Deep dive into Spark Streaming

Tathagata Das (TD)

Matei Zaharia, Haoyuan Li, Timothy Hunter,
Patrick Wendell and many others

amplab
UC BERKELEY

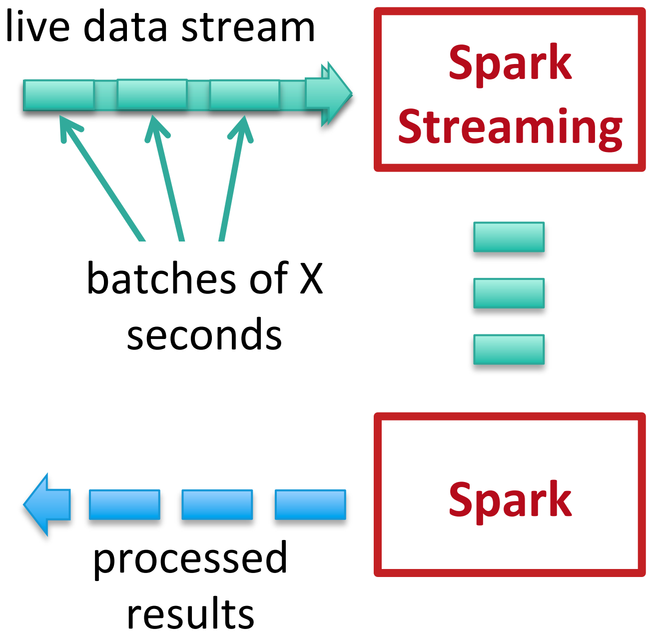
What is Spark Streaming?

- Extends Spark for doing large scale stream processing
- Scales to 100s of nodes and achieves second scale latencies
- Efficient and fault-tolerant stateful stream processing
- Integrates with Spark's batch and interactive processing
- Provides a simple batch-like API for implementing complex algorithms

Discretized Stream Processing

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

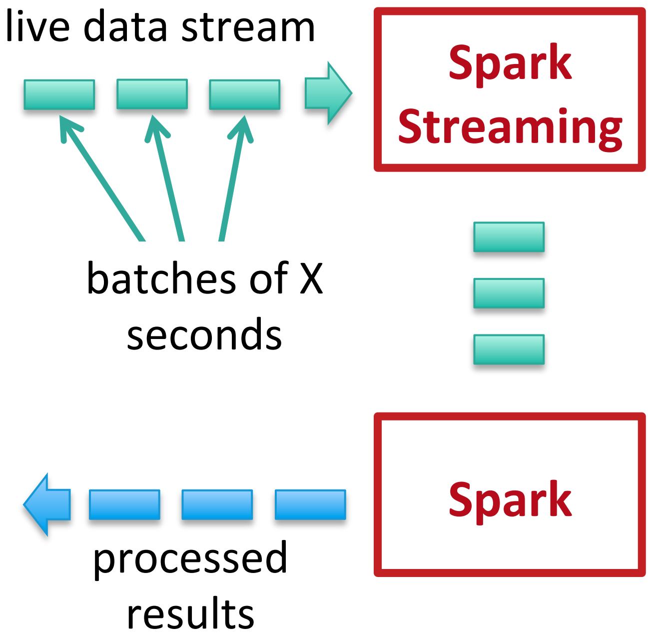
- Chop up the live stream into batches of X seconds
- Spark treats each batch of data as RDDs and processes them using RDD operations
- Finally, the processed results of the RDD operations are returned in batches



Discretized Stream Processing

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

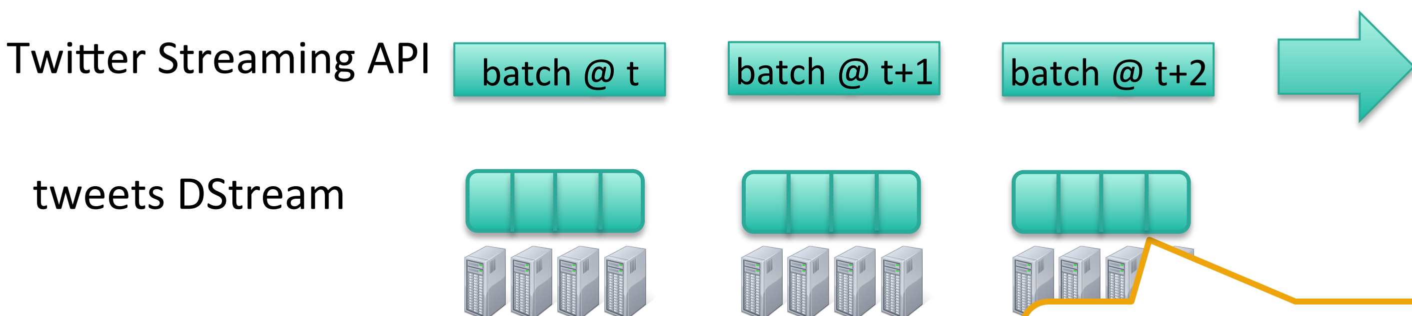
- Batch sizes as low as ½ second, latency ~ 1 second
- Potential for combining batch processing and streaming processing in the same system



Example 1 – Get hashtags from Twitter

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

DStream: a sequence of distributed datasets (RDDs) representing a distributed stream of data



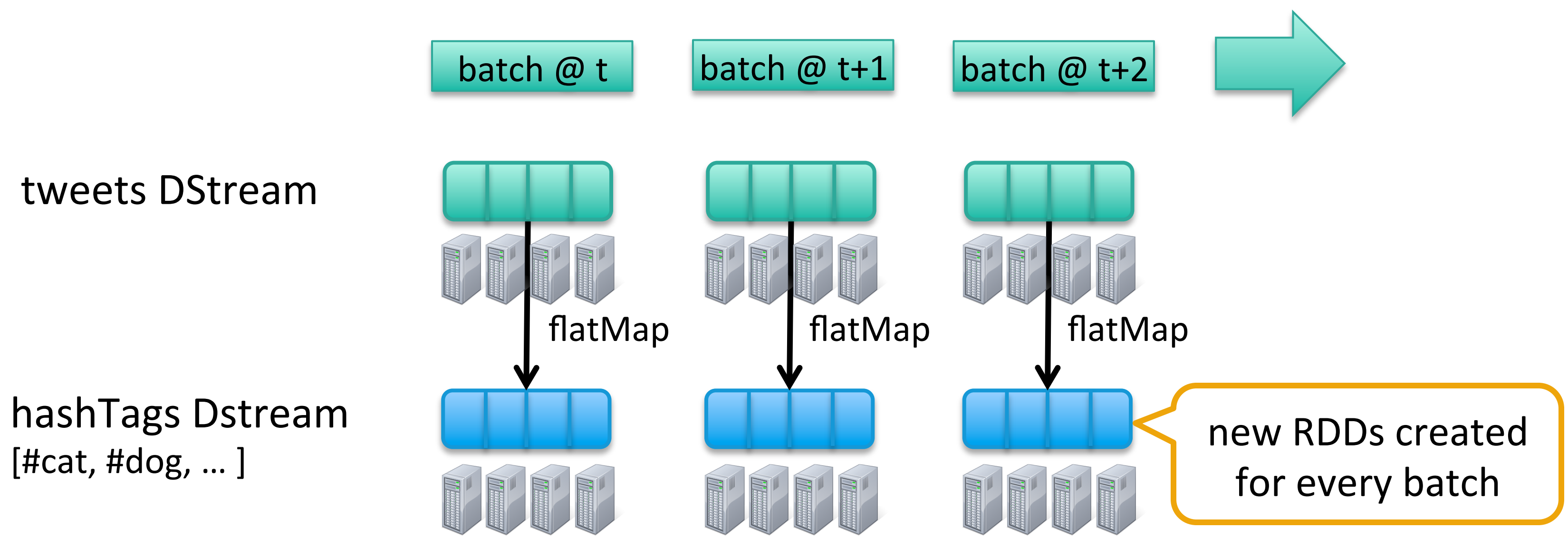
stored in memory as an RDD (immutable, distributed dataset)

Example 1 – Get hashtags from Twitter

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

new DStream

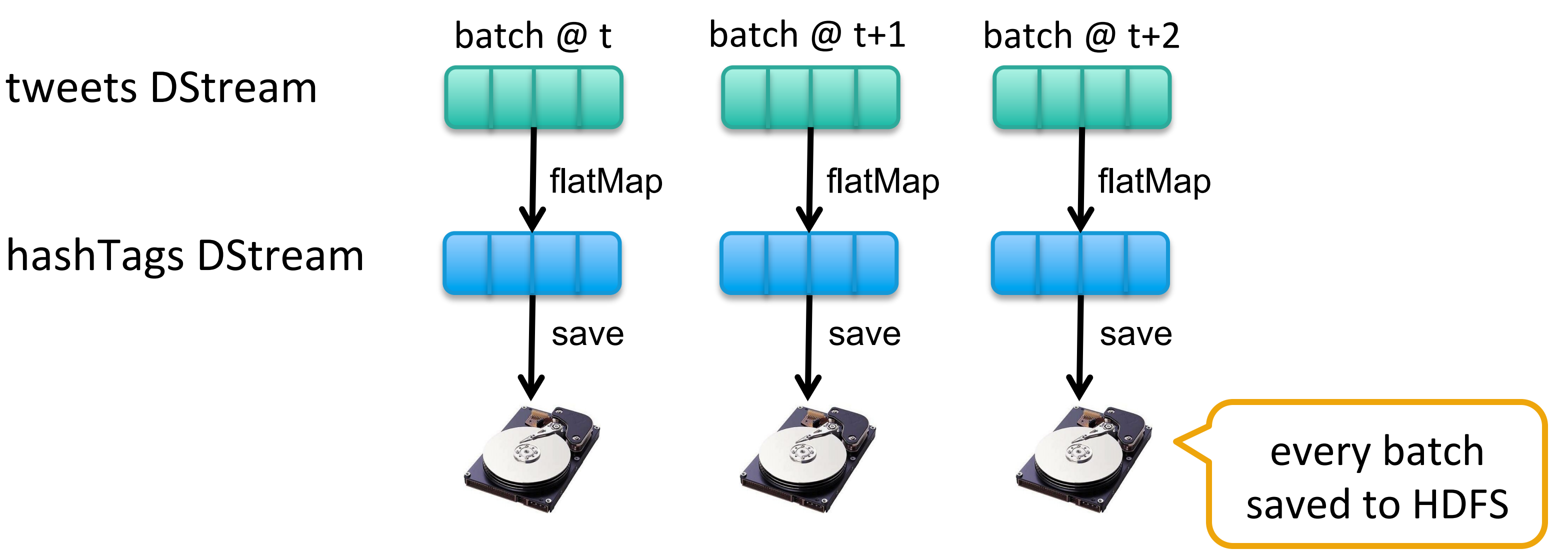
transformation: modify data in one DStream to create another DStream



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



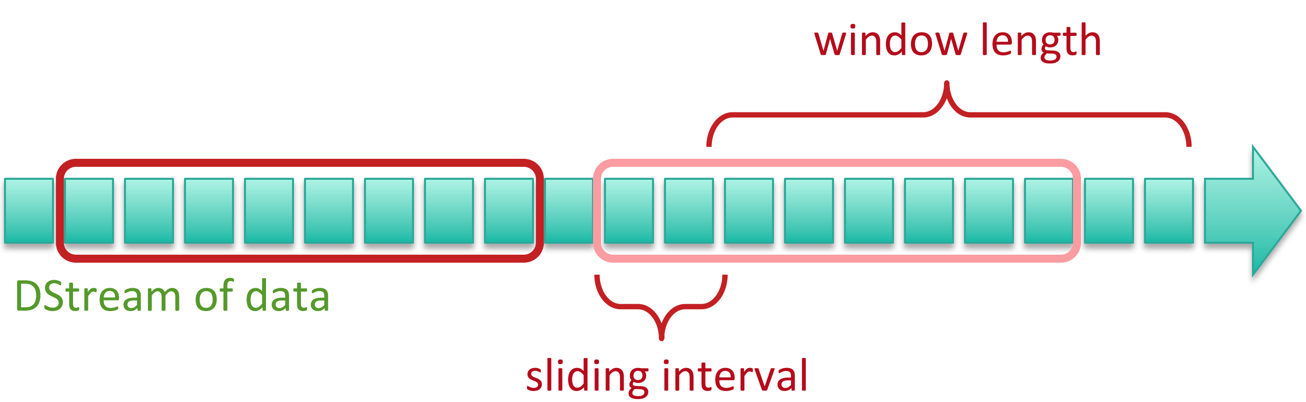
Example 2 – Count the hashtags over last 1 min

```
val tweets = ssc.twitterStream(<Twitter username>, <Twitter password>)
val hashTags = tweets.flatMap (status => getTags(status))
val tagCounts = hashTags.window(Minutes(1), Seconds(1)).countByValue()
```

sliding window operation

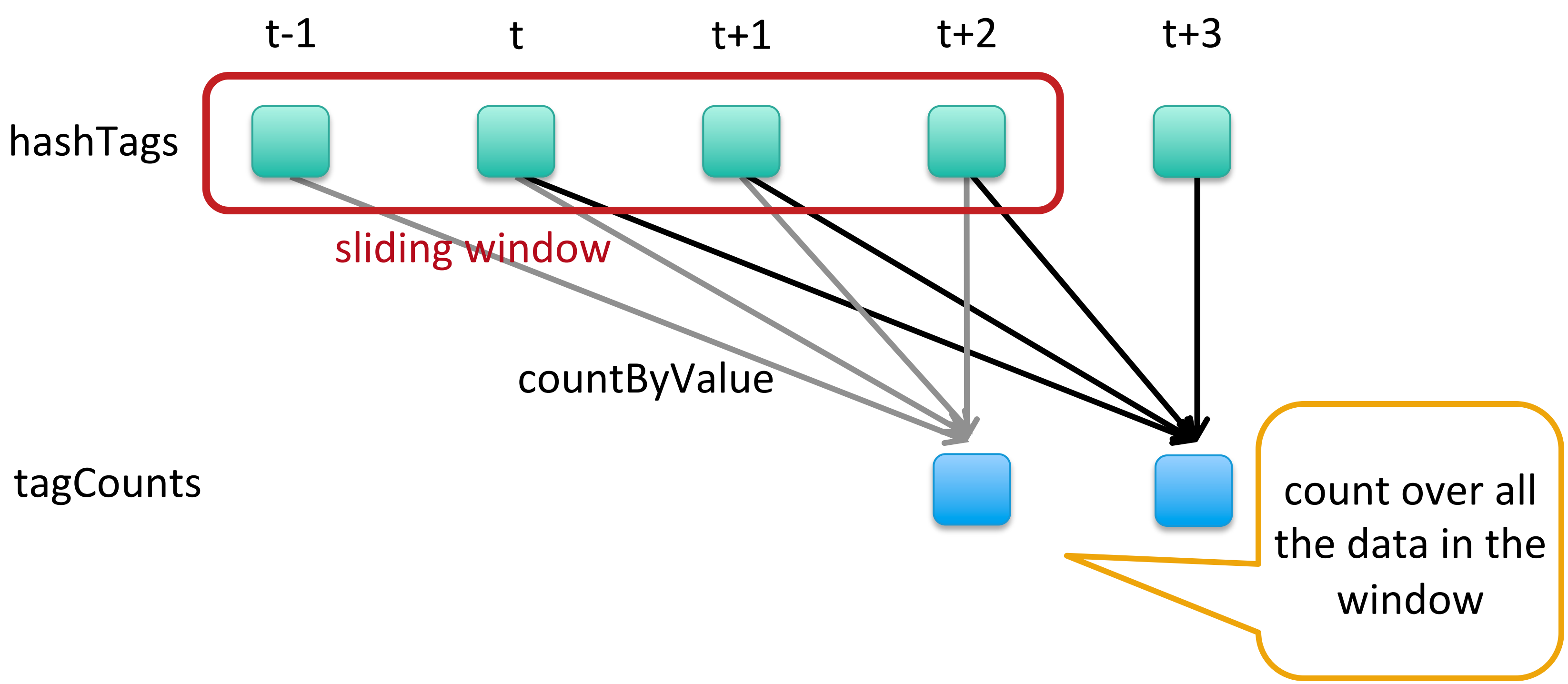
window length

sliding interval



Example 2 – Count the hashtags over last 1 min

```
val tagCounts = hashTags.window(Minutes(1), Seconds(1)).countByValue()
```



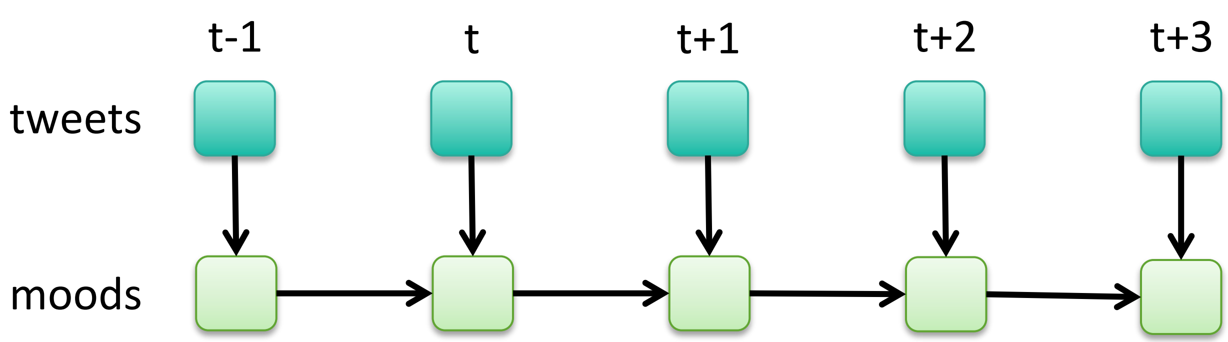
Key concepts

- **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, reduceByKey, join, …
  - Stateful operations – window, countByValueAndWindow, …
- **Output Operations –** send data to external entity
  - saveAsHadoopFiles – saves to HDFS
  - foreach – do anything with each batch of results

Arbitrary Stateful Computations

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

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



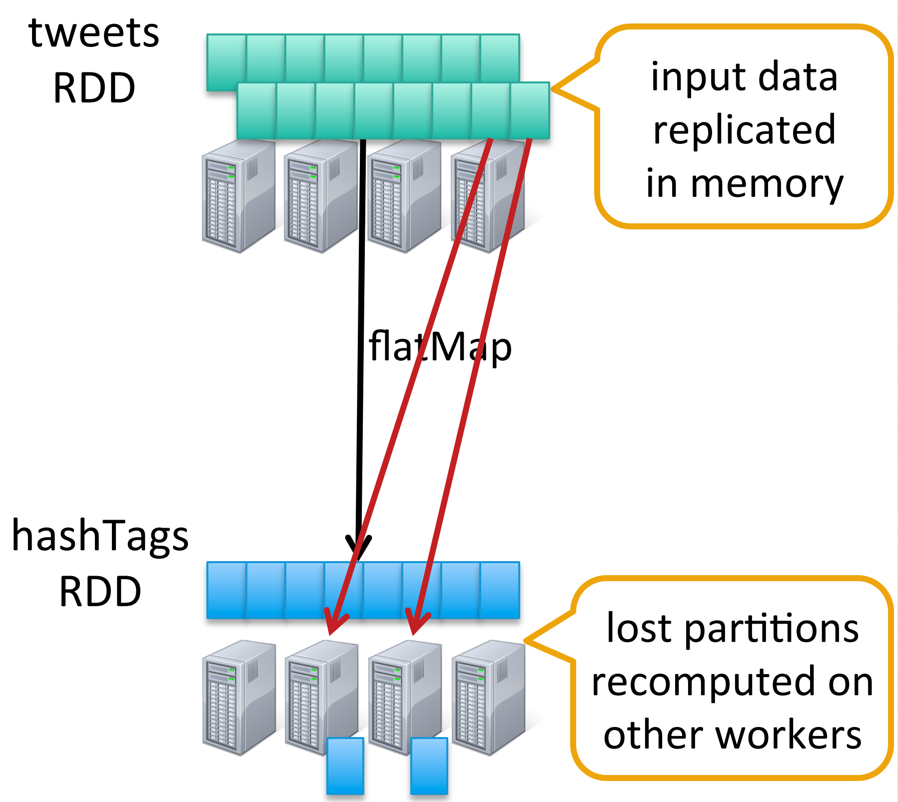
Combine Batch and Stream Processing

- 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(...)
})
```

Fault-tolerance

- RDDs remember the operations that created them
- Batches of input data are replicated in memory for fault-tolerance
- Data lost due to worker failure, can be recomputed from replicated input data
- Therefore, all transformed data is fault-tolerant



Agenda

- Overview
- DStream Abstraction
- System Model
- Persistence / Caching
- RDD Checkpointing
- Performance Tuning

Discretized Stream (DStream)

A sequence of RDDs representing
a stream of data

What does it take to define a DStream?

DStream Interface

The DStream interface primarily defines how to generate an RDD in each batch interval

- List of *dependent (*parent) DStreams
- *Slide Interval,* the interval at which it will compute RDDs
- Function to *compute* RDD at a time t

Example: Mapped DStream

- *Dependencies:* Single parent DStream
- *Slide Interval:* Same as the parent DStream
- *Compute function for time t:* Create new RDD by applying map function on parent DStream's RDD of time t

Example: Mapped DStream

- *Dependencies:* Single parent DStream
- *Slide Interval:* Same as the parent DStream
- *Compute function for time t:* Create new RDD by applying map function on parent DStream's RDD of time t

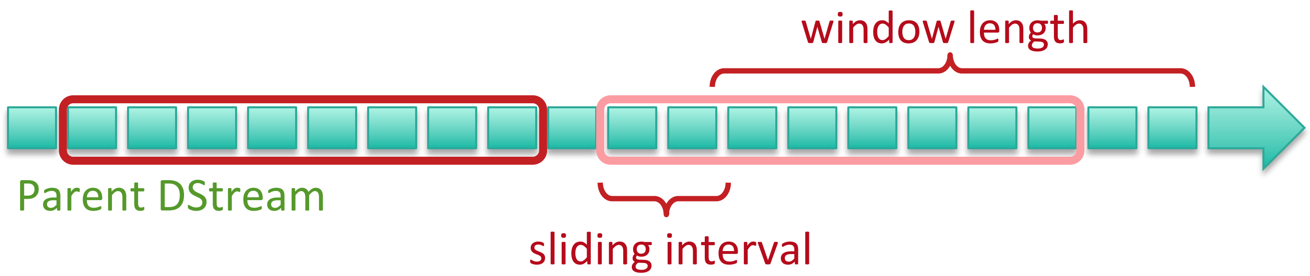
```
override def compute(time: Time): Option[RDD[U]] = {
    parent.getOrCompute(time).map(_.map[U](mapFunc))
}
```

Gets RDD of time t if already computed once, or generates it

Map function applied to generate new RDD

Example: Windowed DStream

Window operation gather together data over a sliding window



- *Dependencies:* Single parent DStream
- *Slide Interval:* Window sliding interval
- *Compute function for time t:* Apply union over all the RDDs of parent DStream between times t and (t – window length)

Example: Network Input DStream

Base class of all input DStreams that receive data from the network

- *Dependencies:* None
- *Slide Interval:* Batch duration in streaming context
- *Compute function for time t:* Create a BlockRDD with all the blocks of data received in the last batch interval
- Associated with a Network Receiver object

Network Receiver

Responsible for receiving data and pushing it into Spark's data management layer (Block Manager)

Base class for all receivers - Kafka, Flume, etc.

Simple Interface:

- What to do *on starting* the receiver
  - Helper object *blockGenerator* to push data into Spark
- What to do *on stopping* the receiver

Example: Socket Receiver

- *On start*:

  Connect to remote TCP server

  While socket is connected,

  Receiving bytes and deserialize

  Deserialize them into Java objects

  Add the objects to *blockGenerator*

- *On stop*:

  Disconnect socket

Other functions in DStream interface

- *parentRememberDuration* – defines how long should
  - Window-based DStreams have *parentRememberDuration = window length*
- *mustCheckpoint* – if set to true, the system will automatically enable periodic checkpointing
  - Set to true for stateful DStreams

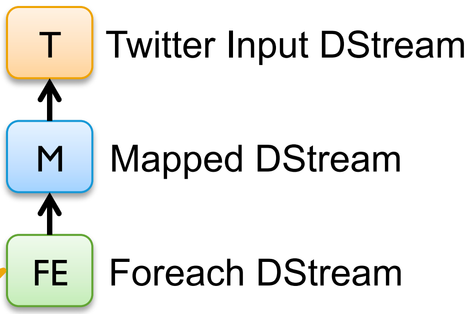
DStream Graph

Spark Streaming program

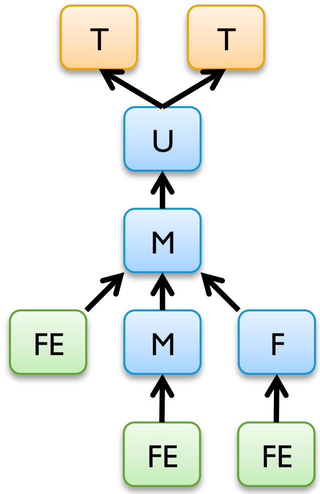
```
t = ssc.twitterStream("…")
       .map(…)
t.foreach(…)
```

```
t1 = ssc.twitterStream("…")
t2 = ssc.twitterStream("…")

t = t1.union(t2).map(…)

t.saveAsHadoopFiles(…)
t.map(…).foreach(…)
t.filter(…).foreach(…)
```

DStream Graph

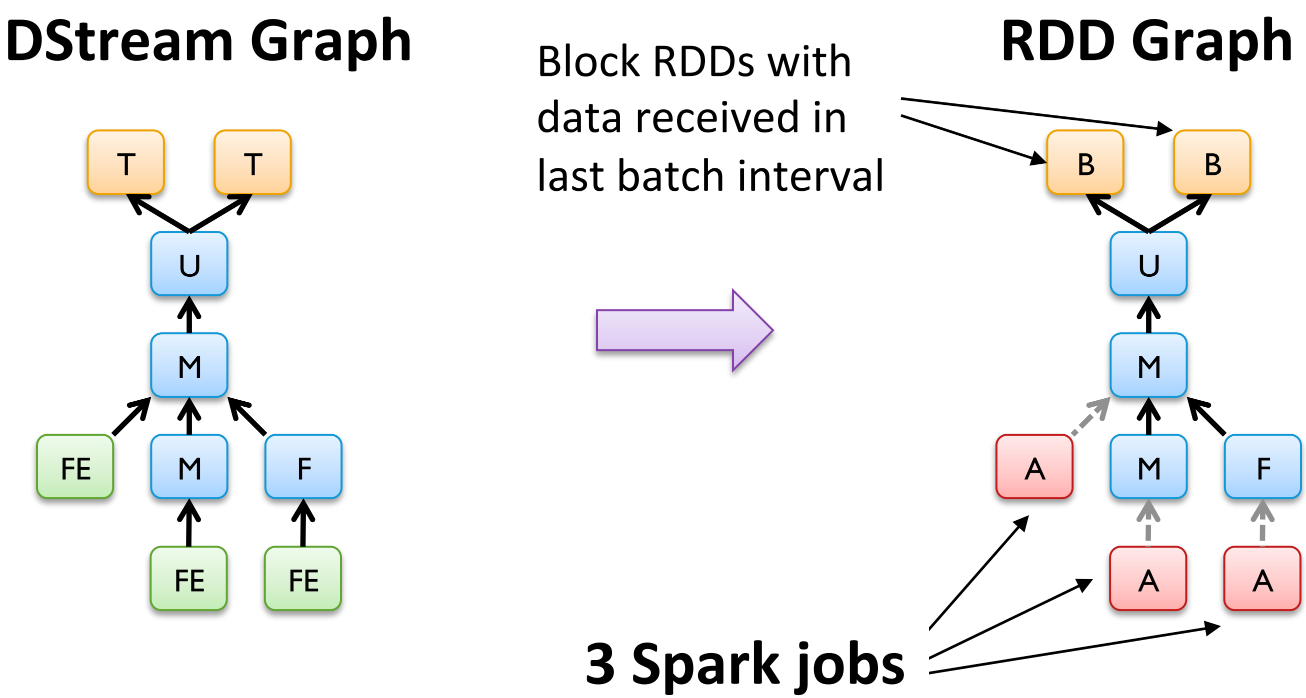


Dummy DStream signifying an output operation



DStream Graph → RDD Graphs → Spark jobs

- Every interval, RDD graph is computed from DStream graph
- For each output operation, a Spark action is created
- For each action, a Spark job is created to compute it

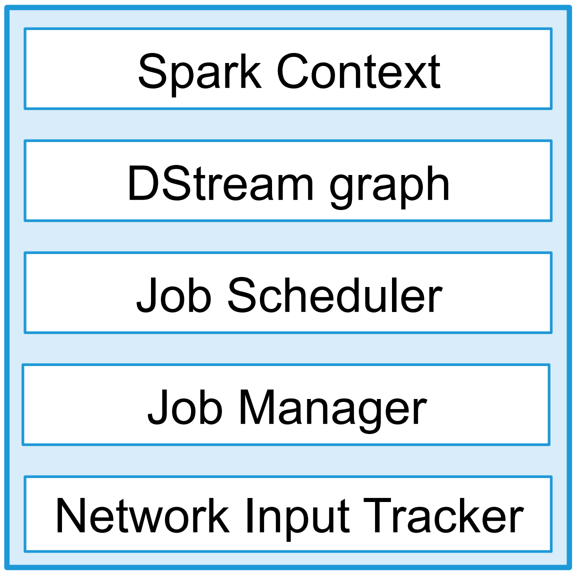


Agenda

- Overview
- DStream Abstraction
- System Model
- Persistence / Caching
- RDD Checkpointing
- Performance Tuning

Components

Spark Streaming Client



Job Scheduler

Periodically queries the DStream graph to generate Spark jobs from received data, and hands them to Job Manager for execution

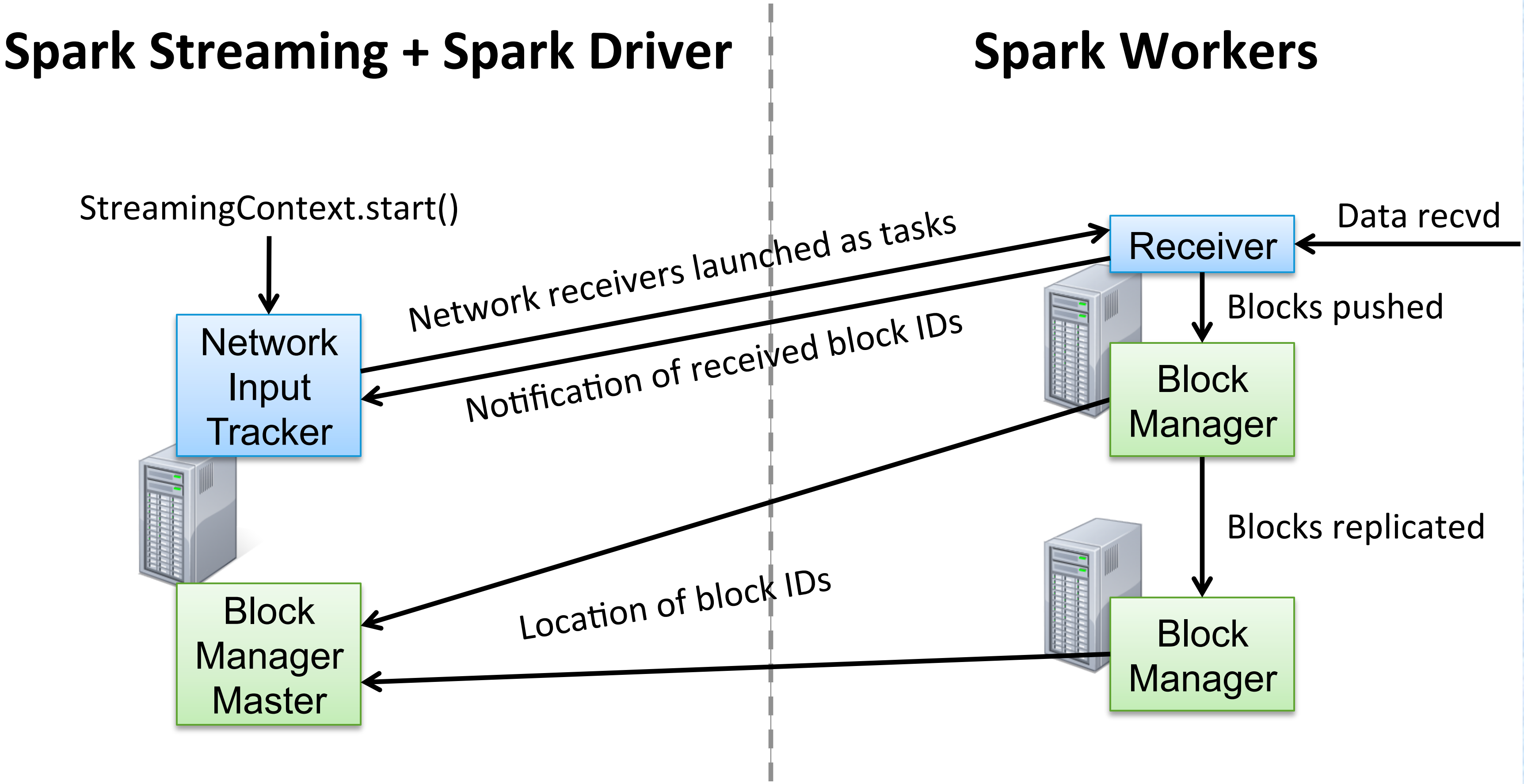
Job Manager

Puts jobs in a queue and runs them in Spark

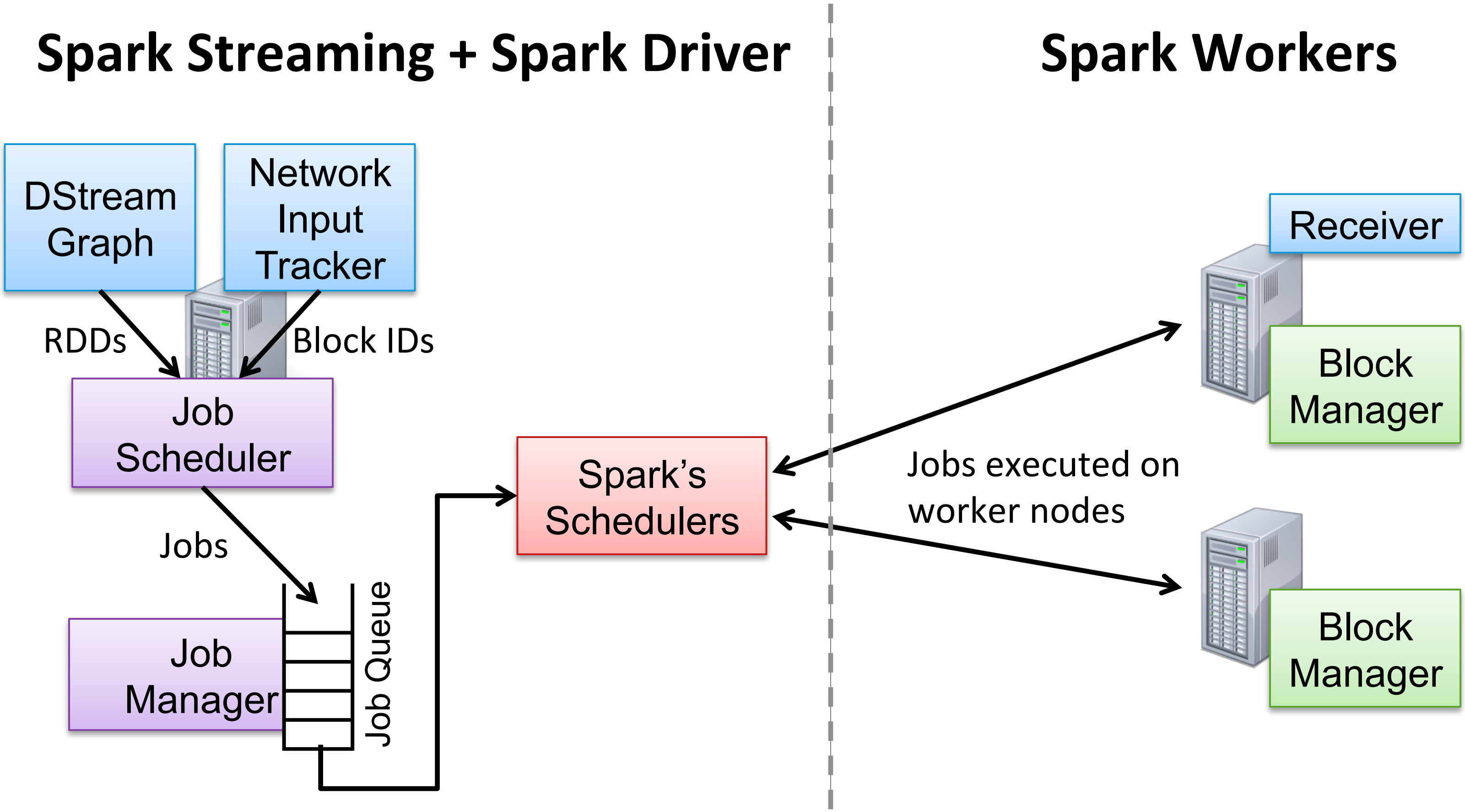
Network Input Tracker

Keeps track of the data received by each network receiver and maps them to the corresponding input DStreams

Execution Model – Receiving Data



Execution Model – Job Scheduling



Job Scheduling

- Each output operation used generates a job
  - More jobs → more time taken to process batches → higher batch duration

- Job Manager decides how many concurrent Spark jobs to run
  - Default is 1, can be set using Java property `spark.streaming.concurrentJobs`
  - If you have multiple output operations, you can try increasing this property to reduce batch processing times and so reduce batch duration

Agenda

- Overview
- DStream Abstraction
- System Model
- Persistence / Caching
- RDD Checkpointing
- Performance Tuning

DStream Persistence

- If a DStream is set to persist at a storage level, then all RDDs generated by it set to the same storage level

- When to persist?
  - If there are multiple transformations / actions on a DStream
  - If RDDs in a DStream is going to be used multiple times

- Window-based DStreams are automatically persisted in memory

DStream Persistence

- Default storage level of DStreams is `StorageLevel.MEMORY_ONLY_SER` (i.e. in memory as serialized bytes)
  - Except for input DStreams which have `StorageLevel.MEMORY_AND_DISK_SER_2`
  - Note the difference from RDD's default level (no serialization)
  - Serialization reduces random pauses due to GC providing more consistent job processing times

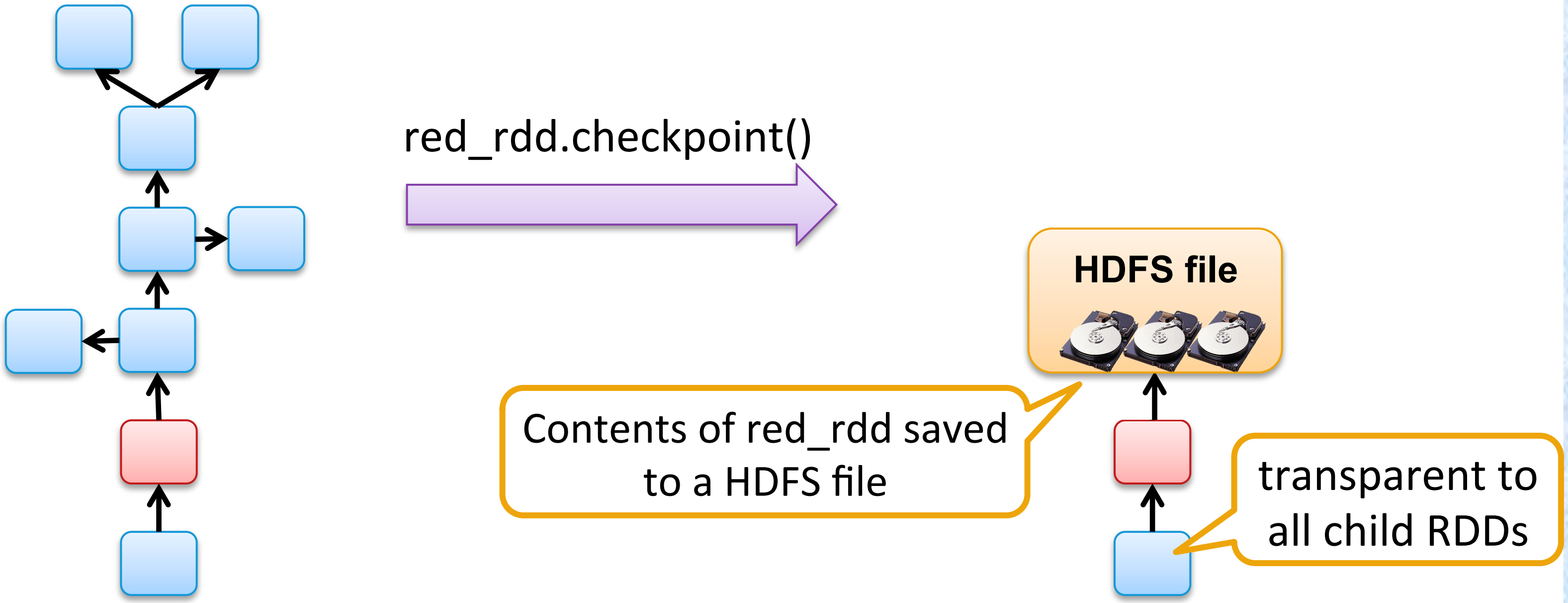
Agenda

- Overview
- DStream Abstraction
- System Model
- Persistence / Caching
- RDD Checkpointing
- Performance Tuning

What is RDD checkpointing?

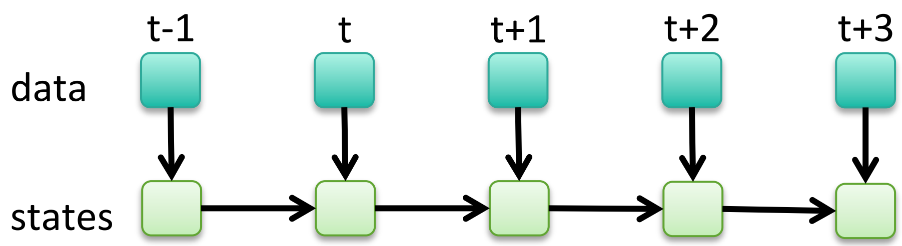
Saving RDD to HDFS to prevent RDD graph from growing too large

- Done internally in Spark transparent to the user program
- Done lazily, saved to HDFS the first time it is computed



Why is RDD checkpointing necessary?

Stateful DStream operators can have infinite lineages

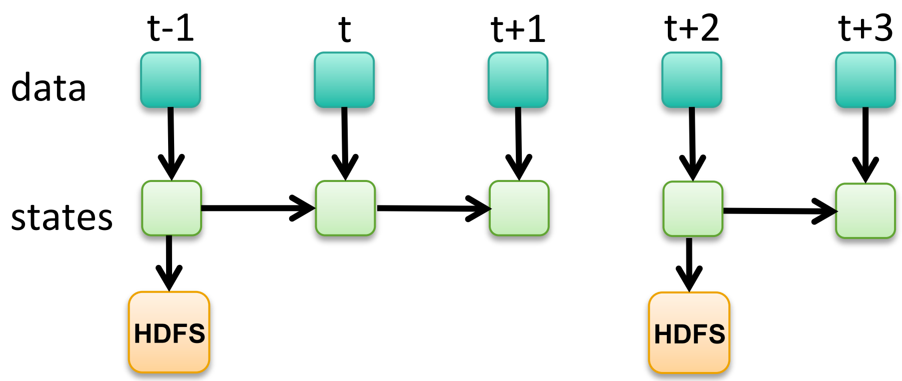


Large lineages lead to …

- Large closure of the RDD object → large task sizes → high task launch times
- High recovery times under failure

Why is RDD checkpointing necessary?

Stateful DStream operators can have infinite lineages



Periodic RDD checkpointing solves this

Useful for iterative Spark programs as well

RDD Checkpointing

- Periodicity of checkpoint determines a tradeoff
  - Checkpoint too frequent: HDFS writing will slow things down
  - Checkpoint too infrequent: Task launch times may increase
  - Default setting checkpoints at most once in 10 seconds
  - Try to checkpoint once in about 10 batches

Agenda

- Overview
- DStream Abstraction
- System Model
- Persistence / Caching
- RDD Checkpointing
- Performance Tuning

Performance Tuning

Step 1

Achieve a stable configuration that can sustain the streaming workload

Step 2

Optimize for lower latency

Step 1: Achieving Stable Configuration

How to identify whether a configuration is stable?

- Look for the following messages in the log

```
Total delay: 0.01500 s for job 12 of time 1371512674000 …
```

- If the total delay is continuously increasing, then unstable as the system is unable to process data as fast as its receiving!
- If the total delay stays roughly constant and around 2x the configured batch duration, then stable

Step 1: Achieving Stable Configuration

How to figure out a good stable configuration?

- Start with a low data rate, small number of nodes, reasonably large batch duration (5 – 10 seconds)
- Increase the data rate, number of nodes, etc.
- Find the bottleneck in the job processing
  - Jobs are divided into stages
  - Find which stage is taking the most amount of time

Step 1: Achieving Stable Configuration

How to figure out a good stable configuration?

- If the first map stage on raw data is taking most time, then try …
  - Enabling delayed scheduling by setting property `spark.locality.wait`
  - Splitting your data source into multiple sub streams
  - Repartitioning the raw data into many partitions as first step
- If any of the subsequent stages are taking a lot of time, try…
  - Try increasing the level of parallelism (i.e., increase number of reducers)
  - Add more processors to the system

Step 2: Optimize for Lower Latency

- Reduce batch size and find a stable configuration again
  - Increase levels of parallelism, etc.

- Optimize serialization overheads
  - Consider using Kryo serialization instead of the default Java serialization for both data and tasks
  - For data, set property `spark.serializer=spark.KryoSerializer`
  - For tasks, set `spark.closure.serializer=spark.KryoSerializer`

- Use Spark stand-alone mode rather than Mesos

Step 2: Optimize for Lower Latency

- Using concurrent mark sweep GC `-XX:+UseConcMarkSweepGC` is recommended
  - Reduces throughput a little, but also reduces large GC pauses and may allow lower batch sizes by making processing time more consistent
- Try disabling serialization in DStream/RDD persistence levels
  - Increases memory consumption and randomness of GC related pauses, but may reduce latency by further reducing serialization overheads
- For a full list of guidelines for performance tuning
  - Spark Tuning Guide
  - Spark Streaming Tuning Guide

Future (Possible) Directions

- Better master fault-tolerance
- Better performance for complex queries
  - Better performance for stateful processing is a low hanging fruit
- Dashboard for Spark Streaming
  - Continuous graphs of processing times, end-to-end latencies
  - Drill down for analyzing processing times of stages for finding bottlenecks
- Python API for Spark Streaming