Evaluating Fault Tolerance of Distributed Stream Processing Systems

Xiaotong Wang[1], Cheng Jiang[1], Junhua Fang[2], Ke Shu[3], Rong Zhang[1](✉), Weining Qian[1], and Aoying Zhou[1]

[1] School of Data Science and Engineering, East China Normal University, Shanghai, China
{wxt,jc}@stu.ecnu.edu.cn, {rzhang,wnqian,ayzhou}@dase.ecnu.edu.cn

[2] Soochow University, Suzhou, China
jhfang@suda.edu.cn

[3] PingCAP Ltd., Shanghai, China
shuke@pingcap.com

Abstract. Since failures in large-scale clusters can lead to severe performance degradation and break system availability, fault tolerance is critical for distributed stream processing systems (DSPSs). Plenty of fault tolerance approaches have been proposed over the last decade. However, there is no systematic work to evaluate and compare them in detail. Previous work either evaluates global performance during failure-free runtime, or merely measures throughout loss when failure happens. In this paper, it is the first work proposing an evaluation framework customized for quantitatively comparing runtime overhead and recovery efficiency of fault tolerance mechanisms in DSPSs. We define three typical configurable workloads, which are widely-adopted in previous DSPS evaluations. We construct five workload suites based on three workloads to investigate the effects of different factors on fault tolerance performance. We carry out extensive experiments on two well-known open-sourced DSPSs. The results demonstrate performance gap of two systems, which is useful for choice and evolution of fault tolerance approaches.

Keywords: Fault tolerance · Benchmarking · Stream processing

1 Introduction

Over the last two decades, numerous stream processing systems (SPS) have sprung up from both academia and industry, catering to the increasing requirements of continuous real-time processing from a wide range of scenarios, including stock trading, network monitoring and fraud detection. Aurora [1] starts the work in data stream management system (DSMS, one branch of SPS), which adopt a centralized architecture. Then the growth and intrinsic dispersity of data stream promote the emergence of distributed stream processing systems (DSPS) which provide advanced features. Representatives include Borealis [3].

X. Wang et al. (Eds.): APWeb-WAIM 2020, LNCS 12318, pp. 101–116, 2020.
https://doi.org/10.1007/978-3-030-60290-1_8

Since the prevalence of cloud computing, superior DSPSs have been developed, e.g. Storm [20], Spark Streaming [22] and Flink [5], which are required to provide highly-scalable and highly-available services.

Researchers have put much effort in benchmarking the characteristics (e.g., scalability) and performance (e.g., throughput and latency) of modern DSPSs [2, 4, 6, 14, 18, 21], shown in Table 1. We roughly classify these benchmarks into two types based on workloads. One type is to simulate real-world scenario, such as advertisement clicking analysis in Yahoo! StreamingBench [6]; the other type is to compose synthetic operators to evaluate the specific characteristic, e.g. sampling and projection in StreamBench [14]. However current stream benchmarks focus on performance evaluation during failure-free runtime, except that StreamBench introduces a penalty factor for latency and throughout during a failure.

Table 1. The overview of existing stream benchmarks.

Benchmarks	System	Datasets	Workloads	Metrics
LinearRoad [2]	Aurora STREAM	Road and vehicle data	Highway toll monitoring	Throughput
StreamBench [14]	Storm Flink Spark Streaming	Search internet traces	Synthetic sampling work counting, et al.	Latency Throughput Penalty factor
Yahoo! StreamingBench [6]	Storm Flink Spark Streaming	Advertising and user clicking data	Clicking analysis	Latency Throughput
YCSB Extension [9]	Storm Flink Spark Streaming	Advertising and user clicking data	Clicking analysis	Latency Throughput
StreamBench [21]	Storm Flink Spark Streaming	Synthetic data	Synthetic word counting, Clicking, KMeans	Latency Throughput

DSPSs are usually deployed on large-scale clusters of commodity servers, with 24/7 running requirement. Failures are ubiquitous, because failure probability increases with the growing scale of cluster and running time [8]. DSPSs should recover rapidly and accurately enough to minimize performance degradation. A bunch of fault tolerance approaches have been proposed. We group them into three categories: (1) *active* scheme replicates data on multiple nodes as well as the processing; (2) *passive* scheme creates a global snapshot of system; (3) *hybrid* scheme combines the strong points of these two schemes. However, there is no systematic work to quantitatively evaluate the quality of a fault tolerance mechanism, which is necessary to consolidate the state-of-the-art research and to guide the development of strong fault-tolerant DSPSs. We address such a gap in this paper, and make the following contributions:

– We first design an evaluation framework dedicated to compare extra runtime overhead and recovery efficiency of fault tolerance of DSPSs.
– We define measurements of general and fault-tolerance-specific metrics.
– We exploit the design principles of fault tolerance mechanisms in classic DSPSs. In order to study the extra cost during runtime, we dissect workloads

employed in previous work, abstract the common operations and construct five fault-tolerance-sensitive workload suites with a set of configurable knobs.
– We present performance analysis of Flink and Storm through extensive experiments. We conclude with interesting observations and future work.

The rest of this paper is organized as follows: Sect. 2 introduces the basic concepts of stream processing and classifies fault tolerance approaches. The design of evaluation framework is described in Sect. 3. Section 4 presents the experimental results and fault tolerance routines of Flink and Storm. Section 5 discusses related work. We conclude the paper in Sect. 6.

2 Background

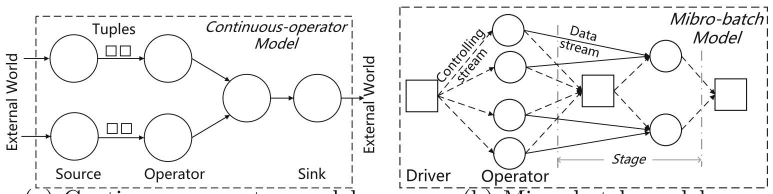
2.1 Stream Processing Model

SPSs can be classified into two categories based on processing model: (1) native *continuous-operator* model, such as Flink and Storm; (2) *micro-batch* model, such as Spark Streaming. As Fig. 1(a) shows, data stream is separated with data processing in continuous-operator model. After job submission, user programs are converted into a directed acyclic graph (DAG, a.k.a, topology) of operators connected by data streams. Each operator encapsulates the real computation logic, and consumes(produces) tuples from input(into output) queues. Operators are usually parallelized into multiple instances and run on different nodes for high throughput. On the contrary, as Fig. 1(b) shows, data stream is bundled with data processing in micro-batch model. In this example, there are two operators in the job[1]. Input streams are divided into continuous series of discrete *t-second* micro-batches. User programs are converted into a directed acyclic graph of *stages*. Within each stage, all the parallel instances of an operator are scheduled by a centralized driver, consume the micro-batches and report the size of output results to the driver. Then the driver launches the next state, and sends data information that another operator should fetch. We mainly concern DSPSs with continuous-operator model. DSPSs with micro-batch model, especially Spark Streaming, are almost inherent fault-tolerant owing to the underlying resilient data structures and lineage information [22].

2.2 Fault Tolerance Overview

Fault tolerance in SPSs is realized via replication, either data processing or internal state. We can group them into *active* scheme [3,12,17], *passive* scheme [5,7,11,13,16] and *hybrid* scheme [10,15,19]. In active scheme, each primary node has $k \geq 1$ standby nodes doing the same job. Once a primary node becomes unavailable, one of the standby nodes takes over it instantly. The main challenge of active scheme lies in replica synchronization between primary and standby nodes. On the contrary, in passive scheme, only the primary node consumes the

[1] Circles in the same column are the parallel instances of an operator.



(a) Continuous-operator model (b) Micro-batch model

Fig. 1. Two common stream processing models.

input stream, but it creates a *checkpoint* of its internal state as well as other information necessary for recovery. Upon failure, operators on the primary node will be relaunched on ≥ 1 standby nodes and rebuild the state from checkpoints. Challenges of passive scheme mainly lie in four aspects: when to do checkpoint, how to coordinate all the operators to create a consistent global snapshot, what data to checkpoint, and where to store checkpoints. Recently, some work proposes hybrid scheme to combine active and passive scheme. But how to find an optimal assignment plan on each node is a key issue.

Table 2. Design goals of fault tolerance.

Runtime overhead	Normal latency	Hardware resource
		CPU, Memory, Network
Recovery efficiency	Recovery latency	Recovery accuracy
		Identical, Duplicated, Disordered, Lost, Incorrect

To measure the quality of fault tolerance, we define *runtime overhead* and *recovery efficiency*, as summarized in Table 2. Obviously, extra overhead is inevitable during failure-free runtime, such as more storage in active scheme, or higher normal latency incurred by checkpointing in passive scheme. Recovery efficiency can be evaluated from two aspects, i.e. recovery latency and degrees of recovery accuracy. These design goals, however have trade-offs between each other. For example, active scheme favors instant recovery at the cost of substantial hardware resource consumption, while passive scheme sacrifices higher recovery latency to save resources, or trades accuracy for lower normal latency by leveraging approximation techniques.

3 Evaluation Framework

Figure 2 shows the overview of our evaluation framework. In-memory *Data Generator* (*DG*) injects tuple into *Data Broker* (*DB*, e.g., Kafka[2].) and controls the input rate as well as skew distribution. DSPS under test is isolated from *DG* and

[2] http://kafka.apache.org.

DB. It executes workloads, fetches input streams from *DB* and exports results to *DB* as well. *Metrics Collector* is integrated into *DB* to collect metrics.

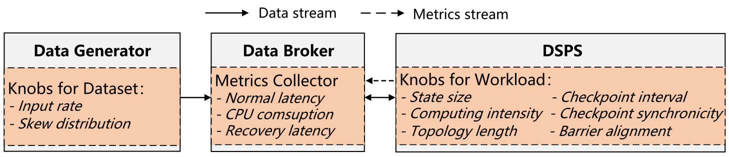


Fig. 2. The evaluation framework.

3.1 Evaluation Metric

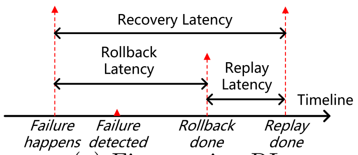
Normal Latency (NL). NL is the average time difference (L_t) between each tuple t generated at DG (T_t^{DG}) and all its causally dependent tuples generated at the sink operators (T_t^{Sink}), shown in Eq. 1. T_t^{DG} is also known as *event time* [5], and thus NL is also called as *event-time latency*.

$$L_t = T_t^{DG} - T_t^{Sink}; \qquad NL = \overline{\sum L_t} \tag{1}$$

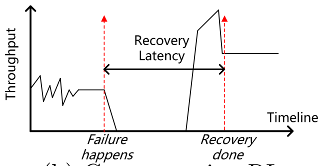
Resource Consumption (CPU). Extra resource overhead often results from (1) scheduling checkpointing procedure and preparing state, which consume CPU cycles; (2) maintaining operator state, which consumes memory; (3) transmitting checkpoints to stable storage, which consumes network bandwidth. During the experiments we observe that the consumption of memory and network bandwidth is proportional to the size of checkpoints. Hence we measure CPU consumption. We monitor CPU every second by `Ganglia`[3], and present both average value and variation trend over time.

Recovery Latency (RL). We define RL from two perspectives: (1) A fine-version RL is defined as the duration from the moment when a failure happens to the moment when DSPS finishes replaying the last tuple. The recovery procedure is composed of rollback phase and replay phase. These two phases execute in sequence and in parallel inside Flink and Storm, respectively. Therefore, we define RL of Flink as the sum of rollback latency and replay latency as shown in Fig. 3(a); we define RL of Storm as the higher one between rollback latency and replay latency. (2) A coarse-version RL can be evaluated indirectly by the evolution of throughput as shown in Fig. 3(b). Once failure happens, throughput fluctuates sharply compared to that during failure-free runtime. Hence, we can measure RL roughly by the duration from the moment when a failure happens to that when throughput comes to be steady.

[3] http://ganglia.sourceforge.net/.



(a) Fine-version RL



(b) Coarse-version RL

Fig. 3. Two versions of recovery latency.

3.2 Data Generation

We use two kinds of datasets: (1) a collection of real-world English novels crawled from Project Gutenberg[4]; (2) a synthetic dataset with different skew distributions of words. We present knobs for datasets in Table 3. To ensure stable running of each system and avoid backpressure, we keep the data generation at a steady rate 5000 tuples/s. But we lower the input rate down to 500 tuples/s in the last set of experiments on Storm due to its performance bottleneck. Whether barriers are aligned has significant influences on performance, especially when input streams are highly-skewed. Hence, we generate synthetic dataset with uniform (skew = 0) and highly-skewed (skew = 1) distributions.

Table 3. Configurable knobs for data generation.

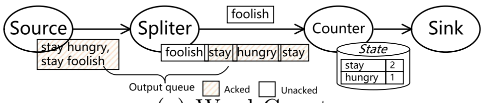
Knob	Abbr	Unit	Value
Input rate	*IR*	tuples/s	500, **5000**
Skew distribution	*Skew*	-	**0**,1

3.3 Workloads Design

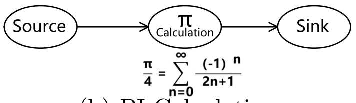
Workloads. Covering all the streaming workloads is impractical. Hence, we choose and generate three typical workloads as depicted in Fig. 4. (1) In most realistic scenarios, aggregation is one of the basic operations. The widely-used task is streaming *Word Count (WC)* that counts the number of occurrences of each word in continuously arriving sentences. As Fig. 4(a) shows, it contains both stateless (i.e., Spliter) and stateful operators (i.e., Counter). (2) Numerical computation is also common in stream processing, and its amount of computation varies in different stream applications, such as CPU-bound model training in machine learning algorithms. Hence, we create a synthetic *PI Calculation (PI)* topology as shown in Fig. 4(b). Within it, we utilize Gregory-Leibniz series formula for π which can control the amount of computation via n. (3) We generate *Dummy Topologies (DT)* with different numbers of operators to simulate the

[4] https://www.gutenberg.org/.

complexity of topology. Each operator in dummy topologies has no real computing task, but receives and sends tuples.



(a) Word Count



(b) PI Calculation

Fig. 4. Three typical workloads.

Workload Suites. After investigating all the fault tolerance approaches of passive scheme, we summarize 6 knobs in Table 4 to compose different workloads suites, by configuring application characteristics and checkpoint requirements. *State* denotes the size in byte of states including in a checkpoint. During the experiments, we find that if we enable the window mode of operators, checkpointing can happen at any moment within a time window. As a result, *State* of each checkpoint is of random distribution, which baffles the evaluation of its impacts on performance. Therefore, we artificially simulate *State* of each operator by maintaining a string variable and conduct the experiments under full-history mode. *Computing intensity* controls the computation amount of operators and is configured as various execution rounds of PI calculation. *Topology length* is the number of operators in a dummy topology. *Checkpoint interval* defines the time interval between two consecutive checkpointing requests. *Checkpoint synchronicity* defines whether checkpointing executes synchronously with normal processing. *Barrier alignment* defines whether barriers are aligned.

The knobs, along with workloads mentioned above, together compose 5 workload suites as listed in Table 5, each of which focuses on distinct effects of knobs on fault tolerance performance. **Suite 1–3** evaluate extra overhead of each DSPS incurred by fault tolerance during failure-free runtime. **Suite 4–5** evaluate the recovery latency of each DSPS after randomly killing worker processes.

Table 4. Configurable knobs for workloads: default value in bold.

Object	Knobs	Abbr	Unit	Values
Application	State size	*State*	MB	1, **10**, 15, 30
	Computing intensity	*CPI*	round	0(low), **2000**(medium), 5000(high)
	Topology length	*TL*	# of operators	**2**, 10
Checkpoint	Checkpoint interval	*CI*	second	NC (disabled), 1 **30**, 45, 60
	Checkpoint Synchronicity	*CS*	-	sync, **async**
	Barrier alignment	*BA*	-	**true**, false

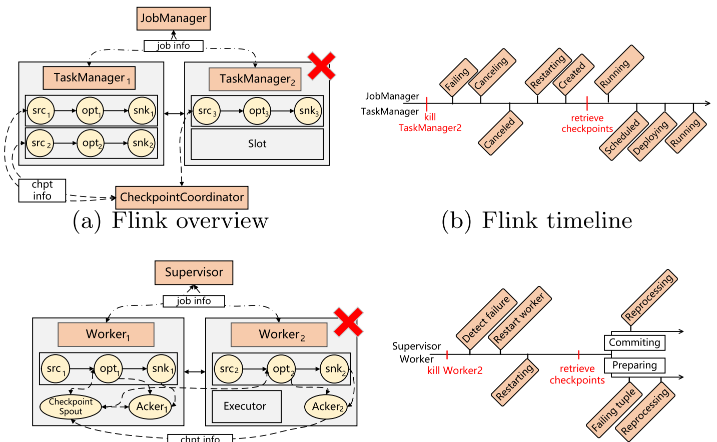
Table 5. Workload suites with distinct purposes.

Workload Suite	Name	Description	Cost
1	*WC*	Effects of *CI* on *NL* and *CPU* for applications with different *State*	Overhead
2	*PI*	Effects of *CS* on *NL* and *CPU* for applications with different *CPI*	
3	*WC*	Effects of *BA* on *NL* and *CPU* when processing streams with different *Skew*	
4	*DT*	Effects of *TL* on *RL*	Recovery efficiency
5	*WC*	Effects of *CI* on *RL*	

4 Experiments

We implement the evaluation framework to compare the fault tolerance performance of Flink and Storm. We perform our experiments on a 5-node cluster where 1 node is equipped with 24 Intel Xeon E5-2620 CPUs and 31 GB of RAM, and the others with 8 Intel Xeon E5606 CPUs and 94 GB of RAM. The nodes are inter-connected via Gigabit Ethernet and run a CentOS Linux operating system with kernel version 6.5.0. We conduct five sets of experiments to understand the design principles of fault tolerance for each system. We roughly simulate failures by killing multiple worker processes randomly.

Flink. As shown in Fig. 5(a), three operators *src*, *opt* and *snk* together compose a topology, and are partitioned into different slots. *JobManager* is the controller process, and *TaskManager* is the worker process. *src* receives periodic checkpointing requests from *CheckpointCoordinator*, and injects barriers into data streams. *opt* with multiple input streams doesn't make checkpoints until it receives barriers with the same ID from all its input streams. When one barrier arrives, *opt* suspends the corresponding input stream and buffers the follow-up tuples into the input queue. Such a step is called *barrier alignment*. It can be disabled, but precise recovery will not be guaranteed. Once all the barriers with the same version ID arrive, *opt* can either block the normal processing to make checkpoints synchronously, or apply the "copy-on-write" technique to make checkpoints asynchronously. When a failure occurs to *TaskManager* (i.e., all the operators it manages fail as well), as shown in Fig. 5(b), operators that communicate with the failed ones receive no responses, and then inform *JobManager*. *JobManager* marks the job as `Failing` and broadcasts a *Canceling* message to each alive *TaskMangager*. Once received, *TaskManager* forcibly terminates all the operators it manages and sends back a `Canceled` message. *JobManager* will never receive a `Canceled` message from the failed *TaskManager*, and then marks the job as *Restarting* when the threshold of waiting time is reached. The whole



(a) Flink overview (b) Flink timeline

(c) Storm overview (d) Storm timeline

Fig. 5. Fault tolerance routines of Flink and Storm.

topology will be reset and restore `Running` status. Accordingly, all the operators redeployed retrieve their own last checkpoints to rollback the state. Once *src* finishes the rollback, it begins to fetch the tuples that were processed but not reflected into the latest checkpoint.

Storm. We use the same topology to discuss Storm, which is inspired by Flink but has some differences. As Fig. 5(c) shows, *Supervisor* is the controller process, *Worker* is the worker process and *Acker* is responsible for message processing guarantee. *CheckpointSpout* is one kind of *src* but only injects periodic checkpointing messages (i.e., the role of barrier in Flink). When fault tolerance is enabled, Storm will acknowledge all tuples that arrive during a checkpoint interval. We call it *batch-acking*. Storm completes a checkpointing procedure in two phases. In *preparing* phase, each operator receives a preparing message, writes a temporary checkpoint meta data into Redis[5] and then acknowledges *Acker*. Once receiving all the acknowledgments from each operator, the *committing* phase is triggered, so *CheckpointSpout* sends a committing message to inform each operator to write down real checkpoints to Redis synchronously. After that, each operator deletes the previous checkpoint, and performs batch-acking. When a failure occurs to *Worker*, as shown in Fig. 5(d), *Supervisor* always monitors the heartbeats of *Worker*. Unlike Flink, only operators that were managed by the failed *Worker* will be reset. If failure happens in preparing phase, the redeployed operators will rollback to the last checkpoint and actively fail the tuples that were processed from that checkpoint to the failure time; if failure happens in committing phase, since the up-to-date state of Storm is already stored in Redis, the redeployed operators directly continue the normal processing after fetching the corresponding checkpoints. Note, *src* will receive two kinds of `Failing` messages

[5] https://redis.io//.

if failure happens in the preparing phase, namely the actively-failed tuples and time-out tuples which are not acknowledged in time by *Acker* due to failure. Tuple replaying and state rollback in Storm execute in parallel, thus it can not provide precise recovery.

4.1 Experimental Results

Workload Suite 1: Since Storm can't function properly with highly frequent checkpointing and large state, we don't run it with 1s-*CI* and 30 MB-*State*. As shown in Fig. 6, for Flink, more frequent checkpointing means more interference on normal processing and more thread scheduling. When checkpointing executes synchronously with normal processing, larger *State* incurs longer pause of normal processing to make and transmit checkpoints. Hence, both *NL* and *CPU* are proportional to *State*, but inversely proportional to *CI*. For Storm, under a certain *CI*, *NL* and *CPU* have a linear relational with *State*. We can conclude that: (1) larger *CI* and *State* usually incur higher *NL* and *CPU*; (2) even the barrier of Flink is aligned which incurs more waiting time, performance of Storm has greater degradation when fault tolerance is enabled, compared with Flink.

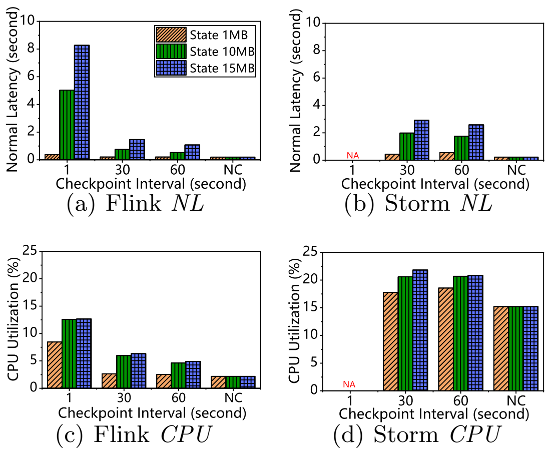
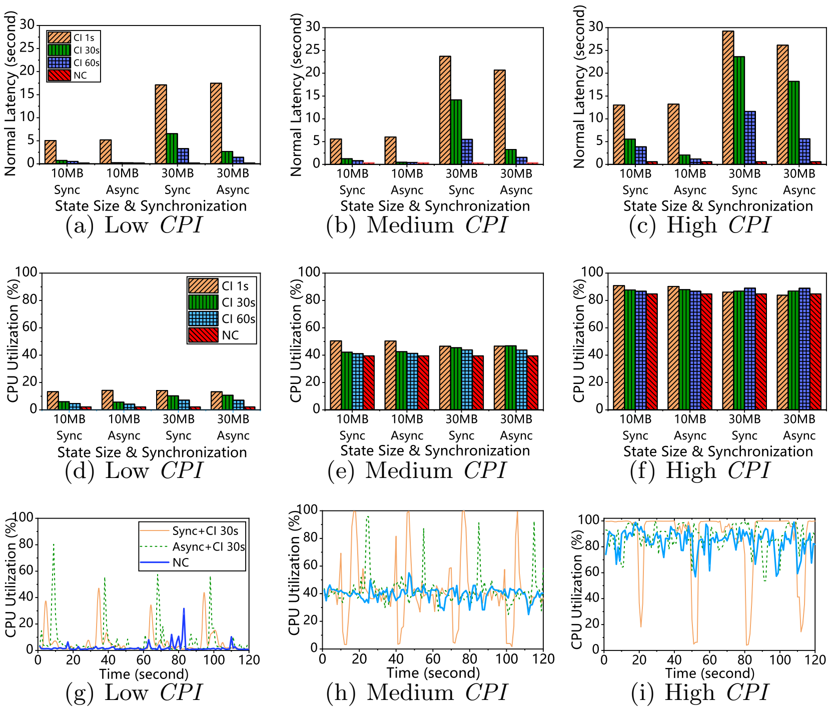


Fig. 6. The effects of *State* and *CI* on *NL* and *CPU*.

Workload Suite 2: Since checkpointing by default executes synchronously with normal processing, and that checkpointing messages are not aligned in Storm, we only evaluate Flink here. As shown in Fig. 7(a)–7(c), if checkpointing executes asynchronously, new incoming tuples will be processed in time; otherwise, they will be buffered in input queues of each operator until checkpointing is finished. Hence, asynchronous checkpointing can optimize the increase of *NL*.



(a) Low *CPI* (b) Medium *CPI* (c) High *CPI*

(d) Low *CPI* (e) Medium *CPI* (f) High *CPI*

(g) Low *CPI* (h) Medium *CPI* (i) High *CPI*

Fig. 7. The effects of *CS* and *CPI* on *NL* and *CPU*.

Moreover, we evaluate *CPU* under different *CPI*. As the total computation amount is constant, the average *CPU* of asynchronous checkpointing is approximate to that of synchronous checkpointing, as shown in Fig. 7(d)–7(f). For further analysis, we evaluate *CPU* trend over time. As illustrated in Fig. 7(g)–7(i), with the increase of *CPI*, *CPU* of synchronous checkpointing changes more violently. Once checkpointing is trigger, each operator suspends its normal processing, which leads to an obvious trough of *CPU*. When checkpointing is finished, the buffered new incoming tuples are first processed, which leads to a peak. In Fig. 7(i), there is no peak but a longer continuously-high period in that *CPU* is already extremely high due to heavy *CPI*. We conclude that asynchronous checkpointing gains lower runtime overhead, but this predominance is weakened under heavy computation amount and highly-frequent checkpointing.

Workload Suite 3: As shown in Fig. 8, with uniform distribution, effects on *NL* have little difference whether barriers are aligned. However, when input streams are highly-skewed, arrival time of barriers on different streams differs sharply with each other, which leads to longer blocking of the stream on which the barrier arrives first, and accordingly higher *NL*. Figure 9 demonstrates *CPU* trend over time under different *Skew*. We also compare high-load nodes with low-load nodes. As Fig. 9(a) and 9(b) demonstrate, when *Skew* is low, there is little difference on CPU whether barriers are aligned or not. But with severe skewness, temporary blocking of some stream causes massive tuples buffered, which leads to an obvious period of peak usage of *CPU*, as shown in Fig. 9(c) and 9(d). We conclude that aligning barriers incurs higher normal latency and unstable CPU status when input streams are of highly-skewed distribution.

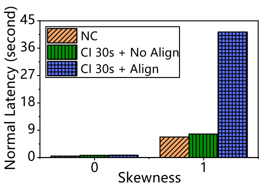


Fig. 8. The effects of *Skew* and *BA* on *NL* in Flink.

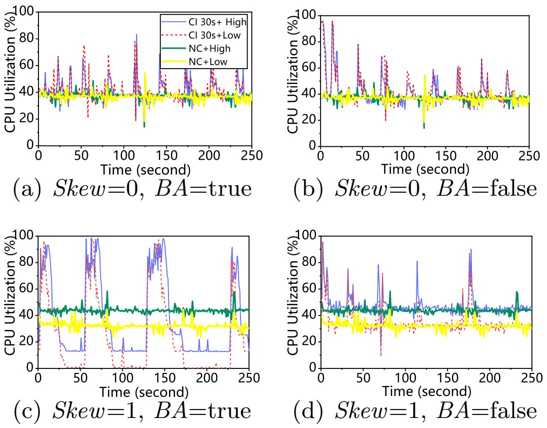


Fig. 9. The effects of *Skew* and *BA* on *NL* and *CPU* in Flink.

Workload Suite 4: *JobManager* decides the failure of a failed *TaskManager*, if it receives no responses from *TaskManager* when the threshold of waiting time is reached. As shown in Fig. 10, longer topology lightly increases the rollback latency, but is bound to the timeout threshold of failure detection. We omit the evaluation of Storm for this workload suite in that we observe that the recovery latency is dominated by the replay latency.

Workload Suite 5: Input rate for Storm is adjusted to 500 tuples/s because its mediocre performance. Figure 11 presents the fine-version recovery latency of Flink and Storm. We observe that the rollback latency of Flink is always bound to 45 s, having nothing to do with *State* and *CI*. After reviewing system logs, we find that the default time-out threshold of failure detection is 45 s. However, as Fig. 11(b) and 11(d) show, the replay latency of Flink has a linear relationship with *CI*, but is not affected by *State*; while *State* has greater influence on the volume of replayed tuples than *CI*. We analyze logs and observe that the replay phase begins before all the operators have finished the rollback phase. Once the source operator has finished its rollback, it begins to fetch tuples from *DB*. The bigger *State* implies more time downstream operators spend on rollback. New incoming tuples are buffered before downstream operators have finished the rollback, which may cause backpressure and consequently lead to higher replay latency. For Storm, *RL* is proportional to *CI* as shown in Fig. 11(d), but almost not affected by *State*. We figure out that *RL* is mainly affected by batch-acking and tuple replaying. When *CI* is relatively large, the impact of batch-acking outweighs that of tuple replaying. Hence, when failure happens in preparing phase, *RL* is higher than that at committing phase. While for the smaller *CI*, we draw an opposite conclusion.

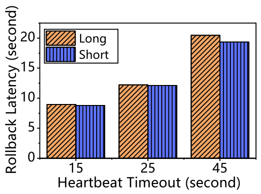


Fig. 10. The effects of *TL* on rollback latency in Flink.

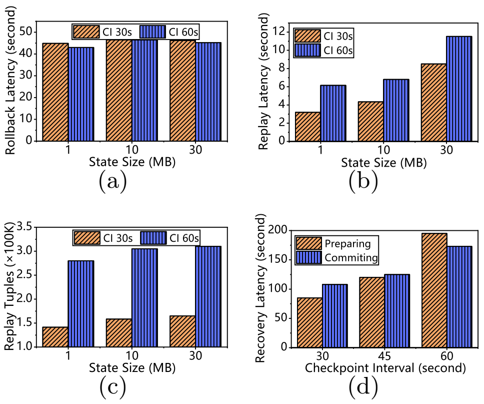


Fig. 11. The effects of *State* and *CI* on fine-version *RL*.

Figure 12 shows the coarse-version recovery latency of Flink and Storm. The first dotted arrow marks the moment when failure occurs, and the second one marks the moment when the system restores the steady throughput. For Flink, after failure is detected, source operators of Flink stop fetching tuples from data broker, which results in a temporary zero throughput. After source operators finish the rollback phase, tuples that were emitted from the latest checkpoint to the failure point are re-fetched, leading to wave crests. For Storm, source operators keep emitting tuples downstream even when failure happens. The wave crests imply the replaying of tuples which are actively failed by downstream operators. We conclude that (1) Flink recovers much more efficiently than Storm.

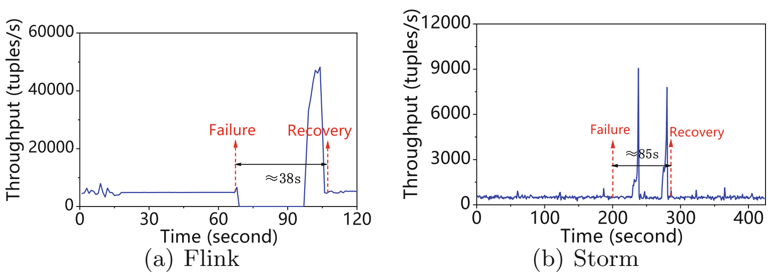


Fig. 12. The coarse-version *RL* of Flink and Storm.

Even all the operators are forced to be redeployed in Flink, Flink gains far less recovery latency than Storm; (2) Checkpoint interval has great effects on recovery latency, as larger interval indicates more tuples to be replayed.

Discussion: Fault tolerance indeed leads to high performance degradation of system. A moderate checkpoint frequency (e.g., 30 s) can incur x4 - x12 severe latency in Flink and Storm. Asynchronous checkpointing technique can mitigate it, but its effect is diminished when the workload is computation-intensive. But if high burst of streams happens to those platforms, checkpointing will occupy the runtime processing all the time. Hence, it is essential to design an adaptive checkpointing technique to adjust checkpoint interval according to real-time workloads. Moreover, the recovery procedure can be dominated by tuple relaying. Even the whole system is reset, retrieving the checkpoints and rebuilding the state are not time-consuming. Until now, the research focuses more on the content to checkpoint instead of the fatal problem in source replaying, and then parallel recovery is necessary in future.

5 Related Work

Linear Road Benchmark [2] is the first benchmark dedicated to DSMSs. It simulates a highway toll system. Lu et al. raise StreamBench [14] to evaluate Storm and Spark Streaming. Apart from performance measurements, StreamBench also evaluates the distributed characteristics of system, such as fault tolerance ability and durability. StreamBench gathers two seed data sets from web log processing and network traffic monitoring, and leverages a message system to inject data into DSPSs on the fly. Based on data type, computation complexity and operator state, StreamBench selects 7 programs and defines 4 synthetic workload suites. However, for the evaluation of fault tolerance ability, StreamBench merely compares the throughput and latency after failures with those before failures. Yahoo! Storm team [6] proposes a open-source streaming benchmark (YSB) which simulates an advertisement analytics pipeline. It uses Kafka and Redis for data fetching and storage respectively. Three DSPSs, namely Flink, Storm and Spark Streaming, are compared in terms of 99th percentile latency and throughput, with checkpointing off by default. Later, Grier [9] extends YSB to measure the latency and maximum throughput of Flink and Storm. Wang [21] also demonstrates a stream benchmark tool called StreamBench to evaluate the event-time latency and throughput of Storm, Flink and Spark Streaming. To distinguish from [14], we rename it as StreamBench'. StreamBench' defines three kinds of workloads to evaluate three representative operations in stream processing, namely aggregation, joining and iterating. Similarly, it utilizes a message system to simulate the on-the-fly data generation, and control the skew distribution.

6 Conclusion

We design an evaluation framework to empirically investigate the fault tolerance performance of DSPSs, which is an urgent requirement from cloud-based/cluster-based applications as the persistent increasing of stream data and strict service quality demands. Experimental results reveal a moderate checkpointing frequency can incur x4 - x12 performance degradation in Flink and Storm. And even Flink forces all the operators to rollback during recovery, it gains x2 recovery efficiency than Storm. These performance gap can be explained by the fault tolerance routine we tease apart. And we conclude that parallel sources recovery and adaptive checkpointing are necessary for improving system performance. Though we have some interesting observations in this work, there are still some points to be covered in the future. First, our work is merely appropriate for DSPSs with continuous-operator model and takes no account of fault tolerance approaches with active and hybrid scheme. The challenge is that SPSs adopting these two fault tolerance schemes either drop behind or with no open-sourced prototypes. Second, all the evaluations in this paper are conducted under stable conditions when DSPSs ingest stable inputs. However, streaming data in real-world is often highly dynamic, and its impact on fault tolerance performance need to be investigated as well. Third, we will involve more complex computation, such as machine learning algorithms, to the framework. Last, a universal measurement of recovery accuracy has to be settled since more stream applications emphasize precise results rather than traditional approximate processing.

Acknowledgement. The work is supported by National Key Research and Development Plan Project (No.2018YFB1003402) and National Science Foundation of China (NSFC) (No.61672233,61802273). Ke Shu is supported by PingCAP.

References

1. Abadi, D.J., Carney, D., Zdonik, S.B., et al.: Aurora: a new model and architecture for data stream management. VLDBJ **12**(2), 120–139 (2003)
2. Arasu, A., Cherniack, M., Tibbetts, R., et al.: Linear road: a stream data management benchmark. In: Proceedings of the 30th VLDB International Conference on Very Large Data Bases, pp. 480–491 (2004)
3. Balazinska, M., Balakrishnan, H., Madden, S., Stonebraker, M.: Fault-tolerance in the borealis distributed stream processing system. In: Proceedings of the 2005 ACM SIGMOD International Conference on Management of Data, pp. 13–24. ACM (2005)
4. Bordin, M.: A Benchmark Suite for Distributed Stream Processing Systems. Ph.D. thesis, Universidade Federal do Rio Grande Do Su (2017)
5. Carbone, P., Katsifodimos, A., Tzoumas, K., et al.: Apache flink^{TM}: stream and batch processing in a single engine. IEEE Data Eng. Bull. **38**(4), 28–38 (2015)
6. Chintapalli, S., Dagit, D., Poulosky, P., et al.: Benchmarking streaming computation engines: storm, flink and spark streaming. In: Proceedings of the 2016 IEEE International Parallel and Distributed Processing Symposium Workshops, pp. 1789–1792 (2016)
7. Fernandez, R.C., Migliavacca, M., Kalyvianaki, E., Pietzuch, P.R.: Integrating scale out and fault tolerance in stream processing using operator state management. In: Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data, pp. 725–736. ACM (2013)

8. Gill, P., Jain, N., Nagappan, N.: Understanding network failures in data centers: measurement, analysis, and implications. In: Proceedings of the 2011 ACM SIGCOMM Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications, pp. 350–361. ACM (2011)
9. Grier, J.: Extending the yahoo! streaming benchmark (2016). https://www.ververica.com/blog/extending-the-yahoo-streaming-benchmark
10. Heinze, T., Zia, M., Fetzer, C., et al.: An adaptive replication scheme for elastic data stream processing systems. In: Proceedings of the 9th ACM DEBS International Conference on Distributed Event-Based Systems, pp. 150–161. ACM (2015)
11. Huang, Q., Lee, P.P.C.: Toward high-performance distributed stream processing via approximate fault tolerance. PVLDB **10**(3), 73–84 (2016)
12. Hwang, J., Çetintemel, U., Zdonik, S.B.: Fast and highly-available stream processing over wide area networks. In: Proceedings of the 24th IEEE ICDE International Conference on Data Engineering, pp. 804–813. IEEE (2008)
13. Kwon, Y., Balazinska, M., Greenberg, A.G.: Fault-tolerant stream processing using a distributed, replicated file system. PVLDB **1**(1), 574–585 (2008)
14. Lu, R., Wu, G., Xie, B., Hu, J.: Streambench: towards benchmarking modern distributed stream computing frameworks. In: Proceedings of the 7th IEEE/ACM International Conference on Utility and Cloud Computing, pp. 69–78. IEEE/ACM (2014)
15. Martin, A., Smaneoto, T., Fetzer, C., et al.: User-constraint and self-adaptive fault tolerance for event stream processing systems. In: Proceedings of the 45th Annual IEEE/IFIP DSN International Conference on Dependable Systems and Networks, pp. 462–473. IEEE/IFIP (2015)
16. Sebepou, Z., Magoutis, K.: CEC: continuous eventual checkpointing for data stream processing operators. In: Proceedings of the 2011 IEEE/IFIP DSN International Conference on Dependable Systems and Networks, pp. 145–156. IEEE/IFIP (2011)
17. Shah, M.A., Hellerstein, J.M., Brewer, E.A.: Highly-available, fault-tolerant, parallel dataflows. In: Proceedings of the 2004 ACM SIGMOD International Conference on Management of Data, pp. 827–838. ACM (2004)
18. Shukla, A., Chaturvedi, S., Simmhan, Y.L.: RIoTbench: an IoT benchmark for distributed stream processing systems. CCPE **29**(21), e4257 (2017)
19. Su, L., Zhou, Y.: Passive and partially active fault tolerance for massively parallel stream processing engines. TKDE **31**(1), 32–45 (2019)
20. Toshniwal, A., Taneja, S., Ryaboy, D.V., et al.: Storm@Twitter. In: Proceedings of the 2014 ACM SIGMOD International Conference on Management of Data, pp. 147–156. ACM (2014)
21. Wang, Y.: Stream Processing Systems Benchmark: StreamBench. Master's thesis, Aalto University (2016)
22. Zaharia, M., Das, T., Stoica, I., et al.: Discretized streams: fault-tolerant streaming computation at scale. In: Proceedings of the 24th ACM SIGOPS Symposium on Operating Systems Principles, pp. 423–438. ACM (2013)