Supplementary Material for Evaluation of Stream Processing Frameworks

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

Abstract—This paper contains the supplementary material to the paper submitted to IEEE TPDS entitled 'Evaluation of Stream Processing Frameworks'. It improves the completeness of the TPDS manuscript.

1 Related work

In Table 1 we listed the related work on which this benchmark was based. We point out the most important differences between them and how our work complements them.

2 Data

In this section, we give additional information on the data used in the main file. Each message is around 200 bytes in size and is published to Kafka with its key and JSON message body. The keys contain the measurement point ID and lane ID. Each measurement point has on average 1.6 lanes. We partition the input stream on the part of the key containing the measurement ID. The number of distinct keys scales linearly with the volume. The JSON message contains 6-7 key-value pairs. The speed messages and flow messages are published onto two different Kafka topics. The timestamps within the messages are replaced by the time of replay.

3 Deployment

In Section 3.4 of the main paper, we explained the four different workloads that we included in this benchmark. In this section, we give a more in-depth view of how these workloads were run. First of all, a diagram of the architecture has been given in Figure 1. An overview of how this diagram relates to data streams between the components running on the DC/OS cluster has been given in Figure 2. In the following, we give a step-by-step runbook for each of the workloads to clarify.

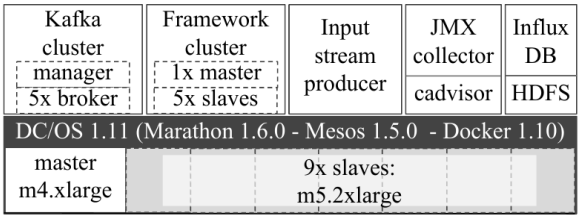

Fig. 1. Benchmark architecture with the active services during runs.

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

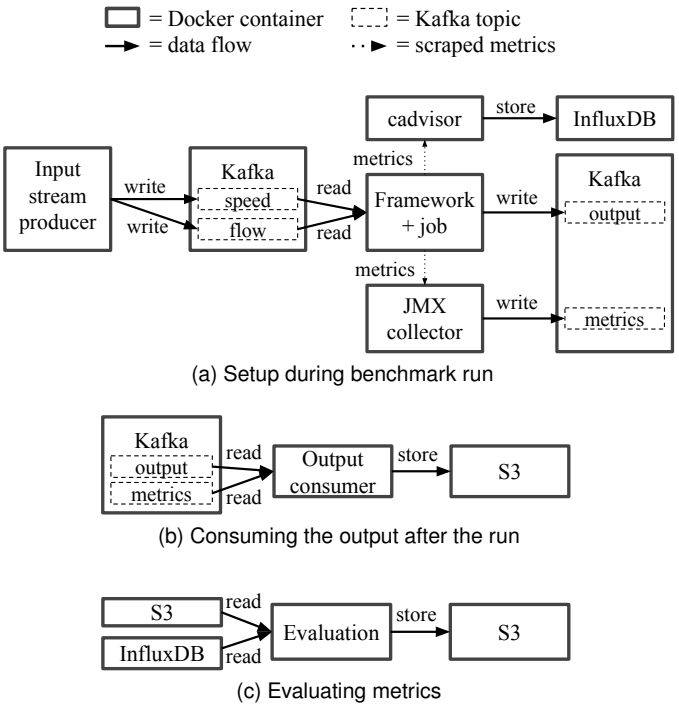

Fig. 2. Setup throughout the benchmark process

3.1 Runbook for workload for latency measurement

1) For each of the frameworks [Spark Streaming, Flink, Kafka Streams and Structured Streaming]
   a) For each of the pipeline complexities [ingest, parse, join, tumbling window, sliding window]
      i) Start cluster of the framework, if it requires one and wait a few minutes to complete startup.
      ii) Create a new Kafka output topic for the results to be published on and a topic for the JMX metrics to be published on.
      iii) Start up the JMX exporter.
      iv) Start up the processing job.
      v) Start the data stream generator.
      vi) Wait for 40 minutes: 30 minutes to process the data and 10 minutes to catch up

TABLE 1
Related work on which the development of this benchmark was based.

Reference	Frameworks	Operations	Workloads	Metrics	Data
Karimov et al.,2018 [1]	Spark Streaming 2.0.1, Flink 1.1.3, Storm 1.0.2	Windowed aggregation, join, large windows	Varying cluster sizes, different data scales	Event and processing-time latency, sustainable throughput	Constant rate at 90% and max throughput rate, periodic bursts, extreme data skew
van Dongen et al.,2018 [2]	Flink 1.3.1	Ingest, parse, join, tumbling window, sliding window	One workload, one scale	Latency	Constant rate, one scale
Karakaya et al., 2017 [3]	Spark Streaming 1.5.0, Flink 0.10.1, Storm 0.10.0	End-to-end pipeline (parse - filter - project - join - window)	Varying cluster sizes and multiple data scales	Saturation level, scale-up ratio, resource consumption	Constant rate
RIoTBench, 2017 [4]	Storm 1.0.1	End-to-end jobs: parse, filter, statistical, predictive, IO, visualization	One workload for each of four datasets	Latency, throughput, jitter, memory, CPU	Two data scales at constant rate, normally distributed rate, bi-modal rate
Yahoo Benchmark, 2016 [5]	Spark Streaming 1.5.0, Flink 0.10.1, Storm 0.10.0-0.11.1	End-to-end pipeline (read - deserialize - filter - projection - join - window)	One workload at varying scales	99th percentile latency and throughput	Constant rate
Shukla and Simmhan, 2016 [6]	Storm 1.0.0	End-to-end applications: parse, filter, statistical, predictive, IO	One workload per dataset	Latency, throughput, jitter, memory, CPU	Two datasets: quasi-constant rate at one scale and bi-modal rate
Qian et al., 2016 [7]	Spark Streaming 1.4.0, Storm 0.9.3, Samza 0.8.0	Identity, sample, projection, grep, statistics, wordcount, distinct count	capability, fault-tolerance	Latency, throughput, TPF, LPF, CPU, network, memory	Constant rate at different scales
SparkBench, 2015 [8] (streaming section)	Spark Streaming (around 1.3.0)	most popular Twitter tag, PageView	one workload, one scale	Resource consumption, job execution time, data process rate, shuffle tasks	Constant rate
StreamBench, 2014 [9]	Spark Streaming 0.9.0-incubating, Storm 0.9.1-incubating	Identity, sample, projection, grep, statistics, wordcount, distinct count	Performance, multi-recipient performance, fault tolerance, durability	Latency, throughput, TPF, LPF, durability index	Constant rate at different scales
This study	**Flink 1.9.1, Structured Streaming and Spark Streaming 2.4.1, Kafka Streams 2.1.0**	**Ingest, parse, join, tumbling window, sliding window, event-driven stateful operations**	**Single broker constant rate, multi broker constant rate, single burst, periodic burst**	**Latency, peak burst throughput, peak sustainable throughput, memory, CPU**	**Constant rate at different scales, periodic bursts, single initial burst**

with possible lags (Figure 2a).

vii) Stop the data stream generator, JMX exporter and streaming job.

2) Start a job to consume the output and metrics from Kafka and write it to S3 (Figure 2b).
3) Start a job to evaluate the output, JMX metrics and cadvisor metrics (Figure 2c).

3.2 Runbook for workload for sustainable throughput measurement

1) For each of the frameworks [Spark Streaming (3 seconds and 5 seconds micro-batch intervals), Flink, Kafka Streams and Structured Streaming]:
   a) For each of the pipeline complexities [ingest, parse, join, tumbling window, sliding window]:
      i) For a list of different throughput levels:
         A) Start cluster of the framework, if it requires one and wait a few minutes to complete startup.
         B) Create a new Kafka output topic for the results to be published on and a topic for the JMX metrics to be published on.
         C) Start up the JMX exporter.
         D) Start up the processing job.
         E) Start the input stream producer.
         F) Wait for 40 minutes: 30 minutes to process the data and 10 minutes to catch up with possible lags (Figure 2a).
         G) Stop the input stream producer, JMX exporter and streaming job.

2) Start a job to consume the output and metrics from Kafka and write it to S3 (Figure 2b).
3) Start a job to evaluate the output, JMX metrics and cadvisor metrics (Figure 2c).

TABLE 2
Resources of DC/OS cluster services

Name	Version	Amt	CPU* units	RAM* GB	Disk* GB
Resources: EC2 Instance Types					
m5.2xlarge		9	8	32	1024
DC/OS Services					
Kafka Multi-Broker	2.1.0	5	2	6	10
Kafka Single-Broker	2.1.0	1	6.5	24	10
Kafka Manager	1.1.0	1	1	2	0
JMX collector	/	1	1	2	0
cAdvisor	0.31	9	0.25	0.25	0
InfluxDB	0.13	1	1	2	0
Data Generator	/	1	3	8	0
HDFS	2.6.0				
Name nodes		2	2	6	10
ZKFC nodes		2	1	2	0
Journal nodes		3	1	2	10
Data nodes		5	1	6	20
Flink JobManager	1.9.1	1	2	8	20
Flink TaskManager	1.9.1	5	4	20	20
Kafka Streams	2.1.0	5	4	20	20
Spark Master	2.4.1	1	2	8	20
Spark Worker	2.4.1	5	4	20	20

* CPU, memory and disk are per instance of the service.

3.3 Runbook for workload for burst at startup

1) For each of the frameworks [Spark Streaming (3 seconds and 5 seconds micro-batch intervals), Flink, Kafka Streams and Structured Streaming]:
   a) For each of the pipeline complexities [ingest, parse, join, tumbling window, sliding window]:
      i) Start cluster of the framework, if it requires one and wait a few minutes to complete startup.
      ii) Create a new Kafka output topic for the results to be published on and a topic for the JMX metrics to be published on.
      iii) Start the input stream producer and let it publish for 5 minutes.
      iv) Start up the JMX exporter.
      v) Start up the processing job
      vi) Wait for 10 minutes. The processing job will catch up with the five minute delay and then continues processing the newly incoming data (Figure 2a).
      vii) Stop the input stream producer, JMX exporter and streaming job.
2) Start a job to consume the output and metrics from Kafka and write it to S3 (Figure 2b).
3) Start a job to evaluate the output, JMX metrics and cadvisor metrics (Figure 2c).

3.4 Runbook for workload with periodic bursts

1) For each of the frameworks [Spark Streaming, Flink, Kafka Streams and Structured Streaming]
   a) For each of the pipeline complexities [ingest, parse, join, tumbling window, sliding window]
      i) Start cluster of the framework, if it requires one and wait a few minutes to complete startup.
      ii) Create a new Kafka output topic for the results to be published on and a topic for the JMX metrics to be published on.
      iii) Start up the JMX exporter.
      iv) Start up the processing job.
      v) Start the input stream producer with periodic bursts.
      vi) Wait for 40 minutes: 30 minutes to process the data and 10 minutes to catch up with possible lags (Figure 2a).
      vii) Stop the input stream producer, JMX exporter and streaming job.
2) Start a job to consume the output and metrics from Kafka and write it to S3 (Figure 2b).
3) Start a job to evaluate the output, JMX metrics and cadvisor metrics (Figure 2c).

We send a low constant volume of data in between bursts. For the frameworks using event time this means that the event time watermark keeps increasing steadily in between bursts. If no data would be send in between bursts, the watermark only advances after the following burst and therefore, events are sent with a latency of at least the interval between bursts unless custom triggers are defined. By sending a low volume in between bursts, data gets processed instantly and we get a more accurate view on the processing time of the events.

4 Frameworks: Implementation and Configuration

In Section 4 of the main paper we laid out the chosen framework configuration parameters. In this section, we give a more thorough explanation of how we tuned the parameters for each workload and the effects they had on performance.

4.1 Apache Flink

4.1.1 Latency-throughput trade-off

As described by [10], configuration parameters can have a significant effect on the trade-off between latency and throughput. In this section, we describe the influence of the buffer timeout for Flink and justify the chosen values in the main paper. Buffers are used to exchange data between operators and are flushed either when they are full or when a timeout condition is reached [10]. Setting the timeout to a low value enables Flink to process with lower latencies. To illustrate this, we ran the latency workload for the ingest phase for a buffer timeout of 0 ms and 100 ms, which is the default. The p99 latency of the run with a buffer timeout of 0 ms was around 1 ms while the p99 latency for buffer timeout 100 ms was between 70 and 90 ms. When throughput is not large enough to fill up the buffers in under 100 ms, the latency grows linear with the buffer timeout.

4.2 Apache Kafka Streams

4.2.1 Latency-throughput trade-off

Similar to what has been described for Apache Flink in Section 4.1.1, Kafka Streams offers several configuration parameters to tune the latency-throughput trade-off [11]. The most important one for Kafka Streams is the linger time of producers. Producers automatically batch messages before sending them over the network to reduce load. The linger time sets the upper bound for the amount of time producers wait for batches to fill. The default linger time in Kafka Streams is 100 ms. For the latency workload we set this value to 0 ms to optimize for latency. When running the latency workload with the default configuration, the median latency of the ingest phase was 90 ms. Setting the linger time to 0 ms, reduced the median latency to less than 1 ms.

4.2.2 Handling bursts and out-of-order data

As explained in [12], Kafka Streams uses a Dual Streaming Model meaning that results of operators are presented as streams of successive updates. The developers of Kafka Streams introduced this model to cope with out-of-order data. For the initial stages of our pipeline this does not have a large effect since one input event leads to one output event. When we use the Kafka Streams DSL for the aggregation and sliding window stage, however, an updated result will be send for each input event leading to redundant output. To prevent this, we had to add a filter step after each stage which filters out all complete events. In addition, the grace period and retention time for each window need to be set. The grace period gives an upper bound for the amount of time a window should allow for out-of-order events after the window ends [13]. Retention time is the lower-level property which sets the amount of time events should be retained in the state stores. Since Kafka Streams publishes each window update, we set the grace period of the inner stream-stream join and tumbling window quite high, to 5s for most workloads. This does not influence latency because updates are not buffered and our stream does not contain out-of-order data. When using the Kafka Streams DSL for the sliding window, we suppress output until event time has passed the window end and a watermark. We therefore use a lower grace period of 50 ms for the final stage since this will have an effect on the latency of the events.

When processing bursts of data with the Kafka Streams DSL, i.e. burst at startup or periodic bursts, we noticed that events got discarded incorrectly. We noticed a similar effect for the low-level API implementation for the workload with a burst at startup. This was caused by the threads not reading equally fast from all topic partitions. Due to this, events were discarded because event time had already progressed past the watermark. To prevent this, we set max.task.idle.ms which sets the maximum amount of time a stream task will stay idle when not all of its partition buffers contain records, to avoid potential out-of-order record processing across multiple input streams [14]. This makes sure records are processed in timestamp order across all topics. This does however have an influence on the latency for the periodic burst workload.

4.3 Apache Spark: Spark Streaming

One of the features we experimented with for Spark Streaming is its backpressure mechanism, which allows limiting the rate at which receivers can ingest data. This can help overcome bursts of data that would otherwise cause the system to overflow. As an experiment, we turned this on for the periodic burst workload. We noticed increased latencies for up to four or five seconds after the burst. Latencies grew to 3000-4000 ms since the events that were published in the burst were processed in smaller chunks in subsequent batches. However, processing data in smaller portions led to lower resource utilization. Since the bursts are not large enough to flood the framework, we chose to turn the backpressure mechanism back off. We did not consider using the backpressure mechanism for the other workloads since they had constant rate input streams and would not benefit from a backpressure mechanism.

4.4 Apache Spark: Structured Streaming

We make use of the DataFrame API of Structured Streaming and avoid conversions to the Dataset API. Datasets are represented by Scala case classes while DataFrames use the InternalRow data type. Conversions happen based on the mapping of column names to the fields of the case class. The extra Java objects that are created during this process lead to extra GC load and should therefore be avoided. GC pauses were further decreased by switching from Parallel GC to G1GC. Parallel GC pauses all threads during a major GC. [15], [16]. The newer G1GC is designed to reduce the length of these pauses. By switching the garbage collector from the default Parallel GC to G1GC the tail latency decreased by 15 seconds. To achieve this performance improvement, we adapted the Java options to use two concurrent GC threads and four parallel GC threads. Increasing the number of concurrent GC threads gives a bit more load on the CPU but reduces the amount of time spent on GC [16]. Parallel GC threads are used during full GCs, while concurrent GC threads are used during other GC activities. Additionally, we decreased the Java option InitiatingHeapOccupancyPercent from 45% to 35%. Decreasing this, makes sure garbage collection gets triggered sooner but is less invasive. For the other frameworks, using G1GC did not lead to similar performance improvements.

When reducing the amount of shuffle partitions, we noticed a large latency reduction as can be seen in Figure 3. For the stateful operations, checkpointing is done for each microbatch. The checkpointing frequency will be reduced in a future version of Spark since it creates a considerable overhead and performance impact. Reducing the shuffle partitions reduced the amount of connections that need to be opened to HDFS to store state which in turn reduced the latency.

5 Supplementary results

In this section we describe supplementary results to those discussed in the main paper.

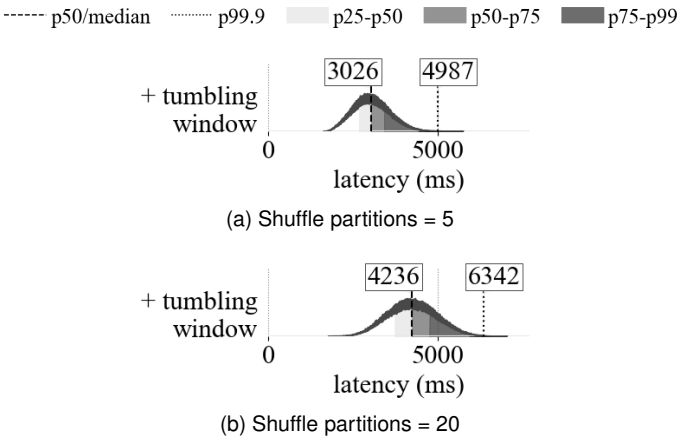

Fig. 3. Structured Streaming: Influence of the amount of shuffle partitions on latency performance

5.1 Workload for latency measurement

In the main paper, we gave an overview of the distribution of latency for each framework and pipeline complexity. To get a better view on how this distribution translates to latency evolution over time, we included the timeseries chart in Figure 4. In this figure, we visualized the 99th, 50th and 1st percentiles of latency for each of the frameworks and implementations.

We can observe that all jobs show stable behavior over time with minimal changes in the percentiles. For Structured Streaming, we see stable behavior with longer and more variable p99 latencies than for the other frameworks. This increases further when the tumbling window stage is added.

5.2 Workload for sustainable throughput measurement

In Figure 5a and Figure 5b, we see that the peak sustainable throughput of the customized tumbling window implementations for Flink and Kafka Streams did not differ substantially from the high-level default implementations. However, we notice that the median latency is much lower for the customized implementations as has been described in the results of the latency measurement workload.

An overview of the peak sustainable throughput levels for each of the frameworks and implementations for the tumbling window stage have been listed in Table 3.

TABLE 3
Overview of peak sustainable throughput per framework and implementation for the tumbling window stage.

framework	implementation	peak sust. throughput
Flink	built-in	26 000 msg/s
	customized	25 000 msg/s
Kafka Streams	DSL	18 500 msg/s
	processor API	20 000 msg/s
Spark Streaming	3 seconds	26 000 msg/s
	5 seconds	30 000 msg/s
Structured Streaming	built-in	115 000 msg/s
	customized	125 000 msg/s

5.3 Workload with burst at startup

Table 4 shows an overview of the results for the single burst workload for each of the frameworks. We show the

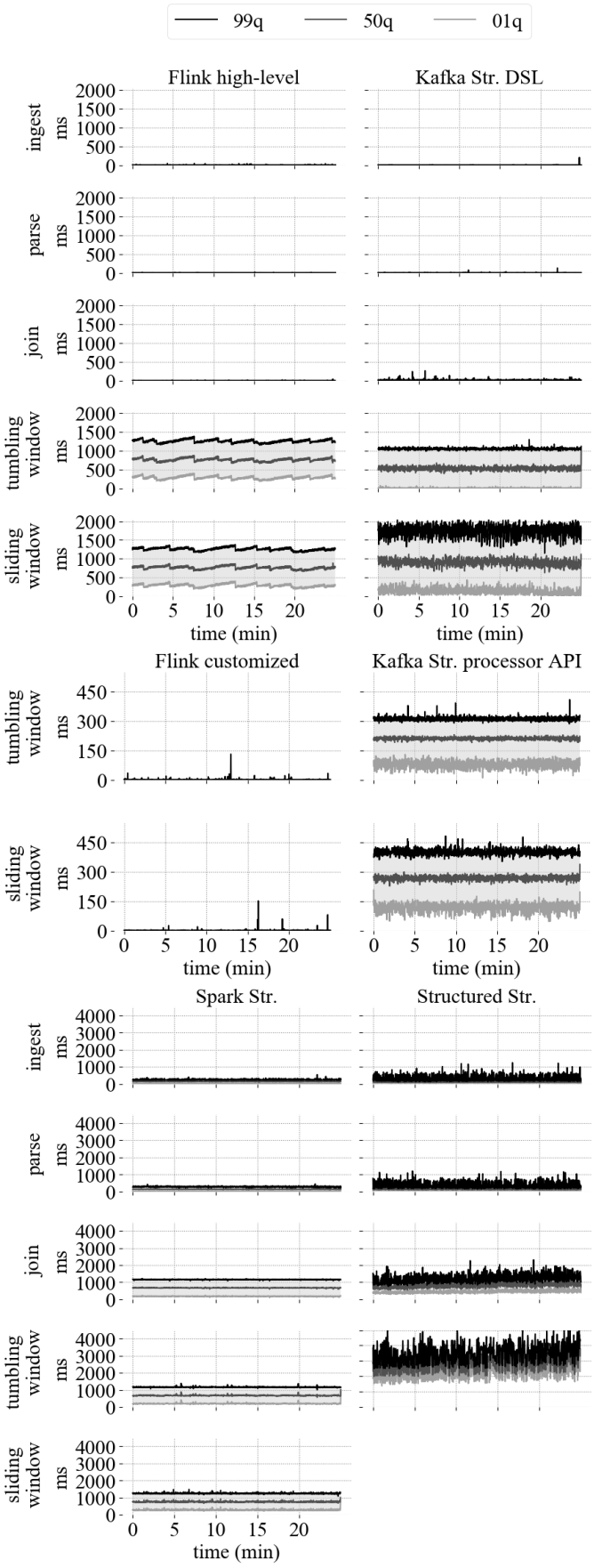

Fig. 4. Latency evolution over time for each framework

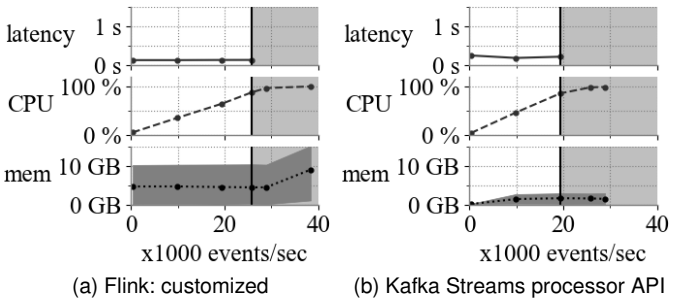

Fig. 5. Sustainable throughput: performance for different throughput levels for the tumbling window stage. Each marker represents a benchmark run for the corresponding throughput level. The light grey zone marks unsustainable levels of throughput.

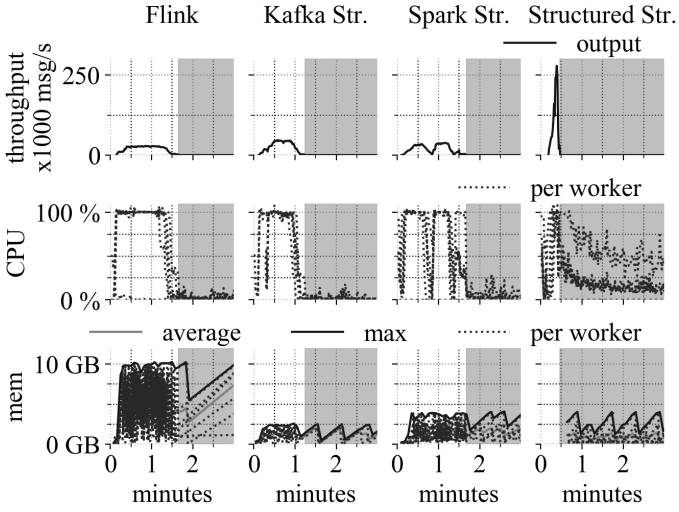

Fig. 6. Single burst workload until parse stage: performance for first three minutes of the run. The grey zone denotes where processing has caught up on the input stream.

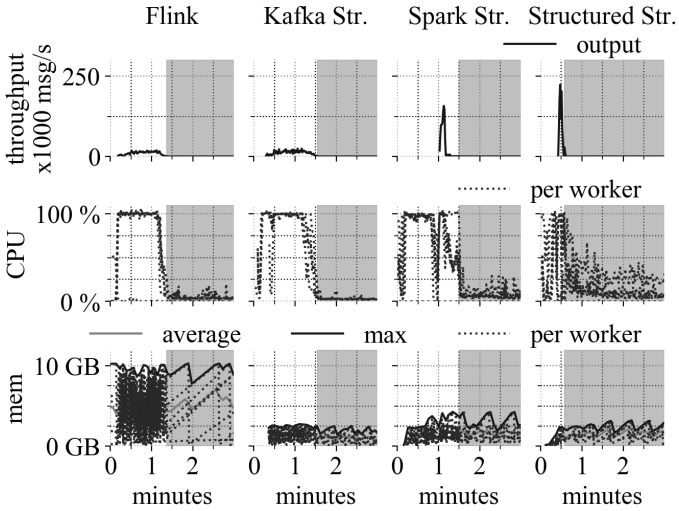

Fig. 7. Single burst workload until join stage: performance for first three minutes of the run. The grey zone denotes where processing has caught up on the input stream.

TABLE 4
Single burst workload results for Flink (FL, with d for default windows and c for customized windows), Kafka Streams (K, with d for default windows and c for customized windows), Spark Streaming (SP) and Structured Streaming (ST). Time in seconds required for initial output and total processing of burst and the peak throughput reached.

End stage	Decision factor	FL	K	SP	ST
Parse	initial output	9	7	6	11
	processing time	99	74	100	27
	peak throughput	30K	47K	39K	279K
Join	initial output	12	18	62	25
	processing time	82	92	90	34
	peak throughput	19K	25K	157K	223K
Tumbling window	initial output	d: 16 c: 18	d: 16 c: 18	69	d: 44 c: 36
	processing time	d: 81 c: 84	d: 118 c: 99	84	d: 54 c: 46
	peak throughput	d: 22K c: 12K	d: 9K c: 10K	110K	d: 150K c: 229K

amount of time before the initial output was published and the entire processing time of the initial burst. For Flink and Kafka, the results of the high-level windowing and low-level customized implementation are shown. We can see that for Flink and Kafka Streams the time before the initial output is published increases as the pipeline becomes more complex. This is due to the increased interdependence between events and hence, the increased buffering. The processing time of the entire batch, however, does not increase as much and in some cases even decreases. Structured Streaming is consistently the fastest in recovering due to its optimized execution engine for microbatch processing [17]. We also see this advantage for Apache Spark as the pipeline gets more complex.

We also visualized the results for the parse and join stage in Figure 6 and Figure 7. In these figures, we can observe that the behavior of most frameworks is similar for each of the pipeline complexities. For Spark Streaming, we see different behavior for the parsing stage. For the parsing stage two streams are read separately from Kafka: a stream with flow events and a stream with speed events. When these two streams are published onto Kafka this leads to two tasks in Spark Streaming, one per input stream. These tasks are executed sequentially, which is the reason that we see two phases in the output of the parsing stage for Spark Streaming and a drop in CPU utilization in the middle. In the later stages these two streams are joined and this is not visible anymore. For Structured Streaming these two input streams translate to two queries which are submitted at the same time and processed in parallel. Therefore, we do not see the same behavior as for Spark Streaming.

5.4 Workload with periodic bursts

In Figure 8 and Figure 9, we show the additional results for the periodic burst workload for the parse and join stage. We notice that the high latencies that we observed for Structured Streaming for the tumbling window stage were not present for the less complex pipelines. This is due to delays in the propagation of watermarks, as explained in Section 5.1 of the main file. For the parsing stage, Structured Streaming reacts the least to bursts and processes bursts

with lower latencies than Flink and Kafka Streams. For the join stage, its latency is similar to Flink and Kafka Streams. As discussed in the main file in Section 5.2, Structured Streaming can sustain the largest throughput due to its dynamic microbatch approach and it also benefits from that when processing bursts in pipelines that do not require watermark propagation.

To give additional detailed information on how a burst is processed on average we added Figure 10. This figure shows the average output throughput in the first seconds after a burst, as well as the latency of the events that were published during those seconds. These values have been computed by taking the average behavior over all the bursts of the run. The amount of output events reduces from stage to stage since the parsing stage has one output event per input event, while the join only outputs one event per pair of input events, and the tumbling window uses on average 3.17 events to generate one output event. We see that as the pipeline gets more complex, it takes more and more time before the largest burst of output events is published. Especially for Structured Streaming this is noticeable. For the parsing stage 28 000 events are published in the first second after the burst while it takes 4 seconds for the output of the tumbling window stage to be published.

We also see that for Flink and Kafka Streams the output is published much faster after the burst when customized state operations are used. For Flink, when using the default event time trigger the p99 latencies remain fairly constant when processing bursts while p50 latencies increase slightly as can be seen in Figure 10c. For the custom trigger implementation (Figure 10d) we see a much larger effect with p99 latencies rising from 125 ms to above one second for the processing of the burst. However, the latency remains lower than for the tumbling window approach with a p50 of 1380 ms and p99 of 1730 ms. As a side note, the median latency (125 ms) in between bursts is much higher than the latencies we clocked during the latency workload (1 ms) because the buffer timeout was set at 100ms. Records are buffered for the entire buffer timeout period during low throughput periods.

Kafka Streams has difficulties with processing bursts since it does not guarantee to read equally fast from all partitions of the Kafka input topics. This has as effect that events get discarded because event time progresses too fast. As explained in Section 4.2.2, increasing the maximum time a task is allowed to be idle mitigates this.

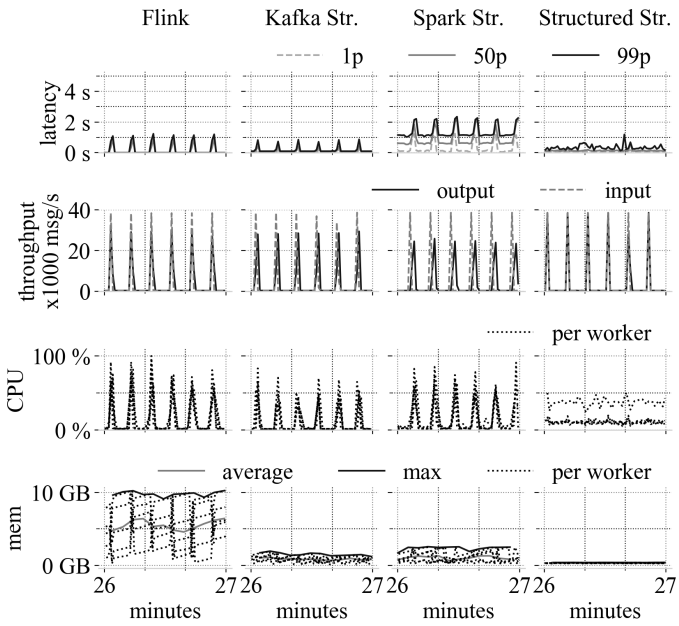

Fig. 8. Periodic burst workload until the parse stage

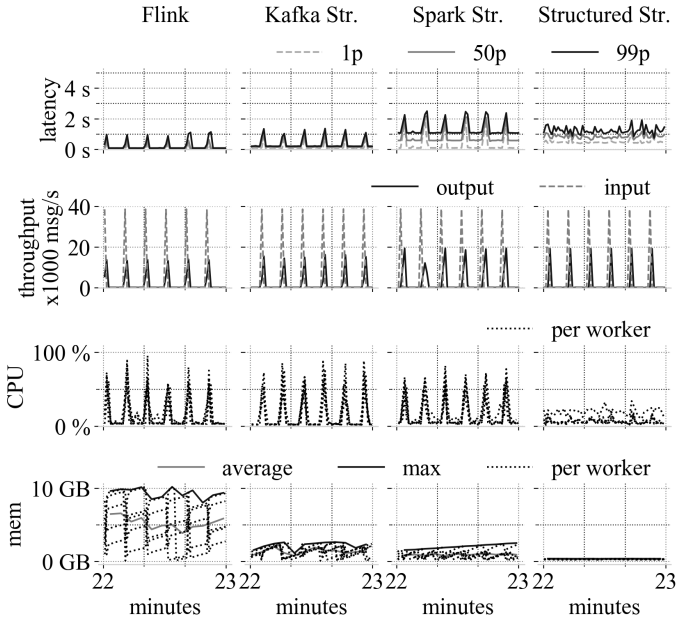

Fig. 9. Periodic burst workload until the join stage

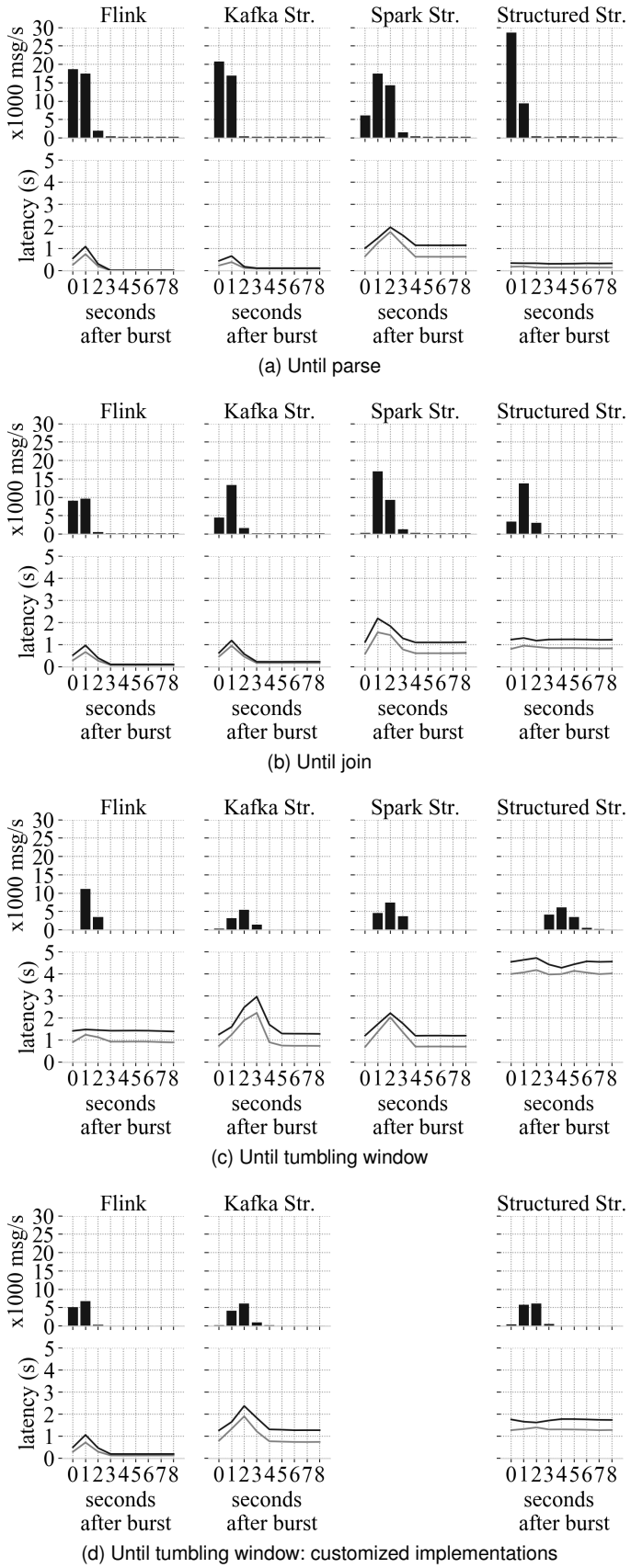

Fig. 10. Periodic burst workload: average output throughput and average p50 and p99 latency of the output throughput in the seconds right after a burst for different pipeline complexities.

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] 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.

[3] 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.

[4] 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.

[5] 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.

[6] 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.

[7] 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.

[8] M. Li, J. Tan, Y. Wang, L. Zhang, and V. Salapura, "Sparkbench: a comprehensive benchmarking suite for in memory data analytic platform spark," in *Proceedings of the 12th ACM International Conference on Computing Frontiers*. ACM, 2015, p. 53.

[9] 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.

[10] 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.

[11] Y. Byzek, "Optimizing Your Apache Kafka Deployment," accessed 2018-12-18. [Online]. Available: https://www.confluent.io/white-paper/optimizing-your-apache-kafka-deployment

[12] 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.

[13] "Kafka Streams Documentation," accessed 2018-12-08. [Online]. Available: https://kafka.apache.org/documentation/streams/

[14] "Kafka Improvement Proposals: KIP-353: Improve Kafka Streams Timestamp Synchronization," accessed 2019-12-02. [Online]. Available: https://cwiki.apache.org/confluence/display/KAFKA/KIP-353%3A+Improve+Kafka+Streams+Timestamp+Synchronization

[15] "Java Garbage Collection Basics," accessed 2018-12-06. [Online]. Available: https://www.oracle.com/webfolder/technetwork/tutorials/obe/java/gc01/index.html

[16] D. Wang and J. Huang, "Tuning Java Garbage Collection for Apache Spark Applications," 2015, accessed 2018-12-06. [Online]. Available: https://databricks.com/blog/2015/05/28/tuning-java-garbage-collection-for-spark-applications.html

[17] 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.