Spark Streaming

2015

Professor Sasu Tarkoma

HELSINGIN YLIOPISTO
HELSINGFORS UNIVERSITET
UNIVERSITY OF HELSINKI

Spark Streaming

Spark extension of accepting and processing of streaming high-throughput live data streams

Data is accepted from various sources

Kafka, Flume, TCP sockets, Twitter, …

Machine learning algorithms and graph processing algorithms can be applied for the streams

Similar systems

Twitter (Storm), Google (MillWheel), Yahoo! (S4)

Discretized Streams: A Fault-Tolerant Model for Scalable Stream Processing. Matei Zaharia,Tathagata Das, Haoyuan Li, Timothy Hunter, Scott Shenker, Ion Stoica. Berkeley EECS (2012-12-14)

Streaming Systems

Traditional streaming systems are based on event-driven event-at-a-time processing model

Each node has state and the state is updated for each event

If the node fails, the state is lost thus creating challenges for fault-tolerance

Well-known systems

- Storm
  - Each record is processed **at least once**
  - State can be lost due to failure
- Trident
  - Each record is processed **exactly once** (replay tuples for fault tolerance, stores additional state information)
  - Transactions are slow

Stream Processing: Discretized

Streaming computation is run as a series of very small deterministic batch jobs

Live stream is divided into **batches of x seconds**

Each batch of data is an RDD and RDD operations can be used

Results are also returned in batches

Batch size as low as 0.5 seconds, results in approx. one second latency

Can combine streaming and batch processing

Achieving Fault-Tolerance

RDDs store the sequence of operations that were used to create it

Batches of input are replicated in memory of multiple workers

Worker failure can be mitigated by recomputing the lost data

Concepts

DStream

- Sequence of RDDs
- Stream data can be based on various sources

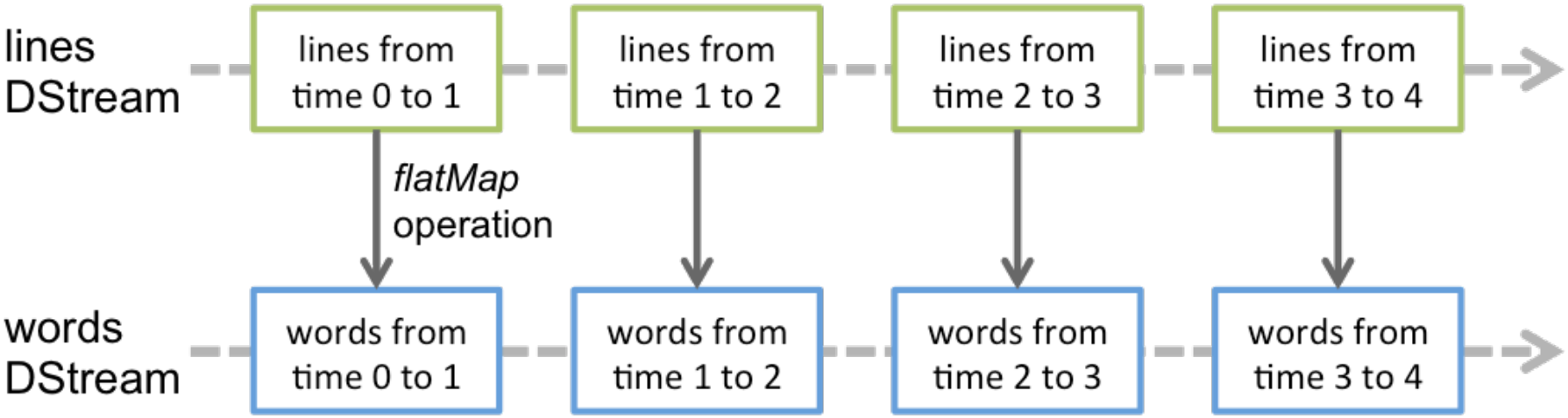
Transformations

- Modifies DStream data and creates a new DStream
- Basic RDD operations: map, countByValue, …
- Stateful operations: window, countbyValueAndWindow, …

Output

- Save to HDFS
- foeachRDD: store each RDD of the stream batch to an external system

DStream (batches of RDDs)



https://spark.apache.org/docs/latest/streaming-programming-guide.html

Creating Streams

A StreamingContext object can be created from a SparkConf object.

```
import org.apache.spark._
import org.apache.spark.streaming._

val conf = new
  SparkConf().setAppName(appName).setMaster(master)
val ssc = new StreamingContext(conf, Seconds(1))

// or from existing SparkContext sc
val ssc = new StreamingContext(sc, Seconds(1))
```

Using Streams

1. Define the input sources by creating input DStreams.
2. Define the streaming computations by applying transformation and output operations to DStreams.
3. Start receiving data and processing it using streamingContext.start().
4. Wait for the processing to be stopped (manually or due to any error) using streamingContext.awaitTermination().
5. The processing can be manually stopped using streamingContext.stop().

Points to Remember

Once a context has been started, no new streaming computations can be set up or added to it.

Once a context has been stopped, it cannot be restarted.

Only one StreamingContext can be active in a JVM at the same time.

Receivers

A Receiver receives data from a source, may acknowledge the data, and stores it in Spark memory

Reliable Receiver - A *reliable receiver* correctly acknowledges a reliable source that the data has been received and stored in Spark with replication.

Unreliable Receiver - These are receivers for sources that do not support acknowledging. Even for reliable sources, one may implement an unreliable receiver that do not go into the complexity of acknowledging correctly.

Transformations

Many Spark transformations are supported: map, flatmap, filter, union, reduce, reduceByKey, join, count, …

UpdateStateByKey updates arbitrary state on the fly

Define the state

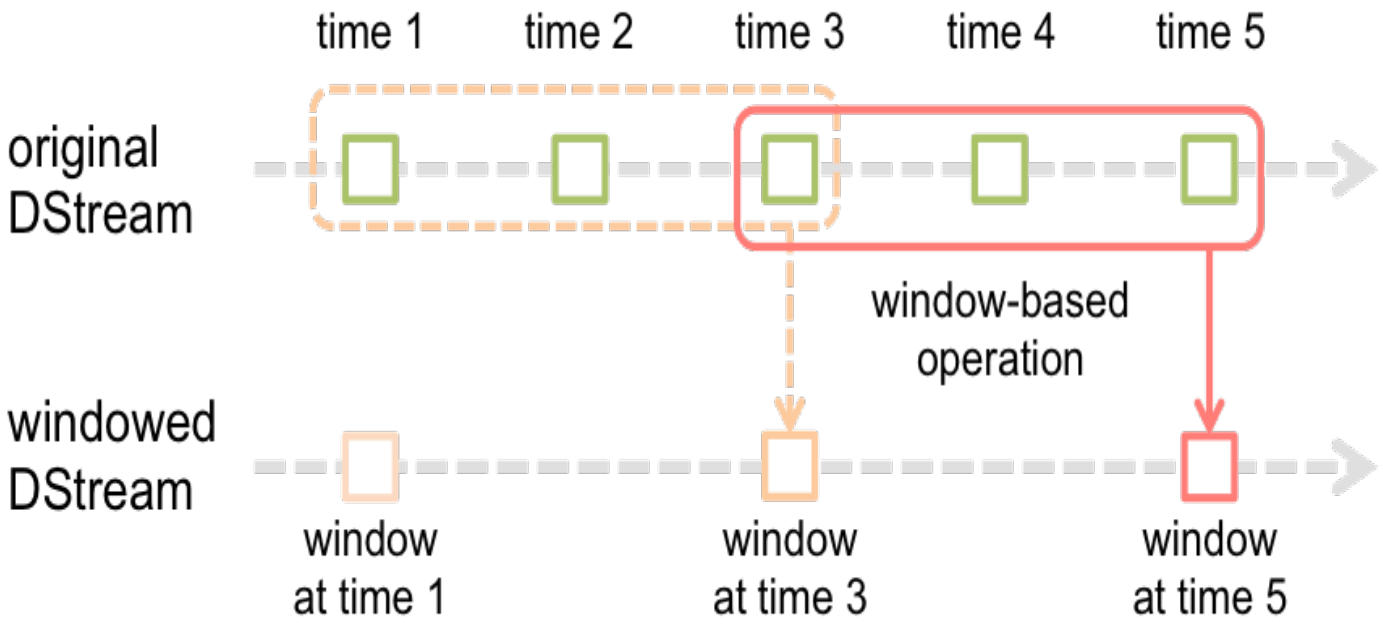
Define the state update function

Transform

```
val cleanedDStream = wordCounts.transform(rdd => {  rdd.join(spamInfoRDD).filter(...) // join data stream with spam information to do data cleaning
 …})
```

Window operations

Source RDDs within the window are combined and processed for the RDDs of the windowed DStream



Window length = 3
Sliding interval = 2

Checkpointing

Spark Streaming uses checkpoints for fault tolerance

Metadata checkpointing: stream information is saved to HDFS or other storage. Can recover from driver failure. Metadata includes: configuration, DStream operations, incomplete batches.

Data checkpointing: Generated RDDs are saved to reliable storage. Needed for some stateful transformations that combine data from multiple batches.

Receiving data

Incoming data needs to be deserialized and stored in Spark

Receive can be parallelized (each receiving DStream running on a single worker machine)

Multiple data streams (multiple DStreams) can be combined

Kafka DStream with two topics → two input streams on two worker nodes

Twitter Example

```
Create a DStream (batches of RDDs)
val tweets = scc.twitterStream(username, password)
Create a new DStream and modify data (new RDDs)
val hashtags =
  tweets.flatMap(status=>getTags(status))
Save to HDFS
hashTags.saveAsHadoopFiles("hdfs://..")
Count how many tags of each type
Val tagC = hashTags.countByValue()
Count hashtags over last 5 minutes
val tagC2 = hashTags.window(Minutes(5),
  Seconds(1)).countByValue()
```

Text Streaming Example

```
import org.apache.spark.streaming._
Import org.apache.spark.streaming.StreamingContext._
val ssc = new StreamingContext(sparkConf, Seconds(10))
val lines = ssc.socketTextStream(serverIP, serverPort)
val words = lines.flatMap(_.split(" "))
// Count each word in each batch!
val pairs = words.map(word => (word, 1))
val wordCounts = pairs.reduceByKey(_ + _)
wordCounts.print()
// The logic has now been defined, we need to start
ssc.start() // Start the computation
ssc.awaitTermination()
// Wait for the computation to terminate
```

Machine Learning

MLLib can be used with streaming

- Streaming machine learning algorithms
  - Linear regression, kmeans, …

Two approaches:

- Simultaneously learn from data and apply model
- First learning a model offline and then using it on the stream

Performance

Better throughput than Storm reported

- Spark streaming 670k records / second / node
- Storm 115k records / second / node
- Apache S4: 7.5k records / second / node

Reported to recover from faults within 1 sec

Conviva case: 1-2 second latency for real-time monitoring of video metadata, linear scalability observed