Evaluation of Stream Processing Frameworks

Giselle van Dongen, *Member, IEEE* and Dirk Van den Poel, *Senior Member, IEEE*

Abstract—The increasing need for real-time insights in data sparked the development of multiple stream processing frameworks. Several benchmarking studies were conducted in an effort to form guidelines for identifying the most appropriate framework for a use case. In this work, we extend this research and present the results gathered. In addition to Spark Streaming and Flink, we also include the emerging frameworks Structured Streaming and Kafka Streams. We define four workloads with custom parameter tuning. Each of these is optimized for a certain metric or for measuring performance under specific scenarios such as bursty workloads. We analyze the relationship between latency, throughput and resource consumption and we measure the performance impact of adding different common operations to the pipeline. To ensure correct latency measurements, we use a single Kafka broker. Our results show that the latency disadvantages of using a micro-batch system are most apparent for stateless operations. With more complex pipelines, customized implementations can give event-driven frameworks a large latency advantage. Due to its micro-batch architecture, Structured Streaming can handle very high throughput at the cost of high latency. Under tight latency SLAs, Flink sustains the highest throughput. Additionally, Flink shows the least performance degradation when confronted with periodic bursts of data. When a burst of data needs to be processed right after startup, however, micro-batch systems catch up faster while event-driven systems output the first events sooner.

Index Terms—Apache Spark, Structured Streaming, Apache Flink, Apache Kafka, Kafka Streams, Distributed Computing, Stream Processing Frameworks, Benchmarking, Big Data

1 Introduction

In response to the increasing need for fast, reliable and accurate answers to data questions, many stream processing frameworks were developed. Each use case, however, has its own performance requirements and data characteristics. Several benchmarks were conducted in an effort to shed light on which frameworks perform best in which scenario, e.g. [1], [2], [3], [4], [5]. Previous work often benchmarked Flink and Spark Streaming. We extend this work by comparing later releases of these frameworks with two emerging frameworks being Structured Streaming, which has shown promising results [6], and Kafka Streams, which has been adopted by large IT companies such as Zalando and Pinterest. We take a two-pronged approach of comparing stream processing frameworks on different workloads and scenarios. In the first place, we assess the performance of different frameworks on similar operations. We look at relationships between metrics such as latency, throughput and resource utilization for different processing scenarios. The parameter configurations of a framework have a large influence on the performance in a certain processing scenario. Therefore, we tune parameters per workload. Additionally, we evaluate the performance impact of adding different, complex operations to the workflow of each framework. Understanding the impact of an operation on the performance of a pipeline, is a valuable insight in the design phase of the processing flow. By combining these two approaches, we can form a more nuanced evaluation of which framework is most suitable for which use case. In short, we make the following contributions:

1) Open-source benchmark and extensive analysis of emerging frameworks, Structured Streaming and Kafka Streams, besides newer releases of Flink and Spark Streaming.
2) Accurate latency measurement of pipelines of different complexities for all frameworks by capturing time on a single Kafka broker.
3) Thorough analysis of relationships between latency, throughput and resource consumption for each of the frameworks and under different processing pipelines, workloads and throughput levels.
4) Dedicated parameter tuning per workload for more nuanced performance evaluations with detailed reasoning behind each of the settings.
5) Inclusion of built-in and customized stateful operators.
6) Guidelines for choosing the right tool for a use case by taking into account inter-framework differences, different pipeline complexities, implementation flexibility and data characteristics.

The code for this benchmark can be found at https://github.com/Klarrio/open-stream-processing-benchmark. The rest of this paper is organized as follows. The next section gives an overview of the related work that formed a basis for this study. In Section 3 we describe the setup that was used to conduct this benchmark. Section 4 dives deeper into the configurations used for the different frameworks. A discussion of the results follows in Section 5, followed by general conclusions. Finally, we outline some of the limitations and opportunities for further research. Additional information on how this benchmark was conducted can be found in the Supplemental File.

• *All authors are with Ghent University. E-mail: giselle.vandongen@klarrio.com, Giselle.vanDongen@ugent.be, Dirk.VandenPoel@ugent.be*

2 Related Work

Several benchmarking studies have been conducted over the last years. In this section, we outline the most important differences. A tabular overview has been given in Section 1 of the Supplemental File. Most of these benchmarks were implemented on one or a combination of the following frameworks: Spark Streaming, Storm and Flink. In [4] and [5], Spark Streaming outperforms Storm in peak throughput measurements and resource utilization but Storm attains lower latencies. The Yahoo benchmark [3] confirmed that under high throughput Storm struggles. Karakaya et al. [2] extended the Yahoo Benchmark and found that Flink 0.10.1 outperforms Spark Streaming 1.5.0 and Storm 0.10.0 for data-intensive applications. However, in applications where high latency is acceptable, Spark outperforms Flink. Karimov et al. [1] confirmed that the general performance of Spark Streaming 2.0.1 and Flink 1.1.3 is better than that of Storm. In this work, we include the latest versions of Spark Streaming 2.4.1 and Flink 1.9.1 due to their superior results. Additionally, we include Spark's new Structured Streaming 2.4.1 API and Kafka Streams 2.1.0, which have not been thoroughly benchmarked before. Micro-batch systems as well as event-driven frameworks are represented in this benchmark. Both Spark frameworks are micro-batch systems [7], meaning that they buffer events on the receiver-side and process them in small batches. Kafka Streams and Flink work event-driven and only apply buffering when events need to be send over the network.

Previous work mainly focused on benchmarking end-to-end applications in areas such as ETL, descriptive statistics and IO, e.g. [2], [3], [8], [9]. Some work has been conducted on benchmarking single operations such as parsing [8], filtering [4], [8], [9] and windowed aggregations [1]. The metrics for these single operations were, however, recorded as if it were end-to-end applications by including ingesting, buffering, networking, serialization, etc. In this work, we also measure the impact of adding a certain operation to a pipeline. We ensured that each of these operations constitutes a seperate stage in the DAG of the job. We based the design of our pipeline on the Yahoo benchmark [3], which has also been adopted by [2]. Besides a tumbling window, we also include a sliding window for the latency measurement workload. For these types of stateful transformations, some frameworks offer more flexibility via low-level APIs, custom triggers and operator access to managed state. In this work we implemented the tumbling and sliding window with built-in as well as low-level customized operators for Flink, Kafka Streams and Structured Streaming to investigate the advantages of using this flexibility.

Two of the most important performance indicators in stream processing systems are latency and throughput, which have been included in most previous benchmarks, e.g. [1], [3], [8], [9]. We use the same methodology to measure latency of single operations as in the initial proposal for this study [10]. The ability of a framework to process events faster and in bigger quantities with a similar setup, leads to cost reductions and a greater ability to overcome bursts of data. Most research, however, merely analyses the maximum throughput that can be sustained for a prolonged period of time, e.g. [1]. In this benchmark, we analyze throughput in two additional scenarios. Firstly, we investigate the behavior of the framework under a bursty workload. Secondly, we monitor the behavior of the framework when it needs to process a big burst of data right after startup. Some papers already studied the behavior of stream processing frameworks under bursty data. Shukla and Simmhan [8], and later RIoTBench [9], studied the behavior of Storm under bi-modal data rates with peaks in mornings and evenings, while Karimov et al. [1] studied the effects of a fluctuating workload for Spark, Flink and Storm. Spark and Flink showed similar behavior in the case of windowed aggregations, while Flink handled windowed joins better than Spark. Storm was the most susceptible to bottlenecks due to data bursts. Most other literature studied constant rate workloads under different data scales, e.g. [2], [3], [4], [5]. To be able to generate these different loads, we use temporal and spatial scaling similar to [8] and [9].

StreamBench [4] and Qian et al. [5] studied the effects of a failing node on throughput and latency. Their results indicated no significant impact for Spark Streaming while Storm experienced a large performance decrease. In this work, in contrast, we investigate the behavior of processing a big burst of data at startup, mimicking the behavior triggered when the job would be down for a period of time.

This benchmark uses Apache Kafka [11] as distributed publish-subscribe messaging system to decouple input generators from processors. In the Kafka cluster, streams of data are made available on partitioned topics. The majority of past benchmarking literature uses Apache Kafka for this purpose as this is representative of real-world usage, e.g. [2], [3], [8], [9], [10]. Furthermore, frameworks, e.g. [6], [12], require a replayable, durable data source such as Kafka to give exactly-once processing guarantees.

Finally, the configuration parameters of a framework have large effects on latency and throughput. Previous research often did not sufficiently document the configuration settings or used default values for all workloads. In this work, framework parameters are tuned separately for each workload to get a more accurate view of the performance of a framework on a certain metric or in a certain scenario. A thorough discussion follows in Section 4, supported by extra material in the Supplemental File.

3 Benchmark Design

In this section, we describe the benchmark setup. First, we introduce the input data source. Next, we elaborate on the operations that were included and the metrics that are monitored. Finally, we describe the different workloads and the architecture on which the processing jobs are run.

3.1 Data

We perform the benchmark on data from the IoT domain because of its increasing popularity and its increasing amount of use cases. We use traffic sensor data from the Netherlands provided by NDW (Nationale Databank Wegverkeersgegevens). This data contains measurements of the number of cars (flow) and their average speed at around 60,000 measurement points in the Netherlands and this for every minute of every lane of the road. The data

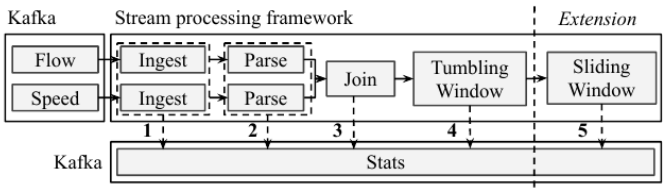


Fig. 1. Processing flow based on [10]

publisher publishes a subset of this data set as a constant rate data stream on the Kafka input topics. For periodic burst workloads, every ten seconds an enlarged batch is published. The volume of data can be inflated by a configurable factor by using spatial scaling, similar to [9]. By changing the characteristics of the data, we mimic different datasets, thereby generalizing the benchmark to other use cases.

3.2 Processing Pipeline

A stream processing benchmark should be able to test the impact of different types of operations. Therefore, we run each workload for different pipeline complexities. We do this by using an extensible processing pipeline (Figure 1) with operations similar to [3] and [10].

1) Ingest: Reading data from the Kafka flow and speed topics: no transformations are done on the data.
2) Parse: Parsing the JSON flow and speed messages.
3) Stream-stream join: inner join of flow and speed messages with the same timestamp, measurement point and lane.
4) Tumbling window: Adding up the number of cars that passed by in the last second and averaging their speed for all the lanes of the road.
5) Extension: Sliding window: Only executed for the latency measurement workload. Computes the relative change in flow and speed for each measurement location over the last two and three seconds. The window length is three seconds and the slide interval is one second.

Initially, we execute the pipeline up to and including the ingest stage. We use the performance of this stage as a baseline. In the second run, we add the parse stage to the pipeline. This mimics ETL jobs with simple data transformations. Afterwards we add a join operation, common in data enrichment scenarios. Finally, we add window stages and customized stateful operations for testing more complex analytics capabilities. Intermediate stages do not publish their outputs to Kafka.

All stateful operations (joining and windowing) are done on event time. For this, we use the Kafka log append timestamp of the input observation. This is possible since the input stream is never out of order. Spark Streaming does not offer event time processing characteristics, therefore some extra logic is required to handle windows accurately in the case of bursty data.

Event-driven frameworks do not apply buffering on the receiver side and can therefore, have significant latency advantages. This advantage disappears when built-in stateful operators such as tumbling windows are used. Since a flexible API is an important asset when optimizing pipeline performance, we included multiple implementations of the stateful operators. For the joining stage of the processing pipeline, we use the most appropriate join semantic available in the framework. For Flink and Kafka Streams, we use an inner interval join [13]. This type of join permits joining events with timestamps that lie in a relative time interval to each other, which is precisely what we want to do. This type of join has a much lower latency than the tumbling window implementation that is typical for micro-batch systems since it can output events directly after receiving the entire pair for the join and it doesn't have to wait for the tumbling of the window.

The Flink API offers flexibility for stateful operations such as the ability to use different state backends [14], to define custom window triggers [12] or to use low-level as well as high-level APIs to do stateful operations [15]. We implement the tumbling window stage with the default event time trigger as well as with a custom trigger that triggers computation when enough data has arrived to do the computation. Furthermore, the sliding window stage was implemented with a built-in sliding window, as well as with the low-level processor API which gives direct access to managed keyed state and timers [15]. Instead of using a window buffer to compute the change in speed and flow over the last seconds, we manage our own list state to do the computation and generate output as soon as an observation enters the processor function. This has the effect that each event can be processed instantly.

Kafka Streams offers two levels of APIs: a high-level DSL (Domain Specific Language) and a low-level Streams Processor API. In this benchmark, we implemented the pipeline with the Kafka Streams DSL for the stateless stages. For the stateful stages we studied two implementations: one using the DSL windowing functionality and the other using the low-level processor API. The processor API allows us to interact with state stores and build customized processing logic. To implement the aggregation and relative change phases, we call the processor API from the DSL by supplying a transformer as described in the documentation [16]. For both stages, we use a persistent key value store backed by RocksDB to store state. RocksDB is the default state backend in Kafka Streams. By using customized low-level implementations we can make sure events are sent out as soon as possible, reaching lower latencies.

Finally, we also include an alternative implementation for Structured Streaming. When we use built-in aggregations such as tumbling window, we experienced issues with the propagation of watermarks leading to very high latencies and making it impossible to chain aggregations, as discussed in Section 5.1. Therefore, we also implemented these stages with mapping functions with custom state management. In this approach, we have fine-grained control over state and the publishing of output so we do not rely on the propagation of watermarks.

The operations discussed in this section cover the main building blocks of stream processing pipelines for most use cases, as can be inferred from the documentation of these frameworks [14], [16], [17], [18]: basic operations, joining,

windowing and processing functions. We believe that by including these operations, we can provide a general performance assessment of these frameworks.

3.3 Metrics

We collect four metrics for every run: the latency of each message, the throughput per second and the memory and CPU utilization of the workers. For each run, except for the single burst workload, we ignore the first five minutes of data since these initial minutes often show increased latency and CPU due to the start up.

3.3.1 Latency

Latency is the amount of time the processing framework requires to generate an output message from one or multiple input messages. This can be difficult to measure since distributed systems do not have a global notion of time. The approach we follow to counter this is described in Section 3.4.1. We unify the way latency is measured across frameworks by subtracting the Kafka log append timestamps of the output and input message. We do not rely on the internal metrics of the frameworks since this does not guarantee us a uniform definition of latency. Furthermore, our definition of latency includes the time the event resided on Kafka before the framework started processing the observation. When backpressure kicks in, this difference between event time latency and processing-time latency can become significant [1]. In the case where multiple input messages are required for one output message, we use the timestamp of the latest input message that was required to do the computation, as proposed by [1].

3.3.2 Throughput

We define throughput as the number of messages a framework processes per second. We measure throughput in three scenarios: peak sustainable throughput under constant load, burst throughput at startup and throughput under periodic bursts. Peak sustainable throughput expresses the maximum throughput that a framework can sustain over an extended period of time without becoming unstable. We consider a system to be stable when the latency and queue size on the input buffer do not continuously increase [1] and when average CPU levels are within reasonable bounds, i.e. lower than 80%. Since this benchmark focuses on (near) real-time processing, we do not allow median latencies to go above 10 seconds. In addition, we measure the burst throughput at startup to capture the speed at which a framework can make up for an incurred lag. Finally, we compare input and output throughput for workloads with periodic bursts.

3.3.3 Memory Utilization

We collect the heap memory usage of the JVMs that run the processing jobs by scraping the JMX metrics exposed by the jobs. The effect on CPU usage of enabling JMX is negligible. Heap memory is used, by all frameworks, to store state and do computations. However, memory management is intrinsically different for each framework, as will be evident from the results.

3.3.4 CPU Utilization

Finally, we look at CPU utilization or the ability to evenly spread work across the different slaves as well as the processing power required to do specific operations. We consider average CPU levels above 80% as unhealthy due to limited resources that are available to the JVM to maintain the running processes when unforeseen bursts of data occur. We monitor CPU at the container level using cAdvisor [19].

3.4 Workloads

Stream processing frameworks need to be able to handle different data characteristics and processing scenarios: bursty workloads, constant rate workloads with different throughput levels, catching up after downtime. We define specific workloads for each of these scenarios. Furthermore, we define separate optimized workloads for measuring latency and throughput since they influence each other, e.g. via buffer sizes and timeouts [12].

3.4.1 Workload for latency measurement

The first workload is the one designed for measuring latency in optimal circumstances. Latency is measured by subtracting the Kafka log append time of the output and input message. Kafka is a distributed service and these timestamps are appended by the leading broker of the topic partition when the message arrives. The leading brokers for the input and output partitions can reside on different machines. Time-of-day clocks can demonstrate noticeable time differences across machines. This makes computations with timestamps from different brokers inaccurate. In an experiment with five brokers, we clocked negative latencies of up to 55 ms for up to ten percent of the events for the earlier stages of the pipeline. Network Time Protocol (NTP) can be used to synchronize time-of-day clocks but still has a minimum error of 35 ms [20]. Due to this, time-of-day clocks are not suitable to measure latencies of less than 10 ms. To counter this, we use a single Kafka broker and we compare the timestamps that were attached to the input and output message by that single Kafka broker [10]. This ensures that all time recordings are done on the same machine and therefore, based on the same clock. Log append time, i.e. the wall clock time of the broker, is accurate up to a millisecond level, which is sufficient for our measurements [21].

This workload is used to determine the lowest obtainable latency. To eliminate the influence throughput has on latency, we use a very low throughput to prevent stressing the framework and the single Kafka broker. In the next workload we gradually increase the load to see how latency holds up as throughput grows. Data flows in at a constant rate of 400 messages per second throughout the entire run. This equals a volume of around 135 MB in one run of thirty minutes. We run this workload for different pipeline complexities to be able to see its impact on latency and we add an additional sliding window stage to the pipeline.

3.4.2 Workload for sustainable throughput measurement

Whereas the previous workload focuses on determining the lowest obtainable latency, this workload investigates how this latency evolves when the framework is used for more realistic and considerable loads and to determine the peak

sustainable throughput. Additionally, we use this workload to investigate the behavior of the frameworks under a constant data input rate which is common for use cases such as monitoring systems (e.g. application logs), preventive maintenance workloads or connected devices (e.g. solar panels).

We determine the peak sustainable throughput by running the workload repeatedly for increasing data scales and monitoring latency, throughput and CPU. For latency, we check whether the median latency does not increase monotonically throughout the run and remains below 10 seconds. When the volume of data is higher than the framework can process, latency increases constantly throughout the workload due to processing delays. Furthermore, we also check whether the framework has processed all events by the time the stream ends. In this work, we assume that once the input stream halts, the framework needs to be able to finish processing in less than ten seconds, otherwise it was either lagging behind on the input data stream or batching at intervals higher than 10 seconds. Finally, we monitor whether the average CPU utilization of the framework containers does not exceed 80%.

3.4.3 Workload with burst at startup

The third workload measures the peak burst throughput that a framework can handle right after startup. This mimics the effect of a job trying to catch up with an incurred delay. For this, we start up the data publisher before the job has been started and let it publish around 6000 events per second, which is a throughput level that all frameworks are able to handle but still imposes a considerable load. Five minutes later, we bring up the processing job and start processing the data on the Kafka topics from the earliest offset. This equals around 1 800 000 events or 350 MB, with linearly increasing event times. Once the processing job is running, we reduce the throughput to around 400 messages per second to let event time progress but not put any more load on the framework. We let this processing job run for ten minutes. Afterwards, we evaluate the throughput at which the framework was able to process the initial burst of data.

3.4.4 Workload with periodic bursts

Finally, we want to monitor the ability of the framework to overcome bursts in the input stream. This mimics use cases such as processing data from a group of coordinated sensors or from connected devices that upload bursts of data every few hours or use cases which need to be able to sustain peaks of data such as web log processing. We have a constant stream of a very low volume and every ten seconds there is a large burst of data. Each burst of data contains approximately 38 000 events, which is approximately 7.5 MB. The publishing of one burst takes around 170 to 180 ms. We look at how long latency persists at an inflated level after a large burst and at the effects on CPU and memory utilization.

3.5 Architecture

To make our benchmark architecture mimic a real-world production analytics stack, we use AWS EC2. We set up a CloudFormation stack which runs DC/OS on nine m5.2xlarge instances (one master, nine slaves). Each of these instances has 8 vCPU, 32 GB RAM, a network bandwith of up to 10 Gbps and a dedicated EBS bandwidth of up to 3500 Mbps. Each instance has an EBS volume attached to it with 150 GB of disk space. We use DC/OS as an abstraction layer between the benchmark components and the EC2 instances on which they run. All benchmark components run in Docker containers on DC/OS. To mitigate the potential performance differences between EC2 instances, we run each benchmark run five times on different clusters and select the run with median performance.

As a message broker, we use Apache Kafka. For most runs, the cluster consists of five brokers (2 vCPU, 6GB RAM) which is the same number as the number of workers for each of the frameworks. As described in Section 3.4.1, we use a single Kafka broker (6.5 vCPU, 24 GB RAM) for the latency measurement workload. We use Kafka Manager for managing the cluster and creating the input and output topics, each of these with 20 partitions since each framework cluster has 20 processing cores. Messages are not replicated to avoid creating too much load on the Kafka brokers. The input topics for the speed and flow messages are partitioned on the ID since this is the key for joining and aggregating. Additionally, all brokers are configured to use LogAppendTime for latency measurements. The roundtrip latency of the Kafka cluster is around 1 ms, tested by publishing and directly consuming the message again. We use cAdvisor, backed by InfluxDB, to monitor and store CPU and networking metrics of the framework containers. Concurrently, the processing jobs expose heap usage metrics on an endpoint via JMX, which is then scraped and stored by a JMX collector. An architecture diagram and detailed information on the distribution of resources over all services has been listed in Section 3 of the Supplemental File. Without the framework clusters, the DC/OS cluster has 45% CPU allocation and 33% memory allocation. The framework cluster adds another 28% CPU allocation and 35% memory allocation.

4 Frameworks

In this section, we discuss each of the frameworks and their configurations, as listed in Table 1. We only discuss the configuration parameters for which we do not use the default values. Parameter tuning is done per workload to get a more accurate measurement of performance. Configurations were chosen based on the documentation of the frameworks, expert advice, and an empirical study, which can be further consulted in the Supplemental File in Section 4.

Some settings are equal for all frameworks. For the frameworks that use a master-slave architecture, i.e. Flink and both Spark frameworks, we deploy a standalone cluster with one master and five workers in Docker containers. Kafka Streams runs with five instances. Each framework gets the same amount of resources for their master (2 vCPU, 8GB RAM) and slaves (4 vCPU, 20GB RAM each). Parallelism is set to 20 since we have 20 Kafka partitions and 20 cores per framework.

For event-driven frameworks we use event time as time characteristic. These frameworks use watermarks to handle

TABLE 1
Framework configuration parameters. Specific workload parameters are noted by L (latency workload), ST (sustainable throughput workload), SB (single burst workload) and PB (periodic burst workload).

a. Apache Flink (v1.9.1)

Parameter	Value	Default
JobManager count	1	/
TaskManager count	5	/
JobManager CPU	2	/
JobManager heap / memory	8 GB / 8 GB	/
TaskManager CPU	4	/
TaskManager heap / memory	18 GB / 20 GB	/
Number of task slots	4	1
Default parallelism	20	1
Time characteristic	event time	processing time
State backend	FileSystem	InMemory
Buffer timeout	100 ms (L: 0 ms)	100 ms
Checkpoint interval	10 000 ms	None
Watermark interval	50 ms	/
Out-of-orderness bound	50 ms	/
Object reuse	enabled	disabled

b. Apache Kafka Streams (v2.1.0)

Parameter	Value	Default
Instances count	5	/
Number of threads / CPUs	4	/
Java heap / memory	18 GB / 20 GB	/
Kafka topic parallelism	20	1
Commit interval	1000 ms	30 000 ms
Message timestamp type	LogAppendTime	CreateTime
Linger ms	100 (L: 0)	100
Grace period join & tumbling window sliding window	 5s (PB, ST: 30s) 50 ms	 / /
Retention time	interval + grace	1 day
Max task idle ms	0 (SB: 300 000)	0
Producer compression	lz4 (L: none)	none
Producer batch size	200 KB (L: 16 KB)	16 KB

c. Spark Streaming and Structured Streaming (v2.4.1)

Parameter	Value	Default
Master count	1	/
Worker count	5	/
Master CPUs / memory	2 / 8 GB	/
Worker CPUs / memory	4 / 20 GB	/
Driver cores / heap	2 / 6 GB	/
Executor cores / heap	4 / 17 GB	/
Number of executors	5	/
Default parallelism	20	parallelism of parent or cores
SQL shuffle partitions	5 (str.), 20 (sp.)	200
Serializer	kryo	java
Locality wait	100 ms	3 s
Garbage collector Initiating heap occupancy Parallel GC threads Concurrent GC threads Max GC pause ms	G1GC 35% 4 2 200	Parallel GC 45% / / 200
Micro-batch interval Initial stages Analytics stages Trigger interval (str.)	 200 ms 1 s (ST: 3 s, 5 s) 0	 / / /
Block interval ms	50 (ST: 150, 250 sp.)	200
Watermark ms (str.)	50	/
minBatchesToRetain (str.)	2	100

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

out-of-order data [12]. We choose an out-of-order bound of 50 ms since we do not have out-of-order data that needs to be handled but we do need to take into account the possible time desynchronization between the brokers. Assume that, in the case of a constant rate of data, two events that need to be joined or aggregated arrive at the extreme beginning and end of the interval. If not all brokers have synchronized time and the out-of-orderness bound is set to zero, it is possible that the watermark had already progressed past the log append time of the second event by the time it entered the system. This might happen if one broker is lagging 50 ms behind on another broker in system time. In this case the join will not take place for this pair of events since it will be seen as 50 ms out of order. We assume that the time difference between the brokers is not more than 50 ms and therefore, we choose an out-of-orderness bound of 50 ms. We do not use a watermark larger than 50 ms because this inherently increases the latency [12], [13].

Event-driven frameworks buffer events before they are sent over the network to reduce the load. Buffers are flushed when they are full or when a configurable timeout has passed. This timeout usually has a default value of 100 ms, which is what we use for throughput measurements. For the latency measurement workload, we disable buffering by setting the buffer timeout of Flink and the linger time of Kafka Streams to 0 ms, thereby optimizing for latency. Structured Streaming and Spark Streaming do micro-batching and, therefore, intrinsically buffer events. A more thorough explanation of this is given in the Supplemental File in Sections 4.1.1 and 4.2.1.

4.1 Apache Flink

For Apache Flink [12], the configuration settings are listed in Table 1(a). The five task managers each have 4 CPUs and therefore, four task slots as suggested by [14]. Additionally, they get 20 GB memory of which 18 GB is assigned to the heap and the other 2 GB is left for off-heap allocation to network buffers and the managed memory of the task manager [14].

Three state backends are currently available in Flink: MemoryStateBackend, FileSystemBackend en RocksDBBackend. We use the FileSystemBackend since this is the recommended backend for large state that fits in heap memory. MemoryStateBackend is used for jobs with little state and in development and debugging stages. RocksDB is a state store kept off-heap and is recommended if the state does not fit in the heap memory of the task managers [14]. The FileSystemBackend we use is backed by HDFS and checkpointing is done every ten seconds. Due to Flink's asynchronous checkpointing mechanism, the processing pipeline is not blocked while checkpointing. Flink's mechanism of keeping state reduces the load on the garbage collector, as opposed to the mechanism used by Spark which still heavily relies on JVM memory management, as described in Section 4.4.

We define a watermark interval of 50 ms. This means the current watermark is recomputed every 50 ms. Finally, we enable object reuse since this can significantly increase performance.

4.2 Apache Kafka Streams

In 2016, Apache Kafka released Kafka Streams [13], a client library for processing data residing on a Kafka cluster. Kafka Streams does not make use of a master-slave architecture and does not require a cluster but runs with different threads that rely on the Kafka cluster for data parallelism, coordination and fault tolerance. All Kafka Streams instances share the same consumer group and application id such that they all process a part of the input topic partitions. Each instance will be running on four threads to optimally use all resources. Kafka Streams stores state on Kafka topics and therefore, does not require a HDFS cluster.

When using the DSL, we need to set the grace period and retention times for the window stages. Since each input record leads to an output update record [13] and only the complete output records are kept, the grace period does not introduce additional latency and has been put at a high number of 5s. The retention time is a lower-level property that sets the time events should be retained, e.g. for interactive queries, and has no influence on the latency. We set this to be equal to the slide interval plus grace period.

We do not use message compression for latency measurements. For measuring throughput and resource consumption, we use slightly different parameters, as recommended by [22]. We keep the linger time at the default 100 ms. We use lz4 message compression to reduce the message sizes and load on the network, thereby increasing throughput. We also increase the size of output batches to reduce the network load.

For processing a single burst, we set the maximum task idle time to 300 000 ms. By doing this, Kafka Streams waits an increased amount of time to buffer incoming events on all topics before it starts processing. This prevents the system from running ahead on some topics and subsequently discarding data on other topics if their event time is too far behind.

4.3 Apache Spark: Spark Streaming

Apache Spark [7] is a cluster computing platform that consists of an extended batch API in combination with a library for micro-batch stream processing called Spark Streaming. The job runs with one driver and five executors. On each worker, we allocate 1 GB of the 20 GB to JVM overhead. The executors use 10 % of the remaining memory for off-heap storage, which leaves 17 GB for heap memory. For the driver we allocate 6 GB of heap memory. Checkpoints are stored in HDFS.

When executing the initial stages of the pipeline, i.e. ingesting and parsing, the jobs are configured to have a micro-batch interval of 200 ms. In the initial stages, the selectivity of the data is one on one so a lower batch interval leads to lower latencies. For the analytics stages we set the micro-batch interval to the same length as the interval at which sensors send their data and on which we will do our aggregations, which is one second. For the sustainable throughput workload, we set the batch interval to three and five seconds since this increases the peak throughput [7]. Spark Streaming splits incoming data in partitions based on the block interval. Therefore, it is recommended to choose the block interval to be equal to the batch interval divided by the desired parallelism [17].

For reading from Kafka we use the direct stream approach with PreferConsistent as location strategy, meaning the 20 Kafka partitions will be spread evenly across available executors [17]. Often it is recommended to use three times the number of cores as the number of partitions. This, however, only leads to performance improvement in the case of data skew. When keys are not equally distributed, having more partitions than the number of cores allows more flexibility in spreading work over all executors. In this use case, the data is equally spread over all keys. Hence, using the same number of partitions as the number of cores leads to less overhead and better performance.

In all workloads, Spark Streaming checkpoints at the default interval which is a multiple of the batch interval that is at least 10 seconds [17]. We use the kryo serializer for serialization and register all classes since it is faster and more compact than the default java serializer.

4.4 Apache Spark: Structured Streaming

In 2016, Apache Spark released a new stream processing API called Structured Streaming [6] which enables users to program streaming applications in the DataFrame API. The DataFrame API is the dominant batch API, and Structured Streaming brings Apache Spark one step closer to unifying their batch and streaming APIs. Structured Streaming offers a micro-batch execution mode and a continuous processing mode for event-driven processing. We use the micro-batch approach because the continuous processing mode is still experimental and does not support stateful operations at the time of this writing. Additionally, the micro-batch API does not support chaining of built-in aggregations, therefore, we do not execute the sliding window stage. However, we also include an implementation of the final two stages using map functions with custom state management, since this circumvents the issues with built-in aggregations.

For Structured Streaming, we trigger job queries as fast as possible, which is enabled by setting the trigger interval to 0 ms. Queries run in the default append mode, which means only new rows of the streaming DataFrame will be written to the sink. Due to the architectural design of Structured Streaming, checkpointing is done for each batch [6]. This leads to a very high load on HDFS. To be able to run the jobs at full capacity, a significant size increase of the HDFS cluster was necessary since the checkpointing for every micro-batch led to seconds latency increase for the windowing phases.

The default parallelism parameter arranges parallelism on the RDD level. Structured Streaming, however, makes use of DataFrames. The parallelism of Dataframes after shuffling is set by the SQL shuffle partitions parameter. Since all workloads have small state and since opening connections for storing state is expensive, performance improves significantly when the number of SQL shuffle partitions is reduced to 5. When doing this for Spark Streaming, we did not notice a similar performance improvement. Structured Streaming checkpoints the range of offsets processed, after each trigger. In contrast, the default checkpoint interval of Spark Streaming is never less than 10 seconds.

Due to the heavier use of checkpointing, Structured Streaming benefits more from reducing the amount of SQL shuffle partitions. We use kryo for fast and compact serialization. We use the G1 garbage collector (GC) with the parameters listed in Table 1 to reduce GC pauses. A more thorough explanation of the reasoning behind this can be found in the Supplemental File. Finally, we set the minimum number of batches that has to be retained to two, which further reduces memory consumption and, therefore, GC pauses.

5 Results

In this section, we discuss the results of each of the workloads. First, we discuss the latency and sustainable throughput workloads and afterwards the burst workloads. For all of the results we did multiple runs with similar results and we discarded the first five minutes of the result timeseries to filter out warm-up effects. An overview of the results has been summarized in Table 2 per workload and for each additional decision factor. These decision factors influence the decision for a framework based on job requirements or pipeline complexity. A higher score refers to better performance. The scores do not have an absolute value and should be interpreted relatively to the scores of the other frameworks.

5.1 Workload for latency measurement

This first workload was designed to measure latency accurately, as was described in Section 3.4. The latency distribution for each of the pipeline complexities and frameworks has been given in Figure 2.

By using event-driven processing and no buffering, Flink and Kafka Streams have the lowest latency for ingesting the data, with a median of 0 ms and a p99 of 1-2 ms. When parsing is added to the pipeline, the median latency of both frameworks increases by 1 ms. Flink has a lower p99 latency than Kafka Streams, 2 ms and 7 ms respectively. When we experiment with higher buffer timeouts and linger times, these latencies rise linearly, as documented in the Supplemental File and confirming [12]. The latencies for Spark Streaming and Structured Streaming are considerably higher, as expected for a micro-batch approach. For Spark Streaming we chose a micro-batch interval of 200 ms for these stateless stages, giving us median latencies of around 140 ms to 150 ms and p99 latencies of 316 ms to 350 ms. Structured Streaming uses the approach of processing as fast as possible. As a result, the median latency is slightly lower than for Spark Streaming, with 105 ms for ingesting and 148 ms when parsing is added. We notice, however, much more pronounced tail latencies with events requiring up to 930 ms of processing time. This can be an important factor for jobs under tight SLAs.

When we include the joining stage in the processing pipeline, the differences in latencies between the frameworks grow. For Spark Streaming, the micro-batch interval was raised to one second because this is equal to the join interval. Since processing one batch takes longer, Structured Streaming also forms larger micro-batches. The median latency of Spark Streaming was 650 ms after joining. The micro-batch interval is 1000 ms so the buffering time for an event is on average 500 ms. Since the median latency is 650 ms, we infer that it takes an additional 150 ms to process all events of the batch. This is confirmed by the form of the distribution, which is rather uniform between 170 ms and 1140 ms. Speed and flow observations of one measurement point are sent immediately after each other. The lower latencies come from pairs of events that arrived at the end of the tumbling window while the higher latencies come from pairs of events that arrived at the beginning of the window. Processing a batch of events, therefore, took between 140-170 ms. Furthermore, we see that the variance of the latency increased after adding the join. This is due to

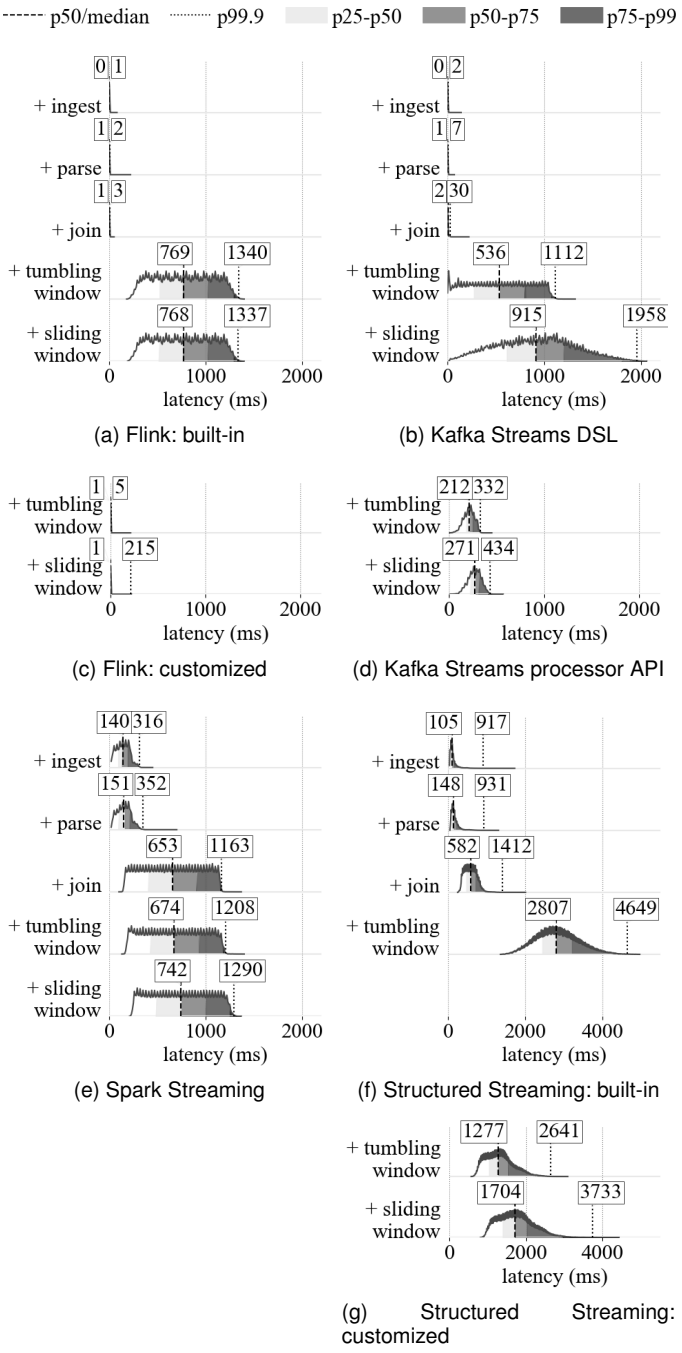


Fig. 2. Latency distribution for all frameworks. The stage until which the pipeline was executed is shown on the y-axis. The chart for Structured Streaming has a different x-axis scale than the other charts.

the fact that the events are joined using a tumbling window. The p99.999 latency is 200 ms higher than the p99 latency, meaning there are some minor outbreaks but that in general latency stays within predictable ranges. The latencies for Structured Streaming suffered from a much longer tail with the p99 above 1400 ms, while the median stays at 582 ms.

For the event-driven processing frameworks, Flink and Kafka Streams, the type of join used is an interval join which can obtain a much lower latency. By joining events that lay within a specified time range from each other, the framework can give an output immediately after both required events arrive and does not wait for an interval to time out. The latency of joining is the lowest for Flink with a median round-trip latency of 1 ms and 99% of the events processed under 3 ms. Kafka Streams reaches median latencies of 2 ms and shows much higher tail latencies with a p99 of 30 ms and a p99.999 of 218 ms. We always compute the latency of the last event that was required to do the join. If we would incorporate the latency of the first event, these latencies would be higher, but we would not be expressing processing time.

The last two stages of the processing pipeline are the windowing stages: a tumbling window, as well as a sliding window are implemented and chained together. At the time of writing, Structured Streaming did not support chaining built-in aggregations. Therefore, we do not execute the built-in sliding window stage for Structured Streaming. We can circumvent this issue by using map functions with custom state management and processing time semantics, as discussed later. The performance of the built-in tumbling window behind the join led to a large performance degradation with p99 latencies of 4649 ms and a median latency of 2807 ms. There are a few main contributors to this 2800 ms median latency. The first one is micro-batching. Despite setting the trigger to 0 ms to achieve processing as fast as possible, we still see micro-batches occurring of around 800-1000 ms with occasional increases due to garbage collection. With a constant stream coming in this means that each event already had an average delay of 400-500 ms before processing has even started. The processing time of the batch in which the event resides then adds another 1000 ms to this time. Once a record has been joined, its propagation to the tumbling window phase is delayed until event time has passed the window end and watermark interval, which means records are buffered for another micro-batch interval. These three factors together add up to a base latency of 2800 ms and an even higher p99 latency. As confirmed by the developers of Structured Streaming in [6], the micro-batch mode of Structured Streaming has been developed to sustain high throughput and should not be used for latency-sensitive use cases. By using customized stateful mapping functionality with processing time semantics, we can improve performance since we do not rely on watermark propagation anymore. As can be seen in Figure 2g, the latency of the tumbling window stage now has a median of 1277 ms. Adding the sliding window stage, leads to a median latency of 1704 ms. We still see large tail latencies for both stages, going up to 3700 ms for the sliding window stage. This can be an issue for use cases with tight latency requirements.

For the older Spark Streaming API, we measure low median latencies of 667 for tumbling window and 765 ms after adding the sliding window. However, we see the 99th percentile latencies increase with complexity. We also notice that the uniform distribution caused by the tumbling window join persists throughout the following stages. The tumbling window as well as sliding window have slide intervals of one second. Therefore, no latency reductions can be obtained by using customized implementations since the latency is directly tight to the fixed micro-batch interval. Spark Streaming becomes a very good contender for these more complex transformations to the built-in high-level APIs of natively faster frameworks Flink and Kafka Streams. For Flink and Kafka Streams we compare two implementations. When using the built-in API, the latency of Kafka Streams increases the least with the addition of the tumbling window, although this behavior is not sustained when the sliding window is added. For all frameworks, we notice growing tail latencies when complexity is added via joins and aggregations. Partly this is due to the checkpointing mechanisms that are required to do stateful stream processing. As state increases, the garbage collection becomes more time consuming and less effective because a larger state needs to be kept and checked at each cycle. Finally, the shuffling and interdependence between the observations to generate the output increases the variance of the latency. Through customizing stateful implementations, we can heavily reduce the latency for some use cases. For Flink, using a custom trigger for the tumbling window reduced the median latency to 1 ms with a p99 of 5 ms. This is much lower than when using the default trigger because the default trigger forces events to be buffered till the end of the window and added an additional delay due to the burst of data that needs to be processed at the end of the interval. By allowing events to be sent out as fast as possible, the latency can be significantly reduced. Similarly, adding a processor function with managed keyed state and timers to do the sliding window keeps the median latency at 1 ms and slightly increases the p99 to 215 ms. This shows that the flexibility of the Flink API enables optimizing the way stateful operations are done by e.g. reducing the time and amount of data that is buffered. For Kafka Streams we also notice large latency reductions (4x) when using the low-level API, however, not as large as those of Flink. Besides, it is important to keep in mind that not all processing flows can be optimized by a customized implementation. In this flow, this was the case because it unnecessarily blocks events until it reaches a window timeout.

We can conclude that event-driven frameworks, such as Flink and Kafka Streams, are preferred when doing simple ETL operations. Built-in windowing operations reduce the difference between micro-batching and event-driven systems, shifting in favor of Spark Streaming. The type of stateful transformation chosen has a large influence on latency. Event-driven frameworks make it possible to do interval joins and stateful operations with custom triggers and state management which can lead to significant latency reductions compared to built-in windowing functionality. Structured Streaming also benefits from a customized implementation since we do not rely on watermarks anymore for output to be send out. When tight latency SLAs are imposed, the longer tail latencies and larger distribution spread for

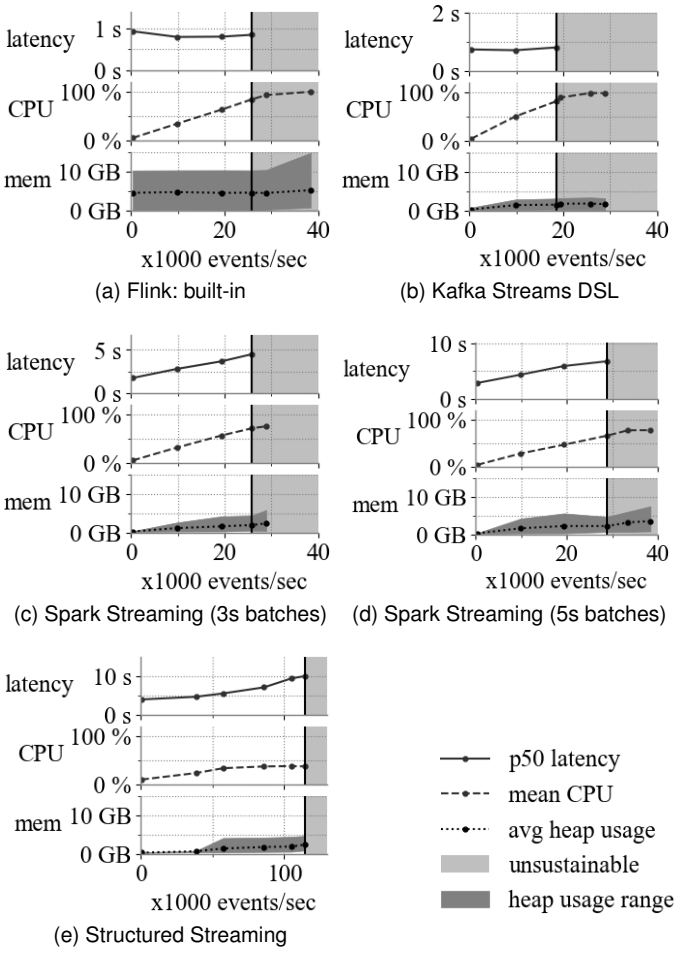


Fig. 3. Sustainable throughput: performance for different throughput levels for the tumbling window stage. Each marker represents a benchmark run for the corresponding throughput level. The light grey zone marks unsustainable levels of throughput. Structured Streaming has different scale on x-axis.

complex pipelines should be kept in mind.

5.2 Workload for sustainable throughput measurement

The sustainable throughput for each of the frameworks is measured for execution of the pipeline up to and including the tumbling window stage. In Figure 3, peak sustainable throughput is denoted by the beginning of the grey zone. In this zone, either mean CPU went above a threshold or processing could not keep up with the input stream. Dot markers represent benchmark runs at corresponding throughput levels. The throughput levels bordering this grey zone are often also at risk of becoming unsustainable since the job does not have any buffer for unforeseen bursts, delays, etc. For unsustainable loads, we show CPU and memory metrics but we do not show latency metrics since increasing delays makes it soar to very high levels of which the median has no interpretative value. For most frameworks, an increase in latency signals throughput levels becoming unsustainable. It is important to note that the latency we show here is computed using a multi-broker Kafka cluster and is, therefore, only accurate to a tenth of a second. Hence, we use this latency merely to compare broad trends. Memory usage fluctuations are shown by the minimum, maximum and average values of the run to avoid cluttering the graph. We take a look at the growth in heap usage and the effectiveness of GC cycles to clean up the heap memory as throughput increases.

For Spark Streaming, we increase the batch interval since this increases the peak sustainable throughput significantly. With a batch interval of one second Spark Streaming was not able to handle a peak sustainable throughput of over 20 000 events per second. With a batch interval of three seconds a sustainable peak throughput of 26 000 events per second is reached. When we increase the batch interval to five seconds, it increases to 30 000 events per second which confirms that sustainable throughput can be increased by increasing the batch interval, but this comes with an inherent latency cost. We notice that for the runs with low load the latency is around 300 ms longer than half of the batch interval, implying that the processing of the batch took approximately 300 ms. As throughput increases, the latency steadily increases as well due to the increased processing time for larger batches. Jobs start incurring delays when the processing time becomes larger than the batch interval. At this point, the latency approaches 1.5 times the batch interval since this includes the buffering at the receiver side. Structured Streaming, on the other hand, uses a dynamic batch interval to improve the data processing speed. When the throughput increases, the batch interval and latencies increase as well. This mechanism allows Structured Streaming to handle a throughput of 115 000 events per second with a CPU utilization of 50%. When the throughput increases any further the batch interval passes the 10 second median latency threshold we imposed, but still remains sustainable. At this point, we are confronted with a trade-off between latency and throughput. As stated in [6], Structured Streaming has been optimized for throughput. For latency-sensitive use cases, they recommend using the continuous processing mode which is still experimental. Since we focus on benchmarking real-time processing systems, we do not allow latency to go above 10 seconds in this workload. Following this definition, Structured Streaming has a peak sustainable throughput of 115 000 events per second while maintaining a median latency under 10 seconds. The pattern of increasing latencies for increasing throughput is not apparent for Flink and Kafka Streams. The main reason is that Flink and Kafka Streams do event-driven processing with very limited buffering on the receiver side. When the constant rate throughput is at such a level that it causes backpressure on the receiver side, the throughput is already at unsustainable levels. Flink can sustain over 30 000 events per second for a buffer timeout of 100 ms. At this point, CPU levels are over 95%. In order to avoid mean CPU utilization above a threshold of 80%, we set the peak sustainable throughput at 26 000 events per second. As an experiment we set the buffer timeout to -1, which means that buffers are only flushed when they are full. However, this did not increase throughput further. This is in agreement with the paper presenting Flink [12], which shows a large throughput increase when increasing the timeout from 0 to 50 ms but a limited throughput increase when increasing from 50 ms to 100 ms. When using the custom trigger implementation, the peak sustainable throughput decreased slightly to 25 000

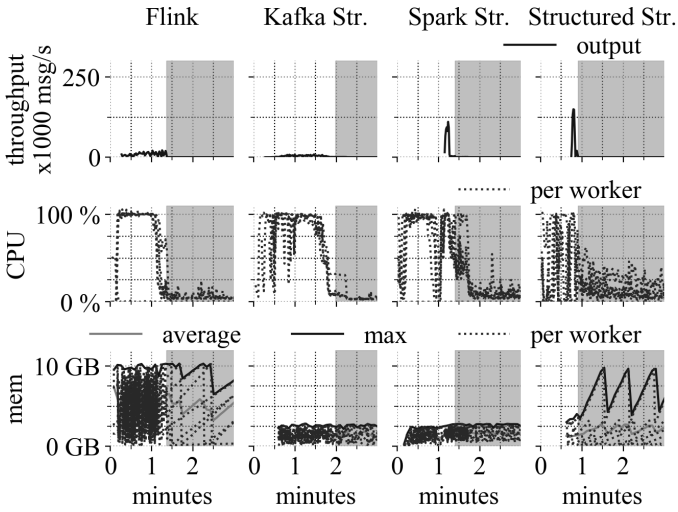


Fig. 4. Single burst workload until tumbling window stage: performance for first three minutes of the run. The grey zone denotes where processing has caught up on the input stream.

events per second, as can be consulted in the Supplemental File. The custom trigger implementation is slightly more CPU intensive than the default implementation and CPU is the bottleneck to increasing throughput further.

Kafka Streams DSL shows relatively constant latencies of around 700 ms for all sustainable throughput levels. At a throughput level of 18 500 events per second, average CPU levels reach 83%. When the throughput increases further, processing starts lagging behind on the input stream and latencies soar. When using the processor API to implement the tumbling window stage, there was a slight increase in peak sustainable throughput to 20 000 events per second, as can be consulted in the Supplemental File.

For all runs, throughput has a linear relationship with CPU usage. We see that the runs with unsustainable throughput levels, i.e. in the grey zone, have average CPU levels of above 80 %. Additionally, for most frameworks, an increase in throughput leads to an increase in memory usage due to the larger input and larger state. At sustainable levels of throughput, Flink is the only framework where we do not notice an increase in average heap memory usage when throughput increases. We see memory ramp up to approximately 9 GB. It then drops back to around 100 MB after it is collected. The maximum heap used only starts increasing after throughput reaches unsustainable levels.

As visualized in Table 2, our results confirm that Structured Streaming can reach higher throughput due to micro-batch execution optimization. Spark Streaming also shows a substantial peak throughput increase when the micro-batch interval is raised. We, however, make the large latency cost that comes with this explicit. For use cases with tighter latency requirements, Flink obtains the best results.

5.3 Workload with burst at startup

We executed the single burst workload for all frameworks and all pipeline complexities and implementations. Since we notice similar behavior for all complexities, we will focus on the results for the tumbling window stage. Additional results have been put in the Supplemental File in Section 5.3.

In Figure 4, the behavior of the framework during the first minutes after startup is shown. The light grey zone shows where the framework caught up to the incoming stream. In the first row of Figure 4, we see the output throughput with clear differences across frameworks that are mainly due to a micro-batch or event-driven approach. Kafka Streams and Flink publish the first output 16 seconds after startup. Structured Streaming takes longer with 44 seconds. Spark Streaming requires more than a minute of processing time before the first output is published. However, these initial batches for Spark Streaming, contain around 80 000 - 110 000 events in six subsequent batches. Structured Streaming shows batch sizes going up to 150 000 events for the first five seconds and manages to catch up the fastest after 53 seconds. Due to its adaptive micro-batch approach it can process large batches of data well and can quickly catch up with the input data, validating the statements made by the developers of Structured Streaming in [6]. Spark Streaming also uses a micro-batch approach, causing a similar large burst in output as observed with Structured Streaming. After 76 seconds most of the data is processed, however batches stay elevated at around 2700 events for another eight seconds afterwards. Flink is able to catch up after 81 seconds by publishing smaller batches over a longer period of time. Kafka Streams requires most time to process all data with 99 seconds and also publishes smaller batches over a longer period of time. To make the built-in window implementation of Kafka Streams process the data in order, the allowed maximum task idle time had to be raised as described in Section 4.2.

During the processing of the burst, all workers of Flink, Kafka Streams and Spark Streaming have 100% CPU utilization. The CPU utilization of Spark Streaming drops immediately before the batches are published onto Kafka. For Structured Streaming, we do not reach 100% CPU for the initial minutes of processing and we notice one worker having increased CPU utilization compared to the other workers of the cluster. Similar to other workloads, Flink uses up to 9GB of the heap while Spark Streaming and Kafka Streams stay around 3 GB.

When executing the workload until other stages or for different implementations, we notice similar patterns as can be consulted in the Supplemental File. The customized implementation finishes faster for Kafka Streams and Structured Streaming and reaches a higher peak throughput. The peak throughput for Structured Streaming goes up to even 229 000 events per second.

In conclusion, micro-batch frameworks take in a big load of data on start-up and take a while to give initial outputs. Event-driven frameworks take in smaller loads of data and start delivering output sooner. Structured Streaming consistently catches up the fastest, followed by Flink, Spark Streaming and finally Kafka Streams.

5.4 Workload with periodic bursts

We ran the periodic burst workload for several pipeline complexities. The results for the tumbling window stage are shown in Figure 5 and for the join stage in Figure 6. Additional results have been put in the Supplemental File in Section 5.4. We zoom in on a one-minute-interval nearing

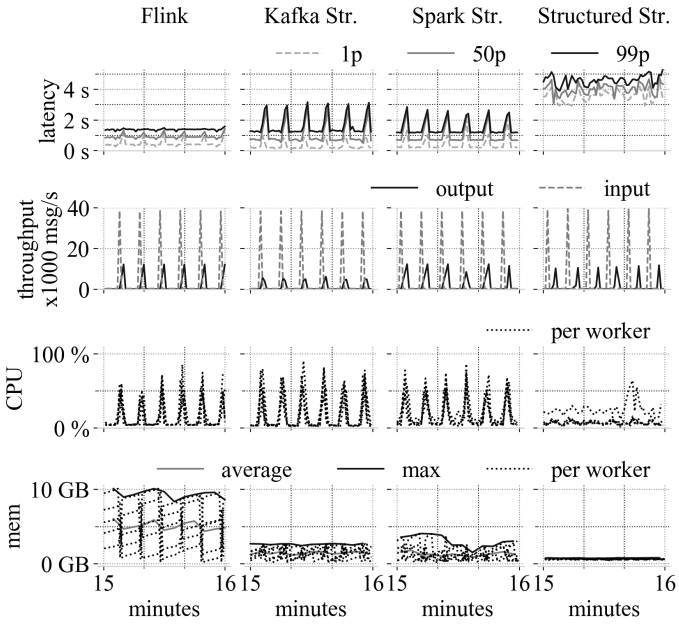


Fig. 5. Periodic burst workload until the tumbling window stage: performance metrics for the last minute of the run

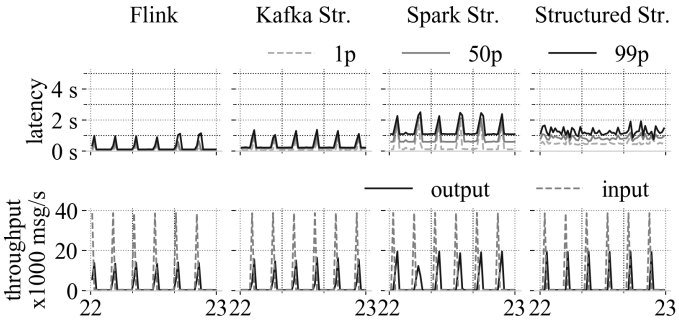


Fig. 6. Periodic burst workload until the join stage: performance metrics for the last minute of the run

the end of our run to get a clearer view of what happens after a burst of data is ingested. Similar to the workload for sustainable throughput, latency measurements are only accurate to tenth of a second.

When we look at the input throughput, we can clearly discriminate the periodic bursts of around 38 000 messages every ten seconds. In between bursts, the input throughput stays at 400 messages per second. For the tumbling window stage, each output event requires on average 3.17 input events. For the join stage, two input events are required. This explains the discrepancy in messages per second between the input and output throughput. The differences in the height of the output throughput rate between frameworks can be explained by the fact that some frameworks ingest the burst at once while others ingest it in several chunks and output the results more gradually.

In Figure 5, the relationships between the different performance metrics reveal some interesting patterns. For Spark Streaming, we see that after a burst of the input throughput, CPU levels rise and are followed by a burst in the output throughput. The latency increases for two to three seconds after the burst, which can be noted by the width of the peak. The median latency during the low load periods is around 700 ms. However, the first seconds after a burst events have an inflated median latency of 1370 ms, then it goes up to 2050 ms after which it falls back to 1350 ms again for the final batch. The main reason for this is the increased processing time required to do all the computations for this burst of data. Besides, the events that come in right after the burst incur a scheduling delay due to this increased processing time and, therefore, have inflated latencies as well.

In Figure 5 we also show the performance of the default trigger implementation for Flink and the DSL implementation for Kafka Streams. We see that the impact of bursts on the latency is the least for Flink with the median latency rising from around 800 ms to 1250 ms and the p99 latency staying fairly constant and under 1500 ms. Due to the lower impact of bursts on the processing time of Flink, the period for which latency remains at higher levels is also shorter. When we execute this workload for earlier stages, such as the parse and join stage (Figure 6), we see a larger effect on the p50 and p99 latency. This can be explained by the fact that the tumbling window applies buffering which dampens latency differences across events. When using custom triggers, this buffering is reduced as can be seen in Figure 7. We now see clear reactions to bursts again and a very low latency in between bursts. Furthermore, the p99 latency after a burst is 1200 ms which is still lower than the almost constant latency of the default trigger implementation. When we use the low-level API of Kafka Streams to implement this stage, the latency of processing a burst is similar to that of Flink and much lower than with the DSL implementation. With the DSL implementation, the median and p99 latency rise considerably after a burst, up to 2800 ms. For the parsing stage, the latency of Kafka Streams is even slightly lower than that of Flink. In contrast, we can see in Figure 5 that the latency of Structured Streaming exhibits a very irregular course with the median fluctuating between 3000 ms and 4500 ms and no clear reactions to bursts. This is due to the delays in watermark propagation that have been explained in Section 5.1. When we use the customized implementation that does not use watermarks (Figure 7), the median latency remains under the two seconds. Structured Streaming can sustain the largest throughput due to its dynamic micro-batch approach and it also benefits from that when processing bursts. For the join stage and the parsing stages, we see lower and less variable latencies and no clear reactions to bursts. Finally, we see that particularly for Kafka Streams and Spark Streaming immediately after a burst, the distribution of the latency becomes much more narrow, with median and p99 latencies almost converging.

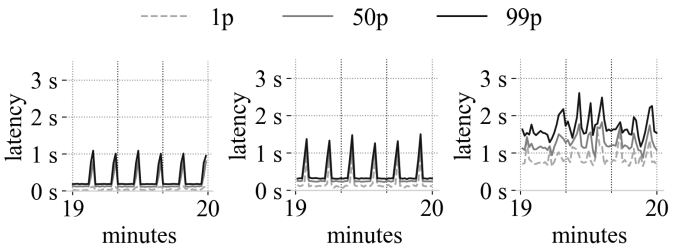
For CPU utilization, we see that for most frameworks all workers are equally utilized when processing bursts, with almost all lines in Figure 5 coinciding. During the processing of a burst we see CPU usage of 60-90% for all workers, while the CPU utilization throughout the rest of the run remains around 2-4%. Structured Streaming shows different behavior with an average load of 6-10% and peaks between 20-40% mainly contributed by one worker. This worker had almost constantly higher CPU usage than the other workers in the cluster.

When looking at the heap usage throughout this run,

TABLE 2
Result overview for Flink (FL), Kafka Streams (K), Spark Streaming (SP) and Structured Streaming (ST). Scores can only be interpreted relatively to the scores of the other frameworks. Higher scores indicate better performance.

Workload	Decision factor	FL	K	SP	ST
Latency	stateless stages	4	4	1	1
	built-in stateful API	4	3	4	1
	customized stateful API	4	4		2
Sustainable throughput	unconstrained latency	2	1	3	4
	constrained latency	4	3	2	1
Single burst	initial output	4	3	1	2
	processing time	2	1	2	4
Periodic burst	stateless stages	3	3	1	4
	built-in stateful API	4	1	3	1
	customized stateful API	4	4		2

we see that Structured Streaming uses the least memory. For Flink we see memory usage ramp up to 10 GB before a GC is triggered. These GC cycles are synchronized with the frequency of the bursts. After a burst has been processed, GC is initiated. For the other frameworks this behavior is not as explicit. The amount of memory that is used is not necessarily important for framework performance. What is important, is that GCs stay effective and that memory frees up well after each GC, which is the case for all frameworks.



(a) Flink: custom trigger (b) Kafka Streams: processor API (c) Structured Streaming: low-level

Fig. 7. Periodic burst workload for customized tumbling window implementations for Flink, Kafka Streams and Structured Streaming: latency metrics for one minute of the run.

As a conclusion, we can state that using custom implementations can not only reduce the latency of the pipeline in general but also the latency of processing bursts. When we use only high-level APIs, Structured Streaming shows much resiliency against bursts for the parsing and joining pipelines and also uses the least resources. Structured Streaming benefits greatly from its micro-batch execution optimizations when processing bursts, confirming [6]. Due to its dynamic micro-batch interval, it reaches much lower latencies than Spark Streaming for these earlier stages. When the pipeline gets more complex Flink suffers the least from processing bursty data, followed by Spark Streaming. Detailed visualizations of the other workloads has been put in the Supplemental File.

6 Conclusion

In this paper we have presented an open-source benchmark implementation and full analysis of two popular stream processing frameworks, Flink and Spark Streaming, and two emerging frameworks, Structured Streaming and Kafka Streams. Four workloads were designed to cover different processing scenarios and ensure correct measurements of all metrics. For each workload, we offer insights on different processing pipelines of increasing complexity and on the parameters that should be tuned for the specific scenario. Our latency workload provides correct measurements by capturing time metrics on a single machine, i.e. Kafka broker. Furthermore, dedicated workloads were defined to measure peak sustainable throughput and the capacity of the frameworks to overcome bursts or catch up on delay. We discuss the metrics latency, throughput and resource consumption and the trade-offs they offer for each workload. By including built-in as well as customized implementations where possible, we aim to show the advantages of having a flexible API at your disposal.

A concise overview of the results is shown in Table 2. For simple stateless operations such as ingest and parse, we conclude that Flink processes with the lowest latency. For these tasks, the newer Structured Streaming API of Spark shows potential with lower median latencies than Spark Streaming. However, it suffers from longer tail latencies making it less predictable to reach SLAs with tight upper bounds. For the joining stage, event-driven frameworks such as Flink and Kafka Streams have an advantage due to the interval join capability. This type of join can output events with a much lower latency, compared to the tumbling window join available in micro-batch frameworks. The latency differences between event-driven and micro-batch systems disappear when the pipeline is extended with built-in tumbling and sliding windows. For these operations, Spark Streaming shows lower tail latencies and outperforms the other frameworks. However, by using the flexibility frameworks offer for these stateful operations through custom triggers and low-level processing capabilities, pipelines can be optimized to reach lower latency.

A second important metric is throughput. Structured Streaming is able to sustain the highest throughput but this comes at significant latency costs. If latency and throughput are both equally important, Flink becomes the most interesting option. When processing a single large burst at startup, Flink outputs the first events the soonest while Structured Streaming finishes processing the entire burst the earliest. For use cases where the input stream exhibits occasional bursts, the least performance impact is perceived when using Flink as the processing framework. When using the default implementation, the p99 latency of Flink stays constant throughout bursts while for other frameworks significant increases are noted. The latency of processing bursts can again be optimized by using low-level APIs.

Finally, the results of this paper can be used as a guideline when choosing the right tool for a processing job by not only focusing on an inter-framework comparison but also highlighting the effects of different pipeline complexities, implementations and data characteristics. When choosing a framework for a use case, the decision should be made based on the most important requirements. If subsecond latency is critical, even in the case of bursts, it is advised to use highly optimized implementations with an event-driven framework such as Flink. When subsecond latency is not required, Structured Streaming offers a high-level,

intuitive API that gives very high throughput with minimal tuning. It is important to keep in mind that at the time of this writing parts of the Structured Streaming API were still experimental and not all pipelines using built-in operations were supported yet. For those use cases, Spark Streaming with a high micro-batch interval offers a good alternative. For ETL jobs on data residing on a Kafka cluster that do not require very high throughput and have reasonable latency requirements, Kafka Streams offers an interesting alternative to Flink with the additional advantage that it does not require a cluster pre-installed.

7 Limitations and Further Research

This benchmark could be extended with joins with static datasets and the stateful operators should be analyzed under different window lengths. As well, other frameworks could be included such as Apex, Beam and Storm. Finally, the fault tolerance and scalability of the frameworks under these workloads are interesting directions for future work.

Acknowledgments

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] J. Karimov, T. Rabl, A. Katsifodimos, R. Samarev, H. Heiskanen, and V. Markl, "Benchmarking distributed stream processing engines," *arXiv preprint arXiv:1802.08496*, 2018.

[2] Z. Karakaya, A. Yazici, and M. Alayyoub, "A comparison of stream processing frameworks," in *Computer and Applications (ICCA), 2017 International Conference on*. IEEE, 2017, pp. 1–12.

[3] S. Chintapalli, D. Dagit, B. Evans, R. Farivar, T. Graves, M. Holderbaugh, Z. Liu, K. Nusbaum, K. Patil, B. J. Peng *et al.*, "Benchmarking streaming computation engines: Storm, flink and spark streaming," in *Parallel and Distributed Processing Symposium Workshops, 2016 IEEE International*. IEEE, 2016, pp. 1789–1792.

[4] R. Lu, G. Wu, B. Xie, and J. Hu, "Stream bench: Towards benchmarking modern distributed stream computing frameworks," in *Utility and Cloud Computing (UCC), 2014 IEEE/ACM 7th International Conference on*. IEEE, 2014, pp. 69–78.

[5] S. Qian, G. Wu, J. Huang, and T. Das, "Benchmarking modern distributed streaming platforms," in *Industrial Technology (ICIT), 2016 IEEE International Conference on*. IEEE, 2016, pp. 592–598.

[6] M. Armbrust, T. Das, J. Torres, B. Yavuz, S. Zhu, R. Xin, A. Ghodsi, I. Stoica, and M. Zaharia, "Structured streaming: A declarative api for real-time applications in apache spark," in *Proceedings of the 2018 International Conference on Management of Data*. ACM, 2018, pp. 601–613.

[7] M. Zaharia, T. Das, H. Li, T. Hunter, S. Shenker, and I. Stoica, "Discretized streams: Fault-tolerant streaming computation at scale," in *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*. ACM, 2013, pp. 423–438.

[8] A. Shukla and Y. Simmhan, "Benchmarking distributed stream processing platforms for iot applications," in *Technology Conference on Performance Evaluation and Benchmarking*. Springer, 2016, pp. 90–106.

[9] A. Shukla, S. Chaturvedi, and Y. Simmhan, "Riotbench: An iot benchmark for distributed stream processing systems," *Concurrency and Computation: Practice and Experience*, vol. 29, no. 21, 2017.

[10] G. van Dongen, B. Steurtewagen, and D. Van den Poel, "Latency measurement of fine-grained operations in benchmarking distributed stream processing frameworks," in *2018 IEEE International Congress on Big Data (BigData Congress)*. IEEE, 2018, pp. 247–250.

[11] J. Kreps, N. Narkhede, J. Rao *et al.*, "Kafka: A distributed messaging system for log processing," in *Proceedings of the NetDB*, 2011, pp. 1–7.

[12] P. Carbone, A. Katsifodimos, S. Ewen, V. Markl, S. Haridi, and K. Tzoumas, "Apache flink: Stream and batch processing in a single engine," *Bulletin of the IEEE Computer Society Technical Committee on Data Engineering*, vol. 36, no. 4, 2015.

[13] M. J. Sax, G. Wang, M. Weidlich, and J.-C. Freytag, "Streams and tables: Two sides of the same coin," in *Proceedings of the International Workshop on Real-Time Business Intelligence and Analytics*. ACM, 2018, p. 1.

[14] "Flink Documentation," accessed: 2019-12-19. [Online]. Available: https://ci.apache.org/projects/flink/flink-docs-stable/

[15] P. Carbone, S. Ewen, G. Fóra, S. Haridi, S. Richter, and K. Tzoumas, "State management in apache flink®: consistent stateful distributed stream processing," *Proceedings of the VLDB Endowment*, vol. 10, no. 12, pp. 1718–1729, 2017.

[16] "Kafka Streams Documentation," accessed: 2019-12198. [Online]. Available: https://kafka.apache.org/documentation/streams/

[17] "Apache Spark: Spark programming guide," 2019, accessed: 2019-12-19. [Online]. Available: https://spark.apache.org/docs/latest/streaming-programming-guide.html

[18] "Structured Streaming," accessed: 2019-12-19. [Online]. Available: https://spark.apache.org/docs/latest/structured-streaming-programming-guide.html

[19] "cAdvisor," accessed: 2019-06-04. [Online]. Available: https://github.com/google/cadvisor

[20] M. Caporaloni and R. Ambrosini, "How closely can a personal computer clock track the utc timescale via the internet?" *European journal of physics*, vol. 23, no. 4, pp. L17–L21, 2002.

[21] "Kafka Improvement Proposals: KIP-32 - Add timestamps to Kafka message," accessed 2019-06-04. [Online]. Available: https://cwiki.apache.org/confluence/display/KAFKA/KIP-32+-+Add+timestamps+to+Kafka+message

[22] Y. Byzek, "Optimizing Your Apache Kafka Deployment," accessed: 2019-12-18. [Online]. Available: https://www.confluent.io/wp-content/uploads/Optimizing-Your-Apache-Kafka-Deployment-1.pdf

Giselle van Dongen is a PhD researcher at Ghent University, teaching and benchmarking real-time distributed processing systems such as Spark Streaming, Flink, Kafka Streams and Storm. Concurrently, she is Lead Data Scientist at Klarrio specialising in real-time data analysis, processing and visualisation.

Dirk Van den Poel Dr. Dirk Van den Poel (PhD) is a Senior Full Professor of Data Analytics/Big Data at Ghent University, Belgium. He teaches courses such as Big Data, Databases, Social Media and Web Analytics, Analytical Customer Relationship Management, Advanced Predictive Analytics, Predictive and Prescriptive Analytics. He co-founded the advanced Master of Science in Marketing Analysis, 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.