SparkStreaming

Large scale near-realtime stream processing

Tathagata Das (TD)
UC Berkeley
Motivation

• Many important applications must process large data streams at second-scale latencies
  – Site statistics, intrusion detection, spam filtering, ...

• Scaling these apps to 100s of nodes, require ...
  – Fault-tolerance: for both crashes and stragglers
  – Efficiency: for being cost-effective

• Also would like to have ...
  – Simple programming model
  – Integration with batch + ad hoc queries

Current streaming frameworks don’t meet both goals together
Traditional Streaming Systems

- **Record-at-a-time** processing model
  - Each node has mutable state
  - For each record, update state & send new records

![Diagram showing a network of nodes connected by arrows representing input records and mutable state. The nodes are labeled as node 1, node 2, node 3, and arrows represent the flow of input records and the push action. The mutable state is indicated with a blue box at each node.]
Traditional Streaming Systems

Fault tolerance via *replication* or *upstream backup*

- **Replication**
  - Fast recovery, but 2x hardware cost

- **Upstream backup**
  - Only need 1 standby, but slow to recover
Traditional Streaming Systems

Fault tolerance via \textit{replication} or \textit{upstream backup}

\textbf{Replication}

\textbf{Upstream backup}

Neither handle stragglers
Observation

• Batch processing models, like MapReduce, do provide fault tolerance efficiently
  – Divide job into deterministic tasks
  – Rerun failed/slow tasks in parallel on other nodes
Idea

• Idea: run a streaming computation as a series of very small, deterministic batch jobs
  – Eg. Process stream of tweets in 1 sec batches
  – Same recovery schemes at smaller timescale

• Try to make batch size as small as possible
  – Lower batch size $\rightarrow$ lower end-to-end latency

• State between batches kept in memory
  – Deterministic stateful ops $\rightarrow$ fault-tolerance
Discretized Stream Processing

input

input stream

batch operations

input

state stream

time = 0 - 1:

time = 1 - 2:

immutable dataset
(stored reliably)

immutable dataset
(output or state);
stored in memory
as Spark RDD
Fault Recovery

- All dataset modeled as RDDs with dependency graph → fault-tolerant with full replication
- Fault/straggler recovery is done in parallel on other nodes → fast recovery

Fast recovery without the cost of full replication
How Fast Can It Go?

- Prototype can process 4 GB/s (40M records/s) of data on 100 nodes at sub-second latency.

Max throughput with a given latency bound (1 or 2s)
Comparison with Storm

- Storm limited to 10,000 records/s/node
- Also tried Apache S4: 7000 records/s/node
- Commercial systems report O(100K)
How Fast Can It Recover?

- Recovers from faults/stragglers within 1 sec

![Graph showing Interval Processing Time (s) vs. Time (s) for Sliding WordCount on 10 nodes with 30s checkpoint interval.](image-url)
Programming Interface

• A Discretized Stream or **DStream** is a sequence of RDDs
  – Represents a stream of data
  – API *very similar* to RDDs

• DStreams can be created...
  – Either from live streaming data
  – Or by transforming other DStreams
DStream Operators

• **Transformations**
  – Build new streams from existing streams
  – Existing RDD ops (map, etc) + new “stateful” ops

• **Output operators**
  – Send data to outside world (save results to external storage, print to screen, etc)
Example 1

Count the words received every second

\[
\text{words} = \text{createNetworkStream("http://...")}
\]

\[
\text{counts} = \text{words.count()}
\]

DStreams

transformation

\[\begin{align*}
\text{time} &= 0 - 1: \\
\text{time} &= 1 - 2: \\
\text{time} &= 2 - 3: \\
\end{align*}\]
Example 2

Count frequency of words received every second

words = createNetworkStream("http://...")
ones = words.map(w => (w, 1))
freqs = ones.reduceByKey(_ + _)

Scala function literal

```
words = createNetworkStream("http://...")
ones = words.map(w => (w, 1))
freqs = ones.reduceByKey(_ + _)
```

Diagram:

- **time = 0 - 1:**
  - words -> ones (map)
  - ones -> freqs (reduce)

- **time = 1 - 2:**
  - words -> ones (map)
  - ones -> freqs (reduce)

- **time = 2 - 3:**
  - words -> ones (map)
  - ones -> freqs (reduce)
Example 2: Full code

```scala
// Create the context and set the batch size
val ssc = new SparkStreamContext("local", "test")
ssc.setBatchDuration(Seconds(1))

// Process a network stream
val words = ssc.createNetworkStream("http://...")
val ones = words.map(w => (w, 1))
val freqs = ones.reduceByKey(_ + _)
freqs.print()

// Start the stream computation
ssc.run
```
Example 3

Count frequency of words received in last minute

```javascript
ones = words.map(w => (w, 1))
freqs = ones.reduceByKeys((a, b) => a + b)
freqs_60s = freqs.window(Seconds(60), Second(1))
```

![Diagram showing the processing steps: map, reduce, window, and reduce for time intervals 0-1, 1-2, and 2-3. The diagram highlights the sliding window operator, window length, and window movement.](image)

- **sliding window operator**
- **window length**
- **window movement**
Simpler window-based reduce

freqs = ones.reduceByKey(_ + _)

freqs_60s = freqs.window(Seconds(60), Second(1))
  .reduceByKey(_ + _)

freqs = ones.reduceByKeyAndWindow(_ + _, Seconds(60), Seconds(1))
“Incremental” window operators

<table>
<thead>
<tr>
<th></th>
<th>words</th>
<th>freqs</th>
<th>freqs_60s</th>
</tr>
</thead>
<tbody>
<tr>
<td>t-1</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>t</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>t+1</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>t+2</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>t+3</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>t+4</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Aggregation function

<table>
<thead>
<tr>
<th></th>
<th>words</th>
<th>freqs</th>
<th>freqs_60s</th>
</tr>
</thead>
<tbody>
<tr>
<td>t-1</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>t</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>t+1</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>t+2</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>t+3</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>t+4</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Invertible aggregation function
**Smarter window-based reduce**

freqs = ones.reduceByKey(_ + _)

freqs_60s = freqs.window(Seconds(60), Second(1)).reduceByKey(_ + _)

freqs = ones.reduceByKeyAndWindow(_ + _, Seconds(60), Seconds(1))

freqs = ones.reduceByKeyAndWindow(_ + _, _ - _, Seconds(60), Seconds(1))
Output Operators

• *save*: write results to any Hadoop-compatible storage system (e.g. HDFS, HBase)

```scala
freqs.save(“hdfs://...”)
```

• *foreachRDD*: run a Spark function on each RDD

```scala
freqs.foreachRDD(freqsRDD => {
  // any Spark/Scala processing, maybe save to database
})
```
Live + Batch + Interactive

- Combining DStreams with historical datasets
  
  ```scala
  freqs.join(oldFreqs).map("")
  ```

- Interactive queries on stream state from the Spark interpreter
  
  ```scala
  freqs.slice("21:00", "21:05").topK(10)
  ```
One stack to rule them all

• The promise of a unified data analytics stack
  – Write algorithms only once
  – Cuts complexity of maintaining separate stacks for live and batch processing
  – Query live stream state instead of waiting for import

• Feedback very exciting

• Some recent experiences ...
Implementation on Spark

• Optimizations on current Spark
  – Optimized scheduling for < 100ms tasks
  – New block store with fast NIO communication
  – Pipelining of jobs from different time intervals

• Changes already in dev branch of Spark on http://www.github.com/mesos/spark

• An alpha will be released with Spark 0.6 soon
More Details

• You can find more about SparkStreaming in our paper: http://tinyurl.com/dstreams

Thank you!
Fault Recovery

Failures:

<table>
<thead>
<tr>
<th>Interval Processing Time (s)</th>
<th>WordCount, 30s checkpoints</th>
</tr>
</thead>
<tbody>
<tr>
<td>Before Failure</td>
<td>1.47</td>
</tr>
<tr>
<td>At Time of Failure</td>
<td>2.31</td>
</tr>
</tbody>
</table>

Stragglers:

<table>
<thead>
<tr>
<th>Interval Processing Time (s)</th>
<th>WordCount</th>
<th>Grep</th>
</tr>
</thead>
<tbody>
<tr>
<td>No straggler</td>
<td>0.79</td>
<td>0.66</td>
</tr>
<tr>
<td>Straggler, with speculation</td>
<td>1.09</td>
<td>1.08</td>
</tr>
</tbody>
</table>
Interactive Ad-Hoc Queries

![Graph showing response time of different queries.]

- Grep on raw sentences: 0.5s
- Lookup in word count RDD: 0.30s
- Top K on word count RDD: 0.28s
- Join with historical counts: 0.51s
Related Work

• Bulk incremental processing (CBP, Comet)
  – Periodic (~5 min) batch jobs on Hadoop/Dryad
  – On-disk, replicated FS for storage instead of RDDs

• Hadoop Online
  – Does not recover stateful ops or allow multi-stage jobs

• Streaming databases
  – Record-at-a-time processing, generally replication for FT

• Approximate query processing, load shedding
  – Do not support the loss of arbitrary nodes
  – Different math because drop rate is known exactly

• Parallel recovery (MapReduce, GFS, RAMCloud, etc)
Timing Considerations

• D-streams group input into intervals based on when records arrive at the system

• For apps that need to group by an “external” time and tolerate network delays, support:
  – **Slack time**: delay starting a batch for a short fixed time to give records a chance to arrive
  – **Application-level correction**: e.g. give a result for time t at time t+1, then use later records to update incrementally at time t+5