IEEE Access
Multidisciplinary ⋮ Rapid Review ⋮ Open Access Journal

Received June 30, 2021, accepted August 1, 2021, date of publication August 5, 2021, date of current version August 11, 2021.

Digital Object Identifier 10.1109/ACCESS.2021.3102645

Influencing Factors in the Scalability of Distributed Stream Processing Jobs

GISELLE VAN DONGEN, (Member, IEEE), AND DIRK VAN DEN POEL, (Senior Member, IEEE)
Department of MIO/Data Analytics, Ghent University, 9000 Ghent, Belgium

Corresponding author: Giselle Van Dongen (giselle.vandongen@ugent.be)

ABSTRACT More and more use cases require fast, accurate, and reliable processing of large volumes of data. To do this, a distributed stream processing framework is needed which can distribute the load over several machines. In this work, we study and benchmark the scalability of stream processing jobs in four popular frameworks: Flink, Kafka Streams, Spark Streaming, and Structured Streaming. Besides that, we determine the factors that influence the performance and efficiency of scaling processing jobs with distinct characteristics. We evaluate horizontal, as well as vertical scalability. Our results show how the scaling efficiency is impacted by many factors including the initial cluster layout and direction of scaling, the pipeline design, the framework design, resource allocation, and data characteristics. Finally, we give some recommendations on how practitioners should undertake to scale their clusters.

INDEX TERMS Apache spark, structured streaming, apache flink, apache kafka, kafka streams, distributed computing, stream processing frameworks, scalability, benchmarking, big data.

I. INTRODUCTION

Near real-time processing has become increasingly important with the rise of new domains such as IoT. The use cases in these domains often require processing large volumes of data at a high velocity. To do this, a single machine is not sufficient or becomes increasingly expensive. Consequently, it becomes necessary to distribute processing over several machines. This heavily increases the complexity of an application in many aspects, e.g. fault recovery [1], accuracy [2], state management [3]. To make abstraction of this complexity, several stream processing frameworks were developed. These frameworks can be used as a generic system to implement a large range of use cases in a distributed fashion. The ability of a system to spread work across several machines is called scalability [4]. A system is considered scalable when an increase in resources leads to a (near) linear increase in throughput. Scalability is often mentioned in relation to networks, systems, or processes [5]. It is also used to describe whether a distributed system can be deployed efficiently in a wide range of scales [6]. A system can be scaled either horizontally or vertically [7]. Horizontal scaling implies an increase in the number of workers, while vertical scaling implies an increase in the resources assigned to the workers.

The associate editor coordinating the review of this manuscript and approving it for publication was Asad Waqar Malik.

It is important to keep in mind that distributed processing introduces significant overheads that vary based on the use case and implementation, as shown by McSherry *et al.* [8]. These overheads are mainly due to increased scheduling, communication, and bookkeeping. McSherry *et al.* [8] computed the Configuration that Outperforms a Single Thread (COST) metric to a set of use cases and found that the use of a single thread often leads to significant advantages.

Multiple benchmarks have studied performance differences between stream processing frameworks, e.g. [9]–[14]. Most of these benchmarks focus on latency and throughput metrics and do not incorporate a scalability analysis. The papers that study scalability often only include horizontal scaling and assume the framework is the main influencing factor of scalability. In contrast, previous work within the general field of distributed systems attributed a large part of scalability issues to the resource bottleneck of the system [5], [6], [15]. In a distributed stream processing framework, the resource bottleneck is the resource that is saturated when the peak throughput is reached and is either related to memory, CPU, or network. We define a throughput bottleneck as a task, aspect, or characteristic of the pipeline which contributes largely to the resource bottleneck. There may be multiple significant contributors. Some common throughput bottlenecks are incessant garbage collection (GC) cycles

This work is licensed under a Creative Commons Attribution 4.0 License. For more information, see https://creativecommons.org/licenses/by/4.0/

due to memory pressure, CPU-intensive state management, network saturation, etc. Scaling alleviates some bottlenecks better than others. In this paper, we evaluate the hypothesis that the scalability of a processing job is influenced by many factors that together create the throughput bottleneck. To do this, we designed two pipelines with different characteristics regarding e.g. state size, output frequency, shuffle operations, and window length. Due to these differences, we expect them to stress different parts of the system and have different bottlenecks. Both pipelines are described in detail in Section III-B. For each pipeline, we determine the throughput bottlenecks for four popular frameworks and analyze the influence scaling has on them.

Most previous benchmarks studied one or a combination of the following frameworks: Spark Streaming, Storm, Flink, Kafka Streams and Structured Streaming [9]–[14], [16]–[20]. Additionally, these frameworks are often included in surveys on distributed stream processing (e.g. [21]–[23]) and are used as a basis for further research in the area (e.g. [24]–[27]). In this work, we include Spark Streaming and Flink due to their superior performance in previous benchmarks (e.g. [12]–[14]). We also include Kafka Streams and Structured Streaming because they are more recent and their performance has not been thoroughly analyzed yet. Moreover, both of them have a rapidly increasing user base. Apache Flink [28] has gained widespread adoption in many large companies and is considered to be one of the front runners in stream processing at the moment. Apache Spark is a popular distributed analytics framework, primarily used for its batch processing capabilities. It has two APIs for stream processing: Spark Streaming and Structured Streaming. When Spark Streaming was released, it quickly gained traction and became a popular alternative for frameworks such as Storm [29] and Samza [30], mainly for use cases where a slightly higher latency is acceptable. In 2016, Apache Spark announced a more high-level stream processing API called Structured Streaming, which integrates more tightly with its batch API. Finally, we include Kafka Streams [31], which targets processing data residing on Kafka [32] clusters. Apache Kafka is a widely used, distributed message broker [33]. Kafka Streams integrates closely with Kafka and uses the cluster for communication, fault tolerance, and parallelization. Kafka Streams is used by companies such as Zalando and Pinterest and is backed by Confluent.

The two processing pipelines, that are included, have very different characteristics and underlying algorithms and, therefore, stress different parts of the system. We give an in-depth analysis of this throughout this work.

In general, the contributions of this paper are:

1) Extensive analysis of the scalability of distributed stream processing jobs.
2) Discussion on throughput bottlenecks based on a wide variety of metrics.
3) Analysis of how scalability is influenced by the throughput bottleneck, which in turn is influenced by multiple factors such as scaling direction, framework design and processing pipeline operations.
4) Analysis of vertical and horizontal scalability for distinct jobs and cluster layouts for the frameworks: Flink, Kafka Streams, Spark Streaming, and Structured Streaming.
5) Guidelines on scaling clusters and optimal cluster layouts for different job complexities.

The rest of this paper is organized as follows. The next section gives an overview of related work in this area. Section III describes the setup and metrics we use to do performance evaluations. Here, we also describe our methodology for measuring scalability. We base ourselves on OSPBench of which the codebase and documentation have been published at https://github.com/Klarrio/open-stream-processing-benchmark. We give an overview of the results in Section IV. In Section V, we outline current and future research directions with regards to scaling and cluster sizes. Finally, we draw conclusions on these results in Section VI and list up some limitations in Section VII.

II. RELATED WORK

Research on the scalability of stream processing jobs and frameworks is very limited. Table 1 shows an overview of this. Previous work investigated horizontal scalability, i.e. increasing the number of workers. Karimov *et al.* [10] compared the performance of clusters with 2, 4, and 8 workers, while Karakaya *et al.* [11] investigated performance for 1, 2, 4, and 6 workers. We take a similar approach for our horizontal scaling experiments. A more recent work [19] takes a different approach and determines the number of processing instances required to process a specific load. They gradually add instances to see if it covers the load. Each instance has 1 CPU and 4 GB memory which is small for industry deployments. Incrementally adding instances leads to large jumps in the available resources. Instead of measuring the required cluster size for a certain throughput, we argue that measuring the peak throughput for a certain cluster size gives a more precise view of the evolution of throughput for specific cluster sizes. In their work, Henning and Hasselbring [19] also investigated the effect of vertical scaling. Similar to them, we also evaluate both scaling directions. Additionally, we adapt the instance configuration along with the instance size to ensure optimal use of resources and to get a complete view of the scalability, as we will elaborate in our results.

All of these works [10], [11], [19] concentrate on describing the throughput levels that can be sustained at different cluster sizes for each framework. We also measure the throughput level for different cluster sizes but evaluate the results from the perspective of the job characteristics with the selected framework as one of these. This gives readers useful information on the scalability of their proper use case, which is what matters most. To do this, we compare scalability for two pipelines with distinct characteristics (see Section III-B). Our pipelines resemble the pipelines of previous

TABLE 1. Scalability workloads in previous benchmarking work.

Reference	Workload	Frameworks	Operations	Metrics
Karimov et al.,2018 [10]	horizontal scalability (2, 4, and 8 workers)	Spark Streaming 2.0.1, Flink 1.1.3, Storm 1.0.2	Windowed aggregation, join, large windows	Event and processing time latency, sustainable throughput
Karakaya et al., 2017 [11]	horizontal scalability (1, 2, 4, and 6 workers)	Spark Streaming 1.5.0, Flink 0.10.1, Storm 0.10.0	End-to-end pipeline (parse - filter - project - join - window)	Saturation level, scale-up ratio, resource consumption
Henning et al., 2020 [19]	horizontal scalability (up to 100 workers), vertical scalability (0.5, 1, and 2 CPU)	Kafka Streams 2.6.0	map + foreach, join + aggregation with feedback loop, tumbling window, aggregation with sliding window	Queue size
This study	**horizontal scalability (1, 2, 4, and 6 workers), vertical scalability (1, 2, 4, and 6 CPU; 5GB memory per CPU)**	**Flink 1.11.1, Kafka Streams 2.6.0 Structured Streaming, Spark Streaming 3.0.0**	**stateful enrichment pipeline, stateful aggregation pipeline**	**Scaling efficiency, latency, throughput, memory, CPU, GC, network I/O, filesystem and disk I/O**

work [9], [12]. Similar to Karimov *et al.* [10], we analyze different window lengths and window aggregations. The enrichment pipeline, which we present in Section III-B, has a similar sequence of operations as [11].

Different metrics were used to evaluate scalability. Both [10] and [19] report sustainable throughput of different cluster sizes. Karakaya *et al.* [11] express scalability by the scale-up ratio which is the percentage increase in throughput of a specific cluster size compared to a cluster with one worker. We define a scaling efficiency metric that expresses the percentage of expected throughput increase that was realized as compared to a base cluster. Previous benchmarks on other aspects of stream processing [9], [10], [12], [17], [18] included a much broader set of metrics. These extra metrics were used to gain a better understanding of the performance and behavior of the framework. The most popular ones are latency, throughput, and CPU and memory utilization. We also include these metrics in our work to do a more in-depth analysis.

Previous work [10] benchmarked Storm 1.0.2, Spark Streaming 2.0.1 and Flink 1.1.3. They found that for larger cluster sizes, Flink's sustainable throughput was bounded by the network bandwidth. The sustainable throughput of Storm and Spark was comparable and did not increase linearly with the number of nodes. The throughput of Spark increased by 70% when going from 2 to 4 nodes and increased by 140% when going from 2 nodes to 8 nodes for the windowed aggregation. Henning and Hasselbring [19] only evaluated Kafka Streams and found that configuration has a large effect on performance. They only tune a few parameters and do not adapt the configuration to the cluster scale and layout. Similar to [9] and [20], we include recent releases of four frameworks: Flink 1.11, Kafka Streams 2.6 and Spark Streaming and Structured Streaming 3.0.

Several studies and modeling exercises have been conducted on the scalability of batch analytics workloads, e.g. [34]–[41]. Since batch and streaming workloads function quite differently, we will discuss the relevant parts of these studies throughout the results section.

III. BENCHMARK DESIGN

To do scalability tests, we need a benchmarking setup with a set of capabilities. First of all, the system needs to be able to automatically run several processing pipelines for a set of frameworks. These pipelines should stress different parts of the system, e.g. CPU, memory, network. On top of that, the system needs to run different throughput levels and cluster sizes in an automated manner. The relevant configuration parameters should change based on the cluster size. Additionally, the setup should collect a broad range of metrics that can be analyzed to generate insights. Ideally, there would exist a single benchmarking suite that incorporates workloads to test several aspects of stream processing frameworks: latency, throughput, fault tolerance, scalability, etc. Currently, no such suite is available. To work to this goal, we prefer extending an existing benchmark over implementing a new suite from scratch. We choose to extend Open Stream Processing Benchmark (OSPBench) [9], [20]. It offers recent implementations of several pipelines in the most popular frameworks. It already includes extensive latency and throughput workloads [9] and fault tolerance workloads [20], but lacks scalability workloads. By extending OSPBench with additional pipelines, metrics, and scalability workloads, we are one step closer to having a single comprehensive suite for testing frameworks on the most important aspects of stream processing. In this section, we explain how we used OSPBench to do scalability experiments and how we extended it to cater to our use case.

A. DATA

The pipelines within OSPBench analyze national traffic data of the Netherlands, i.e. Nationaal Dataportaal Wegverkeer (NDW) [42]. We have two streams: one with speed measurements and one with vehicle counts from a set of measurement points in the Netherlands. The messages in both streams are in JSON format and contain approximately 10 fields. Each record has a measurement ID and lane ID, specifying the measurement point and road lane. These are the fields on which joins and aggregations will be done. The size of one message is around 200 bytes. Each stream is sent to

a dedicated Kafka topic. The throughput of both streams is equal and configurable in the data generator and data is sent at a constant rate, without bursts. The number of measurement IDs increases linearly with the throughput. The number of lanes per measurement ID remains the same as throughput is increased and there are on average 1.58 lanes per measurement ID.

B. PROCESSING PIPELINE

To do our scalability tests, we needed two pipelines with very different characteristics that might influence scalability. Most of these characteristics are related to state and shuffling. The first pipeline has been depicted in Figure 1. We refer to this pipeline as the enrichment pipeline because it contains a join operation. This pipeline was already included in previous versions of OSPBench [9]. The second pipeline is shown in Figure 2. We call this the aggregation pipeline.

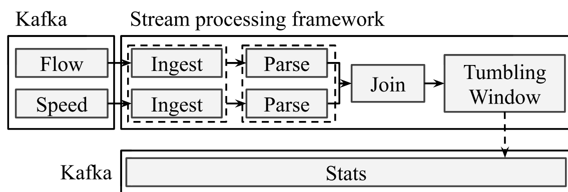

FIGURE 1. Enrichment pipeline adapted from [9] and [43].

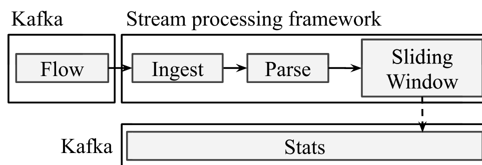

FIGURE 2. Aggregation pipeline.

The enrichment pipeline ingests two streams: one containing speed measurements and the other containing vehicle counts at a configurable number of measurement locations. The data is in JSON format and subsequently parsed into Scala case classes. Both streams are then joined together at one-second intervals. Similar to [9], we do this with interval joins for Flink, Kafka Streams, and Structured Streaming. Spark Streaming does not support this, so here we use a tumbling window join. For its join and aggregation windows, Spark uses a multiple of the batch interval. Flink and Kafka Streams offer more efficient event-driven implementations. Structured Streaming allows defining an interval join, although this will still be executed on a microbatch basis. At this point, the joined stream contains events for every lane of every measurement location. In the next step, we aggregate all lanes of a similar measurement location and compute the average speed and total count. The aggregation is done with an incremental reduce function over a window of one second.

The characteristics of the enrichment pipeline are listed in Table 2. The pipeline does a join and tumbling window operation. The joining key is different from the aggregation key leading to shuffling in between the stages. These frameworks apply shuffling for every key change and grouping operation (e.g. join, window) to make sure that data of the same key or group is present on the same worker. Shuffling leads to a higher network load and higher CPU utilization from serialization and deserialization. Both keys contain the timestamp rounded down to second-level. This leads to many different keys and a quickly changing keyspace. The windows in this pipeline are very short (one second) and trigger output frequently. This affects the behavior of the jobs, as will be explained in the Results Section. The tumbling window does its computations incrementally via a reduce function. Hence, the pipeline does not need to hold state for long periods and has a very small base state. This makes the pipeline lighter on memory requirements. We expect this pipeline to be mainly CPU- and network-intensive due to shuffling and the management of the windows and state.

TABLE 2. Processing pipeline differences.

Characteristic	Enrichment pipeline	Aggregation pipeline
Input streams	2	1
Stateful operations	2	1
State	small	large
Shuffles	2	1
Window length	short	long
Keyspace	quickly changing	static
Key-value ratio	few values per key	many values per key
Output trigger	frequent	infrequent

The second pipeline is the aggregation pipeline (Figure 2). It ingests only one stream of data. The data is also in JSON format. The stream is parsed into Scala case classes. Afterward, it computes a sliding window aggregation with a slide duration of one minute and a window duration of five minutes. This operation aggregates all the lanes of a measurement ID over the entire window duration and computes the average speed and accumulated count of the cars that passed by. It only does a single stateful operation, leading to less shuffling than for the enrichment pipeline. The aggregation can be done incrementally with a reduce function. However, we take a non-incremental approach to stress the state management of the system. We keep all events as state and trigger computation when the window times out. The state accumulates to a few gigabytes for the larger throughput levels. A larger state makes state management heavier. It increases memory requirements and takes longer to checkpoint and to perform garbage collection. The key of the sliding window is the measurement ID. The keyspace does not change since the measurement IDs are a static set of values. We have fewer keys for this pipeline than for the enrichment pipeline and each key has many more values to be kept in state. Output is generated every minute and is, therefore, infrequent compared to the enrichment pipeline. This affects the behavior of the jobs, as explained in the Results Section. We expect this

pipeline to be heavy on memory and CPU utilization due to state management and GC.

C. INFRASTRUCTURE

To make our experiments realistic and reproducible, we run on cloud infrastructure with state-of-the-art technologies. We use AWS EC2 to provide underlying computing power. We allocate six to nine m5n.4xlarge instances. The number of instances grows with the framework cluster size that we are testing. Each instance has 16 vCPUs, 64 GiB memory, and a network bandwidth of up to 25 Gbps. Each instance has an EBS volume of 500 GB with a bandwidth of up to 4750 Mbps.

On top of the EC2 instances, we deploy a DC/OS [44] cluster. DC/OS provides an abstraction layer on top of the EC2 instances and facilitates the deployment of benchmark components as Docker [45] containers. First of all, we deploy a Kafka [33] cluster as a message broker and intermediary between data producers and consumers. We collect metrics of Docker containers with cAdvisor [46] and via JMX. HDFS [47] is used to store the state of processing jobs. When we do a benchmark run, we deploy a framework cluster and a data stream generator that publishes one or two streams of data onto Kafka. When the run has finished, we use a separate component to read the output data from Kafka and save and calculate metrics.

To make sure that the runs are not influenced by an overloaded DC/OS cluster, we only allocate up to 90% of the CPUs and memory of each slave. We also closely monitor the EC2 instances and core components of the benchmark (e.g. Kafka broker, HDFS) to ensure that they are not the bottleneck. Additionally, we execute each benchmark run three times to check for consistent, reproducible results. We relaunch the DC/OS cluster in between runs to exclude the influence of the bad-performing EC2 instances on the performance.

D. METRICS

OSPBench originally only monitored four metrics: latency, throughput, total CPU utilization, and heap memory utilization. To do our bottleneck analysis, we heavily increased the number and depth of metrics that are gathered. OSPBench now scrapes detailed metrics on CPU and memory utilization, garbage collection (GC), network utilization, and filesystem and disk I/O. In this section, we describe each of the metrics and how we collect and compute them.

1) LATENCY

The first metric is latency. Latency is the time required to generate an output message from an input message. In this study, we use latency as an indicator of delays in processing and sustainability of throughput. Latency is computed by subtracting the Kafka message timestamps of the input and output events. When multiple events are required to do a computation, we count latency starting from the last input event. Because the clocks of the different machines of the Kafka brokers are not synchronized, we need to make sure that the input and output events end up on the same brokers, as per [43]. We do this by using the same partitioning key for the input and output.

2) THROUGHPUT

The second metric is throughput. The output throughput is the number of messages per second that the framework emits at its sink. The peak sustainable throughput [9] is defined as the maximum throughput that can be sustained for a prolonged period without leading to instability of the framework. A system is considered to be stable when three conditions are met: (1) the queues on the input buffer and latency are not continuously increasing, (2) the average CPU utilization remains lower than 80% to keep a buffer for maintaining running processes and (3) the median latency remains below 10 seconds for the enrichment pipeline, and below 60 seconds for the aggregation pipeline. A system is considered to be instable when any of the three conditions are not met. The latency requirement is mainly important for Structured Streaming, since it uses a dynamic batch interval for the enrichment pipeline. As throughput increases, the batches become larger and hence, we require latency to remain below 10 seconds to still consider it as stream processing. For the aggregation pipeline, the batch interval of Spark Streaming and Structured Streaming is set at 60 seconds and therefore, we require the latency to remain below 60 seconds as well.

3) CPU UTILIZATION

We use the metric CPU utilization to determine the load on the framework. It has a linear relationship with throughput. Constantly elevated CPU utilization can lead to job instability. It is acceptable for CPU levels to occasionally reach 100% as long as this does not persist for longer periods. We collect CPU metrics of each running container by using cAdvisor [46]. CPU cycles can be spent on a wide variety of tasks such as fetching data, (de)serialization, data copying, data management operations within the JVM or kernel, etc. Therefore, we track not only the total CPU usage but also whether it was spent in user or system mode. Previous work in batch analytics has shown that high total CPU utilization does not always mean that compute is the bottleneck of the system. The amount of time waiting on disk I/O requests can form an important part of CPU utilization [48], [49]. We approximate the I/O wait time by subtracting the system and user CPU utilization from the total CPU utilization. In our visualizations, we report the total CPU usage.

4) MEMORY UTILIZATION

The second resource metric is memory utilization. Memory utilization is subject to the configuration settings of the framework. Memory pressure can induce heavy GC cycles which put load on the CPUs. Workers fail when GC cycles become ineffective and memory grows out of bound. We use memory utilization to check whether it is not monotonously rising and to check whether garbage collection has the desired effect. We use JMX to collect metrics on the heap and

off-heap memory utilization and the size of each of the memory pools: G1 Eden, G1 Survivor, and G1 Old Generation. With cAdvisor, we collect extra metrics on RSS memory, swap memory, cache memory, page faults, and major page faults because they can give us confirmation on whether memory is a bottleneck of the process. In the figures included in this paper, we report heap memory. We report on the other metrics when informative.

5) GARBAGE COLLECTION

All frameworks make use of JVMs and therefore rely on GC algorithms to clean up heap memory. We use the G1 garbage collector for all frameworks, as explained in Section III-E. Some stages in the collection process require threads to stop processing which has a high performance cost. We use JMX to scrape metrics on the different phases of the GC cycle. More specifically, we monitor (1) the collection time and collection counts of minor and major collections and (2) the duration and change in used memory of the last G1 Young Generation and G1 Old Generation collections.

6) NETWORK UTILIZATION

Previous work on batch analytics does not agree on the influence of network bandwidth on performance. Ousterhout *et al.* [50] state that network I/O does not have a large effect on performance, even with 1 Gbps of network bandwidth. On the contrary, when Trivedi *et al.* [35] improved the network of Spark and Flink workers from 1 to 10 Gbps, they saw query response times speed up by a factor of two and more. When improving the network further to 40 Gbps they saw marginal improvements in performance because the CPU had become the new bottleneck. In the field of stream processing, network bandwidth has often proved to be a bottleneck for frameworks such as Flink. Therefore, we use instances that are designed for network-intensive applications and provide up to 25 Gbps. In our runs, we ensure that we do not use more than 4-5 Gbps to prevent network bottlenecks. Additionally, we monitor the bytes that are transmitted and received to identify irregularities and compare shuffling behavior among frameworks. Also, we track whether packages are dropped since this is an important indicator of network congestion. On the level of the EC2 instances, we monitor the CloudWatch metrics for network in and network out. We do not report these metrics throughout the paper because they are only used to check whether or not network bandwidth was becoming a bottleneck for the EC2 instance. When network bandwidth was a bottleneck, we increased the number of EC2 instances.

7) FILESYSTEM AND DISK I/O

As mentioned before, I/O operations can have a significant performance impact. We monitor filesystem usage and the serviced bytes while reading and writing to disk. During normal operations, frameworks only use filesystem or disk if they use a RocksDB state backend [51]. This is only the case for Kafka Streams. For other frameworks, the use of filesystem and disk can indicate that there is memory pressure and that the framework is swapping data between memory and disk. This leads to performance degradation because disk access is slow.

E. FRAMEWORKS

We implement pipelines in four frameworks in this paper. Using the right configuration settings for each job and adapting the configuration for each cluster size is paramount for reliable, generalizable results, as was pointed out by previous work [9], [13], [19]. In this section, we shortly describe each of the frameworks and their most important configuration parameters. The parameters in this work are tuned based on previous experiments with OSPBench [9], [20], the recommendations in the framework documentation, expert advice, and extensive experimentation. The final set of parameters follow standard practices and had optimal performance in our tuning experiments, as we will explain.

We do runs with different cluster sizes to test scalability. To use all the resources allocated to a cluster, some configuration parameters need to scale along with the cluster size, as listed in Table 3. These settings scale linearly with the number of cores per worker. Many of these settings appear in every framework, e.g. instance counts, and CPU, memory, and disk allocation. For vertical scaling, we run the worker containers with a higher allocation for CPU and memory in the Marathon definition. Additionally, we adapt the configuration of the framework to ensure that it uses the additional resources. With these two prerequisites, the frameworks manage the scheduling of tasks across all cores out-of-the-box. The framework configuration that is adapted for vertical scaling is described in Table 3. For Spark frameworks, we increase the setting for the number of worker cores and executor cores. We set the parallelism of the processing jobs equal to the number of cores in the cluster. The parallelism per worker is equal to the number of task slots for Flink and the number of threads for Kafka Streams. This is the recommended setting for pipelines with no data skew [52]. We also adapt the memory settings of each worker, as listed in Table 3. For horizontal scaling, the individual worker settings remain the same but we adapt the total parallelism of the job. For Spark frameworks, we also need to request a higher number of executors when submitting the job.

Another group of parameters is set based on the use case. They do not influence whether or not the added workers are used or not. Therefore, we keep them equal across all cluster sizes. This also ensures that performance differences can be attributed to the increased cluster size and not to configuration differences. These parameters were listed in Table 4. Many settings apply to all frameworks. One of the more important settings is the time characteristic. Stream processing frameworks need to have a notion of time to be able to assign events to time windows and to discard late events. Events can be late due to various reasons such as network delays. To assign these late events to their correct earlier windows, some frameworks support event time processing. With event time processing,

TABLE 3. Cluster configuration parameters.

Parameter	Setting
a. Apache Flink	
JobManager count	1
TaskManager count	1, 2, 4, or 6
JobManager CPU	2
JobManager process memory	8 GB
JobManager Flink memory	7 GB
JobManager disk	20 GB
TaskManager CPU	1, 2, 4 or 6
TaskManager memory	5, 10, 20 or 30 GB
TaskManager Flink size	memory - 2 GB
Memory managed fraction	0.05
TaskManager disk	20 GB
Number of task slots	CPU of container
Default parallelism	total cluster CPUs
b. Apache Kafka Streams	
Instances count	1, 2, 4, or 6
Instance CPU	1, 2, 4 or 6
Instance disk	20 GB
Instance memory	5, 10, 20 or 30 GB
Java heap	$memory * 0.7$
Streams thread count	CPU of container
Kafka parallelism	total cluster CPUs
c. Spark Streaming and Structured Streaming	
Master count	1
Worker count	1, 2, 4, or 6
Master CPUs	2
Master memory	8 GB
Master disk	20 GB
Worker CPUs	1, 2, 4 or 6
Worker memory	5, 10, 20 or 30 GB
Worker disk	20 GB
Driver cores	2
Driver heap	6 GB
Executor cores	CPU of container
Executor heap	$(memory - 1) * 0.9$
Nb executors	worker count
Default parallelism	total cluster CPUs
SQL shuffle partitions	total cluster CPUs

TABLE 4. Job configuration parameters. Specific pipeline parameters are noted with S1 for the enrichment pipeline and S2 for the aggregation pipeline.

a. Apache Flink (v1.11.1)

Parameter	Value	Default
Time characteristic	event time	processing time
Watermark interval	50 ms	/
Out-of-orderness bound	50 ms	/
State backend	FileSystem	InMemory
Checkpoint interval	60s	None
Object reuse	enabled	disabled

b. Apache Kafka Streams (v2.6.0)

Parameter	Value	Default
Commit interval ms	S1: 1s, S2: 60s	30 000
Grace period	5s	0 ms
Producer compression	lz4	none
Producer batch size	200 KB	16 KB
Buffer memory bytes	48 MB	32 MB
Fetch min bytes	10 KB	1 byte
Topology optimization	optimize	none
Window changelog retention	10 min	1 day
Garbage collector	G1GC	Parallel GC

c. Spark Streaming and Structured Streaming (v3.0.0)

Parameter	Value	Default
Serializer	Kryo	Java
Garbage collector	G1GC	Parallel GC
Initiating heap occupancy	35%	45%
Parallel GC threads	total cores	/
Concurrent GC threads	50% of cores	/
MaxGCPauseMillis	200	200
Micro-batch interval (sp.)	S1: 1s, S2: 60s	/
Trigger interval (str.)	S1: 0, S2: 60s	/
Block interval ms	1000/parallelism	200
Locality wait ms	100	3000
Min. batches to retain	2	100
Watermark ms (str.)	50	/
Watermark policy (str.)	max	min

((str.) refers to Structured Streaming; (sp.) refers to Spark Streaming)

the framework bases processing on the timestamps within the events. A detailed explanation of this mechanism has been given in [2]. Most use cases require this time characteristic to ensure correctness in case of late data. Therefore, we use this time characteristic in our experiments when possible. The only exception is Spark Streaming, which only supports processing time. In that case, the event is assigned to windows based on the moment it entered the framework.

When you use event time processing, most frameworks request a tolerance interval for late data. Data is only incorporated in the results if it is not later than this threshold. We set the tolerance interval for late data to 50 ms. This refers to the out-of-orderness bound for Flink and the watermark setting of Structured Streaming. These settings increase the latency of events. It attaches an extra buffering period to the end of the window operations. For Flink, we also set the watermark interval which describes the interval at which the watermark is computed. Kafka Streams makes use of a different mechanism because of its Dual Streaming Model [53]. Window operations in Kafka Streams immediately send out an update for each incoming record. These records can then be filtered for completeness. The grace period in Kafka Streams describes the lateness limit for sending out updates. It has, therefore, no direct influence on the latency so we can put it at a higher value of 5 seconds.

Finally, we tuned the GC algorithms of the frameworks. We compared CMS and G1GC for Kafka Streams and found lower CPU utilization with G1GC. Further tuning of the G1GC configuration parameters did not lead to further improvements so they were left at the defaults. For Spark and Structured Streaming, we compared Parallel GC and G1GC. Based on [54], we adapt the parallel and concurrent GC threads of G1GC to fit the size of the cluster and reduce the initiating heap occupancy to 35% to trigger more frequent but less heavy garbage collection cycles. Also in our sensitivity analysis, we found the best performance for this combination of parameter settings. Flink has its own memory management system and does not require advanced GC tuning [55].

1) APACHE FLINK

Apache Flink [56] is an open-source stream processing framework that positions itself as a fast, in-memory, scalable, distributed processing system for unbounded and bounded data [28]. Since its first release in 2014, it has gained widespread adoption by companies such as Alibaba, AWS, Uber and Zalando.

Most processing pipelines need to store the state of intermediate computations, which are updated as data comes in. This state is stored in a state backend. Flink offers three state backends [28]. The memory state backend is recommended purely for development purposes. The file system backend can be used for production systems with state that fits on the heap. The RocksDB backend should be used for production systems with a large state that cannot fit on the heap. The state of our pipelines fits on the heap and, therefore, we use the file system state backend backed by HDFS for persistent storage.

The interval at which the state is backed up on HDFS is set by the checkpoint interval. A high interval leads to longer recovery periods in case of a failure. When a job fails, Flink restarts it from the last successful checkpoint. The older the checkpoint, the longer the recovery period will be. A low checkpoint interval leads to high processing overheads because checkpointing can become heavy for larger state sizes. Based on this trade-off, we checkpoint every 60 seconds which is not too high and not too low.

The file system backend stores the state on the heap memory. To store as much state as possible, we maximize the heap memory by setting the managed memory fraction to a very low value. Managed memory is not used in our pipeline and this memory can, therefore, be allocated as heap memory. The documentation of Flink recommends this setting [28].

Finally, we enable object reuse since this is recommended by the Flink documentation for production workloads. The setting enables reusing objects for better performance. The setting is not enabled by default because it can cause issues when the user code is not able to handle this [28].

2) APACHE KAFKA: KAFKA STREAMS

Apache Kafka [32], [33] is a distributed message broker, which forms a reliable, scalable, robust intermediary between data producers and data consumers [33], [53]. The developers of Apache Kafka, implemented a library called Kafka Streams [31] that allows data transformations on top of data residing in a Kafka cluster. To run a Kafka Streams application, we spin up one or multiple containers with the same application logic and consumer ID. Each instance of the application connects to the Kafka cluster and relies on Kafka to distribute partitions over threads and for fault tolerance. This means that Kafka Streams applications do not require a pre-deployed processing cluster such as Flink and Spark, which greatly simplifies deployment.

One of the most important parameters of Kafka Streams is the commit interval. This setting manages how frequently records are committed. It, thereby, influences how often output is emitted [31]. Therefore, we set the commit interval equal to the slide intervals of the windows in our pipeline. The commit interval is one second for the enrichment pipeline and 60 seconds for the aggregation pipeline.

We base ourselves on [57] for optimizing our job settings for different objectives such as throughput, latency, and scalability. To reduce the number of requests, we increase the batch size of the producer to 200 KB. This reduces the load on the producers and brokers. We use lz4 compression to reduce the size of messages. We increase the buffer memory and the number of bytes that are fetched per request to the broker, as per [57].

Merely following the recommendations of [57] did not lead to sufficiently good performance. We also had to set fetch min bytes, which determines the minimum amount of data the server should return for a fetch request. The request is only answered when the buffer is full, leading to increased latencies. When we set the parameter to the recommended 100 KB the average CPU utilization decreased 17 percent but the median latency increased from 800 to 1800 ms and the p99 latency rose to 7300 ms. When we leave the parameter at the default value the maximum sustainable throughput is much lower because the CPU threshold of 80% is reached much sooner. Therefore, we put this value to 10 KB which guards our latency performance but still reduces the load on the CPU.

For higher throughput levels, we noticed that the job became unstable after 20 minutes when it started cleaning state. The load of cleaning the state was so high that it made the framework unstable. To mitigate this, we shortened the retention time of state in windowing operations to ten minutes. Now, the job remained stable and state cleanup happened more promptly.

We can conclude that Kafka Streams requires tuning of a broad range of parameters. Many of these parameters have large effects on latency, throughput, and resource utilization. Even though a highly tunable system is an advantage for some use cases, it is still a lengthy process to go through all documentation and find a parameter set that gives good all-round performance.

3) APACHE SPARK: SPARK STREAMING AND STRUCTURED STREAMING

The last two frameworks are part of the Apache Spark ecosystem [58]. They were developed by Databricks and have a strong and active community. Spark Streaming [52] is the older API and is based on the DStream and RDD API [59]. Structured Streaming [60] was released more recently and offers a high-level processing API. Structured Streaming uses the Spark SQL engine to do pipeline optimizations. The Spark SQL and Structured Streaming APIs are unified and can run batch and streaming workloads with minimal code changes.

A Spark cluster runs as a master-slave architecture [58]. The application running on the cluster exists of a driver and executors. The driver runs the application and connects to

the Spark master to request resources for the executors. The executors execute the tasks scheduled by the driver. Spark applications can run in either client mode or cluster mode. In cluster mode, the driver of the application runs on the workers. In client mode, the driver runs as a separate container. We use client mode to fully use the resources of the workers for processing.

As recommended by the Spark documentation [58], we use Kryo serialization since it is faster than the default Java serializer. This setting is not the default because it requires manual registration of classes within the pipeline. We register all classes for both Spark Streaming and Structured Streaming.

Spark Streaming and Structured Streaming are micro-batch frameworks. This means that they buffer data at the pipeline source and trigger processing at specified intervals. Spark Streaming uses a fixed micro-batch interval. We set this equal to the window slide interval which is one second for the enrichment pipeline and one minute for the aggregation pipeline. Within a batch interval, data is split into partitions based on the block interval. Therefore, we set the block interval equal to the batch interval divided by the desired parallelism. Structured Streaming works with a fixed or dynamic batch interval. With a dynamic batch interval, the framework processes batches as fast as possible. The batch interval is then equal to the processing time of the preceding batch [61]. We use this setting for the enrichment pipeline and enable it by setting the trigger of our sink operation to zero. For the aggregation pipeline, we set the trigger of Structured Streaming at a fixed interval of one minute. By setting the batch intervals equal to the window durations, we trigger output for each window as quickly as possible which would be desired in a real-world scenario.

Structured Streaming infers the minimum and maximum event time of a batch after the computation has finished. By default, Structured Streaming uses the minimum timestamp as its internal time notion. This means that the watermark is lower than the event time of most events in the batch and that most events are buffered for multiple intervals before they are emitted. This causes high latencies for stateful operations. To speed up the emission of output, we use the maximum timestamp of the batch as the time notion of the system. Events are then buffered for fewer trigger intervals. However, events of slower streams get dropped aggressively. The streams in our use case were not delayed so we did not experience any message loss.

The scheduler of Spark takes into account several factors when scheduling tasks on executors. One of these factors is data locality. The executor tries to schedule tasks as close to the data as possible, preferably within the same JVM. When there is no core is available in that JVM, it waits a specified time interval before scheduling the task further away. This period is set by the locality wait time parameter. By default, this is set to three seconds, which is too high for streaming jobs because most tasks are small and often take less than a second. Therefore, we reduce the locality wait time to 100 ms to avoid unnecessary delays in the scheduling of tasks. For scalability runs with workers with 6 CPUs and 30 GB, we further lower the locality wait to 0 ms for Spark Streaming because we noticed an uneven distribution of tasks over executors. Finally, to reduce memory pressure, we decrease the minimum number of batches to retain to two.

F. SCALABILITY WORKLOAD

The main selling point of distributed stream processing frameworks is their built-in scalability. In practice, a cluster is scaled up when it needs to sustain higher throughput. Scaling a cluster implies increasing its resources, i.e. CPU and memory (Table 3). These extra resources are either added to the existing workers or via additional workers. To make use of these extra cores, the parallelism of the job also needs to increase. The third type of resource used by the framework is network bandwidth. In our experiments, the network bandwidth available to the cluster is not explicitly scaled but we ensure that the EC2 instances are not using their entire bandwidth, as explained in Section III-D6. Because we make sure that the network bandwidth is not the bottleneck of the processing job, it does not influence the scalability. Moreover, we do not experiment with network saturation in relation to scalability because it is hard to control. Network saturation happens at the level of the EC2 instance. Since we run on public cloud infrastructure, the available maximum network bandwidth can fluctuate and has no stable limit. Additionally, we have multiple components running on the same DC/OS cluster. When the network is saturated, all components suffer, e.g. the data generator which publishes data onto Kafka, the Kafka brokers themselves, etc. Therefore, it is hard to attribute performance degeneration to the right component. Because of this, we focus on studying CPU and memory bottlenecks. These can be easily controlled via resource allocations to the framework containers. At an instance level, we allocate less than 90% of its available CPU and memory. We do this to make sure that the cluster components are always able to fully use their allocated resources.

The analyzed cluster layouts are listed in Table 5. We study two types of scaling: horizontal and vertical. Horizontal scaling implies an increase in the number of workers while the resources per worker remain equal. This form of scaling drives parallelism and is often cheaper because smaller instances are often cheaper than very large instances. On the contrary, it is also the most complex direction for scaling since it increases overheads related to coordination and data transfer. For horizontal scaling, we analyze clusters with 1, 2, 4, and 6 workers. Each worker has 6 CPUs and 30 GB memory. In vertical scaling, the available resources per worker are increased, while keeping the number of workers equal. We experiment with workers with 1 CPU and 5GB memory, 2 CPUs and 10 GB memory, 4 CPUs and 20 GB memory and 6 CPUs and 30 GB memory. By evaluating both types of scalability we can infer the optimal cluster layout for a processing job, i.e. many small instances versus fewer large instances.

TABLE 5. Results for scalability: maximum sustainable throughput in messages per second for each cluster size. Cells with similar colors show clusters with similar total resources but a different cluster layout. The percentages in between the rows represent the scaling efficiency. 100 % indicates perfect scaling, a higher number indicates superlinear scaling, a lower number indicates sublinear scaling.

a. Enrichment pipeline

	Flink	Kafka	Spark	Struct.
Horizontal scaling (6 CPU; 30 GB per worker)				
1 worker	180	40K	220K	80K
2 workers	310K	80K	370K	160K
	86%	100%	84%	100%
4 workers	610K	160K	680K	300K
	98%	100%	92%	94%
6 workers	800K	245K	880K	380K
	87%	102%	86%	84%
Vertical scaling (6 workers)				
1 core; 5 GB	140K	40K	230K	40K
2 cores; 10 GB	350K	95K	410K	160K
	125%	119%	89%	200%
4 cores; 20 GB	610K	160K	770K	310K
	87%	84%	94%	97%
6 cores; 30 GB	800K	245K	870K	350K
	87%	102%	76%	82%

b. Aggregation pipeline

	Flink	Kafka	Spark	Struct.
Horizontal scaling (6 CPU; 30 GB per worker)				
1 worker	40K	/	160K	140K
2 workers	80K	/	330K	310K
	100%	/	103%	111%
4 workers	180K	/	590K	590K
	113%	/	89%	95%
6 workers	280K	/	880K	900K
	104%	/	99%	102%
Vertical scaling (6 workers)				
1 core; 5 GB	40K	/	110K	100K
2 cores; 10 GB	90K	/	290K	290K
/	113%	/	132%	145%
4 cores; 20 GB	160K	/	690K	590K
	89%	/	119%	102%
6 cores; 30 GB	280K	/	880K	900K
	117%	/	85%	102%

With the cluster sizes that are tested, some frameworks can process up to 800 000 messages per second (see Results Section). This number is far above the requirements of most use cases. Therefore, we consider these cluster sizes to be large enough and representative of most real-world deployments for our pipelines.

In our experiments, workers run on different machines and are never colocated. This is the most challenging scenario for scalability because it increases the number of data shuffles over the network. We also consider this to be the most realistic scenario because, in practice, these clusters run on large infrastructures where the chances are high that workers are not colocated on nodes.

For each cluster layout, we determine the peak sustainable throughput and the resource bottleneck. We follow the same approach as OSPBench [9] to measure peak sustainable throughput. We inflate the input stream to increasing throughput levels and analyze the latency and resource utilization to determine whether the framework can handle the throughput.

Because a throughput increase is the main objective of scaling, we define the scaling efficiency as the percentage of intended throughput that was realized. To compute this, we compare the throughput with that of a base cluster. The base cluster is the cluster that is one step smaller in the same scaling direction. We first compute the throughput increase factor (TIF) as the percentage of throughput increase as a result of the scaling, $TIF_P = \frac{TP_P}{TP_{base}}$ and the resource increase factor (RIF) as the percentage of resource increase due to scaling, $RIF_P = \frac{R_P}{R_{base}}$ with P as the increased cluster scale. The scaling efficiency can then be computed by dividing the TIF and RIF.

$$Eff_P = \frac{TIF_P}{RIF_P} = \frac{TP_P * R_{base}}{TP_{base} * R_P}$$

We compare each cluster size with the cluster one step smaller because we believe this gives the most detailed information on how scaling efficiency evolves as the cluster grows. Additionally, these frameworks are not meant to run on small instances. Comparing the performance of large clusters to clusters that are too small, gives a tainted view of high scaling efficiency.

Multiple studies on the scalability of distributed systems [6], [62] state that additional quality-of-service metrics should be taken into account, besides scaling efficiency. Even though the goal of scaling is to increase throughput, it must not harm other performance metrics. Therefore, we also take into account the effect on latency, which is the second-most important metric for stream processing jobs.

IV. RESULTS

In this section, we discuss the results of our experiments. The scalability results have been summarized in Table 5 and Figures 3 and 4. In Table 5, you can find the peak sustainable throughput for each cluster layout and the scaling efficiency. These results are translated into throughput per CPU in Figures 3 and 4. First, we determine the peak sustainable throughput and resource bottleneck of every cluster layout, pipeline, and framework. Afterward, we link these results to the scaling efficiency and describe its influencing factors.

A. PEAK SUSTAINABLE THROUGHPUT AND RESOURCE BOTTLENECK

Before we evaluate scalability, we briefly discuss the behavior and throughput bottleneck of each pipeline and framework.

1) APACHE FLINK

When we execute the enrichment pipeline for Flink, we notice that latency remains at a similar level for all sustainable throughput levels. When the peak throughput is reached, the buffers on the input queues start increasing and latency starts rising. When this happens, CPU utilization is between 60% and 70% so this is not the limiting factor. Once the job becomes unsustainable, the CPU utilization jumps to nearly 100 %. For the aggregation pipeline, CPU utilization

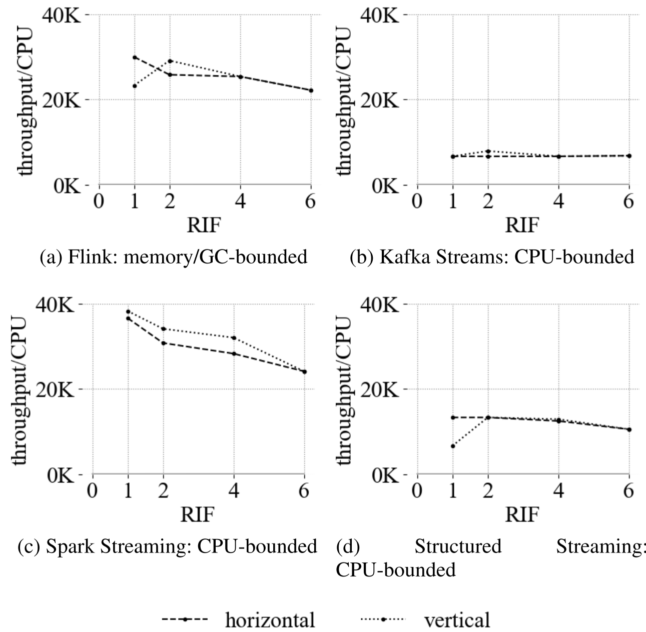

FIGURE 3. Enrichment pipeline: throughput per CPU for each resource increase factor (RIF) for horizontal and vertical scaling.

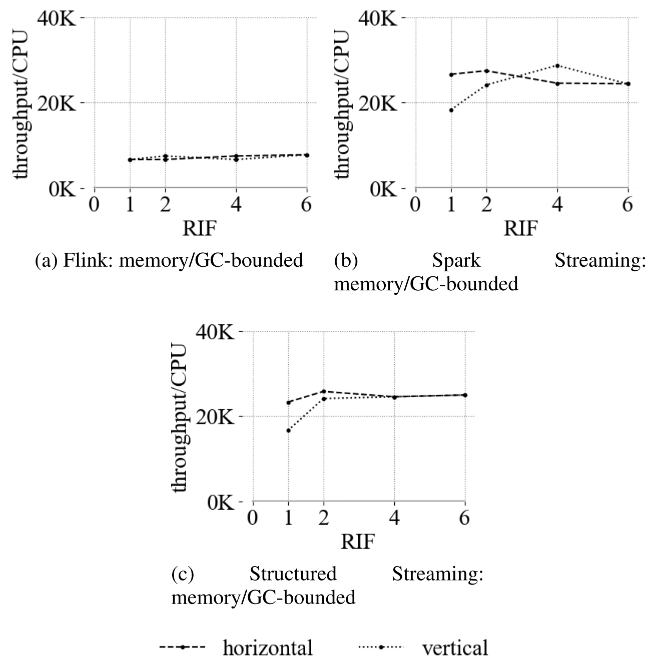

FIGURE 4. Aggregation pipeline: throughput per CPU for each resource increase factor (RIF) for horizontal and vertical scaling.

increases gradually until it passes 80% and the job becomes unstable.

For the unsustainable runs of both pipelines, we notice that CPU utilization in user mode is above 80%. When we compare the metrics of sustainable and unsustainable runs, we notice that this is mainly due to heavy garbage collection. For example, the garbage collection time of the enrichment pipeline rose by 250% (to more than 6 minutes) when the throughput was increased from 38K to 48K msg/s. A throughput of 38K msg/s was still sustainable, while 48K msg/s was unsustainable. The time spent on GC starts increasing when RSS memory hits the container limits, i.e. 5 GB for the smaller containers and 28 GB for the larger ones. Flink makes use of G1GC. By default, this algorithm tries to keep the duration of minor GC cycles below 200 ms. When we look at the duration of the G1 Young Generation collection, we notice that this is getting out of control with peaks above 2 and 5 seconds. When we increase the load further, we see major GC (G1 Old Generation) kicking in with stop-the-world pauses of up to 8 seconds. This means GC is losing its effectiveness and the system is becoming unstable.

As expected, the base state of the enrichment pipeline is much smaller than that of the aggregation pipeline. The aggregation pipeline buffers each incoming event for 5 minutes. The enrichment pipeline can discard state after one second, so the used heap memory after a GC cycle is only a few MBs large. This makes GC very efficient and fast. The G1 Young Generation duration remains below 200 ms at all times for sustainable runs. For the aggregation pipeline, GC is much heavier. The G1 Young Generation duration often takes more than one second when a new window has been processed, even for sustainable runs.

To conclude, memory pressure and heavy GC cycles are the cause of unsustainability for both Flink pipelines. GC is heavier for the aggregation pipeline. Therefore, the sustainable throughput of the aggregation pipeline is much lower than that of the enrichment pipeline for Flink.

2) APACHE KAFKA: KAFKA STREAMS

Kafka Streams becomes unsustainable when CPU utilization passes the 80% threshold. These CPU cycles are mainly spent in user mode. The GC collection time over the entire run is not high enough to be the main culprit of this. When we do more detailed profiling, we notice that tasks related to state management are the main CPU hotspot. Kafka Streams uses RocksDB to keep state and CPU time is mainly spent on flushing, seeking, putting, and getting data, etc. We did several experiments to see if we could alleviate the load. In version 2.6, the flushing of the state stores is tight to the commit interval (one second). This gives a high load on the state store when it needs to process considerable amounts of state. When we raise the commit interval to ten seconds, peak throughput increases, confirming the results of [19], but the median latency grows to almost 6 seconds and p99 latency becomes more than 10 seconds. In a future version of Kafka Streams, the flush interval will be decoupled from committing results so that latency and cache/ RocksDB performance are not linked anymore. Also, RocksDB spills to disk which is a very expensive operation. As an experiment, we used the in-memory window stores but the load remained high. Mainly because there are many small windows in our job, which causes methods such as the one for registering new window stores to consume a lot of CPU time.

As explained, Kafka Streams had difficulties with state management for the enrichment pipeline. The aggregation

pipeline requires much more state to be kept and Kafka Streams is not able to reach reasonable performance. The aggregation pipeline is designed to stress the state management aspects of the framework. The same pipeline can be implemented with incremental windows which would alleviate the issues for this specific use case but would fail the objective of this experiment. Because of this, we left Kafka Streams out of the comparison for the aggregation pipeline.

3) APACHE SPARK: SPARK STREAMING

Spark Streaming is a micro-batch framework. This influences its behavior. For both pipelines, latency does not remain constant as throughput increases because the size of a batch becomes larger and it takes longer to process it. CPU utilization increases linearly with throughput, as for the event-driven frameworks. Both pipelines have different throughput bottlenecks.

For the enrichment pipeline, Spark Streaming becomes unsustainable as soon as it cannot get through the processing delay at startup. The first batches always take longer due to the heavy initialization overhead of Spark. If there is no spare CPU capacity to catch up with this delay, the system becomes unstable. In the last sustainable run, it took the framework almost 15 minutes to catch up. Previous work argues that Spark is not very CPU efficient, e.g. [39], [41]. Due to inefficiencies in their micro-batch approach, Spark Streaming and Structured Streaming do not reach 100% CPU utilization. When we look at the DAG of the enrichment pipeline, we see that the job gets split into four stages. The first two stages are parallel stages for the ingest of the flow records and the speed records. The third stage is the joining stage and the final stage is the aggregation stage. When the batch interval times out, the batch is processed stage by stage and the framework waits for a stage to finish before continuing with the next one. This means that when one task takes longer than another task, all the other cores remain idle until the task finishes and they can continue to the next stage. This idle time leads to resource inefficiency and increased latencies because each stage is as slow as its slowest task. This can be improved by increasing the number of (shuffle) partitions. However, this didn't improve performance for our use case because it increased scheduling overheads and other delays.

The aggregation pipeline behaves differently. The processing of the initial batches is much lighter because the batch interval is much larger and not much state has accumulated yet. Every minute the window is triggered and we see a peak in latency, CPU, and memory. Eventually, the memory becomes the bottleneck. For cluster layouts with small instances memory fills up, while for cluster layouts with large instances the memory management overheads become too large.

In the beginning of unsustainable runs with small instances, memory usage increases and page faults show peaks. When memory limits are hit, Old Generation GC is triggered which sets off a train of effects. CPU utilization is 100% for the instance doing the Old Generation GC but is 0% for the other instances. Once Old Generation GC is triggered, latency starts increasing continuously. To reduce memory pressure, the job also occasionally starts writing to disk. The final sustainable runs also triggered Old Generation GC but did not yet write to disk. When the job starts writing to disk, the GC collection time soars, and eventually, the executor dies.

When we run the aggregation pipeline with one large instance, we see slightly different behavior when throughput becomes unsustainable. The job does not run out of memory but GC starts taking so much time that CPU becomes the bottleneck. The CPU utilization is continuously above 80% to process the batches in time and clean up memory. The peaks in the duration of G1 Young Generation GC go far above the 200 ms pause threshold, up to 2 seconds, and the total GC collection time is above 8 minutes for a run of 30 minutes. There is no writing to disk yet and all three memory pools clean up well after a GC cycle. The heavy GC cycles cause a high load on the CPU cores. When throughput is increased further, the CPU utilization remains constantly at 100% and the job becomes unstable and eventually fails.

To conclude, the enrichment pipeline became unstable because the batch processing time did not leave enough buffer to catch up with the startup delay. The aggregation pipeline was memory-intensive and failed either because the executor went out-of-memory or because GC started consuming too much CPU time.

4) APACHE SPARK: STRUCTURED STREAMING

The behavior of Structured Streaming is quite similar to that of Spark Streaming. For the enrichment pipeline, the framework becomes unsustainable when there is not enough CPU buffer to catch up with the startup delay. The final sustainable run was only able to get through the startup period after 25 minutes. The initialization overhead of Structured Streaming is larger than that of Spark Streaming because it is based on Spark SQL and does more optimizations under-the-hood at startup. This leads to a lower sustainable throughput for Structured Streaming than for Spark Streaming in the enrichment pipeline.

The aggregation pipeline requires heavy state management. Since it takes a while for the state to build up and since the batch interval is quite large, the startup period is easier and is not the cause of unsustainability. For this pipeline, memory management becomes the bottleneck. During the first minutes of an unsustainable run, we see a steep increase in heap memory usage. After six to seven minutes, G1 Young Generation GC duration starts peaking and G1 Old Generation GC is triggered. However, GC is not effective at cleaning up memory. The collection time skyrockets. The executors start writing to disk and eventually they fail. For the final sustainable runs, the duration of Young Generation GC surpasses the 200 ms pause goal and Old Generation GC is frequently triggered, but the job still shows stable latency and memory patterns.

Besides the different throughput bottlenecks, we also notice some other differences in behavior between the two pipelines. The enrichment pipeline processes as fast as possible with a dynamic batch interval which grows as throughput increases. The median latency is around 10 seconds when the framework becomes unsustainable. At this point, the batch interval is around 3 to 4 seconds. Events are buffered for multiple intervals due to the mechanism of Structured Streaming for watermark propagation, as described earlier. This happens for both the join and the window operation. The aggregation pipeline, on the contrary, only has one stateful operation. Also, we manually set the processing trigger at one minute, which leads to a very different latency pattern that is similar to that of Spark Streaming. The batching mechanism now works very similarly for both frameworks and the sustainable throughput is also approximately equal. Structured Streaming benefits more from longer batch intervals than Spark Streaming. Two reasons for this are that Structured Streaming checkpoints every batch interval and that the overhead from event-time processing is larger. Further CPU and memory utilization differences between the pipelines are similar as for Spark Streaming.

In summary, we notice different causes of unsustainability for different pipelines. The enrichment pipeline is CPU-bounded, while the aggregation pipeline suffers from memory pressure.

5) CONCLUSION

The enrichment pipeline has two chained stateful operations that require shuffling, short windows with frequent output, and a quickly changing state. The shuffling and the managing of the short windows, make this pipeline CPU-intensive. Due to this, Kafka Streams, Spark Streaming and Structured Streaming are CPU-bounded. The bottleneck of Kafka Streams is in the maintenance of its RocksDB state backend. Due to the short windows and commit interval, a lot of CPU cycles are spent on flushing, seeking, and putting data into the state backend. Spark Streaming and Structured Streaming mainly have difficulties with catching up with the startup delays. Both frameworks have a heavy initialization overhead, and if there is not enough spare CPU capacity per batch interval, delays start accumulating and the framework becomes unstable.

The aggregation pipeline uses only one stateful operation with a larger window, a larger base state, and less frequent output. This causes memory and memory management (GC) to become the bottleneck. To stress the state management system of the framework, we use a non-incremental window implementation that requires a large state to be kept to compute the window results. With small instance sizes, jobs go out of memory and fail. With larger instances, GC starts taking up a lot of the CPU capacity. Eventually, the framework does not have enough CPU cycles left to process all events and becomes unstable. We see this happening for Flink, Spark Streaming, and Structured Streaming. For Kafka Streams, the management of the RocksDB state backend becomes so heavy that it is incapable of reaching reasonable throughput so we left it out of the results for this pipeline.

Because both pipelines have very different characteristics, the peak sustainable throughput also varies significantly (Table 5). Flink, Kafka Streams, and Spark Streaming reach much higher throughput with the enrichment pipeline than with the aggregation pipeline. The only exception is Structured Streaming which suffers from the chained stateful operations of the enrichment pipeline because of its watermark propagation mechanism. It also benefits from the large batch interval of the aggregation pipeline because it has a significant processing overhead for each batch (e.g. checkpointing, watermark propagation, etc). The large batch interval and small state at the beginning of the run make it easier to get through the startup period.

Finally, we want to stress again that the largest cluster sizes of some frameworks can process over 800 000 messages per second. We consider this to be more than would be required for most real-world use cases. Therefore, we do not further increase our clusters.

B. SCALING EFFICIENCY

Before we discuss the scaling efficiency, we want to remind the reader that throughput is measured by step-wise increases in throughput while checking for sustainability. Sustainable throughput is therefore only precise up to the step size, which is around 10K for smaller cluster sizes and around 20K for larger cluster sizes. This means that the throughput and scaling efficiency are approximations and should not be seen as precise up to a percentage level. Measuring the exact peak sustainable throughput is impossible because it is not a static number. Every run is subject to some external uncontrollable factors such as a momentary increase in network delay. These factors differ across runs and can cause runs on the edge of sustainability to sometimes succeed and other times fail. Additionally, it would take a very large number of runs to determine throughput at a fine-grained level. A single run needs to last at least 30 minutes which makes it very time-consuming with little extra gains. Therefore, we opted for a precision of 10K-20K. Most runs have a sustainable throughput above 150K so this gives us satisfactory precision. Finally, we only consider a throughput level sustainable if all three runs for that level were sustainable.

When we look at the results, we can determine some interesting patterns. These patterns help us determine the influencing factors of scalability. Since scalability is tightly related to peak throughput, these factors also influence the throughput bottleneck. Based on our experiments, we determine five factors that influence the scalability of processing jobs.

1) INFLUENCE OF INITIAL CLUSTER LAYOUT AND SCALING DIRECTION ON SCALING EFFICIENCY

When looking at the results, we notice that the efficiency of scaling depends for a large part on the initial cluster layout and the scaling direction.

First of all, we often see a superlinear throughput increase when scaling from one core to two cores. We see an increase in throughput per CPU when going from one core to two cores for most frameworks in Figures 3 and 4, and in Table 5 we see efficiency numbers above 100%. The frameworks themselves create an overhead that is significant for small cluster sizes. A part of this overhead is fixed and is equally large for small and large instances. An example of this is the heap memory of Spark frameworks. The formula for heap memory is $(memory - 1GB) * 0.9$. This leads to an overhead of 28% for an instance with 5 GB memory but only an overhead of 13% for an instance with 30 GB. Consequently, the overhead is especially large for a cluster with many small instances (6 workers, 1 CPU, 5 GB). The heap size of one instance with 30 GB memory is 26 GB. The total heap size of six instances with 5 GB is only 21 GB. Therefore, the superlinear throughput increase when scaling up from the smallest worker sizes can be contributed to the superlinear scaling of available resources for the pipeline computations. Additionally, using many small instances increases the overhead in communication, coordination, and data transfer. This overhead is much smaller for a few large instances.

On a side note, it is important to keep in mind that multiple smaller workers increase fault tolerance. When one worker fails, there are multiple other workers available to take over. This is not the case when there are just one or two workers.

Scaling horizontally or vertically leads to a trade-off between GC time and communication overhead. Using multiple smaller workers increases communication costs between workers and between the workers and the master. With larger workers, the heap size increases and GC becomes more costly. For our pipelines, vertical scaling works better for smaller instances. As instance size grows, the relative overhead per instance reduces. When instances have reached 2 to 4 cores, the efficiency of horizontal and vertical scaling converges. Once instances are large enough to suffer less from the overheads, we see slightly better performance for more smaller instances than for fewer large instances. This can be seen in Figures 3 and 4 by the vertical scaling line which is often slightly higher than the horizontal scaling line.

2) INFLUENCE OF PIPELINE DESIGN ON SCALING EFFICIENCY

The second influencing factor is pipeline design. The characteristics and complexity of the pipeline operations heavily influence the throughput bottleneck. Independent of the framework, an inefficient pipeline definition can induce bottlenecks and limit scalability.

For the enrichment pipeline, we observe a diminishing efficiency for increasing cluster sizes for all frameworks except for Kafka Streams. We see a clear diminishing slope in Figure 3 and a decrease in the scaling efficiency in Table 5. Scaling from 1 to 2 cores or from 1 to 2 workers has the highest return in terms of scaling efficiency. As more cores or workers are added, the extra gains diminish. These results are consistent with Amdahl's law [63] which, in simple terms, states that the efficiency of scaling diminishes for larger cluster sizes due to sequential tasks that do not benefit from the extra parallelism. This phenomenon has been consistently proven and confirmed for batch workloads in literature, e.g., [35], [37]–[39], [41]. Amannejad *et al.* [41] describe the increased shuffling overhead that comes with larger cluster sizes. This leads to more (de)serialization operations, network latencies, delays, synchronizations between threads, blocking I/O operations, etc. The inefficiencies caused by this have been confirmed by [35] and [39]. Most of this overhead is related to shuffling and is more demanding for CPU than memory. The enrichment pipeline is mainly CPU-bounded so this causes a sublinear throughput increase.

The scaling efficiency of the aggregation pipeline is approximately constant when scaling up higher than 2 workers or 2 cores (Figure 4). This pipeline is mainly heavy on memory usage and management and much less on CPU and shuffling. As the cluster size increases, overheads reduce and scaling stays fairly efficient.

We conclude that different processing pipelines stress different parts of the computing system and, thereby, influence the efficiency of scaling. In this section, we did not yet discuss the combined effects of framework design decisions and certain characteristics of the pipeline (e.g., frequent output, multiple stateful operations, large state, etc.). This will be the topic of the next section.

3) INFLUENCE OF FRAMEWORK DESIGN ON SCALING EFFICIENCY

The framework design also influences the scaling efficiency. Figure 3 and Table 5 show that Kafka Streams scales approximately linearly as cluster size increases and the difference between horizontal and vertical scaling is marginal. This can be attributed for a large part to the framework design. The Kafka Streams job becomes unsustainable due to CPU pressure from maintaining the state in the RocksDB backend. As we explained in the previous section, the commit interval and flushing interval are coupled. When we trigger frequent output, this causes heavier state management. Since CPU is the bottleneck, doubling CPU power doubles the peak throughput. However, for most frameworks, this would also increase communication costs and therefore lead to a sublinear throughput increase. This is not the case for Kafka Streams because shuffling does not happen directly in between instances or threads but is done by channeling all data through intermediate Kafka topics. This means that the shuffling cost of an event remains the same for all possible cluster layouts and sizes. For example, we measure the same network transfer for a cluster with a few large instances (1 worker, 6 CPU, 30 GB) as for a cluster with many small instances (6 workers, 1 CPU, 5 GB). This mechanism leads to a lot of redundant shuffling. Because of this, the peak throughput is low compared to other frameworks, but we do not notice a diminishing efficiency when scaling. Our results give more nuance to the only previous study [19] on

the scalability of Kafka Streams. This previous study did not adapt the configuration parameters to the cluster size. When investigating instances of different sizes (0.5 CPU with 2GB, 1 CPU with 4 GB, and 2 CPU with 8GB), each instance still had only one processing thread configured. Therefore, no substantial performance difference was noted between using 1 CPU and 2 CPUs. When going from 0.5 CPU to 1 CPU, they found that they could handle more than double the amount of throughput. Here, the single processing thread was able to benefit from the increase in resources. We also found that Kafka scales linearly when it makes full use of its resources. We also show that Kafka Streams is still able to benefit from vertical scaling as instance sizes get larger than 2 CPUs. Even though it was not explicitly mentioned in [19], their results also show no signs of diminishing efficiency, similar to ours.

Additionally, in the previous section, we described the superlinear throughput increase of vertical scaling of very small instances. Structured Streaming shows the highest superlinear throughput increase and this is related to its design. It has a scaling efficiency of 200% for the enrichment pipeline and 145% for the aggregation pipeline. This is also visible in Figures 3 and 4 because the vertical scaling line is often slightly higher than the horizontal scaling line. When we use a cluster with six small instances, there is not enough memory to ingest the first batches of data. To avoid overflowing memory, we set the maximum number of offsets per trigger to the equivalent of 20 seconds of input data. This increases the maximum throughput that can be sustained. Nonetheless, it is still much less than the throughput of Spark Streaming for the enrichment pipeline. We described the differences between Spark Streaming and Structured Streaming in Section IV-A. We explained how Structured Streaming does not perform well on stateful pipelines with short micro-batch intervals due to high startup delays, state management overhead, and its watermark mechanism. For larger cluster sizes, the peak throughput of Structured Streaming is reached when latency crosses the threshold. The reason for these high latencies is mainly due to is watermark mechanism, as explained earlier. Spark Streaming does not offer event time processing which simplifies processing and reduces the overhead. It also keeps its latency low and, thereby, increases its peak throughput.

The aggregation pipeline only contains one stateful operation and uses infrequent triggers. This is a much better match for Structured Streaming because the overhead is lower, due to larger batches and less stateful operations. We now see approximately similar throughput and scaling efficiency as Spark Streaming.

Spark Streaming has the most stable behavior across both pipelines. The throughput of both pipelines does not differ very much. The performance of the aggregation pipeline degrades more when we use small instances, while the enrichment pipeline leads to a stronger diminishing efficiency. The scaling efficiency of Spark Streaming drops when going from 4 cores to 6 cores for both pipelines. This confirms the study of Awan *et al.* [34] who found that for batch jobs Spark scales almost linearly up to 4 cores per executor and does not benefit from more than 12 cores per executor. Beyond four cores, the speed-up was sublinear. As the input data size or throughput increases, the wait time during I/O operations and garbage collection increases as well. GC becomes a bottleneck since GC time increases superlinear with data size.

4) INFLUENCE OF RESOURCE ALLOCATION ON SCALING EFFICIENCY

When we scaled our cluster, we didn't change the ratio of CPU cores to GBs of memory. However, jobs that are especially heavy on either CPU or memory, may benefit from different ratios. This should be taken into account when scaling. If a processing job is limited due to a specific resource, increasing only that resource may be enough to alleviate the bottleneck and scale most efficiently.

5) INFLUENCE OF DATA CHARACTERISTICS ON SCALING EFFICIENCY

Finally, we briefly discuss the influence of data characteristics on scaling. Data characteristics can induce processing bottlenecks in the pipeline. For example, data skew is known to have very large performance implications, mainly for join operations [64]. Data skew occurs when the load is not evenly distributed over all partitions. Some partitions contain much more data than others. In some cases, this can become quite extreme with over 80% of the load on the same partition. The cores which need to process these partitions will often become the bottleneck of the pipeline. A job with high data skew does not scale well since adding more cores does not alleviate the load on the core which needs to process the heavy partition. To make the job scalable, the heavy partition needs to be split into smaller partitions via techniques such as salting. In our experiments data was uniformly distributed over keys so we do not suffer from skew and scaling is effective.

V. CURRENT AND FUTURE RESEARCH DIRECTIONS

Improving the scalability and optimizing the cluster size of stream processing jobs are active areas of development and research. In this section, we touch upon some areas of improvement related to scalability and we introduce the concept of elasticity which is closely related to scalability.

A. IMPROVING JOB PERFORMANCE

A lot of effort has been done to make these frameworks performant. These efforts relieve throughput bottlenecks and, thereby, influence the required cluster size for a processing load. Framework development has focused on several topics such as memory management, checkpoint efficiency, the networking stack, etc. In this section, we address a few substantial ones.

The Spark community put substantial effort into improving the efficiency of memory and CPU utilization. This umbrella project existed of several smaller improvements and

was called Project Tungsten [65]. One of the initiatives was to enable binary processing and improve memory management. The main objective here was to leverage application semantics to manage memory explicitly and eliminate the overhead of the JVM object model and garbage collection. This comes from the idea that Spark should understand the lifecycle of memory blocks much better than the standard GC algorithms which base themselves on heuristics and estimations. Besides that, Java objects have a significant memory overhead. To tackle this, an explicit memory manager was implemented which can work directly on the binary data [65]. A second initiative under Project Tungsten was to make more effective use of the L1/L2/L3 CPU caches to improve processing speed. Keeping data in the CPU cache reduces the wait time of fetching data from the main memory. A third initiative was whole-stage code generation and was centered around improving code generation for SQL query evaluation and optimization.

Similar to Tungsten, Flink also operates on binary data [55]. Since its early releases, Flink has been focusing on keeping memory representations as efficient as possible and reducing GC pressure. It has a custom memory management model. Flink v1.10 contained some improvements to its memory management model and extra configuration options [66]. It now allows both high-level and fine-grained tuning of memory components. This makes it easier to manage and debug Flink applications. However, fine-grained tuning requires expert Flink knowledge and is a laborious trial-and-error process.

Since memory and GC are often the bottlenecks, several academic studies also aimed to improve these mechanisms. Some studies propose new GC algorithms tailored to big data applications or improve their memory management, e.g., [67]–[69]. In [25], a set of GC algorithms are evaluated for big data applications in Spark. A few efforts have been done to reduce the GC overhead in Spark, e.g., [70], [71]. Finally, the reader can find a comprehensive overview of GC algorithms for big data systems in [72].

We can conclude that the performance of these frameworks is improved with every release. Improvements may alleviate certain bottlenecks and introduce others. Therefore, it is important to continuously evaluate how they evolve and how these improvements influence our results.

B. ELASTICITY

In our experiments, we scaled the cluster by either adding a worker or by increasing the size of a worker. Both approaches took manual intervention and a relaunch of the cluster. The resources of the cluster were also fixed at a certain size. This means that there is over-provisioning during off-peak hours and, possibly, under-provisioning during peak hours. An active area of study is adapting the resources automatically based on demand. This is called autoscaling [7]. Autoscaling is closely related to the concept of elasticity [21]. Elasticity is the ability of the framework to respond to increased or diminishing resource demands by adding or removing workers from the cluster. Most framework communities have been working on this because it is an important characteristic for many use cases.

The Spark framework includes a feature called Dynamic Resource Allocation, which allows a job to adapt the number of executors based on the number of pending tasks. Executors are released when they are idle and new ones are added when there are pending tasks. Scaling up happens exponentially to make sure applications can react promptly to resource demands. In the first round, only one executor is added. When this is not enough two more executors are added, then four executors, and so on. Since this feature allows increasing the number of executors and not the number of workers, it is mainly useful for large clusters with multiple processing jobs, where jobs can share resources more efficiently based on their current processing load.

Kafka Streams currently does not include autoscaling capabilities. A solution is to use the autoscaling features of the underlying infrastructure. For example, when running on Kubernetes, the Horizontal Pod Scaler feature [73] could be used. This allows scaling the number of instances of a service based on monitoring metrics such as CPU load. This feature could also be used to scale up the number of workers of a Spark cluster.

In May 2021, Flink released a new experimental feature called Reactive Mode [74]. This feature allows automatically adjusting the parallelism of a Flink job. The system still relies on an external system to scale the number of task managers based on metrics. Flink will then increase the parallelism of the job as more task slots become available. To scale the number of task managers, features such as Kubernetes Horizontal Pod Scaler or AWS autoscaling groups can be used. Currently, Reactive Mode is only supported for standalone cluster deployments and not for deployments with active resource providers such as Kubernetes, YARN, or Mesos. At the moment, adjusting the job parallelism still requires a job restart which leads to processing delays and may not be acceptable for all use cases.

Finally, improving the automatic scaling of stream processing frameworks is an active area of academic research. For more information on this topic, we refer the reader to the following literature: [75]–[84].

VI. CONCLUSION

Scalability is the ability of a system to spread work across several machines. The scalability of stream processing jobs is of utmost importance in production deployments but has been overlooked in benchmarking literature. It is measured as the increase in peak throughput induced by an increase in cluster resources. Previous benchmarking work mostly studied scalability as a framework feature. Many frameworks claim to be scalable. However, we assume that the eventual scalability of a processing job is much more complex and impacted by several additional factors. Of course, it is a prerequisite that the framework can distribute work over several machines. But once it has this capability, the scaling efficiency of

a processing job in that framework is influenced by a set of factors. In this paper, we list and study these factors. Therefore, the goal of this paper was twofold: benchmark the scalability of processing jobs in four popular frameworks and investigate framework-related and non-framework-related factors which influence this scalability.

To this end, we contribute a broader framework of what should be analyzed when benchmarking scalability. We implemented two pipelines with very distinct characteristics in four popular distributed stream processing frameworks: Flink, Kafka Streams, Spark Streaming, and Structured Streaming. We see differences in the height of throughput, throughput bottleneck, and the efficiency of scaling. The scaling efficiency varies greatly for different combinations of a framework, pipeline, and cluster layout. We determined five main factors which influence scalability.

The first influencing factor is the initial cluster layout and scaling direction. When choosing between horizontal and vertical scaling, there is an important performance trade-off between communication costs and the cost of GC when scaling up. When instances are small, vertical scaling leads to much higher scaling efficiency because it creates fewer overhead. As instances become larger, the efficiency of horizontal and vertical scaling converges, as we notice for Flink, Spark Streaming, and Structured Streaming. Once the instances are large enough to suffer less from the overheads, we notice slightly better performance for more smaller instances than for fewer large instances.

Secondly, the characteristics of the processing pipeline influence the throughput bottleneck and scalability of the framework. For Flink, Spark Streaming, and Structured Streaming, we see different scaling behavior for the different pipelines. The enrichment pipeline is heavier on shuffling and CPU usage and the scaling efficiency diminishes more. The aggregation pipeline is heavier on state management and memory, and the scaling efficiency is more constant.

Thirdly, the design of the framework has an influence. An example of this is the shuffling behavior of Kafka Streams which is independent of its cluster layout and therefore, the efficiency diminishes less. Its inefficient shuffling mechanism and eager state management cause heavy load on CPU and the network and lead to low peak sustainable throughput for our pipeline. Also, the batching behavior of Structured Streaming leads to different bottlenecks for both pipelines which in turn affects scalability. Spark Streaming was able to reach high throughput and showed the most stable performance across both pipelines. However, this can be contributed partly to the fact that Spark Streaming does not support event time processing which significantly simplifies its processing mechanism and reduces the batch overhead.

Another factor that influences the scaling efficiency is the resource allocation. When scaling, the ratio between CPU and memory should be evaluated and adapted to fit the performance characteristics of the pipeline.

The final factor is the input data characteristics. Here, mainly data skew is known to be detrimental for the scaling efficiency.

From our experiments, we conclude that the scaling efficiency of a certain stream processing job is determined by a complex interplay of factors. We also showed that the scalability of frameworks differs over pipelines and that one framework cannot be deemed more scalable than another. Each framework has a distinct design with varying advantages and disadvantages for different pipelines and processing scenarios. Because of this, it is difficult to make recommendations that are generally applicable. A practitioner with a specific use case should start with evaluating the throughput bottleneck and its relation to the current cluster layout. Once this has been determined, the resource which limits throughput should be increased. Possibly, this shifts the throughput bottleneck and saturated resource. Ideally, scaling is first done in a vertical direction. Horizontal scaling can be applied when vertical scaling is not possible anymore or when the efficiency of this reduces because instance sizes become too large to manage. Of course, it should be kept in mind that minimal horizontal scaling is required for fault tolerance. When choosing a scaling direction, the pipeline characteristics play a role as well. A pipeline with many shuffle operations will suffer more from a diminishing efficiency than a pipeline with only one or no shuffle operations.

VII. LIMITATIONS AND FURTHER RESEARCH

An interesting direction for further research is to develop a broader set of pipelines to investigate further how these pipeline characteristics influence scalability. Furthermore, it would be interesting to see how scaling efficiency evolves for very large clusters. Although, we believe very few use cases require a throughput above 800 000 messages per second.

ACKNOWLEDGMENT

This research was done in close collaboration with Klarrio, a cloud-native integrator and software house specialized in bidirectional ingest and streaming frameworks aimed at IoT & Big Data/Analytics project implementations (https://klarrio.com).

REFERENCES

[1] F. Cristian, "Understanding fault-tolerant distributed systems," *Commun. ACM*, vol. 34, no. 2, pp. 56–78, Feb. 1991.

[2] T. Akidau, R. Bradshaw, C. Chambers, S. Chernyak, R. J. Fernández-Moctezuma, R. Lax, S. McVeety, D. Mills, F. Perry, E. Schmidt, and S. Whittle, "The dataflow model: A practical approach to balancing correctness, latency, and cost in massive-scale, unbounded, out-of-order data processing," *Proc. VLDB Endowment*, vol. 8, no. 12, pp. 1792–1803, Aug. 2015.

[3] K. M. Chandy and L. Lamport, "Distributed snapshots: Determining global states of distributed systems," *ACM Trans. Comput. Syst.*, vol. 3, no. 1, pp. 63–75, Feb. 1985.

[4] M. D. Hill, "What is scalability?" *ACM SIGARCH Comput. Archit. News*, vol. 18, no. 4, pp. 18–21, 1990.

[5] A. B. Bondi, "Characteristics of scalability and their impact on performance," in *Proc. 2nd Int. Workshop Softw. Perform. (WOSP)*, 2000, pp. 195–203.

[6] P. Jogalekar and M. Woodside, "Evaluating the scalability of distributed systems," *IEEE Trans. Parallel Distrib. Syst.*, vol. 11, no. 6, pp. 589–603, Jun. 2000.
[7] T. Lorido-Botran, J. Miguel-Alonso, and J. A. Lozano, "A review of auto-scaling techniques for elastic applications in cloud environments," *J. Grid Comput.*, vol. 12, no. 4, pp. 559–592, Dec. 2014.
[8] F. McSherry, M. Isard, and D. G. Murray, "Scalability! But at what {COST}?" in *Proc. 15th Workshop Hot Topics Oper. Syst. (HotOS XV)*, 2015, pp. 1–6.
[9] G. V. Dongen and D. V. D. Poel, "Evaluation of stream processing frameworks," *IEEE Trans. Parallel Distrib. Syst.*, vol. 31, no. 8, pp. 1845–1858, Dec. 2020.
[10] J. Karimov, T. Rabl, A. Katsifodimos, R. Samarev, H. Heiskanen, and V. Markl, "Benchmarking distributed stream data processing systems," 2018, *arXiv:1802.08496*. [Online]. Available: http://arxiv.org/abs/1802.08496
[11] Z. Karakaya, A. Yazici, and M. Alayyoub, "A comparison of stream processing frameworks," in *Proc. Int. Conf. Comput. Appl. (ICCA)*, Sep. 2017, pp. 1–12.
[12] S. Chintapalli, D. Dagit, B. Evans, R. Farivar, T. Graves, M. Holderbaugh, Z. Liu, K. Nusbaum, K. Patil, B. J. Peng, and P. Poulosky, "Benchmarking streaming computation engines: Storm, flink and spark streaming," in *Proc. IEEE Int. Parallel Distrib. Process. Symp. Workshops (IPDPSW)*, May 2016, pp. 1789–1792.
[13] S. Qian, G. Wu, J. Huang, and T. Das, "Benchmarking modern distributed streaming platforms," in *Proc. IEEE Int. Conf. Ind. Technol. (ICIT)*, Mar. 2016, pp. 592–598.
[14] R. Lu, G. Wu, B. Xie, and J. Hu, "Stream bench: Towards benchmarking modern distributed stream computing frameworks," in *Proc. IEEE/ACM 7th Int. Conf. Utility Cloud Comput. (UCC)*, Dec. 2014, pp. 69–78.
[15] C. B. Weinstock and J. B. Goodenough, "On system scalability," Softw. Eng. Inst., Carnegie-Mellon Univ., Pittsburgh, PA, USA, Tech. Rep. CMU/SEI-2006-TN-012, 2006.
[16] M. Li, J. Tan, Y. Wang, L. Zhang, and V. Salapura, "SparkBench: A comprehensive benchmarking suite for in memory data analytic platform spark," in *Proc. 12th ACM Int. Conf. Comput. Frontiers*, May 2015, p. 53.
[17] A. Shukla and Y. Simmhan, "Benchmarking distributed stream processing platforms for IoT applications," in *Proc. Technol. Conf. Perform. Eval. Benchmarking*. Cham, Switzerland: Springer, 2016, pp. 90–106.
[18] A. Shukla, S. Chaturvedi, and Y. Simmhan, "RIoTBench: An IoT benchmark for distributed stream processing systems," *Concurrency Computation: Pract. Exper.*, vol. 29, no. 21, p. e4257, Nov. 2017.
[19] S. Henning and W. Hasselbring, "Theodolite: Scalability benchmarking of distributed stream processing engines in microservice architectures," 2020, *arXiv:2009.00304*. [Online]. Available: http://arxiv.org/abs/2009.00304
[20] G. van Dongen and D. V. D. Poel, "A performance analysis of fault recovery in stream processing frameworks," *IEEE Access*, vol. 9, pp. 93745–93763, 2021.
[21] H. Röger and R. Mayer, "A comprehensive survey on parallelization and elasticity in stream processing," *ACM Comput. Surv.*, vol. 52, no. 2, pp. 1–37, May 2019.
[22] M. D. da Assunção, A. da S. Veith, and R. Buyya, "Distributed data stream processing and edge computing: A survey on resource elasticity and future directions," *J. Netw. Comput. Appl.*, vol. 103, pp. 1–17, Feb. 2018.
[23] Q.-C. To, J. Soto, and V. Markl, "A survey of state management in big data processing systems," *Int. J. Very Large Data Bases*, vol. 27, no. 6, pp. 847–872, Dec. 2018.
[24] P. F. Silvestre, M. Fragkoulis, D. Spinellis, and A. Katsifodimos, "Clonos: Consistent causal recovery for highly-available streaming dataflows," in *Proc. Int. Conf. Manage. Data*, Jun. 2021, pp. 1637–1650.
[25] L. Xu, T. Guo, W. Dou, W. Wang, and J. Wei, "An experimental evaluation of garbage collectors on big data applications," in *Proc. 45th Int. Conf. Very Large Data Bases (VLDB)*, 2019, pp. 1–14.
[26] D. Le Quoc, R. Chen, P. Bhatotia, C. Fetze, V. Hilt, and T. Strufe, "Approximate stream analytics in apache flink and apache spark streaming," 2017, *arXiv:1709.02946*. [Online]. Available: http://arxiv.org/abs/1709.02946
[27] M. Petrov, N. Butakov, D. Nasonov, and M. Melnik, "Adaptive performance model for dynamic scaling apache spark streaming," *Procedia Comput. Sci.*, vol. 136, pp. 109–117, Jan. 2018.
[28] (2021). *Apache Flink: Stateful Computations Over Data Streams*. Accessed: Apr. 23, 2021. [Online]. Available: https://flink.apache.org/
[29] (2021). *Apache Storm*. Accessed: Apr. 23, 2021. [Online]. Available: https://storm.apache.org/
[30] (2021). *Apache Samza: A Distributed Stream Processing Framework*. Accessed: Apr. 23, 2021. [Online]. Available: http://samza.apache.org/
[31] (2021). *Apache Kafka: Kafka Streams*. Accessed: Apr. 23, 2021. [Online]. Available: https://kafka.apache.org/documentation/streams/
[32] (2021). *Apache Kafka*. Accessed: Apr. 23, 2021. [Online]. Available: https://kafka.apache.org/
[33] J. Kreps, N. Narkhede, and J. Rao, "Kafka: A distributed messaging system for log processing," in *Proc. NetDB*, 2011, pp. 1–7.
[34] A. J. Awan, M. Brorsson, V. Vlassov, and E. Ayguade, "How data volume affects spark based data analytics on a scale-up server," in *Proc. BPOE*. Cham, Switzerland: Springer, 2015, pp. 81–92.
[35] A. Trivedi, P. Stuedi, J. Pfefferle, R. Stoica, B. Metzler, I. Koltsidas, and N. Ioannou, "On the [ir] relevance of network performance for data processing," in *Proc. 8th USENIX Workshop Hot Topics Cloud Comput. (HotCloud)*, 2016, pp. 1–6.
[36] A. J. Awan, M. Brorsson, V. Vlassov, and E. Ayguade, "Architectural impact on performance of in-memory data analytics: Apache spark case study," 2016, *arXiv:1604.08484*. [Online]. Available: http://arxiv.org/abs/1604.08484
[37] O.-C. Marcu, A. Costan, G. Antoniu, and M. S. Pérez-Hernández, "Spark versus flink: Understanding performance in big data analytics frameworks," in *Proc. IEEE Int. Conf. Cluster Comput. (CLUSTER)*, Sep. 2016, pp. 433–442.
[38] J. Veiga, R. R. Expósito, X. C. Pardo, G. L. Taboada, and J. Tourifio, "Performance evaluation of big data frameworks for large-scale data analytics," in *Proc. IEEE Int. Conf. Big Data (Big Data)*, Dec. 2016, pp. 424–431.
[39] D. Richins, T. Ahmed, R. Clapp, and V. J. Reddi, "Amdahl's law in big data analytics: Alive and kicking in TPCx-BB (BigBench)," in *Proc. IEEE Int. Symp. High Perform. Comput. Archit. (HPCA)*, Feb. 2018, pp. 630–642.
[40] Z. Li, F. Duan, and H. Che, "A unified scaling model in the era of big data analytics," in *Proc. 3rd Int. Conf. High Perform. Compilation, Comput. Commun.*, Mar. 2019, pp. 67–77.
[41] Y. Amannejad, S. Shah, D. Krishnamurthy, and M. Wang, "Fast and lightweight execution time predictions for spark applications," in *Proc. IEEE 12th Int. Conf. Cloud Comput. (CLOUD)*, Jul. 2019, pp. 493–495.
[42] (2021). *NDW: Nationale Databank Wegverkeersgegevens*. Accessed: Apr. 5, 2021. [Online]. Available: http://www.ndw.nu/
[43] G. van Dongen, B. Steurtewagen, and D. Van den Poel, "Latency measurement of fine-grained operations in benchmarking distributed stream processing frameworks," in *Proc. IEEE Int. Congr. Big Data (BigData Congress)*, Jul. 2018, pp. 247–250.
[44] (2021). *DC/OS*. Accessed: Apr. 23, 2021. [Online]. Available: https://dcos.io/
[45] (2021). *Docker*. Accessed: Apr. 23, 2021. [Online]. Available: https://www.docker.com/
[46] (2021). *Cadvisor*. Accessed: Apr. 23, 2021. [Online]. Available: https://github.com/google/cadvisor
[47] (2021). *Hadoop: HDFS Architecture Guide*. Accessed: Apr. 23, 2021. [Online]. Available: https://hadoop.apache.org/docs/r1.2.1/hdfs_design.html
[48] N. Poggi, A. Montero, and D. Carrera, "Characterizing bigbench queries, hive, and spark in multi-cloud environments," in *Proc. Technol. Conf. Perform. Eval. Benchmarking*. Cham, Switzerland: Springer, 2017, pp. 55–74.
[49] T. Ivanov and M.-G. Beer, "Performance evaluation of spark SQL using bigbench," in *Big Data Benchmarking*. Cham, Switzerland: Springer, 2015, pp. 96–116.
[50] K. Ousterhout, R. Rasti, S. Ratnasamy, S. Shenker, and B.-G. Chun, "Making sense of performance in data analytics frameworks," in *Proc. 12th USENIX Symp. Netw. Syst. Design Implement. (NSDI)*, 2015, pp. 293–307.
[51] S. Dong, M. Callaghan, L. Galanis, D. Borthakur, T. Savor, and A. M. Strum, "Optimizing space amplification in RocksDB," in *Proc. CIDR*, vol. 3, 2017, p. 3.
[52] (2021). *Apache Spark: Spark Streaming Programming Guide*. Accessed: Apr. 25, 2021. [Online]. Available: https://spark.apache.org/docs/latest/streaming-programming-guide.html
[53] M. J. Sax, G. Wang, M. Weidlich, and J.-C. Freytag, "Streams and tables: Two sides of the same coin," in *Proc. Int. Workshop Real-Time Bus. Intell. Anal.*, Aug. 2018, pp. 1–10.
[54] D. Wang and J. Huang. (2015). *Tuning Java Garbage Collection for Apache Spark Applications*. Accessed: Aug. 5, 2021. [Online]. Available: https://databricks.com/blog/2015/05/28/tuning-java-garbage-collectionfor-spark-applications.html
[55] (2015). *Juggling With Bits and Bytes*. Accessed: Jun. 14, 2021. [Online]. Available: https://flink.apache.org/news/2015/05/11/Juggling-with-Bits-and-Bytes.html

[56] P. Carbone, A. Katsifodimos, S. Ewen, V. Markl, S. Haridi, and A. K. Tzoumas, "Apache flink: Stream and batch processing in a single engine," *Bull. IEEE Comput. Soc. Tech. Committee Data Eng.*, vol. 36, no. 4, pp. 1–11, 2015.

[57] Y. Byzek. *Optimizing Your Apache Kafka Deployment*. Accessed: Aug. 5, 2021. [Online]. Available: https://www.confluent.io/wp-content/uploads/Optimizing-Your-Apache-Kafka-Deployment-1.pdf

[58] (2021). *Apache Spark: Lightning-Fast Unified Analytics Enginge*. Accessed: Apr. 5, 2021. [Online]. Available: https://spark.apache.org/

[59] M. Zaharia, T. Das, H. Li, T. Hunter, S. Shenker, and I. Stoica, "Discretized streams: Fault-tolerant streaming computation at scale," in *Proc. 24th ACM Symp. Oper. Syst. Princ.*, Nov. 2013, pp. 423–438.

[60] (2021). *Apache Spark: Structured Streaming Programming Guide*. Accessed: Apr. 25, 2021. [Online]. Available: https://spark.apache.org/docs/latest/structured-streaming-programming-guide.html

[61] T. Das, Y. Zhong, I. Stoica, and S. Shenker, "Adaptive stream processing using dynamic batch sizing," in *Proc. ACM Symp. Cloud Comput.*, Nov. 2014, pp. 1–13.

[62] L. Duboc, D. Rosenblum, and T. Wicks, "A framework for characterization and analysis of software system scalability," in *Proc. 6th Joint Meeting Eur. Softw. Eng. Conf. ACM SIGSOFT Symp. Found. Softw. Eng. (ESEC-FSE)*, 2007, pp. 375–384.

[63] G. M. Amdahl, "Validity of the single processor approach to achieving large scale computing capabilities," in *Proc. Spring Joint Comput. Conf.-AFIPS*, 1967, pp. 483–485.

[64] M. A. H. Hassan and M. Bamha, "Towards scalability and data skew handling in GroupBy-joins using MapReduce model," *Procedia Comput. Sci.*, vol. 51, pp. 70–79, Jan. 2015.

[65] (2015). *Project Tungsten: Bringing Apache Spark Closer to Bare Metal*. Accessed: Jun. 14, 2021. [Online]. Available: https://databricks.com/blog/2015/04/28/project-tungsten-bringing-spark-closer-to-bare-metal.html

[66] (2020). *Memory Management Improvements With Apache Flink 1.10*. Accessed: Jun. 14, 2021. [Online]. Available: https://flink.apache.org/news/2020/04/21/memory-management-improvements-flink-1.10.html

[67] K. Nguyen, L. Fang, G. Xu, B. Demsky, S. Lu, S. Alamian, and A. O. Mutlu, "Yak: A high-performance big-data-friendly garbage collector," in *Proc. 12th USENIX Symp. Oper. Syst. Design Implement. (OSDI)*, 2016, pp. 349–365.

[68] L. Gidra, G. Thomas, J. Sopena, M. Shapiro, and N. Nguyen, "NumaGiC: A garbage collector for big data on big NUMA machines," *ACM SIGPLAN Notices*, vol. 50, no. 4, pp. 661–673, May 2015.

[69] Y. Bu, V. Borkar, G. Xu, and M. J. Carey, "A bloat-aware design for big data applications," in *Proc. Int. Symp. Int. Symp. Memory Manage. (ISMM)*, 2013, pp. 119–130.

[70] L. Lu, X. Shi, Y. Zhou, X. Zhang, H. Jin, C. Pei, L. He, and Y. Geng, "Lifetime-based memory management for distributed data processing systems," 2016, *arXiv:1602.01959*. [Online]. Available: http://arxiv.org/abs/1602.01959

[71] X. Shi, Z. Ke, Y. Zhou, H. Jin, L. Lu, X. Zhang, L. He, Z. Hu, and F. Wang, "Deca: A garbage collection optimizer for in-memory data processing," *ACM Trans. Comput. Syst.*, vol. 36, no. 1, pp. 1–47, Mar. 2019.

[72] R. Bruno and P. Ferreira, "A study on garbage collection algorithms for big data environments," *ACM Comput. Surv.*, vol. 51, no. 1, pp. 1–35, Apr. 2018.

[73] (2021). *Kubernetes: Horizontal Pod Autoscaler*. Accessed: Mar. 28, 2021. [Online]. Available: https://kubernetes.io/docs/tasks/run-application/horizontal-pod-autoscale/

[74] T. Rohrmann. (2021). *Flip-159: Reactive Mode*. Accessed: Mar. 28, 2021. [Online]. Available: https://cwiki.apache.org/confluence/display/FLINK/FLIP-159%3A+Reactive+Mode

[75] T. Heinze, V. Pappalardo, Z. Jerzak, and C. Fetzer, "Auto-scaling techniques for elastic data stream processing," in *Proc. IEEE 30th Int. Conf. Data Eng. Workshops*, Mar. 2014, pp. 296–302.

[76] K. G. S. Madsen, Y. Zhou, and J. Cao, "Integrative dynamic reconfiguration in a parallel stream processing engine," in *Proc. IEEE 33rd Int. Conf. Data Eng. (ICDE)*, Apr. 2017, pp. 227–230.

[77] T. M. Ahmed, F. H. Zulkernine, and J. R. Cordy, "Proactive auto-scaling of resources for stream processing engines in the cloud," in *Proc. CASCON*, 2016, pp. 226–231.

[78] M. M. Belkhiria and C. Tedeschi, "A fully decentralized autoscaling algorithm for stream processing applications," in *Proc. Eur. Conf. Parallel Process.* Cham, Switzerland: Springer, 2019, pp. 42–53.

[79] R. P. Singh, B. Kumarasubramanian, P. Maheshwari, and S. Shetty, "Auto-sizing for stream processing applications at linkedin," in *Proc. 12th USENIX Workshop Hot Topics Cloud Comput. (HotCloud)*, 2020, pp. 1–8.

[80] L. Baresi and G. Quattrocchi, "Towards vertically scalable spark applications," in *Proc. Eur. Conf. Parallel Process.* Cham, Switzerland: Springer, 2018, pp. 106–118.

[81] T. Z. Fu, J. Ding, R. T. Ma, M. Winslett, Y. Yang, and A. Zhang, "DRS: Auto-scaling for real-time stream analytics," *IEEE/ACM Trans. Netw.*, vol. 25, no. 6, pp. 3338–3352, Dec. 2017.

[82] R. K. Kombi, N. Lumineau, and P. Lamarre, "A preventive auto-parallelization approach for elastic stream processing," in *Proc. IEEE 37th Int. Conf. Distrib. Comput. Syst. (ICDCS)*, Jun. 2017, pp. 1532–1542.

[83] N. Hidalgo, D. Wladdimiro, and E. Rosas, "Self-adaptive processing graph with operator fission for elastic stream processing," *J. Syst. Softw.*, vol. 127, pp. 205–216, May 2017.

[84] L. R. Moore, K. Bean, and T. Ellahi, "Transforming reactive auto-scaling into proactive auto-scaling," in *Proc. 3rd Int. Workshop Cloud Data Platforms (CloudDP)*, 2013, pp. 7–12.

GISELLE VAN DONGEN (Member, IEEE) is currently a Ph.D. Researcher with Ghent University, teaching and benchmarking real-time distributed processing systems, such as spark streaming, flink, and kafka streams. She is also a Lead Data Scientist with Klarrio specializing in real-time data analysis, processing, and visualization.

DIRK VAN DEN POEL (Senior Member, IEEE) is currently a Senior Full Professor of data analytics/big data with Ghent University, Belgium. He teaches courses, such as big data, databases, social media and web analytics, analytical customer relationship management, advanced predictive analytics, and predictive and prescriptive analytics. He co-founded the Advanced Master of Science in Marketing Analysis (now renamed to Master of Science in Data Science for Business), the first (predictive) analytics master program in the world as well as the Master of Science in Statistical Data Analysis and the Master of Science in Business Engineering/Data Analytics.