Spark Streaming

Spark Streaming

- In this chapter, you will learn
  - What Spark Streaming is and why it is valuable
  - How to use Spark Streaming
  - How to work with Sliding Window operations

Chapter Topics

- Spark Streaming Overview
- Example: Streaming Request Count
- DStreams
- Hands-On Exercise: Exploring Spark Streaming
- State Operations
- Sliding Window Operations
- Developing Spark Streaming Applications
- Flume Vs. Storm Vs. Spark Streaming
- Summary
- Review
- References
- Hands-On Exercise: Writing A Spark Streaming Application

The Big Idea

- Currently, you know how to process the contents of individual RDDs using Spark
- But what if you have a continuous flow of RDDs streaming into an application?
  - You want to leverage the knowledge you have already acquired for processing individual RDDs to be able to process continuous streams of RDDs flowing into your application on an ongoing basis
  - Toward this end we introduce the concept of a DStream (Discretized Stream) an abstraction for a sequence of RDDs representing a continuous stream of data
    - A DStream is to Spark Streaming what an RDD is to Spark (with equivalent types of functionality)

What Is Spark Streaming?

- Spark Streaming provides real-time processing of stream data as it arrives in the cluster
  - An extension of core Spark
  - Currently supports Scala, Java, and Python (as of Spark 1.2)
  - https://spark.apache.org/docs/latest/streaming-programming-guide.html

Why Spark Streaming?

- Many big-data applications need to process large data streams in real time
  - Monitoring a web site against a DOS attack
  - Monitoring credit card transactions to prevent fraud
  - Monitoring search queries to choose relevant advertisements
- What do these applications share in common?
  - The need to respond with very low latency

Spark Streaming Features

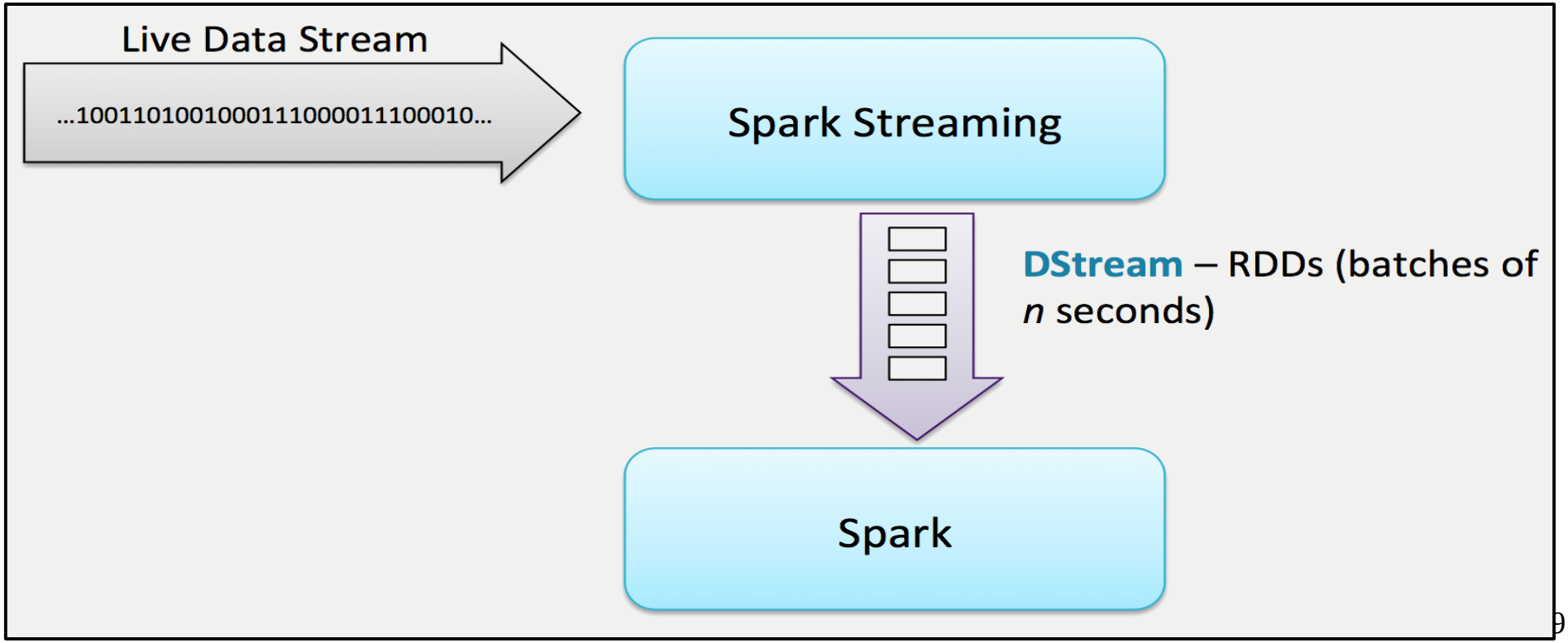
- Second-scale latencies
  - The lower limit is approximately 1/2 second
- Scalability and efficient fault tolerance
- "Once and only once" processing
  - As opposed to "At least once" processing (which requires a different, idempotent programming model)
- Integrates batch and real-time processing
  - The basic approach is to batch all data arriving within a small time frame for processing as a unit in a new RDD
    - 1/2 second – 3 seconds
  - So the same programming model used for Spark batch processing can be applied to streaming data
- Easy to develop
  - Uses Spark's high level API

How Does Spark Streaming Differ From Apache Storm?

- FYI, Storm is a competing Apache project offering low latency streaming (lower than Spark Streaming) analytic capabilities
  - https://storm.incubator.apache.org
- Among the differences of note, Spark Streaming addresses some of the shortcomings of Storm
  - Fault tolerance
    - Spark Streaming guarantees once and only once processing
    - Storm guarantees at least once processing, which requires a slightly more complex (idempotent) programming model
  - Integrates batch and realtime processing
    - Spark Streaming treats realtime data the same as stored data
    - Storm treats them differently, requiring extra logic to integrate the two
    - Useful for joining streaming data (e.g. logs) with static data (e.g. customer account information)
- http://xinhstechblog.blogspot.com/2014/06/storm-vs-spark-streaming-side-by-side.html
  - A nice comparison of Storm and Spark Streaming

Spark Streaming Overview

- Divide a data stream into batches of n seconds
  - Perhaps using Flume or Kafka
- Process each batch in Spark as a separate RDD
- Return the results of the RDD operations in batches



Chapter Topics

- Spark Streaming Overview
- Example: Streaming Request Count
- DStreams
- Hands-On Exercise: Exploring Spark Streaming
- State Operations
- Sliding Window Operations
- Developing Spark Streaming Applications
- Flume Vs. Storm Vs. Spark Streaming
- Summary
- Review
- References
- Hands-On Exercise: Writing A Spark Streaming Application

Streaming Example: Streaming Request Count (1 Of 2)

```
import org.apache.spark.streaming.{Seconds, StreamingContext}
import org.apache.spark.streaming.StreamingContext._
import org.apache.spark.SparkContext._
import org.apache.spark.SparkConf
object StreamingRequestCount {
    def main(args: Array[String]) {
        if (args.length < 2) {
            System.err.println("Usage: StreamingRequestCount <host> <port>")
            System.exit(1)
        }
        val host = args(0)
        val port = args(1).toInt
        val ssc = new StreamingContext(new SparkConf(), Seconds(2))
        val logs = ssc.socketTextStream(host, port)
        val userreqs = logs
             .map(line => (line.split(' ')(2), 1))
             .reduceByKey((v1, v2) => v1 + v2)
        userreqs.print()
        ssc.start()
        ssc.awaitTermination()
    }
}
```

Streaming Example: Streaming Request Count (2 Of 2)

- This example simulates streaming data by reading from a file, whose contents have been redirected to a network socket
  - The file represents a web log
  - The program counts the frequency with which different users access a web site

Streaming Example: Configuring A StreamingContext (1 Of 2)

```
import org.apache.spark.streaming.{Seconds, StreamingContext}
import org.apache.spark.streaming.StreamingContext._
import org.apache.spark.SparkContext._
import org.apache.spark.SparkConf
object StreamingRequestCount {
    def main(args: Array[String]) {
        if (args.length < 2) {
            System.err.println("Usage: StreamingRequestCount <host> <port>")
            System.exit(1)
        }
        val host = args(0)
        val port = args(1).toInt
        val ssc = new StreamingContext(new SparkConf(), Seconds(1))
        // Create a stream of user requests
        val userreqs = ssc.socketTextStream(host, port)
        // Count requests by user id
        val reqcounts = userreqs.map(x => (x, 1)).reduceByKey(_ + _)
        // Print the first ten elements of each RDD generated in this stream to the console
        reqcounts.print()
        ssc.start()
        ssc.awaitTermination()
    }
}
```

- A StreamingContext is the main entry point for Spark Streaming apps
- Equivalent to SparkContext in core Spark
- Configured with the same parameters as a SparkContext, plus a batch
- duration – an instance of Milliseconds, Seconds, or Minutes
- Named ssc by convention

Streaming Example: Configuring A StreamingContext (2 Of 2)

- The parameters for constructing a StreamingContext are identical to those for a SparkContext with one additional parameter: batchDuration
  - An instance of an org.apache.spark.streaming.Duration, representing the duration of each batch (latency)
  - Smallest practical duration for Spark Streaming is 500 milliseconds

Streaming Example: Creating A DStream (1 Of 2)

```
import org.apache.spark.streaming.{Seconds, StreamingContext}
import org.apache.spark.streaming.StreamingContext._
import org.apache.spark.SparkContext._
import org.apache.spark.SparkConf
object StreamingRequestCount {
    def main(args: Array[String]) {
        if (args.length < 2) {
            System.err.println("Usage: StreamingRequestCount <host> <port>")
            System.exit(1)
        }
        val host = args(0)
        val port = args(1).toInt
        val userreqs = logs
            .map(line => (line.split(' ')(2), 1))
            .reduceByKey((v1, v2) => v1 + v2)
        userreqs.print()
        ssc.start()
        ssc.awaitTermination()
    }
}
```

- Get a DStream (Discretized Stream) from a streaming data source, e.g. text
- from a socket

Streaming Example: Creating A DStream (2 Of 2)

- DStreams are the Spark Streaming analog of RDDs in regular Spark.
  - In Spark you usually start by creating an RDD from some file
  - In Spark Streaming you start by creating a DStream from some data source

Beneath The Hood

- Beneath the hood, Spark Streaming like Spark, uses Hadoop Input and Output Formats to give structure to the data being processed
  - SocketTextStream relies on Hadoop's TextInputFormat and LineRecordReader, meaning that the data from the stream is processed line by line
    - The resulting RDDs will contain one element for each ''line'' (newline delimited text) read from the input stream
    - Consequently, you can process the stream just like you would any line-delimited file, as you will see in the next slide

Streaming Example: DStream Transformations (1 Of 2)

```
import org.apache.spark.streaming.{Seconds, StreamingContext}
import org.apache.spark.streaming.StreamingContext._
import org.apache.spark.SparkContext._
import org.apache.spark.SparkConf
object StreamingRequestCount {
    def main(args: Array[String]) {
        if (args.length < 2) {
            System.err.println("Usage: StreamingRequestCount <host> <port>")
            System.exit(1)
        }
        val host = args(0)
        val port = args(1).toInt
        val ssc = new StreamingContext(new SparkConf(), Seconds(2))
        val logs = ssc.socketTextStream(host, port)
```

- DStream operations are applied to each batch RDD in the stream
- Similar to RDD operations - filter, map, reduceByKey, join, etc.

```
        userreqs.print()
        ssc.start()
        ssc.awaitTermination()
    }
}
```

Streaming Example: DStream Transformations (2 Of 2)

- Here we perform two operations (frequency counting) on the base DStream, resulting in two child DStreams
  - DStream lineages work just like RDD lineages
- The only difference between this and the "count hits by user id" example covered earlier is that we are performing these operations on a DStream rather than on an RDD

Streaming Example: DStream Result Output (1 Of 2)

```
import org.apache.spark.streaming.{Seconds, StreamingContext}
import org.apache.spark.streaming.StreamingContext._
import org.apache.spark.SparkContext._
import org.apache.spark.SparkConf
object StreamingRequestCount {
    def main(args: Array[String]) {
        if (args.length < 2) {
            System.err.println("Usage: StreamingRequestCount <host> <port>")
            System.exit(1)
        }
        val host = args(0)
        val port = args(1).toInt
        val ssc = new StreamingContext(new SparkConf(), Seconds(2))
        val logs = ssc.socketTextStream(host, port)
        val userreqs = logs
        userreqs.print()
        ssc.start()
        ssc.awaitTermination()
    }
}
```

- Print out the first 10 elements of each RDD

Streaming Example: DStream Result Output (2 Of 2)

- The print function is a convenience function on DStreams to display the first 10 elements of each batch of data
  - It doesn't do any formatting, but it does display the time of each batch

Streaming Example: Starting The Streams (1 Of 2)

```
import org.apache.spark.streaming.{Seconds, StreamingContext}
import org.apache.spark.streaming.StreamingContext._
import org.apache.spark.SparkContext._
import org.apache.spark.SparkConf
object StreamingRequestCount {
    def main(args: Array[String]) {
        if (args.length < 2) {
            System.err.println("Usage: StreamingRequestCount <host> <port>")
            System.exit(1)
        }
        val host = args(0)
            .map(line => (line.split(' ')(2), 1))
            .reduceByKey((v1, v2) => v1 + v2)
        userreqs.print()
        ssc.start()
        ssc.awaitTermination()
    }
}
```

- start: Starts the execution of all DStreams
- awaitTermination: Waits for all background threads to complete before
- ending the main thread

Streaming Example: Starting The Streams (2 Of 2)

- The actual streaming doesn't start until the start function is invoked on the StreamingContext
- At that point, the DStream connects to its data source (a network port in this example) and starts reading data from the data source, storing the data in RDDs
- From that point on, it runs indefinitely, creating new RDDs and performing the specified operations on those RDDs every 2 seconds (in this particular example)
- The StreamingContext stop() method provides graceful shutdown capabilities
  - e.g. Terminating an application without losing messages that have already arrived

Streaming Example: A Brief Aside

- Note that this example requires 2 executor threads
  - One thread for the socket
  - One thread for the processing
  - So if you run locally, your virtual machine will have to be configured to run with 2+ cores to avoid hanging
- Note also that if you run this example interactively, (from the Spark shell) you would have to change the statement initializing the Streaming Context from

```
val ssc = new StreamingContext(new SparkConf(), Seconds(2))
```

- to avoid a runtime error

```
val ssc = new StreamingContext(sc, Seconds(2))
```

Streaming Example: Streaming Request Count Recap

```
import org.apache.spark.streaming.{Seconds, StreamingContext}
import org.apache.spark.streaming.StreamingContext._
import org.apache.spark.SparkContext._
import org.apache.spark.SparkConf
object StreamingRequestCount {
    def main(args: Array[String]) {
        if (args.length < 2) {
            System.err.println("Usage: StreamingRequestCount <host> <port>")
            System.exit(1)
        }
        val host = args(0)
        val port = args(1).toInt
        val ssc = new StreamingContext(new SparkConf(), Seconds(2))
        val logs = ssc.socketTextStream(host, port)
        val userreqs = logs
             .map(line => (line.split(' ')(2), 1))
             .reduceByKey((v1, v2) => v1 + v2)
        userreqs.print()
        ssc.start()
        ssc.awaitTermination()
    }
}
```

Streaming Example: Output (1 Of 3)

```
-------------------------------------------
Time: 1401219545000 ms
-------------------------------------------
(23713,2)
(53,2)
(24433,2)
(127,2)
(93,2)
...
```

Starts 2 seconds after ssc.start()

Streaming Example: Output (2 Of 3)

```
-------------------------------------------
Time: 1401219545000 ms
-------------------------------------------
(23713,2)
(53,2)
(24433,2)
(127,2)
(93,2)
...
-------------------------------------------
Time: 1401219547000 ms
-------------------------------------------
(42400,2)
(24996,2)
(97464,2)
(161,2)
(6011,2)
...
```

2 seconds later…

Streaming Example: Output (3 Of 3)

```
-------------------------------------------
Time: 1401219545000 ms
-------------------------------------------
(23713,2)
(53,2)
(24433,2)
(127,2)
(93,2)
...
-------------------------------------------
Time: 1401219547000 ms
-------------------------------------------
(42400,2)
(24996,2)
(97464,2)
(161,2)
(6011,2)
...
-------------------------------------------
Time: 1401219549000 ms
-------------------------------------------
(44390,2)
(48712,2)
(165,2)
(465,2)
(120,2)
...
```

2 seconds later…

Continues until termination…

Streaming Demo

- Reminder: This example requires 2 executor threads
  - One thread for the socket
  - One thread for the processing
  - So if you run locally, your Spark master will have to be configured to run with 2 threads (or allocate 2 cores to the Virtual Machine) to avoid hanging

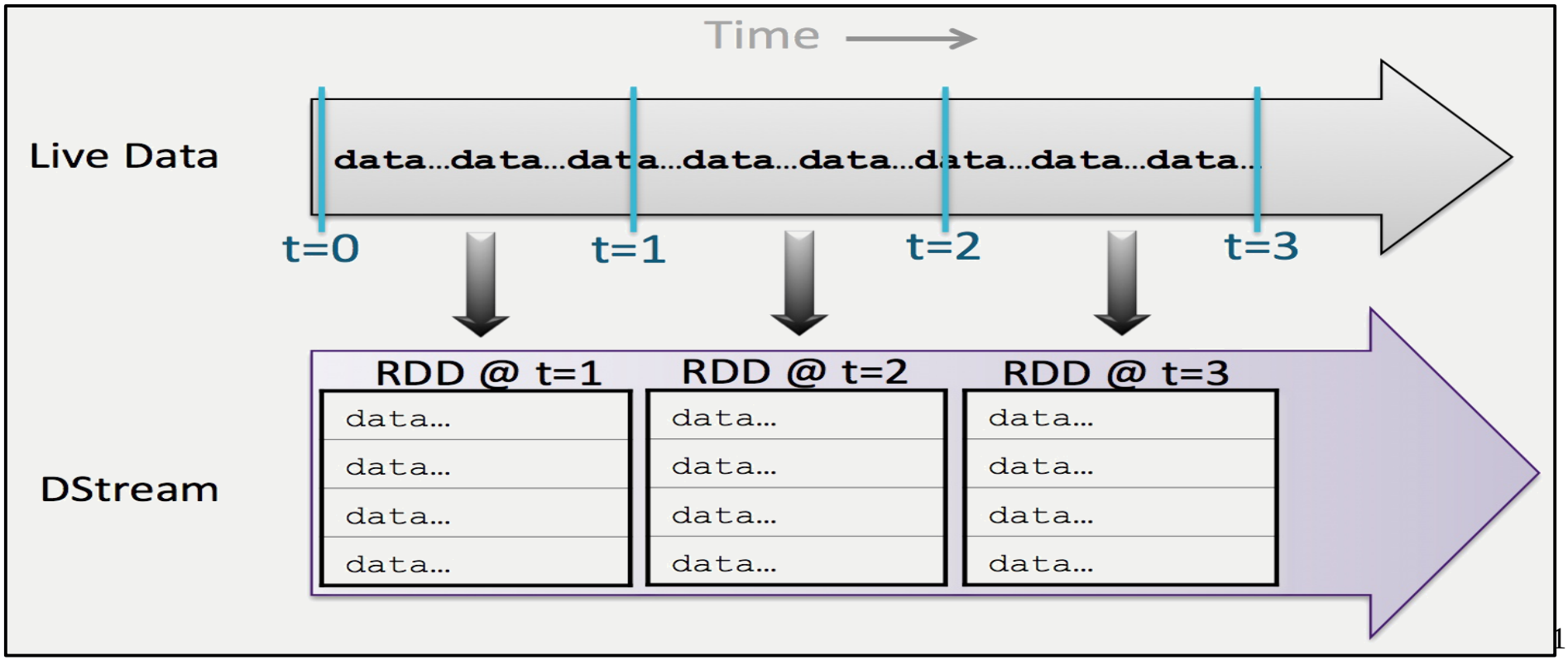
```
sh> cd ~/training_materials/sparkdev/examples
sh> gedit streamtest.py
sh> cd ~/training_materials/sparkdev/projects/streamingrequestcount/src/main/scala
sh> gedit StreamingRequestCount.scala
# Comment out Examples 2 - 4 individually, up to, but not including ssc.start().
# Optional
# Replace: userreqs.saveAsTextFile("streamreq/reqcounts")
# With: userreqs.print()
sh> cd ~/training_materials/sparkdev/examples
sh> python streamtest.py localhost 1234 50 ~/training_materials/sparkdev/data/weblogs/* &
# Enter the following commands from a separate terminal.
sh> cd ~/training_materials/sparkdev/projects/streamingrequestcount
sh> mvn package
sh> spark-submit --class StreamingRequestCount target/streamreqcount-1.0.jar localhost 1234
```

Chapter Topics

- Spark Streaming Overview
- Example: Streaming Request Count
- DStreams
- Hands-On Exercise: Exploring Spark Streaming
- State Operations
- Sliding Window Operations
- Developing Spark Streaming Applications
- Flume Vs. Storm Vs. Spark Streaming
- Summary
- Review
- References
- Hands-On Exercise: Writing A Spark Streaming Application

DStreams

- A DStream is a sequence of RDDs representing a data stream
  - Stands for Discretized Stream
  - Think of it as a continual bitstream



DStream Data Sources

- DStreams are defined for a given input stream (e.g., a Unix socket)
  - Created by the StreamingContext
    - ssc.socketTextStream (hostname, port)
  - Similar to how RDDs are created by the SparkContext
- Out-of-the-box data sources
  - Network
    - Sockets
    - Flume
    - Akka Actors (Next Page)
    - Kafka
    - ZeroMQ (Next Page)
    - Twitter
  - Files
    - Monitors an HDFS directory for new content (possibly delivered by one of the network out-of-the-box data sources)

Out-Of-Box DStream Data Sources

- Akka
  - http://akka.io/
  - A toolkit and runtime for building highly concurrent, distributed, fault tolerant, event-driven applications that run on the JVM
  - Designed to make it easier to write, test and maintain concurrent and distributed systems by allowing you to focus on the workflow, rather than on low level primitives like threads, locks and sockets
- ZeroMQ
  - http://zeromq.org
  - A high-performance asynchronous messaging library aimed at use in scalable distributed or concurrent applications, capable of running without a dedicated message broker
    - Similar to Spring Integration, Apache Camel, etc.
  - Employs a familiar socket-style API
- Apache Kafka
  - http://kafka.apache.org
  - A distributed, partitioned, replicated commit log service
  - Sometimes used as a lightweight alternative to Apache Flume

DStream Operations

- DStream operations are applied to every RDD in a stream
  - Executed once per duration
- Two types of DStream operations are supported
  - Transformations
    - Create a new DStream from an existing one
    - Like RDD transformations
  - Output operations
    - Write data (for example, to a file system, database, or console)
    - Like RDD Actions
- The DStream API exposes many common RDD operations
  - map, flatMap, reduceByKey
- Again, the payoff being that the programming model does not change as you move from batch oriented data sources to streaming data sources

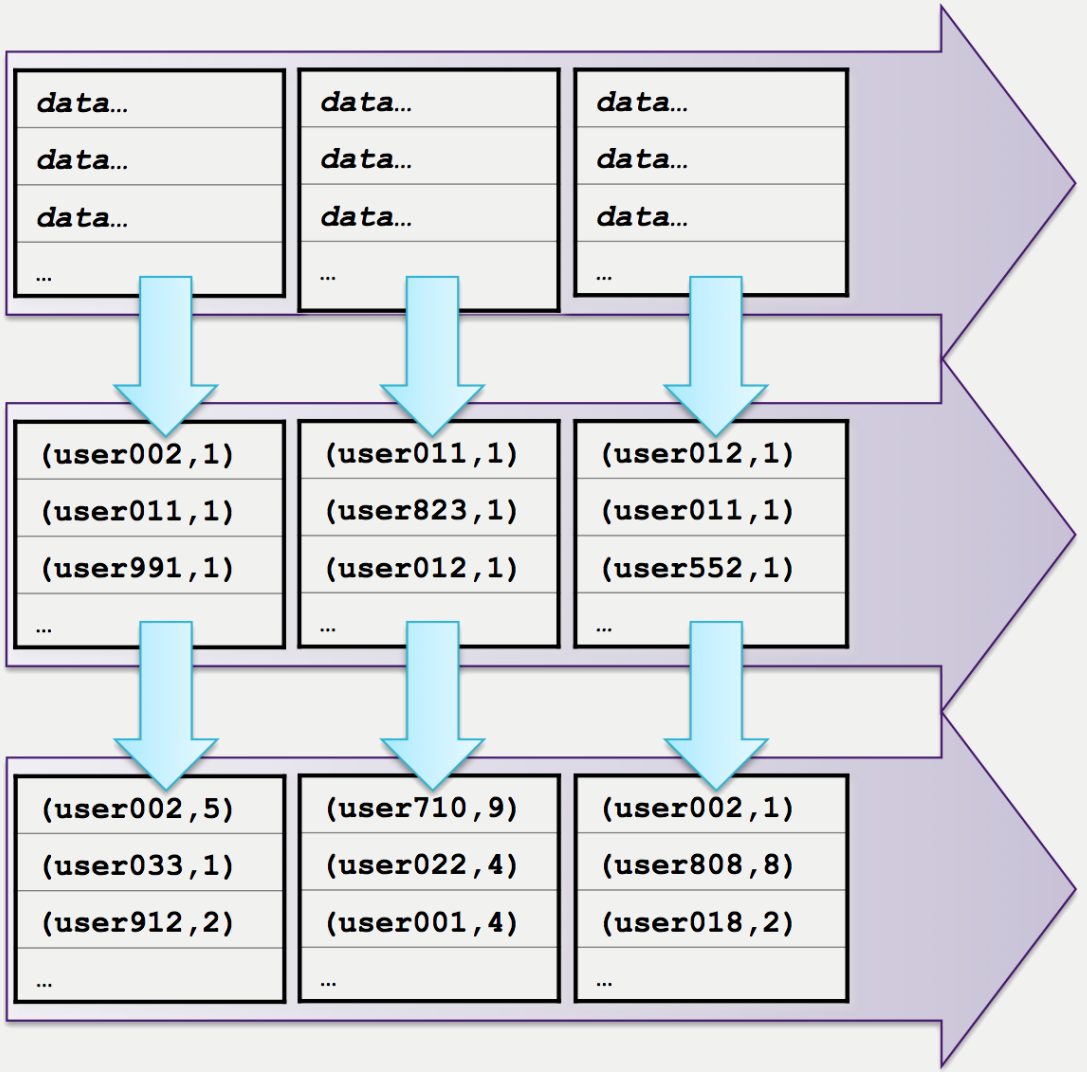
DStream Transformations (1 Of 2)

- Many RDD transformations are also available on DStreams
  - Regular transformations such as map, flatMap, filter
  - Pair transformations such as reduceByKey, groupByKey, join
- What if you want to do something else?
  - Not all RDD operations are available for DStreams
    - e.g. sortByKey, distinct, fold, ...
  - transform(function)
    - Creates a new DStream by executing a function on each RDD in the current DStream
      - Same idea as map

DStream Transformations (2 Of 2)

```
userreqs = logs.map(line =>
  (line.split(" ")(2),1))
```

```
reqcounts = userreqs.
  reduceByKey((x,y) => x+y)
```



DStream Output Operations

- http://spark.apache.org/docs/latest/api/scala/index.html#org.apache.spark.streaming.dstream.DStream
- Console output
  - print
    - Prints out the first 10 elements of each RDD
- File output
  - saveAsTextFiles
    - Saves data as text
  - saveAsObjectFiles
    - Saves as a Hadoop Sequence file
  - saveAsHadoopFiles
    - Saves data using standard Hadoop Input / Output formats
- Executing other functions
  - foreachRDD (function)
    - Performs a function on each RDD in the DStream
    - Parameter requirements:
      - RDD
        - The RDD to which to apply the function
      - Time
        - The time stamp of the RDD (optional)

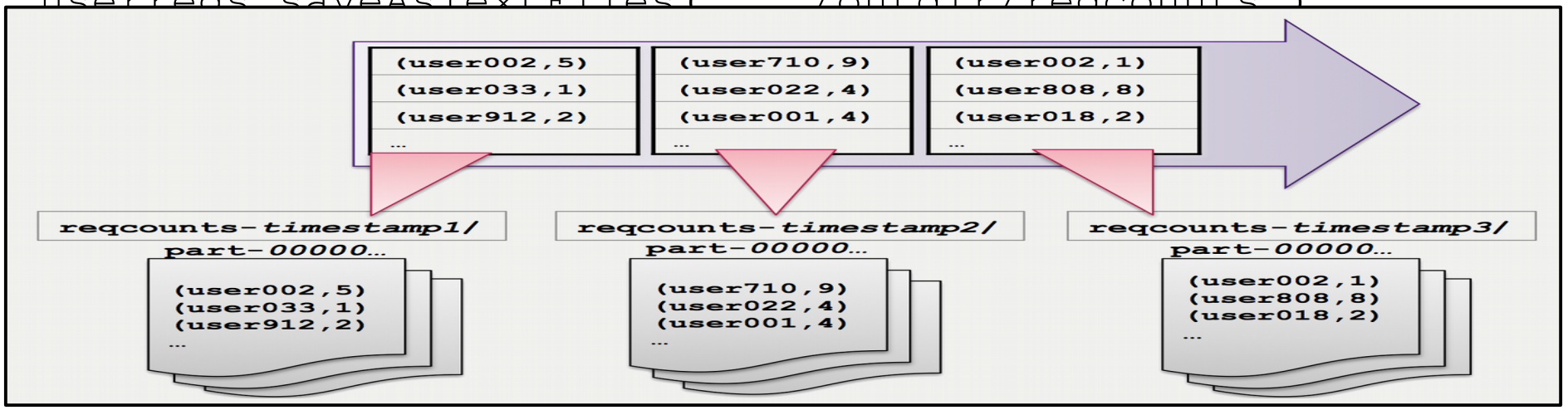
The forEach Operation

- foreachRDD is the fundamental output operation
  - All the others call this function
  - Applies the designated function to each RDD in the stream
- foreachRDD differs from other transformations in that it does not return a new DStream, it just applies a function to each RDD in the DStream being processed
  - Just like the way foreach differs from other RDD transformations
  - This might be useful, for example, if you wanted to do some processing on an entire RDD in a Dstream as a unit, such as saving it to a file

Saving DStream Results As Files

- In this example, output is written to a file rather than the display
  - Each batch written to its own directory whose name (by default) derives from a combination of the provided basename and a timestamp
  - Each file named "part-" followed by a 5+ digit partition identifier

```
val userreqs = logs
  .map(line => (line.split(' ')(2), 1))
  .reduceByKey((x, y) => x + y)
// userreqs.print()
userreqs.saveAsTextFiles("/outdir/reqcounts")
```



Case Study: Find Top Users (1 Of 3)

- The next example builds on to the previous example
  - In addition to saving the list of all users and their request counts, we will calculate and print out the users with the highest hit count for each interval
- To accomplish the latter task, we need to sort the users by the frequency with which they appear in each interval (count), so we:
  - Apply a map to swap the userId (key) and count (value), so that we can sort using count as the key
  - Invoke sortByKey on the count
- Unfortunately, sortByKey is not exposed in the DStream API
  - So we will call it indirectly, using transform

Case Study: Find Top Users (2 Of 3)

```
...
val userreqs = logs
    .map(line => (line.split(' ')(2), 1))
    .reduceByKey((v1, v2) => v1 + v2)
val sortedreqs = userreqs
    .map(pair => pair.swap)
    .transform(rdd => rdd.sortByKey(false))
    .foreachRDD(
    ((rdd, time) => {
        println("Top users @ " + time)
        rdd.take(5).foreach {
            (pair => printf("User: %s (%s)\n", pair._2, pair._1))}
        })}
ssc.start()
ssc.awaitTermination()
...
```

Transform each RDD: Swap userId (key) with count (value), then sort by count. False indicates that the sort is NOT in ascending order.

Case Study: Find Top Users (3 Of 3)

```
...
val userreqs = logs
    .map(line => (line.split(' ')(2), 1))
    .reduceByKey((v1, v2) => v1 + v2)
val sortedreqs = userreqs
    .map(pair => pair.swap)
    .transform(rdd => rdd.sortByKey(false))
sortedreqs.foreachRDD {
    ((rdd, time) => {
        println("Top users @ " + time)
        rdd.take(5).foreach {
        })}
ssc.start()
ssc.awaitTermination()
...
```

Print out the top 5 users as "User: userId(count)"

Case Study: Find Top Users - Output (1 Of 3)

```
Top users @ 1401219545000 ms
User: 16261 (8)
User: 22232 (7)
User: 66652 (4)
User: 21205 (2)
User: 24358 (2)
```

t = 0 (2 seconds after ssc.start())

Case Study: Find Top Users - Output (2 Of 3)

```
Top users @ 1401219545000 ms
User: 16261 (8)
User: 22232 (7)
User: 66652 (4)
User: 21205 (2)
User: 24358 (2)
Top users @ 1401219547000 ms
User: 53667 (4)
User: 35600 (4)
User: 62 (2)
User: 165 (2)
User: 40 (2)
Top users @ 1401219549000 ms
```

t = 1
(2 seconds later)

Case Study: Find Top Users - Output (3 Of 3)

```
Top users @ 1401219545000 ms
User: 16261 (8)
User: 22232 (7)
User: 66652 (4)
User: 21205 (2)
User: 24358 (2)
Top users @ 1401219547000 ms
User: 53667 (4)
User: 35600 (4)
User: 62 (2)
User: 165 (2)
User: 40 (2)
Top users @ 1401219549000 ms
User: 31 (12)
User: 6734 (10)
User: 14986 (10)
User: 72760 (2)
User: 65335 (2)
Top users @ 1401219551000 ms
…
```

t =2
(2 seconds later)

Continues until termination…

Streaming Demo

- Reminder: This example requires 2 executor threads
  - One thread for the socket
  - One thread for the processing
  - So if you run locally, your Spark Master will have to be configured to run with 2 threads (or allocate 2 cores to the VM) to avoid hanging

```
sh> cd ~/training_materials/sparkdev/examples
sh> gedit streamtest.py
sh> cd ~/training_materials/sparkdev/projects/streamingrequestcount/src/main/scala
sh> gedit StreamingRequestCount.scala
# Uncomment Example 2.
# Optional
# Write output of Example 1 to the file system.
sh> cd ~/training_materials/sparkdev/examples
sh> python streamtest.py localhost 1234 50 ~/training_materials/sparkdev/data/weblogs/* &
# Enter the following commands from a separate terminal.
sh> cd ~/training_materials/sparkdev/projects/streamingrequestcount
sh> mvn package
sh> spark-submit --class StreamingRequestCount target/streamreqcount-1.0.jar localhost 1234
```

Using Spark Streaming With The Spark Shell

- Spark Streaming is designed for batch applications, not interactive use
- However, the Spark Shell can be used for limited testing
  - Adding operations after the StreamingContext has been started (ssc.start()) is unsupported
    - In other words, all Dstream transformation operations must appear before the ssc.start() method invocation
  - Stopping and later restarting the StreamingContext is also unsupported
  - Remember you'll need to allocate at least two threads for this example, one for the socket the other for DStream processing

```
$ spark-shell --master local[2]
```

Chapter Topics

- Spark Streaming Overview
- Example: Streaming Request Count
- DStreams
- Hands-On Exercise: Exploring Spark Streaming
- State Operations
- Sliding Window Operations
- Developing Spark Streaming Applications
- Flume Vs. Storm Vs. Spark Streaming
- Summary
- Conclusion
- Review
- Hands-On Exercise: Writing A Spark Streaming Application

Hands-On Exercise

- Exploring Spark Streaming
  - Explore Spark Streaming using the Scala Spark Shell
  - Count words, use netcat to simulate a data stream
- Please refer to the Hands-On Exercise Manual

Chapter Topics

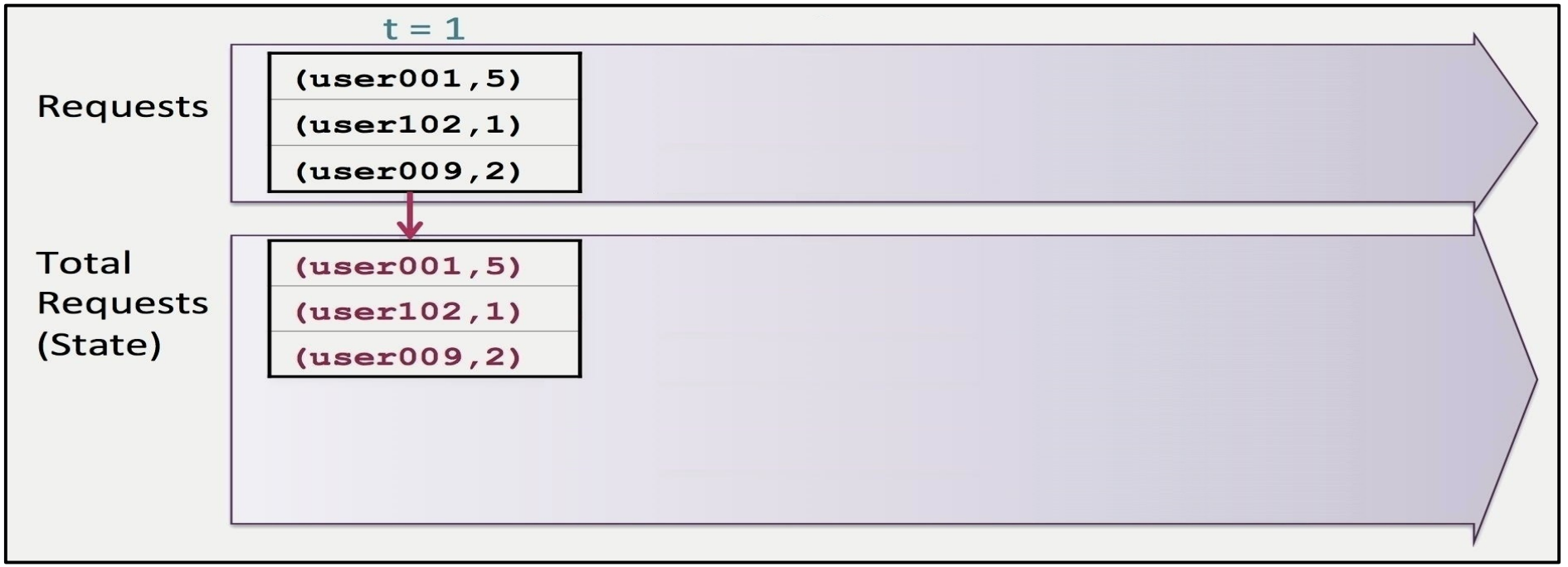
- Spark Streaming Overview
- Example: Streaming Request Count
- DStreams
- Hands-On Exercise: Exploring Spark Streaming
- State Operations
- Sliding Window Operations
- Developing Spark Streaming Applications
- Flume Vs. Storm Vs. Spark Streaming
- Summary
- Review
- References
- Hands-On Exercise: Writing A Spark Streaming Application

Food For Thought

- The examples you have seen thusfar have only processed one batch of data at a time, statelessly
  - Each batch processed independently of its predecessors
- But often you want to accumulate state across batches, updating the state each time a new batch of data is processed
  - e.g. In addition to counting requests by user every 2 seconds, you might want to count total requests by user, since you began streaming, as illustrated in the next few slides

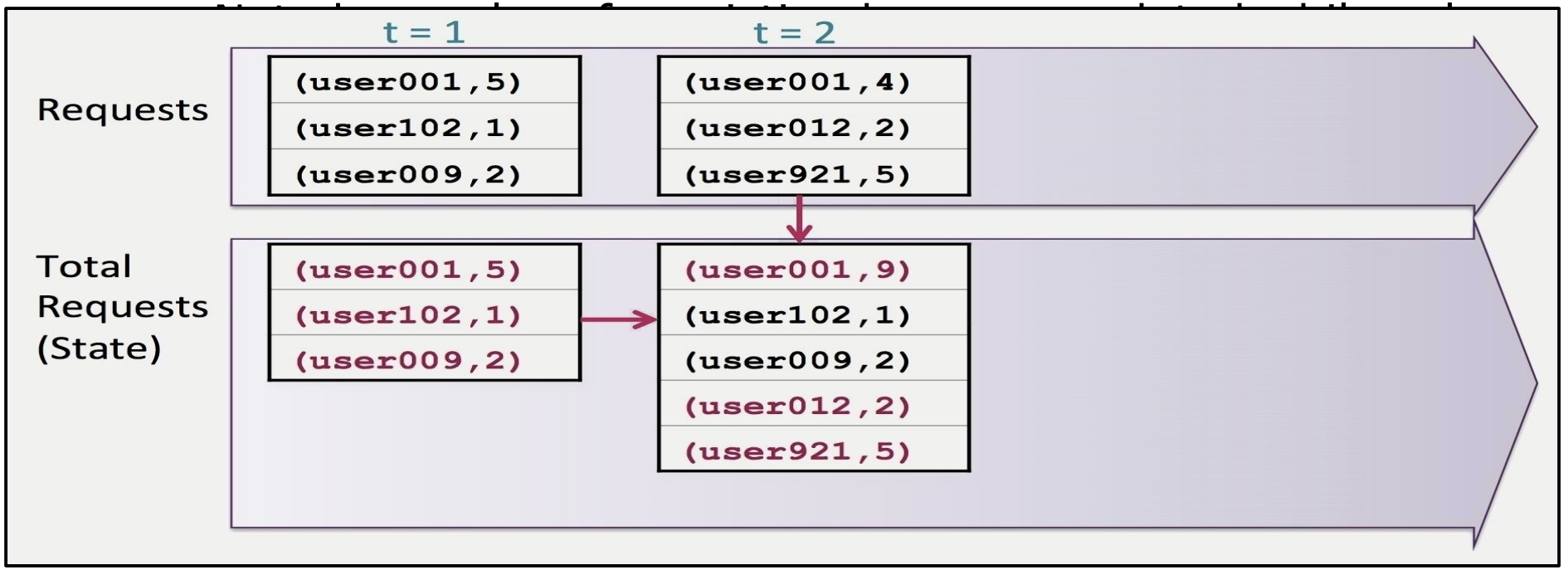
State DStreams (1 Of 3)

- Use the updateStateByKey function to create a state DStream
  - Example: Total request count by User ID



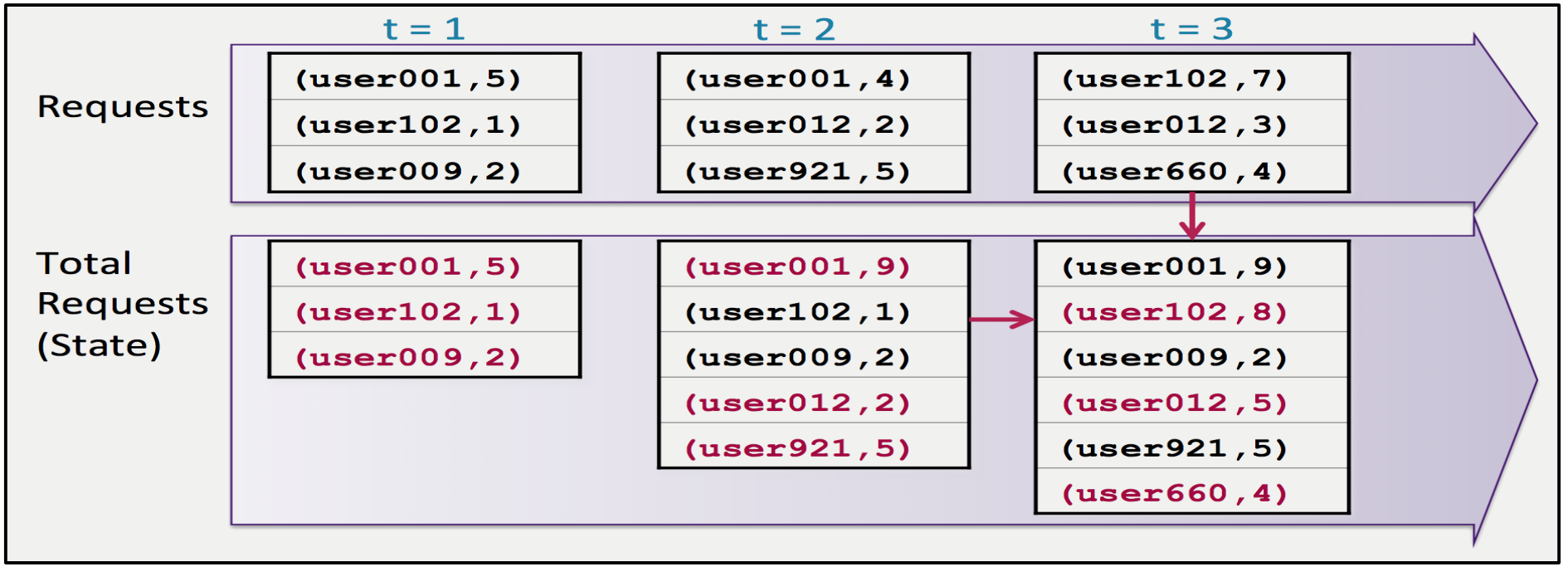
State DStreams (2 Of 3)

- Use the updateStateByKey function to create a state DStream
  - Example: Total request count by User ID



State DStreams (3 Of 3)

- Use the updateStateByKey function to create a state DStream
  - Example: Total request count by User ID



Example: Total User Request Count (1 Of 2)

```
...
val userreqs = logs
    .map(line => (line.split(" ")(2), 1))
    .reduceByKey((x, y) => x + y)
...
ssc.start()
ssc.awaitTermination()
...
```

Set the checkpoint directory to enable checkpointing. This is required to prevent infinite lineages. Failure to set a checkpoint directory when using the updateStateByKey function will generate an error

Example: Total User Request Count (2 Of 2)

```
...
val userreqs = logs
    .map(line => (line.split(" ")(2), 1))
ssc.checkpoint("checkpoints")
val totalUserreqs = userreqs.updateStateByKey(updateCount)
totalUserreqs.print()
ssc.start()
ssc.awaitTermination()
...
```

Compute a state DStream based on the previous states updated with the values from the current batch of request counts

Example: Total User Request Count Update Function (1 Of 3)

```
def updateCount = (newCounts: Seq[Int], state: Option[Int]) => {
    val newCount = newCounts.foldLeft(0)(_ + _)
    val previousCount = state.getOrElse(0)
```

Given an array of values associated with a single key from the new data stream (newCounts), and the value representing the current state associated with that key (state), sum the values from the new data stream (foldLeft), get the value for the current state (or 0 if the key represents a new user) (getOrElse) and generate a state for the key (Some)

- For those of you unfamiliar with Scala's Option class, it is a manifestation of the Null Object design pattern which allows us to represent Null or absent values as objects (without the need for conditional logic to differentiate null from non-null values)
  - This is what enables the use of the getOrElse function further down in the code

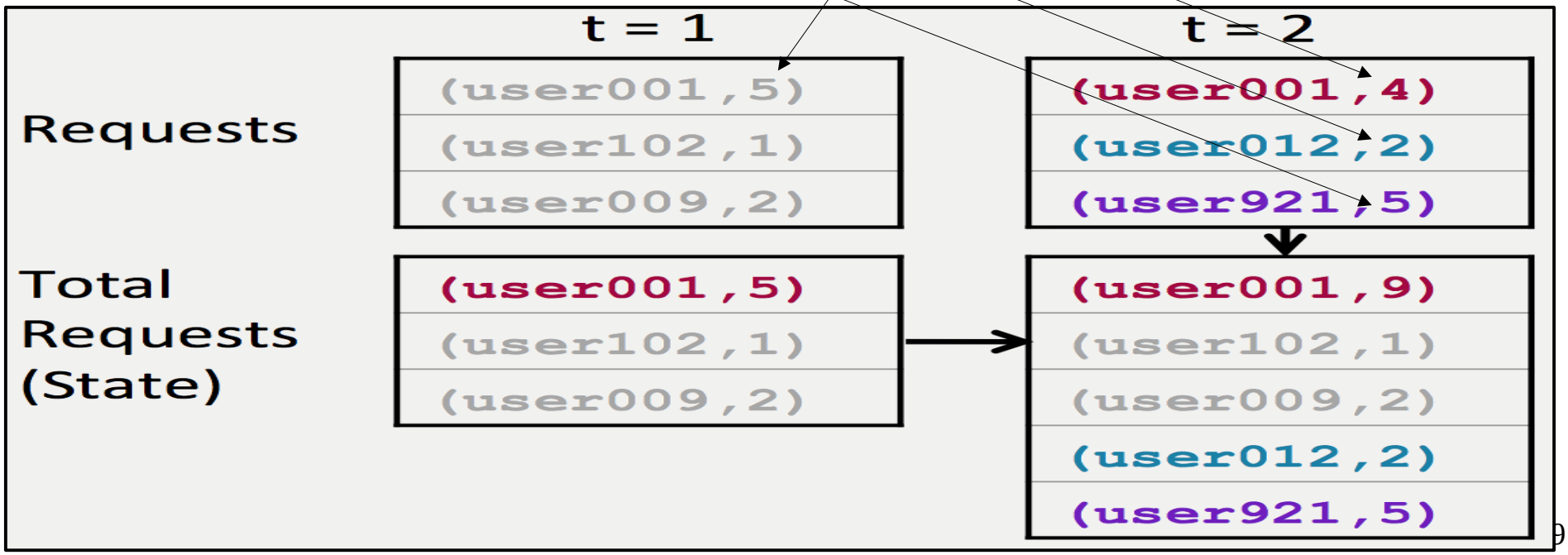
Example: Total User Request Count Update Function (2 Of 3)

- The function passed as an argument to updateStateByKey must conform to the following requirements
  - The first parameter must be an array
    - The reason this must be an array is that the function could have received many values
      - e.g. Had the previous operation not been a reduceByKey
  - In this case the array contains a single value representing the number of times a particular visitor visited our site during this batch
    - 4 for user001
  - The second parameter must be an array of Options representing the existing state (count) for this user, which is 5, from the t=1 state
    - Each member of this array is always either Some(value) or None
    - In this case, because user001 had already made 5 requests, the value passed in would be a Some object whose value is 5
    - But had user001 not made any requests from state t=1 (as is true for the next user, user012) the value would be a None object (whose value would be treated as 0 for the purpose of accumulating prior visits)

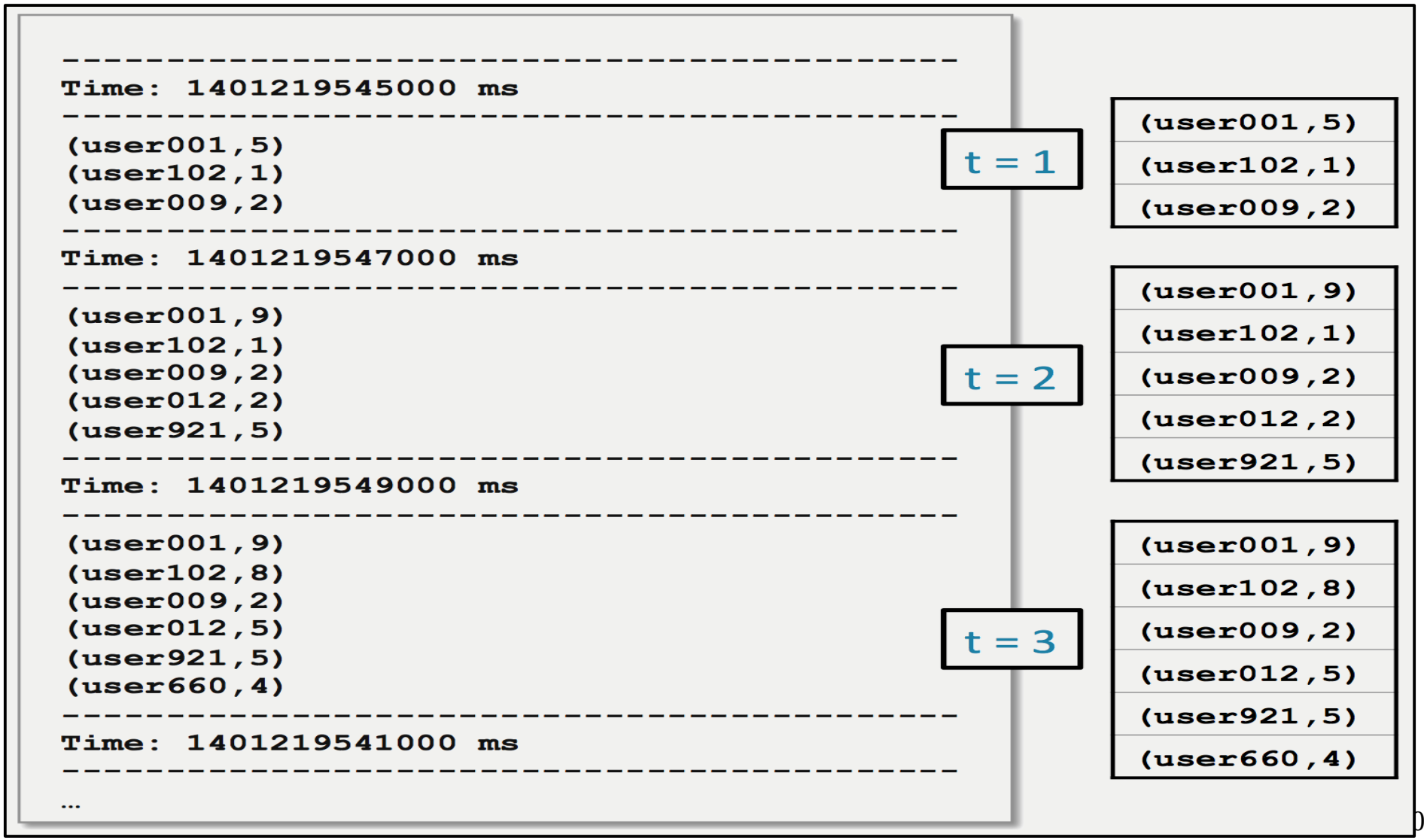
Example: Total User Request Count Update Function (3 Of 3)

- Example: t = 2

Key	Values	Result
user001	updateCount([4], Some[5])	9
user012	updateCount([2], None)	2
user921	updateCount([5], None)	5



Example: Maintaining State Output



Streaming Demo

- Reminder: This example requires 2 executor threads
  - One thread for the socket
  - One thread for the processing
  - So if you run locally, your Spark master will have to be configured to run with 2 threads (or allocate 2 cores to the Virtual Machine) to avoid hanging
- Note that since you are accumulating data over a large data set, it may take a while to see the cumulative effect of this implementation, unless you modify the code, command line arguments or data sets accordingly

```
sh> cd ~/training_materials/sparkdev/examples
sh> gedit streamtest.py
sh> cd ~/training_materials/sparkdev/projects/streamingrequestcount/src/main/scala
sh> gedit StreamingRequestCount.scala
# Uncomment Example 3.
sh> cd ~/training_materials/sparkdev/examples
sh> python streamtest.py localhost 1234 50 ~/training_materials/sparkdev/data/weblogs/* &
# Enter the following commands from a separate terminal.
sh> cd ~/training_materials/sparkdev/projects/streamingrequestcount
sh> mvn package
sh> spark-submit --class StreamingRequestCount target/streamreqcount-1.0.jar localhost 1234
```

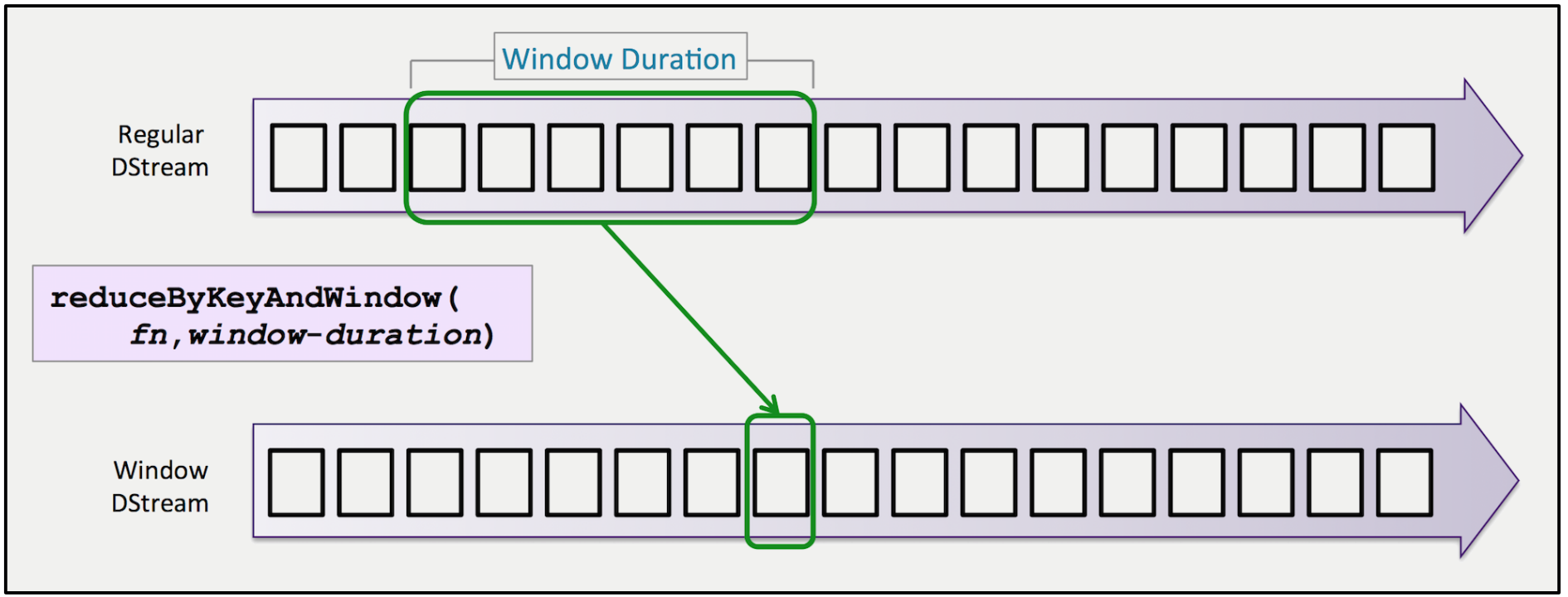
Chapter Topics

- Spark Streaming Overview
- Example: Streaming Request Count
- DStreams
- Hands-On Exercise: Exploring Spark Streaming
- State Operations
- Sliding Window Operations
- Developing Spark Streaming Applications
- Flume Vs. Storm Vs. Spark Streaming
- Summary
- Review
- References
- Hands-On Exercise: Writing A Spark Streaming Application

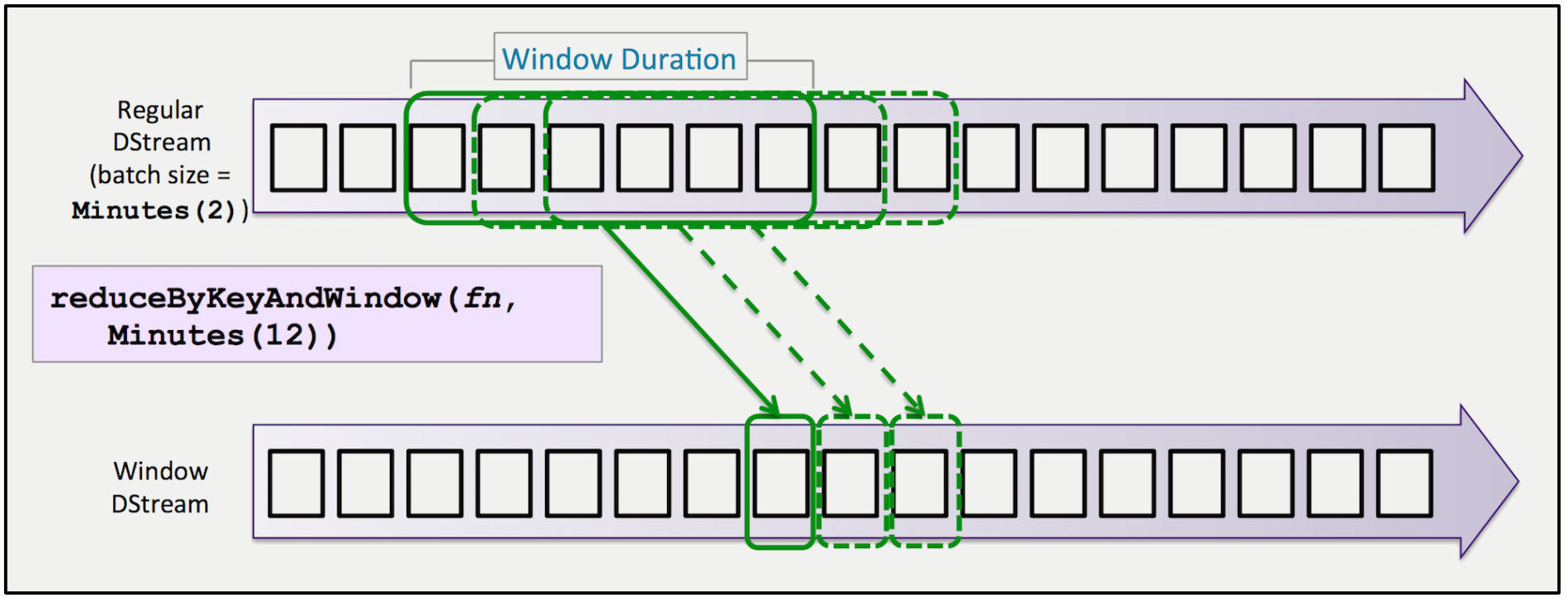
Food For Thought

- Thusfar you have seen two approaches for processing a stream of RDDs
  - Process each batch independently
    - Each RDD in the new stream depends only on a single RDD from the base stream
  - Accumulate state
    - Each RDD in the state stream depends on all RDDs from the base stream back to the beginning of the stream
- Now we introduce a third approach
  - Process a set of RDD batches into a single new RDD
    - Each RDD in the new stream depends on a specific set of RDDs from the base stream, those over a specified window of time
      - e.g. A rolling average
      - Allows us to answer questions like
        - What was the average price of a stock for the previous hour, computed every 5 minutes
        - What topics are trending in Twitter
    - This is known as a "Sliding Window"

Sliding Window Operations (1 Of 4)



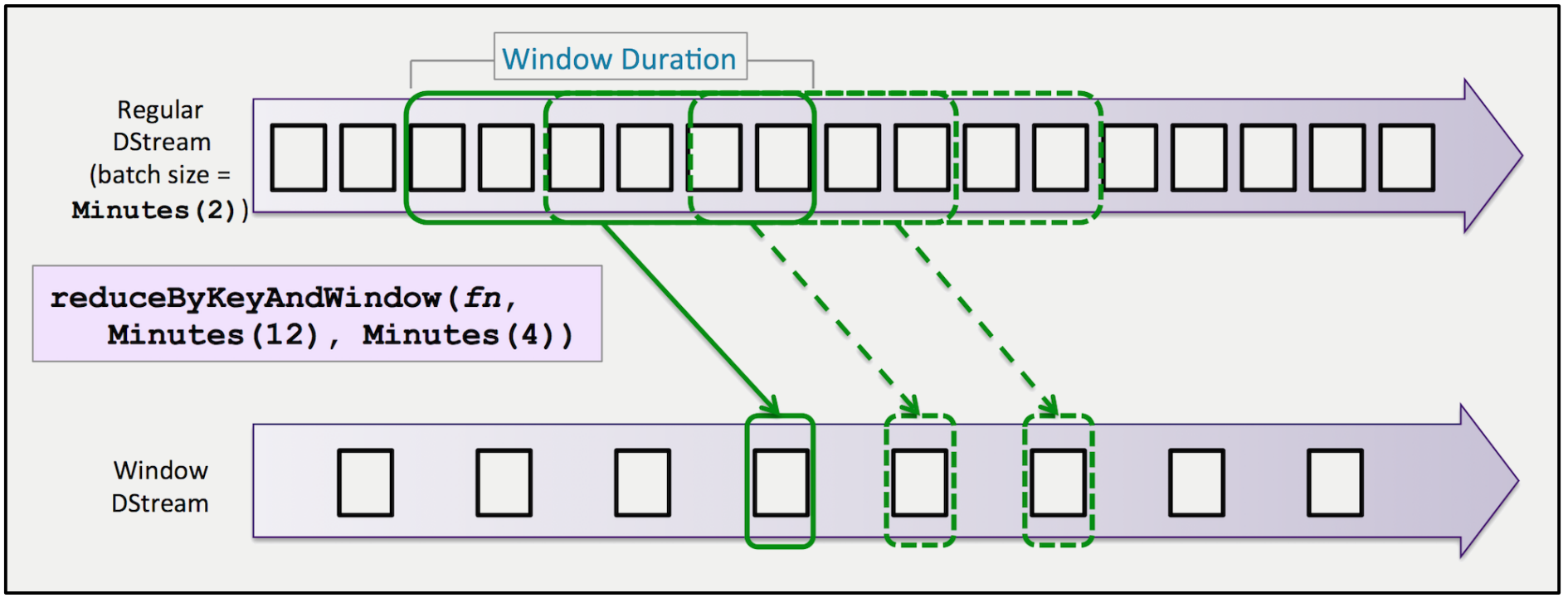
Sliding Window Operations (2 Of 4)



Sliding Window Operations (3 Of 4)

- What's happening in the example illustrated in the previous slide?
  - We are analyzing the previous 12 minutes worth of data every 2 minutes
  - At 10:00 am, we would analyze data generated from 9:48 – 10:00 am
  - At 10:02 am, we would analyze data generated from 9:50 – 10:02 am
  - At 10:04 am, we would analyze data generated from 9:52 – 10:04 am, etc ...

Sliding Window Operations (4 Of 4)



Things To Keep In Mind As You Determine The Parameters Of Your Analysis

- The window duration must be a multiple of the batch duration
- The minimum duration Spark can handle is 500 ms
- Make sure that the processing time for the window does not exceed the batch duration or you will end up falling behind as the application continues to execute
- Bottom Line: Think carefully about your batch size

Example: Count And Sort User Requests By Window (1 Of 2)

- This code is almost identical to the earlier example in which we computed the top user requests without windowing
  - The only difference is that we have replaced reduceByKey with reduceByKeyAndWindow to process multiple RDDs at a time, rather than just one

```
...
val reqCountsByWindow = logs
    .map(line => line.split(' ')(2), 1))
    .reduceByKeyAndWindow((x: Int, y: Int) => x + y,
Every 30 seconds, count requests by user for the previous 5 minutes
val topReqsByWindow = reqCountsByWindow
    .map(pair => pair.swap)
    .transform(rdd => rdd.sortByKey(false))
topReqsByWindow.map(pair => pair.swap).print()
ssc.start()
ssc.awaitTermination()
...
```

Example: Count And Sort User Requests By Window (2 Of 2)

```
...
val ssc = new StreamingContext(new SparkConf(), Seconds(2))
val logs = ssc.socketTextStream(host, port)
...
val reqCountsByWindow = logs
    .reduceByKeyAndWindow((x: Int, y: Int) => x + y,
        Minutes(5), Seconds(30))
val topReqsByWindow = reqCountsByWindow
    .map(pair => pair.swap)
    .transform(rdd => rdd.sortByKey(false))
topReqsByWindow.map(pair => pair.swap).print()
ssc.start()
ssc.awaitTermination()
...
```

Sort and print the top users for every RDD (every 30 seconds)

Streaming Demo

- Reminder: This example requires 2 executor threads
  - One thread for the socket
  - One thread for the processing
  - So if you run locally, your Spark master will have to be configured to run with 2 threads (or allocate 2 cores to the Virtual Machine) to avoid hanging
- Note that since you are accumulating data over a large data set, it may take a while to see the cumulative effect of this implementation, unless you modify the code, command line arguments or data sets accordingly

```
sh> cd ~/training_materials/sparkdev/examples
sh> gedit streamtest.py
sh> cd ~/training_materials/sparkdev/projects/streamingrequestcount/src/main/scala
sh> gedit StreamingRequestCount.scala
# Uncomment Example 4.
sh> cd ~/training_materials/sparkdev/examples
sh> python streamtest.py localhost 1234 50 ~/training_materials/sparkdev/data/weblogs/* &
# Enter the following commands from a separate terminal.
sh> cd ~/training_materials/sparkdev/projects/streamingrequestcount
sh> mvn package
sh> spark-submit --class StreamingRequestCount target/streamreqcount-1.0.jar localhost 1234
```

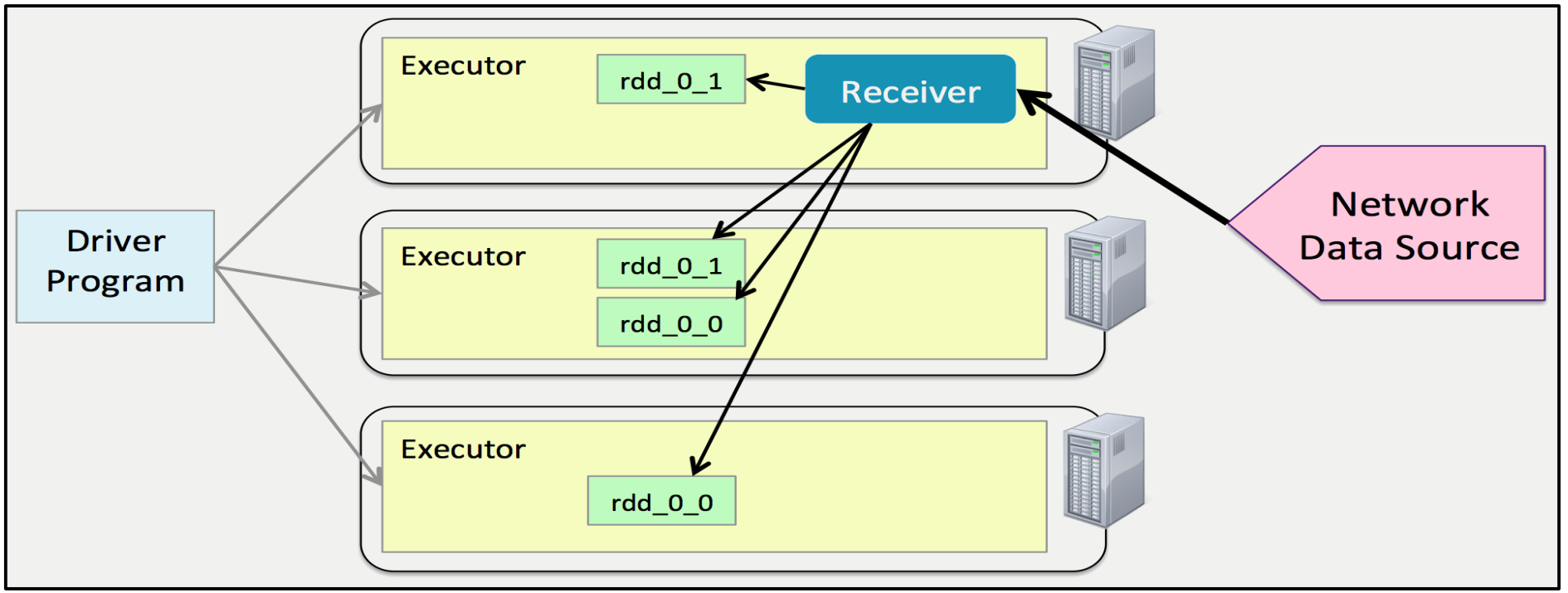
Chapter Topics

- Spark Streaming Overview
- Example: Streaming Request Count
- DStreams
- Hands-On Exercise: Exploring Spark Streaming
- State Operations
- Sliding Window Operations
- Developing Spark Streaming Applications
- Flume Vs. Storm Vs. Spark Streaming
- Summary
- Review
- References
- Hands-On Exercise: Writing A Spark Streaming Application

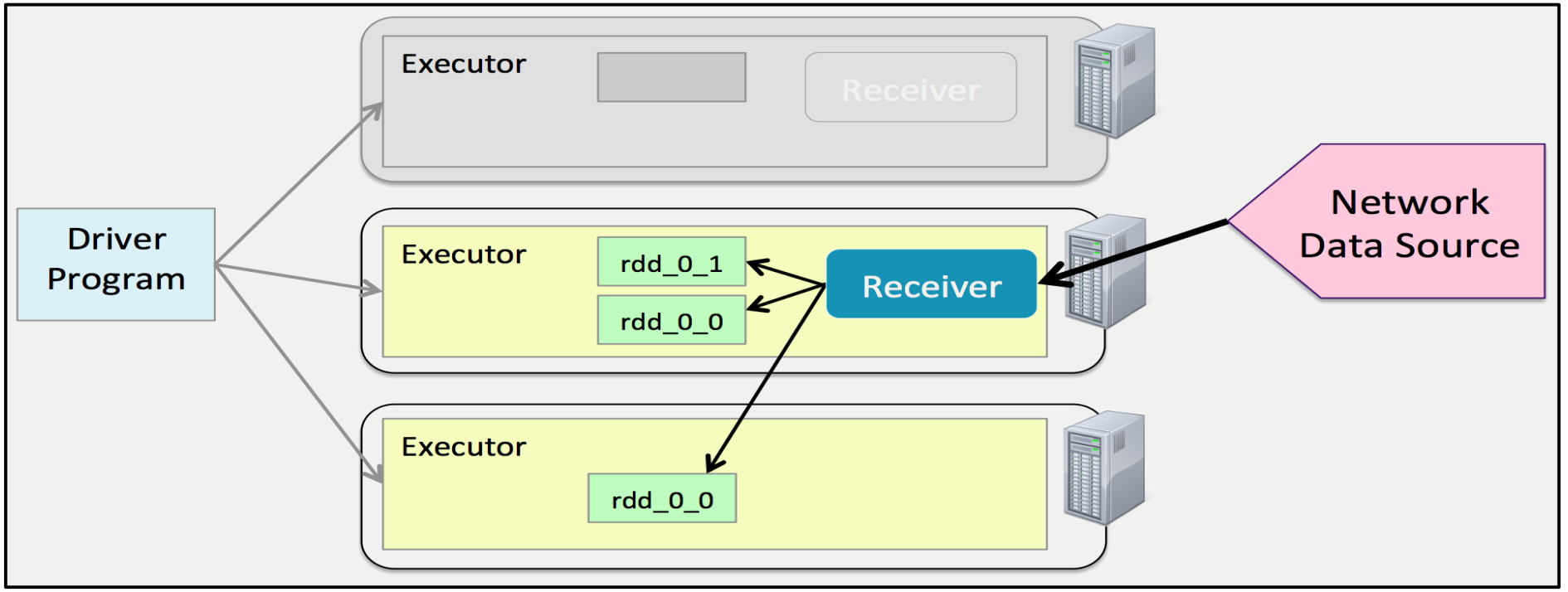
Special Considerations For Streaming Applications

- Spark Streaming applications are by definition long-running
  - Long running application require some different approaches than typical Spark applications
- Metadata accumulates over time
  - Use checkpointing to trim RDD lineage data
  - Required for state operations (but not windowed operations that use inverse functions)
    - There is no need to checkpoint windowed operations that use inverse functions (add new values, then subtract old values) (to improve performance) because there really are no long RDD dependencies created for these operations
      - Your data for any sliding window comes from that window, its predecessor and successor only
  - Enable by setting the checkpoint directory:
    - ssc.checkpoint (directory)
  - For stateful transformations, the default interval for setting the checkpoint should be a multiple of the batch interval, at least 10 seconds in length, as excessive checkpointing degrades performance
    - Set this value by invoking dstream.checkpoint(checkpointInterval)
    - A checkpoint interval 5 - 10 times the sliding interval of a DStream is generally considered a good setting
- Monitoring
  - The StreamingListener API lets you collect statistics

Spark Fault Tolerance (1 Of 2)



Spark Fault Tolerance (2 Of 2)



What If The Driver Fails? (1 Of 2)

- Spark Streaming allows a streaming computation to be resumed even after the failure of the driver node, by periodically checkpointing metadata for the DStreams that were setup through the StreamingContext to HDFS
- Upon failure of the driver, the lost StreamingContext can be recovered and restarted, assuming the driver is written to support such failover
  - When the program is being started for the first time, it will create a new StreamingContext, set up all the streams and then call start()
  - When the program is being restarted after failure, it will re-create a StreamingContext from the checkpoint data in the checkpoint directory

What If The Driver Fails? (2 Of 2)

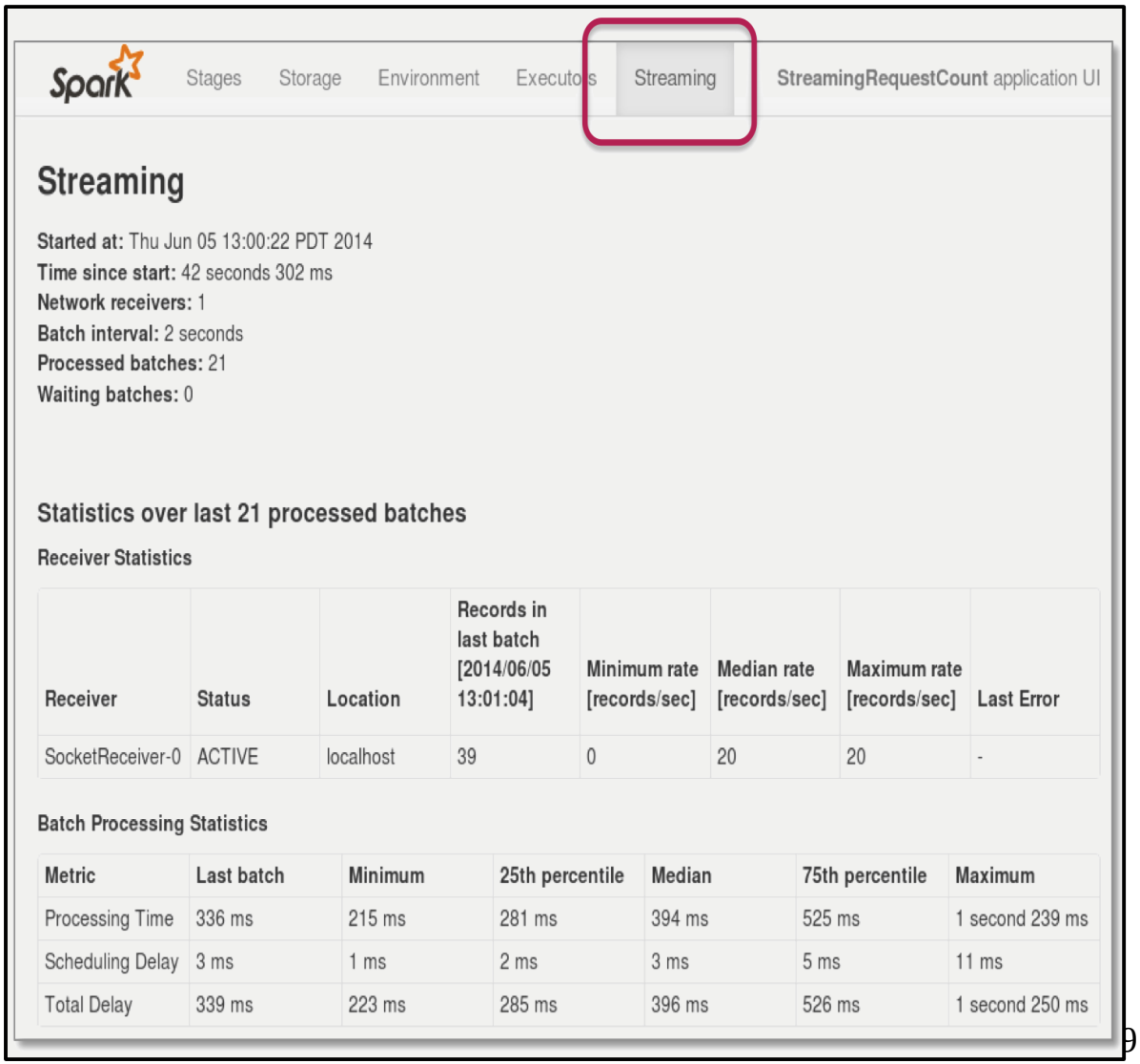
```
// Function to create and setup a new StreamingContext
def functionToCreateContext(): StreamingContext = {
    val ssc = new StreamingContext(...)   // New context.
    val lines = ssc.socketTextStream(...) // Create DStreams.
    ...
    ssc.checkpoint(checkpointDirectory)   // Set checkpoint
    ssc                                   // directory.
}
// Get the StreamingContext from the checkpoint
// data or create a new one.
val context = StreamingContext.getOrCreate
    (checkpointDirectory, functionToCreateContext _)
// Do any additional setup on context that needs to be done,
// irrespective of whether it is being started or restarted.
...
// Start the context
context.start()
context.awaitTermination()
```

Building And Running Spark Streaming Applications

- Building Spark Streaming Applications
  - Link with the main Spark Streaming library (included with Spark)
  - Link with additional Spark Streaming libraries if necessary
    - e.g,. Kafka, Flume, Twitter
- Running Spark Streaming Applications
  - Remember to use 2+ threads if running locally
    - One for streaming
    - One for processing

The Spark Streaming Application UI

- The Streaming tab in the Spark Application UI provides basic metrics about the application



Twitter Using Spark Streaming

- Our courseware provides an example of using Spark Streaming with Twitter channel adapters
  - This program generates output summarizing tweets every 10 seconds, and then again summarizing every 60 seconds
- This program requires 2 executor threads
  - One thread for the socket
  - One thread for the processing
  - So if you run locally, your Spark master will have to be configured to run with 2 threads (or allocate 2 cores to the Virtual Machine) to avoid hanging

```
> cd /home/training/training_materials/sparkdev/examples
> # Read the source file.
> gedit TwitterPopularTags.scala
[Ctrl][Q]
> cd /usr/lib/spark
# Run the demo
> ./bin/run-example streaming.TwitterPopularTags
TwitterPopularTags <consumer key> <consumer secret>
<access token> <access token secret> [<filters>]
> # Note the need for Twitter Security credentials.
> # Run the demo with your personal credentials and an optional filter.
> ./bin/run-example streaming.TwitterPopularTags ...
```

Chapter Topics

- Spark Streaming Overview
- Example: Streaming Request Count
- DStreams
- Hands-On Exercise: Exploring Spark Streaming
- State Operations
- Sliding Window Operations
- Developing Spark Streaming Applications
- Flume Vs. Storm Vs. Spark Streaming
- Summary
- Conclusion
- Review
- Hands-On Exercise: Writing A Spark Streaming Application

Their Intent Is Different

- Apache Flume is a tool for ingesting data, and any processing you do during ingest should be minimal
  - e.g. normalizing timestamps, format conversion, or dropping corrupt records (ETL)
- In contrast, both Storm and Spark Streaming are intended to do much more serious processing on data as it streams in (ELT)
  - e.g. analytics, machine learning, etc
  - Thus, Flume is not really a competitor to either one in this regard
- This begs the question, "How do Storm and Spark Streaming compare to one another?"
  - In general, we feel that Spark Streaming is the preferred solution for reasons summarized in the next few slides

Long Term Investment

- Although HortonWorks and MapR distributions have Storm while Cloudera s CDH does not, all three distributions have (or soon will have) Spark Streaming
  - Thus, Spark Streaming is actually more portable and a better long-term investment for your code

Consistent Programming Model / Learning Curve

- Storm is intended only for processing streaming data, while Spark offers both batch and stream processing with similar APIs and operational models
  - This is much easier than having to learn two totally separate frameworks

Idempotence Requirements

- By default, Storm guarantees that a message will be processed "at least once," thus making it unsuitable for anything that s not idempotent
  - e.g. financial transactions
  - Although this can be addressed with an additional component called Trident, but at a price in complexity and performance
  - https://github.com/nathanmarz/storm/wiki/Trident-tutorial

Maintenance

- With Spark Streaming, you will be able to take advantage of centralized security and resource management instead of having to set up and maintain Hadoop/Spark and Storm separately

APIs

- Spark Streaming provides better APIs to interface with data sets on Hadoop very easily
  - This can be much more onerous in Storm

Developer Support

- Spark has a more active developer community than Storm

Chapter Topics

- Spark Streaming Overview
- Example: Streaming Request Count
- DStreams
- Hands-On Exercise: Exploring Spark Streaming
- State Operations
- Sliding Window Operations
- Developing Spark Streaming Applications
- Flume Vs. Storm Vs. Spark Streaming
- Summary
- Review
- References
- Hands-On Exercise: Writing A Spark Streaming Application

Summary

- Spark Streaming is an add-on to core Spark to process real-time streaming data
- DStreams are "discretized streams" of streaming data, batched into RDDs by time interval
  - Operations applied to DStreams are applied to each RDD individually
  - Transformations produce new DStreams by applying a function to each RDD in the base DStream
- You can update state based on prior state
  - e.g. Total requests by user
- You can perform operations on "windows" of data
  - e.g. Number of logins in the last hour

Chapter Topics

- Spark Streaming Overview
- Example: Streaming Request Count
- DStreams
- Hands-On Exercise: Exploring Spark Streaming
- State Operations
- Sliding Window Operations
- Developing Spark Streaming Applications
- Flume Vs. Storm Vs. Spark Streaming
- Summary
- Review
- References
- Hands-On Exercise: Writing A Spark Streaming Application

Review

- What is the motivation for Spark Streaming?

Review Answer

- What is the motivation for Spark Streaming?
  - To be able to process a continuous flow of RDDs streaming into an application
  - Using the same basic programming model that we use in Spark
    - Otherwise, we might use Apache Storm

Review

- Spark Streaming is currently supported for what languages?

Review Answer

- Spark Streaming is currently supported for what languages?
  - Scala
  - Java
  - Python

Review

- What is the smallest batch duration recommended for use in Spark Streaming?

Review Answer

- What is the smallest batch duration recommended for use in Spark Streaming?
  - 1/2 second

Review

- Spark Streaming uses the same SparkContext as regular Spark applications. True or False?

Review Answer

- Spark Streaming uses the same SparkContext as regular Spark applications. True or False?
  - Spark relies on a special StreamingContext

Review

- The central abstraction in Spark Streaming is called a(n) _____. What is its role?

Review Answer

- The central abstraction in Spark Streaming is called a(n) DStream. What is its role?
  - Discretized Stream
  - A sequence of RDDs representing a data stream
  - The contribution of the DStream is that it provides a mechanism for processing a continual stream of data as a set of RDDs, leveraging the same programming model applicable to the rest of Spark

Review

- List 3 out-of-the-box data sources that Spark Streaming recognizes

Review Answer

- List 3 out-of-the-box data sources that Spark Streaming recognizes
  - Network
    - Sockets
    - Flume
    - Akka Actors
    - Kafka
    - ZeroMQ
    - Twitter
  - Files
    - Monitor an HDFS directory for new content (possibly delivered by one of the network out-of-the-box data sources)

Review

- To process the contents of a DStream just use the same methods you would use processing an RDD. True or False?

Review Answer

- To process the contents of a DStream just use the same methods you would use processing an RDD. True or False?
  - Remember, the Spark API works directly with RDDs, not DStreams, while the Spark Streaming API works with DStreams, not RDDs
    - While they expose similar methods, they work at different levels of abstraction
      - A DStream being a wrapper for a set of RDDs
  - So you use the DStream wrapper methods, which perform the same logical operations as the RDD methods, but applies them individually to each RDD in the DStream

Review

- What's the big idea behind a State DStream?

Review Answer

- What's the big idea behind a State DStream?
  - Allows you to maintain state across batches, updating variables like accumulators each time a new batch of data is processed

Review

- What's the big idea behind a Sliding Window?

Review Answer

- What's the big idea behind a Sliding Window?
  - To regulate room temperature ;-)
  - To process a flow by analyzing different portions of the flow over time
    - Rolling computations
      - What was the average price of a stock for the previous hour, computed every 5 minutes
      - What topics are trending in Twitter every 15 minutes

Chapter Topics

- Spark Streaming Overview
- Example: Streaming Request Count
- DStreams
- Hands-On Exercise: Exploring Spark Streaming
- State Operations
- Sliding Window Operations
- Developing Spark Streaming Applications
- Flume Vs. Storm Vs. Spark Streaming
- Summary
- Review
- References
- Hands-On Exercise: Writing A Spark Streaming Application

References

- https://spark.apache.org/docs/latest/streaming-programming-guide.html
  - Spark Streaming Programming Guide
- https://spark.apache.org/streaming/
  - Spark Streaming home page
- http://horicky.blogspot.com/2014/11/spark-streaming.html
  - A brief, simple example overview of Spark Streaming
- https://www.cs.duke.edu/~kmoses/cps516/dstream.html#spark-stream
  - Using Spark Streaming for data intensive processing

Chapter Topics

- Spark Streaming Overview
- Example: Streaming Request Count
- DStreams
- Hands-On Exercise: Exploring Spark Streaming
- State Operations
- Sliding Window Operations
- Developing Spark Streaming Applications
- Flume Vs. Storm Vs. Spark Streaming
- Summary
- Review
- References
- Hands-On Exercise: Writing A Spark Streaming Application

Hands-On Exercise

- Writing A Spark Streaming Application
  - Write a Spark Streaming application to process web logs using a Python script to simulate a data stream
- Please refer to the Hands-On Exercise Manual