Towards Evaluating Stream Processing Autoscalers

George Siachamis Job Kanis Wybe Koper Kyriakos Psarakis
Marios Fragkoulis* Arie van Deursen Asterios Katsifodimos
*Delivery Hero SE, Delft University of Technology
{initial.lastname}@tudelft.nl

2023 IEEE 39th International Conference on Data Engineering Workshops (ICDEW) | 979-8-3503-2244-6/23/$31.00 ©2023 IEEE | DOI: 10.1109/ICDEW58674.2023.00021

***Abstract*—In this work, we evaluate autoscaling solutions for stream processing engines. Although autoscaling has become a mainstream subject of research in the last decade, the database research community has yet to evaluate different autoscaling techniques under a proper benchmarking setting and evaluation framework. As a result, every newly proposed autoscaling solution only performs a shallow performance evaluation and comparison against existing solutions. In this paper, we evaluate autoscaling solutions by employing two streaming queries and a dynamic workload that follows a cosinus pattern. Our experiments reveal that current autoscaling techniques fail to account for generated lag due to rescaling or underprovisioning and cannot efficiently handle practical scenarios of intensely dynamic workloads.**

***Index Terms*—autoscaling, stream processing**

I. Introduction

In previous decades, data processing was mainly taking place in dedicated clusters of fixed resources. However, in the cloud, resources are not fixed, while different pricing schemes provide incentives for the dynamic provision of resources. In addition, the inclusion of spot instances allows for better utilization of idle resources with cost-saving incentives.

Most of the widely adopted stream processing engines (SPEs) were originally developed for deployment on clusters of fixed resources. As a result, these SPEs provide limited autoscaling capabilities and require substantial operational effort to adapt to changes in needs and workloads. An operations team has to always monitor the performance of the deployed system or application, estimate the required resources, decide whether to scale or not and perform the actual scaling manually. This process is time-consuming and can lead to slow reactions to workload changes with serious performance implications.

To provide automated solutions, specialized autoscalers have been developed, in order to equip SPEs with the missing self-managing capabilities. However, it remains unclear how these autoscalers perform in different practical scenarios, and under a proper comparison framework. We argue that without a principled and configurable experimental analysis, it is doubtful that these autoscalers will have the desired impact on modern stream processing engines.

In this paper, we report on the first step towards a principled and configurable framework that will allow for a comprehensive experimental analysis of autoscalers for SPEs. In short, the contributions of this paper go as follows:

Partially funded by the AI for Fintech Research Lab.

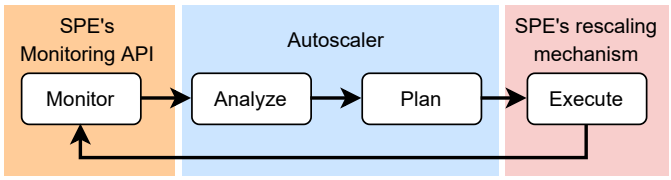

Fig. 1. MAPE loop for stream processing autoscaling

- We stress the importance of extensive experimental evaluation of autoscalers for stream processing.
- We reproduce state-of-the-art autoscalers for stream processing under a common framework.
- We present our preliminary experimental results over two NEXmark queries and a heavily dynamic workload.

II. Background

A. Autoscaling Process

The process of autoscaling resembles the MAPE loop from control theory. As depicted in Figure 1, the first step includes *monitoring* of a stream processing job and acquiring all the metrics needed both for the evaluation of its performance and for deciding on performing rescaling actions. After these metrics are collected, the *analysis* step takes place where we evaluate the current state of the job and calculate the job's needs to adhere to the enforced SLOs. The outcome of the analysis is then used from the *planning* step to decide on the proper rescaling actions. The goal is to satisfy the calculated needs while minimizing the resources employed. The last step is *executing* the devised plan. Responsible for retrieving the metrics is usually either the monitoring API of the SPE or any applicable monitoring tool, while execution usually falls on the SPE and its rescaling mechanism. The analysis and planning steps are handled from the *autoscaler*.

B. Common notions

Before discussing the selected autoscalers and the experimental results we discuss a few notions that we deem necessary for understanding what follows.

Task Managers & Operators. In this paper we use Apache Flink as our SPE. We choose Flink among other SPEs since it is the current state-of-the-art and the most widely adopted system in production, and it provides all the expected by the autoscalers mechanisms. A task manager is the fundamental processing unit of Apache Flink. By default, a task manager

2473-3490/23/$31.00 ©2023 IEEE
DOI 10.1109/ICDEW58674.2023.00021

runs multiple operators that share its resources. However, for this paper, we have configured Flink to isolate operators and assign a single operator to each task manager.

Back pressure. Back pressure is a rate control mechanism employed by many SPEs. When an operator cannot handle the input rate, the system uses the back pressure mechanism to regulate the output rate of the upstream operator. The backpressure can be propagated up to the source operator.

Lag & Latency. Lag is defined as the number of unprocessed records waiting in the input queue or the operator buffers. Latency is defined as the time a record spends in the input queue until the SPE ingests it.

III. AUTOSCALERS

There is plenty of research done in autoscaling for cloud computing [6], however, only a handful of the existing solutions can be applied or target stream processing. In this paper, we select for evaluation the state-of-the-art DS2 [5] and Dhalion [3], as the most cited and easily deployable solutions. We also consider DRS [4] and the solution from Varga et al. [8]. However, DRS requires metrics that are not natively supported by Flink and, as a queue-based model, it is not easily applicable for different queries. On the other hand, Vargas does not support per-operator autoscaling and it is not easily extensible to do so. Finally, we selected the Horizontal Pod Autoscaler [2] as both a simple baseline and an applied solution from a commercial product.

A. Dhalion

Dhalion [3] is a framework that provides self-regulating capabilities to underlying stream processing systems that employ a backpressure mechanism to perform rate control. It utilizes user-defined policies to handle performance issues related to different underlying causes, such as load skew, slow instances, and provisioning. In this work, we are only interested in its proposed policy for autoscaling. The policy distinguishes two cases: the overprovisioning and the underprovisioning case.

Overprovisioning. For an operator of a running job to be considered overprovisioned, two conditions must hold: (a) there is no backpressure anywhere in the pipeline, and (b) the input queue of the operator has a length of almost zero. For each operator considered overprovisioned, new parallelism is calculated using a provided *scale down factor*. At the end of the monitoring iteration, a rescaling operation is triggered, and the overprovisioned operators are scaled down.

Underprovisioning. If there is any backpressure along the pipeline the job is considered to be in an unhealthy state and underprovisioned. To resolve the issue, the first step is to identify the operator which is the root of the backpressure. Then a *scale up factor* is calculated for this operator based on the amount of time the job managed to process the input normally and the amount of time backpressure occurred over the monitoring window. More precisely, the *scale up factor* is provided by the following formula:

$$scaleUpFactor = \frac{backpressuredTime_{w_i}}{normalProcessingTime_{w_i}} \quad (1)$$

where w_i is the current monitoring window.

Since we consider Kafka as our source we also need to scale up/down Flink's KafkaSource operators. Since there is no backpressure information available for these operators we decided to use the increase of lag in Kafka as an indicator of backpressure caused by the KafkaSource operators. The *scale up factor* for these operators is calculated as:

$$scaleUpFactor_{KS} = \frac{pendingRecordsRate_{w_i}}{consumedRecordsRate_{w_i}} \quad (2)$$

where pendingRecordsRate is the average lag increase per second and consumedRecordsRate is the average number of records consumed per second over the monitoring window w_i.

We gather the needed metrics using the monitoring API of Flink and Prometheus. Since Flink does not report the input queue size of each individual operator we use the percentage of input buffers used to decide on the lag in the input queues.

The only tunable parameter for Dhalion is the scale down factor that is user-provided.

B. DS2

In contrast to Dhalion which scales each operator independently, DS2 [5] attempts to combine the scaling of all operators in a single step by leveraging the topology of the streaming query. To do so, it introduces the notions of *useful time*, *true processing rate*, and *true output rate*. *Useful Time* is the time spent by an operator in (de)serializing and processing records. *True processing rate* is the number of records an operator processes per unit of useful time, while *true output rate* is the number of records an operator outputs per unit of *useful time*. Based on these notions, DS2 calculates progressively the optimal parallelism of each operator o_i as follows:

$$OP_{o_i} = \frac{\sum \text{true output rate of upstream operators}}{avg(\text{true processing rate}) \text{ of } o_i} \quad (3)$$

In this work, in order to calculate the optimal parallelism for the KafkaSource operators, we use the rate at which records are written to Kafka as the *true output rate* of the upstream operators. In addition, we extend DS2 with a user-provided overprovisioning factor in order to help DS2 to handle noisy spikes and the lag accumulated due to scaling actions. This is also the only tunable parameter of DS2.

C. HPA

The Horizontal Pod Autoscaler (HPA) [2] is the default autoscaling solution shipped with Kubernetes. As its name implies, HPA tries to automatically scale horizontally a deployment by adding or removing pods, in order to match user-provided target values on the observed metric. The observed metric can be either the standard average CPU/memory utilization or any custom user-defined metric. HPA [2] attempts to match the user-provided target value of the metric by scaling up or down based on Equation (4).

$$desiredPods = \lceil currentPods \times \frac{currentMetricValue}{targetMetricValue} \rceil \quad (4)$$

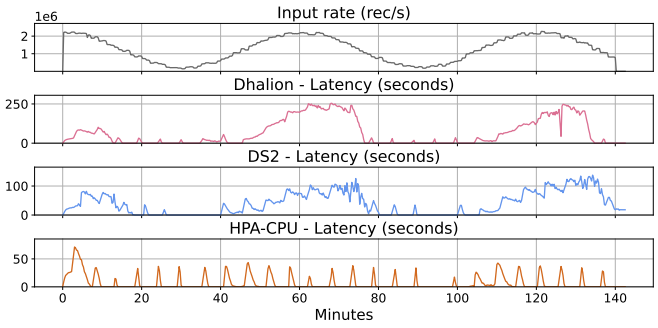

Fig. 2. Latency results of query 1.

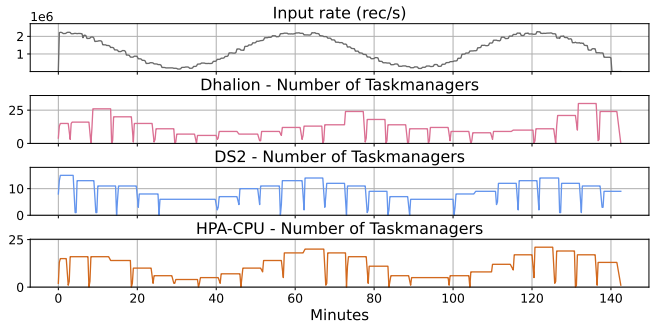

Fig. 3. Number of task managers deployed over time on query 1.

When scaling down HPA opts for a conservative approach. It records the scaling recommendations over a stabilization window and picks the highest recommendation as the desired amount of resources. This way it ensures a gradual scaledown that is not affected by fluctuations in the metric values.

HPA as a Streaming Topology Autoscaler. Since a given task manager in the streaming topology runs on an individual pod, HPA can be used as a basis for building a streaming topology autoscaler, that will add or remove task managers when required. HPA works over the task managers' deployment of Flink and is agnostic of the underlying operators. Thus, it cannot be used as an autoscaler for streaming topologies out of the box.

We now describe the changes that we applied to HPA in order to turn it into a streaming topology autoscaler. Our own version of HPA monitors the actual operators within a pod, instead of the deployment of the task managers. We employ as a metric the average CPU utilization. In what follows, we will refer to this custom version of HPA as HPA-CPU. HPA-CPU has two tunable parameters: the target value of the CPU utilization and the length of the stabilization window.

IV. Experiments

A. Experimental Setup

The experiments are conducted on a 3-node Kubernetes cluster with AMD EPYC 7H12 2.60GHz CPUs. On top of this Kubernetes cluster, we have configured an Apache Flink cluster in application mode. The JobManager instance is provided with 1 CPU and 8GB of memory, while each employed TaskManager consists of 1CPU and 4GB of memory. An NFS server is deployed as a persistence layer for the Apache Flink deployment, Prometheus(https://prometheus.io/) is used for scraping and gathering all the metrics, and an Apache Kafka(https://kafka.apache.org/) deployment is used as a source for the experiments. We cap the available resources to 80 task managers, resulting in a maximum of 80 CPUs and 320GB of memory available for processing.

B. Queries & Workload

For the evaluation of the autoscalers, we employ queries from the original NEXMark benchmark [7] and the extended version provided by the Apache Beam project [1]. We use a scalable generator that utilizes the NEXMark entity generators to create dynamic workloads following specified patterns.

Queries. More specifically, we first evaluate the autoscalers using *Q1* of the original NEXMark benchmark [7]. *Q1* is a simple map query that performs a currency conversion from U.S. dollars to Euros. We choose *Q1* as a representative stateless query with a simple topology and a low computational load. In addition, we employ *Q11* from the extended version of the benchmark [1]. *Q11* computes the number of bids a user made in each active session. It represents a windowed aggregate (count) over a session window and, therefore it comprises a stateful complex computation task.

Workload. We choose to develop our own data generator in order to mimic the periodic/seasonal workloads seen in real-life deployments. We opted for a dynamic workload that follows a cosine pattern. This cosinus workload has a mean value of 1.2M records per second, a max-divergence of 1M, and a period of 60 minutes. Some moderate noise of up to 100K records per second is also introduced to mimic real-world conditions. Every experiment has a total duration of 140 minutes. Due to the length and the high input rate, we set for each record a time-to-live of 10 minutes in our Kafka queue to avoid saturating the storage system.

Autoscalers' Configuration. For our experiments, we set Dhalion's scale down factor to 0.2, a value suggested in the original work. We use an overprovisioning factor of 0.2 for DS2, which we consider to be sufficient as the intention of DS2 is to avoid any overshooting of resources. We use the default stabilization window of 5 minutes for HPA-CPU, and we choose a target CPU utilization of 70% as the best performing among the values tested. In addition, we employ a cooldown window of 5 minutes after every scaling action to allow time for the system to reach a stable state and avoid back-to-back scaling actions due to a slow restart of the system or the lag produced by the scaling action.

C. Preliminary Experimental Results

Query 1 (Q1). After some dry runs, we provide a starting parallelism of 5 for each operator for *Q1* in order to ensure that the system starts with close to optimal resources and to avoid spending time adjusting to the starting load. In terms of *latency* (fig. 2), HPA-CPU greatly outperforms both Dhalion and DS2. It manages to retain a very low average latency throughout the experiment(table I). The few spikes of latency in fig. 2 are due to the rescaling actions that take place. DS2 and Dhalion manage to keep a low level of latency while the input rate was

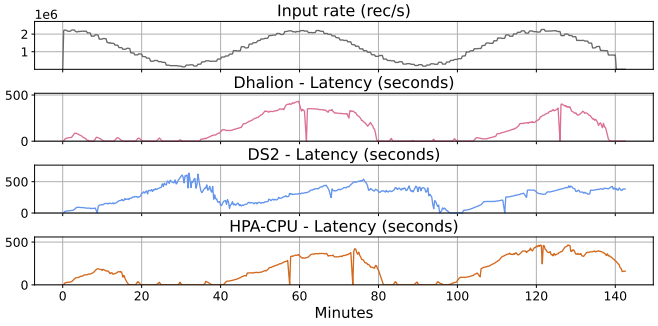

Fig. 4. Latency results of query 11.

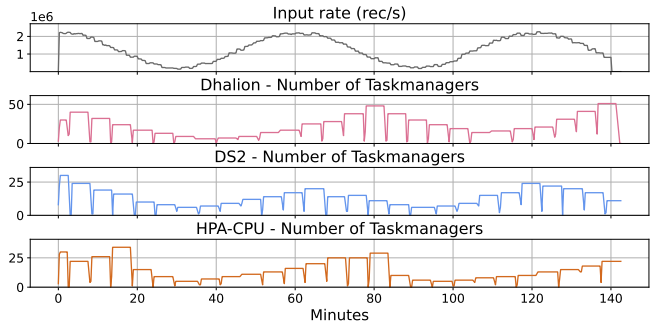

Fig. 5. Number of task managers deployed over time on query 11.

low but failed to adapt when the input rate was increasing. For Dhalion this is a result of scaling a single operator per scaling action, while for DS2 is a result of the inability to handle the lag accumulated from backpressure during the cooldown period and the rescaling action. In terms of *resource usage* (fig. 3), DS2 assigns on average fewer resources than HPA-CPU and Dhalion, and significantly fewer resources when the system operates under high input rates (table I). These differences in the assigned resources also justify why HPA-CPU achieves a better performance in terms of latency. In terms of *rescaling actions*, DS2 triggers the least rescaling actions since it is more stable and avoids triggering a rescaling action when is not necessary as we can see in Figure 3 for low input rates.

Query 11 (Q11). Similarly, we start the execution of *Q11* with a parallelism of 10 for each operator. *Q11* consists of heavier computational workload; fact that reflects on the results. In terms of *latency* (fig. 4), none of the autoscalers manages to scale the system well enough to handle high input rates. HPA-CPU and Dhalion manage to keep latency low for lower input rates, while DS2 fails. Again the main reason for this behavior of DS2 is that the lag that exists in the Kafka queue is not considered when calculating the required parallelism. In terms of *resource usage* (fig. 5), DS2 and HPA-CPU assign a significantly lower amount of resources, while Dhalion recommends on average 8-9 more task managers and reaches a maximum number of task managers that is higher than the rest by 20 units. For Dhalion, this is mostly attributed to the excessive scaling of the source operator due to the $scaleUpFactor_{KS}$. DS2 has a slightly better performance than HPA-CPU as it recommends on average one less task manager. In terms of *scaling actions* (table I), all autoscalers showed the same performance.

TABLE I
SUMMARY OF RESULTS FOR THE FULL DURATION OF THE EXPERIMENTS.

Query	Autoscaler	Rescaling Actions	TaskManagers		Latency(s)	
			Max	*Avg*	*Max*	*Avg*
Q1	*Dhalion*	27	30	13.2	254.4	68.3
	DS2	23	**15**	**9.4**	137.6	39.4
	HPA-CPU	26	21	11.4	**71.3**	**6.4**
Q11	*Dhalion*	27	51	22.8	**432.5**	**130.7**
	DS2	26	**30**	**13.2**	636.6	284.1
	HPA-CPU	26	34	14.2	466.3	178.0

Summary of Findings. Surprisingly, HPA-CPU has the better overall performance for both queries. Although it recommends slightly more task managers than DS2 across the experiments, it achieves low latency results for the simple map query (Q1) and similar to Dhalion latency for the window aggregate (Q11), while recommending on average 8 fewer task managers. It is evident that DS2 recommends resources that follow the same pattern as the input rate, however, it does not account for lag; hence, the bad performance. Dhalion has the worst performance among the autoscalers both in terms of resources and latency. Of course, these performance results depend heavily on parameter tuning, the deployment decisions of the underlying system, and our design decisions to allow all the autoscalers to work on an end-to-end basis under real-world assumptions. We try to tune fairly the autoscalers and the underlying system and we intend to better explore parameter tuning and deployment properties in future work.

V. CONCLUSION

In this paper, we point out the lack of significant comparison between existing solutions proposed for autoscaling in stream processing, and the need for an extensive experimental evaluation to identify the best-performing existing solutions and the biggest challenges remaining unsolved. We showcase the preliminary results of such an experimental evaluation of autoscalers using a dynamic workload and two NEXMark queries. Surprisingly, in contrast to existing literature, a simple CPU usage-based solution outperforms state-of-the-art solutions. Using these experiments, we argue that an efficient solution should take into account the impact in terms of generated lag of the rescaling actions and the existing lag when recommending optimal parallelism for the current load.

REFERENCES

[1] Extended nexmark benchmark from apache beam project. https://beam.apache.org/documentation/sdks/java/testing/nexmark/. Accessed: 2022-12-20.
[2] Kubernetes horizontal pod autoscaling. https://kubernetes.io/docs/tasks/run-application/horizontal-pod-autoscale/. Accessed: 2022-12-20.
[3] A. Floratou, A. Agrawal, B. Graham, S. Rao, and K. Ramasamy. Dhalion: Self-regulating stream processing in heron. *PVLDB*, 2017.
[4] T. Z. J. Fu, J. Ding, R. T. B. Ma, M. Winslett, Y. Yang, and Z. Zhang. Drs: Auto-scaling for real-time stream analytics. *IEEE/ACM Trans. Netw.*, 25(6):3338–3352, dec 2017.
[5] V. Kalavri, J. Liagouris, M. Hoffmann, D. Dimitrova, M. Forshaw, and T. Roscoe. Three steps is all you need: Fast, accurate, automatic scaling decisions for distributed streaming dataflows. In *Usenix OSDI*, 2018.

[6] T. Lorido-Botran, J. Miguel-Alonso, and J. A. Lozano. A review of auto-scaling techniques for elastic applications in cloud environments. *Journal of grid computing*, 12(4):559–592, 2014.
[7] P. Tucker, K. Tufte, V. Papadimos, and D. Maier. Nexmark–a benchmark for queries over data streams (draft). Technical report, 2008.
[8] B. Varga, M. Balassi, and A. Kiss. Towards autoscaling of apache flink jobs. *Acta Universitatis Sapientiae, Informatica*, 13:1–21, 04 2021.