Fault-Tolerant and Elastic Streaming MapReduce with Decentralized Coordination

Alok Kumbhare[1], Marc Frincu[1], Yogesh Simmhan[2] and Viktor K. Prasanna[1]
[1]University of Southern California, Los Angeles, California 90089
[2]Indian Institute of Science, Bangalore 560012
Email: kumbhare@usc.edu, frincu@usc.edu, simmhan@serc.iisc.in, prasanna@usc.edu

Acknowledgement
This material is based upon work supported by the Department of Energy under Award Number DE-OE0000192.

Fault-Tolerant and Elastic Streaming MapReduce with Decentralized Coordination

Alok Kumbhare[1], Marc Frincu[1], Yogesh Simmhan[2] and Viktor K. Prasanna[1]
[1]University of Southern California, Los Angeles, California 90089
[2]Indian Institute of Science, Bangalore 560012
Email: kumbhare@usc.edu, frincu@usc.edu, simmhan@serc.iisc.in, prasanna@usc.edu

***Abstract*—The MapReduce programming model, due to its simplicity and scalability, has become an essential tool for processing large data volumes in distributed environments. Recent Stream Processing Systems (SPS) extend this model to provide low-latency analysis of high-velocity continuous data streams. However, integrating MapReduce with streaming poses challenges: first, the runtime variations in data characteristics such as data-rates and key-distribution cause resource overload, that in-turn leads to fluctuations in the Quality of the Service (QoS); and second, the stateful reducers, whose state depends on the complete tuple history, necessitates efficient fault-recovery mechanisms to maintain the desired QoS in the presence of resource failures. We propose an *integrated streaming MapReduce architecture* leveraging the concept of *consistent hashing* to support runtime elasticity along with locality-aware *data and state replication* to provide efficient load-balancing with low-overhead fault-tolerance and parallel fault-recovery from multiple simultaneous failures. Our evaluation on a private cloud shows up to $2.8\times$ improvement in peak throughput compared to Apache Storm SPS, and a low recovery latency of $700 - 1500$ ms from multiple failures.**

***Keywords*-Distributed stream processing; Streaming mapreduce; Runtime elasticity; Fault-tolerance; Big data**

I. Introduction

The MapReduce (MR) programming model [1] and its execution frameworks have been central to building “Big Data” [2], [3] applications that analyze huge volumes of data. Recently, Big Data applications for processing *high-velocity data* and providing rapid results – on the order of seconds or milliseconds – are gaining importance. These applications, such as fraud detection using real-time financial activity [4], [5], trend analysis and social network modeling [6], and online event processing to detect abnormalities in complex systems, operate on a large and diversified pool of streaming data sources. As a result, distributed Stream Processing Systems (SPS) [7] have been developed to scale with high-velocity data streams by exploiting data parallelism and distributed processing. They allow applications to be composed of continuous operators, called *Processing Elements (PEs)*, that perform specific operations on each incoming tuple from the input streams and produce tuples on the output streams to be consumed by subsequent PEs in the application.

The stateful *Streaming MapReduce (SMR)* [8] combines the simplicity and familiarity of the MR pattern with the non-blocking, continuous model of SPSs. SMR consists of mappers and reducers connected using dataflow edges with a key-based routing such that tuples with a specific *key* are always routed to and processed by the same reducer instance. However, unlike the *batch* MR, which guarantees that the reducers start processing only after all mapper instances finish execution and the intermediate data transfer is completed, in SMR a reducer instance continuously receives tuples from the mappers, with different keys interleaved with each other. Hence the reducers need to store and access some *state* associated with an individual key whenever a tuple with that key needs to be processed.

When SMR with stateful reducers are used to process high-velocity, variable-rate data streams with low-latency, a couple of challenges arise:
(1) Unlike batch MR, the system load is not known in advance and data stream characteristics can vary over time – in terms of data rate as well as the reducer’s key distribution across tuples. This can cause computational imbalances across the cluster, with overloaded machines that induce high stream processing latency. *Intelligent load-balancing and elastic runtime scaling techniques are required to account for such variations and maintain desired QoS.*
(2) The distributed execution of such applications on large commodity clusters (or clouds) is prone to random failures [9]. Techniques used for fault-recovery in batch MR, such as persistent storage and replication using HDFS with re-execution of failed tasks [10], incur high latency cost of seconds to minutes. Other solutions [11], [12], [13] that rely on process replication are unsuitable for SMR due to their high resource cost and need for tight synchronization. To meet the QoS of streaming applications, *SMR must support fault-tolerance mechanisms with minimal runtime overhead during normal operations while also providing low-latency, parallel recovery from one or more concurrent resource failures*.

In this paper we address these two challenges through an integrated architecture for SMR on commodity clusters and clouds that provides: (1) adaptive load-balancing to redistribute keys among the reducers at runtime, (2) runtime elasticity for reducers to maintain QoS and reduce costs in the presence of load variations, and (3) tolerance to random fail-stop failures at the node or the network level. While existing systems for batch and continuous stream processing partially offer these solutions (see § II), our unique contribution is to build an integrated system for streaming MapReduce that supports above features, by extending well-established concepts such as *Consistent Hashing* [14], Distributed Hash Tables (DHT) [15], and incremental checkpointing [16] to the streaming context while maintaining the latency and throughput requirements in

such scenarios. Specifically, the contributions of this paper are:
(1) We propose a dynamic key-to-reducer (re)mapping scheme based on consistent hashing [14]. This minimizes key reshuffling and state migration during auto-scaling and failure-recovery without needing an explicit routing table (§V-A).
(2) We propose a decentralized monitoring and coordination mechanism that builds a peer-ring among the reducer nodes. Further, a locality-aware tuple peer-backup (§V-B) and incremental peer-checkpointing (§V-C) enables low-overhead load-balancing, auto-scaling as well as fault-tolerance and recovery. Specifically, it can tolerate and efficiently recover from $r \leq x \leq \frac{nr}{(r+1)}$ faults, where r is the replication factor and n is number of nodes in the peer-ring. (§VII).
(3) Finally, we implement these features into the *Floe* SPS and evaluate its low-overhead operations on a private cloud, comparing its throughput with Apache Storm SPS which uses upstream backup and external state for fault-recovery (§VIII). We also analyze the latency characteristics during load-balancing and during fault-recovery, exhibiting a constant recovery time from one or more concurrent faults.

II. Related Work

MR and Batch Processing Systems. MR introduced a simplified programming abstraction for distributed large scale processing of high volume data. A number of MR variations that provide high level programming and querying abstractions, such as PIG, HIVE, Dryad [2], Apache Spark [3], and Orleans [17], as well as extensions such as iterative [18] and incremental MR [19] have been proposed for scalable high volume data processing. However, these fail to consider *runtime elasticity* of mappers and reducers as the workload, and required resources, can be estimated and acquired at deployment time. Further, simple fault-tolerance and recovery techniques such as replicated persistent storage and re-execution of failed tasks suffice since the overall *batch* runtime outweighs the recovery cost. In contrast, SMR requires *runtime load-balancing and elasticity*, as the stream data behavior varies over time, and *low latency fault-tolerance and recovery*, to maintain the desired QoS. These are the focus of this paper.

Scalable SPS and SMR. SPSs (Apache Storm [20], Apache S4 [7], Granules [21], and TimeStream [22]) enable loosely coupled applications, often modeled as task graphs, to process high-velocity data streams on distributed clusters with support for SMR operators. Here, mappers continuously emit data tuples routed to a fixed number of reducers based on a given key. Their main drawback is the limited or lack of support for *runtime elasticity* in the number of mappers and reducers to account for variations in data rates or resource performance. Storm supports limited load balancing by allowing the user to add new machines at runtime and redistribute the load across the cluster, but requires the application to be temporarily suspended, leading to a transient spike in latency. Other systems [23], including our previous work [24], support elasticity and dynamic load-balancing but assume stateless operators or involve costly processes and state migration, and hence unsuitable for SMR.

Other SMR approaches (Spark Streaming [25] and StreamMapReduce [8]) convert the incoming stream into small batches (windows) of tuples and perform batch MR within each window; the reducer state is embedded in the batch output. Runtime elasticity can be achieved at the window boundaries by varying the number of machines based on the load observed in the previous window. However, this has two downsides. First, depending on the window size, the queuing latency of the tuples can grow to tens of seconds [25]. And second, since these systems use a simple hash-based key-to-reducer mapping function, scaling in/out causes a complete reshuffle of the key mappings, incurring a high overhead due to the large state transfer required.

Fault-tolerance and state management in SPS. Traditional SPSs support fault-tolerance using techniques such as *active replication* [26] which rely on two or more copies of data and processes. Recent systems such as S4 [7] and Granules [21] provides partial fault-tolerance by using a centralized coordination system such as Zookeeper. They offer automatic fail-over by launching new instances of the failed processes on a new or standby machine. They also perform periodic, non-incremental checkpointing and backup for individual processes, including the tuple buffer, by using synchronous (de)serialization which requires pausing the process during checkpointing. Further, they use an external in-memory or persistent storage for backup. These approaches add considerable overhead and lead to high processing latency during checkpointing as well as recovery. In addition, they do not guarantee against tuple loss as any in-flight and uncheckpointed tuples may be lost during the failure.

Systems such as Storm [20], Timestream [22] guarantee “atleast once” tuple semantics using a combination of upstream-backup and an explicit acknowledgment tree, even in the presence of multiple node failures. Trident, an abstraction over Storm improves that to “exactly once” semantics using an external, persistent coordination system (zookeeper). However, neither of these support state recovery, and any local state associated with the failed processes is lost. A user may implement their own state management system using persistent distributed cache systems (DHT, Zookeeper) but this increases the complexity and processing overhead per tuple. SEEP, [27], [28] integrate elastic scale-out and fault-tolerance for general purpose SPSs using a distributed, partitioned state and explicit routing table. Their solution, while applicable to SMR, incurs higher overhead during load-balancing, scaling and fault-recovery as it fails to take advantage of the key grouping and state locality property of the reducers. This causes significant reshuffling of the key-to-reducer mapping.

Martin et. al. [29] proposed a streaming map reduce system with low-overhead fault tolerance similar to our proposed system. The key distinguishing factor is our support for runtime elasticity and load-balancing to handle variability in data streams observed at runtime. Further, their system enables deterministic execution (*exactly-once semantics*) and relies on the virtual synchrony [30] method for synchronization and synchronous checkpointing at the end of each epoch (checkpoint interval) which increases overall latency and further requires total ordering of messages from different mappers, which is difficult to achieve.

On the other hand, our approach provides efficient state management and fault-recovery, and guarantees "atleast once" tuple semantics with no tuple loss during failure. It combines asynchronous, incremental peer-checkpointing for reducer state and peer tuple backup with intelligent collocation to reduce the recovery overhead by minimizing state and tuple transfer during recovery. We also employ a decentralized monitoring and recovery mechanism which isolates the fault to a small subset of reducers while the rest can continue processing without interruptions.

III. Background

Streaming MapReduce. SMR [8], [31] extends the batch MR model by leveraging stateful operators in SPS and using a key-based routing strategy for sending tuples from mappers to reducers. The mapper is a stateless operator that transforms the input tuples and produces one or more output tuples of the form $t = \langle k_i, v \rangle$. Unlike batch MR, SMR relaxes the "strict phasing" restriction between map and reduce. Hence, the system does not need to wait for all mappers to complete their execution (i.e., produce all tuples for a given key) before starting the reducers. Instead, tuples are routed to the matching reducer as they are emitted and the reduce function is executed on each incoming tuple, producing continuous results.

Unlike the batch MR, where reducers are stateless as they can access all the tuples for a given key during execution, in SMR, *reducers must be stateful.* As tuples with different keys may arrive interleaved at a reducer, a single reducer will operate on different keys, while maintaining an independent state for each key. Specifically, the reducer function takes a tuple, and a state associated with the given key, performs a stateful transformation and produces a new state with an optional output key-value pair, i.e. $\mathbb{R} : \langle k_i, v \rangle, s_j^{k_i} \rightarrow [\langle \widehat{k_i}, \widehat{v} \rangle,] s_{j+1}^{k_i}$, (for e.g., see Fig. 9 in Appendix).

Execution Model. The SMR execution model is based on existing well established SPSs such as Storm, S4 [7], and Floe [32]. Figure 1 shows an example of a SMR application on five hosts (physical or virtual). The mapper and reducer instances are distributed across the available hosts to exploit inter-node parallelism. Each host executes multiple instances in parallel threads, exploiting intra-node parallelism. Although mappers and reducers may be collocated (preferred), for simplicity we assume that they are isolated on different hosts. We thus refer to a node hosting a number of mappers or reducers as a *Mapper node* or *Reducer node*, respectively. Each mapper consumes incoming tuples in parallel, processes and emits tuples of the form $\langle k_i, v \rangle$ where k_i is the key used for routing the tuple to a specific reducer. Given the stateless and independent nature of the mappers, simple load-balancing and elastic scaling mechanisms [33], [24] are adequate and are not discussed here. Instead, we focus on *load-balancing, elasticity, and fault-tolerance* only for the reducer nodes.

The tuples are reliably transmitted to the reducers with assured FIFO ordering between a pair of mapper and reducer.

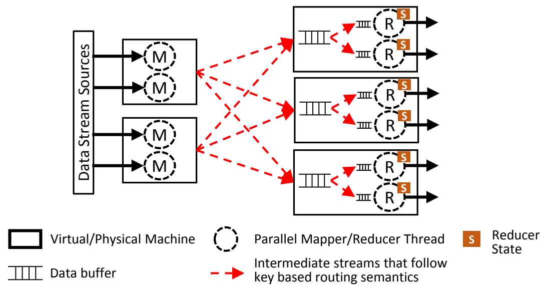

Data Stream Sources
M
R
S
Virtual/Physical Machine
Parallel Mapper/Reducer Thread
Reducer State
Data buffer
Intermediate streams that follow key based routing semantics

Fig. 1. Distributed SMR execution model.

However, given that the several mappers may process and produce tuples in parallel, the *total ordering* among the tuples generated by them, even for a single key, is undefined. Hence, the reducer logic should be agnostic to the order in which the tuples arrive. This constraint is similar to the traditional MR, as well as other stream processing systems. For applications that require specific ordering, techniques such as used by Flux [26] or Virtual Synchrony may be used but are out of scope.

Fault Model. We only consider and guard against fail-stop failures. These occur due to hardware failures, critical software bugs or planned upgrade and reboot in the cluster or cloud environment. In such cases we assume that the acquired resource becomes unavailable and we lose all the buffered data and state associated with that host. Further, individual network link failures are treated as host failures and the fault-tolerance techniques reused. Recovering the lost data and state by replaying tuples from the beginning is not possible as it would cause an unacceptable latency, and hence requires efficient state-management and low-latency fault-tolerance techniques to recover the lost state to continue processing. Using simple techniques that store the state in an external distributed shared, persistent [7], [20] memory (DHT, HDFS, databases) would also suffer from high processing and recovery latency since these systems lack a notion of locality and the state updates and recovery require expensive read/write operations from the external system.

IV. Goals of Proposed System

Performance and Availability. In SPSs, QoS is defined not only in terms of overall system availability and its capability to eventually process all incoming tuples, but is also measured in terms of *average response time* – the latency between when a tuple is generated by the mappers and when it is processed by the reducer, including the corresponding state update. Brito et al. [8] identified several classes of applications with expected response time ranging from minutes or hours for traditional batch data analysis, to less than 5 ms per incoming tuple for real-time applications. Here we focus on fast streaming applications with average response times ranging from $10 - 500$ ms, but real-time applications with millisecond or sub-millisecond response time is out of scope.

The response time includes *transmission*, *queuing*, and *processing* delays. While the transmission and processing delays are a function of the hardware and application logic, the queuing delay is a function of the variable load on the host. The queuing delay may be reduced by transferring its current load to other hosts or newly acquired resources. Such *load transfer* should incur low overhead and not interfere with the regular system operations. Application performance should be maintained during failure and fault-recovery procedure while reducing recovery overhead. These low-latency requirements dictate the design of our system and preclude the use of persistent storage for fault-tolerance. Replicated in-memory distributed hash tables (DHTs) can persist state for failure-recovery but are costly during normal operations for accessing and updating the distributed state since they are not sensitive to the locality of the state on reducer(s).

Deterministic Operations. We assume that the reducer is "an order agnostic, deterministic operator", similar to batch MR, i.e., it does not require tuples to arrive at a certain time or in a particular order to produce results, as long as each tuple is processed exactly once by the reducer. This implies that the application can be made deterministic if we ensure the following: (D1) the state associated with a key at each reducer is maintained between successive executions, i.e., the state is not lost, even during failures; (D2) none of the tuples produced by the mappers are lost; and (D3) none of the tuples produced by the mappers are processed more than once for two different state updates. The last condition might occur during failure recovery or when shifting the load to another reducer (§VI). A tuple is non-deterministic if used for multiple different state transformations, a case which may occur if: (1) the state, or part of it, is lost during recovery (which should never occur as it violates D1), or (2) the reducer instance is unable to determine if the replayed tuple was processed earlier and applies the update again on the recovered state.

Achieving strict determinism that satisfies D1, D2, and D3 is expensive and requires complex ordering and synchronization [26]. In this paper, we relax these and allow "atleast-once" tuple semantics, instead of "exactly-once", i.e., we ensure that D1 and D2 are strictly enforced, but some of the tuples may be processed more than once by the reducers, thus violating D3. Although this limits the class of applications supported by our system, this is an acceptable compromise for a large class of big data applications and can be mitigated at the application level using simple techniques such as *last seen* timestamp.

V. Proposed System Architecture

To achieve the dual low-latency goals of response time performance and fault-recovery under variable data streams, we need to support adaptive load-balancing and elastic runtime scaling of reducers to handle the changing load on the system. Further, we need to support low-latency fault-tolerance mechanism that has minimal overhead during regular operations and supports fast, parallel fault-recovery.

We achieve adaptive runtime load-balancing and elastic scaling of stateful reducers by efficiently *re-mapping* a subset of keys assigned to each reducer, at runtime, from overloaded reducer nodes to less loaded ones. However, the reducer semantics dictate that all the tuples corresponding to a particular key should be processed by a single reducer which conflicts with this requirement. Hence, to transparently meet these

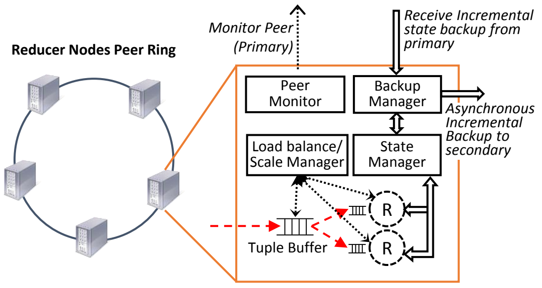

Reducer Nodes Peer Ring
Monitor Peer (Primary)
Receive Incremental state backup from primary
Peer Monitor
Backup Manager
Asynchronous Incremental Backup to secondary
Load balance/ Scale Manager
State Manager
R
R
Tuple Buffer

Fig. 2. Reducer node components.

semantics, we efficiently manage the state associated with the overloaded reducer and migrate the *partial* state corresponding to the re-mapped keys.

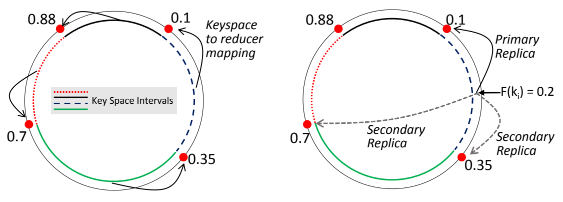
In addition, to support efficient fault-tolerance and recovery, we backup tuples with minimal network and memory overhead, and perform asynchronous incremental checkpointing and backup such that the tuple and state backup corresponding to a specific key are collocated on a single node. The asynchronous checkpointing ensures that performance of regular operations is not affected, and state and tuple collocation ensures that the recovery from failure is fast, without requiring any state or tuple transfer over the network.

The proposed system design integrates these two approaches by building a reducer *peer-ring* as shown in Fig. 2. We overlay a decentralized monitoring framework where each reducer node is identical and acts both as a *primary node* responsible for processing incoming tuples, as well as a *secondary node* responsible for backing up its neighbor's tuples and state as well as to monitor it for failures. A reducer node consists for several components (Fig. 2): a state manager responsible for maintaining the local partitioned state for each key processed by the reducer; a backup manager responsible for receiving and storing state checkpoints from its peers as well as to perform asynchronous incremental checkpointing for the local state; a load-balancer/scaler responsible for monitoring the load on the reducer and perform appropriate load-balance or scaling operations; and a peer monitor responsible for monitoring its peer for failure and perform recovery actions. We discuss these techniques in detail and describe the advantages offered by the peer-ring design in meeting our system goals.

A. Dynamic Re-Mapping of Keys

The shuffle phase of batch MR uses a mapping function $F = \text{HASH}(k) \bmod n$, where k is the key and n is the number of reducers, to determine where the tuple should be routed. The shuffle phase is performed in two stages. First, the mapper applies F to each outgoing tuple and sends the aggregated results to the designated reducer. Once all mappers are done, each reducer aggregates all the tuples received for each unique key from several mappers, and then executes the reducer function for each key over its complete list of values. Scalable SMR is similar except that the tuples are routed immediately to the corresponding reducer which continuously processes the incoming tuples and updates its state (§III).

However, when performing elastic scaling, the number of reducers can change at runtime. So the above mapping



(a) Consistent hashing. (b) Peer-backup for r=2.

Fig. 3. Consistent hashing examples with 4 reducers.

function F becomes infeasible since adding or removing even a single reducer would cause a shuffle and re-map of many of the existing keys to different reducers, leading to a large number of state migrations between the reducers. This $\mathcal{O}(|K|)$ remapping, where $|K|$ is the number of unique keys, will introduce a high processing latency. Existing SPSs, such as Storm, S4, SEEP and Granules, are limited in their runtime scale in/out due to the use of such a hash function.

To overcome this, we rely on *consistent hashing* [14] for key-to-reducer mapping whose *monotonocity* property ensures that adding or removing reducers, only affects a small portion, $\mathcal{O}(\frac{|K|}{n})$, of the key set, needing less state migrations and incurring a smaller latency. The idea is to assign each of the reducers a unique *"token"* $\in [0,1]$ to form a *ring* (Fig. 3a). The keys are always mapped to $[0,1]$ using a well-distributed hash function. We then select the closest reducer in the counter-clockwise direction of the ring from that point. The complexity of mapping the key to a reducer is $\mathcal{O}(1)$.

Whenever a reducer is added, it is assigned a new unique token which puts it between two existing consecutive reducers, dividing the keys space mapped to the original reducer into two subsets, one of which is mapped to the newly added reducer without affecting key mapping for any of the other existing reducers. Similarly whenever a reducer is removed (e.g., due to a fault) its keys are dynamically mapped to the next reducer on the ring in the counter-clockwise direction.

The basic consistent hashing algorithm assigns a random position for each of the nodes on the ring possibly leading to non-uniform load distribution. It is also vulnerable to changes in the system's load due to the variations in data rates and key distribution over time. A virtual nodes approach [34] addresses only the initial non-uniform load distribution issue. Instead, we use a dynamic approach, similar to Cassandra, that allows a node to move along the ring at runtime in response to the variations in the system load.

B. Peer-Backup of Tuples

To achieve efficient backup of incoming tuples to a reducer, we again use a variation of *consistent hashing* that assigns each key to $r+1$ contiguous buckets on the ring onto which the tuple is backed up; r is the tuple replication factor. Fig. 3b shows a sample configuration with 4 buckets and $r = 2$. A key k_i is hashed to the interval $[0,1]$ and the *primary replica* is selected by finding the nearest neighbor on the ring in the counter-clockwise direction as before. In addition, $r = 2$ neighboring buckets are chosen as *secondary replicas* by traversing the ring in the clockwise direction. The mapper then sends the tuple to all 3 of the nodes (1 primary, 2 secondary), using a reliable multi-cast protocol (e.g., PGM) to minimize network traffic.

On receiving a tuple, each node checks if it is the primary by verifying if it appears first in the counter-clockwise direction from the tuple's position on the ring. If so, the tuple is dispatched to the appropriate reducer thread on the machine for processing. Else, the node is a secondary and the tuple is backed-up in-memory, to be used later for load-balancing or fault-recovery. Note that each node can determine if it is a primary in $\mathcal{O}(1)$ time, and needs to navigate to at most r clockwise neighbors. Note that it is important to clear out tuple replicas which have been successfully processed to reduce the memory footprint. This eviction policy is discussed in V-D.

C. Reducer State Model and Incremental Peer-Checkpointing

The state model must support (1) *partial state migration*, i.e., migrating the state associated with only a subset of keys at a reducer, and (2) *reliable, incremental and asynchronous checkpointing* [35], [16], [36], i.e., the state must be deterministic, not concurrently updated by the reducer during checkpointing, and without the need to pause the reducers during this process.

Partial states can be managed by decoupling the state from reducer instance, and partitioning it based on the individual keys being processed by the reducer (Fig. 4). This allows incremental backup only for the keys processed and updated during the last checkpointing period. We can further reduce the size of the incremental backup by restricting the state representation to a set of key-value pairs (*Master State* in Fig. 4). Given this two-level state representation, an incremental backup can be obtained by keeping track of reducer keys updated during the latest checkpoint and the set of key-value pairs updated for each of the reducer keys.

We divide the state associated with each key into two parts, a *master state* and a *state fragment*. The former represents a stable state, i.e., a state that has been checkpointed and backed up onto a neighbor. The latter represents the incrementally updated state which has to be checkpointed at the end of this checkpoint interval. After the fragment is checkpointed, it can be merged into the master state and cleared for the next checkpoint interval. This allows efficient incremental checkpointing, but can lead to *unreliable* checkpoints as the state fragment may be updated by the reducers during the checkpointing process. We can avoid this by pausing the reducers during checkpointing, but it incurs high latency. Instead we propose an *asynchronous* incremental checkpointing process.

Here, we divide the state fragment into two mutually exclusive sub-fragments: *active* and *inactive*. The active fragment is used by the reducers to make state updates, i.e., the key-value pairs are added/updated only in the active fragment, while the inactive fragment is used only during checkpointing, as follows. At the end of a checkpoint interval, the active fragment contains all the state updates that took place during that checkpoint interval, while the inactive fragment is empty. To start the checkpointing process, the state manager *atomically* swaps the pointers to the active and inactive fragments and the

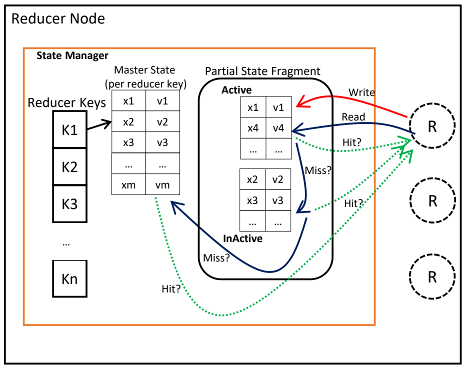


Fig. 4. State representation with master state and partial state fragments.

corresponding timestamp is recorded. The reducers continue processing as usual and update the state in the active fragment, while the inactive fragment contains a *reliable* snapshot of updates that occurred in the previous interval. The inactive fragment is then asynchronously serialized, checkpointed, and transferred to the backup nodes using a multi-cast protocol. After completion, the inactive fragment is merged with the master state and is cleared for the next checkpoint cycle.

With respect to freshness, the active fragment contains the most recent updates, followed by the inactive fragment and finally the master state. Thus, a "state read" request for a key-value pair from the reducer is first sent to the active fragment, then the inactive fragment, and finally the master state until the corresponding key-value pair is found (Fig. 4). While this involves three seeks, it can be mitigated with $\mathcal{O}(1)$ data structures like hash tables.

The combination of partitioned state and active/inactive state fragments allows the reducer thread to continue processing incoming tuples and update the state in the active fragment during the checkpointing process without any conflicts or interruptions thus minimizing the overhead (See § VIII). As with tuple-backup, we follow an optimistic checkpointing process where the primary host does not wait for an acknowledgment from any of the backup hosts, letting reducers execute uninterrupted. This optimistic replication works as long as atleast one backup node is available. We rely on the peer ring to determine the hosts to be used for backup to ensure tuple and state collocation for fast recovery.

D. Tuple Ordering and Eviction Policy

The tuple eviction policy determines which tuples can be safely removed from the backup nodes such that the state associated with the failed host can be recovered by replaying the remaining tuples and updating the checkpointed state. Tuple eviction allows us to keep a low memory footprint.

The proposed eviction policy allows for a small subset of tuples to be processed more than once. We assume that all the mappers are loosely time synchronized and that the maximum time skew between any two mappers is less than the maximum network latency (L_n) to transfer tuples from mappers to reducers. Such assumption have been used in systems such as Flux [26] to identify missing messages and is practically justified since using NTP an error bound within few μ-seconds may be achieved and network latency is typically observed in the milli-seconds range.

Each mapper emits tuples with an associated timestamp. We assume a reliable multi-cast protocol to ensure that emitted tuples are delivered in FIFO order. The primary host then processes the tuples in the order they arrive from different mappers and marks the state update with the latest timestamp of the tuples that affect the current state. Given the time skew between mappers and the maximum network latency, L_n, a tuple with lower timestamp may be received and processed at a later point. In this case the timestamp mark on the state is not updated.

The backup hosts also receive tuples from different mappers for backup. These hosts store the backup tuples in decreasing order of their timestamps. Maintaining this order is not costly since the tuples are usually received in an increasing order of timestamp (except for few due to clock skew and network latency). Whenever a checkpoint is received from the primary, the secondary host retrieves the associated timestamp T_s and evicts all tuples with a timestamp less than $T_s - 2L_n$ (instead of tuples with TS less than T_s) since tuples with an earlier timestamp may arrive later and may have not been processed yet by the primary node. This leaves some potential tuples in the backup which may have already been processed and reflected in the checkpointed state. If the primary host fails, the backup node tries to recover the state by replaying the backup tuples and updating the checkpointed state. This may lead to some tuples being processed twice, violating condition D3. Certain measures can be implemented at the application level to handle this scenario but are out of our scope. As long as the time skew is bounded by network latency, L_n, there will be no tuple loss during failure, hence condition D2 will be met. Note that the issue of variability in network latency over time can be mitigated in one of the two ways: first, by setting a much high value of the threshold (L_n) compared to the observed maximum network latency (10-20x) such that any latency higher than the threshold may be considered as network failure. However, a side effect of high threshold is that it significantly increases the number of messages that may be replayed during a fault recovery or scaling-in scenario. Second, we can monitor the network latency over time and update the threshold ($2 \times L_n$) to reflect the variations in network latency. Note that this can be achieved efficiently in a decentralized manner since each of the nodes in the ring can monitor the latency and use a gossip protocol to propagate the changes to other nodes in the ring. We use the first approach for system evaluation in Section VIII.

VI. Adaptive Load Balancing and Elasticity

Because of fluctuations in the data streams two scenarios can occur. First, the number of keys mapped to a particular interval (i.e., on a reducer node) may fluctuate at runtime, and second, the rate of generated tuples associated with a specific key may vary at runtime. Both scenarios can lead to processing load imbalances among the reducer nodes.

The proposed architecture supports adaptive load balancing by leveraging the fact that the neighboring nodes (secondary instances) associated with a key already possess a recently cached state (peer-checkpointing) as well as the unprocessed tuples (peer-backup) associated to that key.

We use a simple load-balancing strategy where each host monitors its buffer queue length (q_L) and the overall CPU usage (c) while processing the incoming tuples. A host is said to be overloaded if $q_L \geq \tau_q^{high}$ *AND* $c \geq \tau_c^{high}$. While this strategy is prone to oscillation in resource allocation if frequent variations are observed, we have previously proposed robust runtime adaptation algorithms that not only consider the current system load but also observe data patterns and use predictive techniques to avoid such oscillations as well as to minimize resource cost [33], [24], which can be applied here.

An overloaded node (A) negotiates with its clockwise neighbor (B) (i.e., the first backup host) to see if it can share some of the load (i.e., $q_L \leq \tau_q^{low}$ *AND* $c \leq \tau_c^{low}$ for that node). If so, it requests B to expand the primary interval associated with that node by moving in the counter-clockwise direction on the ring. Figure 10 in Appendix shows a sample configuration and the transformation that happens during the load balancing phase. The distance by which the backup node should move can be determined by the exact load on the two systems and the current distribution of keys along the path. However, for simplicity, we assume that the backup node always moves by half the distance between the two at a time.

Node B then trivially recovers the state associated with the transferred keys that was checkpointed earlier by A. It then replays and processes the tuples buffered in its backup (i.e., those tuples not reflected in the currently checkpointed state) and starts processing the new tuples associated with that interval. The backup node now becomes the primary for the given interval and relieves the load on the original primary (A) by taking over some of its load. It also notifies the mappers about the change using multi-cast. Note that the backup node need not wait for this to be delivered to the mappers, since, being the secondary node, it already receives all the relevant tuples. On receiving the notification, the mappers will update their ring and stop sending the tuples to A and start sending it to B and the load-balancing will be completed.

The adaptive load-balancing technique allows an overloaded node to negotiate with its immediate neighbor to offload some of its work. However, if the neighboring host does not have spare cycles and is working to its capacity, it can in turn request its neighbor to offload before accepting the load from the primary. If none of the hosts can accept additional load, the primary host will elastically scale out as follows.

Scaling Out. Scaling out involves provisioning a new host, putting the new host on the ring such that it offloads some of the work from the overloaded primary node, and transferring the corresponding state on to it. To minimize the overhead we start the state transfer in background only state corresponding to the keys that will be offloaded. We also start sending (duplicating) tuples associated with the keys to the new host for backup (using the multi-cast protocol, hence without the additional hop). During this process, the primary node continues to process the incoming tuples as before (albeit at a slower rate due to overload) and update the active state fragment as before. Once the master state transfer is completed, the previously described checkpointing process is performed on the backup nodes as well as on the newly acquired node. This step ensures that the new host has the latest state and that the processed tuples are evicted from its backup buffer. Finally, the new host is put on the peer-ring mid-way between the overloaded node and its neighbor and its r neighbors are informed about the change (see Fig. 11 in Appendix). The new host thus takes over some work from the overloaded node.

Scaling In. A primary host can be scaled in if that host along with its clockwise neighbor are lightly loaded (i.e., $q_L \leq \tau_q^{low}$ *AND* $c \leq \tau_c^{low}$ for both hosts). The primary host can offload all of its load to the neighbor and can return to the resource pool and be available for other requirements or shutdown to conserve cost and energy. The process is similar to the load-balancing process including the checkpoint, tuple replay, and state recovery, except that the neighbor's interval is now expanded to cover the primary's entire interval by setting its token to be equal to the primary's token and removing the primary from the ring (Fig. 12 in Appendix).

VII. Fault-Tolerance

A system is said to be r-fault-tolerant if it can tolerate r arbitrary faults (host failures cf. §III) and can resume its operations (i.e., satisfy D1, D2) without having to restart/redeploy the entire application. The proposed system can tolerate at most r failures in *consecutive* neighbors on the ring since the state and the tuples are backed up on $r+1$ hosts. However, it can tolerate more than r failures if the failures do not occur in consecutive neighbors on the ring. Specifically, in the best case scenario, it can tolerate up to $x = \frac{nr}{(r+1)}$ node failures as long as the faults do not occur in more than r consecutive neighbors (where r is the replication factor and n is number of nodes). For example, with $r = 1$, the system can still be functional (i.e., no state or tuple loss) even if every alternate node on the ring (i.e., $n/2$ nodes) fail simultaneously.

Recent large scale studies [37] have shown that a significant portion of failures observed in a datacenter are spatially collocated (e.g., overheating, rack failures, cluster switch failures) and that the datacenter can be divided into several "fault zones" such that the probability of simultaneous failures for machines in different fault zones is lower than that for machines within a single fault zone. This is also the basis for the "fault domains" or "availability zones" feature provided by Microsoft Azure and Amazon AWS which provide atleast 99.95% availability guarantees if VMs are placed in distinct fault zones. We can exploit this property and place neighbors on the ring in distinct fault zones (Fig. 13 in Appendix) to achieve higher fault-tolerance.

Fault-tolerance involves two activities: fault-detection and recovery. To achieve the former in a decentralized manner we overlay a monitoring ring on top of the peer-ring where each host monitors its immediate **counter-clockwise** neighbor (i.e., a secondary node which holds tuple and state backup for a subset of the key space monitors the primary node). Whenever a fault is detected, to initiate fault recovery, it contacts the one-hop neighbor of the failed node to see if that node is still alive and continues this process until r hops or a live host is found. Note that at this point the backup host will have the checkpointed state and tuples for all failed nodes. Hence, it can employ a procedure similar to the *scale in* process to take over the load from one or more ($\leq r$) failed nodes. As before, no state transfer or additional tuple transfer is required.

The fault-recovery process by itself does not provision additional resources but uses the backup nodes for recovery so as to minimize downtime. However, during or after the takeover, if the backup node becomes overloaded, the adaptive load-balance or scale-out process will be initiated to offload some of its work without interrupting the regular operations.

VIII. Evaluation

We implemented a prototype of our proposed architecture on top of Floe SPS [32]. It provides a distributed execution environment and programming abstractions similar to Storm and S4. In addition, it has built-in support for elastic operations and allows the user to add or remove parallel instances of (stateless) PEs at runtime using existing or acquired resources. We extend it to support elasticity as well as fault-tolerance for stateful reducers via the proposed enhancements.

The goal of the experiments is not to study the scalability of the system on hundreds of nodes, but to evaluate the dynamic nature of the proposed system and the overhead it incurs. The setup consists of a private Eucalyptus cloud with up to 20 VMs (4 cores, 4GB RAM) connected using gigabit Ethernet. Each map/reduce node holds at most 4 corresponding instances. We use a streaming version of the *word frequency count application* that keeps track of the overall word frequency for each unique word seen in the incoming stream as well as a count over a different sliding windows (e.g. past 15 mins, 1 hr, 12 hrs etc.) which represents recent trends observed in the stream. Such application may be used in analysis of social data streams to detect top "k" trending topics by ranking them based on their relative counts observed during a recent window. In such applications, the exact count for each of the topics is not required since the relative ranking among the topics is sufficient. As a result, the system's atleast-once semantics for message delivery is acceptable for the application. We emulate the data streams by playing the text extracted from the corpus of text data from the Gutenberg project. Each mapper randomly selects a text file and emits a stream of words which is routed to the reducers. To demonstrate various characteristics, we emulate: (1) variations in overall data rate by dynamically scaling up/down the number of mappers, and (2) variations in data rate for a subset of keys (load imbalance), by using streams that repeatedly emit a small subset of words.

We synchronize the VMs using NTP and get a loose synchronization bound within few μsec. Further, we determine the maximum network latency (L_n) to be around 1ms by executing a number of ping requests between the VMs. Nonetheless we use a conservative estimate of 15ms for our experiments and a value of $2 \times L_n$ = 30ms as a bound to evict tuples from the

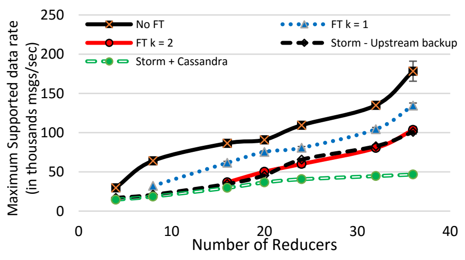


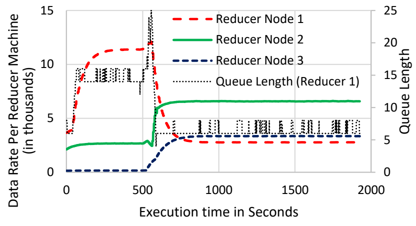
Fig. 5. Achieved Peak throughput for different number of reducers.

backup buffer to account for variations in the network latency and potential time drift observed over time.

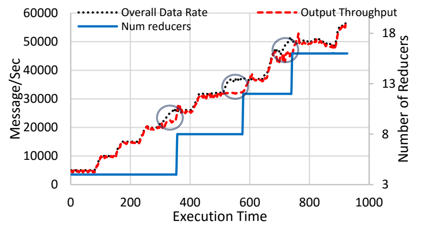
A. Empirical Results

We first evaluate the system under static configurations to determine the overhead due to the checkpointing and backup mechanisms. We fix the number of VMs and reducers at deployment time and progressively increase data rates to determine the maximum achievable cumulative throughput (processing rate of the reducers). We examine the system under different tolerance levels $r = 0, 1, 2$, where 0 indicates that no fault-tolerance mechanisms are in place. We compare our system against two variations of the application deployed using Storm SPS. The first uses Storm's upstream-backup feature with explicit acknowledgments ensuring that no tuples are lost during failure. However, the state is stored locally and may be lost if the corresponding node fails. The second relies on an external distributed reliable in-memory storage (Cassandra) to store the state associated with each key. This version provides fault-tolerance and recovery as well as protection against tuple loss similar to the proposed system, but incurs significant overhead. Fig. 5 shows the peak throughput achieved by these systems as a function of number of reducers. Following key observations can be made from Fig. 5: (1) The peak throughput achieved by Floe at $r = 0$ is consistently higher than others due to minimal overhead and it drops by around 15% as we increase the tolerance level. This is expected since higher r values require additional tuple and state transfer and adds to the load on secondary nodes. (2) Floe achieves higher throughput than both versions of Storm giving around 2.8x improvement for $r = 1$ compared to Storm with state backup using Cassandra due to high latency incurred during state access. (3) Finally, we observe that Floe scales (almost) linearly as we increase the number of resources, while Storm's peak throughput flattens out after a certain point due to the bottleneck caused by the external state management system.

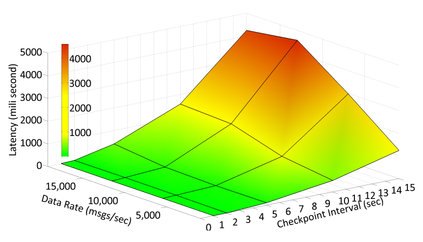
Next we study the throughput and latency characteristics of the proposed load-balancing and elasticity mechanisms. Fig. 6a shows an example load-balancing scenario with fixed resources. It shows the *last 1-min* average data rate per node for a deployment with 3 reducer nodes and average queue length for one of the nodes. The system is initially imbalanced (due to random placement of small number of reducer nodes) but stable (i.e., $q_L \leq \tau_q^{high}$ for all nodes). At around 500s, we repeatedly emit a small subset of words causing further imbalance. The pending queue length for reducer 1 starts increasing beyond the threshold indicating that the incoming data rate is beyond its processing capacity and it initiates the load-balancing process with its neighbor, reducer 3, which in



(a) Load balancing example (fixed resources).



(b) Throughput characteristics as a function of data-rate and scale-out.



(c) Backup node latency.

Fig. 6. Throughput and Latency Characteristics for Load Balance and Scale-out.

turn transfers some of its load to reducer 2 and reaches a stable state around 700s. Note that the system may not reach stable state if the increase in data rate is beyond the cumulative capacity of the cluster, in which case a "scale out" operation will be performed. Fig. 6b shows the system response and throughput as a function of increasing data rate. We observe that for given set of resources, the achieved throughput initially increases with increase in the incoming data rate. However, as the system reaches its capacity, the observed throughput flattens out (as indicated in Fig. 6b). As a result, the pending queue length for the resources goes beyond the threshold and a load-balance or scale-out process is initiated on an idle or acquired resource which allows the system to catchup with the incoming data rate. We observe a reaction time for scale out to be around 1.5–3 seconds which includes both the detection latency as well as message replay and state restoration. While the former is a function of monitoring interval, the latter depends on both the checkpointing interval and the data rate (at the overloaded reducer), which we study next.

Fig. 6c shows the latency characteristics for the load-balancing scenario as a function of data rate and the check-pointing interval on the latency. We observe that the absolute value of the latency is very low (10 - 500ms) for moderate data rates and checkpointing interval of up to 10s. However, latency increases as the checkpointing interval and the data rate increases as this causes the number of tuples backed up by the node to increase along with the number of tuples that need to be replayed. Further, it adds significant memory overhead and that contributes to the performance degradation. Thus smaller checkpointing intervals are preferred. Another benefit of our approach is due to the proposed state representation and use of state fragments which allows us to asynchronously and consistently checkpoint a part of the state without pausing the regular operations, eliminating the effect of frequent write operations caused by small checkpointing intervals. As shown in Fig. 7, the proposed incremental checkpointing significantly

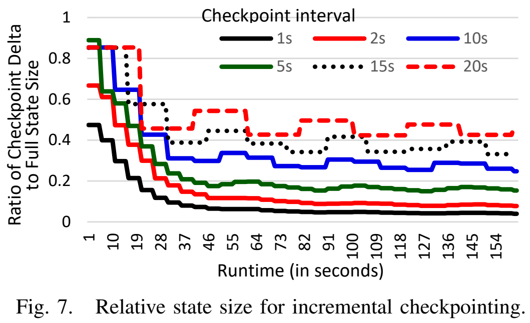
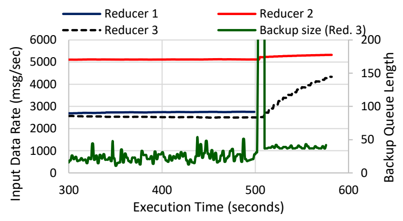
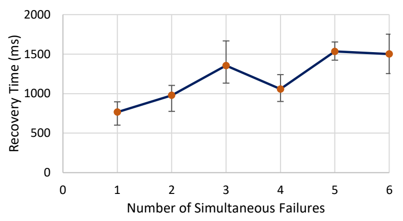


Fig. 7. Relative state size for incremental checkpointing.



(a) Fault-recovery example.



(b) Recovery latency as a function of number of simultaneous node failures (n = 12, r = 1, data rate = 15,000 msgs/sec).

Fig. 8. Fault Tolerance Throughput and Latency Characteristics.

reduces the size of the checkpoint when using smaller intervals (the checkpoint size for 2s interval stabilizes around 7% of the size of the entire state stored by the reducer node), which further supports our argument for a smaller checkpointing interval. Since load balancing, elasticity, and fault-tolerance use the same backup mechanism, Fig. 6c is a good indicator of the overall performance of our process.

Finally, we demonstrate the fault-tolerance and recovery process. Fig. 8a shows a snapshot of an execution with 3 reducer nodes (12 reducers) with fixed data rate. It also shows the size of the tuple backup for one of the reducers (reducer 3). We induce a fault in the system by manually stopping reducer 1 at around 500s. Reducer 3 stops receiving checkpoint data from the failed reducer (which is also treated as a heartbeat) leading to large tuple backup during the recovery process. After detecting the fault, it decides to take over the execution from the failed reducer and replays all the backed-up tuples to recover the state and finally moves itself on the ring so that the tuples originally destined for the failed node are now transmitted to reducer 3 (as is evident by the increasing data rate). Note that the latency characteristics of fault-recovery due to a *single node fault* are similar to that of the load-balancing process (Fig. 6c) and hence are omitted for brevity. We further study the recovery latency of the system under multiple concurrent failures. Since the recovery is performed in parallel, the overall recovery latency is the maximum of latencies to recover all the failed nodes in parallel. Figure 8b shows the average recovery latency observed for multiple (m)

simultaneous failures such that no two consecutive neighbors fail simultaneously. We observe that the recovery latency does not increase linearly and stabilizes around 1,200ms to 1,500ms for 3 to 6 simultaneous node failures. Note that the recovery latency for multiple simultaneous failures is measured as the maximum recovery latency incurred by any of the backup nodes in the ring. The observed variations in the recovery time (Fig. 8b) are due to the imbalances in the load (which leads to varying recovery latency for different failed nodes) and is not due to the increase in the number simultaneous failures.

IX. Conclusions

As high-velocity data becomes increasingly common, using the familiar MapReduce model to process it is valuable. In this paper, we presented an integrated approach to support fault-tolerance, load-balancing and elasticity for the streaming MapReduce model. Our novel approach extends the concept of consistent hashing, and provides locality-aware tuple peer-backup and peer-checkpointing. These allows low-latency dynamic updates to the system, including adaptive load-balance and elasticity, as well as offer low-latency fault recovery by eliminating the need for explicit state or tuple transfer during such operations. Our decentralized coordination mechanism helps make autonomic decisions based on the local information and eliminates a single point of failure. Our experiments show up to $2\times$ improvement in throughput compared to Storm SPS and demonstrated low-latency recovery of $10 - 1500\ ms$ from multiple concurrent failures. As future work, we plan to extend the evaluation to larger systems and real-world, long running application and also to extend the idea to a general purpose stream processing systems for wider applicability.

Acknowledgment

This work was supported by the U.S. National Science Foundation under grant ACI-1339756.

References

[1] W. Yin, Y. Simmhan, and V. Prasanna, "Scalable regression tree learning on hadoop using openplanet," in *MAPREDUCE*, 2012.

[2] M. Isard, M. Budiu, Y. Yu, A. Birrell, and D. Fetterly, "Dryad: distributed data-parallel programs from sequential building blocks," in *SIGOPS Operating Systems Review*, vol. 41, no. 3. ACM, 2007.

[3] M. Zaharia, M. Chowdhury, M. J. Franklin, S. Shenker, and I. Stoica, "Spark: cluster computing with working sets," in *HotCloud*, 2010.

[4] N. Parikh and N. Sundaresan, "Scalable and near real-time burst detection from ecommerce queries," in *SIGKDD*. ACM, 2008, pp. 972–980.

[5] A. Martin, C. Fetzer, and A. Brito, "Active replication at (almost) no cost," in *SRDS*. IEEE, 2011, pp. 21–30.

[6] M. Gatti, P. Cavalin, S. B. Neto, C. Pinhanez, C. dos Santos, D. Gribel, and A. P. Appel, "Large-scale multi-agent-based modeling and simulation of microblogging-based online social network," in *Multi-Agent-Based Simulation XIV*. Springer, 2014, pp. 17–33.

[7] L. Neumeyer, B. Robbins, A. Nair, and A. Kesari, "S4: Distributed stream computing platform," in *ICDMW*, 2010.

[8] A. Brito, A. Martin, T. Knauth, S. Creutz, D. Becker, S. Weigert, and C. Fetzer, "Scalable and low-latency data processing with stream mapreduce," in *CloudCom*. IEEE, 2011, pp. 48–58.

[9] P. Gill, N. Jain, and N. Nagappan, "Understanding network failures in data centers: measurement, analysis, and implications," in *SIGCOMM Computer Communication Review*, vol. 41, no. 4. ACM, 2011.

[10] K. Shvachko, H. Kuang, S. Radia, and R. Chansler, "The hadoop distributed file system," in *Symposium on Mass Storage Systems and Technologies (MSST)*. IEEE, 2010, pp. 1–10.

[11] T. Gunarathne, B. Zhang, T.-L. Wu, and J. Qiu, "Scalable parallel computing on clouds using twister4azure iterative mapreduce," *Future Generation Computer Systems*, vol. 29, no. 4, pp. 1035 – 1048, 2013.

[12] L. Wang, J. Tao, R. Ranjan, H. Marten, A. Streit, J. Chen, and D. Chen, "G-hadoop: Mapreduce across distributed data centers for data-intensive computing," *Future Generation Computer Systems*, vol. 29, no. 3, 2013.

[13] Q. Zheng, "Improving mapreduce fault tolerance in the cloud," in *International Symposium on Parallel Distributed Processing, Workshops and Phd Forum (IPDPSW)*, April 2010, pp. 1–6.

[14] D. Karger, A. Sherman, A. Berkheimer, B. Bogstad, R. Dhanidina, K. Iwamoto, B. Kim, L. Matkins, and Y. Yerushalmi, "Web caching with consistent hashing," *Computer Networks*, vol. 31, no. 11, 1999.

[15] M. F. Kaashoek and D. R. Karger, "Koorde: A simple degree-optimal distributed hash table," in *Peer-to-peer systems II*. Springer, 2003.

[16] S. Agarwal, R. Garg, M. S. Gupta, and J. E. Moreira, "Adaptive incremental checkpointing for massively parallel systems," in *Supercomputing*. ACM, 2004.

[17] S. Bykov, A. Geller, G. Kliot, J. R. Larus, R. Pandya, and J. Thelin, "Orleans: cloud computing for everyone," in *Symposium on Cloud Computing*. ACM, 2011, p. 16.

[18] J. Ekanayake, H. Li, B. Zhang, T. Gunarathne, S.-H. Bae, J. Qiu, and G. Fox, "Twister: A runtime for iterative mapreduce," in *Workshop on MapReduce and its Applications (MAPREDUCE)*, 2010.

[19] P. Bhatotia, A. Wieder, R. Rodrigues, U. A. Acar, and R. Pasquin, "Incoop: Mapreduce for incremental computations," in *Symposium on Cloud Computing*. ACM, 2011, p. 7.

[20] Storm, distributed and fault-tolerant realtime computation. Last accessed 19 Dec. 2014. [Online]. Available: http://storm.apache.org/

[21] S. Pallickara, J. Ekanayake, and G. Fox, "Granules: A lightweight, streaming runtime for cloud computing with support, for map-reduce," in *CLUSTER*. IEEE, 2009.

[22] 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 *ECCS*, 2013.

[23] B. Satzger, W. Hummer, P. Leitner, and S. Dustdar, "Esc: Towards an elastic stream computing platform for the cloud," in *CLOUD*, 2011.

[24] A. Kumbhare, Y. Simmhan, and V. Prasanna, "Plasticc: Predictive look-ahead scheduling for continuous dataflows on clouds," in *CCGrid*. IEEE/ACM, May 2014, pp. 344–353.

[25] M. Zaharia, T. Das, H. Li, S. Shenker, and I. Stoica, "Discretized streams: an efficient and fault-tolerant model for stream processing on large clusters," in *USENIX HotCloud*, 2012.

[26] M. A. Shah, J. M. Hellerstein, and E. Brewer, "Highly available, fault-tolerant, parallel dataflows," in *SIGMOD*. ACM, 2004.

[27] M. Migliavacca, D. Eyers, J. Bacon, Y. Papagiannis, B. Shand, and P. Pietzuch, "Seep: scalable and elastic event processing," in *Middleware'10 Posters and Demos Track*. ACM, 2010, p. 4.

[28] R. Castro Fernandez, M. Migliavacca, E. Kalyvianaki, and P. Pietzuch, "Integrating scale out and fault tolerance in stream processing using operator state management," in *SIGMOD*. ACM, 2013, pp. 725–736.

[29] 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 *Distributed Computing Systems (ICDCS), 2011 31st International Conference on*, June 2011, pp. 689–699.

[30] L. E. Moser, Y. Amir, P. M. Melliar-Smith, and D. A. Agarwal, "Extended virtual synchrony," in *Distributed Computing Systems, 1994., Proceedings of the 14th International Conference on*. IEEE, 1994.

[31] C. Fetzer, "Streammine: A scalable and dependable event processing platform," in *DEBS*. ACM, 2010.

[32] Y. Simmhan, A. Kumbhare, and C. Wickramachari, "Floe: A dynamic, continuous dataflow framework for clouds," USC, Tech. Rep., 2013.

[33] A. Kumbhare, Y. Simmhan, and V. K. Prasanna, "Exploiting application dynamism and cloud elasticity for continuous dataflows," in *SuperComputing,*. ACM, 2013.

[34] G. DeCandia, D. Hastorun, M. Jampani, G. Kakulapati, A. Lakshman, A. Pilchin, S. Sivasubramanian, P. Vosshall, and W. Vogels, "Dynamo: Amazon's highly available key-value store," *SIGOPS Oper. Syst. Rev.*, vol. 41, no. 6, 2007.

[35] J. S. Plank, J. Xu, and R. H. Netzer, "Compressed differences: An algorithm for fast incremental checkpointing," *University of Tennessee, Tech. Rep. CS-95-302*, 1995.

[36] R. Gioiosa, J. C. Sancho, S. Jiang, F. Petrini, and K. Davis, "Transparent, incremental checkpointing at kernel level: A foundation for fault tolerance for parallel computers," in *SuperComputing*. IEEE Computer Society, 2005.

[37] J. Dean, "Designs, lessons and advice from building large distributed systems," *Keynote from LADIS*, 2009.

Appendix

We include here additional content to clarify some of the core concepts outlined in the paper. Fig. 9 shows the streaming word frequency count application using stateful streaming map reduce (SMR) programming model. The Map function is executed for each incoming tuple (e.g. once per tweet), which then emits a tuple as a key value pair $\langle Word, 1\rangle$. Each tuple is then mapped to a reducer instance based on the specific key, similar to the batch mapreduce version. However, unlike batch MR, the reducer function is executed for each tuple it receives. Hence the reducer does not have access to all tuples for the corresponding key, instead the SMR framework maintains a local state associated with each "key" processed by the reducer (§V-C), which is passed along with the tuple to the reducer function. Hence the reducer keeps a running count of all the words seen so far. It can choose to emit the current count based on certain condition such as an external signal.

```
1: procedure MAP($Tuple$, $Emitter$)
2:         $\rightarrow Tuple$ : Incoming data tuple
3:         $\rightarrow Emitter$ : Emit tuples to the reducers
4:     for Word in Tuple.value do
5:         $Out \leftarrow \langle Word, 1\rangle$
6:         $Emitter.Write(Out)$
7:     end for
8: end procedure
1: procedure REDUCE($Tuple$, $State$, $Emitter$)
2:         $\rightarrow Tuple\langle k_i, v\rangle$ : Incoming tuple with key $k_i$
3:         $\rightarrow State$ : State associated with the key $k_i$
4:         $\rightarrow Emitter$ : Used to emit output tuples
5:     $Word \leftarrow Tuple.key$
6:     $ct \leftarrow$ State.$get$("count")
7:     $ct \leftarrow ct + Tuple.value$
8:     State.$Update$("count", ct)
9:     if $Condition$ then
10:        $out \leftarrow \langle Word, ct\rangle$
11:        $Emitter.write(out)$
12:    end if
13: end procedure
```

Fig. 9. Stateful Streaming MapReduce Word Count Example.

Fig. 10 shows a sample peer-ring configuration consisting of 4 reducer nodes. Assuming that the node (A) is overloaded, it negotiates with its neighbor (B) to check if it has spare cycles (i.e. $q_L \leq \tau_q^{low}$ AND $c \leq \tau_c^{low}$). If so, it requests B to take over some of its load (i.e. share the key space). Node B is then moved along the circle in the counter clockwise direction and its key-space is extended and hence it shares some load from node A. Note that during this process, no explicit state transfer is required since B acted as a *secondary* before load-balancing and hence had a recent checkpoint for the state associated with the given keys.

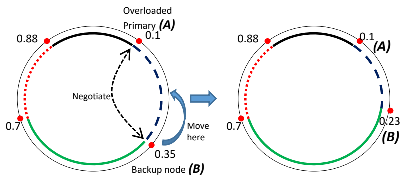


Fig. 10. Load balancing example.

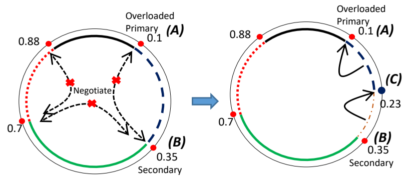


Fig. 11. Scaling out example

Fig. 11 shows a similar transition for scaling out. An overloaded node (A), first sends a load-balance request to its neighbor (B). However, unlike before, the neighbor (B) does not have enough free processing cycles and hence in turn tries to offload some load to its neighbor and so on. If it is observed that all the reducers are processing at its capacity, a new node C is provisioned and placed between A and B, which takes over some of the load from node A. However, note that in this case, node C does not have access to the state associated with the keys. Hence we perform a delayed transition (i.e. we start asynchronous state transfer to node C) while A continues to process the incoming tuple, and the load is transferred only after the state transfer is completed. Finally, Fig. 12 shows the transition for scaling in resources. In this case, since node A is lightly loaded, it checks with its neighbor node B if it has spare cycles to take over all of its load. If so, node A can be removed from the peer-ring and node B is moved into its position. As before, no state or tuple transfer is required to complete the scaling-in transition. Note that even though node B now processes a larger key-space compared to other nodes, the low tuple density allows it to process the tuples without getting overloaded.

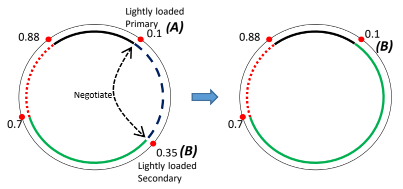


Fig. 12. Scaling in example.

Fig. 13 shows a peer-ring ($r = 1$) distributed across two fault zones. Given the property of fault-zones that the probability of simultaneous failures across two zones is much less than that of within a fault zone, the configuration with consecutive neighbors in the ring lying in distinct fault-zones as shown in fig. 13 gives an optimal fault-tolerance level. In the best case scenario, the system will still be operational without any state or tuple loss even if simultaneous failures are observed in *all* the nodes in a single fault zone (i.e. even if $\frac{n*1}{(1+1)} = n/2$ of the nodes fail at the same time).

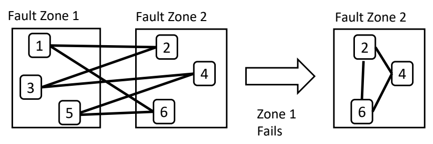


Fig. 13. Reducer peer ring with two fault zones: before and after failure.