Decentralized Scaling for Stream Processing Engines

Mehdi Belkhiria, Cédric Tedeschi

▶ **To cite this version:**

Mehdi Belkhiria, Cédric Tedeschi. Decentralized Scaling for Stream Processing Engines. 2019. hal-02127609

HAL Id: hal-02127609
https://inria.hal.science/hal-02127609

Preprint submitted on 13 May 2019

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Decentralized Scaling for Stream Processing Engines

Mehdi Belkhiria, Cédric Tedeschi

Univ Rennes, Inria, CNRS, IRISA

May 13, 2019

Abstract

Stream Processing was recently introduced as a paradigm to easily develop and deploy applications targeting the near real-time processing of data getting continuously produced. Stream Processing engines such as Storm, Flink or Spark Streaming are today regarded as mature software platforms offering high-level programming abstraction easing the development of stream processing applications. Moreover, they ease the deployment of such applications over computing facilities such as clusters or clouds.

Autoscaling, i.e., the ability to scale elastically and autonomously according to the incoming load (in our case the input data stream) gained recently momentum in the stream processing context. Autoscaling in this context saw the design and implementation of different mechanisms to be able to finely adapt the amount of computing resources dedicated to each step of the processing.

Yet, with the advent of new platforms gathering smaller resources but at a larger geographic scale, as in the Fog computing paradigm, the need for decentralizing such autoscaling mechanisms appears, a topic which, to our knowledge, has been mostly ignored.

In this paper, we tackle the problem of decentralized autoscaling for stream processing applications. Based on the common assumption that a stream processing application is a combination of operators (in a pipeline) potentially distributed over distant compute nodes, we provide a protocol letting each operator takes its own scaling decisions based on local information. While each operator maintains only a view on its neighbors in the pipeline, our protocol is able to ensure that all operators keep a consistent view while these neighbors are scaled out or scaled in in response to load variations, ensuring global correctness, e.g. that no data record is lost due to an outdated connection to a deleted neighbor. Our protocol is presented in detail, and its correctness validated. Its performances are captured through both analysis and simulation experiments.

1 Introduction

While data is produced at an increasing pace in many different domains, ranging from smart cities to military applications, the need for a fast analysis of data streams so as to extract value from them when the data is still relevant appears more and more crucial. Stream Processing gained momentum in the quest for adequate abstractions and tools to address this issue.

Stream processing engines such as Storm [15], Flink [4], Spark Streaming [17] or Heron [10] are today's inevitable tools for a scalable analysis of continuously produced data. While these software suites offer high level programming models, tools to facilitate the deployment of stream processing applications, as well as strong reliability guarantees, a number of issues regarding their performance, efficiency and scalability have been identified [9]. Amongst them, autoscaling of these applications have been recently widely studied. Autoscaling of stream processing applications typically relies on a subsystem running within these engines and which is dedicated to collecting the right application metrics and resources' states so as to dynamically adapt the amount of resources dedicated to the application facing a varying data velocity as input.

More precisely speaking, a stream processing application consists generally in a directed acyclic graph where vertices represent the operators through which each data record in the input stream, (or *tuple*) must go, and edges the path followed by the tuples from one operator to another. These operators can be stateless or stateful. In this context, autoscaling means dynamically adapting the number of instances of each operator so as to cope with the varying velocity of the input stream, each operator possibly having different speeds and thus incurring different delays at different places in the graph and influencing the velocity downstream in the graph in a different way.

The topic of autoscaling of stream processing applications has gained attention recently. Several works proposed different kinds of mechanisms (online or static), on different kinds of operators (stateless or stateful) with different targeted platforms (fixed cluster or elastic cloud). These autoscaling subsystems are typically centralized and thus suffer from a limited scalability when these operators are deployed over geographically dispersed computing nodes, as typically advocated by Fog computing [11].

Deploying stream processing applications and autoscaling them in a Fog context brings new challenges: Each operator being hosted on a compute node potentially geographically distant from the compute node hosting its successor operator in the graph, keeping everywhere a centralized view of the graph and its state can become difficult. This aspect is again reinforced by the limited performance of resources constituting the fog, typically loosely-coupled micro-datacenters. Then, each operator can only maintain a local, very limited view of the graph. Scaling in this context while keeping a globally consistent graph to ensure no data is sent to an outdated link is a challenge in itself.

In this paper, we consider graphs that are pipelines of stateless operators. We focus on devising a decentralized autoscaling protocol in which each operator instance takes its own decisions towards duplication or self-termination, based on the local knowledge of the graph and locally experienced varying load. The autoscaling policy relies on a probabilistic decision taken locally that globally lead to an accurate level of parallelism with regards to the current global velocity. We show that our protocol is able, in spite of the decentralization and only a partial view of the graph on each node, to maintain the global graph consistent while the number of instances of each operator is modified concurrently by each of these instances taking scaling decisions independently. We present the protocol in detail, discuss its correctness facing concurrent independent duplication and deletion of operators' instances, and help capturing the expected performance of such a mechanism through simulation experiments.

The related work is presented in Section 2. In Section 3, the system model used to describe applications and platforms considered is given. Our decentralized scaling protocol, including the scaling policy in both in and out cases, as well as a sketch of proof regarding correctness facing concurrency is presented in Section 4. Simulation results are detailed in Section 5. Section 6 concludes.

2 Related Works

The scaling of distributed stream processing applications has been the subject of a recent series of works [6,9]. It comes mainly from the fact that there is an uncertainty in i) the velocity of the data stream to be processed, that can be hard to estimate before runtime and that can vary significantly over time, and ii) the actual computation cost of the operators included in the computation graph can be also difficult to estimate and moreover different from one operator to another.

These works can be classified according to at least two dimensions, namely the elasticity *direction*, and the elasticity *temporality*.

2.1 Elasticity actions

There are a limited set of elasticity actions to be used to scale the graph of operators that can help the performance of the graph, and its adaptation to the varying load. Three of them [9] are generally used in such a purpose: fusion, fission, and deletion

Fusion refers to the merging of two contiguous operators in the graph and hosted by two different compute nodes, into a single compute node (typically one of the compute nodes amongst the two). While it may

increase the load on the compute node chosen to host both operators, it targets the reduction of the network load by keeping within a node the traffic of the data stream that was traversing the network links between the two initial nodes. We do not use this operation in this work, as it is more a *placement* action than a *scaling* action.

Fission refers to an operator being duplicated: a new instance of an already running operator gets started so the level of parallelism of this operator is increased. Still, a fission brings an actual extra computing power dedicated to this operator if the new thread or process started use computing resource that was not (or at least not fully) leveraged prior to the fission. We discuss this in more detail in Section 2.2. Note that the type of operator influences the difficulty of its fission. When an operator is stateful, its parallelization as several instances requires to add splitters and mergers before and after the split operator, respectively, so as to merge partial states maintained on each instance. In contrast, a stateless operator is easier to parallelize. It is simply a matter of sharing the incoming stream to this operator between the instances. For that reason, some works even consider stateful operators as non parallelizable [14]. Having this in mind, and because statefulness is not a focus in this work, we consider only stateless operators in this paper.

2.2 Elasticity direction.

Elasticity approaches for distributed systems, would they be meant to host stream processing applications or another kind of applications, are generally classified into two broad categories: *vertical* approaches and *horizontal* approaches. Vertical approaches can be seen as a better usage of available computing nodes: it may be for instance an increase in the level of parallelism of an application (by spawning new processes or new thread) that will in return allocate more CPU and memory. Horizontal approaches, in contrast relies on the allocation of new computing nodes to support the application's workload. The work presented in this paper is actually *direction-agnostic*. Our approach considers each operator in the stream processing application to exist in an elastic number of instances. These instances, when created (for instance during a fission operation) can be deployed over an already allocated node or over a new node. In other words, the actual elasticity direction is left to the implementer of the solution described. Yet, in a Fog context, we consider small dispersed facilities to be already present and ready to receive some new work / new operator's instances to run. Yet this work focus on defining a protocol to decide when and what to scale (in a decentralized fashion). The *where*, i.e., the precise placement of the instances started is orthogonal to our contribution.

The topic of operator placement in Stream Processing applications have been recently studied by several series of works [12, 13].

2.3 Elasticity temporality.

Another dimension allowing to separate elasticity approaches is what we refer to as *temporality*, i.e., *when* the scaling mechanism is actually triggered. We broadly generally distinguish two families of mechanisms in this regard: the *static* ones and the *dynamic* ones.

Static approaches rely on the static analysis of the processing graph so as to infer an optimal parallelization of the graph. This analysis is run once before the application gets started. Among these works, we can mention the work by Schneider et al. [14], which propose a heuristic-based traversal of the graph so as to group operators together in different *parallel areas*, each area being a contiguous set of operators that are not stateful, stateful operators being considered here again as not trivial to parallelize without incurring an extra merging cost that may be higher than the gain of parallelism. While this static analysis is a necessary first step when the graph of operators, it is unable to find a continuous adequate level of parallelism able to adapt to the actual velocity of the incoming data stream to be processed. Unprecisely setting this level may lead to either an *overshoot*, i.e., two much resources allocated compared with the actual demand, leading to a waste of resources, or conversely to an insufficient amount of allocated resources leading to a performance degradation.

Dynamic (or *online*) techniques typically rely on two elements [5, 7, 8, 16]:

1. A centralized subsystem able to collect all the information required (current traffic, bottlenecks, available resources and their current usage, etc.) to be able to take correct decisions so as to optimize a certain metric.

2. A scaling policy which states *when* to trigger a scale-out, scale-in or reconfiguration mechanism

In [5], the authors propose a mechanism to monitor the CPU utilization so as to detect bottlenecks and trigger a scaling-out phase. They focus on the scaling of partitioned stateful operators by devising a mechanism to repartition the key spaces of tuples assigned to the different instances of a given operator at scale-out time.

Designed as an extension of Storm [15], T-Storm [16] introduces a mechanism of dynamic load rebalance triggered periodically, with a focus on trying to reduce internode communication (by grouping operators efficiently). Aniello et al. proposes a similar approach [1].

StreamCloud [8] provides a set of techniques to identify parallelizable zones of operators called *subqueries* into which the whole operator graph (or *query, by analogy with the database domain*) is split. A subquery starts with a stateful operator and ends with another one, called the *sink* and contains all stateless operators in between. This splitting algorithm share some similarities with the work in [14]: each subquery can be parallelized independently. yet, on top of this splitting mechanism, a dynamic scheduling is introduced to balance the load of a stream produced by the sink of a subquery to the dowmstream subqueries. The paper introduces interesting techniques to ensure the order of processing amongst parallelized stateful operators.

In [7], the authors propose a dynamic technique to adjust the number of instances of each operator in the graph so as to cope with the changing velocity of the input stream. The technique relies on a set of rules that try to ensure that the system sticks to a behaviour respecting the SASO properties (Settling time, Accuracy, Stability, Overshoot). In other words, the requirement is to be able to allocate the right number of instances that will ensure the performance of the system (accuracy), that this number is reached quickly (settling time), that it does not oscillate artificially (stability) and that no useless extra resources are used (overshoot). This work appears to be the closest to the work presented in the following section. Our objectives are similar yet, our assumptions, as detailed in Section 3 are different, and above all, our approach targets decentralized systems where Gedik et al still relies on a centralized subsystem to collect information and take and enforce decisions.

2.4 Introducing decentralization

To our knowledge, decentralizing autoscaling for stream processing application is an open problem. While we can find works giving clues to decentralize auto-scaling, in particular the DEPAS aproach [3], these works are not specifically targeted at stream processing, and focus on a distributed multi-cloud infrastructures with local schedulers taking decisions independently. The similarity between DEPAS and the present work stands in the fact that autonomous instances take scaling decisions based on a probabilistic policy. Still, DEPAS instances are local schedulers while our instances are instances of operators in a stream processing graph. While DEPAS focus on the probabilistic policies, the present work focuses on devising an autoscaling decentralized protocol maintaining the global consistency of the graph while it evolves.

3 System Model

3.1 Platform Model

We consider a distributed system composed of a set of (geographically dispersed) compute nodes. These nodes can be either physical or virtual nodes. The number of nodes is not bounded. In other words, we assume that the amount of computing resources available is not strictly limited. The allocation of a new (virtual) node is represented hereafter by the $createNode()$ method. Our contribution is orthogonal to the scheduling problem: we focus on enabling a decentralized scaling mechanism on resources that be allocated or deallocated as needed by the scaling policy. Compute nodes are homogeneous. Homogeneity brings two

benefits. Firstly, it makes scaling decisions easier. Secondly, it eases the allocation of Fog resources: all virtual machines allocated have the same size. Also, compute nodes are reliable. Besides cases of deallocations, they do not become unavailable. In other words, we assume this is the duty of the Cloud provider to ensure this reliability.

These nodes communicate through the network using FIFO reliable channels.

1. A message reaches its destination in a finite time
2. Two messages sent into the same channel are processed at the destination in the same order they were sent by the sender.

We abstract the communication through the following method:

send(type,ctnt,dest)

This method is non-blocking and takes three parameters

- **type** denotes the message type. It can take values such as *duplication* and *deletion_ack*.
- **ctnt** is the content of the message. Its structure can vary from one message to another.
- **dest** is the address of the destination node.

Finally, let us mention a higher-level communication primitive that sends multiple messages sequentially. Its pseudo-code is given in Algorithm 1. As detailed in Section 4, it is used in particular to inform other nodes of their new neighbors at duplication time.

Algorithm 1 $sendInformation(type, succs, preds, addresses)$

Input: *type*: scaling type (*duplication* or *deletion*)
Input: *succs*: Successors of the current OI
Input: *preds*: Predecessors of the current OI
Input: *addresses*: Addresses of the new or terminating nodes
  for all *node* **in** $succs \cup preds$ **do**
    $send(type, addresses, node)$
  end for

3.2 Application Model

We consider stream processing applications that can be represented as directed pipelines in which vertices represent operators to be applied on each record to be processed and edges represent streams between these operators. Edges can be also seen as data dependencies between operators. As discussed previously, the operators support stateless operations.

At starting time, each operator is launched on one particular compute nodes. Then, the scaling mechanism can duplicate the operators. Each operator running in a different computing unit. Each replica of an operator is referred to as an *operator instance* (OI) in the following. OIs running the same operator are referred to as *siblings*.

Conversely, an OI can be terminated for the sake of scaling in, for instance when the velocity of input data goes down. Fig. 1 illustrates a simple application made of three operators and its state after duplication of the middle operator.

Note that the load of an operator is shared by all its instances. As illustrated by Figure 1, each operator O_i can exist in several instances OI_ij where i is the index of the operator and j the index of the instance amongst instances of operator O_i. In the example of Fig. 1, the pipeline is made of three operators. The logical pipeline being shown on top of the figure. At the bottom of the figure is illustrated the potential concrete version of the pipeline after two duplications of operator O_1.

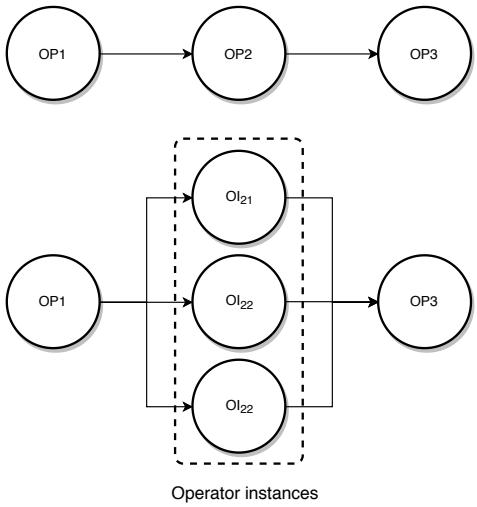

Figure 1: Scaling out a simple application.

4 Scaling Algorithm

The algorithm proposed and described in this section enables each OI to decide locally and independently when to get duplicated or deleted. The algorithm is run periodically by each OI (at possibly different times, and with possibly different frequencies). The algorithm starts with the decision phase. During this phase, the OI computes, based on information available locally, how loaded it is. As we assume that OIs are homogeneous and that the load is fairly distributed amongst the instances of a given operator, OIs are able to take uncoordinated yet relevant decisions at the operator scale.

Once an OI decided to get duplicated or deleted, it starts the second phase of the algorithm, responsible to actually trigger the duplication and inform the OIs pertained by this change in the graph (namely the successors and predecessors of the duplicated operator).

In the following, we start with detailing the decision process in Section 4.1. Then, in Section 4.2, we give the details of the protocol enabling the decentralized scaling.

4.1 Scaling decision

4.1.1 Duplication decision

As previously mentioned, each OI decides to get duplicated or deleted locally and independently from other OIs Recall that each OI runs on a different node, so the terms *OI* and *node* can be used interchangeably.

Let C denote the capacity of the (homogeneous) nodes, i.e. the number of records they can process per time unit without incurring performance degradation. Let n_t denote the current replication factor for this operator, i.e. the number of OIs currently running this operator. n_t can be known by an OI through communication with one of its predecessor in the graph that necessarily has this information, as it is required to share its outgoing stream between its own successors. Let l_t the current load experienced by the OI, and represent the number of records received during the last time unit. Finally, r, with $0 < r \leq 1$ denotes the desired load ratio for this particular operator (it is the same for all instances of a given operator but may vary from one operator to another one). r is set statically and can be seen as a user parameter.

The objective at scaling decision time for an OI is to find the replication factor to be applied to itself so

all OIs for this operator globally reach their targeted ratio. This replication factor is calculated as follows. The current experienced load for an operator can be estimated as $L_t = l_t \times n_t$.

Note that under the assumption of an homogeneous capacity of OIs and a fair distribution of the load on the OIs of a given operator, if the current OI is overloaded, so are its siblings. Yet if all OIs take a duplication decision concurrently, we may end up having an overshoot. For scaling up to the right number of OIs without the need for a global coordination between them, each node computes a *probabilistic replication factor*, that can have a value greater than 1. Let us first note that the desired load is $r \times C$.

The number of OIs to support L_t while ensuring that each OI reaches the desired load is then $\frac{L_t}{r \times C}$. Thus the number of nodes to add is $n_{diff} = \frac{L_t}{r \times C} - n_t$. Locally, this target is translated into the local replication factor to be applied $p = \frac{n_{diff}}{n_t}$.

If $p < 1$, this can be interpreted as a duplication probability: the node will get duplicated with probability p. Otherwise, the node will get duplicated $\lfloor p \rfloor$ times and then one final time with probability $p - \lfloor p \rfloor$.

4.1.2 Deletion decision

The inverse decision, i.e., for a node to terminate itself, follows the same principle as the previous decision. However, the factor calculated in this case has a value which is necessarily between 0 and 1. The other difference is that this decision is triggered not when the load is above a certain threshold, but below another, low, threshold. In this case, the risk is that all OIs for a given operator take this decision at the approximate same time, leading to a collective termination, ending up with no instance for this operator. We solve this by introducing a particular node (called the *operator keeper*) that cannot terminate itself whatever its load.

The decision process (would it lead to deletion or duplication) can be summarized by Algorithm 2

Algorithm 2 $getProbability(C, r, l_t, n_t) : float$

Input: C: Processing capacity
Input: r: Target load ratio
Input: l_t: current OI's load
Input: n_t: current number of siblings
1: $L_t \leftarrow l_t \times n_t$
2: $n_{diff} \leftarrow |\frac{L_t}{r \times C} - n_t|$
3: **Return** $\frac{n_{diff}}{n_t}$

Finally, let us mention a simple primitive function allowing to transform this probability into a boolean actually stating whether the deletion or duplication action will get enabled. It is given by Algorithm 3.

Algorithm 3 $applyProba(p : real) : int$

Input: $0 \leq p \leq 1$: Calculated duplication or termination's probability
$r \leftarrow rand()$
Return $r < p \ ? \ 1 : 0$

4.2 Scaling protocol

Algorithm 4 presents the pseudo-code of the procedure triggered when the duplication decision was taken. It manipulates the notions presented above plus two extra inputs: i)$thres_{\uparrow}$, the value above which the load ratio triggers the duplication policy, and ii) the successor's and predecessor's list.

The first part of the algorithm consists in calculating the amount of duplication needed to reach the targeted load ratio r (in Lines 2-4). From Lines 5 to Lines 8, new nodes are started, with the computed factor, according to the policy explained above. The $createNode()$ method actually triggers the launch of a new OI for the operator. Note that these new nodes are not yet *active*: they are idle, waiting for a message of the current node to initialize its neighbors — this will be done by Algorithm 6 — and start processing

incoming data. In the meantime, the current node, in Lines 10-13, spreads the information of the new nodes to be taken into account to its own neighbors. A counter of the expected responses is initialized: To validate the duplication and actually initialize the new, currently idle, nodes, the OI needs to collect the acknowledgement of all of its neighbors.

Algorithm 4 Scale-out protocol: initialization.

Input: $thres_{\uparrow}$: threshold
Input: $succs, preds$: arrays of successors and predecessors
1: **procedure** $operatorScale - Out()$
2: $p \leftarrow getProbability(C, r, l_t, n_t)$
3: $newAddrs \leftarrow List()$
4: $n \leftarrow \lfloor p \rfloor + applyProba(p)$
5: **if** $n > 1$ **then**
6: **for** $i \leftarrow 1$ **to** n **do**
7: $newNode \leftarrow createNode()$
8: $newAddrs.add(newNode)$
9: **end for**
10: $sendInformation($"$duplication$", $succs, preds, newAddrs)$
12: $nbAck \leftarrow 0$
13: $nbAckExpected \leftarrow |succs| + |preds|$
14: **end if**

The remainder of the duplication protocol is given by Algorithms 5, 6 and 7. Algorithm 5 shows what is done on the successors and predecessors of the duplicating OI on receipt of a message informing them of the duplication. In Lines 2-7, the case of a *duplication* message coming from a successor is processed: the addresses received are new predecessors for the current node, which are then added to the corresponding set. There are still two cases to consider: if the node receiving the message is itself not yet *active*, i.e., it is itself a new node waiting for its starting message (this can happen as detailed in Section 4.3), it will store the new neighbor in a particular *succsToAdd* set which contains non-active neighbors: the node may start processing incoming data but cannot yet send new data to its successors to avoid lost tuples, as detailed in Section 4.3.

Then, in Lines 8-13, the case of a duplication coming from a predecessor is processed similarly. Finally, the node acknowledges the message to the duplicating node by sending a *duplication_ack* message.

Algorithm 5 Scale-out protocol: receipts on succs and preds.

1: **upon** receipt **of** ("*duplication*", *addrs*) **from** p
2: **if** $p \in$ succs **then**
3: **if** isActive **then**
4: $succs = succs \cup addrs$
5: **else**
6: $succsToAdd = succsToAdd \cup addrs$
7: **end if**
8: **else if** $p \in$ preds
9: **if** isActive **then**
10: $preds = preds \cup addrs$
11: **else**
12: $predsToAdd = predsToAdd \cup addrs$
13: **end if**
14: $send($"$duplication_ack$", $p)$

Algorithms 6 and 7 show the final step in this protocol: once all the acknowledgements have been received by the duplicating OI from its neighbors, the new nodes can finally become active and start processing records. To this end, in Line 5 of Algorithm 4, the duplicating OI sends a *start* message to all its new siblings. On receipt, the new siblings just need to initialize the sets of its neighbors by combining the sets sent by their initiator (the duplicating OI) and the other information received in the meantime, and stored into $*ToAdd$ and $*ToDelete$ variables. Refer to Section 4.3) for more information about this possibility. This is done in Lines 2-3 of Algorithm 7.

Algorithm 6 Scale-out protocol: receipts of acks.

```
1: upon receipt of ("duplication_ack")
2:   nbAck++
3:   if nbAck = nbAckExpected then
4:     for all newSibling in newAddrs do
5:       send("start", succs, preds, newSibling)
6:     end for
7:   end if
```

Algorithm 7 Scale-out protocol: receipt of the start signal.

```
1: upon receipt of ("start", succs_, preds_) from p
2:   succs = succs_ ∪ succsToAdd \ succsToDelete
3:   preds = preds_ ∪ predsToAdd \ predsToDelete
4:   isActive ← true
5:   activate()
```

Let us now review the protocol for a terminating node, detailed in Algorithms 8, 9 and 10. It is very similar to the protocol presented previously enabling the scale-out case. Algorithm 8 shows the initialization of the protocol: the current OI, about to self-terminate must ensure that every node pertained by the deletion (each neighbor) is aware of it before actually terminate itself.

Algorithm 8 Scale-in protocol: initialization.

```
Input: thres_↓: threshold
Input: succs, preds: array of successors and predecessors
1: procedure operatorScale − In()
2:   p ← getProbability(C, r, l_t, n_t)
3:   if applyProba(p) then
4:     sendInformation("deletion", succs, preds, me)
5:     nbAck ← 0
6:     nbAckExpected ← |succs| + |preds|
7:   end if
```

On receipt of this upcoming termination information, we again have to consider two cases, depending whether we are currently active or not: if the receiving node is active, then, the node is simply removed from the list of its neighbors (either from *pred* or *succ*) and an acknowledgement is sent back. Otherwise, the node is stored in a *to be deleted* set of nodes, that will be taken into account at starting time.

Algorithm 9 Scale-in protocol: receipts on succs and preds.

```
1: upon receipt of ("deletion", addr) from p
2:   if P ∈ succs then
3:     if isActive then
4:       succs ← succs \ addr
5:     else
6:       succsToDelete ← succsToDelete ∪ addr
7:     end if
8:   else if p ∈ preds
9:     if isActive then
10:      preds ← preds \ addr
11:    else
12:      predsToDelete ← predsToDelete ∪ addr
13:    end if
14:  send("deletion_ack", p)
```

The final step consists, on the node about to terminate, to count the number of acknowledgements. As discussed in Section 4.3, the terminating node must wait for all the acknowledgement of the nodes it considers as neighbors. Once it is done, it flushes its data queue and triggers its own termination.

Algorithm 10 Scale-in protocol: receipt of acks and termination.

```
1: upon receipt of ("deletion_ack")
2:   nbAck + +
3:   if nbAck = nbAckExpected then
4:     // wait current tuples to be processed
5:     terminate()
6:   end if
```

Algorithm 11 presents the global algorithm's pseudo-code putting it all together. It is triggered periodically by every node, but potentially at different times. Notice that to enter a deletion action, a node cannot be the leader: we ensure that there is always at least one instance of every operator. This leader can for instance be the initial instance of the operator at pipeline's starting time. Without such an assumption, some operators could definitely disappear, leading to the pipeline's failure.

Algorithm 11 Periodically triggered autoscaling mechanism

```
1: if l_t ≥ thres_↑ then
2:   operatorScale − Out()
3: else if l_t ≤ thres_↓ and !isLeader then
4:   operatorScale − In()
5: end if
```

4.3 Sketch of correctness proof

The protocol's correctness relies on several aspects. First, nodes do not crash and messages reach their destination systematically and are processed in a finite time. Yet, we adopted an asynchronous model: each node moves at its own pace, and the time for a message to reach its destination is finite but not bounded. The final assumption is that channels are FIFO: two messages sent through a given order are received and processed in the same order.

Let us informally justify these choices. Firstly, let us mention that our *ack* messages are necessary. When a node duplicates itself, it must wait for an acknowledgement from all of its neighbors to actually start the

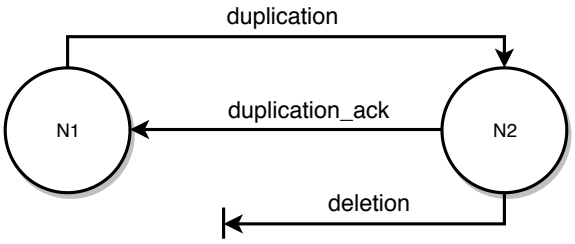

Figure 2: A potential synchronization issue.

new OI. Without such a precaution, the new node could start emitting messages to some successors that may not consider the new OI as a predecessor.

Secondly, without the FIFO assumption, our protocol may also lead to some inconsistencies. Let us consider the following case, illustrated by Figure 2. In this case, two neighboring nodes in the graph, N2 being the successor of N1 triggers two antagonist operations at the approximate same time. N1 triggers a duplication while N2 triggers its own termination. Consequently, N1 sends a *duplication* message while N2 sends a *deletion* message. Let us assume that N2's deletion takes far longer to reach N1 than N1's message to reach N2. Without the FIFO assumption, the *duplication_ack* message sent back by N2 to N1 may be processed on N1 before the deletion message, resulting in N1 counting N2 as a neighbor of the to-get-started OI that will send data records to N2, without being aware of N2's deletion.

Introducing the FIFO assumptions for the channels solve this concurrency problem. We show that by distinguishing two cases:

1. N2 sends the deletion message before processing N1's duplication message. In this case, due to the FIFO assumption, N1 will first receive N2's deletion message and remove N2 from its set of successors, so that, once N1 receives all of the *duplication_ack* from its neighbours (including N2), N1 will send the starting message to the new OI with a set of successors not including N2, leading both N1 and the new OI to not consider N2 as a successor.

2. N2 processes the *duplication* message sent by M1 before sending its own *deletion* message. In this case, N2 sends the *duplication_ack* message before the *deletion* message. So they will arrive in this order on N1. On receipt of the first message, N1 still considers N2 as a neighbour for the future OI and may send it to the new OI at starting time. Yet it is not a problem, as N2 now knows about the new OI and will send its deletion message also to it. While not yet active, the new OI will receive the deletion message and keep the information that N2 is to be deleted from the successors at starting time, as enforced by Line 6 in Algorithm 5.

The case of two concurrent duplications is even simpler. Yet this case can be processed similarly to the previous one, by distinguishing two cases. Either the two duplications are not really concurrent, and the later duplication message received will be sent after the first duplication message is processed, or they are really concurrent, and each duplication messages arrives before the other one is processed. In the latter case, each node will learn its new neighbor independently and start their new OI with the information of that new neighbors' OI.

5 Simulation results

In this section, we present the results of our solution based on simulation results. We evaluate the accuracy, rapidity of scaling and the network traffic induced by the solution itself.

5.1 Simulation set-up

To evaluate our protocol, we developed a discrete-time simulator in Java. Each time step t sees the following operations: a subset of the nodes starts testing the conditions for triggering a scaling operation, would it be duplication or deletion. In case the protocol is actually initiated, the first message (*duplication* or *deletion*) is received by the neighbors of the initiating node. The following steps respect the following rule: messages sent at step t are processed at step $t+1$ and the newly sent messages during this process is sent. These new messages will be processed at time $t+2$, and so on.

Having this in mind, let us remark that one scale-out operation takes three steps: i) a node takes the scale-out decision and sends the *duplication* message which is processed at this same first iteration so its successors and predecessors treat it and send a *duplication_ack* message. ii) The initiating node receives all the acks and sends a start message to its new siblings to activate them. iii) the new siblings receive the *start* message and start processing data records. Scaling-in takes only 2 steps. i) a node decides to terminate itself, so it sends a deletion message to their successors and predecessors which process the message and send the *deletion_ack* message . ii) The node receives the acks and terminates itself.

The variation of the workload is modelled by a stochastic process, mimicking a Browning motion. Using Browning motion allows us to evaluate our algorithm with a quick yet swift variation of the workload and give it a more realistic aspect. The graph tested is a pipeline composed of 5 operators, each operator having a workload evolving independently (through a Browning motion). Initially, each operator is duplicated on 7 OIs (for a total number of OIs of 35). Compute nodes hosting OIs have a processing capacity of 500 (tuples per time step). The other parameters of our simulation are listed below:

- Ideal load ratio $r = 0.7$
- top_threshold $thres_{\downarrow} = 0.8$
- down_threshold $thres_{\uparrow} = 0.6$
- Nodes try to start the scaling protocol every 5 steps
- Simulation runs for 200 steps.

In the following sections, we present our results on the *rapidity of scale* with which our algorithm is able to converge to the adequate number of instances, both locally for one operator, and globally. This accuracy is shown with two metrics: the distance to the ideal throughput (in other words, the relation between the number of tuples received and the number of tuples we are able to process) as well as the distance to the ideal number of nodes, to show that in spite of the fact that decisions are taken locally, the actual number of nodes is close to what would be an ideal one. Finally, the network traffic is estimated first analytically, and then put in perspective to our results obtained through simulation.

5.2 Rapidity of scale

We first discuss our algorithm's ability to quickly adapt to load's variations and reach an adequate number of instances through local decisions taken by operators' instances to duplicate or terminate themselves.

Fig. 3(a) shows the result of the experiment having the previously devised parameters. The blue curve shows the load (aggregated number of tuples received by the nodes, whatever their position in the pipeline). The red curve shows how the global number of OIs evolved during the experiment.

Firstly, we observe that the number of nodes decreases with the decline of the workload during the first 25 iterations. Then, it increases until reaching the peak of 114 nodes at iteration 104 quickly after the load itself reached the peak of 40396 tuples by time step at iteration 100. In the rest of the experiment, the load (and consequently the number of OIs, between 95 and 111) fluctuates a bit.

Secondly, we observe that the number of nodes scales quickly in some steps. It shows, that the delay between a variation in the load and the adaptation an be reduced to a very small time. This shows also that nodes are able, without coordination, and only based on decisions taken locally based on local information,

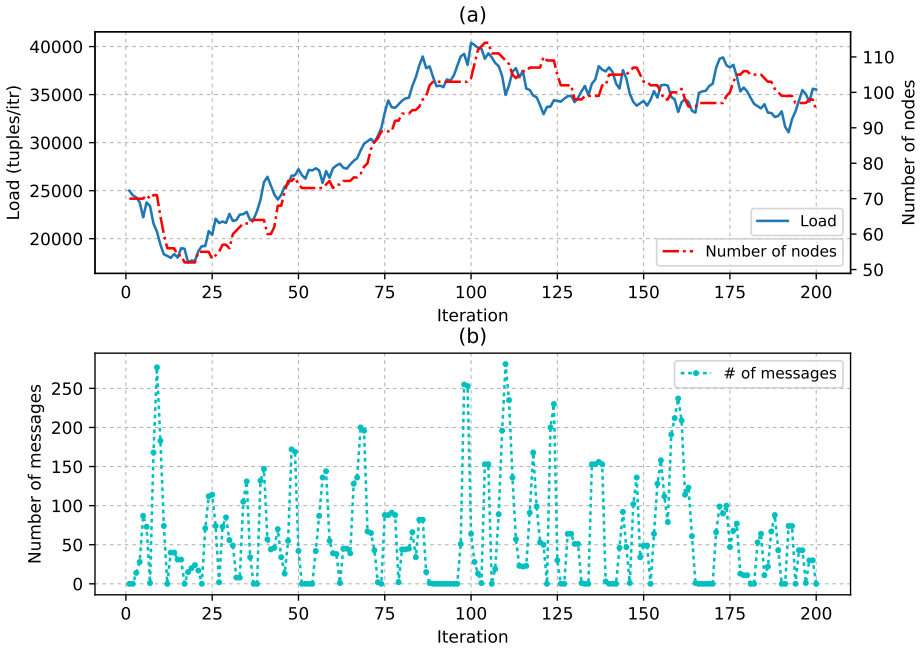

Figure 3: Load and number of nodes, global scale.

to start or remove nodes in a *batch* fashion. It means that if X more nodes are needed, X nodes will be added over a short period of time, the burden of starting these X nodes being shared by the existing nodes.

Fig. 4(a) presents the same experiment, but focused on one of the operator in the pipeline. The behavior observed at the global scale is even more visible at this scale: the proportion of OIs for this operator closely follows the proportion of tuples received by this operator.

Fig. 4(b) and Fig. 4(b) shows the traffic incurred by the scaling protocol: each significant increase or decrease in the number of OIs leads to a proportional traffic. We come back to this aspect in more detail in Section 5.4.

5.3 Accuracy

In this section, we focus on capturing the accuracy of the algorithm. For this, we study two metrics. The *accuracy* which is calculated as the ratio between the current number of instances and the ideal one. The ideal number of nodes is obtained by dividing the load by the capacity of nodes, the whole multiplied by the ideal load ration r. In other words, it represents the number of nodes we need to exactly handle the load, each node being loaded at a ratio equal to r. The accuracy is shown for the same experiment as in Section 5.2, in blue in Fig. 5. An accuracy of 1 means that actual number of nodes is the ideal one. An accuracy higher than 1 means that the operator has more instances than necessary. Finally, an accuracy of less than 1 means that we would need more instances. In this last case, we see a departure from 100% of the ideal throughput.

The ideal throughput is the ratio between the aggregated capacity and the aggregated load received at a given step. In other words, when the accuracy is lower than 1, in a discrete vision of the arrivals of tuples, there is a certain amount of tuples that are lost, or at least may get delayed because the instances are not able to *absorb* the load received at one iteration before the end of this iteration.

Fig. 5 shows two peaks in the accuracy (between iterations 25 and 50). During these peaks, the algorithm led to what is referred to as an *overshoot* in the literature: there are 2.5 times too many instances regarding

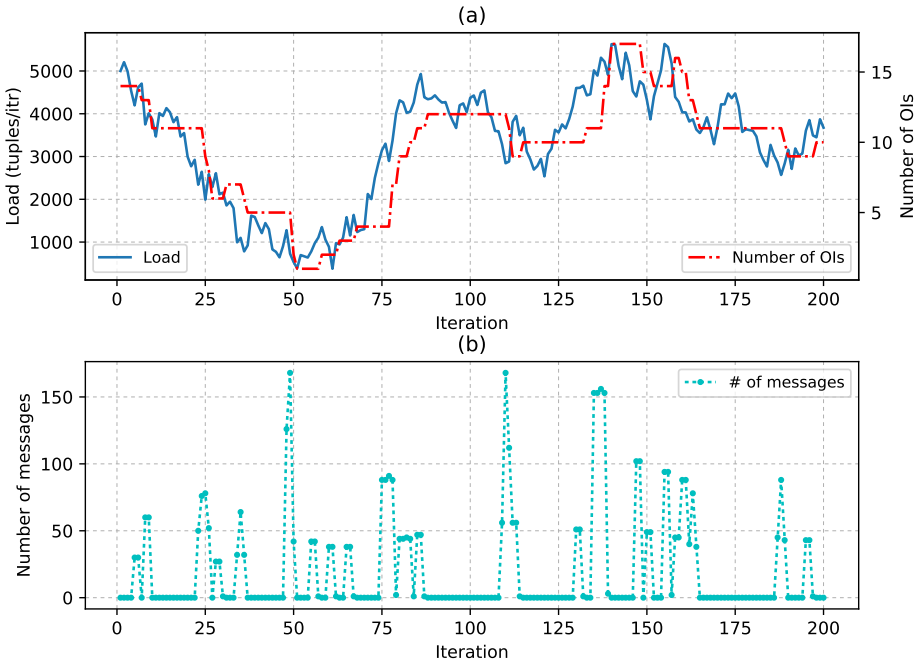

Figure 4: Load and number of nodes, local scale.

the instantaneous load. Note that iteration 46 sees a sudden load decrease of the load down to only 643 tuples received at this iteration. In this case we would actually need only 2 instances as the capacity of a single instance is 500, but there are 5 instances due to previous scaling operations. Yet the system scale-in to reduce to these 2 instances at iteration 49 and get back to the ideal accuracy.

Later, in iteration 57, the workload increases suddenly to 1104 and then 1351 in iteration 58. At this time, the operator has only 1 instance, but it needs 3 instances to reach an accuracy of 1. For this reason, we observe an accuracy of 0.33 and a decrease in the percentage of the ideal throughput down to 45% corresponding to a load of 1104 while the system has just one instance of capacity equal to 500.

Such inaccuracies do not incur performance degradation but may have a useless extra cost. They are difficult to avoid in the sense that the faster the load evolves, the more difficult it is to actually stick to an accuracy of 1. Fine tuning is needed and depends on the actual context, and on the aspect of the data stream processed. Such a tuning falls beyond the scope of this paper and is anyway specific to a given application and data stream.

5.4 Network Traffic

We now give both analysis and simulation-based elements to estimate the traffic incurred by the protocol.

Let us first consider the traffic generated by a single duplication phase for one operator that is suppose to reach an amount of OIs which depends on the variation of the load. Let us assume that according to the last variation in the load, k new nodes are needed, and that the current number of OI for the operator considered is n. Let us finally denote *succ* and *pred* the set of successors and predecessors for this operator, respectively.

We start with a simple case where a single instance is created by one of the instances in place. In this case, the number of messages will be $2(|succs| + |preds|) + 1$, as detailed in Section 4: the instance sends a duplication message to all its successors and its predecessors, then waits for their *acks*, to finally send a start message to the new sibling to be activated.

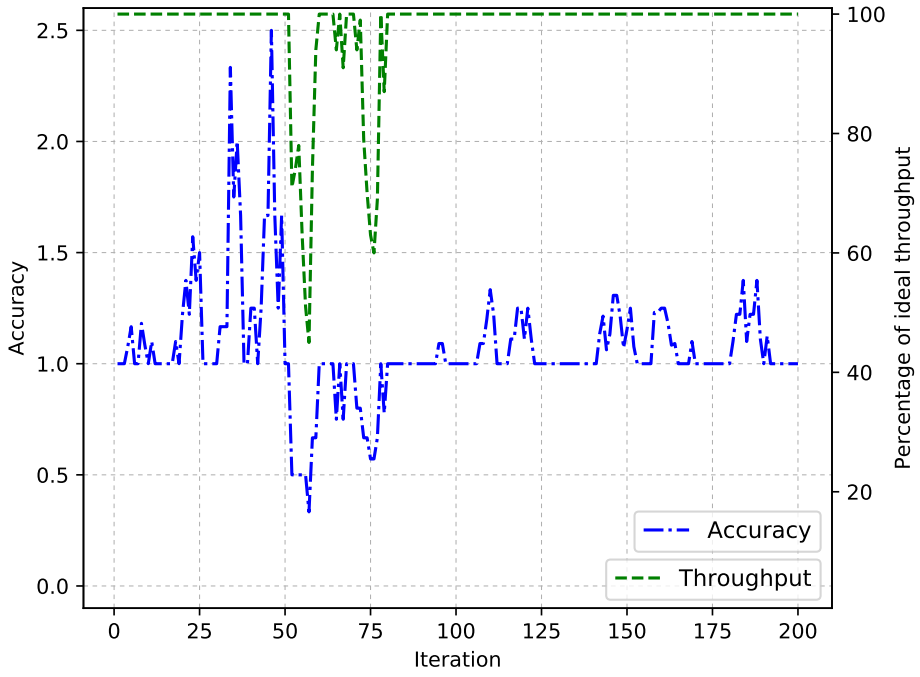

Figure 5: Accuracy and percentage of ideal throughput.

In the general case, we have k new siblings to be created by the n current instances. There are two cases. If $k < n$, it means k instances trigger their duplication. If $k > n$, then all the instances trigger their duplication. Then the number of messages is:

$$2min(k, n)(|succs| + |preds|) + k$$

.

Generalizing this at the global level, the total traffic becomes quadratic in the number of nodes, as the previous result needs to be summed over all the levels, and the number of successors and successors appears then twice as a factor.

Let us now visualize this traffic in the previously describe simulation experiments. Fig. 3(b) and 4(b) shows the number of messages per iteration throughout the simulation run. Messages starts to be sent at iteration 5 where 30 messages are sent (at the local level, see Fig. 4). This number is due to the fact that for the operator considered, there are 14 successor nodes and 16 predecessor nodes.

Then we have a peak of 168 messages at iteration 49 due to the huge decrease of the workload. During this iteration, one instance triggers its own deletion, as the operator considered has 21 successors nodes and 21 predecessors nodes, it sends 42 messages. We need to add messages due to the deletion of 3 other instances of the same operator decided at the previous iteration, leading to 126 acks sent at iteration 49, which, combined with the 42 others, equal 168. More globally speaking, the amount of messages varies with the variation of the workload.

Let us finally have a look at the network traffic when the workload is artificially increased monotonically, so as to check how the traffic evolves with the number of nodes.

Fig. 6 confirms an evolution of the traffic which is quadratic in the number of nodes (at iterations during which actual duplications are done), the number of nodes being, as already established, proportional to the workload.

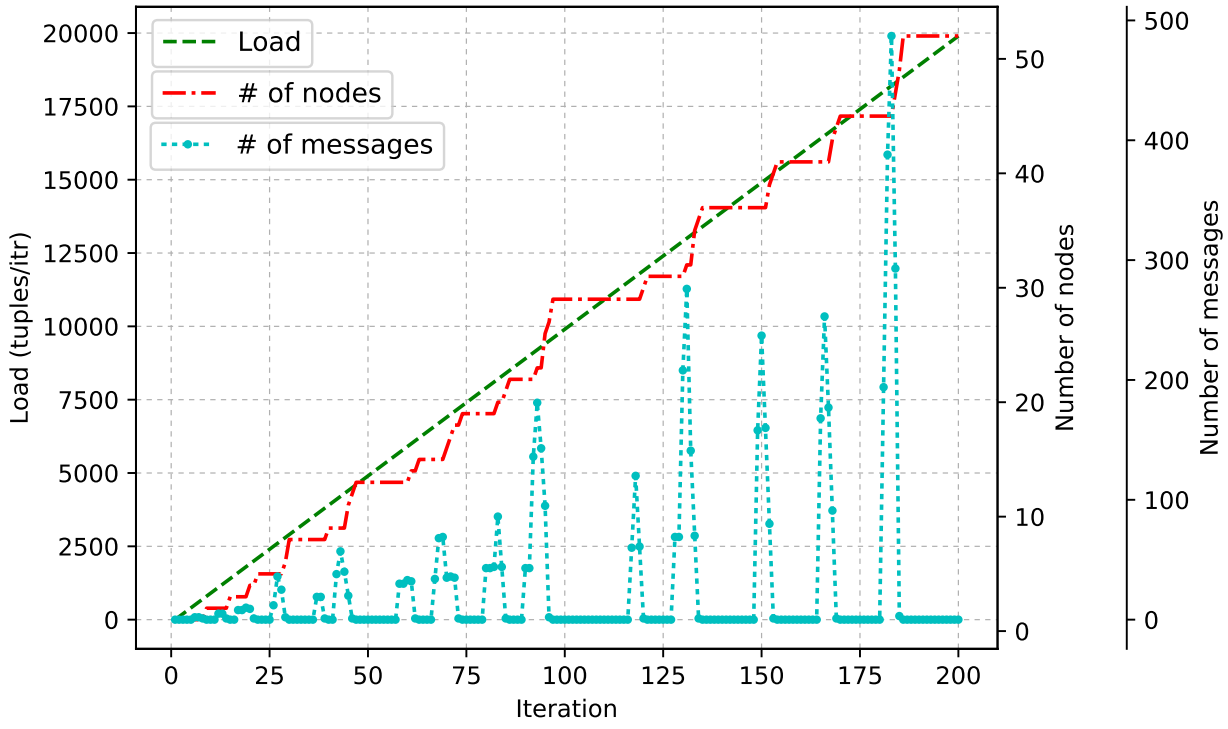

Figure 6: Traffic when facing a monotonically increasing load.

6 Conclusion

We presented an autoscaling algorithm for stream processing applications to be deployed over a geographically-dispersed set of resources. In such a Fog context, maintaining a centralized subsystem responsible of the scaling of the whole platform becomes complicated.

Our algorithm relies on independant, local autoscaling decisions taken by operators having only a partial view of the experienced load and the graph. Maintaining a globally consistent graph in these conditions is a challenge in itself. we present the mechanisms ensuring this correctness along with the results of simulation experiments showing that local probabilistic decision can lead to a global accurate level of parallelism in regard to the globally experienced load.

So as to validate this approach in a more real context, we are currently prototyping a distributed stream processing engine relying on the protocol presented for scaling. A first large scale validation of it is planned on the distributed experimental platform Grid'5000 [2].

Acknowledgment

This project was partially funded by ANR grant ASTRID SESAME ANR-16-ASTR-0026-02.

References

[1] Leonardo Aniello, Roberto Baldoni, and Leonardo Querzoni. Adaptive online scheduling in storm. In *Proceedings of the 7th ACM International Conference on Distributed Event-based Systems*, DEBS '13, pages 207–218, New York, NY, USA, 2013. ACM.

[2] Daniel Balouek, Alexandra Carpen Amarie, Ghislain Charrier, Frédéric Desprez, Emmanuel Jeannot, Emmanuel Jeanvoine, Adrien Lèbre, David Margery, Nicolas Niclausse, Lucas Nussbaum, Olivier Richard, Christian Pérez, Flavien Quesnel, Cyril Rohr, and Luc Sarzyniec. Adding virtualization capabilities to the Grid'5000 testbed. In Ivan I. Ivanov, Marten van Sinderen, Frank Leymann, and Tony Shan, editors, *Cloud Computing and Services Science*, volume 367 of *Communications in Computer and Information Science*, pages 3–20. Springer International Publishing, 2013.

[3] Nicolò M. Calcavecchia, Bogdan A. Caprarescu, Elisabetta Di Nitto, Daniel J. Dubois, and Dana Petcu. Depas: a decentralized probabilistic algorithm for auto-scaling. *Computing*, 94(8):701–730, Sep 2012.

[4] Paris Carbone, Asterios Katsifodimos, Stephan Ewen, Volker Markl, Seif Haridi, and Kostas Tzoumas. Apache flink™: Stream and batch processing in a single engine. *IEEE Data Eng. Bull.*, 38(4):28–38, 2015.

[5] Raul Castro Fernandez, Matteo Migliavacca, Evangelia Kalyvianaki, and Peter Pietzuch. Integrating scale out and fault tolerance in stream processing using operator state management. In *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data*, SIGMOD '13, pages 725–736, New York, NY, USA, 2013. ACM.

[6] Marcos Dias de Assunção, Alexandre Da Silva Veith, and Rajkumar Buyya. Distributed data stream processing and edge computing: A survey on resource elasticity and future directions. *J. Network and Computer Applications*, 103:1–17, 2018.

[7] Bugra Gedik, Scott Schneider, Martin Hirzel, and Kun-Lung Wu. Elastic scaling for data stream processing. *IEEE Trans. Parallel Distrib. Syst.*, 25(6):1447–1463, 2014.

[8] V. Gulisano, R. Jimnez-Peris, M. Patio-Martnez, C. Soriente, and P. Valduriez. Streamcloud: An elastic and scalable data streaming system. *IEEE Transactions on Parallel and Distributed Systems*, 23(12):2351–2365, Dec 2012.

[9] Martin Hirzel, Robert Soulé, Scott Schneider, Buğra Gedik, and Robert Grimm. A catalog of stream processing optimizations. *ACM Comput. Surv.*, 46(4):46:1–46:34, March 2014.

[10] Sanjeev Kulkarni, Nikunj Bhagat, Maosong Fu, Vikas Kedigehalli, Christopher Kellogg, Sailesh Mittal, Jignesh M. Patel, Karthik Ramasamy, and Siddarth Taneja. Twitter heron: Stream processing at scale. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, SIGMOD '15, pages 239–250, New York, NY, USA, 2015. ACM.

[11] Redowan Mahmud and Rajkumar Buyya. Fog computing: A taxonomy, survey and future directions. *CoRR*, abs/1611.05539, 2016.

[12] Boyang Peng, Mohammad Hosseini, Zhihao Hong, Reza Farivar, and Roy Campbell. R-storm: Resource-aware scheduling in storm. In *Proceedings of the 16th Annual Middleware Conference*, Middleware '15, pages 149–161, New York, NY, USA, 2015. ACM.

[13] P. Pietzuch, J. Ledlie, J. Shneidman, M. Roussopoulos, M. Welsh, and M. Seltzer. Network-aware operator placement for stream-processing systems. In *22nd International Conference on Data Engineering (ICDE'06)*, pages 49–49, April 2006.

[14] Scott Schneider, Martin Hirzel, Bugra Gedik, and Kun-Lung Wu. Auto-parallelizing Stateful Distributed Streaming Applications. In *International Conference on Parallel Architectures and Compilation Techniques, PACT '12*, pages 53–64, Minneapolis, USA, September 2012.

[15] Ankit Toshniwal, Siddarth Taneja, Amit Shukla, Karthikeyan Ramasamy, Jignesh M. Patel, Sanjeev Kulkarni, Jason Jackson, Krishna Gade, Maosong Fu, Jake Donham, Nikunj Bhagat, Sailesh Mittal, and Dmitriy V. Ryaboy. Storm@twitter. In *International Conference on Management of Data (SIGMOD 2014)*, pages 147–156, Snowbird, USA, June 2014.

[16] J. Xu, Z. Chen, J. Tang, and S. Su. T-storm: Traffic-aware online scheduling in storm. In *2014 IEEE 34th International Conference on Distributed Computing Systems*, pages 535–544, June 2014.

[17] Matei Zaharia, Tathagata Das, Haoyuan Li, Timothy Hunter, Scott Shenker, and Ion Stoica. Discretized streams: fault-tolerant streaming computation at scale. In *24th ACM SIGOPS Symposium on Operating Systems Principles (SOSP'13)*, pages 423–438, Farmington, USA, November 2013.