This article has been accepted for inclusion in a future issue of this journal. Content is final as presented, with the exception of pagination.

Minimum Backups for Stream Processing With Recovery Latency Guarantees

Hongliang Li, *Member, IEEE*, Jie Wu, *Fellow, IEEE*, Zhen Jiang, *Member, IEEE*, Xiang Li, and Xiaohui Wei, *Member, IEEE*

***Abstract*—The stream processing model continuously processes online data in an on-pass fashion that can be more vulnerable to failures than other big-data processing schemes. Existing fault-tolerant (FT) approaches have been presented to enhance the reliability of stream processing systems. However, the fundamental tradeoff between recovery latency and FT overhead is still unclear, so these scheme cannot provide recovery latency guarantees. This paper introduces the FT Configuration (FTC) problem and presents a solution for guaranteed recovery latency with minimum backups. A failure effect model is presented to describe the relationship between recovery latency and FTC (the amount and locations of backups). With this model, we design an algorithm to compute FTCs for different types of *stream topologies* according to recovery latency requirements. Extensive experiments are conducted to verify the correctness and effectiveness of our approach. We prove that our algorithm guarantees recovery latencies for all directed acyclic graph (DAG) stream topologies. For *line(s)* and *tree* topologies, our algorithm solves the FTC problem with a time complexity of $O(N)$. For a general DAG topology, a heuristic function is used to generate FTCs. This causes fewer than 10% more backups on average compared to the optimal solution with a time complexity of $O(N^2)$.**

***Index Terms*—Distributed system, fault-tolerant (FT), recovery latency, stream processing, upstream backup.**

Manuscript received January 30, 2017; revised April 19, 2017; accepted June 1, 2017. This work was supported in part by the National Natural Science Foundation of China (NSFC) under Grant 61602205, in part by the National Key Research and Development Plan of China under Grant 2016YFB0201503, in part by the Provincial Key Research Project on Science and Technology of Jilin under Grant 20160203008GX, in part by the China Scholarship Council, and in part by the NSF of the U.S. under Grant CNS 1629746, Grant CNS 1564128, Grant CNS 149860, Grant CNS 1461932, Grant CNS 1460971, Grant CNS 1439672, Grant CNS 1301774, and Grant ECCS 1231461. Associate Editor: S.-Y. Hsieh. (*Corresponding author: Hongliang Li.*)

H. Li is with the College of Computer Science and Technology, Jilin University, Changchun 130022, China, with the Key Laboratory of Symbolic Computation and Knowledge Engineering of the Ministry of Education, Changchun 130012, China, and also with the Department of Computer and Information Sciences, Temple University, Philadelphia, PA 19122 USA (e-mail: lihongliang@jlu.edu.cn).

J. Wu are with the Department of Computer and Information Sciences, Temple University, Philadelphia, PA 19122 USA (e-mail: jiewu@temple.edu).

X. Li and X. Wei are with the College of Computer Science and Technology, Jilin University, Changchun 130022, China, and also with the Key Laboratory of Symbolic Computation and Knowledge Engineering of the Ministry of Education, Changchun 130012, China (e-mail: lxiang@jlu.edu.cn; weixh@jlu.edu.cn).

Z. Jiang is with the Department of Computer Science, West Chester University of Pennsylvania, West Chester, PA 19382 USA (e-mail: zjiang@wcupa.edu).

Color versions of one or more of the figures in this paper are available online at http://ieeexplore.ieee.org.

Digital Object Identifier 10.1109/TR.2017.2712563

Acronyms

FTC	Fault-tolerant configuration.
f-cover	forward cover.
b-cover	backward cover.

Notations

$G(V,E)$	The DAG that represents a stream topology.
N	Amount of tasks in a stream topology, $N = \|V\|$.
$T(v)$	Reprocessing latency of $v \in V$.
$U(v)$	Upstream latency of $v \in V$.
$L(v)$	Task recovery latency of $v \in V$.
O	Recovery latency of a stream topology.
P_v	The set of adjacent upstream tasks of task v, $v \in V$, $P_v \subset V$.
Q_v	The set of adjacent downstream tasks of task v, $v \in V$, $Q_v \subset V$.
V_s	The set of *source* of a stream topology.
V_t	The set of *sink* of a stream topology.
$D(v_1, v_2)$	*Distance* between two *tasks* v_1 and v_2.
R	Recovery latency requirement.
FTC(G,R)	FT configuration of stream topology G under recovery constraint R, $\{M(v_1),..,M(v_N)\}$, $M(v)=\{0,1\}$.
V_b	Set of *backup tasks* of a stream topology, $V_b \subseteq V$.
M	Amount of *backup tasks*, $M = \|V_b\| = \sum_{v\in V} M(v)$.
C_v	A set of all adjacent upstream backup tasks of task v, $v \in V$.
$I(v)$	Task index according Breadth-first Search.
$F(v)$	The percentage of workload a candidate backup task can b-cover.
$B(v)$	The combined workload a candidate backup task can b-cover.

I. Introduction

Recently, a new class of "Big Data" applications has been widely recognized: applications that demand that large-scale data streams be processed and analyzed in real time or near real time. Examples can be found in various areas, including trading high-frequency stocks, monitoring and controlling production lines, and detecting and managing traffic congestion. These applications take one or more data streams as input, perform a series of predefined functions, and generate output in the form of data streams again. They share a common

0018-9529 © 2017 IEEE. Personal use is permitted, but republication/redistribution requires IEEE permission.
See http://www.ieee.org/publications_standards/publications/rights/index.html for more information.

This article has been accepted for inclusion in a future issue of this journal. Content is final as presented, with the exception of pagination.

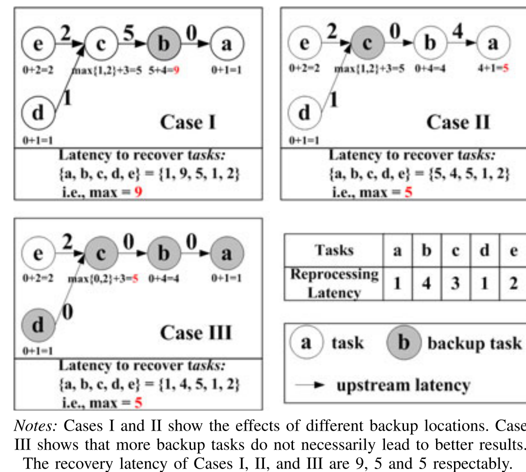
characteristic: Strict processing latency constraints, i.e., their processing rates must match the data arrival rate to provide a fast and accurate response. This leads to an increase in the popularity of a new computing paradigm known as the Stream Processing Model (SPM) [1]–[3].

Over the last decade, many approaches have been proposed and various stream processing systems have been implemented in both academia and industry [2], [4]–[8]. In these systems, a stream processing application is usually modeled as a directed acyclic graph (DAG), i.e., *Stream Topology* [9], which consists of a number of interconnected *tasks* (also denoted as *processing elements* in the literature). Each task consumes data from *upstream* task(s), processes the data, and emits results as data stream(s) to *downstream* task(s) [10]. Data streams arrive continuously and are usually considered infinite. The inability to obtain complete data beforehand has led to a computing paradigm completely different from the traditional "process-after-store" mode. The SPM performs "one-pass" processing over data on the fly before or even without storing the data.

This unique characteristic poses a novel fault-tolerant (FT) problem. It is one of the fundamental open challenges of the SPM [1], [11]. On one hand, failures in stream processing systems can cause severe damages, such as data loss and inaccurate or incorrect results, and the SPM can be more vulnerable to failures than other big-data processing schemes. On the other hand, the stream processing systems that have been implemented over distributed systems and deployed on data centers and cloud centers, known as distributed stream processing systems (DSPS) [6], [7], [10], have achieved higher processing throughput and scalability but increased the failure rates at the same time [12].

Although FT and reliability in distributed systems [13] is a classic problem that has been widely studied, SPM introduces new challenges, like strict recovery latency constraints and complex recovery dependencies. Existing FT approaches enable DSPS to recover from host-level failures with a short recovery latency [14]–[17]. Replication-based approaches [18], [19] maintain a backup instance for each primal instance [20] and introduce almost no recovery latencies. However, they suffer from considerable FT overhead. In recent years, the upstream backup model [10], [15], [16], [21] has been widely applied due to its short recovery latency and reasonable FT overhead. In the upstream backup model, each task can maintain a backup of its output data for its downstream tasks. The tasks that perform backups are called *backup tasks*. Although extensive results have been presented in the related literature to illustrate the performance of different upstream backup approaches, to the best of our knowledge, none provides recovery latency guarantees. The relationship between recovery latency and FT overhead (i.e., the amount of upstream backups) is still unclear.

There are recovery dependencies among a recovering task and its dependent upstream backup task(s). The recovery latency of a recovering task is closely related to the *backup configuration* (the amount and location of backup tasks) of a stream topology. When failure occurs, the recovery latency of the stream topology is equal to the largest task recovery latency. The recovery latency of a task means the time consumed from when it is restarted to when it is recovered back to its prefailure status, which consists of two parts: 1) *upstream latency*, the time used to obtain recent



Notes: Cases I and II show the effects of different backup locations. Case III shows that more backup tasks do not necessarily lead to better results. The recovery latency of Cases I, II, and III are 9, 5 and 5 respectably.

Fig. 1. Example of recovery latencies of the same stream topology under different FTCs.

backup data from its adjacent upstream task(s); and 2) *reprocess latency*, the time to recover a task to its latest state. Given the failure rate and backup interval, the reprocessing latency can be estimated [10], [22], [23]. A task's upstream latency depends on the location of the dependent backup tasks. If an adjacent upstream task is not a backup task, it has to start its own recovery procedure recursively to regenerate the data needed. This causes extra latencies. For example, in Fig. 1 Case II, task *b* will be involved in the recovery process of task *a*.

Setting all tasks as backup tasks [10], [21] ensures that any recovering task's adjacent upstream task(s) is(are) its dependent backup task(s) and that all required backup data are available right after task restarting. This introduces zero upstream latencies and minimizes the recovery latency. However, maintaining backup data introduces overhead in runtime. Even though the overhead can be reduced by trimming the backup data queue with checkpointing [5], [7], upstream backups could still bring considerable runtime overheads to both space and time [15], [16]. Decreasing the number of backup tasks reduces FT overhead. Therefore, the other extreme is to perform backup only on the input streams [6], [15] to minimize backup overhead. In this method, all upstream tasks will be involved recursively to regenerate the needed data and introduce maximum extra latencies. Both the number of backups (FT overhead) and the location of backups affect the recovery latency. As shown in Fig. 1, different configurations with the same number of backups may lead to different recovery latencies (Case I and Case II), and more backups do not necessarily reduce recovery latency (Case II and Case III).

It is straightforward to ask what is the relationship between FT configuration (FTC) (the amount and location of backups) and recovery latency. Given a stream topology [9], how can we setup the FTC to minimize backups and guarantee recovery latency at the same time (defined as FTC problem in Section III-B)? This is a very practical question that does not appear to have been addressed in previous literature. The purpose of this paper is to introduce a model that describes the tradeoff between FTC and recovery latency and to present an approach to compute FTC with minimum backups (FT overhead) according to recovery

This article has been accepted for inclusion in a future issue of this journal. Content is final as presented, with the exception of pagination.

latency requirements. The main contributions are summarized as follows.

1) We present a quantitative failure effect model that describes the relationship between the *recovery latency* and the FTC of a stream topology. We introduce the FTC problem based on this model.
2) We propose an algorithm for different types of stream topologies: sequential/parallel *lines*, *tree*, and general DAG. We prove that our algorithm keeps recovery latency guarantees for all DAG stream topologies.
3) We conduct extensive simulations to verify the correctness and effectiveness of our approach with different applications and setups.

The rest of this paper is organized as follows. In the next section, we summarize related works. Section III presents the problem modeling and analysis. We propose and analyze our approach in Section IV, and Section V discusses the experimental results. Section VI concludes this paper.

II. Related Work

A. Reliable Stream Processing

Active replication and checkpoint/recovery are two widely studied traditional FT mechanisms, and both have applications in distributed stream processing systems. Active replication maintains at least one active replica instance to enable instant switch from the primary instance to its replication when failure occurs. This ensures a minimum response time, but suffers from a high overhead that at least doubles resource consumption. It is applied in earlier stream processing systems and in data engines [1], [18] hosted by a small cluster of machines. With the scale of applications increasing rapidly [8], the active replication model becomes inefficient or even impractical for DSPS, which is why most recent research explores FT approaches based on checkpoint/recovery [5], [19], [21], [24].

Hwang *et al.* [10] introduced an *upstream backup* model that takes advantage of close upstream–downstream dependencies. Upstream tasks keep output buffer as a backup for a downstream task for as long as necessary. If a downstream task fails, backup data are replayed to restore the latest running state and to generate correct results. This is an efficient approach to the stream processing model, but it only supports applications that depend on recent data, not those that depend on a complete history of previous data. Therefore, recent works combine upstream backup with checkpoint/recovery to solve this problem [15], [19], [21]. The runtime status, including the accumulated results, of a task is stored in checkpoint files and restored after failure. Checkpointing events are used to trim upstream backup data. This approach achieves short recovery latency and has become the most commonly used FT method for SPM. However, maintaining upstream backups may still introduce considerable FT overhead (both computing time and memory space) [15], [16].

B. FT Strategy

Recent works have begun to pay attention to the tradeoff between FT overhead and recovery latency. Some of the works are based on hybrid FT schemes, such as active replication and checkpoint/recovery [16], [17], [19], [25], and they adaptively choose different FT strategies according to application types and/or recovery time thresholds. Martin *et al.* [25] used spare resources to run active replicas of operators. When the system is under heavy workloads, operators switch to passive replicas. Other works present optimization models to choose between different FT schemes. FTOps [17] selects the optimal fault tolerance mechanism in offline fashion, while [19] makes adaptive decisions based on online estimations of processing performance. FTOps only supports tree-structured jobs, not DAG-structured plans, which narrows its applications. Heinze *et al.* [19] minimized recovery time violations, but the recovery latency is not guaranteed. Su *et al.* [16] model correlated failures and make optimized plans for replications to avoid tentative outputs.

Apart from FT strategies for stream processing systems, related works study the FT strategies for workflow applications [23], [26]. Salama *et al.* [23] presented a materialization configuration for workflow applications with the objective of minimizing overall execution time. Like our work, [23] makes decisions about the amount and the location of materialization tasks. However, their application backgrounds and objectives are different. In the stream processing model, all tasks execute concurrently as a pipeline. In the workflow model, tasks are executed step-by-step and only tasks without upstream-downstream dependencies can execute concurrently. Therefore, the FT strategy for the stream processing model focuses on reducing the maximum task latency to avoid performance bottlenecks instead of on reducing the overall latency of the entire workflow.

C. Processing Latency and Recovery Latency Modeling

Chain [27] is one of the earliest works focusing on the modeling of processing latency and task allocation strategies for stream processing systems. It presents an approach to assign resources to tasks with the objective of minimizing the makespan of a data flow job in the single machine environment. In recent years, the task allocation problem for DSPS has been widely studied [9], [28], [29]. Studies seek task allocation plans that avoid processing latency bottleneck. These works use similar models for processing latency and stream topology in the SPM, and they provide the background for our work. Eidenbenz *et al.* [9] presented strong theoretical results for a common type of stream topology (i.e., SPD). They propose solutions for computing the optimal resource shares of stream processing tasks under a continuous resource partitioning scenario. This work differs from our work because its objective is to minimize the overall processing latency. Optimal resource shares can be used as a guide for placing backup resources, and therefore, we use it as a comparison method in our simulations.

The recovery latency of a stream processing task is related to multiple parameters such as state size, queue length, window size, and checkpoint intervals [19]. There are two approaches to estimate recovery latency, experimental methods, and theoretical methods. Heinze *et al.* [19] designed a clustering method based on historical samples to estimate recovery time. Salama *et al.* [23] used the reliability model [22] to estimate the recovery latency assuming that the checkpoint interval and the failure

This article has been accepted for inclusion in a future issue of this journal. Content is final as presented, with the exception of pagination.

rates of individual tasks are given. This work provides the background of our failure effect model. The reprocessing latency of a recovering stream processing task can be estimated based on the same method. Note that this method estimate the reprocessing latencies of individual tasks independently, and it supports tasks with nonidentical failure rates. However, these methods are not suitable for the stream processing model when tasks have recovery dependencies with their upstream backups. In this paper, we propose a failure effect model based on the dependencies between recovering tasks and their dependent backup tasks.

III. Problem Formulation and Analysis

A. Stream Processing Model

A *stream topology* is modeled as a directed acyclic graph (DAG) [9] $G = (\mathrm{V}, \mathrm{E})$, where $v \in V$ and $e \in E$ represent a task and a directed link between two adjacent upstream and downstream tasks [10], respectively. The number of tasks in a stream topology is $|V| = N$. Each task performs a predefined function on the data from the input stream(s) and emits data to output stream(s). A task without any incoming link is called *source*, and one without any outgoing link is called *sink*. There could be multiple *sources* and *sinks* in a stream topology. V_s and V_t denote sets of *source* and *sink*, respectively.

A task state consists of a data state and an internal state [5], [15]. There are two types of tasks, known as stateful tasks (e.g., join and aggregate), which have both states, and stateless tasks (e.g., map and filter), which have only a data state [21]. We assume each task is able to perform upstream backup [10] and that *upstream task* will backup output data for *downstream task* for as long as necessary in case *downstream task* needs to recover from failure. The tasks that are chosen to perform upstream backup are called *backup tasks*, denoted by set ($V_b \subseteq V$). Next, we give the definition of FTC.

Definition 1: FTC of a stream topology $G(V, E)$ is a set $\{M(v_1), .., M(v_n)\}$, where $v \in V$, $n = |V|$ and $M(v) = \{0, 1\}$. $M(v) = 1$ when $v \in V_b$, and $M(v) = 0$ otherwise.

We also assume that stateful tasks perform periodical checkpointing [21], [30] to preserve internal states and trim backup data. The data state of a stream task is usually much larger than the internal state [15], and a great proportion of stream processing tasks are stateless tasks that do not have internal states. Therefore, we only consider data state in our model, and we assume that the FT overhead of a stream topology is reflected by the amount of backup tasks ($M = \sum_{v \in V} M(v)$). When a task failure occurs, the task is revived to its latest materialized state using recent checkpoint file. Then, the recovering task fetches the needed backup data from upstream backup task(s) and reprocesses them to get back to the latest state before the failure.

B. Failure Effect Model

We present a failure effect model to describe the relationship between the FTC (the amount and location of backups) and performance (recovery latency) of a stream topology. In this paper, we consider task failures that may be caused by various reasons, such as processor failure, network failure, and software malfunction. When a task failure occurs, a restarted task seeks previous input data from all of its adjacent upstream task(s), and then, it reprocesses them to recover its latest state. A task's recovery latency is denoted as $L(v)$ and consists of two parts: 1) upstream latency ($U(v)$), the time it takes to obtain previous data from adjacent upstream tasks and 2) reprocessing latency ($T(v)$), the time it takes to reprocess data from the last checkpoint. Let R denote the recovery latency requirement of a stream topology. R is satisfied when $\forall v \in V, L(v) \leq R$:

$$L(v) := U(v) + T(v). \tag{1}$$

A cost function $T : V \rightarrow \mathbb{R}^+$ determines the time needed for a task to reprocess data during failure recovery. Next, we discuss how to estimate the reprocessing latency $T(v)$ in practice. In this paper, we assume tasks have independent failure rates. The reprocessing latencies of individual tasks can be estimated independently based on the classic FT theory for sequential application [22]. Given the failure rate of a task, the optimal checkpoint interval can be computed based on the method in [22]. Let $T(v)^{\text{ckpt}}$ denotes the checkpoint interval of task v. Furthermore, the reprocessing latency of a task can be estimated based on its checkpointing interval ($T(v) \approx 1/2 \cdot T(v)^{\text{ckpt}}$) [10], [22], [23]. Note that tasks can have nonidentical failure rates in our failure effect model.

Based on the aforementioned discussion, we assume without loss of generality that for all tasks $v \in V$, $T(v)$ is given as an input. Next, we show how to model the upstream latency, i.e., the time needed to retrieve all backup data from upstream task(s). We give a recursive version based on *upstream backup*[10], [21] and then present a nonrecursive version (used in our algorithm) in Section IV.

If task $v \in V$ is recovering, P_v is a set of its adjacent upstream tasks, and $P_v \subseteq V$, then for each upstream task $u \in P_v$, $U(u, v)$ denotes the upstream latency from u to v:

$$U(u, v) := \begin{cases} 0, & \text{if } M(u) = 1 \\ L(u), & \text{otherwise.} \end{cases} \tag{2}$$

As shown in (2), if an upstream task u is a backup task (i.e., $M(u) = 1$), then it is able to provide the data v needs right away (i.e., $U(u, v) = 0$). Otherwise, it has to engage its own recovery process to regenerate data for v so that $U(u, v) = L(u)$. Since all tasks execute concurrently in SPM [9], $U(v) = \max_{u \in P_v} \{U(u, v)\}$. The recovery latency of a task can be formulated recursively as

$$L(v) := \max_{u \in P_v} \{(1 - M(u))L(u)\} + T(v). \tag{3}$$

A backup task preserves results from upstream tasks and provides backup data for downstream tasks. Next, we give the definition of *forward cover* and of *backward cover*, hereafter referred as *f-cover* and *b-cover*, to help us with a nonrecursive model of task upstream latency.

Definition 2: Forward-cover (Backward-cover): We say a backup task u *f-covers* (*b-covers*) task v and v is *f-covered* (*b-covered*) by u, denoted by $u \underset{fc}{\rightarrow} v$ ($v \underset{bc}{\leftarrow} u$), when $u \in V$, $v \in V$, and u is an adjacent upstream (downstream) backup task for v,

This article has been accepted for inclusion in a future issue of this journal. Content is final as presented, with the exception of pagination.

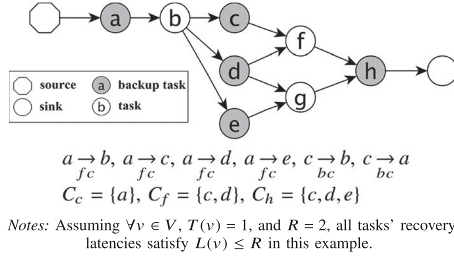


Fig. 2. Example of forward-cover (backward-cover) and upstream cover set.

e.g., there is no other backup task on the path from u to v, that satisfies recovery latency requirement $L(v) \leq R$.

Definition 3: Upstream Cover Set of task v is a set of all upstream tasks that f-cover v, denoted by C_v ($C_v \subseteq V$) and $\forall u \in C_v, u \underset{fc}{\rightarrow} v$.

As exampled in Fig. 2, one backup task can f-cover (b-cover) multiple tasks (task a), and a task may depend on multiple backup tasks in its recovery process (task h). According to the definition, a task's upstream cover (UC) set contains sufficient backups to provide all data needed in a task failure recovery. Let us say $u \in C_v$, and a set of all nonbackup tasks on a path from u to v (including v) is denoted by $J_{(u,v)}$. The *distance* between u and v is the largest latency among all possible paths from u to v and is denoted by $D(u, v) = \max\{\sum_{z \in J_{(u,v)}} T(z)\}$. Therefore, the recovery latency of a task v equals the largest distance from all backup tasks in its UC set C_v:

$$L(v) = U(v) + T(v) = \max_{u \in C_v}\{D(u, v)\}. \tag{4}$$

C. FTC Problem

We define the FTC problem based on the failure effect model. We assume all tasks have independent failure rates, and our goal is to tolerate a single failure at one time. The overall recovery latency of a stream topology is equal to the largest recovery latency of a task, i.e., $O = \max_{v \in V}\{L(v)\}$. Therefore, the recovery latency requirement is fulfilled when all tasks' recovery latencies are less than or equal to R.

Problem I (FTC): Given a stream topology graph $G(V, E)$ $(N = |V|)$ and the recovery latency requirement R, find the fault-tolerant configuration $FTC(G, R) = \{M(v_1), .., M(v_N)\}$ with minimum backups (M) that satisfies R:

$$\begin{aligned} &\text{minimize} \quad M := \sum_{v \in V} M(v) \\ &\text{subject to} \\ &\qquad\qquad M(v) := \{0, 1\} \quad v \in V \\ &\qquad\qquad L(v) \leq R \qquad\quad v \in V. \end{aligned} \tag{5}$$

D. Assumptions

In this paper, we make the following assumptions. First, we assume independent failure rates on each task. The reprocessing time of each task can be estimated as [10], [22], and [23], and are given as inputs. A scenario in which the failure rate of each task is not independent will be studied in our future work. Second, we make the assumption that there is, at most, one failed task for each stream topology at one time, which is realistic when tasks are distributed on different physical machines. Finally, we assume that the backup overhead of the data state dominates that of the internal state, i.e., the FT overhead of a stream topology is reflected by the amount of backup tasks. This assumption is discussed in Section III-A.

IV. ALGORITHM

In this section, we propose an algorithm ($algorithm^{FTC}$) to solve the FTC problem for different types of stream topologies: *line(s) topology*, *tree topology*, and *general DAG*. We show how $algorithm^{FTC}$ computes optimal FTCs for the first two types of topologies with a complexity of O(N). Then, we present a general version of the algorithm that uses a heuristic ranking method to compute FTCs for all DAG with a complexity of ($\mathrm{O}(\mathrm{N}^2)$).

A. Main Algorithm

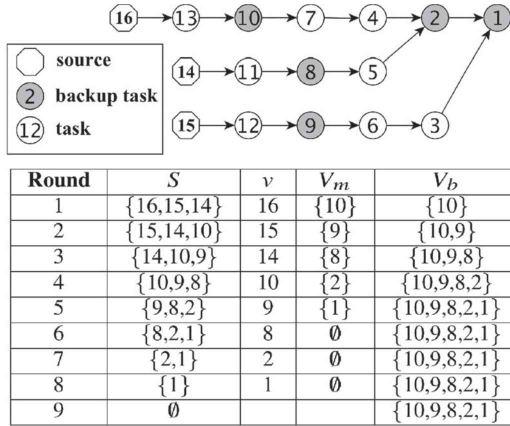
For any input stream topology $G(V, E)$ we assume that tasks are indexed according to their $Breadth_First_Search$ orders. If we add one virtual sink to connect all sinks when G has multiple sinks, then we can perform a breadth-first search from the sole sink as *root* and index the tasks from 1 to N (virtual sink is indexed 0) according to the search orders. After the traversal, tasks with more hops from *root* have a larger index. We refer to the *index* of a task as $I(v)$. Tasks are in descending orders according to their indexes in $V = \{v_1, .., v_N\}$.

$Algorithm^{FTC}$ scans all tasks in descending order according to topological indexes. This is based on the fact that in order to achieve the FT of a source, the input stream must always be backuped, e.g., using Apache Kafuka [31]. Next the algorithm maintains a set of pending search sources, denoted as S, initialized with all source of the stream topology. Then, it launches a round of search starting from each task in S to set up new backup task(s). At each round, it chooses a task from S with the largest index ($I(v)$), removes it from S, scans a candidate set of downstream tasks to choose new backup task(s), and put the new backup task(s) into S. V_m contains tasks that are chosen to be backup tasks in the current search round. At the end of each round, V_m will be merged into S. The algorithm ends when the pending search set is empty. Since at least one backup will be set up each round, $Algorithm^{FTC}$ performs, at most, M rounds of searches. It is easy to prove that all tasks are scanned at least once in this process.

In each round, function $ChooseBackup()$ is called to choose new backup(s). New backups must satisfy the following constraints.

1) *Rule I*—The new backup task is *f-covered* (*definition*. 2) by the current searching source.
2) *Rule II*—The processing result of the current searching source is backuped by all new backup task(s).

This article has been accepted for inclusion in a future issue of this journal. Content is final as presented, with the exception of pagination.



Round	S	v	V_m	V_b
1	{16,15,14}	16	{10}	{10}
2	{15,14,10}	15	{9}	{10,9}
3	{14,10,9}	14	{8}	{10,9,8}
4	{10,9,8}	10	{2}	{10,9,8,2}
5	{9,8,2}	9	{1}	{10,9,8,2,1}
6	{8,2,1}	8	∅	{10,9,8,2,1}
7	{2,1}	2	∅	{10,9,8,2,1}
8	{1}	1	∅	{10,9,8,2,1}
9	∅			{10,9,8,2,1}

Notes: Tasks are indexed according to the breadth-first search orders using the sink as root. $V = \{1, 2, 3, .., 16\}$, assume $\forall v \in V, T(v) = 1$, and $R = 2$.

Fig. 3. Example of applying $Algorithm^{FTC}$ to line(s) topology.

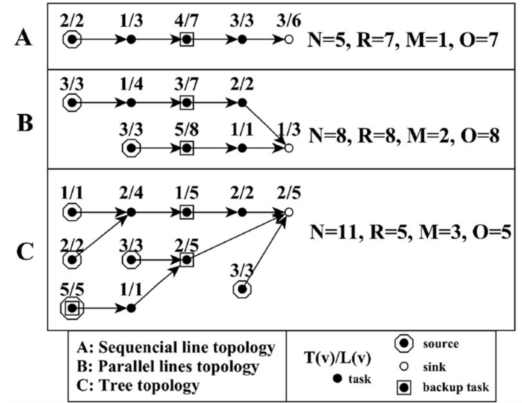


Fig. 4. FTC examples of line(s) and tree topology.

Algorithm 1: $Algorithm^{FTC}$ (G, R).

Input: A stream topology graph $G(V, E)$ and recovery latency requirement R
Output: FTC set V_b
1: Initialize N, $M(v)$ and searching source $S := S \cup V_s$;
2: **while** ($S \neq \emptyset$) **do**
3: $v := arc\max_{v \in S}\{I(v)\}$; //choose current searching source
4: $V_m := ChooseBackup(v,\ V,\ R,\ V_b)$; //choose new backup tasks
5: update V_b and S;
6: **end while**
7: **return** V_b;

Algorithm 2: ChooseBackup-Line(s)Topology (v, V, R, V_b).

Input: Current *source* v, *BreadthFirstSearch* order searching array V, recovery latency requirement R and current backup tasks V_b
Output: Set of chosen backup tasks V_m
1: Initlize $V_m := \emptyset$; $v_{mark} := v$;
2: get v's adjacent downstream task u;
3: **while** ($R >= d(u, v) + T(u)$) **do**
4: $v_{mark} := u$;
5: get v's adjacent downstream task u;
6: **end while**
7: $V_m := V_m \cup \{v_{mark}\}$;
8: **return** V_m

3) *Rule III*—The combined workloads of the tasks *b-covered* by chosen backup tasks in current round are maximized ($\sum_{v \underset{bc}{\leftarrow} u, u \in V_m} T(v)$).

The first rule ensures that the chosen backup task and its upstream tasks from the searching source do not exceed the recovery latency requirement. The second rule states that the processing results of the current searching source must be fully backed up in their downstream backup tasks to prevent data loss. Finally, the last rule seeks to maximize the benefits of each backup tasks and to reduce the total number of backup tasks.

Next, we show how this algorithm computes the FTC for different stream topologies, which eventually follows the same ranking method (discussed in Section IV-D).

B. Line(s) Topology

The first type of stream topology is called *line(s) topology*, where each graph consists of either a sequential line of *series tasks* [9] connected one-by-one or multiple parallel lines joining a common sink, as illustrated in Figs. 3 and 4(a) and (b). This type of stream topology is common when multiple processing steps are connected in a sequence like in production lines.

Function *ChooseBackup-Line(s)Topology* outlines the process of setting up backup tasks for line(s) topologies. As illustrated in Fig. 3, tasks are indexed according to the breadth-first search orders. Tasks in pending searching sources S are in decreasing order according to task indices. $Algorithm^{FTC}$ calls *ChooseBackup-Line(s)Topology* for the first searching source in S at each round. *ChooseBackup-Line(s)Topology* chooses new backup tasks (V_m). The current FTC set V_b is then updated accordingly. New backup tasks from the current round are added into S as pending searching sources for future rounds. Finally, $Algorithm^{FTC}$ finishes when $S = \emptyset$.

In order to satisfy Rule I, a candidate task set is generated according to recovery latency R. The candidate set contains tasks that are f-covered by the current searching source. In both sequential and parallel line(s) topologies, each task has one adjacent downstream task. Any newly chosen backup tasks can b-cover all processing results from the current search source, thus satisfying Rule II. We apply an approach that simply chooses new backup tasks with the smallest indices among all candidate tasks to apply Rule III. Fig. 4 (a) and (b) provides more examples of when tasks are given nonidentical reprocessing weights.

Lemma 1: Algorithm *ComputeFTC* keeps recovery latency guarantees for *line(s)* stream topologies.

This article has been accepted for inclusion in a future issue of this journal. Content is final as presented, with the exception of pagination.

Algorithm 3: ChooseBackup-TreeTopology (v, V, R, V_b).

Input: Current *source* v, *BreadthFirstSearch* order searching array V, recovery latency requirement R and current backup tasks V_b
Output: Set of chosen backup tasks V_m

1: Initlize $V_m := \emptyset$; $r := T(v)$;
2: **if** $(M(v) == 1)$ **then**
3: $r := 0$;
4: **end if**
5: **while** $(r <= R)$ **do**
6: $v_{mark} := v$;
7: get v's adjacent downstream task u;
8: **if** $(M(v) == 1)$ **then**
9: $V_m := \emptyset$; //End current search
10: return;
11: **end if**
12: $r := r + T(u)$;
13: $v := u$;
14: **end while**
15: $V_m := V_m \cup \{v_{mark}\}$;
16: **return** V_m

C. Tree Topology

The second type of stream topology is called *tree topology*, where tasks are connected as a tree rooted by a common sink. In this topology, each task has only one adjacent downstream task, as illustrated in Fig. 4(c). This type of stream topology represents a type of application (e.g., monitoring applications) that aggregates information from multiple sources to generate results. Unlike line(s) topology, tree topology has multiple joint nodes that may introduce opportunities for sharing backup tasks. Therefore, the *ChooseBackup* function follows a similar routine as the line(s) topology, but finishes the search on an earlier path where a scanned task is already set up as new backup task. The function for tree topology is shown in *ChooseBackup-TreeTopology()*.

Lemma 2: Algorithm *ComputeFTC* keeps recovery latency guarantees for *tree* stream topologies.

D. General DAG Topology

In this section, we present a general version of *ChooseBackup* that computes FTCs for all types of stream topologies, which we call general DAG topologies. Note that general DAG topologies include line(s) and tree topologies. We first discuss how previous functions solve the FTC problem for line(s) and tree topologies. 1) Tasks in *line(s)* and *tree topologies* only have one adjacent downstream task. A downstream task is capable of b-covering current searching sources and satisfies Rule II all by itself. 2) In *line(s)* and *tree topologies*, all downstream tasks are connected as a sequence. The combined workloads a candidate task could cover increases linearly with its distance from current searching source. Therefore, candidate tasks of a longer distance should be chosen first according to Rule III. The *line(s)* and *tree topologies* use similar methods to rank candidates based on their distance

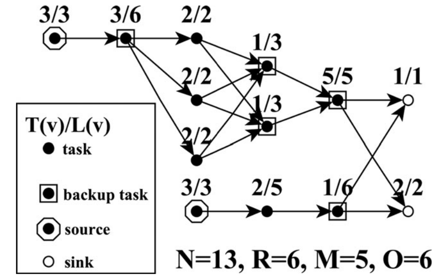


Fig. 5. FTC of general DAG topology.

from the current searching source (6). For the tree topology, if a current search task is marked a *backup task* (by previous searching tasks), its rank is set to a value big enough to guarantee that it will be picked. Otherwise, tasks are ranked in the same way as *line(s) topology*. As a result, the *line(s) topology* is a special case of *tree topology*:

$$\underset{v \in V, u \underset{c}{\rightarrow} v}{\text{rank}(v)} := \begin{cases} D(u,v), & \text{line}(s) \\ D(u,v) + (1 - M(v)) \cdot R & \text{tree.} \end{cases} \tag{6}$$

Unlike line(s) and tree topologies, some stream topologies contains tasks with multiple upstream/downstream neighbors, which we call general *DAG stream topologies*. This type of topology is also practical in the stream processing model, e.g., a stream topology running in DSPS supporting autoscale [21], [32] technology, as illustrated in Fig. 5.

We present a heuristic function to compute the FTC for the general DAG topology. First, function *GenerateCandidates()* generates a candidate set Z that contains all possible backup tasks for this round. All downstream tasks within a distance of R from the current source are in Z. This process requires a traversal of all the tasks in the worst case. Second, we compute the b-cover percentage $F(v)$ (7) of each candidate. The search ends when the current source is fully b-covered by new backup tasks, i.e., the combined b-cover percentages of all chosen backup tasks in this round equals "1," which satisfies Rule II in Section IV-A. Third, according to Rule III, we compute the ranks of candidate tasks according to the amount of workload a candidate can b-cover. We refer to this as *backup workload* $B(v)$(8). $B(v)$ is computed in a recursive manner according to the *distance* from the current searching source and how the workload is partitioned along the way. Fig. 6 illustrates the b-cover percentage and backup workload. The candidate with highest rank $B(v)$ is chosen in each round. Finally, we remove tasks that have been *b-covered* from the new backup task to eliminate redundant backups. It is easy to prove that *ChooseBackup-DAGTopology()* is a general version that supports line(s) or tree topologies:

$$F(v) := \begin{cases} 1, & \text{if } v \in V_s \\ \sum_{u \in P_v, u \in Z} \frac{F(u)}{|Q_u|}, & \text{otherwise} \end{cases} \tag{7}$$

$$B(v) := \begin{cases} T(v), & \text{if } v \in V_s \\ \sum_{u \in P_v, u \in Z} \frac{B(u)+T(u)}{|Q_u|}, & \text{otherwise.} \end{cases} \tag{8}$$

Lemma 3: Algorithm *ComputeFTC* keeps recovery latency guarantees for general DAG stream topologies.

This article has been accepted for inclusion in a future issue of this journal. Content is final as presented, with the exception of pagination.

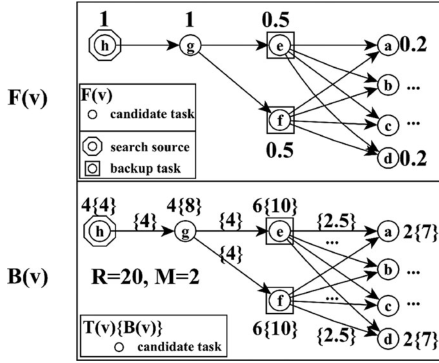


Fig. 6. Example of backward-cover percentage and backup workload.

Algorithm 4: ChooseBackup-DAGTopology (v, V, R, V_b).

Input: Current source v, searching array in *BreadthFirstSearch* orders V, recovery latency requirement R and current backup tasks V_b

Output: Set of chosen backup tasks V_m

1: $V_m := \emptyset$;
2: $Z := \overline{v_1}, .., \overline{v_z} := GenerateCandidate(v, V, R, V_b)$;
3: $F(v) := ComputePercentage()$; // (7)
4: $B(v) := ComputeRank()$; // (8)
5: **while** $(Z \neq \emptyset)$ **do**
6: $E := arc \max_{\overline{v} \in Z}\{B(\overline{v})\}$; //choose tasks with highest B(v)
7: $\overline{v_{mark}} := arc \max_{\overline{v} \in E}\{D(v, \overline{v})\}$; //tasks with longest distance
8: **if** $(M(v_{mark}) \neq 1)$ **then**
9: $V_m := V_m \cup \{\overline{v_{mark}}\}$
10: **end if**
11: remove all tasks on the paths from v to $\overline{v_{mark}}$ in Z;
12: **end while**
13: **return** V_m

E. Discussions

In this section, we discuss the fulfillment of the recovery latency requirement and the complexity of the proposed algorithm for different stream topology types.

Theorem 1: Algorithm *ComputeFTC* computes an FTC for a stream topology G that guarantees recovery latency R with a complexity of O(N^2) .

Theorem 2: Algorithm *ComputeFTC* computes an optimal FTC for a (*line(s)* or *tree*) stream topology G with recovery latency R.

Note that the function *ChooseBackup-DAGTopology()* is a greedy heuristic that updates current task coverage dynamically and chooses the task that b-covers the most workload in each round. We leave discussion of the optimality of Algorithm *ComputeFTC* over general DAG topology open. Instead, we give an exhaust algorithm that computes the optimal FTC for general DAG topology for comparison. $Algorithm^{exhaust}$ tries all possible numbers of backups $1 \leq M \leq |V|$ for a given $G(V, E)$ and R so that at each round, it tries all possible backup locations and chooses the FTC that leads to the smallest overall recovery latency O. It is easy to prove that $Algorithm^{exhaust}$ computes an optimal FTC for the FTC problem. The computing complexity of the algorithm is $O(M \cdot N!)$.

TABLE I
COMPARING ALGORITHMS

Algorithm	Description
Exhaust	A exhaust search algorithm
Line	ComputeFTC for line(s) topology
Tree	ComputeFTC for tree topology
DAG	ComputeFTC for general DAG topology
Shared-based	A algorithm based on balanced share [9]

TABLE II
STREAM TOPOLOGY TYPES

Type	Description
Line(s)	Multiple lines of task join at one *sink*.
Tree	Multiple *source* and one *sink*.
Sequential dominated	DAG with long paths (SignalGuru [33])
Parallel dominated	DAG with many autoscale tasks (TwitterSentiment in [34])

V. Experiment Results

We conduct several simulation experiments to illustrate 1) the accuracy of our failure-effect model compared to solutions computed with $algorithm^{exhaust}$; 2) the efficiency of our approach for different types of queries and the difference in failure rates; and 3) the scalability of the proposed approach and how it solves real application topologies.

A. Experimental Settings

In this section, we compare five different approaches, listed in Table I. $Algorithm^{exhaust}$ performs a full search on all solution space to compute *Opt* for the FTC problem as a baseline. We test our algorithm to show the accuracy of our model compared to OPT. Moreover, we implement a share-based algorithm modeled after the resource allocation method in [9]. This algorithm sorts all candidate tasks according to their optimal resource share and chooses the one with the largest share as the backup task in each round. We use this algorithm to measure the efficiency of our approach.

We choose four types of stream topologies for our experiments, listed in Table II. Sequential line topology and tree topology are dummy topologies that are common in practice. Then, we demonstrate how our approach solves two types of real application topologies. Sequential-dominated topology (SignalGuru [33]) illustrates how our approach solves applications with long paths. Parallel-dominated topology (Twitter Sentiment [34]) has large number of nodes and edges caused by autoscale tasks. It is used to test the scalability of our approach.

This article has been accepted for inclusion in a future issue of this journal. Content is final as presented, with the exception of pagination.

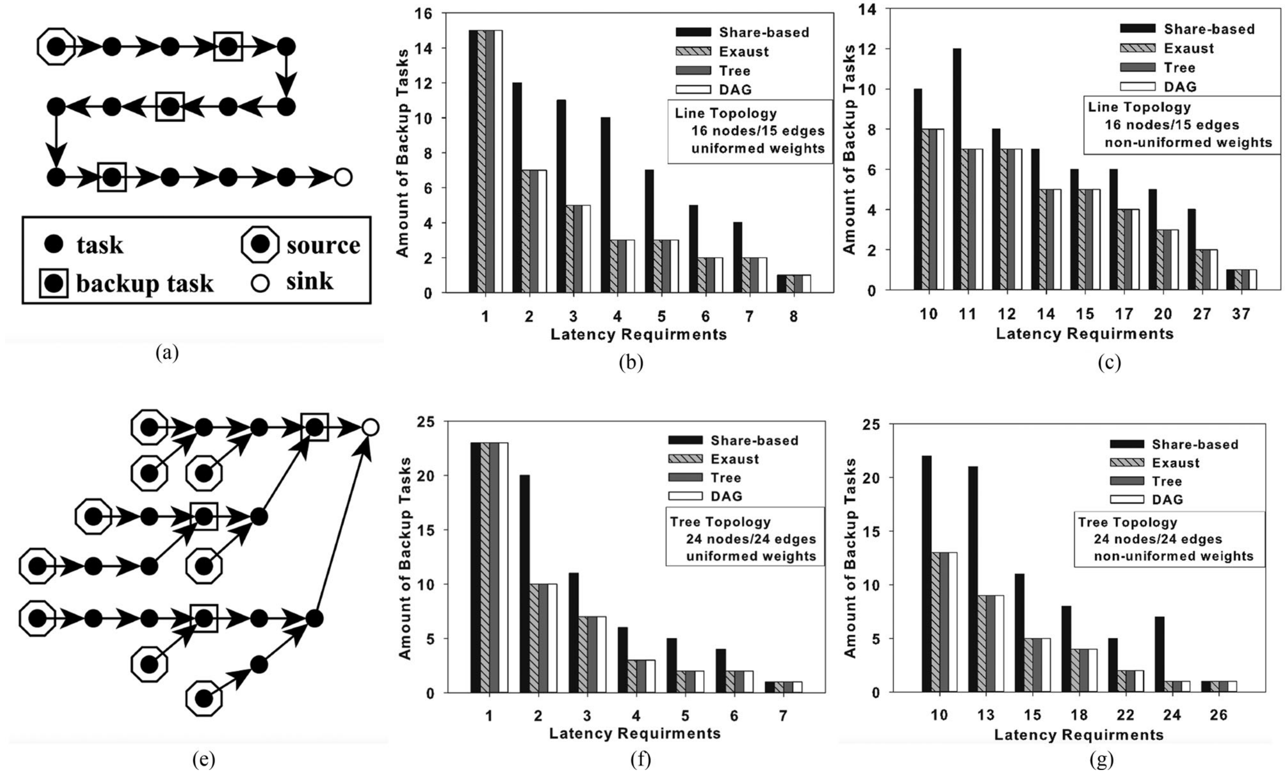


Fig. 7. Experimental results on the amount of backups and the recovery latencies for line(s) and tree topologies (four different scenarios defined by two different topologies and two different weight settings). (a) Line topology. (b) Line topology result 1. (c) Line topology result 2. (d) Tree topology. (e) Tree topology result 1. (f) Tree topology result 2.

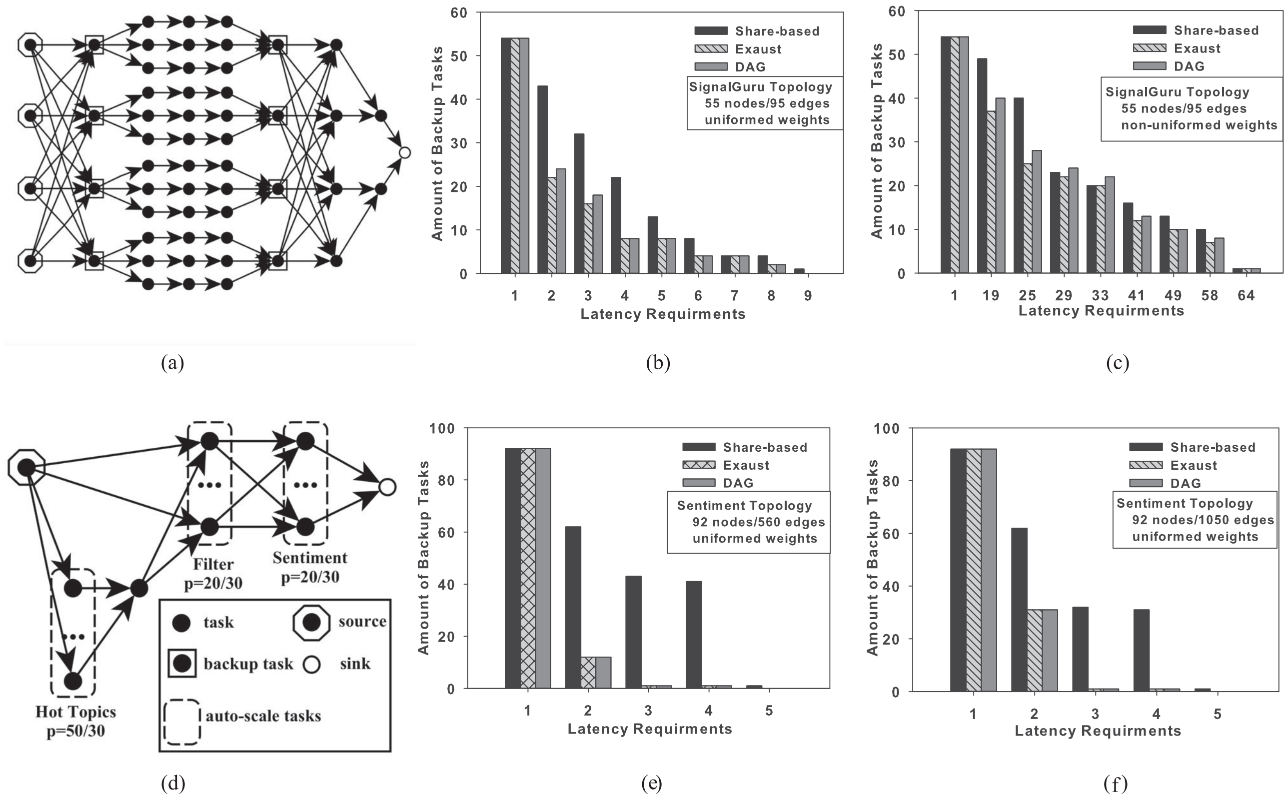


Fig. 8. Experimental results on the amount of backups and the recovery latencies for general DAG topologies (four different scenarios defined by two different topologies and two different weight settings). (a) Sequential-dominated topology. (b) Sequential-dominated result 1. (c) Sequential-dominated topology result 2. (d) Parallel-dominated topology. (e) Parallel-dominated result 1. (f) Parallel-dominated result 2.

This article has been accepted for inclusion in a future issue of this journal. Content is final as presented, with the exception of pagination.

B. Results

We use exhaust algorithm to compute Opt FTCs for all types of topologies. These results are used as benchmarks to test the performance of proposed algorithms. For each topology, we set up two different reprocessing time scenarios: uniform weight with a value of 1 and nonuniform weight with a value ranging from 1 to 10.

1) FT Overheads: As illustrated in Fig. 7, we test both uniform and nonuniform scenarios on the line(s) and tree topologies. In all four tests, $Algorithm^{FTC}$ uses exactly the same amount of backups as the exhaust algorithm, which illustrates the accuracy of both our failure effect model and the proposed approach. The share-based algorithm uses 21%–43% more backups under different settings.

SignalGuru [33] is an application with paths of nine steps, as shown in Fig. 8(a). In each step, the task parallelism is different, and therefore, this topology can help demonstrate how $Algorithm^{FTC}$ ranks tasks according to their locations in the topology. As shown in Fig. 8(b) and (c), $Algorithm^{FTC}$'s results are very close to the optimal solution computed with the exhaust algorithm. $Algorithm^{FTC}$ introduces an extra backup task of 3% on average with a uniform weight scenario and of 10% with a nonuniform scenario.

We use the Twitter Sentiment application [34] as demonstration of how our approach performs when solving topologies with large numbers of nodes and edges, i.e., *autoscale* scenario. Fig. 8(d) shows the topology, which contains three autoscale tasks (hot topics, filter, and sentiment). We tested two cases: one where the autoscale is set to (50, 20, 20), resulting in 560 edges, and one where the autoscale is set to (30, 30, 30) and has 1050 edges. Note that the number of tasks in both cases is kept the same to illustrate the effect of different cases of parallelism. As shown in Figs. 8(e) and (f), our approach works as well with these topologies as the others. $Algorithm^{FTC}$ introduces, on average, 4% extra overhead compared to the optimal results computed by $Algorithm^{exhaust}$. The share-based approach suffers from high FT overhead, but when the required recovery latency is short, none of the listed approaches are able to compute the FTCs with a small overhead. We get similar results in the SignaGuru case. This is because the task parallelism introduces too many parallel tasks, and there is no way to achieve guaranteed recovery latency except by increasing backup tasks.

2) Execution Time: In this section, we show the execution time of each algorithm. Note that $Algorithm^{Tree}$ is only tested on Line(s) and tree topologies. As illustrated in Table III, all comparison algorithms except the exhaust algorithm generate FTCs in a short time. In SigmaGuru and Sentiment, which involve 55 and 92 nodes, respectively, the exhaust algorithm finishes in several minutes, which makes it in practical. The share-based algorithm uses less computational time than our proposed algorithm because it uses a greedy method based on a sort function.

TABLE III
TIME CONSUMED TO COMPUTE FTCs

	Exhaust	Tree	DAG	Share Based
Line-u	54 ms	0.5 ms	1.2 ms	10 ms
Line-n	57 ms	0.6 ms	1.5 ms	10 ms
Tree-u	892 ms	0.8 ms	1.6 ms	10 ms
Tree-n	904 ms	0.9 ms	1.8 ms	10 ms
Guru-u	1.16 sec	NA	20 ms	8 ms
Guru-u	1.21 sec	NA	23 ms	10 ms
Senti-560	12.34 sec	NA	287 ms	28 ms
Senti-1050	108.22 sec	NA	501 ms	29 ms

Notes: Line-u and Line-n mean uniform and nonuniform scenarios, respectively.

C. Application Scenario

Nowadays, many big data applications are facilitated by public Cloud service providers, such as Google, Amazon EC2, and Microsoft Azure. Our work focuses on the FT strategy of the stream processing system. It can be applied to any stream processing system that supports the upstream backup model, and it can be deployed on any private or public Cloud service.

Given a stream topology G, estimated task reprocessing time $T(v), v \in V$, and recovery latency requirement R, $Algorithm^{FTC}$ computes FT configuration $FTC(G, R)$ using millisecond level time. FTCs can be computed dynamically in runtime, which makes application to real production environments possible. With the proposed failure-effect model, one can evaluate the FTCs of a stream processing system. The proposed algorithm, $Algorithm^{FTC}$, can be used in task allocation, online autoscaling, and load balancing to further improve system throughput and efficiency.

1) *Autoscale adjustment:* When stream processing tasks perform autoscaling, the stream topology changes, and we can recompute the FTC to adapt to the new topology.
2) *Task allocation:* When the system scheduler makes task allocation decisions for a stream topology, it can use FTCs to estimate FT overheads and reserve redundant resources for potential failure recoveries.
3) *Load balance:* When the resource manager performs periodic load balancing and task migrations, it can use the failure-effect model to evaluate current FT plans and to recompute FTC when necessary.

VI. CONCLUSION

This paper focuses on FT strategy for distributed stream processing systems. We proposed a novel quantitative failure effect model to describe the relationship between the recovery latency and FTC (the amount and location of backup tasks) of a stream topology. We introduce the FTC problem based on the failure effect model and we propose an approach to compute FTC with guaranteed recovery latency and minimum backups. We prove that the proposed method can guarantee the recovery latency requirement for all DAG stream topologies. It computes optimal FTCs for sequential and parallel line(s) topologies and for tree topologies with a time complexity of ($O(N)$). For other DAG topologies, we use a heuristic ranking function to generate configurations, causing fewer than 10% more backups on average than the optimal solution with a time complexity of ($O(N^2)$).

This article has been accepted for inclusion in a future issue of this journal. Content is final as presented, with the exception of pagination.

APPENDIX

Here, we give proofs for the theorems given in the main body of this paper.

Proof of Lemma 1: According to *ChooseBackup-Line(s) Topology() line* 3, a new backup task is chosen from a candidate set that contains tasks which satisfy the recovery latency requirement R. For multiple *line(s) topologies*, the algorithm treats each line independently and equally, which proves the claim. ■

Proof of Lemma 2: According to function *ChooseBackup-TreeTopology() line* 5, all tasks are guaranteed with a recovery latency of less than R. ■

Proof of Lemma 3: According to function *ChooseBackup-DAGTopology() line* 2, all candidate tasks are chosen according to recovery latency R so that the constraint is satisfied for all tasks. ■

Proof of Theorem 1: According to Lemmas 1–3, Algorithm *ComputeFTC* keeps the recovery latency guarantees. It is left to proof that the time complexity of the algorithm is $\mathrm{O}(\mathrm{N}^2)$.

1) For the *line(s) topology*, each task in the graph is scanned once. The complexity is $\mathrm{O}(\mathrm{N})$.
2) For the tree topology, each task is scanned once, but a backup task may be scanned multiple times in *ChooseBackup-TreeTopology() line* 8. Repeated scanning occurs when a backup task is shared by two search sources, which is at most $N-2$ times. The complexity is $\mathrm{O}(\mathrm{N})$.
3) For the DAG topology, similar to tree topology, each task in the graph is scanned once. Extra scans are introduced to compute both $F(v)$ (7) and $B(v)$ (8). They will cause $2(N-2)$ scans in total. Therefore, the complexity of algorithm *ComputeFTC* for general DAG topology is $\mathrm{O}(\mathrm{N}^2)$.

We prove the claim in all three conditions. ■

Proof of Theorem 2: Theorem 2 is easy to prove in the case of line(s) topologies. In tree topologies, each task has only one adjacent downstream task, e.g., the candidate tasks are lined up as a search path. A backup task either covers n tasks along the path so that $\sum T(v) \leq R$ or when a backup task chosen by an earlier search round exists on the path the search stops at function *ChooseBackup-TreeTopology()* line 8. This procedure will not involve setting up extra backup tasks, and therefore, we prove the claim. ■

REFERENCES

[1] M. Stonebraker, U. Çetintemel, and S. Zdonik, "The 8 requirements of real-time stream processing," *ACM SIGMOD Record*, vol. 34, no. 4, pp. 42–47, 2005.
[2] A. Arasu *et al.*, "Stream: The stanford stream data manager (demonstration description)," in *Proc. ACM SIGMOD Int. Conf. Manage. Data*, 2003, pp. 665–665.
[3] G. Hesse and M. Lorenz, "Conceptual survey on data stream processing systems," in *Proc. IEEE 21st Int. Conf. Parallel Distrib. Syst.*, 2015, pp. 797–802.
[4] S. Chandrasekaran *et al.*, "Telegraphcq: Continuous dataflow processing," in *Proc. ACM SIGMOD Int. Conf. Manage. Data*, 2003, pp. 668–668.
[5] T. Akidau *et al.*, "Millwheel: Fault-tolerant stream processing at internet scale," *Proc. VLDB Endowment*, vol. 6, no. 11, pp. 1033–1044, 2013.
[6] A. Toshniwal *et al.*, "Storm@ twitter," in *Proc. ACM SIGMOD Int. Conf. Manage. Data*, 2014, pp. 147–156.
[7] L. Neumeyer, B. Robbins, A. Nair, and A. Kesari, "S4: Distributed stream computing platform," in *Proc. IEEE Int. Conf. Data Mining Workshops*, 2010, pp. 170–177.
[8] S. Kulkarni *et al.*, "Twitter heron: Stream processing at scale," in *Proc. ACM SIGMOD Int. Conf. Manage. Data*, 2015, pp. 239–250.
[9] R. Eidenbenz and T. Locher, "Task allocation for distributed stream processing," in *Proc. 35th Annu. IEEE Int. Conf. Comput. Commun.*, 2016, pp. 1–9.
[10] J.-H. Hwang, M. Balazinska, A. Rasin, U. Cetintemel, M. Stonebraker, and S. Zdonik, "High-availability algorithms for distributed stream processing," in *Proc. 21st Int. Conf. Data Eng.*, 2005, pp. 779–790.
[11] G. Krempl *et al.*, "Open challenges for data stream mining research," *ACM SIGKDD Explorations Newslett.*, vol. 16, no. 1, pp. 1–10, 2014.
[12] L. Xu, L. Lin, S. Zhou, and S.-Y. Hsieh, "The extra connectivity, extra conditional diagnosability, and t/m-diagnosability of arrangement graphs," *IEEE Trans. Rel.*, vol. 65, no. 3, pp. 1248–1262, Sep. 2016.
[13] J. Wu, *Distributed System Design*. Boca Raton, FL, USA: CRC, 1998.
[14] R. Ananthanarayanan *et al.*, "Photon: Fault-tolerant and scalable joining of continuous data streams," in *Proc. ACM SIGMOD Int. Conf. Manage. Data*, 2013, pp. 577–588.
[15] Z. Qian *et al.*, "Timestream: Reliable stream computation in the cloud," in *Proc. 8th ACM Eur. Conf. Comput. Syst.*, 2013, pp. 1–14.
[16] L. Su and Y. Zhou, "Tolerating correlated failures in massively parallel stream processing engines," in *Proc. IEEE 32nd Int. Conf. Data Eng.*, 2016, pp. 517–528.
[17] P. Upadhyaya, Y. Kwon, and M. Balazinska, "A latency and fault-tolerance optimizer for online parallel query plans," in *Proc. ACM SIGMOD Int. Conf. Manage. Data*, 2011, pp. 241–252.
[18] M. Balazinska, H. Balakrishnan, S. R. Madden, and M. Stonebraker, "Fault-tolerance in the borealis distributed stream processing system," *ACM Trans. Database Syst.*, vol. 33, no. 1, 2008, Art. no. 3.
[19] T. Heinze, M. Zia, R. Krahn, Z. Jerzak, and C. Fetzer, "An adaptive replication scheme for elastic data stream processing systems," in *Proc. 9th ACM Int. Conf. Distrib. Event-Based Syst.*, 2015, pp. 150–161.
[20] J. Wu and K. Huang, "The balanced hypercube: A cube-based system for fault-tolerant applications," *IEEE Trans. Comput.*, vol. 46, no. 4, pp. 484–490, Apr. 1997.
[21] R. Castro Fernandez , M. Migliavacca, E. Kalyvianaki, and P. Pietzuch, "Integrating scale out and fault tolerance in stream processing using operator state management," in *Proc. ACM SIGMOD Int. Conf. Manage. Data*, 2013, pp. 725–736.
[22] J. W. Young, "A first order approximation to the optimum checkpoint interval," *Commun. ACM*, vol. 17, no. 9, pp. 530–531, 1974.
[23] A. Salama, C. Binnig, T. Kraska, and E. Zamanian, "Cost-based fault-tolerance for parallel data processing," in *Proc. ACM SIGMOD Int. Conf. Manage. Data*, 2015, pp. 285–297.
[24] P. Carbone, A. Katsifodimos, S. Ewen, V. Markl, S. Haridi, and K. Tzoumas, "Apache flink: Stream and batch processing in a single engine," *IEEE Comput. society Tech. Comm. Data Eng.*, vol. 36, no. 4, pp. 28–38, 2015.
[25] A. Martin, C. Fetzer, and A. Brito, "Active replication at (almost) no cost," in *Proc. 30th IEEE Symp. Reliable Distrib. Syst.*, 2011, pp. 21–30.
[26] G. Aupy, A. Benoit, H. Casanova, and Y. Robert, "Checkpointing strategies for scheduling computational workflows," *Int. J. Netw. Comput.*, vol. 6, no. 1, pp. 2–26, 2016.
[27] B. Babcock, S. Babu, R. Motwani, and M. Datar, "Chain: Operator scheduling for memory minimization in data stream systems," in *Proc. ACM SIGMOD Int. Conf. Manage. Data*, 2003, pp. 253–264.
[28] V. Cardellini, V. Grassi, F. Lo Presti, and M. Nardelli, "Optimal operator placement for distributed stream processing applications," in *Proc. 10th ACM Int. Conf. Distrib. Event-Based Syst.*, 2016, pp. 69–80.
[29] A. Chatzistergiou and S. D. Viglas, "Fast heuristics for near-optimal task allocation in data stream processing over clusters," in *Proc. 23rd ACM Int. Conf. Conf. Inf. Knowl. Manage.*, 2014, pp. 1579–1588.
[30] Z. Zhang *et al.*, "A hybrid approach to high availability in stream processing systems," in *Proc. IEEE 30th Int. Conf. Distrib. Comput. Syst.*, 2010, pp. 138–148.
[31] N. Garg, *Apache Kafka*. Olton, U.K.: Packt, 2013.
[32] B. Gedik, "Partitioning functions for stateful data parallelism in stream processing," *VLDB J.*, vol. 23, no. 4, pp. 517–539, 2014.
[33] E. Koukoumidis, L.-S. Peh, and M. R. Martonosi, "Signalguru: Leveraging mobile phones for collaborative traffic signal schedule advisory," in *Proc. 9th Int. Conf. Mobile Syst. Appl. Services*, 2011, pp. 127–140.
[34] B. Lohrmann, P. Janacik, and O. Kao, "Elastic stream processing with latency guarantees," in *Proc. IEEE 35th Int. Conf. Distrib. Comput. Syst.*, 2015, pp. 399–410.

This article has been accepted for inclusion in a future issue of this journal. Content is final as presented, with the exception of pagination.

Hongliang Li (M'13) received the Ph.D. degree from the College of Computer Science and Technology (CCST), Jilin University, Changchun, China.

He is currently an Associate Professor with the CCST and a Visiting Scholar with the Department of Computer and Information Sciences, Temple University, Philadelphia, PA, USA. His research interests include resource scheduling and fault tolerance in distributed computing systems. He is working on a China NFS project on theory & technology of realtime FT for stream processing systems.

Jie Wu (M'90–SM'93–F'09) received his Ph.D. degree from Florida Atlantic University, Boca Raton, FL, USA. He is the Associate Vice Provost for International Affairs with Temple University, Philadelphia, PA, USA, where he also serves as the Chair and Laura H. Carnell Professor with the Department of Computer and Information Sciences. Prior to joining Temple University, he was a Program Director with the National Science Foundation and was a Distinguished Professor with Florida Atlantic University. His current research interests include mobile computing and wireless networks, routing protocols, cloud and green computing, network trust and security, and social network applications. He regularly publishes in scholarly journals, conference proceedings, and books.

Dr. Wu serves on several editorial boards, including the IEEE TRANSACTIONS ON SERVICE COMPUTING and the *Journal of Parallel and Distributed Computing*. He was a General Cochair/Chair for the 2006 IEEE International Conference on Mobile Adhoc and Sensor Systems, IEEE International Parallel and Distributed Processing Symposium 2008, IEEE International Conference on Distributed Computing Systems 2013, and ACM International Symposium on Mobile Ad Hoc Networking and Computing 2014, as well as a Program Cochair for the IEEE International Conference on Computer Communications 2011 and China Computer Federation (CCF) China National Computer Congress 2013. He was an IEEE Computer Society Distinguished Visitor, ACM Distinguished Speaker, and the Chair for the IEEE Technical Committee on Distributed Processing. He is a CCF Distinguished Speaker. He received the 2011 CCF Overseas Outstanding Achievement Award.

Zhen Jiang (M'02) received the B.S. degree from Shanghai Jiaotong University, Shanghai, China, in 1992, the M.S. degree from Nanjing University, Nanjing, China, in 1998, and the Ph.D. degree from Florida Atlantic University, Boca Raton, FL, USA, in 2002.

He is currently an Associate Professor with the Computer Science Department, West Chester University of Pennsylvania (WCU), West Chester, PA, USA, the Director of National Security Agency (NSA) certified Information Security Center at WCU, and an Adjunct Professor with Temple University, Philadelphia, PA, USA. His research interests include information system development and wireless communication.

Dr. Jiang received the Best Paper Award for protocols and algorithms in the 7th IEEE International Conference on Mobile Ad-hoc and Sensor Systems in 2010. He is also active in many committees and is a member of the Association for Computing Machinery, where he is involved in the organization of many conferences and workshops.

Xiang Li received her master degree from University of Auckland, New Zealand. She is currently working toward her Ph.D. degree with the College of Computer Science and Technology (CCST), Jilin University, Changchun, China.

She is a faculty member with the CCST, Jilin University. Her research interests include cloud computing and distributed systems.

Xiaohui Wei (M'13) received his Ph.D. degree from the College of Computer Science and Technology (CCST), Jilin University, Changchun, China. He is a Professor and the Dean of the College of Computer Science and Technology, Jilin University, Changchun, China.

He is currently the Director of the High Performance Computing Center, Jilin University. His current research interests include resource scheduling for large distributed systems, infrastructure level virtualization, large-scale data processing systems, and fault-tolerant computing. He has published more than 50 journal and conference papers in the above areas.