Cost of Fault-Tolerance on Data Stream Processing

Valerio Vianello[1(✉)], Marta Patiño-Martínez[1], Ainhoa Azqueta-Alzúaz[1], and Ricardo Jimenez-Péris[2]

[1] Universidad Politécnica de Madrid, Madrid, Spain
{vvianello,mpatino,aazqueta}@fi.upm.es
[2] LeanXcale, Madrid, Spain
rjimenez@leanxcale.com

Abstract. Data streaming engines process data on the fly in contrast to databases that first, store the data and then, they process it. In order to process the increasing amount of data produced every day, data streaming engines run on top of a distributed system. In this setting failures will likely happen. Current distributed data streaming engines like Apache Flink provide fault tolerance. In this paper we evaluate the impact on performance of fault tolerance mechanisms of Flink during regular operation (when there are no failures) on a distributed system and the impact on performance when there are failures. We use the Intel HiBench for conducting the evaluation.

Keywords: Data streaming · Fault tolerance · Evaluation · HiBench

1 Introduction

Data streaming has become a popular data processing model in the last decade with the increase of the amount of data that is produced every second that must be processed on the fly. Typical examples of streaming applications include quick detection of price changes in the stock market, credit card fraud detection, detection of attacks by inspecting network traffic among others. Data streaming engines run on top of distributed systems in order to process the high volumes of data produced every second (from thousands to millions of events per second). Distributed streaming engines like StreamCloud [8], Borealis [4] and Flink [1] have incorporated fault-tolerance mechanisms in order to ensure the availability of the system when a failure happens. Fault-tolerance mechanisms resort to checkpointing the state of the data streaming application and the data streams in order to be replayed when the system recovers after the failure, ensuring that all the data is processed. In this paper, we evaluate the performance overhead that the fault-tolerance mechanisms introduce during regular operation running the Intel HiBench benchmark [10] with Flink on top of a distributed system. We also evaluate the time the system needs to resume regular processing and the impact on performance till the system returns to regular operation (the

© Springer Nature Switzerland AG 2019
G. Mencagli et al. (Eds.): Euro-Par 2018 Workshops, LNCS 11339, pp. 17–27, 2019.
https://doi.org/10.1007/978-3-030-10549-5_2

system processes all the queued records during the failure). These performance evaluation is important for practitioners in order to dimension the system when fault-tolerance mechanisms are present and understand the behavior of the system when it recovers.

The rest of the paper is organized as follows. First we present an introduction to Flink (Sect. 2) then, the fault-tolerance mechanisms of Flink are described (Sect. 3). Section 4 presents the performance evaluation. Finally conclusions are presented in Sect. 5.

2 Flink Architecture

Apache Flink is an open-source distributed and fault-tolerant stream processing framework. A Flink program transforms the incoming data streams and returning results through sinks that can write them to different destinations. The transformations are also known as operators and the set of operators linked by the incoming and outgoing data streams form a topology that logically is a DAG (directed acyclic graph).

Flink provides several built-in operators which can be classified as stateless or stateful. Stateless operators do not keep any state. They simply transform the incoming data. Examples of stateless operators are *map, filter and union.* Stateful operators keep events in memory (windows) apply a function and produce an output (time windows) or a number of records are received (record based windows). Examples of stateful operators are *fold, aggregates, join.*

At the core of the Flink architecture there are two components that are JobManager and TaskManager. The JobManager is the master of a Flink cluster. More than one JobManager can be started in a Flink cluster to provide high availability. The JobManager is not directly involved in data processing, it is in charge of coordinating the distributed execution. The TaskManager runs topologies (or part of them) and manages the data exchange using streams.

Figure 1 shows how a client application (Flink Program) runs on a Flink cluster made by one JobManager and two TaskManagers. Each TaskManager (process) has its own Memory and Network Manager and can be configured with several task slots. On one hand, task slots are used to split and isolate TaskManager dedicated memory for different topologies. On the other hand, they fix the maximum number of concurrent sub-task (part of a topology) that can be running on a given TaskManager. In Fig. 1, TaskManagers are configured with three task slots, it means that three sub-tasks from three different topologies can be executed by the TaskManager. It is worth noting that Flink allows the deployment of different sub-tasks of a given topology to share the same task slot. The JobManager keeps track of the registered topologies and their corresponding dataflow graph. It also schedules the tasks and decides on which TaskManager they are executed. On the client side, a Flink program is used to build an optimized dataflow graph from the topology and deploy it on the Flink cluster sending it to the JobManager.

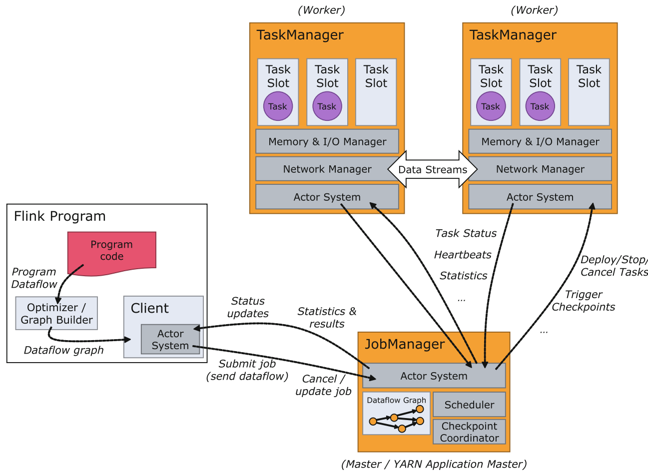


Fig. 1. Apache Flink runtime [7].

3 Fault Tolerance

Fault tolerance in Flink [6] is based on durable data sources and state checkpointing. A durable data source is able to replay records from a specified point in time in the past. Typically, a durable data source reads records from a persistent messaging system, such as Apache Kafka [3] or RabbitMQ [12], so in case a failure happens Flink can go back in time and re-read the input streams. Flink uses state checkpointing to save the state of topologies into a persistent storage. This state is recovered in case of failures. The persistent state must be accessible by all JobManagers and TaskManagers running in the Flink cluster in order to recover the state after a failure, hence a distributed filesystem, such as Hadoop Distributed File System [2], can be used for this purpose. This approach is similar to the one in [11]. Flink allows users to set different parameters to tune the checkpointing duration like the time between two consecutive checkpoints, the maximum time to wait for a checkpoint to be completed, the number of stored checkpoints.

A snapshot of an operator is taken when a special tuple, called barrier, is received from all its input streams. Then, the operator sends the barrier in all its outgoing streams. The JobManager injects the barriers in the streams at the data sources in order to take a distributed consistent snapshot. When a sink receives barrier n from all its incoming streams, it informs the snapshot coordinator. When the snapshot coordinator (the JobManager) receives this message from all the sinks in the topology, the n-th snapshot is completed. The snapshot can be taken synchronously or asynchronously. The former has an impact on the performance. If the snapshot is taken asynchronously, the state is copied as a background process and the operator immediately sends the barrier in its output streams. Once the state is copied, the operator informs the snapshot coordinator.

A snapshot is considered complete when the coordinator is informed by all sinks that they have received the corresponding barriers and stateful operators have completed their backup. At this point, the state at the sources corresponding to that snapshot will never be needed again.

When a failure happens, and the JobManager detects that one of the TaskManagers is not available, the affected topology is undeployed and a new deployment is scheduled on the available task slots. The JobManager cannot re-deploy the topology if there are not enough task slots left, in this case the topology is suspended until new TaskManagers join the Flink cluster making available their task slots. After a redeployment, the latest completed snapshot is selected (n). The state for checkpoint n is read from persistent storage and the streams are resent from the n offset.

Flink ensures at least once semantics. That is some tuples may be processed more than once. That is, records sent after the latest completed snapshot might be processed more than once.

Flink has recently introduced end-to-end exactly once semantics, where each incoming event affects the final result exactly once. For this purpose Flink uses a two-phase commit protocol that together with new special sink components, durable data source and checkpoint is able to ensure that there are no duplicate results in case of failures happen [5].

4 Evaluation

The goal of the performance evaluation is to evaluate the overhead that the fault-tolerance introduces in a regular processing and the cost of recovery. For this purpose the *HiBench* big data benchmark is used [9] and deployed in a cluster.

4.1 Benchmark

The *Hibench* provides a set of topologies already implemented for Apache Flink among them we picked the one that has window operators (*Fixwindow*) in order to test the performance of window operations in streaming frameworks. The benchmark creates records representing the visits of users to a web server. Each record has a total of 200 bytes and among the other fields it includes a timestamp taken at record creation time and the IP address of the client. Figure 2 depicts the graph representing the *Fixwindow* topology. The *Kafka source* source operator fetches records from the remote Kafka server. The *Map* operator projects *Timestamp* and *IP* fields of records from the input stream to the output stream ones. *KeyBy* partitions the stream using the *IP* field. *Window* stores events from each partition for a given amount of time. *Reduce* counts the elements in the window and emits one record with the *IP*, oldest *Timestamp* among the records in the window, and the number of elements in the window. The second *Map* operator adds a *Timestamp* to the record and writes it into Kafka.

The benchmark evaluates the latency of the operation calculating for each output record the difference in time between the *Timestamp* added in Flink by the latest Map in the topology and the *Timestamp* added by the benchmark at record creation time.

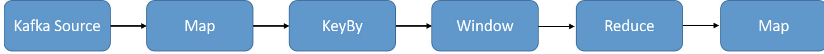


Fig. 2. HiBench fixwindow topology.

4.2 Setup

The evaluation was performed in a cluster with six homogeneous nodes. Each node is equipped with 2 CPU sockets with Intel XEON E5-2620 v3 with 6 cores (12 virtual cores), a total of 24 virtual cores, 128 GB RAM divided into 8 slots. Each slot contains a 16 GB RAM card. Each node is equipped with a directly attached SSD Intel SD3510 480 GB. All of them connected by a 1Gbit Ethernet. The software running on the nodes is: Intel HiBench 7.0, Flink 1.4.2, Kafka 2.10-0.8.2.2, Hadoop 2.6.5 and Zookeeper 3.4.8. Figure 3 shows where this software is running. *Node1* runs the HiBench benchmark. We used from 2 to 5 instances of the benchmark to increase the load. *Node2* runs HDFS to store Flink checkpoints and the HiBench data seed and Zookeeper for coordinating the Kafka cluster and the JobManager of Flink. *Node3* and *Node4* run 6 Kafka Brokers each. *Node5* and *Node6* run 12 JobManagers each. JobManagers are configured with 2 task slots (for a total availability of 48 task slots) and 8 GB of memory.

The experiments are run with different configurations and loads summarized in the Table 1. Varying the number of HiBench instances generates loads from 200,000 records per second up to 500,000 records per second. We ran experiments with and without Flink checkpointing mechanism in order to measure the overhead of the checkpointing mechanism during regular operation. Checkpoints are taken every second and stored in HDFS. Later, failures are injected in both configurations and the time for recovery is measured.

Table 1. Experiments configurations.

Input load (r/sec)	Window size	Checkpointing	Fault injection
200k–500k	50 Records	No	No
200k–500k	30 to 50 Records	HDFS	No
200k–500k	30 to 50 Records	HDFS	Yes
200k–500k	50 Records	HDFS + RocksDB	No
200k–500k	50 Records	HDFS + RocksDB	Yes

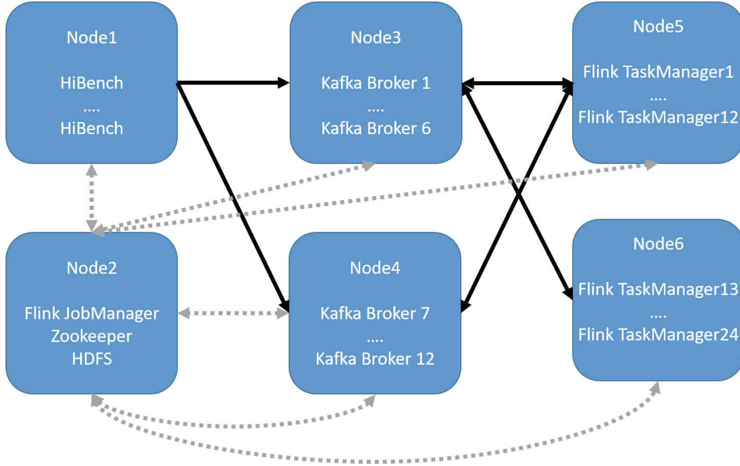


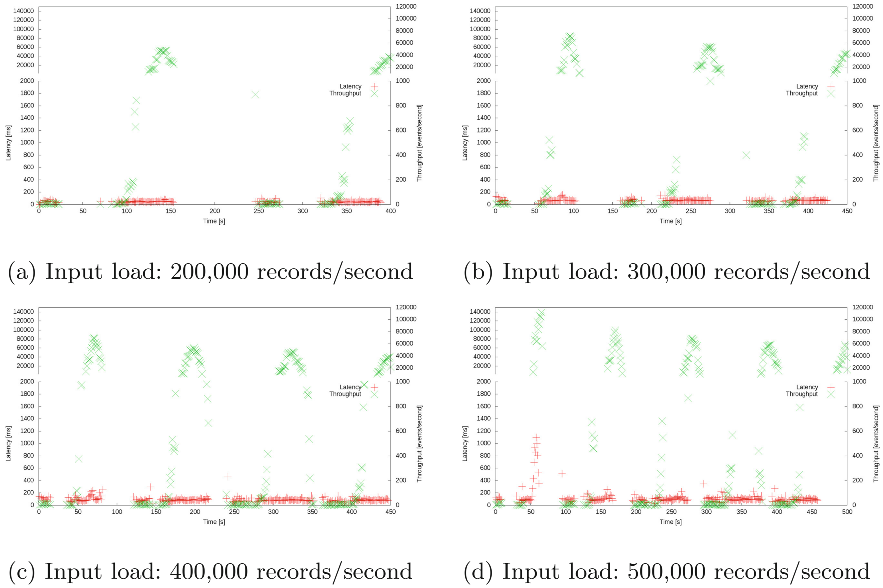
Fig. 3. Evaluation setup.

4.3 Performance Evaluation Results

For all experiments we show measure both latency (on the left axis) and throughput (on the right axis). The latency represents the difference in time between the timestamp of the newest record that falls in a window and the timestamp taken when the result record of the window is generated; it is measured in milliseconds and we report the mean value per second. The throughput shows the number of result records created per second, that is the number of windows that are triggered per second. The x-axis shows the time evolution during an experiment. Second 0 in the x-axis corresponds to the first output received from Flink. First, we run experiments without the checkpointing mechanism. Figure 4 reports the results of these experiments with four different loads.

In all cases, Flink is able to process the load with a very low latency that is always smaller than 200 ms. The maximum throughput is around 40K, 70K, 80K and 100K records per second for the increasing load. This maximum is reached twice with a load of 200K records per second (Fig. 4a), three times with 300K (Fig. 4b) and four times with 400K and 500K per second (Figs. 4c and d). These peaks happen because the load is increased by adding more HiBench instances but, the key space remains the same causing the same windows (there is a window per key) to be triggered more times. As the load increases, windows are filled at a faster pace.

Figure 5 shows the experiments with the checkpointing mechanism enabled in Flink and the same workloads. Comparing Figs. 4a and 5a we observe that the latency of the window processing with the checkpointing mechanism enabled is almost equal to the baseline case. This happens because Flink stores the snapshot of the state asynchronously and if the load is not too high it is able to perform both operations without a noticeable penalty on the latency. However, as the load increases, the latency increases up to 1 second with a load of 300K (Fig. 5b) and



(a) Input load: 200,000 records/second (b) Input load: 300,000 records/second

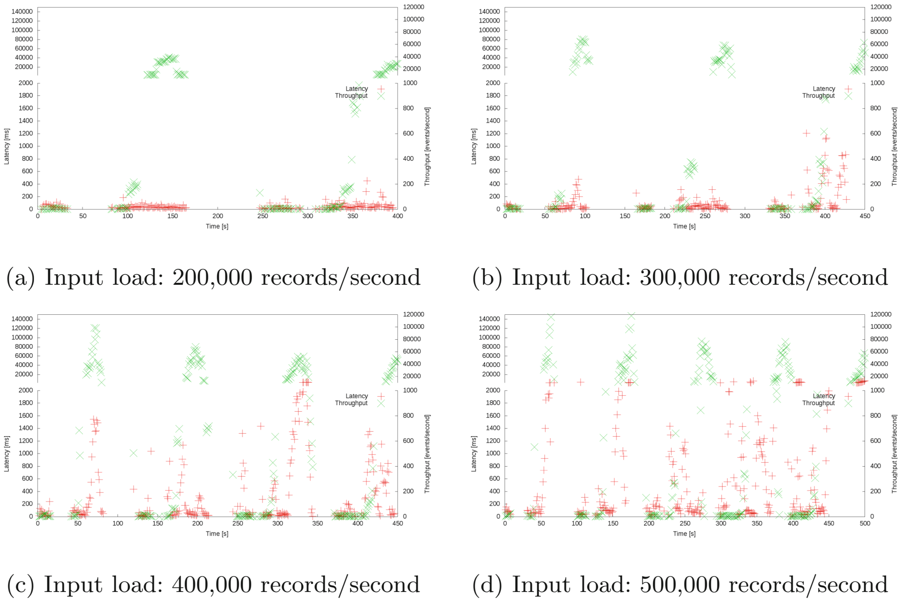
(c) Input load: 400,000 records/second (d) Input load: 500,000 records/second

Fig. 4. Experiments results with checkpointing disabled.

up to 2 s when the load increases to 400k and 500k records per second (Fig. 5c and d). This happens because there are more concurrent windows to checkpoint and taking and storing the snapshot consumes CPU cycles that cannot be used to process the input load and therefore, the processing time of records increases.

Figure 6 shows the CPU utilization per core in one of the two nodes used for running Flink in the experiments with 200k and 500k record per second checkpointing the state to HDFS. It can be observed that with 200K the CPU usage is on average 40% while with a load of 500K the system is almost saturated with 70% CPU usage on average.

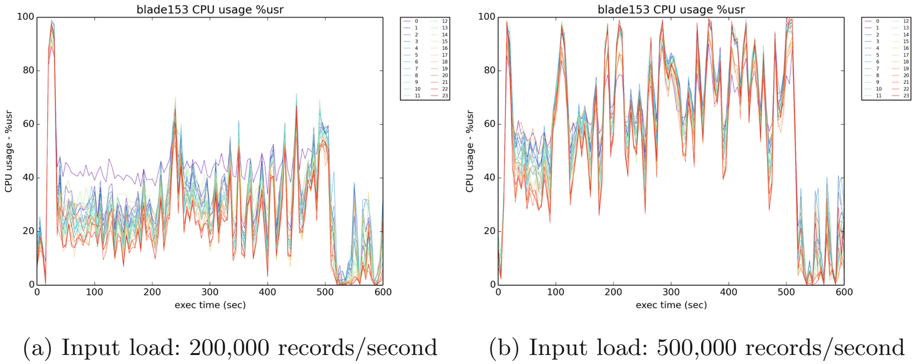
Figure 7 presents the results of the experiments with failures in order to measure impact of failures when the system recovers. The fault is injected by killing one of the TaskManagers running the topology 90 s after the first outputs are produced. Flink takes around 90 s to detect the failure and resume processing that is, detect the failure of the TaskManager, undeploy the topology, redeploy the topology on the available task-slots, load the state and restart the normal processing. During that period there is no throughput (Figs. 7a, b and c). Then, the latency is very high in all setups: up to 1 min with a load of 200 records per second and reaching up to 2 min with the other configurations. This happens because the data needs to be resent from the source and there are a lot of data that are waiting to be processed while the system recovers. These data are processed in 60 s with a load of 200K records (after second 210 latency is below 200 ms), 150 s with a load of 300K records, 170 s for a load of 400K. The system



(a) Input load: 200,000 records/second (b) Input load: 300,000 records/second

(c) Input load: 400,000 records/second (d) Input load: 500,000 records/second

Fig. 5. Performance with checkpointing on HDFS

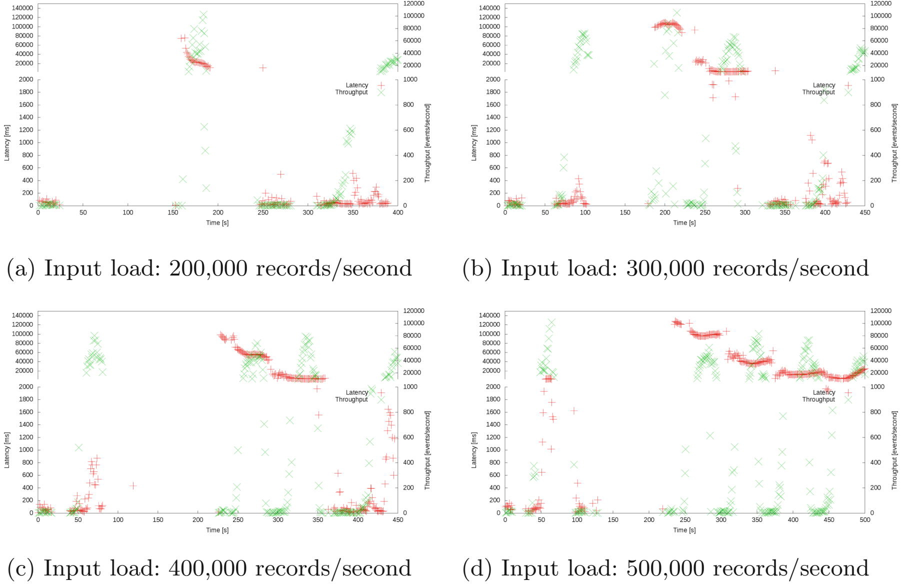


(a) Input load: 200,000 records/second (b) Input load: 500,000 records/second

Fig. 6. CPU utilization on one of the Flink nodes

is not able to return to regular latencies after 260 s with a load of 500K records, showing latencies higher than 20 s during that period.

Figure 8 reports the CPU usage per core in the two nodes running Flink when the input load rate is 500,000 record per second. Both nodes have a CPU consumption similar to the one of Fig. 6b (checkpoint enabled without faults) at the beginning of the experiment before the failure. When the failure happens, CPU usage goes to 0 and after the recovery both nodes are completely saturated processing the pending load.



(a) Input load: 200,000 records/second

(b) Input load: 300,000 records/second

(c) Input load: 400,000 records/second

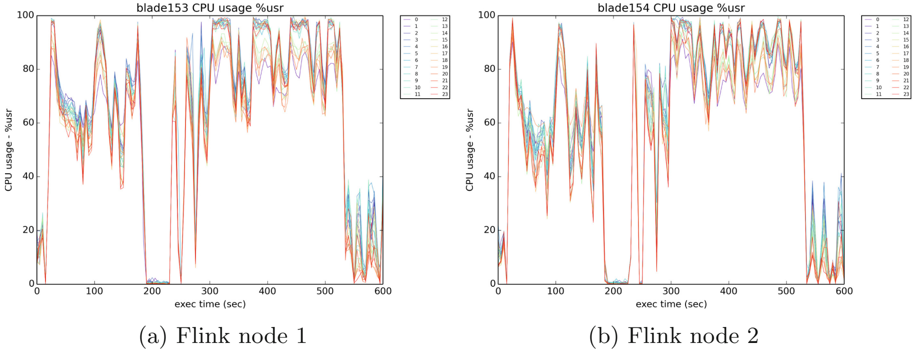
(d) Input load: 500,000 records/second

Fig. 7. Performance with checkpointing on HDFS and fault injection

To study the impact of the state size to be checkpointed on the latency, we run a set of experiments, with the checkpointing, with different window size 30, 40 and 50 records. The window size represents the state to be checkpointed. Figure 9 shows the latency graphs with the four loads.

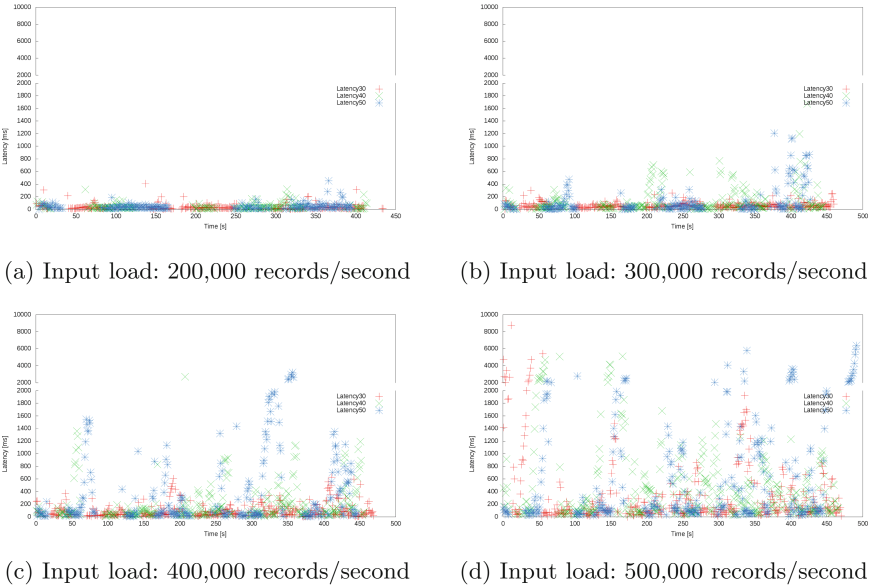
The latency for different window sizes is similar for low loads (200K records per second). As the load increases, the latency increases first for the larger windows (with 300K records) and then for all the window sizes with a high load (500K records per second). As expected the window size has an impact on the time to retrieve and store the checkpoint and therefore in the regular latency.

Table 2 reports the latency percentiles (75% and 95%) for each of the experiments with different window size. For the window size of 50 records we also report the latency percentiles when there is no checkpointing. The 75% percentile is smaller than 200 ms in any configuration when the input load is either 200,000 or 300,000 records per second. When the load is 400,000 records per second, the latency (75% percentile) when the state is 40 or 50 records reaches up to 768 ms in the case of 50-records window. With the highest workload, the 75% percentile latency is between 3 and 10 times higher than the case with no checkpointing depending on the state size. The 95% percentile shows latency values much greater than the 75% percentile due to the peaks in the latency that happen when there are many windows triggered at the same time. The impact of the window size on latency is clearly shown with the largest window comparing the latencies with and without checkpointing. The latency is at least double when chekpointing is enabled.



(a) Flink node 1 (b) Flink node 2

Fig. 8. CPU usage on Flink nodes with failures



(a) Input load: 200,000 records/second (b) Input load: 300,000 records/second

(c) Input load: 400,000 records/second (d) Input load: 500,000 records/second

Fig. 9. Latency varying the window size

Table 2. Latency. Percentiles 75% and 95%

Input Load (r/sec)	Window size 30	Window size 40	Window size 50	Window size 50 no chekpointing
200k	25–113 ms	32–281 ms	34–286 ms	73–158 ms
300k	60–279 ms	125–855 ms	172–947 ms	89–215 ms
400k	127–622 ms	391–2062 ms	768–4186 ms	108–282 ms
500k	681–2499 ms	885–3194 ms	1922–4982 ms	212–1060 ms

5 Conclusions

This paper describes and evaluate the fault tolerance mechanisms available in Apache Flink, the current de facto standard for streaming processing engines. The paper focuses on the overhead of these mechanisms on the latency and throughput through a comprehensive set of experiments. The analysis of the results shows that when the fault tolerance mechanisms are enabled, the latency can grow up the 10 times the baseline values. In presence of failures the system is able to recover quite quickly if it has enough available resources to process the peak on the input load after that the failure happens. As future work, we are interested in evaluating the performance of the system in presence of multiple topologies deployed at same time and the overhead of the exactly once end-to-end protocols.

Acknowledgments. This research has been partially funded by the European Commission under projects CloudDBAppliance, CrowdHealth and BigDataStack (grants H2020-732051, H2020-727560 and H2020-779747), the Madrid Regional Council, FSE and FEDER, project Cloud4BigData (grant S2013TIC2894), the Ministry of Economy and Competitiveness (MINECO) under project CloudDB (grant TIN2016-80350).

References

1. Apache flink. https://flink.apache.org/. Accessed 11 May 2018
2. Apache hadoop. http://hadoop.apache.org/. Accessed 11 May 2018
3. Apache kafka. https://kafka.apache.org/. Accessed 11 May 2018
4. Balazinska, M., Balakrishnan, H., Madden, S., Stonebraker, M.: Fault-tolerance in the borealis distributed stream processing system. ACM Trans. Database Syst. **33**(1), 3:1–3:44 (2008)
5. Flink an overview of end-to-end exactly-once processing in apache flink. https://flink.apache.org/features/2018/03/01/end-to-end-exactly-once-apache-flink.html. Accessed 11 May 2018
6. Flink checkpointing. https://ci.apache.org/projects/flink/flink-docs-release-1.4/dev/stream/state/checkpointing.html. Accessed 11 May 2018
7. Flink runtime. https://ci.apache.org/projects/flink/flink-docs-release-1.4/concepts/runtime.html. Accessed 11 May 2018
8. Gulisano, V., Jiménez-Peris, R., Patiño-Martínez, M., Soriente, C., Valduriez, P.: Streamcloud: an elastic and scalable data streaming system. IEEE Trans. Parallel Distrib. Syst. **23**(12), 2351–2365 (2012)
9. Hibench, a big data benchmark suite. https://github.com/intel-hadoop/HiBench. Accessed 11 May 2018
10. Huang, S., Huang, J., Dai, J., Xie, T., Huang, B.: The HiBench benchmark suite: characterization of the MapReduce-based data analysis. In: 22nd International Conference on Data Engineering Workshops, pp. 41–51 (2010). https://doi.org/10.1109/icdew.2010.5452747
11. Kwon, Y., Balazinska, M., Greenberg, A.: Fault-tolerant stream processing using a distributed, replicated file system. Proc. VLDB Endow. **1**(1), 574–585 (2008). https://doi.org/10.14778/1453856.1453920
12. Rabbitmq. https://www.rabbitmq.com/. Accessed 11 May 2018