Spark Streaming
Large-scale near-real-time stream processing

Tathagata Das (TD)
UC Berkeley
What is Spark Streaming?

- Framework for large scale stream processing
  - Scales to 100s of nodes
  - Can achieve second scale latencies
  - Integrates with Spark’s batch and interactive processing
  - Provides a simple batch-like API for implementing complex algorithm
  - Can absorb live data streams from Kafka, Flume, ZeroMQ, etc.
Motivation

- Many important applications must process large streams of live data and provide results in near-real-time
  - Social network trends
  - Website statistics
  - Intrusion detection systems
  - etc.

- Require large clusters to handle workloads
- Require latencies of few seconds
Need for a framework ...

... for building such complex stream processing applications

But what are the requirements from such a framework?
Requirements

- **Scalable** to large clusters
- **Second-scale** latencies
- **Simple** programming model
Case study: Conviva, Inc.

- Real-time monitoring of online video metadata
  - HBO, ESPN, ABC, SyFy, ...

- Two processing stacks
  - Custom-built distributed stream processing system
    - 1000s complex metrics on millions of video sessions
    - Requires many dozens of nodes for processing
  - Hadoop backend for offline analysis
    - Generating daily and monthly reports
    - Similar computation as the streaming system
Case study: XYZ, Inc.

- Any company who wants to process live streaming data has this problem
- *Twice* the effort to implement any new function
- *Twice* the number of bugs to solve
- *Twice* the headache

- Two processing stacks

Custom-built distributed stream processing system:
- 1000s of complex metrics on millions of videos
- Requires many dozens of nodes for processing

Hadoop backend for offline analysis:
- Generate daily and monthly reports
- Similar computation as the streaming system

Any company who wants to process live streaming data has this problem:
- Twice the effort to implement any new function
- Twice the number of bugs to solve
- Twice the headache

Two processing stacks

Custom-built distributed stream processing system:
- 1000s of complex metrics on millions of videos
- Requires many dozens of nodes for processing

Hadoop backend for offline analysis:
- Generate daily and monthly reports
- Similar computation as the streaming system
Requirements

- **Scalable** to large clusters
- **Second-scale** latencies
- **Simple** programming model
- **Integrated** with batch & interactive processing
Stateful Stream Processing

- Traditional streaming systems have a event-driven record-at-a-time processing model
  - Each node has mutable state
  - For each record, update state & send new records

- State is lost if node dies!

- Making stateful stream processing be 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 updates slow
Requirements

- **Scalable** to large clusters
- **Second-scale** latencies
- **Simple** programming model
- **Integrated** with batch & interactive processing
- **Efficient fault-tolerance** in stateful computations
Spark Streaming
Discretized Stream Processing

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
Discretized Stream Processing

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

- Batch sizes as low as ½ second, latency ~ 1 second
- Potential for combining batch processing and streaming processing in the same system

---

![Diagram](image-url)
Example 1 – Get hashtags from Twitter

```scala
val tweets = ssc.twitterStream(<Twitter username>, <Twitter password>)
```

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

Twitter Streaming API

tweets DStream

stored in memory as an RDD (immutable, distributed)
Example 1 – Get hashtags from Twitter

```scala
val tweets = ssc.twitterStream(<Twitter username>, <Twitter password>)
val hashTags = tweets.flatMap (status => getTags(status))
```

**transformation**: modify data in one Dstream to create another DStream.
Example 1 – Get hashtags from Twitter

val tweets = ssc.twitterStream(<Twitter username>, <Twitter password>)
val hashTags = tweets.flatMap (status => getTags(status))
hashTags.saveAsHadoopFiles("hdfs://...")

output operation: to push data to external storage

tweets DStream

hashTags DStream

every batch saved to HDFS
Java Example

Scala
val tweets = ssc.twitterStream(<Twitter username>, <Twitter password>)
val hashTags = tweets.flatMap(status => getTags(status))
hashTags.saveAsHadoopFiles("hdfs://...")

Java
JavaDStream<Status> tweets = ssc.twitterStream(<Twitter username>, <Twitter password>)
JavaDstream<String> hashTags = tweets.flatMap(new Function<...> { })
hashTags.saveAsHadoopFiles("hdfs://...")

Function object to define the transformation
Fault-tolerance

- RDDs are remember the sequence of operations that created it from the original fault-tolerant input data

- Batches of input data are replicated in memory of multiple worker nodes, therefore fault-tolerant

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

- **DStream** – sequence of RDDs representing a stream of data
  - Twitter, HDFS, Kafka, Flume, ZeroMQ, Akka Actor, TCP sockets

- **Transformations** – modify data from one DStream to another
  - Standard RDD operations – map, countByValue, reduce, join, ... 
  - Stateful operations – window, countByValueAndWindow, ...

- **Output Operations** – send data to external entity
  - saveAsHadoopFiles – saves to HDFS
  - foreach – do anything with each batch of results
Example 2 – Count the hashtags

```scala
val tweets = ssc.twitterStream(<Twitter username>, <Twitter password>)
val hashTags = tweets.flatMap (status => getTags(status))
val tagCounts = hashTags.countByKeyValue()
```

![Diagram showing the flow of data from tweets to tagCounts](image)
Example 3 – Count the hashtags over last 10 mins

val tweets = ssc.twitterStream(<Twitter username>, <Twitter password>)
val hashTags = tweets.flatMap (status => getTags(status))
val tagCounts = hashTags.window(Minutes(10), Seconds(1)).countByValue()
Example 3 – Counting the hashtags over last 10 mins

val tagCounts = hashTags.window(Minutes(10), Seconds(1)).countByValue()
val tagCounts = hashtags.countByValueAndWindow(Minutes(10), Seconds(1))
Smart window-based *reduce*

- Technique to incrementally compute count generalizes to many reduce operations
  - Need a function to “inverse reduce” (“subtract” for counting)

- Could have implemented counting as:
  ```scala
  hashTags.reduceByKeyAndWindow(_ + _, _ - _, Minutes(1), ...)
  ```
Demo
Fault-tolerant Stateful Processing

All intermediate data are RDDs, hence can be recomputed if lost.

---

tagCounts

---

hashTags
Fault-tolerant Stateful Processing

- State data not lost even if a worker node dies
  - Does not change the value of your result

- Exactly once semantics to all transformations
  - No double counting!
Other Interesting Operations

- Maintaining arbitrary state, track sessions
  - Maintain per-user mood as state, and update it with his/her tweets
    
    ```
    tweets.updateStateByKey(tweet => updateMood(tweet))
    ```

- Do arbitrary Spark RDD computation within DStream
  - Join incoming tweets with a spam file to filter out bad tweets
    
    ```
    tweets.transform(tweetsRDD => {
      tweetsRDD.join(spamHDFSFile).filter(...)  
    })
    ```
Performance

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

- Tested with 100 streams of data on 100 EC2 instances with 4 cores each
Comparison with Storm and S4

Higher throughput than Storm

- Spark Streaming: \textbf{670k} records/second/node
- Storm: \textbf{115k} records/second/node
- Apache S4: 7.5k records/second/node
Fast Fault Recovery

Recovers from faults/stragglers within 1 sec

Sliding WordCount on 10 nodes with 30s checkpoint interval
Real Applications: Conviva

Real-time monitoring of video metadata

- Achieved 1-2 second latency
- Millions of video sessions processed
- Scales linearly with cluster size

![Graph showing active sessions vs. number of nodes in cluster]
Real Applications: 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
Vision - *one stack to rule them all*

Stream Processing

**Spark**
+ **Shark**
+ **Spark Streaming**

Batch Processing

Ad-hoc Queries

---

*Ad-hoc Queries*  
*Batch Processing*  
*Spark Streaming*  
*Spark*  
*Shark*
Spark program vs Spark Streaming program

Spark Streaming program on Twitter stream

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

Spark program on Twitter log file

```scala
val tweets = sc.hadoopFile("hdfs://...")
val hashTags = tweets.flatMap (status => getTags(status))
hashTags.saveAsHadoopFile("hdfs://...")
```
Vision - one stack to rule them all

- Explore data interactively using Spark Shell / PySpark to identify problems
- Use same code in Spark stand-alone programs to identify problems in production logs
- Use similar code in Spark Streaming to identify problems in live log streams

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

```scala
object ProcessLiveStream {
  def main(args: Array[String]) {
    val sc = new StreamingContext(...) 
    val stream = sc.kafkaStream(...) 
    val filtered = file.filter(_.contains("ERROR"))
    val mapped = file.map(...) 
  }
}
```
Explore data interactively using Spark Shell / PySpark to identify problems

Use same code in Spark stand-alone programs to identify problems in production logs

Use similar code in Spark Streaming to identify problems in live log streams

Vision - *one stack to rule them all*

- Stream Processing
- Spark + Shark
- Spark Streaming
- Batch Processing
- Ad-hoc Queries

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

```
object ProcessLiveStream {
  def main(args: Array[String]) {
    val sc = new StreamingContext(...)
    val stream = sc.kafkaStream(...)
    val filtered = file.filter(_.contains("ERROR"))
    val mapped = file.map(...)
  }
}
```
Alpha Release with Spark 0.7

- Integrated with Spark 0.7
  - Import `spark.streaming` to get all the functionality

- Both Java and Scala API

- Give it a spin!
  - Run locally or in a cluster

- Try it out in the hands-on tutorial later today
Summary

- Stream processing framework that is ...
  - Scalable to large clusters
  - Achieves second-scale latencies
  - Has simple programming model
  - Integrates with batch & interactive workloads
  - Ensures efficient fault-tolerance in stateful computations

- For more information, checkout our paper: [http://tinyurl.com/dstreams](http://tinyurl.com/dstreams)