A predictive approach for dynamic replication of operators in distributed stream processing systems

Daniel Wladdimiro
Sorbonne University, INRIA, CNRS, LIP6
Paris, France
daniel.wladdimiro@lip6.fr

Luciana Arantes and Pierre Sens
Sorbonne University, INRIA, CNRS, LIP6
Paris, France
luciana.arantes@lip6.fr
pierre.sens@lip6.fr

Nicolas Hidalgo
Universidad Diego Portales
Santiago, Chile
nicolas.hidalgoc@mail.udp.cl

***Abstract*—Stream Processing Systems (SPSs) can present significant fluctuation in input rate. To address this issue, some existing solutions propose reconfiguring the SPS by replicating its operators. However, such reconfiguration usually induces a high system downtime cost. Moreover, reconfiguration decisions are based only on resource utilization without balancing the load between replicas.**

We propose in this paper a predictive SPS that dynamically defines the necessary number of replicas of each operator based not only on the current resource utilization and input rate variation but also on the events that, due to the operator's overloading, could not be processed yet and are, thus, kept in the operator's queue. In addition, our SPS implements a load balancer that distributes incoming events more evenly among replicas of an operator. Our solution has been integrated into Storm. To avoid system reconfiguration downtime, our SPS pre-allocates a pool of replicas where each of them can be activated or deactivated based on per operator input load predictions. Using real traffic traces with different applications, we have conducted experiments on Google Cloud Platform (GCP), evaluating our SPS and comparing it with Storm and DABS-Storm.

***Index Terms*—Stream processing, Adaptive SPS, Predictive algorithm, Replication, Google Cloud Platform**

I. Introduction

There is fast growth in the volume of data created by existing applications or systems on the Web due to large-scale user interactions (e.g., Twitter streaming data, real time stock trades, multiplayer game iterations, etc.). Consequently, processing such data in real-time, and delivering helpful information results in short periods, is increasingly requested by companies in different areas such as trading, security, and research, among others. To this end, Stream Processing Systems (SPSs) are intensely used [1]. The aim is to process the largest amount of information (number of events), avoiding as much as possible the loss of processing data which can decrease the quality of the delivered results.

SPSs are based on directed acyclic graphs (DAG) where vertices and unidirectional edges correspond to operators and event data flows, respectively [2]. An external source provides data continuously. Operators are based on light programming tasks (such as filters, counters, storage, etc.) that process the desired information in short periods in a pipeline way. Deployed in a processing infrastructure (e.g., Clouds, clusters, etc.), resources (e.g., VMs) are allocated to operators and often replicated for performance sake. However, in most SPSs, the number of replicas per operator is defined beforehand and does not change during execution. This static behavior might induce bottlenecks in processing events due to the dynamic nature of the data flows that usually present input rate fluctuations. Sudden traffic up spikes may overload some operators, increasing end-to-end processing latency. Increasing the number of such operators replicas is necessary to overcome this issue. On the other hand, in the case of down spikes, resources may become underloaded and, therefore, the number of replicas should be reduced to save resources.

Another critical feature of SPSs is the shuffle grouping technique which is responsible for sharing the input load among operators' replicas. For instance, the round-robin policy is one of the most common approaches to implementing shuffle grouping. However, the latter does not consider the input load of each replica, i.e., current queued events waiting to be processed.

By focusing on stateless operators, this article proposes a predictive DAG-based SPS algorithm that dynamically defines the current number of per operator replicas necessary to process the input stream. The flow of events is divided into time intervals, and our SPS defines, for each operator O, the events that O should process within each time interval. These events concern not only the ones that O's direct operator predecessors sent to it but also those related to previous time intervals that O could not process yet and are thus kept in a queue. In order to process these events, the ideal number O's replicas should be estimated at each time interval by taking into account both the number of such events as well as the average event execution time. Hence, the number of O's replicas dynamically increases or decreases over time according to its input rate. A new grouping technique that considers the current input load of operator replicas is also proposed.

Our SPS extends Apache Storm [1] and uses a predictive algorithm that follows a MAPE model [3], which relies on a four-stage control loop widely used in autonomic systems.

In order to avoid the downtime cost of stopping and restarting Storm when the number of an operator replicas needs to be reconfigured, our SPS allocates, at initialisation phase, a pool of per operator replicas where each of them can be

either in *active* or *inactive* state. An inactive replica of an operator consumes a negligible percentage of CPU and can be dynamically activated when the system detects the need to increase the current number of its replicas.

Experiments have been conducted on the Google Cloud Platform (GCP) with applications that process Twitter or DNS-traffic stream traces. We have evaluated our predictive adaptive SPS with different configurations and compared it with the original Storm and the predictive SPS DABS-Storm [4], which also adjusts resource usage dynamically. Results related to metrics, such as latency, resource utilization, and the number of processed events, are presented and discussed.

The next sections are organized as follows. Section II summarizes some SPS concepts and definition. Section III introduces adaptive SPS using predictive system adaptation models. Section IV presents our adaptive SPS. Results of experiments carried out on GCP are presented and discussed in Section V while some existing adaptive SPS works from the literature. Finally, Section VI concludes the article and presents some future directions.

II. Stream processing systems

The goal of SPS is to process high volumes of data in real-time [5]. The DAG defines the processing logic of the SPS where each vertex represents an operator, and unidirectional edges represent the data flow. An operator is usually a lightweight task (e.g., filtering, counting, etc.) that, based on the DAG, receives one or more dataflow, processes them, and sends the processed data over its output DAG edges. An operator can be classified as either *stateless* or *stateful*. The former handles each event completely independent from the preceding ones while the latter keeps a state based on previous processed events and, therefore, past events can influence the way current events are processed. Furthermore, an operator can have several replicas, and each replica of the operator is associated with a thread. Thus, they can process the data in parallel. A data source provides the input raw data stream to be processed by the operators over the DAG. Raw data are homogeneous [6], i.e., a set of structures (tuples) identified by key-values.

In the presence of replicas, the data flow is partitioned and shared among them. For instance, in the Shuffle Grouping approach, tuples are sent randomly to each replica, while in the Field Grouping approach, the tuple's key determines which replica will receive it. The drawback of this approach is the potential lack of load balance. To cope with such a problem, other existing SPS propose hash-based data partition [7], partial-key based [8] or executor-centric [9] solutions.

Figure 1 shows an SPS logical design (DAG) example, composed of an input data source, three replicated operators, and three edges. The source is in charge of sending the raw data to the operator O_1 by distributing the stream data input to each of O_1's replicas, based on some partition approach. After processing the data, O_1 sends the result data to O_2, its neighbor in the oriented DAG. Every operator processes the received data and sends the processed output data to the next, based on the grouping technique. If there is no adjacent neighbor, the data flow is terminated. The last operator that processes the data in the figure is O_3.

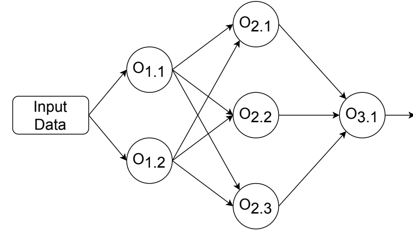


Fig. 1: The SPS logical architecture.

The processing logic (DAG) must be mapped to a physical environment for its execution, usually nodes (machines, cores, VMs) of a distributed platform such as a Grid, Cluster, or Cloud. The scheduling algorithm is in charge of performing the mapping. In this process, load balance problems may arise. For instance, when a random policy is applied (as Storm does), there is no guarantee that the workload will be homogeneously distributed among the operators since an operator may be more complex than another one [10] and machine resources could be heterogeneous. Even with homogeneous computation nodes, unbalance issues can also arise [11]. According to some scheduling algorithms, each operator replica is placed in an available node.

A. Storm

Storm [12] is an SPS framework implemented in Java that enables the processing of unbounded data flows. A Storm application is a DAG, denoted *topology*.

There are three types of components in a topology: *Streams*, *Spouts*, and *Bolts*. *Streams* or data flows are shared among operators following the DAG model. They are composed of key-value tuples. *Spouts* are responsible for capturing the input data of the topology from external sources. They structure the tuples sending them through one or more *Streams* to the next components of the topology. *Bolts* are the operators. Similarly to *Spouts*, *Bolts* can send the processed tuples through one or more *Streams*. At runtime, operators of the topology are executed by several threads called *executors*, which are instances of the operators.

The architecture comprises Storm and Zookeeper clusters. The Storm cluster contains a master node, called Nimbus, and Supervisor nodes. The latter provides a fixed number of processes, called workers, that run executors. The Nimbus is responsible for distributing the application code across the cluster, scheduling executors to available workers, monitoring the state of nodes, and detecting failures. Zookeeper provides a distributed coordination service enabling communication among Storm cluster nodes, load balance, and fault tolerance.

III. Related Work

In this section, we focus on existing works of the literature that, similar to ours, propose predictive SPS solutions.

DABS-Storm, a congestion prevention SPS, is presented in [4]. Its aim is to reduce the degradation of the quality of the results. To this end, a metric is used to estimate the level of activity of the operators. A monitor gathers statistics about the operators activity and then, based on a metric, decides if the amount of resource allocated to each operator should be modified or not. Such a metric is defined by predicting the system input by using a regression function as well as taking into account pending events. The capacity of the operators is also estimated, considering both the physical capacity of the machine where the operator is located and the latency of the system. As DABS-Storm has been implemented in Storm, its operators reconfiguration approach carries the drawback of Storm reconfiguration downtime cost, contrarily to our SPS that avoids it with the pre-allocated pool of inactive replicas.

The authors in [13] propose a predictive model implemented in Borealis SPS [14], taking into account not only the input rate as a metric, but also the capacity of the nodes as well as data processing complexity. Then, the model provides an equation that characterizes the workload of the system and determines the amount of required parallelism for processing events. Therefore, its objective is both the balance of the workload between the nodes and the reduction of latency. Although the system is capable of scaling-out, it does not perform scale-in, so it does not consider the reduction of allocated resources that our SPS provides.

Based on look-ahead approach, PLAStiCC is a predictive scheduling proposed by [15]. Its model analyzes the system performance through the balance of resource overload. Furthermore, as it is conceived to run on clouds, allocated resources can have different costs. Therefore, the model considers not only the workload of the system, but also the costs associated with the increase in resources. Contrarily to our experiments which were conducted on the public cloud GCP with real Twitter traces, the work uses for evaluation the cloud simulator CloudSim [16], as well as synthetic dataflows.

The Elastic-PPQ SPS [17] proposes to analyze the system at short-term and medium/long-term levels. The first one performs an analysis on the events that arrive in a time interval while the second one takes into account longer periods to perform a more complex analysis, using Fuzzy Logic Controller. To this end, an autonomous system, based on QoS, manages the system resources according to a runtime strategy, which considers the complexity of the system components. In this way, the parallelism of the tasks, associated with a set of threads, can increase or decrease. For the evaluation and validation of system load analysis, both synthetic and real data were used. Although the solution is quite robust, since it is implemented in FastFlow [18] framework, its focus is more on high performance processing than on distributed data processing.

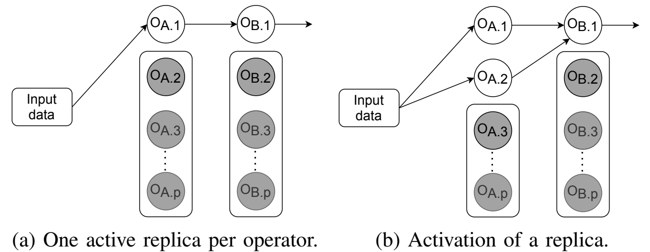
The predictive MEAD SPS [19] was implemented in Flink [20]. Operator auto-scaling takes place based Markovian Arrival Processes approach, where the system load is analysed according to a queuing model. The SPS proposes a MAPE-K for the control flow. Evaluation experiments used both synthetic and real environments. However, if the authors state that the MEAD supports operators scaling-out, such a feature has not been implemented. On the other hand, similar to our SPS and other works that use Flink, such as [21], reconfiguration does not induce performance degradation.

IV. Our predictive Storm-based SPS

Our predictive model aims to process all input events, reacting quickly and dynamically to system adaptation requirements due to input data fluctuation.

Based on DAGs composed of stateless operators, our SPS provides a predictive algorithm that dynamically estimates, for each operator, the current number of replicas necessary for processing as much as possible the received incoming events. The prediction of an operator number of replicas depends both on the dynamics of event input rate and the operator capacity to process events.

At initialization, our SPS assigns, for each operator, a set (pool) of replicas deployed by the Storm scheduler.



(a) One active replica per operator. (b) Activation of a replica.

Fig. 2: Per operator replica pool example.

Replicas can be either in *active* or *inactive* state. An inactive (resp., active) replica consumes negligible (resp., consumes) CPU power and can be dynamically activated (resp., deactivated) whenever the prediction model of our SPS detects the need for increasing (resp., decreasing) the resources for the operator in question. Figure 2 considers a DAG with the operators O_A and O_B and their respective replica pools. In Figure 2 (a), there is only one active replica per operator while in Figure 2 (b), a inactive replica of O_A has been activated.

$$r_i(t+1) = \frac{\widehat{\lambda_i}(t+1) \times et_i}{td} \tag{1}$$

$$\widehat{\lambda_i^r}(t+1) = \lambda_G(t) \times \theta_i(t) \tag{2a}$$

$$\theta_i(t) = \sum_{p \in pred(O_i)} \theta_i^p(t) \times \theta_p(t) \tag{2b}$$

$$\theta_i^p(t) = \frac{\lambda_i^p(t)}{\mu_p(t)} \tag{2c}$$

$$\widehat{\lambda_i^q}(t+1) = |q_i(t)| \tag{3}$$

$$\widehat{\lambda_i}(t+1) = \widehat{\lambda_i^r}(t+1) + \widehat{\lambda_i^q}(t+1) \tag{4}$$

Table I summarizes all the notations used by our predictive model where $\widehat{v}$ designates a *predicted* value of variable v.

Execution time is divided into equal intervals identified by t whose duration is td in which the statistics of all the operators are collected. The value of value must be greater than the average DAG execution time of one event.

At the end of each interval, the number of active replicas of an operator O_i for the next time interval is dynamically recalculated by Equation 1. The objective of this equation is to estimate how many active replicas would be necessary for O_i to process all $\widehat{\lambda}_i(t+1)$ estimated events within $t+1$, considering that O_i processes each event in et_i units of time. The value of et_i has been calculation with a benchmark at the beginning of the deployment of the application.

Parameter	***Description***
O_i	operator i
t	time interval number
td	time interval duration
et_i	average execution time of one event by O_i
$q_i(t)$	queue of events received and not processed by O_i at the end of t
$\lambda_G(t)$	number of events sent by input data during t
$\lambda_i^r(t)$	number of events received by O_i during t
$\lambda_i^p(t)$	number of events received by O_i sent from O_p during t
$\mu_i(t)$	number of events processed by O_i during t
$\theta_x(t)$	percentage of events processed of $\lambda_G(t)$ by O_x during t
O_i^p	predecessor operator of O_i in the SPS DAG
$\theta_i^p(t)$	percentage of events produced by O_i^p sent to O_i during t
$\widehat{\lambda}_i(t+1)$	predicted number of events to process by O_i during $t+1$
$\widehat{\lambda_i^r}(t+1)$	predicted number of events received by O_i during $t+1$
$\widehat{\lambda_i^q}(t+1)$	predicted number of queued events to be processed by $O_i$during $t+1$
$r_i(t+1)$	number of replicas of O_i computed at the end of t

TABLE I: Parameters notation and their description.

For instance, let's consider that the time interval duration (td) equals to 1000 ms. Figure 3 shows an example composed by three independent operators (O_1, O_2, and O_3) which receive the same input number of events $\widehat{\lambda}_i(t+1)$ to process. However, they have different event execution time et_i. Initially, at the beginning of t, all the three operators have two replicas. However, due to et_i's differences, Equation 1 will render $r_1(t+1) = 2$, $r_2(t+1) = 1$, and $r_3(t+1) = 4$ at the end of t. Such results inform that the number of O_1's active replicas should not change but that of O_2 (resp., O_3) is overestimated (resp., underestimated) and should be reduced (resp., increased) to one (resp., four).

Note that the input number of events $\widehat{\lambda}_i(t)$ will be distributed among the active replicas $r_i(t)$ of O_i by applying our grouping policy (see Section IV-A). The value of $\widehat{\lambda}_i(t+1)$ is determined by Equation 4, which in turn is determined by Equation 2.

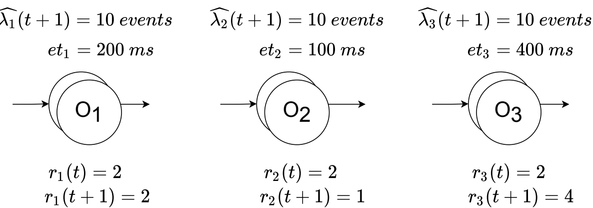


Fig. 3: Example of the number of replicas calculation, according Equation 1

Since operators of the SPS are related to each other by the DAG, there exists a dependency between sent and processed events, as shown in Figure 4, a linear SPS DAG with two operators, O_1 and O_2, and their respective values of $\lambda_i^r(t)$ and $\mu_i(t)$. (See Table I). $\mu_1(t)$ and $\lambda_2^r(t)$ are equal since operator O_1 has sent all the events it has processed to its single successor O_2. If i is the initial single DAG operator, then $\lambda_p^r(t) = \lambda_G(t)$ ($\lambda_1^r(t) = \lambda_G(t)$). Note that the increase of O_p's number of active replicas at the end of the interval t has a direct impact in O_p's successors, since, in this case, $\mu_p(t+1)$ increases and thus, $\lambda_i^r(t+1)$ too, inducing a domino effect that the prediction formulations should avoid. For example, in Figure 4, if $\mu_1(t+1)$ increased from 5 to 10 events due to replication of O_1 at the end of t, $\lambda_2^r(t+1)$ would increase as well. Hence, if the operators process all received events during $t+1$, we have that $\lambda_G(t+1) = \lambda_1^r(t+1) = \mu_1(t+1) = \lambda_2^r(t+1)$ and, consequently, all operators O_i are dependent on $\lambda_G(t)$.

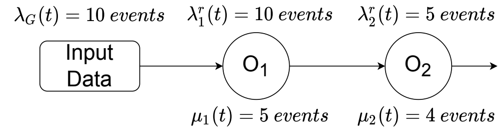


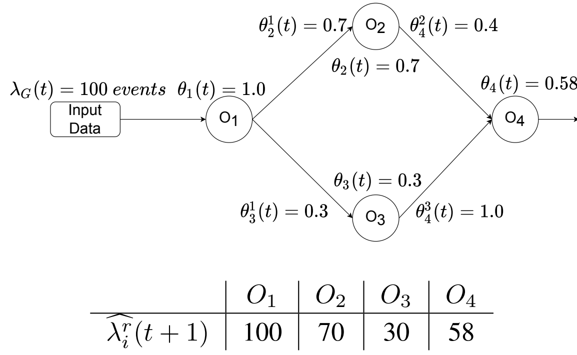
Fig. 4: DAG operators dependence example.

In a SPS execution, not always all the output processed events of O_i^p, the predecessor operator of O_i, will be sent to the latter. It might happen that O_i^p splits, filters, or replicates the events into several streams, sending each of them to one of its different successor operators in the DAG. The θ_i^p parameter informs the percentage of processed events of O_i^p sent to O_i. Its value is calculated by Equation 2c.

Figure 5 shows a DAG SPS with the respective values of Equation 2a for each operator. We observe that, since θ_1 is equal to 1, O_1 receives all the events sent from the input date and then splits them among O_2 and O_3. As these two operators do not receive all the events from its predecessor O_1, θ_1 and θ_1 have values 0.7 and 0.3 respectively. Finally, operator O_4 receives the events of its predecessor O_2 and O_3. However, O_2 does not send all its processed events to O_4, but only $\theta_4^2 = 0.4$, unlike O_3 which sends all processed events to O_4 ($\theta_4^2 = 1$). The value of θ_4 is, therefore, 0.58, according to Equation 2b.

Events received and not processed by O_i are kept in $q_i(t)$. Hence, the number of input events $\lambda_i(t)$ that O_i should actually process in t is composed not only of received events $\lambda_i^r(t)$ but also the events queued in O_i, which correspond to $\lambda_i^q(t)$. $\lambda_i^q(t)$ is defined as the number of events queued to process in O_i during t, considering there is one queue per operator. Thus, the predicted value of $\widehat{\lambda_i^q}(t+1)$ is defined by the events queued by O_i at the end of the time interval t as defined in Equation 3.

Figure 6 shows a linear DAG SPS, with operators O_1, O_2, and O_3. The values of the parameters obtained in the time interval t are presented in Table II (a). The calculation of $r_i(t+1)$ (Equation 1) requires the value of $\widehat{\lambda}_i(t+1)$ (Equation 4), which in its turn is defined by $\widehat{\lambda_i^r}(t+1)$ (Equation 2a) and



	O_1	O_2	O_3	O_4
$\widehat{\lambda_i^r}(t+1)$	100	70	30	58

Fig. 5: DAG example of predicted received number of events, according to Equation 2

.

$\widehat{\lambda_i^q}(t+1)$ (Equation 3). In order to obtain the value of $\widehat{\lambda_i^r}(t+1)$, the value of $\theta_i(t)$ for each O_i needs to be computed. Therefore, given the dependence between the operators, it is necessary to start from the initial operators to the last one. The value of each θ_i^p is defined in Figure 6, which were calculated according to Equation 2c. Finally, having obtained the predicted values, the number of replicas $r_i(t+1)$ for each operator O_i can be calculated. Such values are presented in Table II (b).

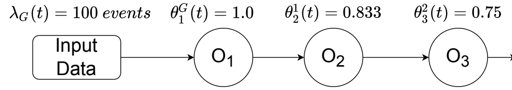


Fig. 6: Predictive analysis example for the SPS DAG.

	et_i	$\lambda_i^r(t)$	$\lambda_i^q(t)$	$\mu_i(t)$	$q_i(t)$	$\theta_i(t)$
O_1	16.6 ms	100	40	140	0	1.0
O_2	25 ms	117	10	120	7	0.833
O_3	100 ms	90	20	90	20	0.625

(a) Parameters values calculated for a time interval t.

	$\widehat{\lambda_i^r}(t+1)$	$\widehat{\lambda_i^q}(t+1)$	$\widehat{\lambda}_i(t+1)$	$r_i(t+1)$
O_1	100	0	100	2
O_2	84	7	91	3
O_3	63	20	83	9

(b) Parameters values predicted using Equation 1.

TABLE II: Analysis example for DAG presented in Figure 6.

A. Grouping

In shuffle grouping, the received input events are evenly distributed among active replicas without considering that the loaded replicas can receive new events while pending events are in their queue. A replica j of an operator i is defined as $O_{i.j}$

Parameter	***Description***
$\mu_{i.j}(t)$	number of events processed by $O_{i.j}$ during t
$U_{i.j}(t)$	utilization rate of $O_{i.j}$ computed at the end of t

TABLE III: Parameters notations of the grouping algorithm.

$$U_{i.j}(t) = \frac{\mu_{i.j}(t) \times et_i}{td} \tag{5}$$

To overcome such a constraint, we propose the *Load balancing grouping* strategy, which considers the load of active replicas in each time interval (t). In this way, the distribution of events is proportional to the load of active replicas, computed using Equation 5, where U is a value between 0 and 1: 0 informs that the replica has no load, and 1 the 100% utilization of the replica. If the value of U is the same for all replicas, the subsequent events are sent in round-robin. If not, these new events are sent to the replica with the lowest load. Algorithm 1 shows the pseudo-code of the *Load balancing grouping* which selects the active replica that will process the next event. We point out that the grouping implementation in Storm requires an independent queue for each active replica of an operator.

Algorithm 1 Load balancing grouping for operator O_i.

Require: Statistics of r_i replicas of O_i in interval t.
Ensure: Replica $O_{i.m}$ that should process the event.

```
1:  m ← 0
2:  for j : 1 → r_i do
3:      if U_{i.j} < U_{i.m} then
4:          m ← j
5:      end if
6:  end for
7:  if U_{i.m} = 1 then
8:      m ← getReplicaRoundRobin(O_i)
9:  else
10:     U_{i.m} ← U_{i.m} + et_i/td
11: end if
12: sendEvent(O_{i.m})
```

B. MAPE implementation

The MAPE loop control is in charge of providing the self-adaptation feature of our SPS. Each of the four MAPE modules performs a specific task:

1) *Monitor*: a module in charge of gathering and centralizing statistics from the DAG. At each time interval, the monitor requests the values of $\lambda_i(t)$, et_i, and the number of queued events q_i.
2) *Analysis*: a module in charge of computing Equation 4 in order to get $\lambda_i(t)$. Note that the analysis will be performed from the beginning of the graph till the last operator.
3) *Plan*: module that, based on the previous analysis and the current number of active replicas of an operator, defines whether it is necessary to modify the operator's current number of active replicas. Algorithm 2 shows the pseudo-code of the *Plan* module, responsible for increasing/decreasing the current number of active replicas, if necessary. The $getReplicas(O_i)$ function returns the number of current active replicas of O_i.
4) *Execute*: a module which is in charge of carrying out the change in the current number of replicas of an operator, if required by the *Plan* module.

V. Performance Evaluation

This section presents performance results related to the evaluation of our predictive SPS and its ability to adapt to

Algorithm 2 Adaptive Plan algorithm for operator O_i.

Require: Statistics Operator O_i in time interval t.
Ensure: Modifying the current number of active replicas of operator O_i.

1: $r_i(t+1) \leftarrow$ computeReplicas($\widehat{\lambda}_i(t+1)$, et_i, td)
2: $k_i \leftarrow r_i(t+1)$ - getReplicas(O_i)
3: **if** $k_i > 0$ **then**
4: Add k_i active replicas to O_i
5: **else if** $k_i < 0$ **then**
6: Remove k_i active replicas from O_i
7: **end if**

the dynamics of the event stream, without reducing the rate of processed events, as well as comparisons with other SPSs of the literature. In Section V-A, we introduce our system settings and the use-case application which analyses tweets streaming.

The evaluation is composed of five parts: (1) an analysis of the impact of the system parameters (Section V-B); (2) a comparison of different grouping strategies (Section V-C); (3) a comparison of our system with both the original Storm using a fix number of replicas (Section V-D) and with (4) the DABS-Storm adaptive solution proposed by [4] (Section V-F); and finally (5) evaluation results of two other applications: a tweet-based more complex application and a second one that has DNS traffic as input streaming (Section V-G).

A. Testbed and experimental settings

1) Testbed: Experiments were conducted on Google Cloud Platform (GCP) using eleven Virtual Machines (VMs): three in charge of Zookeeper, seven as Supervisor nodes, and one for running both the Nimbus and our SPS. Two types of machines were used: a `n1-standard-1` (1 CPU, 2.2 GHz, 3.75 GB of RAM) machine for hosting Zookeeper VMs, the Nimbus, and the adaptive system, and a `n1-highcpu-8` (8 CPU, 2.2GHz, 7.2GB of RAM) machine for the Supervisors VMs.

2) Application and scenario: The same input rate has been used and applied for all the experiments except the last one. We deployed an application composed of four operators in charge of analyzing and classifying tweet events, as shown in Figure 7. The events (tweets) were previously collected from Twitter and extracted with Twitter API. Events classification is based both on the content of the tweets and the identity of the person who has published the *tweet*. The analysis of the tweets are stored in the database

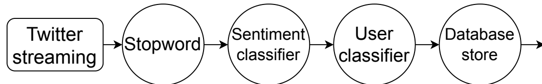


Fig. 7: Twitter application in SPS.

The traffic model is based on data from Twitter related to COVID-19 with 237 million tweets [22]. However, for our experiments, we have used only a sample of these tweets. The sample of selected tweets considers periods of the datasets that present high variation. In other words, we select a combination of traffic spikes and under spikes. The methodology for the creation of the testing data set is presented in [23].

3) Metrics: We have defined four evaluation metrics.

- *Saved resources*: proposed in [24], this metric expresses the proportion of resources (active replicas) saved with respect to a statically over-provisioned configuration.. It is defined by $1-\frac{r}{r_{over}}$, with r the number of active replicas, and r_{over} the overestimated number of replicas. r_{over} is the number of replicas needed to process all the events during the highest input rate peak of the benchmark. Note that if the value of the metric is close to 1, a high number of resources has been saved.
- *Throughput degradation*: this metric, also described in [24], aims at analyzing the behavior of the system in terms of throughput stability. It is defined by $\frac{|input_{rate}-output_{rate}|}{input_{rate}}$. If the metric value is close to 0, the system has a good stability. On the other hand, if it is close to 1, the system is not capable to process the input rate and the system is unstable.
- *Latency*: is the average time taken by an event between the moment it enters and leaves the SPS (end-to-end latency). This metric is relevant since SPSs are supposed to deliver real-time processed events.
- *Difference in the number of processed events*: is the difference between the total number of processed events and the total number of received events. It is an important metric since SPSs are used to process high volumes of data.

B. Impact of the parameters

Aiming at tuning their value, we propose in this section to discuss the impact of the three SPS parameters in the metrics. The parameters are: time interval duration (td), timeout to detect the failure of an event (t_{out}), and queue size (q_{size}).

We consider that an event has failed when its processing time on all operators exceeds the end-to-end timeout, t_{out}. This parameter removes events that have been queued for a long time, reducing the load on an operator's replicas.

By Analyzing the Twitter dataset, the value of r_{over} was set to 32.

Table IV shows the values of the four metrics when the value of td varies. We set the value of t_{out} to $30s$ and of q_{size} to 100000. Note that the greater the time interval, the greater the number of samples used for calculating Equation 1. We observe an improvement in the results when the time interval is small, which is also in accordance with the dynamic behavior of the input rate. Latency and throughput degradation confirm the latter, given that by increasing td the system needs to wait longer to adjust the number of replicas and stabilize.

It is important to highlight that, unlike Storm's traditional solution, which must restart the application to reconfigure the number of resources of each operator, our SPS should only activate or deactivate replicas in the pool. Therefore, the reconfiguration downtime does not exist. Furthermore, the computational cost of calculating the equations is minimal since they are basic operations carried out by the system.

Consequently, even if reconfiguration occurs quite often, we do not observe a decrease in the number of processed events.

Time interval	Saved Resources	Throughput Degradation	Diff. Proc. Events	Latency
$td = 30s$	0.5617	0.1831	0.9987	2098.91
$td = 60s$	0.5390	0.4332	0.9979	5271.01
$td = 120s$	0.5219	0.9221	0.9976	17068.33
$td = 180s$	0.5563	0.8028	0.9992	22364.86

TABLE IV: System metric values with different time intervals.

Having set the value of td to $30s$ and of q_{size} to 100000, Table V shows the four metrics, when the value of t_{out} varies. We observe that t_{out} has an impact in both latency and loss of processed events. For example, a $t_{out} = 1s$ improves the latency in 82.40% when compared to $t_{out} = 30s$ but, at the same time, there is a decrease of 9.5% in the difference of processed events. Therefore, on the one hand, if the proposed application does not require full data processing but low latency, one solution is to set the timeout to a low value for system deployment. On the other hand, if the application requires full event processing, it is recommended to use a high timeout value.

Timeout	Saved Resources	Throughput Degradation	Diff. Proc. Events	Latency
$t_{out} = 1s$	0.6164	0.1000	0.9030	369.35
$t_{out} = 5s$	0.5875	0.1038	0.9303	946.69
$t_{out} = 10s$	0.5633	0.1056	0.9529	1298.45
$t_{out} = 30s$	0.5617	0.1831	0.9987	2098.91

TABLE V: System metric values with different timeout.

Table VI summarizes the four metrics values for different sizes of the pending message queue, q_{size} when $td = 30s$ and $t_{out} = 30s$. The greater the queue size, the higher the number of queue events, thus reducing the loss rate and increasing the number of processing events ($Diff.\ Proc.\ Events$), as we can observe in the table.

On the other hand, since many incoming events are dropped when using small queues, operators are less loaded and we observe an increase of the saved resources (the number of replicas computed by Equation 1 is then smaller than the ones with big queue sizes).

For example, with $q_{size} = 100$, we observe a 19.61% improvement in saved resources and a 98.57% decrease of the latency compared to $q_{size} = 100000$. However, the number of dropped events highly increases, inducing a decrease of 46.12% in processed events.

Queue size	Saved Resources	Throughput Degradation	Diff. Proc. Events	Latency
$q_{size} = 100$	0.6719	0.1919	0.6835	29.91
$q_{size} = 1000$	0.6656	0.1375	0.6922	311.93
$q_{size} = 10000$	0.6602	0.1687	0.7249	1394.56
$q_{size} = 100000$	0.5617	0.1831	0.9987	2098.91

TABLE VI: System metric values with different queue size.

Based on the current study, in the following sections, we select parameter values for which our SPS processes the greatest number of events without significantly degrading latency. This means that $td = 30s$, $t_{out} = 30s$ and $q_{size} = 100000$.

C. Grouping

This section compares our load balancing grouping strategy with the shuffle grouping where events are randomly distributed among the replicas.

Table VII shows the metrics for both grouping strategies. There is no significant difference in terms of processed events and only an increase of 4.04% in saved resources using our grouping strategy. Regarding the use of VMs resources, CPU utilization increases by 0.9% with respect to a random distribution and the difference in memory usage is negligible.

On the other hand, there is a great difference in latency and throughput metrics of the two strategy since shuffle grouping does not take into account replicas' load. When a new replica is activated, the old loaded replica must process both the new events that arrive and the ones it previously queued, which explains a decrease of 60.18% in latency and 57.73% in throughput degradation.

Grouping	Saved Resources	Throughput Degradation	Diff. Proc. Events	Latency
Shuffle	0.5390	0.4332	0.9979	5271.01
Load balancing	0.5617	0.1831	0.9987	2098.91

TABLE VII: System metric values with different grouping.

D. Comparison with Storm

We have compared our SPS, denoted Predictive SPS (PSPS) with the original *Storm* where the number r of replicas per operator is fixed. We have considered three configurations for *Storm*: no replication ($r = 1$); four replicas ($r = 4$); eight replicas ($r = 8$). The latter corresponds to the *overprovisioning* configuration where the total number of replicas, r_{over}, is equal to 32 in our SPS.

Table VIII summarizes the results of the different configurations. When comparing the values for each metric, we observe considerable difference.

In configuration with no replication ($r = 1$), the input rate of events is too high and, therefore, Storm can not process all the events without additional replicas. Operators are overloaded, inducing a high average latency. Although few resources are used, such a configuration is not suitable to cope with the input rate.

On the other hand, in the configuration with $r = 4$, Storm can process a large number of events within each time interval, presenting a decrease of only 1.1% when compared to PSPS. Latency is 87.14% lower than PSPS since a constant number of replicas is always available. The difference in latency can be explained since in Storm the distribution of events is homogeneous whereas in PSPS events are distributed based on the load among the dynamic set of replicas where some of them may have queued events which will then take more time to be processed. However, in this case, Storm is more unstable compared to PSPS as shown by the throughput degradation which is much higher even if a great number of events can be processed. We also observe an increase of 10.98% in resource usage compared to PSPS.

Finally, in configuration with $r = 8$, Storm is able to process all incoming events within each time interval, which is reflected by a 0 throughput degradation. Since there are never queued events, Storm latency is lower than PSPS one. However, this configuration uses much more resources than the others and PSPS.

System	Saved Resources	Throughput Degradation	Diff. Proc. Events	Latency
PSPS	0.5617	0.1831	0.9987	2098.91
$r = 1$	0.8750	0.5153	0.3319	196962.95
$r = 4$	0.5	0.8559	0.9874	269.75
$r = 8$	0.00	0.00	1.00	153.51

TABLE VIII: System metric values with *DPSP* and *Storm*.

Figure 8 shows the ability of PSPS to dynamically modify the number of operator replicas. The latter follows the evolution of the events input rate as we can see in the inflection points at $t = 200s$, $t = 500s$, and $t = 900s$. At $t = 900s$, PSPS increases the number of replicas above $r_{over} = 32$ to process not only input events but also events queued on loaded operators.

The number of events processed per time interval in the three Storm configurations and PSPS is presented in Figure 9. As explained above, the configuration with $r = 1$ has a limited amount of resources. Consequently, the throughput rate is constant since it is impossible to process more events with the available resources. The same behaviour is observed in configuration with $r = 4$ at $t = 120s$, $t = 420s$, and $t = 800s$. The throughput rate is constant until replicas have ended to process the queued events, as seen at $t = 350s$ or $t = 700s$. However, in configuration with $r = 8$, the throughput rate is similar to the input rate because there are always enough available resources to process all events. On the other hand, PSPS tolerates peaks of load by queuing events during periods of adaptation and then processing them later, as we can observe at $t = 200$, $t = 500$, or $t = 950$. Therefore, our SPS dynamically adapts the number of active replicas to the input rate fluctuation in order to process the largest number of events.

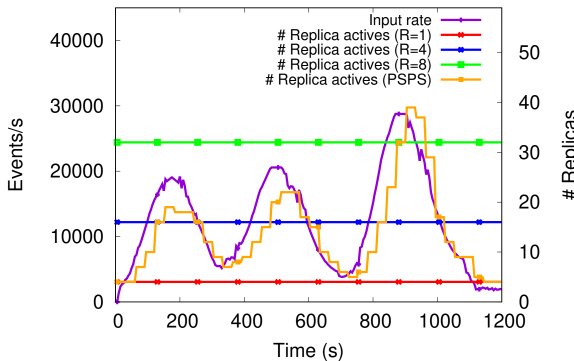


Fig. 8: Total number of replicas of *Storm* and *PSPS*.

E. Twitter raw stream

In order to evaluate the impact of having used in the previous experiments Twitter smoothed traces instead of the original one (*raw data*, described in V-A), we conducted an experiment with the same application but with Twitter raw data as input. We have considered $r_{over} = 32$ and $r = 1$ for Storm. Table IX shows the obtained results, which are globally consistent with those of Table VIII with smoothed data. However, in the case of raw data, PSPS is less stable and, therefore, many reconfigurations take place, inducing an increase of the throughput degradation as well as a decrease of the number events processed.

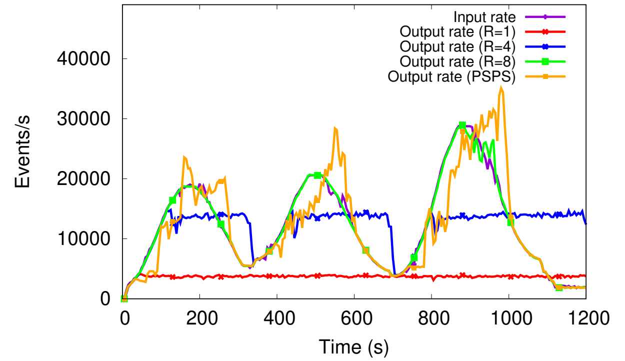


Fig. 9: Throughput of *Storm* and *PSPS*.

Nevertheless, we observe that PSPS improves by 246.95% the proportion of processed events compared to Storm with no replication ($r = 1$). Also, PSPS is more stable, since the throughput degradation has an improvement of 40.35%. As the system is more stable, PSPS processes a larger amount of queued events, thus, the latency decreases by 94.82%.

System	Saved Resources	Throughput Degradation	Diff. Proc. Events	Latency
PSPS	0.745	0.4	0.9791	2501.46
$r = 1$	0.875	0.6706	0.2822	48379.95

TABLE IX: Twitter raw stream metric values.

F. Comparison with DABS-Storm

Table X gathers the metric values related to PSPS and DABS-Storm, the SPS proposed in [4] (see Section III). Regarding the used resources, PSPS improves the number of them by 41.77% when compared to DABS-Storm. In addition, the throughput degradation of PSPS is 35.73% lower than DABS. The most important differences are in latency and the number of processed events. As DABS needs to restart the system at every reconfiguration, operator queues are emptied and their events are dropped. On the other hand, in PSPS, there exists a pool of replicas and it is only necessary to activate or deactivate the pre-allocated replicas at each reconfiguration, keeping the existing pending events of the queues. Hence, since in DABS-Storm the queued events are not processed during reconfiguration, we observe a decrease in both the number of processed events and latency: DABS processes 17.06% fewer events than PSPS and latency is decreased by 33.71%.

Figure 10 shows the number of replicas used by the two SPS. DABS and PSPS can dynamically adapt the number of replicas according to the input rate. However, DABS downtime

System	Saved Resources	Throughput Degradation	Diff. Proc. Events	Latency
PSPS	0.5617	0.1831	0.9987	2098.91
DABS	0.3962	0.2849	0.8283	1391.28

TABLE X: PSPS and DABS-Storm metric values.

at each reconfiguration has a direct impact in the throughput, as we can observe in Figure 11, inducing a higher instability and a decrease in the number of processed events.

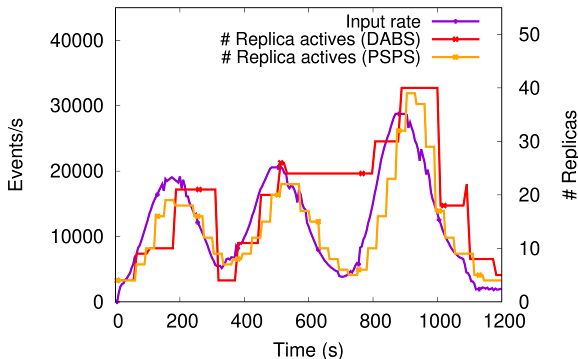


Fig. 10: Total number of replicas of *PSPS* and *DABS-Storm*.

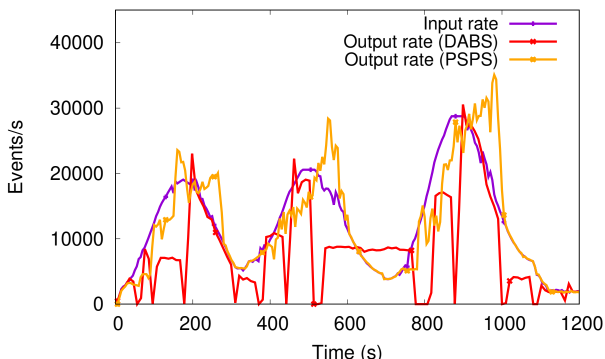


Fig. 11: Throughput of *PSPS* and *DABS-Storm*.

G. Other applications

1) Complex application: We have also evaluated PSPS with a more complex application, whose DAG is represented in Figure 12. It analyzes Twitter streaming containing information such as news or opinions. Depending on the type of information, the flow can be splitted. Result are stored in a database. In the experiments, $r_{over} = 40$.

Table XI shows evaluation results obtained with DPS and Storm with a fixed per operator number of replicas of five ($r = 5$) for Storm. In PSPS, we observe a high reduction of used resources with 74.5% fewer active replicas, when compared to Storm. Such a decrease has an impact on the physical used resources: CPU consumption of PSPS (resp. Storm) is in average, 8.41% (resp. 14.66%). This difference happens because each replica is associated with a thread. Therefore, having a fixed number of 5 replicas, Storm requires more CPU than PSPS where the number of replicas dynamically varies.

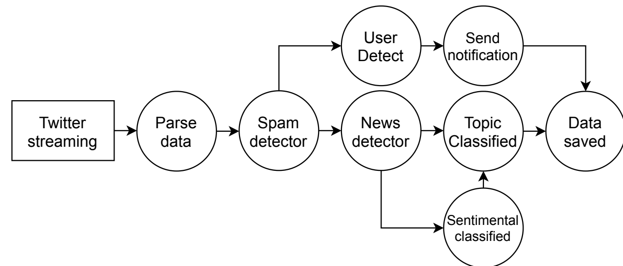


Fig. 12: A Twitter more complex application in *PSPS*.

Complex App	Saved Resources	Throughput Degradation	Diff. Proc. Events	Latency
PSPS	0.745	0.0911	0.9970	219.13
$r = 5$	0.0	0.0	1.0	31.99

TABLE XI: Complex application metric values.

2) DNS traffic data stream: We deployed an application composed of four operators for analyzing and classifying events based on DNS traffic, as shown in Figure 13. In the experiments, we use the dataset presented in [25] and fixed $r_{over} = 12$ (i.e., $r_i = 3$).

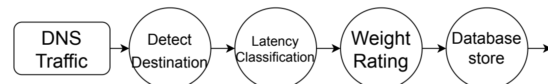


Fig. 13: DNS application in PSPS.

The aim of the experiment is to verify that PSPS also adapts to other types of input rates. Table XII summarizes the obtained results, which are in accordance with PSPS proposal to process the greatest number of events: the percentage of unprocessed events is only 1.16%. PSPS also improves latency, decreasing it by 29.44% when compared to Storm with no replication ($r = 1$). We also observe an improvement in system stability, since the throughput degradation of PSPS is 17.40% lower than the latter.

Scenario	Saved Resources	Throughput Degradation	Diff. Proc. Events	Latency
$r = 1$	0.6666	0.4521	0.9759	476.12
$PSPS$	0.6062	0.3734	0.9884	335.95

TABLE XII: DNS scenario metric values.

Figure 14 shows both the input rate and the throughput of PSPS. Despite the high dynamics of the input rate, PSPS is able to adapt its resources in order to process the largest number of events in each time interval. Such adaptation ability is positively reflected on the throughput degradation, whose value is closer to 0 (see Table XII) which confirms the stability of PSPS. There is also an improvement in the use of resources with a decrease of 60.62% compared to an overestimated replica configuration.

VI. Conclusion

We have presented in this article a predictive Storm-based SPS, which, in order to cope with input data fluctuation,

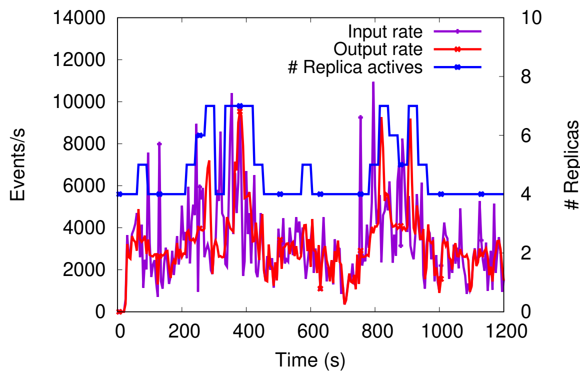


Fig. 14: Total number of replicas and throughput of *PSPS* with DNS traffic.

dynamically adapts the number of active replicas of the DAG operators. To do this end, we have proposed a set of equations that, firstly deduce the number of incoming events and then predict the number of such replicas, as well as the *Load Balancing* grouping strategy which is based on current replicas load. Evaluation results confirm the effectiveness of both our predictive and grouping approaches and that our SPS outperforms Storm and DABS-Storm SPSs. In the experiments, the *Load Balancing* grouping reduced latency by 60.18%, when compared to the traditional Storm grouping, while PSPS reduced used resources by 41.77% and increased the number of processed events by 20.57%, when compared to DABS-Storm.

In future work, we intend to evaluate our solution against an existing benchmark, such as [26] and also predict the input rate, exploiting a mathematical model. In this case, the estimation of the number of active replicas would be based on values given by the model, using two intervals for the analysis of the SPS, as well as the queued events of the predecessor operators for the prediction.

Acknowledgement

This work was funded by the National Agency for Research and Development National Agency for Research and Development (ANID) / Scholarship Program / DOCTORADO BECAS CHILE / 2018 - 72190551. This material is based upon work supported by Google Cloud. Nicolas Hidalgo wants to thank the project CONICYT FONDECYT Nº 11190314, Chile and to STIC-AmSud ADMITS Nº 20-STIC-01.

References

[1] J. Leibiusky, G. Eisbruch, and D. Simonassi, *Getting Started with Storm - Continuous Streaming Computation with Twitter's Cluster Technology*. O'Reilly, 2012.

[2] S. Chakravarthy and Q. Jiang, *Stream Data Processing: A Quality of Service Perspective - Modeling, Scheduling, Load Shedding, and Complex Event Processing*, ser. Advances in Database Systems. Kluwer, 2009, vol. 36.

[3] A. Computing *et al.*, "An architectural blueprint for autonomic computing," *IBM White Paper*, vol. 31, no. 2006, pp. 1–6, 2006.

[4] R. K. Kombi, N. Lumineau, P. Lamarre, N. Rivetti, and Y. Busnel, "Dabs-storm: A data-aware approach for elastic stream processing," *Trans. Large Scale Data Knowl. Centered Syst.*, vol. 40, pp. 58–93, 2019.

[5] M. Kleppmann, *Making Sense of Stream Processing*. O'Reilly Media, Incorporated, 2016.

[6] H. Andrade, B. Gedik, and D. Turaga, *Fundamentals of Stream Processing: Application Design, Systems, and Analytics*. Cambridge University Press, 2014.

[7] M. A. Shah, J. M. Hellerstein, S. Chandrasekaran, and M. J. Franklin, "Flux: An adaptive partitioning operator for continuous query systems," in *ICDE*. IEEE Computer Society, 2003, pp. 25–36.

[8] M. A. U. Nasir, G. D. F. Morales, D. Garcia-Soriano, N. Kourtellis, and M. Serafini, "The power of both choices: Practical load balancing for distributed stream processing engines," in *ICDE*. IEEE Computer Society, 2015, pp. 137–148.

[9] L. Wang, T. Z. J. Fu, R. T. B. Ma, M. Winslett, and Z. Zhang, "Elasticutor: Rapid elasticity for realtime stateful stream processing," in *SIGMOD Conference*. ACM, 2019, pp. 573–588.

[10] Y. Xing, S. B. Zdonik, and J. Hwang, "Dynamic load distribution in the borealis stream processor," in *Proceedings of the 21st International Conference on Data Engineering, ICDE 2005, 5-8 April 2005, Tokyo, Japan*, 2005, pp. 791–802.

[11] 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*. IEEE Computer Society, 2014, pp. 535–544.

[12] A. Toshniwal, S. Taneja, A. Shukla, K. Ramasamy, J. M. Patel, S. Kulkarni, J. Jackson, K. Gade, M. Fu, J. Donham *et al.*, "Storm@ twitter," in *Proceedings of the 2014 ACM SIGMOD international conference on Management of data*. ACM, 2014, pp. 147–156.

[13] C. Balkesen, N. Tatbul, and M. T. Özsu, "Adaptive input admission and management for parallel stream processing," in *DEBS*. ACM, 2013, pp. 15–26.

[14] D. J. Abadi, Y. Ahmad, M. Balazinska, U. Çetintemel, M. Cherniack, J. Hwang, W. Lindner, A. Maskey, A. Rasin, E. Ryvkina, N. Tatbul, Y. Xing, and S. B. Zdonik, "The design of the borealis stream processing engine," in *CIDR*. www.cidrdb.org, 2005, pp. 277–289.

[15] A. G. Kumbhare, Y. Simmhan, and V. K. Prasanna, "Plasticc: Predictive look-ahead scheduling for continuous dataflows on clouds," in *CCGRID*. IEEE Computer Society, 2014, pp. 344–353.

[16] R. N. Calheiros, R. Ranjan, C. A. F. D. Rose, and R. Buyya, "Cloudsim: A novel framework for modeling and simulation of cloud computing infrastructures and services," *CoRR*, vol. abs/0903.2525, 2009.

[17] G. Mencagli, M. Torquati, and M. Danelutto, "Elastic-ppq: A two-level autonomic system for spatial preference query processing over dynamic data streams," *Future Gener. Comput. Syst.*, vol. 79, pp. 862–877, 2018.

[18] M. Aldinucci, M. Danelutto, P. Kilpatrick, and M. Torquati, "Fastflow: high-level and efficient streaming on multi-core," *Programming multi-core and many-core computing systems, parallel and distributed computing*, 2017.

[19] G. R. Russo, V. Cardellini, G. Casale, and F. L. Presti, "MEAD: model-based vertical auto-scaling for data stream processing," in *CCGRID*. IEEE, 2021, pp. 314–323.

[20] P. Carbone, A. Katsifodimos, S. Ewen, V. Markl, S. Haridi, and K. Tzoumas, "Apache flink™: Stream and batch processing in a single engine," *IEEE Data Eng. Bull.*, vol. 38, no. 4, pp. 28–38, 2015.

[21] H. Arkian, G. Pierre, J. Tordsson, and E. Elmroth, "Model-based stream processing auto-scaling in geo-distributed environments," in *30th Inter. Conference on Computer Communications and Networks*, 2021.

[22] A. Gruzd and P. Mai, "COVID-19 Twitter Dataset," 2020. [Online]. Available: https://doi.org/10.5683/SP2/PXF2CU

[23] P. Bodík, A. Fox, M. J. Franklin, M. I. Jordan, and D. A. Patterson, "Characterizing, modeling, and generating workload spikes for stateful services," in *SoCC*. ACM, 2010, pp. 241–252.

[24] F. Lombardi, L. Aniello, S. Bonomi, and L. Querzoni, "Elastic symbiotic scaling of operators and resources in stream processing systems," *IEEE Trans. Parallel Distrib. Syst.*, vol. 29, no. 3, pp. 572–585, 2018.

[25] M. MontazeriShatoori, L. Davidson, G. Kaur, and A. H. Lashkari, "Detection of doh tunnels using time-series classification of encrypted traffic," in *IEEE Intl Conf on Dependable, Autonomic and Secure Computing*. IEEE, 2020, pp. 63–70.

[26] A. Shukla, S. Chaturvedi, and Y. Simmhan, "Riotbench: An iot benchmark for distributed stream processing systems," *Concurr. Comput. Pract. Exp.*, vol. 29, no. 21, 2017.