Discretized Streams: An Efficient and Fault-Tolerant Model for Stream Processing on Large Clusters

Matei Zaharia, Tathagata Das, Haoyuan Li, Scott Shenker, Ion Stoica
University of California, Berkeley

Abstract

Many important "big data" applications need to process data arriving in real time. However, current programming models for distributed stream processing are relatively low-level, often leaving the user to worry about consistency of state across the system and fault recovery. Furthermore, the models that provide fault recovery do so in an expensive manner, requiring either hot replication or long recovery times. We propose a new programming model, *discretized streams (D-Streams)*, that offers a high-level functional programming API, strong consistency, and efficient fault recovery. D-Streams support a new recovery mechanism that improves efficiency over the traditional replication and upstream backup solutions in streaming databases: *parallel recovery* of lost state across the cluster. We have prototyped D-Streams in an extension to the Spark cluster computing framework called Spark Streaming, which lets users seamlessly intermix streaming, batch and interactive queries.

1 Introduction

Much of "big data" is received in real time, and is most valuable at its time of arrival. For example, a social network may want to identify trending conversation topics within minutes, an ad provider may want to train a model of which users click a new ad, and a service operator may want to mine log files to detect failures within seconds.

To handle the volumes of data and computation they involve, these applications need to be distributed over clusters. However, despite substantial work on cluster programming models for batch computation [6, 22], there are few similarly high-level tools for stream processing. Most current distributed stream processing systems, including Yahoo!'s S4 [19], Twitter's Storm [21], and streaming databases [2, 3, 4], are based on a *record-at-a-time* processing model, where nodes receive each record, update internal state, and send out new records in response. This model raises several challenges in a large-scale cloud environment:

- **Fault tolerance:** Record-at-a-time systems provide recovery through either *replication*, where there are two copies of each processing node, or *upstream backup*, where nodes buffer sent messages and replay them to a second copy of a failed downstream node. Neither approach is attractive in large clusters: replication needs $2\times$ the hardware and may not work if two nodes fail, while upstream backup takes a long time to recover, as the entire system must wait for the standby node to recover the failed node's state.
- **Consistency:** Depending on the system, it can be hard to reason about the global state, because different nodes may be processing data that arrived at different times. For example, suppose that a system counts page views from male users on one node and from females on another. If one of these nodes is backlogged, the ratio of their counters will be wrong.
- **Unification with batch processing:** Because the interface of streaming systems is event-driven, it is quite different from the APIs of batch systems, so users have to write two versions of each analytics task. In addition, it is difficult to *combine* streaming data with historical data, *e.g.,* join a stream of events against historical data to make a decision.

In this work, we present a new programming model, *discretized streams (D-Streams)*, that overcomes these challenges. The key idea behind D-Streams is to treat a streaming computation as a *series of deterministic batch computations on small time intervals*. For example, we might place the data received each second into a new interval, and run a MapReduce operation on each interval to compute a count. Similarly, we can perform a running count over several intervals by adding the new counts from each interval to the old result. Two immediate advantages of the D-Stream model are that consistency is well-defined (each record is processed atomically with the interval in which it arrives), and that the processing model is easy to unify with batch systems. In addition, as we shall show, we can use similar recovery mechanisms to batch systems, albeit at a much smaller timescale, to mitigate failures more efficiently than existing streaming systems, *i.e.,* recover data faster at a lower cost.

There are two key challenges in realizing the D-Stream model. The first is making the latency (interval granularity) low. Traditional batch systems like Hadoop and Dryad fall short here because they keep state on disk between jobs and take tens of seconds to run each

job. Instead, to meet a target latency of several seconds, we keep intermediate state in memory. However, simply putting the state into a general-purpose in-memory storage system, such as a key-value store [17], would be expensive due to the cost of data replication. Instead, we build on Resilient Distributed Datasets (RDDs) [23], a storage abstraction that can rebuild lost data *without* replication by tracking the operations needed to recompute it. Along with a fast execution engine (Spark [23]) that supports tasks as small as 100 ms, we show that we can achieve latencies as low as a second. We believe that this is sufficient for many real-world big data applications, where the timescale of events monitored (*e.g.,* trends in a social network) is much higher.

The second challenge is recovering quickly from failures. Here, we use the deterministic nature of the batch operations in each interval to provide a new recovery mechanism that has not been present in previous streaming systems: *parallel recovery* of a lost node's state. Each node in the cluster works to recompute part of the lost node's RDDs, resulting in faster recovery than upstream backup without the cost of replication. Parallel recovery was hard to implement in record-at-a-time systems due to the complex state maintenance protocols needed even for basic replication (*e.g.,* Flux [20]),[1] but is simple with the deterministic model of D-Streams.

We have prototyped D-Streams in Spark Streaming, an extension to the Spark cluster computing engine [23]. In addition to enabling low-latency stream processing, Spark Streaming interoperates cleanly with Spark's batch and interactive processing features, letting users run ad-hoc queries on arriving streams or mix streaming and historical data from the same high-level API.

2 Discretized Streams (D-Streams)

The key idea behind our model is to treat streaming computations as a series of deterministic batch computations on small time intervals. The input data received during each interval is stored reliably across the cluster to form an *input dataset* for that interval. Once the time interval completes, this dataset is processed via deterministic parallel operations, such as *map*, *reduce* and *groupBy*, to produce new datasets representing program outputs or intermediate state. We store these results in *resilient distributed datasets (RDDs)* [23], an efficient storage abstraction that avoids replication by using lineage for fault recovery, as we shall explain later.

A *discretized stream* or D-Stream groups together a *series* of RDDs and lets the user manipulate them to through various operators. D-Streams provide both *stateless* operators, such as *map*, which act independently on each time interval, and *stateful* operators, such as aggregation over a sliding window, which operate on multiple intervals and may produce intermediate RDDs as state.

[1] The one parallel recovery algorithm we are aware of, by Hwang *et al.* [11], only tolerates one node failure and cannot mitigate stragglers.

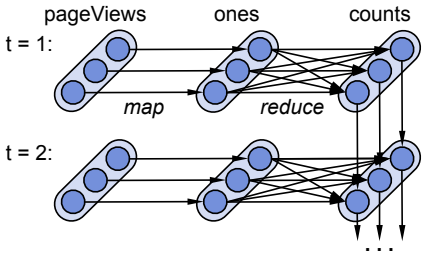


Figure 1: Lineage graph for the RDDs in the view count program. Each oblong shows an RDD, whose partitions are drawn as circles. Lineage is tracked at the granularity of partitions.

We illustrate the idea with a Spark Streaming program that computes a running count of page view events by URL. Spark Streaming provides a language-integrated API similar to DryadLINQ [22] in the Scala language. The code for the view count program is:

```
pageViews = readStream("http://...", "1s")
ones = pageViews.map(event => (event.url, 1))
counts = ones.runningReduce((a, b) => a + b)
```

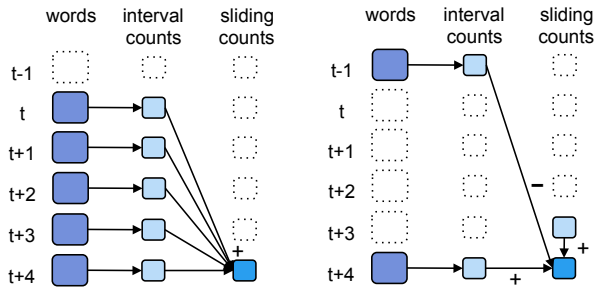
This code creates a D-Stream called `pageViews` by reading an event stream over HTTP, and groups it into 1-second intervals. It then transforms the event stream to get a D-Stream of (URL, 1) pairs called `ones`, and performs a running count of these using the *runningReduce* operator. The arguments to *map* and *runningReduce* are Scala syntax for a closure (function literal).

Finally, to recover from failures, both D-Streams and RDDs track their *lineage*, that is, the set of deterministic operations used to build them. We track this information as a dependency graph, similar to Figure 1. When a node fails, we recompute the RDD partitions that were on it by rerunning the *map*, *reduce*, etc. operations used to build them on the data still in the cluster. The system also periodically checkpoints state RDDs (*e.g.,* by replicating every fifth RDD) to prevent infinite recomputation, but this does not need to happen for all data, because recovery is often fast: the lost partitions can be recomputed *in parallel* on separate nodes, as we shall discuss in Section 3.

D-Stream Operators D-Streams provide two types of operators to let users build streaming programs:

- *Transformation* operators, which produce a new D-Stream from one or more parent streams. These can be either *stateless* (*i.e.,* act independently on each interval) or *stateful* (share data across intervals).
- *Output* operators, which let the program write data to external systems (*e.g.,* save each RDD to HDFS).

D-Streams support the same stateless transformations available in typical batch frameworks [6, 22], including *map*, *reduce*, *groupBy*, and *join*. We reused all of the op-



(a) Naïve evaluation (b) Incremental evaluation

Figure 2: Comparing the naïve variant of *reduceByWindow* (a) with the incremental variant for invertible functions (b). Both versions compute a per-interval count only once, but the second avoids re-summing each window. Boxes denote RDDs, while arrows show the operations used to compute window $[t, t+5)$.

erators in Spark [23]. For example, a program could run a canonical MapReduce word count on each time interval of a D-Stream of sentences using the following code:

```
words = sentences.flatMap(s => s.split(" "))
pairs = words.map(w => (w, 1))
counts = pairs.reduceByKey((a, b) => a + b)
```

In addition, D-Streams introduce new *stateful* operators that work over multiple intervals. These include:

- *Windowing:* The *window* operator groups all of the records from a range of past time intervals into a single RDD. For example, in our earlier code, calling `pairs.window("5s").reduceByKey(_+_)` yields a D-Stream of word counts on intervals $[0, 5)$, $[1, 6)$, $[2, 7)$, etc. *Window* is the most general stateful operator, but it is also often inefficient, as it repeats work.
- *Incremental aggregation:* For the common use case of computing an aggregate value, such as a count or sum, over a sliding window, D-Streams have several variants of a *reduceByWindow* operator. The simplest one only takes an associative "merge" operation for combining values. For example, one might write:

  ```
  pairs.reduceByWindow("5s", (a, b) => a + b)
  ```

  This computes a per-interval count for each time interval only once, but has to add the counts for the past five seconds repeatedly, as in Figure 2a. A more efficient version for *invertible* aggregation functions also takes a function for "subtracting" values and updates state incrementally (Figure 2b).
- *Time-skewed joins:* Users can join a stream against its own RDDs from some time in the past to compute trends—for example, how current page view counts compare to page views five minutes ago.

Finally, the user calls *output operators* to transfer results out of D-Streams into external systems (*e.g.,* for display on a dashboard). We provide two such operators: *save*, which writes each RDD in a D-Stream to a storage system,[2] and *foreach*, which runs a user code snippet (any Spark code) on each RDD in a stream. For example, a user can print the counts computed above with:

```
counts.foreach(rdd => println(rdd.collect()))
```

Unification with Batch and Interactive Processing Because D-Streams follow the same processing model as batch systems, the two can naturally be combined. Spark Streaming provides several powerful features to unify streaming and batch processing.

First, D-Streams can be combined with static RDDs computed, for example, by loading a file. For example, one might join a stream of incoming tweets against a precomputed spam filter, or compare it with historical data.

Second, users can run a D-Stream program as a batch job on previous historical data. This makes it easy compute a new streaming report on past data as well.

Third, users can attach a Scala console to a Spark Streaming program to run ad-hoc queries on D-Streams *interactively*, using Spark's existing fast interactive query capability [23]. For example, the user could query the most popular words in a time range by typing:

```
counts.slice("21:00", "21:05").topK(10)
```

The ability to quickly query any state in the system is invaluable for users troubleshooting issues in real time.

3 Fault Recovery

Classical streaming systems update mutable state on a per-record basis and use either replication or upstream backup for fault recovery [12].

The replication approach creates two or more copies of each process in the data flow graph [2, 20]. Supporting one node failure therefore doubles the hardware cost. Furthermore, if two nodes in the same replica fail, the system is not recoverable. For these reasons, replication is not cost-effective in our large-scale cloud setting.

In upstream backup [12], each upstream node buffers the data sent to downstream nodes locally until it gets an acknowledgement that all related computations are done. When a node fails, the upstream node retransmits all unacknowledged data to a standby node, which takes over the role of the failed node and reprocesses the data. The disadvantage of this approach is long recovery times, as the system must wait for the standby node to catch up.

To address these issues, D-Streams employ a new approach: *parallel recovery*. The system periodically checkpoints some of the state RDDs, by asynchronously replicating them to other nodes. For example, in a view count program computing hourly windows, the system

[2] We can use any storage system supported by Hadoop, *e.g.,* HDFS or HBase, by calling into Hadoop's I/O interfaces to these systems.

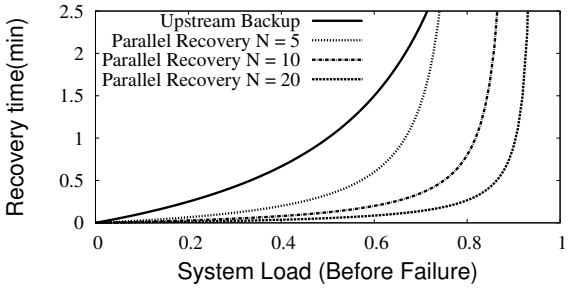


Figure 3: Recovery time for parallel recovery vs. upstream backup on N nodes, as a function of the load before a failure.

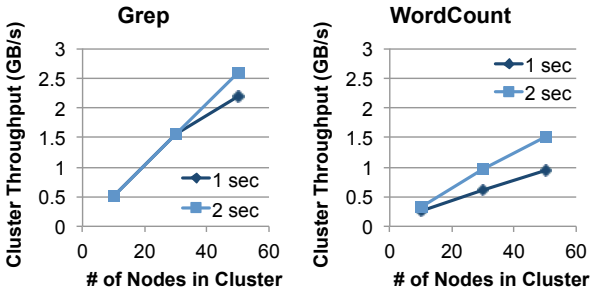


Figure 4: Performance of Grep and sliding WordCount on different cluster sizes. We show the maximum throughput attainable under an end-to-end latency below 1 second or 2 seconds.

could checkpoint results every minute. When a node fails, the system detects the missing RDD partitions and launches tasks to recover them from latest checkpoint. Many fine-grained tasks can be launched *at the same time* to compute different RDD partitions on different nodes. Thus, parallel recovery finishes faster than upstream backup, at a much lower cost than replication.

To show the benefit of this approach, we present results from a simple analytical model in Figure 3. The model assumes that the system is recovering from a minute-old checkpoint and that the bottleneck resource in the recovery process is CPU. In the upstream backup line, a single idle machine performs all of the recovery and then starts processing new records. It takes a long time to catch up at high system loads because new records for it continue to accumulate while it is rebuilding old state.[3] In the other lines, all of the machines partake in recovery, while also processing new records. With more nodes, parallel recovery catches up with the arriving stream much faster than upstream backup.[4]

One reason why parallel recovery was hard to perform in previous streaming systems is that they process data on a per-record basis, which requires complex and costly bookkeeping protocols (*e.g.,* Flux [20]) even for basic replication. In contrast, D-Streams apply deterministic transformations at the much coarser granularity of RDD partitions, which leads to far lighter bookkeeping and simple recovery similar to batch data flow systems [6].

Finally, beside node failures, another important concern in large clusters is stragglers [6]. Fortunately, D-Streams can also recover from stragglers in the same way as batch frameworks like MapReduce, by executing speculative backup copies of slow tasks. This type of speculation would again be difficult in a record-at-a-time system, but becomes simple with deterministic tasks.

[3]For example, when the load is 0.5, the standby node first has to spend 0.5 minutes to recompute the 1 minute of state lost since the checkpoint, then 0.25 minutes to process the data that arrived in those 0.5 minutes, then 0.125 minutes to process the data in this time, etc.

[4]Note that when the system's load before failure is high enough, it can never recover because the load exceeds the available resources.

4 Results

We implemented a prototype of Spark Streaming that extends the existing Spark runtime and can receive streams of data either from the network or from files periodically uploaded to HDFS. We briefly evaluated its scalability and fault recovery using experiments on Amazon EC2. We used nodes with 4 cores and 15 GB RAM.

Scalability We evaluated the scalability of the system through two applications: Grep, which counts input records matching a pattern, and WordCount, which performs a sliding window word count over 10 second windows. For both applications, we measured the maximum throughput achievable on different-sized clusters with an end-to-end latency target of either 1 or 2 seconds. By end-to-end latency, we mean the total time between when a record enters the system and when it appears in a result, including the time it waits for its batch to start. We used a batching interval of 0.5s and 100-byte records.

Figure 4 plots the results. We see that the system can process roughly 40 MB/second/node (400K records/s/node) for Grep and 20 MB/s/node (200K records/s/node) for WordCount at sub-second latency, as well as slightly more data if we allow 2s of latency. The system also scales nearly linearly to 50 nodes. The scaling is not perfect because there are more stragglers with more nodes.

Parallel Recovery We evaluated parallel fault recovery using two applications, both of which received 10 MB/s of data per node on 10 nodes, and used 2-second batching intervals. The first application, *MaxCount*, performed a word count in each 2-second interval, and computed the maximum count for each word over the past 30 seconds using a sliding window. Because max is not an invertible operation, we used the naïve `reduceByWindow` that recomputes every 2s. We ran this application both without any checkpointing (except for replication of the input data). Each interval took **1.66s** to process before the failure, whereas the average processing time of the interval during which a failure happened was **2.64s**

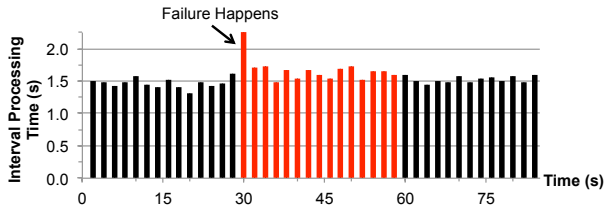


Figure 5: Processing time of intervals during one run of the *WordCount* job. After a failure, the jobs for the next 30s (shown in red) can take slightly longer than average because they may need to recompute the local counts from 30s ago.

(std.dev. 0.19s). Even though results dating back 30 seconds had to be recomputed, this was done in parallel, costing only one extra second of latency.

The second application performed a sliding word count with a 30s window using the the incremental `reduceByWindow` operator, and checkpointed data every 30s. Here, a failure-free interval took **1.47s**, while an interval with a failure took on average **2.31s** (std.dev. 0.43s). Recovery was faster than with *MaxCount* because each interval's output only depends on three previous RDDs (the total count for the previous interval, the local count for the current interval, and the local count for 30 seconds ago). However, one interesting effect was that *any* interval within the next 30s after a failure could exhibit a slowdown, because it might discover that part of the local counts for the interval 30s before it were lost. Figure 5 shows an example of this, where the interval at 30s, when the failure occurs, takes 2.26s to recover, but the intervals at times 32, 34, 46 and 48 also take slightly longer. We plan to eagerly recompute lost RDDs from the past to mitigate this.

5 Related Work

The seminal academic work on stream processing was in streaming databases such as Aurora, Borealis, Telegraph, and STREAM [4, 2, 3, 1]. These systems provide a SQL interface and achieve fault recovery through either replication (an active or passive standby for each node [2, 13]) or upstream backup [12]. We make two contributions over these systems. First, we provide a general programming interface, similar to DryadLINQ [22], instead of just SQL. Second, we provide a more efficient recovery mechanism: parallel recomputation of lost state. Parallel recovery is feasible due to the *deterministic* nature of D-Streams, which lets us recompute lost partitions on other nodes. In contrast, streaming DBs update mutable state for each incoming record, and thus require complex protocols for both replication (*e.g.,* Flux [20]) and upstream backup [12]. The only parallel recovery protocol we are aware of, by Hwang et al [11], only tolerates one node failure, and cannot handle stragglers.

In industry, most stream processing frameworks use a lower-level message passing interface, where users write stateful code to process records in a queue. Examples include S4, Storm, and Flume [19, 21, 7]. These systems generally guarantee at-least-once message delivery, but unlike D-Streams, they require the user to manually handle state recovery on failures (*e.g.,* by keeping all state in a replicated database) and consistency across nodes.

Several recent research systems have looked at online processing in clusters. MapReduce Online [5] is a streaming Hadoop runtime, but cannot compose multiple MapReduce steps into a query or recover stateful reduce tasks. iMR [15] is an in-situ MapReduce engine for log processing, but does not support more general computation graphs and can lose data on failure. CBP [14] and Comet [10] provide "bulk incremental processing" by running MapReduce jobs on new data every few minutes to update state in a distributed file system; however, they incur the high overhead of replicated on-disk storage. In contrast, D-Streams can keep state in memory without costly replication, and achieve order of magnitude lower latencies. Naiad [16] runs computations incrementally, but does not yet have a cluster implementation or a discussion of fault tolerance. Percolator [18] performs incremental computations using triggers, but does not offer consistency guarantees across nodes or high-level operators like *map* and *join*.

Finally, our parallel recovery mechanism is conceptually similar to recovery techniques in MapReduce, GFS, and RAMCloud [6, 9, 17], which all leverage repartitioning. Our contribution is to show that this mechanism can be applied on small enough timescales for stream processing. In addition, unlike GFS and RAMCloud, we *recompute* lost data instead of having to replicate all data, avoiding the network and storage cost of replication.

6 Conclusion

We have presented discretized streams (D-Streams), a stream programming model for large clusters that provides consistency, efficient fault recovery, and powerful integration with batch systems. The key idea is to treat streaming as a series of short batch jobs, and bring down the latency of these jobs as much as possible. This brings many of the benefits of batch processing models to stream processing, including clear consistency semantics and a new parallel recovery technique that we believe is the first truly cost-efficient recovery technique for stream processing in large clusters. Our implementation, Spark Streaming, lets users seamlessly intermix streaming, batch and interactive queries.

In future work, we plan to use Spark Streaming to build integrated systems that combine these types of processing [8], and to further explore the limits of D-Streams. In particular, we are interested in pushing the latency even lower (to about 100 ms) and in recovering from failures faster by providing approximate results.

7 Acknowledgements

This research is supported in part by an NSF CISE Expeditions award, gifts from Google, SAP, Amazon Web Services, Blue Goji, Cisco, Cloudera, Ericsson, General Electric, Hewlett Packard, Huawei, Intel, MarkLogic, Microsoft, NetApp, Oracle, Quanta, Splunk, and VMware, by DARPA (contract #FA8650-11-C-7136), and by a Google PhD Fellowship.

References

[1] A. Arasu, B. Babcock, S. Babu, M. Datar, K. Ito, I. Nishizawa, J. Rosenstein, and J. Widom. STREAM: The Stanford stream data management system. *SIGMOD*, 2003.

[2] M. Balazinska, H. Balakrishnan, S. R. Madden, and M. Stonebraker. Fault-tolerance in the Borealis distributed stream processing system. *ACM Trans. Database Syst.*, 2008.

[3] S. Chandrasekaran, O. Cooper, A. Deshpande, M. J. Franklin, J. M. Hellerstein, W. Hong, S. Krishnamurthy, S. Madden, V. Raman, F. Reiss, and M. Shah. TelegraphCQ: Continuous dataflow processing for an uncertain world. In *CIDR*, 2003.

[4] M. Cherniack, H. Balakrishnan, M. Balazinska, D. Carney, U. Cetintemel, Y. Xing, and S. B. Zdonik. Scalable distributed stream processing. In *CIDR*, 2003.

[5] T. Condie, N. Conway, P. Alvaro, and J. M. Hellerstein. MapReduce online. *NSDI*, 2010.

[6] J. Dean and S. Ghemawat. MapReduce: Simplified data processing on large clusters. In *OSDI*, 2004.

[7] Apache Flume. http://incubator.apache.org/flume/.

[8] M. Franklin, S. Krishnamurthy, N. Conway, A. Li, A. Russakovsky, and N. Thombre. Continuous analytics: Rethinking query processing in a network-effect world. *CIDR*, 2009.

[9] S. Ghemawat, H. Gobioff, and S.-T. Leung. The Google File System. In *Proceedings of SOSP '03*, 2003.

[10] 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 *SoCC '10*.

[11] J. Hwang, Y. Xing, and S. Zdonik. A cooperative, self-configuring high-availability solution for stream processing. In *ICDE*, 2007.

[12] J.-H. Hwang, M. Balazinska, A. Rasin, U. Cetintemel, M. Stonebraker, and S. Zdonik. High-availability algorithms for distributed stream processing. In *ICDE*, 2005.

[13] S. Krishnamurthy, M. Franklin, J. Davis, D. Farina, P. Golovko, A. Li, and N. Thombre. Continuous analytics over discontinuous streams. In *SIGMOD*, 2010.

[14] D. Logothetis, C. Olston, B. Reed, K. C. Webb, and K. Yocum. Stateful bulk processing for incremental analytics. In *SoCC*, 2010.

[15] D. Logothetis, C. Trezzo, K. C. Webb, and K. Yocum. In-situ MapReduce for log processing. In *USENIX ATC*, 2011.

[16] F. McSherry, R. Isaacs, M. Isard, and D. G. Murray. Naiad: The animating spirit of rivers and streams. In *SOSP Poster Session*, 2011.

[17] D. Ongaro, S. M. Rumble, R. Stutsman, J. K. Ousterhout, and M. Rosenblum. Fast crash recovery in RAMCloud. In *SOSP*, 2011.

[18] D. Peng and F. Dabek. Large-scale incremental processing using distributed transactions and notifications. In *OSDI 2010*.

[19] Apache S4. http://incubator.apache.org/s4/.

[20] M. Shah, J. Hellerstein, and E. Brewer. Highly available, fault-tolerant, parallel dataflows. *SIGMOD*, 2004.

[21] Storm. https://github.com/nathanmarz/storm/wiki.

[22] Y. Yu, M. Isard, D. Fetterly, M. Budiu, Ú. Erlingsson, P. K. Gunda, and J. Currey. DryadLINQ: A system for general-purpose distributed data-parallel computing using a high-level language. In *OSDI '08*, 2008.

[23] M. Zaharia, M. Chowdhury, T. Das, A. Dave, J. Ma, M. McCauley, M. Franklin, S. Shenker, and I. Stoica. Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing. In *NSDI*, 2012.