Benchmarking Stream Processing Frameworks for Large-Scale Data Shuffling

Sören Henning[1,2], Adriano Vogel[1,2], Michael Leichtfried[2], Otmar Ertl[2], Rick Rabiser[1]
[1]Co-Innovation Lab LIT CPS Lab, Johannes Kepler University Linz, Austria
[2]Dynatrace Research, Linz, Austria
{firstname.lastname}@{jku.at, dynatrace.com}

Abstract

Distributed stream processing frameworks help building scalable and reliable applications that perform transformations and aggregations on continuous data streams. We outline our ongoing research on designing a new benchmark for distributed stream processing frameworks. In contrast to other benchmarks, it focuses on use cases where stream processing frameworks are mainly used for re-distributing data records to perform state-local aggregations, while the actual aggregation logic is considered as black-box software components. We describe our benchmark architecture based on a real-world use case, show how we implemented it with four state-of-the-art frameworks, and give an overview of initial experimental results.

1 Introduction and Background

Distributed stream processing frameworks such as Spark, Flink, Kafka Streams, or Hazelcast (Jet) have gained widespread adoption over the last years not only for building data analytics pipelines, but also for implementing core business logic in software-based organizations [1]. Such frameworks provide primitives and high-level APIs to express directed acyclic processing graphs that filter, transform, aggregate, and merge data streams. Accompanied by sophisticated semantics for time windows and key-based joins, stream processing frameworks thus offer a programming model to design and implement large-scale distributed applications. While this surely has many advantages, there are cases where building entire applications this way does not work. For example, squeezing complex business logic or existing software components into such programming models and architectures can be difficult. Nevertheless, stream processing frameworks can still be helpful to build applications with mainly custom logic as they provide, for example, abstractions for cluster management, means to scale out the data processing, fault-tolerant state management, well-defined processing guarantees (e.g., exactly-once or at-most-once), and a rich ecosystem of documentation, support, and associated tooling.

Several papers on benchmarking stream processing frameworks have been published [6]. However, they usually focus on specific use cases utilizing the wide range of framework features including complex dataflow graphs and window semantics. It has been shown that the performance of such frameworks highly depends on the use case [4] and, hence, it can be challenging to map the result of general purpose benchmarks to production systems [5]. With this paper, we give an insight into our ongoing research on evaluating the performance of different stream processing frameworks. We introduce *ShuffleBench*, our proposal for a new stream processing benchmark focusing on the use of stream processing frameworks for *shuffling* (i.e., re-partitioning) data streams to perform state-local aggregations. In contrast to many other benchmarks, it is inspired by requirements of a large real-world software system and can be configured for various use cases of large-scale data shuffling.

2 The ShuffleBench Benchmark

Real-world Inspiration Our benchmark is designed to address the requirements for continuous dashboard queries and real-time alerting of a market-leading cloud observability platform. Using a powerful query language, internal and external clients can define complex rules to aggregate and correlate different data sources such as metrics, events, logs, and traces.

Stream processing frameworks are used to filter, transform, aggregate, or correlate data streams in near real time. The actual processing logic can be considered as queries on data streams, either defined with programmatic APIs or via dedicated SQL-like languages. With ShuffleBench we look into use cases in which a high amount of queries have to be executed in parallel, although each query only requires a very small portion of the overall data volume. Therefore, it is not required to parallelize or even distribute the execution of a single query, but to only route those data records to a query instance as needed. Moreover, dedicated software components such as anomaly detection models might already include the logic for operations such as joins or sliding windows such that these features are not required by the framework.

Benchmark Dataflow Architecture A well-suited way to meet the listed requirements is

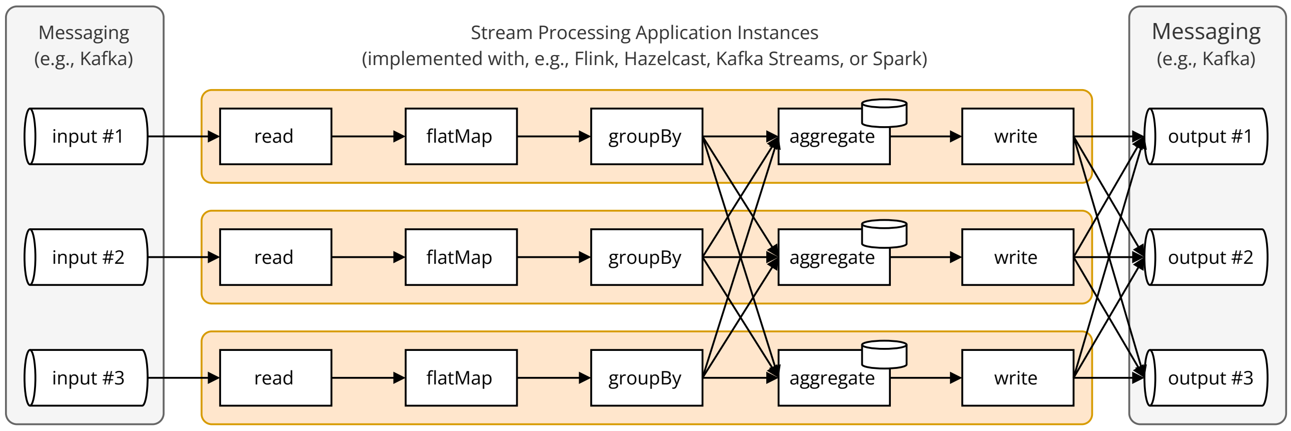


Figure 1: The *ShuffleBench* dataflow architecture at runtime for three stream processing application instances.

a MapReduce-like architecture on continuous data streams as it can be built with modern stream processing frameworks. Figure 1 depicts our benchmark architecture for a corresponding stream processing application. It can be deployed with multiple instances, which execute the same processing logic, but on different data subsets. The data processing starts by reading data records from a messaging system such as Kafka. Kafka topics are partitioned, allowing each instance of the stream processing application to subscribe to a dedicated set of partitions. After ingestion, the *matcher service* finds the relevant queries for each record. The matcher logic is wrapped in a *flatMap* operation of a stream processing framework that duplicates an incoming record for each relevant query, while assigning a query-identifying key to each duplicate. In a subsequent operation, the data is repartitioned among all instances such that all records which have the same query key assigned are forwarded to the same instance. This is done with an operation of the stream processing framework often called *groupBy*. In the next step, the actual black-box query logic is executed, which is stateful by aggregating multiple records. We abstract the query logic in *real-time consumers*, wrapped in an *aggregate* operation of the stream processing framework that manages the state. These real-time consumers adhere to a simple interface: They consume an incoming record and the previous state and output the updated state and, optionally, an alerting event. Finally, these alerting events are written to another Kafka topic.

Benchmark Implementations We implemented the proposed benchmark for the four stream processing frameworks Flink, Hazelcast, Kafka Streams, and Spark Structured Streaming; as well as a corresponding load generator. To provide a fair comparison, the implementation of the matcher service and the real-time consumers (which would be domain-specific in production) are shared among all frameworks.

To simulate incoming observability data, our load generator generates data records at configurable frequency with random byte content of configurable size. It can be deployed in a distributed fashion and writes the generated records to Kafka.

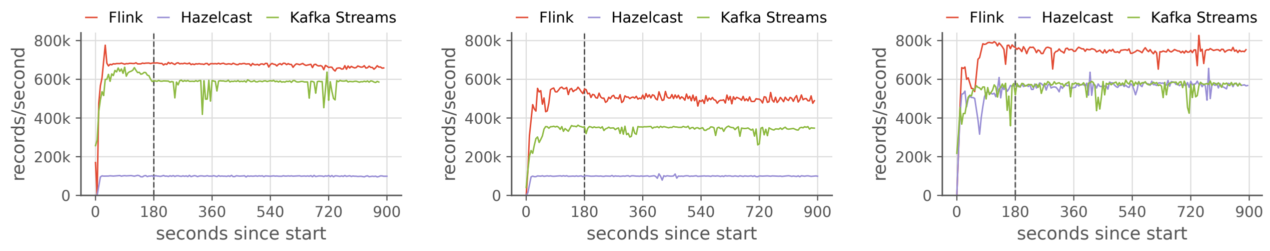
The matcher service is configured with a set of rules, which define their selectivity, i.e., the probability that a record is matched by this rule. The actual stateful aggregations logic is currently the same for each query: We count the number of received records and generate an alerting event if the count is dividable by a configured value.

Benchmark Configuration Options Our ShuffleBench implementations are highly configurable to evaluate frameworks for different use cases of large-scale data shuffling tasks. This includes the size of incoming records, the number of different real-time consumers, the total selectivity for all real-time consumers as well as the distribution of individual selectivities, the size of the aggregation state, and the output frequency of the real-time consumers. Additionally, all stream processing frameworks have a wide range of configuration options that impact throughput, latency, scalability, and fault-tolerance.

Automated Benchmark Execution We utilize and extend the Theodolite benchmarking framework [2, 3] to automate the benchmark execution in Kubernetes-based cloud environments. This includes the declarative definition of benchmark experiments, automated setup and teardown of all involved software components (i.e., stream processing frameworks, load generator, and Kafka) as well as the collection of measurement data.

3 Initial Experimental Results

We conduct initial experiments with our proposed benchmark to evaluate the throughput of the stream processing frameworks Flink, Hazelcast, and Kafka Streams. The benchmark setup is deployed on an AWS-managed Kubernetes cluster, consisting of 9 nodes: 3 *m6i.xlarge* nodes run the stream processing framework, 3 *m6i.2xlarge* nodes run one Kafka bro-



(a) 10k consumers, 1024 bytes/record (b) 1M consumers, 1024 bytes/record (c) 1M consumers, 128 bytes/record

Figure 2: Throughput of Flink, Hazelcast, and Kafka Streams benchmarked with *ShuffleBench*.

ker each, and 3 *m6i.xlarge* nodes run the load generator instances plus additional benchmarking infrastructure. We generate a constant rate of 750 000 records per second for 15 minutes and monitor the number of records per second processed by the stream processing framework. The stream processing application is deployed with 9 single-CPU application instances (3 per cluster node), resulting in a total parallelism of 9. Except for a few adjustments for better comparability, we test all frameworks with their default configurations. In all experiments, the selectivities for all real-time consumers sum up to 20 % and each consumer emits an altering event for every tenth record.

Figure 2 shows the throughput results for different experiments. We can see that in all experiments the observed throughput is quite stable after a warm-up period of at most 3 minutes (dashed gray line). Figure 2a shows the throughput for 10 000 real-time consumers and records of 1024 bytes. We observe that Flink achieves the highest throughput, followed by Kafka Streams with a 13.8 % lower throughput. Hazelcast shows a significantly lower throughput, which is 85.2 % lower than Flink.

Fig. 2b shows the results for repeating the experiments but with one million real-time consumers. Increasing the number of consumers increases the managed state (but not the total number of state updates) as well as the runtime of the *flatMap* processing step. We observe that Flink's throughput decreases by only 25.5 % compared to 10 000 consumers, while Kafka Streams' throughput decreases by even 40.8 %. Hazelcast's throughput does not change, which indicates that the bottleneck in Hazelcast's processing is unrelated to the number of real-time consumers.

Considering that performance losses in Hazelcast compared to other frameworks were not observed in the related literature [4, 6], we conduct a further experimental scenario with one million real-time consumers, but smaller sized records of 128 bytes. Figure 2c shows that with smaller records Hazelcast's performance increases significantly, achieving a processing rate similar to Kafka Streams. Moreover, Flink again achieves the highest throughput by processing records at the same rate as they are generated, which indicates that even higher throughputs are achievable.

4 Conclusions and Road Ahead

With ShuffleBench, we propose a new benchmark to assess the performance of stream processing frameworks for stateful aggregation use cases. While this paper already shows initial throughput results for Flink, Hazelcast, and Kafka Streams, we now work on detailed evaluations of different framework and use case configurations for qualities such as throughput, latency, fault tolerance, scalability, and operational costs. This encompasses an in-depth investigation of the fundamental causes behind the noted variations in performance. As one of the next steps, we will open-source our implementations to engage other researchers and practitioners to join us in this research.

Acknowledgments We would like to thank JKU and Dynatrace for co-funding this research.

References

[1] P. Carbone et al. “Beyond Analytics: The Evolution of Stream Processing Systems”. *International Conference on Management of Data*. 2020.

[2] S. Henning and W. Hasselbring. “Theodolite: Scalability Benchmarking of Distributed Stream Processing Engines in Microservice Architectures”. *Big Data Research* 25 (2021).

[3] S. Henning and W. Hasselbring. “A Configurable Method for Benchmarking Scalability of Cloud-Native Applications”. *Empirical Software Engineering* 27.6 (2022).

[4] S. Henning and W. Hasselbring. “Benchmarking scalability of stream processing frameworks deployed as microservices in the cloud” (2023). DOI: 10.48550/arXiv.2303.11088.

[5] J. Rank, A. Hein, and H. Krcmar. “The Role of Performance in Streaming Analytics Projects: Expert Interviews on Current Challenges and Future Research Directions”. *Softwaretechnik-Trends* 43.1 (2023). (SSP 2022).

[6] A. Vogel et al. “A systematic mapping of performance in distributed stream processing systems”. *Euromicro Conference on Software Engineering and Advanced Applications*. In press. 2023.