Summary of "Discretized Streams: Fault-Tolerant Streaming Computation at Scale"

Pei-Xuan Xie (peixuanx), Tai-Ying Lin (taiyingl), Changfeng Liu (changfel)

Problem and Motivation

Nowadays, running many "big data" applications in real-time requires parallel platforms that automatically handle faults and straggler problems. These two problems are inevitable in large clusters, thus it is important to perform fast recovery from them. However, most of existing systems, including Storm, TimeStream, MapReduce Online, and streaming databases are based on the continuous operator model. These systems provide fault recovery costly and do not handle stragglers since they perform recovery through two approaches: replication (costs 2X in the hardware) and upstreaming backup (takes long time to recovery).

Hypothesis

In the paper, Discrete Streams (D-Streams), a new stream processing model is proposed to enable fast (often sub-second) recovery from both faults and stragglers without the overhead of replication. D-Streams enables a parallel recovery mechanism so that it improves the efficiency over traditional replication and backup schemes, and tolerates stragglers. Furthermore, D-Streams can seamlessly be composed with batch and interactive computation since it use the same execution model and data structures (RDDs) as batch platforms. Noted that the applications with latency needed below a few hundred milliseconds are not targeted in this paper.

Solution Overview

Problem1:

In continuous operator models, systems perform recovery through two approaches:
Replication: Get two copies of each node
Upstream backup: Nodes buffer sent messages and replay them to a new copy of a failed node.
However, these two approaches doesn't work well in large clusters: replication costs 2× the hardware, while upstream backup takes a long time to recover, because the whole system must wait for a new node to serially rebuild the failed node's state.

Solution to Problem1:

1. Parallel Recovery
   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.

- 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.
- Lost datasets on different timesteps can be recomputed in parallel.

- In D-Streams, all of the machines partake in recovery, while processing new records.

2. Master Recovery
   The tolerate failures of Spark's master is a requirement for Spark Streaming, and the following features help to simplify the recovery:

- It is acceptable for D-streams if a given RDD is computed twice as the operations are deterministic.
- Master recovery is ensured by:
    - Writing the state of the computation reliably when starting each timestep
    - Having workers connect to the new Master and report their RDDs

Problem2: Straggler Handling

Mitigate stragglers is difficult in traditional systems for it requires launching a new copy of the node, populating its state, overtaking the slow copy and replication algorithms require synchronization.

Solution to Problem2 : Straggler Mitigation

D-Streams help us mitigate stragglers like batch systems do, by running speculative backup copies of slow tasks.

Problem3

Unification with Batch and Interactive Processing is uneasy for continuous operator model.

Solution to Problem3

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

- Seamless combination with batch systems
- New streaming report can be generated using old data in "batch mode"
- Users can run ad-hoc queries interactively on D-Streams

Limitations and Possible Improvements

1. Minimum Latency: D-Streams have a fixed minimum latency due to batching data. However, the total delay of D-Streams can still be as low as 1–2 seconds, which is enough for many practical cases.
2. Expressiveness: With the abstraction of an execution strategy, D-Streams should be able to handle most streaming algorithms. For example, it might be doable and beneficial to use languages like streaming SQL or Complex Event Processing models over them.
3. Setting the batch interval: Batch interval is an essential element in the trade-off of end-to-end latency and throughput. However, in current D-Streams, developers have to tune this parameter by themselves which can be tedious and inefficient. It will be welcomed if D-Streams can determine the batch interval automatically.
4. Memory usage: D-Streams generate a new state RDD after processing each batch of data. For the lineage-based fault recovery, D-Streams have to store different

versions of these state RDDs, which cause large memory usage. It might be improved by only storing the deltas between state RDDs instead.

5. Approximate results: Besides recomputing lost work, it might also be possible to provide approximate results when encountering a failure. That is to say, a child node can launch a task and compute partial results before all of its parents are ready. The information of missing parent nodes can be included in the lineage data.

Comparison to "Storm @Twitter"

Feature	Spark Streaming	Storm
Processing Model	D-Streams	Continuous Operator Model
Fault Tolerance	Parallel Recovery Straggler Mitigation	Upstream Backup
State Management	Using Resilient Distributed Datasets (RDDs) to keep data in memory.	Nimbus uses a combination of the local disk(s) and Zookeeper to store state about the topology.

Summary of Class Discussion

1. Why don't the previous model handle straggler? Would they encounter the straggler problem?
   They do encounter straggler problem. For upstream backup, a straggler must be treated as a failure, incurring a costly recovery step. For replicated systems, synchronization protocols are used to coordinate replicas causing a straggler to slow down both replicas.
2. The zookeeper would maintain the cluster state. Thus, if the master node fails and another node needs to be updated as the new master, the zookeeper would take care of it, so there will always be single master node.
3. Where does those data be duplicated soon after they arrive?
   Those data will be assigned to multiples workers by master, and then the master will assign task related to those data to one of those workers.
4. The trade-off between low latency and high through-put is common among processing systems. By batching input data before processing, we can get higher throughput but also higher latency. Conversely, if we processed data by tuples, we can get lower latency yet also get lower throughput.
5. Stragglers are much more common than failure. Say a failure might happen once a month or once a year and sometimes it's easy to simply restart the systems. However there are always many nodes slower than others, which can be the bottleneck of the performance of whole system. This is why we should pay attention to straggler problem.