Spark Streaming
Real-time big-data processing

Tathagata Das (TD)
What is Spark Streaming?

- Extends Spark for doing big data stream processing
- Project started in early 2012, alpha released in Spring 2013 with Spark 0.7
- Moving out of alpha in Spark 0.9
Why Spark Streaming?

Many big-data applications need to process large data streams in realtime

Website monitoring

Fraud detection

Ad monetization
Why Spark Streaming?

Need a framework for big data stream processing that

- Scales to hundreds of nodes
- Achieves second-scale latencies
- Efficiently recover from failures
- Integrates with batch and interactive processing
Integration with Batch Processing

- Many environments require processing same data in live streaming as well as batch post-processing

- Existing frameworks cannot do both
  - Either, stream processing of 100s of MB/s with low latency
  - Or, batch processing of TBs of data with high latency

- Extremely painful to maintain two different stacks
  - Different programming models
  - Double implementation effort
Stateful Stream Processing

- Traditional model
  - Processing pipeline of nodes
  - Each node maintains mutable state
  - Each input record updates the state and new records are sent out

- Mutable state is lost if node fails

- Making stateful stream processing fault tolerant is challenging!
Existing Streaming Systems

- **Storm**
  - Replays record if not processed by a node
  - Processes each record *at least once*
  - May update mutable state twice!
  - Mutable state can be lost due to failure!

- **Trident** – Use transactions to update state
  - Processes each record *exactly once*
  - Per-state transaction to external database is slow
Spark Streaming
Spark Streaming

Run a streaming computation as a series of very small, deterministic batch jobs

- Chop up the live stream into batches of X seconds
- Spark treats each batch of data as RDDs and processes them using RDD operations
- Finally, the processed results of the RDD operations are returned in batches
Spark Streaming

Run a streaming computation as a series of very small, deterministic batch jobs

- Batch sizes as low as $\frac{1}{2}$ second, latency of about 1 second
- Potential for combining batch processing and streaming processing in the same system
Example – Get hashtags from Twitter

```scala
val tweets = ssc.twitterStream()
```

**DStream**: a sequence of RDDs representing a stream of data

Twitter Streaming API

tweets DStream

stored in memory as an RDD (immutable, distributed)
Example – Get hashtags from Twitter

val tweets = ssc.twitterStream()
val hashTags = tweets.flatMap(status => getTags(status))

new DStream

**transformation**: modify data in one DStream to create another DStream

tweets DStream

hashTags Dstream

[new RDDs created for every batch]
Example – Get hashtags from Twitter

val tweets = ssc.twitterStream()
val hashTags = tweets.flatMap(status => getTags(status))
hashTags.saveAsHadoopFiles("hdfs://...")

**output operation**: to push data to external storage

tweets DStream

hashTags DStream

每一批次保存到HDFS
Example – Get hashtags from Twitter

```scala
val tweets = ssc.twitterStreamStream()
val hashTags = tweets.flatMap(status => getTags(status))
hashTags.foreach(hashTagRDD => {
  ...
})

foreach: do whatever you want with the processed data
```

tweets DStream

- batch @ t
  - flatMap
  - foreach

hashTags DStream

- batch @ t+1
  - flatMap
  - foreach

- batch @ t+2
  - flatMap
  - foreach

Write to a database, update analytics
UI, do whatever you want
Demo
Java Example

Scala

```scala
val tweets = ssc.twitterStream()
val hashTags = tweets.flatMap(status => getTags(status))
hashTags.saveAsHadoopFiles("hdfs://...")
```

Java

```java
JavaDStream<Status> tweets = ssc.twitterStream()
JavaDstream<String> hashTags = tweets.flatMap(new Function<>(){
    
})
hashTags.saveAsHadoopFiles("hdfs://...")
```
Window-based Transformations

val tweets = ssc.twitterStream()
val hashTags = tweets.flatMap(status => getTags(status))
val tagCounts = hashTags.window(Minutes(1), Seconds(5)).countByValue()
Arbitrary Stateful Computations

Specify function to generate new state based on previous state and new data

- Example: Maintain per-user mood as state, and update it with their tweets

```python
def updateMood(newTweets, lastMood) => newMood

moods = tweetsByUser.updateStateByKey(updateMood _)
```
Arbitrary Combinations of Batch and Streaming Computations

Inter-mix RDD and DStream operations!
- Example: Join incoming tweets with a spam HDFS file to filter out bad tweets

```scala
tweets.transform(tweetsRDD => {
    tweetsRDD.join(spamHDFSFile).filter("
})
```
DStreams + RDDs = Power

- Online machine learning
  - Continuously learn and update data models (*updateStateByKey* and *transform*)

- Combine live data streams with historical data
  - Generate historical data models with Spark, etc.
  - Use data models to process live data stream (*transform*)

- CEP-style processing
  - window-based operations (*reduceByWindow*, etc.)
Input Sources

- Out of the box, we provide
  - Kafka, HDFS, Flume, Akka Actors, Raw TCP sockets, etc.

- Very easy to write a receiver for your own data source

- Also, generate your own RDDs from Spark, etc. and push them in as a “stream”
Fault-tolerance

- Batches of input data are replicated in memory for fault-tolerance

- Data lost due to worker failure, can be recomputed from replicated input data

- All transformations are fault-tolerant, and exactly-once transformations
Performance

Can process **60M records/sec (6 GB/sec)** on **100 nodes** at **sub-second latency**

<table>
<thead>
<tr>
<th>Cluster</th>
<th>Throughput (GB/s)</th>
</tr>
</thead>
<tbody>
<tr>
<td># Nodes</td>
<td>in Cluster</td>
</tr>
<tr>
<td>1 sec</td>
<td></td>
</tr>
<tr>
<td>2 sec</td>
<td></td>
</tr>
</tbody>
</table>

**Grep**

![Cluster Throughput (GB/s) vs # Nodes in Cluster for Grep]

<table>
<thead>
<tr>
<th># Nodes in Cluster</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
</tr>
<tr>
<td>50</td>
</tr>
<tr>
<td>100</td>
</tr>
</tbody>
</table>

**WordCount**

![Cluster Throughput (GB/s) vs # Nodes in Cluster for WordCount]

<table>
<thead>
<tr>
<th># Nodes in Cluster</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
</tr>
<tr>
<td>50</td>
</tr>
<tr>
<td>100</td>
</tr>
</tbody>
</table>
Comparison with other systems

Higher throughput than Storm
- Spark Streaming: 670k records/sec/node
- Storm: 115k records/sec/node
- Commercial systems: 100-500k records/sec/node
Fast Fault Recovery

Recovers from faults/stragglers within 1 sec

Sliding WordCount on 10 nodes with 30s checkpoint interval
Mobile Millennium Project

Traffic transit time estimation using online machine learning on GPS observations

- Markov-chain Monte Carlo simulations on GPS observations
- Very CPU intensive, requires dozens of machines for useful computation
- Scales linearly with cluster size

![Graph showing GPS observations per second vs. number of nodes in cluster.](image)
Advantage of an unified stack

- Explore data interactively to identify problems
- Use same code in Spark for processing large logs
- Use similar code in Spark Streaming for realtime processing

```scala
$ ./spark-shell
scala> val file = sc.hadoopFile("smallLogs")
...
scala> val filtered = file.filter(_.contains("ERROR"))
...
scala> val mapped = filtered.map(....

object ProcessProductionData {
  def main(args: Array[String]) {
    val sc = new SparkContext(...)
    val file = sc.hadoopFile("productionLogs")
    val filtered = file.filter(_.contains("ERROR"))
    val mapped = filtered.map(....
    ...
  }
}

object ProcessLiveStream {
  def main(args: Array[String]) {
    val sc = new StreamingContext(...)
    val stream = sc.kafkaStream(...)
    val filtered = stream.filter(_.contains("ERROR"))
    val mapped = filtered.map(....
    ...
  }
}
```
Roadmap

- **Spark 0.8.1**
  - Marked alpha, but has been quite stable
  - Master fault tolerance – manual recovery
    - Restart computation from a checkpoint file saved to HDFS

- **Spark 0.9 in Jan 2014 – out of alpha!**
  - Automated master fault recovery
  - Performance optimizations
  - Web UI, and better monitoring capabilities
Roadmap

- Long term goals
  - Python API
  - MLlib for Spark Streaming
  - Shark Streaming

- Community feedback is crucial!
  - Helps us prioritize the goals

- Contributions are more than welcome!!
Today’s Tutorial

- Process Twitter data stream to find most popular hashtags over a window

- Requires a Twitter account
  - Need to setup Twitter OAuth keys to access tweets
  - All the instructions are in the tutorial

- Your account will be safe!
  - No need to enter your password anywhere, only the keys
  - Destroy the keys after the tutorial is done
Conclusion

- Streaming programming guide – spark.incubator.apache.org/docs/latest/streaming-programming-guide.html

- Research Paper – tinyurl.com/dstreams