# FP4S: Fragment-based Parallel State Recovery for Stateful Stream Applications

Pinchao Liu, Hailu Xu, Dilma Da Silva[‡], Qingyang Wang[†], Sarker Tanzir Ahmed[‡], Liting Hu

Florida International University, [‡]Texas A&M University, [†]Louisiana State University
Email:{pliu002, hxu017}@cs.fiu.edu, dilma@cse.tamu.edu, qywang@csc.lsu.edu, tanzir@tamu.edu, lhu@cs.fiu.edu

***Abstract*—Streaming computations are by nature long-running. They run in highly dynamic distributed environments where many stream operators may leave or fail at the same time. Most of them are stateful, in which stream operators need to store and maintain large-sized state in memory, resulting in expensive time and space costs to recover them. The state-of-the-art stream processing systems offer failure recovery mainly through three approaches: replication recovery, checkpointing recovery, and DStream-based lineage recovery, which are either slow, resource-expensive or fail to handle many simultaneous failures.**

**We present FP4S, a novel fragment-based parallel state recovery mechanism that can handle many simultaneous failures for a large number of concurrently running stream applications. The novelty of FP4S is that we organize all the application's operators into a distributed hash table (DHT) based consistent ring to associate each operator with a unique set of neighbors. Then we divide each operator's in-memory state into many fragments and periodically save them in each node's neighbors, ensuring that different sets of available fragments can reconstruct lost state in parallel. This approach makes this failure recovery mechanism extremely scalable, and allows it to tolerate many simultaneous operator failures. We apply FP4S on Apache Storm and evaluate it using large-scale real-world experiments, which demonstrate its scalability, efficiency, and fast failure recovery features. When compared to the state-of-the-art solutions (Apache Storm), FP4S reduces 37.8% latency of state recovery and saves more than half of the hardware costs. It can scale to many simultaneous failures and successfully recover the states when up to 66.6% of states fail or get lost.**

## I. Introduction

Stream processing technology has become a critical building block of many real-time applications, such as making business decisions from marketing streams, identifying spam campaigns from social network streams, predicting tornados and storms from radar streams, and analyzing genomes in different labs and countries to track the sources of a potential epidemic. Over the last decade, a boom of stream processing systems has been developed including Storm [8], Trident [10], Spark Streaming [20], Borealis [23], TimeStream [48], S4 [44], etc.

A driving need is that today many stream applications need to store and update the large-sized application state along with their processing, and process live data streams in a timely fashion from massive and distributed data sets. This poses a significant challenge to the failure recovery mechanism of state-of-the-art stream processing systems. This is because (1) stream operators are by nature long-running in which failures and stragglers are inevitable and very difficult to predict; (2) a large number of stream applications may run concurrently on the same platform, and many distributed operators may fail simultaneously; and (3) large distributed states must be restored efficiently after node failures.

This work is supported by the National Science Foundation (NSF) under NSF-SPX-1919126.

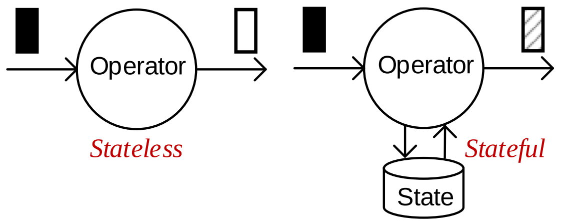


Figure 1: Contrast of stateless stream processing and stateful stream processing.

A stream is an unbounded sequence of tuples (e.g., online social network's microblog streams) generated continuously in time. A stream processing system creates a logical topology of stream processing operators, connected in a directed acyclic graph (DAG), processes the tuples of a stream as they flow through the DAG, and outputs the results in a short time. DAGs can be implemented via many patterns, such as the partition/aggregate pattern which scales out by partitioning tasks into many sub-tasks (e.g., Dryad [36]), sequential/dependent pattern in which streams are processed sequentially and subsequent streams depend on the results of previous ones (e.g., Storm [8]), and hybrid pattern with sequential/dependent and partition/aggregate (e.g., Spark Streaming [20], Naiad [42]). Figure 1 shows the contrast of stateless stream processing vs stateful stream processing. Input records are shown as black bars. A stateless operator transforms each input record one at a time and outputs each result based solely on that last record (white bar). A stateful operator maintains the value of state for some of the records processed so far (in local memory or remote storage) and updates it with each new input, such that the output (the bar with the pattern) reflects results that take into account both historical records and the new input. State recovery is the process of recovering application states when one or many operators fail or lose their states.

Application developers are facing significant challenges in handling many simultaneous failures for a large number of concurrently running stream applications.

The first challenge is "*how to scale recovery with the size of the state, the number of simultaneous failures and the number of concurrently running stream applications on a shared platform?*" Existing studies [2, 5, 6, 8, 10, 20, 26, 39, 40, 44] mostly inherit MapReduce's "single master/many workers" architecture, where the central master is responsible for all scheduling activities. As such, they are limited to a fixed computation model, e.g., asynchronous stream processing like Storm [8], synchronous mini-batch processing like Spark[7], etc. Note that the recovery operation is a critical consumer of time and space. It must quickly recover all failure operators' lost states on failover nodes (if any) without blocking the normal processing of stream applications. As a result, it is difficult (or even impossible) for the centralized master to manage state recovery of a large number of concurrently running applications due to the inherent centralized bottlenecks.

The second challenge is "*how can we handle many simultaneous failures while achieving fast recovery and imposing low hardware cost?*" State-of-the-art stream processing systems offer failure recovery mainly through three approaches: replication recovery [28, 52], checkpointing recovery [8, 10, 48] and DStream-based lineage recovery [1, 29, 53, 58], which are either slow, resource-expensive or fail to handle many simultaneous failures. Replication recovery adds significant hardware cost because multiple copies must concurrently run on distinct nodes for failover. Checkpointing recovery is known to be prohibitively expensive, and users in many domains disable it as a result [27, 34, 42, 46, 47]. DStream-based lineage recovery is slow when the lineage graph is long and falls short in handling multiple simultaneously failures.

We present FP4S, a novel fragment-based parallel state recovery mechanism to address the challenges listed above: to efficiently handle many simultaneous failures for a large number of concurrently running stream applications in a fast, scalable, and lightweight manner.

FP4S operates as follows: (1) we first organize all the application's operators into a distributed hash table (DHT) based consistent ring [51] to provide each operator with a unique set of neighbors; (2) afterward, we divide each operator's in-memory state into many fragments using erasure codes [49]. Erasure codes operate by converting a data object into a larger set of code blocks such that any sufficiently large subset of the generated code blocks can be used to reconstruct the original data object; and (3) finally, we periodically checkpoint each node's state in its neighbors, ensuring that different sets of available fragments can be used to reconstruct failed state in parallel. By doing that, this failure recovery mechanism is extremely scalable to the size of the lost state, significantly reduces the failure recovery time and can tolerate many simultaneous operator failures.

We apply FP4S on Apache Storm and evaluate it using large-scale experiments with real-world datasets. Experimental results demonstrate the scalability, efficiency, and fast failure recovery of FP4S. When compared to the state-of-the-art solutions (Apache Storm [8]), FP4S reduces in 37.8% the state recovery latency and reduces more than half of the hardware costs. It can scale to many simultaneous failures and successfully recover the states when up to 66.6% of states fail or get lost.

**Contributions.** We make the following technical contributions:

- We propose a decentralized architecture using a DHT-based consistent ring and erasure codes to recover the distributed states for numerous concurrently running stream applications. To the best of our knowledge, FP4S is the first work to use a fully decentralized architecture for state recovery (Sec. III).
- We implement the FP4S prototype on the state-of-the-art stream processing system Storm and demonstrate its portability to many other stream processing systems. The source code of FP4S is publicly available at https://github.com/fiu-elves/FP4S. (Sec. III).
- We make a comprehensive evaluation of the scalability, fast recovery and robustness of FP4S on a large cluster using real-world stream application's datasets (Sec. IV).

**Roadmap.** The remainder of this paper is organized as follows. Section II discusses the related work. Section III describes the FP4S design and implementation. Sections IV shows the experimental setup and performance evaluation. We conclude with some directions for future work in Section V.

## II. Related Work

Designing a state recovery mechanism for stateful stream processing systems is non-trivial, and existing failure recovery techniques for stream processing do not achieve the necessary scalability and efficiency. We first summarize existing stateful stream processing systems and then examine why their failure recovery techniques are either slow, resource-expensive or fail to handle multiple failures. Then we propose our solution.

### A. Stateful Stream Processing Systems

Many industrial stream processing systems either do not support state (Heron [39], S4 [44], early version of Storm [8]), or rely on in-memory data structures such as hash tables and hash table variants to store state. For example, Muppet [40] and Trident [10] (an extension of Storm) store state via hash tables. Spark Streaming [20] enables state computation via RDDs [57] which are inherent hashmaps. Some other systems such as Millwheel [25], and Dataflow [26] choose to separate state from the application logic and have state centralized in a remote storage [24, 27, 30] (e.g., a database management system, HDFS or GFS) shared among applications, along with periodically checkpointing state for fault tolerance. A few other systems such as Kafka [5], Samza [6, 45], Spark Streaming [20], and Flink [1, 29] use a combination of "soft state" stored in in-memory data structures along with "hard state" persisted in on-disk data store (e.g., RocksDB [18], LevelDB [14]).

Scaling to large distributed states and recovering from failures in such systems is quite expensive, because when a

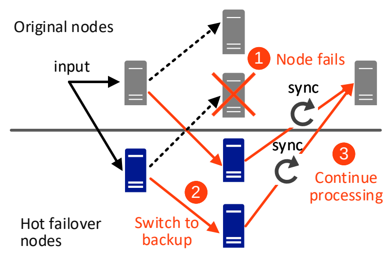


Figure 2: The replication recovery workflow.

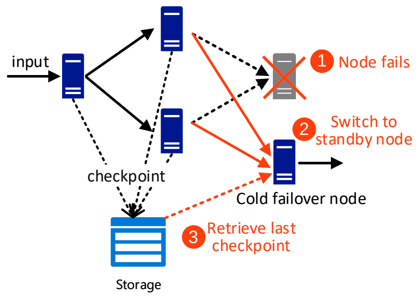


Figure 3: The checkpointing recovery workflow.

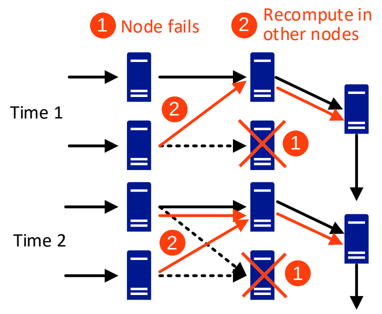


Figure 4: The DStream-based lineage recovery workflow.

single node fails, the distributed states for all dependent nodes must be reset to the last checkpoint, and computation must resume from that point, costing a lot of extra time and space to accomplish recovery. Moreover, these systems rely on a single master for handling failures and stragglers, exhibiting significant overhead from centralization bottlenecks.

### B. Failure Recovery in Stream Processing Systems

Existing stream processing systems offer failure recovery mainly through the use of three approaches: replication recovery, checkpointing recovery, and DStream-based lineage recovery.

**Replication recovery.** In the process of replication recovery, as shown in Figure 2, there is a completely separate set of hot failover nodes, which processes the same stream in parallel with the primary set of nodes. The input records are sent to both. When there is a failure or multiple failures in the primary nodes, the system automatically switches over to the secondary set of nodes and the system can continue processing with very little or no disruption. The replication recovery has been widely used in systems such as Flux [52] and Borealis [28]. The failover is fast, and it can handle multiple concurrent failures. However, the replication recovery costs twice the hardware.

**Checkpointing recovery.** In the process of checkpointing recovery, as shown in Figure 3, each of the nodes in the pipeline has a buffer in memory, which retains a backup of the records that it has forwarded to the downstream nodes since the last checkpoint. All nodes periodically checkpoint their states to a remote storage such as HDFS or GFS. A standby set of nodes is maintained in the system. The checkpointing recovery has been widely used in systems such as TimeStream [48], Trident [10], Drizzle [56] and Multi-level Checkpointing from LLNL [41]. Drizzle [56] introduced group scheduling and pre-scheduling to avoid the centralized scheduling bottleneck. However, it used batch processing model and focus on scheduling tasks for one application while FP4S uses a record-at-a-time processing model and focus on many concurrently running jobs.

**DStream-based lineage recovery.** To achieve both fast recovery and small hardware overhead, the DStream-based lineage recovery was proposed, as shown in Figure 4, which is used in systems such as Apache Spark based systems [1, 29, 53, 58]. The most recent state is stored in each node's memory using a data structure called Resilient Distributed Dataset (RDD) [57], together with the lineage graph, that is, the graph of deterministic operators used to build RDDs. The entire recovery processing is linear, that is, the lost tasks need to be executed or computed strictly in line with the original lineage graph on other nodes. As such, the recovery process may be slow when the lineage graph is long and incur multiple data uploads through the network, consuming a critical resource in geo-distributed network settings.

To our best knowledge, the very few research projects that are broadly relevant to state management solutions are [4, 15, 35, 55], which either point out the criticality of making state explicit [35, 55] or develop mechanisms for reprocessing state [4, 15], but propose no effective solutions for fast state recovery for concurrently running stream applications.

## III. System Design and Implementation

In this section, we describe the basic workflow of FP4S, introduce each component, show how stream applications' distributed states are recovered by the FP4S-enabled stream processing system, and explain the performance, scalability and flexibility benefits of using FP4S.

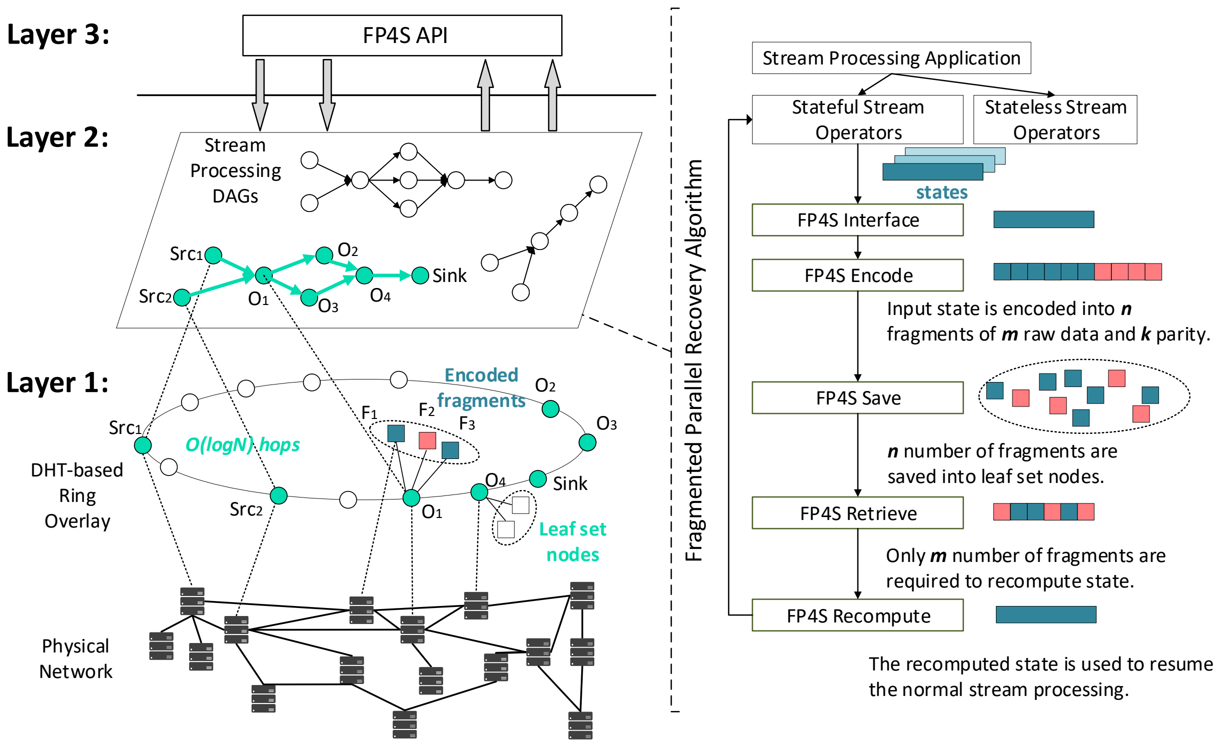
### A. Overview

The FP4S aims to achieve the following goals:

- *Resource efficient.* Avoid the replication hardware overhead.
- *Fast recovery.* Avoid the slow recovery of retrieving state from disk and replaying the data input that hurts the service quality of stream applications.
- *Resilient to multiple failures.* The mechanism needs to handle multiple simultaneous failures due to the much higher node dynamics in large clusters.

As show in Figure 5, the FP4S system consists of three layers: *The DHT-based consistent ring overlay*, *the fragmented parallel state recovery mechanism*, and *the high-level FP4S interfaces* that are exposed to the stream processing systems (e.g., Storm [8], Spark Streaming [20], Heron [39]) for implementing the state recovery for stream applications.

- **Layer 1: DHT-based ring overlay.** Each data center server is installed with one or many in-situ stream operators, also called "nodes" in this study. We organize



Layer 3:
FP4S API
Layer 2:
Stream Processing DAGs
Src1
Src2
O1
O2
O3
O4
Sink
Layer 1:
Encoded fragments
F1
F2
F3
O(logN) hops
DHT-based Ring Overlay
Leaf set nodes
Physical Network
Fragmented Parallel Recovery Algorithm
Stream Processing Application
Stateful Stream Operators
Stateless Stream Operators
states
FP4S Interface
FP4S Encode
Input state is encoded into n fragments of m raw data and k parity.
FP4S Save
n number of fragments are saved into leaf set nodes.
FP4S Retrieve
Only m number of fragments are required to recompute state.
FP4S Recompute
The recomputed state is used to resume the normal stream processing.

Figure 5: FP4S system design.

these potentially hundreds of thousands of nodes into a distributed hash table (DHT) based ring overlay (e.g., Pastry [51], Chord [54]) which is commonly used in Bitcoin [43], BitTorrent [32], and FAROO [12]. This overlay is self-organizing and self-repairing. To do that, each node needs to maintain two data structures: a routing table and a leaf set, in which the routing table is used for looking for the state (within $log(N)$ hops) and the leaf set nodes are used for recovering the application state if one or more nodes fail.

- **Layer 2: fragmented parallel state recovery.** Periodically, the state in each node's memory is divided into $m$ identically-sized blocks, which are encoded into $n$ blocks, where $n > m$. The $n$ blocks of the state are replicated to $n$ nodes from the original node's leaf set nodes in parallel, guaranteeing that the original state can be reconstructed from any $m$ blocks.
- **Layer 3: high-level interfaces to stream processing systems.** The high-level FP4S programming API (Table I) is exposed to the state-of-the-art stream processing systems and programmers for implementing the parallel state recovery policies for concurrently running stream applications, e.g., Storm [8], Spark [20], and Flink [1].

## B. DHT-based Ring Overlay

FP4S leverages DHT-based consistent overlay [51, 54] to support parallel recovery of distributed states for a large number of concurrently running stream applications. In this DHT-based consistent ring overlay (e.g., Pastry [51], Chord [54]), each node is equal to the other nodes, and they have the same rights and duties. The primary purpose of this model is to enable all nodes to work collaboratively to deliver a specific service. For example, in BitTorrent [32], if someone downloads some file, the file is downloaded to her computer in parts that come from many other computers in the system that already have that file. At the same time, the file is also sent (uploaded) from her computer to others that ask for it.

Similar to BitTorrent in which many machines work collaboratively to undertake the task of downloading files and uploading files, we enable distributed stream operators to work collaboratively to undertake the original centralized master's failure recovery task. First, each stream operator maintains an in-memory buffer to store the application state. Instead of storing states at a remote storage, these distributed stream operators store the states for each other. Second, these distributed stream operators (nodes) are self-organized into a DHT-based overlay. Each node is randomly assigned a unique `NodeId` in a large circular `NodeId` space. `NodeIds` are used to identify the nodes and route stream data. It is guaranteed that any data can be routed to a node whose `NodeId` is numerically closest to the destination node within $O(logN)$ hops. To do that, each node maintains two data structures: a routing table and a leaf set.

1) *Routing table:* The routing table consists of physical node characteristics (`NodeId`, `IP`) organized in rows by the length of common prefix. When routing a message, each node

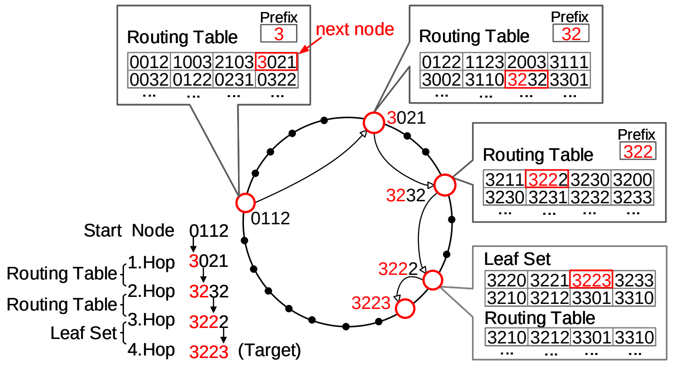


Figure 6: The routing process is cooperatively fulfilled by the routing table and the leaf set.

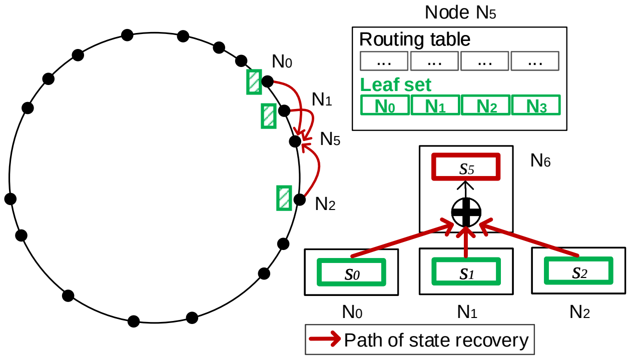


Figure 7: The fragment-based parallel state recovery process.

forwards it to the node in the routing table with the longest prefix in common with the destination `NodeId`. Figure 6 shows this routing process of Pastry's DHT [51]. At each routing step, given a key, Pastry routes messages to the node whose `NodeId` is numerically closest to the key. The node first checks if the key falls in the range of the `NodeIds`' leaf set. If so, the message is directly forwarded to that node. If not, the message is forwarded to another node in the routing table whose `NodeId` shares a common prefix with the key by at least one more digit (see Figure 6 first, second, and third hops). In some cases, there is no appropriate entry in the routing table or the associated node is not reachable. Then the message is forwarded to a node whose prefix is the same as the local node, but numerically closer.

2) *Leaf set:* The leaf set contains a fixed number of nodes whose `NodeIds` are numerically closest to each node, which assists in rebuilding routing tables and reconstructing application's state when any operator fails (see Sec. III-C, next, for more details).

## C. Fragmented Parallel State Recovery

The parallel recovery mechanism of FP4S leverages the key idea from erasure codes. Erasure codes operate by converting a data object into a larger set of code blocks such that any sufficiently large subset of the generated code blocks can be used to reconstruct the original data object. For example, (32, 16)-Reed-Solomon (RS) code [49] divides a data object into 16 blocks and transforms these blocks into 32 coded blocks, guaranteeing that any 16 out of the 32 coded blocks are sufficient to reconstruct the original data object. Erasure codes have been widely used in massive storage systems (e.g., OceanStore [38]), Bar codes (e.g., QR Code [37]), data transmission technologies (e.g., DSL [33]) and space transmission technologies (e.g., Galileo Probe). Figure 7 shows the steps of the erasure-code-based parallel recovery algorithm.

Built upon Sec. III-B's DHT-based ring overlay, each node maintains a routing table and a leaf set. Periodically, the state in each node's memory is encoded into $n$ identically-sized fragments, which include $m$ raw data fragments and $k$ parity fragments, where $k >= 1, n = m + k$. Then these $n$ fragments of the state are replicated to $n$ nodes in the original node's leaf set in parallel. The error correction mechanism of the erasure codes guarantees that any $m$ out of the $n$ fragments are sufficient to correctly recompute, even though when some fragments are not available in the leaf set (denoted as $e$), to reconstruct the original state. Thus, as long as $n$–$e$ $>= m$, the original state is safe to be accurately recomputed from the node's leaf set nodes.

- **Step 1: encode state.** For each node, FP4S converts its current version of state in a sliding window into $n$ fragments (configurable parameter) according to RSCodes algorithm [49]. These $n$ fragments include $m$ raw data fragments and $k$ parity fragments. The amount of $m$ and $k$ are configurable.
- **Step 2: save state.** Each node sends these $n$ fragments to any $n$ of its leaf set nodes. We ensure that the size of the leaf set is larger than $n$. We assign the `NodeIds` to reflect the physical proximity in order to ensure that the leaf set nodes are also geographical closest nodes that have abundant bandwidth.
- **Step 3: retrieve state.** Once any failure happens, the retrieve routine is triggered. A request to obtain the lost state's fragments will be sent out. To recompute the lost state, FP4S only requires $m$ amount of $n$ total fragments. These fragments are stored at the leaf set nodes that are quite easy to access.
- **Step 4: recompute state.** Finally, the state recompute routine is triggered, which reconstructs the lost state using erasure codes [49]. After that, the recovered state will be used as input for the downstream operators and we can resume the normal stream processing.

The benefits are the following: (1) it allows for tolerating a maximum of $(n-m)$ simultaneous failures; (2) the recovery process is fast. For multiple failures, different nodes from non-overlapping leaf set nodes can work in parallel to recompute the lost state, which is faster than DStream's line-structured recovery that executes strictly in line with the original lineage graph; and (3) we achieve data locality because the leaf set contains nodes that are geographically close to the original nodes (e.g., in the same rack or in the same site) that have abundant upload bandwidth.

### D. FP4S API

FP4S is platform-agnostic and can be easily integrated with stream processing platforms such as Storm [8], Spark Streaming [20], Flink [1], Timely Dataflow [48], Heron [39], etc. In our design, using FP4S is essentially a configuration option. Depending on the usage scenario (e.g., stateful or stateless, latency requirement, reliability requirement), users can choose to configure whether and when they want FP4S support. Table I shows the FP4S API.

Table I: FP4S API

| | |
|---|---|
| **List <Fragment>`Encode`(int rawDataNumber, int parityNumber, State inputState)** | The function is invoked to encode a state into many fragments. The fragment number is decided by the inputs of `rawDataNumber` and `parityNumber`. The output is a list of encoded fragments with the length of `rawDataNumber` + `parityNumber`. |
| **Boolean[] `Save`(List<>fragment, DHTNetwork dhtNetwork, int numberOfThreads)** | The function is invoked to save state into the DHT's overlay. It generates multiple threads to concurrently save the fragments. The inputs are the fragments, the DHT overlay information and the number of threads. The output is a Boolean array that indicates the status of each fragment. |
| **List<Fragment>`Retrieve`(String stateName, DHTNetwork dhtNetwork, int numberOfThreads)** | The function is invoked when a state recovery request is issued. |
| **String Recompute(List<>fragments)** | The function is invoked to recover the state. It loops through all the retrieved fragments. If the number of fragments is equal or larger than the number of raw fragments, the function will perform further computation to recompute the retrieved fragments into the original state. |

### E. Instrumentation requirements

Here we describe the instrumentation requirements FP4S imposes and discuss the issues we encountered when integrating it with the Apache Storm processing engine.

In Apache Storm [8], stream processing applications are deployed and executed as *topologies*. The *topologies* contain the business logics. These logics are formed as a DAG (directed acyclic graph) and implemented by *spouts* and *bolts*. *Spouts* are the data sources of the stream, which accept input data from raw data sources like Twitter Streaming API [21], Apache Kafka queue [5], etc. *Bolts* are the logical processing units. *Spouts* pass data to bolts and *bolts* process and produce a new output stream. `IRichBolt` is the common interface for implementing *bolts*.

FP4S interacts with the `IRichBolt` interface in Storm [8]. If FP4S is enabled, FP4S periodically saves the states into the DHT-based ring overlay for all stateful operators (*bolts*). For record-at-a-time systems like Storm, saving every operator's state may incur a lot of overhead. Instead, we aggregate the states for all the operators except for sources (*spouts*) and *sinks*. The aggregated state size is configurable in order to satisfy different real-world stream applications' requirements. After the size reaches a certain threshold, the `Encode` function encodes the states into fragments and the `Save` function puts these fragments into the DHT-based overlay. If any node fails, the leaf set nodes call the routines to `Retrieve` and `Recompute` states on failover nodes. Any qualified available subset of fragments will be sufficient to recover the lost states by the `Recompute` function.

Table II: Real-world application's dataset.

| **Application** | **Dataset** | **Size** |
|---|---|---|
| Trending Topics | Twitter Streaming API [21] | >1TB |
| Bargain Index | Google Finance [13] | >1TB |
| Word Count | Project Gutenberg [17] | 8GB |
| | Wikimedia Dumps [22] | 9GB |
| Traffic Monitoring | Dublin Bus Traces [11] | 4GB |

## IV. Evaluation

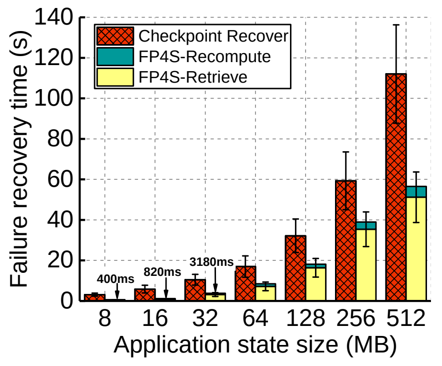
We integrate FP4S with Apache Storm and evaluate it using large scale real-world experiments, demonstrating its scalability, efficiency, and fast failure recovery. Experimental evaluations answer the following questions:

- How does the FP4S-enabled Storm system scale with the size of state, the number of concurrently running applications and the number of simultaneously failed operators?
- How does the efficiency of the fragment-based parallel state recovery algorithm change with different parameters, e.g., the number of the raw fragments ($m$) and the number of the coded fragments ($n$), and how does FP4S balance the workload?
- What are the performance and functionality benefits of FP4S compared to the state-of-the-art solutions?
- What is the overhead and the instrumentation used by FP4S?

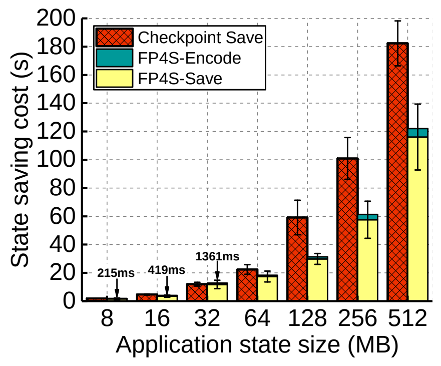
### A. Setup

We run all FP4S experiments on up to 4 machines, each with 16 Intel Xeon Gold 6130@2.10GHz cores and 256GB of RAM, running GNU/Linux 3.10.0. On top of these machines, we boot up 50 virtual machines to host 650 stream operators in total, each with 4 cores and 8GB of memory, running Linux Ubuntu 4.4.0. We use Apache Storm 2.0.0 [9] configured with 10 TaskManagers, each with 4 slots (maximum parallelism per operator = 36). We use Pastry 2.1 [16] configured with leafset size of 24, max open sockets of 5000 and transport buffer size of 6MB.

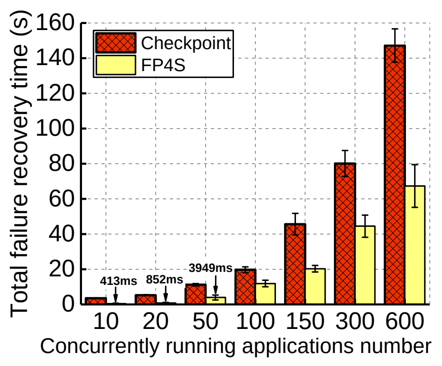
To demonstrate generality across diverse computations and streaming operators. We deploy Yahoo streaming benchmarks [31] and real-world stream applications to FP4S (see Table II).



(a) State recovery time for different input state sizes.



(b) State saving time for different input state sizes.



(c) Total failure recovery time by varing # concurrently running stream applications.

Figure 8: Performance comparison FP4S performance vs checkpointing strategy for different input states size and applications number.

These stream applications contain various representative streaming operators: stateless streaming transformations (e.g., `map`, `filter`), stateful operators (e.g., `incremental join`), and various window operators (e.g., sliding window, tumbling window and session window). We compare FP4S with a state-of-the-art failure recovery solution: the checkpointing recovery approach commonly used in TimeStream [48], Storm [8], and Trident [10]. We were not able to compare with Drizzle [56] because its source code is not publicly available. We choose the checkpointing recovery approach as the baseline approach because the replication recovery already costs twice the hardware and the DStream-based lineage recovery approach is not generalized because it sacrifices programming model transparency by forcing programmers to declare and maintain state using Spark's RDDs [57].

The base value of raw fragments ($m$) and the coded fragments ($n$) are derived from production systems such as Pond [50] and Sia [19], which set $m = 16$, $n = 16$ and $m = 10$, $n = 20$ respectively. To fully evaluate the FP4S performance, we vary the values of $m$, $n$ and the input state size.

## B. FP4S vs Checkpointing Recovery

We compare the failure recovery time of FP4S with the checkpointing recovery by varying the size of the state and the number of concurrently running stream applications.

The FP4S fragmented parallel recovery process consists of two steps: saving the state to leaf set nodes in the DHT-based overlay, and recomputing the state if any failure happens. Similarly, the checkpointing recovery process also consists of two steps: checkpointing the state to the HBase [4] or HDFS [3], and retrieving the state from HBase or HDFS if failure happens. Note that, for both approaches, the first step can run asynchronously with the second step so the first step may not impact the failure recovery time if they are executed in a pipeline.

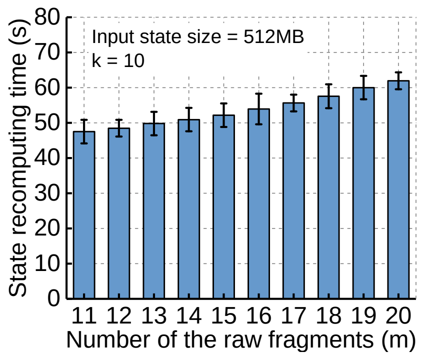
**Failure recovery time.** Figure 8a shows the failure recovery time comparison of FP4S vs Storm's checkpointing recovery. In this experiment, we focus on a single stream application that has only one operator failure and we vary the state size. As Figure 8a shows, FP4S achieves 40.3% to 87.1% less failure recovery time compared to Storm's checkpointing recovery. The improvement gap increases as the size of the state increases. The rationale behind the result lies in that FP4S fragmented parallel recovery involves many nodes to help recompute the state in parallel which significantly reduces the failure recovery time. Instead, the checkpointing recovery only relies on a single node to retrieve the state which is constrained by the HBase I/O rate and the network bandwidth.

**State saving cost.** Figure 8b shows the state saving cost comparison of FP4S vs Storm's checkpointing recovery. The FP4S state saving cost includes the time cost for dividing the state into fragments, encoding each state, and then writing the encoded fragments into leaf set nodes. We write them into the leaf set nodes serially to enable a fair comparison with the checkpointing recovery. As Figure 8b shows, the state saving cost of FP4S is less than the state saving cost of the checkpointing recovery especially for large state because it runs in parallel with the operators execution.

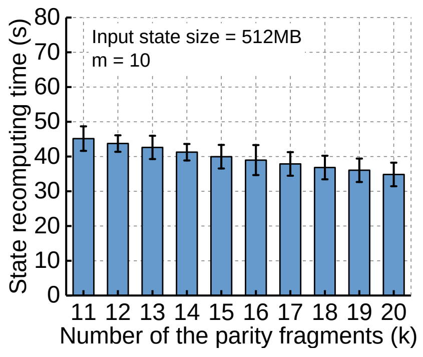
**Scale with the number of applications.** Figure 8c shows the total failure recovery time for FP4S and Storm's checkpointing recovery when there are a large number of concurrently running stream applications on the platform. We set the failure rate of stream operators to be 1% according to Zorro [47]. As Figure 8c shows, compared to Storm's checkpointing recovery that has linearly increasing state recovery time, FP4S can handle many simultaneous failures with relatively stable state recovery time. This is because FP4S leverages the DHT consistent ring overlay to distribute the total failure recovery load across all participating nodes and thus simultaneously failing operators' states can be recovered in parallel, which significantly improves the scalability.

## C. Fragmented Parallel Recovery Algorithm

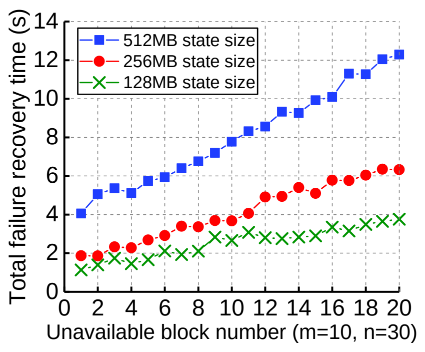
Several important factors affect the recovery performance, including the number of the raw fragments $m$ in a state, the



(a) Adjust raw fragment ($m$) parameter.

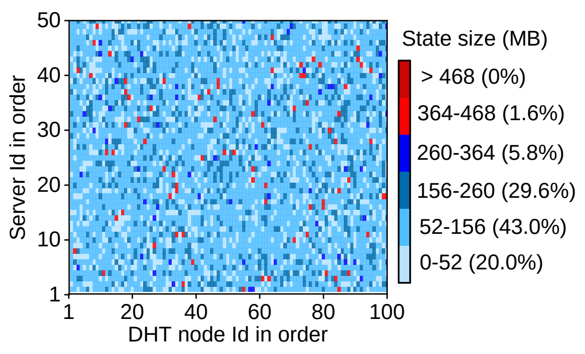


(b) Adjust parity fragment ($k$) parameter.

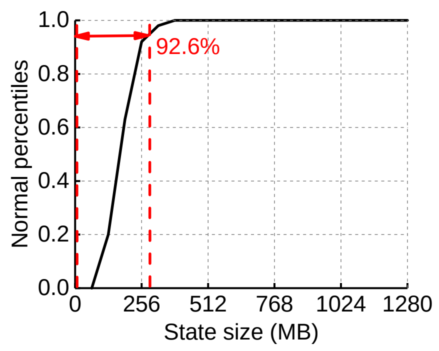


(c) Adjust unavailable block number ($e$).

Figure 9: The recovery performance evaluation by adjusting number of raw fragments, number of parity fragments, and number of unavailable blocks.



(a) Heatmap of the state size across all DHT nodes.



(b) Normal probability plot of the state size across all DHT nodes.

Figure 10: The load balance evaluation of FP4S for a collection of concurrently running stream applications.

number of the parity fragments $k$ in a state, the number of unavailable blocks $e$ in a state and the amount of leaf nodes.

**Number of the raw fragments ($m$).** We evaluate the impact of the number of the raw fragments $m$ in a state on the recovery performance by varying $m$ from 11 to 20, where $k$ is set to be 10. Figure 9a plots the performance of state recomputing time when recovering from single failure by varying $m$. We can observe that the state recomputing time increases as the number of raw fragments $m$ increases. The reason lies in that the recovery time of FP4S is mainly determined by $mB/(m+k-1)$, where $B$ is the amount of data that any providing peer uploads. $mB/(m+k-1)$ increases with the increases of $m$ when the values of $k$ and $B$ are given. Thus, the performance of FP4S is more sensitive to $m$ when $k$ is smaller.

**Number of the parity fragments ($k$).** We evaluate the impact of the number of the parity fragments $k$ in a state on recovery performance by varying $k$ from 11 to 20, where $m$ is set to be 10. Figure 9b plots the performance of state recomputing time when recovering from single failure by varying $k$. We can observe that FP4S achieves better recovery performance when $k$ is increasing from 11 to 20. The reason is that the recovery time of FP4S is mainly determined by by $mB/(m+k-1)$, where $B$ is the amount of data that any providing peer uploads. $mB/(m+k-1)$ decreases with the increases of $k$ when the values of $m$ and $B$ are given.

**Number of unavailable blocks ($e$).** We evaluate the impact of the number of unavailable blocks $e$ on recovery performance by varying $e$ from 1 to 20. We can see from Figure 9c that the total failure recovery time of FP4S increases linearly with the increase of $e$. This is due to the current star-like structure of the FP4S prototype, in which the performance bottleneck of FP4S is mainly the upload speeds of providing peers. Thus, the recovery performance of FP4S is inversely proportional to the amount of data a providing peer uploads. In the future, we plan to enable fragments to be transmitted and combined through a spanning tree covering the replacing node and all providing nodes to mitigate this performance bottleneck.

## D. *Load Balance*

FP4S has an attractive load balance feature because it assigns each operator a non-overlapping set of leaf set nodes, which distributes the total load of state saving and recovery all over the overlay. We evaluate the load balance of FP4S by running 500 stream applications on the platform of 50 virtual servers that have 5000 DHT nodes. Each application has 512 MB state and we have 780GB states in total that need to be saved in the above 5000 nodes.

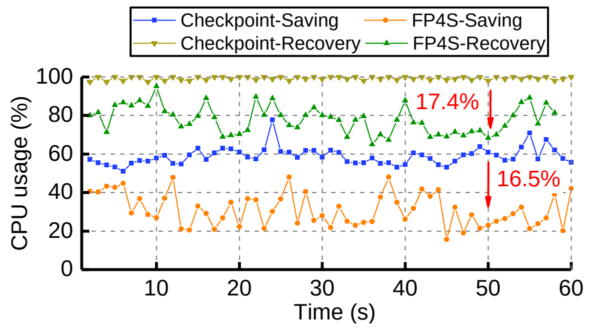
Figure 10a plots the heatmap of the state size on each DHT node. Figure 10b shows the normal probability plot for the size of stored state per node. We can observe that over 90% (20%+43%+29.6%) of nodes store state with less than 260 MB (see the blue zone in Figure 10a), while less than 7.5% nodes store state over 260 MB, demonstrating its attractive load balance and scalability features.

### *E. Overhead Analysis*

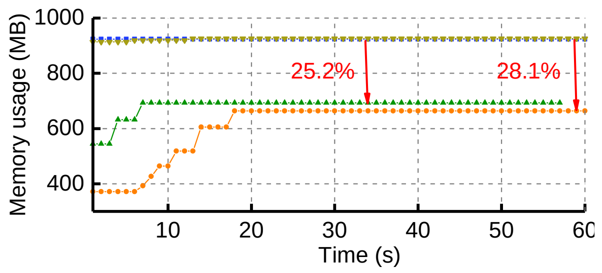
We evaluate FP4S runtime overhead, particularly those pertaining to its saving and recovery execution, and compare them with the checkpointing recovery approach. The FP4S saving and recovery require additional CPU to compute the fragments and additional memory for maintaining intermediate results. Figure 11 presents these costs, explained next.

**CPU overhead.** Figure 11a shows the per-node CPU runtime overhead comparison of FP4S vs checkpointing recovery. The CPU runtime overhead of FP4S is less than the checkpointing recovery. This is because fragment calculations account for only a small fraction (<10%) of the entire save and restore execution.

**Memory overhead.** Figure 11b shows the per-node memory run-time overhead comparison of FP4S vs checkpointing recovery. The memory overhead of FP4S is less than the checkpointing recovery. This is because the checkpointing recovery involves a centralized daemon process such as Zookeeper for coordination. Instead, FP4S nodes are independent, which does not require the centralized daemon process to maintain these relationships.



(a) The CPU overhead.



(b) The memory overhead.

Figure 11: The overhead analysis of the FP4S-enabled Storm at runtime.

## V. Conclusion

In this paper we have described and evaluated FP4S, a novel fragment-based parallel state recovery mechanism that can handle many simultaneous failures for stateful stream applications. Unlike existing failure recovery approaches based on replication or checkpointing which are either slow, resource-expensive or fail to handle many simultaneous failures, FP4S leverages DHTs and erasure codes to divide each operator's in-memory state into many fragments and periodically save them in each node's leaf set nodes in a DHT ring, ensuring that different sets of available fragments can reconstruct failed state in parallel. By doing that, this failure recovery mechanism is scalable to the size of the lost state, which significantly reduces the failure recovery time and can tolerate many simultaneous operator failures.

FP4S is framework-agnostic and broadly applicable to a range of streaming systems. We have implemented FP4S atop the state-of-the-art stream processing engine Apache Storm, and demonstrated its scalability, efficiency, and fast failure recovery features that incur negligible instrumentation overheads. Note that dividing data blocks, encoding raw fragments, uploading encoded fragments, and reconstructing states are non-overlapping operations, so many interesting questions for future work arise: such as *how to pipeline them to speed up the recovery process? How to adjust the ratio of $m$ and $n$ to tradeoff the reduced upload data and the increased computation complexity? How to provide a theoretic model to estimate network I/O cost and computation complexity?*

## Acknowledgment

We would like to thank editors and the anonymous reviewers for their insightful suggestions and comments that improved this paper. This work is supported by the National Science Foundation (NSF) under NSF-SPX-1919126.

## References

[1] Apache flink. http://flink.apache.org/.
[2] Apache flume. http://flume.apache.org/.
[3] Apache hadoop hdfs. https://hadoop.apache.org/docs/current/hadoop-project-dist/hadoop-hdfs/.
[4] Apache hbase. http://hbase.apache.org/.
[5] Apache kafka. http://kafka.apache.org/.
[6] Apache samza. http://samza.apache.org/.
[7] Apache spark. http://spark.apache.org/.
[8] Apache storm. http://storm.apache.org/.
[9] Apache storm 2.0.0. https://storm.apache.org/2019/05/30/storm200-released.html.
[10] Apache trident. http://storm.apache.org/releases/current/Trident-tutorial.html.
[11] Dublin bus gps sample data from dublin city council. https://data.gov.ie/dataset/.
[12] Faroo. https://en.wikipedia.org/wiki/FAROO.
[13] Google finance data api. http://finance.google.com/finance/feeds/.
[14] Leveldb. https://github.com/google/leveldb/.
[15] Mongodb. http://www.mongodb.com/.
[16] Pastry. https://www.freepastry.org/FreePastry/.
[17] Project gutenberg. http://www.gutenberg.com/.
[18] Rocksdb. http://rocksdb.org/.
[19] Sia: a decentralized storage platform secured by blockchain technology. http://sia.tech/.
[20] Spark streaming. https://spark.apache.org/streaming/.
[21] Twitter streaming apis. https://developer.twitter.com/en/docs/tutorials/consuming-streaming-data.

[22] Wikimedia dumps. https://dumps.wikimedia.org/.
[23] Daniel J Abadi, Yanif Ahmad, Magdalena Balazinska, Ugur Cetintemel, Mitch Cherniack, Jeong-Hyon Hwang, Wolfgang Lindner, Anurag Maskey, Alex Rasin, Esther Ryvkina, et al. The design of the borealis stream processing engine. In *Cidr*, volume 5, pages 277–289, 2005.
[24] Daniel J Abadi, Don Carney, Ugur Çetintemel, Mitch Cherniack, Christian Convey, Sangdon Lee, Michael Stonebraker, Nesime Tatbul, and Stan Zdonik. Aurora: a new model and architecture for data stream management. *the VLDB Journal*, 12(2):120–139, 2003.
[25] Tyler Akidau, Alex Balikov, Kaya Bekiroğlu, Slava Chernyak, Josh Haberman, Reuven Lax, Sam McVeety, Daniel Mills, Paul Nordstrom, and Sam Whittle. Millwheel: fault-tolerant stream processing at internet scale. *Proceedings of the VLDB Endowment*, 6(11):1033–1044, 2013.
[26] Tyler Akidau, Robert Bradshaw, Craig Chambers, Slava Chernyak, Rafael J Fernández-Moctezuma, Reuven Lax, Sam McVeety, Daniel Mills, Frances Perry, Eric Schmidt, et al. The dataflow model: a practical approach to balancing correctness, latency, and cost in massive-scale, unbounded, out-of-order data processing. *Proceedings of the VLDB Endowment*, 8(12):1792–1803, 2015.
[27] Arvind Arasu, Brian Babcock, Shivnath Babu, John Cieslewicz, Mayur Datar, Keith Ito, Rajeev Motwani, Utkarsh Srivastava, and Jennifer Widom. Stream: The stanford data stream management system. In *Data Stream Management*, pages 317–336. Springer, 2016.
[28] Magdalena Balazinska, Hari Balakrishnan, Samuel Madden, and Michael Stonebraker. Fault-tolerance in the borealis distributed stream processing system. In *Proceedings of the 2005 ACM SIGMOD international conference on Management of data*, pages 13–24. ACM, 2005.
[29] Paris Carbone, Stephan Ewen, Gyula Fóra, Seif Haridi, Stefan Richter, and Kostas Tzoumas. State management in apache flink®: consistent stateful distributed stream processing. *Proceedings of the VLDB Endowment*, 10(12):1718–1729, 2017.
[30] Sirish Chandrasekaran, Owen Cooper, Amol Deshpande, Michael J Franklin, Joseph M Hellerstein, Wei Hong, Sailesh Krishnamurthy, Samuel Madden, Vijayshankar Raman, Frederick Reiss, et al. Telegraphcq: Continuous dataflow processing for an uncertain world. In *Cidr*, volume 2, page 4, 2003.
[31] Sanket Chintapalli, Derek Dagit, Bobby Evans, Reza Farivar, Thomas Graves, Mark Holderbaugh, Zhuo Liu, Kyle Nusbaum, Kishorkumar Patil, Boyang Jerry Peng, et al. Benchmarking streaming computation engines: Storm, flink and spark streaming. In *2016 IEEE international parallel and distributed processing symposium workshops (IPDPSW)*, pages 1789–1792. IEEE, 2016.
[32] Bram Cohen. Incentives build robustness in bittorrent. In *Workshop on Economics of Peer-to-Peer systems*, volume 6, pages 68–72, 2003.
[33] Philip Golden, Hervé Dedieu, and Krista S Jacobsen. *Implementation and applications of DSL technology*. CRC press, 2007.
[34] Joseph E Gonzalez, Reynold S Xin, Ankur Dave, Daniel Crankshaw, Michael J Franklin, and Ion Stoica. Graphx: Graph processing in a distributed dataflow framework. In *11th Symposium on Operating Systems Design and Implementation*, pages 599–613, 2014.
[35] Zhiming Hu, Baochun Li, and Jun Luo. Flutter: Scheduling tasks closer to data across geo-distributed datacenters. In *IEEE INFOCOM 2016-The 35th Annual IEEE International Conference on Computer Communications*, pages 1–9. IEEE, 2016.
[36] Michael Isard, Mihai Budiu, Yuan Yu, Andrew Birrell, and Dennis Fetterly. Dryad: distributed data-parallel programs from sequential building blocks. In *ACM SIGOPS operating systems review*, volume 41, pages 59–72. ACM, 2007.
[37] Peter Kieseberg, Manuel Leithner, Martin Mulazzani, Lindsay Munroe, Sebastian Schrittwieser, Mayank Sinha, and Edgar Weippl. Qr code security. In *Proceedings of the 8th International Conference on Advances in Mobile Computing and Multimedia*, pages 430–435. ACM, 2010.
[38] John Kubiatowicz, David Bindel, Yan Chen, Steven Czerwinski, Patrick Eaton, Dennis Geels, Ramakrishna Gummadi, Sean Rhea, Hakim Weatherspoon, Westley Weimer, et al. Oceanstore: An architecture for global-scale persistent storage. In *ACM SIGARCH Computer Architecture News*, volume 28, pages 190–201. ACM, 2000.
[39] Sanjeev Kulkarni, Nikunj Bhagat, Maosong Fu, Vikas Kedigehalli, Christopher Kellogg, Sailesh Mittal, Jignesh M Patel, Karthik Ramasamy, and Siddarth Taneja. Twitter heron: Stream processing at scale. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, pages 239–250. ACM, 2015.
[40] Wang Lam, Lu Liu, Sts Prasad, Anand Rajaraman, Zoheb Vacheri, and AnHai Doan. Muppet: Mapreduce-style processing of fast data. *Proceedings of the VLDB Endowment*, 5(12):1814–1825, 2012.
[41] Adam Moody, Greg Bronevetsky, Kathryn Mohror, and Bronis R De Supinski. Design, modeling, and evaluation of a scalable multi-level checkpointing system. In *SC'10: Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 1–11. IEEE, 2010.
[42] Derek G Murray, Frank McSherry, Rebecca Isaacs, Michael Isard, Paul Barham, and Martín Abadi. Naiad: a timely dataflow system. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, pages 439–455. ACM, 2013.
[43] Satoshi Nakamoto et al. Bitcoin: A peer-to-peer electronic cash system. 2008.
[44] Leonardo Neumeyer, Bruce Robbins, Anish Nair, and Anand Kesari. S4: Distributed stream computing platform. In *2010 IEEE International Conference on Data Mining Workshops*, pages 170–177. IEEE, 2010.
[45] Shadi A Noghabi, Kartik Paramasivam, Yi Pan, Navina Ramesh, Jon Bringhurst, Indranil Gupta, and Roy H Campbell. Samza: stateful scalable stream processing at linkedin. *Proceedings of the VLDB Endowment*, 10(12):1634–1645, 2017.
[46] Daniel Peng and Frank Dabek. Large-scale incremental processing using distributed transactions and notifications. 2010.
[47] Mayank Pundir, Luke M Leslie, Indranil Gupta, and Roy H Campbell. Zorro: Zero-cost reactive failure recovery in distributed graph processing. In *Proceedings of the Sixth ACM Symposium on Cloud Computing*, pages 195–208. ACM, 2015.
[48] Zhengping Qian, Yong He, Chunzhi Su, Zhuojie Wu, Hongyu Zhu, Taizhi Zhang, Lidong Zhou, Yuan Yu, and Zheng Zhang. Timestream: Reliable stream computation in the cloud. In *Proceedings of the 8th ACM European Conference on Computer Systems*, pages 1–14. ACM, 2013.
[49] Irving S Reed and Gustave Solomon. Polynomial codes over certain finite fields. *Journal of the society for industrial and applied mathematics*, 8(2):300–304, 1960.
[50] Sean C Rhea, Patrick R Eaton, Dennis Geels, Hakim Weatherspoon, Ben Y Zhao, and John Kubiatowicz. Pond: The oceanstore prototype. In *FAST*, volume 3, pages 1–14, 2003.
[51] Antony Rowstron and Peter Druschel. Pastry: Scalable, decentralized object location, and routing for large-scale peer-to-peer systems. In *IFIP/ACM International Conference on Distributed Systems Platforms and Open Distributed Processing*, pages 329–350. Springer, 2001.
[52] Mehul A Shah, Joseph M Hellerstein, and Eric Brewer. Highly available, fault-tolerant, parallel dataflows. In *Proceedings of the 2004 ACM SIGMOD international conference on Management of data*, pages 827–838. ACM, 2004.
[53] Prateek Sharma, Tian Guo, Xin He, David Irwin, and Prashant Shenoy. Flint: Batch-interactive data-intensive processing on transient servers. In *Proceedings of the Eleventh European Conference on Computer Systems*, page 6. ACM, 2016.
[54] Ion Stoica, Robert Morris, David Karger, M Frans Kaashoek, and Hari Balakrishnan. Chord: A scalable peer-to-peer lookup service for internet applications. *ACM SIGCOMM Computer Communication Review*, 31(4):149–160, 2001.
[55] Radu Tudoran, Gabriel Antoniu, and Luc Bouge. Sage: geo-distributed streaming data analysis in clouds. In *2013 IEEE International Symposium on Parallel & Distributed Processing, Workshops and Phd Forum*, pages 2278–2281. IEEE, 2013.
[56] Shivaram Venkataraman, Aurojit Panda, Kay Ousterhout, Michael Armbrust, Ali Ghodsi, Michael J Franklin, Benjamin Recht, and Ion Stoica. Drizzle: Fast and adaptable stream processing at scale. In *Proceedings of the 26th Symposium on Operating Systems Principles*, pages 374–389. ACM, 2017.
[57] Matei Zaharia, Mosharaf Chowdhury, Tathagata Das, Ankur Dave, Justin Ma, Murphy McCauley, Michael J Franklin, Scott Shenker, and Ion Stoica. Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing. In *Proceedings of the 9th USENIX conference on Networked Systems Design and Implementation*, pages 2–2. USENIX Association, 2012.
[58] Matei Zaharia, Tathagata Das, Haoyuan Li, Timothy Hunter, Scott Shenker, and Ion Stoica. Discretized streams: Fault-tolerant streaming computation at scale. In *Proceedings of the twenty-fourth ACM symposium on operating systems principles*, pages 423–438. ACM, 2013.