Chapter 09

SPARK STREAMING

What is Spark Streaming?

- **Spark Streaming provides real time processing of stream data**
- **An extension of core Spark**
- **Currently supports Python, Scala and Java**

Why Spark Streaming?

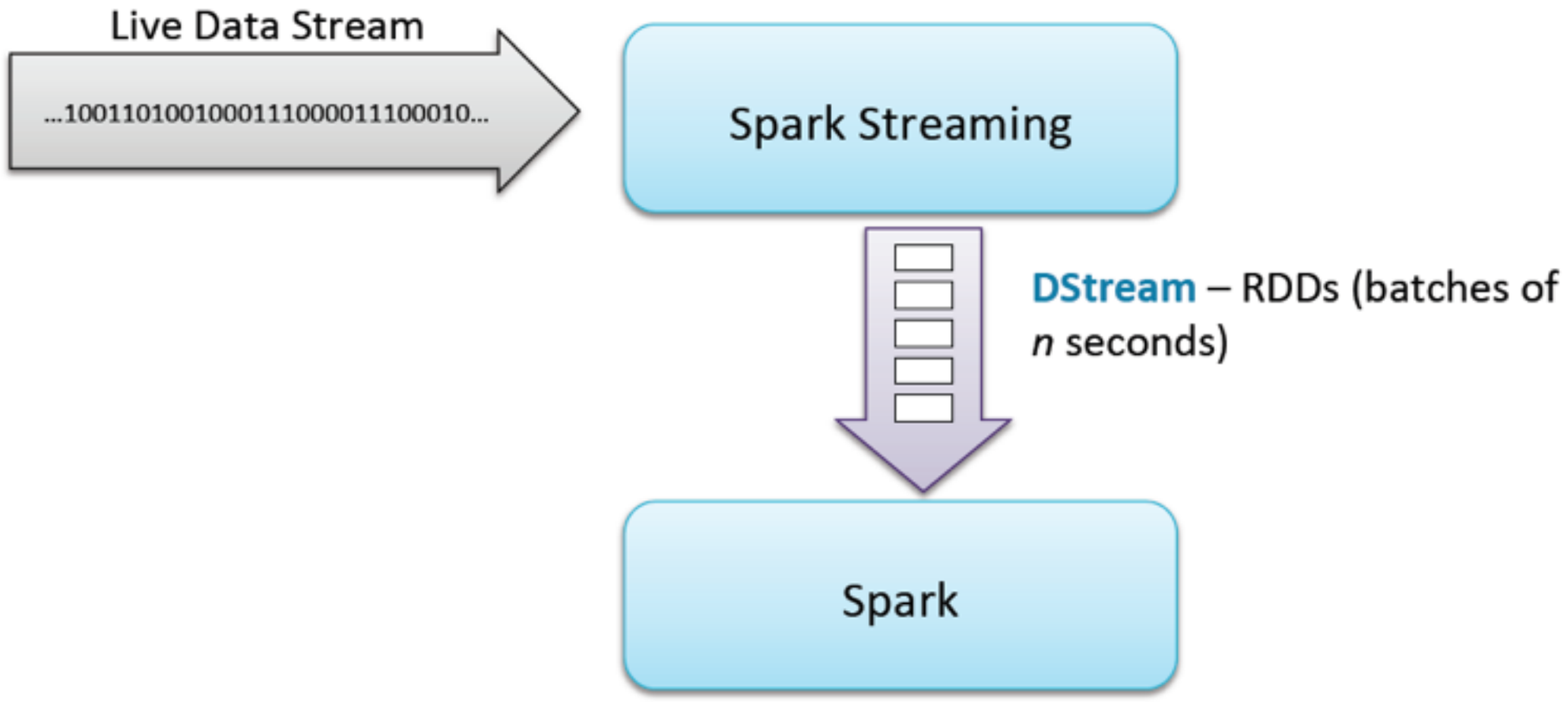
- **Many big data applications need to process large data streams in real time**
  - **Website monitoring**
  - **Fraud detection**
  - **Ad monetization**

Spark Streaming Features

- **Second scale latencies**
- **Scalability and efficient fault tolerance**
- **“Once and only once” processing**
- **Integrates batch and real-time processing**
- **Easy to develop**
  - **Uses Spark’s high level API**

Spark Streaming Overview

- **Divide up data stream into batches of n seconds**
- **Process each batch in Spark as an RDD**
- **Return results of RDD operations in batches**



Streaming Example: Streaming Request Count

```
object StreamingRequestCount {

  def main(args: Array[String]) {

    val ssc = new StreamingContext(new SparkConf(),Seconds(2))
    val mystream = ssc.socketTextStream(hostname, port)
    val userreqs = mystream
      .map(line => (line.split(" ")(2),1))
      .reduceByKey((x,y) => x+y)

    userreqs.print()

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

Streaming Example: Configuring StreamingContext

```
object StreamingRequestCount {

  def main(args: Array[String]) {

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

    use


    ssc
    ssc
  }
}
```

- 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 *batch duration* – instance of **`Milliseconds`**, **`Seconds`**, or **`Minutes`**
- Named **`ssc`** by convention

Streaming Example: Creating a DStream

```
object StreamingRequestCount {

  def main(args: Array[String]) {

    val ssc = new StreamingContext(new SparkConf(),Seconds(2))
    val logs = ssc.socketTextStream(hostname, port)
    val
    use

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

- Get a **DStream** (“**D**iscretized **Stream**”) from a streaming data source, e.g., text from a socket

Streaming Example: Dstream Transformations

```
object StreamingRequestCount {

  def main(args: Array[String]) {

    val ssc = new StreamingContext(new SparkConf(),Seconds(2))
    val logs = ssc.socketTextStream(hostname, port)
    val userreqs = logs
      .map(line => (line.split(" ")(2),1))
      .reduceByKey((x,y) => x+y)

    use

    ssc
    ssc
  }
}
```

- DStream operations are applied to each batch RDD in the stream
- Similar to RDD operations – **`filter`**, **`map`**, **`reduce`**, **`joinByKey`**, etc.

Streaming Example: Dstream Result Output

```
object StreamingRequestCount {

  def main(args: Array[String]) {

    val ssc = new StreamingContext(new SparkConf(),Seconds(2))
    val logs = ssc.socketTextStream(hostname, port)
    val userreqs = logs
      .map(line => (line.split(" ")(2),1))
      .reduceByKey((x,y) => x+y)

    userreqs.print()

    ss
    ss
  }
}
```

- Print out the first 10 elements of each RDD

Streaming Example: Starting the Streams

```
object StreamingRequestCount {

  def main(args: Array[String]) {

    val ssc = new StreamingContext(new SparkConf(),Seconds(2))
    val logs = ssc.socketTextStream(hostname, port)
    va
    us

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

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

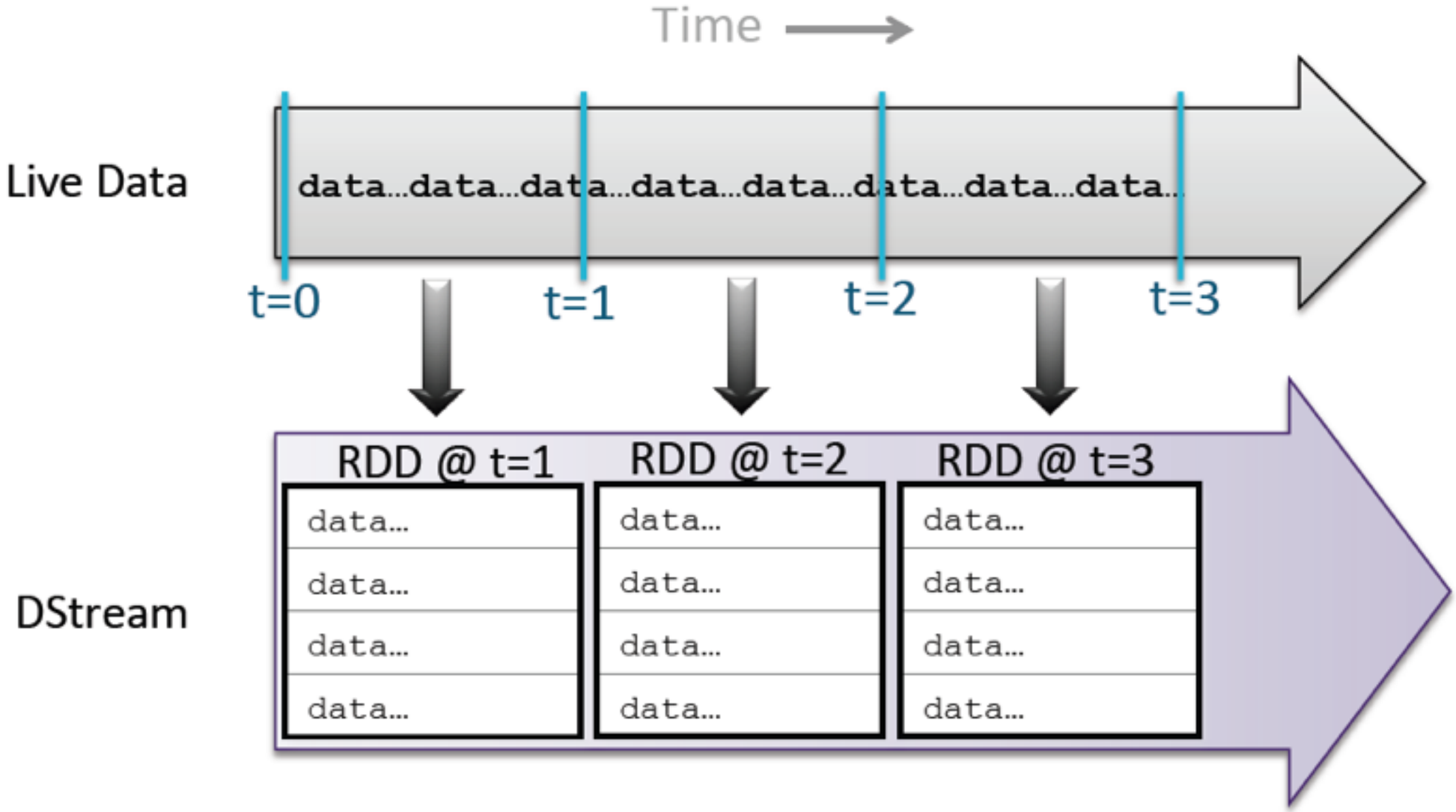
Streaming Example Output

```
-------------------------------------------
Time: 1401219545000 ms
-------------------------------------------
(23713,2)
(53,2)
(24444,2)
(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)
…
```

Continues until termination…

DStreams

- **A Dstream is a sequence of RDDs representing a data stream**
  - **"Discretized Stream"**



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
  - Other network sources, e.g., Flume, Akka Actors, Kaga, Twitter
  - Files
    - Monitors an HDFS directory for new content

Dstream Operations

- Dstream operations are applied to every RDD in the stream
  - Executed once per duration
- Two types of Dstream operations
  - Transformations
    - Create a new Dstream from an existing one
  - Output operations
    - Write data (for example, to a file system, database, or console)
    - Similar to RDD actions

Dstream Transformations

- Many RDD transformations are also available on DStreams
  - Regular transformations such as map, flatMap, filter
  - Pair transformations such as reduceByKey, groupByKey, joinByKey
- What if you want to do something else?
  - transform(function)
    - Creates a new Dstream by executing function on RDDs in the current DStream

Dstream Transformations

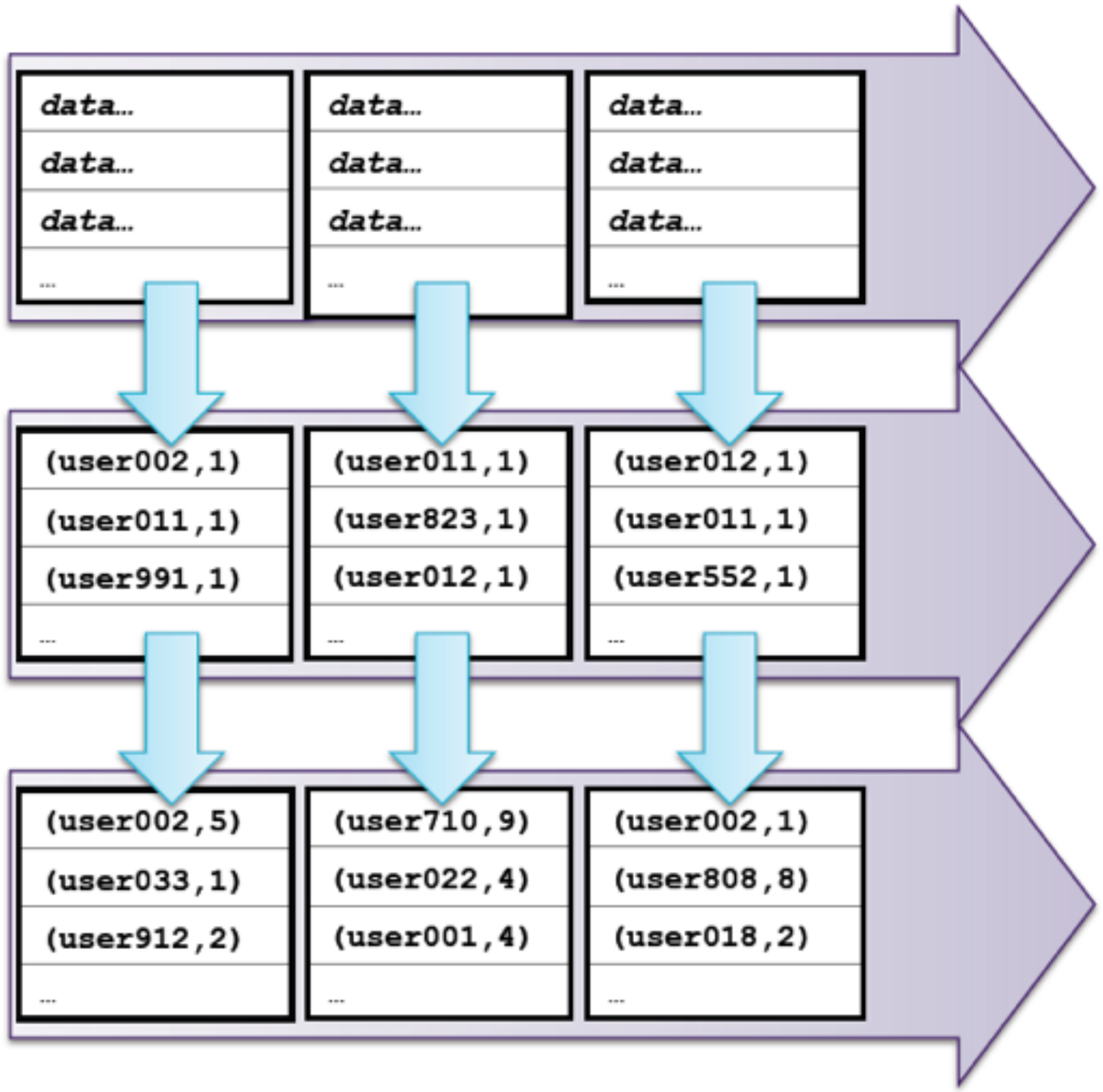
```
userreqs = logs.map(line =>
  (line.split(" ")(2),1))
```

logs

userreqs

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

reqcounts

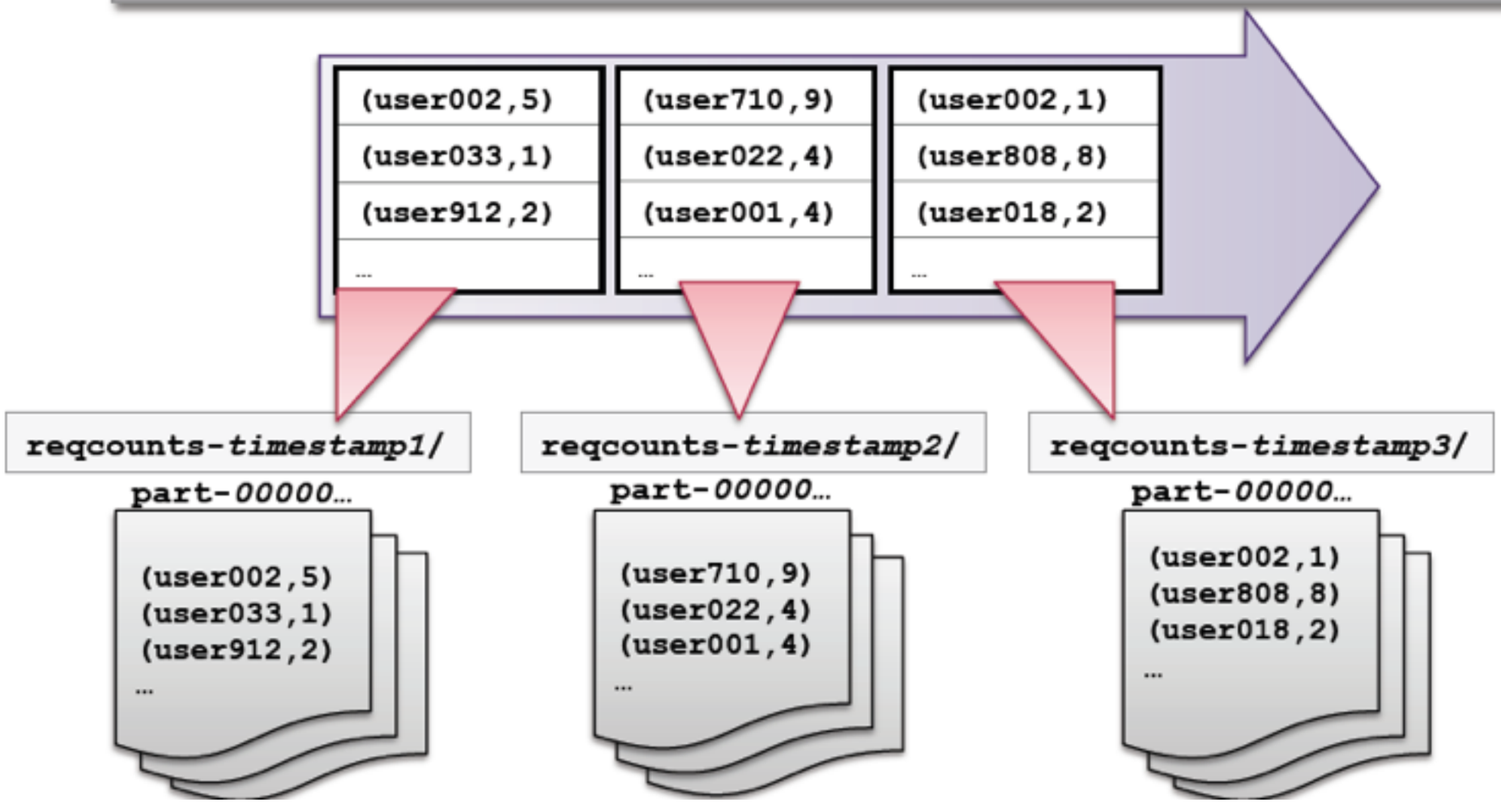


Dstream Output Operations

- **Console output**
  - **print - prints out the first 10 elements of each RDD**
- **File output**
  - **saveAsTextFiles - save data as text**
  - **saveAsObjectFiles - save as serialized object files**
- **Other types of functions**
  - **foreachRDD (function, time) - performs a function on each RDD in the Dstream**

Saving Dstream Results as Files

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



Find Top Users

```
…
val userreqs = logs
  .map(line => (line.split(" ")(2),1))
  .reduceByKey((x,y) => x+y)
userreqs.saveAsTextFiles(path)

val sortedreqs = userreqs
  .map(pair => pair.swap)
  .transform(rdd => rdd.sortByKey(false))

sortedreqs.fo
  println("To
  rdd.take(5)
    pair => printf("User: %s (%s)\n",pair._2, pair._1))
  }
)

ssc.start()
ssc.awaitTermination()
…
```

Transform each RDD: swap userID/count, sort by count

Find Top Users

```
…
val userreqs = logs
  .map(line => (line.split(" ")(2),1))
  .reduceByKey((x,y) => x+y)
userreqs.saveAsTextFiles(path)

val sortedreq
  .map(pair =
  .transform
```

Print out the top 5 users as *userID* (*count*)

```
sortedreqs.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()
…
```

Find Top Users – Output

```
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…

Using Spark Streaming with Spark Shell

- **Spark Streaming is designed for batch applications, not interactive use**
- **Spark Shell can be used for limited testing**
  - **Adding operations after the Streaming Context has been started is unsupported**

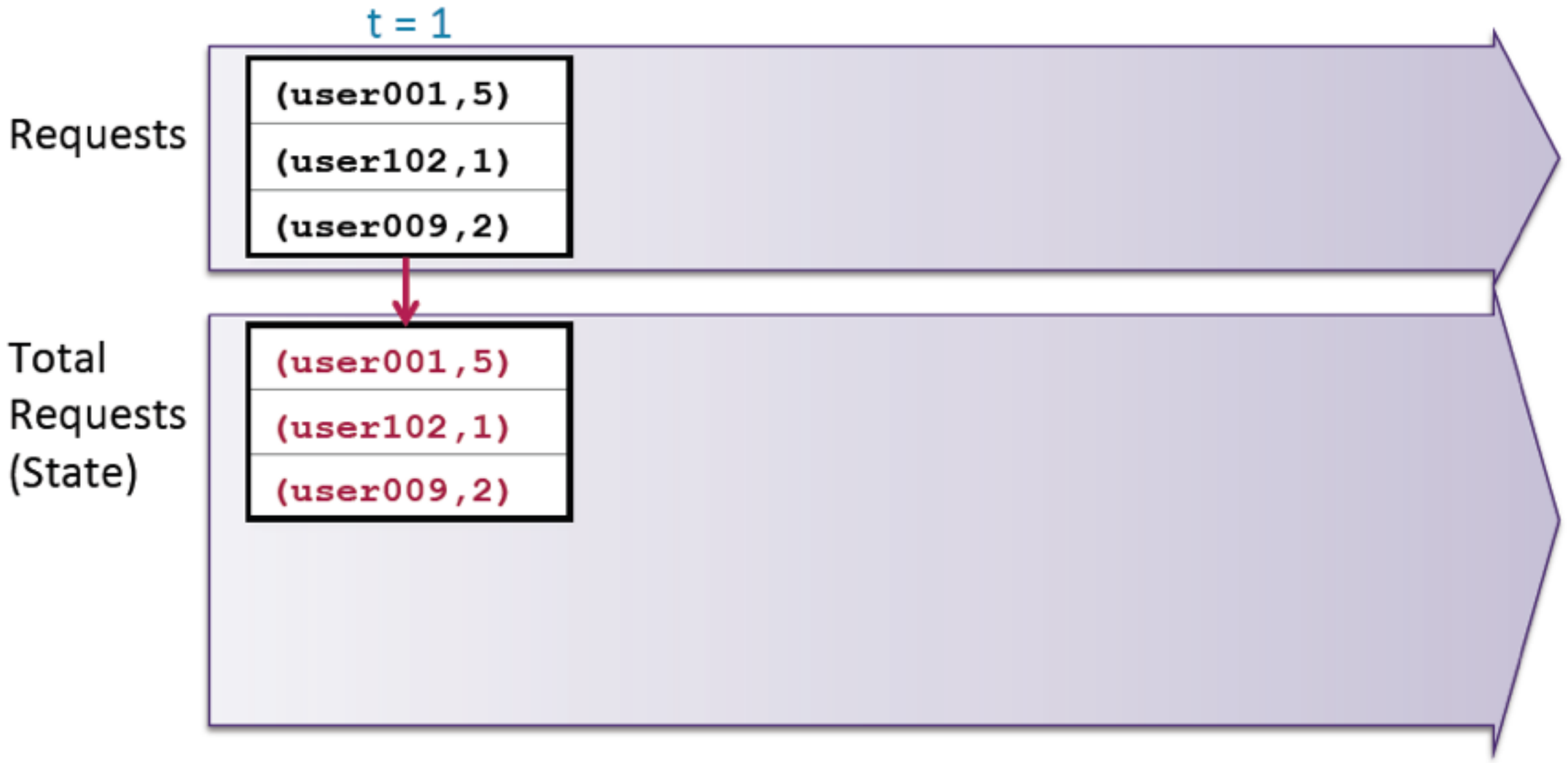
```
$ spark-shell --master local[2]
```

  - **Stopping and restarting the Streaming Context is unsupported**

HANDS-ON EXERCISE: EXPLORING SPARK STREAMING

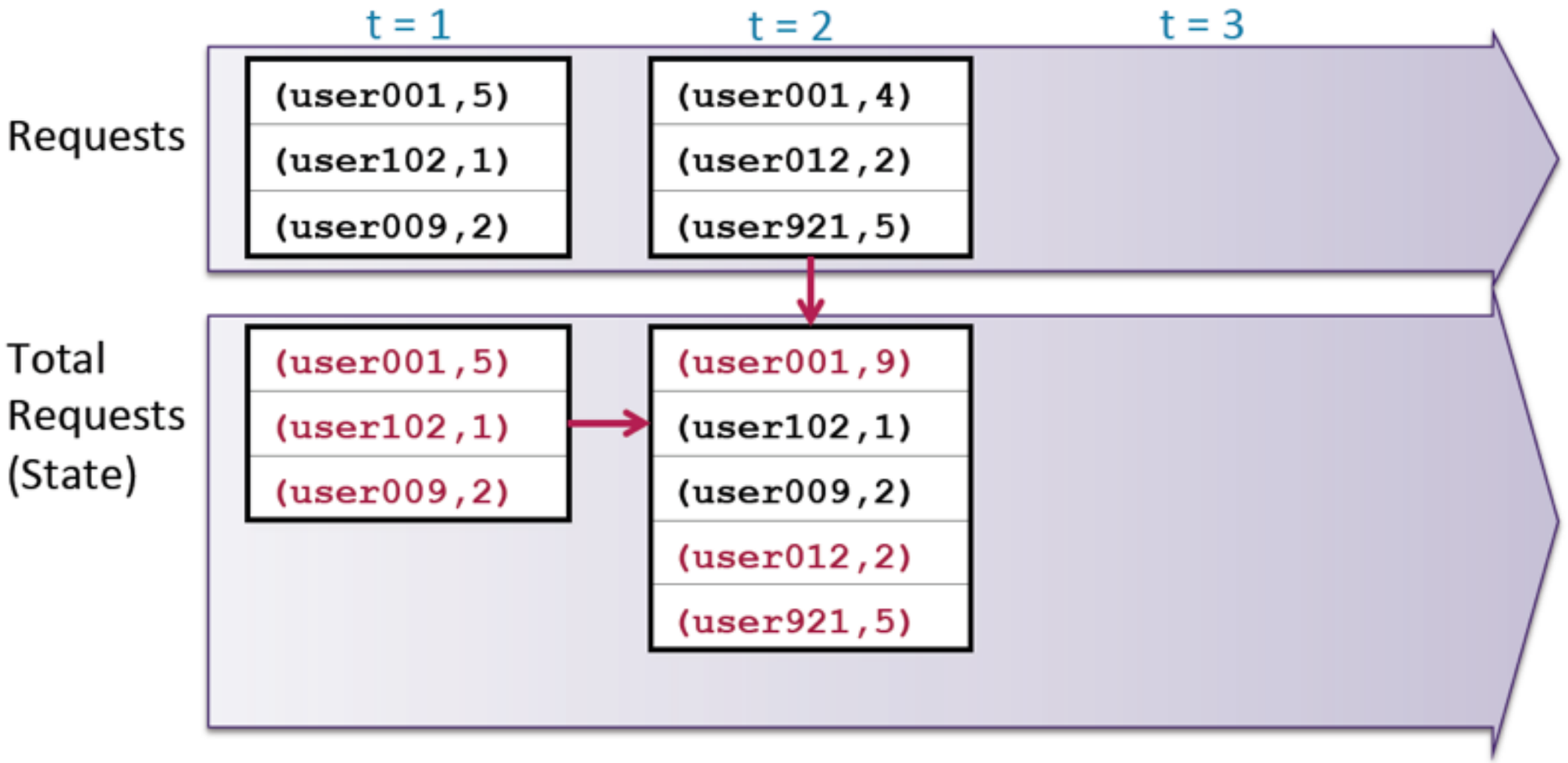
State Dstreams

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



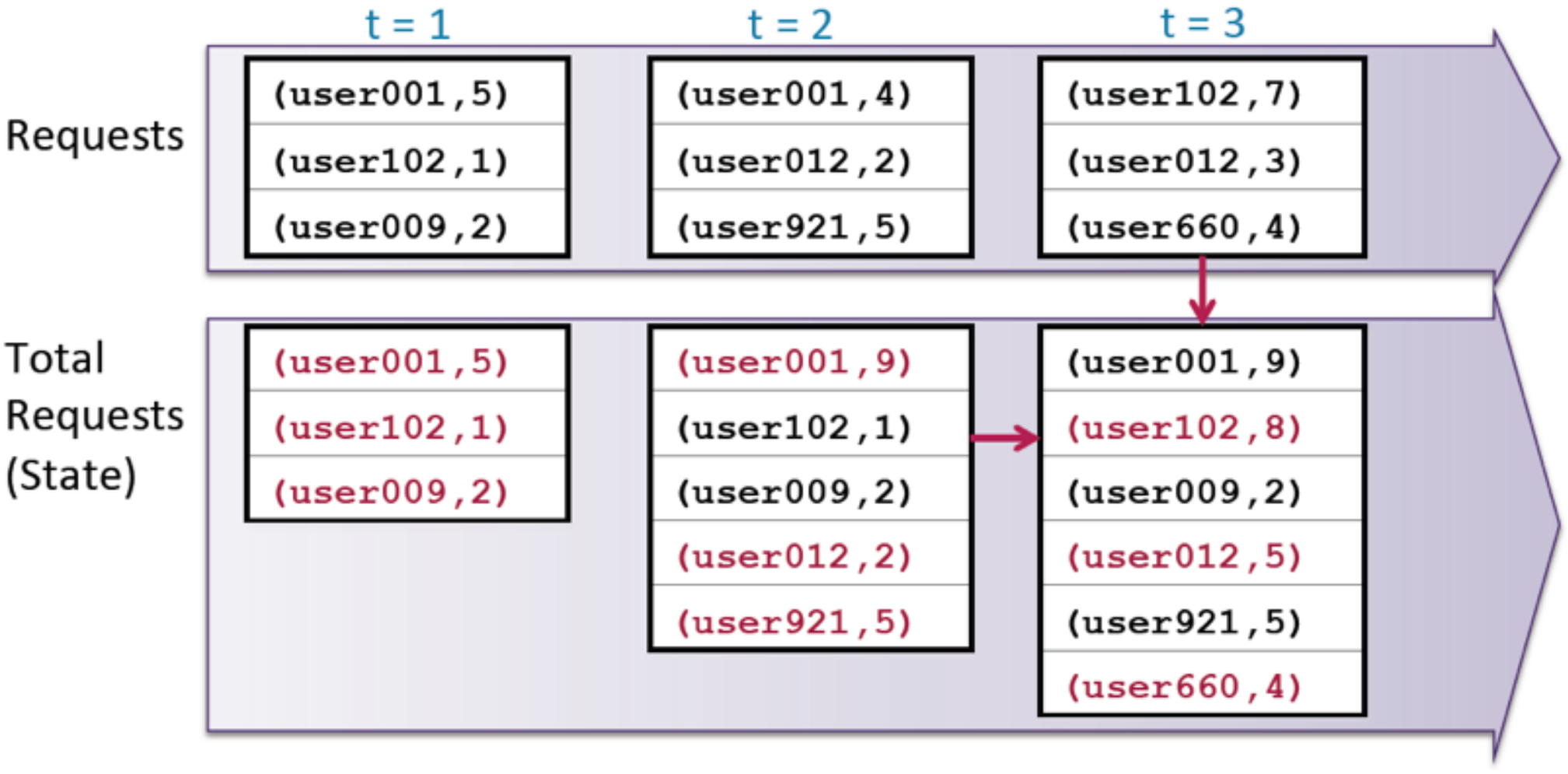
State Dstreams

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



State Dstreams

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



Total User Request Count

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

ssc.checkpoint("checkpoints")
val totalUserreqs = userreqs.updateStateByKey(updateCount)
totalUserreqs.print()

ssc.start()
ssc.awaitTermi
…
```

Set checkpoint directory to enable checkpointing.

Required to prevent infinite lineages.

Total User Request Count

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

ssc.checkpoint("checkpoints")
val totalUserreqs = userreqs.updateStateByKey(updateCount)
totalUserreqs.print()

ssc.start()
ssc.awaitTe
…
```

next slide...

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

Total User Request Count – Update Function

New Values

Current State (or **None**)

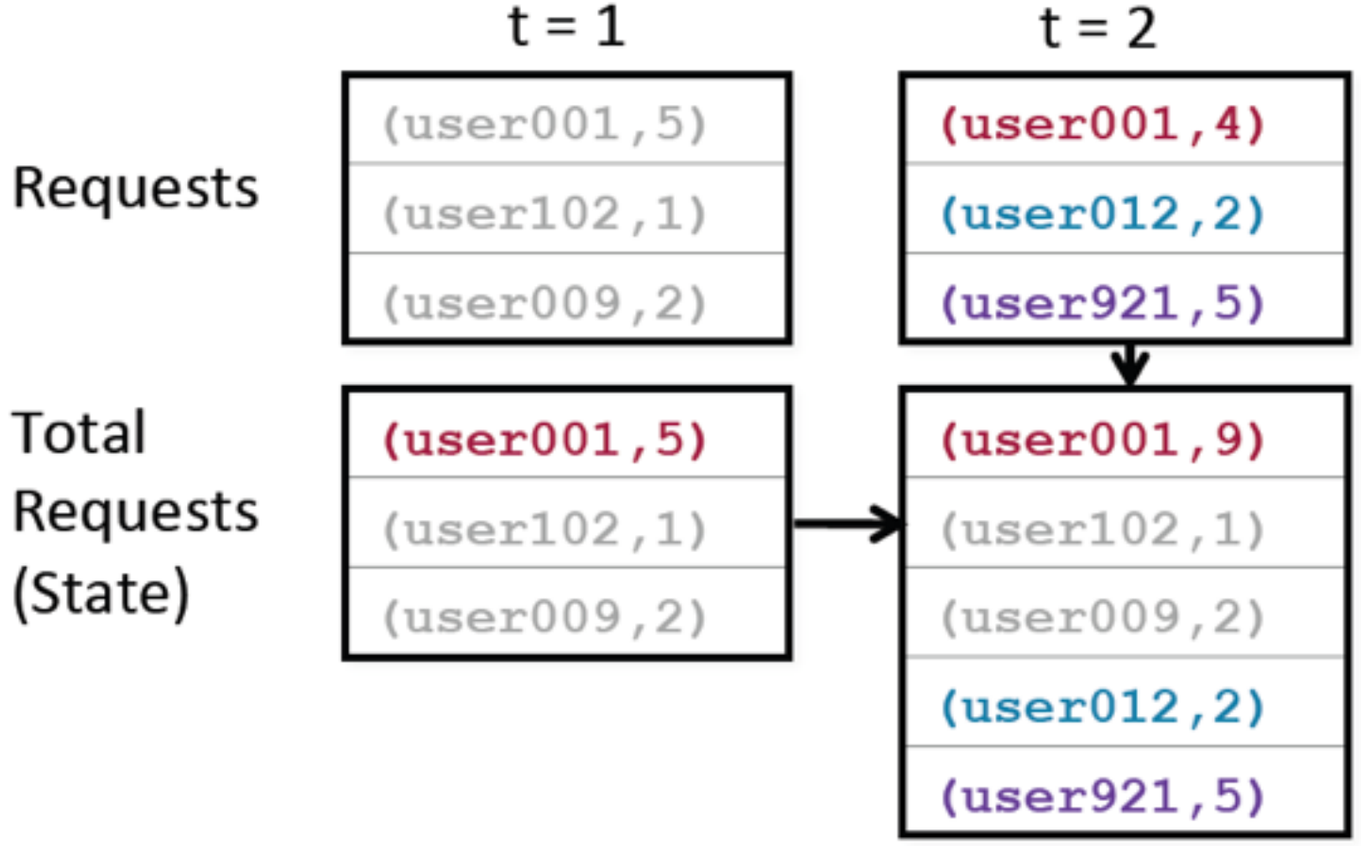
```
def updateCount = (newCounts: Seq[Int], state: Option[Int]) => {
   val newCount = newCounts.foldLeft(0)(_ + _)
   val previousCount = state.getOrElse(0)
   Some(newCount + previousCount)
}
```

New State

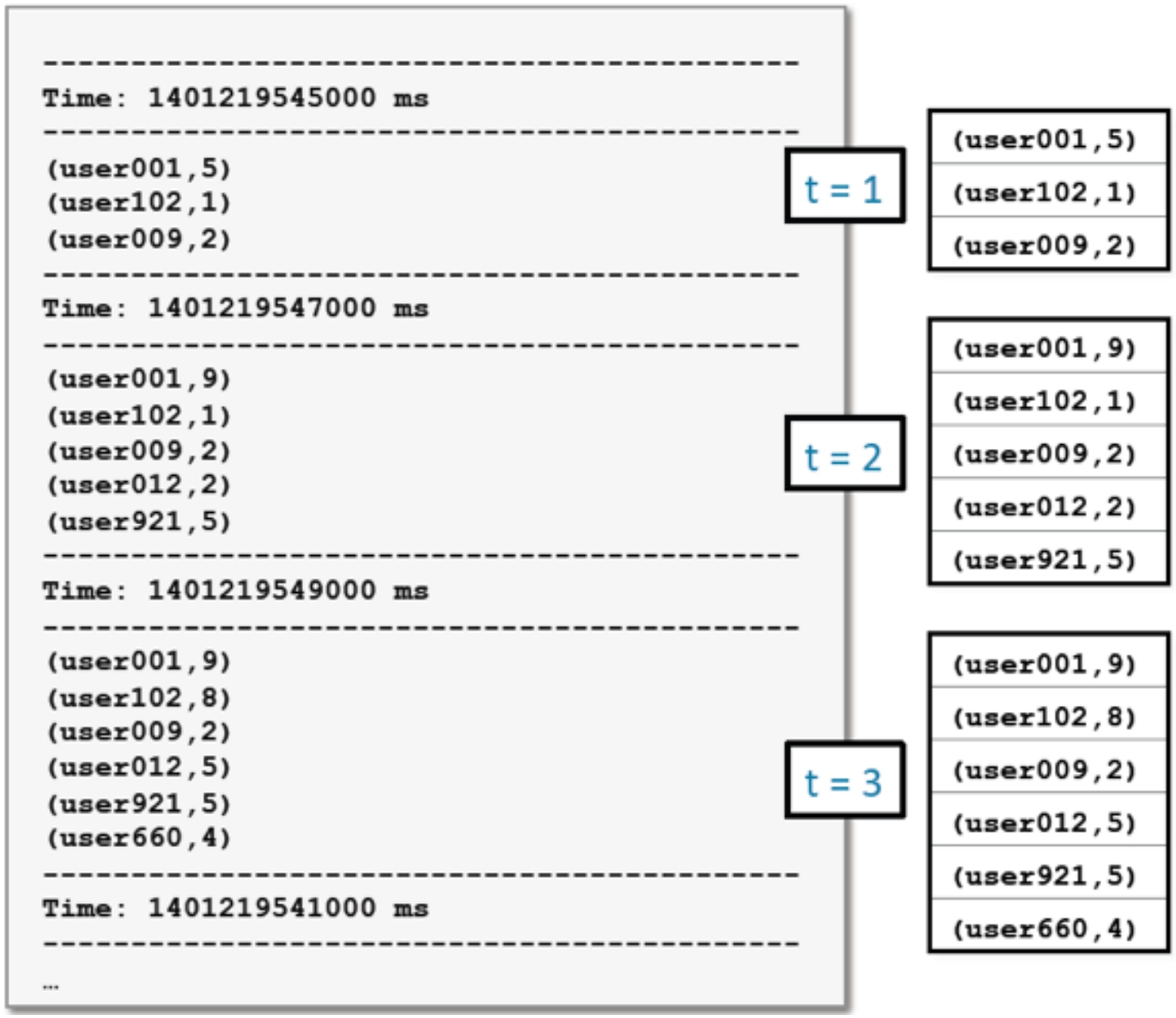
Given an existing state for a key (user), and new values (counts), return a new state (sum of current state and new counts)

Total User Request Count – Update Function

- **Example at t=2**
  - **user001: updateCount([4],Some[5]) 9**
  - **user012: updateCount([2],None)) 2**
  - **user921: updateCount([5],None)) 5**

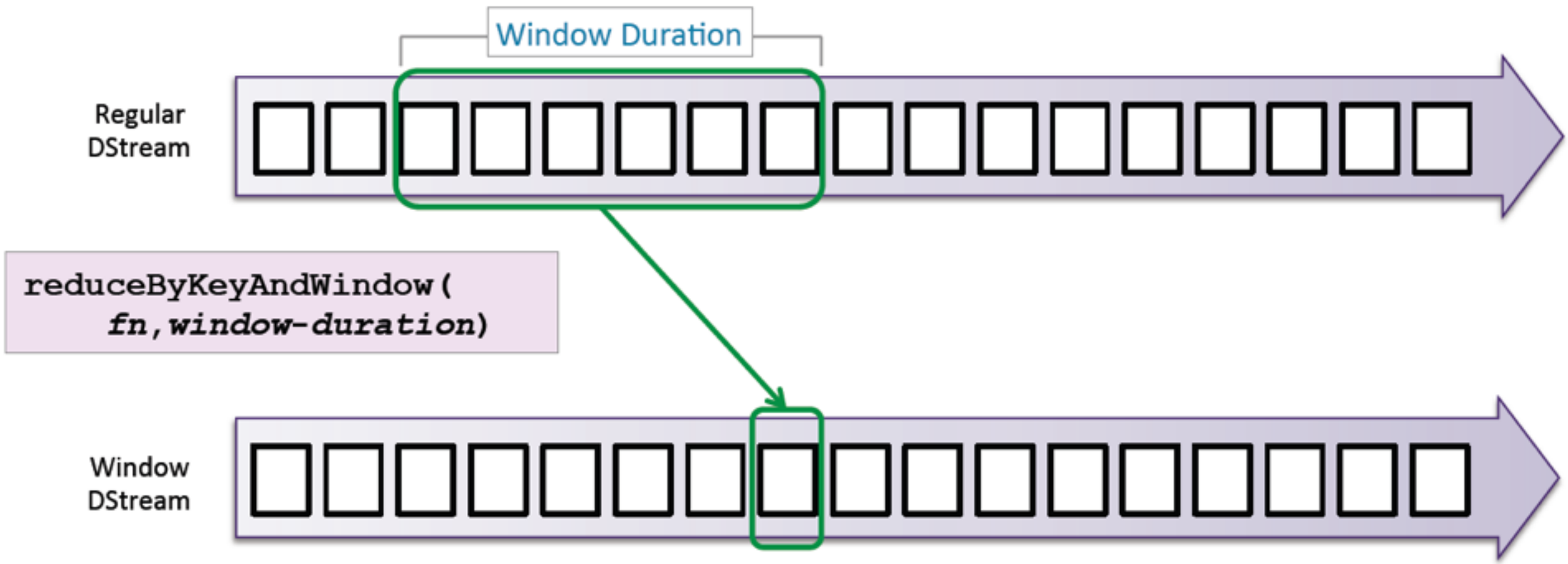


Maintaining State – Output



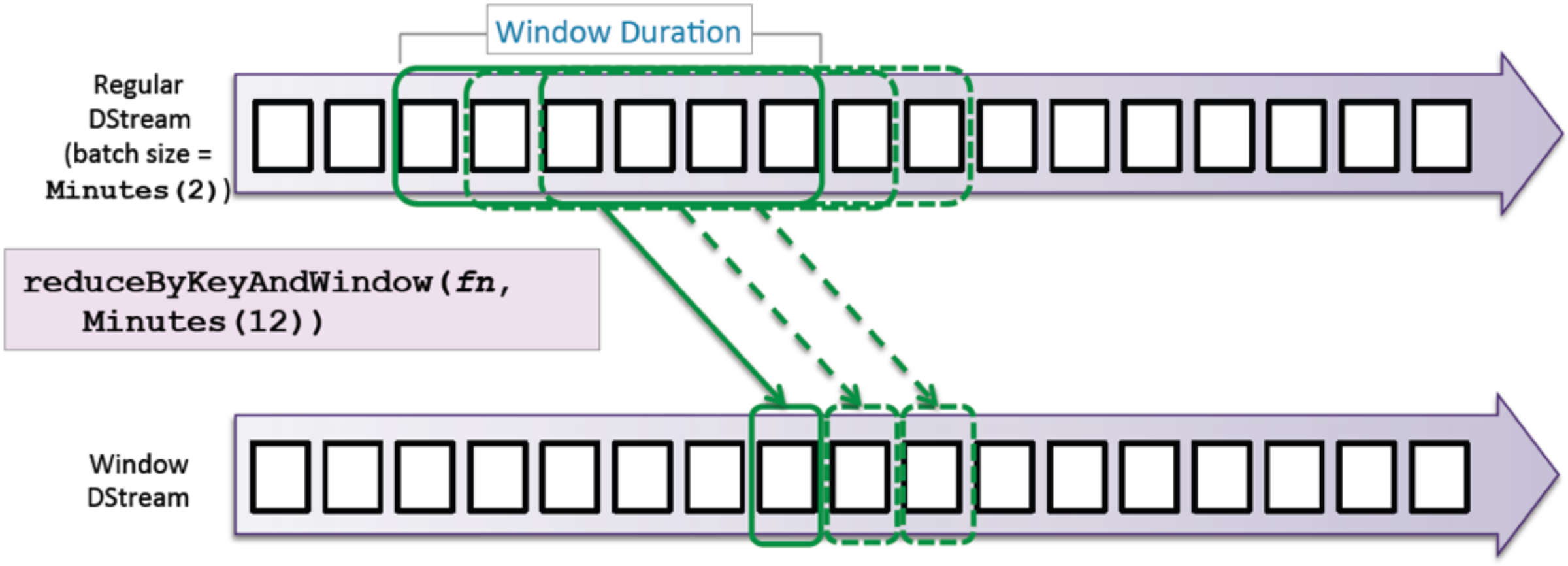
Sliding Window Operations

- **Regular Dstream operations execute for each RDD based on SSC duration**
- **"Window" operations span RDDs over a given duration**
  - **e.g., reduceByKeyAndWindow, countByWindow**



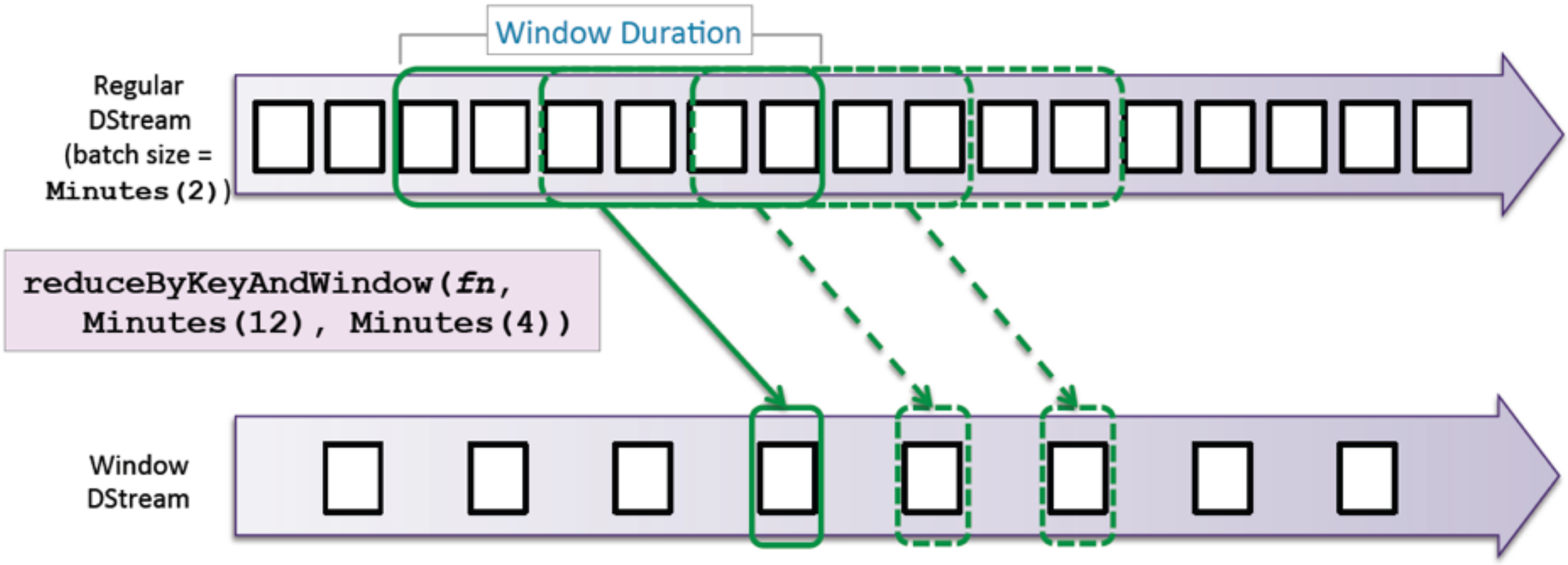
Sliding Window Operations

- **By default window operations will execute with an "interval" the same as the SCC duration**
  - **i.e., for 2 minute batch duration, window will "slide" every 2 minutes**



Sliding Window Operations

- **You can specify a different slide duration (must be a multiple of the SSC duration)**



Count and Sort User Requests by Window

```
…
val ssc = new StreamingContext(new StreamConf(), Seconds(2))
val logs = ssc.socketTextStream(hostname, port)
…
val reqcountsByWindow = logs.
  map(line => (line.split(' ')(2),1)).
  reduceByKeyAndWindow((x: Int, y: Int) => x+y,
     Minutes(5),Seconds(30))

val topreqsBy
  map(pair =>
  transform(r
topreqsByWind

ssc.start()
ssc.awaitTermination()
…
```

Every 30 seconds, count requests by user over the last 5 minutes

Count and Sort User Requests by Window

```
…
val ssc = new StreamingContext(new StreamConf(), Seconds(2))
val logs = ssc.socketTextStream(hostname, port)
…
val reqcountsB
  map(line =>
  reduceByKeyA
     Minutes(5
```

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

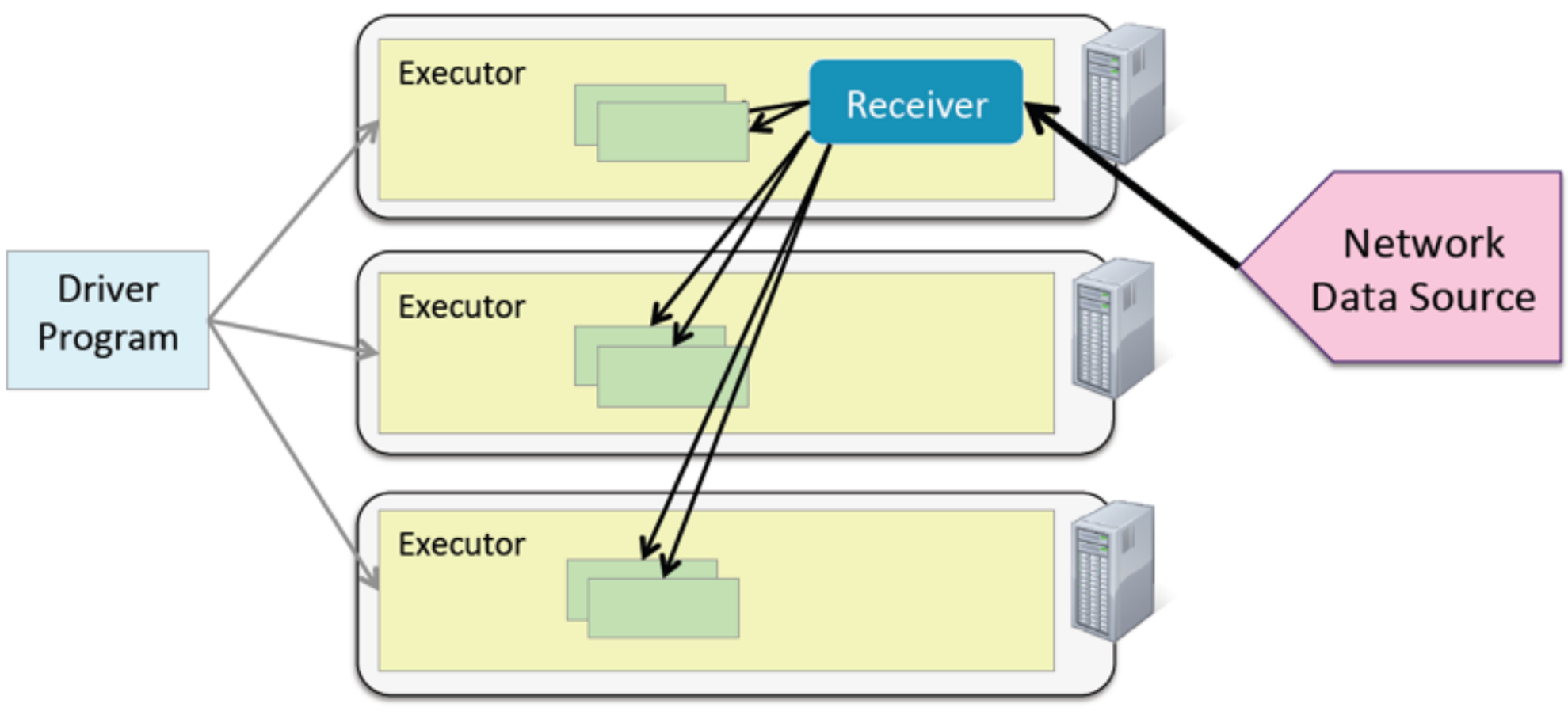
```
val topreqsByWindow=reqcountsByWindow.
  map(pair => pair.swap).
  transform(rdd => rdd.sortByKey(false))
topreqsByWindow.map(pair => pair.swap).print()

ssc.start()
ssc.awaitTermination()
…
```

Special Considerations for Streaming Applications

- **Spark Streaming applications are by definition long-running**
  - **Require some different approaches than typical Spark applications**
- **Metadata accumulates over time**
  - **Use check pointing to trim RDD lineage data**
    - **Required to use windowed and state operations**
    - **Enable by setting the checkpoint directory: ssc.checkpoint(directory)**
- **Monitoring**
  - **The StreamingListener API lets you collect statistics**

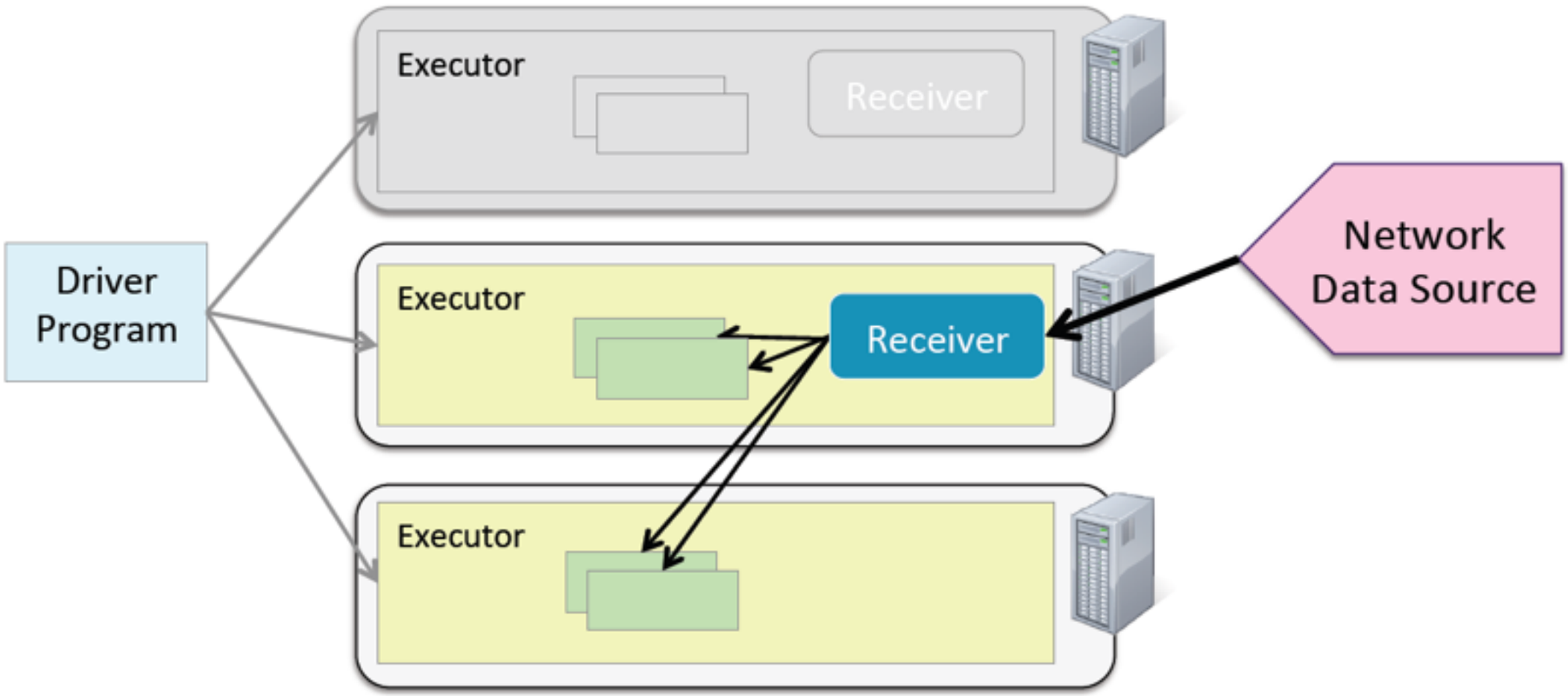
Spark Fault Tolerance

- **Network data is received on a worker node**
  - **Receiver distributes data (RDDs) to the cluster**
- **Spark Streaming persists windowed RDDs by default (replication = 2)**



Spark Fault Tolerance

- **If the receiver fails, Spark will restart it on a different Executor**
  - **Potential for brief loss of incoming data**

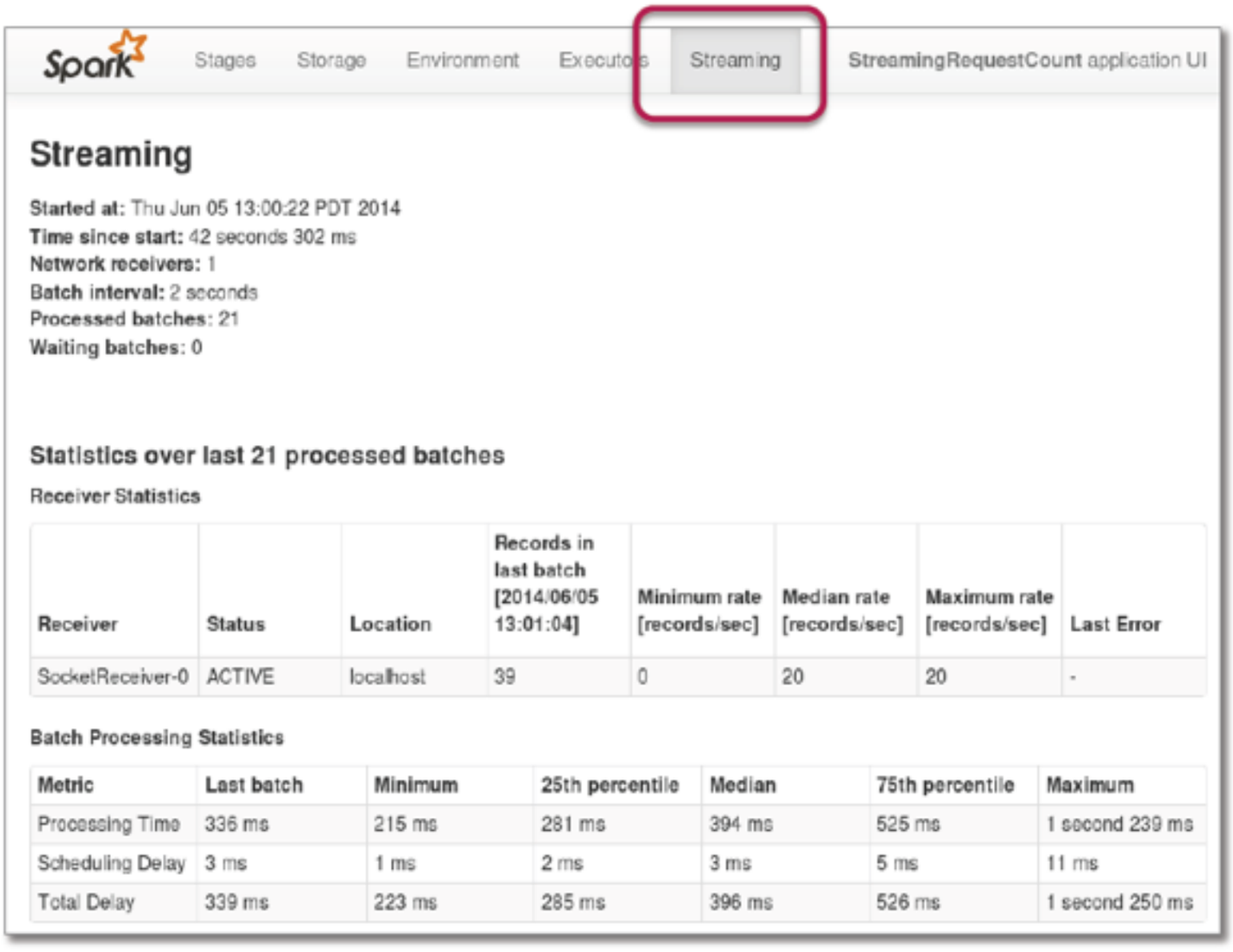


Building and Running Spark Streaming Applications

- **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**
  - **Use at least two threads if running locally**

The Spark Streaming Application UI

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



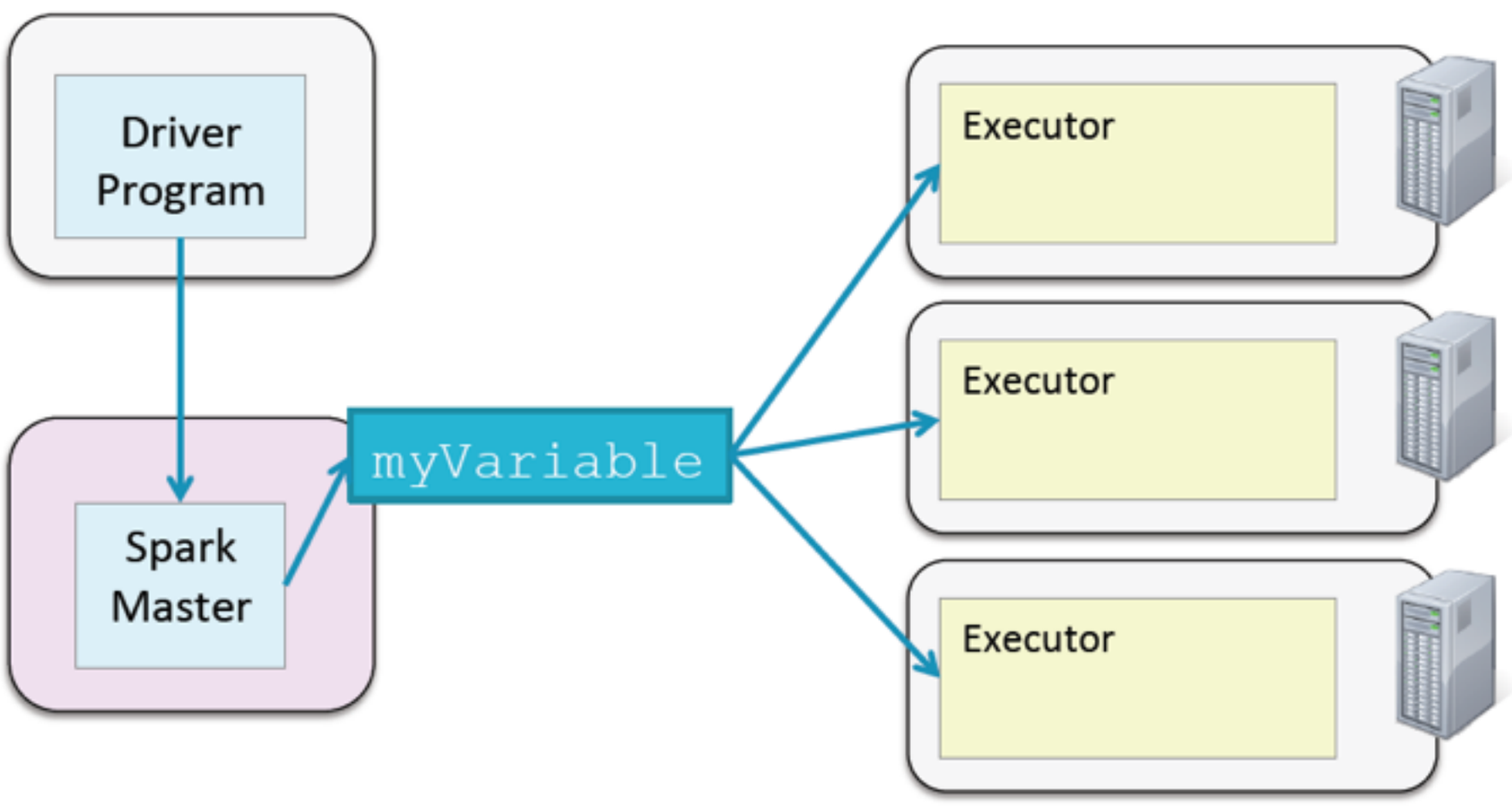
HANDS-ON EXERCISE: WRITING A SPARK STREAMING APPLICATION

Chapter 11

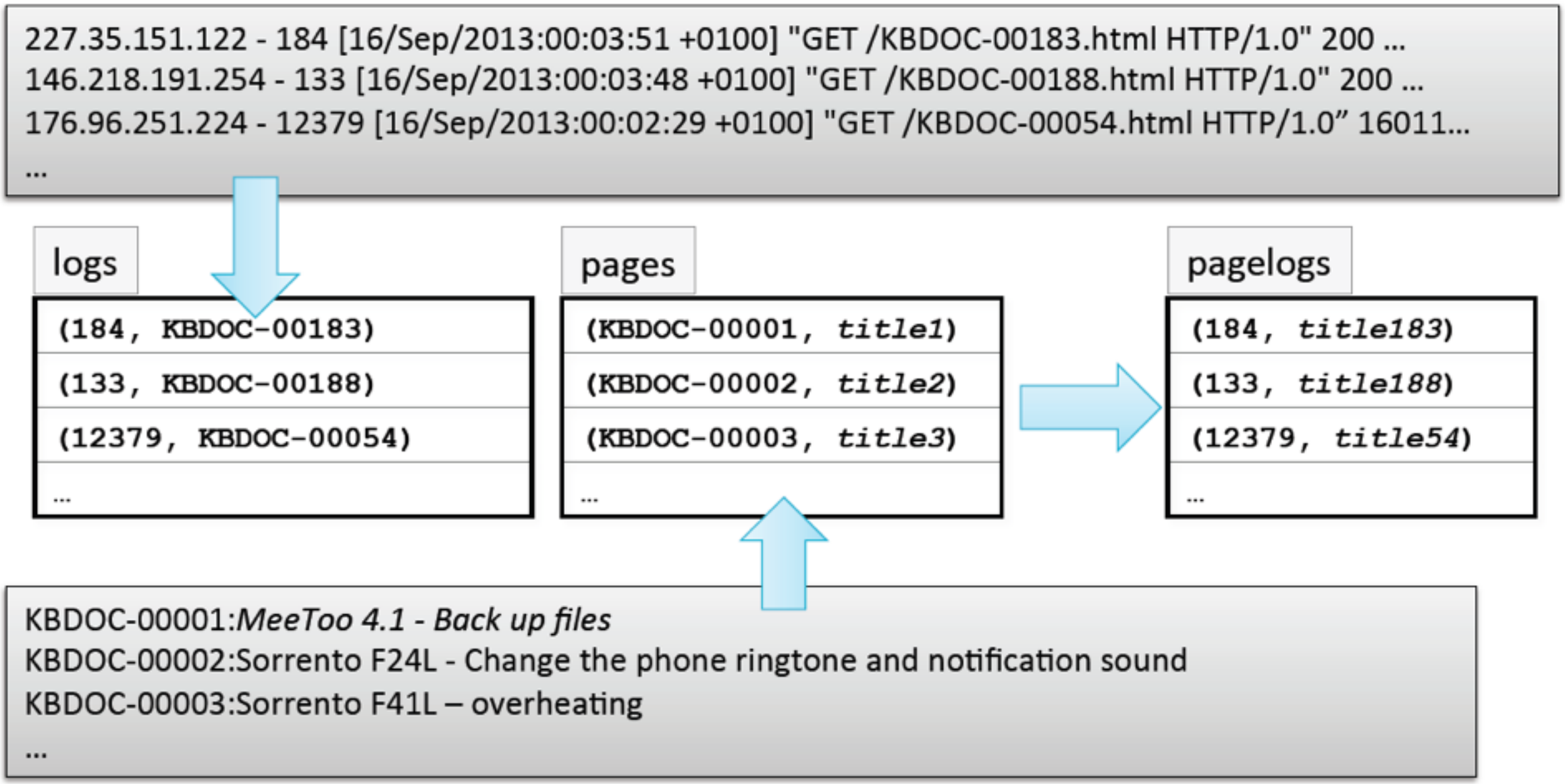
IMPROVING SPARK PERFORMANCE

Broadcast Variables

- **Broadcast variables are set by the driver and retrieved by the workers**
- **They are read only once set**
- **The first read of a Broadcast variable retrieves and stores its value on the node**

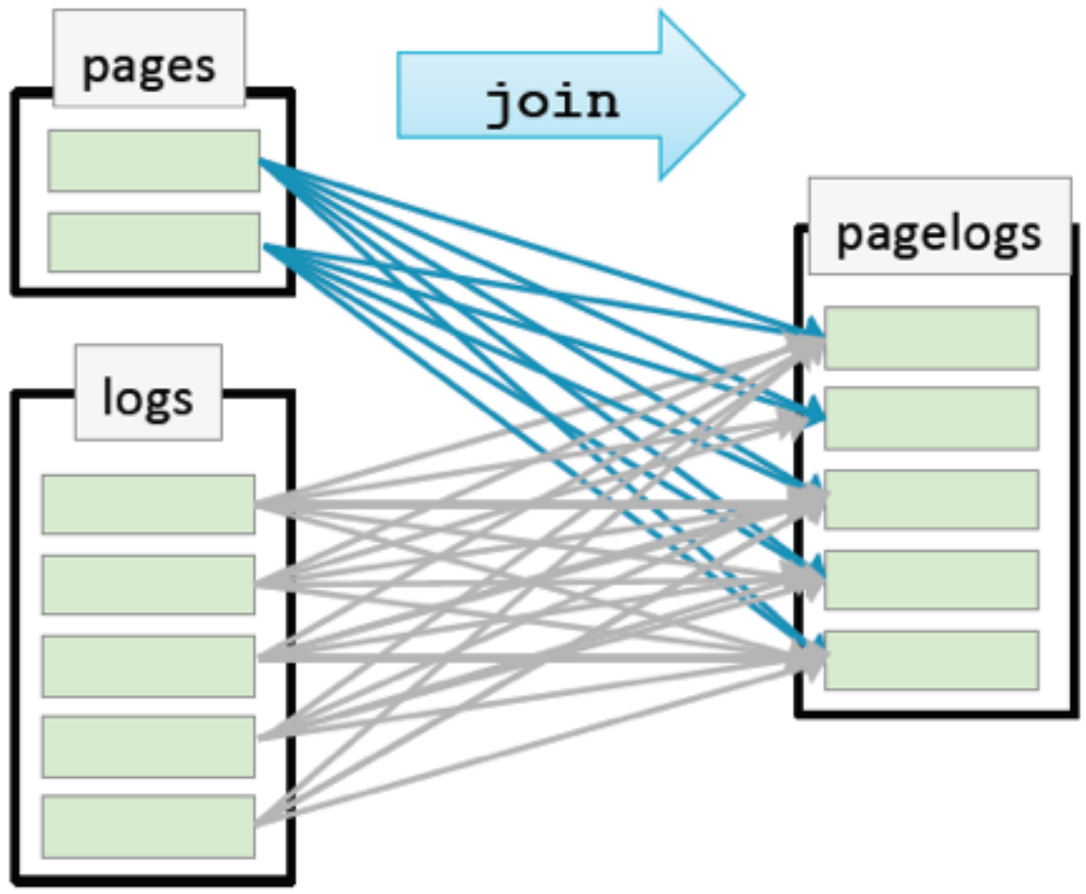


Match User IDs with Requested Page Titles



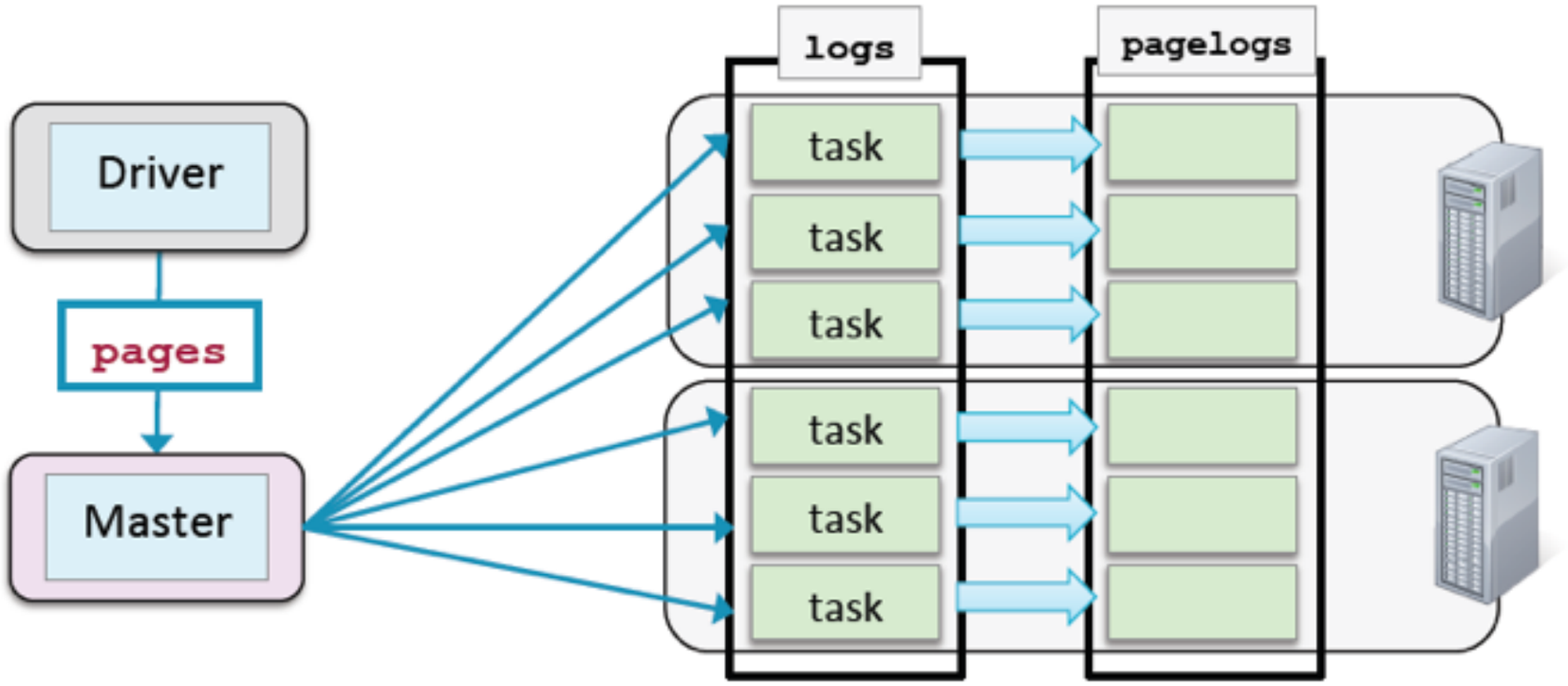
Join a Web Server Log with Page Titles

```
logs = sc.textFile(logfile).map(fn)
pages = sc.textFile(pagefile).map(fn)
pagelogs = logs.join(pages)
```



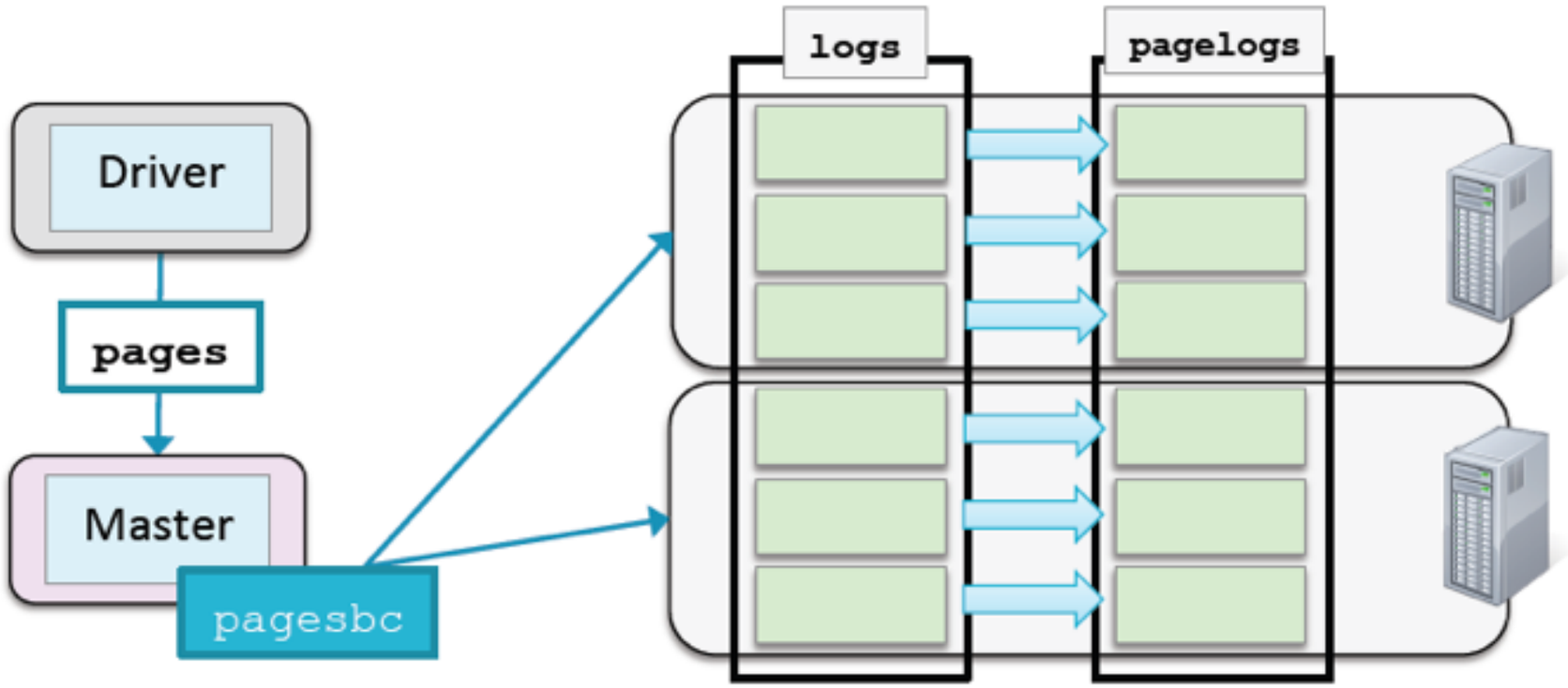
Pass a Small Table as a Parameter

```
logs = sc.textFile(logfile).map(fn)
pages = dict(map(fn,open(pagefile)))
pagelogs = logs.map(lambda (userid,pageid):
   (userid,pages[pageid]))
```



Broadcast a Small Table

```
logs = sc.textFile(logfile).map(…)
pages = dict(map(fn,open(pagefile)))
pagesbc = sc.broadcast(pages)
pagelogs = logs.map(lambda (userid, pageid):
   (userid,pagesbc.value[pageid]))
```



Broadcast Variables

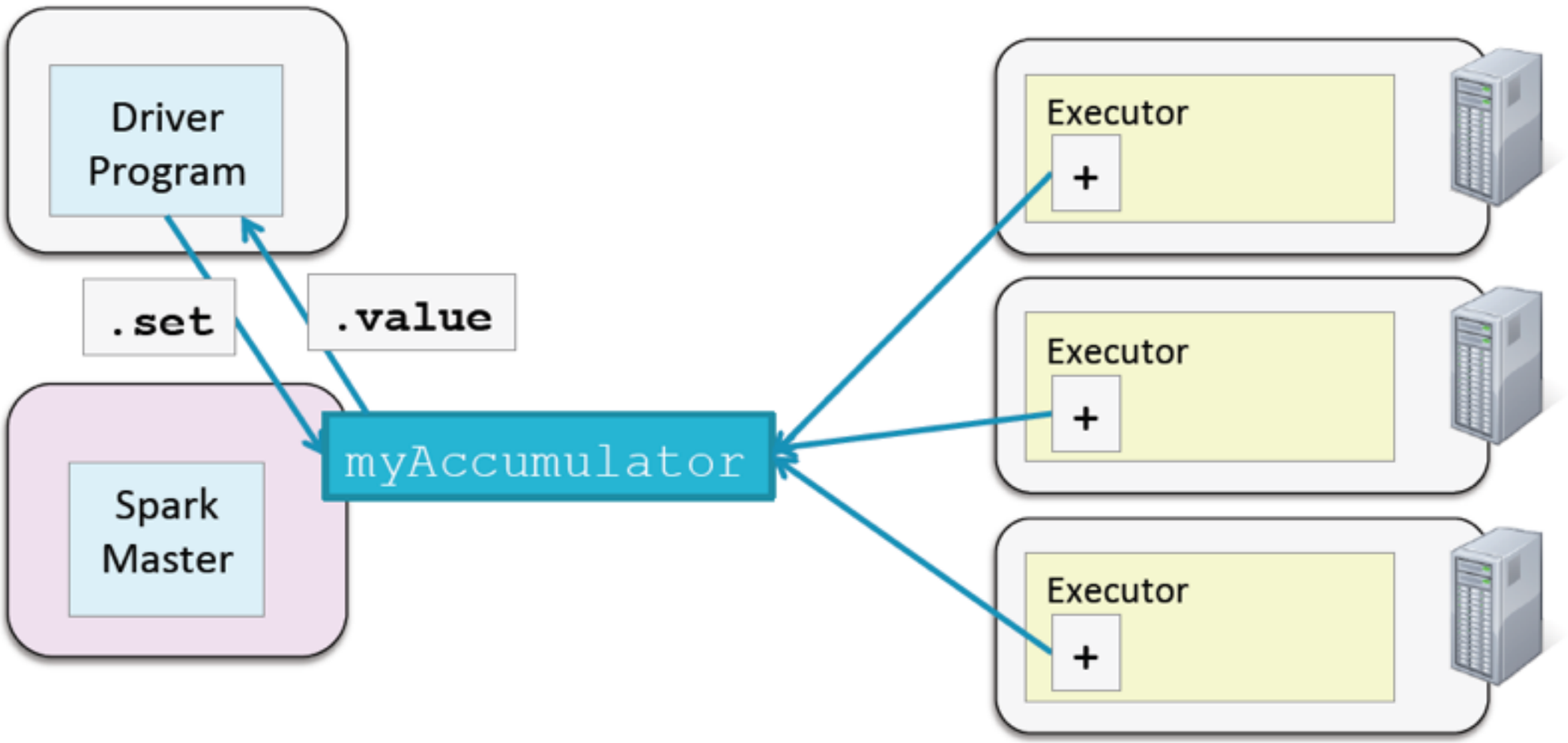
- **Why use Broadcast variables?**
  - **Use to minimize transfer of data over the network, which is usually the biggest bottleneck**
  - **Spark Broadcast variables are distributed to worker nodes using a very efficient peer-to-peer algorithm**

In this Hands-On Exercise, you will filter web server logs for requests

HANDS-ON EXERCISE: USING BROADCAST VARIABLES

Accumulators

- **Accumulators are shared variables**
  - **Worker nodes can add to the value**
  - **Only the driver application can access the value**



Accumulator Example: Average Word Length

```
def addTotals(word,words,letters):
    words += 1
    letters += len(word)

totalWords = sc.accumulator(0)
totalLetters = sc.accumulator(0.0)

words = sc.textFile(myfile) \
  .flatMap(lambda line: line.split())

words.foreach(lambda word: \
       addTotals(word,totalWords,totalLetters))

print "Average word length: ", \
   totalLetters.value/totalWords.value
```

More About Accumulators

- Accumulators will only be incremented once per task
  - If tasks must be rerun due to failure, Spark will correctly add only for the task which succeeds
- Only the driver can access the value
  - Updates are only sent to the master, not to all workers
  - Code will throw an exception if you use .value on worker nodes
- Supports the increment (+=) operator
- Can use integers or doubles
  - sc. accumulator(0)
  - sc. accumulator(0.0)
- Can customize to support any data type
  - Extend the Accumulator Param class

HANDS-ON EXERCISE: USING ACCUMULATORS

Performance Issue: Serialization

- **Serialization affects**
  - **Network bandwidth**
  - **- Memory (save memory by serializing)**
- **Default method of serialization in Spark is basic Java serialization**
  - **- Simple but slow**

Using Kryo Serialization

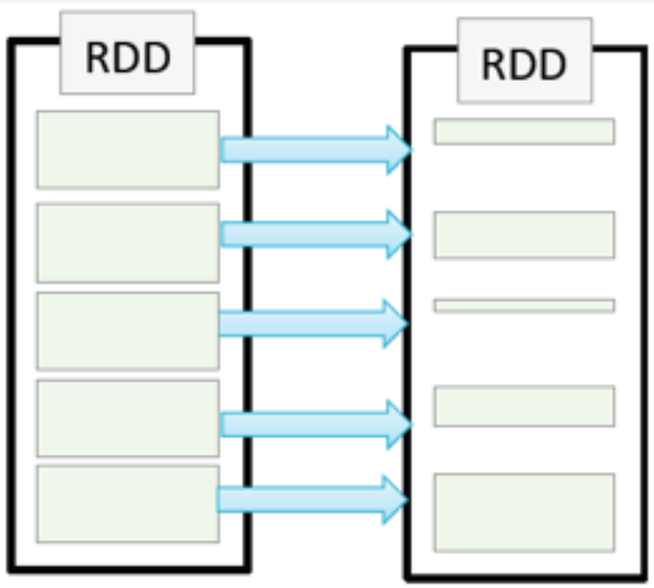
- Use Kryo serialization for Scala and Java
  - To enable, set spark.serializer = spark.KryoSerializer
- To enable Kryo for your custom classes
  - Create a KryoRegistrar class and set spark.kryo.registrator=MyRegistrator
  - Register your classes with Kryo

```
class MyRegistrator extends spark.KryoRegistrator {
  def registerClasses(kryo: Kryo) {
    kryo.register(classOf[MyClass1])
    kryo.register(classOf[MyClass2])
    …
  }
}
```

Performance Issue: Small Partitions

- **Problem: filter() can result in partitions with small amounts of data**
  - **Results in many small tasks**

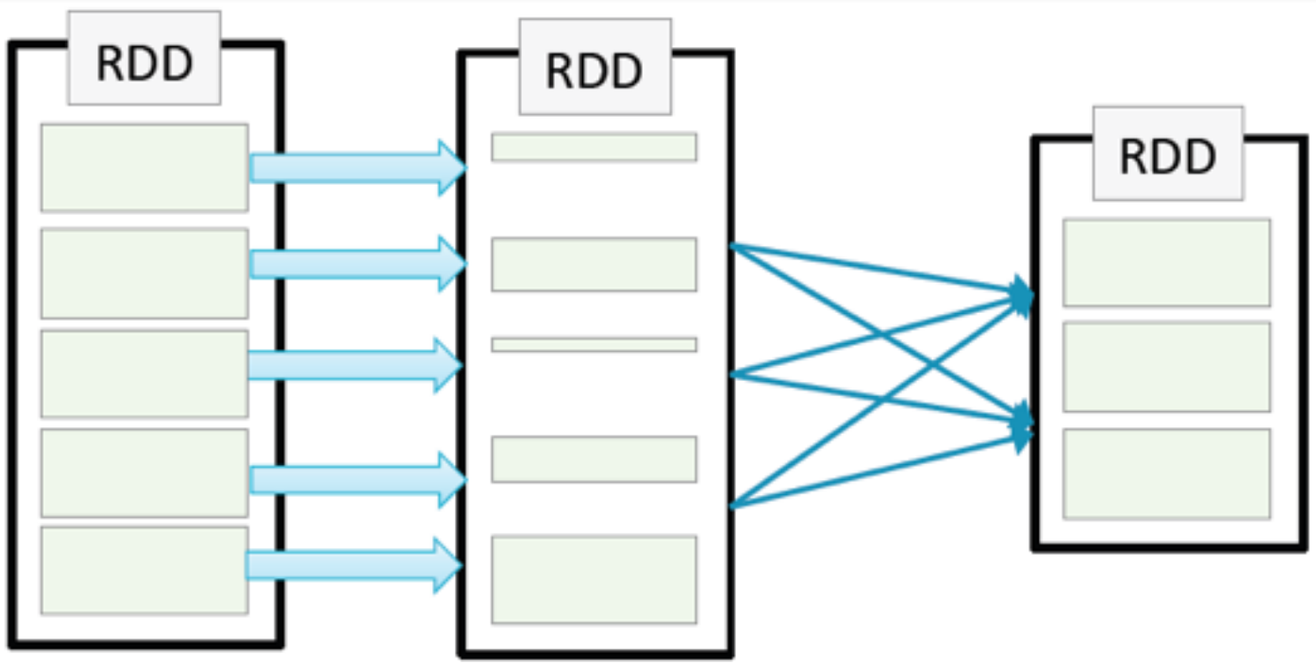
```
sc.textFile(file) \
  .filter(lambda s: s.startswith('I')) \
  .map(lambda s: \
     (s.split()[0],(s.split()[1],s.split()[2])))
```



Solution: Repartition/Coalesce

- **Solution: repartition(n)**
  - **This is the same as coalesce(n, shuffle=false)**

```
sc.textFile(file) \
  .filter(lambda s: s.startswith('I')) \
  .repartition(3) \
.map(lambda s: \
   (s.split()[0],(s.split()[1],s.split()[2])))
```



Performance Issue: Passing Too Much Data in Functions

- **Problem: Passing large amounts of data to parallel functions results in poor performance**

```
hashmap = some_massive_hash_map()
…
myrdd.map(lambda x: hashmap(x)).countByValue()
```

Performance Issues: Passing Too Much Data in Functions

- **Solution:**
  - **If the data is relatively small, use a Broadcast variable**

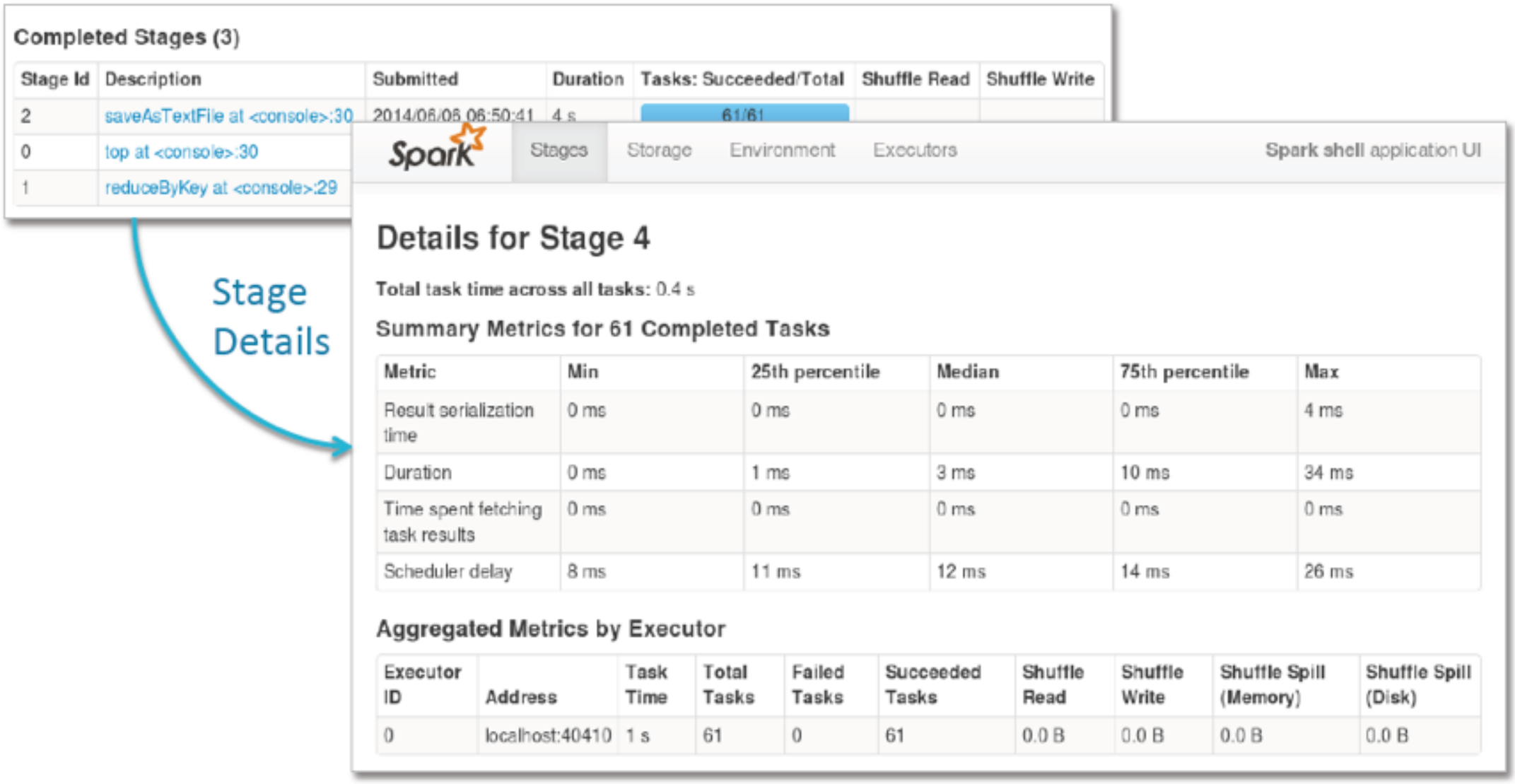
```
hashmap = some_massive_hash_map()
bhashmap = sc.broadcast(hashmap)
…
myrdd.map(lambda x: bhashmap(x)).countByValue()
```

  - If the data is very large, parallelize into an RDD

```
hashmap = some_massive_hash_map()
hashmaprdd = sc.parallelize(hashmap)
…
myrdd.join(bhashmaprdd).countByValue()
```

Diagnosing Performance Issues

- **The Spark Application UI provides useful metrics to find performance problems**



Diagnosing Performance Issues

- **Where to look for performance issues**
  - **Scheduling and launching tasks**
  - **Task execution**
  - **Shuffling**
  - **Collecting data**

Scheduling and Launching Issues

- **Scheduling and launching taking too long**
  - **Are you passing too much data to tasks?**
    - **myrdd.map(lambda x: HugeLookupTable(x))**
  - **Use a Broadcast variable or an RDD**

Summary Metrics for 11 Completed Tasks

Metric	Min	25th percentile	Median	75th percentile	Max
Result serialization time	0 ms	0 ms	0 ms	0 ms	4 ms
Duration	20 ms	23 ms	30 ms	44 ms	0.3 s
Time spent fetching task results	0 ms	0 ms	0 ms	0 ms	0 ms
Scheduler delay	11 s	11 s	12 s	12 s	15 s

Task Execution Issues

- **Task execution taking too long?**
  - **Are there tasks with a very high per-record overhead?**
    - **e.g., mydata.map (dbLookup)**
    - **Each lookup call opens a connection to the DB, reads, and closes**
  - **Try mapPartitions**

Task Execution Issues

- **Are a few tasks taking much more time than others?**
  - **Repartition, partition on a different key, or write a custom partitioner**

Summary Metrics for 182 Completed Tasks

Metric	Min	25th percentile	Median	75th pe…	
Result serialization time	0 ms	0 ms	0 ms	0 ms	1 ms
Duration	11 ms	15 ms	17 ms	20 ms	50 ms
Time spent fetching	0 ms	0 ms	0 ms	0 ms	0 ms

Task durations should be fairly even

Example: empty partitions due to filtering

182 Partitions

Block Name	Storage Level	Size in Memory ▲	Size on Disk	Executors
rdd_8_87	Memory Deserialized 1x Replicated	2.2 MB	0.0 B	localhost:56859
rdd_8_0	Memory Deserialized 1x Replicated	112.0 B	0.0 B	localhost:56859
rdd_8_1	Memory Deserialized 1x Replicated	112.0 B	0.0 B	localhost:56859
rdd_8_10	Memory Deserialized 1x Replicated	112.0 B	0.0 B	localhost:56859
rdd_8_100	Memory Deserialized 1x Replicated	112.0 B	0.0 B	localhost:56859

Shuffle Issues

- **Writing shuffle results taking too long?**
  - **Make sure you have enough memory for buffer cache**
  - **Make sure spark.local.dir is a local disk, ideally dedicated**

Completed Stages (3)

Stage Id	Description	Submitted			fle Read	Shuffle Write
2	saveAsTextFile at <console>:30	2014/06/06 06:50:41				
0	top at <console>:30	2014/06/06 06:49:56	3 s	61/61		
1	reduceByKey at <console>:29	2014/06/06 06:49:47	9 s	61/61		168.9 KB

Saves to disk if too big for buffer cache

Tasks

Look for big write times

Task Index	Task ID	Status	Locality Level	Executor	Time	Duration	GC Time	Result Ser Time	Write Time ▲	Shuffle Write	Errors
26	26	SUCCESS	PROCESS_LOCAL	localhost	2014/06/06 05:11:44	66 ms	4 ms		9 ms	2.4 KB	
21	21	SUCCESS	PROCESS_LOCAL	localhost	2014/06/06 05:11:44	74 ms			8 ms	2.4 KB	
22	22	SUCCESS	PROCESS_LOCAL	localhost	2014/06/06 05:11:44	0.1 s			6 ms	2.4 KB	

Collecting Data to the Driver

- **Are results taking too long?**
  - Beware of returning large amounts of data to the driver, for example with `collect()`
  - Process data on the workers, not the driver
  - Save large results to HDFS

Watch for disproportionate result serialization times

Tasks

Task Index	Task ID	Status	Locality Level	Executor	Launch Time	Duration	GC Time	Result Ser Time ▲	Errors
153	215	SUCCESS	PROCESS_LOCAL	localhost	2014/06/06 08:17:23	27 ms	85 ms	94 ms	
87	149	SUCCESS	PROCESS_LOCAL	localhost	2014/06/06 08:17:21	35 ms	72 ms	87 ms	
116	178	SUCCESS	PROCESS_LOCAL	localhost	2014/06/06 08:17:22	44 ms	54 ms	77 ms	

Performance Analysis and Monitoring

- Spark supports integration with other performance tools
  - Configurable metrics system built on the Coda Hale Metrics Library
  - Metrics can be
    - Saved to files
    - Output to the console
    - Viewed in the JMX console
    - Sent to reporting tools like Graphite or Ganglia

THANK YOU!