Outline

• Introduction
• DStreams
• Thinking about time
• Recovery and Fault tolerance
• Conclusion
About Me

Andrew Psaltis
Data Engineer @ Shutterstock

Fun outside of Shutterstock:
• Sometimes ramble here: @itmdata
• Author of Streaming Data
• Dreaming about streaming since 2008
• Conference Speaker
• Content provider for SkillSoft
• Lacrosse crazed
Why Streaming?

“Without stream processing there’s no big data and no Internet of Things” – Dana Sandu, SQLstream

• **Operational Efficiency** - 1 extra mph for a locomotive on its daily route can lead to $200M in saving (Norfolk Southern)

• **Tracking Behavior** - McDonalds (Netherlands) realized a 700% increase in offer redemptions using personalized advertising based on location, weather, previous purchase, and preference.

• **Predict machine failure** - GE monitors over 5500 assets from 70+ customer sites globally. Can predict failure and determine when something needs maintenance

• **Improving Traffic Safety and Efficiency** – According to EU Commission congestion in EU urban areas costs ~ €100 billion or 1 percent of EU GDP annually
What is Spark Streaming?

- Provides efficient, fault-tolerant stateful stream processing
- Provides a simple API for implementing complex algorithms
- Integrates with Spark’s batch and interactive processing
- Integrates with other Spark extensions
High-level Architecture

Handles scheduling the jobs to run on the workers

Your program
Contains Spark Streaming Context (Spark Client)

Where your algorithm runs

The streaming data source (Twitter, IoT, Network, File,..) and output store
Discretized Streams (DStreams)

- The basic abstraction provided by Spark Streaming
- Continuous series of RDDs
DStreams

• 3 Things we want to do
  • Ingest
  • Transform
  • Output
Input DStreams (Ingestion)

There are 3 ways to get data in:

- Basic sources
- Advanced sources
- Custom Sources
Basic Input DStreams

- Basic sources
  - Built-in (file system, socket, Akka actors)
  - Non-built in (Avro, CSV, ...)
  - Not reliable
Advanced Input DStreams

• Advanced sources
  • Twitter, Kafka, Flume, Kinesis, MQTT, ….
• Require external library
• Maybe reliable or unreliable
Custom Input DStreams

- Implement two classes
  - Input DStream
  - Receiver
Custom Input DStream

```scala
class CustomInputDStream(  @transient ssc_: StreamingContext,  storageLevel: StorageLevel) extends ReceiverInputDStream[String](ssc_) {

  def getReceiver(): Receiver[String] = {
    new CustomReceiver(storageLevel)
  }
}
```
Custom Receiver

Start threads, open sockets, etc.
MUST BE non-blocking

class CustomReceiver(storageLevel: StorageLevel)
    extends Receiver[String](storageLevel){

    def onStart() {
    }

    def onStop() {
    }

    //Defined in Receiver class
    def store(...) {
    }

    Cleanup everything started in onStart. Stops receiving data

    Call store (item, buffer, iterator)
Receiver Reliability

Two types of receivers

• Unreliable Receiver
• Reliable Receiver
Receiver Reliability

Unreliable Receiver

- Simple to implement
- No fault-tolerance
- Data loss when receiver fails
Receiver Reliability

Reliable Receiver

- Complexity depends on the source
- Strong fault-tolerance guarantees (zero data loss)
- Data source must support acknowledgement
Input DStream and Receiver

1. Submit your job
2. Create Input Get Receiver
3. Receiver Sent
4. Data Source
5. Spark Worker
   Input Receiver Block manager
6. Replicate Blocks
7. Spark Master
   Task Scheduler
   Block Tracker
   Report Block ID's

Receive Data
Store Blocks
Data Source(s)
Creating DStreams

2 Ways to create DStreams

• Input – a streaming source
• Transforming a DStream
Creating a DStream via Transformation

• Transformations modify data from one DStream to another

  ![Diagram showing map transformation from input DStream to output DStream]

• Two general classifications:
  • Standard RDD operations – map, countByValue, reduceByKey, join,…
  • Stateful operations – window, updateStateByKey, transform, countByValueAndWindow, …
Transforming the input - Standard Operation

```scala
val myStream = createCustomStream(streamingContext)
val events = myStream.map(....)
```
Stateful Operation - UpdateStateByKey

Provides a way for you to maintain arbitrary state while continuously updating it.

• For example – In-Session Advertising, Tracking twitter sentiment
Stateful Operation - UpdateStateByKey

Need to do two things to leverage it:

- Define the state – this can be any arbitrary data
- Define the update function – this needs to know how to update the state using the previous state and new values

Requires Checkpoint to be configured
Using `updateStateByKey`

Maintain per-user mood as state, and update it with his/her tweets

```java
moods = tweets.updateStateByKey(tweet => updateMood(tweet))
updateMood(newTweets, lastMood) => newMood
```
### Transform

Allows arbitrary RDD-to-RDD functions to be applied on a DStream

```scala
transform (transformFunc: RDD[T] => RDD[U]): DStream[U]
```

**Example:** We want to eliminate “noise” words from crawled documents:

```scala
val noiseWordRDD = ssc.sparkContext.newAPIHadoopRDD(...)
val cleanedDStream = crawledCorpus.transform(rdd => {
  rdd.join(noiseWordRDD).filter(...)
})
```
Outputting data

```scala
val myStream = createCustomStream(streamingContext)
val events = myStream.map(....)
events.countByValue().foreachRDD{...}
```
From Streaming Program to Spark jobs

```
myStream = createCustomStream

events = myStream.map(…)

events.countByValue()

.foreachRDD{…}
```
Thinking about time
Thinking about time

- Windowing – Tumbling, Sliding
- Stream time vs. Event time
- Out of order data
Windowing

• Common Types
  • Tumbling
  • Sliding
Tumbling (Count) Windowing

Window length
Sliding interval

Time (in seconds)

Tumbling count-window
Tumbling (temporal) Windowing

Window length
Sliding interval

Time (in seconds)
Tumbling temporal window
Sliding Window

Window length

Sliding interval

Time (in seconds)
Spark Streaming -- Sliding Windowing

• Two types supported:
  • Incremental
  • Non-Incremental
Non-Incremental Sliding Windowing

\[
\text{reduceByKeyAndWindow}((a,b)\Rightarrow(a + b), \text{Seconds}(5), \text{Seconds}(1))
\]
Incremental Sliding Windowing

```
reduceByKeyAndWindow((a,b) => (a + b), (a,b) => (a-b), Seconds(5), Seconds(1))
```
More thinking about time

Stream time vs. Event time

- **Stream time** -- the time when the record arrives into the streaming system.
- **Event time** – the time that the event was generated, not when it entered the system.
- Spark Streaming uses stream time

Out of order data

- Does it matter to your application?
- How do you deal with it?
Handling Out of Order Data

Imagine we want to track ad impressions between time $t$ and $t + 1$.
Recovery and Fault Tolerance
Recovery

- Checkpointing
  - Metadata checkpointing
  - Data checkpointing
Recovery

Without

With
Recovery

• Too frequent: HDFS writing will slow things down

• Too infrequent: Lineage and task sizes grow

• Default setting: Multiple of batch interval at least 10 seconds

• **Recommendation:** checkpoint interval of 5 - 10 times of sliding interval
Fault Tolerance

• All properties of RDDs still apply

• We are trying to protect two things
  • Failure of a Worker
  • Failure of the Driver Node

• Semantics
  • At most once
  • At least once
  • Exactly once

• Where we need to think about it
  • Receivers
  • Transformations
  • Output
Conclusion

• Introduction
• High-level Architecture
• DStreams
• Thinking about time
• Recovery and Fault tolerance
Thank you

Andrew Psaltis
@itmdatal
psaltis.andrew@gmail.com
https://www.linkedin.com/in/andrewpsaltis