# Latency Measurement of Fine-Grained Operations in Benchmarking Distributed Stream Processing Frameworks

Giselle van Dongen
*Ghent University*
Ghent, Belgium
giselle.vandongen@UGent.be

Bram Steurtewagen
*Ghent University*
Ghent, Belgium
bram.steurtewagen@UGent.be

Dirk Van den Poel, *Senior Member, IEEE,*
*Ghent University*
Ghent, Belgium
dirk.vandenpoel@UGent.be

***Abstract*—This paper describes a benchmark for stream processing frameworks allowing accurate latency benchmarking of fine-grained individual stages of a processing pipeline. By determining the latency of distinct common operations in the processing flow instead of the end-to-end latency, we can form guidelines for efficient processing pipeline design. Additionally, we address the issue of defining time in distributed systems by capturing time on one machine and defining the baseline latency. We validate our benchmark for Apache Flink using a processing pipeline comprising common stream processing operations. Our results show that joins are the most time consuming operation in our processing pipeline. The latency incurred by adding a join operation is 4.5 times higher than for a parsing operation, and the latency gradually becomes more dispersed after adding additional stages.**

***Index Terms*—big data applications, distributed stream computing, benchmark, Flink, Kafka**

## I. Introduction

Over the last decade, much research has been put into the development of fast, scalable, fault-tolerant stream processing frameworks. Because of their scalable, low-latency engine designed to run on commodity clusters or in the cloud, these frameworks are especially suitable for dealing with the increasing amount of continuously generated data coming from various domains: IoT, social media, web logs, etc.

Due to this vast development of stream processing frameworks, establishing clear guidelines about which framework to use for which use case has become cumbersome. Preliminary work in this field tends to focus on defining benchmark proposals consisting of common stream processing workloads along with appropriate performance metrics [1]–[3]. Latency is one of these performance metrics, since low-latency stream processing jobs are becoming more and more business-critical for many companies, unprovisioned delays can have significant consequences for a business [4].

The setup described in this paper, enables performance benchmarks of separate stages of a processing pipeline. These outcomes can be used as a guideline when designing efficient processing pipelines for other use cases. The end-to-end processing pipeline of our benchmark includes common operations on data streams, closely following those of [5]: reading from a Kafka topic, parsing, joining, aggregating, window operations and writing to a Kafka topic. We define our baseline as merely consuming from Kafka topics and directly publishing the raw observations back to Kafka. The performance change observed after adding a stage to the pipeline is then interpreted as the framework's performance when executing that specific stage.

A fundamental issue when measuring latency in distributed systems is the absence of absolute, global time to which we can appeal. Every machine in the cluster has a local quartz clock for measuring time but due to clock drift these cannot be deemed accurate. Typical local clock drift is assumed to be around 30 ppm (2.6 seconds per day) and can fluctuate further due to temperature differences [6]. TrueTime, the globally synchronized clock used in Google's Spanner database, reports an average clock drift of 4 ms [7]. Several clock synchronization methods have been developed, e.g. Network Time Protocol (NTP) but none of them reach full precision [8]–[11]. The minimum error when synchronizing over NTP is 35 ms according to [12]. Previous benchmarking literature does not take this into account. In our latency benchmark, we address this issue by capturing the timestamps of incoming and outgoing messages on a single machine of a message system. In our benchmark, we use Apache Kafka as a message system and we capture time on one single broker for all Kafka topics and their partitions (cf. Section III-B).

Overall, we make the following contributions to benchmarking literature of stream processing frameworks:

1) A new way of benchmarking latency (cf. supra) for stream processing frameworks, validated on Apache Flink.
2) Fine-grained latency measurement of common operations in a processing pipeline to lay out guidelines for more efficient processing pipeline design.
3) Correct latency measurements by capturing time on one machine.

The rest of this paper is organized as follows. The next section gives a brief overview of previous work in this domain. In Section III our latency benchmark is presented, and we

provide an overview of the environment set up to conduct this benchmark for the popular framework Apache Flink. A more detailed description of Apache Flink is given in Section IV. The discussion of results follows in Section V. General conclusions are drawn in Section VI. Finally, we state the limitations of our research and issues for further research.

## II. Related Work

In the last few years, some initial work has been done on benchmarking modern distributed stream processing frameworks. In this section, we will briefly review the past literature that served as a basis for our research.

In 2014, an initial benchmark definition for stream processing frameworks was developed called Stream Bench [2]. It included methodologies for selecting and generating data, workloads and metrics. The program set was implemented for Apache Spark and Apache Storm. As proposed in Stream Bench, we use a message system to decouple data generation from data consumption.

Qian et al. [13] extended Stream Bench to benchmark Apache Spark (receiver and direct approach) and Apache Storm (including Trident) with tuning of parameters. They determined the optimal settings for each framework. Lastly, they compared the characteristics of hardware utilization (CPU, memory, network and disk) for these platforms.

To guide users in optimizing configurations and cluster deployments of Apache Spark, Li et al. [1] presented SparkBench. The benchmark was designed to test both the batch and streaming components of Apache Spark for different settings.

A benchmark of Spark, Storm and Flink was conducted at Yahoo [5]. The incorporated metrics were throughput and 99th percentile latency. Apache Kafka was used as a message system and Redis was used for storage. The benchmark simulates an advertisement analytics pipeline with the following steps: reading event data from Kafka, deserializing JSON messages, filtering out irrelevant events, taking a projection of the relevant fields, joining events, storing the information in Redis and finally taking a windowed count and storing each window in Redis. The pipeline we designed for our benchmark follows this one closely, although we will use a different methodology for extracting latency measurements.

In 2017, Karakaya et al. [14] extended the Yahoo Streaming Benchmark by measuring resource usage and performance scalability against a varying number of cluster sizes. They found that Flink outperforms Spark and Storm under equal constraints. Latency measurement was not included in their setup.

Shukla and Simmhan [3], proposed an IoT benchmark for distributed stream processing systems, based on common IoT micro-benchmark tasks and two IoT applications for statistical analysis and predictive analytics. This benchmark later evolved into RIoTBench [15]. The depth of the benchmark was increased by adding new tasks and new applications to the existing ones. Furthermore, four real-world streams with different distributions were used for evaluation. Five metrics were incorporated: end-to-end latency, peak throughput, memory usage, CPU usage and jitter. The benchmark was tested for Apache Storm.

In [16], characteristics of Spark and Storm were analyzed and the latency of performing certain tasks was compared. Three different tasks were executed: WordCount, Grep and Top K Words. Latency was defined as the time it took to process 10 000 000 records. As a second part of the paper, the latency of subparts of the pipeline was measured by looking at the DAG generated by the frameworks. For measuring these subparts, the jobs were run on a single machine. We will also measure the latency of separate stages of our pipeline, however, we will run the jobs on a cluster to mimic big deployments more closely.

Previous work has put no emphasis on analyzing separate operations and has ignored the complexity of accurate latency measurement in distributed systems. In the following section we will lay out how our methodology mitigates this.

## III. Benchmark Definition

### A. Processing Pipeline

In order to investigate the latency differences among common stream processing operations, we implemented a processing pipeline resembling that of [5]. It aims to compute the evolution of traffic intensity at the traffic measurement locations. The stages included in the pipeline are the following (as shown by Fig. 1):

1) Ingest or baseline: consuming speed and flow measurements from Kafka. No operations are done on the ingested data. This stage will serve as our baseline for the other stages.
2) Parse: parsing the JSON data.
3) Join: joining speed and flow measurement streams on timestamp, road lane and measurement location.
4) Aggregation: tumbling window of one second to compute the total amount of cars and average speed per measurement location.
5) Sliding window: sliding window with a ten-second lookback period and a trigger interval of one second to compute the short-term (5 seconds) and long-term (10 seconds) relative change in traffic intensity.

At the end of each stage, the data is published to Kafka.

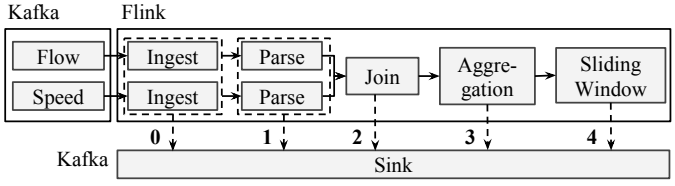


Fig. 1. Processing flow.

### B. Latency Measurement Methodology

As stated in [3], the latency of an output message is the time that it took for a task to process one or several inputs in

order to generate its output message. We compute latency as the difference between the Kafka consumer record timestamps of the incoming records and the processed records. If multiple input messages are required to generate one output message, we take the average of the latencies of each input message. We first compute the latency of merely ingesting from Kafka and publishing back to Kafka, which we call the baseline (cf. label 0 in Fig. 1). We then add one stage at a time to extract the latency incurred from adding an extra stage (cf. labels 1 to 4 in Fig. 1). By doing this, we are able to observe the latency increase of adding a specific stage to the pipeline.

The workload applied to the framework is specifically designed for accurate latency measurement. The data stream generator (cf. Fig. 2) publishes the input data stream to Kafka. Every partition of a Kafka topic has one broker which is elected as the leader. The leader takes in the new data published on that partition and assigns the timestamp to the consumer record. Different partitions can elect different brokers as leader and since these brokers are not necessarily on the same machine, timestamps may again not be comparable. To prevent this, we will make sure that all partitions of the input and output topics elect the same broker as leader. By doing this, we make sure time is always captured on the same machine, guaranteeing correct latency measurements. It is important to note that our Kafka cluster will still comprise five brokers to keep our setup as realistic as possible. The other brokers are now merely used for data replication. Finally, we make sure that latency is tested under sustainable throughput conditions.

### C. Architecture

An overview of our architectural set up has been given in Fig. 2. To run our experiments we set up a DC/OS cluster for resource management. The three master nodes of the DC/OS cluster each run with 4 vCPU's and 32GB RAM. The cluster has 12 worker nodes each consisting of two Intel Xeon X5670 processors (2x 6 cores with hyper threading = 24 vCPU) with 96GB RAM and two 300GB RAID1 hard drives.

The benchmark consists of seven main components that are all running in containers on DC/OS (as shown in Fig. 2):

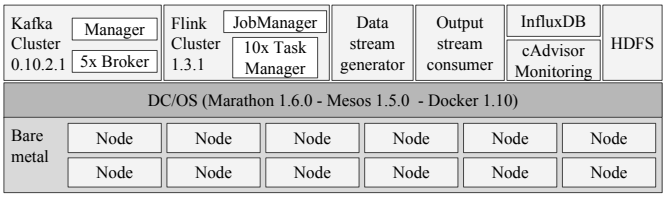


Fig. 2. Benchmark Architecture.

- *Data stream generator*: publishes input data on Kafka.
- *Kafka cluster*: messaging system consisting of five brokers. Each broker runs in a Docker container with 4 CPU's and 16GB RAM.
- *Flink cluster*: cluster with one job manager and ten task managers, each with twelve task slots. Each task manager has 36 GB memory (12 GB heap) and 12 CPU's.
- *Output consumer*: writes the processed data to HDFS.
- *Evaluator*: computes the latency distributions by reading in the processed data from HDFS. This is run as a batch job after the entire stream has been processed and is, therefore, not present in Fig. 2.
- *cAdvisor and InfluxDB*: cAdvisor monitors the containers and stores CPU and memory metrics in InfluxDB.
- *HDFS*: for data storage, we use HDFS with 10 data nodes and 150 GB disk allocated to each node.

No other workloads were active while benchmarking.

### D. Data

IoT use cases often require outcomes to be generated as fast as possible and at a massive scale. The requirement for high throughput and low latency makes it an interesting use case for testing stream processing frameworks. Furthermore, by using real-world data we will be able to mimic realistic usage of the stream processing frameworks. For this benchmark we will use traffic sensor data originating from the Nationale Databank Wegverkeersgegevens (NDW) [17]. Sensors at every measurement location in the Netherlands publish one minute aggregates of the amount of cars (flow measurements) and the average speed of the cars (speed measurements) that passed on each lane of the road.

To ensure thorough testing of our framework, we make use of a data publisher that simulates the traffic data at a higher speed and granularity using temporal and spatial scaling [15]. Per second we publish 26 000 observations to the Kafka speed and flow measurements topics. The size of one observation lies between 180 and 200 bytes.

## IV. Apache Flink

We use Apache Flink, a distributed, open-source stream processing framework [18], to validate the benchmarking methodology. For distributed execution, Flink chains operator subtasks together into tasks. We ensure that each of the stages of our processing flow translates to one task. Each of these tasks is executed on one thread to enhance the efficiency of parallel computation [19]. Apache Flink works with a master-slave architecture in which the masters are called Job Managers and the slaves are called Task Managers.

## V. Results

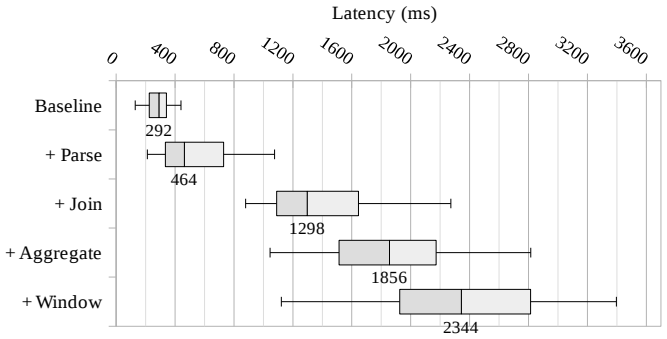


Fig. 3. Cumulative latency of the processing pipeline for Apache Flink.

After running the benchmark for Apache Flink, we find the following results, as visualized in Fig. 3. The median latency of merely executing the baseline is 292 ms. We suppose this time was mainly spent on network transfer and queueing in Kafka. After computing the baseline, we add the parsing stage to the pipeline which leads to a median latency increase by 172 ms. Subsequently, adding the join stage leads to the highest latency increase of all stages by a median value of 835 ms, from 464 ms to 1298 ms. Joining the speed and the flow stream takes more than 4.5 times longer than parsing the incoming data. Additionally, the tail of the distribution becomes longer, thereby increasing the variability of how long it can take to process an observation. The difference between the 75th and 95th percentile was 81 ms for baseline-parse execution while this rose to 279 ms after adding the join operation. When designing a processing pipeline, joining streams should be avoided where possible. Aggregating the data using a tumbling window of one second increases the latency by 558 ms. The aggregation stage was expected to have a latency higher than 500 ms, due to the tumble interval that needs to finish before output is generated. The final stage of the pipeline is a sliding window in which the long-term and short-term evolution of the speed are computed. The sliding window has a lookback period of ten seconds (ten minutes in the data due to temporal scaling). Computing the relative change adds a median latency of 488 ms, leading to a median end-to-end latency of 2344 ms. We observe the latency becoming more dispersed as we add stages to the processing flow. After adding the last stage, the 5th percentile was at 1121 ms and the 95th percentile at 3397 ms, a difference of more than 2000 ms.

## VI. Conclusion

In conclusion, the described setup allows latency benchmarking of separate stages of a processing pipeline. This can be used for fine-grained framework comparisons and to derive insights for more conscious processing pipeline design. Furthermore, we enforce correct latency measurements by capturing time on one machine. Running the benchmark for Apache Flink brings to light that joining data streams takes about 4.5 times longer than merely parsing the data. Joins should, therefore, be avoided where possible. Additionally, the latency becomes more dispersed after adding subsequent stages. The code for this project can be found at http://www.bigdata.ugent.be/benchmark.htm.

## VII. Limitations and Further Research

Further research could use this work to benchmark Apache Flink against other frameworks such as Apache Spark. Other frameworks might show different results and patterns. Furthermore, other performance metrics, e.g. throughput and resource usage, could be added to the benchmark. Finally, our approach allows more accurate benchmarking of latency, but due to the architectural changes that had to be made, it would not be a good setup for measuring other metrics such as throughput.

## VIII. Acknowledgment

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

## References

[1] 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.

[2] 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.

[3] 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.

[4] A. D. Ferguson, P. Bodik, S. Kandula, E. Boutin, and R. Fonseca, "Jockey: Guaranteed job latency in data parallel clusters," in *Proceedings of the 7th ACM European Conference on Computer Systems*, ser. EuroSys '12. New York, NY, USA: ACM, 2012, pp. 99–112.

[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] R. Ostrovsky and B. Patt-Shamir, "Optimal and efficient clock synchronization under drifting clocks," in *Proceedings of the eighteenth annual ACM symposium on Principles of distributed computing*. ACM, 1999, pp. 3–12.

[7] J. C. Corbett, J. Dean, M. Epstein, A. Fikes, C. Frost, J. J. Furman, S. Ghemawat, A. Gubarev, C. Heiser, P. Hochschild *et al.*, "Spanner: Googles globally distributed database," *ACM Transactions on Computer Systems (TOCS)*, vol. 31, no. 3, p. 8, 2013.

[8] D. L. Mills, "Internet time synchronization: the network time protocol," *IEEE Transactions on communications*, vol. 39, no. 10, pp. 1482–1493, 1991.

[9] M. Maróti, B. Kusy, G. Simon, and Á. Lédeczi, "The flooding time synchronization protocol," in *Proceedings of the 2nd international conference on Embedded networked sensor systems*. ACM, 2004, pp. 39–49.

[10] J. Elson, L. Girod, and D. Estrin, "Fine-grained network time synchronization using reference broadcasts," *ACM SIGOPS Operating Systems Review*, vol. 36, no. SI, pp. 147–163, 2002.

[11] L. Lamport, "Time, clocks, and the ordering of events in a distributed system," *Communications of the ACM*, vol. 21, no. 7, pp. 558–565, 1978.

[12] 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.

[13] 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.

[14] 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.

[15] 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.

[16] P. Córdova, "Analysis of real time stream processing systems considering latency," University of Toronto, Tech. Rep., 2014.

[17] "NDW: Nationale Databank Wegverkeersgegevens," http://www.ndw.nu/, 2017, [Online; accessed 25-July-2017].

[18] "Apache Flink: Flink Programming Guide," https://ci.apache.org/projects/flink/flink-docs-release-1.3/, 2017, accessed: 25-July-2017.

[19] "Apache Flink Docs: Distributed Runtime Environment," https://ci.apache.org/projects/flink/flink-docs-release-1.3/concepts/runtime.html, 2017, [Online; accessed 25-July-2017].