Article

Performance Evaluation Analysis of Spark Streaming Backpressure for Data-Intensive Pipelines

Kassiano J. Matteussi[1]*, Julio C.S. dos Anjos [2]*, Valderi R.Q. Leithardt [3,4]* and Claudio F.R. Geyer[1]

[1] Federal University of Rio Grande do Sul, Institute of Informatics, UFRGS/PPGC, Porto Alegre, RS, Brazil, 91501-970; {kjmatteussi,geyer}@inf.ufrgs.br
[2] Federal University of Ceará, Graduate Program in Teleinformatics Engineering (PPGETI/UFC), Center of Technology, Campus of Pici, Fortaleza, Ceará, Brazil, 60455-970; jcsanjos@ufc.br
[3] COPELABS, Universidade Lusófona de Humanidades e Tecnologias, 1749-024 Lisboa, Portugal
[4] VALORIZA, Research Center for Endogenous Resource Valorization, Polytechnic Institute of Portalegre, Portalegre, Portugal, 7300-555; valderi@ipportalegre.pt
* Correspondence: kjmatteussi@inf.ufrgs.br; jcsanjos@ufc.br; valderi@ipportalegre.pt;

Abstract: In the past decades, a significant rise in the adoption of streaming applications has changed the decision-making process for the industry and academia sectors. This movement led to the emergence of a plurality of Big Data technologies such as Apache Storm, Spark, Heron, Samza, Flink, and other systems to provide in-memory processing for real-time Big Data analysis at high throughput. Spark Streaming represents one of the most popular open-source implementations which handles an ever-increasing data ingestion and processing by using the Unified Memory Manager to manage memory occupancy between storage and processing regions dynamically, which is the focus of this study. The problem behind memory management for data-intensive stream processing pipelines is that the incoming data is faster than the downstream operators can consume. Consequently, the backpressure of Spark acts in the opposite direction of downstream operators. In such a case, the incoming data overwhelms the memory manager and provokes memory leak issues. As a result, it affects the performance of applications generating, e.g., high latency, low throughput, or even data loss. In such a case, the initial intuition motivating our work is that memory management became the critical factor in keeping processing at scale and system stability of Spark. This work provides a deep dive into Spark backpressure, evaluates its structure, presents the main characteristics to support data-intensive streaming pipelines, and investigates the current in-memory-based performance issues.

Keywords: Backpressure, Big Data, Spark Streaming, Stream Processing

Abbreviations

The following abbreviations are used in this manuscript:

API	Application Programming Interface	**PT**	Processing Time
CPU	Central Processing Unit	**RAM**	Random Access Memory
DAG	Direct Acyclic Graph	**RAMFS**	Random Access Memory File System
HDFS	Hadoop Distributed File System	**RDD**	Resilient Distributed Datasets
GC	Garbage Collection	**RC**	Rate Controller
JVM	Java Virtual Machine	**RE**	Rate Estimator
LRU	Least Recently Used	**RL**	Rate Limiter
MPI	Message Passing Interface	**SMM**	Static Memory Manager
MQ	Message Queue	**SD**	Scheduling Delay
OOM	Out Of Memory	**SP**	Stream Processing
OMM	Out Of Memory Management	**SS**	Spark Streaming
OS	Operating System	**UMM**	Unified Memory Management
PID	Proportional-Integral-Derivative		

 doi:10.20944/preprints202205.0334.v1

1. Introduction

Stream Processing (SP) is a trending topic representing a remarkable milestone for real-time data-intensive processing and analysis in both industry and research fields [1,2]. Moreover, SP systems have provided real-time data analysis for numerous network-based applications and services in the most varied areas and domains such as financial services, healthcare, education, manufacturing, retail, social media, and sensor networks [3,4]. .

Nonetheless, it is noticeable the considerable growth of distributed frameworks for the most varied purposes of Big Data analytics such as Apache Storm [5], Samza [6], Apache Spark [7], Flink [8], Amazon Kinesis Streams [9] and others. These frameworks were designed to enable flexible solutions to persist and process data-intensive workloads in memory [10]. Besides, the memory processing minimizes disk I/O movements and reduces the data processing time significantly and outperforms the well-established Hadoop MapReduce implementation [7].

However, data pipelines for SP sometimes produces faster data than the downstream operators can consume. This phenomenon is called backpressure [11]. The backpressure mechanisms have been widely adopted in the most varied domains of SP systems. This mechanism helps applications keep data processing under control by managing data ingestion and processing rates. The management reacts to the processing needs for a graceful response to sudden and intensive loads of data rather than face a system crash [11].

Spark Streaming (SS) provides iterative in-memory data processing with low latency using the Resilient Distributed Datasets (RDD). The RDD processing and cache rely on the Unified Memory Management (UMM), which dynamically manages storage and execution regions in the Java Virtual Machine (JVM). The execution region from Spark supports runtime processing operations like *shuffle*, *join*, *sort*, and *aggregation*. On the other hand, the storage region caches RDD data blocks for both current processing and re-processing tasks as well as stores the incoming data to be further processed [12].

However, the Spark could present performance degradation due to a lack of memory management for very dynamic memory-borrowing operations between execution and storage regions at the UMM level [10,12]. The borrowing operations at the UMM are quite intensive and frequent. Thus, more space is used for the UMM execution and storage regions when data loss is processed. Therefore, the storage area will keep storing the incoming data during the whole processing life cycle, resulting in sudden borrowing operations between the regions. The UMM gives a higher priority to execution memory than to storage memory [13]. Therefore, overloading execution and storage areas will lead to several implications such as significant recomputing overhead, unnecessary data block eviction, long and numerous Garbage Collection (GC), Out Of Memory (OOM) exceptions, throughput degradation, high processing latency, data loss, and memory contention.

Our recent studies unveils that resource management for Big Data analysis is a concern in both batch and stream processing systems [14–19]. Therefore, extending this problem to other SP systems like Flink and Storm can help the JVM data processing and storage operation support by using varied data persisting approaches such as on-heap, on-heap, disk only, and off-heap. The common point of these approaches with Spark is the limited support for data-intensive caching operations due to the restricted size of JVM heap space, disk, or other combinations. Also, the complexity behind the configuration of each approach is hard to manage.

This work provides a deep dive into SS backpressure and its underlying components. Also, it presents an empirical study with an on-heap persisting approach to analyze possible performance crash scenarios for data-intensive SP pipelines. The main contributions are:

i) This study not only presents the SS backpressure model and architecture but unveils its internal stages in terms of in-memory -management for data-intensive pipelines;
ii) This paper proposes a performance evaluation with a data streaming intensive approach similar to real production scenarios. Still, the assessment and remarks of this study point out varied performance insights which can help discussions in SP communities and for other solution sources;
iii) This work proposes a well-defined evaluation that exposes the current limitation of data caching and backpressure for SS. Finally, the current investigation demonstrates that the

Preprints (www.preprints.org) | NOT PEER-REVIEWED | Posted: 24 May 2022 doi:10.20944/preprints202205.0334.v1

constraints may affect SS and other SP systems that have the target to provide in-memory data processing and analysis.

The remainder of this paper is organized as follows. Section 2 presents SS backpressure, its model, and architecture. Section 3 presents in-memory management need for SP systems. Section 4 presents the SS evaluation, results, and insights. Section 5 presents the related work and presents a related discussion section. Finally, Section 6 presents the conclusions.

2. Spark Backpressure: Model and Architecture

Spark backpressure provides dynamic management of processing rates of Spark executors according to the Processing Time (PT) and Scheduling Delay (SD) application performance metrics. It uses the Proportional-Integral-Derivative (PID) controller as core implementation, also known as three-term controller created by Ziegler–Nichols and Nathaniel B. [20]. PID implements a feedback loop mechanism widely used in industry to decrease the complexity behind the configuration of dynamic control systems like the behavior found in data-intensive SP systems. At the implementation level, the backpressure model includes the performance counters of Spark, as seen in Table 1 in the PID model.

Table 1. Spark Performance Counters

Performance Counters	Definition
Time	The timestamp of the current batch interval that just finished;
Events	The number of records that were processed in this batch;
Processing Time (PT)	Time in *ms* it took the job to complete;
Scheduling Delay (SD)	The time in *ms* that the job spent in the scheduling queue;

Figure 1 presents the PID controller implemented by Spark. The goal of the PID model is to automatically calculate processing rates to be applied in the Spark executors' continuously.

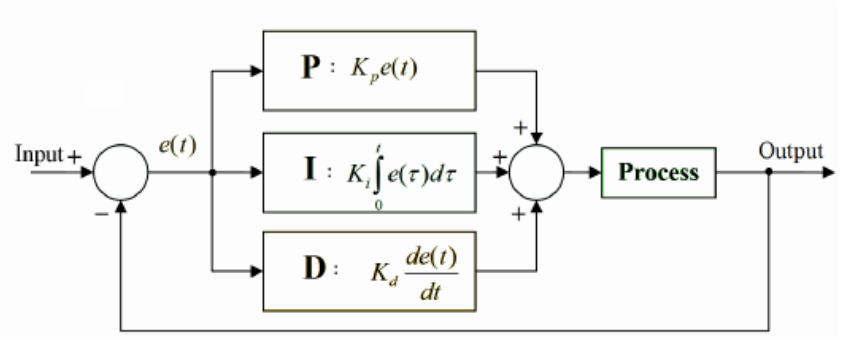


Figure 1. PID Controller Model Implementation

The main components of PID controller implementation[1] are:

- Error $e(t)$: Represents the number of records that overloaded the defined time window from a given batch (**Process**). It is used as the basis to for PID measurements;
- **Proportional (P)**: Term P is proportional to the current obtained error $e(t)$, and this value can be positive or zero, being proportionately adjusted taking into account a gain factor value d - default value is 1. If there is no error, there is no corrective response. At Spark-level, K_p is defined by spark.streaming.backpressure.pid.proportional (default weight: 1 non-negative) and represents the current error over the PT rate $e(t)$;
- **Integral (I)**: The I value is measured based on the past values and integrated on-the-fly to produce output. For instance, if there is a residual error $e(t)$ from proportional control, the integral tries to eliminate this residual error by applying a weight-based factor. If the error is minimized, the I term will be proportional to the error decreases. At Spark-level, K_i, is

[1] PID implementation: https://github.com/apache/spark/blob/master/streaming/src/main/scala/org/apache/spark/streaming/scheduler/rate/PIDRateEstimator.scala

Preprints (www.preprints.org) | NOT PEER-REVIEWED | Posted: 24 May 2022 doi:10.20944/preprints202205.0334.v1

defined by spark.streaming.backpressure.pid.integral (default weight: 0.2, non-negative) and it is well known as a historical error that uses the current SD as an indicator for the overflowing elements that could not be processed in previous batches;

- **Derivative (D)**: Term D represents an estimation related to the current rate of change. It is also called "anticipatory control" because it reduces the effect of error $e(t)$ by rating the rate of error change. If the state changes fastly, this value reacts in the same proportion, controlling or damping the effect. At Spark-level, this is kept as zero since the controller did not expect abrupt oscillation in the processing rates.

In summary, the new processing rate represents the error value $e(t)$ obtained by the difference between a desired rate calculated for the previous output batch ($y(t)$) and a measured rate obtained for the current input batch ($r(t)$), where $r(t)$ represents the number of processed events divided by time window, resulting in $e(t) = r(t) - y(t)$.

As indicated in Figure 1, Spark covers K_p for reacting to the error proportionally, for instance, how much the correction should depend on the current error. If the value chosen for this term is considerable, the controller could overshoot the established point, and a small one makes the controller too insensitive. At the same time, K_i measures how much the correction should depend on the accumulation of past errors. So, K_i accelerates a movement towards the desired value. However, an enormous value may lead to overshooting. The K_d term provides some speculation to seek how much the correction should depend on predicting future errors based on the current rate of change. Usually, K_d is not used very frequently since it can impact SP system stability. Finally, the PID controller presents an exciting property that lacks a guarantee of a stable fixed point [21]. Due to its nature, the algorithm will oscillate around a (near-optimal) constant throughput until stead.

Behind the hype, Spark implements some architectural components to support the PID model, such as Rate Controller (RC), Rate Estimator (RE), and Rate Limiter (RL). Figure 2 presents the Spark backpressure PID architecture. Indeed, RC and RE components execute side by side to deliver the maximum processing speed on the *Driver* side. At the same time, the RL updates the maximum processing rate into the *executors* right after *Driver's* notification update.

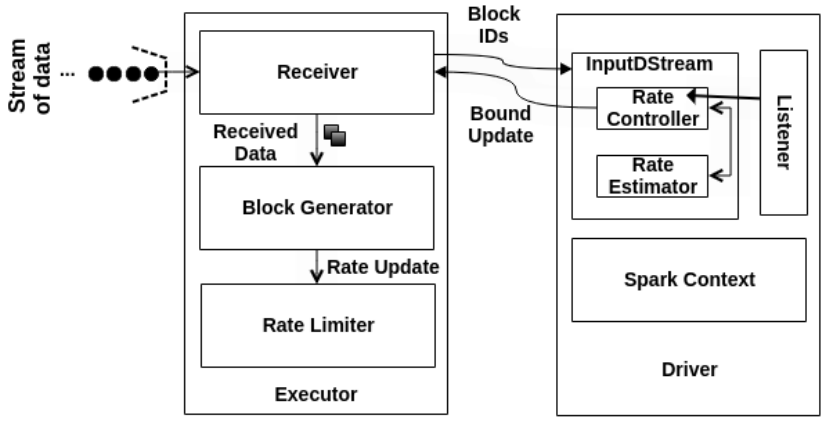


Figure 2. Spark Backpressure PID Architecture

The RC [2] represents a contract for a single Dstream that grabs information from Spark listener for every batch completed to measure the current processing rate and then estimates a new one [22]. The DStream concept allows the processing of the continuous data stream comprised of a RDDs' sequence.

The listener *onBatchCompleted* receives periodic information from all batches jobs, such as the current number of processed events, PT, and SD. Then, this information will be taken by

[2] Rate Controller Implementation: https://github.com/apache/spark/blob/master/streaming/src/main/scala/org/apache/spark/streaming/scheduler/RateController.scala

Preprints (www.preprints.org) | NOT PEER-REVIEWED | Posted: 24 May 2022 doi:10.20944/preprints202205.0334.v1

PID implementation in RE [3] to estimate a reasonable data processing rate. The RE could have multiple-way implementations, but the deploy PID-based [4] version is the default in this work.

Moreover, the PID controller only computes a new rate if there are events to process. Otherwise, the rate estimation is skipped, and no rate limit is returned. Then, RE stores the new rate and forwards it to RL. RL sets the maximum rate of events not to exceed the current load, usually by assigning a value lower than the last established rate. The new bound is related to the number of events that the receivers will allow for processing per second at the executor level.

3. Spark Streaming Memory Management

Spark allows the use of the UMM to provide dynamic allocation for both execution and storage areas. Thus, if the storage or execution environments get insufficient spaces, the memory region will be handled according to a dynamic occupancy mechanism [12]. Finally, Spark relies on JVM for execution and storage and needs to use GC and Least Recently Used (LRU) [23] mechanisms for providing some cleaning of old objects and blocks from memory.

The UMM [5] was introduced in Spark 1.6 to replace the Static Memory Manager (SMM) model. The UMM allocates ***execution*** and ***storage*** as an unified memory region that provides dynamic memory management. Thus, when ***execution*** memory is not used, the ***storage*** memory could acquire all the available memory, and vice versa [12]. Figure 3 shows the UMM diagram.

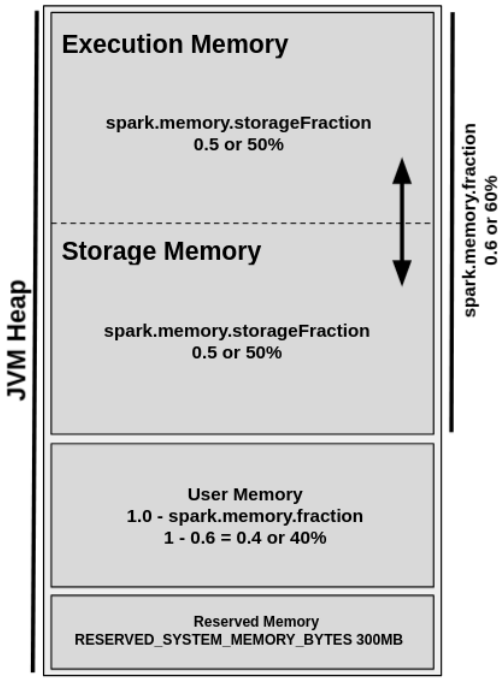


Figure 3. Unified Memory Manager

By default, the *Reserved Memory*[6] is hard-coded, and its size cannot be changed in any way without Spark recompilation or by changing *spark.testing.reservedMemory* setting. The **User Memory** represents 40% of JVM heap memory and stores user-defined data structures and functions, Spark internal metadata, data needed for RDD conversion and operations, and all the information regarding RDD' dependency and others. It is also commonly used as a "backup" to prevent OOM issues.

[3] Rate Estimator Implementation: https://github.com/apache/spark/blob/master/streaming/src/main/scala/org/apache/spark/streaming/scheduler/rate/RateEstimator.scala

[4] PID Rate Estimator Implementation: https://github.com/apache/spark/blob/master/streaming/src/main/scala/org/apache/spark/streaming/scheduler/rate/PIDRateEstimator.scala

[5] Unified Memory Manager: https://github.com/apache/spark/blob/branch-2.4/core/src/main/scala/org/apache/spark/memory/UnifiedMemoryManager.scala

[6] Executor Size: it is recommended executor memory at least 1.5 times of reserved memory size.

Preprints (www.preprints.org) | NOT PEER-REVIEWED | Posted: 24 May 2022 doi:10.20944/preprints202205.0334.v1

Spark Memory Fraction represents 60% of JVM heap memory and unifies the regions responsible for the management of all cached, persisted or other intermediate data. This region is allowed by the configuration of *spark.memory.storageFraction* parameter and enables the management of *storage* and *execution* sub-regions.

The ***storage*** region represents 50% of the **Spark Memory Fraction** region and aims to store all cached data, broadcast variables, unroll process (deserializing data contained in RDD' partitions), and so on. On the other hand, ***execution*** region allows shuffle buffering and stores all the objects required during the execution of Spark tasks and related operations like *shuffle*, *join*, *sort*, and *aggregations*.

Still, ***execution*** region represents 50% from the **Spark Memory Fraction** region and could be defined as "short-lived" than the ***storage*** one because the data blocks could be evicted from JVM heap memory immediately after each operation, making space for new computation. However, even in intensive conditions, these data blocks cannot be forcefully evicted by other running threads (tasks).

3.1. Inefficient Cache Management in Spark

Efficient memory allocation is a critical task for performance in SP systems. In-memory processing frameworks like SS should allow data-intensive processing for various applications. However, it is noticed that Spark could present performance issues due to a lack of memory management for very dynamic and intensive operations at the UMM level. In such a case, the RDD data blocks could be evicted from the UMM storage memory region or even neglected before processing. Consequently, the execution can suffer from several OOM exceptions, job and system crashes, data loss, throughput degradation, and high latency. Figure 4 presents the memory management behavior of the UMM' *storage* and *execution* regions.

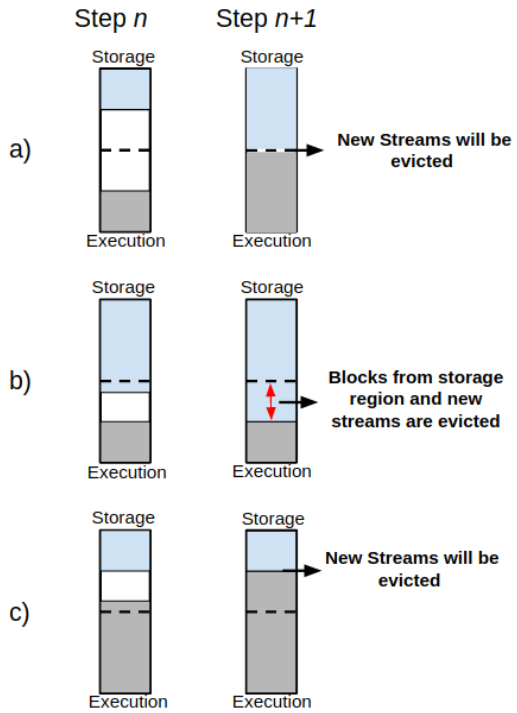


Figure 4. Memory Management Behaviour

In such a case, it is possible to find the following behaviors:

1. The ***storage*** region can borrow space from ***execution*** one only if blocks are not used in ***execution*** region;
2. The ***execution*** region can also borrow space from ***storage*** one if blocks are not used in ***storage*** memory;

 doi:10.20944/preprints202205.0334.v1

3. If space on both ***execution*** and ***storage*** regions are insufficient, then the new stream of data will be spilled off, see Figure 4, a);
4. If ***storage*** region borrows space from ***execution*** region, and ***execution*** region claims it back. The data blocks from the ***storage*** region in the borrowed space from ***execution*** will be evicted gradually until the established limit for the ***storage*** region gets reached, and then the new streams of data will be spilled off until this procedure ends, see Figure 4, b);
5. If ***execution*** region borrows space from ***storage*** region, but ***storage*** region claims it back, then the new streams of data will be spilled off while the ***execution*** region releases the memory space from ***storage***. See Figure 4, c).

Usually, JVM enables user memory to prevent the Out Of Memory Management (OMM) issues, as described earlier in this section. However, the eviction process may also occur between memory borrowing operations since the memory is under pressure due to excessive use from UMM regions.

As noticed, the UMM design prioritizes the execution region and its operations over the storage region. In practice, the UMM design avoids any modification of runtime operations in the execution memory because they are costly for performance[24]. Still related to the performance, the UMM design relies on the premise that the more available memory exists, the better the performance will be. However, it is not a valid premise if memory is under pressure in data-intensive processing scenarios.

Furthermore, if the size for storage memory region is too large or if there is too much data cached, then it will lead to frequent full garbage collection operations, thus reducing task execution performance since the cached RDD data is usually resident in memory for a long time.

Still at the UMM level, Spark provides the LRU strategy to help with the task of Direct Acyclic Graph (DAG)-dependency eviction by removing old cached objects. In such a context, recent studies proposed LRU-based solutions with noticeable improvements in comparison to the Spark LRU strategy [10,12]. However, the main observations highlight that lack of flexibility and related priority is given to the execution over the storage region from UMM.

Summarizing, the JVM heap may represent a point of failure during application processing due to high memory consumption at the UMM level, leading Spark to an under-pressure scenario. Consequently, executors could spend long processing time due to the numerous GC operations, causing long processing delays and system crashes.

Therefore, the less memory space RDD takes up in executors' JVM heap memory, the more heap space will be available for application processing, thus increasing the GC efficiency. Otherwise, excessive memory consumption by RDD leads to significant performance loss (shortened by 10% or 20%) due to a large number of buffered objects at the heap level[25]. The next chapter will present the state-of-the-art memory management for SP systems.

4. Performance Evaluation

This section introduces an investigation of how Spark's Streaming backpressure performs in data-intensive pipelines.

4.1. Environmental Setup

The experimental setup used at this work rely on top of an open-source French Grid5000[7] consortium that provides a modern hardware environment for experimental purposes. The environments chosen for the experiments were Parasilo cluster in Rennes and Dahu cluster in Grenoble. In both cases, the processing scenarios comprise at least a minimum set of 18 nodes and a maximum set of 25 nodes per cluster in total, varying only the number of Message Queue (MQ)s set for each experiment. The hardware of the clusters are presented as follows.

- **Dahu** cluster uses Dell PowerEdge C6420 nodes comprised by Intel Xeon Gold 6130 (Skylake, 2.10GHz, 2 CPUs/node, sixteen cores/CPU) - 64 cores on total (hyper-threading), 192 GB of memory, two interfaces i) 10 Gbps, model: Intel Ethernet Controller X710 for 10GbE SFP+, and ii) Omni-Path, 100 Gbps, model: Intel Omni-Path HFI Silicon 100 Series;

[7] Grid5000 and the hardware specification: https://www.grid5000.fr/w/Hardware

 doi:10.20944/preprints202205.0334.v1

- **Parasilo** cluster uses Dell PowerEdge R630 nodes comprised by Intel Xeon E5-2630 v3 (Haswell, 2.40GHz, 2 CPUs/node, eight cores/CPU) - 32 cores on total (hyper-threading), 128 GB of memory, two 10 Gbps network cards interconnected with a 10G Nexus 56128P network switch.

Each setup allows the use of the software stack presented in Table 2.

Table 2. Software Stack

Operating System Debian 9, Kernel 4.9.0-11 amd64
Hadoop 3.1.2
Spark Streaming 2.4.3
Java 1.8.081
Scala 2.13
OpenMpi 4.0.1
ZeroMQ 3.1.1

Besides, both SS and Hadoop Distributed File System (HDFS) used a Random Access Memory File System (RAMFS) mounted folder to write shuffle data and checkpoints. The RAMFS also supports all installation folders such as data and name nodes of Hadoop, Spark, or even logs of applications like GC, or monitoring tools like dstat monitor[8]. Also, at the Spark side, the monitoring and user interface were deactivated to avoid extra communication costs or any level of interference at the experiments. At the configuration level, the proposed evaluation will consider real-world guidelines, as follows presented.

- Spark could run by using a minimum of 8 GB to hundreds of gigabytes of memory per machine. In general, The recommended work memory space allocation is at least 75% to Spark, leaving the rest for the Operating System (OS) [26]. The proposed testbed pushes up this limit to use up to 90% of available memory to observe the behavior in OS as well;
- JVM does not always behave well with more than 200 GB of Random Access Memory (RAM). In this case, it is recommended to launch multiple executors per worker node and balance all available resources such as memory and cores [26]. At this work this limit was not reached, and each worker in the Spark cluster will hosts only one *Executor* that will receive all the available resources from the given node. Still, the *Executor* will host only a single receiver;
- The communication must rely on high-speed and low latency networks like a 10 Gigabit switches or higher to allow data processing at scale [26]. The cluster used in this work provides 10 Gigabit switches;
- Although SS allows the use of varied data persisting approaches. This work will make the use of on-heap JVM approach to handle both data processing and storage at memory level only;
- G1GC[9] garbage collector is indicated by Spark to improve performance in bottleneck scenarios [27]. This work keeps this standard as a default configuration.

Table 3 summarizes the configuration set for Spark in both clusters. The batch interval represents the window time (2000ms) in which Spark will receive and process data. The RDD block generation happen every 400ms. The values chosen reflects a near-real time processing in which the applications could process data at high-scale. All the experiments will be performed using only one DAG in order to observe how master process behave in the driver node. This work keep default parallelism of Spark since it follows the number of partitions in RDDs, and in this case we optimized the partitioning to fits with the number of the available cores in the cluster. Thus, it is possible to extract the maximum performance, pushing up memory utilization as happens in real-world use cases. Finally, the JVM heap set follows 90% from the available memory per node, and the table also shows the conceptual values from the Spark UMM regions.

Table 4 presents the pipeline configurations used in the evaluations. It is proposed two configurations to push data processing until the limit. The idea is to obtain a better understanding of the current limitations from SS surrounding data SP in scenarios with and without a backpressure

[8] Dstat-based monitor tool: https://github.com/mvneves/dstat-monitor
[9] G1GC is the low-pause, server-style generational garbage collector for Java HotSpot VM

Preprints (www.preprints.org) | NOT PEER-REVIEWED | Posted: 24 May 2022 doi:10.20944/preprints202205.0334.v1

Table 3. Spark Configuration

Parameters	Rennes Parasilo Cluster	Grenoble Dahu Cluster
Window (batch interval)	2000ms	2000ms
Block interval	400ms	400ms
Concurrent DAGs	1	1
Spark Parallelism	default	default
#Driver instances	1	1
#Executors instances	8	8
#Receivers per Executor	1	1
Main Memory per node	128GB	192GB
Driver JVM Heap Memory	117GB	174GB
Executor JVM Heap Memory	117GB	174GB
Executor UMM storage Region	33GB	49GB
Executors Global Storage Region Memory	264GB	392GB
Executors Global JVM Heap Memory	936GB	1392GB
Cores per Executors (HT)	32	64
# Total Cores in the Spark Cluster	256	512
JVM Memory Schema	On-heap	On-heap
GC Type	G1GC	G1GC

mechanism. *Pipeline 1* represents a soft-sized pipeline that uses a single MQ for data forwarding. *Pipeline 2* represents high-sized once it increases to eight the number of MQs in data forwarding layer, introducing a data-intensive approach.

Table 4. Pipeline Configurations

Cod.	Size	#Data Sources	#MQs Nodes	#Driver Nodes	#Executors (one per worker node)
Pipeline 1	Soft	8	1	1	8
Pipeline 2	High	8	8	1	8

It is necessary to point out some observations regarding the pipeline configurations and its usage. This work aim to investigate and understand how backpressure works in intensive conditions. For this reason, some rules were adopted: i) the cluster will be not shared; ii) the experiments will be made in low-latency and high-speed network connections; iii) Spark processing should be performed in a non-intrusive environment. In this case, each worker node hosts only one executor, each executor will be comprised by a single receiver, and each executor will receive all available resources from such worker node.

For all experiments, a data-intensive scientific simulation called Stencyl was used to generate numeric data at scale by using Message Passing Interface (MPI). The Stencyl implementation generates data based on a heat distribution simulation in a 2D domain relied on the Jacobi iterative method. The progression of the simulation is recorded as synchronous time steps. At the end of a time step, the simulation data is forwarded to the MQ. The messages were configured comprises 10000 floating points elements (approximately size of 10KB per message) per Message Passing Interface (MPI) rank for each application time step. The MPI ranks were set to 256 in the Parasilo cluster and 512 (representing the number total of processing cores per cluster) for the Dahu Cluster, representing simultaneous queues connected to the MQ to maximize the usage of resources during the execution of an application.

The flow of messages is intensive and continuous to keep data communication and processing at high rates. However, if any sending operation fails, the messages will be added to a buffer in the respective queue. Still, if the queue buffer is full, the application starts the re-transmission phase, avoiding any data loss. Besides, ZMQ provides I/O threads that take care of the message transmission, thus immediately unlocking the MPI process for computations.

 doi:10.20944/preprints202205.0334.v1

In the current implementation of ZMQ, the ZMQ buffer was set equal to 1000 (default value), and the number of I/O threads is the same number of CPU cores for each MQ node. It is essential to mention that one I/O thread can support at least one gigabyte of data in or out per second [28]. Still, the MQ layer uses intermediary nodes in which its amount depends on the system's configuration, as presented at Table 4. Those intermediary nodes receive messages from N simulation processes and retain them in-memory into an input queue buffer before forwarding them to M Spark receivers in the executors.

At the consumer level, we provide a Stateful SUMServer application to be measuring a mean computation of scientific simulation data, ensuring all MPI ranks from a given time step into a window will be considered in the current analysis. This application uses a Dstream concept where the RDD is independent, and no data is kept between two *time windows*.

Thus, to keep information from one RDD to another, a state operator must be used. The state operator is a memory-based element that synchronizes a state over a distributed file system between two *time windows*. The state operator associates states' information to a key-value pair. The behavior of the state operator must be manually written. Hence, the first time a key-value pair is met, the state is created. When meeting the key again, the user can update the state value.

In this context, data must be removed manually from the state operator to avoid memory overflow. Thus, the implementation focused on *mapWithState()* function because it provides better performance, being able to provide 8x lower latency (i.e. processing time) and having the capacity to maintain 10x more keys than *updateStateByKey()* [29]. Furthermore, this function allows native support for state timeouts by the *isTimingOut()* function. Then, it is possible to define the period to hold the states in memory before checkpointing. In this work, each state remains in memory for *timeout(new Duration(20000)))* ms.

The execution time for each experiment was defined as a minimum of 1800 seconds. The evaluation will present only a subset of the numerous experiments performed to validate our assumptions. Finally, the measured metrics in the experiments were PT, SD in milliseconds, timestamp in seconds, and throughput in megabytes per second (MBps).

4.2. Native-Based Performance Analysis

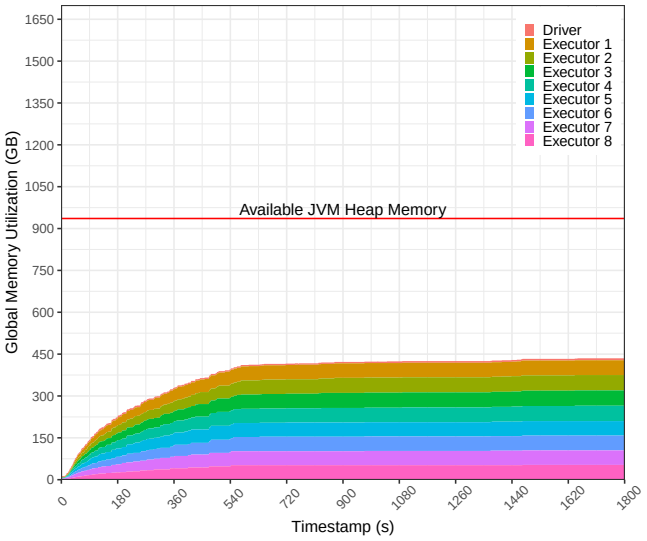
This section presents a native-based performance analysis of SS without a backpressure mechanism for data-intensive processing pipelines. This experiment aims to verify potential memory-based issues and their related performance impacts on the Stateful SUMServer application. The pipelines used in this evaluation are described in Table 4, and the Spark configuration is presented in detail in Table 3.

Figure 5 summarizes the experiments for the SUMServer Stateful application over Pipeline 1. Figures 5(a) (Parasilo) and 5(b) (Dahu) presents a global JVM heap memory utilization in GB. These figures are stacked and aim to present the total memory used per Executor and Driver components over time in seconds. Still, the Figures 5(c) (Parasilo) and 5(d) (Dahu) presents the total delay related to PT, SD metrics, and their respective averages in ms over time in seconds. Finally, the Figures 5(e) (Parasilo) and 5(f) (Dahu) the aggregated throughput average for data sent by the MQ and processed by Spark.

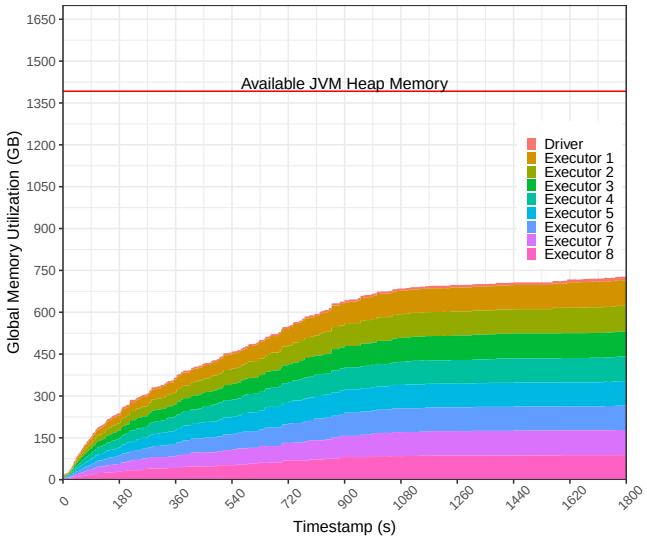
The experiments in Pipeline 1 demonstrate how simple changes in the application life cycle may affect the overall SP pipeline performance. Figure 5(a) presents a regular use of the JVM heap memory in the Parasilo cluster. It demonstrates that almost 50% of allocated heap memory is free to be used for *Executors* and *Driver* instances.

It occurs because Spark completely takes the incoming data, and there is no data being cached in heap memory, leading to regular and stable use of the heap memory. The PT and SD reinforce this assumption since the metrics were intensive at the begging due to high data ingestion but steady when data processing was faster than data sending to the MQs. In this case, the PT stabilized below the defined time window (1585 ms on average), and the SD did not present any intrusiveness (519 ms on average), indicating underutilization of resources, see Figure 5(c).

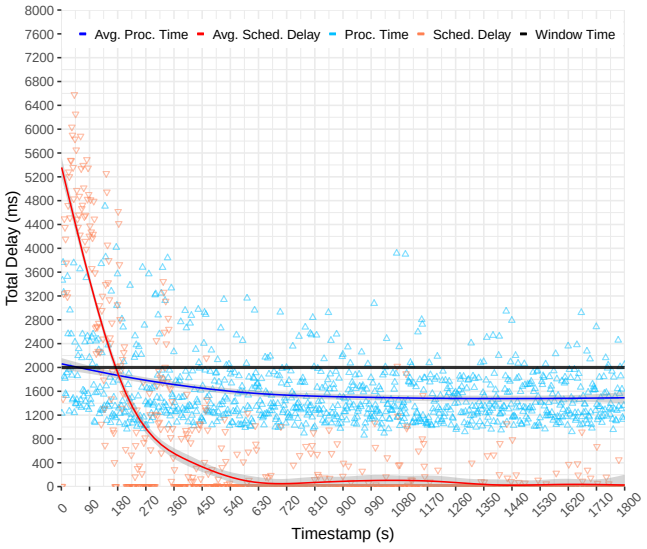
Preprints (www.preprints.org) | NOT PEER-REVIEWED | Posted: 24 May 2022 doi:10.20944/preprints202205.0334.v1



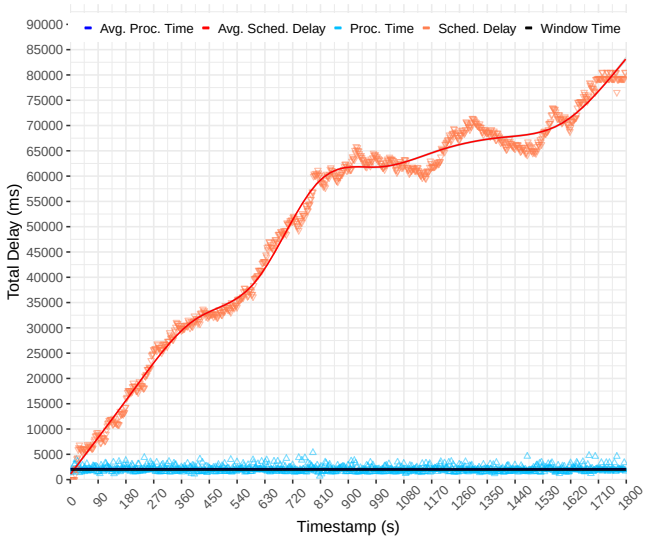
(a) Global JVM Heap UtilizationParasilo Cluster



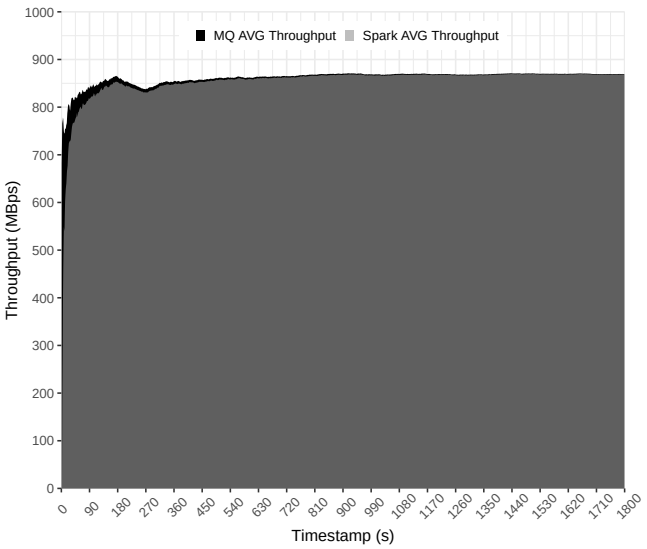
(b) Global JVM Heap UtilizationDahu Cluster



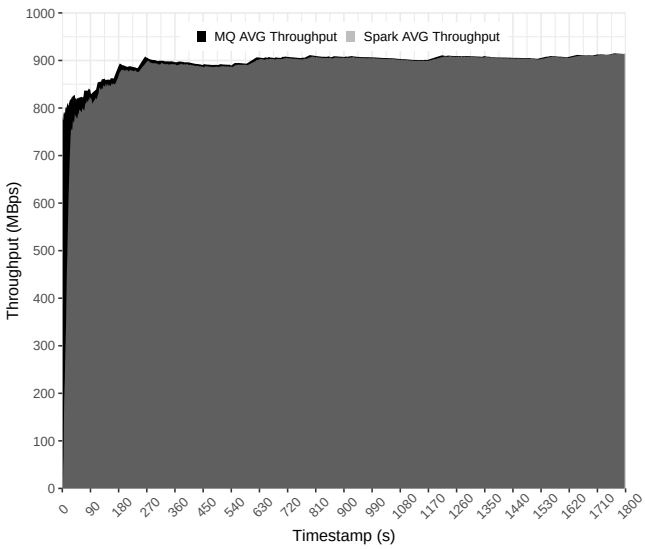
(c) Total Delay, Parasilo Cluster



(d) Total Delay, Dahu Cluster



(e) Throughput Average, Parasilo Cluster



(f) Throughput Average, Dahu Cluster

Figure 5. Stateful SumServer Application Without Backpressure - Pipeline 1

 doi:10.20944/preprints202205.0334.v1

In the opposite direction, Figure 5(b) for cluster Dahu. The JVM hep memory utilization grows over time, and the heap is not uniform, leading to a potentially unstable processing scenario. It happens because Spark Streaming queues data up in memory for later processing, and this behavior is reflected in heap memory utilization.

In this case, data ingestion is slightly faster than the processing capacity for a single time window. The PT, for instance, reached up to 2094ms on average, see Figure 5(d) (Grenoble). Unfortunately, case Spark can not keep PT alongside window time. The SD will keep growing until the JVM heap memory gets full, overloading Spark UMM execution, storage, and user memory regions in the JVM. It is important to mention unpredicted data ingestion as presented in the Dahu cluster, which reached a SD of 50679ms on average, which may lead the application to an unstable processing scenario. As a consequence, the system falls behind due to OOM at JVM heap memory.

Finally, it is possible to observe Figures 5(e) and 5(f) presented a similar average throughput, slightly better in the Dahu cluster. Although the Dahu cluster allowed promising throughput gain for long-term applications, the average PT obtained overpasses window time, demonstrating the system starts to be under pressure. Table 5 summarizes the obtained performance indicators for the evaluated scenario.

Table 5. Performance Indicators of Stateful SUMServer Application - Pipeline 1 without Backpressure

Metrics	Parasilo	Dahu
AVG Th (MBps)	870	918
AVG PT (ms)	1585	2094
AVG SD (ms)	519	50679
AVG Proc. Events	95051	100279

The second set of experiments uses the Pipeline 2 to evaluate how SS reacts to the data-intensive pipelines. Figure 6 summarizes the experiments for the SUMServer Stateful application over Pipeline 2. Figures 6(c) (Parasilo) and 6(d) (Dahu) presents the per node JVM heap utilization in Parasilo and Dahu Clusters. The figures aim to present the total memory used per Executor and Driver components over time in *seconds*. Still, the Figures 6(a) and 6(b) presents the total delay related to PT, SD metrics, and their respective averages in ms over time in seconds.

Figures 6 revealed limitations related to the internals of SS and its incapacity to support data management and memory coordination for a sudden surge of data. As we can see in Figures 6(a) (Parasilo) and 6(b) (Dahu), PT is at least five times greater than the time window. Although this behavior helps to increase data throughput, the Spark UMM execution and storage memory will be fastly overwhelmed with data.

Thus, as UMM has priority during processing tasks. It can disturb memory-boring operations at the UMM level and raise data block exceptions. It means data will be lost, and performance issues will be imminent. In fact, Table 6 presents exactly this behavior. In such a case, after data ingestion peaks, several exceptions happened, and the loss of data was noticed.

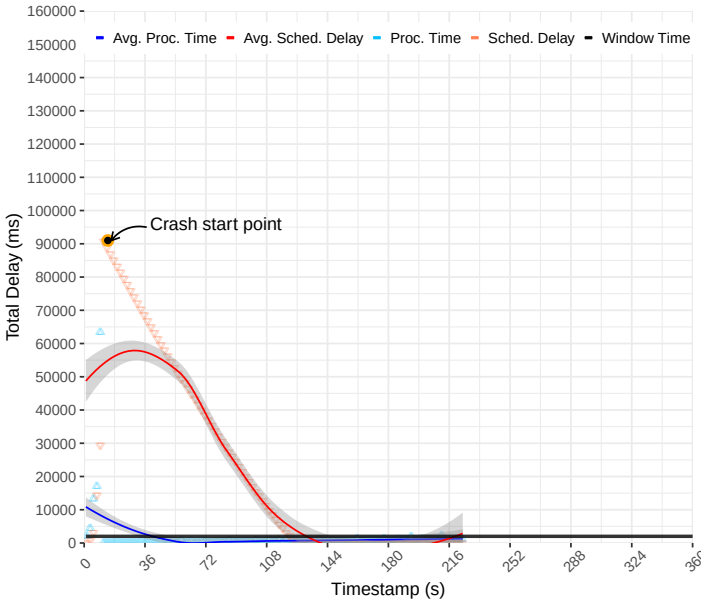
Table 6. Performance Crashing Indicators of Stateful SUMServer Application - Pipeline 2 without Backpressure

Metrics	Parasilo	Dahu
MAX PT (ms)	63371	69044
MAX SD (ms)	90792	137480
Crashing Start Time (sec)	14	26

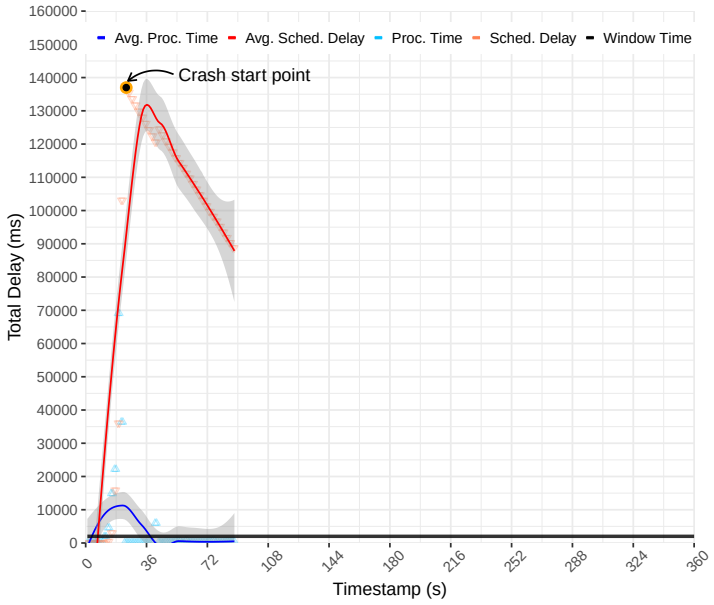
Consequently, the SD increases fast since the Spark cluster cannot handle the incoming data properly. It may lead to a crash scenario since too much data is waiting to be processed, more than one minute in Parasilo and more than two minutes in the Dahu cluster.

It is possible to see in Figures 6(c) and 6(d) the UMM memory regions (execution and storage) from executors' getting full of data, spilling data to user memory. In such a case, the executors' may start to evict data blocks from memory by using the LRU algorithm to free some space for be storing the new RDD blocks.

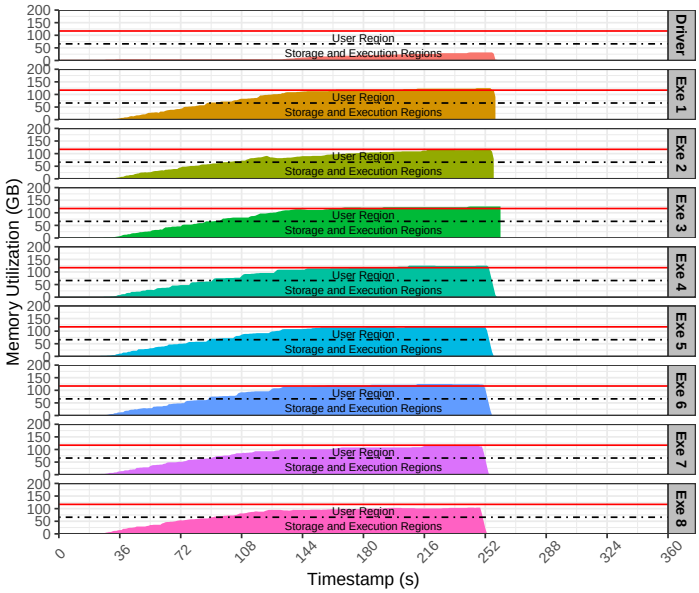
Preprints (www.preprints.org) | NOT PEER-REVIEWED | Posted: 24 May 2022 doi:10.20944/preprints202205.0334.v1



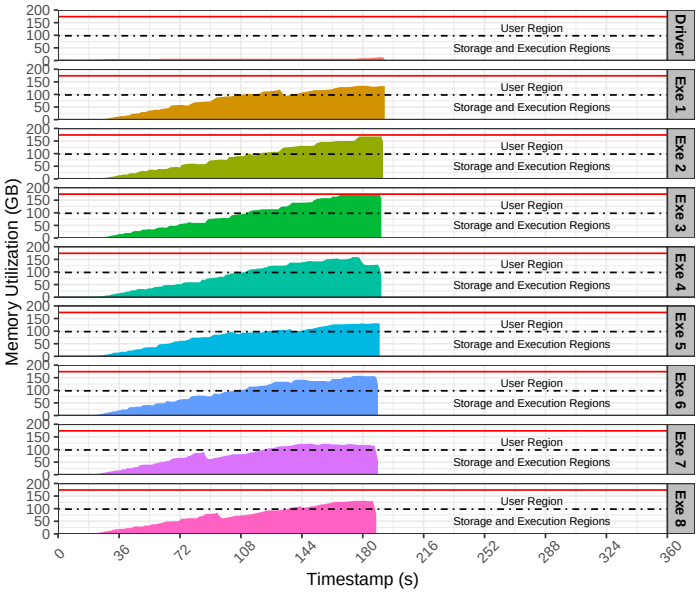
(a) Total Delay, Parasilo Cluster



(b) Total Delay, Dahu Cluster



(c) Per Executor JVM Heap Utilization, Parasilo Cluster



(d) Per Executor JVM Heap Utilization, Dahu Cluster

Figure 6. Stateful SumServer Application Without Backpressure - Pipeline 2

Furthermore, the sudden surges of data may congest Spark Block Manager at the Driver's daemon. It overwhelms the Driver with metadata from RDD block, states, shuffle operations, data caching, and processing references. Consequently, the communication between the executors' process will take longer, reproducing an invalid state of the system, and incurring in application processing failures at any execution point. See The **Start Crash Points** at Figures 6(a) (Rennes) and 6(b) (Grenoble).

These points represent an internal noise at the system level because receivers continuously receive incoming data and perform data processing tasks. In contrast, PT decreases until 0, while SD follows the same direction. In summary, if the incoming is not under control and PT is not steady, the eviction process becomes inevitable. In such a case, Spark cannot be considered stable anymore, losing data and collapsing completely due to OOM issue at JVM heap memory.

Preprints (www.preprints.org) | NOT PEER-REVIEWED | Posted: 24 May 2022 doi:10.20944/preprints202205.0334.v1

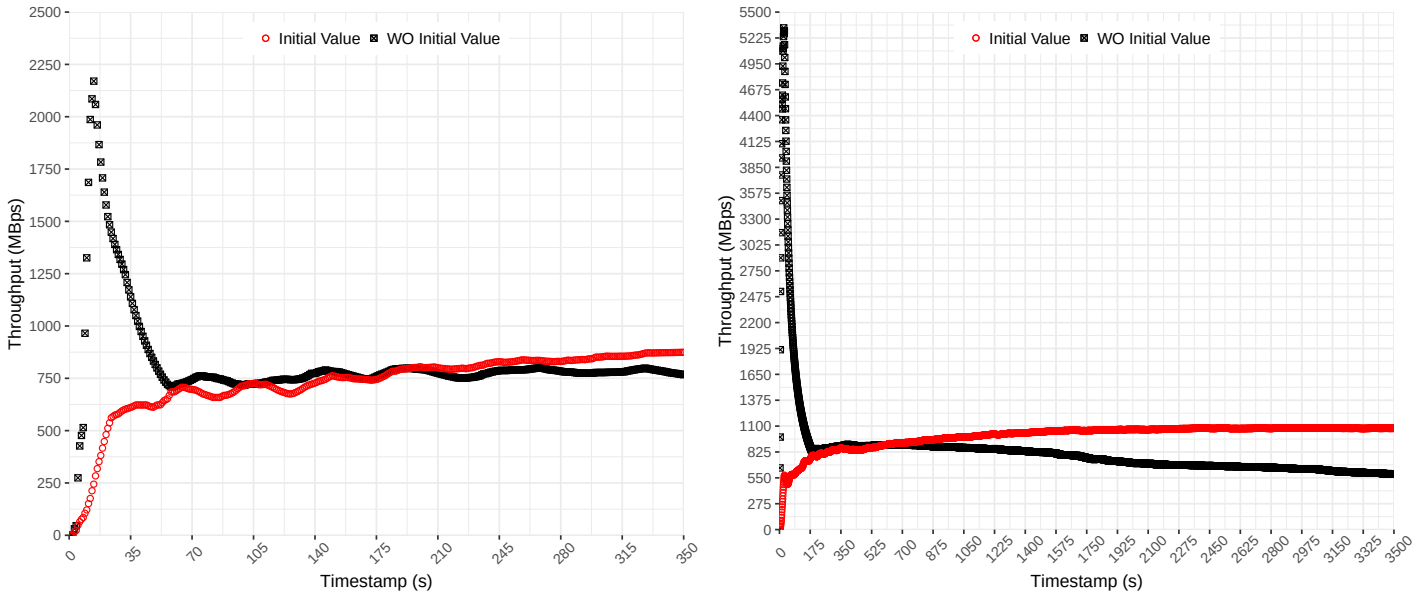
Finally, the preliminary results indicated Spark could not handle a sudden data surge. Still, it is noticeable the need for a system or feature to help in controlling data processing for SP applications. This is the point where Spark backpressure comes in. It was designed to handle the sudden surge of data and to keep processing stable. In such a case, the following section evaluates how Spark handles data-intensive pipelines by applying the backpressure mechanism.

4.3. Backpressure-based Performance Analysis

This section presents a backpressure-based Performance analysis. This evaluation investigates how Spark reacts to data-intensive processing pipelines using the backpressure mechanism and related components. The first step of this analysis evaluates the use of spark.streaming.backpressure.initialRate property and its related impact on application performance.

The spark.streaming.backpressure.initialRate property helps Spark backpressure to define the initial maximum receiving rate of the executors. The experiments were conducted using the Stateful SUMServer application on Dahu and Parasilo clusters over Pipeline 2. Observe the pipeline configuration at Table 4, and the Spark Configuration at Table 3.

Figure 7 shows Spark's throughput for Stateful SUMServer application in MBps for Dahu and Parasilo clusters. Each figure presents the application performance with and without the initial rate property configured. The initial rate value was set to allow up to 2000 records per executor. It represents a small value that introduces a slow start to help the PID algorithm in the preliminary measurements.



(a) Parasilo Cluster (b) Dahu Cluster

Figure 7. Backpressure Initial Rate Feature Comparison for Stateful SUMServer Application - Pipeline 2

In the experiments performed without an initial value set for the backpressure, it is possible to observe a sudden surge of data at the beginning of execution, see Figure 7(b). Although it presented a high throughput for a short time, it ramped up the SD up quickly. Thus, even backpressure keep processing rates under control. Spark accumulated a huge amount of data in the executors' cache, requiring a considerable effort from PID backpressure to stabilize the processing rates on the fly. See the obtained average throughput for this comparison in Table 7.

Then, backpressure identified that the processing rate was greater than the time window and suddenly decreased the number of events allowed to be processed per executor. This process is effective but time-consuming since the state remains in memory for a long time, degrading performance over time.

In comparison, in the experiments performed with an initial value set for the backpressure, the initial rate slowed down the application's processing at the beginning of execution. Although

Preprints (www.preprints.org) | NOT PEER-REVIEWED | Posted: 24 May 2022 doi:10.20944/preprints202205.0334.v1

processing slows at the beginning, it helped the PID controller to adjust to steady processing rates, see obtained throughput average (Initial Values) in the Figures 7(a), 7(b).

Table 7. Backpressure Initial Value Comparison

	Initial value set Average Throughput (MBps)	**Initial value not set** Average Throughput (MBps)
Parasilo	874	764
Dahu	1076	590

Finally, the results indicate that initial values improved performance to 14% in the Parasilo cluster and 82% in the Dahu cluster. The following section presents the backpressure evaluation for data-intensive SP pipelines. The following experiments uses Stateful SUMServer application with backpressure spark.streaming.backpressure.initialRate feature enabled and set to 2000 records per executor.

Figure 8 presents the global JVM Heap memory utilization in GB for the Stateful SUMServer application in Pipeline 2 at Figures 8(a) (Parasilo) and 8(b) (Dahu). The figures are stacked and aim to present the total memory used per Executor and Driver components over time in seconds. Figures 8(c) and 8(d) shows per executor JVM heap utilization. Still, Figures 8(e) (Parasilo) and 8(f) (Dahu) presents the total delay, which shows the PT, SD metrics, and their respective averages over time in seconds.

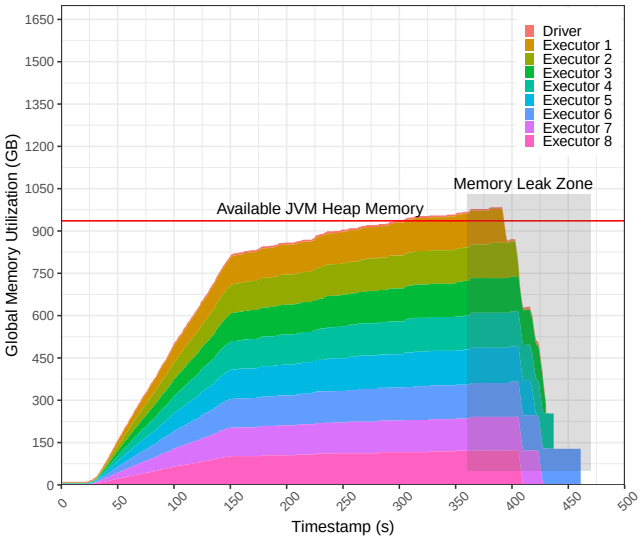
It is possible to observe that backpressure failed to keep processing under control, leading to a crash issue in both clusters. It occurred due to a lack of management of incoming data at the Spark level, meaning the MQ keep sending data to Spark receivers without any control. Internally, the backpressure took care of SD and PT metrics by managing executor processing rates to fits with window time as presented in Figures 8(e) and 8(f).

However, backpressure could not keep UMM healthy since Spark UMM execution has priority over the storage region, leading to a memory starvation condition. In this case, Spark keeps filling the storage region that tries to borrow space from the execution region, the user region will be used as "backup" to avoid OOM issues since Spark was configured to use the on-heap data persistence. Thus, case JVM gets full of data, a new stream of data, and a set of old RDD blocks will be evicted from memory to allow free space for new computation and storage needs. Thus, as SUMServer can not compute their jobs due to data loss, the application falls behind, raising an issue known as *BlockNotFoundException*.

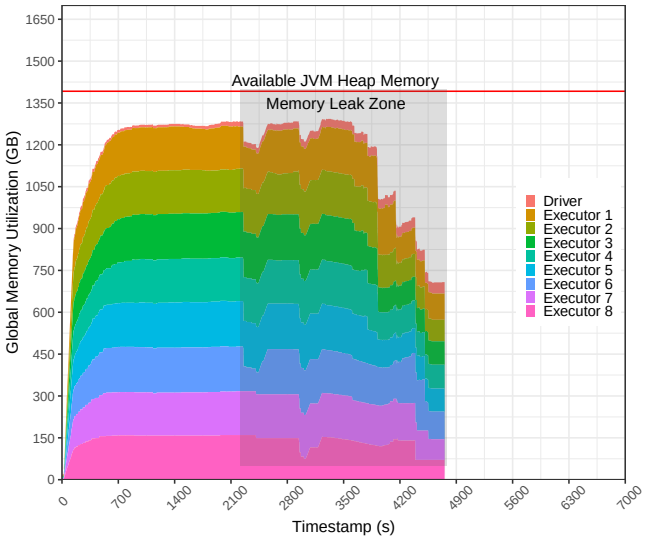
The eviction process may be observed in Figures 8(a) and 8(b), observe the **Memory Leak zone** regions in the Figures. Still, it is possible to observe in depth the JVM heap utilization per executor at Figures 8(c) and 8(d). Figure 8(a) reveals the exact moment of the OOM issue, looks at second 220 and analyzes the Memory Utilization of the executors'. Still, it is possible to see them reaching the JVM capacity side by several SD oscillations before a full outage, as demonstrated in Figure 8(e). Similarly, the Dahu cluster presents the eviction process at the time slice 2200, 3000, 3800 (seconds), and others, as observed in Figure 8(f).

Such a situation may occur because the storage region at executors gets full of data, then borrow space from execution and fill it, and finally overload the user region. Thus, if a single executor starts to fail, it can lead to a system crash that will result in several GC operations or data eviction from memory by the LRU strategy from Spark. Finally, it is possible to observe that small-sized or non-intensive applications may use backpressure without any problem since those scenarios do not push Spark to the limit.

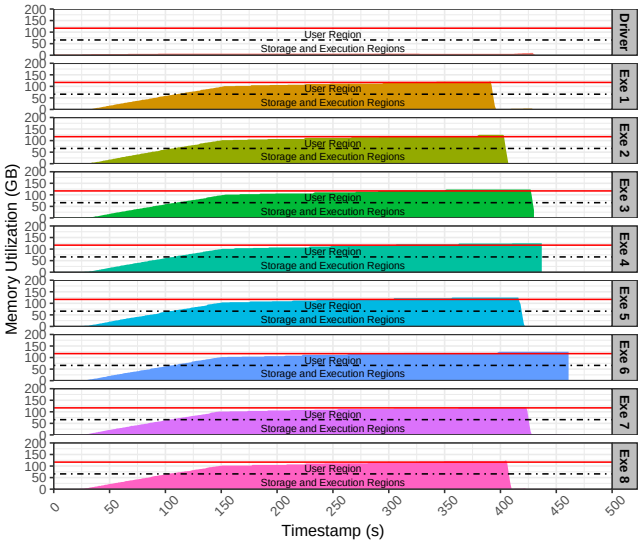
Preprints (www.preprints.org) | NOT PEER-REVIEWED | Posted: 24 May 2022 doi:10.20944/preprints202205.0334.v1



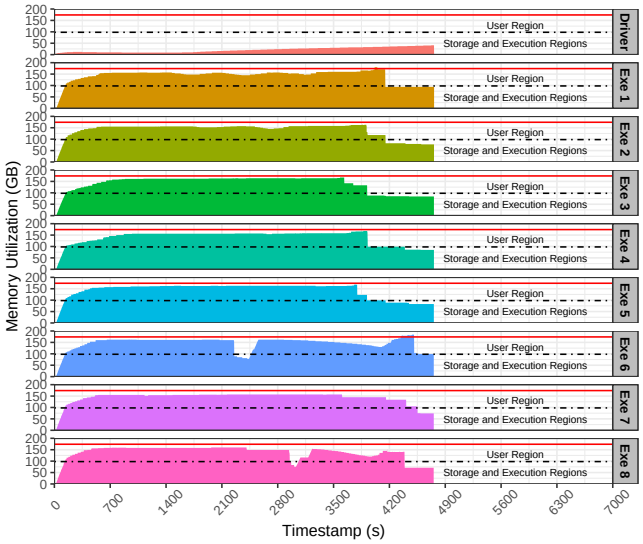
(a) Global Heap Memory Utilization, Parasilo Cluster



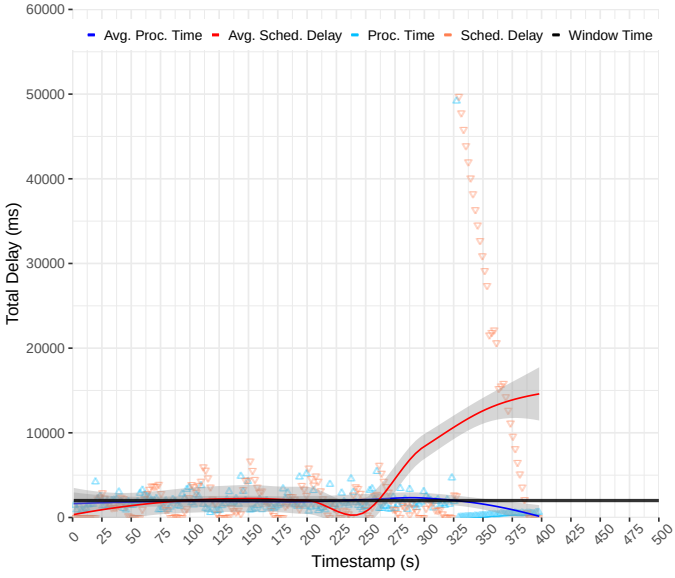
(b) Global Memory Utilization, Dahu Cluster



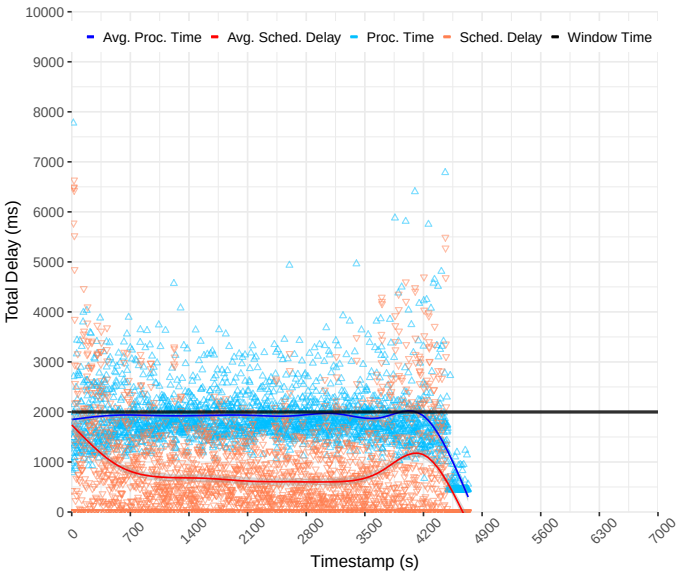
(c) Per Executor Heap Memory Utilization, Parasilo Cluster



(d) Per Executor Heap Memory Utilization, Dahu Cluster



(e) Total Delay, Parasilo Cluster



(f) Total Delay, Dahu Cluster

Figure 8. Stateful SumServer Application With Backpressure - Pipeline 2

 doi:10.20944/preprints202205.0334.v1

4.4. Garbage Collection Comparison Analysis

This section aims to understand how GC operations imply in the performance of data-intensive SP scenarios. It presents a performance comparison of the stateful SUMServer application performed under stress conditions with and without Spark backpressure over Pipeline 2. The GC metrics were collected for both Spark driver and executor instances by setting the spark.driver.extraJavaOptions configuration parameter and the respective value *"-XX:+PrintGCDetails -XX:+PrintGCDateStamps -Xloggc:file.log"*. The obtained data were parsed through the *Universal GC Log Analyzer tool Gceasy*[10]. The Gceasy application provides the following GC processes metrics from the logs:

- **Ergonomics**: it is a JVM and garbage auto-tuning process which dynamically tunes the JVM heap size to meet the application needs with minimum pauses;
- **Allocation Failure**: it happens when there is not enough free space to create new objects;
- **GCLocker Initiated GC**: this process prevents GC operations when JNI code is in a critical region. So, if GC is needed while a thread is in a critical region, then it will allow them to complete, i.e., call the corresponding release function;
- **Metadata GC Threshold**: this process happens when configured meta-space size is smaller than the current system requirements;

These processes may lead the JVM to perform the following GC operations:

- **Minor GC stats**: it collects garbage from JVM spaces. Minor GC is always triggered when JVM cannot allocate space for new objects, e.g., one region getting full. So, the higher the allocation rate is, the more frequently Minor GC will be executed.
- **Full GC stats**: it cleans the entire Heap – all memory spaces.

Furthermore, we can also observe some performance indicators, such as:

- **Throughput (%)**: percentage of time spent in processing real transactions vs. time spent in GC activity. A higher percentage indicates GC overhead is low;
- **Avg Pause GC Time (ms)**: this is the average amount of time taken by one Stop-the-World GC pause;
- **Max Pause GC Time (ms)** this is the maximum amount of time taken by one Stop the World GC to run;

Table 8 summarizes the sources of GC operations, the GC operations, and some performance indicators for the scenario Stateful SUMServer application without backpressure. Initially, it is possible to observe that the Driver instance is achieving almost 100% of *Throughput* in both environments. It demonstrates that JVM heap memory at the Driver node is healthy and does not represent a bottleneck. It occurs because JVM heap memory utilization at the Driver is under control, resulting in a lower number of GC operations and maintaining low latency when operations are required. Although Driver is healthy, the Executors will keep processing and storing data at a high level in memory, overloading the JVM heap.

Thus, the pressure on the JVM introduced some performance degradation. For instance, the throughput metric for executor degraded by at least 55% for both environments compared to the scenario with (Table 9) and without (Table 8) backpressure enabled. It is possible to observe that the performance degradation is mainly related to the successive GC operations made in both scenarios. Also, they mainly came from allocation failures that happen when there is not enough free memory space to create new objects in memory. Thus, if the system is not under control, minor GC events will continuously try to allocate space for new objects to keep processing stable. However, if the Eden region is getting full, a major GC event may occur, clearing the entire JVM heap.

[10] GC Analyser tool: https://gceasy.io/

Preprints (www.preprints.org) | NOT PEER-REVIEWED | Posted: 24 May 2022 doi:10.20944/preprints202205.0334.v1

Table 8. GC Statistcs for Stateful SUMServer Application Without Backpressure - Pipeline 2

	Dahu Cluster		Parasilo Cluster	
	#GC Events Driver	#GC Events Executors	#GC Events Driver	#GC Events Executors
Ergonomics	0	133	0	19
Allocation Failure	13	1392	36	184
GCLocker Initiated GC	6	25	0	9
Metadata GC Threshold	0	22	3	2
Total	19	1572	39	214
GC Operations				
Minor GC stats	16	1428	36	194
Full GC stats	3	144	3	20
Total	19	1572	39	214
Performance Indicators				
Throughput %	99	45	99	45
Avg Pause GC Time (ms)	34	835	57	591
Max Pause GC Time (ms)	90	10184	160	7920

Table 9 summarizes the sources of GC operations, the GC operations, and some performance indicators for the scenario running Stateful SUMServer application with backpressure. Initially, it is possible to see Driver achieving 100% of *Throughput* in both environments like presented before. The GC operations at the Driver side increased since it must be more active than a scenario where Spark suddenly crashes.

Table 9. GC Statistcs' for Stateful SUMServer Application With Backpressure over Pipeline 2

	Dahu Cluster		Parasilo Cluster	
	#GC Events Driver	#GC Events Executors	#GC Events Driver	#GC Events Executors
Ergonomics	1	125	0	38
Allocation Failure	725	1045	95	152
GCLocker Initiated GC	6	18	6	0
Metadata GC Threshold	1	32	0	30
Total	733	1220	101	220
GC operations				
Minor GC stats	731	1079	98	167
Full GC stats	6	141	3	53
Total	737	1220	101	220
Performance Indicators				
Throughput %	100	99	100	95
Avg Pause GC Time (ms)	26	338	21	238
Max Pause GC Time (ms)	300	1854	90	755

Finally, the obtained results at the Executors' were slightly better than a non-backpressure scenario. Both applications were kept alive and stable for a longer time, more than 300 in Parasilo and more than 2000 in the Dahu cluster, and maintained satisfactory throughput level near 100%. Furthermore, backpressure was quite suitable since Dahu decreased the number of GC operations by more than 30%. At the same time, Parasilo maintained a similar number of operations but for a longer execution time.

5. Related Work

An essential requirement of SP systems is robustness against variations in streaming workloads. For example, the SP should adapt quickly to sudden spikes in workload demands. This section investigates how SP systems handle incoming data from varied streaming sources without degrading applications throughput. Still, this section aims to understand the existing solutions and their

current limitations to point out opportunities regarding memory management for data-intensive SP pipelines.

Das, T. et all. [30] presents an adaptive batch sizing strategy for SP systems. It is based on a fixed-point iteration solution, a well-known numerical optimization technique that allows the system to adapt window size when incoming data dynamically varies. Thus, it is possible to minimize end-to-end latency while keeping the system stable based on the statistics of the last two completed batches. This strategy allows better use of resources since it avoids high processing delays and load spikes, which lead the SP system to build up batches in memory, and results in low throughput performance and system crashes.

Research gap and opportunities: although the solution does not present related-memory management strategies that directly impact the memory governance task. It prototypes an end-to-end controller that introduces data orchestration and load balancing using a batching strategy. Still, it can be considered a promising solution that introduces the queue concept widely adopted by MQ systems. Thus, a queue-based global controller can manage incoming data between up and downstream operators in a data-intensive SP pipeline.

Birke, R. et all. [31] proposed a data-driven latency controller that estimates how much data can be processed in a single time window. The solution is based on performance metrics obtained from Spark execution, such as SD and PT. Then, for each time step, the solution will measure the current SD and PT to define the new processing rate. Still, if the incoming data overflows system capacity, a shedding threshold will be set to drop data out. Thus, the new data blocks will be accepted if they fit into the current time window. Otherwise, they will be dropped out by the shedding strategy to avoid high load spikes.

Research gap and opportunities: this work is quite similar to the current backpressure mechanism allowed by Spark. However, the major drawback of this work relies on the data shedding strategy. Thus, even decreasing processing delays, the solution does not avoid data loss and may get the worst results in data-intensive SP pipelines. Still, this solution ignores the memory utilization from Spark or related-memory strategies that directly impact processing performance.

Chen, Xin. et all. [32] presented a checkpointing feedback controller as a complementary mechanism to act alongside Spark backpressure to manage checkpointing time. It was made to achieve a solid execution and a high throughput, similar to PID schema in the Spark framework. The solution collects historical data such as SD and PT from past batch jobs between a set of checkpoints. In this case, the author defines one region as a collection of 10 seconds (10 jobs), where nine are set for processing and one for the checkpoint. Then, based on the retrieved information from processing, it is possible to measure the number of tuples and minimize the data ingestion of the next jobs to decrease the delay cost associated with checkpoints task gradually. It represents the similar behavior applied by PID to Spark receivers.

Research gap and opportunities: Spark allows the use of the *.timeout()* [11] function to control state checkpointing persistence. In such a case, it is highly recommended to specify this feature for data-intensive SP applications. Otherwise, the state checkpoint becomes bigger, and the system could run out of memory. Still, the author implemented a module to collect information. However, Spark also allows the use of a well-defined listener interface like *onBatchCompleted()*[12] for receiving information about an ongoing streaming computation. Finally, although it lets the processing under control, this solution ignores the incoming data and related use from memory at the Spark level, directly impacting processing performance.

Hanif, Muhammad et all. [11] presented a backpressure mitigation mechanism for in-memory data SP frameworks. This study reveals how Flink's backpressure is propagated in the opposite direction of downstream operators. It means backpressure is not fully aware of operator performance, and it may affect the performance due to memory management problems at the JVM level. Still, the upstreams may produce data faster than the downstream operators can consume, overloading JVM memory.

[11] State timeout: https://spark.apache.org/docs/2.0.0/api/java/org/apache/spark/streaming/State.html

[12] Spark listeners: https://spark.apache.org/docs/2.1.0/api/java/org/apache/spark/streaming/scheduler/StreamingListener.html

 doi:10.20944/preprints202205.0334.v1

In such a context, the proposed strategy focuses on stateful applications and aims to improve its performance by adjusting the level of parallelism of each operator on the fly. Thus, a feedback loop was made to identify whether the data ingestion is faster than the downstream operators can consume. This strategy uses a ratio-based algorithm that measures Central Processing Unit (CPU) utilization to set up a ratio value to establish the current sensibility of processing. The ratio varies from 0 to 1 and indicates the current system's condition. For instance, a value near zero does not represent a backpressure scenario, but a valuer near 1 indicates a backpressure one. Based on the ratio value, it is possible to increase or decrease the level of parallelism from operators on the fly in order to alleviate buffer overflow.

Research gap: although Flink has been taking advantage of several Application Programming Interface (API)s such as backpressure to help in task performance management. This work does not measure memory utilization from the operators, and it may lead to incorrect decisions. Still, as the authors described, the OMM should occur in extreme conditions without controlling incoming data, thus leading to a memory starvation problem. Unfortunately, the author provided a narrow evaluation that does not comprise a real-world case scenario and leads the system to a bad state. Finally, this work reinforces the need for a global controller to keep incoming data under control. At least, Flink and Spark rely upon JVM for execution overflow and manage OMM issues by spilling data to the disk, degrading performance.

De Souza. Paulo et all. [15] introduces BurstFlow, a tool for enhancing communication across data sources located at the edges of the Internet and big data SP applications located in cloud infrastructures. BurstFlow introduces a strategy for adjusting the micro-batch sizes dynamically according to the time required for communication and computation. Also, it presents an adaptive data partition policy for distributing incoming data across available machines by considering memory and CPU capacities. This approach leads to overcoming resource contention scenarios while maintaining network stability. Real-world experiments show an improvement of over 9% in the execution time, over 49% better CPU and memory utilization compared to methods applied to data partitioning in Apache Flink and state of the art.

Research gap: The up-streams components could produce data faster than the downstream operators can consume, thus overloading JVM memory. The author proposes a dynamic strategy to batch data that maximizes throughput and minimizes network latency over heterogeneous environments. However, the scheduling and the data imbalance problem, leaving JVM free to keep receiving data, even in intensive conditions.

As far as we can see, the goal of finding an efficient memory management solution became a key that allows high-performance processing in the most varied areas and domains. Unfortunately, SP systems hide the memory management scheme from users who do not have the opportunity to monitor and configure the memory resources properly. For instance, the control of the JVM or internal mechanism from frameworks like Spark; the current cache replacement policies based on the DAG like Spark LRU which do not consider the dynamic change in cache capacity needed by data-intensive SP applications, and so on. Both cases may lead data blocks to be evicted, producing significant recomputing overhead or data loss.

6. Conclusion

It is well known that the available memory of computing systems is constantly increasing, thus allowing in-memory data processing at a high scale. Moreover, in-memory data-intensive frameworks have been widely used to handle challenging problems in various domains such as machine learning, graph computing, and SP. Related applications benefit from in-memory operations.

The use of backpressure seems to guarantee a kind of application's stability for non-intensive scenarios, i.e., receiving data as fast as Spark can process in a single time window. Thus, the application could achieve stable processing at high throughput in controlled conditions. However, backpressure shifts the task of buffering incoming records to the sender until the stream application processes them. This approach can fail in high-intensive scenarios, as demonstrated in this work, due to a lack of management from the UMM. The PID algorithm only handles the execution region

 doi:10.20944/preprints202205.0334.v1

during processing, leading to memory faults like OOM issues due to high utilization from the storage region.

It occurs because UMM storage and execution share the same regions. Thus, while backpressure performs rate adjustments, the receivers will keep pushing data and filling memory up to the JVM heap memory limit, reducing the space for execution due to high storage needs. In such a context, a surge of data may generate high SD peaks that lead to a crash issue because they are considered unhealthy for processing since the system is not stable. It means the processing could not be made in a single time window since plenty of data is waiting in the cache to be processed later, letting the JVM heap memory on pressure at the executor side. Furthermore, when the UMM needs to evict data from memory, the execution has priority over the storage region, incurring data loss in intensive processing scenarios.

The time to evict data blocks is relative and really depends on cluster capacity and processing performance. Due to data overload and memory contention, the system may become unhealthy case executors stop sending periodic processing information. Still, several borrowing operations at the UMM level may lead to multiple full GC operations to clean up RDD objects from JVM heap memory. Still, the cleaning operations are critical for processing because some past states may be needed for the current processing steps. Thus, Spark will fail if a block is already evicted from the executors' cache.

The related works do not present agnostic solutions for the memory management problem. The solutions relied upon small and medium-sized changes in the core of SP frameworks, such as a new memory manager, a new eviction policy, or improvements based on batching or GC counters.

Author Contributions: Conceptualization, K.J.M and C.F.R.G; methodology, K.J.M, J.C.S.A, C.F.R.G; software; K.J.M; validation, K.J.M and J.C.S.A; supervision C.F.R.G; writing–original draft preparation, K.J.M, J.C.S.A; and V.R.Q.L; writing–review and editing, K.J.M, V.R.Q.L and J.C.S.A

Funding: This work has partially supported by PROPESQ-UFRGS-Brasil and Junta De Castilla y León-Consejería de Economía Y Empleo: System for simulation and training in advanced techniques for the occupational risk prevention through the design of hybrid-reality environments with ref. J118.

Institutional Review Board Statement: Not applicable.

Informed Consent Statement: Not applicable.

Data Availability Statement: Not applicable.

Acknowledgments: This work was supported by national funds through the "SmartSent" (#17/2551-0001 195-3), CAPES (Finance Code 001), CNPq, PROPESQ-UFRGS-Brasi, program PRONEX 122014. Also, CEREIA Project (#2020/09706-7), PRINT- FAPESP – MCTIC - CGI.BR, and FAPERGS Project "GREEN-CLOUD - Computação em Cloud com Computação Sustentável" (#16/2551-0000 488-9). Fundação para a Ciência e a Tecnologia, I.P. (Portuguese Foundation for Science and Technology) by the project UIDB/05064/2020 (VALORIZA–Research Centre for Endogenous Resource Valorization), and Project UIDB/04111/2020, ILIND–Instituto Lusófono de Investigação e Desenvolvimento, under project COFAC/ILIND/ COPELABS/3/2020.

Conflicts of Interest: The authors declare no conflict of interest.

References

1. Hassanien, A.E.; Darwish, A. "Machine Learning and Big Data Analytics Paradigms: Analysis, Applications and Challenges". *Springer Nature* **2020**, *77*.
2. Avgeris, M.; Spatharakis, D.; Dechouniotis, D.; Leivadeas, A.; Karyotis, V.; Papavassiliou, S. ENERDGE: Distributed Energy-Aware Resource Allocation at the Edge. *Sensors* **2022**, *22*. https://doi.org/https://doi.org/10.3390/s22020660.
3. Tang, Z.; Zeng, A.; Zhang, X.; Yang, L.; Li, K. "Dynamic Memory-Aware Scheduling in Spark Computing Environment". Journal of Parallel and Distributed Computing **2020**, *141*, 10 – 22. https://doi.org/https://doi.org/10.1016/j.jpdc.2020.03.010.
4. da Silva Veith, A.; Dias de Assuncao, M.; Lefevre, L. Latency-Aware Strategies for Deploying Data Stream Processing Applications on Large Cloud-Edge Infrastructure. *IEEE Transactions on Cloud Computing* **2021**, pp. 1–1. https://doi.org/10.1109/TCC.2021.3097879.
5. Toshniwal, A.; Taneja, S.; Shukla, A.; Ramasamy, K.; Patel, J.M.; Kulkarni, S.; Jackson, J.; Gade, K.; Fu, M.; Donham, J.; et al. "Storm@twitter". In Proceedings of the Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD; ACM: New York, NY, USA, 2014; p. 147–156. https://doi.org/10.1145/2588555.2595641.

6. Noghabi, S.A.; Paramasivam, K.; Pan, Y.; Ramesh, N.; Bringhurst, J.; Gupta, I.; Campbell, R.H. "Samza: Stateful Scalable Stream Processing at LinkedIn". Journal of Very Large Data Base Endowment. **2017**, *10*, 1634–1645. https://doi.org/10.14778/3137765.3137770.
7. Zaharia, M.; Chowdhury, M.; Franklin, M.J.; Shenker, S.; Stoica, I. "Spark: Cluster Computing With Working Sets". In Proceedings of the Proceedings of the 2nd USENIX Workshop on Hot Topics in Cloud Computing (HotCloud 10); USENIX Association: Berkeley, CA, USA, 2010; pp. 1–7.
8. Carbone, P.; Katsifodimos, A.; Ewen, S.; Markl, V.; Haridi, S.; Tzoumas, K. "Apache Flink: Stream and Batch Processing In A Single Engine". Bulletin of the IEEE Computer Society Technical Committee on Data Engineering **2015**, *36*, 4. https://doi.org/XX.
9. Amazon Web Services, Inc.. "Collect Streaming Data, at Scale, for Real-Time Analytics, 2021. Available at: https://aws.amazon.com/kinesis/data-streams/.
10. Xu, L.; Li, M.; Zhang, L.; Butt, A.R.; Wang, Y.; Hu, Z.Z. "MEMTUNE: Dynamic Memory Management for In-Memory Data Analytic Platforms". In Proceedings of the Proceedings of the IEEE International Parallel and Distributed Processing Symposium (IPDPS); IEEE Computer Society: Chicago, IL, USA, 2016; pp. 383–392. https://doi.org/10.1109/IPDPS.2016.105.
11. Hanif, M.; Yoon, H.; Lee, C. "A Backpressure Mitigation Scheme in Distributed Stream Processing Engines". In Proceedings of the Proceedings of the 2020 International Conference on Information Networking (ICOIN); IEEE Computer Society: XX, 2020; pp. 713–716. https://doi.org/10.1109/ICOIN48656.2020.9016513.
12. Zhao, Z.; Zhang, H.; Geng, X.; Ma, H. "Resource-Aware Cache Management for In-Memory Data Analytics Frameworks". In Proceedings of the Proceedings of the IEEE Intl Conf on Parallel Distributed Processing with Applications, Big Data Cloud Computing, Sustainable Computing Communications, Social Computing Networking (ISPA/BDCloud/SocialCom/SustainCom). IEEE Computer Society, 2019, pp. 364–371. https://doi.org/10.1109/ISPA-BDCloud-SustainCom-SocialCom48970.2019.00060.
13. Jia, D.; Bhimani, J.; Nguyen, S.N.; Sheng, B.; Mi, N. "Atumm: Auto-Tuning Memory Manager in Apache Spark". In Proceedings of the Proceedings of the IEEE International Conference on Performance, Computing and Communications (IPCCC). IEEE Computer Society, 2019, pp. 1–8. https://doi.org/10.1109/IPCCC47392.2019.8958724.
14. Matteussi, K.J.; Zanchetta, B.F.; Bertoncello, G.; Dos Santos, J.D.; Dos Anjos, J.C.; Geyer, C.F. "Analysis and Performance Evaluation of Deep Learning on Big Data". In Proceedings of the Proceedings of the IEEE Symposium on Computers and Communications (ISCC). IEEE Computer Society, 2019, pp. 1–6. https://doi.org/10.1109/ISCC47284.2019.8969762.
15. De Souza, P.R.R.; Matteussi, K.J.; Veith, A.D.S.; Zanchetta, B.F.; Leithardt, V.R.; Murciego, A.L.; De Freitas, E.P.; Dos Anjos, J.C.; Geyer, C.F. "Boosting Big Data Streaming Applications in Clouds With BurstFlow". IEEE Access **2020**, *8*, 219124–219136. https://doi.org/10.1109/ACCESS.2020.3042739.
16. Matteussi, K.J.; Geyer, C.F.R.; Xavier, M.G.; De Rose, C.A. "Understanding and Minimizing Disk Contention Effects for Data-Intensive Processing in Virtualized Systems". In Proceedings of the Proceedings of International Conference on High Performance Computing Simulation (HPCS). IEEE Computer Society, 2018, pp. 901–908. https://doi.org/10.1109/HPCS.2018.00144.
17. Dos Anjos, J.C.; Matteussi, K.J.; De Souza, P.R.; Grabher, G.J.; Borges, G.A.; Barbosa, J.L.; Gonzalez, G.V.; Leithardt, V.R.; Geyer, C.F. "Data Processing Model to Perform Big Data Analytics in Hybrid Infrastructures". IEEE Access **2020**, *8*, 170281 – 170294. https://doi.org/10.1109/10.1109/ACCESS.2020.3023344.
18. Dos Anjos, J.C.S.; Gross, J.L.G.; Matteussi, K.J.; González, G.V.; Leithardt, V.R.Q.; Geyer, C.F.R. An Algorithm to Minimize Energy Consumption and Elapsed Time for IoT Workloads in a Hybrid Architecture. *Sensors* **2021**, *21*. https://doi.org/10.3390/s21092914.
19. Pereira Fábio, C.P.; R.Q., L.V. PADRES: Tool for PrivAcy, Data REgulation and Security. *SoftwareX* **2022**, *17*, 100895. https://doi.org/https://doi.org/10.1016/j.softx.2021.100895.
20. Ziegler, J.G.; Nichols, N.B.; et al. "Optimum Settings for Automatic Controllers". Journal of Trans. ASME **1942**, *64*, 759–765. https://doi.org/https://doi.org/10.1115/1.2899060.
21. Startin, R. "Tuning Spark Back Pressure by Simulation", 2020. Available at: https://richardstartin.github.io/posts/tuning-spark-back-pressure-by-simulation.
22. Zaharia, M.; Das, T.; Li, H.; Hunter, T.; Shenker, S.; Stoica, I. "Discretized Streams: Fault-Tolerant Streaming Computation at Scale". In Proceedings of the Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles; Association for Computing Machinery: New York, NY, USA, 2013; p. 423–438. https://doi.org/10.1145/2517349.2522737.
23. Dessokey, M.; Saif, S.M.; Salem, S.; Saad, E.; Eldeeb, H. "Memory Management Approaches in Apache Spark: A Review". In Proceedings of the Proceedings of the International Conference on Advanced Intelligent Systems and Informatics 2020; Springer International Publishing: Cham, 2021; pp. 394–403. https://doi.org/10.1007/978-3-030-58669-0_36.
24. Or, A.; Rosen, J. "Unified Memory Management in Spark 1.6", 2021. Available at: https://www.linuxprobe.com/wp-content/uploads/2017/04/unified-memory-management-spark-10000.pdf.
25. Daoyuan, W.; Huang, J. "Tuning Java Garbage Collection for Apache Spark Applications", 2015. Available at: https://databricks.com/blog/2015/05/28/tuning-java-garbage-collection-for-spark-applications.html.
26. Spark™, A. "Hardware Provisioning", 2021. Available at: https://spark.apache.org/docs/3.0.0/hardware-provisioning.html.
27. Spark™, A. "Memory Management Overview", 2021. Available at: https://spark.apache.org/docs/latest/tuning.html#memory-management-overview.
28. Guide, M.T. "0MQ - The Guide", 2020. Available at: http://zguide.zeromq.org/php:chapter2.
29. Data, F.T. "Apache Spark DStream", 2021. Available at: https://data-flair.training/blogs/spark-tutorial/.

30. Das, T.; Zhong, Y.; Stoica, I.; Shenker, S. "Adaptive Stream Processing Using Dynamic Batch Sizing". In Proceedings of the Proceedings of the ACM Symposium on Cloud Computing; Association for Computing Machinery: New York, NY, USA, 2014; p. 1–13. https://doi.org/10.1145/2670979.2670995.
31. Birke, R.; Björkqvist, M.; Kalyvianaki, E.; Chen, L.Y. "Meeting Latency Target in Transient Burst: A Case on Spark Streaming". In Proceedings of the Proceedings of the 2017 IEEE International Conference on Cloud Engineering (IC2E); IEEE Computer Society: XX, 2017; pp. 149–158. https://doi.org/10.1109/IC2E.2017.17.
32. Chen, X.; Vigfusson, Y.; Blough, D.M.; Zheng, F.; Wu, K.L.; Hu, L. GOVERNOR: Smoother Stream Processing Through Smarter Backpressure. In Proceedings of the Proceedings of the IEEE International Conference on Autonomic Computing (ICAC). IEEE Computer Society, 2017, pp. 145–154. https://doi.org/10.1109/ICAC.2017.31.