IEEE Access
Multidisciplinary ⋮ Rapid Review ⋮ Open Access Journal

Received May 11, 2021, accepted June 25, 2021, date of publication June 28, 2021, date of current version July 7, 2021.
Digital Object Identifier 10.1109/ACCESS.2021.3093208

A Performance Analysis of Fault Recovery in Stream Processing Frameworks

GISELLE VAN DONGEN, (Member, IEEE), AND DIRK VAN DEN POEL, (Senior Member, IEEE)
MIO/Data Analytics, Ghent University, 9052 Ghent, Belgium
Corresponding author: Giselle van Dongen (giselle.vandongen@ugent.be)

This work did not involve human subjects or animals in its research.

ABSTRACT Distributed stream processing frameworks have gained widespread adoption in the last decade because they abstract away the complexity of parallel processing. One of their key features is built-in fault tolerance. In this work, we dive deeper into the implementation, performance, and efficiency of this critical feature for four state-of-the-art frameworks. We include the established Spark Streaming and Flink frameworks and the more novel Spark Structured Streaming and Kafka Streams frameworks. We test the behavior under different types of faults and settings: master failure with and without high-availability setups, driver failures for Spark frameworks, worker failure with or without exactly-once semantics, application and task failures. We highlight differences in behavior during these failures on several aspects, e.g., whether there is an outage, downtime, recovery time, data loss, duplicate processing, accuracy, and the cost and behavior of different message delivery guarantees. Our results highlight the impact of framework design on the speed of fault recovery and explain how different use cases may benefit from different approaches. Due to their task-based scheduling approach, the Spark frameworks can recover within 30 seconds and in most cases without necessitating an application restart. Kafka Streams has only a few seconds of downtime, but is slower at catching up on delays. Finally, Flink can offer end-to-end exactly-once semantics at a low cost but requires job restarts for most failures leading to high recovery times of around 50 seconds.

INDEX TERMS Apache spark, structured streaming, apache flink, apache kafka, kafka streams, distributed computing, stream processing frameworks, fault tolerance, benchmarking, big data.

I. INTRODUCTION

The demand for near real-time processing has been soaring in the last decade with the rise of the IoT domain and a surge in time-sensitive use cases such as fraud detection and monitoring. Often, this requires processing large volumes of data for which a single machine does not suffice or becomes increasingly expensive. Consequently, parallel processing becomes necessary. Processing in parallel makes many aspects of the system more complex, e.g., fault tolerance [1], accuracy [2], managing state [3], time recording [4]. Distributed stream processing frameworks have become a popular tool to process large amounts of real-time data because they allow abstraction of these complex features in a scalable system.

Many benchmarks were developed to study performance differences between these frameworks, e.g., [5]–[10]. Most of these benchmarks, however, focus on latency and throughput metrics. Much less work has been done on studying fault tolerance in depth. In this paper, we address this gap in the literature. We study the fault tolerance characteristics of each of the frameworks and we run experiments for the different types of faults that can present themselves. The distributed architecture of these frameworks implies that several components can fail. Both Spark frameworks and Flink use a master for job scheduling. For these frameworks, we experiment with single master and high-availability (HA) setups with Zookeeper. Besides a master, Spark also uses a driver component per application that does task scheduling. We test driver failures for both Spark frameworks, i.e. Spark Streaming and Structured Streaming. The workers are the second principal components that can fail. With default framework configurations, events are processed at least once. Some use cases require the guarantee that events are processed exactly once. Therefore, we also experiment with enabling exactly-once semantics to determine the performance impact.

We selected four frameworks for this comparison. We include Apache Flink [11] because it has gained widespread adoption in many large companies and is considered

The associate editor coordinating the review of this manuscript and approving it for publication was Fabrizio Messina.

This work is licensed under a Creative Commons Attribution 4.0 License. For more information, see https://creativecommons.org/licenses/by/4.0/

TABLE 1. Fault tolerance workloads in previous benchmarking work.

Reference	Workload	Frameworks	Operations	Metrics
Lopez et al., 2016 [16]	single worker failure	Spark Streaming 1.6.1, Storm 0.9.4, Flink 0.10.2	end-to-end analytics pipeline	throughput, recovery time, message loss
Qian et al., 2016 [9]	single worker failure	Spark Streaming 1.4.0, Storm 0.9.3	identity	TPF, LPF
StreamBench, 2014 [10]	single worker failure	Spark Streaming 0.9.0-inc, Storm 0.9.1-inc	identity, sample, projection, grep, statistics, wordcount, distinct count	TPF, LPF
This study	**master failure (single and high-availability), Spark driver failure, single worker failure (exactly-once and at-least-once), job failure, task failure**	**Flink 1.11.1, Kafka Streams 2.6.0 Structured Streaming, Spark Streaming 3.0.0**	**stateful pipeline**	**recovery time, downtime, (peak) latency, throughput, memory, CPU**

to be one of the front runners in stream processing at the moment. Secondly, we include both Spark APIs for stream processing, Spark Streaming, and Structured Streaming. Apache Spark is a popular distributed analytics framework, primarily used for its batch processing capabilities. When Spark Streaming was released, it quickly gained traction and it became a popular alternative for frameworks such as Storm [12] and Samza [13], mainly for use cases where a slightly higher latency is acceptable. In 2016, Apache Spark announced a more high-level stream processing API called Structured Streaming, which integrates more tightly with its batch API. Finally, we include Kafka Streams [14], which targets processing data residing on Kafka [15] clusters. One of its most significant advantages is that it does not require a pre-deployed processing cluster which reduces overhead from maintenance and idle time. Communication happens via the Kafka brokers. Kafka Streams is used by companies such as Zalando and Pinterest and is backed by Confluent.

In general, the contributions of this paper are:

1) Extensive analysis of the fault recovery mechanisms of four distributed stream processing systems: Flink, Kafka Streams, Spark Streaming, and Structured Streaming.
2) Discussion and experiments on the performance impact of a master failure with a single master and a high-availability setup with Zookeeper.
3) Discussion and experiments on the performance impact of Spark driver failures for Spark Streaming and Structured Streaming with at-most-once and at-least-once semantics.
4) Discussion and experiments on worker failures for at-least-once and exactly-once processing semantics.
5) Discussion and experiments on application, job, stage, and task failure.
6) Implementation of three fault tolerance workloads on top of OSPBench.

The rest of this paper is organized as follows. The next section gives an overview of related work on this topic. In Section III, we describe the fault tolerance mechanisms of the included frameworks. To run our experiments, we implemented three fault tolerance workloads on top of Open Stream Processing Benchmark (OSPBench) [5]. The details on how we did this have been described in Section IV. In Sections V-VII, we discuss three common failure types and their influence on the performance of the framework. Master failures with and without high-availability setups are studied in Section V. In this section, we also experiment with driver failures for the Spark frameworks. Section VI covers worker failures under different processing semantics. Failures of applications, jobs, stages, and tasks are discussed in Section VII. Finally, we form conclusions on these results in Section VIII and list some limitations in Section X. Our workload implementations have been merged into the codebase of OSPBench, which is available at https://github.com/Klarrio/open-stream-processing-benchmark.

II. RELATED WORK

The previous work on fault tolerance benchmarking of stream processing frameworks is scarce, as can be seen in the overview in Table 1. To start with, previous work only included worker failures with the default at-least-once semantics. No previous work evaluated master failures with and without high-availability setups, worker failures for exactly-once semantics, Spark driver failures, and application, job, stage and task failures. We include these important other dimensions of fault recovery in our study.

The first stream processing benchmark to study worker failure was StreamBench [10] and its extension [9]. They evaluate Spark Streaming (up to v1.4.0) and Storm (up to v0.9.3). In both [9] and [10], a worker fails in the middle of the execution and the node does not come back up. In our worker failure workload, we let the node fail and come back up after a few seconds. By doing this, we mimic a modern system in which failed services are recovered, possibly on a different node. The processing pipeline used in [9] did no operations on the ingested data. StreamBench [10] included a larger variety of processing pipelines to test fault recovery but did not give

detailed reasoning behind performance differences. In our work, we use a stateful pipeline because this makes the fault recovery process heavier and more complex. In [9] and [10], the impact of the failure was computed by comparing the latency and throughput of runs with and without node failure. They call this the latency and throughput penalty factor. The results of [10] showed that the worker failure did not have a significant impact on any metric for the incubating release of Spark Streaming. Storm performed worse: throughput dropped by one third and latency increased five times. Qian *et al.* [9] confirmed these results. Both studies look at the overall impact for an end-to-end run. We investigate the influence at a more fine-grained level since we assume that frameworks will be able to recover and that processing performance will be comparable before and after recovery. Therefore, we focus on the evolution of latency, throughput, and resource utilization throughout the failure and failure recovery process. Additionally, we determine downtime and recovery speed since these are very important metrics in production setups.

The third work which investigated worker failures [16] evaluated the response of Flink v0.10.2, Storm v0.9.4 and Spark Streaming v1.6.1 on a single node failure. They tested an end-to-end threat detection pipeline with a neural network classifier. Lopez *et al.* [16] induce a node failure after 50 seconds of execution. We wait almost 10 minutes to allow more state to be built up before the node fails, which makes fault recovery more demanding. The results of [16] confirm that the impact of the worker failure is the least for Spark Streaming. No messages are lost and the time to redistribute tasks is short. For Flink, the time to redistribute the tasks took much longer and more messages got lost during the fault. Storm redistributed tasks faster but suffered more from message loss. We compare this with our results and we add Kafka Streams and Structured Streaming. To our knowledge, the fault tolerance of these frameworks has not yet been benchmarked.

To conclude, our work is the first to incorporate a broad range of possible failures: (1) master failure for a setup with a single master and for a high-availability setup, (2) Spark driver failures with at-most-once and at-least-once semantics, (3) single worker failure with exactly-once and at-least-once processing guarantees, (4) job and task failures. We test these failures for each of the four frameworks on a stateful pipeline. Finally, we look at a broad range of metrics such as downtime and peak latency. We use these metrics to do a fine-grained analysis of the consecutive stages of the recovery process: failure, restart, rebalancing, etc.

III. FAULT RECOVERY MECHANISMS

When processing unbounded streams, jobs are intended to keep running endlessly. This means they will inevitably be confronted with machine failures, maintenance, and other interruptions. Distributed computing clusters are composed of several components: master, worker, application, job, stage, task, etc. Each of these components can fail. Frameworks handle these failures in different ways. Sometimes it suffices to reschedule some tasks, while other times the job needs to be restarted. Therefore, the effects on processing are not the same for all frameworks.

Recovering an interrupted processing job is complex. The system needs to restart the job from a set of offsets that correspond to a globally consistent state [3] while preventing data duplication and guaranteeing correctness. This is a heavy operation for which sufficient resources need to be available. In Section III-B, we elaborate on the complexity of this and explain the fault recovery prerequisites and mechanisms of stream processing systems. Without these mechanisms, data might get lost, or SLAs are broken. In Section III-C, we discuss how these frameworks can recover from master failures. Unlike the workers, the master is a single point of failure. The master does not do any processing but has a managing role. To recover from master failures, the cluster needs to be high-available. The fault recovery mechanisms of the frameworks included in this paper have been thoroughly explained in previous research. Therefore, we do not give an in-depth explanation but guide the reader to the relevant resources.

A. MESSAGE DELIVERY GUARANTEES

An important term when talking about fault tolerance is message delivery guarantees or processing semantics. There are three main message delivery guarantees. At-most-once processing means that messages might get lost in case of a failure. This happens when no fault recovery mechanism is in place. When a worker fails, another one comes back up and starts processing newly incoming data, skipping older events that were not processed yet. At-least-once message delivery means that messages might get processed twice in case of a failure. In this case, processing starts from the last available stored state or checkpoint. Events that had been processed after the checkpoint might get processed twice. Exactly-once processing is the ideal scenario in which events are processed exactly one time, meaning messages never get lost or processed twice. Most stream processing frameworks rephrase this and guarantee that each incoming event affects the final results exactly once and is committed at the sinks exactly once. To reach this guarantee, thorough fault recovery mechanisms need to be in place. Even then, it is still not possible to hold this guarantee in scenarios such as pipelines with side effects [17], as we explain later. For some use cases, exactly-once semantics are very important, e.g. fraud detection in the financial industry. However, losing messages is not equally detrimental for all use cases, e.g. IT monitoring system that loses a few CPU metrics. The frameworks included in this research allow choosing the appropriate processing semantics for each use case.

B. RECOVERY OF PROCESSING JOBS

Imagine a processing job that computes a running average on an incoming stream. Recovering such a processing job brings up some difficulties. These difficulties are mainly related to

the consistency of the state of the job. Stream processing jobs keep two types of state: intermediate results and offset metadata [18].

1) RESTORING INTERMEDIATE RESULTS

The first type of state is intermediate results. In the example of the processing job that computes moving averages, this is a list of previous values over which the average is computed. If this information is lost due to a job restart or failure, the results after the restart will not be correct. To guarantee correctness, the processing job needs to take periodic backups of this state so it can rebuild it after a failure [19]. Since the state is constantly written to and distributed over several machines, it is difficult to determine the global, consistent state of the system [3]. There are some different techniques to do this.

The most common technique is to back up state via periodic checkpointing [3], [20]. In essence, this means storing a copy of the state on a reliable, durable filesystem such as HDFS or S3. This generates an overhead that can be reduced by doing incremental checkpoints [20]. This technique avoids writing state that did not modify between consecutive checkpoints, leading to much smaller checkpoints. Another technique to reduce the overhead is by checkpointing asynchronously. This means that a separate thread is spun up to do the checkpointing and the main processing threads are not stopped.

Apache Flink uses checkpointing for recovering state and job metadata [11], [21]. It uses a mechanism called Asynchronous Barrier Snapshotting (ABS) [22] to guarantee consistent checkpoints. This mechanism is based on the Chandy Lamport algorithm [3] and punctuation [23]. This is used by many other event-driven frameworks [24], [25]. It provides a way of taking snapshots with low impact on performance and a low space cost. In short, this mechanism injects checkpoint barriers in the input stream. When an operator has received a barrier for each input channel, it triggers a checkpoint. Flink provides asynchronous checkpoints for the filesystem backend, which we also use in our experiments. Incremental checkpointing is, at this moment, only available for the RocksDB backend.

Spark Streaming and Structured Streaming use checkpointing and write-ahead-logs (WAL) to provide fault tolerance [26]–[29]. Both are micro-batch frameworks which simplifies their fault recovery mechanism [16]. They use a blocking operator model in which an operator has to produce its complete result before a downstream operator can start processing it. This leads to a simplified fault tolerance model since tasks only read intermediate results from upstream operators when they are complete. Spark Streaming jobs take a snapshot every 5-10 sliding intervals, while Structured Streaming checkpoints after every batch. Checkpointing in Spark Streaming and Structured Streaming is not incremental, nor asynchronous.

Kafka Streams takes a different approach for fault recovery and makes use of its close integration with Kafka as a messaging layer to backup state [14], [30]. In-memory state stores are backed by a replicated changelog topic on Kafka which tracks state updates. Each task that accesses the store keeps a changelog. If a task fails, it is automatically restarted on one of the remaining instances and the state stores will be restored by replaying its changelog.

2) RESTORING OFFSETS

The second type of state describes which events were successfully processed so far. If the job was in the middle of processing when the failure happened, some records were already successfully emitted at the sinks while others were not. Therefore, there needs to be a mechanism in place to keep track of which records were emitted and which ones were not. This can be seen as job metadata. When this is not tracked precisely, the job will restart processing from a point in the stream which is either too late or too early. When it starts processing from a point that is too late, data is lost (at-most-once semantics). In the other case, data is processed and emitted twice, leading to duplicate output (at-least-once semantics). Flink and the Spark frameworks include the job metadata in the checkpoint. This means that the mechanisms described in the previous section store the intermediate results together with the position in the input streams.

The need to track progress and start processing from an earlier point in the stream, places some extra requirements on the pipeline. These requirements count for most frameworks [24]–[26], [31], [32]. First of all, the source needs to be replayable. After a restart, the events that were not successfully processed are replayed and reprocessed. This means the source also needs to be deterministic [33]. A deterministic source returns the same data in the same order as before the failure. Secondly, the source needs to be persistent which means that older data is still available for replay. A popular source of streaming data is Kafka [15]. It makes data available as topics that consist of several partitions. The records in a partition are in a deterministic, static order [34] and are retained for a fixed period of time, which can go up to days. We use Kafka as data source because of its popularity and because it is a replayable, persistent, and deterministic source.

Besides the sources, processing also needs to be deterministic. This is not evident. For example, if a pipeline queries an external database for side input, the database records may have changed during the failure. Another example is a pipeline that uses processing time windows. At restart, this pipeline will take in all the data that was produced during the downtime, leading to a larger number of events in the window bucket and different results compared to normal operations. Therefore, the processing logic on the window bucket needs to be robust to this.

Finally, there are also some prerequisites for the sinks. If a failure occurs, the system cannot rely on downstream systems to roll back [35]. The outside world needs to perceive consistent behavior of the system despite any failures. This is called the output commit problem [20]. There are two possible approaches here [36]. One option is to have an

idempotent sink operation. This means the sink would be able to handle duplicate data, e.g. overwriting data in a database, overwriting a file with identical data.

Another option is to implement transactional and atomic writes in the sink, implying that they either complete entirely or roll back. Flink implements this [18], [31] via the Two-Phase Commit Protocol [37]. This protocol coordinates rollbacks and commits in the sinks and works in two phases. In the pre-commit phase, the checkpoint is started and the barrier passes through all the operators and triggers snapshots. In the commit phase, the checkpoint has succeeded and the job manager communicates this to all operators. When a failure happens, uncommitted data can end up on Kafka. During the reprocessing, this data is sent to Kafka again and now committed correctly. To avoid reading duplicate messages in downstream systems, the isolation level of their Kafka consumers needs to be set to only read committed data.

Kafka Streams does three steps atomically to guarantee exactly-once processing: (1) committing offsets of the source topics, (2) tracking state updates on highly-available changelog topics and (3) acknowledging records at the sinks [38].

Spark Streaming or Structured Streaming use write-ahead-logs to enable at-least-once (and exactly-once) semantics. The WAL is a concept from database design and is a JSON append-only log that supports durable, atomic writes. It can be used to replay data for recovery or rollbacks [26]. It offers a reliable log of data that has been successfully written to output sinks. We store the WAL on HDFS together with the checkpoints. The end-to-end processing guarantee of a Spark Streaming or Structured Streaming application depends on the semantics of the sources and the sinks of the application. In our pipeline, we use a Kafka sink for Structured Streaming and a foreachRDD sink for Spark Streaming. Both of these sinks only provide at-least-once semantics at the time of this writing. Currently, none of the stream processing APIs of Spark offers writing to Kafka with out-of-the-box exactly-once guarantees. To reach exactly-once semantics, the user needs to do a custom implementation of transactional updates to Kafka. We leave this out of the scope of this paper because the goal of this paper is to test the built-in features of the frameworks. As far as we are aware, the only sink which currently offers built-in exactly-once semantics is the file sink of Structured Streaming because it is idempotent.

C. RECOVERY FROM MASTER FAILURE

The master of the cluster has different responsibilities in different frameworks. In Flink [11], the master is the job manager which coordinates execution, schedules tasks, triggers checkpoints, coordinates worker failures, etc. A Spark cluster works differently. A standalone cluster exists of a master and several workers. An application running on a cluster exists of a driver and several executors. The executors run on the workers and the driver runs either on one of the workers or separately. The driver is the main entry point of the application. It houses the SparkContext and schedules tasks on the executors that were assigned to it. The master of the cluster does tasks such as monitoring the health of the workers and tracking the jobs that are running on it [36]. Kafka Streams does not use a master-slave architecture and runs as separate instances which are identified as one application by the Kafka brokers.

Because of these differences, the effect of a master failure also differs across frameworks, e.g. inability to schedule new jobs, job restart, etc. We elaborate further on this in the Results Section. Because a master has a different set of responsibilities than the workers, other fault-tolerance schemes need to be in place. A single master is a single point of failure. Setups that tolerate master failures are high-availability setups. Several high-availability setups have been proposed [33]. Currently, the most popular approach for standalone clusters is having multiple masters running concurrently, where one is the leader and the other is in standby. In this setup, Zookeeper [39] is used for coordination. The job metadata are kept partly in Zookeeper and partly in a reliable filesystem, e.g. HDFS, S3. When a failure happens, Zookeeper notices the failure and elects the other master as the new leader. Some frameworks require a job restart after this to ensure the consistency of state. Whether or not this is required, depends on the framework. If it is required, this will be handled as described in the previous section, i.e. based on the latest checkpoint. In this work, we evaluate both setups: single master and high availability with Zookeeper.

IV. BENCHMARK DESIGN

As said earlier, several benchmark suites have implemented latency and throughput experiments. However, there is no single benchmark suite that covers all key aspects of stream processing and has recent implementations of varying pipelines in several frameworks. To repeatedly benchmark newer releases on several features and keep results comparable, we would need to work towards this goal. Therefore, we prefer extending an existing benchmark suite, with extensive fault tolerance workloads, over implementing a new suite from scratch. For this purpose, we extend OSPBench [5] with three fault tolerance workloads. OSPBench is an open-source stream processing benchmark that includes extensive latency and throughput workloads for pipelines of varying complexity and under varying data characteristics. It offers recent implementations of pipelines in four state-of-the-art frameworks and includes an ecosystem for collecting and analyzing performance metrics. One of the limitations of OSPBench was the lack of fault tolerance workloads. OSPBench only looked at the key performance metrics (i.e. latency and throughput) in optimal conditions. Nonetheless, it does include a workload to test the peak throughput during an input data burst. This is closely related to the time it takes to catch up on processing delays after failure. However, this does not take into account the fault recovery mechanism, task rescheduling, state restoring, and rebalancing operations that take place during a failure. Therefore, the ability to catch up quickly is only a small portion of the end-to-end

performance on fault recovery. As such, OSPBench requires dedicated fault recovery workloads. This study is focused on investigating the behavior under several kinds of fault scenarios. The three workloads, we implemented, were merged into the codebase of OSPBench, which now offers a one-stop benchmark suite for latency, throughput, and fault tolerance testing. The features of OSPBench that we use are its processing pipeline implementations, infrastructure and automation scripts, and metrics collection system. These are shortly described throughout this section. Finally, we also discuss the chosen framework configurations for the fault tolerance workloads.

A. PROCESSING PIPELINE

We use a processing pipeline that analyzes NDW traffic data [40] of the Netherlands, as shown in Figure 1. The job ingests two streams: one containing speed measurements and the other containing vehicle counts at a configurable number of measurement locations. The total input throughput is 18 000 events per second. The incoming data is in JSON format and subsequently parsed into Scala case classes. Both streams are then joined together at one-second intervals. At this point, the joined stream contains events for every lane of every measurement location. Next, we join all the different lanes of a similar measurement location and compute the average speed and total count.

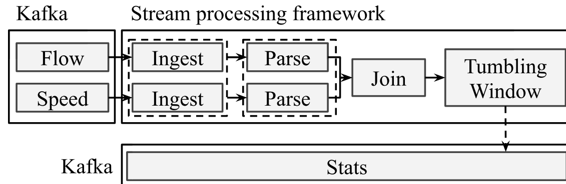

FIGURE 1. Processing pipeline adapted from [5] and [41].

Resuming a failed pipeline mainly involves rescheduling failed tasks and operators, restoring state, rebalancing partitions over worker threads, and catching up on the processing lag. In our experiments, we use a stateful pipeline because restoring state is one of the heaviest parts of fault recovery. The larger the state, the heavier it is to reload it into the workers. The size of the state is the most important determinant here. The type of operation which keeps the state (e.g. tumbling window, sliding window) is not relevant for recovery. Besides that, the other aspects of fault recovery also have no clear relation to the exact operators in the pipeline. The fault recovery mechanism of the framework influences the rebalancing of partitions and rescheduling of tasks. Additionally, previous work [5] tested the speed of catching up on a delay for several pipeline complexities. For all the pipelines, they found similar rankings and differences between the frameworks. Therefore, we believe the results of our workloads on one stateful pipeline will be representative of other pipelines.

B. INFRASTRUCTURE

In the design of the underlying infrastructure of our benchmark runs, we mimic modern real-world deployments by running on the cloud and using popular state-of-the-art technologies. We run our experiments in the cloud on AWS EC2 instances to make the benchmark reproducible. The EC2 instances have 16 vCPUs, 64 GiB memory, and a network bandwidth of up to 25 Gbps. Each instance has an EBS volume of 500 GB with a bandwidth of up to 4750 Mbps. We ensured the network bandwidth is sufficient for our processing job. On top of these instances, we set up a DC/OS cluster [42] which serves as an abstraction layer on the underlying machines. Moreover, DC/OS allows easy deployment of Docker [43] container services. All the components of this benchmark run as a Docker container on top of DC/OS.

To run the benchmark, we need several other services. We set up a Kafka [34] cluster to serve as a message broker between data producers and data consumers. For the collection of metrics, we set up a cAdvisor [44] instance on every node and a JMX exporter to scrape JVM-related metrics from the running containers. We backup the state of the processing jobs on an HDFS [45] cluster. During a benchmark run, we deploy a framework cluster when required (for Flink, Spark Streaming, and Spark Structured Streaming). Finally, we also have a component that generates a data stream and publishes it on Kafka. The generated stream is read by the processing job which runs on the cluster and results are published back to Kafka.

C. METRICS

In this section, we describe the metrics that we use to discuss the fault recovery capabilities of the frameworks. We also cover how we collect and compute them.

1) LATENCY

The first metric is latency. Latency is the time required to generate an output message from an input message. In this study, we use latency as an indicator of delays in processing. We compute latency by subtracting the Kafka message timestamps of the input and output events. When multiple events are required to do a computation, we count latency starting from the last input event. Since the clocks of the different machines of the Kafka brokers are not synchronized, we need to make sure that input and output events end up on the same brokers, as per [41]. We do this by using the same partitioning key for the input and output.

2) THROUGHPUT

The second metric is throughput. Throughput is the processing rate of the framework in messages per second. We discuss two types of throughput in this work. Input throughput is the number of events per second entering the framework at the sources of the pipeline. Output throughput is the number of events per second emitted from the processing job at its sink.

3) CPU UTILIZATION

We include CPU utilization because it indicates the load on the framework and whether the framework is making efficient use of its allocated resources. We use cAdvisor to collect CPU metrics of each running container [44].

4) MEMORY UTILIZATION

The second resource metric we monitor is memory utilization. Memory utilization is subject to the configuration settings of the framework. Memory pressure can induce heavy GC cycles which put load on the CPUs. Workers fail when GC cycles become ineffective and memory grows out of bound. We use memory utilization to check whether it is not monotonously rising and to check whether garbage collection has the desired effect. We use JMX to collect metrics on the heap and off-heap memory utilization. Memory utilization is not a direct indicator of speedy recovery. Therefore, we do not include memory metrics in our visualizations but report on them when informative.

D. FRAMEWORKS

We include four frameworks in this comparison. To get reliable, generalizable results, it is paramount to use the right configuration settings for each job and workload, as has been highlighted by previous work [5], [9], [46]. Tuning the parameters ensures that we are comparing the optimal performance of the frameworks. We do not want to test how well the default parameters are set but how well the framework can perform on a certain feature. Therefore, we tuned the most important parameters through extensive experimentation and based on expert advice. Most of these parameter settings are recommended by the framework documentation or follow standard practices, as will be explained.

In Table 2, we list the cluster-related configuration parameter settings. Many of these are equal for all frameworks, such as instance counts, CPU, memory, and disk allocation. We set the parallelism of all frameworks equal to the number of cores in the cluster. This refers to the setting for the number of task slots for Flink and the number of threads for Kafka Streams.

In Table 3, we list the configuration parameters that are related to the processing job itself. Some of these were, again, equal for every framework. An example of this is the setting for event time processing. Stream processing jobs are frequently confronted with delayed data. For example, when processing sensor data, one sensor can be temporarily offline. When it comes back up, it uploads its backlog in bulk. Another example is where one data generator incurs higher network delays than another. A stream processing job needs to be robust against these types of scenarios and needs to still be able to generate accurate results. To do this, a job needs to be able to base its processing on the timestamp at which an event was generated instead of the timestamp at which it entered the framework. The mechanism which allows doing this is called event time processing [2]. Since most use cases require correctness in the case of late data, we enable event time processing where possible. The only exception is Spark Streaming, which only offers processing time semantics.

TABLE 2. Cluster configuration parameters.

Parameter	Setting
a. Apache Flink	
JobManager count, CPU, memory, disk	1, 2 CPU, 8 GB, 20 GB
TaskManager count, CPU, memory, disk	5, 4 CPU, 20 GB, 20 GB
JobManager Flink memory	7 GB
TaskManager Flink size	18 GB
Memory managed fraction	0.05
Number of task slots	4
Default parallelism	20
b. Apache Kafka Streams	
Instances count, CPU, memory, disk	5, 4 CPU, 20 GB, 20 GB
Java heap	14 GB
Streams thread count	4
Kafka parallelism	20
c. Spark Streaming and Structured Streaming	
Master count, CPU, memory, disk	1, 2 CPU, 8 GB, 20 GB
Worker count, CPU, memory, disk	5, 4 CPU, 20 GB, 20 GB
Driver cores, heap	2, 6 GB
Executor count, cores, heap	5, 4 cores, 17 GB
Default parallelism	20
SQL shuffle partitions	20

For the frameworks which allow event time processing, the tolerance interval for late data is set to 50 ms. We chose this number because it is an approximation of the maximum time difference between the Kafka brokers. In our setup, we are sure that data is generated in order. The only disordering that can happen is due to having several Kafka brokers on different machines. Kafka only guarantees order within partitions and not across partitions. The allowed lateness of 50 ms refers to the out-of-orderness bound for Flink and the watermark setting of Structured Streaming. We increase the grace period of Kafka Streams to 30 seconds since it prevents wrongly discarding events when some partitions are processed faster than others and it has no negative influence on the latency. We run all workloads with default at-least-once processing semantics. Additionally, we also experiment with at-most-once semantics for the Spark driver failure workload and exactly-once semantics for the worker failure workload.

Finally, we use G1GC as the garbage collector for all frameworks since this is often recommended for Spark and Kafka Streams [15], [47] and is the default for Flink. We ran some tests with different GC algorithms and found best performance with G1GC for our pipeline.

1) APACHE FLINK

Apache Flink [11] is an open-source stream processing framework that positions itself as a fast, in-memory, scalable, distributed processing system for unbounded and bounded data. Since its first release in 2014, it has gained widespread adoption by companies such as Alibaba, AWS, Uber and Zalando.

TABLE 3. Job configuration parameters.

a. Apache Flink (v1.11.1)

Parameter	Value	Default
Time characteristic	event time	processing time
Watermark interval	50 ms	/
Out-of-orderness bound	50 ms	/
State backend	FileSystem	InMemory
Checkpoint interval	10s	none
Object reuse	enabled	disabled
Exactly-once semantics		
Producer semantic	exactly-once	at-least-once
Enable idempotence	true	false
Isolation level	committed	uncommitted
Transaction timeout	1h	15min

b. Apache Kafka Streams (v2.6.0)

Parameter	Value	Default
Commit interval	1s	30s
Grace period	30s	0 ms
Producer compression	lz4	none
Producer batch size	200 KB	16 KB
Max task idle	5 min	0
Topology optimization	optimize	none
Window changelog retention	10 min	1 day
Garbage collector	G1GC	Parallel GC
Processing guarantee	exactly-once	at-least-once

c. Spark Streaming and Structured Streaming (v3.0.0)

Parameter	Value	Default
Serializer	Kryo	Java
Write ahead log	enable	disable
Garbage collector	G1GC	Parallel GC
Initiating heap occupancy	35%	45%
Parallel GC threads	4	/
Concurrent GC threads	2	/
MaxGCPauseMillis	200	200
Micro-batch interval (sp.)	1 s	/
Trigger interval (str.)	0	/
Max offsets per trigger (str.)	$20 * throughput$	/
Block interval	50 ms	200 ms
Locality wait	100 ms	3000 ms
Min. batches to retain	2	100
Watermark (str.)	50 ms	/
Watermark policy (str.)	max	min

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

When a stateful processing pipeline is used, Flink requires a state backend to be able to store state. There are three state backend implementations available in Flink: memory backend, filesystem backend, and RocksDB backend. As stated in the Flink documentation [11], the memory backend is encouraged for local development and debugging. The filesystem backend should be used for high-availability setups with a large state which fits on the task manager heap. And finally, the RocksDB backend should be used for high-availability setups where the state is too large to fit on the heap. Since our job requires a high-availability setup and the state is small enough to fit on the heap, we use the filesystem backend with HDFS for persistent storage.

For stateful pipelines, Flink requires setting a checkpoint interval [11]. Setting this interval leads to a trade-off. Checkpointing is, on the one hand, a CPU-heavy operation so a lower interval takes away more resources from the main computation pipeline. A higher interval, on the other hand, leads to longer recovery times since Flink restarts jobs from the latest successful checkpoint in case of a failure. We set our checkpoint interval to 10 seconds because this is low enough for a speedy recovery and high enough to not become the bottleneck of our pipeline. This is also the default checkpoint interval of Spark Streaming.

The filesystem backend is heap-based. This means that the task managers need to have enough heap memory to store the entire state. To this end, the Flink documentation recommends setting the managed memory fraction to a very low value because it makes sure that the maximal amount of memory is allocated to heap space.

We enable object reuse because this improves performance, as discussed in [11]. Flink does not use this by default because it can cause issues when the user code is not adapted for this behavior.

To enable exactly-once semantics, some settings need to be adapted as explained in the Flink documentation [11]. First of all, we enable the Two-Phase Commit Protocol [18] by setting exactly-once semantics for the Kafka sink. Additionally, we set the transaction timeout of the Kafka brokers to 1 hour to deal with longer outages. To avoid reading duplicates in case of a failure, we set the isolation level of the Kafka consumer in the output consumer to only read committed messages.

2) APACHE KAFKA: KAFKA STREAMS

Kafka [15] is a distributed message broker that serves as a reliable, scalable, robust intermediary between data producers and data consumers [34]. Kafka Streams [14] is a library that allows data transformations on top of data residing in a Kafka cluster. Kafka Streams applications do not require a separate processing cluster but run as individual threads that rely on Kafka for parallelism and fault tolerance.

We set the commit interval to one second because this is the slide interval of the windowing operations. This triggers output of the aggregation step every second, as we do for the other frameworks.

During startup or restarts of a container, some threads process faster than others. We noticed that events got discarded when processing did not progress equally fast on all partitions. Event time progressed together with the fast-moving partitions. This caused events to be discarded on the slower partitions. To mitigate this, we use a higher grace period of 30 seconds. With this setting, Kafka Streams will tolerate late data of up to 30 seconds. This does not impact the latency because Kafka Streams sends an update from aggregation operations at every commit interval and does not wait for the end of the grace period to emit results. This mechanism is called the Dual Streaming Model of Kafka Streams [30]. Additionally, we increased the maximum time tasks are allowed to wait idly for data to arrive on all partitions. This further reduced the incorrect discarding of data.

For most of the other settings in Table 3, we follow the tuning recommendations of the Kafka Streams developers [48]. We increase the producer batch size to 200 KB. This leads to larger batches which reduces the number of requests and therefore, the load on the producers and brokers [48]. We use lz4 compression to reduce the size of messages, as recommended in [48].

To ensure that threads clean up state promptly, we set the general window changelog retention time to ten minutes and additionally, define shorter retention times for all individual windowing operations. Without these settings, the job becomes unsustainable after 20 minutes when it starts cleaning state. Finally, when desired, we enable exactly-once semantics by setting this as the processing guarantee of the pipeline.

3) APACHE SPARK: SPARK STREAMING AND STRUCTURED STREAMING

Apache Spark is an open-source unified analytics engine that targets large-scale data processing use cases and is actively developed by Databricks and the open-source community. It currently includes two streaming APIs: the older Spark Streaming [36] API and the more recent Structured Streaming API [49]. Spark Streaming is based on the DStream and RDD API [27], whereas Structured Streaming is more high-level and is built on the Spark SQL engine, which does a lot of optimizations under the hood. Structured Streaming is more closely integrated with the batch API. Users can switch from batch to streaming implementations with minimal code changes.

We run all applications in client mode, meaning that we run the driver in a separate container that connects to the Spark master to request resources and run the application. We set the supervise flag when submitting applications to ensure application restarts in the case of failures.

As recommended in the documentation [36], we use Kryo serialization since it is faster than the default Java serializer. Kryo serialization is not the default since it requires manual registration of the classes used throughout the pipeline. In our codebase, we do this registration for both Spark Streaming and Structured Streaming.

We tune the most important garbage collection parameters, based on [47]. We adapt the parallel and concurrent GC threads of G1GC based on the number of cores per JVM. We set the parallel GC threads equal to the total number of cores and we set the concurrent GC threads equal to 50% of the cores. Furthermore, we reduce the initiating heap occupancy to 35% to trigger more frequent but less heavy garbage collection cycles.

Spark Streaming uses a fixed micro-batch interval at which batches of data are collected and processed. We set the batch interval to one second which is equal to the slide interval of the aggregation step. This is a logical choice since we want to compute aggregates per second in our use case. As said, Spark starts processing a batch when the batch interval times out. To benefit from parallel processing, the data is split up into partitions while it is arriving. This is done based on the block interval. We set the block interval equal to the batch interval divided by the desired parallelism to get the right amount of partitions in each interval. For Structured Streaming, we set the trigger of our sink operation to zero, meaning processing as fast as possible. This setting leads to dynamically sized batches based on the processing time of the previous batch. Dynamically sized batches can lead to advantages in the case of changing processing loads [50], which we will be confronted with during restarts. With dynamic batch sizes, the framework can increase the batch size to be able to sustain peaks in throughput and can lower the batch size to reach lower latencies at times where throughput is lower.

By default, Structured Streaming uses the minimum timestamp across all partitions as the event time of a batch. The event time of the system is updated after the processing has finished. This means that the event time of some events in the batch has not yet passed the watermark. These events are buffered until the next trigger before they are outputted. Because of this, chained stateful operations cause high latencies. To speed up the generation of output, we use the maximum timestamp of the batch as the event time. This avoids buffering events longer than one interval at each stage. We do this by setting the watermark policy to take the maximum instead of the minimum event time as the global watermark. This watermark policy makes sure that the watermark follows the fastest input stream and that data from the slower stream is dropped aggressively. We did not experience message loss since both streams are well synchronized and this allows for a much lower latency.

During a restart, a processing backlog builds up. We noticed that when the job came back up, the first batches were so large that they tended to overflow the memory of the executors. To avoid this, we set the maximum offsets per trigger to the equivalent of 20 seconds of data. By doing this, we regulate the amount of data that is ingested in the initial batches. We decrease the minimum number of batches to retain to reduce the memory footprint.

Spark includes a mechanism to preserve data locality when scheduling jobs. This means that Spark tries to schedule tasks as close to their data as possible. First, it will try to schedule it in the same JVM, then on another executor on the same node, and then on the same rack. The configuration setting spark.locality.wait sets the amount of time Spark waits for a core to free up before it tries to schedule the task on the second-best locality level. The default locality wait time is 3 seconds, which is reasonable for batch executions but not for streaming jobs since tasks take much less than 3 seconds. When we put the locality wait on 100 ms, tasks are distributed more equally and latencies are reduced. Reducing this setting is standard practice for streaming jobs in both Spark Streaming and Structured Streaming.

Structured Streaming generates a high load on HDFS due to checkpointing. For stateful operations, Structured Streaming checkpoints at each batch interval (mostly between one and three seconds). This is much more frequent than the other

frameworks. For example, Spark Streaming checkpoints at an interval that is a multiple of the batch interval, at least 10 seconds and typically 5-10 times the sliding interval of the stateful operation. Because Structured Streaming checkpoints more frequently, we had to increase the HDFS cluster to handle the load.

V. MASTER FAILURE

In the following sections, we describe the three main types of failures that can happen in master-slave architectures. First, we describe behavior in the case of master failures and Spark driver failures. The next section covers worker failures and Section VII discusses other failures such as application, job, and stage failures. An overview of the mechanisms and performance of the different frameworks has been given in Table 4.

TABLE 4. Failure recovery mechanism and performance. When a feature is not available or not applicable for the framework, we denote this by N.A.

	Flink	Kafka	Spark	Struct.
Fault Recovery Mechanism				
Checkpointing	yes	no	yes	yes
State store	HDFS	Kafka	HDFS	HDFS
Write-ahead-logs	no	no	yes	yes
Transactional Kafka sink	yes	yes	no	no
Master failure				
Single Master	app killed	N.A.	app killed	app killed
HA (Zookeeper)	restart	N.A.	no impact	no impact
Driver failure (at-most-once)	N.A.	N.A.	restart + data loss	restart + data loss
Driver failure (at-least-once)	N.A.	N.A.	restart + no data loss	restart + no data loss
Worker failure				
Job recovery	OK	OK	OK	OK
Job restart	yes	no	no	no
Downtime	50s	2s	6s	4s
Recovery time	53s	21s	30s	20s
Peak p99 latency	53s	19s	8.5s	10.5s
Stability after failure	good	good	good	good
Cost exactly-once	no	large	N.A.	N.A.
Duplicate output	yes	no	no	no
Application failure				
Restarts	yes	no, via Marathon	supervise	supervise

In the context of master failures, we experiment with a single master setup and a high-availability setup with Zookeeper. We discuss the results for Flink, Spark Streaming, and Spark Structured Streaming since these frameworks make use of a master-slave architecture and therefore, have a master which can fail. Kafka Streams does not have a master component and therefore, is not included in this section.

A. SINGLE MASTER SETUP

When the job manager of Flink gets killed, the task manager's report that heartbeats are timing out and that they lost their leader. When this happens, the running applications fail and no new applications can be submitted anymore. When Marathon has rescheduled the job manager and the task managers have joined the cluster again, the job still does not recover. By default, the job state is not stored anywhere and the new job manager does not know which jobs were running.

We see similar behavior when the Spark master crashes: no new applications can be scheduled and the application fails after numerous timeouts in its communication to the master.

B. HIGH-AVAILABILITY WITH ZOOKEEPER

In a second experiment, we enable high-availability with Zookeeper [39] as a coordination mechanism that stores the location of the latest checkpoint and does leader election.

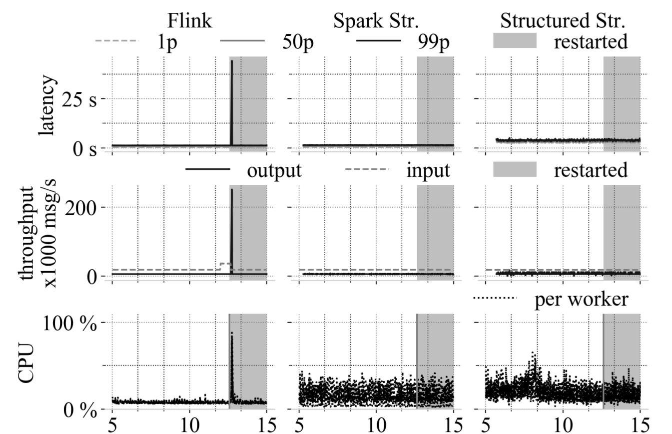

FIGURE 2. Master failure workload with high available master setup with Zookeeper for flink, Spark streaming and spark structured streaming. Kafka streams is not included since it does not use a master-slave architecture and therefore, does not have a master component.

In a high-availability setup, Flink stores its dataflow graphs on HDFS. We tried two approaches for high availability. In the first one, we had a standby master running during the job. In the second one, we relied on Marathon to bring the job manager back up after the failure. We noticed similar performance for both approaches. When we rerun the workload and let the master fail after 10 minutes, we see that the job fails but comes back up when the new job manager has been elected as leader. As we described in Section III-C, the job is restarted to maintain state consistency. When a Flink job restarts, it starts processing from the latest successful checkpoint. Because the job fails, there is a performance impact. We notice a gap of four seconds in the output. After that, it takes 2 seconds to finish processing the backlog and before all metrics are back to normal levels. In total, the impact was noticeable for 6 seconds. When we use at-least-once processing, we see duplicate output for the messages between the last checkpoint and the failure. The impact described here is visible in Figure 2. The input throughput chart of Flink shows that events that were published right before master failure were ingested two times in the framework. This line does not show the number of events that were published in a certain second but shows the number of events of that specific second that

were processed. Therefore, this bump indicates that these events were processed twice, as is expected with at-least-once processing semantics. We see a large increase in latency due to the outage of 4 seconds and the reprocessing of the old events that were published since the last checkpoint. In Section VI, we describe the implications of job restarts in more detail.

When the Spark master fails in a high-availability setup, a new one is started while the application keeps running without interruptions. This is possible since the driver of the application does not run within the master container but runs as a separate container or on one of the workers and therefore, keeps running. Due to this approach, there is no impact on any of the performance metrics of the job: latency, throughput, resource utilization, etc. Making use of a standby master has the same effect. As can be seen in Figure 2, the fail-over did not impact the performance metrics of the processing jobs of Spark and Structured Streaming.

When we compare failure-free runs with and without a high-availability setup, we saw no noticeable performance impact on the processing job from using a high-availability setup. The reason for this is that processing happens on the workers and not on the master. Running in a high-availability setup mainly creates extra work for the master. It does not significantly influence the key metrics of the processing job that is running on the workers, such as latency and resource utilization. Moreover, the CPU utilization of the master is very low for both setups and shows no significant difference. So also for the masters themselves, the overhead is limited. This can be explained by the fact that storing job metadata on Zookeeper is not a heavy operation.

Finally, it is important to note that running a master in standby leads to a larger resource allocation to the cluster and, therefore, a higher resource cost. In our setup, we allocate 2 CPUs and 8 GB of memory to the master. On the contrary, if we use Marathon to reschedule the master, only one master is running at all times and there is no additional cost for high-availability.

C. SPARK DRIVER FAILURE

As described in the previous section, both Spark Streaming and Structured Streaming applications have a driver to do task scheduling. The driver houses the Spark Context through which it connects to the Spark cluster [36]. It can be seen as the master of the Spark application and is, therefore, an essential component. Because of this, we also experiment with driver failures. This experiment can only be done for the two Spark frameworks since the other frameworks do not have a separate driver component. The results have been visualized in Figure 3.

The driver can either run on the workers or run in client mode as a separate container. We use client mode. When the driver is killed, it comes back up automatically because the container gets rescheduled by Marathon. If the driver would run on one of the workers, we can enable automatic restarts by submitting the application with the supervise flag.

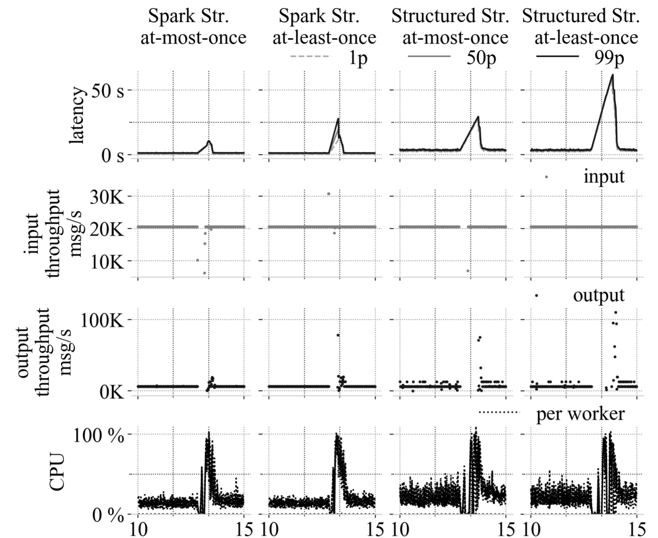

FIGURE 3. Driver failure workload for spark streaming and spark structured streaming. Without WAL and proper checkpointing enabled, we get at-most-once semantics. With WAL and proper checkpointing enabled, we get at-least-once semantics.

We run the driver failure workload two times. The first time we do not enable write-ahead-logs and proper checkpointing, leading to at-most-once semantics. The second time, we enable write-ahead-logs and proper checkpointing, leading to at-least-once semantics. Write-ahead-logs ensure that there is no data loss in the case of driver failures. Checkpoints ensure that intermediate state and Kafka offsets are backed up.

In Figure 3, the results for both frameworks and semantics are visualized. When the driver goes down, the application and its executors are killed. Therefore, we see a gap in the output throughput during this restart period. When the driver and the application come back up, new executors are assigned. Subsequently, the state is restored from the latest successful checkpoint. Figure 3 shows a gap in input throughput for applications with at-most-once semantics. The input throughput chart shows the number of input events of a specific second that were processed by the framework. Ideally, this should be a flat line because the data stream generator generates the same number of events for each second. If there is an increase or a drop in the input throughput, this indicates that the framework either skipped events (data loss) or processed events twice (duplicate processing). Therefore, the gap in input throughput for the runs with at-most-once semantics indicates data loss. The number of lost events depends on the duration of the outage and the throughput. Longer outages and higher throughput lead to more lost events. In the case of a stateful pipeline, these lost events also lead to an incomplete state and, thereby, incorrect results. How long this persists, depends on the type of operations that are done. For example, when windows are used, the state will be incomplete or incorrect up to the longest window duration. In our pipeline, the window duration is only a few seconds so results also

only remain incorrect for a few seconds after the restart. For Spark Streaming, the downtime was 18 seconds which equals 370 000 lost input events. The first three seconds after the restart, executors are starting up and a portion of the events are still skipped. In total, approximately 400 000 events were lost and five seconds of processing was based on an inconsistent state. To extrapolate these results to other pipelines, the reader should take into account the input throughput and pipeline characteristics.

For at-least-once semantics, we see that there are no gaps in the input throughput. All data of every second was processed. For Spark Streaming, we see an increase in the input throughput around the restart. This is caused by the events that got processed twice due to the at-least-once semantics. For Structured Streaming, we do not see this increase. However, this does not mean that Structured Streaming offers exactly-once semantics. Whether or not there is duplicate processing, depends on the exact moment of the failure. The driver may fail while the application is publishing the results of a batch. In this case, the results that were published were not committed yet and will be reprocessed. In the run visualized in Figure 3, the failure happened before Structured Streaming started publishing the output of the batch and therefore no duplicate output was published. Additionally, the number of reprocessed events depends on the recency of the latest checkpoint and the throughput. When a recent checkpoint is available, less data needs to be reprocessed. The correctness of these duplicate events depends largely on the prerequisites which we described in Section III. One of these is the determinism of the processing pipeline. For example, when the pipeline makes use of processing time windows, all events ingested at startup end up in the same window bucket. This leads to different outputs as without a failure. Of course, this is the expected behavior of processing windows. The pipeline needs to be robust against this. We use processing time windows for Spark Streaming because it does not support event time processing. We mitigate this in our implementation by including a truncated timestamp in the key of the grouping operations. In our experiment, one second of data was reprocessed, i.e. approximately 21 000 input events. It is important to note that even if the reprocessed events lead to identical output, downstream systems still need to be able to handle this correctly.

D. CONCLUSION

We can conclude that high-availability setups are crucial to ensure the continuation and restart of processing applications in both Flink and Spark. With a high-availability setup, Flink still requires a job restart when the master fails to ensure the consistency of state. Spark Streaming and Structured Streaming do not notice any impact of a master failures. Driver failures, on the contrary, do lead to job restarts and significant downtime in Spark applications. With the appropriate fault tolerance mechanisms, data loss can be prevented but duplicate processing may still occur.

VI. WORKER FAILURE

In the worker failure workload, we make one worker node fail deliberately by killing it in the middle of the execution. The worker node comes back up immediately. We monitor the behavior of the framework during the recovery process. We compare the performance of at-least-once and exactly-once semantics, as shown in Figure 4. The grey zone shows the recovery period. In the first few seconds of the grey zone, the Docker container of the new worker restarts and joins the cluster. For the rest of the period, the job recovers. Three metrics are shown: (1) median and p99 latency, (2) input and output throughput, and (3) CPU utilization of each worker. The input throughput is 3.16 times higher than the output throughput since the pipeline contains a join and aggregation step. The CPU utilization of the worker that fails and restarts, is marked in black. In the following, we discuss some different aspects of fault recovery, as reported in Table 4. First of all, we discuss whether or not the job was able to recover. Secondly, we measure the length of the outage or downtime as the gap in output throughput. After the downtime, the job needs to catch up on the delay before latencies can return to normal levels. We define the recovery time as the sum of the downtime and the catch-up period. We report the peak p99 latency that we encounter during the recovery process. When the job has fully recovered, we evaluate the stability of the job. Finally, we compare the accuracy, performance and cost of exactly-once and at-least-once semantics.

A. DOWNTIME AND RECOVERY TIME

When the cluster manager notices that a worker has failed and a new one has joined, it needs to redistribute tasks across the workers. The time it takes for the new worker to join the cluster and go through the redistribution phase is visible by the downtime and highly inflated latencies.

Flink has the longest downtime. For both processing semantics, no output is generated for around 50 seconds. Flink does fault recovery based on checkpoints. When a task or task manager fails, the job is canceled completely. It gets rescheduled as soon as there are enough task slots available, so when the task manager is back up. When the job is rescheduled, it needs to rebuild the state from the last successful checkpoint and start processing data from thereon. In our workload, the checkpoint interval is 10 seconds so the maximum timespan of data that needs to be reprocessed is 10 seconds of data plus the downtime of the task manager and the processing job. Once the job is running again, it takes only eight seconds for Flink to rebuild the state and go through the entire backlog of 45 seconds. The first seconds of processing reach a maximum throughput of 164 000 output events in one second, equivalent to an input of 520 000 events per second.

The Spark frameworks implement fault recovery from a worker failure differently. Once the master and driver notice that a worker went down, it reschedules the tasks that were running on that worker on one of the other workers. This is visible in Figure 4 as the throughput stays almost unchanged

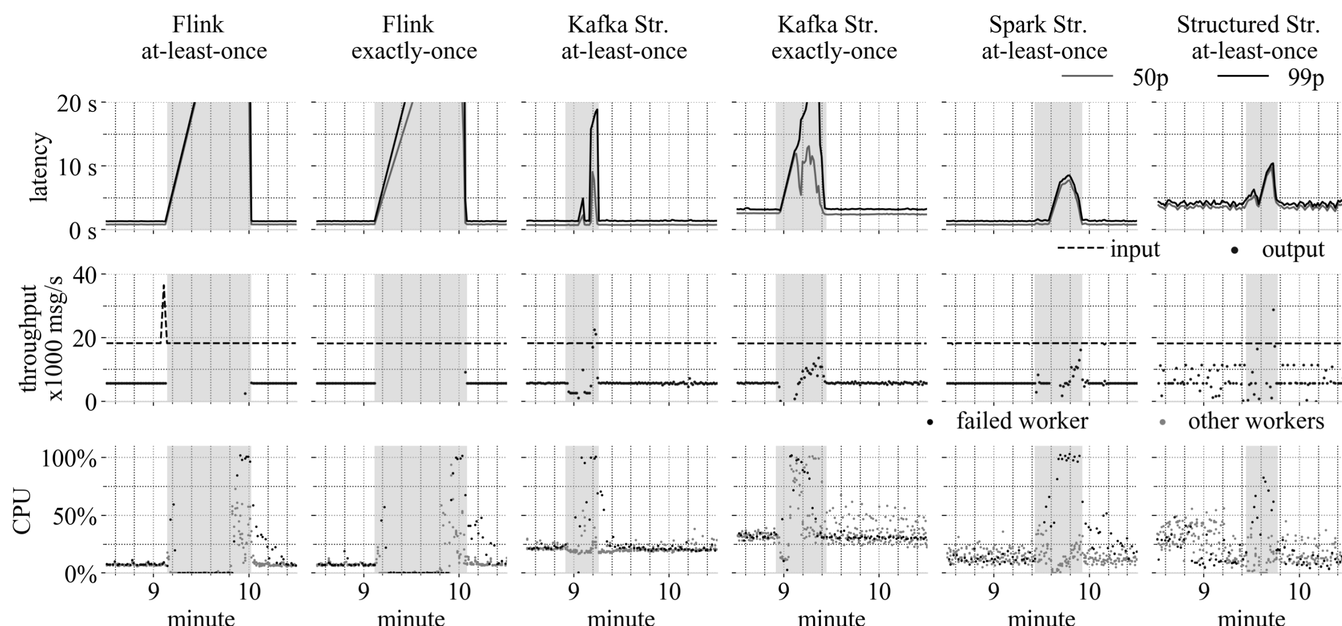

FIGURE 4. Performance metrics for workload with failure of worker-1. The grey zone denotes the recovery period.

for both frameworks during the restart (grey zone). We see slightly increased latency during this restart period (grey zone) since tasks had to be rescheduled and therefore took longer. When the grey zone ends and the worker has restarted, we see a larger latency increase and a gap in throughput of a few seconds. This is the phase where the job is scaling up again and the new executor is added to the job. After a downtime of 4 seconds, it takes 16 seconds for the latency of Structured Streaming to return to a normal state.

The fourth framework, Kafka Streams, takes yet another approach to recover from worker failures. A Kafka Streams application runs with several instances that do not communicate with each other. All communication goes via the Kafka brokers. Scaling of Kafka Streams applications can be done by adding or removing instances. Hence, a failing instance is the same as downscaling. Every instance has several processing threads running that get partitions of the input topics assigned to them. When one instance falls out, the cluster rebalances and reassigns partitions to the remaining threads. In earlier versions, Kafka Streams stopped processing during a rebalancing operation and reassigned partitions in a round-robin fashion. Of course, this leads to long processing pauses and unnecessary migration of partitions. In fact, only the partitions of the failed worker have to be reassigned and migrated, not all partitions. Since version 2.4, Kafka Streams implements a different approach called the incremental cooperative rebalancing protocol [51]. The newer protocol implements a smarter way of incrementally moving partitions to other workers and is better at handling temporary imbalances. To find the new partition assignment, it uses a cooperative sticky assignor. This partition assignor attempts to remove the least amount of partitions, i.e. sticky, to reach a new balance. Instead of stopping the processing of all partitions, this protocol just halts processing for the partitions that need to change ownership. This is done by doing two rebalancing operations right after each other, as is reflected in the latency pattern in Figure 4. In the first rebalancing operation, only the partitions that need to move ownership are revoked. In the second rebalance, the revoked partitions are assigned to new owners. During these two rebalancing operations, processing for the other partitions continues. This is why the figures show a decrease in throughput during the failure but no drop to zero. To avoid multiple subsequent rebalancing operations in the case of restarts or when multiple instances are added or removed, Kafka Streams allows short periods of imbalance. During the restart (beginning of the grey zone), we see a drop in throughput, although latency stays fairly constant. After the worker comes back up, the system starts rebalancing and we see a steep increase in latency coming from the newly assigned partitions that accumulated a processing backlog. The latency chart shows the latency of the events that were outputted in that second. Therefore, the latency is so low in the first seconds of the grey zone because it only shows the events from the partitions that were still processed. The data from the halted partitions are represented in the peak in latency after the rebalancing operation.

B. PEAK p99 LATENCY

The peak p99 latency depends on the design of the framework. The latency of Flink climbs up to 53 seconds for the first seconds after the job comes back up. This number is equal to the downtime and recovery time because all processing was halted. The downtime of Kafka Streams (2s) is much lower than its peak p99 latency (19s). This can be explained by the fact that the partitions that were assigned to the failed worker are the only ones that were not processed anymore.

For those, the peak latency is a time estimate of the entire rebalancing operation. The Spark frameworks have the lowest latency peak since the downtime is quite short and the framework catches up with the delay in a few subsequent batches. Processing happens equally fast on all partitions at all times, as can be seen from the similar median and p99 latency. This is clearly different behavior than for Kafka Streams, where the median is much lower than the p99 latency. Structured Streaming has a peak p99 latency of 10.5 seconds and the downtime lasted 4 seconds. The p99 peak latency of Spark Streaming is slightly lower, at 8.5 seconds shortly after a gap in the output of 6 seconds. Without a failure, the latency and batch interval of Spark Streaming are much lower than that of Structured Streaming [5]. Hence, the relative latency increase is much larger for Spark Streaming than for Structured Streaming. Structured Streaming can make use of its dynamic micro-batches to catch up with the delay faster and more efficiently, as confirmed by [5].

As can be seen, the influence of the design decisions on the peak latency during recovery is large and can be an important aspect to keep in mind for jobs with tight latency SLAs. The latency of event-driven systems strongly depends on the manner of fault recovery. And due to the ease of rescheduling tasks, a micro-batch approach leads to a lower latency cost throughout recovery. Despite that, we need to keep in mind that the latency of micro-batch frameworks is much higher during failure-free execution for most pipelines [5].

C. JOB STABILITY AFTER RECOVERY

When we compare performance before and after failure, we see stable performance for all frameworks. All frameworks can recover and redistribute work to the new worker, as can be seen from the similar CPU utilization across all workers. When we look at throughput, we see that Flink has a very stable output throughput. When it starts processing again we see a few seconds of elevated throughput, after that the output throughput does not fluctuate anymore and is extremely stable. For the other frameworks, the stabilization takes longer. Structured Streaming has a bursty throughput pattern due to its dynamic batch interval. If the batch interval takes longer than a second, multiple batches may be sent out at once because the watermark has passed event time for both batches, leading to a peak for the second in which that happens. The peaks mostly contain double the number of events of the normal output throughput since it is the output of two batches. We see this behavior before and after the failure.

D. MESSAGE DELIVERY GUARANTEES: IMPLICATIONS ON ACCURACY, PERFORMANCE, AND COST

Every processing semantic uses different fault tolerance mechanisms within the framework. In this section, we have a look at the impact on accuracy, performance, and cost of this. For the performance impact, we mainly look at latency and CPU utilization. We do not measure the impact on peak throughput but assume that the influence on latency and CPU utilization can serve as a proxy for this.

As we elaborated earlier, Flink and Kafka Streams use different implementations to provide exactly-once semantics. Flink was the only framework that had duplicate output in case of a worker failure. The reason for this is that the job got canceled and had to start processing from the last checkpoint, while the other frameworks were able to continue processing and only processed each data point once. In our experiment with at-least-once processing for Flink, two seconds of data right before the failure got processed twice. The number of duplicate messages depends on the exact moment of the failure. The longer the time between the latest checkpoint and the moment of failure, the more messages are duplicated. In our experiment, we process an input throughput of around 20K messages per second. This means that if the job would fail five seconds after a successful checkpoint, five seconds of data would have to be reprocessed. This would lead to 100K events being processed twice and to approximately 31K duplicate output records on Kafka. Besides data duplication, this redundant processing also slows down recovery. Therefore, the checkpoint interval needs to be carefully tuned to avoid an excessive checkpointing overhead but guarantee a speedy recovery.

For Flink, two important settings need to be adapted to get exactly-once semantics. It has to be set as the semantic of the Kafka sink. Besides that, the Kafka consumer that reads the outputs of this sink should have its isolation level set to read_committed. Hence, to provide exactly-once semantics with its Kafka sink, Flink uses Kafka features and depends on the correct configuration of downstream processing systems. This is not optimal because uncommitted events can end up on Kafka during a failure. These events are published again during the recovery and are now committed successfully. When the downstream system is not set correctly, the uncommitted and committed events are processed, leading to duplicates. Finally, enabling exactly-once semantics had no real impact on latency and CPU utilization.

Kafka Streams is also able to deliver correct output for both cases. We see, however, a large impact of enabling exactly-once semantics. Average CPU utilization increases by 10 percent from 20 percent to 30 percent. There is an increase in latency from 730 ms to 2400 ms. Finally, recovery takes ten seconds longer with exactly-once semantics.

The current Kafka sink implementation of Spark Streaming and Structured Streaming does not offer exactly-once semantics. For worker failures, the Spark job also does not restart, so we have correct and unaffected behavior. Spark jobs only restart in the case of driver failures. We refer back to Section V-C on driver failures for the behavior of job restarts with at-most-once and at-least-once semantics.

As we mentioned earlier, the preferred message delivery guarantee depends on the use case. Losing a few data points is not equally detrimental for all use cases. However, for most use cases, it will be preferable not to lose any data. For the systems that only offer at-least-once processing, this means that downstream systems will need to be robust against duplicates. Additionally, these frameworks have some

prerequisites to ensure correctness (see Section III-B). In short, the sink either needs to be idempotent or implement atomic transactions. Furthermore, processing needs to be deterministic. Flink and Kafka Streams offer exactly-once semantics. Nonetheless, the same correctness prerequisites count here and there is a dependency on downstream systems.

E. CONCLUSION

When confronted with worker failures, both Spark systems have an advantage since they can easily reschedule tasks, leading to low downtime, low recovery times, and lower peak latencies. The dynamic micro-batch approach of Structured Streaming speeds up recovery even further. Event-driven systems halt processing of at least part of the partitions and therefore, show higher peak latencies. Flink restarts the entire job from the latest checkpoint to guarantee consistency, causing long downtime and recovery time. Additionally, Flink relies on the settings of downstream processing systems to prevent duplicate output. Kafka Streams recovers faster because it continues processing the partitions of the healthy instances and therefore, has less of a delay. However, we see a large performance hit when enabling exactly-once semantics. Table 4 gives a concise overview of these results.

VII. APPLICATION, JOB, STAGE, AND TASK FAILURES

It is also possible that the streaming application itself fails. Applications can fail for numerous reasons: invalid pipeline definitions, configuration issues, memory leaks, etc. Since application failures can be triggered by a broad range of issues, we do not include a generic experiment for this type of failure. When a Spark application fails, it is automatically rescheduled if the supervise flag is passed in the submit command. Flink also automatically restarts the application when this is possible. For Kafka Streams, whether or not an instance restarts depends on the underlying scheduling infrastructure that was used. In our setup, we use Marathon on Mesos to reschedule exited containers.

Issues can lead to different types of failures for different frameworks. An example of this is uncaught exceptions within the application, e.g. due to data quality issues. As an experiment, we inserted some corrupted data in the stream and monitored the effect.

Spark and Structured are micro-batch frameworks. Each micro-batch is processed in a micro-batch job consisting of stages and each stage consists of multiple tasks. The stages represent the different nodes of the JobGraph, while the tasks represent the partitions that are processed. When corrupted data is ingested in Spark, the ingestion task fails, causing that stage and therefore that micro-batch job to fail as well. The application, i.e. driver and executors, keeps running. The micro-batch job will be retried up to 4 times by default (as set by the parameter spark.task.maxFailures). When the task fails four times, the application will give up on the micro-batch job and continue with the following one. When we ingest the same corrupted data in Flink, we get very different default behavior. The entire application gets canceled and is restarted because the ingest stage fails. We use the default fixed-delay restart strategy with maximum restart attempts. This means that Flink will attempt to restart the job a specified number of times and will wait a fixed amount of time in between restarts [11]. Since the job cannot get past the faulty events, the job never continues processing. For Kafka Streams, ingesting corrupted data caused an exception in the stream thread coming from the LogAndFailExceptionHandler. Even though the processing thread died, the containers did not shut down. The developers of Kafka Streams have been improving this behavior in recent versions. Now, when one or multiple threads fail, users can initiate an application shutdown or replace the failed thread.

We can conclude by saying that it is evident that error handling should be part of the application design and implementation and should be adapted to the use case. Users should be aware of the behavior of the framework under different types of faults.

VIII. DISCUSSION

The results of the fault recovery experiments have been summarized in Table 4. We noticed large differences in the way component faults are handled. Most of these differences lead back to the characteristics of the framework and mainly whether it is micro-batch or event-driven and whether it uses a master-slave architecture or not.

Both Spark frameworks are quite resilient against faults due to their task-based scheduling approach. When tasks fail, for a variety of possible reasons, they are retried on other executors and the job can continue processing. This behavior is apparent in both Spark Streaming and Structured Streaming jobs for worker failures and task failures. Compared to the other frameworks, Spark Streaming and Structured Streaming recover fast and have low peak latencies during recovery. In the case of master failures, Spark requires a high-availability setup to continue processing without any impact. Driver failures are the only scenario in which the application gets killed (besides single master failure). Even though Spark Streaming and Structured Streaming are resilient against component faults, exactly-once semantics are not guaranteed for most sinks. For use cases for which this is required, this would require a custom implementation which is error-prone and more complex.

Other frameworks such as Flink take a more careful approach and restart the entire job from the last successful checkpoint in case of a failure, e.g. master failure, worker failure, and task failure. This leads to longer outages, but correct results, albeit with dependencies on downstream systems for exactly-once semantics.

Due to the architecture of Kafka Streams, instances work more independently, which makes it resilient to worker failures because processing can continue for the unaffected partitions. Due to this, Kafka Streams experiences almost no downtime. It only accumulates a processing backlog for the partitions that were halted and can, therefore, catch up fast. However, the exactly-once processing mechanism of

Kafka Streams causes more performance degradation than that of Flink.

A. LESSONS AND BEST PRACTICES

When choosing a framework for a specific use case, fault tolerance may be one of the decision criteria. Many use cases have SLAs on the allowed downtime and peak latencies. The user should define his objectives for recovery time and data loss or duplication and determine the best fitting framework based on the results listed in Table 4.

When it is most important that recovery happens fast and that peak latencies remain low and when duplicates can be tolerated, Spark Streaming and Structured Streaming are the most stable and only have significant downtime in the case of driver failures. Although, it needs to be kept in mind that their latencies are higher during fault-free processing than those of event-driven frameworks. It should be evaluated whether they are still low enough for the use case at hand. Also, the Spark frameworks do not support exactly-once semantics so any downstream systems need to be able to handle duplicate data.

When minimal downtime or only partial downtime is a requirement, Kafka Streams performs best. Furthermore, it has no master or driver components so it can only suffer from worker failures, to which it is resilient. Previous work showed that the latency and throughput of Kafka Streams is often worse than that of other frameworks [5]. The exactly-once semantics of Kafka Streams degrade performance even further, so if this is required, Flink may be the preferred option. Furthermore, Kafka Streams makes use of Kafka for fault tolerance so can only be used when a Kafka cluster is available.

For some use cases, data loss or duplicate processing is detrimental. These types of use cases, which require exactly-once semantics and correctness under all types of faults, benefit most from using Flink. It has higher recovery times because of job restarts but it also has a much better latency-throughput trade-off during failure-free execution, as shown by numerous previous studies. The Flink framework is well adapted to use cases with a large state because they offer advanced state management options, e.g. asynchronous and incremental checkpoints with RocksDB. Additionally, many checkpointing parameters can be tuned such as the checkpoint interval and the minimum time in between checkpoints.

In short, the fault recovery mechanism of Spark is stable, fast, and resilient but may output duplicates. The mechanism of Flink is correct, tunable, and careful but slow. Kafka Streams has the lowest downtime and is very resilient against failures but often has worse general performance.

Once a framework has been selected, the implementation should follow some best practices to fully benefit from the fault tolerance capability. First of all, production workloads should use a high-availability setup. Otherwise, the master is a single point of failure. Single master failures make jobs fail permanently.

Secondly, to get accurate results, some additional prerequisites need to be fulfilled. We described these prerequisites throughout Section III. For example, the pipeline needs to be deterministic. We described, throughout this work, how determinism cannot be guaranteed by default for certain operations, such as processing time windows and retrieving side inputs from external APIs or databases.

Some frameworks offer exactly-once semantics. However, these have some additional limitations. They only guarantee that each incoming event affects the final results exactly once and is committed at the sinks exactly once. They do not apply to the execution of user code within functions, known as side outputs. For example, when the user writes to a database in the user code, this may be executed twice when confronted with a failure. Therefore, these operations should be idempotent. Also, sinks need to be idempotent or implement transactions.

B. FUTURE RESEARCH DIRECTIONS

The developer and academic communities actively research how to improve the fault tolerance of these stream processing systems. There are numerous publications in this field on various subjects. In this section, we briefly give an overview of some of these efforts. For more extensive surveys on fault tolerance and high-availability in stream processing, we refer to the following surveys: [20], [52]–[55]. Before we begin this review, we want to stress that frameworks that are actively improved, provide the best guarantees for continued future adoption and support. It should, therefore, be taken into account when choosing a stream processing framework for a use case.

1) FAULT TOLERANCE

One topic with room for improvement is exactly-once semantics and output commit protocols (Section III). Flink uses the Two-Phase Commit Protocol to commit transactions when writing to Kafka. The Two-Phase Commit Protocol has some disadvantages such as that it requires two steps to complete. Some academic efforts study approaches to make transaction commits more performant and less heavy, e.g., [56], [57]. The Structured Streaming community has acknowledged the need for exactly-once processing semantics in its Kafka sink. However, as far as we know, there is no concrete progress on this topic.

Our results showed that Flink often had a slow recovery because of job restarts. The community is working on features called reactive mode [58], declarative resource management [59] and an adaptive scheduler [60]. With these features, Flink would be able to react to newly available resources and a job would be able to run when not all required task slots are available. Currently, this is not allowed. This would also make it possible to do local failover in which only a part of the topology is restarted when a failure happens. Currently, Flink only supports global failover in case of a failure. These features are planned for version 1.13 and can mean big steps forward for Flink as a framework.

In this work, we mentioned the importance of determinism for fault tolerance. Some studies are conducted on fault tolerance for non-deterministic pipelines. An interesting work

is Clonos [61] which is implemented on top of Flink and uses checkpointing, causal logging, and standby operators to enable exactly-once consistent local recovery with support for non-deterministic computations.

Another research area uses predictions to improve fault tolerance. For example, Huang and Lee [62] present a system that uses approximate fault tolerance. It tries to estimate the errors upon failures and adaptively triggers state backups when the error goes beyond a user-defined threshold. Another example is the prediction of failures to enable proactive preventive failure management [63], [64].

In our experiments, the input stream had a stable data rate. However, this is not the case for all use cases. There have been some studies on combining mechanisms for fault tolerance and elasticity by dynamically scaling resources according to the runtime workload fluctuations [65]–[67]. Resources are scaled by either activating redundant replicas or by acquiring new resources during high-load periods.

As can be noticed from our experiments, the fault tolerance of a processing job is largely dependent on the management and backup of state. Most frameworks use checkpointing. This becomes heavy in pipelines with a large state. The development communities of stream processing frameworks still actively investigate how to make checkpointing lighter and more efficient. Recent releases of Flink included some new features regarding checkpointing. Flink now supports unaligned checkpoints [52], [68]. Flink uses markers to mark the frontiers of a checkpoint epoch and uses an alignment phase to synchronize markers from different input channels. During this alignment phase, the faster channels are temporarily blocked, slowing down the processing speed. Unaligned checkpointing is a non-blocking alternative that allows records to overtake an epoch frontier. Another recent addition is concurrent checkpointing which allows multiple checkpointing operations to be in progress at the same time. For lengthy checkpointing operations, this reduces the time between checkpoints and, thereby, makes incremental checkpoints smaller and fault recovery lighter.

As we mentioned in the previous section, the Spark frameworks are not a good fit for latency-sensitive use cases. The Spark community is working on a Continuous Processing API for Structured Streaming [69]. This API offers low-latency event-driven processing on top of Spark. Currently, the API is still experimental and supports limited operations. This API also bases itself on the Chandy-Lamport algorithm [3] for checkpointing, similar to Flink and many other event-driven frameworks [24], [25]. These developments may make Structured Streaming a competitor for Flink on latency-sensitive use cases.

Improving checkpointing is also a topic in academic research. Some of this work is centered around optimizing the checkpoint interval [70], [71]. As we described in our results, the checkpoint interval offers a trade-off between recovery speed and checkpointing overhead. Some new checkpointing methods have been proposed. To *et al.* [54] give a thorough review of research on several checkpointing methods such as correlated checkpoints, independent checkpoints, and incremental checkpoints.

Additionally, some academic work investigates alternative mechanisms for state management in general, e.g. replicating state on workers [72]. We refer to [54] for an extensive overview on the topic of state management and checkpointing in streaming systems.

2) HIGH-AVAILABILITY SCHEMES

A second active area of research is centered around high-availability schemes. At the moment, most popular stream processing frameworks make use of a passive replication approach. However, there are some other schemes: e.g. active replication [73], [74] and hybrid active-passive replication [75]–[78]. An interesting work on this topic is [79], in which several resilience strategies are compared and evaluated. They model the cost of different strategies through a series of experiments. A thorough overview of research on high-availability setups in stream processing has been given in [33] and [52].

Finally, from the four frameworks we compared here, Flink is the most active when it comes to releasing new fault tolerance features. It is also used most often in academic research to try out new fault-tolerance mechanisms, e.g. [61]. However, the stream processing field is still very dynamic. New frameworks such as Structured Streaming Continuous Processing API and Hazelcast Jet [25] are actively developed and may change the focus of the field.

IX. CONCLUSION

Fault tolerance of stream processing jobs is of utmost importance in production deployments but has been overlooked in benchmarking literature. In this work, we contribute a broader framework of what should be included when benchmarking this feature. We conduct tests on the behavior during several types of failures and with several setups and configurations. To do a range of experiments, we implemented three fault-tolerance workloads on top of OSPBench. Our workload implementations have been incorporated in the OSPBench codebase at https://github.com/Klarrio/open-stream-processing-benchmark. Based on these workloads, we conduct an extensive analysis of four popular distributed stream processing frameworks: Flink, Kafka Streams, Spark Streaming, and Structured Streaming. We identified guidelines of how these results influence the selection of a framework for a use case and we listed some lessons and best practices. Finally, we also describe some current and future research directions to improve fault tolerance in stream processing frameworks.

X. LIMITATIONS AND FURTHER RESEARCH

Some further research can be done on the other high-availability schemes that make use of cluster managers such as YARN, Mesos, and Kubernetes. Also, as recovery becomes heavier when the state is large, the influence of state sizes on fault recovery can be investigated. In our fault recovery workload, the state was fairly small, under 20 MB.

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 and Big Data/Analytics project implementations (https://klarrio.com).

REFERENCES

[1] F. Cristian, "Understanding fault-tolerant distributed systems," *Commun. ACM*, vol. 34, no. 2, pp. 56–78, Feb. 1991.

[2] T. Akidau, R. Bradshaw, C. Chambers, S. Chernyak, R. J. Fernández-Moctezuma, R. Lax, S. McVeety, D. Mills, F. Perry, E. Schmidt, and S. Whittle, "The dataflow model: A practical approach to balancing correctness, latency, and cost in massive-scale, unbounded, out-of-order data processing," *Proc. VLDB Endowment*, vol. 8, no. 12, pp. 1792–1803, Aug. 2015.

[3] K. M. Chandy and L. Lamport, "Distributed snapshots: Determining global states of distributed systems," *ACM Trans. Comput. Syst.*, vol. 3, no. 1, pp. 63–75, Feb. 1985.

[4] L. Lamport, "Time, clocks, and the ordering of events in a distributed system," in *Proc. Concurrency, Works Leslie Lamport*, 2019, pp. 179–196.

[5] G. V. Dongen and D. V. D. Poel, "Evaluation of stream processing frameworks," *IEEE Trans. Parallel Distrib. Syst.*, vol. 31, no. 8, pp. 1845–1858, Dec. 2020.

[6] J. Karimov, T. Rabl, A. Katsifodimos, R. Samarev, H. Heiskanen, and V. Markl, "Benchmarking distributed stream data processing systems," 2018, *arXiv:1802.08496*. [Online]. Available: http://arxiv.org/abs/1802.08496

[7] Z. Karakaya, A. Yazici, and M. Alayyoub, "A comparison of stream processing frameworks," in *Proc. Int. Conf. Comput. Appl. (ICCA)*, Sep. 2017, pp. 1–12.

[8] S. Chintapalli, D. Dagit, B. Evans, R. Farivar, T. Graves, M. Holderbaugh, Z. Liu, K. Nusbaum, K. Patil, B. J. Peng, and P. Poulosky, "Benchmarking streaming computation engines: Storm, flink and spark streaming," in *Proc. IEEE Int. Parallel Distrib. Process. Symp. Workshops (IPDPSW)*, May 2016, pp. 1789–1792.

[9] S. Qian, G. Wu, J. Huang, and T. Das, "Benchmarking modern distributed streaming platforms," in *Proc. IEEE Int. Conf. Ind. Technol. (ICIT)*, Mar. 2016, pp. 592–598.

[10] R. Lu, G. Wu, B. Xie, and J. Hu, "Stream bench: Towards benchmarking modern distributed stream computing frameworks," in *Proc. IEEE/ACM 7th Int. Conf. Utility Cloud Comput.*, Dec. 2014, pp. 69–78.

[11] (2020) *Apache Flink: Stateful Computations Over Data Streams*. Accessed: 2020-01-28. [Online]. Available: https://flink.apache.org/

[12] (2020). *Apache Storm*. Accessed: Jan. 28, 2020. [Online]. Available: https://storm.apache.org/

[13] (2020). *Apache Samza: A Distributed Stream Processing Framework* Accessed: Mar. 23, 2020. [Online]. Available: http://samza.apache.org/

[14] (2020). *Apache Kafka: Kafka Streams*. Accessed: Jan. 28, 2020. [Online]. Available: https://kafka.apache.org/documentation/streams/

[15] (2020). *Apache Kafka*. Accessed: Mar. 28, 2020. [Online]. Available: https://kafka.apache.org/

[16] M. A. Lopez, A. G. P. Lobato, and O. C. MBDuarte, "A performance comparison of open-source stream processing platforms," in *Proc. IEEE Global Commun. Conf. (GLOBECOM)*, Dec. 2016, pp. 1–6.

[17] T. Akidau, S. Chernyak, and R. Lax, *Streaming Systems: What, Where, When, How Large-Scale Data Processing*. Newton, MA, USA: O'Reilly Media, 2018.

[18] *An Overview of End-to-End Exactly-Once Processing in Apache Flink* Accessed: Sep. 4, 2020. [Online]. Available: https://flink.apache.org/features/2018/03/01/end-to-end-exactly-once-apache-flink.html

[19] Y. Kwon, M. Balazinska, and A. Greenberg, "Fault-tolerant stream processing using a distributed, replicated file system," *Proc. VLDB Endowment*, vol. 1, no. 1, pp. 574–585, Aug. 2008.

[20] E. N. Elnozahy, L. Alvisi, Y.-M. Wang, and D. B. Johnson, "A survey of rollback-recovery protocols in message-passing systems," *ACM Comput. Surv.*, vol. 34, no. 3, pp. 375–408, Sep. 2002.

[21] P. Carbone, A. Katsifodimos, S. Ewen, V. Markl, S. Haridi, and A. K. Tzoumas, "Apache flink: Stream and batch processing in a single engine," *Bull. IEEE Comput. Soc. Tech. Committee Data Eng.*, vol. 36, no. 4, pp. 1–12, 2015.

[22] P. Carbone, G. Fóra, S. Ewen, S. Haridi, and K. Tzoumas, "Lightweight asynchronous snapshots for distributed dataflows," 2015, *arXiv:1506.08603*. [Online]. Available: http://arxiv.org/abs/1506.08603

[23] P. A. Tucker, D. Maier, T. Sheard, and L. Fegaras, "Exploiting punctuation semantics in continuous data streams," *IEEE Trans. Knowl. Data Eng.*, vol. 15, no. 3, pp. 555–568, May 2003.

[24] G. Jacques-Silva, F. Zheng, D. Debrunner, K.-L. Wu, V. Dogaru, E. Johnson, M. Spicer, and A. E. Sariyäce, "Consistent regions: Guaranteed tuple processing in IBM streams," *Proc. VLDB Endowment*, vol. 9, no. 13, pp. 1341–1352, Sep. 2016.

[25] C. Gencer, M. Topolnik, V. Durina, E. Demirci, E. B. Kahveci, A. Gürbüz, O. Lukáš, J. Bartók, G. Gierlach, F. Hartman, U. Yilmaz, M. Dogan, M. Mandouh, M. Fragkoulis, and A. Katsifodimos, "Hazelcast Jet: Low-latency stream processing at the 99.99th percentile," 2021, *arXiv:2103.10169*. [Online]. Available: http://arxiv.org/abs/2103.10169

[26] 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 *Proc. Int. Conf. Manage. Data*, 2018, pp. 601–613.

[27] M. Zaharia, T. Das, H. Li, T. Hunter, S. Shenker, and I. Stoica, "Discretized streams: Fault-tolerant streaming computation at scale," in *Proc. 24th ACM Symp. Oper. Syst. Princ.*, Nov. 2013, pp. 423–438.

[28] M. Zaharia, T. Das, H. Li, T. Hunter, S. Shenker, and I. Stoica, "Discretized streams: Fault-tolerant streaming computation at scale," in *Proc. 24th ACM Symp. Oper. Syst. Princ.*, Nov. 2013, pp. 423–438.

[29] M. Zaharia, M. Chowdhury, T. Das, A. Dave, J. Ma, M. McCauley, M. J. Franklin, S. Shenker, and I. Stoica, "Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing," in *Proc. 9th USENIX Conf. Netw. Syst. Design Implement.*, 2012, pp. 15–28.

[30] M. J. Sax, G. Wang, M. Weidlich, and J.-C. Freytag, "Streams and tables: Two sides of the same coin," in *Proc. Int. Workshop Real-Time Bus. Intell. Anal.*, 2018, p. 1.

[31] P. Carbone, S. Ewen, G. Fóra, S. Haridi, S. Richter, and K. Tzoumas, "State management in Apache Flink: Consistent stateful distributed stream processing," *Proc. VLDB Endowment*, vol. 10, no. 12, pp. 1718–1729, Aug. 2017.

[32] S. A. Noghabi, K. Paramasivam, Y. Pan, N. Ramesh, J. Bringhurst, I. Gupta, and R. H. Campbell, "Samza: Stateful scalable stream processing at Linkedin," *Proc. VLDB Endowment*, vol. 10, no. 12, pp. 1634–1645, Aug. 2017.

[33] J.-H. Hwang, M. Balazinska, A. Rasin, U. Cetintemel, M. Stonebraker, and S. Zdonik, "High-availability algorithms for distributed stream processing," in *Proc. 21st Int. Conf. Data Eng. (ICDE)*, 2005, pp. 779–790.

[34] J. Kreps, N. Narkhede, and J. Rao, "Kafka: A distributed messaging system for log processing," in *Proc. NetDB*, 2011, pp. 1–7.

[35] R. F. Pausch, "Adding input and output to the transactional model," Carnegie Mellon Univ., Pittsburgh, PA, USA, Tech. Rep. AAI8905256, 1989.

[36] (2020). *Apache Spark: Spark Streaming Programming Guide*. Accessed: Jan. 28, 2020. [Online]. Available: https://spark.apache.org/docs/latest/streaming-programming-guide.html

[37] J. N. Gray, "Notes on data base operating systems," in *Operating Systems*. Berlin, Germany: Springer, 1978, pp. 393–481.

[38] *Enabling Exactly-Once in Kafka Streams*. Accessed: Sep. 4, 2020. [Online]. Available: https://www.confluent.io/blog/enabling-exactly-once-kafka-streams/

[39] *Apache Zookeeper*. Accessed: Sep. 4, 2020. [Online]. Available: https://zookeeper.apache.org/

[40] (2017). *NDW: Nationale Databank Wegverkeersgegevens*. Accessed: Jul. 25, 2017. [Online]. Available: http://www.ndw.nu/

[41] G. V. Dongen, B. Steurtewagen, and D. V. D. Poel, "Latency measurement of fine-grained operations in benchmarking distributed stream processing frameworks," in *Proc. IEEE Int. Congr. Big Data*, Jul. 2018, pp. 247–250.

[42] (2021). *DC/OS*. Accessed: Jan. 28, 2021. [Online]. Available: https://dcos.io/

[43] (2021). *Docker*. Accessed: Jan. 28, 2021. [Online]. Available: https://www.docker.com/

[44] *Advisor*. Accessed: Jun. 4, 2019. [Online]. Available: https://github.com/google/cadvisor

[45] (2021). *Hadoop: HDFS Architecture Guide*. Accessed: Mar. 28, 2021. [Online]. Available: https://hadoop.apache.org/docs/r1.2.1/hdfs_design.html

[46] S. Henning and W. Hasselbring, "Theodolite: Scalability benchmarking of distributed stream processing engines in microservice architectures," 2020, *arXiv:2009.00304*. [Online]. Available: http://arxiv.org/abs/2009.00304
[47] D. Wang and J. Huang. (2015). *Tuning Java Garbage Collection for Apache Spark Applications*. Accessed: Dec. 6, 2018. [Online]. Available: https://databricks.com/blog/2015/05/28/tuning-java-garbage-collection-for-spark-applications.html
[48] Y. Byzek. *Optimizing Your Apache Kafka Deployment*. Accessed: Dec. 18, 2018. [Online]. Available: https://www.confluent.io/wp-content/uploads/Optimizing-Your-Apache-Kafka-Deployment-1.pdf
[49] (2020). *Apache Spark: Structured Streaming Programming Guide*. Accessed: Jan. 28, 2020. [Online]. Available: https://spark.apache.org/docs/latest/structured-streaming-programming-guide.html
[50] T. Das, Y. Zhong, I. Stoica, and S. Shenker, "Adaptive stream processing using dynamic batch sizing," in *Proc. ACM Symp. Cloud Comput.*, Nov. 2014, pp. 1–13.
[51] (2020). *From Eager to Smarter in Apache Kafka Consumer Rebalances*. Accessed: Aug. 28, 2020. [Online]. Available: https://www.confluent.jp/blog/cooperative-rebalancing-in-kafka-streams-consumer-ksqldb/
[52] M. Fragkoulis, P. Carbone, V. Kalavri, and A. Katsifodimos, "A survey on the evolution of stream processing systems," 2020, *arXiv:2008.00842*. [Online]. Available: http://arxiv.org/abs/2008.00842
[53] M. Dayarathna and S. Perera, "Recent advancements in event processing," *ACM Comput. Surv.*, vol. 51, no. 2, pp. 1–36, Jun. 2018.
[54] Q.-C. To, J. Soto, and V. Markl, "A survey of state management in big data processing systems," *VLDB J.*, vol. 27, no. 6, pp. 847–872, Dec. 2018.
[55] P. Carbone, M. Fragkoulis, V. Kalavri, and A. Katsifodimos, "Beyond analytics: The evolution of stream processing systems," in *Proc. ACM SIGMOD Int. Conf. Manage. Data*, Jun. 2020, pp. 2651–2658.
[56] J. Gray and L. Lamport, "Consensus on transaction commit," *ACM Trans. Database Syst.*, vol. 31, no. 1, pp. 133–160, Mar. 2006.
[57] M. Bravo and A. Gotsman, "Reconfigurable atomic transaction commit," in *Proc. ACM Symp. Princ. Distrib. Comput.*, Jul. 2019, pp. 399–408.
[58] T. Rohrmann. (2021). *Flip-159*. Accessed: Mar. 28, 2021. [Online]. Available: https://cwiki.apache.org/confluence/display/FLINK/FLIP-159%3A+Reactive+Mode
[59] C. Schepler. (2021). *Flip-138: Declarative Resource Management*. Accessed: Mar. 28, 2021. [Online]. Available: https://cwiki.apache.org/confluence/display/FLINK/FLIP-138%3A+Declarative+Resource+management
[60] T. Rohrmann. (2021) *Flip-160: Adaptive Scheduler*. Accessed: Mar. 28, 2021. [Online]. Available: https://cwiki.apache.org/confluence/display/FLINK/FLIP-160%3A+Adaptive+Scheduler
[61] P. F. Silvestre, M. Fragkoulis, D. Spinellis, and A. Katsifodimos, "Clonos: Consistent causal recovery for highly-available streaming dataflows," in *Proc. Int. Conf. Manage. Data*, Jun. 2021, pp. 1637–1650.
[62] Q. Huang and P. P. C. Lee, "Toward high-performance distributed stream processing Via approximate fault tolerance," *Proc. VLDB Endowment*, vol. 10, no. 3, pp. 73–84, Nov. 2016.
[63] X. Gu, S. Papadimitriou, P. S. Yu, and S.-P. Chang, "Online failure forecast for fault-tolerant data stream processing," in *Proc. IEEE 24th Int. Conf. Data Eng.*, Apr. 2008, pp. 1388–1390.
[64] X. Gu and K. Nahrstedt, "On composing stream applications in peer-to-peer environments," *IEEE Trans. Parallel Distrib. Syst.*, vol. 17, no. 8, pp. 824–837, Aug. 2006.
[65] R. Castro Fernandez, M. Migliavacca, E. Kalyvianaki, and P. Pietzuch, "Integrating scale out and fault tolerance in stream processing using operator state management," in *Proc. Int. Conf. Manage. Data*, 2013, pp. 725–736.
[66] P. Bellavista, A. Corradi, S. Kotoulas, and A. Reale, "Adaptive fault-tolerance for dynamic resource provisioning in distributed stream processing systems," in *Proc. EDBT*, 2014, pp. 85–96.
[67] K. G. S. Madsen, P. Thyssen, and Y. Zhou, "Integrating fault-tolerance and elasticity in a distributed data stream processing system," in *Proc. 26th Int. Conf. Sci. Stat. Database Manage.*, 2014, pp. 1–4.
[68] A. Heise. (2020). *Flip-76: Unaligned Checkpoints*. Accessed: Mar. 28, 2021. [Online]. Available: https://cwiki.apache.org/confluence/display/FLINK/FLIP-76%3A+Unaligned+Checkpoints
[69] J. Torres, M. Armbrust, T. Das, and S. Zhu. (2018). *Introducing Low-Latency Continuous Processing Mode in Structured Streaming in Apache Spark 2.3*. Accessed: Mar. 28, 2021. [Online]. Available: https://databricks.com/blog/2018/03/20/low-latency-continuous-processing-mode-in-structured-streaming-in-apache-spark-2-3-0.html
[70] S. Jayasekara, A. Harwood, and S. Karunasekera, "A utilization model for optimization of checkpoint intervals in distributed stream processing systems," *Future Gener. Comput. Syst.*, vol. 110, pp. 68–79, Sep. 2020.
[71] Z. Zhang, W. Li, X. Qing, X. Liu, and H. Liu, "Research on optimal checkpointing-interval for flink stream processing applications," *Mobile Netw. Appl.*, vol. 4, pp. 1–10, Jan. 2021.
[72] X. Liu, A. Harwood, S. Karunasekera, B. Rubinstein, and R. Buyya, "E-storm: Replication-based state management in distributed stream processing systems," in *Proc. 46th Int. Conf. Parallel Process. (ICPP)*, Aug. 2017, pp. 571–580.
[73] M. Balazinska, H. Balakrishnan, S. Madden, and M. Stonebraker, "Fault-tolerance in the Borealis distributed stream processing system," in *Proc. ACM SIGMOD Int. Conf. Manage. Data*, 2005, pp. 13–24.
[74] M. A. Shah, J. M. Hellerstein, and E. Brewer, "Highly available, fault-tolerant, parallel dataflows," in *Proc. ACM SIGMOD Int. Conf. Manage. Data*, 2004, pp. 827–838.
[75] L. Su and Y. Zhou, "Passive and partially active fault tolerance for massively parallel stream processing engines," *IEEE Trans. Knowl. Data Eng.*, vol. 31, no. 1, pp. 32–45, Jan. 2019.
[76] L. Su and Y. Zhou, "Tolerating correlated failures in massively parallel stream processing engines," in *Proc. IEEE 32nd Int. Conf. Data Eng. (ICDE)*, May 2016, pp. 517–528.
[77] Z. Zhang, Y. Gu, F. Ye, H. Yang, M. Kim, H. Lei, and Z. Liu, "A hybrid approach to high availability in stream processing systems," in *Proc. IEEE 30th Int. Conf. Distrib. Comput. Syst.*, Jun. 2010, pp. 138–148.
[78] T. Heinze, M. Zia, R. Krahn, Z. Jerzak, and C. Fetzer, "An adaptive replication scheme for elastic data stream processing systems," in *Proc. 9th ACM Int. Conf. Distrib. Event-Based Syst.*, Jun. 2015, pp. 150–161.
[79] B. Chandramouli and J. Goldstein, "Shrink: Prescribing resiliency solutions for streaming," *Proc. VLDB Endowment*, vol. 10, no. 5, pp. 505–516, Jan. 2017.

GISELLE VAN DONGEN (Member, IEEE) is currently a Ph.D. Researcher with Ghent University, teaching and benchmarking real-time distributed processing systems, such as spark streaming and flink and kafka streams. Concurrently, she is a Lead Data Scientist at Klarrio, specialising in real-time data analysis, processing, and visualisation.

DIRK VAN DEN POEL (Senior Member, IEEE) is currently a Senior Full Professor of data analytics/big data with Ghent University, Belgium. He teaches courses such as big data, databases, social media and web analytics, analytical customer relationship management, advanced predictive analytics, and predictive and prescriptive analytics. He co-founded the advanced Master of Science in marketing analysis (currently renamed to Master of Science in data science for business), the first (predictive) analytics master program in the world as well as the Master of Science in statistical data analysis, and the Master of Science in business engineering/data analytics.