Discretized Streams: Fault-Tolerant Streaming Computation at Scale

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

Abstract

Many "big data" applications must act on data in real time. Running these applications at ever-larger scales requires parallel platforms that automatically handle faults and stragglers. Unfortunately, current distributed stream processing models provide fault recovery in an expensive manner, requiring hot replication or long recovery times, and do not handle stragglers. We propose a new processing model, *discretized streams* (D-Streams), that overcomes these challenges. D-Streams enable a *parallel recovery* mechanism that improves efficiency over traditional replication and backup schemes, and tolerates stragglers. We show that they support a rich set of operators while attaining high per-node throughput similar to single-node systems, linear scaling to 100 nodes, sub-second latency, and sub-second fault recovery. Finally, D-Streams can easily be composed with batch and interactive query models like MapReduce, enabling rich applications that combine these modes. We implement D-Streams in a system called Spark Streaming.

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 wish to detect trending conversation topics in minutes; a search site may wish to model which users visit a new page; and a service operator may wish to monitor program logs to detect failures in seconds. To enable these low-latency processing applications, there is a need for streaming computation models that scale transparently to large clusters, in the same way that batch models like MapReduce simplified offline processing.

Designing such models is challenging, however, because the scale needed for the largest applications (*e.g.,* realtime log processing or machine learning) can be hundreds of nodes. At this scale, two major problems are *faults* and *stragglers* (slow nodes). Both problems are inevitable in large clusters [11], so streaming applications must recover from them quickly. Fast recovery is even *more* important in streaming than it was in batch jobs: while a 30 second delay to recover from a fault or straggler is a nuisance in a batch setting, it can mean losing the chance to make a key decision in a streaming setting.

Unfortunately, existing streaming systems have limited fault and straggler tolerance. Most distributed streaming systems, including Storm [36], TimeStream [32], MapReduce Online [10], and streaming databases [5, 8, 9], are based on a *continuous operator* model, in which long-running, stateful operators receive each record, update internal state, and send new records. While this model is quite natural, it makes it difficult to handle faults and stragglers.

Specifically, given the continuous operator model, systems perform recovery through two approaches [19]: *replication*, where there are two copies of each node [5, 33], or *upstream backup*, where nodes buffer sent messages and replay them to a new copy of a failed node [32, 10, 36]. Neither approach is attractive in large clusters: replication costs 2× the hardware, while upstream backup takes a long time to recover, as the whole system must wait for a new node to serially rebuild the failed node's state by rerunning data through an operator. In addition, neither approach handles stragglers: in upstream backup, a straggler must be treated as a failure, incurring a costly recovery step, while replicated systems use synchronization protocols like Flux [33] to coordinate replicas, so a straggler will slow down both replicas.

This paper presents a new stream processing model, *discretized streams* (D-Streams), that overcomes these challenges. Instead of managing long-lived operators, the idea in D-Streams is to structure a streaming computation as a series of *stateless, deterministic batch computations* on small time intervals. For example, we might place the data received every second (or every 100ms) into an interval, and run a MapReduce operation on each interval to compute a count. Similarly, we can run a rolling count over several intervals by adding the new count from each interval to the old result. By structuring computations this way, D-Streams make (1) the *state* at each timestep fully deterministic given the input data, forgoing the need for synchronization protocols, and (2) the *dependencies* between this state and older data visi-

SOSP'13, Nov. 3–6, 2013, Farmington, Pennsylvania, USA.
ACM 978-1-4503-2388-8/13/11.
http://dx.doi.org/10.1145/2517349.2522737

ble at a fine granularity. We show that this enables powerful recovery mechanisms, similar to those in batch systems, that outperform replication and upstream backup.

There are two challenges in realizing the D-Stream model. The first is making the latency (interval granularity) low. Traditional batch systems, such as Hadoop, fall short here because they keep state in replicated, on-disk storage systems between jobs. Instead, we use a data structure called Resilient Distributed Datasets (RDDs) [42], which keeps data in memory and can recover it without replication by tracking the *lineage graph* of operations that were used to build it. With RDDs, we show that we can attain sub-second end-to-end latencies. We believe that this is sufficient for many real-world big data applications, where the timescale of the events tracked (*e.g.,* trends in social media) is much higher.

The second challenge is recovering quickly from faults and stragglers. Here, we use the determinism of D-Streams to provide a new recovery mechanism that has not been present in previous streaming systems: *parallel recovery* of a lost node's state. When a node fails, each node in the cluster works to recompute part of the lost node's RDDs, resulting in significantly faster recovery than upstream backup without the cost of replication. Parallel recovery was hard to perform in continuous processing systems due to the complex state synchronization protocols needed even for basic replication (*e.g.,* Flux [33]),[1] but becomes simple with the fully deterministic D-Stream model. In a similar way, D-Streams can recover from stragglers using speculative execution [11], while previous streaming systems do not handle them.

We have implemented D-Streams in a system called Spark Streaming, based on the Spark engine [42]. The system can process over 60 million records/second on 100 nodes at sub-second latency, and can recover from faults and stragglers in sub-second time. Spark Streaming's per-node throughput is comparable to commercial streaming databases, while offering linear scalability to 100 nodes, and is 2–5× faster than the open source Storm and S4 systems, while offering fault recovery guarantees that they lack. Apart from its performance, we illustrate Spark Streaming's expressiveness through ports of two real applications: a video distribution monitoring system and an online machine learning system.

Finally, because D-Streams use the same processing model and data structures (RDDs) as batch jobs, a powerful advantage of our model is that streaming queries can seamlessly be *combined* with batch and interactive computation. We leverage this feature in Spark Streaming to let users run ad-hoc queries on streams using Spark, or join streams with historical data computed as an RDD. This is a powerful feature in practice, giving users a single API to combine previously disparate computations. We sketch how we have used it in our applications to blur the line between live and offline processing.

[1] The only parallel recovery algorithm we are aware of, by Hwang *et al.* [20], only tolerates one node failure and cannot handle stragglers.

2 Goals and Background

Many important applications process large streams of data arriving in real time. Our work targets applications that need to run on tens to hundreds of machines, and tolerate a latency of several seconds. Some examples are:

- **Site activity statistics:** Facebook built a distributed aggregation system called Puma that gives advertisers statistics about users clicking their pages within 10–30 seconds and processes 10^6 events/s [34].
- **Cluster monitoring:** Datacenter operators often collect and mine program logs to detect problems, using systems like Flume [3] on hundreds of nodes [16].
- **Spam detection:** A social network such as Twitter may wish to identify new spam campaigns in real time using statistical learning algorithms [38].

For these applications, we believe that the 0.5–2 second latency of D-Streams is adequate, as it is well below the timescale of the trends monitored. We purposely do *not* target applications with latency needs below a few hundred milliseconds, such as high-frequency trading.

2.1 Goals

To run these applications at large scales, we seek a system design that meets four goals:

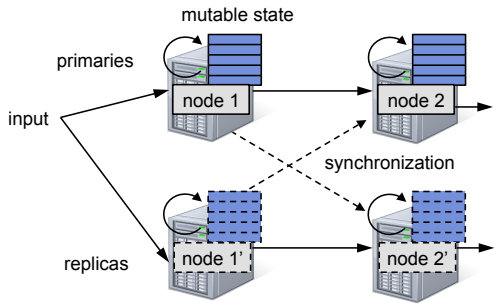
1. Scalability to hundreds of nodes.
2. Minimal cost beyond base processing—we do not wish to pay a 2× replication overhead, for example.
3. Second-scale latency.
4. Second-scale recovery from faults and stragglers.

To our knowledge, previous systems do not meet these goals: replicated systems have high overhead, while upstream backup based ones can take tens of seconds to recover lost state [32, 40], and neither tolerates stragglers.

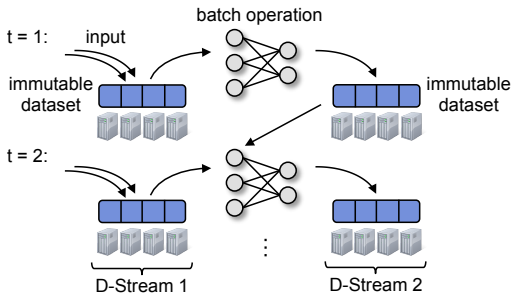
2.2 Previous Processing Models

Though there has been a wide set of work on distributed stream processing, most previous systems use the same *continuous operator* model. In this model, streaming computations are divided into a set of long-lived stateful operators, and each operator processes records as they arrive by updating internal state (*e.g.,* a table tracking page view counts over a window) and sending new records in response [9]. Figure 1(a) illustrates.

While continuous processing minimizes latency, the stateful nature of operators, combined with nondeterminism that arises from record interleaving on the network, makes it hard to provide fault tolerance efficiently. Specifically, the main recovery challenge is rebuilding



(a) Continuous operator processing model. Each node continuously receives records, updates internal state, and emits new records. Fault tolerance is typically achieved through replication, using a synchronization protocol like Flux or DPC [33, 5] to ensure that replicas of each node see records in the same order (*e.g.,* when they have multiple parent nodes).



(b) D-Stream processing model. In each time interval, the records that arrive are stored reliably across the cluster to form an immutable, partitioned dataset. This is then processed via deterministic parallel operations to compute other distributed datasets that represent program output or state to pass to the next interval. Each series of datasets forms one D-Stream.

Figure 1: Comparison of traditional record-at-a-time stream processing (a) with discretized streams (b).

the state of operators on a lost, or slow, node. Previous systems use one of two schemes, *replication* and *upstream backup* [19], which offer a sharp tradeoff between cost and recovery time.

In replication, which is common in database systems, there are two copies of the processing graph, and input records are sent to both. However, simply replicating the nodes is not enough; the system also needs to run a *synchronization protocol*, such as Flux [33] or Borealis's DPC [5], to ensure that the two copies of each operator see messages from upstream parents in the same order. For example, an operator that outputs the union of two parent streams (the sequence of all records received on either one) needs to see the parent streams in the same order to produce the same output stream, so the two copies of this operator need to coordinate. Replication is thus costly, though it recovers quickly from failures.

In upstream backup, each node retains a copy of the messages it sent since some checkpoint. When a node fails, a standby machine takes over its role, and the parents replay messages to this standby to rebuild its state. This approach thus incurs high recovery times, because a single node must recompute the lost state by running data through the serial stateful operator code. TimeStream [32] and MapReduce Online [10] use this model. Popular message queueing systems, like Storm [36], also use this approach, but typically only provide "at-least-once" delivery for *messages*, relying on the user's code to handle state recovery.[2]

More importantly, neither replication nor upstream backup handle stragglers. If a node runs slowly in the replication model, the whole system is affected because of the synchronization required to have the replicas receive messages in the same order. In upstream backup, the only way to mitigate a straggler is to treat it as a failure, which requires going through the slow state recovery process mentioned above, and is heavy-handed for a problem that may be transient.[3] Thus, while traditional streaming approaches work well at smaller scales, they face substantial problems in a large commodity cluster.

3 Discretized Streams (D-Streams)

D-Streams avoid the problems with traditional stream processing by structuring computations as a set of *short, stateless, deterministic tasks* instead of continuous, stateful operators. They then store the state in memory across tasks as fault-tolerant data structures (RDDs) that can be recomputed deterministically. Decomposing computations into short tasks exposes dependencies at a fine granularity and allows powerful recovery techniques like parallel recovery and speculation. Beyond fault tolerance, the D-Stream model gives other benefits, such as powerful unification with batch processing.

3.1 Computation Model

We treat a streaming computation as a series of deterministic batch computations on small time intervals. The data received in 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 either program outputs or intermediate state. In the former case, the results may be pushed to an external sys-

[2] Storm's Trident layer [25] automatically keeps state in a replicated database instead, writing updates in batches. This is expensive, as all updates must be replicated transactionally across the network.

[3] Note that a speculative execution approach as in batch systems would be challenging to apply here because the operator code assumes that it is fed inputs serially, so even a backup copy of an operator would need to spend a long time recovering from its last checkpoint.

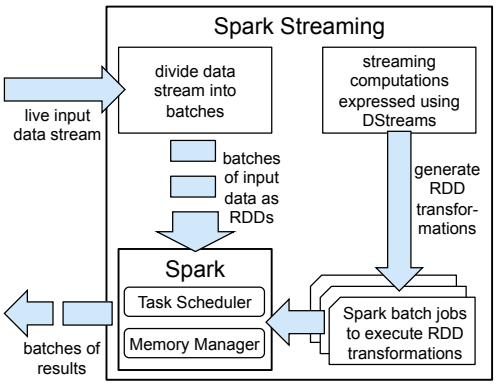


Figure 2: High-level overview of the Spark Streaming system. Spark Streaming divides input data streams into batches and stores them in Spark's memory. It then executes a streaming application by generating Spark jobs to process the batches.

tem in a distributed manner. In the latter case, the intermediate state is stored as *resilient distributed datasets (RDDs)* [42], a fast storage abstraction that avoids replication by using lineage for recovery, as we shall explain. This state dataset may then be processed along with the next batch of input data to produce a new dataset of updated intermediate states. Figure 1(b) shows our model.

We implemented our system, Spark Streaming, based on this model. We used Spark [42] as our batch processing engine for each batch of data. Figure 2 shows a high-level sketch of the computation model in the context of Spark Streaming. This is explained in more detail later.

In our API, users define programs by manipulating objects called *discretized streams (D-Streams)*. A D-Stream is a sequence of immutable, partitioned datasets (RDDs) that can be acted on by deterministic *transformations*. These transformations yield new D-Streams, and may create intermediate *state* in the form of RDDs.

We illustrate the idea with a Spark Streaming program that computes a running count of view events by URL. Spark Streaming exposes D-Streams through a functional API similar to LINQ [41, 2] in the Scala programming language.[4] The code for our 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 these into 1-second intervals. It then transforms the event stream to get a new D-Stream of (URL, 1) pairs called `ones`, and performs a running count of these with a stateful *runningReduce* transformation. The arguments to *map* and *runningReduce* are Scala function literals.

[4]Other interfaces, such as streaming SQL, would also be possible.

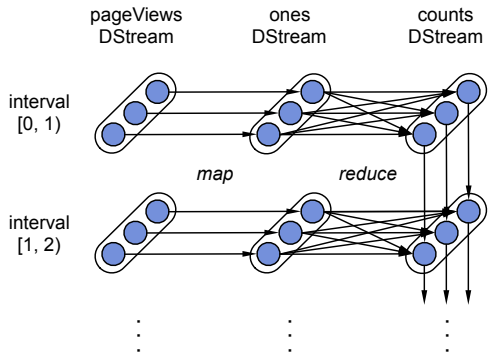


Figure 3: Lineage graph for RDDs in the view count program. Each oval is an RDD, with partitions shown as circles. Each sequence of RDDs is a D-Stream.

To execute this program, Spark Streaming will receive the data stream, divide it into one second batches and store them in Spark's memory as RDDs (see Figure 2). Additionally, it will invoke RDD transformations like *map* and *reduce* to process the RDDs. To execute these transformations, Spark will first launch *map* tasks to process the events and generate the url-one pairs. Then it will launch *reduce* tasks that take both the results of the maps and the results of the previous interval's reduces, stored in an RDD. These tasks will produce a new RDD with the updated counts. Each D-Stream in the program thus turns into a sequence of RDDs.

Finally, to recover from faults and stragglers, both D-Streams and RDDs track their *lineage*, that is, the graph of deterministic operations used to build them [42]. Spark tracks this information at the level of *partitions* within each distributed dataset, as shown in Figure 3. When a node fails, it recomputes the RDD partitions that were on it by re-running the tasks that built them from the original input data stored reliably in the cluster. The system also periodically checkpoints state RDDs (*e.g.,* by asynchronously replicating every tenth RDD)[5] 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. In a similar way, if a node straggles, we can speculatively execute copies of its tasks on other nodes [11], because they will produce the same result.

We note that the parallelism usable for recovery in D-Streams is higher than in upstream backup, even if one ran multiple operators per node. D-Streams expose parallelism across both *partitions* of an operator and *time*:

1. Much like batch systems run multiple tasks per node, each timestep of a transformation may create multiple RDD partitions per node (*e.g.,* 1000 RDD partitions on a 100-core cluster). When the node fails, we can recompute its partitions in parallel on others.

[5]Since RDDs are immutable, checkpointing does not block the job.

2. The lineage graph often enables data from different timesteps to be rebuilt in parallel. For example, in Figure 3, if a node fails, we might lose some *map* outputs from each timestep; the maps from different timesteps can be rerun in parallel, which would not be possible in a continuous operator system that assumes serial execution of each operator.

Because of these properties, D-Streams can parallelize recovery over hundreds of cores and recover in 1–2 seconds even when checkpointing every 30s (§6.2).

In the rest of this section, we describe the guarantees and programming interface of D-Streams in more detail. We then return to our implementation in Section 4.

3.2 Timing Considerations

Note that D-Streams place records into input datasets based on the time when each record *arrives* at the system. This is necessary to ensure that the system can always start a new batch on time, and in applications where the records are generated in the same location as the streaming program, *e.g.,* by services in the same datacenter, it poses no problem for semantics. In other applications, however, developers may wish to group records based on an *external timestamp* of when an event happened, *e.g.,* when a user clicked a link, and records may arrive out of order. D-Streams provide two means to handle this case:

1. The system can *wait* for a limited "slack time" before starting to process each batch.
2. User programs can correct for late records at the *application level*. For example, suppose that an application wishes to count clicks on an ad between time t and $t+1$. Using D-Streams with an interval size of one second, the application could provide a count for the clicks received between t and $t+1$ as soon as time $t+1$ passes. Then, in future intervals, the application could collect any further events with external timestamps between t and $t+1$ and compute an updated result. For example, it could output a *new* count for time interval $[t, t+1)$ at time $t+5$, based on the records for this interval received between t and $t+5$. This computation can be performed with an efficient incremental *reduce* operation that adds the old counts computed at $t+1$ to the counts of new records since then, avoiding wasted work. This approach is similar to order-independent processing [22].

These timing concerns are inherent to stream processing, as any system must handle external delays. They have been studied in detail in databases [22, 35]. In general, any such technique can be implemented over D-Streams by "discretizing" its computation in small batches (running the same logic in batches). Thus, we do not explore these approaches further in this paper.

3.3 D-Stream API

Because D-Streams are primarily an execution strategy (describing how to break a computation into steps), they can be used to implement many of the standard operations in streaming systems, such as sliding windows and incremental processing [9, 4], by simply batching their execution into small timesteps. To illustrate, we describe the operations in Spark Streaming, though other interfaces (*e.g.,* SQL) could also be supported.

In Spark Streaming, users register one or more streams using a functional API. The program can define *input* streams to be read from outside, which receive data either by having nodes listen on a port or by loading it periodically from a storage system (*e.g.,* HDFS). It can then apply two types of operations to these streams:

- *Transformations*, which create a new D-Stream from one or more parent streams. These may be *stateless*, applying separately on the RDDs in each time interval, or they may produce state across intervals.
- *Output operations*, which let the program write data to external systems. For example, the *save* operation will output each RDD in a D-Stream to a database.

D-Streams support the same stateless transformations available in typical batch frameworks [11, 41], including *map*, *reduce*, *groupBy*, and *join*. We provide all the operations in Spark [42]. For example, a program could run a canonical MapReduce word count on each time interval of a D-Stream of words using the following code:

```
pairs = words.map(w => (w, 1))
counts = pairs.reduceByKey((a, b) => a + b)
```

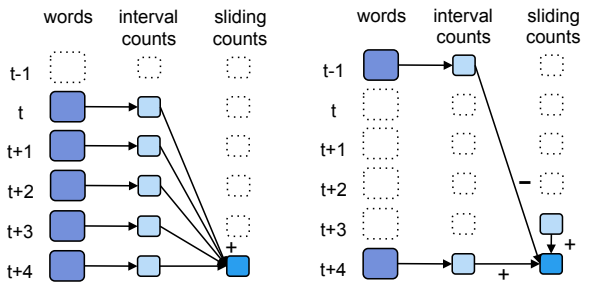
In addition, D-Streams provide several *stateful* transformations for computations spanning multiple intervals, based on standard stream processing techniques such as sliding windows [9, 4]. These include:

Windowing: The *window* operation groups all the records from a sliding window of past time intervals into one RDD. For example, calling `words.window("5s")` in the code above yields a D-Stream of RDDs containing the words in intervals $[0,5)$, $[1,6)$, $[2,7)$, etc.

Incremental aggregation: For the common use case of computing an aggregate, like a count or max, over a sliding window, D-Streams have several variants of an incremental *reduceByWindow* operation. The simplest one only takes an associative merge function for combining values. For instance, in the code above, one can 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 shown in Figure 4(a). If the aggregation function is also *invertible*, a more efficient version also takes a function for "subtracting" values and main-



(a) Associative only (b) Associative & invertible

Figure 4: *reduceByWindow* execution for the associative-only and associative+invertible versions of the operator. 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 $[t, t+5)$.

tains the state incrementally (Figure 4(b)):

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

State tracking: Often, an application has to track *states* for various objects in response to a stream of events indicating state changes. For example, a program monitoring online video delivery may wish to track the number of active *sessions*, where a session starts when the system receives a "join" event for a new client and ends when it receives an "exit" event. It can then ask questions such as "how many sessions have a bitrate above X."
D-Streams provide a *track* operation that transforms streams of (Key, Event) records into streams of (Key, State) records based on three arguments:

- An *initialize* function for creating a State from the first Event for a new key.
- An *update* function for returning a new State given an old State and an Event for its key.
- A *timeout* for dropping old states.

For example, one could count the active sessions from a stream of (ClientID, Event) pairs called as follows:

```
sessions = events.track(
  (key, ev) => 1,          // initialize function
  (key, st, ev) =>         // update function
    ev == Exit ? null : 1,
  "30s")                   // timeout
counts = sessions.count()  // a stream of ints
```

This code sets each client's state to `1` if it is active and drops it by returning `null` from update when it leaves. Thus, `sessions` contains a (ClientID, 1) element for each active client, and `counts` counts the sessions.

These operators are all implemented using the batch operators in Spark, by applying them to RDDs from different times in parent streams. For example, Figure 5

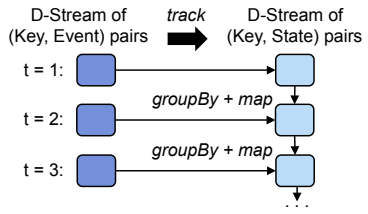


Figure 5: RDDs created by the *track* operation.

shows the RDDs built by *track*, which works by grouping the old states and the new events for each interval.

Finally, the user calls *output operators* to send results out of Spark Streaming into external systems (*e.g.,* for display on a dashboard). We offer two such operators: *save*, which writes each RDD in a D-Stream to a storage system (*e.g.,* HDFS or HBase), and *foreachRDD*, which runs a user code snippet (any Spark code) on each RDD. For example, a user can print the top K counts with `counts.foreachRDD(rdd => print(rdd.top(K)))`.

3.4 Consistency Semantics

One benefit of D-Streams is that they provide clean consistency semantics. Consistency of state across nodes can be a problem in streaming systems that process each record eagerly. For instance, consider a system that counts page views by country, where each page view event is sent to a different node responsible for aggregating statistics for its country. If the node responsible for England falls behind the node for France, *e.g.,* due to load, then a snapshot of their states would be inconsistent: the counts for England would reflect an older prefix of the stream than the counts for France, and would generally be lower, confusing inferences about the events. Some systems, like Borealis [5], synchronize nodes to avoid this problem, while others, like Storm, ignore it.

With D-Streams, the consistency semantics are clear, because time is naturally discretized into intervals, and each interval's output RDDs reflect *all* of the input received in that and previous intervals. This is true regardless of whether the output and state RDDs are distributed across the cluster—users do not need to worry about whether nodes have fallen behind each other. Specifically, the result in each output RDD, when computed, is the same as if all the batch jobs on previous intervals had run in lockstep and there were no stragglers and failures, simply due to the determinism of computations and the separate naming of datasets from different intervals. Thus, D-Streams provide consistent, "exactly-once" processing across the cluster.

3.5 Unification with Batch & Interactive Processing

Because D-Streams follow the same processing model, data structures (RDDs), and fault tolerance mechanisms as batch systems, the two can seamlessly be combined.

Aspect	D-Streams	Continuous proc. systems
Latency	0.5–2 s	1–100 ms unless records are batched for consistency
Consistency	Records processed atomically with interval they arrive in	Some systems wait a short time to sync operators before proceeding [5, 32]
Late records	Slack time or app-level correction	Slack time, out of order processing [22, 35]
Fault recovery	Fast parallel recovery	Replication or serial recovery on one node
Straggler recovery	Possible via speculative execution	Typically not handled
Mixing w/ batch	Simple unification through RDD APIs	In some DBs [14]; not in message queueing systems

Table 1: Comparing D-Streams with record-at-a-time systems.

Spark Streaming provides several powerful features to unify streaming and batch processing.

First, D-Streams can be combined with static RDDs computed using a standard Spark job. For instance, one can *join* a stream of message events against a precomputed spam filter, or compare them with historical data.

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

Third, users run ad-hoc queries on D-Streams *interactively* by attaching a Scala console to their Spark Streaming program and running arbitrary Spark operations on the RDDs there. For example, the user could query the most popular words in a time range by typing:

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

Discussions with developers who have written both offline (Hadoop-based) and online processing applications show that these features have significant practical value. Simply having the data types and functions used for these programs in the same codebase saves substantial development time, as streaming and batch systems currently have separate APIs. The ability to also query state in the streaming system interactively is even more attractive: it makes it simple to debug a running computation, or to ask queries that were not anticipated when defining the aggregations in the streaming job, *e.g.,* to troubleshoot an issue with a website. Without this ability, users typically need to wait tens of minutes for the data to make it into a batch cluster, even though all the relevant state is in memory on stream processing nodes.

3.6 Summary

To end our overview of D-Streams, we compare them with continuous operator systems in Table 1. The main difference is that D-Streams divide work into small, deterministic tasks operating on batches. This raises their minimum latency, but lets them employ highly efficient recovery techniques. In fact, some continuous operator systems, like TimeStream and Borealis [32, 5], *also* delay records, in order to deterministically execute operators that have multiple upstream parents (by waiting for periodic "punctuations" in streams) and to provide consistency. This raises their latency past the millisecond scale and into the second scale of D-Streams.

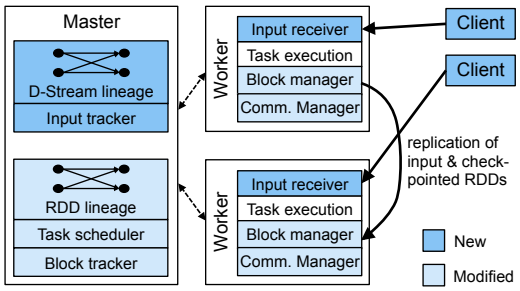


Figure 6: Components of Spark Streaming, showing what we added and modified over Spark.

4 System Architecture

We have implemented D-Streams in a system called Spark Streaming, based on a modified version of the Spark processing engine [42]. Spark Streaming consists of three components, shown in Figure 6:

- A *master* that tracks the D-Stream lineage graph and schedules *tasks* to compute new RDD partitions.
- *Worker nodes* that receive data, store the partitions of input and computed RDDs, and execute tasks.
- A *client library* used to send data into the system.

As shown in the figure, Spark Streaming reuses many components of Spark, but we also modified and added multiple components to enable streaming. We discuss those changes in Section 4.2.

From an architectural point of view, the main difference between Spark Streaming and traditional streaming systems is that Spark Streaming divides its computations into short, stateless, deterministic *tasks*, each of which may run on any node in the cluster, or even on multiple nodes. Unlike the rigid topologies in traditional systems, where moving part of the computation to another machine is a major undertaking, this approach makes it straightforward to balance load across the cluster, react to failures, or launch speculative copies of slow tasks. It matches the approach used in batch systems, such as MapReduce, for the same reasons. However, tasks in Spark Streaming are far shorter, usually just 50–200 ms, due to running on in-memory RDDs.

All state in Spark Streaming is stored in fault-tolerant data structures (RDDs), instead of being part of a long-running operator process as in previous systems. RDD partitions can reside on any node, and can even be com-

puted on multiple nodes, because they are computed deterministically. The system tries to place both state and tasks to maximize data locality, but this underlying flexibility makes speculation and parallel recovery possible.

These benefits come naturally from running on a batch platform (Spark), but we also had to make significant changes to support streaming. We discuss job execution in more detail before presenting these changes.

4.1 Application Execution

Spark Streaming applications start by defining one or more input streams. The system can load streams either by receiving records directly from clients, or by loading data periodically from an external storage system, such as HDFS, where it might be placed by a log collection system [3]. In the former case, we ensure that new data is replicated across two worker nodes before sending an acknowledgement to the client library, because D-Streams require input data to be stored reliably to recompute results. If a worker fails, the client library sends unacknowledged data to another worker.

All data is managed by a *block store* on each worker, with a tracker on the master to let nodes find the locations of blocks. Because both our input blocks and the RDD partitions we compute from them are immutable, keeping track of the block store is straightforward—each block is simply given a unique ID, and any node that has that ID can serve it (*e.g.,* if multiple nodes computed it). The block store keeps new blocks in memory but drops them in an LRU fashion, as described later.

To decide when to start processing a new interval, we assume that the nodes have their clocks synchronized via NTP, and have each node send the master a list of block IDs it received in each interval when it ends. The master then starts launching tasks to compute the output RDDs for the interval, *without* requiring any further kind of synchronization. Like other batch schedulers [21], it simply starts each task whenever its parents are finished.

Spark Streaming relies on Spark's existing batch scheduler within each timestep [42], and performs many of the optimizations in systems like DryadLINQ [41]:

- It pipelines operators that can be grouped into a single task, such as a *map* followed by another *map*.
- It places tasks based on data locality.
- It controls the *partitioning* of RDDs to avoid shuffling data across the network. For example, in a *reduceByWindow* operation, each interval's tasks need to "add" the new partial results from the current interval (*e.g.,* a click count for each page) and "subtract" the results from several intervals ago. The scheduler partitions the state RDDs for different intervals in the same way, so that data for each key (*e.g.,* a page) is consistently on the same node across timesteps. More details are given in [42].

4.2 Optimizations for Stream Processing

While Spark Streaming builds on Spark, we also had to make sigificant optimizations and changes to this batch engine to support streaming. These included:

Network communication: We rewrote Spark's data plane to use asynchronous I/O to let tasks with remote inputs, such as reduce tasks, fetch them faster.

Timestep pipelining: Because the tasks inside each timestep may not perfectly utilize the cluster (*e.g.,* at the end of the timestep, there might only be a few tasks left running), we modified Spark's scheduler to allow submitting tasks from the next timestep *before* the current one has finished. For example, consider our first *map* + *runningReduce* job in Figure 3. Because the maps at each step are independent, we can begin running the maps for timestep 2 before timestep 1's reduce finishes.

Task Scheduling: We made multiple optimizations to Spark's task scheduler, such as hand-tuning the size of control messages, to be able to launch parallel jobs of hundreds of tasks every few hundred milliseconds.

Storage layer: We rewrote Spark's storage layer to support asynchronous checkpointing of RDDs and to increase performance. Because RDDs are immutable, they can be checkpointed over the network without blocking computations on them and slowing jobs. The new storage layer also uses zero-copy I/O for this when possible.

Lineage cutoff: Because lineage graphs between RDDs in D-Streams can grow indefinitely, we modified the scheduler to forget lineage after an RDD has been checkpointed, so that its state does not grow arbitrarily. Similarly, other data structures in Spark that grew without bound were given a periodic cleanup process.

Master recovery: Because streaming applications need to run 24/7, we added support for recovering the Spark master's state if it fails (Section 5.3).

Interestingly, the optimizations for stream processing also improved Spark's performance in batch benchmarks by as much as 2×. This is a powerful benefit of using the same engine for stream and batch processing.

4.3 Memory Management

In our current implementation of Spark Streaming, each node's block store manages RDD partitions in an LRU fashion, dropping data to disk if there is not enough memory. In addition, the user can set a maximum history timeout, after which the system will simply forget old blocks without doing disk I/O (this timeout must be bigger than the checkpoint interval). We found that in many applications, the memory required by Spark Streaming is not onerous, because the state within a computation is typically much smaller than the input data (many appli-

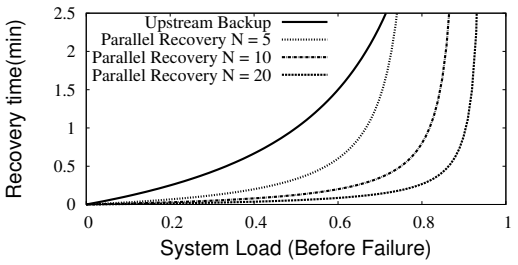


Figure 7: Recovery time for single-node upstream backup vs. parallel recovery on N nodes, as a function of the load before a failure. We assume the time since the last checkpoint is 1 min.

cations compute aggregate statistics), and any reliable streaming system needs to replicate data received over the network to multiple nodes, as we do. However, we also plan to explore ways to prioritize memory use.

5 Fault and Straggler Recovery

The deterministic nature of D-Streams makes it possible to use two powerful recovery techniques for worker state that are hard to apply in traditional streaming systems: parallel recovery and speculative execution. In addition, it simplifies master recovery, as we shall also discuss.

5.1 Parallel Recovery

When a node fails, D-Streams allow the state RDD partitions that were on the node, and all tasks that it was currently running, to be recomputed in parallel on other nodes. The system periodically *checkpoints* some of the state RDDs, by asynchronously replicating them to other worker nodes.[6] For example, in a program computing a running count of page views, the system could choose to checkpoint the counts every minute. Then, when a node fails, the system detects all missing RDD partitions and launches tasks to recompute them from the last checkpoint. Many tasks can be launched *at the same time* to compute different RDD partitions, allowing the whole cluster to partake in recovery. As described in Section 3, D-Streams exploit parallelism both across *partitions* of the RDDs in each timestep and across *timesteps* for independent operations (*e.g.,* an initial *map*), as the lineage graph captures dependencies at a fine granularity.

To show the benefit of parallel recovery, Figure 7 compares it with single-node upstream backup using a simple analytical model. The model assumes that the system is recovering from a minute-old checkpoint.

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 loads because new records for it continue to arrive while it is rebuilding old state. Indeed, suppose that the load before failure was λ. Then during each minute of recovery, the backup node can do 1 min of work, but receives λ minutes of new work. Thus, it fully recovers from the λ units of work that the failed node did since the last checkpoint at a time t_{up} such that $t_{\text{up}} \cdot 1 = \lambda + t_{\text{up}} \cdot \lambda$, which is

$$t_{\text{up}} = \frac{\lambda}{1-\lambda}.$$

In the other lines, all of the machines partake in recovery, while also processing new records. Supposing there where N machines in the cluster before the failure, the remaining $N-1$ machines now each have to recover λ/N work, but also receive new data at a rate of $\frac{N}{N-1}\lambda$. The time t_{par} at which they catch up with the arriving stream satisfies $t_{\text{par}} \cdot 1 = \frac{\lambda}{N} + t_{\text{par}} \cdot \frac{N}{N-1}\lambda$, which gives

$$t_{\text{par}} = \frac{\lambda/N}{1-\frac{N}{N-1}\lambda} \approx \frac{\lambda}{N(1-\lambda)}.$$

Thus, with more nodes, parallel recovery catches up with the arriving stream much faster than upstream backup.

5.2 Straggler Mitigation

Besides failures, another concern in large clusters is stragglers [11]. Fortunately, D-Streams also let us mitigate stragglers like batch systems do, by running speculative backup copies of slow tasks. Such speculation would be difficult in a continuous operator system, as it would require launching a new copy of a node, populating its state, and overtaking the slow copy. Indeed, replication algorithms for stream processing, such as Flux and DPC [33, 5], focus on *synchronizing* two replicas.

In our implementation, we use a simple threshold to detect stragglers: whenever a task runs more than $1.4\times$ longer than the median task in its job stage, we mark it as slow. More refined algorithms could also be used, but we show that this method still works well enough to recover from stragglers within a second.

5.3 Master Recovery

A final requirement to run Spark Streaming 24/7 was to tolerate failures of Spark's master. We do this by (1) writing the state of the computation reliably when starting each timestep and (2) having workers connect to a new master and report their RDD partitions to it when the old master fails. A key aspect of D-Streams that simplifies recovery is that *there is no problem if a given RDD is computed twice*. Because operations are deterministic, such an outcome is similar to recovering from a failure.[7] This means that it is fine to lose some running tasks while the master reconnects, as they can be redone.

[6] Because RDDs are immutable, checkpointing does not block the current timestep's execution.

[7] One subtle issue here is output operators; we have designed operators like *save* to be idempotent, so that the operator outputs each timestep's worth of data to a known path, and does not overwrite previous data if that timestep was already computed.

Our current implementation stores D-Stream metadata in HDFS, writing (1) the graph of the user's D-Streams and Scala function objects representing user code, (2) the time of the last checkpoint, and (3) the IDs of RDDs since the checkpoint in an HDFS file that is updated through an atomic rename on each timestep. Upon recovery, the new master reads this file to find where it left off, and reconnects to the workers to determine which RDD partitions are in memory on each one. It then resumes processing each timestep missed. Although we have not yet optimized the recovery process, it is reasonably fast, with a 100-node cluster resuming work in 12 seconds.

6 Evaluation

We evaluated Spark Streaming using both several benchmark applications and by porting two real applications to it: a commercial video distribution monitoring system and a machine learning algorithm for estimating traffic conditions from automobile GPS data [18]. These latter applications also leverage D-Streams' unification with batch processing, as we shall discuss.

6.1 Performance

We tested the performance of the system using three applications of increasing complexity: Grep, which finds the number of input strings matching a pattern; WordCount, which performs a sliding window count over 30s; and TopKCount, which finds the k most frequent words over the past 30s. The latter two applications used the incremental *reduceByWindow* operator. We first report the raw scaling performance of Spark Streaming, and then compare it against two widely used streaming systems, S4 from Yahoo! and Storm from Twitter [28, 36]. We ran these applications on "m1.xlarge" nodes on Amazon EC2, each with 4 cores and 15 GB RAM.

Figure 8 reports the maximum throughput that Spark Streaming can sustain while keeping the end-to-end latency below a given target. By "end-to-end latency," we mean the time from when records are sent to the system to when results incorporating them appear. Thus, the latency includes the time to wait for a new input batch to start. For a 1 second latency target, we use 500 ms input intervals, while for a 2 s target, we use 1 s intervals. In both cases, we used 100-byte input records.

We see that Spark Streaming scales nearly linearly to 100 nodes, and can process up to 6 GB/s (64M records/s) at sub-second latency on 100 nodes for Grep, or 2.3 GB/s (25M records/s) for the other, more CPU-intensive jobs.[8] Allowing a larger latency improves throughput slightly, but even the performance at sub-second latency is high.

[8] Grep was network-bound due to the cost to replicate the input data to multiple nodes—we could not get the EC2 network to send more than 68 MB/s per node. WordCount and TopK were more CPU-heavy, as they do more string processing (hashes & comparisons).

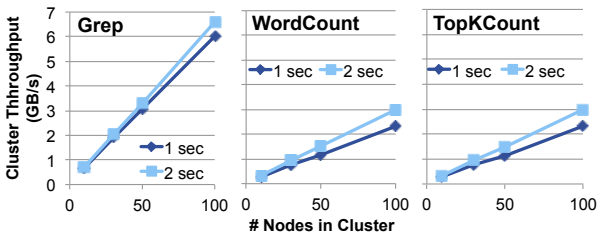


Figure 8: Maximum throughput attainable under a given latency bound (1 s or 2 s) by Spark Streaming.

Comparison with Commercial Systems Spark Streaming's per-node throughput of 640,000 records/s for Grep and 250,000 records/s for TopKCount on 4-core nodes is comparable to the speeds reported for commercial single-node streaming systems. For example, Oracle CEP reports a throughput of 1 million records/s on a 16-core machine [30], StreamBase reports 245,000 records/s on 8 cores [39], and Esper reports 500,000 records/s on 4 cores [12]. While there is no reason to expect D-Streams to be slower or faster per-node, the key advantage is that Spark Streaming scales nearly linearly to 100 nodes.

Comparison with S4 and Storm We also compared Spark Streaming against two open source distributed streaming systems, S4 and Storm. Both are continuous operators systems that do not offer consistency across nodes and have limited fault tolerance guarantees (S4 has none, while Storm guarantees at-least-once delivery of records). We implemented our three applications in both systems, but found that S4 was limited in the number of records/second it could process per node (at most 7500 records/s for Grep and 1000 for WordCount), which made it almost 10× slower than Spark and Storm. Because Storm was faster, we also tested it on a 30-node cluster, using both 100-byte and 1000-byte records.

We compare Storm with Spark Streaming in Figure 9, reporting the throughput Spark attains at sub-second latency. We see that Storm is still adversely affected by smaller record sizes, capping out at 115K records/s/node for Grep for 100-byte records, compared to 670K for Spark. This is despite taking several precautions in our Storm implementation to improve performance, including sending "batched" updates from Grep every 100 input records and having the "reduce" nodes in WordCount and TopK only send out new counts every second, instead of each time a count changes. Storm was faster with 1000-byte records, but still 2× slower than Spark.

6.2 Fault and Straggler Recovery

We evaluated fault recovery under various conditions using the WordCount and Grep applications. We used 1-second batches with input data residing in HDFS, and set the data rate to 20 MB/s/node for WordCount and

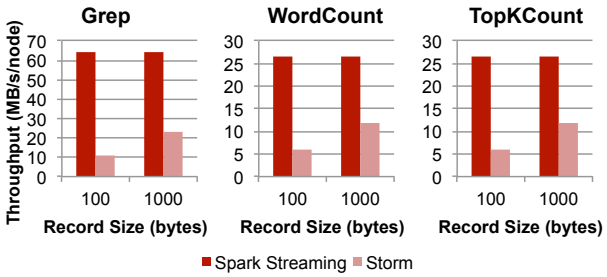


Figure 9: Throughput vs Storm on 30 nodes.

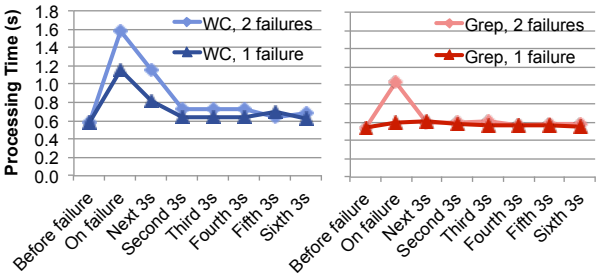


Figure 10: Interval processing times for WordCount (WC) and Grep under failures. We show the average time to process each 1s batch of data before a failure, during the interval of the failure, and during 3-second periods after. Results are over 5 runs.

80 MB/s/node for Grep, which led to a roughly equal per-interval processing time of 0.58s for WordCount and 0.54s for Grep. Because the WordCount job performs an incremental *reduceByKey*, its lineage graph grows indefinitely (since each interval subtracts data from 30 seconds in the past), so we gave it a checkpoint interval of 10 seconds. We ran the tests on 20 four-core nodes, using 150 map tasks and 10 reduce tasks per job.

We first report recovery times under these these base conditions, in Figure 10. The plot shows the average processing time of 1-second data intervals before the failure, during the interval of failure, and during 3-second periods thereafter, for either 1 or 2 concurrent failures. (The processing for these later periods is delayed while recovering data for the interval of failure, so we show how the system restabilizes.) We see that recovery is fast, with delays of at most 1 second even for two failures and a 10s checkpoint interval. WordCount's recovery takes longer because it has to recompute data going far back, whereas Grep just loses four tasks on each failed node.

Varying the Checkpoint Interval Figure 11 shows the effect of changing WordCount's checkpoint interval. Even when checkpointing every 30s, results are delayed at most 3.5s. With 2s checkpoints, the system recovers in just 0.15s, while still paying less than full replication.

Varying the Number of Nodes To see the effect of parallelism, we also tried the WordCount application on

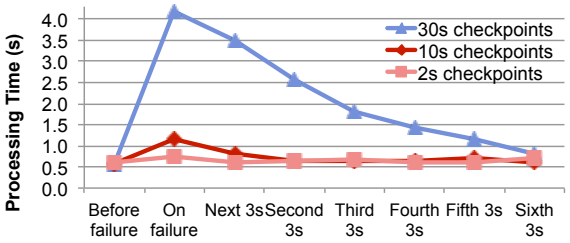


Figure 11: Effect of checkpoint time in WordCount.

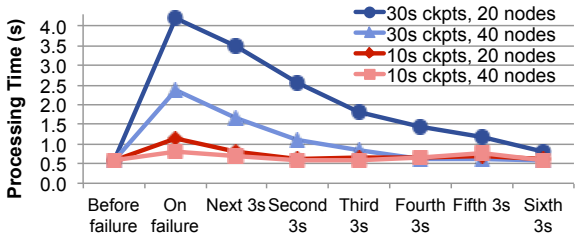


Figure 12: Recovery of WordCount on 20 & 40 nodes.

40 nodes. As Figure 12 shows, doubling the nodes reduces the recovery time in half. While it may seem surprising that there is so much parallelism given the linear dependency chain of the sliding *reduceByWindow* operator in WordCount, the parallelism comes because the *local* aggregations on each timestep can be done in parallel (see Figure 4), and these are the bulk of the work.

Straggler Mitigation Finally, we tried slowing down one of the nodes instead of killing it, by launching a 60-thread process that overloaded the CPU. Figure 13 shows the per-interval processing times without the straggler, with the straggler but with speculative execution (backup tasks) disabled, and with the straggler and speculation enabled. Speculation improves the response time significantly. Note that our current implementation does *not* attempt to remember straggler nodes across time, so these improvements occur despite repeatedly launching new tasks on the slow node. This shows that even unexpected stragglers can be handled quickly. A full implementation would blacklist slow nodes.

6.3 Real Applications

We evaluated the expressiveness of D-Streams by porting two real applications. Both applications are significantly more complex than the test programs shown so far, and both took advantage of D-Streams to perform batch or interactive processing in addition to streaming.

6.3.1 Video Distribution Monitoring

Conviva provides a commercial management platform for video distribution over the Internet. One feature of this platform is the ability to track the performance across different geographic regions, CDNs, client devices, and ISPs, which allows the broadcasters to quickly

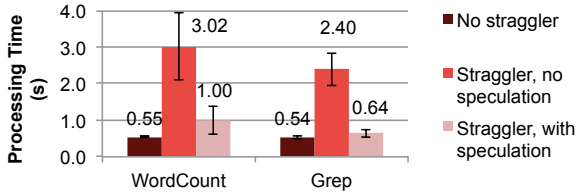
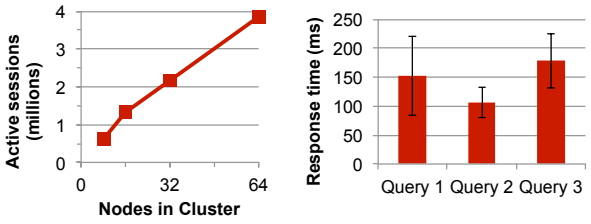


Figure 13: Processing time of intervals in Grep and WordCount in normal operation, as well as in the presence of a straggler, with and without speculation.



(a) Scalability (b) Ad-hoc queries

Figure 14: Results for the video application. (a) shows the number of client sessions supported vs. cluster size. (b) shows the performance of three ad-hoc queries from the Spark shell, which count (1) all active sessions, (2) sessions for a specific customer, and (3) sessions that have experienced a failure.

idenify and respond to delivery problems. The system receives events from video players and uses them to compute more than fifty metrics, including complex metrics such as unique viewers and session-level metrics such as buffering ratio, over different grouping categories.

The current application is implemented in two systems: a custom-built distributed streaming system for live data, and a Hadoop/Hive implementation for historical data and ad-hoc queries. Having both live and historical data is crucial because customers often want to go back in time to debug an issue, but implementing the application on these two separate systems creates significant challenges. First, the two implementations have to be kept in sync to ensure that they compute metrics in the same way. Second, there is a lag of several minutes minutes before data makes it through a sequence of Hadoop import jobs into a form ready for ad-hoc queries.

We ported the application to D-Streams by wrapping the *map* and *reduce* implementations in the Hadoop version. Using a 500-line Spark Streaming program and an additional 700-line wrapper that executed Hadoop jobs within Spark, we were able to compute all the metrics (a 2-stage MapReduce job) in batches as small as 2 seconds. Our code uses the *track* operator described in Section 3.3 to build a session state object for each client ID and update it as events arrive, followed by a sliding *reduceByKey* to aggregate the metrics over sessions.

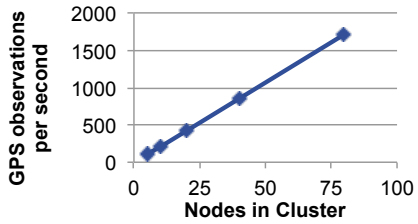


Figure 15: Scalability of the *Mobile Millennium* job.

We measured the scaling performance of the application and found that on 64 quad-core EC2 nodes, it could process enough events to support 3.8 million concurrent viewers, which exceeds the peak load experienced at Conviva so far. Figure 14(a) shows the scaling.

In addition, we used D-Streams to add a *new* feature not present in the original application: ad-hoc queries on the live stream state. As shown in Figure 14(b), Spark Streaming can run ad-hoc queries from a Scala shell in less than a second on the RDDs representing session state. Our cluster could easily keep ten minutes of data in RAM, closing the gap between historical and live processing, and allowing a single codebase to do both.

6.3.2 Crowdsourced Traffic Estimation

We applied the D-Streams to the *Mobile Millennium* traffic information system [18], a machine learning based project to estimate automobile traffic conditions in cities. While measuring traffic for highways is straightforward due to dedicated sensors, *arterial roads* (the roads in a city) lack such infrastructure. *Mobile Millennium* attacks this problem by using crowdsourced GPS data from fleets of GPS-equipped cars (*e.g.,* taxi cabs) and cellphones running a mobile application.

Traffic estimation from GPS data is challenging, because the data is noisy (due to GPS inaccuracy near tall buildings) and sparse (the system only receives one measurement from each car per minute). *Mobile Millennium* uses a highly compute-intensive expectation maximization (EM) algorithm to infer the conditions, using Markov Chain Monte Carlo and a traffic model to estimate a travel time distribution for each road link. The previous implementation [18] was an iterative batch job in Spark that ran over 30-minute windows of data.

We ported this application to Spark Streaming using an online version of the EM algorithm that merges in new data every 5 seconds. The implementation was 260 lines of Spark Streaming code, and wrapped the existing *map* and *reduce* functions in the offline program. In addition, we found that only using the real-time data could cause overfitting, because the data received in five seconds is so sparse. We took advantage of D-Streams to also combine this data with *historical* data from the same time during the past ten days to resolve this problem.

Figure 15 shows the performance of the algorithm on

up to 80 quad-core EC2 nodes. The algorithm scales nearly perfectly because it is CPU-bound, and provides answers more than 10× faster than the batch version.[9]

7 Discussion

We have presented discretized streams (D-Streams), a new stream processing model for clusters. By breaking computations into short, deterministic tasks and storing state in lineage-based data structures (RDDs), D-Streams can use powerful recovery mechanisms, similar to those in batch systems, to handle faults and stragglers.

Perhaps the main limitation of D-Streams is that they have a fixed minimum latency due to batching data. However, we have shown that total delay can still be as low as 1–2 seconds, which is enough for many real-world use cases. Interestingly, even some continuous operator systems, such as Borealis and TimeStream [5, 32], add delays to ensure determinism: Borealis's SUnion operator and TimeStream's HashPartition wait to batch data at "heartbeat" boundaries so that operators with multiple parents see input in a deterministic order. Thus, D-Streams' latency is in a similar range to these systems, while offering significantly more efficient recovery.

Beyond their recovery benefits, we believe that the most important aspect of D-Streams is that they show that streaming, batch and interactive computations can be unified in the same platform. As "big" data becomes the *only* size of data at which certain applications can operate (*e.g.,* spam detection on large websites), organizations will need the tools to write both lower-latency applications and more interactive ones that use this data, not just the periodic batch jobs used so far. D-Streams integrate these modes of computation at a deep level, in that they follow not only a similar API but also the same data structures and fault tolerance model as batch jobs. This enables rich features like combining streams with offline data or running ad-hoc queries on stream state.

Finally, while we presented a basic implementation of D-Streams, there are several areas for future work:

Expressiveness: In general, as the D-Stream abstraction is primarily an *execution* strategy, it should be possible to run most streaming algorithms within them, by simply "batching" the execution of the algorithm into steps and emitting state across them. It would be interesting to port languages like streaming SQL [4] or Complex Event Processing models [13] over them.

Setting the batch interval: Given any application, setting an appropriate batch interval is very important as it directly determines the trade-off between the end-to-end latency and the throughput of the streaming workload. Currently, a developer has to explore this trade-off and determine the batch interval manually. It may be possible for the system to tune it automatically.

Memory usage: Our model of stateful stream processing generates new a RDD to store each operator's state after each batch of data is processed. In our current implementation, this will incur a higher memory usage than continuous operators with mutable state. Storing different versions of the state RDDs is essential for the system perform lineage-based fault recovery. However, it may be possible to reduce the memory usage by storing only the deltas between these state RDDs.

Approximate results: In addition to recomputing lost work, another way to handle a failure is to return approximate partial results. D-Streams provide the opportunity to compute partial results by simply launching a task *before* its parents are all done, and offer lineage data to know *which* parents were missing.

8 Related Work

Streaming Databases Streaming databases such as Aurora, Telegraph, Borealis, and STREAM [7, 8, 5, 4] were the earliest academic systems to study streaming, and pioneered concepts such as windows and incremental operators. However, *distributed* streaming databases, such as Borealis, used replication or upstream backup for recovery [19]. We make two contributions over them.

First, D-Streams provide a more efficient recovery mechanism, parallel recovery, that runs faster than upstream backup without the cost of replication. Parallel recovery is feasible because D-Streams discretize computations into stateless, deterministic tasks. In contrast, streaming databases use a stateful continous operator model, and require complex protocols for both replication (*e.g.,* Borealis's DPC [5] or Flux [33]) and upstream backup [19]. The only parallel recovery protocol we are aware of, by Hwang et al [20], only tolerates one node failure, and cannot handle stragglers.

Second, D-Streams also tolerate stragglers, using speculative execution [11]. Straggler mitigation is difficult in continuous operator models because each node has mutable state that cannot be rebuilt on another node without a costly serial replay process.

Large-scale Streaming While several recent systems enable streaming computation with high-level APIs similar to D-Streams, they also lack the fault and straggler recovery benefits of the discretized stream model.

TimeStream [32] runs the continuous, stateful operators in Microsoft StreamInsight [2] on a cluster. It uses a recovery mechanism similar to upstream backup that tracks which upstream data each operator depends on and replays it serially through a new copy of the operator. Recovery thus happens on a single node for each op-

[9] Note that the raw rate of records/second for this algorithm is lower than in our other programs because it performs far more work for each record, drawing 300 Markov Chain Monte Carlo samples per record.

erator, and takes time proportional to that operator's processing window (*e.g.,* 30 seconds for a 30-second sliding window) [32]. In contrast, D-Streams use stateless transformations and explicitly put state in data structures (RDDs) that can (1) be checkpointed asynchronously to bound recovery time and (2) be rebuilt in parallel, exploiting parallelism across data partitions and timesteps to recover in sub-second time. D-Streams can also handle stragglers, while TimeStream does not.

Naiad [26, 27] automatically incrementalizes data flow computations written in LINQ and is unique in also being able to incrementalize *iterative* computations. However, it uses traditional synchronous checkpointing for fault tolerance, and cannot respond to stragglers.

MillWheel [1] runs stateful computations using an event-driven API but handles reliability by writing all state to a replicated storage system like BigTable.

MapReduce Online [10] is a streaming Hadoop runtime that pushes records between maps and reduces and uses upstream backup for reliability. However, it cannot recover reduce tasks with long-lived state (the user must manually checkpoint such state into an external system), and does not handle stragglers. Meteor Shower [40] also uses upstream backup, and can take tens of seconds to recover state. iMR [24] offers a MapReduce API for log processing, but can lose data on failure. Percolator [31] runs incremental computations using triggers, but does not offer high-level operators like *map* and *join* or consistency guarantees across nodes.

Finally, to our knowledge, none of these systems support *combining* streaming with batch and interactive queries, like D-Streams do. Some streaming databases have supported combining tables and streams [14].

Message Queueing Systems Systems like Storm, S4, and Flume [36, 28, 3] offer a message passing model where users write stateful code to process records, but they generally have limited fault tolerance guarantees. For example, Storm ensures "at-least-once" delivery of *messages* using upstream backup at the source, but requires the user to manually handle the recovery of *state*, *e.g.,* by keeping all state in a replicated database [37]. Trident [25] provides a functional API similar to LINQ on top of Storm that manages state automatically. However, Trident does this by storing all state in a replicated database to provide fault tolerance, which is expensive.

Incremental Processing CBP [23] and Comet [17] provide "bulk incremental processing" on traditional MapReduce platforms by running MapReduce jobs on new data every few minutes. While these systems benefit from the scalability and fault/straggler tolerance of MapReduce *within* each timestep, they store all state in a replicated, on-disk filesystem *across* timesteps, incurring high overheads and latencies of tens of seconds to minutes. In contrast, D-Streams can keep state unreplicated in memory using RDDs and can recover it across timesteps using lineage, yielding order-of-magnitude lower latencies. Incoop [6] modifies Hadoop to support incremental recomputation of job outputs when an input file changes, and also includes a mechanism for straggler recovery, but it still uses replicated on-disk storage between timesteps, and does not offer an explicit streaming interface with concepts like windows.

Parallel Recovery Our parallel recovery mechanism is conceptually similar to techniques in MapReduce, GFS, and RAMCloud [11, 15, 29], which all leverage partitioning of recovery work on failure. Our contribution is to show how to structure a streaming computation to allow the use of this mechanism across data partitions and time, and to show that it can be implemented at a small enough timescale for stream processing.

9 Conclusion

We have proposed D-Streams, a new model for distributed streaming computation that enables fast (often sub-second) recovery from both faults and stragglers without the overhead of replication. D-Streams forgo conventional streaming wisdom by *batching* data into small timesteps. This enables powerful recovery mechanisms that exploit parallelism across data partitions and time. We showed that D-Streams can support a wide range of operators and can attain high per-node throughput, linear scaling to 100 nodes, sub-second latency, and sub-second fault recovery. Finally, because D-Streams use the same execution model as batch platforms, they compose seamlessly with batch and interactive queries. We used this capability in Spark Streaming to let users combine these models in powerful ways, and showed how it can add rich features to two real applications.

Spark Streaming is open source, and is now included in Spark at http://spark-project.org.

10 Acknowledgements

We thank the SOSP reviewers and our shepherd for their detailed feedback. This research was supported in part by NSF CISE Expeditions award CCF-1139158 and DARPA XData Award FA8750-12-2-0331, a Google PhD Fellowship, and gifts from Amazon Web Services, Google, SAP, Cisco, Clearstory Data, Cloudera, Ericsson, Facebook, FitWave, General Electric, Hortonworks, Huawei, Intel, Microsoft, NetApp, Oracle, Samsung, Splunk, VMware, WANdisco and Yahoo!.

References

[1] T. Akidau, A. Balikov, K. Bekiroglu, S. Chernyak, J. Haberman, R. Lax, S. McVeety, D. Mills, P. Nordstrom, and S. Whittle. MillWheel: Fault-

tolerant stream processing at internet scale. In *VLDB*, 2013.

[2] M. H. Ali, C. Gerea, B. S. Raman, B. Sezgin, T. Tarnavski, T. Verona, P. Wang, P. Zabback, A. Ananthanarayan, A. Kirilov, M. Lu, A. Raizman, R. Krishnan, R. Schindlauer, T. Grabs, S. Bjeletich, B. Chandramouli, J. Goldstein, S. Bhat, Y. Li, V. Di Nicola, X. Wang, D. Maier, S. Grell, O. Nano, and I. Santos. Microsoft CEP server and online behavioral targeting. *Proc. VLDB Endow.*, 2(2):1558, Aug. 2009.

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

[4] 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*.

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

[6] P. Bhatotia, A. Wieder, R. Rodrigues, U. A. Acar, and R. Pasquin. Incoop: MapReduce for incremental computations. In *SOCC '11*, 2011.

[7] D. Carney, U. Çetintemel, M. Cherniack, C. Convey, S. Lee, G. Seidman, M. Stonebraker, N. Tatbul, and S. Zdonik. Monitoring streams: a new class of data management applications. In *VLDB '02*, 2002.

[8] 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.

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

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

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

[12] EsperTech. Performance-related information. http://esper.codehaus.org/esper/performance/performance.html, Retrieved March 2013.

[13] EsperTech. Tutorial. http://esper.codehaus.org/tutorials/tutorial/tutorial.html, Retrieved March 2013.

[14] 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.

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

[16] J. Hammerbacher. Who is using flume in production? http://www.quora.com/Flume/Who-is-using-Flume-in-production/answer/Jeff-Hammerbacher.

[17] 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*, 2010.

[18] T. Hunter, T. Moldovan, M. Zaharia, S. Merzgui, J. Ma, M. J. Franklin, P. Abbeel, and A. M. Bayen. Scaling the Mobile Millennium system in the cloud. In *SOCC '11*, 2011.

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

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

[21] M. Isard, M. Budiu, Y. Yu, A. Birrell, and D. Fetterly. Dryad: distributed data-parallel programs from sequential building blocks. In *EuroSys 07*, 2007.

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

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

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

[25] N. Marz. Trident: a high-level abstraction for realtime computation. http://engineering.twitter.com/2012/08/trident-high-level-abstraction-for.html.

[26] F. McSherry, D. G. Murray, R. Isaacs, and M. Isard. Differential dataflow. In *Conference on Innovative Data Systems Research (CIDR)*, 2013.

[27] D. Murray, F. McSherry, R. Isaacs, M. Isard, P. Barham, and M. Abadi. Naiad: A timely dataflow system. In *SOSP '13*, 2013.

[28] L. Neumeyer, B. Robbins, A. Nair, and A. Kesari. S4: Distributed stream computing platform. In *Intl. Workshop on Knowledge Discovery Using Cloud and Distributed Computing Platforms (KDCloud)*, 2010.

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

[30] Oracle. Oracle complex event processing performance. http://www.oracle.com/technetwork/middleware/complex-event-processing/overview/cepperformancewhitepaper-128060.pdf, 2008.

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

[32] 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 *EuroSys '13*, 2013.

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

[34] Z. Shao. Real-time analytics at Facebook. XLDB 2011, http://www-conf.slac.stanford.edu/xldb2011/talks/xldb2011_tue_0940_facebookrealtimeanalytics.pdf.

[35] U. Srivastava and J. Widom. Flexible time management in data stream systems. In *PODS*, 2004.

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

[37] Guaranteed message processing (Storm wiki). https://github.com/nathanmarz/storm/wiki/Guaranteeing-message-processing.

[38] K. Thomas, C. Grier, J. Ma, V. Paxson, and D. Song. Design and evaluation of a real-time URL spam filtering service. In *IEEE Symposium on Security and Privacy*, 2011.

[39] R. Tibbetts. Streambase performance & scalability characterization. http://www.streambase.com/wp-content/uploads/downloads/StreamBase_White_Paper_Performance_and_Scalability_Characterization.pdf, 2009.

[40] H. Wang, L.-S. Peh, E. Koukoumidis, S. Tao, and M. C. Chan. Meteor shower: A reliable stream processing system for commodity data centers. In *IPDPS '12*, 2012.

[41] 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.

[42] 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.