Toward Stream Processing Elasticity in Realistic Geo-Distributed Environments

Khaled Arsalane, Guillaume Pierre, Shadi Ibrahim

▶ **To cite this version:**

Khaled Arsalane, Guillaume Pierre, Shadi Ibrahim. Toward Stream Processing Elasticity in Realistic Geo-Distributed Environments. IC2E 2024 - 12th IEEE International Conference on Cloud Engineering, IEEE, Sep 2024, Paphos, Cyprus. pp.1-9. hal-04655408v2

HAL Id: hal-04655408

https://inria.hal.science/hal-04655408v2

Submitted on 19 Aug 2024

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.

Distributed under a Creative Commons Attribution 4.0 International License

Toward Stream Processing Elasticity in Realistic Geo-Distributed Environments

Khaled Arsalane
Univ Rennes, Inria, CNRS, IRISA
khaled.arsalane@irisa.fr

Guillaume Pierre
Univ Rennes, Inria, CNRS, IRISA
guillaume.pierre@irisa.fr

Shadi Ibrahim
Inria, Univ Rennes, CNRS, IRISA
shadi.ibrahim@inria.fr

***Abstract*—Stream data processing is a widely used technology for analysing IoT-generated data shortly after being produced, and delivering timely insights about them. Executing such analysis in geo-distributed platforms enables shorter delays between data production and processing and fewer disturbances due to potential instability of long-distance networks, while retaining the ability to scale the processing capacity up and down according to the demand. However, current stream processing systems were designed for environments made of homogeneous servers connected together using high-speed network links. We experimentally study the performance of Apache Flink coupled with the Gesscale auto-scaler in conditions which resemble those of geo-distributed platforms. We demonstrate that Flink's backpressure mechanism should not be used as the only trigger for rescaling operations in heterogeneous network conditions. Raw performance, as well as performance predictability, also degrade quickly in the presence of stateful data processing operators and/or high network latency between the processing nodes.**

***Index Terms*—Data Stream Processing, geo-distributed environment, elasticity, stateful operators**

I. Introduction

In the era of rapidly growing real-time data, the emergence of Data Stream Processing (DSP) has revolutionized the management, analysis, and timely utilization of large volumes of continuous data [1]. DSP systems excel at handling high-velocity data streams and enabling applications to promptly extract actionable insights. However, their deployment at a large scale poses significant challenges, especially as the number of IoT devices producing input data increases. The projected amount of data generated by IoT devices is expected to reach 163 zettabytes (10^{21} bytes) by 2025 [2].

As the number of data sources increases, so does the usage of long-distance networks to transport the raw data to cloud data centers where they may be processed [3]. Geo-distributed environments emerge as a promising paradigm to address the limitations of such datacenter-centric approaches [4], [5]. Such environments extend centralized cloud platforms with additional resources close to the data sources to reduce data transfer latency and to enable faster decision-making processes. Incorporating DSP into geo-distributed environments offers significant advantages, allowing the DSP system to leverage the proximity to data sources for timely analytics [3].

An important challenge when processing data produced by IoT devices is that these data streams are often non-stationary [6]. It is therefore necessary to dynamically scale the processing capacity up and down according to the time-varying demand. Several strategies for managing the elasticity of DSP systems have been proposed in the literature [7]. They differ based on the deployment environment of the DSP, the monitored data type, the targeted quality-of-service objective, and the utilized optimization method.

DSP systems such as Apache Spark and Apache Flink were initially designed to operate in cluster-like environments based on a homogeneous set of powerful servers connected with high-capacity network links. However, these hypotheses are not necessarily met when the DSP system is deployed using multiple geo-distributed servers located close to the sources of input data. This paper aims to identify the strengths and weaknesses of current DSP systems and their auto-scalers when they execute stateless as well as stateful operators in realistic geo-distributed settings.

We focus our study on the Apache Flink DSP system [8] coupled with the Gesscale auto-scaler [9]. We evaluate the system throughput of stateless and stateful workflows when varying the number of servers, and when the network latency between servers is large and fluctuating. We show that:

- Stateful applications suffer from unstable throughput when scaled on nodes with heterogeneous network conditions;
- Backpressure is not a reliable signal for scaling decisions in heterogeneous network conditions;
- Skewed data distribution causes performance to be unpredictable during rescale operations.

Based on these observations we propose a research agenda to improve the way DSP systems and their auto-scalers work in heterogeneous geo-distributed environments.

The remainder of the paper is organized as follows: Section II presents the technical background and Section III details the experimental system. Section IV presents our evaluations. Section V discusses the strengths and weaknesses of current DSP systems. Finally, Section VI presents the related work and Section VII concludes.

II. Background

A. DSP elasticity

Horizontal elasticity is an essential feature of DSP systems. However, there is no one-size-fits-all solution for achieving elasticity [7]. Different designs depend on the considered metrics, desired QoS goals, and the underlying infrastructures. The most common Quality of Service (QoS) objectives include

minimizing end-to-end application processing latency [10], maximizing application throughput [9], [11], [12], and reducing costs [13], [14]. In this work, we focus on Maximum Sustainable Throughput (MST) as QoS objective [11]. MST is the highest level of output that a system can sustain with acceptable quality for a desired period.

Geo-distributed platforms usually consist of different types of compute and network resources than those found in traditional cloud data centers. This reduces deployment and operation costs but increases resource heterogeneity in the infrastructure. Therefore, it is important to carefully consider these types of resources, particularly when developing models that aim to implement elasticity. Network resources can have a significant impact on the performance of the DSP application if the elasticity model does not properly account for them.

Stream processing operators can be classified as either stateless or stateful. Stateless operators perform simple operations on every data record, such as mapping and filtering. In contrast, stateful operators, such as window joins and data aggregations, require the retention of past operation states and results in memory. During scaling operations in an elastic resource adjustment context, it is important to maintain the state of stateful operators to ensure accurate results of the stream processing application. However, transferring the state to new replicas can prolong the application's reconfiguration time, leading to a temporary halt in record processing and a significant decrease in overall performance.

To ensure consistent throughput, DSP systems use network buffers to transmit data between operator instances. When an operator experiences a bottleneck, its input buffers may fill, risking a temporary data loss. DSP systems use the so-called "backpressure" mechanism to request a task's upstream operators to slow down their rate of records emission, until there is space again in buffers [15], [16]. Many works on autoscaling also use backpressure as a signal that the system does not have enough compute resources to process incoming data, and to trigger elastic resource rescaling [12], [17], [18].

B. The Gesscale auto-scaler

Multiple strategies can be used to achieve QoS objectives, from simple threshold-based approaches to specialized models and machine learning techniques. This work focuses on Gesscale, an auto-scaler for geo-distributed DSP systems [9]. Gesscale relies on a performance model to predict the system performance after it has been reconfigured [19]. Like many DSP auto-scalers, it aims to guarantee a sufficient Maximum Sustainable Throughput (MST) to process the incoming workload. Gesscale's performance model also takes into account the impact of inter-node latency. It relies on three parameters that are calibrated during the first few re-configurations. Its prediction accuracy therefore increases over time.

However, Gesscale has only been tested with known and static network latencies, which may limit its effectiveness in dynamic heterogeneous network environments. Furthermore, its performance model has not been tested for stateful DSP operators. This study aims to analyze the model's behavior with this category of operators, identify any gaps, and propose potential approaches to enhance its functionality.

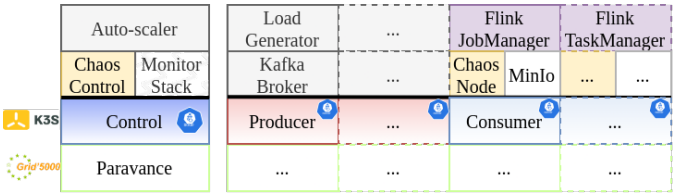


Fig. 1. Experimental platform.

III. Evaluation methodology

A. Experimental setup

1) Platform

We conducted all experiments in the Paravance cluster of Grid'5000 experimental testbed [20]. Each machine is equipped with an Intel Xeon E5-2630 v3 processor (x86_64, 2.40 GHz, 2 CPUs/node, 8 cores/CPU), 128 GB of RAM, and a 10 Gbps network card. We chose to use a single cluster as it allows us to precisely control the inter-node network latencies.

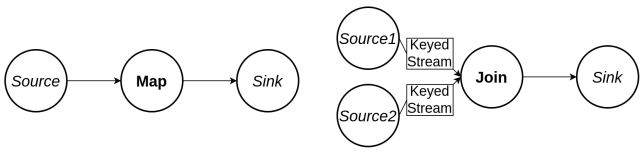
As shown in Figure 1, we deploy Kubernetes (K3S v1.28.3+k3s2) [21] on the 15 nodes of the cluster. Kubernetes manages the life cycle and the orchestration of the DSP system. We limit resources to 1 CPU core and 4 GB of memory to all applications deployed on top of the k3s nodes. The Kubernetes cluster has 5 nodes labeled as producers and 10 nodes labeled as consumers. Producers host the load generators and the Kafka broker replicas, whereas consumers host the DSP system instances and its distributed storage system.

To emulate network degradation between nodes, we add chaos-mesh (v2.6.2) [22] in the consumer nodes alongside the DSP system. It helps to create a controlled environment with different network profiles, such as bandwidth limitation and artificial network delays.

2) Stream processing system

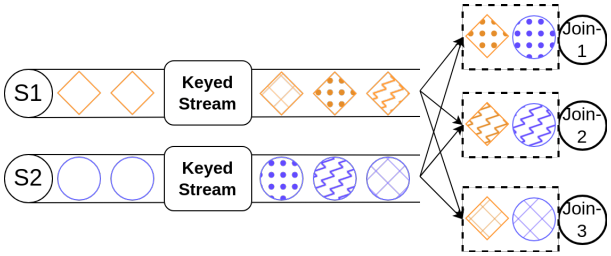
We use the Apache Flink (v1.17.2) DSP system [8]. Flink is a powerful open-source stream processing framework which provides low-latency and high-throughput processing of data streams with exactly-once semantics. It is designed to execute as a distributed system, making it ideal for running in geo-distributed environments.

Flink's run-time consists of two types of processes: one JobManager, and one or more TaskManagers. The TaskManagers are responsible for executing the tasks assigned to them by the JobManager. The JobManager is responsible for scheduling tasks, coordinating checkpoints, and managing the overall execution of the job. A job in Flink is a program that describes a pipeline, composed of operators, in which data flows and where each operator applies transformations to data flows. In Kubernetes, JobManager and TaskManager run in separate containers, making scaling of TaskManagers a trivial task. A single task slot per TaskManager is set to force the allocation of a single operator's replica per TaskManager. This allows us to apply targeted network degradation to a subset

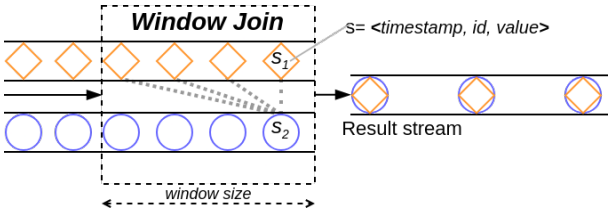


(a) Stateless job with Map. (b) Stateful job with Window Join.

Fig. 2. Tested jobs.



(a) Keyed streams mechanism.



(b) Window mechanism.

Fig. 3. Window join mechanisms.

of TaskManagers while observing how heterogeneous network conditions affect part of the stream processing pipeline.

To handle failure recovery, Flink leverages its internal checkpointing mechanism [23]. It allows Flink to recover state and positions in the stream by storing consistent snapshots of all the state in timers and stateful operators, including connectors, windows, and any user-defined state. Similar to checkpoints, a savepoint is a consistent image of the execution state of a streaming job. Savepoints allow jobs to be gracefully stopped and reconfigured (e.g., re-configuring the parallelism level of a job with a stateful operator) while minimizing data loss. We setup Flink to use MinIo [24] as its distributed storage system for checkpoints and savepoints.

3) Workload

We implement two pipelines to evaluate the performance of two types of operator: a stateless one and a stateful one. To emulate compute work, each operator applies a custom function consisting of a Fibonacci function evaluation. Figure 2(a) presents the pipeline with a *Map* acting as a stateless operator. It processes incoming data injected at the *Source*, performs computations by applying a custom *Map* function, and outputs the results to a *Sink* operator.

Figure 2(b) shows the pipeline with a *Window Join* acting as a stateful operator. Input data are produced in two *Source* operators at equal rate and with the same range of keys. We then transform each data stream to KeyedStream by using the record IDs. KeyedStreams [25] allow Flink to partition streams between tasks, thus helping to route records of a key subspace to independent task instances without incurring a synchronization overhead, as shown in Figure 3(a). Figure 3(b) illustrates the *Window Join* operation. Records are collected in a time-sized tumbling event window of 4 seconds. Each record of the first stream is compared to all other records of the other stream within this window. If the join condition used for the comparison holds true, a custom *Join* function is applied between the records. Given records s_1 from stream S_1 and s_2 from S_2, for the implementation of the join condition, we focus on two possible join criteria:

1) s_1.id $= s_2$.id (ID-ID join)
2) s_1.id $= s_2$.value (ID-value join)

In the first scenario, the join condition is applied on the same IDs used for the two keyed streams. This creates a compute-intensive pipeline with a high occurrence of joins, due to both streams being symmetrical. The second scenario involves a join condition applied between the ID of the first stream's record and the value of the second stream's record. We generate values randomly in a key subspace smaller than the IDs to force the join condition on keys different than those used for the keyed stream. We can thus observe skew in key distribution and the impact on performance caused by non-deterministic triggers of the join function.

4) Auto-scaler

Gesscale [9] is a DSP auto-scaler that adapts resources by using a performance model [19]. We deploy it in the master node of the Kubernetes cluster. It estimates the inter-node network delays of consumer nodes using the Vivaldi network coordinates algorithm [26] implemented by the Serf module [27] of Consul (v1.17) [28].

Based on the backpressure metrics on *Source* nodes and current data throughput, the auto-scaler predicts the system's performance and decides to scale the number of TaskManagers on the best available nodes. With the adequate number of TaskManagers, it adjusts the parallelism of every operator of the Flink application. Gesscale always tries to select nodes that have the lowest inter-node delay. Particularly, from all the available nodes, it looks at the maximum inter-node delay that a node would add to the system and it tries to select that with the minimal one, as it will be the weakest link in the graph, in terms of processing latency.

B. Experimental Procedure

1) Workload injection

The load generators inject data streams into the Kafka broker, which provides data for consumption to the stream processing application. Each stream consists of a sequence of $< id, timestamp, value >$ data records produced by multiple identical emulated sensors, each of which is identified by a unique identifier between 0 and $x - 1$. Each emulated sensor generates one message per second. In the *Map* use case, a single stream emulating $x = 100,000$ sensors injects data into the pipeline. In contrast, in the *Window Join* use cases, two

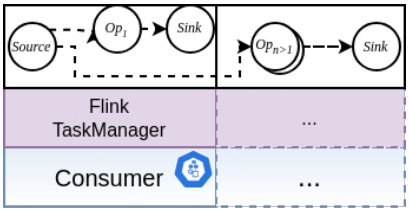
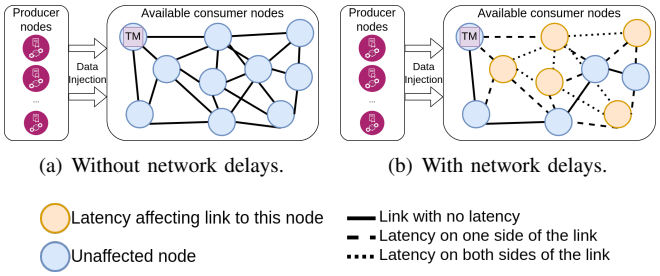


Fig. 4. Pipeline deployment.



(a) Without network delays. (b) With network delays.

Fig. 5. Deployed consumer nodes.

streams emulating $x = 50,000$ sensors each inject data into each *Source* of the pipeline. This results in a constant number of comparisons being made within the *Window Join* between triplets of the two streams. Injecting a static workload which significantly exceeds the pipeline's processing capacity allows us to observe the evolution of throughput under conditions of both stable and degraded nodes following a scale-up operation.

2) Operator scaling

Figure 4 presents the typical deployment of a stream processing application. The experiment starts with the deployment of the job with a parallelism value of 1. This results in the entire pipeline being deployed on a single TaskManager. As the application scales out, Flink's scheduler co-locates the first instance of the operator with the *Source* operator, and all other instances on subsequent available TaskManagers. The auto-scaler gives the application a warm-up interval of 180 seconds, then it monitors the job metrics for an interval of 60 seconds at the end of which it computes a prediction matrix. This matrix is computed using metrics representing: the ingestion throughput of the operator (numRecordsInPerSecond); the backpressure generated on upstream operators (backPressured-TimeMsPerSecond); and the estimated inter-node network delays. Based on the prediction matrix, the auto-scaler adds TaskManager instances, if necessary, on available Consumer nodes and rescales the pipeline with a new parallelism level for the bottleneck operator. The reconfiguration of an operator is triggered according to a backpressure threshold. If, during a one-second interval, an instance is backpressured for at least 500 milliseconds, then a scale-out decision is executed by the auto-scaler, according to the prediction matrix. The first run of the model produces a matrix with modest accuracy, as the system has not gathered enough information on the job. After 3 rescaling actions, the prediction matrix starts to converge to a more precise forecast. The experiment concludes when the auto-scaler is no longer able to adjust parallelism due to shortage of resources or when the backpressure drops below the defined threshold of 500 ms, as this indicates that upstream operators are no more throttled down.

3) Wide-area network conditions emulation

Figure 5(a) shows a deployment with no artificial network delays. Since the testbed server nodes are co-located in a single rack, they result in an average inter-node latency lower than 1 ms. We run each job in this environment to observe the minimum number of instances required to process the imposed workload. Figure 5(b) depicts a network configuration in which 50% of consumer nodes are subject to network delays. This implies that the inbound connection of each affected node experiences an artificial delay. For example, two nodes affected with 25 ms injected delay will experience an overall 50 ms round-trip latency on the link connecting them.

In the initial phase of the experiment, the auto-scaler selects nodes that are not impacted by network delays. This allows us to observe how scaling affects throughput when network conditions are not a factor. Once all unaffected nodes have been consumed, the application starts using nodes affected by network delays, which impacts the performance of the job and the prediction model. We explore two scenarios with delay injection: (i) fixed 25 ms network delays; and (ii) dynamic network delays with values fluctuating between 15 ms and 35 ms. The first scenario illustrates the impact on the job's performance when it is rescaled on nodes with heterogeneous network conditions. The second scenario highlights the impact of a dynamic heterogeneity in network conditions.

IV. Evaluation

We now evaluate stateless and stateful operators using three jobs: a stateless job implementing the *Map* operator, and two stateful jobs implementing the *Window Join* operator, one with join condition demanding equality between $id_1 - id_2$ and the other between $id_1 - value_2$ of triplets s_1 and s_2 from streams S_1 and S_2. Each job runs with either no artificial latency, fixed 25 ms latency, or variable 25±10 ms latency with jitter.

We conduct five runs of each experiment to ensure the reliability of the results. The results display the minimum, maximum, and average throughput ingested by the operator per level of parallelism. We also include the model's predictions for each level of parallelism, along with the error percentage compared to the actual measured throughput.

A. Effects of skewed data distribution on system busyness

Figure 6 presents the average busyness of each operator instance under our standardized workloads. The experiment, conducted without artificial latency, illustrates the minimum and maximum values of busyness experienced by any instance of the operator. It can be observed that all operators have at least one instance that is always 100% busy. In the *Map* and *Id-Id Join* use cases, this always occurs for the first instance of the operator that is co-located by Flink in the same TaskManager as the *Source* connectors, as illustrated in Figure 4. This

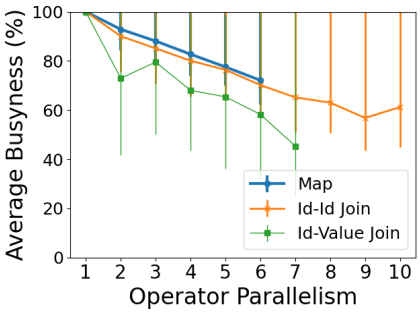


Fig. 6. Average busyness of individual TaskManagers when scaling streaming jobs with no artificial latency.

TaskManager therefore handles an extra task to ingest and distribute input data to downstream operators, which explains the additional busyness of this TaskManager compared to the others. However, the distribution of keys among all instances of the operator remains balanced. This is evidenced by the linear decrease in busyness for increasing levels of parallelism.

In contrast, the *Id-Value Join* operator experiences large variations in busyness, which can be attributed to the skewed distribution of matching id-value pairs. Unlike previous operators, the instance that is always busy changes at every re-scale operation, while at the same time, other instances experience low levels of busyness. This observation clearly highlights the effects of data skewness on stateful applications.

B. Flink's scaling behaviour in different latency setups

1) With no artificial latency

Figure 7 shows the throughput of the stateless and stateful operators in homogeneous environments with no artificial latency. We see that *Map* and *Id-Id Window Join* scale almost linearly with the number of instances. For the *Map* operator, we observe that each additional instance increases throughput by 90% of that of the first instance. In addition, the application manages to sustain the full workload (indicated in the figure by the red line) when it reaches 7 instances.

Stateful operators however behave differently as state management operations such as checkpointing cause an overhead that reduces throughput. Figure 7(b) shows the behavior of the *Id-Id Window Join* operator. The throughput again grows monotonously with the number of operator instances, but the throughput increase per added instance is only 70% of the throughput of a single operator. As a result, the *Id-Id Window Join* operator requires more instances to achieve the target throughput. Note that the experiment ends at 10 instances – the maximum number of consumer nodes in our settings.

Finally, the *Id-Value Window Join* operator does reach its target throughput using only 7 instances, but the increase in throughout is not consistent as some of the new instances bring more extra throughput than others. This is due to the nature of the join condition, which causes skewness in the distribution of data. Data skewness causes an imbalance in the load across instances, which affects the overall throughput (as discussed in Section IV-A). We observe that this join condition does not generate a deterministic amount of triplets for each comparison. Consequently, the join condition triggers the custom function less often, resulting in less computational work compared to the *Id-Id Window Join* use case. This allows the operator to ingest more incoming data without producing enough records.

We also see that Gesscale predicts the throughput of the *Map* and *Id-Id Window Join* operators with high accuracy. However, it fails to predict the throughput of *Id-Value Window Join* operator. We observe significant errors such as -52% because the model does not consider data partitioning and the impact of data skew. These results complement previous work [9], which only shows an expected near-linear scalability for *Map* and *Id-Id Window Join* under homogeneous network conditions. This suggests that performance models should take data skewness into account in their predictions.

2) With fixed latency

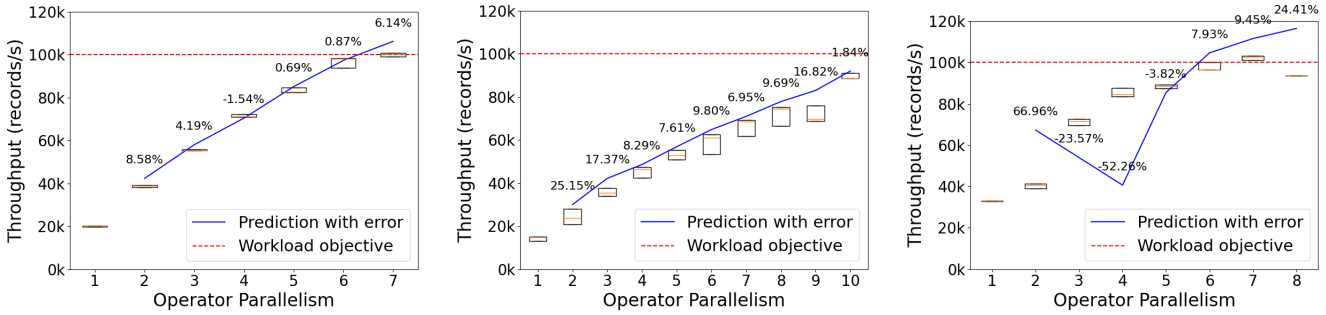
Figure 8 shows the data processing throughput when scaling DSP operators in heterogeneous environments where 50% of the consumer nodes are affected by artificial network latency (i.e., by adding 25 ms ingress network latency). Note that the auto-scaler selects new nodes based on their latency to previous nodes, so the impact of artificial network latency is only observed when scaling to 6 instances and above.

We make two observations. First, although the *Map* operator scales normally despite the network heterogeneity, the overall performance of the *Id-Id Window Join* operator degrades in heterogeneous environments when scaling beyond 5 instances. Most runs (4 out of 5) of the *Id-Id Window Join* operator show a peak throughput at parallelism level 7, with an average ingestion of 62k records/s. The throughput eventually drops to an average of 58k records/s at parallelism level 8. At this point, experiments stop scaling beyond 8 instances, even though the target throughput is not reached, because the backpressure level drops below the defined threshold of 500 ms. Only one run of the experiment achieves parallelism 10, with a throughput of 88k records/s. The *Id-Value Window Join* operator exhibits a comparable behavior to that observed in homogeneous environments. This operator, however, demonstrates its ability to ingest the full workload at parallelism 7. This is due to the join condition not being triggered often enough to saturate the operator with the ingested data.

Second, Gesscale's predictions for the *Map* operator are reasonably accurate. In contrast, *Id-Id Window Join* shows poor predictions in the presence of latency. As heterogeneity appears from parallelism 6 and beyond, we observe prediction errors reaching 34% for the *Id-Id Window Join* operator. Finally, *Id-Value Window Join* exhibits very poor predictions, with errors up to 62%. Similar to the situation with no artificial latency, prediction errors of +62% and -36% render the auto-scaler highly unreliable although the DSP system reaches its target throughput between parallelism levels 6 and 7.

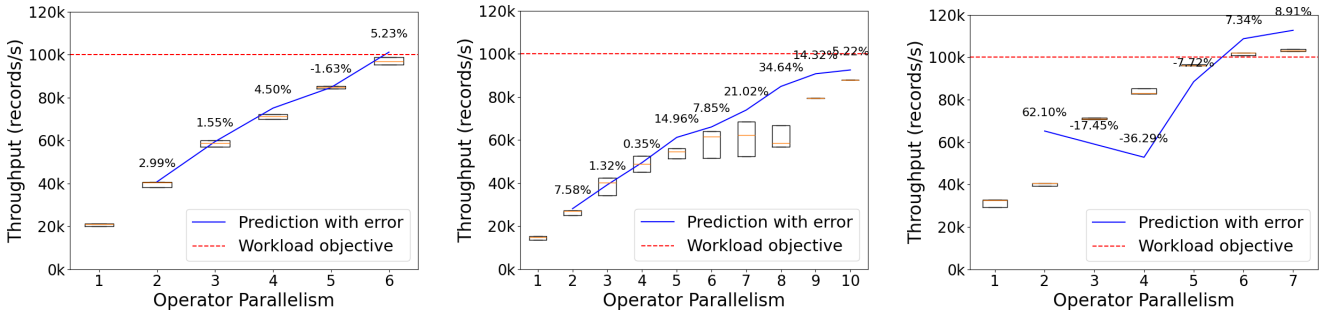
3) With fluctuating latency

Figure 9 shows the results when scaling DSP operators in dynamic environments, where 50% of consumer nodes are



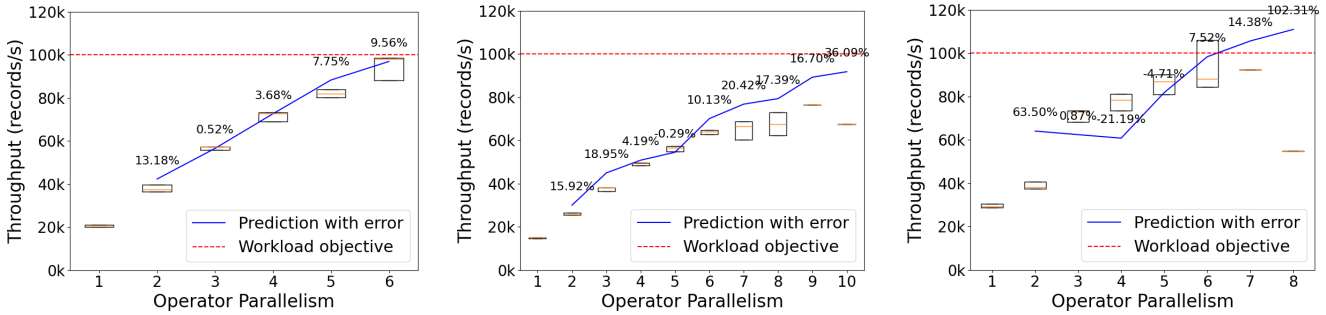
(a) Map. (b) Id-Id Window Join. (c) Id-Value Window Join.

Fig. 7. Throughput measurements – No latency scenario.



(a) Map. (b) Id-Id Window Join. (c) Id-Value Window Join.

Fig. 8. Throughput measurements – Fixed 25 ms latency scenario.



(a) Map. (b) Id-Id Window Join. (c) Id-Value Window Join.

Fig. 9. Throughput measurements – Variable 25±10 ms latency with jitter scenario.

affected by latency jitter (i.e., with continuously changing network ingress delays between 15 ms and 35 ms). Once again, we observe that the Gesscale model correctly accounts for network delays when scaling stateless operators. The full workload is ingested at parallelism level 6, with low prediction errors. However, the behavior differs for stateful operators. The performance of the *Id-Id Window Join* and *Id-Value Window Join* operators begins to decrease when using more than 6 instances. For instance, the throughput of the *Id-Id Window Join* operator drops by ∼13%, from 77k records/s to 67k records/s, when the number of instances is scaled from 9 to 10. The model predictions diverge from the measured throughput with errors up to 34%. This can be explained by the heterogeneous network conditions affecting half of the nodes. This degradation is more pronounced for the *Id-Value Window Join* operator. This use-case shows instability in throughput and prediction errors similar to the fixed latency experiment, which can be attributed to the nature of the application.

We also observe that scaling DSP operators usually helps to increase the data processing throughput. However, the expected increase in throughput is different depending on whether the operator to be scaled is stateless or stateful. In fact, DSP applications composed of stateful operators have an increased overhead due to state management and other factors. In some cases, the nature of the stateful operation itself causes unexpected variations in throughput. Furthermore, heterogeneous network conditions can significantly impact the throughput of stateful operators, potentially leading to perfor-

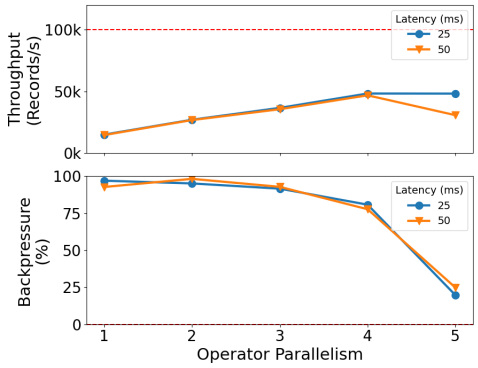


Fig. 10. Throughput and backpressure measurements – Id-Id Window Join with fixed network delays affecting 70% of nodes.

mance degradation. The current prediction model accurately accounts for the impact of heterogeneous network conditions on stateless operators, enabling precise forecasts. However, the model struggles to accurately predict the throughput of stateful operators, resulting in significant prediction errors.

C. Auto-scaling in heterogeneous network conditions

We saw that the auto-scaler sometimes cannot rescale even though it does not reach the target throughput because the backpressure signal drops below the defined threshold. This results in the application not scaling as intended.

Figure 10 presents the effect of network delays on backpressure for the *Id-Id Window Join* operator. Here, 70% of consumer nodes are subject to fixed artificial network delays. We explore two cases: the first one injects 25 ms inter-node delay, whereas the second injects 50 ms. Increasing the number of instances of an operator can alleviate backpressure on upstream operators, provided that network conditions are not the limiting factor. The case with 25 ms injected delay shows that, with 3 parallel instances of the operator not affected by heterogeneity, the throughput scales linearly with small decrease in backpressure. However, when a pair of nodes with degraded network conditions is added, the throughput levels off. Simultaneously, backpressure experiences a sudden drop below the threshold level, which affects the auto-scaler's decision-making. This trend is even more pronounced in the case with 50 ms injected delay, where the backpressure drops significantly while the throughput also deteriorates, ultimately affecting the application performance. In this case, the throughput bottleneck is due to network delays rather than limited computation capabilities. Backpressure is produced only locally within each TaskManager with limited computation capability, which explains the reason why the auto-scaler does not receive the correct trigger.

Although we conducted these experiments using Gesscale, we expect other auto-scalers which rely on backpressure to suffer from the same problem [12], [17], [18], [29].

V. Discussion

Our evaluations highlight strengths and weaknesses when using Apache Flink and Gesscale in realistic geo-distributed environments. We derive research directions for future work.

Stateless vs. stateful operators: Flink and Gesscale handle stateless operators with predictable and scalable performance, even in heteogeneous networking conditions. However, managing stateful operators is more challenging, and applications with a skewed data distribution exhibit significant performance instability during rescaling. Fluctuating network performance also degrades the performance of stateful operators as well as the quality of rescaling decisions.

Heterogeneous infrastructure: DSP systems were designed for homogeneous environments such as data centers. When the data sources are geo-distributed such as in IoT use-cases, it becomes useful to deploy the DSP in an Edge/Fog infrastructure. In our experiments, we could see that even in scenarios with just a couple of nodes suffering from degraded network performance, slow TaskManagers become the overall performance bottleneck, and force the unaffected nodes to slow down as well. This is a concern both in terms of absolute performance of the DSP system and for its auto-scaler.

Backpressure as rescaling signal: Backpressure is crucial for DSP systems to ensure exactly-once semantics without data loss. Many works also utilize it to identify potential bottlenecks and to trigger rescaling operations. However, in geo-distributed environments, this signal may be distorted, causing incorrect resource adaptation decisions. In such environments, additional signals should be considered for the adaptation of the stream processing pipeline.

VI. Related Work

Numerous approaches have been proposed for run-time adaptation of stream processing applications [7]. We focused our study on horizontal scaling techniques in geo-distributed environments, specifically targeting the networking infrastructure conditions and their impact on data processing performance and predictability. We did not consider the impact of heterogeneity of the computation resources.

Most DSP auto-scaling works use backpressure as the main signal for triggering rescaling actions. AdCom pre-aggregates tuples in mini-batch intervals to achieve sustainable throughput [12]. A PID controller monitors the workload and adjusts the interval based on the level of network buffer fill to reduce backpressure. SpinStreams is a static optimization tool for DSP systems that analyzes and suggests a new pipeline configuration based on potential bottlenecks that can cause backpressure [17]. TransScale combines scaling and approximate data processing to respond to workload fluctuations at run-time [18]. Finally, GOVERNOR exploits a backpressure controller which takes into consideration the checkpointing costs to adapt the input sizes [29]. Our study adds to these works by highlighting the unreliability of backpressure in heterogeneous network conditions. Attention should be paid to DSP adaptation triggers in geo-distributed environments.

Several works focus on performance control for stateful DSP operators. Elasticutor uses both vertical and horizontal scaling, where CPU cores are dynamically added to a subset of keys before using distant nodes [30]. This reduces migration

times and therefore application reconfiguration times. This solution has been extensively studied under different types of workloads (velocity, variety, and volume). However, the heterogeneous network conditions have not been considered in the case of migration. In geo-distributed environments, migration of operator states to different nodes may be costly.

A performance model that estimates the throughput and latency of streaming window-joins is presented in [31]. The work explores different join configurations and evaluates the impact on DSP application performance. However, it does not account for the impact of heterogeneous network latency. In contrast, we implement two join configurations that represent the scenarios of deterministic (*Id-Id Window Join*) and non-deterministic (*Id-Value Window Join*) joins fed by multiple streams.

Taking networking latency into account is important in geo-distributed DSP systems. Klink presents an optimization for Flink's scheduler which monitors window queries at run-time and analyzes their watermark progression [32]. It reduces end-to-end processing latency by dynamically releasing windows and firing tuples to meet the QoS objective. The authors also investigate the impact of latency on DSP systems. They estimate network delays based on event watermarks, whereas we rather employ network coordinates. Our work builds on theirs by demonstrating how network heterogeneity affects elasticity, particularly in terms of predictability and raw performance of window joins and other stateful queries.

Finally, Theodolite [33] is a benchmark designed to evaluate the scalability of DSP systems. It supports multiple DSP systems (including Apache Flink) and multiple stateless and stateful applications. In our work, we implemented test jobs inspired by these applications and used Theodolite's standalone load generators to inject workload.

VII. Conclusion

In this paper, we experimentally studied the performance and predictability of Apache Flink and Gesscale when handling stateless and stateful data processing tasks in realistic geo-distributed environments. We identified domains where these systems perform sufficiently well, in particular concerning stateless operators in heterogeneous networking environments. Stateful operators constitute a greater challenge in terms of absolute performance (especially for tasks which exhibit data skew), predictability, and rescaling trigger. These topics have attracted limited attention so far in the state of the art, and constitute promising avenues for future research.

Acknowledgments

Experiments presented in this paper were carried out using the Grid'5000 testbed, supported by a scientific interest group hosted by Inria and including CNRS, RENATER and several Universities as well as other organizations (see https://www.grid5000.fr).

References

[1] V. Kalavri, "Why stream processing systems are now more relevant than ever," SIGOPS blog, Jun. 2020, https://www.sigops.org/2020/streams/.

[2] RedHat Research, "Real-time data stream processing," Nov. 2020, https://research.redhat.com/blog/publication/real-time-data-stream-processing/.

[3] R. Muñoz *et al.*, "Integration of IoT, transport SDN, and edge/cloud computing for dynamic distribution of IoT analytics and efficient use of network resources," *IEEE Journal of Lightwave Technology*, vol. 36, no. 7, Apr. 2018.

[4] A. Ahmed *et al.*, "Fog computing applications: Taxonomy and requirements," *CoRR*, vol. abs/1907.11621, 2019, http://arxiv.org/abs/1907.11621.

[5] A. Yousefpour *et al.*, "All one needs to know about fog computing and related edge computing paradigms: A complete survey," *Journal of Systems Architecture*, vol. 98, 2019.

[6] U. Tadakamalla and D. A. Menascé, "Characterization of IoT workloads," in *Proc. IEEE EDGE*, 2019.

[7] V. Cardellini *et al.*, "Runtime adaptation of data stream processing systems: The state of the art," *ACM Computing Surveys*, vol. 54, no. 115, Sep. 2022.

[8] Apache Software Foundation, "Apache Flink ® – stateful computations over data streams," https://flink.apache.org/.

[9] H. Arkian *et al.*, "Model-based stream processing auto-scaling in geo-distributed environments," in *Proc. ICCCN*, 2021.

[10] K. Wang *et al.*, "Spur: Mitigating slow instances in large-scale streaming pipelines," in *Proc. ACM SIGMOD*, 2020.

[11] T. Lambert *et al.*, "Rethinking operators placement of stream data application in the edge," in *Proc. ACM CIKM*, 2020.

[12] F. Gutierrez *et al.*, "AdCom: Adaptive combiner for streaming aggregations," in *Proc. EDBT*, 2021.

[13] C. Hochreiner *et al.*, "Elastic stream processing for the Internet of Things," in *Proc. IEEE CLOUD*, 2016.

[14] V. Cardellini *et al.*, "Optimal operator deployment and replication for elastic distributed data stream processing," *Concurrency and Computation: Practice and Experience*, vol. 30, no. 9, 2018.

[15] U. Celebi, "How Apache Flink™ handles backpressure," Ververica blog, https://www.ververica.com/blog/how-flink-handles-backpressure.

[16] S. Kulkarni *et al.*, "Twitter Heron: Stream processing at scale," in *Proc. ACM SIGMOD*, 2015.

[17] G. Mencagli *et al.*, "Spinstreams: a static optimization tool for data stream processing applications," in *Proc. ACM Middleware*, 2018.

[18] A. Pagliari and G. Pierre, "TransScale: Combined-approach elasticity for stream processing in fog environments," in *Proc. MobileCloud*, 2023.

[19] H. Arkian *et al.*, "An experiment-driven performance model of stream processing operators in fog computing environments," in *Proc. ACM SAC*, 2020.

[20] D. Balouek *et al.*, "Adding virtualization capabilities to the Grid'5000 testbed," in *Cloud Computing and Services Science*, ser. Communications in Computer and Information Science. Springer, 2013, vol. 367.

[21] K3s authors, "K3s," https://k3s.io/.

[22] The Linux Foundation, "Chaos Mesh: a powerful chaos engineering platform for Kubernetes," https://chaos-mesh.org/.

[23] P. Carbone *et al.*, "State management in Apache Flink®: Consistent stateful distributed stream processing," *Proc. VLDB Endow.*, vol. 10, no. 12, aug 2017.

[24] MinIO Authors, "Minio," https://min.io/.

[25] Apache Software Foundation, "Stateful stream processing," https://nightlies.apache.org/flink/flink-docs-release-1.17/docs/concepts/stateful-stream-processing/.

[26] J. Ledlie *et al.*, "Network coordinates in the wild," in *Proc. NSDI*, 2007.

[27] Serf Authors, "Serf," https://www.serf.io/docs/internals/coordinates.html.

[28] Consul Authors, "Consul," https://www.consul.io/.

[29] X. Chen *et al.*, "GOVERNOR: Smoother stream processing through smarter backpressure," in *Proc. ICAC*, 2017.

[30] L. Wang *et al.*, "Elasticutor: Rapid elasticity for realtime stateful stream processing," in *Proc. ACM SIGMOD*, 2019.

[31] V. Gulisano *et al.*, "Performance modeling of stream joins," in *Proc. DEBS*, 2017.

[32] O. Farhat *et al.*, "Klink: Progress-aware scheduling for streaming data systems," in *Proc. ACM SIGMOD*, 2021.

[33] S. Henning and W. Hasselbring, "Theodolite: Scalability benchmarking of distributed stream processing engines in microservice architectures," *Big Data Research*, vol. 25, 2021.