Toward High-Performance Distributed Stream Processing via Approximate Fault Tolerance

Qun Huang
The Chinese University of Hong Kong
qhuang@cse.cuhk.edu.hk

Patrick P. C. Lee
The Chinese University of Hong Kong
pclee@cse.cuhk.edu.hk

Abstract

Fault tolerance is critical for distributed stream processing systems, yet achieving error-free fault tolerance often incurs substantial performance overhead. We present *AF-Stream*, a distributed stream processing system that addresses the trade-off between performance and accuracy in fault tolerance. AF-Stream builds on a notion called *approximate fault tolerance*, whose idea is to mitigate backup overhead by adaptively issuing backups, while ensuring that the errors upon failures are bounded with theoretical guarantees. Our AF-Stream design provides an extensible programming model for incorporating general streaming algorithms, and also exports only few threshold parameters for configuring approximation fault tolerance. Experiments on Amazon EC2 show that AF-Stream maintains high performance (compared to no fault tolerance) and high accuracy after multiple failures (compared to no failures) under various streaming algorithms.

1. INTRODUCTION

Stream processing becomes an important paradigm for processing data at high speed and large scale. As opposed to traditional batch processing that is designed for static data, stream processing treats data as a continuous stream, and processes every item in the stream in real-time. For scalability, we can make stream processing *distributed*, by processing streaming data in parallel through multiple processes (or workers) or threads.

Given that failures can happen at any time and at any worker in a distributed environment, fault tolerance is a critical requirement in distributed stream processing. In particular, streaming algorithms often keep internal states in main memory, which is fast but unreliable. Also, streaming data is generated continuously in real-time, and will become unavailable after being processed. Thus, we need to provide fault tolerance guarantees for both internal states and streaming data. One approach is to issue regular *backups* for internal states and streaming data, so that when failures happen, they can recover from backups and resume processing as normal without any error. However, frequent backups can incur significant network and disk I/Os, thereby compromising the stream processing performance. Some stream processing systems (e.g., S4 [44] and Storm [55]) aim for best-effort fault tolerance to trade accuracy for performance, but can incur unbounded errors and hence compromise the correctness of outputs.

We propose *AF-Stream*, a distributed stream processing system that addresses the trade-off between performance and accuracy in fault tolerance. AF-Stream builds on a notion called *approximate fault tolerance*, whose idea is to adaptively issue backup operations for both internal states and unprocessed items, while incurring only *bounded* errors after failures are recovered. Specifically, AF-Stream estimates the errors upon failures with the aid of user-specified interfaces, and issues a backup operation only when the errors go beyond the user-defined acceptable level.

We justify the trade-off with two observations. First, to mitigate computational complexities, streaming algorithms tend to produce "quick-and-dirty" results rather than exact ones (e.g., data synopsis) (§2.2). It is thus feasible to incur small additional errors due to approximate fault tolerance, provided that the errors are bounded. Also, the errors can often be amortized after the processing of large-volume and high-speed data streams. Second, although failures are prevalent in distributed systems, their occurrences remain relatively infrequent over the lifetime of stream processing. Thus, incurring errors upon failures should bring limited disturbance. We point out that the power of approximation has been extensively addressed in distributed computing (e.g., [2, 3, 35, 47, 49, 59, 60]). To the best of our knowledge, AF-Stream is the first work that leverages approximation in achieving fault tolerance in distributed stream processing. Our contributions are summarized as follows.

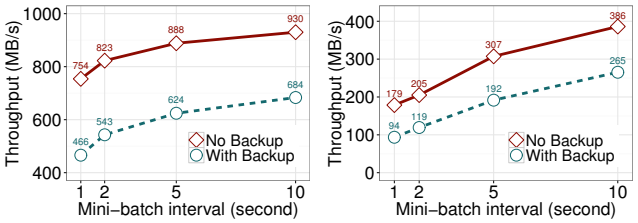
First, AF-Stream provides an extensible programming model for general streaming algorithms. In particular, it exports built-in interfaces and primitives that make fault tolerance intrinsically supported and transparent to programmers.

Second, AF-Stream realizes approximate fault tolerance, which bounds errors upon failures in two aspects: state divergence and number of lost items. We prove that the errors are bounded independent of the number of failures and the number of workers in a distributed environment. Also, the error bounds are tunable with only *three* user-specified parameters to trade between performance and accuracy. Note that AF-Stream adds no error to the common case when failures never happen.

Third, we implement an AF-Stream prototype, with emphasis on optimizing its inter-thread and inter-worker communications.

Finally, we evaluate the performance and accuracy of our AF-Stream prototype on a 10 Gb/s Amazon EC2 cluster for various streaming algorithms. AF-stream only degrades the throughput by up to 4.7%, 5.2%, and 0.3% in heavy hitter detection, online join, and online logistic regression, respectively, when compared to disabling backups; meanwhile, its accuracy after 10 failures only drops by a small percentage (based on algorithm-specific metrics) when compared to without failures.

Proceedings of the VLDB Endowment, Vol. 10, No. 3



(a) Grep (b) WordCount

Figure 1: Throughput of Spark Streaming with and without backups on Amazon EC2.

2. STREAMING ALGORITHMS

We first overview general streaming algorithms. We narrow down our study to three classes of streaming algorithms, in which we identify their commonalities to guide our system design.

2.1 Overview and Motivation

A streaming algorithm comprises a set of *operators* for processing a data stream of *items*. Each operator continuously receives an input item, processes the item, and produces one or multiple output items. It also keeps an in-memory *state*, which holds a collection of values corresponding to the processing of all received input items. The state is updated after each input item is processed. Also, an operator produces new items with respect to the updated state. Since items are produced and processed asynchronously and will become unavailable after being processed, an operator often buffers items until they are fully processed.

If failures never happen, we refer to the produced state and output as the *ideal state* and *ideal output*, respectively. However, the actual deployment environment is failure-prone. In this case, an operator needs to generate its *actual state* and *actual output* in a fault-tolerant manner. Specifically, an operator needs to handle: (i) missing in-memory state and (ii) missing unprocessed items.

Suppose that we want to achieve error-free fault tolerance for an operator, and ensure that both ideal and actual cases are identical after recovery. This necessitates the availability of a prior state and any following items, so that when a failure happens, we can retrieve the prior state and resume the processing of the following items. In particular, in stream processing, the items to be processed are generated continuously, so we need to make periodic state backups and issue a backup for *every* item in order to achieve error-free fault tolerance. For example, Spark Streaming saves each state as a Resilient Distributed Dataset (RDD) [62] in a mini-batch fashion, and also saves each item via write-ahead logging [63].

However, making regular backups for both the state and each item incurs excessive I/O overhead to normal processing. We motivate this claim by evaluating the backup overhead of Spark Streaming (v1.4.1) versus the mini-batch interval (i.e., the duration within which items are batched). Note that the performance of Spark Streaming is sensitive to the number of items in a mini-batch: having small mini-batches aggravates backup overhead, while having very large mini-batches increases the processing time to even exceed the mini-batch interval and hence makes the system unstable [15]. Thus, for a given mini-batch interval, we tune the stream input rate that gives the maximum stable throughput. Figure 1 shows the throughput of Spark Streaming for Grep and WordCount, measured on Amazon EC2 (see §4 for the details on the datasets and experimental setup). We see that the throughput drops significantly due to backups, for example, by nearly 50% for WordCount when the mini-batch interval is 1 second.

2.2 Classes of Streaming Algorithms

This paper focuses on three classes of streaming algorithms that have been well studied and widely deployed.

Data synopsis. Data synopsis algorithms summarize vital information of large-volume data streams into compact in-memory data structures with low time and space complexities. Examples include sampling, histograms, wavelets and sketches [12], and have been used in areas such as anomaly detection in network traffic [13, 17] and social network analysis [53]. To bound memory usage of the data structures, data synopsis algorithms are often designed to return estimates with bounded errors. For example, sampling algorithms (e.g., [17]) perform computations on a subset of items; sketch-based algorithms (e.g., [13, 17]) map a large key space into a fixed-size two-dimensional array of counters.

Stream database queries. Stream databases manage data streams with SQL-like operators as in traditional relational databases, and allow queries to be continuously executed over unlimited streams. Since some SQL operators (e.g., join, sorting, etc.) require multiple iterations to process items, stream database queries need to adapt the semantics of SQL operators for streaming data. For example, they restrict the processing of items over a time window, or return approximate query results using data synopsis techniques (e.g., sampling in online join queries [21, 40]).

Online machine learning. Machine learning aims to model the properties of data by processing the data (possibly over iterations) and identifying the optimal parameters toward some objective function. It has been widely used in web search, advertising, and analytics. Traditional machine learning algorithms assume that all data is available in advance and iteratively refine parameters towards a global optimization objective on the entire dataset. To support streaming data, online machine learning algorithms define a local objective function for each item with respective to the current model parameters, and search for the locally optimal parameters. After a large number of items are processed, it has been shown that the local approach can converge to a global optimal point, subject to certain conditions [24, 25, 65].

2.3 Common Features

We identify the common features of existing streaming algorithms to be addressed in our system design. Table 1 elaborates how our design choices are related to the common features.

Common primitive operators. We can often decompose an operator of a streaming algorithm into a small number of *primitive operators*, which form the building blocks of the same class of streaming algorithms. For example, stream database queries are formed by few operators such as map, union, and join [1, 20, 48]; sketch-based algorithms are formed by hash functions, matching, and numeric arrays [61].

Intensive and skewed updates. The state of an operator often holds *update-intensive* values, such as sketch counters [61] and model parameters in online machine learning [35]. Also, state values are updated at different frequencies. For example, the counters corresponding to frequent items are updated at higher rates; in online machine learning, only few model parameters are frequently updated due to the sparsity nature of machine learning features [35].

Bounding by windows. Streaming algorithms often work on a bounded sequence of items of a stream. For example, some operators of stream database queries process a time window of items (§2.2); some incremental processing systems (e.g., [8, 22, 38, 62]) divide a stream into mini-batches for processing.

Table 1: Common features of streaming algorithms and their corresponding design choices.

Common features	Corresponding design choices
Common primitive operators	AF-Stream abstracts a streaming algorithm into a set of operators and maintains fault tolerance for each operator (§3.2). It also realizes a rich set of built-in primitive operators (§3.6).
Intensive and skewed state updates	AF-Stream supports partial state backup to reduce the backup size (§3.4.1). It also exposes interfaces to let users specify which parts of a state are actually included in a backup (§3.3).
Bounding by windows	AF-Stream resets thresholds based on windows (§3.5.2). It also exposes interfaces to let users specify window types and lengths (§3.3).

2.4 Distributed Implementation

For scalability, we can parallelize a streaming algorithm through a distributed implementation. We identify two common distributed approaches, which can be used individually or in combination.

Pipelining divides an operator into multiple stages, each of which corresponds to an operator or a primitive operator (§2.3). The output items of one stage can serve as input items to the next stage, while different stages can process different items in parallel.

Operator duplication parallelizes stream processing through multiple copies of the same operator. There are two ways to distribute loads across operator copies: *data partitioning*, in which each operator copy processes a subset of data items of a stream, and *state partitioning*, in which each operator copy manages a subset of values of a state. Both approaches can be used simultaneously in a streaming algorithm.

3. AF-STREAM DESIGN

AF-Stream abstracts a streaming algorithm as a set of operators. It maintains fault tolerance for each operator by making backups for its state and unprocessed items. To realize approximate fault tolerance, AF-Stream issues a backup operation only when the actual state and output deviate much from the ideal state and output, respectively. This mitigates backup overhead, while incurring bounded errors after failures are recovered.

AF-Stream's approximate fault tolerance inherently differs from existing backup-based approaches for distributed stream processing. Unlike the approaches that achieve error-free fault tolerance (e.g., [4,30,48]), AF-Stream issues fewer backups, thereby improving stream processing performance. Unlike the approaches that achieve best-effort fault tolerance by also making fewer backups (e.g., [44,55]), AF-Stream ensures that the errors are bounded with theoretical guarantees. AF-Stream also differs from approximate processing approaches (§5) in that it only makes approximations in maintaining fault tolerance rather than in normal processing.

3.1 Design Assumptions

To bound the errors upon failures, AF-Stream makes the following assumptions on streaming algorithms.

AF-Stream assumes that a single lost item (without any backup) brings limited degradations to accuracy. Instead of focusing on identifying a specific item (e.g., finding an outlier item), AF-Stream is designed to analyze the aggregated behavior over a large-volume stream of items, so each item has limited impact on the overall analysis. For example, in network monitoring, we may want to identify the flows whose sums of packet sizes exceed a threshold. Each item corresponds to a packet, whose maximum size is typically limited by the network (e.g., 1,500 bytes). Note that this assumption is also made by existing stream processing systems that build on approximation techniques (§5).

Note that after we recover a failed operator, there may be errors when the restored operator resumes processing from an actual state instead of from the ideal state. Nevertheless, such errors can often be amortized or compensated after processing a sufficiently large number of items. For example, online machine learning algorithms can converge to the optimal solution even if they start from a non-ideal state [23, 34, 52]. Thus, this type of errors brings limited accuracy degradations, as also validated by our experiments (§4).

Finally, AF-Stream assumes that the errors across multiple duplicate operator copies can be aggregated. For example, machine learning algorithms maintain linearly additive states [35], thereby allowing the errors of multiple copies to be summable.

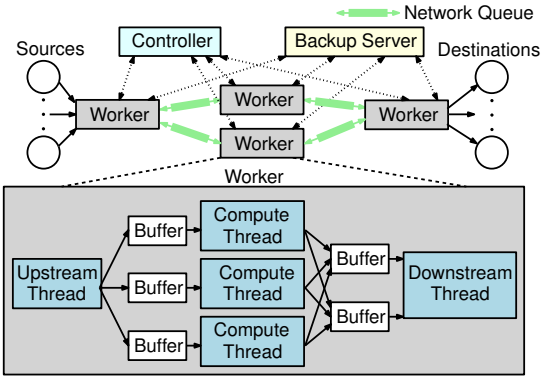


Figure 2: AF-Stream architecture.

3.2 Architecture

Figure 2 shows the architecture of AF-Stream. AF-Stream comprises multiple processes, including a single *controller* and multiple *workers*. Each worker manages a single operator of a streaming algorithm, while the controller coordinates the executions of all workers. AF-stream organizes workers as a graph, in which one or multiple sources originate data streams, and one or multiple destinations store the final results. For a pair of neighboring workers, say w_1 and w_2, we call w_1 an *upstream* worker and w_2 a *downstream* worker if the stream processing directs from w_1 to w_2. Specifically, a worker receives input items from either a source or an upstream worker, processes the items, and forwards output items to either a destination or a downstream worker.

AF-Stream also supports the *feedback* mechanism, which is essential for some streaming algorithms (e.g., model convergence in online machine learning [34]). It allows a downstream worker to optionally send feedback messages to an upstream worker. In other words, the communication between each pair of neighboring workers is bi-directional.

The controller manages the execution of each worker, which periodically sends heartbeats to the controller. If a worker fails, the controller recovers the failed state and data in a new worker. Also, each worker issues backups to a centralized *backup server*, which keeps backups in reliable storage. The backup server should be viewed as a *logical* entity that can be substituted with any external storage system (e.g., HDFS [51]). This paper assumes that the controller and the backup server are always available and have suf-

ficient computational resources, yet we can deploy multiple controllers and backup servers for fault tolerance and scalability.

Each worker in AF-Stream comprises one upstream thread, one downstream thread, and multiple compute threads. The upstream thread forwards input items from either a source or an upstream worker to one of the compute threads, while the downstream thread forwards the output items from the compute threads to either a destination or a downstream worker. In particular, multiple compute threads can collaboratively process items, such that a compute thread can partially process an item and forward the intermediate results to another compute thread for further processing. Furthermore, the downstream thread can collect and forward any feedback message from a downstream worker to the compute threads for processing, and the upstream thread can forward any new feedback message from the compute threads to an upstream worker. Our implementation experience is that it suffices to have only one upstream thread and one downstream thread per worker to achieve the required processing performance. Thus, we can reserve the remaining CPU cores for compute threads to perform heavy-weight computations.

AF-Stream connects workers and threads as follows. For inter-worker communications, AF-Stream connects each pair of upstream and downstream workers via a bi-directional network queue. For inter-thread communications, AF-Stream shares data across threads via in-memory circular ring buffers. We carefully optimize both network queue and ring buffer implementations so as to mitigate communication overhead (§3.6).

3.3 Programming Model

AF-Stream manages two types of objects: *states* and *items* (§2.1), whose formats are user-defined. Each operator is associated with a state, and each state holds an array of binary values. Also, each operator supports three types of items: (i) *data items*, which collectively refer to the input and output items that traverse along workers from upstream to downstream in stream processing, (ii) *feedback items*, which traverse along workers from downstream to upstream, and (iii) *punctuation items*, which specify the end of an entire stream or a sub-stream for windowing (§2.3).

AF-Stream has two sets of interfaces: *composing interfaces* and *user-defined interfaces*. We list them in our technical report [27] in the interest of space.

The composing interfaces are used to define the AF-Stream architecture and the stream processing workflows. Their functionalities can be summarized as follows: (i) connecting workers (in server hostnames) and the source/destination (in file pathnames), (ii) adding threads to each worker, (iii) connecting threads within each worker, (iv) pinning a thread to a CPU core, and (v) specifying the windowing type (e.g., hopping window, sliding window, decaying window) and window length.

On the other hand, user-defined interfaces allow programmers to add specific implementation details. AF-Stream automatically calls the user-defined interfaces and processes items based on their implementations. Specifically, the upstream thread can be customized to receive items, dispatch them to the compute threads, and optionally send feedback items to upstream workers. Similarly, the downstream thread can be customized to send output items and optionally receive feedback items from downstream workers. The compute thread can be customized to process data items, feedback items, and punctuation items.

AF-Stream provides three user-defined interfaces for building operators that realize approximate fault tolerance for state backup (§3.4.1): (i) `StateDivergence`, which quantifies the divergence between the current state and the most recent backup state, (ii) `BackupState`, by which operators provide the state to be saved in reliable storage via the backup server, and (iii) `RecoverState`, by which operators obtain the most recent backup state from the backup server. For example, suppose that we implement a sketch-based algorithm and maintain a fixed-length array of counters as the state (§2.2). Then `StateDivergence` can specify the divergence as the Manhattan distance, Euclidean distance, or maximum difference of counter values; `BackupState` can issue backups for the array of counter values; and `RecoverState` can restore the array of counter values.

3.4 Approximate Fault Tolerance

AF-Stream maintains approximate fault tolerance for both the state and items of each operator. We introduce both state backup (§3.4.1) and item backup (§3.4.2) as individual backup mechanisms that are complementary to each other and are configured by different thresholds.

3.4.1 State Backup

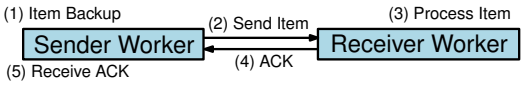
Recall that AF-Stream calls `BackupState` to issue a backup operation for the state of an operator to the backup server. Instead of making frequent calls to `BackupState`, AF-Stream defers the call to `BackupState` until the current state deviates from the most recent backup state by some threshold (denoted by θ). Specifically, AF-Stream caches a copy of the most recent backup state of the operator in local memory. Each time when an item updates the current state, AF-Stream calls `StateDivergence` to compute the divergence between the current state and the cached backup state. If the divergence exceeds θ, AF-Stream calls `BackupState` to issue a backup for the current state, and updates the cached copy accordingly.

Making the backup of an entire state may be expensive, especially if the state is large. To further mitigate backup overhead, AF-Stream supports the backup of a *partial* state, such that programmers can only specify the list of updated values (together with their indices) that make a state substantially deviate as the returned state of `BackupState`. Partial state backup is useful when only few values of a state are updated (§2.3).

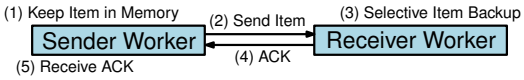
Note that each state update triggers a call to `StateDivergence`. For most operators and divergence functions (e.g., difference of cardinalities, Manhattan distance, Euclidean distance, or maximum difference of values), `StateDivergence` only involves few arithmetic operations. For example, suppose that a divergence function returns the sum of differences (denoted by D) of all values between two states. If we update a value, then we can compute the new sum of differences (denoted by D') as $D' = D + \Delta$, where Δ is the change of the value. Thus, if the operator has only one compute thread, the compute thread itself can evaluate the divergence with limited overhead. On the other hand, if the operator has multiple compute threads to process a state simultaneously, say by operator duplication (§2.4), then summing the divergence of all compute threads can be expensive due to inter-thread communications. AF-Stream currently employs a smaller threshold in each compute thread, such that the sum of the thresholds does not exceed θ. For example, we can set the threshold as θ/n if there are $n > 1$ compute threads.

3.4.2 Item Backup

AF-Stream makes item backups *selective*, such that a worker decides to make a backup for an item based on the item type and a pre-defined threshold. For a punctuation item, a worker always makes a backup for it to accurately identify the head and tail of a sequence to be processed; for a data item or a feedback item, the



(a) Upstream backup



(b) Our receiver-side backup

Figure 3: Comparisons between upstream backup and our receiver-side backup.

worker counts the number of pending items that have not yet been processed. If the number exceeds a threshold (denoted by l), the worker makes backups for all pending items so as to bound the number of missing items upon failures.

In AF-Stream, item backups are issued by the receiver-side worker (i.e., a downstream worker handles the backups of data and punctuation items, while an upstream worker handles the backups of feedback items). The rationale is that the receiver-side worker can exactly count the number of unprocessed items and decide when to issue item backups. Consider a data item that traverses from an upstream worker to a downstream worker. After the upstream worker sends the data item, it keeps the data item in memory. When the downstream worker receives the data item, it examines the number of unprocessed data items. If the number exceeds the threshold l, then the worker makes backups for all pending data items before dispatching the data item to the compute threads for processing. It also returns an acknowledgment (ACK) to the upstream worker, which can then release the cached data item from memory.

Performing item backups on the receiver side enables us to handle ACKs differently from the upstream backup approach [4, 30, 48], as shown in Figure 3. Upstream backup makes item backups on the sender side. For example, an upstream worker is responsible for making item backups for data items. It caches the backup items in memory and waits until its downstream worker replies an ACK. However, the downstream worker sends the ACK only when the item is completely processed, and the upstream worker may need to cache the item for a long time. In contrast, our receiver-side approach can send an ACK before an item is processed, and hence limits the caching time in the upstream worker.

The rate of item backups in AF-Stream is responsive to the current load of a worker. Specifically, when a worker is heavily loaded, it will accumulate more unprocessed items and hence trigger more backups. Nevertheless, the backup overhead remains limited since item processing is the bottleneck in this case. On the other hand, when a worker is lightly loaded, it issues fewer backups and avoids compromising the processing performance.

Note that the sender-side worker may miss all unacknowledged items if it fails. Currently AF-Stream does not make backups for such unacknowledged items, but instead bounds the maximum number of unacknowledged items (denoted by γ) that a worker can cache in memory. If the number of unacknowledged items reaches γ, then the worker is blocked from sending new items until the number drops. The value of γ can be generally very small as our receiver-side approach allows a worker to reply an ACK immediately upon receiving an item.

3.4.3 Recovery

If the controller detects a failed worker, it activates recovery and restores the state of the failed worker in a new worker. The new worker calls `RecoverState` to retrieve the most recent backup state. Also, AF-Stream replays the backup items into the new worker, which can then process them to update the restored state.

Some backup items may have already been processed and updated in the recovered state, and we should avoid the duplicate processing of those items. AF-Stream uses the sequence number information for backup and recovery. When a worker receives an item, it associates the item with a sequence number. Each state also keeps the sequence number of the latest item that it includes. When AF-Stream replays backup items during recovery, it only chooses the items whose sequence numbers are larger than the sequence number kept in the restored state. To reduce the recovery time, our current implementation restores fewer items by replaying only the most recent sequence of consecutive items. Since the number of unprocessed items before the replayed sequence is at most l, the errors are still bounded.

3.4.4 User-Configurable Thresholds

AF-Stream exports three user-configurable threshold parameters: (i) Θ, the maximum divergence between the current state and the most recent backup state, (ii) L, the maximum number of unprocessed non-backup items and (iii) Γ, the maximum number of unacknowledged items. It automatically *tunes* the thresholds θ, l, and γ at runtime with respect to Θ, L and Γ, respectively, such that the errors are bounded independent of the number of failures that have occurred and the number of workers in a distributed environment. In Appendix, we present both theoretical analysis and numerical examples on how these parameters are translated to the accuracy of some streaming algorithms.

AF-Stream currently requires users to have domain knowledge on configuring the parameters with respect to the desired level of accuracy. We pose the issue of configuring the parameters without user intervention as future work.

3.5 Error Analysis

We analyze how AF-Stream bounds the errors upon failures for different failure scenarios in a distributed environment. We quantify the error bounds in two aspects: (i) the divergence between the actual and ideal states and (ii) the number of lost output items, based on our assumptions (§3.1). In particular, we assume that each lost item brings limited accuracy degradations. To quantify the degradations, after updating the current state with an item, we let the divergence between the current state and the most recent backup state increase by at most α and the number of output items generated by the update is at most β, where α and β are two constants specific for a streaming algorithm. Note that both α and β are introduced purely for our analysis, while programmers only need to configure Θ, L, and Γ for building a streaming algorithm. Also, our analysis does not assume any probability distribution of failure occurrences.

3.5.1 Single Failure of a Worker

Suppose that AF-Stream sees only a single failure of a worker over its lifetime. We consider the worst case when the failed worker is restored, as shown in Figure 4. First, for the divergence between the actual and ideal states, the restored state diverges from the actual state before the failure by at most θ, and each of the lost unprocessed items (with l at most) changes the divergence by at most α. Thus, the total divergence between the actual and ideal states is at most $\theta + l\alpha$, which is bounded.

Second, for the number of lost output items, AF-Stream loses at most γ unacknowledged output items, and each of the unprocessed items (with l at most) leads to at most β lost output items. Thus, the total number of lost output items is at most $\gamma + l\beta$, which is also bounded.

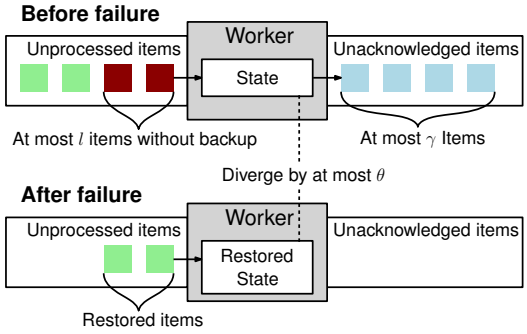


Figure 4: Errors due to a single failure of a worker.

3.5.2 *Multiple Failures of a Single Worker*

Suppose that AF-Stream sees multiple failures of a single worker over its lifetime, while other workers do not fail. In this case, the errors after each failure recovery will be accumulated. AF-Stream bounds the accumulated errors by adapting the three thresholds θ, l, and γ with respect to the three user-specified parameters Θ, L and Γ, respectively, and the number of failures denoted by k. First, a worker initializes the thresholds with $\theta = \frac{\Theta}{2}$, $l = \frac{L}{2}$, and $\gamma = \frac{\Gamma}{2}$. After each failure recovery, the worker halves each threshold; in other words, after recovering from k failures, the thresholds become $\theta = \frac{\Theta}{2^{k+1}}$, $l = \frac{L}{2^{k+1}}$, and $\gamma = \frac{\Gamma}{2^{k+1}}$. By summing up the errors accumulated over k failures, we can show that the divergence between the actual and ideal states is at most $\Theta + L\alpha$ and the number of lost output items is at most $\Gamma + L\beta$. Note that the errors are bounded independent of the number of failures k.

Our adaptations imply that the thresholds become very small after many failures, so AF-Stream reduces to error-free fault tolerance and makes frequent backups. AF-Stream addresses this issue by allowing the thresholds to be reset. In particular, many streaming algorithms work with windowing, and AF-Stream can use different strategies to reset the thresholds for different windowing types (§3.3). For example, for the hopping window, AF-Stream resets the thresholds to the initial values $\theta = \frac{\Theta}{2}$, $l = \frac{L}{2}$, and $\gamma = \frac{\Gamma}{2}$ at the end of a window; for the sliding window, AF-Stream tracks the time of the last failure and increases the thresholds once a failure is not included in the window; for the decaying window, AF-Stream always keeps $\theta = \Theta$, $l = L$, and $\gamma = \Gamma$ and disables threshold adaptations, as the errors fade over time.

3.5.3 *Failures in Multiple Workers*

We address the general case when a general number of failures can happen in multiple workers in a distributed environment. To make the error bounds independent of the number of workers, AF-Stream employs small initial thresholds, such that the accumulated errors are the same as those in the single-worker scenario. Specifically, for operator duplication with n copies, AF-Stream initializes each copy with $\theta = \frac{\Theta}{2n}$, $l = \frac{L}{2n}$, and $\gamma = \frac{\Gamma}{2n}$; for a pipeline with m operators, AF-Stream initializes the i-th operator in the pipeline with $\theta = \frac{\Theta}{2\beta^{m-i}}$, $l = \frac{L}{2\beta^{m-i}}$, and $\gamma = \frac{\Gamma}{2\beta^{m-i}}$ (since each lost item can lead to at most β^m lost output items after m pipeline stages). By summing the errors over all failures and all workers, we can obtain the same error bounds as in §3.5.2.

3.5.4 *Discussion*

Our analysis shows that the incurred errors due to approximate fault tolerance are bounded in terms of state divergence and number of lost output items. How the error bounds quantify the accuracy of a streaming algorithm is specific to the algorithmic design and cannot be directly generalized for all algorithms. In Appendix, we theoretically analyze two specific algorithms, namely heavy-hitter detection in Count-Min Sketch [14] and Ripple Join [21], and show how their accuracy will be degraded under approximate fault tolerance with respect to Θ, L, and Γ. We also resort to experiments to empirically evaluate the accuracy for various parameter choices in AF-Stream (§4).

3.6 Implementation

We have implemented a prototype of AF-Stream in C++. AF-Stream connects the controller and all workers using ZooKeeper [28] to manage fault tolerance. It also includes a backup server, which is now implemented as a daemon that receives backup states and items from the workers via TCP and writes them to its local disk. While we currently implement AF-Stream as a clean-slate system, we explore how to integrate our approximate fault tolerance notion into existing stream processing systems in future work. Our current prototype has over 45,000 LOC in total.

Our prototype realizes a number of built-in primitive operators and their corresponding implementations of `StateDivergence`, `BackupState`, and `RecoverState`, as listed in Table 2. We now support the numeric variable, vector, matrix, hash table, and set, and provide them with built-in fault tolerance. For example, we can keep elements in our built-in hash table, whose fault tolerance is automatically enabled. Programmers can also build their own fault-tolerant operators via implementing the above three interfaces.

Communication optimization. Our prototype specifically optimizes both inter-thread and inter-worker communications for high-throughput stream processing. For inter-thread communications, we implement a lock-free multi-producer, single-consumer (MPSC) ring buffer [43]. We assign one MPSC ring buffer per destination, and group output items from different compute threads by destinations (§3.2). This offloads output item scheduling to compute threads, and simplifies subsequent inter-worker communications. Also, we only pass the pointers to the items to the ring buffer, so that the rate of the number of items that can be shared is independent of the item size.

For inter-worker communications, we implement a bi-directional network queue with ZeroMQ [64]. ZeroMQ itself uses multiple threads to manage TCP connections and buffers, and the thread synchronization is expensive. Thus, we modify ZeroMQ to remove its thread layer, and make the upstream and downstream threads of a worker directly manage TCP connections and buffers. The modifications enable us to achieve high-throughput stream processing in a 10 Gb/s network (§4).

Asynchronous backups. We further mitigate the performance degradation of making backups using an asynchronous technique similar to [19]. Specifically, our prototype employs a dedicated backup thread for each worker. It collects all backups of a worker and sends them to the backup server, allowing other threads proceed normal processing after generating backups. Note that the asynchronous technique does not entirely eliminate backup overhead, since the backup thread can still be overloaded by frequent backup operations. Thus, we propose approximate fault tolerance to limit the number of backup operations.

4. EXPERIMENTS

We evaluate AF-Stream on an Amazon EC2 cluster located in the `us-east-1b` zone. The cluster comprises a total of 12 instances: one `m4.xlarge` instance with four CPU cores and 16 GB RAM, and 11 `c3.8xlarge` instances with 32 CPU cores and 60 GB RAM each. We deploy the controller and the backup server in the

Table 2: Built-in fault-tolerant primitive operators.

Operator	`StateDivergence`	`BackupState`	`RecoverState`
Numeric variables	Difference of two values	Returns the variable value	Assigns the variable value
Vector and matrix	Manhattan distance; Euclidean distance; or maximum difference of values of an index	Returns a list of (index, value) pairs and the length of the list	Fills in restored (index, value) pairs
Hash table	Manhattan distance, Euclidean distance, maximum difference of values of a key, difference of the numbers of keys	Returns a list of (key, value) pairs and the length of the list	Inserts the restored (key, value) pairs
Set	Difference of the numbers of keys	Returns a list of set members and the length of the list	Inserts the restored set members

Table 3: Summary of streaming algorithms in our evaluation.

Algorithm	Upstream stage (# workers)	Downstream stage (# workers)	State (Divergence)	Trace		
				Source	# items	Size
Grep	Parse and send the lines with the matched pattern (10)	Merge matched lines (1)	None	Gutenberg [46]	300M	15 GB
WordCount	Parse and send words with intermediate counts (4)	Aggregate intermediate word counts (7)	Hash table of word counts (maximum difference of word counts)	Gutenberg [46]	300M	15 GB
HH detection	Update packet headers into a local sketch and send the local sketch and local HHs (10)	Merge local sketches into a global sketch, and check if local HHs are actual HHs with the global sketch (1)	Matrix of counters (maximum difference of counter values in bytes)	Caida [9]	1G	40 GB
Online join	Find and send tuples that have matching packet headers in two streams (2)	Perform join and aggregation (9)	Set of sampled items (difference of number of packets)	Caida [9]	1G	40 GB
Online LR	Train and send the local model, and update the local model with feedback items (10)	Merge local models to form a global one, and send the global model as feedback items (1)	Hash table of model parameters (Euclidean distance)	KDD Cup 2012 [45]	110M	42 GB

`m4.xlarge` instance, and a worker in each of the `c3.8xlarge` instances. We connect all instances via a 10 Gb/s network.

Our experiments consider five streaming algorithms: Grep, WordCount, heavy hitter detection, online join, and online logistic regression. The latter three algorithms are chosen as the representatives for data synopsis, stream database queries, and online machine learning, respectively (§2.2). We pipeline each streaming algorithm in two stages, in which the upstream stage reads traces, processes items, and sends intermediate outputs to the downstream stage for further processing. Each stage contains one or multiple workers. We evenly partition a trace into subsets and assign each subset to an upstream worker, which partitions its intermediate outputs to different downstream workers if more than one downstream worker is used. Table 3 summarizes each algorithm, including the functionalities and number of workers for both upstream and downstream stages, the definitions of the state and the corresponding state divergence, as well as the source, number of items, and size of each trace. We elaborate the algorithm details later when we present the results.

Each experiment shows the average results over 20 runs. Before each measurement, we load traces into the RAM of each upstream worker, which then reads the traces from RAM during processing. This eliminates the overhead of reading on-disk traces, and moves the bottleneck to AF-Stream itself. In some of our experiments, we may observe throughput on the order of GB/s.

We evaluate the throughput and accuracy of AF-Stream for each algorithm, and show the trade-off with respect to Θ, L, and Γ. For throughput, we measure the rate of the amount of data processed in each upstream worker, and compute the sum of rates in all upstream workers as the resulting throughput. For accuracy, we provide the specific definition for each algorithm when we present the results.

4.1 Baseline Performance

We benchmark the baseline performance of AF-Stream using two algorithms: Grep, which returns the input lines that match a pattern, and WordCount, which counts the occurrence frequency of each word. Note that Grep does not need any state to be kept in a worker, while WordCount defines a state as a hash table of word counts. We implement both algorithms based on their implementations in open-source Spark Streaming [62]. We use the documents on Gutenberg [46] as the inputs, with the total size 15 GB.

Experiment 1 (Comparisons with existing fault tolerance approaches). We compare AF-Stream with two open-source stream processing systems: Storm [55] and Spark Streaming [62]. We consider three setups for each of them. The first setup disables fault tolerance to provide baseline results. The second setup uses their own available fault tolerance mechanisms. Specifically, Storm tracks every item until it is fully processed (via a component called `Acker`), yet it only achieves only best-effort fault tolerance as it does not support state backups. On the other hand, Spark Streaming achieves error-free fault tolerance by making state backups as RDDs in mini-batches and making item backups via write-ahead logging [63]. We set the mini-batch interval of Spark Streaming as 1 second (see details in §2.1). For both setups, we configure the systems to read traces from memory to avoid the disk access overhead. Finally, the third setup achieves fault tolerance through Kafka [6], a reliable messaging system that persists items to disk for availability. We configure the systems to read input items from the disk-based storage of Kafka, so that Kafka serves as an extra item backup system to replay items upon failures. We configure the fault tolerance mechanisms of Storm and Spark Streaming based on the official documentations; for Kafka integration, we refer to [54, 56]. In particular, Kafka and write-ahead logging in Spark Streaming have the same functionality, so for performance efficiency, it is suggested to disable write-ahead logging when Kafka is used [54]. On the other hand, other fault tolerance approaches can be used in conjunction with Kafka. We present all combinations in our results.

Table 4: Experiment 1 (Comparisons with existing fault tolerance approaches).

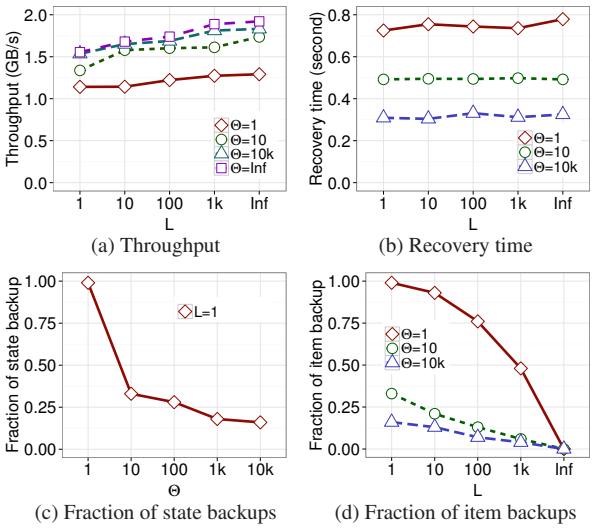
	Grep	WordCount
Storm		
No fault tolerance	262.04 MB/s	901.47 MB/s
With item backup only	100.65 MB/s	571.31 MB/s
With Kafka only	85.79 MB/s	192.66 MB/s
With Kafka + item backup	80.14 MB/s	174.30 MB/s
Spark Streaming		
No fault tolerance	178.61 MB/s	754.02 MB/s
With RDD + write-ahead logging	93.61 MB/s	466.07 MB/s
With Kafka only	81.49 MB/s	147.28 MB/s
With Kafka + RDD	75.09 MB/s	140.80 MB/s
AF-Stream implementation		
No fault tolerance	3.55 GB/s	1.92 GB/s
Mini-batch	358.16 MB/s	380.49 MB/s
Upstream backup	379.61 MB/s	373.89 MB/s
Approximate fault tolerance	3.48 GB/s	1.88 GB/s

In addition, we implement existing fault tolerance approaches in AF-Stream and compare them with approximate fault tolerance under the same implementation setting. We consider *mini-batch* and *upstream backup*. Mini-batch follows Spark Streaming [62] and makes backups for mini-batches. To realize the mini-batch approach in AF-Stream, we divide a stream into mini-batches with the same number of items, such that the number of items per mini-batch is the maximum number while keeping the system stable (§2.1). Our approach generates around 800 mini-batches. Our modified AF-Stream then issues state and item backups for each mini-batch. On the other hand, upstream backup [29] provides error-free fault tolerance by making backups in upstream workers. To realize upstream backup, we issue a backup for every data item, while we issue a state backup every 1% of data items. In addition to saving items via the backup server, both approaches also keep the items in memory for ACKs. Once the memory usage exceeds a threshold (1 GB in our case), we save any new item to local disk. Finally, we set $\Theta = 10^4$, $L = 10^3$, and $\Gamma = 10^3$ in AF-Stream for approximate fault tolerance.

Table 4 shows the throughput of different fault tolerance approaches. Both Storm and Spark Streaming see throughput drops when fault tolerance is used. Compared to the throughput without fault tolerance, even the most modest case degrades the throughput by around 37% (i.e., Storm's item backup for Grep). We find that the bottlenecks of both systems are mainly due to frequent item backups. Also, Kafka integration achieves even lower throughput since Kafka incurs extra I/Os to read traces from disk. In contrast, AF-Stream with approximate fault tolerance issues fewer backup operations. It achieves 3.48 GB/s for Grep and 1.78 GB/s for WordCount, both of which are close to when AF-Stream disables fault tolerance. Note that AF-Stream outperforms Spark Streaming and Storm even when they disable fault tolerance. The reason is that AF-Stream has a more simplified implementation.

Experiment 2 (Impact of thresholds on performance). We examine how the thresholds Θ, L, and Γ affect the performance of AF-Stream in different aspects. Since Grep does not keep any state, we focus on WordCount. We vary Θ and L, and fix $\Gamma = 10^3$. When Θ and L are sufficiently large (i.e., close to infinity), we in essence disable both state and item backups, respectively.

Figure 5(a) presents the throughput of AF-Stream versus L for different settings of Θ and the case of disabling state backups. Compared to disabling state backups, the throughput loss is 33% when $\Theta = 1$, but we reduce the loss to 10% by setting $\Theta = 10$.



(a) Throughput (b) Recovery time

(c) Fraction of state backups (d) Fraction of item backups

Figure 5: Experiment 2 (Impact of thresholds on WordCount).

Figure 5(b) also presents the recovery time when recovering a worker failure, starting from the time when a new worker process is resumed until it starts normal processing. A smaller Θ implies a longer recovery time, as we need to make backups for more updated state values in partial backup (§3.4.1). Nevertheless, the recovery time in all cases is less than one second.

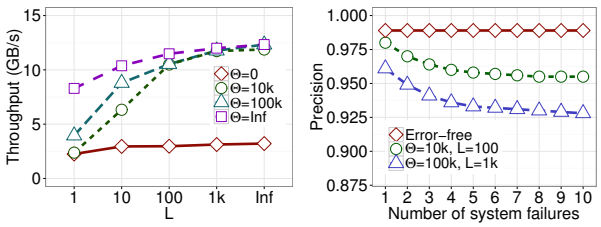
We reason the throughput by showing the fractions of state and item backup operations over the total number of items. Figure 5(c) shows that AF-Stream issues state backups for less than 30% of items when $\Theta \geq 10$ (where we fix $L = 1$). Figure 5(d) shows that increasing Θ also reduces the fraction of item backups (e.g., to less than 20% when $\Theta \geq 10$), mainly because the compute threads have more available resources to process items rather than perform state backups. This reduces the number of unprocessed items, thereby issuing fewer item backups.

4.2 Performance-accuracy Trade-offs

We examine how AF-Stream trades between performance and accuracy. We evaluate its throughput when no failure happens, while we evaluate its accuracy after recovering from system failures in which all workers fail. To mimic a system failure, we inject a special item in the stream to a worker. When the worker reads the special item, it sends a remote stop signal to kill all worker processes. We then resume all worker processes, recover all backup states, and replay the backup items. We inject the special item multiple times to generate multiple failures over the entire stream. We consider the other three streaming algorithms, which are more complicated than Grep and WordCount.

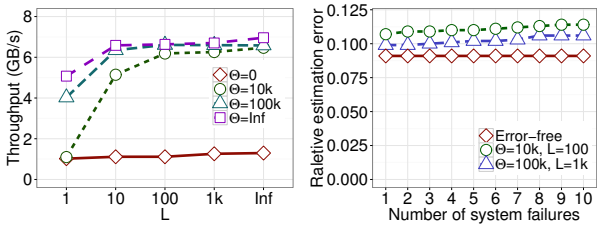
In the following, we vary Θ and L, and fix $\Gamma = 10^3$. Here, Γ represents the maximum number of unacknowledged items in upstream workers. We observe that the actual number of unacknowledged items is small and they have limited impact on both performance and accuracy in our experiments. Thus, we focus on the physical meanings of L and Θ in each algorithm and justify our choices of the two parameters.

Experiment 3 (Heavy hitter detection). We perform heavy hitter (HH) detection in a stream of packet headers from Caida [9], where the total packet header size is 40 GB. We define an HH as a source-destination IP address pair whose total payload size ex-



(a) Throughput (b) Precision

Figure 6: Experiment 3 (Heavy hitter detection).



(a) Throughput (b) Relative estimate error

Figure 7: Experiment 4 (Online join).

ceeds 10 MB. We implement HH detection based on Count-Min Sketch [14]. Each worker holds a Count-Min Sketch with four rows of 8,000 counters each as its state. Each packet increments one counter per row by its payload size. We maintain the counters in a matrix that has built-in fault tolerance (§3.6). Also, an upstream worker sends local sketches and detection results to a downstream worker as punctuation items, for which we ensure error-free fault tolerance (§3.4.2).

In our implementation, Θ is the maximum difference of counter values (in bytes) between the current state and the most recent backup state and L is the maximum number of unprocessed non-backup packets. We choose $\Theta \leq 10^5$ and $L \leq 10^3$. If the packet size is at most 1,500 bytes, our choices account for at most 1.5 MB of counter values, much smaller than our selected threshold 10 MB.

We measure the throughput of AF-Stream as the total packet header size processed per second. To measure the accuracy, one important fix is that we need to address the missing updates due to approximate fault tolerance. In particular, the missing updates cause the restored counter values to be smaller than the original counter values before a failure. To compensate the missing updates, we add each counter by $\Theta/2^k + L\alpha/2^k$ when restoring counter values after the k-th failure (where $k \geq 1$). This ensures the zero false negative rate of Count-Min Sketch, while increasing the false positive rate by a bounded value. We measure the accuracy by the precision, defined as the ratio of the number of actual HHs to the number of returned HHs (which include false positives).

Figure 6 presents the throughput and precision of AF-Stream for HH detection. If we disable both state and item backups (i.e., Θ and L are close to infinity), the throughput is 12.33 GB/s, and the precision is 98.9% when there is no failure. With approximate fault tolerance, if $\Theta = 10^5$ and $L = 10^3$, the throughput decreases by up to 4.7%, yet the precision only decreases to 92.8% after 10 system failures. If we set $\Theta = 10^4$ and $L = 100$, the throughput drops by around 15%, while the precision decreases to 95.5%.

Experiment 4 (Online join). Online join is a basic operation in stream database queries. This experiment considers an online join operation that correlates two streams of packet headers of different cities obtained from Caida [9], with the total packet header size 40 GB. Our goal is to return the tuples of destination IP address and timestamp (in units of seconds) that have matching packet headers in both streams, meaning that both streams visit the same host at about the same time. We partition the join operator into multiple workers, each of which runs a Ripple Join algorithm [21] to sample a subset of packets for online join at a sampling rate 10%. Each worker keeps the sampled packets in a set with built-in fault tolerance (§3.6).

Here, Θ represents the maximum difference of the numbers of packets between the current state and the most recent backup state, and L is the maximum numbers of unprocessed non-backup packets. We find that our sampling rate can obtain around 100 million packets. Thus, we set $\Theta \leq 10^5$ and $L \leq 10^3$ to account for at most 0.1% of sampled packets.

We again measure the throughput of AF-Stream as the total packet header size processed per second. To measure the accuracy, we issue an aggregation query for the total number of joined tuples. Ripple join returns the estimated number of tuples by dividing the number of joined tuples in the sampled set by the sampling rate. We measure the accuracy as the relative estimation error, defined as the percentage difference of the estimated number from the actual number without any sampling.

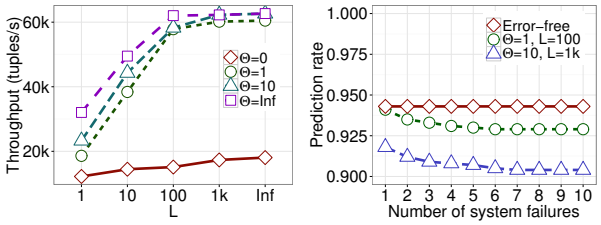
Figure 7 presents the throughput and relative estimation error of AF-Stream for online join. If we disable both state and item backups, the throughput is 6.96 GB/s. The relative estimation error is 9.1% when no failure happens. If we enable approximate fault tolerance, the throughput drops by 5.2% for $\Theta = 10^5$ and $L = 10^3$, and by 12% for $\Theta = 10^4$ and $L = 100$, while the relative estimation error only increases to 11.3% and 10.6%, respectively, even after 10 system failures.

Experiment 5 (Online logistic regression). Logistic regression (LR) is a classical algorithm in machine learning. We use online LR to predict advertisement click-throughs for a public trace in KDD Cup 2012 Track 2 [45], which contains a list of 110 million tuples with a total size 42 GB. Each tuple is associated with a label and multiple features. We evenly divide the entire trace into two halves, one as a training set and another as a test set. We train the model with a distributed stochastic gradient descent (SGD) technique [34], in which each upstream worker trains its local model with a subset of the training set, and regularly sends its local model to a single downstream worker (every 10^3 tuples in our case) in the form of a punctuation item (§3.4.2). The online LR algorithm has a feedback loop, in which the downstream worker computes the average of the model parameters to form a global model, and sends the global model to each upstream worker in the form of a feedback item. The upstream worker updates its local model accordingly. Each upstream worker stores the model parameters in a hash table as its state.

Here, Θ represents the Euclidean distance of the model parameters, and L is the maximum number of unprocessed non-backup tuples for model training. As our model has more than 10^6 features, we set $\Theta \leq 10$ and $L \leq 10^3$ to limit the errors to the model parameters. This setting implies an average Euclidean distance of less than $\sqrt{\Theta/10^6} = 0.01$ for each feature parameter and at most $L/(110 \times 10^6) = 0.001\%$ of lost tuples.

We measure the throughput as the number of tuples processed per second. To measure the accuracy, we predict a label for each tuple in the test set based on its features by LR, and check if the predicted label is identical to the true label associated with the tuple. We measure the accuracy as the prediction rate, defined as the fraction of correctly predicted tuples over all tuples in the test set.

Figure 8 presents the throughput and prediction rate of AF-Stream



(a) Throughput (b) Prediction rate

Figure 8: Experiment 5 (Online logistic regression).

Table 5: Comparisons of fault tolerance approaches for HH detection, online join and online LR with $L = 10^3$ and $\Gamma = 10^3$.

	HH detection, $\Theta = 10^5$	Online join, $\Theta = 10^5$	Online LR, $\Theta = 10$
No fault tolerance	12.33 GB/s	6.96 GB/s	62,699 tuples/s
Mini-batch	0.91 GB/s	0.89 GB/s	14,770 tuples/s
Upstream backup	0.89 GB/s	0.87 GB/s	12,827 tuples/s
Approximate fault tolerance	11.77 GB/s	6.59 GB/s	62,292 tuples/s

for online LR. The throughput is 62,000 tuples/s when we disable state backups. If we set $\Theta = 10$ and $L = 10^3$, or $\Theta = 1$ and $L = 100$, the throughput is only less than that without state and item backups by up to 0.3%. The reason is that the model has a large size with millions of features and incurs intensive computations. Thus, the throughput drops due to state backups become insignificant. In addition, the prediction rate is 94.3% when there is no failure. The above two settings of Θ and L decrease the prediction rate to 90.4% and 92.9%, respectively, after 10 system failures.

Discussion. Our results demonstrate how AF-Stream addresses the trade-off between performance and accuracy for different parameter choices. We also implement the three algorithms under existing fault tolerant approaches as in Experiment 1, and Table 5 summarizes the results. We observe that both mini-batch and upstream backup approaches reduce the throughput to less than 25% compared to disabling fault tolerance, while approximate fault tolerance achieves over 95% of the throughput.

5. RELATED WORK

Best-effort fault tolerance. Some stream processing systems only provide best-effort fault tolerance. They either discard missing items (e.g., [44]) or missing states (e.g., [5, 33, 55]), or simply monitor data loss and return the data completeness to developers (e.g., [31, 39]). These systems may have unbounded errors in the face of failures, making the processing results useless. In contrast, AF-Stream bounds errors upon failures with theoretical guarantees.

Error-free fault tolerance. Early stream processing systems extend traditional relational databases to distributed stream databases to support SQL-like queries on continuous data streams. They support error-free fault tolerance, and often adopt *active standby* (e.g., [7, 50]) or *passive standby* (e.g., [10, 29, 30, 32]). Active standby employs backup nodes to execute the same computations as primary nodes, while passive standby makes periodic backups for states and items to backup nodes. Both approaches, however, are expensive due to maintaining redundant resources (for active standby) or issuing frequent backup operations (for passive standby).

Some stream processing systems realize *upstream backup*, in which an upstream worker keeps the items that are being processed in its downstream workers until all downstream workers acknowledge the completion of the processing. The upstream worker replays the kept items when failures happen. This approach is used extensively by previous studies [4, 18, 26, 37, 42, 48, 57, 58]. Upstream backup generally incurs significant overhead to normal processing, as a system needs to save a large number of items for possible replays (§3.4.2).

To mitigate the impact on normal processing, asynchronous state checkpoints [19, 41] allow normal processing to be performed in parallel with state backups. However, they are not designed for stream processing and do not address item backups, which could be expensive. StreamScope [36] provides an abstraction to handle failure recovery, but does not address how to trade between performance and accuracy in fault tolerance as in AF-Stream.

Incremental processing. Incremental processing systems extend batch processing systems for streaming data, by incrementally batching processing at small timescales. Some systems extend MapReduce [16] by pipelining mapper and reducer tasks (e.g., [11]), while others explicitly divide a stream into mini-batches and run batch-based processing for each mini-batch (e.g., [8, 22, 38, 62]). Incremental processing systems inherently support error-free fault tolerance because all data is available to regenerate states upon failures, but they incur high I/O overhead in saving all items for availability.

Approximation techniques. Recent distributed systems have extensively adopted approximation techniques to improve performance. For example, BlinkDB [2] is a database which samples a subset of data to reduce query latency. Some machine learning systems (e.g., [35, 59, 60]) defer synchronization to reduce communication costs. In the context of stream processing, JetStream [49] deploys tunable operators that automatically trade accuracy for bandwidth saving in wide-area stream processing. Approximate Spark Streaming [3] samples a subset of items in continuous streams for actual processing. AF-Stream differs from them by leveraging approximations in fault tolerance for distributed stream processing.

Zorro [47] introduces approximation techniques to handle failures in graph processing. It exploits vertex replication in distributed graph processing to reconstruct lost states with a high probability. While the idea is similar to approximate fault tolerance, it is not applicable for stream processing since streaming data does not exhibit such replication nature.

6. CONCLUSIONS

We propose AF-Stream, a distributed stream processing system that realizes approximate fault tolerance for both internal states and unprocessed items. AF-Stream achieves not only high performance by reducing the number of backup operations, but also high accuracy by bounding errors upon failures with theoretical guarantees. It provides an extensible programming model and exports user-specified threshold parameters for configuring the performance-accuracy trade-off. Experiments on our AF-Stream prototype demonstrate its high performance and high accuracy in various streaming algorithms. The source code of our AF-Stream prototype is available at: **http://ansrlab.cse.cuhk.edu.hk/software/afstream**.

Acknowledgments: This work was supported by grant (project number: YB2014110015) from Huawei Technologies.

7. REFERENCES

[1] D. J. Abadi, D. Carney, U. Cetintemel, M. Cherniack, C. Convey, S. Lee, M. Stonebraker, N. Tatbul, and S. Zdonik. Aurora: A New Model and Architecture for Data Stream Management. *The VLDB Journal*, 12:120–139, 2003.

[2] S. Agarwal, B. Mozafari, A. Panda, H. Milner, S. Madden, and I. Stoica. BlinkDB: Queries with Bounded Errors and Bounded

Response Times on Very Large Data. In *Proc. of EuroSys*, pages 29–42, 2013.

[3] S. Agarwal and K. Zeng. BlinkDB and G-OLA: Supporting Continuous Answers with Error Bars in SparkSQL. In *Spark Summit*, 2015.

[4] T. Akidau, A. Balikov, K. Bekiro, S. Chernyak, J. Haberman, R. Lax, S. Mcveety, D. Mills, P. Nordstrom, and S. Whittle. MillWheel: Fault-Tolerant Stream Processing at Internet Scale. In *Proc. of VLDB Endowment*, volume 6, pages 1033–1044, 2013.

[5] Apache Flume. `http://flume.apache.org`.

[6] Apache Kafka. `http://kafka.apache.org`.

[7] M. Balazinska, H. Balakrishnan, S. R. Madden, and M. Stonebraker. Fault-Tolerance in the Borealis Distributed Stream Processing System. In *Proc. of SIGMOD*, pages 13–24, 2005.

[8] P. Bhatotia, A. Wieder, R. Rodrigues, U. Acar, and R. Pasquin. Incoop: Mapreduce for Incremental Computations. In *Proc. of SoCC*, pages 7:1–7:14, 2011.

[9] CAIDA Anonymized Internet Traces 2014. `http://www.caida.org/data/passive/passive_2014_dataset.xml`.

[10] M. Cherniack, H. Balakrishnan, M. Balazinska, D. Carney, U. Cetintemel, Y. Xing, and S. Zdonik. Scalable Distributed Stream Processing. In *CIDR*, 2003.

[11] T. Condie, N. Conway, P. Alvaro, J. M. Hellerstein, K. Elmeleegy, and R. Sears. MapReduce Online. In *Proc. of NSDI*, pages 21–21, 2010.

[12] G. Cormode, M. Garofalakis, P. J. Haas, and C. Jermaine. *Synopses for Massive Data: Samples, Histograms, Wavelets, Sketches*. Now Publishers Inc., 2012.

[13] G. Cormode and S. Muthukrishnan. What's New: Finding Significant Differences in Network Data Streams. In *Proc. of INFOCOM*, pages 1534 – 1545, 2004.

[14] G. Cormode and S. Muthukrishnan. An Improved Data Stream Summary: The Count-Min Sketch and its Applications. *Journal of Algorithms*, 55(1):58–75, 2005.

[15] T. Das, Y. Zhong, I. Stoica, and S. Shenker. Adaptive Stream Processing using Dynamic Batch Sizing. In *Proc. of SoCC*, pages 16:1–16:13, 2014.

[16] J. Dean and S. Ghemawat. MapReduce: Simplified Data Processing on Large Clusters. In *Proc. of OSDI*, pages 107–113, 2004.

[17] C. Estan and G. Varghese. New Directions in Traffic Measurement and Accounting. In *Proc. of SIGCOMM*, pages 323–336, 2002.

[18] R. C. Fernandez, M. Migliavacca, E. Kalyvianaki, and P. Pietzuch. Integrating Scale Out and Fault Tolerance in Stream Processing using Operator State Management. In *Proc. of SIGMOD*, pages 725–736, 2013.

[19] R. C. Fernandez, M. Migliavacca, E. Kalyvianaki, and P. Pietzuch. Making State Explicit for Imperative Big Data Processing. In *Proc. of USENIX ATC*, pages 49–60, 2014.

[20] V. Gulisano, R. Jimenez-Peris, M. Patino-Martinez, and P. Valduriez. StreamCloud: A Large Scale Data Streaming System. In *Proc. of ICDCS*, pages 126–137, 2010.

[21] P. J. Haas and J. M. Hellerstein. Ripple Joins for Online Aggregation. In *Proc. of SIGMOD*, pages 287–298, 1999.

[22] B. He, M. Yang, Z. Guo, R. Chen, B. Su, W. Lin, and L. Zhou. Comet : Batched Stream Processing for Data Intensive Distributed Computing. In *Proc. of SoCC*, pages 63–74, 2010.

[23] Q. Ho, J. Cipar, H. Cui, J. K. Kim, S. Lee, P. B. Gibbons, G. A. Gibson, G. R. Ganger, and E. P. Xing. More Effective Distributed ML via A Stale Synchronous Parallel Parameter Server. In *Proc. of NIPS*, pages 1223–1231, 2013.

[24] M. Hoffman, D. Blei, and F. Bach. Online Learning for Latent Dirichlet Allocation. In *Proc. of NIPS*, pages 856–864, 2010.

[25] M. D. Hoffman, D. M. Blei, C. Wang, and J. Paisley. Stochastic Variational Inference. *Journal of Machine Learning Research*, 14(1):1303–1347, 2013.

[26] L. Hu, K. Schwan, H. Amur, and X. Chen. ELF: Efficient Lightweight Fast Stream Processing at Scale. In *Proc. of USENIX ATC*, pages 25–36, 2014.

[27] Q. Huang and P. P. C. Lee. Toward High Performance Distributed Stream Processing via Approximate Fault Tolerance. Technical report, CUHK, 2016. `http://www.cse.cuhk.edu.hk/~pclee/www/pubs/tech_afstream.pdf`.

[28] P. Hunt, M. Konar, F. P. Junqueira, and B. Reed. ZooKeeper: Wait-free Coordination for Internet-scale Systems. In *Proc. of USENIX ATC*, pages 11–11, 2010.

[29] J. H. Hwang, M. Balazinska, A. Rasin, U. Çetintemel, M. Stonebraker, and S. Zdonik. High-Availability Algorithms for Distributed Stream Processing. In *Proc. of ICDE*, pages 779–790, 2005.

[30] J.-h. Hwang, Y. Xing, U. Cetintemel, and S. Zdonik. A Cooperative, Self-Configuring High-Availability Solution for Stream Processing. In *Proc. of ICDE*, pages 176 – 185, 2007.

[31] N. Jain, P. Mahajan, D. Kit, P. Yalagandula, M. Dahlin, and Y. Zhang. Network Imprecision: A New Consistency Metric for Scalable Monitoring. In *Proc. of OSDI*, pages 87–102, 2008.

[32] S. Krishnamurthy, M. J. Franklin, J. Davis, D. Farina, P. Golovko, A. Li, and N. Thombre. Continuous Analytics over Discontinuous Streams. In *Proc. of SIGMOD*, pages 1081–1092, 2010.

[33] S. Kulkarni, N. Bhagat, M. Fu, V. Kedigehalli, C. Kellogg, S. Mittal, J. M. Patel, K. Ramasamy, and S. Taneja. Twitter Heron: Stream Processing at Scale. In *Proc. of SIGMOD*, pages 239–250, 2015.

[34] J. Langford, A. Smola, and M. Zinkevich. Slow Learners are Fast. In *Proc. of NIPS*, pages 2331–2339, 2009.

[35] M. Li, D. G. Andersen, J. W. Park, A. J. Smola, A. Ahmed, V. Josifovski, J. Long, E. J. Shekita, and B.-Y. Su. Scaling Distributed Machine Learning with the Parameter Server. In *Proc. of OSDI*, pages 583–598, 2014.

[36] W. Lin, Z. Qian, J. Xu, S. Yang, J. Zhou, and L. Zhou. StreamScope: Continuous Reliable Distributed Processing of Big Data Streams. In *Proc. of NSDI*, pages 439–454, 2016.

[37] Q. Liu, J. C. Lui, C. He, L. Pan, W. Fan, and Y. Shi. SAND: A Fault-Tolerant Streaming Architecture for Network Traffic Analytics. In *Proc. of DSN*, pages 80–87, 2014.

[38] D. Logothetis, C. Olston, B. Reed, K. C. Webb, and K. Yocum. Stateful Bulk Processing for Incremental Analytics. In *Proc. of SoCC*, pages 51–62, 2010.

[39] D. Logothetis, C. Trezzo, K. C. Webb, and K. Yocum. In-situ MapReduce for Log Processing. In *Proc. of USENIX ATC*, pages 9–9, 2011.

[40] G. Luo, C. J. Ellmann, P. J. Haas, and J. F. Naughton. A Scalable Hash Ripple Join Algorithm. In *Proc. of SIGMOD*, pages 252–262, 2002.

[41] A. Martin, T. Knauth, S. Creutz, D. Becker, S. Weigert, C. Fetzer, and A. Brito. Low-overhead Fault Tolerance for High-throughput Data Processing Systems. In *Proc. of ICDCS*, pages 689 – 699, 2011.

[42] D. G. Murray, F. McSherry, R. Isaacs, M. Isard, P. Barham, and M. Abadi. Naiad: A Timely Dataflow System. In *Proc. of SOSP*, pages 439–455, 2013.

[43] NatSys Lab. `http://natsys-lab.com`.

[44] L. Neumeyer, B. Robbins, A. Nair, and A. Kesari. S4: Distributed Stream Computing Platform. In *KDCloud*, pages 170 – 177, 2010.

[45] Y. Niu, Y. Wang, G. Sun, A. Yue, B. Dalessandro, C. Perlich, and B. Hamner. The Tencent Dataset and KDD-Cup'12. In *KDD-Cup Workshop*, 2012.

[46] Project Gutenberg. `http://www.gutenberg.org`.

[47] M. Pundir, L. M. Leslie, I. Gupta, and R. H. Campbell. Zorro: Zero-Cost Reactive Failure Recovery in Distributed Graph Processing. In *Proc. of SoCC*, pages 195–208, 2015.

[48] Z. Qian, Y. He, C. Su, Z. Wu, H. Zhu, T. Zhang, L. Zhou, Y. Yu, and Z. Zhang. TimeStream: Reliable Stream Computation in the Cloud. In *Proc. of EuroSys*, pages 1–14, 2013.

[49] A. Rabkin, M. Arye, S. Sen, V. S. Pai, and M. J. Freedman. Aggregation and Degradation in JetStream: Streaming Analytics in the Wide Area. In *Proc. of NSDI*, pages 275–288, 2014.

[50] M. a. Shah, J. M. Hellerstein, and E. Brewer. Highly Available, Fault-Tolerant, Parallel Dataflows. In *Proc. of SIGMOD*, pages 827–838, 2004.

[51] K. Shvachko, H. Kuang, S. Radia, and R. Chansler. The Hadoop Distributed File System. In *Proc. of IEEE MSST*, pages 1–10, 2010.

[52] A. Smola and S. Narayanamurthy. An Architecture for Parallel Topic Models. In *Proc. of VLDB Endowment*, volume 3, pages 703–710, 2010.

[53] H. H. Song, T. W. Cho, V. Dave, Y. Zhang, and L. Qiu. Scalable Proximity Estimation and Link Prediction in Online Social Networks. In *Proc. of IMC*, pages 322–335, 2009.
[54] Spark Kafka Integration Guide. `http://spark.apache.org/docs/latest/streaming-kafka-integration.html`.
[55] Storm. `http://storm.apache.org`.
[56] Storm Kafka Integration Guide. `http://storm.apache.org/releases/2.0.0-SNAPSHOT/storm-kafka.html`.
[57] Trident. `http://storm.apache.org/documentation/Trident-tutorial.html`.
[58] H. Wang, L. S. Peh, E. Koukoumidis, S. Tao, and M. C. Chan. Meteor Shower: A Reliable Stream Processing System for Commodity Data Centers. In *Proc. of IPDPS*, pages 1180 – 1191, 2012.
[59] J. Wei, W. Dai, A. Qiao, Q. Ho, H. Cui, G. R. Ganger, P. B. Gibbons, G. A. Gibson, and E. P. Xing. Managed Communication and Consistency for Fast Data-Parallel Iterative Analytics. In *SoCC*, pages 381–394, 2015.
[60] E. P. Xing, Q. Ho, W. Dai, J. K. Kim, J. Wei, S. Lee, and X. Zheng. Petuum: A New Platform for Distributed Machine Learning on Big Data. In *Proc. of KDD*, pages 49 – 67, 2015.
[61] M. Yu, L. Jose, and R. Miao. Software Defined Traffic Measurement with OpenSketch. In *Proc. of NSDI*, pages 29–42, 2013.
[62] M. Zaharia, T. Das, H. Li, T. Hunter, S. Shenker, and I. Stoica. Discretized Streams: Fault-Tolerant Streaming Computation at Scale. In *Proc. of SOSP*, pages 423–438, 2013.
[63] Zero Data Loss in Spark Streaming. `http://databricks.com/blog/2015/01/15/improved-driver-fault-tolerance-and-zero-data-loss-in-spark-streaming.html`.
[64] ZeroMQ. `http://zeromq.org`.
[65] M. Zinkevich. Online Convex Programming and Generalized Infinitesimal Gradient Ascent. In *Proc. of ICML*, pages 928–936, 2003.

APPENDIX

A. ANALYSIS

In this section, we extend the analysis in §3.5 to derive the error bounds of two streaming algorithms with respect to Θ, L, and Γ.

A.1 Heavy Hitter Detection

We consider the heavy hitter (HH) detection with Count-Min Sketch [14]. Count-Min Sketch maintains a matrix of counters with r rows and w counters per row to keep track of the values of the keys. Given a threshold ϕ, a key x is reported as an HH if its estimated value exceeds ϕ. Lemma 1 presents the error probability of HH detection for an arbitrary key x with the true value $T(x)$.

LEMMA 1 ([14]). *Consider a Count-Min Sketch with $r = \log_{1/2}\delta$ rows and $w = \frac{U}{\epsilon\phi}$ counters per row, where U denotes the sum of the values of all keys, and ϵ and ϕ are error parameters. It reports every HH x where $T(x) \geq \phi$ without any error. For a non-HH x with $T(x) < (1-\epsilon)\phi$, it is reported as an HH with an error probability at most δ.*

THEOREM 1. *Suppose that AF-Stream deploys a Count-Min Sketch with the same setting as Lemma 1 and the compensation method in Experiment 3 (§4.2). The sketch reports all HHs. On the other hand, upon a single failure, for an arbitrary key x with the true value $T(x) < (1-\epsilon)\phi - \Theta - \alpha L$, it is reported as an HH with an error probability at most δ.*

PROOF. With the compensation method, AF-Stream overestimates a key by at most $\Theta + \alpha L$, but does not underestimate any keys. Thus, we ensure that every HH x with $T(x) \geq \phi$. On the other hand, let $\widetilde{T}(x)$ denote the estimated value of key x. For a non-HH x with $T(x) < (1-\epsilon)\phi - \Theta - \alpha L$, it is reported as an HH if and only if its estimate value $\widetilde{T}(x) \geq \phi$. Thus, $\Pr\{\widetilde{T}(x) \geq \phi\} = Pr\{\widetilde{T}(x) - T(x) \geq \phi - T(x)\} \leq \Pr\{\widetilde{T}(x) - T(x) \geq \epsilon\phi + \Theta + \alpha L\} \leq \delta$, due to Markov's inequality [14]. □

Example. Consider a network traffic stream where the total volume of flows in a window is $U = 40$ GB and flows exceeding $\phi = 10$MB are HHs. Let $\delta = 1/16$ and $\epsilon = 1/2$. By setting $r = \log_{1/2}\delta = 4$ and $w = \frac{U}{\epsilon\phi} = 8192$, a Count-Min Sketch reports all HHs. For any non-HH less than 5 MB (i.e., $(1-\epsilon)\phi$), it is falsely reported as an HH with an error probability at most $\delta = 1/16$ based on Lemma 1.

Suppose that we run HH detection with Count-Min Sketch on AF-Stream. Here, we assume $\alpha = 1{,}500$, denoting the maximum packet size in bytes. If we configure $L = 10^3$ and $\Theta = 10^5$, then the keys with sizes less than 3.5 MB (i.e., $(1-\epsilon)\phi - L\alpha - \Theta$) will be falsely reported as HHs with an error probability at most $\delta = 1/16$ based on Theorem 1.

A.2 Stream Database Queries

We study Ripple Join [21], an online join algorithm that samples a subset of items in two streams and performs join operations on the sampled subset. Let n be the number of sampled items. For aggregation queries (e.g., SUM, COUNT, and AVG), we denote the true value and estimate value by μ and $\hat{\mu}$, respectively. Ripple Join provides the following guarantee of the aggregation error.

LEMMA 2 ([21]). *When Ripple Join is applied to n sampled items, it guarantees that $\Pr\{\hat{\mu} \in [\mu - \epsilon_n, \mu + \epsilon_n]\} \geq 1 - \delta$, where $\epsilon_n = \frac{z}{\sqrt{n}}$ and z is a constant number depending on δ and the specific aggregation.*

AF-Stream loses sampled items in failure recovery, which is equivalent to decreasing the sampling rate in Ripple Join. Therefore, AF-Stream (slightly) increases the aggregation error as fewer items are sampled. Theorem 2 quantifies the new aggregation error.

THEOREM 2. *When AF-Stream applies Ripple Join to n sampled items, it guarantees that $\Pr\{\hat{\mu} \in [\mu - \epsilon_n, \mu + \epsilon_n]\} \geq 1 - \delta$, where $\epsilon_n = \frac{z}{\sqrt{n - \Theta - L\beta - \Gamma}}$ and z is a constant depends on δ, n and the specific aggregation.*

PROOF. Recall that AF-Stream defines the state divergence as the difference of the numbers of items between the current state and the most recent backup state in Experiment 4. Thus, $\alpha = 1$, since each lost item implies a difference of one in the number of items. AF-Stream ensures that the total number of lost sampled items is at most $\Theta + \alpha(\Gamma + L\beta) = \Theta + \Gamma + L\beta$. By replacing n in Lemma 2 with $n - \Theta - \Gamma - L\beta$, the theorem concludes. □

Example. Consider two streams with 10^9 items in a window and Ripple Join employs a sampling rate 10%, leading to an average of 10^8 sampled items. In this case, the error is less than $\frac{z}{10{,}000}$ with a probability at least $1 - \phi$ based on Lemma 2.

Suppose that we run Ripple Join on AF-Stream. Here, β is the maximum number of items with which an item can join. Its value depends on how fast a stream is generated and the number of items that we apply the join operation. For example, in Experiment 4 (§4.2), we find that the number of items being joined is no more than 100, so we let $\beta = 100$. Suppose that we configure $\Theta = 10^5$, $L = 10^3$, $\Gamma = 10^3$, AF-Stream loses at most $201{,}000$ (i.e., $\Theta + L\beta + \Gamma$) sampled items and increases the error from $\frac{z}{10{,}000}$ to $\frac{z}{9{,}989}$ based on Theorem 2.

B. INTERFACES AND BUILT-IN PRIMITIVE OPERATORS IN AF-STREAM

Table 6: Composing Interfaces in C++ Syntax.

Entities	Functions	Descriptions
Worker	`void AddUpstreamWorker(string& upName)`	Adds an upstream worker
	`void AddDownstreamWorker(string& downName)`	Adds a downstream worker
	`void AddUpstreamThread(Thread& thread)`	Plugs in the upstream thread
	`void AddDownstreamThread(Thread& thread)`	Plugs in the downstream thread
	`void AddComputeThread(Thread& thread)`	Plugs in a compute thread
	`void PinCPU(Thread& thread, int core)`	Pins a thread to a CPU core
	`void SetWindow(int type, int length)`	Sets the type and length of a window (the window type can be the hopping window, the sliding window, or the decaying window)
	`void Start()`	Starts the execution of the worker
Compute thread	`void ConnectFromUpstreamThread()`	Associates the compute thread with the upstream thread
	`void ConnectToDownstreamThread()`	Associates the compute thread with the downstream thread
	`void ConnectToComputeThread(Thread& dstThread)`	Connects the compute thread to another compute thread
	`void SendToUp(string& upName, Item& feedback)`	Sends a feedback item to an upstream worker
	`void SendToDown(string& downName, Item& data)`	Sends a data item to a downstream worker

Table 7: User-defined Interfaces in C++ Syntax.

Entities	Functions	Descriptions
Upstream thread	`Item ReceiveDataItem()`	Receives an input item from data sources or upstream workers
	`int GetDestComputeThread(Item& item)`	Returns the compute thread which an item is dispatched
	`void SendFeedbackItem(Item& feedback)`	Sends a feedback item to an upstream worker
Downstream thread	`void SendDataItem(Item& feedback)`	Sends output items
	`Item ReceiveFeedbackItem()`	Receives a feedback item from downstream workers
Compute thread	`bool ProcessData(Item& data)`	Processes a data item
	`bool ProcessFeedback(Item& feedback)`	Processes a feedback item
	`bool ProcessPunctuation(Item& punc)`	Processes a punctuation item
Fault-tolerant operator	`double StateDivergence()`	Gets the divergence of the up-to-date state and the backup state
	`State BackupState()`	Returns the state to be saved by AF-Stream
	`void RecoverState(State& state)`	Obtains the most recent backup state from AF-Stream