0.025 whereas in the second case where the costs differ in the range $\pm$50% the error becomes 0.107. This shows that even when the actual network links exhibit very high deviance from the simulated ones, the estimation error of our algorithms is on average small, e.g., 10%. Thus, we can claim that our solution is robust to metadata inaccuracies.

5.1.2. Optimization time

The running times for the optimization algorithms were measured on a machine with the following specs: AMD FX-6300 six-core CPU @ 3.5 GHz, 8 GB of RAM. The median times are presented in Table 8. For *latOpt* and *qualOpt*, the overhead is

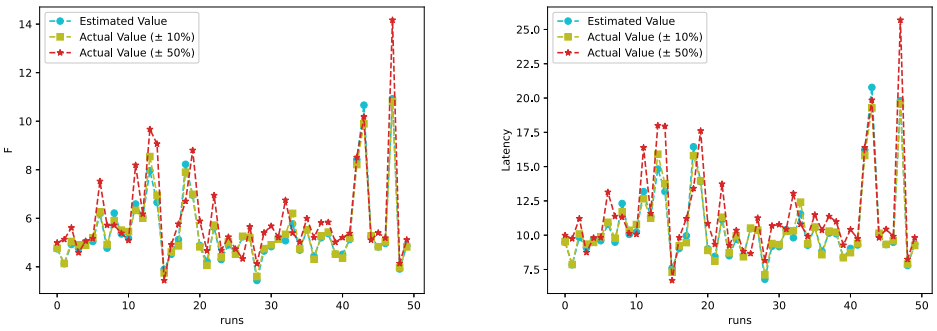

Fig. 7. Estimated and actual F and latency values considering the communication cost inaccuracy for the Diamond DAG ($\alpha = \overline{comCost}/10$, $n = 10$, $\beta = 1$).

Table 8
Time overheads for $n = 10$ (in secs).

DAG	LP	Spring	latOpt	qualOpt
A	0.65	1.31	4.70	0.59
B	0.72	1.30	1.06	0.05
C	1.34	1.58	3.08	2.27
D	1.84	1.71	3.69	3.27
E	0.81	1.31	2.05	0.73
Replicated	0.66	0.98	2.07	0.01
Diamond	1.83	1.53	1.12	0.01
Sequential	0.58	1.34	3.43	0.60

directly affected by the number of iterations, which is kept fixed to 10 in all our experiments. Also, in these experiments, the machines are always 5 times the number of DAG operators.

We conduct another experiment and we consider the number of devices to be 10*n*. Then, for the Sequential DAG, for n = 5, 10, and 15 the optimization median times for *latOpt*, become 0.12, 5.72 and 38.19 secs, respectively. For the simple *LP* solution, the corresponding times are 0.26, 2.09 and 6.61 secs, respectively. Please note that in smaller graphs the chance for latOpt and qualOpt to find a better solution decreases and in some cases the *applyChangesDownstream()* function may not be called making the runtime of the algorithms lower than the LP one.

These experiments prove that our solutions are suitable for a real-time streaming scenario since they do not incur high overhead relatively, given that DAGs may correspond to either long-running or even continuous analyses.

5.1.3. Convergence to the optimal solution

In order to investigate the convergence of our algorithms to the optimal solution regarding latency, we conducted another experiment. In a small sequential DAG with n=3 and 15 devices, we find the optimal solution that minimizes the transfer time through solving the corresponding Quadratic Programming problem; i.e., in this flow, we jointly optimize the placement of both operators in a single QP, rather than stage-by-stage. Then we compare the relative *F*, *DQ* and latency values of optimal solution in terms of latency with our algorithms. The results are presented in Fig. 8. The lower the value, the less our algorithms deviate from the optimal solution. On average, in this DAG, our algorithms achieve 34.2% higher latency than the optimal, while the maximum observed deviation is 2.51X. In 40% of the runs, the deviation does not exceed 10%. In general, the relative differences are smaller for the *F* criterion. Moreover, in 44% of the runs, our algorithms yield a higher DQ value. This experiment provides insights into the capability of our techniques to find a close to optimal solution.

5.1.4. Moving data quality check away from the sources

In this section we relax the assumption that data quality check can take place only in source nodes. Since the data quality check

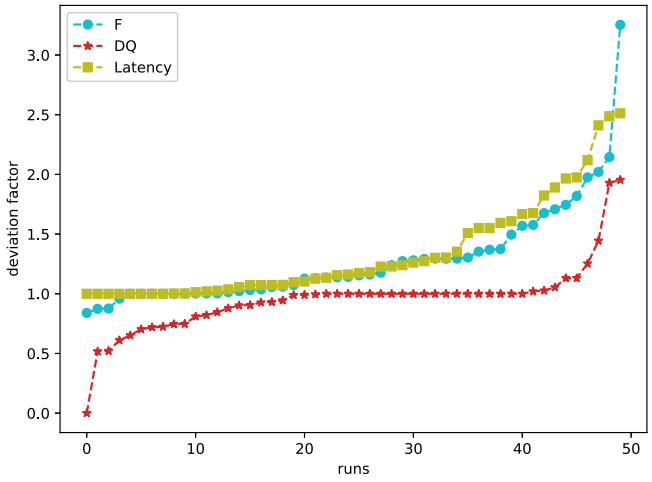

Fig. 8. Convergence to the optimal solution for a sequential DAG with n=3 ($\alpha = \overline{comCost}/10$, $\beta = 1$) (Y axis - 1 equals to no deviation from the optimal, whereas values lower than 1 denote that our techniques are superior to the one that finds the optimal latency).

needs to be completed before the actual analysis of data, which will transform or reduce them, the check can only be assigned either to the sources or to the nodes right downstream (the children nodes in the DAG). We implemented and experimented with an additional data quality check assignment. More specifically, the experiments compared the following two assignment types against each other:

1. The data quality check is assigned only on the devices that act as data sources. This is the main technique used throughout the paper.
2. In case the resources of the devices of a source node are saturated, the devices that are responsible for (part of) the source's children nodes are assigned the quality check, if this assignment results in a higher data quality fraction.

The results showed no difference between the two alternatives because, even in the second case, in almost all cases, the most beneficial assignment was to place data quality checks on source devices rather than using the devices running the children nodes. This is mainly due to the nature of our technique, which implicitly aims to employ as few devices as possible when assigning tasks in order to incur less data transfers. Thus, the devices that take on the children nodes end up being more saturated than the source devices and not able to take on the quality check. Therefore, the approach of assigning data quality check only on source devices is simpler and, given that the differences from more complicated solutions are negligible, does not compromise the performance. Based on this evidence, the fact that we place data quality checks

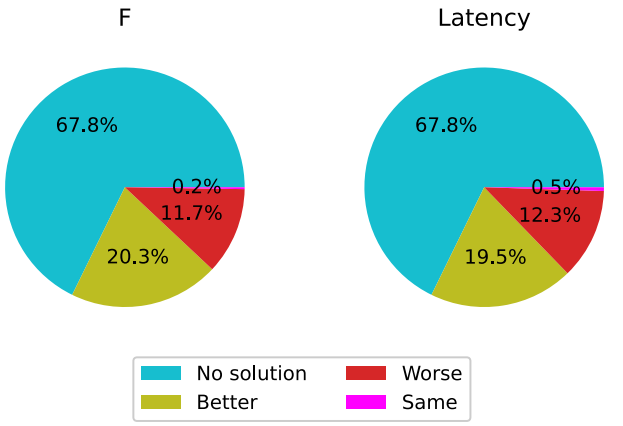

Fig. 9. Performance of the no-parallelism ILP solution compared to ours (400 runs).

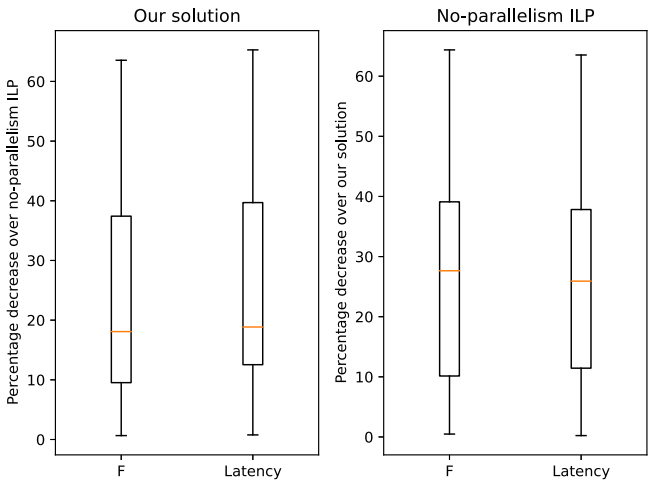

Fig. 10. Relative improvements of each solution when no-parallelism ILP finds an allocation.

only on data sources should not be interpreted as a limitation, but as a simplification that does not impact on the final task assignment result.

5.1.5. Comparison against a non-parallel task placement solution

In order to further examine if techniques with no operator parallelism are suitable to our scenario, we conducted the following experiment. We compare our solution with a method, denoted as no-parallelism ILP (Integer Linear Programming) that finds the optimal operator placement of the whole graph, but without parallelizing operators, a widely used approach in state-of-the-art papers [9,24]. This technique assigns each operator to only one device by solving the ILP once for each graph. The experiment was performed for all the DAGs and was repeated 50 times for each one of them, resulting to 400 runs. The results presented in Fig. 9 provide useful insights into the performance of the no-parallelism ILP solution. As can be seen, in most cases, the technique was not able to find a solution, i.e., to detect a single device for each operator meeting the constraints. In general, no-parallelism ILP achieves a better outcome only in 20% of the runs. Fig. 10 analyzes the decrease over no-parallelism ILP in the cases where our solution yields a better outcome and vice-versa. From the figures we can observe that in case no-parallelism ILP finds a better solution, the Interquartile Range falls between 11% and 39% with a median value of 27%, while in the cases our solution is better, the Interquartile Range also falls between 11% and 39% with a median value of 18%. More importantly, although it can be claimed that our solution is superior based on the data above, we do not treat no-parallelism ILP solutions as competitors. Given that we advocate a hybrid solution, any such method can be leveraged to provide an initial assignment and then perform modifications similarly to the spring relaxation case. An immediate consequence of this is that all results presented previously can be slightly improved through modifying our approach to choose the best among the LP-based, spring relaxation and non-parallel-ILP-based solutions (when the latter manages to yield a solution).

5.2. Prototype experiments

Using the custom scheduler to enforce the optimization decisions and the Fogify emulator described in Section 4, we conducted experiments to prove the efficiency of our algorithms in a real environment even when we are interested in latency exclusively. We performed experiments for three types of DAGs, shown in Fig. 11. The first one is a simple chain DAG, the second DAG includes a binary logical operator and the third is a DAG with two sources that are joined. We ran our algorithms with 5 random combinations of selectivities and communication costs for 10 devices using the same procedure as previously to derive the random values, and came up with optimized placement decisions. However, each source is fixed and placed in only one of the devices. Then we emulate the characteristics of the network and we run the topologies using both original Storm and a Storm flavor that encapsulates the custom scheduler to enforce a predefined task placement on the devices. In each experiment, we compare our placement decision with the equal distribution of the tasks across all 10 devices (this is the uniform approach, where each node is equally distributed across all devices) and also with the default Storm scheduler. We repeated the 5 experiments for each DAG 5 times each (in total 75 runs) and calculated the median latency (in msecs).

Table 9
Estimated and actual percentage reduction in latency over equal task distribution.

Exp. no.	Sequential		Diamond		Two-sources	
	Est.	Actual	Est.	Actual	Est.	Actual
1	48.54	95.18	25.85	87.72	4.88	34.74
2	39.14	89.14	38.52	92.88	44.07	67.30
3	48.08	90.66	36.46	79.72	38.73	47.45
4	48.94	91.74	33.37	90.17	42.76	65.75
5	40.95	45.36	38.96	58.70	25.67	16.57

The results are presented in Fig. 12. The first observation is that our algorithms outperform the equal distribution of tasks in all cases; the improvements reach 95.2% of reduction (i.e., approximately 20X faster) with an average reduction of 70%. The second observation is that in 13 out of the 15 experimental settings (86.7% of the cases), our solutions were faster than the default Storm scheduler that does not necessarily split the workload evenly across available executor nodes. More specifically, in the 13 out of the 15 cases that our solution is superior, it is on average more than 2X faster; the average reduction in latency is 52.5%. In one Diamond setting, this reduction is 87.9%, i.e., we have achieved a performance improvement of latency by a factor of 8X. On the other hand, in the two cases that the default Storm scheduler was faster, this was by 39.9% and 7.2%, respectively.

Finally, in Table 9 we compare the estimated percentage reduction that our algorithms achieved over the equal distribution with the actual reduction. As shown, in most cases, the reduction estimated internally by our cost model was lower than the actual results. Based on our overall findings, we can claim that our

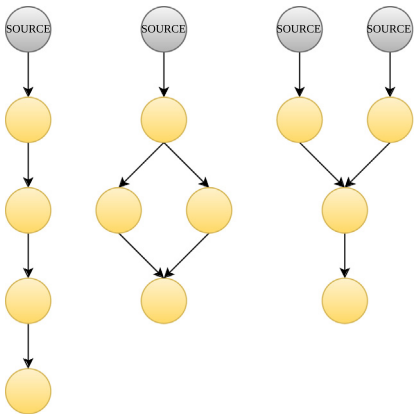

Fig. 11. Sequential, Diamond and Two-sources DAGs used in prototype experiments.

optimizations are robust and are capable of yielding performance improvements over simpler solutions like equal distribution of tasks as well as state-of-the-art frameworks like Apache Storm, in heterogeneous and geo-distributed environments even when the user is interested in latency solely. This is attributed to the fact that our solution considers inter-device communication costs explicitly.

6. Discussion

In the previous sections, we have presented the algorithmic and system part of EQUALITY along with thorough experimental results. Our novelty is not only in considering quality checks during task allocation, but also in dealing with a new task allocation problem per se, where aspects such as massive parallelism, partial execution of an operator and complex analytics jobs are combined. However, a broader vision may cover additional dimensions like those presented below:

(1) Running data quality checks is important in its own right; however, a more complete solution should also address the problem of trading resource usage and latency for increased effectiveness of the quality checks, e.g., increased recall values regarding the anomalies detected. Another direction is to allow quality checks to be performed after merging multiple input data streams and not on each source individually, before any other type of data processing. These issues are out of the scope of this work but they are important.

(2) Fog environments are subject to frequent changes in terms of the nodes available and the communication costs. Therefore, a practical task allocation solution should also account for adaptivity (and elasticity) and fault tolerance aspects. Modern platforms, such as Storm, are fault tolerant by design. However, devising adaptive flavors of EQUALITY raises new challenges, especially when the operators are stateful. In parallel with investigating adaptivity issues, a complete solution should also account for (i) cases, where the computation costs have a tangible impact on latency and (ii) acquiring CPU and RAM requirements in a practical manner.

Finally, we clarify that our work can be easily adapted to stateful operators, where the fraction of workload allocated does not refer to input tuples but to keys, e.g., as is the case in partitioned group-by. Also, we can easily cover cases, where data sources differ in aspects such as validity of data produced and trustworthiness. This can be achieved as follows. When computing the $DQ_{fraction}$ variable, instead of regarding all sources as of equal weight, we can use a formula, where each source's quality check ratio is weighted by a factor indicating the validity/trustworthiness of that source.

7. Related work

We present related work on four main research fields: (A) Geo-Distributed Data Analytics, (B) Edge Computing, (C) Data Quality and (D) Apache Storm scheduler proposals. Each of these fields is examined in turn.

Geo-Distributed Data Analytics. Edge Computing inherently falls within the geo-distributed research field. The main similarities include the representation of the jobs as DAGs, the massive parallelism of its nodes' execution and the geo-distribution of the executor nodes. When designing an edge computing placement algorithm, we can benefit from previous works that deal with geo-distributed environments. One of the differences is the type of data that needs to be analyzed. Edge analytics refer to streaming data whereas the works described below deal mostly with batch processing. Also in an edge computing setting, the requirements for latency, security and placement are usually stricter than those in cluster/cloud/data center-based analytics. Finally, the heterogeneity of executor nodes in computing capacities and network bandwidth is higher.

The most notable proposals on geo-distributed analytics include the proposal of Flutter [24], which focuses on the placement of tasks closer to the data-centers, where the data are generated and optimizes each stage of the graph independently. WANalytics [4] also optimizes each node of the graph separately using a heuristic mechanism with a view to minimizing the network usage. WhiteWater [25] focuses on maximizing the throughput by setting the order and placement of operators using hill-climbing algorithms. Viswanathan et al. [26] deal with the scheduling and placement of the queries' tasks in a WAN-aware manner in order to minimize the response time. The accompanying query optimizer, called Clarinet, chooses the best execution plan using a modified shortest job first algorithm. Li et al. [27] solve a Mixed Integer Linear Programming formula for task placement in order to minimize the WAN traffic. Xiao et al. [28] balance bandwidth, storage, latency, computing and migration cost by formulating the problem into a joint stochastic integer nonlinear optimization one. Gu et al. [29] propose a communication cost model and place operators by solving a relaxed MILP formulation. Pietzuch et al. [19] describe a multidimensional cost space and place operators using a spring relaxation algorithm to minimize the network usage while keeping the delay low.

Nardelli et al. [9] propose heuristics to determine the placement of streaming analytics jobs while considering heterogeinity in network and computation resources as well as different quality of service metrics. Similar to Nardelli et al.'s [9] system model, Niu et al. [30] also formulate the task placement problem as a bi-partitioned graph matching (tasks to machines). However the machine computation and memory capacities are not heterogeneous. Tetrium [11] on the other hand considers heterogeneous geo-distributed clusters but does not deal with streaming data. In our previous works, [18,31], we have built upon Iridium [23], a system that minimizes the query response time. In those works, we proposed algorithms that jointly optimized both latency and network traffic and we also optimized the whole graph (and not each stage independently). However the heterogeneity of the machines referred only to network speeds and the data we dealt with was not streaming.

The present work can be deemed as an extension of the aforementioned geo-distributed data analytics proposals, where not

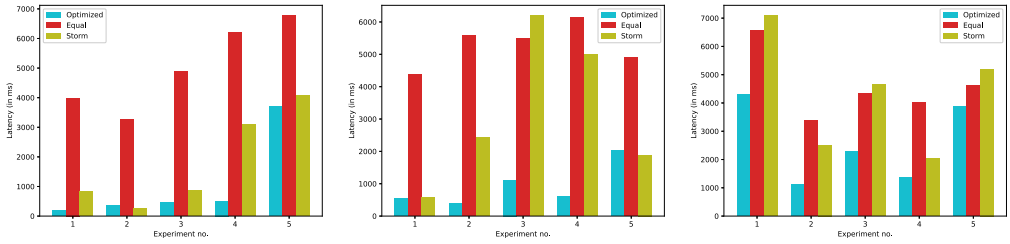

Fig. 12. Prototype experiments results for the Sequential (left), Diamond (middle) and Two-sources (right) DAGs.

only heterogeneity and a streaming setting are taken into account but data quality is included as a first-class optimization objective that needs to be traded for latency.

Edge Computing. EdgeWise [32] is a Stream Processing Engine built on top of Apache Storm, which aims to improve latency and throughput by assigning operations to workers. However, it does not focus on network characteristics and privacy of the data as we do in our work through the availability constraint. They assign at-most one worker to any operation, thus not parallelizing them across multiple resources. SpanEdge [33] approaches the minimization of latency and bandwidth usage by placing operators on edge and central workers. Geelytics [6] is a system that considers heterogeneous compute nodes when placing the tasks and also supports on-demand edge analytics. Foggy [34] is a framework that offers an automated IoT application deployment and update but the job placement targets only the minimization of latency. Wan et al. [35] consider mobile devices with limited storage and computation capacities. They target the maximum system throughput and place tasks based on a 3-dimensional distance metric (considering CPU, memory and geographical distance) from the edge nodes. Skarlat et al. [36] organize the devices into fog colonies containing fog cells. Each fog cell has certain CPU, storage and RAM capacities. The colonies communicate with each other and with the central cloud. Finally, Renart et al. [37] propose a solution on how to efficiently split applications among edge devices and the cloud improving metrics like latency, bandwidth consumption and messaging cost. We differ from the solutions mentioned above and we enrich the research in edge computing operator placement with the data quality objective and by combining edge analytics with partitioned parallelism while respecting any resource and privacy-related constraints.

Data Quality There are several recent studies that investigate the quality of data in streaming analytics. Zhang et al. [38] profile the accuracy-bandwidth trade-off by adjusting the data rate in Wide-Area Streaming analytics applications. They also extend JetStream [39], which adjusts the quality of data in multiple ways; through sampling, roll-ups on the storage cube, setting thresholds for the values and synopsis approximation. By doing so, JetStream can achieve lower bandwidth usage. Heintz et al. [40] also studied a trade-off between timeliness and accuracy but focused on windowed grouped aggregation of streaming data. ApproxIoT [41] and StreamSight [42] apply sampling on the streaming data in order to achieve higher throughput with a trade-off in accuracy. MobiQoR [43] is a mobile edge analytics framework that focuses on finding the optimal trade-off between acceptable quality of results decreasement, energy consumption and response time by solving a Linear Programming formula. Kuemper et al. [44] propose a framework that evaluates and ensures the quality of streaming data from IoT sensors. As can be seen, there is an increasing interest in data quality-related solutions and many of the works try to decrease the quality of data to attain faster response time. However, the problem we tackle in this work is the opposite. We try to find at which extent can we measure the quality of data at the expense of increased latency.

Apache Storm schedulers. Many works propose schedulers for Apache Storm. T-Storm [45] is a traffic-aware system that is based on the idea of not using all available worker nodes. R-Storm [46] and T3-Scheduler [47] focus on placing communicating tasks closer with each other. D-Storm [48] is also a resource-aware scheduler which is dynamic and self-adaptive. Aniello et al. [49] propose two schedulers; an offline topology-based and an online traffic-based one. Eskandari et al. [50] make use of graph partitioning heuristics in order to fuse communicating tasks into groups. Fan et al. [51] propose a scheduler consisting of a runtime load tracker, and both static and dynamic scheduling strategies. AdaStorm [52] makes use of machine learning to adaptively adjust the Storm configuration based on the streaming data rate. In the future, we plan to investigate combining our solution with such proposals to deliver an adaptive quality-aware scheduler that judiciously allocates tasks to specific resources.

8. Conclusions

In this work, we examine a task allocation problem for complex analytics over edge devices, while taking into account both latency and data quality. The two examined optimization objectives are contradicting, therefore we propose a solution that trades latency for an increased fraction of incoming data, for which data quality checks are performed. The overall solution is a hybrid one, consisting of (i) a first part that computes locally optimal task allocations and then improves the partial solution employing local search heuristics and (ii) a second part that places emphasis on limited degree of partitioned parallelism. We thoroughly evaluate our proposal using both extensive simulations and emulations after importing our optimizations into Apache Storm. The results are particularly encouraging in the sense that we manage to reach cost-based trade-offs between latency and data quality checks, we improve upon state-of-the-art by up to 2.56X in our simulations, and we drop the latency compared to the default scheduler in Apache Storm up to 8X.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Acknowledgments

The research work was supported by the Hellenic Foundation for Research and Innovation (H.F.R.I.), Greece under the "First Call for H.F.R.I. Research Projects to support Faculty members and Researchers and the procurement of high-cost research equipment grant" (Project Number: 1052).

References

[1] S. Yi, C. Li, Q. Li, A survey of fog computing: Concepts, applications and issues, in: Proc. of Workshop on Mobile Big Data, 2015, pp. 37–42.

[2] J. Lin, W. Yu, N. Zhang, X. Yang, H. Zhang, W. Zhao, A survey on internet of things: Architecture, enabling technologies, security and privacy, and applications, IEEE Internet Things J. 4 (5) (2017) 1125–1142.

[3] D. Reinsel, J. Gantz, J. Rydning, The Digitization of the Worldfrom Edge To Core, Tech. Rep., International Data Corporation, 2018.

[4] A. Vulimiri, C. Curino, B. Godfrey, K. Karanasos, G. Varghese, Wanalytics: Analytics for a geo-distributed data-intensive world, in: CIDR 2015, 2015.

[5] L. Liu, Z. Chang, X. Guo, S. Mao, T. Ristaniemi, Multiobjective optimization for computation offloading in fog computing, IEEE Internet Things J. 5 (2018) 283–294.

[6] B. Cheng, A. Papageorgiou, M. Bauer, Geelytics: Enabling on-demand edge analytics over scoped data sources, in: BigData Congress, 2016, pp. 101–108.

[7] X. Sun, N. Ansari, Edgeiot: Mobile edge computing for the internet of things, IEEE Commun. Mag. 54 (12) (2016) 22–29.

[8] A. Karkouch, H. Mousannif, H.A. Moatassime, T. Noel, Data quality in internet of things: A state-of-the-art survey, J. Netw. Comput. Appl. 73 (2016) 57–81.

[9] M. Nardelli, V. Cardellini, V. Grassi, F. Presti, Efficient operator placement for distributed data stream processing applications, IEEE Trans. Parallel Distrib. Syst. 30 (08) (2019) 1753–1767.

[10] T. Hiessl, V. Karagiannis, C. Hochreiner, S. Schulte, M. Nardelli, Optimal placement of stream processing operators in the fog, in: 3rd Int. Conf. on Fog and Edge Computing, ICFEC, 2019, pp. 1–10.

[11] C.-C. Hung, G. Ananthanarayanan, L. Golubchik, M. Yu, M. Zhang, Wide-area analytics with multiple resources, in: EuroSys, 2018, pp. 12:1–12:16.

[12] T. Toliopoulos, A. Gounaris, K. Tsichlas, A. Papadopoulos, S. Sampaio, Continuous outlier mining of streaming data in flink, Inf. Syst. 93 (2020) 101569.

[13] M. Khayati, A. Lerner, Z. Tymchenko, P. Cudré-Mauroux, Mind the gap: An experimental evaluation of imputation of missing values techniques in time series, Proc. VLDB Endow. 13 (5) (2020) 768–782.

[14] D.J. DeWitt, J. Gray, Parallel database systems: The future of high performance database systems, Commun. ACM 35 (6) (1992) 85–98.

[15] M. Zaharia, M. Chowdhury, T. Das, A. Dave, J. Ma, M. McCauly, M.J. Franklin, S. Shenker, I. Stoica, Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing, in: S.D. Gribble, D. Katabi (Eds.), NSDI, 2012, pp. 15–28.

[16] J. Leskovec, A. Rajaraman, J.D. Ullman, Mining of Massive Datasets, third ed., Cambridge University Press, 2020, URL http://www.mmds.org/.

[17] V. Cardellini, V. Grassi, F. Lo Presti, M. Nardelli, Optimal operator placement for distributed stream processing applications, in: Proc. of the Int. Conf. on Distributed and Event-based Systems, DEBS, 2016, pp. 69–80.

[18] A.-V. Michailidou, A. Gounaris, A fast solution for bi-objective traffic minimization in geo-distributed data flows, in: IDEAS, 2019, pp. 27:1–27:10.

[19] P. Pietzuch, J. Ledlie, J. Shneidman, M. Roussopoulos, M. Welsh, M. Seltzer, Network-aware operator placement for stream-processing systems, in: ICDE, 2006, p. 49.

[20] F. Dabek, R. Cox, F. Kaashoek, R. Morris, Vivaldi: A decentralized network coordinate system, in: Proc. of Conf. on Applications, Technologies, Architectures, and Protocols for Computer Communications, 2004, pp. 15–26.

[21] M. Symeonides, Z. Georgiou, D. Trihinas, G. Pallis, M.D. Dikaiakos, Fogify: A fog computing emulation framework, in: SEC, New York, NY, USA, 2020.

[22] A. Gounaris, G. Kougka, R. Tous, C.T. Montes, J. Torres, Dynamic configuration of partitioning in spark applications, IEEE Trans. Parallel Distrib. Syst. 28 (7) (2017) 1891–1904.

[23] Q. Pu, G. Ananthanarayanan, P. Bodik, S. Kandula, A. Akella, P. Bahl, I. Stoica, Low latency geo-distributed data analytics, in: SIGCOMM, 2015, pp. 421–434.

[24] Z. Hu, B. Li, J. Luo, Flutter: Scheduling tasks closer to data across geo-distributed datacenters, in: INFOCOM, 2016, pp. 1–9.

[25] I. Stanoi, G. Mihaila, C. Lang, T. Palpanas, Whitewater: Distributed processing of fast streams, IEEE Trans. Knowl. Data Eng. 19 (2007) 1214–1226.

[26] R. Viswanathan, G. Ananthanarayanan, A. Akella, CLARINET: Wan-aware optimization for analytics queries, in: OSDI, 2016, pp. 435–450.

[27] P. Li, S. Guo, T. Miyazaki, X. Liao, H. Jin, A.Y. Zomaya, K. Wang, Traffic-aware geo-distributed big data analytics with predictable job completion time, IEEE Trans. Parallel Distrib. Syst. 28 (6) (2017) 1785–1796.

[28] W. Xiao, W. Bao, X. Zhu, L. Liu, Cost-aware big data processing across geo-distributed datacenters, IEEE Trans. Parallel Distrib. Syst. 28 (11) (2017) 3114–3127.

[29] L. Gu, D. Zeng, S. Guo, Y. Xiang, J. Hu, A general communication cost optimization framework for big data stream processing in geo-distributed data centers, IEEE Trans. Comput. 65 (1) (2016) 19–29.

[30] Z. Niu, B. He, C. Zhou, C.T. Lau, Multi-objective optimizations in geo-distributed data analytics systems, in: ICPADS, 2017, pp. 519–528.

[31] A.-V. Michailidou, A. Gounaris, Bi-objective traffic optimization in geo-distributed data flows, Big Data Res. 16 (2019) 36–48.

[32] X. Fu, T. Ghaffar, J.C. Davis, D. Lee, Edgewise: A better stream processing engine for the edge, in: D. Malkhi, D. Tsafrir, (Eds.), USENIX.

[33] H.P. Sajjad, K. Danniswara, A. Al-Shishtawy, V. Vlassov, Spanedge: Towards unifying stream processing over central and near-the-edge data centers, in: Symposium on Edge Computing, SEC, 2016, pp. 168–178.

[34] E. Yigitoglu, M. Mohamed, L. Liu, H. Ludwig, Foggy: A framework for continuous automated iot application deployment in fog computing, in: Int. Conf. on AI Mobile Services, AIMS, 2017, pp. 38–45.

[35] Z. Wan, X. Deng, Z. Cao, H. Zhang, Mobile resource aware scheduling for mobile edge environment, in: IEEE Int. Conf. on Communications, ICC, 2018, pp. 1–6.

[36] O. Skarlat, M. Nardelli, S. Schulte, S. Dustdar, Towards qos-aware fog service placement, in: Int. Conf. on Fog and Edge Computing, ICFEC, 2017, pp. 89–96.

[37] E. Gibert Renart, A. Da Silva Veith, D. Balouek-Thomert, M.D. De Assunção, L. Lefèvre, M. Parashar, Distributed operator placement for iot data analytics across edge and cloud resources, in: CCGRID, 2019, pp. 459–468.

[38] B. Zhang, X. Jin, S. Ratnasamy, J. Wawrzynek, E.A. Lee, Awstream: Adaptive wide-area streaming analytics, in: SIGCOMM, 2018, pp. 236–252.

[39] A. Rabkin, M. Arye, S. Sen, V.S. Pai, M.J. Freedman, Aggregation and degradation in jetstream: Streaming analytics in the wide area, in: NSDI, 2014, pp. 275–288.

[40] B. Heintz, A. Chandra, R.K. Sitaraman, Trading timeliness and accuracy in geo-distributed streaming analytics, in: Proc. of Symposium on Cloud Computing, SoCC, 2016, pp. 361–373.

[41] Z. Wen, D.L. Quoc, P. Bhatotia, R. Chen, M. Lee, Approxiot: Approximate analytics for edge computing, in: Int. Conf. on Distributed Computing Systems, ICDCS, 2018, pp. 411–421.

[42] Z. Georgiou, M. Symeonides, D. Trihinas, G. Pallis, M.D. Dikaiakos, Streamsight: A query-driven framework for streaming analytics in edge computing, in: UCC, 2018, pp. 143–152.

[43] Y. Li, Y. Chen, T. Lan, G. Venkataramani, Mobiqor: Pushing the envelope of mobile edge computing via quality-of-result optimization, in: ICDCS, 2017, pp. 1261–1270.

[44] D. Kuemper, T. Iggena, R. Toenjes, E. Pulvermueller, Valid.iot: A framework for sensor data quality analysis and interpolation, in: Proc. of the 9th ACM Multimedia Systems Conference, 2018, pp. 294–303.

[45] J. Xu, Z. Chen, J. Tang, S. Su, T-storm: Traffic-aware online scheduling in storm, in: ICDCS, 2014, pp. 535–544.

[46] B. Peng, M. Hosseini, Z. Hong, R. Farivar, R.H. Campbell, R-storm: resource-aware scheduling in storm, 2019, CoRR abs/1904.05456.

[47] L. Eskandari, J. Mair, Z. Huang, D. Eyers, T3-scheduler: A topology and traffic aware two-level scheduler for stream processing systems in a heterogeneous cluster, Future Gener. Comput. Syst. 89 (2018) 617–632.

[48] X. Liu, R. Buyya, D-storm: Dynamic resource-efficient scheduling of stream processing applications, in: ICPADS, 2017, pp. 485–492.

[49] L. Aniello, R. Baldoni, L. Querzoni, Adaptive online scheduling in storm, in: DEBS, 2013, pp. 207–218.

[50] L. Eskandari, J. Mair, Z. Huang, D. Eyers, Iterative scheduling for distributed stream processing systems, in: DEBS, 2018, pp. 234–237.

[51] Jiahua Fan, Haopeng Chen, Fei Hu, Adaptive task scheduling in storm, in: Int. Conf. on Computer Science and Network Technology, ICCSNT, 01, 2015, pp. 309–314.

[52] Z. Weng, Q. Guo, C. Wang, X. Meng, B. He, Adastorm: Resource efficient storm with adaptive configuration, in: ICDE, 2017, pp. 1363–1364.