Cluster Computing
https://doi.org/10.1007/s10586-019-02939-9

Reliable stream data processing for elastic distributed stream processing systems

Xiaohui Wei[1,2] · Yuan Zhuang[1] · Hongliang Li[1,2] · Zhiliang Liu[1]

Received: 13 October 2017 / Revised: 4 May 2019 / Accepted: 4 May 2019

Abstract
Distributed stream processing system (DSPS) has proven to be an effective way to process and analyze large-scale data streams in real-time fashions. The reliability problem of DSPS is becoming a popular topic in recent years. Novel elastic DSPSs provide the ability to seamlessly adapt to stream workload changes, which introduce new reliability challenges: (1) operators can be scaled up and down at runtime, requiring fault tolerant methods to maintain data backup consistency under the runtime dynamics. (2) Rollback recovery to the last checkpoint may undo recent auto-scaling adjustments, which will introduce high cost and unacceptable impact to the system. In this paper, we put forward a novel fault-tolerant mechanism to deal with these issues. In particular, we propose a self-adaptive backup unit, elastic data slice (EDS), that can partition and merge data backups according to operator auto-scaling at runtime. The consistency of recovery is guaranteed by new upstream backup protocols, which restart the system from the status after auto-scaling instead of last checkpoint and avoid high recovery latency. Based on them, we implement a prototype system named SPATE. Evaluations on SPATE show that our mechanism supports auto-scaling changes with similar overhead compared to existing approaches, while achieving low recovery latency despite auto-scaling.

Keywords Distributed stream processing system · Fault tolerance · Upstream backup

1 Introduction

In recent years, distributed stream processing systems (DSPSs) [5, 7] have been proven to be an effective way to solve large-scale data stream problems [15, 20, 23, 32, 33]. These systems are able to organize shared distributed computing resources and process multiple data streams in real-time. The failure of a single node can significantly disrupt or even halt overall stream processing, leading to incomplete datasets and inaccurate results that eventually jeopardize the integrity of data-sensitive applications (e.g., health-care monitoring, financial analysis and alarm systems). Therefore, it is crucial for DSPSs to achieve reliable stream processing.

There are two major fault-tolerant mechanisms applied in state-of-the-art DSPSs: replication [27] and upstream backup [18, 22]. It is relatively easier for active replication to achieve failure recovery, but at a prohibitively high cost in extra required computational resources. In contrast, upstream backup ensures fault tolerance with low overhead. Especially, when combined with checkpointing, upstream backup avoids the infeasible recovery latency of operators whose state depends on all past tuples. Thus, upstream backup has been widely applied in DSPSs.

As streaming applications are characterized by highly variable arrival rates, mechanisms for auto-scaling in stream processing have received considerable attention. However, runtime dynamics of auto-scaling increase the difficulties of fault-tolerance, new challenges of upstream backup have emerged as follows:

✉ Hongliang Li
lihongliang@jlu.edu.cn

[1] College of Computer Science and Technology, Jilin University, Changchun, China

[2] Key Laboratory of Symbolic Computation and Knowledge Engineering of Ministry of Education, Changchun, China

Published online: 21 May 2019

1.1 Consistency guarantee under the auto-scaling

Elastic DSPSs change stream topology by altering operator parallelism on-demand, which makes it hard for fault-tolerance to achieve the consistency of recovery. Generally, operators in upstream backup retain their output events as backups for downstream neighbors, and correlate with each other by their data dependencies. Owing to the topology changes, the backups and their data dependencies between operators are not consistent before and after auto-scaling. Therefore, the backups should be able to re-partition according to operator scaling. Furthermore, online adjustment of their data dependencies between operators is also needed.

1.2 Fast recovery with low overhead

For stateful operators, upstream backup usually works with checkpointing to achieve their recovery guarantee. Upon failure, the state of a failed task can be restored from its latest checkpoint. However, this rollback recovery undoes any recent auto-scaling adjustment, after which the restarted operator may possibly experience abrupt workload or unpredicted workload skews. In particular, this can again incur high scaling overhead, which can dampen the system performance seriously. Thereby, it requires the capability of online checkpoint updates to capture operator auto-scaling changes to ensure fast recovery.

The challenges listed above call for a fault-tolerance mechanism capable of dynamic adjustments to support elastic DSPSs. Unfortunately, existing upstream backup mechanisms [18, 21, 34, 38] are mostly static, and cannot adapt to dynamic scaling changes. Therefore, how to ensure fault tolerance in elastic stream processing is still an open question. Recent works have started to focus on dynamic characteristic of elastic DSPSs, but they either simplify the problem by scaling an operator from its most recent checkpoint (MRC) [4] or just inherit existing fault tolerance [13] without considering the topology changes. While these adopted strategies achieve a certain level of reliability for elastic DSPSs, they fail to enable transparent failure recovery without putting considerable pressure on overall performance.

In this paper, we propose a novel upstream backup mechanism to solve these reliability problems in elastic DSPSs.

The main contributions of this paper are as follows:

1. We introduce an adaptive fault-tolerant problem for elastic distributed stream processing systems.
2. We propose a novel adaptive upstream backup data unit, elastic data slice (EDS), which can be partitioned and merged according to topology changes.
3. We present an asynchronous incremental checkpoint mechanism to trace operator auto-scaling changes at runtime.
4. We propose a set of fault-tolerant protocols that guarantees the status consistency under auto-scalings and supports fast recovery from failures.
5. We conduct extensive evaluation data based on a prototype system.

The rest of the paper is organized as follows: In Sect. 2, we discuss related works; in Sect. 3, we define the fault-tolerant problems. In Sect. 4, we propose the fault-tolerant mechanism, and in Sect. 5 we present protocols and algorithms. In Sect. 6, we discuss the related implementation of our proposed mechanism, designated SPATE, and in Sect. 7 we present experimental results. We conclude the paper and discuss planned future work in Sect. 8.

2 Related work

2.1 Elastic distributed stream processing systems

To handle the continuous changing data streams, elastic distributed stream processing systems [8] (EDSPS) support auto-scaling technology [9, 19, 26, 29, 36] to seamlessly accommodate changes in workload online.

Earlier proposals of data stream processing systems, such as Aurora [1] and Borealis [2], cannot support elastic operator execution. Among the existing systems are Apache S4 [30] and Twitter Storm [28], two typical works that support flexible job topologies. S4 supports operator scaling according to workload. Storm allows users to specify parallelism level and supports stream partitioning based on key. However, the latter two works both have limited failure-recovery guarantees and do not provide state management from system level. Meanwhile, some proposed frameworks [3, 14, 38] have chosen to treat stream processing as a continuous series of MapReduce-style mini-batch processing jobs, such as Spark Streaming [30]. The advantage of this method is the ability to inherit MapReduce-based applications and the possibility of static data and stream data combined processing. The shortcoming is that these solutions lack dynamic parallelism at runtime.

Recently, much interest has been shifted to elasticity of stateful operators at runtime. Existing researches have proposed elasticity policies so far in literature. For instance, [26] uses a dynamically configured threshold to

improve the system adaptivity. Other works use more complex centralized policies to determine the scaling decisions, such as control theory [9], rare pattern mining [24], and fuzzy logic [29]. The proactive elastic strategies also emerged: PEC [36] introduces a proactive elastic resource scheduling, which aims at meeting the latency requirement with the minimal energy cost. These general EDSPSs feature dynamic resource scheduling and operator state migration, which make it hard to maintain recovery consistency under the runtime dynamics.

2.2 Fault-tolerant mechanisms in EDSPS

There are two different classes of operators in DSPS: stateless operators (e.g., map, fitter), characterized by each input data item that produces an output independently of any previous input tuple, and stateful operators (e.g., aggregate) that require keeping an internal state as a representation of the history data and results. Thus, for failure recovery, the former depends solely on present stream data, while the latter depends on all past data.

Generally, stateless operators can be preserved by data replications stored in either precursor nodes (upstream backup) [18, 34] or peer nodes [2, 27]. In comparison to peer replications, upstream backup is more efficient because it reuses existing communication connections rather than establishing new nodes.

In upstream backup, the upstream node stores past data for the downstream node as a backup, which will be replayed during recovery. Existing works focus on how to reduce the effect of periodical synchronization and message replay. Chen et al. [6] proposes a backtrack mechanism for failure recovery that allows continuous processing without waiting for ACKs. Wang et al. [34] allows recovery from multiple upstream nodes to accelerate data replay.

In order to support fault tolerance of stateless and stateful operators, upstream backup usually works with active/passive replication [27, 38, 39]. Zhang et al. [39] introduces a hybrid schema, in which the replica using upstream backup is switched to an active standby when a crash is detected. Martin et al. [27] uses CPU idle time to run active replications but under the constraint that the average CPU utilization should be under (50%). In D-Stream [38], a distributed memory abstraction, called resilient distributed dataset (RDD) is proposed. D-Stream selectively checkpoints RDDs to stable storage. Apache Flink depends on asynchronous distributed snapshots as consistent checkpoints, to which the system can fall back on in case of failure.

Most related works fail to pay enough attention to the dynamic characteristic of elastic stream processing. As auto-scaling introduces on-demand operator parallelism and dynamic topology between operators, online backup adjustment is necessary to maintain the recovery consistency with job status at runtime [4, 31, 37] have begun to explore this issue.

Elasticity is integrated with fault tolerance in Ref. [4]. It scales up the bottleneck operator from its most recent checkpoint (i.e., the point to which the system is forced back), causing a period of processing-time waste. ChronoStream [37] replicates the state of each UDF to a peer node, enabling elastic and fault-tolerant execution of stateful dataflows by this explicit state management. Obviously, this requires expensive I/O accesses to local storage that is persistent. While these elasticity implementations are bundled with the fault-tolerant feature, none of them offer online backup adjustment as in our work.

Unlike the above works, Timestream [31] introduces a lightweight data dependency tracking mechanism to record the mapping relationship between input and output data under auto-scaling. However, it fails to present a complete fault tolerant model. In addition, it backs up the entire processing chain to maintain the data dependencies, which introduces considerable fault-tolerant overheads, e.g., up to 14.82% network overhead. Therefore, a novel fault tolerant mechanism is urged to guarantee fast recovery and support online job topology adaptations with affordable overheads in EDSPSs.

3 Problem formulation

3.1 Overview

Here, we give an overview of the problems faced by upstream backup in EDSPS and showcase it in an operator auto-scaling scenario.

A stream-processing job can be modeled as a graph topology $G(V, A)$ in Fig. 1, which consists of a set of operators V and a set of directed arcs A that interconnect them. As shown in Fig. 2a, operator $v_i \in V$ may have multiple instances (i.e., task $t_{ij} \in v_i$), and the operator's

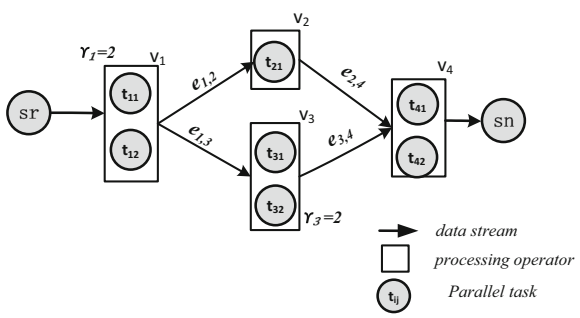
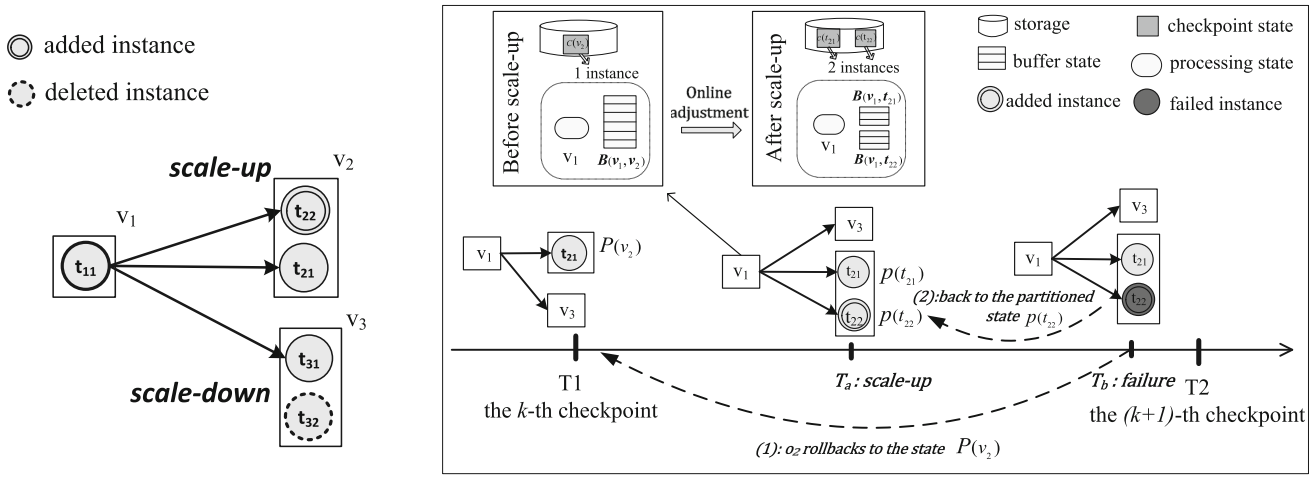


Fig. 1 Stream topology model



(a) Operator auto-scaling **(b)** Operator scale-up

Fig. 2 Example of operator auto-scaling

parallelism degree γ_i can be adjusted by scaling the instances up and down in response to workload changes, i.e., scale-up and scale-down. Table 1 elaborates the notations used in our model.

We assume that each operator backs up output for downstream neighbors and keeps periodical checkpointing in case of stateful operators. However, auto-scaling introduces runtime dynamics (e.g, operator topology changes) that greatly increase the difficulty of fault-tolerance. Considering the scale-up scenario depicted in Fig. 2b, we summarize new reliability problems as follows:

3.1.1 Backup consistency under dynamic topology

In upstream backup, tracking data dependencies between the input and output streams is crucial for efficient recomputation-based recovery. However, auto-scaling can change operator topology at runtime, breaking the mapping consistency between upstream backups and downstream neighbors. As shown in Fig. 2, when v_2 scales up at T_a, the backups buffered in v_1 should be repartitioned in order to keep the recovery consistency. Similarly, when v_3 scales-down at runtime, operator $v_1$also needs to merge its buffer state in response to the topology changes.

Table 1 Notations

$G(V, A)$	DAG that represents a stream topology
V	A set of operators in G $V = \{v_i \mid i \in \{1, .., n\}\}$
γ_i	The parallelism degree of operator $v_i \in V$
t_{ij}	A task of parallel operator $v_i \in V$ We say that $t_{ij} \in v_i$ when $j \in \{1, .., \gamma_i\}$
$P(v)$	Processing state of $v \in V$ $P(v) = \bigcup_{t \in v}\{p(t)\}$ where $p(t)$ represents the processing state of task t belonging to parallel operator v
$C(v)$	Checkpoint state of $v \in V$ $C(v) = \bigcup_{t \in v}\{c(t)\}$ where $c(t)$ represents the checkpoint state of task t belonging to parallel operator v
$B(v)$	Output buffer of $v \in V$ $B(v) = \bigcup_{t \in v}\{b(t)\}$, where $b(t)$ denotes task $t's$ output buffer
$B(v, d)$	Partitioned output buffer of operator v for downstream neighbor d, $B(v, d) = \bigcup_{t \in v}\{b(t, d)\}$
$a_{i,j}$	$a_{i,j} \in E$ is the data stream connecting two adjacent operators v_i and v_j
$e_{a,q}$	The qth event of a stream a, $e_{a,q} = \{q, <key, value>\}$
$s_{v,k}$	The kth data slice that goes through operator v
$s_{ij,k}$	The kth data slice that goes through task t_{ij}
$I(a, s_{v,k})$	$I(a, s_{v,k}) \in \mathbb{N}$ is the sequence number of the last input event from stream a to generate slice $s_{v,k}$
$O(a, s_{v,k})$	$O(a, s_{v,k}) \in \mathbb{N}$ is the sequence number of the last output event on stream a sent from slice $s_{v,k}$
$Q(v)$	A set of event sequence number that indicates the last events for each output stream of v
IST	A table to record pre-process slices in each task
OST	A table to record post-process slices in each task

3.1.2 Fast recovery with low overhead

For stateful operators, if the last checkpoint is achieved before the auto-scaling, rollback recovery means to undo the recent scaling adjustment. As in the case shown in Fig. 2, if any task failure occurs after the scale-up, it must roll back to the state preserved at $T1$. With unpredictable workload changes, this may trigger scale-up once again, resulting in serious delays to real-time processing.

In summary, the design goal in this paper is to provide a flexible, low-overhead upstream backup mechanism for EDSPSs, satisfying the requirements of recovery consistency and fast recovery under auto-scaling. To achieve this goal, we introduce an online adjustment function that enables upstream backup adaptive to elastic scaling (see Sect. 3.3). Before that, we first focus on the running status of operators (Table 2) preserved in case of failure in the follow section.

3.2 Running status of operators

In order to tolerate system failures, the running status of an operator should be preserved and restored before and after the failure. For each operator, the continuous input stream is divided into sub-streams by given time interval, i.e., data slices, as mini-batch processing units. Therefore, the snapshot of a running operator usually includes its processing state and data slices, which are preserved in the form of checkpoint state and upstream buffer state respectively. In particular, as an operator may have multiple tasks, we regard the topology relationship between operators as a part of the running status, i.e., mapping state.

3.2.1 Processing state

Operators perform user-defined processing functions (e.g., mapper or reducer tasks [10]) on each passing data slice and emit output events to downstream. Stateful operators need to maintain an internal summary of all past input slices, called processing state. Upon failure, traditional upstream backup suffers from long recovery times to rebuild the missing state. To avoid that, we should periodically checkpoint operator processing state to reliable storage.

For each processing data, we denote its partitioning key by $key \in \kappa$, where κ is the key domain. As an operator v_i may be parallelized into a set of tasks, the key domain κ will be partitioned for γ_i running tasks, which works as a partitioning function $D(\kappa) \rightarrow [1, \gamma_i]$. For each task t belonging to v_i, we define its checkpoint state $c(t)$ as a representation of the processing state $p(t) \in P(v_i)$, where $P(v_i)$ is the processing state of v_i.

3.2.2 Data slices

As the mini-batch processing unit, a data slice consists of all events flowing into an operator in a certain time unit, the states of which could be classified as pre-process, processing, or post-process. In our model, each operator has an Input Queue QI and an Output Queue QO. As shown in Fig. 3, QI is used to collect input events forming into the next data slice, while QO is in charge of buffering post-process slices. Meanwhile, for each slice, the data dependencies with upstream operators and downstream operators are preserved in table by table IST and OST (see Sect. 4.1), respectively.

There are two types of data slices in the output buffer: (1) *a part of current operator state*, output buffers store the slices that have not yet been processed by downstream operators, and therefore must be restored after failure; (2) *data backup for downstream neighbors*, these slices have been processed by downstream operators, and will be trimmed when they are no longer needed for downstream recovery. In other words, they are used to restore the former for downstream neighbors.

Data slices buffered in operator v are denoted $B(v)$. When a downstream operator d has multiple tasks, the partitioned buffer state of v specified for the downstream task t is denoted $B(v, t)$. For example, operator v_i has two downstream tasks(d_1, d_2). Then, the partitioned buffer states of v_i are denoted $B(v_i, d_1)$ and $B(v_i, d_2)$ in Fig. 3.

3.2.3 Mapping state

An operator in a stream-processing job may correspond to multiple tasks, so upstream operators have to decide which one to route an output event to based on its key partitioning. Given an operator v, we term its routing relationship with downstream neighbors as the mapping state $Map(v)$.

Table 2 A brief reference of the states of an operator

Status	Description
Processing state	An internal summary of all past input slices maintained by the operator
Data buffer	Slices stored in QO that have not yet been processed by downstream operators
Mapping state	Operator's routing relationship with downstream neighbors
Checkpoint state	Replication of the processing state
Backup buffer	Data backup buffered in QO for downstream neighbors

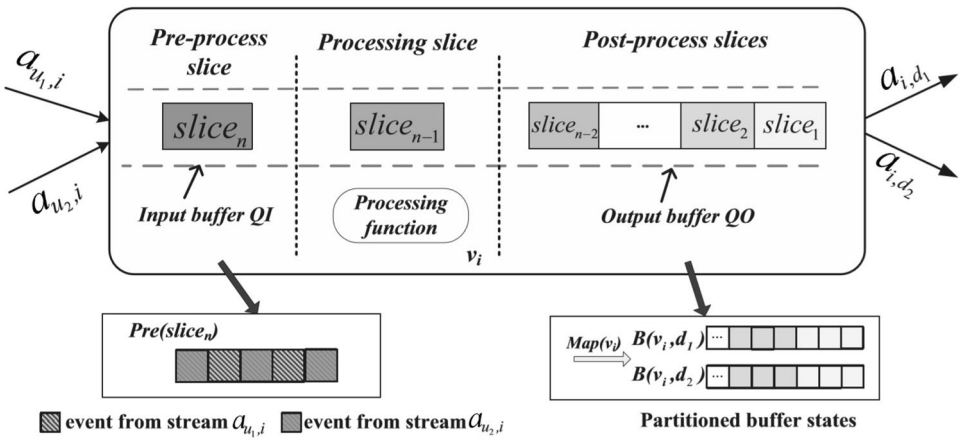


Fig. 3 Operator model

The mapping state guarantees that all the output events with the same partitioning key are routed to the same successor. In case of failure, the mapping state should be saved on the reliable storage. In particular, the auto-scaling changes operator parallelism dynamically, so these explicit mapping states may vary from time to time.

3.3 Upstream backup supporting auto-scaling

As EDSPSs enable operators scale up and down at runtime, we need a flexible upstream backup to preserve operator running status, which can keep recovery consistency under the auto-scaling while ensuring fast recovery. To handle this problem, we define an online adjustment function for upstream backup:

$$f : [B^{(T_a)}(u,v), C^{(T_a)}(v)] \xrightarrow{Map^{(T_a+1)}(u)} [B^{(T_a+1)}(u,v), C^{(T_a+1)}(v)]$$

Let f be the adjustment function used during time period of backup process. Once a given operator v auto-scales at T_a, the mapping state of upstream operator u changes as the parallelism of scaling operator changes from $\gamma^{(T_a)}$ to $\gamma^{(T_a+1)}$. Therefore, the backup states need to be updated accordingly. In particular, it involves dynamic repartition of upstream data backup and online adjustment of checkpoint state as follows:

3.3.1 Data backup repartition

Elastic scaling changes operator topology at runtime, and the upstream backups should be repartitioned to ensure that buffered events are mapped to the correct partition, i.e., upstream backup slices need to adjust the mapping relationship with downstream neighbors automatically.

As shown in Fig. 2, v_2 scales up into t_{21} and t_{22} at time T_a, and the relevant buffer state in v_1 should be split as $B(v_1, v_2) \xrightarrow{Map^{(T_a+1)}(v_1)} \{B(v_1, t_{21}), B(v_1, t_{22})\}$. Similarly, when v_3 scales down, its upstream backup should be joined accordingly.

3.3.2 Dynamic checkpoint update

To guarantee fast recovery of stateful operators, the checkpoint mechanism should keep track of the auto-scale changes of operator state. For example, v_2 scales up at T_a, leading to the new key partition $D^{(T_a+1)}(v_2)$. Based on this new key partition, the checkpoint state can be repartitioned as $C(v_2) \rightarrow \{c(t_{21}), c(t_{22})\}$. In this way, we ensure dynamic checkpoint adjustment (i.e., state re-partition) with small overhead, further avoiding the long recovery time in case of operator auto-scaling again.

In summary, online backup adjustment for auto-scaling operators means their upstream operators re-partition data buffers to the correct partitioned operator. Moreover, for stateful operators, dynamic update of checkpoint states to reflect operator scaling changes is also needed. That way, we propose a novel upstream backup approach that supports auto-scaling with these adjustments, as well as ensures the low impact on processing performance.

3.4 Assumption

We make the following assumptions. First, we assume that the execution plan of a stream-processing job is deployed over multiple nodes of a shared-nothing cluster. We further assume that one or more tasks of an operator are executed in parallel over partitions on individual nodes, i.e., a node can host one or multiple tasks that belong to the same operator. For example, in Storm [28], these tasks are called "executors". Meanwhile, we assume single-node failure at one time, which is realistic when tasks are distributed on different physical machines. The node will stop once the failure occurs. Besides, our proposed fault-tolerant approach is to support deterministic operators [18]. An operator is deterministic if it can produce the same output

every time when starts from the same initial state and receives the same events on each input stream, which are of critical to the upper layer of stream processing in existing works [4, 12, 16, 39].

4 Fault-tolerant mechanism

In this section, we present a novel fault-tolerant mechanism that enables upstream backup re-partition and dynamic checkpoint adjustment under the auto-scaling scenario. First, we propose the elastic data slice (EDS) as basic data backup unit, which enables data buffer re-partition for recovery consistency, and we then introduce a dynamic checkpointing mechanism supporting online state splitting and merging, which ensures fast recovery in case of auto-scaling.

4.1 Elastic data slice

With auto-scaling, the number of operator instances can increase and decrease at runtime, causing the changes of its mapping state with upstream neighbors. To guarantee consistency of data backup, we strive to design a self-adaptive data backup unit. More specifically, we propose an EDS model that enables upstream backup repartition for operator auto-scaling.

4.1.1 Data model

A data stream is an infinite sequence of events, each of which has a sequence number and a key-value pair [10]. The sequence number of an event is usually decided by its arriving order. The qth event of a stream $a \in A$ is denoted $e_{a,q} = \{q,\ \langle key, value\rangle\}$.

Given a processing operator, the events may come from different sources and be sent to multiple targets, and we define an upstream dependency function *du* as the data mapping relationship with input streams, and a downstream dependency function *dd* to be the data mapping relationship with output streams.

As exampled in Fig. 4a, task t_{21} has input streams $a_{11,2}, a_{12,2}$ and output streams $a_{2,41}, a_{2,42}$. For the 6*th* local input event $e_{*,6}$, the upstream dependency maps its local sequence number generated by the arrival order to the sourcing sequence number, i.e., $du : e_{*,6} \rightarrow e_{a_{12,2},4}$.

Similarly, the downstream dependency maps the local sequence number of each output event to the corresponding sequence number on each partitioned target stream, i.e., $dd : e_{*,5} \rightarrow e_{a_{2,41},3}$.

4.1.2 Slice model

Data slice is a finite set of continuous events that goes through an operator, which discretizes the continuous stream into data segments. It also represents the data backup unit in our fault-tolerant model.

Assuming task t_i has m input streams and n output streams, the upstream dependency of a given slice s works as a mapping $su(s) \rightarrow \{I(a_{u_1,i}, s)...I(a_{u_m,i}, s)\}$, i.e., a sequence number set of the last events in each input stream that form into the pre-process slice.

Similarly, the downstream dependency of slice s is a mapping $sd(s) \rightarrow \{O(a_{i,d_1}, s)...O(a_{i,d_n}, s)\}$, i.e., a sequence number set of the last events in each output stream corresponding to the post-process slice.

Therefore, the k*th* slice that goes through operator v_i can be further denoted $s_{i,k} = \{k, su(s_{i,k}), sd(s_{i,k})\}$. For instance, the first slice on task t_{21} is denoted by $\{1, (60, 40), (10, 10)\}$ in Fig. 4b.

4.1.3 Data re-partition

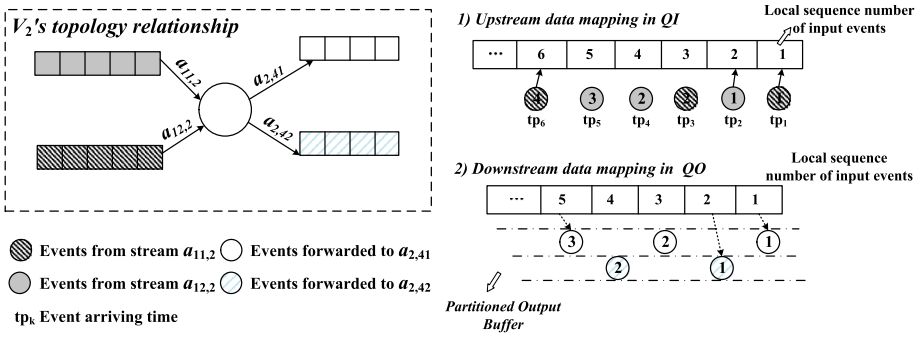
When an operator scales up and down, the proposal slice model supports data splitting and merging to adapt the topology changes. More specifically, if an operator d scales up at T_a, its upstream operator v_i re-partitions the output buffer to keep the data consistency. For slices in $B(v_i, d)$, this means adjusting their downstream dependencies according to the new mapping state, i.e., $s_{i,k} \xrightarrow{Map^{(T_a+1)}(v)} \{k, su(s_{i,k}), sd^{(T_a+1)}(s_{i,k})\}$. The data repartition is shown as follows.

4.1.3.1 Slice splitting If operator d scales up into tasks $t_{d_1}, t_{d_2}...t_{d_n}$, the backup slices $s_{i,k}$ buffered in each upstream task $t_i \in v$ should change their data dependency with d. Therefore, slice splitting is used to adjust its downstream data dependency as follows:

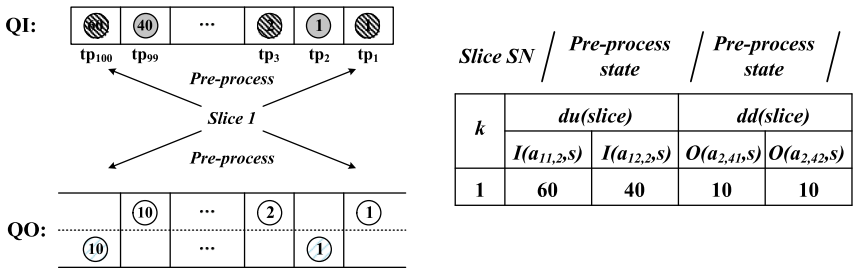
$$O(a_{i,d}, s) \xrightarrow{Map^{(T_a+1)}(i)} \{O(a_{i,d_1}, s), O(a_{i,d_2}, s)...O(a_{i,d_n}, s)\}. \quad (1)$$

4.1.3.2 Slice merging Similarly, assuming parallel tasks $(t_{d_1}, t_{d_2}...t_{d_m})$ of operator d scale down into one, the backup slices buffered in upstream task t_i should adjust their downstream dependencies accordingly. Given a backup slice $s_{i,k}$, the data merging is denoted as follows:

$$\{O(a_{i,d_1}, s), O(a_{i,d_2}, s)...O(a_{i,d_m}, s)\} \xrightarrow{Map^{(T_a+1)}(i)} O(a_{i,d}, s) \quad (2)$$



(a) Data dependency



k	du(slice)		dd(slice)	
	$I(a_{11,2},s)$	$I(a_{12,2},s)$	$O(a_{2,41},s)$	$O(a_{2,42},s)$
1	60	40	10	10

(b) Slice model

Fig. 4 Example of data slices on task t_{21}

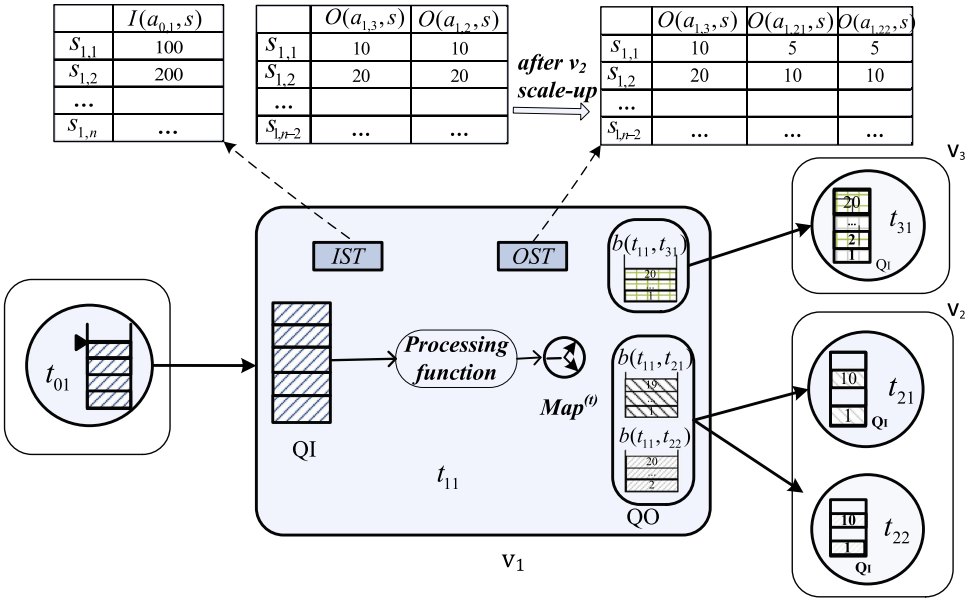


Fig. 5 Data dependency tracking: after scale-up, $O(a_{1,2},s)$ is split into $O(a_{1,21},s)$ and $O(a_{1,22},s)$, and then $dd(s_{1,1})$ is denoted $(10,5,5)$.

Note that we do not discuss data re-partition details of the auto-scaling operator here, as the running status (i.e., data slices and processing state) partition can be achieved by certain partitioning mechanisms like [11], which is not the focus of our work.

4.1.4 Dependency tracking

To support the above adjustments of the EDS, we present an online dependency tracking mechanism with two tables, *IST* and *OST*, defined as follows.

Input slice table (IST): a table that each operator (e.g., v_i) maintains to record pre-process slices with their upstream dependencies as $\{\langle s_{i,1}, su(s_{i,1})\rangle, \langle s_{i,2}, su\ (s_{i,2})\rangle, \ldots, \langle s_{i,k}, su(s_{i,k})\rangle\}$, where k is the slice sequence number.

Output slice table (OST):a table that each operator (e.g., v_i) maintains to record post-process slices with their downstream dependencies as $\{\langle s_{i,1}, sd(s_{i,1})\rangle, \langle s_{i,2}, sd(s_{i,2})\rangle, \ldots, \langle s_{i,k}, sd(s_{i,k})\rangle\}$, where k is the slice sequence number.

At normal runtime, *IST* and *OST* record each passing *slices* with their pre-process and post-process state, i.e., they maintain the data dependencies with input and output streams, respectively. These slice recordings will be removed from *IST* and *OST* after they are processed and acknowledged by all the downstream neighbors, i.e., queue trimming (see Sect. 5.1).

Once an auto-scaling operator occurs, its upstream operator's *OST* will be synchronized with the data re-partition, i.e., existing *slice* recordings will be updated with their split or merged states according to Eqs.1 and 2. As shown in Fig. 5, when v_2 scales up into parallel tasks t_{21} and t_{22}, the existing *OST* recording of slice $s_{1,1}$ is changed from $\langle s_{1,1}, (10, 10)\rangle$ to $\langle s_{1,1}, (10, 5, 5)\rangle$ on upstream task t_{11}. Similarly, the downstream operator's *IST* will update slice upstream dependencies in response to the new mapping state of the current scaling operator.

Upon failure, the operator loses all the running status including tables *IST* and *OST*. As upstream operators have buffered all the events needed for the operator's recovery, these tables will be restored along with the re-processing of these data. In particular, to guarantee the data consistency after recovery, downstream operators will send a set of latest event sequence number received from the failed operator based on *IST*.

4.2 Dynamic checkpointing adjustment

To implement fault tolerance for stateful operators, we execute periodic checkpoints along with upstream backup (see Sect. 5.1). Generally, operator checkpointing is executed asynchronously and triggered every interval T_c. However, auto-scaling introduces dynamic partitioning of operator running status, requiring online checkpoint state adjustment. To handle this problem, we introduce a dynamic checkpointing mechanism [37] to support online state merging and splitting.

Given an operator v_i, the processing state $P(v_i)$ and checkpoint state $C(v_i)$ correspond to the same key domain k_i. At normal runtime, every update to the processing state is directly reflected at the corresponding key-value store. Upon triggering the checkpointing timing, all the maintained key-value stores are scanned and the corresponding updated states are saved as a checkpoint file in remote storage.

Once dynamic scaling of v_i is triggered, as discussed in Sect. 3.3, the new key space partition $D^{(T_a+1)}(v_i)$ will be sent to the backup side, which is used to guide the splitting and merging of the checkpoint state. In case of failure, the checkpoint state C_i will be reconstructed according to the new key distribution. For instance, in Fig. 2, after auto-scaling, v_2's checkpoint state is split into $c(t_{21})$ and $c(t_{22})$. In this way, we can guarantee fast recovery despite auto-scaling.

5 Protocol and algorithm

In this section, we present an integrated upstream backup protocol that consists of (i) upstream backup with periodical checkpointing at the normal runtime, and (ii) elastic backup adjustment during auto-scaling. We then propose a failure recovery algorithm based on this protocol. In this way, we guarantee the data consistency at runtime while offering fast recovery in case of failure.

QTP part1: upstream operator

BEGIN
// before v_i starts to process
T1: Send $s_{i-1,k}$ to v_i
T3: $L0\text{-}ACK\ (s_{i-1,k}) \leftarrow ReceiveACKMessage()$
// after v_{i+1} has received the output events of $s_{i,k}$
T5: $L1\text{-}ACK\ (s_{i-1,k}) \leftarrow ReceiveACKMessage()$
FOR s in $s_{i-1,j}, s_{i-1,j+1} \ldots s_{i-1,k}$ DO {
trim s from QO }
END

QTP part2: current processing operator

BEGIN
// after v_i has sent *slice* $s_{i-1,k}$
T1: $s_{i,k} \leftarrow ReceiveNewEvents()$
$RecordSliceInIST(s_{i,k})$
Send $L0\text{-}ACK(s_{i-1,k})$ to v_{i-1}
T2: $post(s_{i,k}) \leftarrow ProcessingFunction(s_{i,k})$
$RecordSliceInOST(post(s_{i,k}))$
IF v_i is *stateful* THEN {
$C(v_i) \leftarrow Checkpoint(P(v_i))$}
T3: Send $post(s_{i,k})$ to v_{i+1}
// after v_{i+1} have sent $L0\text{-}ACK$
T4: $L0\text{-}ACK\ (s_{i,k}) \leftarrow ReceiveACKMessage()$
$UpdateStateInOST(L0\text{-}ACK(s_{i,k}))$
T5: Send $L1\text{-}ACK(s_{i-1,k})$ to v_{i-1}
END

QTP part3: downstream operator

BEGIN
// after v_i finishes $s_{i,k}$ processing
T3: $s_{i+1,k} \leftarrow ReceiveNewEvents()$
$RecordSliceInIST(s_{i+1,k})$
T4: Send $L0\text{-}ACK(s_{i,k})$ to v_i
END

5.1 Upstream backup

An upstream backup model coordinating with checkpointing is proposed to control the backup process and tolerate operator failure with minimum storage. The key idea of this model is to introduce precise trimming of slice backups with ACK messages [4] among operators.

5.1.1 Queue trimming

Upstream nodes act as backups for downstream neighbors by preserving their output events in *QO*s. These backups will be removed from upstream nodes when the downstream neighbors completely process them and send results to the successors safely. For stateful operators, the data trimming must be postponed until all the backup slices in a checkpoint cycle are acknowledged by successors.

The trimming of backup slices is achieved by the two-level ACK mechanism, where slices are marked as outdated only if they get acknowledged by all the dependent successors. Take a trimming iteration as an example, Figure 6 shows a typical communication sequence between three nodes v_{i-1}, v_i, v_{i+1}. Note that notations $T1$, $T2$, $T3$, $T4$ and $T5$ represent the synchronization time points.

Based on the two-level ACK model, we put forward a queue trimming protocol *QTP* for basic stream processing scenario without operator auto-scaling. Given the last slice (e.g, $s_{i-1,k}$) of one iteration, we show details of data and message communication among operators (i.e., v_{i-1}, v_i, v_{i+1}) in different parts of *QTP* as follows.

Each node produces and sends events downstream while storing them in QOs. Each node also periodically acknowledges the reception of input slices by sending *L0-ACK*(i.e., the level-0 ack message) to upstream neighbors. When a node v_i receives *L0-ACK* from downstream neighbor v_{i+1}, it notifies its upstream neighbor v_{i-1} about the earliest buffered events that contributed to producing the acknowledged slice. After the notification *L1-ACK* (i.e., the level-1 ack message) is received, each upstream task prunes its output buffer accordingly. In particular, if v_i is stateful, the message *L1-ACK* will be postponed to send until v_i finishes the next checkpoint.

5.1.2 Online backup adjustment

To guarantee recovery correctness in case of operator auto-scaling, we present an asynchronous protocol *OAP* for online backup adjustment. During the execution of *QTP* among operators v_{i-1},v_i, and v_{i+1}, data consistency is ensured under the topology changes. As shown in Fig. 7, a scale-up case of v_i is used to show the asynchronous maintenance and update of *ISTs/OSTs* among the operators.

We represent more details in the following protocol *OAP*, focusing on each operator (i.e., v_{i-1}, v_i, v_{i+1}) in part1, part2 and part3, respectively.

OAP part1: current processing operator v_i
BEGIN
// v_i is to scale-up
T0: *StopSendingOutput*()
Send *Scale_Up_Init*($\gamma_i^{(T0+1)}$) to v_{i-1} and v_{i+1}
$D^{(T0+1)}(v_i) \leftarrow$ *GetNewKeyDistribution*($\kappa, \gamma^{(T0+1)}(v_i$
Send *Scale_Up_Init*[$\gamma_i^{(T0+1)}, D^{(T0+1)}(v_i)$] to backup node v_{i-1}
// after partitioning $v_i's$ running status
T1: *PartitionCheckpointState*($C(v_i), D^{(T0+1)}(v_i)$)
FOR t_{ij} in (v_i) DO {
copy *PartitionedCheckpointStatus* to t_{ij} }
Send signal *Scale_Up_FIN* to v_{i-1}
// after receiving *events* from v_{i-1}
T3: FOR t_{ij} in (v_i) DO {
$s_{ij,k} \leftarrow$ *ReceiveNewEvents()*
RecordSliceInIST($s_{ij,k}$) }
// after receiving *Downstream_Get_Ready* from v_{i+1}
T4: *ResumeSendingOutput*(v_{i+1})
FOR t_{ij} in (v_i) DO {
Send $s_{ij,k}$ to v_{i+1} }
END

OAP part2: upstream *operator* v_{i-1}
BEGIN
// after receiving *Scale_up_Init* from v_i
T1: *StopSendingOutput*(v_i)
$Map^{(T0+1)}(v_i) \leftarrow$ *GetMappingState*($v_i, D^{(T0+1)}(v_i)$)
FOR $s_{i-1,j}$ in $B(v_{i-1})$ DO {
GetSliceSplit($O(a_{i-1,i}, s_{i-1,j}), Map^{(T0+1)}(v_i)$)
UpdateSliceInOST(*SliceSplitStatus*($s_{i-1,j}$)) }
// after receiving message *Scale_up_FIN*
T2: *ResumeSendingOutput*(v_i)
Send $s_{i-1,k}$ to v_i
END

OAP part3: downstream *operator* v_{i+1}
BEGIN
// after receiving *Scale_Up_Init* from v_i
T1: *StopSendingACKs*(v_i)
UpdateUpstreamDependency($v_i, \gamma_i^{(T0+1)}$);
// after receiving message *Slice_Partition_State*
T2: *UpdateStateInIST*(Slice_Partition_State)
ResumeSendingACKs(v_i)
Send *Downstream_Get_Ready* to v_i
// after receiving events from v_i' tasks
T5: $s_{i+1,k} \leftarrow$ *ReceiveNewEvents()*
RecordSliceInIST($s_{i+1,k}$)
END

In the example, operator v_i scales up at time $T0$, and then notifies its upstream neighbor v_{i-1} and downstream neighbor v_{i+1}. Upon receiving the scaling notification (i.e.,*Scale_up_Init*) from v_i at $T1$, upstream operator v_{i-1} stops sending output to v_i and generates a new mapping state $Map^{(T0+1)}(v_i)$. Base on the new mapping state, v_{i-1} repartitions the buffered slices $B(v_{i-1})$ while updating their recordings in *OST* accordingly. For downstream operator

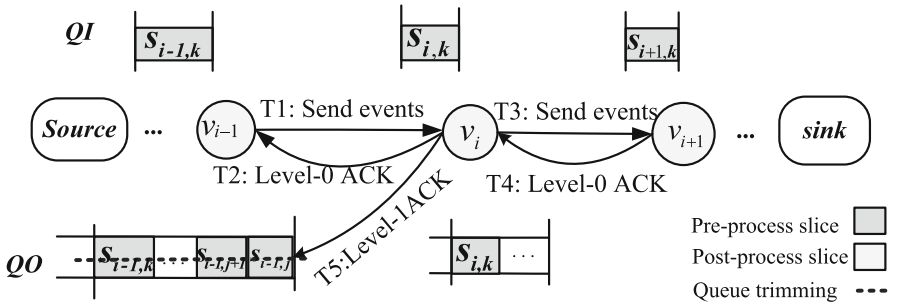


Fig. 6 One trimming cycle of operator backup

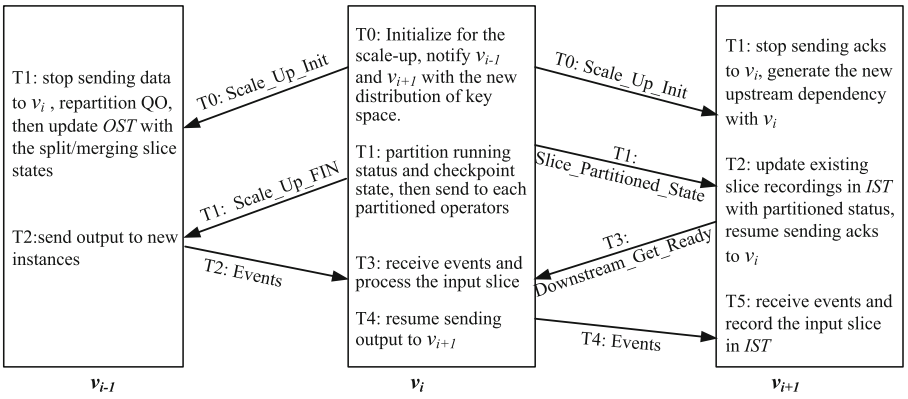


Fig. 7 Online backup adjustment for operator scale-up

v_{i+1}, it stops sending ACK messages to v_i after the notification *Scale_up_Init* arrives.

When v_i finishes the running status partition at $T1$, it sent the messages of slice partitioned states (i.e., *Slice_Partition_State*) to downstream neighbors. Based on the slice partitioned states, v_{i+1} updates existing slice recordings in *IST*, and then resumes sending ACK to v_i after sending the notification *Downstream_Get_Ready* to v_i at $T2$. At the same time, v_{i-1} resumes sending output events to v_i after receiving the message *Scale_Up_FIN* at $T2$.

After the above adjustments, the ACKs (e.g,*L0-ACK* and *L1-ACK*) of v_{i+1} will be sent to the partitioned operator v_i correctly. Eventually, v_i's tasks start to receive events from v_i at $T3$, and then send the output to v_{i+1} after processing them at $T4$.

Therefore, *OAP* ensures the correctness of queue trimming among operators (i.e., v_{i-1}, v_i, v_{i+1}) despite the topology changes. In this way, we guarantee data consistency under auto-scaling.

5.1.3 Discussion

The above two-level ACK model is able to resist single-point failure. For multiple concurrent node failure, we can extend the model with *n-level* ACK, where an operator is in charge of keeping previous output *slices* for an *n-1* level of successors. Specifically, each *operator* is able to look back slices from its n-level precursors. For m multiple simultaneous failures on adjacent operators, each recovering operator is able to track previous slices from $n-1$level precursors. Meanwhile, we can still apply our online backup adjustment to the extended model, solving the data consistency issues under auto-scaling. Although not within the scope of this paper, this extended model that resists n−1 level failures can be a future avenue of our work.

5.2 Failure recovery

Algorithm 1 details the steps to recover a failure operator v_i. When operator v_i fails, a new one just restarts another reallocated task $v_i^{'}$ by system automatically. Then it restores buffer state $Q(v_i)$ from the recovery information sent by each successor d, and the processing state $P^{'}(v_i)$ is recovered by the last checkpoint state $C(v_i)$ if v_i is *stateful*. For any operator $u \in up(v_i)$, where $up(v_i)$ denotes the set of $v_i^{'}$ upstream operators, the buffered *slices* of $B(u, v_i)$ will be replayed by v_i.

Note that, for highly time-sensitive applications, we adopt parallel recovery [4] to reduce processing latency, avoiding the cascading failure effect on facing workload peaks. Parallel recovery combines the recovery with operator auto-scaling, i.e., each upstream operator u will re-partition its output buffer with the parallelism degree γ_i (line 12) before the replay.

Algorithm 1 Algorithms for recovery of operator v_i

```
1: v_i' ← get-new-operator()
2: Q(v)' ← retrieve_EN_state
   (d.send_RecoveryInformation(v_i'))
3: if v_i is stateful then
4:     P'(v_i) ← retrieve_processing_state(c(v_i))
5: else
6:     P'(v_i) ← null
7: end if
8: restore_state(v_i', P'(v_i), Q(v)')
9: start-operator(v_i')
10: for all u in up(v_i) do
11:     stop-operator(u)
12:     // Re-partition buffer-state if it is parallel recovery
13:     if Parallel_recovery is true then
14:         partition_buffer_state(B(u, v_i), γ_i);
15:     end if
16:     replay-buffer-state(B(u, v_i))
17:     start-operator(u)
18: end for
```

5.2.1 Consistency guarantee

We consider a recovery method to be consistent [12] if and only if a job after the recovery always produces processing results as if no failure had happened. Our two-level ACK model ensures recovery consistency for single-node failure despite auto-scaling.

Suppose there is a failure to v_i during the period between its two checkpoints; then, all its running status is lost. Before the failure, v_i has processed up to slice $s_{i,k}$ and its processing state is $P^{t_k}(v_i)$. Clearly, the upstream neighbors v_{i-1} are not affected by the failure. Thus, this ensures v_i can correctly resume processing after recovery.

After v_i is recovered from the last checkpoint, it starts to replay data slices stored in v_{i-1}'s output buffers. Note that all these necessary data contributing to $P^{t_k}(v_i)$ from the last checkpoint are still in the upstream output queues according to *QTP*. Thus, v_i's running status will be restored after the replay. The results flow into downstream neighbors as these history events are re-processed by v_i. The downstream operators will drop duplicated data and eventually process new events produced by v_i.

5.2.2 Discussion

We guarantee exactly-once data delivery by using the ordered event sequence number, i.e., there is no duplicated and out-of-order data elements during the recovery phase. Thereby, when a deterministic operator is recovered, we ensure that it produces exactly the same output as in the case in which no failure has occurred.

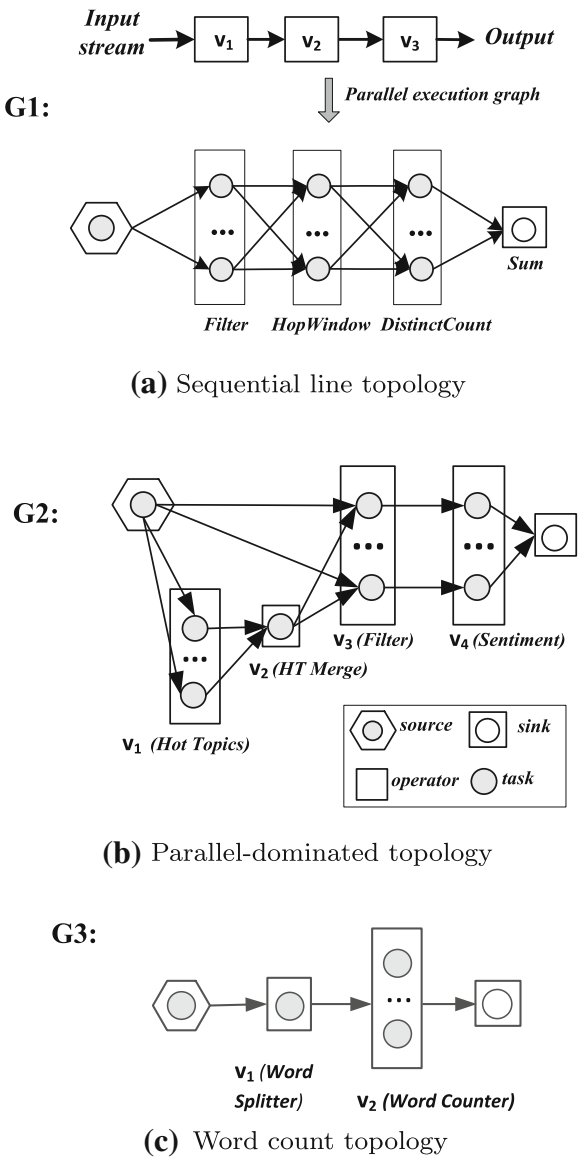
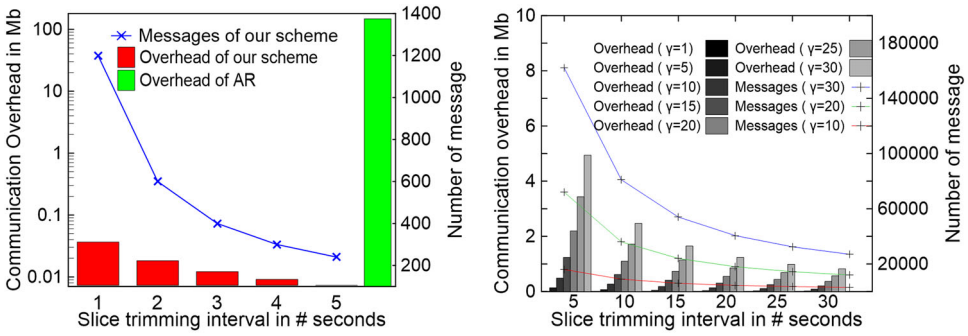


Fig. 8 Stream topologies used in the experiments of runtime overhead

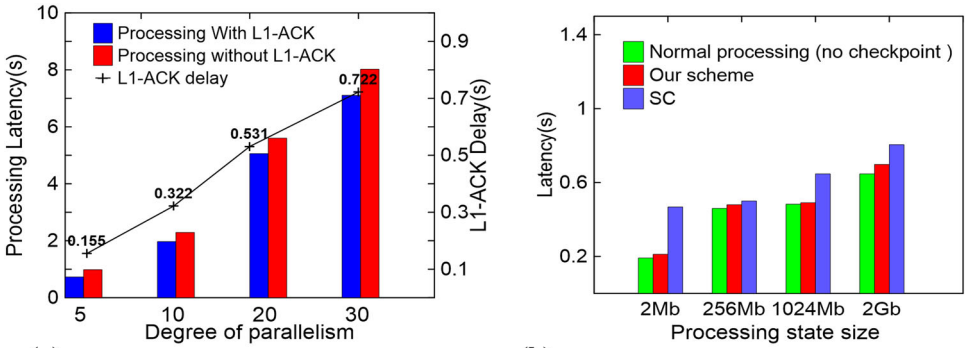
5.3 Overhead analysis

For stateless operators, the overhead of our backup model comes from the ACK messages transmission. A lower ACK frequency reduces bandwidth utilization, but increases output queue size and recovery time. For stateful operators, upstream operators keep track of the slices after the last checkpoint. The checkpoint interval determines the trade-off between runtime overhead (i.e., latency overhead and memory overhead) and recovery time.



(a) Sequential scenario of topology G1 **(b)** Parallel scenario of topology G1

Fig. 9 Communication overhead comparison



(a) Communication latency in parallel scenario of G1 **(b)** Processing latency at v_2 in sequential scenario of G1

Fig. 10 Latency comparions

5.3.1 Communication overhead

For upstream backup, we consider the *QTP* protocol for complexity analysis as it has the highest communication cost. The communication overhead in our work refers to the bandwidth consumed for slice trimming, i.e., the transmission payload of ACK messages. We begin our analysis by assuming that an *EDSPS* job is performed in the cluster environment with a scale of $O(M)$, where $M = L \times N$. L denotes the maximum parallelism of operators and N represents the number of operators in the job. Furthermore, for analysis, we consider that v_i is *stateful* and triggers its checkpointing every τ seconds. The parallelism of operator v_{i-1}, v_i, v_{i+1} is denoted by l_1, l_2 and l_3 respectively. The time bucket per slice is 1s by default.

At T1 in *QTP*, v_{i-1} emits output to v_i with receiving $l_1 \times l_2$ *L0-ACK* messages. After T2, v_i gathers all events from its precursors to form *slice* $s_{i,k}$ and emits the output at T3. Then v_i receives $l_2 \times l_3$ *L0-ACK* messages from its successors until T4, i.e.,v_{i+1} receives $s_{i+1,k}$. After v_{i+1} sends *L0-ACK* messages to v_i, the notifications will be moved forward to the predecessor v_{i-1}, i.e., $l_1 \times l_2$ *L1-ACK* messages.

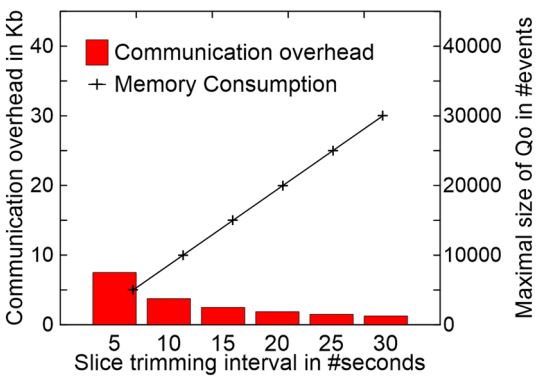


Fig. 11 Maximal size of *QO* at v_1 in G1

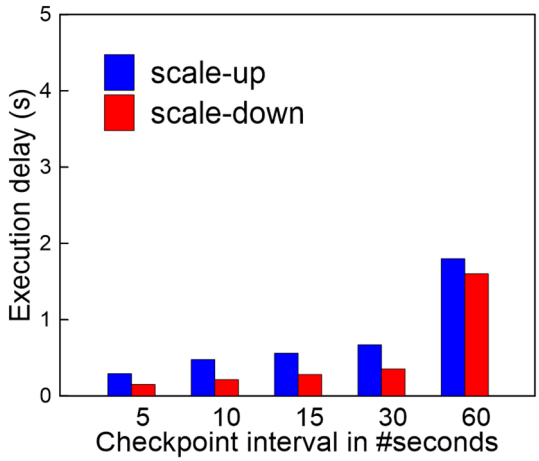


Fig. 12 Execution delay at v_3 in G2

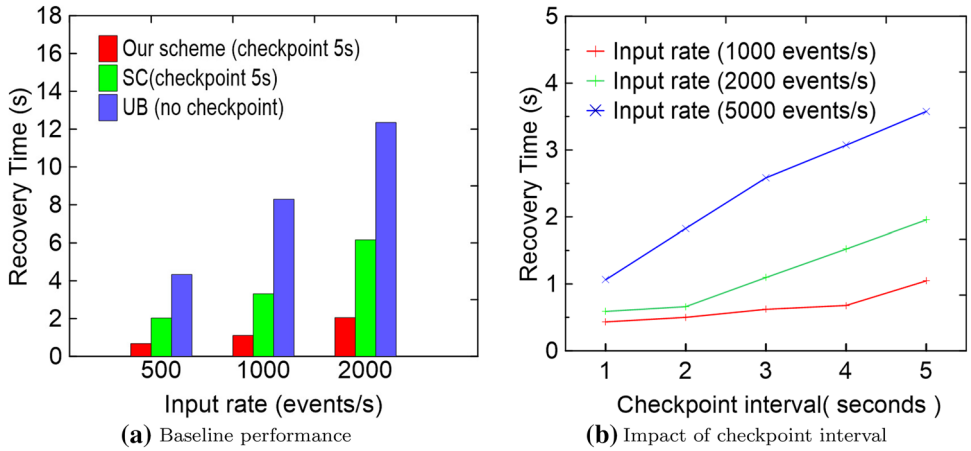


Fig. 13 Recovery time comparison of v_2 in G3

(a) Baseline performance **(b)** Impact of checkpoint interval

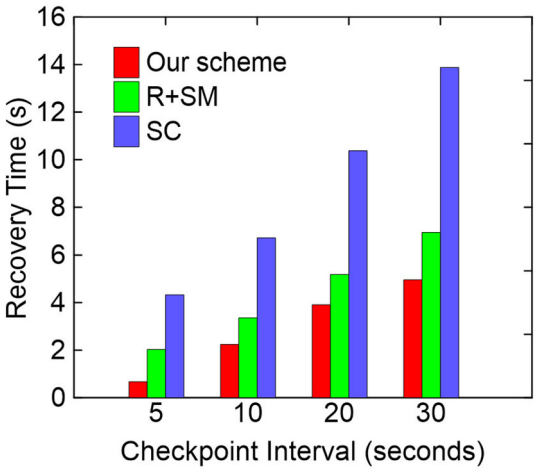


Fig. 14 Recovery time of v_2 in G3 after scale-up

The total communication complexity of one trimming circle is $T_{QTP} = O(L^2) + O(L^2) + O(L^2)$, which can be simplified as $T_{QTP} = O(L^2)$. We can see that the communication complexity analyzed above has a square relationship with computing scale. Comparing with the processing data size, we consider the overhead on communication messages is acceptable as analysed in the following.

Assuming that $M_{L0/1-ack}$ denotes the size of *ACK* message, M_{slice} denotes the size of the data slice and ω represents the input rate of v_i, we have the processing data size of v_i, $D = \omega \times \tau \times l_1 \times M_{slice}$, and the communication overhead of one trimming circle $C = l_1 \times l_2 \times M_{L0-ack} + l_2 \times l_3 \times M_{L0-ack} + l_1 \times l_2 \times M_{L1-ack}$. If we take L to replace l_1, l_2, l_3, and m to replace $M_{L0/1-ack}$, the ratio of our communication overhead to the data size can be given as $C/D = (3 * L/\tau) \times (m/M_{slice})$.

To mimic the scenario with high communication overhead, we set a large empirical value of 30 for L and 5 s for the slice trimming interval according to existing work [4] (i.e., $L = 30$ and $\tau = 5$); we consider a small empirical value for v_i's input stream rate, i.e., 1000 events/s (usually 500–60,000), $M_{slice} = 172$ bytes and $m = 32$ bytes according to Sect. 7.2.1. Consequently, the value of C/D is as small as 0.00335.

In summary, we consider that the proposed protocol *QTP* ensures a relatively low communication overhead compared to the processing data size in DSPS.

5.3.2 Latency overhead

For stateless operators, the processing overhead from our ACK messages transmission is usually negligible (see Sect. 7.2.1). For stateful operators, the latency overhead consists of generating and processing checkpoint messages (proportional to the bandwidth overhead), which is also small in our approach since the checkpoint messages only contain processing states. The latency impact of online backup adjustment on normal processing has been evaluated in Sect. 7.

5.3.3 Memory overhead

Generally the consumption of main memory at a host that an operator induces can be divided into three parts: one is the memory used for input buffer (i.e.,QI) and intermediate processing results. We ignore this part as it contains no specific overhead for our approach. The other part is the memory used by all data stored solely for the purpose of enabling efficient upstream backup. This part reflects the size of *QO*, which is determined by the ACK frequency or checkpoint interval (in case of stateful operators). Thus, we explore the impact of different slice trimming (i.e., checkpoint intervals) in Sect. 7.2.1. Note IST and OST are not considered as they only record some index IDs for each slice. The size of these tables can be quite small compared to the processing data buffered in QO.

Table 3 Comparing approaches

Type	Description
AR	Active replication
UB	Upstream backup
SB	Source backup
SC	Synchronous checkpointing technology
R+SM	Recovery with state management

Table 4 Stream topology types

Topology	Description	Num of servers
G1	DAG with sequential line topology	5
G2	DAG with parallel-dominated topology	6
G3	Word count query	4

6 Implementation

We implement our protocols and algorithm on SPATE [35], an experimental stream processing system prototype built on an improved combination of Hadoop version1.1.2 and S4 [30]. In our system, a logically centralised topology manager, called TP_Manager, is automatically generated and appended to the user-submitted job topology. The responsibilities of the manager include collecting workload statistics, handling topology tracking and node failure.

To evaluate our dynamic backup adjustment, a simple yet effective elastic scaling policy has been realized on SPATE. We enforce an upper threshold and a lower threshold for the CPU usage of an individual host. Every 5 s, operator hosts submit CPU utilization reports to the monitor in TP_Manager used to identify bottleneck operators. When two consecutive reports from an operator are above the upper threshold, the manager notifies the scale-up of this overloaded operator. Similarly, a host is marked as underloaded and notified to scale-down, if the utilization drops below the lower threshold for two consecutive measurements.

In our approach, detecting overload and failure are handled in the same fashion. The fault-tolerance coordinator in TP_Manager detects node failures by checking their heartbeats. Upon detection of a failure, the coordinator calls the recovery algorithm presented in Sect. 5.2 to restore it by restarting the new host. The status of TP_Manager is periodically persisted to the reliable file system in case of failure.Note that the TP_Manager is implemented by ZooKeeper [17], which has been used in Hadoop as a high-performance service for coordinating processes of distributed applications.

Table 5 Experimental setup for topology G1

Components	#Tasks of serial operator	#Tasks of parallel operator
Source	1	γ
Filter (v_1)	1	γ
HopWindow (v_2)	1	γ
DistinctCount (v_3)	1	γ
Sum	1	1

Table 6 Experimental setup for topology G2

Components	Num of tasks
Source	1
Hot topics (v_1)	5
HT merge (v_2)	1
Filter (v_3)	5
Sentiment (v_4)	10\2\5
Sink	1

Table 7 Experimental setup for topology G3

Components	Num of tasks
Source	1
Word splitter (v_1)	1
Word counter (v_2)	1\2
Sink	1

Table 8 Communication latency in the sequential scenario of G1

Slice trimming interval(s)	1	2	3	4	5
Delay (ms)	28	24	59	46	51

7 Evaluations

In this section, we evaluate our fault-tolerant protocols and algorithm. The goals of our experimental evaluation are to investigate: (i) the runtime overheads introduced by our fault-tolerance, and (ii) the effectiveness of our recovery mechanism in a typical query application of an EDSPS.

Table 9 Comprehensive performance comparison

Method	Communication_overhead(Mb)	Latency_overhead(s)	Recovery_time(s)	rt_after_ Scale_up(s)
AR	147.6	Negligible	Negligible	X
UB	Negligible	0.024	12.35	X
SB	Negligible	0.028	15.015	X
SC	22.14	0.157	3.312	13.878
R+SM	18.45	0.136	3.127	6.939
Our scheme	0.007	0.051	2.054	4.957

7.1 Experimental settings

The experiments are conducted using our stream processing system prototype SPATE [35] (see Sect. 6). We deploy it on a computer cluster running *RedhatLinux* 2.6.32 − 358.*el*6.*x*86_64 operating system for the query operators. Each server has 16 cores, 32 Gb of RAM and 2.60 GHz CPU. All servers are interconnected with 1 Gb Ethernet switches.

To examine our runtime overhead, we implemented different fault tolerant methods, including active replication [18] (AR), which offers seamless failure recovery by switching among active backup tasks. Since it introduces at least doubling communication overhead of the number of events sent through the network, we compare the communication overhead with AR as the worst case. For latency overhead, our approach is compared with the checkpoint-based methods (i.e.,SC), which also generate the latency overhead due to the checkpoint. Meanwhile, we also compare recovery efficiency with upstream backup [18] (UB) and source backup (SB) in Storm [28] in the baseline performance experiments. UB buffers events as downstream backup in each operator and requires the entire history of events to recover the operator state, and SB is viewed as a variant of UB, where the events are only buffered and replayed by the source operator.

In addition, we compare the recovery time under the auto-scaling with recovery using state management [4] (R+SM), which can handle operator scale-up by partitioning the MRC instead of its current running status, and Synchronous checkpoint (SC), a traditional checkpoint technology used in Spark streaming [38], where operators perform checkpoints synchronously without dynamic updates for auto-scaling. SC is chosen as the worst case that needs to redo the scale-up after recovery. The comparison approaches are shown in Table 3.

In the first set of experiments, we implemented a sequential line topology *G*1 (Distinct Count [31]) to study runtime overhead of the proposed approach with respect to three aspects: communication overhead, processing latency, and the volatile storage of *QO*. In particular, we implement a real stream application topology *G*2 (Twitter Sentiment [25]) that has many tasks and edges caused by auto-scaling, to explore the overhead introduced by our online backup adjustment. The structure of these topologies are depicted in Fig. 8. To mimic the real-world scenario, each operator in the same topology was deployed on different servers.

Table 4 lists the total number of servers being used in the experiments. Moveover, the details of how the operators are parallelized, i.e., the task number of each operator in G1 and G2, are given by Tables 5 and 6, respectively. Note that the parallelism degree of operators v_1, v_2 and v_3 in G1 is set to the same value, i.e., $\gamma_1 = \gamma_2 = \gamma_3 = \gamma$, which can vary in the parallel scenario as shown in Fig. 9b.

In the following set of experiments, we implemented a windowed word frequency query *G*3 as shown in Fig. 8c to examine the recovery performance that executes over a stream of sentence fragments and counts word number over a 30-s window.

In G3, all of the source tasks produce input events for their downstream neighboring tasks in a specified rate (1000 events/s or 2000 events/s). The word splitter (i.e., v_1) tokenizes input slice into events and outputs the word number of the sentence in each event; and the word counter (i.e., v_2)is responsible for counting words per slice like local reduce, the processing state of which is maintained as a hash table of word counts. Table 7 shows the structure of topology G3 and its resource mapping.

7.2 Results

7.2.1 Runtime overhead

7.2.1.1 Communication overhead With a 1000-event/s input rate, we investigated how much additional data (i.e., communication messages) is sent over the network within 5 mins in the sequential scenario of G1. The overhead depends on *slice* ACK frequency, so we observe the impact of *slice* trimming interval on it. ACK messages are all

32 bytes (8 bytes for the source operator sendport, 8 bytes for the acked event sequence number, and 16 bytes for the identifier of ACK), and an event is 172 bytes (32 bytes for some transmitting information such as event sequence, source operator, slice sequence number, and ACK sequence number, 140 bytes for payload).

Figure 9a shows that communication overhead decreases as queue trimming interval (i.e., the checkpoint interval of v_2) increases. For the same amount of processing data, the shorter interval means more of v_2's ACK messages will be sent in one backup circle. The results that show our approach induces less communication overhead than the compared approach AR.

Figure 9b shows the communication overhead under different operator parallelism value γ of topology G1. The maximum communication overhead is 4.94 Mb, which is under the parallelism of 30 and a trimming interval of 5 s. Compared with the size of data (147.6 Mb) to be processed, the bandwidth cost for communication overhead in our approach is sufficiently small, which illustrates the analysis conclusion in Sect. 5.3.

7.2.1.2 Latency overhead To measure our proposed mechanism overhead on processing latency, we obtained its effect by comparing the difference of *slice* processing latency between our approach and a baseline without backup under an 1000-events/s input rate. Table 8 shows that the latency is so small that it can be negligible in a sequential scenario of G1, i.e., $\gamma = 1$.

We increase the input rate from 2500 to 15,000 event/s in G1, and the corresponding parallelism of v_1 varies from 5 to 30. In Figure 10a shows that our latency is negligible in both cases ($\gamma = 5$ and $\gamma = 10$). Even the maximum latency in the case of $\gamma = 30$ is no more than 1 s. These results illustrate that our two-level ACK mechanism brings such a small latency overhead that it has little effect on normal stream processing.

For stateful operators, we illustrate the impact of our asynchronous checkpointing on the processing latency of a single node. We measure the average latency time per slice of 10 runs on v_2 in the sequential scenario of G1 after processing 100,000 events, where the input rate is 2000 events/s. The system runtime checkpoints the computation states every 5 s. Figure 10b captures the results with the state capacity growing from 200 Kb to 2 Gb as we synthetically vary the dictionary size. When the state size expands to 2 Gb, the total latency from SC compared to normal processing rises to 2 s after all the events past (i.e., almost 10 checkpoints), while our asynchronous checkpointing has little affect, only incurring an extra delay around 0.5 s.

7.2.1.3 Memory consumption We explored the effect of queue trimming interval (i.e., v_2's L1-ACK frequency) on the maximal size of v_1's *QO* and the communication overhead under an input rate of 2000 events/s. Figure 11 shows that a larger trimming interval reduces bandwidth utilization, but increases maximal output queue size. As the above plots include the maximal size of operator QO, which decides the expected recovery time for that queue trimming interval, there is the following trade-off: the longer the queue trimming interval, the lower the impact on event processing, but at the expense of a longer recovery time after failure.

7.2.1.4 Online auto-scaling support overhead To measure the overhead introduced by online auto-scaling support, we compared two sets of 10 runs of *G*2 on six nodes, with and without dynamic backup updates, and the time bucket of a default slice is 1 s. The input rate in our experiment was initially 50,000 events/s, handled by 10 *Sentiment* instances. Then, we elevated the rate to 120,000 events/s after 60 s (i.e., 20 source instances)and dropped to 40,000 events/s after another minute. During the high load period, operator *Sentiment* scaled up into 20 parallel instances, and then returned to five instances after the rate dropped back. In addition, we used upstream operator hosts as the backup nodes to store the checkpoints, so the overall overhead could be observed more intuitively.

In Fig. 12, we show the processing delay on upstream operator caused by *Sentiment*'s scale-up and scale-down. When *Sentiment* scales up, it leads to a brief latency spike of the upstream neighbor to do backup re-partition. The second short latency spike occurs when *Sentiment* scales-down in five instances. Meanwhile, we observe the divergence of their CPU and network overhead, which is 1.75% and 0.23%, respectively. As described in Sect. 5.1, online backup adjustments involve the extra communication overhead among operators. These results show that our mechanism of online backup updates does not introduce noticeable overhead and can be accomplished in a seamless manner.

7.2.2 Recovery efficiency

In order to evaluate failure recovery, we first compared recovery time with other fault-tolerant approaches. Then, we measured the impact of queue trimming interval on recovery time, i.e., the checkpoint interval in case of stateful operator.

7.2.2.1 Baseline performance We ran topology G3 on four servers and observed recovery times of the server hosting operator v_2 for the approaches SR, UB and ours. During the experiment, we set the checkpointing interval to

5 s. The recovery latency was calculated from the moment that the failure was detected to the instant when the failed operator recovered the running status before failure. In addition, we ignored the time to allocate the new server as pre-deployed operators were provided in the experiment.

Figure 13a shows results averaged over 10 runs. The standard deviation of recovery time under different input rates (i.e. 500 events/s, 1000 events/s and 2000 events/s) is 98 ms, 117 ms and 285 ms, respectively. Our scheme achieves lower recovery time than both SR and UB, since our mechanism enables a shorter queue trimming interval, and it reprocesses fewer events to recover the stateful operator, especially at higher input stream rates.

Next, we investigated the impact of checkpoint interval on recovery time under different input rates. Figure 13b shows that, for a shorter interval, a faster recovery time can be achieved. This can not avoid increasing some certain latency overhead, but it is acceptable since we have adopted a asynchronous incremental checkpointing referred to as none-blocking. However, recovery time increases with larger input rate because more events are replayed. In general, our approach guarantees a fast recovery from failure.

7.2.2.2 Under auto-scaling To verify our recovery effectiveness under auto-scaling, we compared the recovery time obtained with SC and R+SM, in which the checkpointing interval varied from 10 to 30 s. In the middle between two checkpoints, v_2 scaled up into two parallel tasks t_{21} and t_{22} as we increased the input rate (i.e., 2000 events/s) by two times. Then, we failed v_2 after its scale-up and measured the recovery time.

Figure 14 demonstrates the fast recovery achieved under auto-scaling. Owing to dynamic backup updates, we guarantee parallel recovery from the state after scale-up. While R+SM re-partitions the backup states with scale-up, its parallel recovery must start from the last checkpoint. The recovery latencies with SC are much higher, as it rolls back to the last checkpoint state, leading to the replay of more events with one operator instance during recovery. In particular, with a higher input rate, bad latencies may occur that can significantly dampen system performance if the operator scales up again. Compared to SC, our approach reduces the recovery time by up to 50%.

To sum up, we compare the comprehensive performance of different methods regarding to the runtime overhead and recovery time in Table 9. The results of runtime overhead are from the experiments of Sect. 7.2.1, which are based on the sequential line topology G1($\gamma = 1$) with a 30-s window. For the comparison of recovery time, the forks indicate that the method in the current row doesn't support the auto-scaling scenario. According to the comparison of various aspects, our method proposed in this paper has the best comprehensive performance in runtime overhead and recovery time.

8 Conclusion and future work

Focusing on elastic processing demand of EDSPSs, we propose a high-efficiency and flexible fault-tolerant mechanism for general EDSPSs, which enables online backup adjustment as an adaptation to operator auto-scaling, including data backup re-partitioning and dynamic checkpoint updating. In particular, the EDS is proposed as basic data backup unit that can be merged and split, which ensures recovery consistency under dynamic topology. Based on these attributes, our fault-tolerant protocols coordinate low-overhead backup, dynamic adjustment, and fast recovery from failures. We implemented a prototype system named SPATE, and the results of evaluating it show that our approach can be used effectively to provide fast recovery with low overhead. Comparing to existing works, our approach reduces the recovery time under scale-up by up to 50%. In the future, we plan to extend our fault-tolerant mechanism to tolerate multiple task failures with minimum overhead for ESPSs.

Acknowledgements This work was supported by the National Natural Science Foundation of China (NSFC) (Grant Nos. 61602205 and 61772228), the National Key Research and Development Program of China (Grant Nos. 2017YFC1502306, 2016YFB0201503 and 2016YFB0701101), the Major Special Research Project of Science and Technology Department of Jilin Province (20160203008GX), and the Jilin Scientific and Technological Development Program (20170520066JH).

References

1. Abadi, D.J., Carney, D., Çetintemel, U., Cherniack, M., Convey, C., Lee, S., Stonebraker, M., Tatbul, N., Zdonik, S.: Aurora: a new model and architecture for data stream management. VLDB J. **12**(2), 120–139 (2003)
2. Balazinska, M., Balakrishnan, H., Madden, S.R., Stonebraker, M.: Fault-tolerance in the borealis distributed stream processing system. ACM Trans. Database Syst. (TODS) **33**(1), 3 (2008)
3. Brito, A., Martin, A., Knauth, T., Creutz, S., Becker, D., Weigert, S., Fetzer, C.: Scalable and low-latency data processing with stream mapreduce. In: Proceedings of the IEEE Third International Conference on Cloud Computing Technology and Science (CloudCom), 2011 , IEEE, pp. 48–58 (2011)
4. Castro Fernandez, R., Migliavacca, M., Kalyvianaki, E., Pietzuch, P.: 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, ACM, pp. 725–736 (2013)
5. Chandrasekaran, S., Cooper, O., Deshpande, A., Franklin, M.J., Hellerstein, J.M., Hong, W., Krishnamurthy, S., Madden, S.R., Reiss, F., Shah, M.A.: Telegraphcq: continuous dataflow processing. In: Proceedings of the 2003 ACM SIGMOD

International Conference on Management of Data, ACM, pp. 668–668 (2003)
6. Chen, Q., Hsu, M., Malu, C.: Fault tolerant distributed stream processing based on backtracking. Int. J. Netw. Distrib. Comput. **1**(4), 226–238 (2013)
7. Cherniack, M., Balakrishnan, H., Balazinska, M., Carney, D., Cetintemel, U., Xing, Y., Zdonik, S.B.: Scalable distributed stream processing. CIDR **3**, 257–268 (2003)
8. de Assuncao, M.D., da Silva, A., Buyya, R.: Distributed data stream processing and edge computing: a survey on resource elasticity and future directions. J. Netw. Comput. Appl. **103**, 1–17 (2018)
9. De Matteis, T., Mencagli, G.: Elastic scaling for distributed latency-sensitive data stream operators. In: Proccedings of the 2017 25th Euromicro International Conference on Parallel, Distributed and Network-based Processing (PDP), IEEE, pp. 61–68 (2017)
10. Dean, J., Ghemawat, S.: Mapreduce: simplified data processing on large clusters. Commun. ACM **51**(1), 107–113 (2008)
11. Gedik, B., Schneider, S., Hirzel, M., Wu, K.L.: Elastic scaling for data stream processing. Parallel Distrib. Syst. IEEE Trans. **25**(25), 1447–1463 (2014)
12. Gu, Y., Zhang, Z., Ye, F., Yang, H., Kim, M., Lei, H., Liu, Z.: An empirical study of high availability in stream processing systems. In: Proceedings of the 10th ACM/IFIP/USENIX International Conference on Middleware, Springer-Verlag New York, Inc., p. 23 (2009)
13. Gulisano, V., Jimenez-Peris, R., Patino-Martinez, M., Soriente, C., Valduriez, P.: Streamcloud: an elastic and scalable data streaming system. Parallel Distrib. Syst. IEEE Trans. **23**(12), 2351–2365 (2012)
14. He, B., Yang, M., Guo, Z., Chen, R., Su, B., Lin, W., Zhou, L.: Comet: batched stream processing for data intensive distributed computing. In: Proceedings of the 1st ACM Symposium on Cloud Computing, ACM, pp. 63–74 (2010)
15. He, F., Wei, P.: Research on comprehensive point of interest (poi) recommendation based on spark. Clust. Comput. (2018). https://doi.org/10.1007/s10586-018-2061-y
16. Heinze, T., Zia, M., Krahn, R., Jerzak, Z., Fetzer, C.: An adaptive replication scheme for elastic data stream processing systems. In: Proceedings of the 9th ACM International Conference on Distributed Event-Based Systems, ACM, pp. 150–161 (2015)
17. Hunt, P., Konar, M., Junqueira, F.P., Reed, B.: Zookeeper: Wait-free coordination for internet-scale systems. In: USENIX Annual Technical Conference, Boston, MA, USA, vol. 8 (2010)
18. Hwang, J.H., Balazinska, M., Rasin, A., Çetintemel, U., Stonebraker, M., Zdonik, S.: High-availability algorithms for distributed stream processing. In: Proceedings of the 21st International Conference on Data Engineering. ICDE 2005, IEEE, pp. 779–790 (2005)
19. Imai, S., Patterson, S., Varela, C.A.: Uncertainty-aware elastic virtual machine scheduling for stream processing systems. In: 2018 18th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGRID), IEEE, pp. 62–71 (2018)
20. Javed, M.H., Lu, X., Panda, D.K.: Cutting the tail: designing high performance message brokers to reduce tail latencies in stream processing. In: 2018 IEEE International Conference on Cluster Computing (CLUSTER), IEEE, pp. 223–233 (2018)
21. Koldehofe, B., Mayer, R., Ramachandran, U., Rothermel, K., Völz, M.: Rollback-recovery without checkpoints in distributed event processing systems. In: Proceedings of the 7th ACM International Conference on Distributed Event-Based Systems, ACM, pp. 27–38 (2013)
22. Li, H., Wu, J., Jiang, Z., Li, X., Wei, X.: Minimum backups for stream processing with recovery latency guarantees. IEEE Trans. Reliab. **PP**(99), 1–12 (2017)
23. Li, M., Tan, J., Wang, Y., Zhang, L., Salapura, V.: Sparkbench: a spark benchmarking suite characterizing large-scale in-memory data analytics. Clust. Comput. **20**(3), 2575–2589 (2017)
24. Liu Z., Huang H., He Q., Chiew K., Gao Y.: Rare category exploration on llnear time complexity. In: Renz M., Shahabi C., Zhou X., Cheema M. (eds) Database Systems for Advanced Applications. DASFAA 2015. Lecture Notes in Computer Science, vol. 9050, pp. 37–54. Springer, Cham (2015). https://doi.org/10.1007/978-3-319-18123-3_3
25. Lohrmann, B., Janacik, P., Kao, O.: Elastic stream processing with latency guarantees. In: IEEE International Conference on Distributed Computing Systems, pp. 399–410 (2015)
26. Lombardi, F., Aniello, L., Bonomi, S., Querzoni, L.: Elastic symbiotic scaling of operators and resources in stream processing systems. IEEE Trans. Parallel Distrib. Syst. **29**(3), 572–585 (2018)
27. Martin, A., Fetzer, C., Brito, A.: Active replication at (almost) no cost. In: 2011 30th IEEE Symposium on Reliable Distributed Systems (SRDS), IEEE, pp. 21–30 (2011)
28. Marz, N.: Storm: distributed and fault-tolerant realtime computation (2013)
29. Mencagli, G., Torquati, M., Danelutto, M.: Elastic-ppq: a two-level autonomic system for spatial preference query processing over dynamic data streams. Future Gener. Comput. Syst. **79**, 862–877 (2018)
30. Neumeyer, L., Robbins, B., Nair, A., Kesari, A.: S4: Distributed stream computing platform. In: 2010 IEEE International Conference on Data Mining Workshops (ICDMW), IEEE, pp. 170–177 (2010)
31. Qian, Z., He, Y., Su, C., Wu, Z., Zhu, H., Zhang, T., Zhou, L., Yu, Y., Zhang, Z.: Timestream: reliable stream computation in the cloud. In: Proceedings of the 8th ACM European Conference on Computer Systems, ACM, pp. 1–14 (2013)
32. Sîrbu, A., Babaoglu, O.: Towards operator-less data centers through data-driven, predictive, proactive autonomics. Clust. Comput. **19**(2), 865–878 (2016)
33. Sumalatha, M., Ananthi, M.: Efficient data retrieval using adaptive clustered indexing for continuous queries over streaming data. Clust. Comput. (2017). https://doi.org/10.1007/s10586-017-1093-z
34. Wang, H., Peh, L.S., Koukoumidis, E., Tao, S., Chan, M.C.: Meteor shower: a reliable stream processing system for commodity data centers. In: 2012 IEEE 26th International on Parallel & Distributed Processing Symposium (IPDPS), IEEE, pp. 1180–1191 (2012)
35. Wei, X., Xiang, L., Hongliang, L., Cong, L., Yuan, Z.: Flexible online mapreduce model and topology protocols supporting large-scale stream data processing. J. Jilin Univ. (Eng. Technol. Edn.) **46**(4), 1222–1231 (2016)
36. Wei, X., Li, L., Li, X., Wang, X., Gao, S., Li, H.: Pec: proactive elastic collaborative resourcescheduling in data stream processing. In: Proceedings of the IEEE Transactions on Parallel and Distributed Systems (2019)
37. Wu, Y., Tan, K.L.: Chronostream: elastic stateful stream computation in the cloud. In: IEEE International Conference on Data Engineering, pp. 723–734 (2015)
38. Zaharia, M., Das, T., Li, H., Hunter, T., Shenker, S., Stoica, I.: Discretized streams: a fault-tolerant model for scalable stream processing. Technical Report, DTIC Document (2012)
39. Zhang, Z., Gu, Y., Ye, F., Yang, H., Kim, M., Lei, H., Liu, Z.: A hybrid approach to high availability in stream processing systems. In: 2010 IEEE 30th International Conference on Distributed Computing Systems (ICDCS), IEEE, pp. 138–148 (2010)

Xiaohui Wei is a Professor and Dean of the College of Computer Science and Technology (CCST), Jilin University, China. He is currently the Director of High Performance Computing Center of JilinUniveristy. His currentmajor research interests includes resource scheduling for large distributed systems, infrastructure level virtualization, largescale data processing system and fault-tolerant computing. He has published more than 50 journal and conference papers in the above areas.

Yuan Zhuang is a doctoral student in the College of Computer Science and Technology (CCST), Jilin University, China. Her research interests include HPC for big data processing and fault tolerance in distributed stream processing systems.

Hongliang Li received his Ph.D. from the College of Computer Science and Technology (CCST), Jilin University, China. He is currently an associate professor of CCST. His research interests include resource scheduling and fault tolerance in HPC systems. He is working on a China NFS project on elastic virtual cluster and parallel job scheduling under virtual HPC environment.

Zhiliang Liu is a master and received her bachelor's degree from the College of Computer Science and Technology (CCST), Jilin University, China, in 2017. His research interests include fault tolerance and checkpoint scheduling in stream processing systems.