Parallel Processing With Spark

Parallel Processing With Spark

- In this chapter, you will learn
  - How RDDs are distributed across a cluster
  - How Spark executes RDD operations in parallel

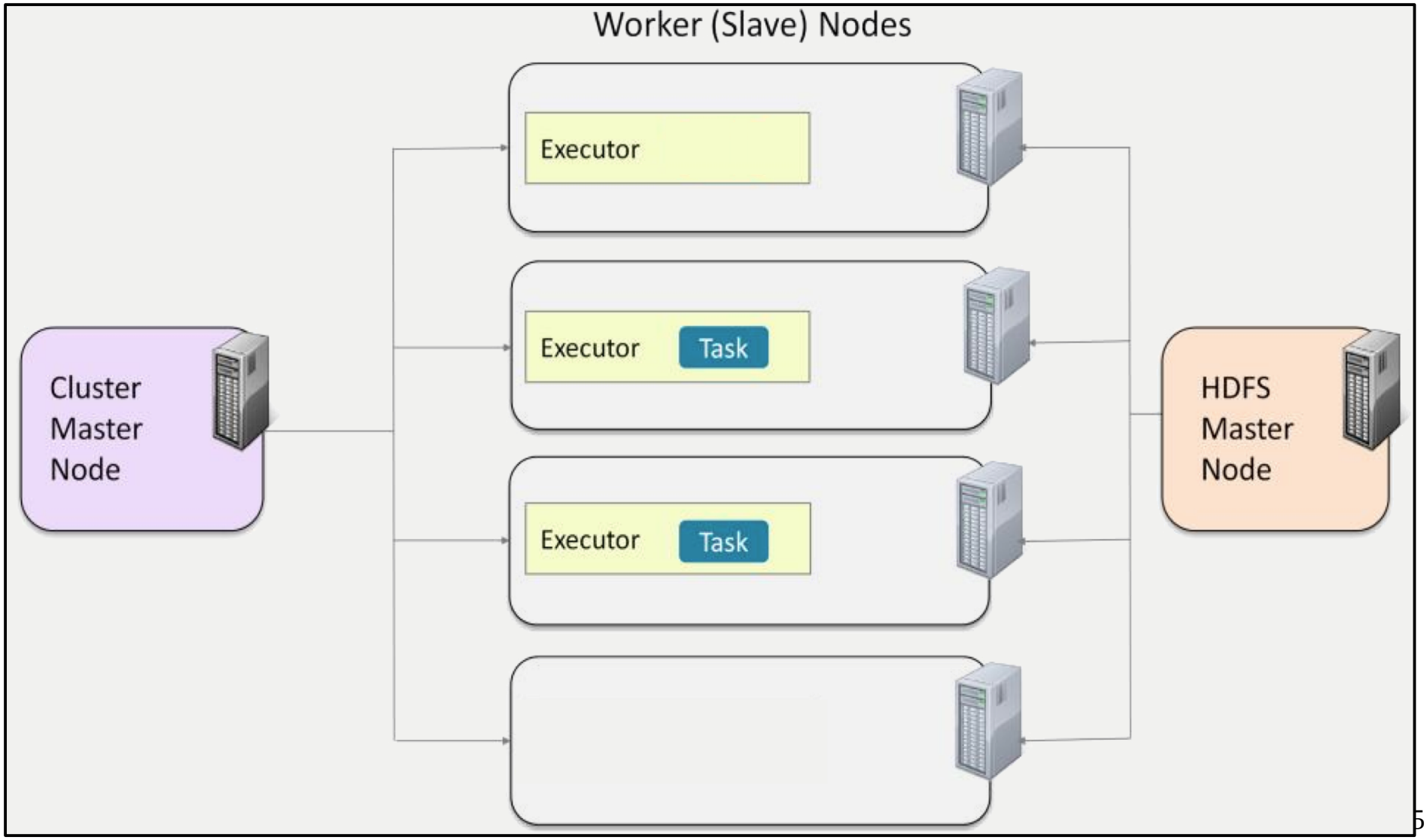
Chapter Topics

- Review: Spark On A Cluster
- RDD Partitions
- Partitioning Of File-Based RDDS
- HDFS And Data Locality
- Hands-On Exercise: Working With Partitions
- Executing Parallel Operations
- Stages And Tasks
- Summary
- Review
- References
- Hands-On Exercise: Viewing Stages And Tasks In The Spark Application UI

Introduction

- To this juncture we have been treating RDDs as though they represented individual monolithic collections of data
  - Like arrays
  - Actually, that's one of the nice things about Spark
    - You can operate on data without having to know the underlying details about how or where it is stored
- But what happens when an RDD gets too big to fit in the memory available on any single node?
  - It gets broken down and distributed across the cluster, reaping the benefits of scalability and parallel processing in the process
- To get the most from Spark, and to write the most efficient Spark code, you need to understand how RDD's work beneath the hood
  - That is the focus of this chapter

Spark Cluster Review (1 Of 2)



Spark Cluster Review (2 Of 2)

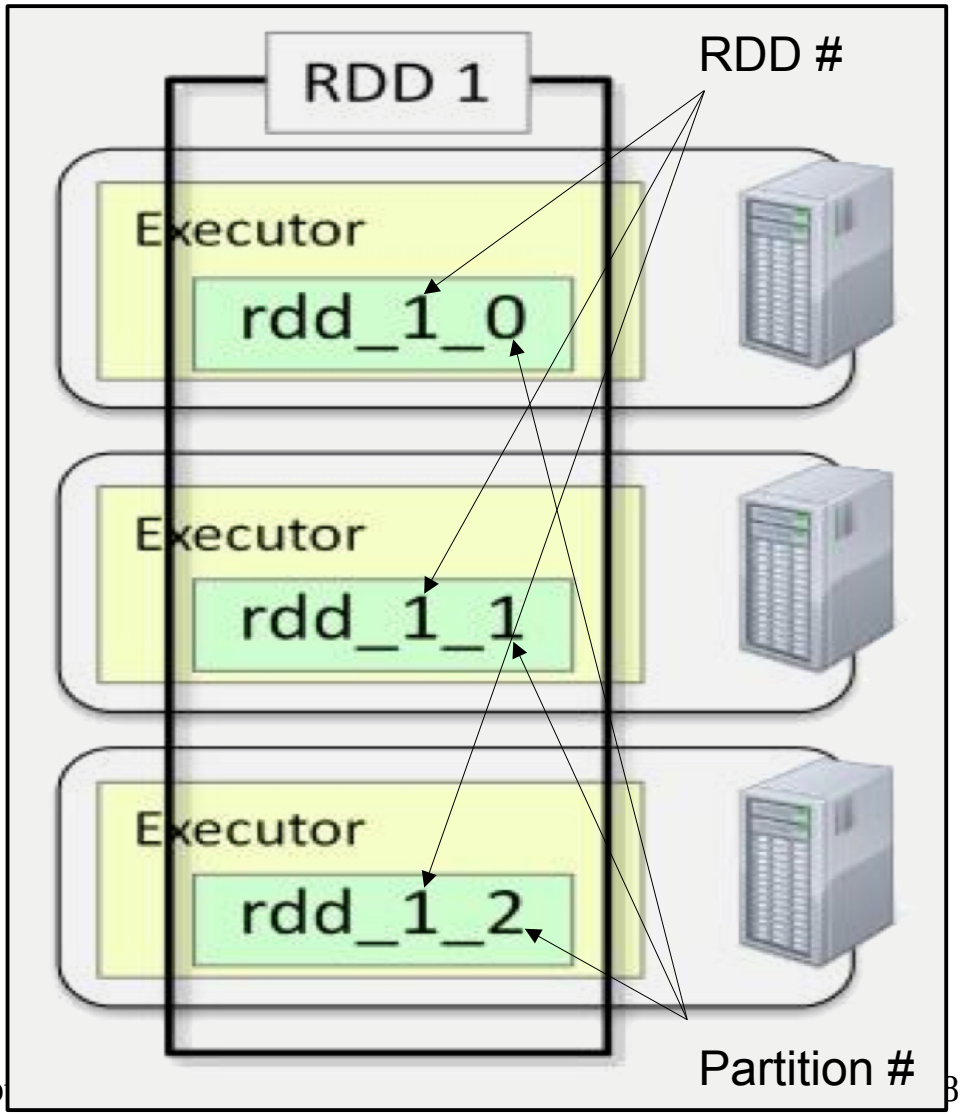
- Our cluster is administered by 2 Master daemons
  - Cluster Master Node (e.g. Spark Standalone)
    - Manages the Executors within which tasks run on behalf of the client
  - HDFS Master Node (NameNode)
    - Manages the HDFS file system, tracking
      - Each file in the system
      - The blocks mapped to each file in the system
      - The location of each block in the system

Chapter Topics

- Review: Spark On A Cluster
- RDD Partitions
- Partitioning Of File-Based RDDS
- HDFS And Data Locality
- Hands-On Exercise: Working With Partitions
- Executing Parallel Operations
- Stages And Tasks
- Summary
- Review
- References
- Hands-On Exercise: Viewing Stages And Tasks In The Spark Application UI

RDDs On A Cluster

- Resilient Distributed Datasets
  - Live in memory
  - Data is automatically partitioned across worker nodes, so that
    - Datasets too large to fit in the memory of a single node can still be processed
    - Data can be processed locally on the node where it lives
- Partitioning is done automatically by Spark
  - If the data is being read from HDFS, each partition usually correlates with a block
  - But for dynamically generated data, you can control how many partitions are created



Chapter Topics

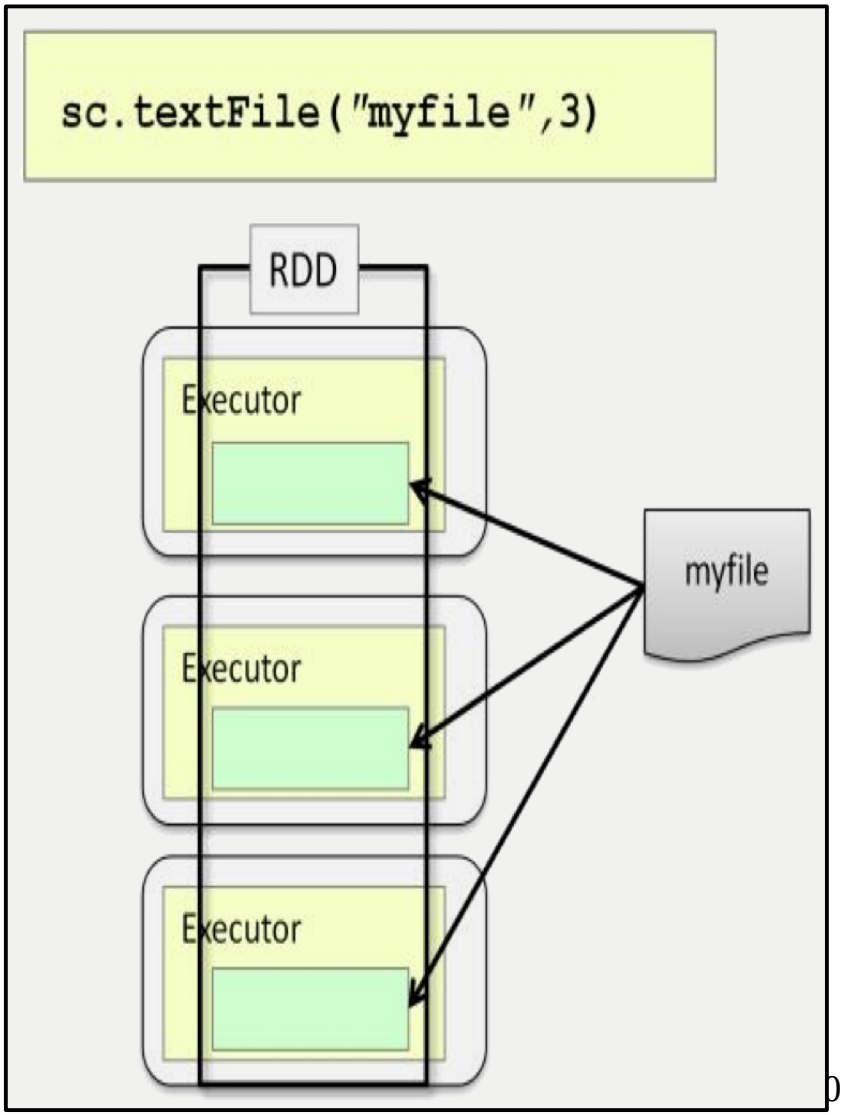
- Review: Spark On A Cluster
- RDD Partitions
- Partitioning Of File-Based RDDS
- HDFS And Data Locality
- Hands-On Exercise: Working With Partitions
- Executing Parallel Operations
- Stages And Tasks
- Summary
- Review
- References
- Hands-On Exercise: Viewing Stages And Tasks In The Spark Application UI

File Partitioning: Single Files

- When creating a partition from a single file
  - The number of partitions is generally based on the file size
    - One partition per HDFS block
  - You can optionally specify a minimum number of partitions
    - The default minimum is 2 partitions, even if the single core of thread

```
sc.textFile("filename")       // 2 Partitions
sc.textFile("filename", 4)    // 4 Partitions
```

  - More partitions mean more parallelization
- You should partition even small datasets to make sure your program works identically regardless of whether or not the data is partitioned
  - e.g. Some operations might yield different results with multiple partitions than with a single partition
- A partition is analogous to an InputSplit in MapReduce
  - A chunk of data processed by a single executor



Loading Multiple Files (1 Of 3)

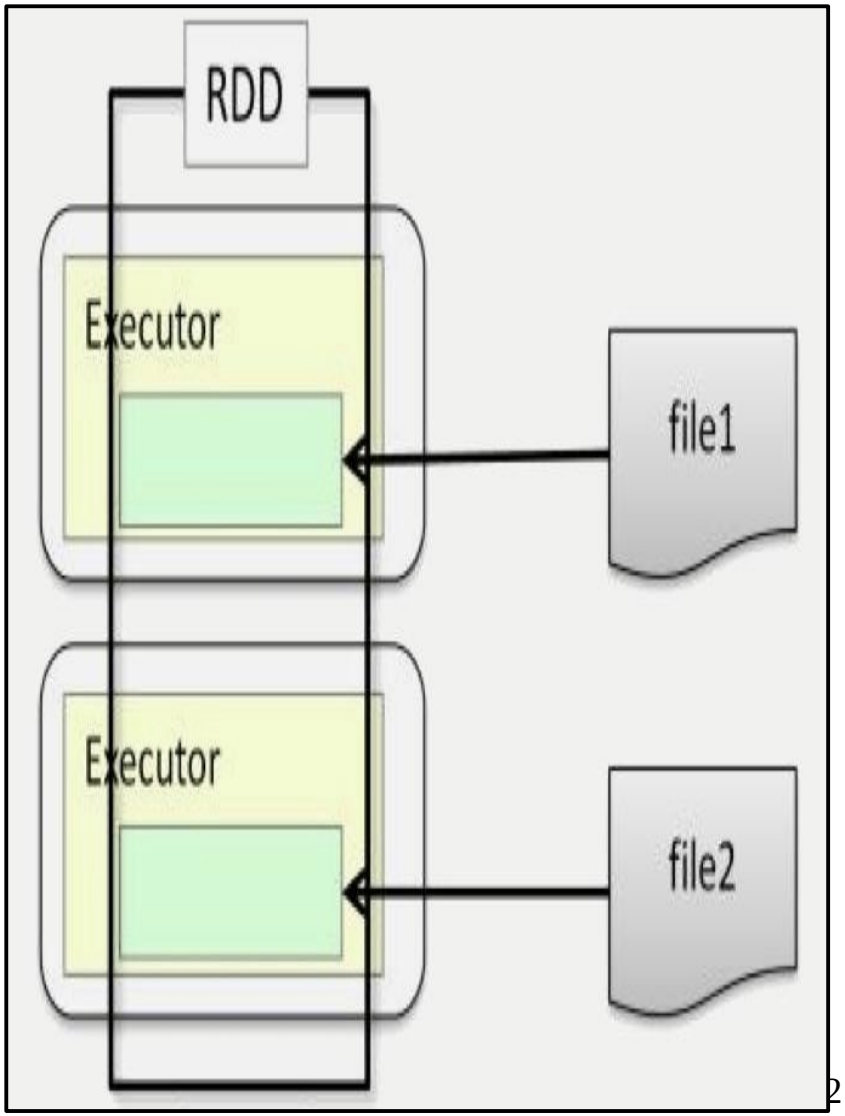
- Recall that there are two approaches to loading multiple files, one for large files or files of varying or unknown size, the other for small files
  - In general, Cloudera recommends not putting small files into HDFS
    - Remember, HDFS is optimized for handling a “moderate” number of large files
  - But sometimes, you may have no choice in the matter, so the question becomes, if you must store small files in HDFS, how might you process those files efficiently?

File Partitioning: Multiple Files (2 Of 3)

- Large files or files of varying or unknown size

```
sc.textFile("mydir/*")
sc.textFile("mydir/file1, mydir/file2")
```

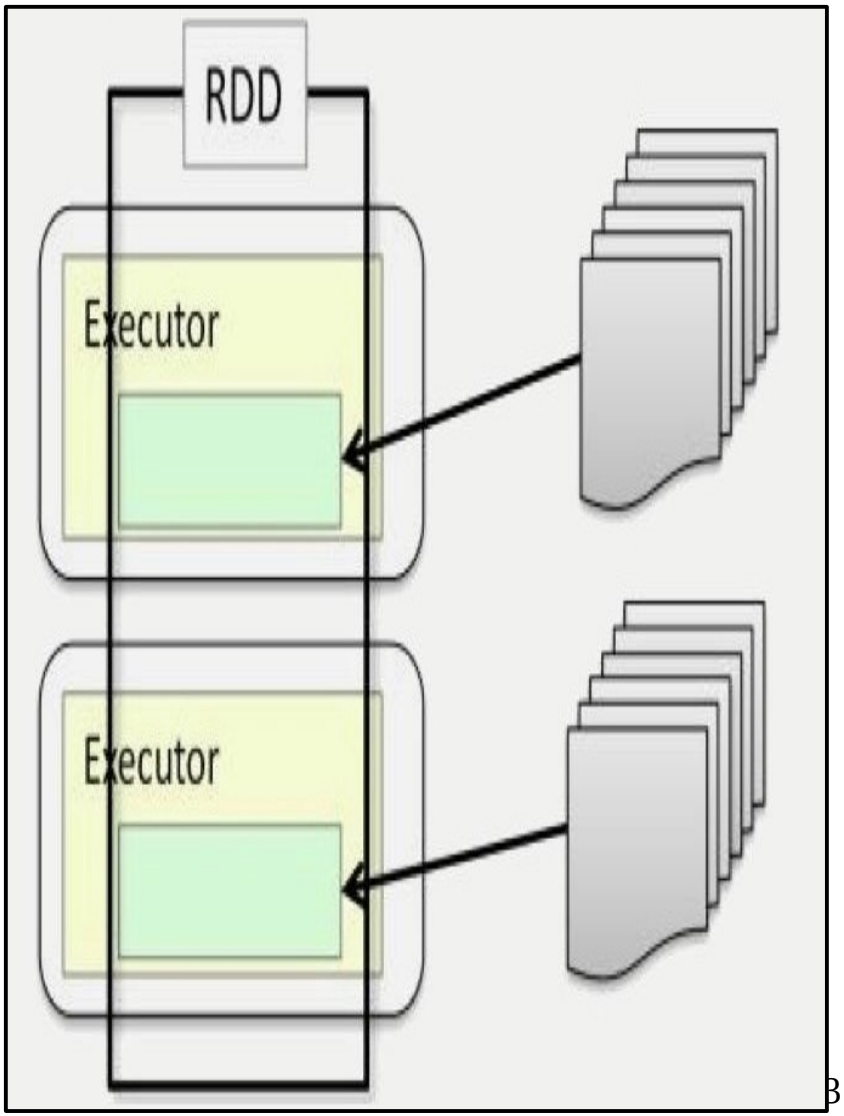
  - One partition
    - More, if the file size exceeds the default partition (HDFS block) size
  - File-based operations can be performed per partition
    - e.g. Processing a file as a unit (distinct from other files) rather than processing the contents of many files collectively



File Partitioning: Multiple Files (3 Of 3)

- Small files (each small enough to fit into a single record)
  - Creates a key-value PairRdd for each file, each of whose:
    - Key: File name
    - Value: File contents
    - Analogous to a Hadoop SequenceFile containing many small files
  - Puts the contents of many files in a single partition, rather than creating lots of small partitions, one per file, which turns out to be very inefficient
  - Also useful for working with records spanning multiple lines

```
sc.wholeTextFiles("mydir")
```



Operating On Partitions

- Most RDD operations work individually on each element of an RDD
  - e.g. map, flatMap, reduceByKey
- But a few work directly on the partition itself
  - foreachPartition
    - Calls a function for each partition (rather than for each element of the RDD)
    - Like forEach on an RDD in that it is neither a transformation (does not produce a new RDD) nor an action (does not return a value)
      - Its purpose is to produce a side effect
  - mapPartitions
    - Creates a new RDD by executing a function on each partition in the current RDD (rather than on each element of the RDD)
  - mapPartitionsWithIndex
    - Same as mapPartitions but includes the index of the RDD within the RDD
- Functions for partition operations take iterators as arguments, not elements

Example: Count JPG Requests Per File (1 Of 2)

```
> def countJpgs(index, partIter):
      jpgCount = 0
      for line in partIter:
          if "jpg" in line:
              jpgCount += 1
      yield(index, jpgCount)       # yield is not necessary.
> jpgCounts = sc.textFile("weblogs/*") \
      .mapPartitionsWithIndex(countJpgs)
```

```
> def countJpgs
      (index: Int, partIter: Iterator[String]):
          Iterator[(Int, Int)] = {
              var jpgCount = 0
              for (line <- partIter)
                  if (line.contains(".jpg"))
                      jpgCount += 1
              Iterator((index, jpgCount))
          }
> val jpgCounts = { sc.textFile("weblogs/*")
      .mapPartitionsWithIndex(countJpgs) }
```

```
(0, 237)
(1, 132)
(2, 188)
(3, 193)
...
```

Example: Count JPG Requests Per File (2 Of 2)

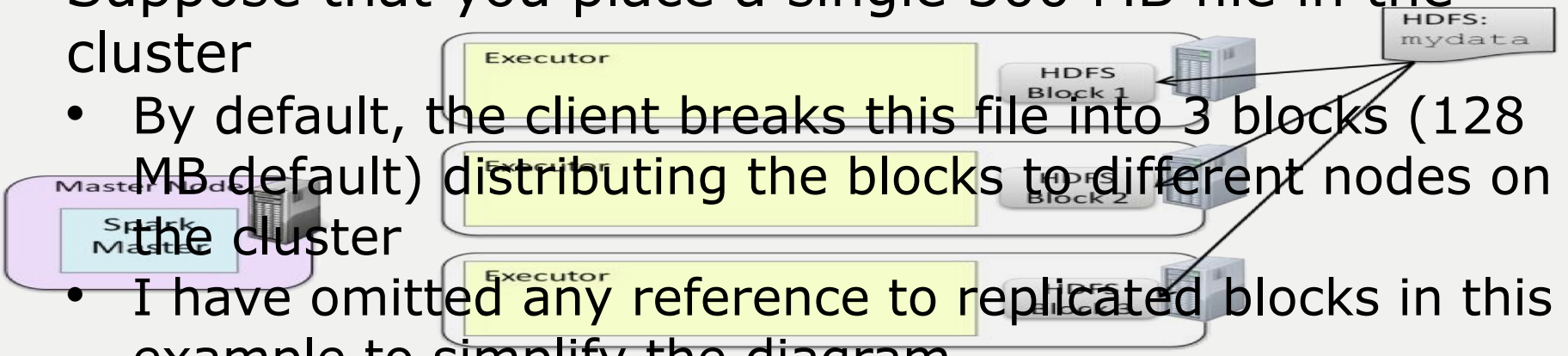
- This example counts the number of JPG requests represented in a set of log files
  - Each file in its own partition
- When might partition based operations like this be useful?
  - For operations that might be prohibitively expensive when applied at the element level
    - e.g. Opening/closing a resource once per partition to process many elements in a batch process, rather than opening/closing the resource once per element
  - For subgroup analysis if each file represents a different subgroup
  - Working with multiple elements of an RDD at once, when those elements reside in the same partition

Chapter Topics

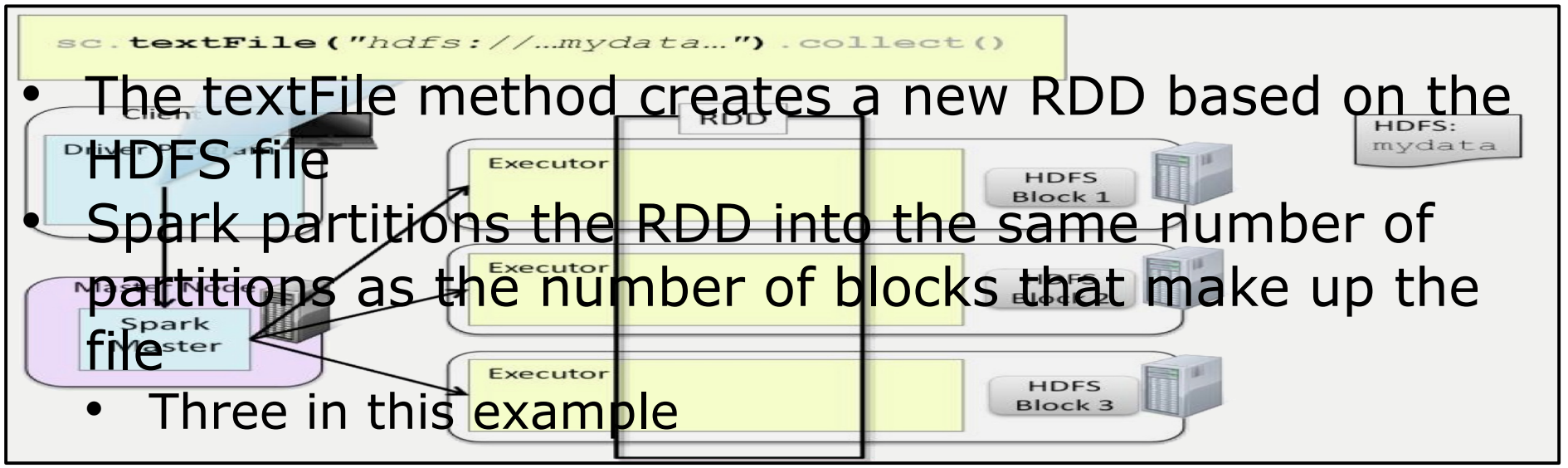
- Review: Spark On A Cluster
- RDD Partitions
- Partitioning Of File-Based RDDS
- HDFS And Data Locality
- Hands-On Exercise: Working With Partitions
- Executing Parallel Operations
- Stages And Tasks
- Summary
- Review
- References
- Hands-On Exercise: Viewing Stages And Tasks In The Spark Application UI

HDFS And Data Locality (1 Of 4)

- Suppose that you place a single 300 MB file in the cluster
  - By default, the client breaks this file into 3 blocks (128 MB default) distributing the blocks to different nodes on the cluster
  - I have omitted any reference to replicated blocks in this example to simplify the diagram

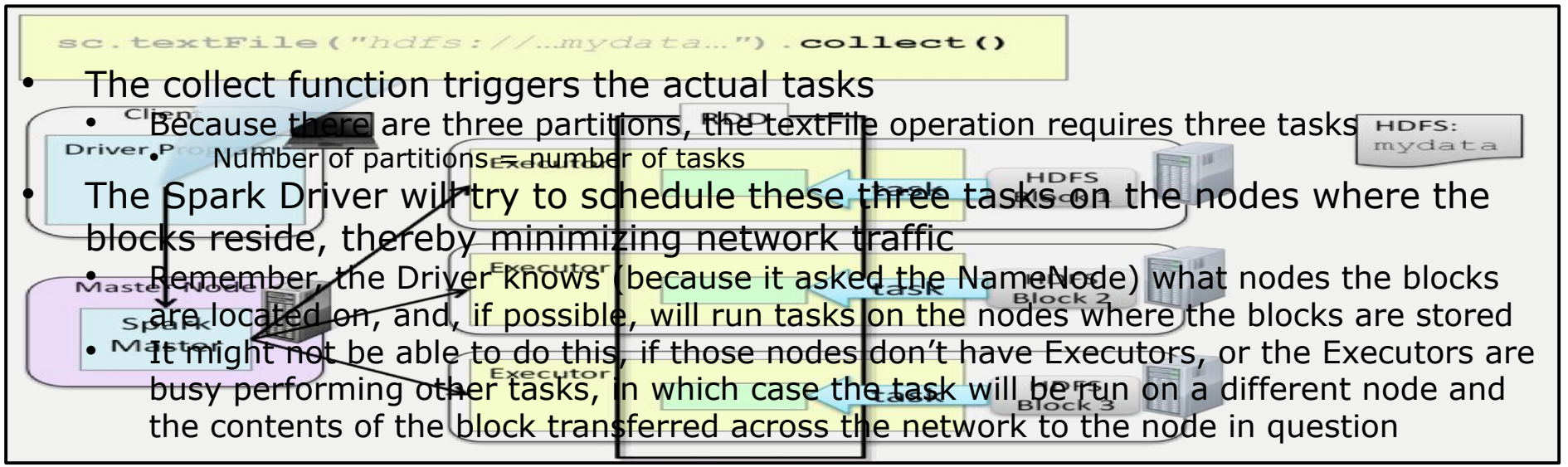


HDFS And Data Locality (2 Of 4)



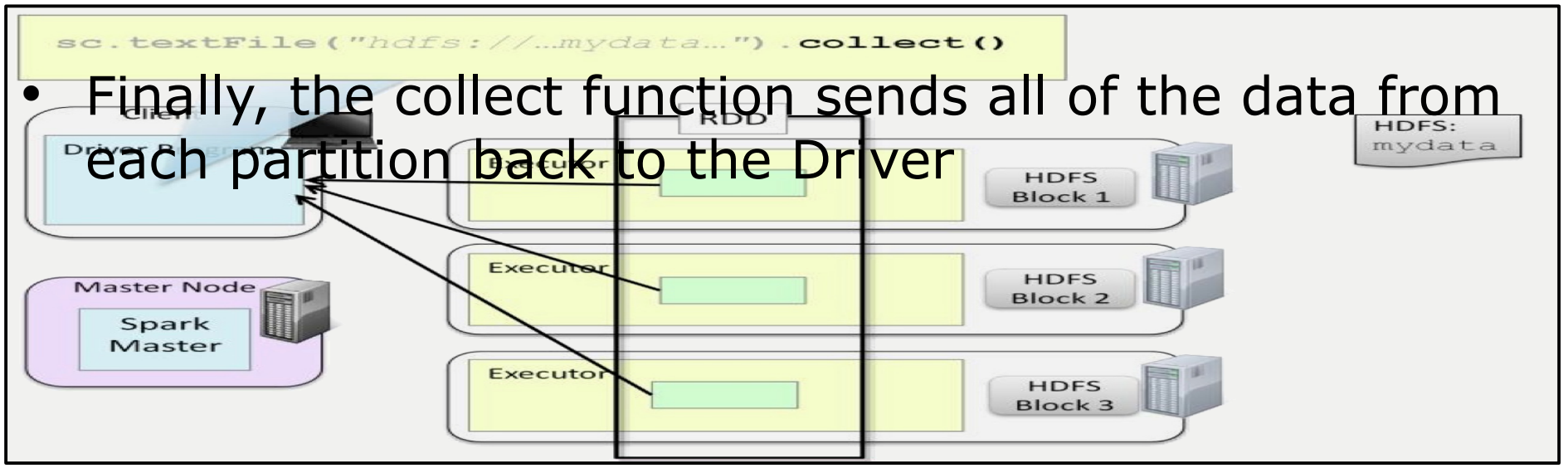
- The textFile method creates a new RDD based on the HDFS file
- Spark partitions the RDD into the same number of partitions as the number of blocks that make up the file
  - Three in this example

HDFS And Data Locality (3 Of 4)



- The collect function triggers the actual tasks
  - Because there are three partitions, the textFile operation requires three tasks
    - Number of partitions = number of tasks
- The Spark Driver will try to schedule these three tasks on the nodes where the blocks reside, thereby minimizing network traffic
  - Remember, the Driver knows (because it asked the NameNode) what nodes the blocks are located on, and, if possible, will run tasks on the nodes where the blocks are stored
  - It might not be able to do this, if those nodes don't have Executors, or the Executors are busy performing other tasks, in which case the task will be run on a different node and the contents of the block transferred across the network to the node in question

HDFS And Data Locality (4 Of 4)



- Finally, the collect function sends all of the data from each partition back to the Driver

Chapter Topics

- Review: Spark On A Cluster
- RDD Partitions
- Partitioning Of File-Based RDDS
- HDFS And Data Locality
- Hands-On Exercise: Working With Partitions
- Executing Parallel Operations
- Stages And Tasks
- Summary
- Review
- References
- Hands-On Exercise: Viewing Stages And Tasks In The Spark Application UI

Hands-On Exercise

- Working With Partitions
  - Parse multiple small XML files containing device activation records
    - Use the XML parsing functions provided in the exercise stubs
  - Find the most common device models in the dataset
- Please refer to the Hands-On Exercise Manual

Chapter Topics

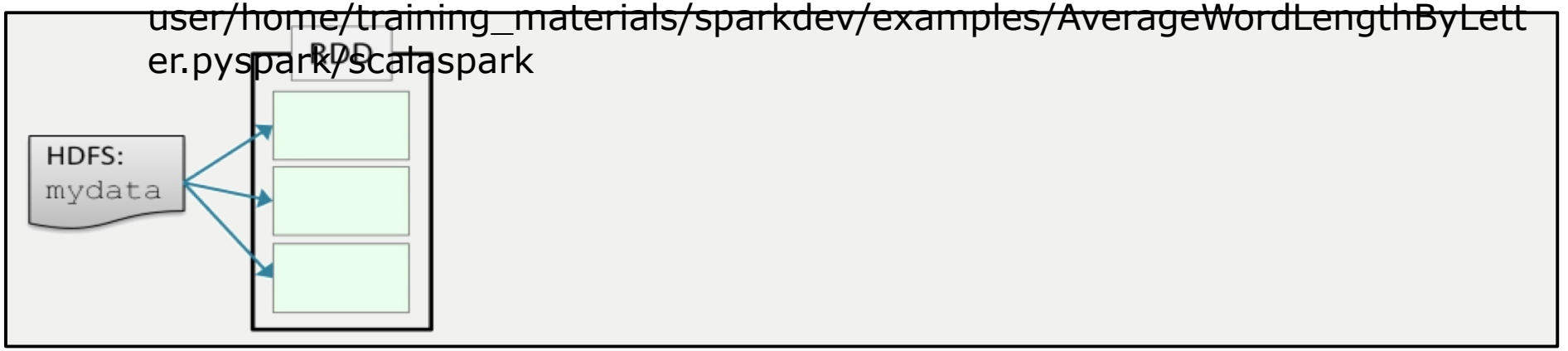
- Review: Spark On A Cluster
- RDD Partitions
- Partitioning Of File-Based RDDS
- HDFS And Data Locality
- Hands-On Exercise: Working With Partitions
- Executing Parallel Operations
- Stages And Tasks
- Summary
- Review
- References
- Hands-On Exercise: Viewing Stages And Tasks In The Spark Application UI

Parallel Operations On Partitions

- RDD operations are executed in parallel on each partition
  - When possible, tasks execute on the worker nodes on which the data already resides
- Some operations preserve partitioning
  - e.g., map, flatMap, filter
  - When all of the input data comes from a single partition
  - The output partition typically resides on the same node as its input partition
- Some operations repartition (shuffle data)
  - e.g., reduceByKey, sortByKey, groupByKey
  - When the input data comes from multiple partitions (distributed across multiple nodes)

Example: Average Word Length By Letter (1 Of 5)

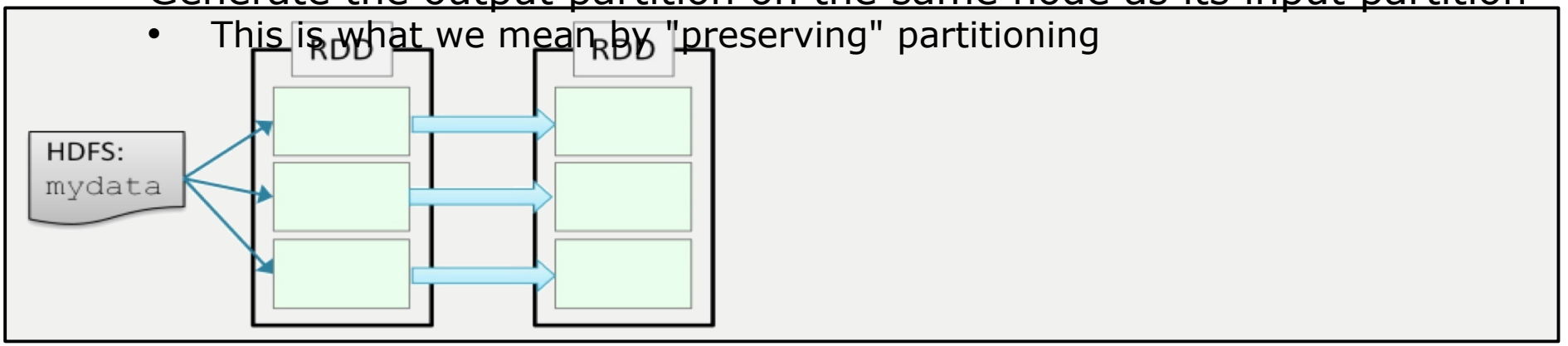
- `> avglens = sc.textFile("fileName")`
- Start by reading a file
  - This creates 1 partition per HDFS block (or more if requested by the client)
- Source Code:
  - /user/home/training_materials/sparkdev/examples/AverageWordLengthByLetter.pyspark/scalaspark



Example: Average Word Length By Letter (2 Of 5)

```
> avglens = sc.textFile("fileName") \
    .flatMap(lambda line: line.split())
```

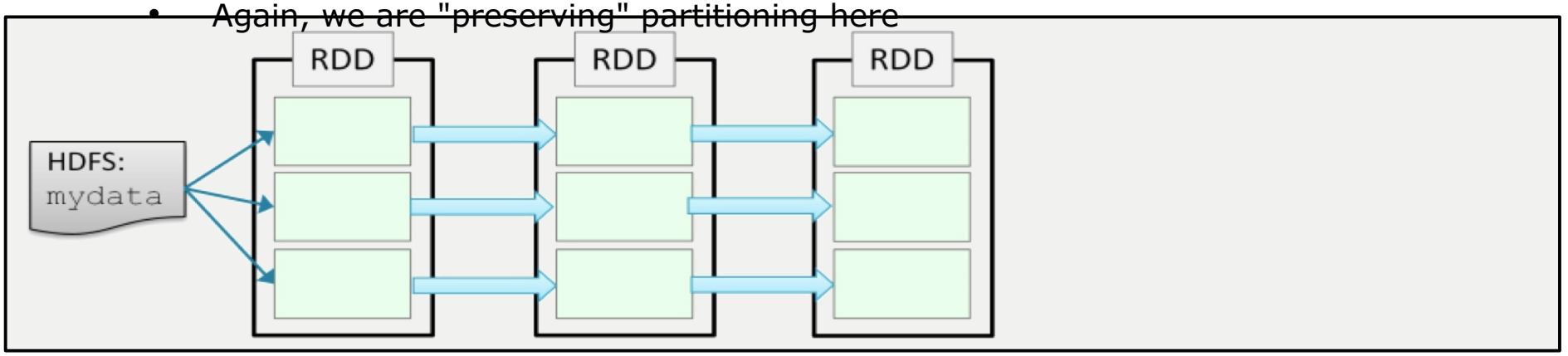
- flatMap each partition in parallel on each node
  - In this instance, processing one line at a time on each node, splitting each line into words
  - Generate the output partition on the same node as its input partition
    - This is what we mean by "preserving" partitioning



Example: Average Word Length By Letter (3 Of 5)

```
> avglens = sc.textFile("fileName") \
  .flatMap(lambda line: line.split()) \
  .map(lambda word: (word[0], len(word)))
```

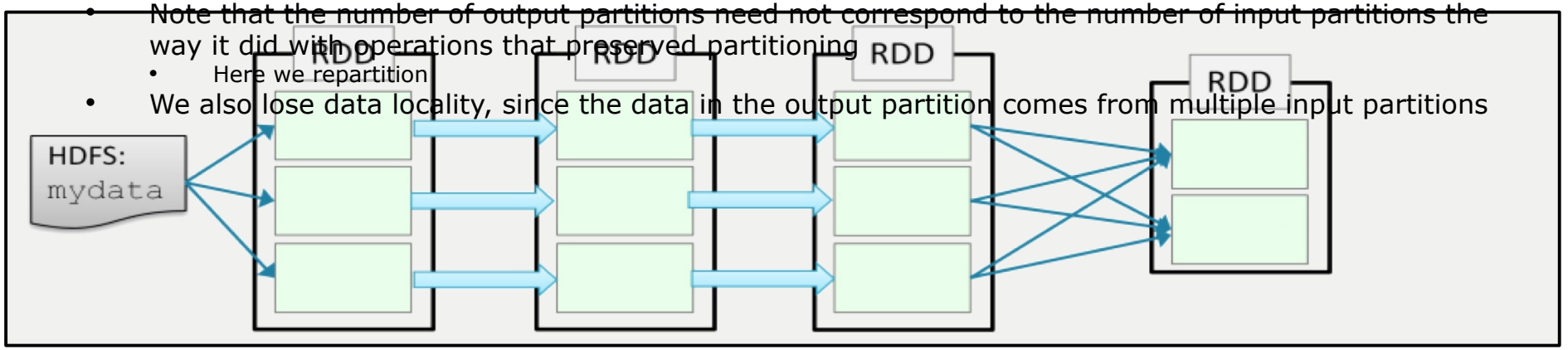
- Map each partition in parallel on each node
  - In this instance, operating one word at a time on each node, emitting the first letter of the word as a key and the length of the word as its value
  - Generate the output partition on the same node as its input partition
    - Again, we are "preserving" partitioning here



Example: Average Word Length By Letter (4 Of 5)

```
> avglens = sc.textFile("fileName") \
    .flatMap(lambda line: line.split()) \
    .map(lambda word: (word[0], len(word))) \
    .groupByKey()
```

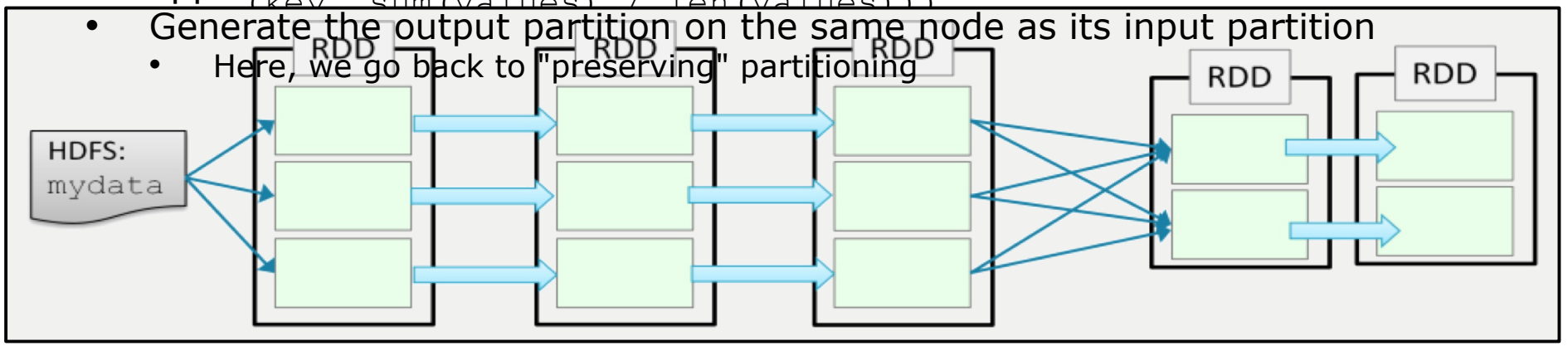
- Group the key/value pairs by their key (the first letter of each word)
  - GroupByKey differs from its predecessors in that it is a reducer, combining data from different partitions into a single partition in a process known as the "Shuffle"
  - In this instance, group the values (word lengths) by the first letter in each word (key) to improve the efficiency of the subsequent calculation
  - Note that the number of output partitions need not correspond to the number of input partitions the way it did with operations that preserved partitioning
    - Here we repartition
  - We also lose data locality, since the data in the output partition comes from multiple input partitions



Example: Average Word Length By Letter (5 Of 5)

```
> avglens = sc.textFile("fileName") \
    .flatMap(lambda line: line.split()) \
    .map(lambda word: (word[0], len(word))) \
    .groupByKey() \
    .map(lambda (key, values): \
      (key, sum(values) / len(values)))
```

- Map each "grouped" partition in parallel on each node
  - In this instance, dividing the sum of all of the lengths of words beginning with a particular letter of the alphabet by the frequency with which those words appeared
  - Generate the output partition on the same node as its input partition
    - Here, we go back to "preserving" partitioning



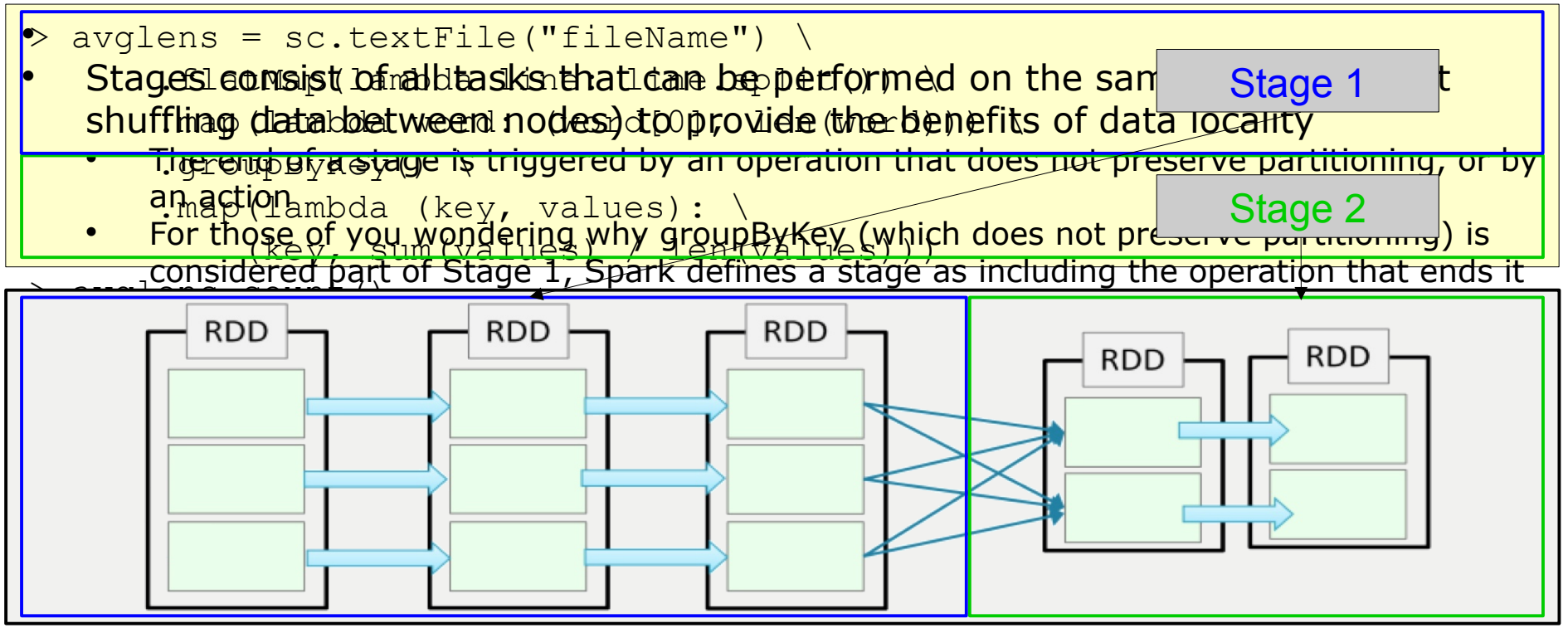
Chapter Topics

- Review: Spark On A Cluster
- RDD Partitions
- Partitioning Of File-Based RDDS
- HDFS And Data Locality
- Hands-On Exercise: Working With Partitions
- Executing Parallel Operations
- Stages And Tasks
- Summary
- Review
- References
- Hands-On Exercise: Viewing Stages And Tasks In The Spark Application UI

Stages

- Operations that can run on the same partition are executed in a single "stage"
  - In contrast, operations that repartition run in new stages
- Tasks within a stage are pipelined together
  - Executed sequentially on a single RDD element before the next RDD element is processed
- Developers should be aware of stages, because they offer opportunities to optimize performance

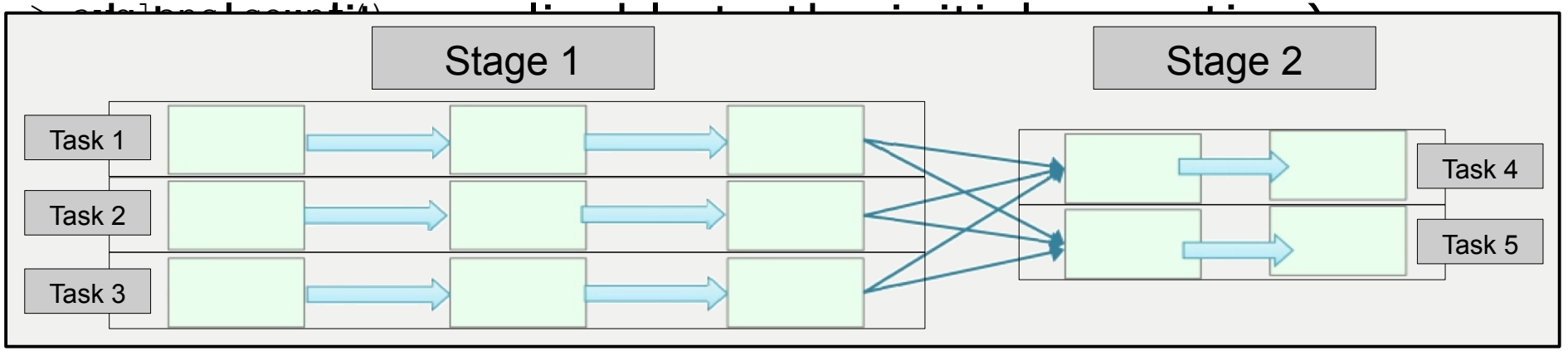
Spark Execution Stages (1 Of 5)



- Stages consist of all tasks that can be performed on the same partition of data without shuffling data between nodes, to provide the benefits of data locality
  - The end of a stage is triggered by an operation that does not preserve partitioning, or by an action
  - For those of you wondering why groupByKey (which does not preserve partitioning) is considered part of Stage 1, Spark defines a stage as including the operation that ends it

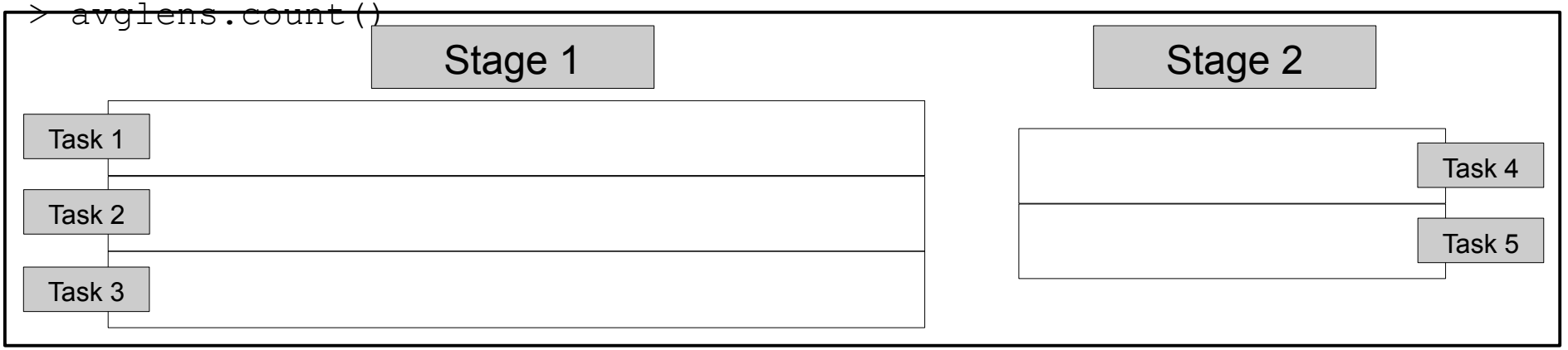
Spark Execution Stages (2 Of 5)

```
> avglens = sc.textFile("fileName") \
      .flatMap(lambda line: line.split()) \
      .map(lambda word: (word[0], len(word))) \
      .groupByKey() \
      .map(lambda (key, values): \
          (key, sum(values) / len(values)))
```



Spark Execution Stages (3 Of 5)

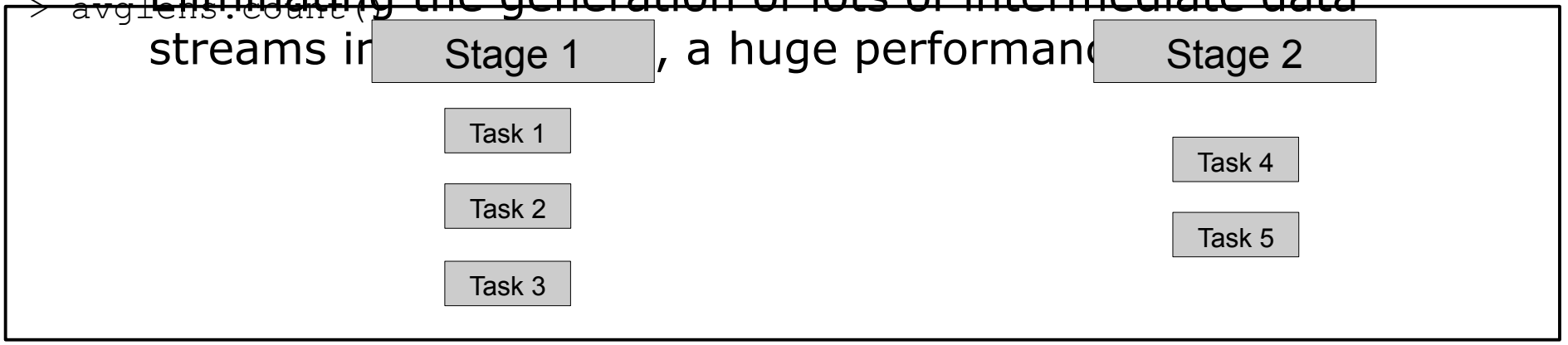
```
> avglens = sc.textFile("fileName") \
      .flatMap(lambda line: line.split()) \
      .map(lambda word: (word[0], len(word))) \
      .groupByKey() \
      .map(lambda (key, values): \
          (key, sum(values) / len(values)))
> avglens.count()
```



Spark Execution Stages (4 Of 5)

```
> avglens = sc.textFile("fileName") \
      .flatMap(lambda line: line.split()) \
      .map(lambda word: (word[0], len(word))) \
      .groupByKey() \
      .map(lambda (key, values): \
          (key, sum(values) / len(values)))
> avglens.count()
```

- Eliminating the generation of lots of intermediate data streams in , a huge performanc



Spark Execution Stages (5 Of 5)

- Think of it this way
  - You write (pseudo)code like this

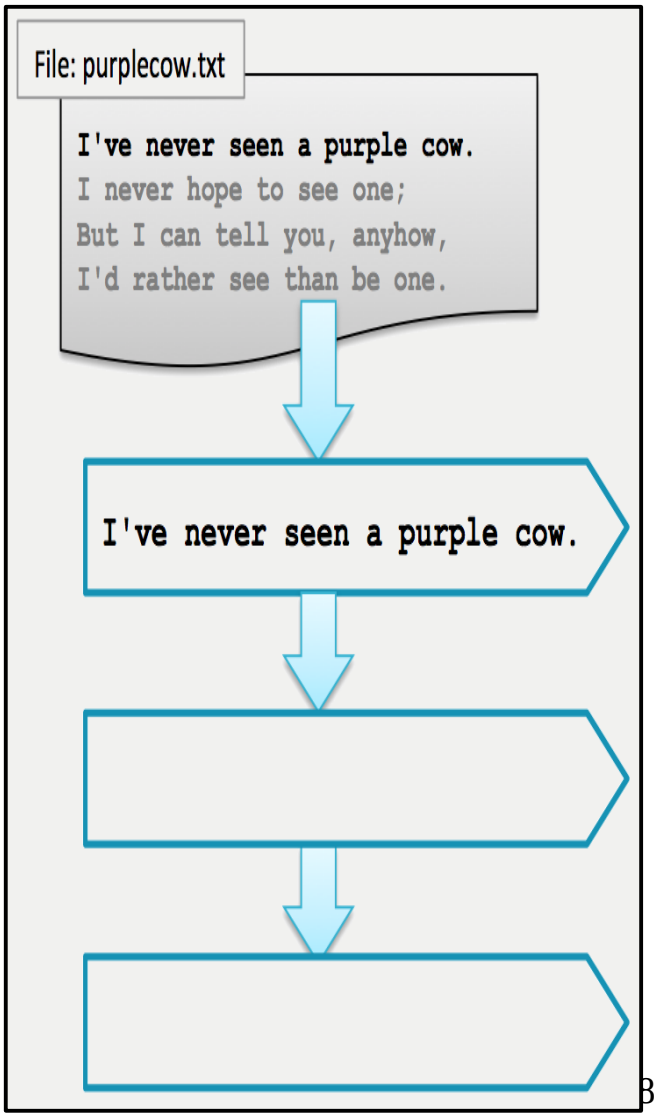
```
mapSquare(input)                          # Input is a list of numbers.
    declare output                        # Output is a list of numbers.
    foreach datum in input                # Square the list members in 1st pass.
        output.append(square(datum))
    return output
mapSqrt(input)
    declare output
    foreach datum in input                # Sqrt the list members in 2nd pass.
        output.append(sqrt(datum))
    return output

declare input
declare output
load input
output = mapSqrt(mapSquare(input))
```

```
declare input
declare output
load input
foreach datum in input
    output.append(sqrt(square(datum)))    # Do both transformations in 1 pass.
```

Pipelining (1 Of 8)

- When possible, Spark will perform sequences of transformations by row so no data is stored

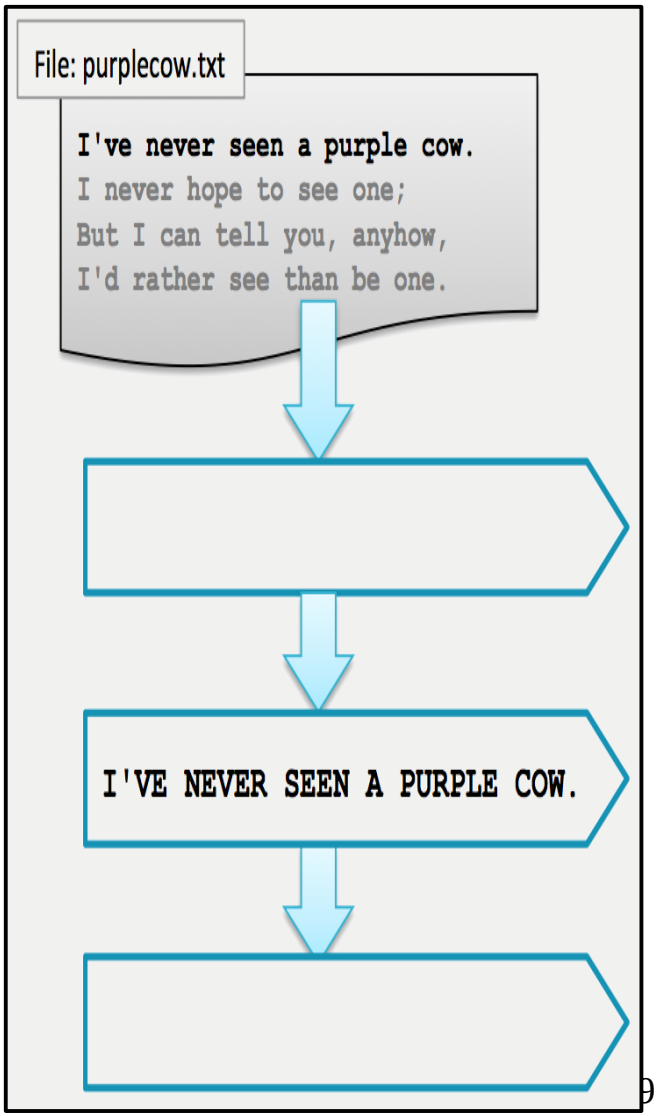
```
> val mydata_filt =
      sc.textFile("purplecow.txt")
> val mydata_uc =
      mydata.map(line => line.toUpperCase())
> val mydata_filt = mydata_uc
      .filter(line => line.startswith('I'))
> mydata_filt.take(2)
```



Pipelining (2 Of 8)

- When possible, Spark will perform sequences of transformations by row so no data is stored

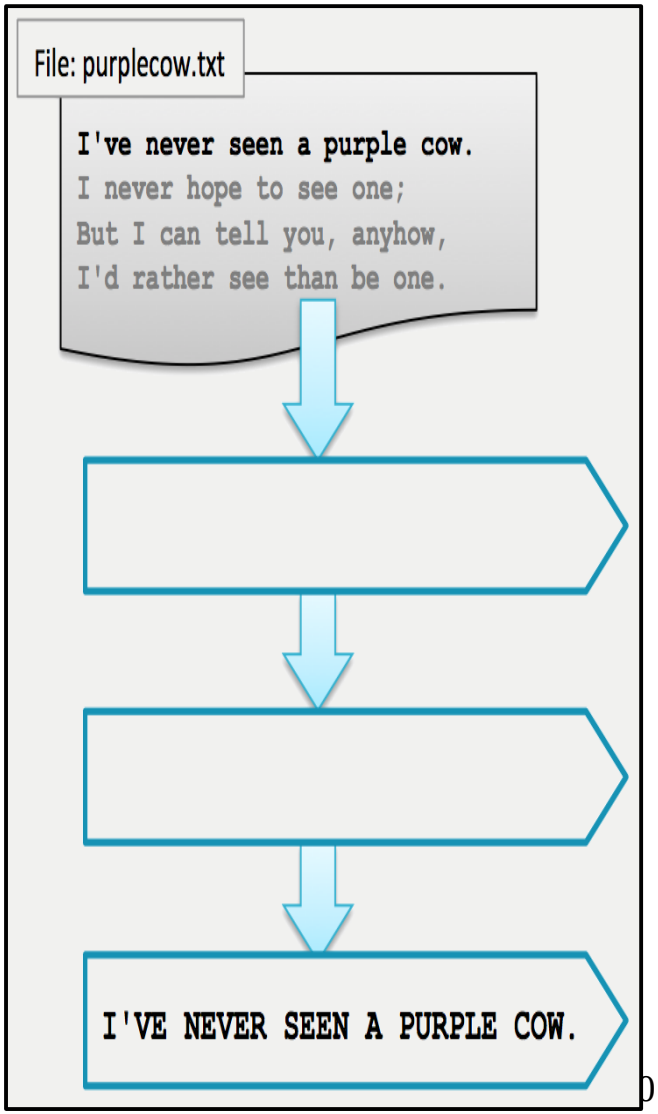
```
> val mydata_filt =
      sc.textFile("purplecow.txt")
> val mydata_uc =
      mydata.map(line => line.toUpperCase())
> val mydata_filt = mydata_uc
      .filter(line => line.startswith('I'))
> mydata_filt.take(2)
```



Pipelining (3 Of 8)

- When possible, Spark will perform sequences of transformations by row so no data is stored

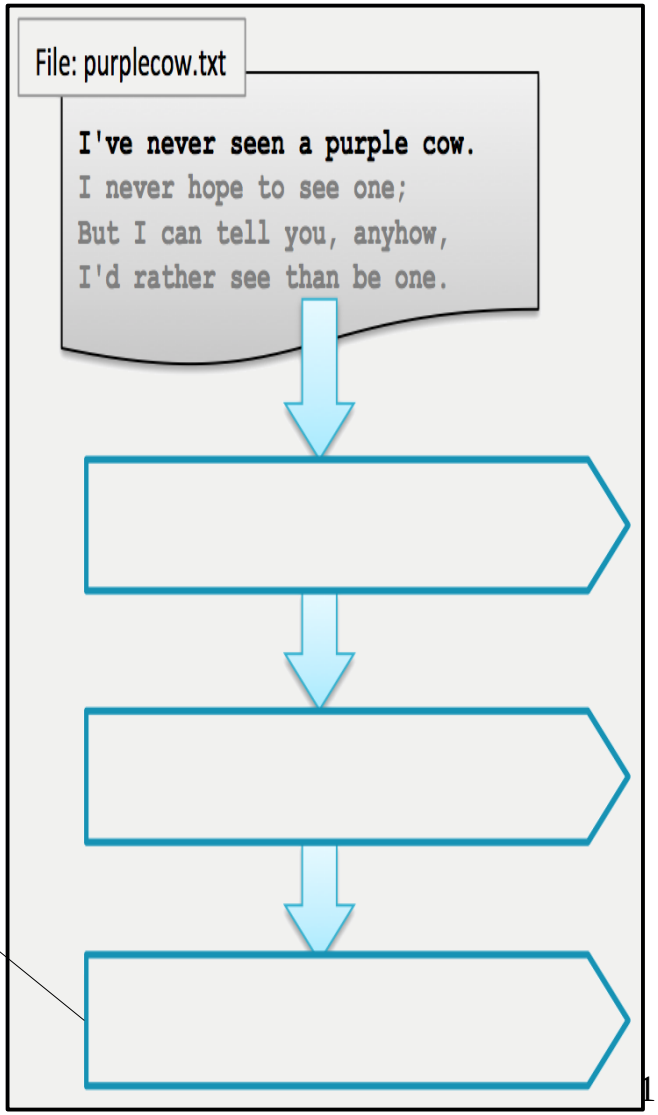
```
> val mydata_filt =
      sc.textFile("purplecow.txt")
> val mydata_uc =
      mydata.map(line => line.toUpperCase())
> val mydata_filt = mydata_uc
      .filter(line => line.startswith('I'))
> mydata_filt.take(2)
```



Pipelining (4 Of 8)

- When possible, Spark will perform sequences of transformations by row so no data is stored

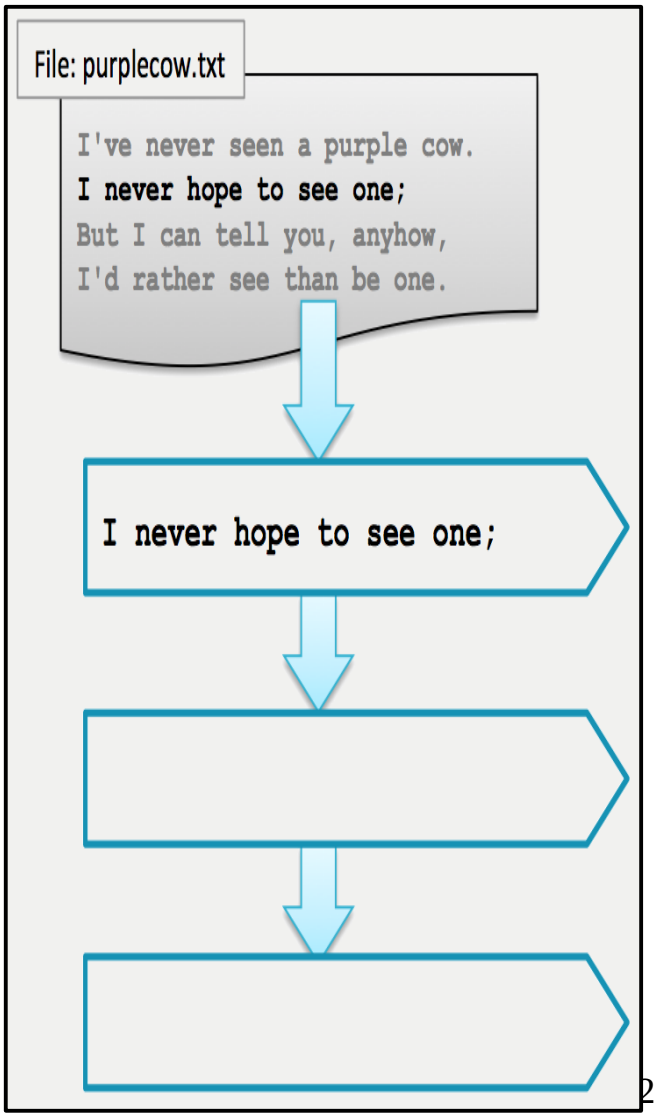
```
> val mydata_filt =
      sc.textFile("purplecow.txt")
> val mydata_uc =
      mydata.map(line => line.toUpperCase())
> val mydata_filt = mydata_uc
      .filter(line => line.startswith('I'))
> mydata_filt.take(2)
I'VE NEVER SEEN A PURPLE COW.
```



Pipelining (5 Of 8)

- When possible, Spark will perform sequences of transformations by row so no data is stored

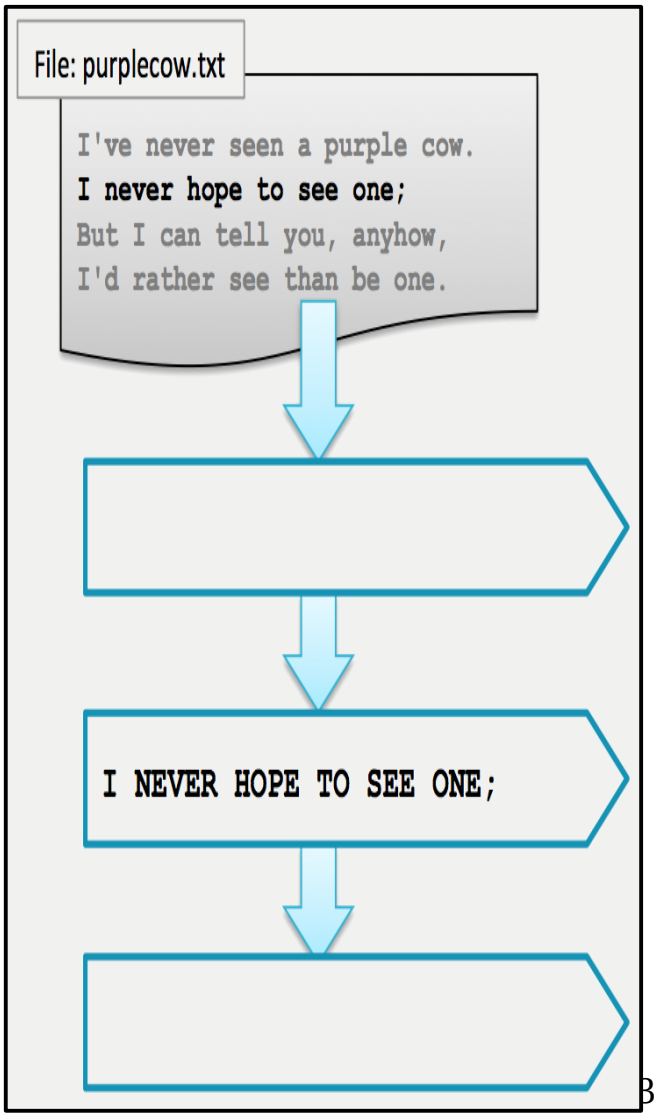
```
> val mydata_filt =
      sc.textFile("purplecow.txt")
> val mydata_uc =
      mydata.map(line => line.toUpperCase())
> val mydata_filt = mydata_uc
      .filter(line => line.startswith('I'))
> mydata_filt.take(2)
I'VE NEVER SEEN A PURPLE COW.
```



Pipelining (6 Of 8)

- When possible, Spark will perform sequences of transformations by row so no data is stored

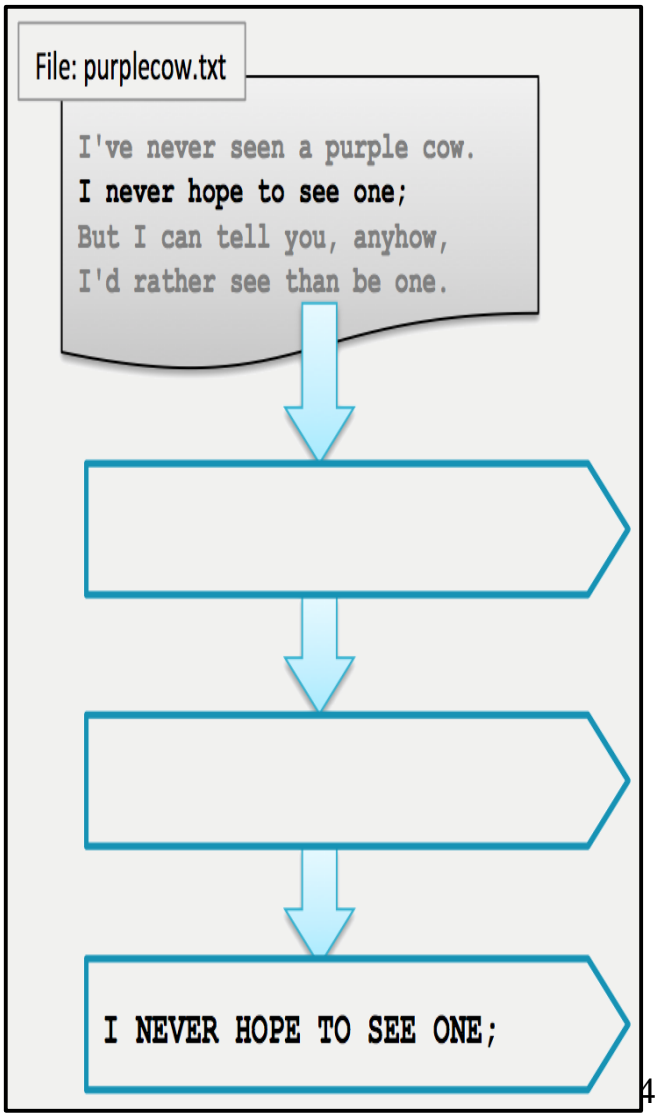
```
> val mydata_filt =
      sc.textFile("purplecow.txt")
> val mydata_uc =
      mydata.map(line => line.toUpperCase())
> val mydata_filt = mydata_uc
      .filter(line => line.startswith('I'))
> mydata_filt.take(2)
I'VE NEVER SEEN A PURPLE COW.
```



Pipelining (7 Of 8)

- When possible, Spark will perform sequences of transformations by row so no data is stored

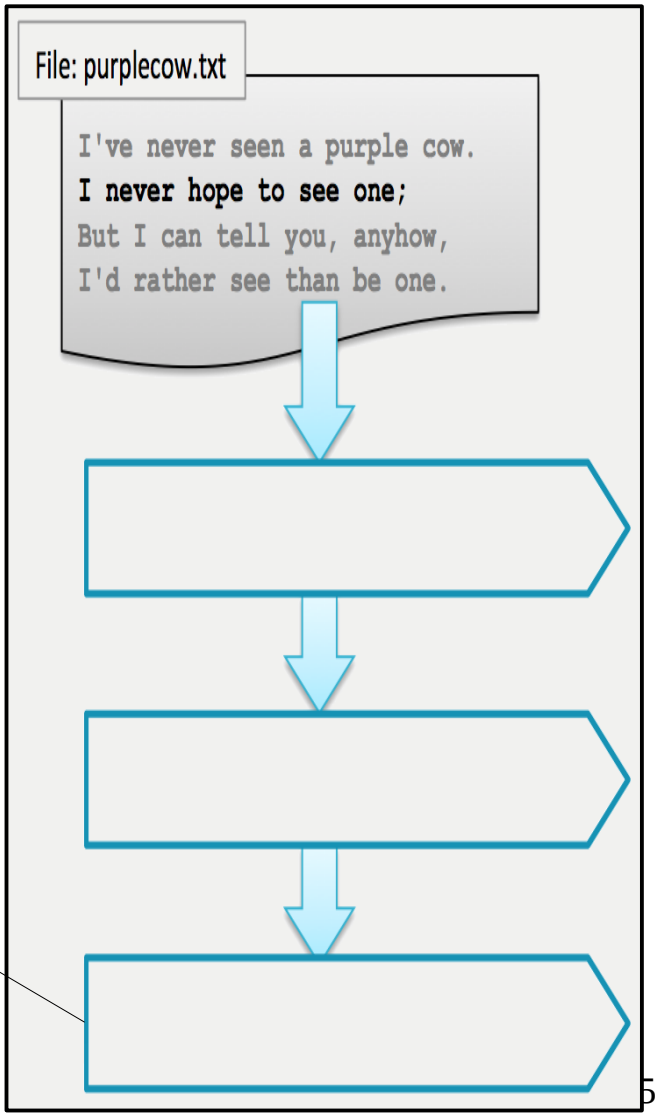
```
> val mydata_filt =
      sc.textFile("purplecow.txt")
> val mydata_uc =
      mydata.map(line => line.toUpperCase())
> val mydata_filt = mydata_uc
      .filter(line => line.startswith('I'))
> mydata_filt.take(2)
I'VE NEVER SEEN A PURPLE COW.
```



Pipelining (8 Of 8)

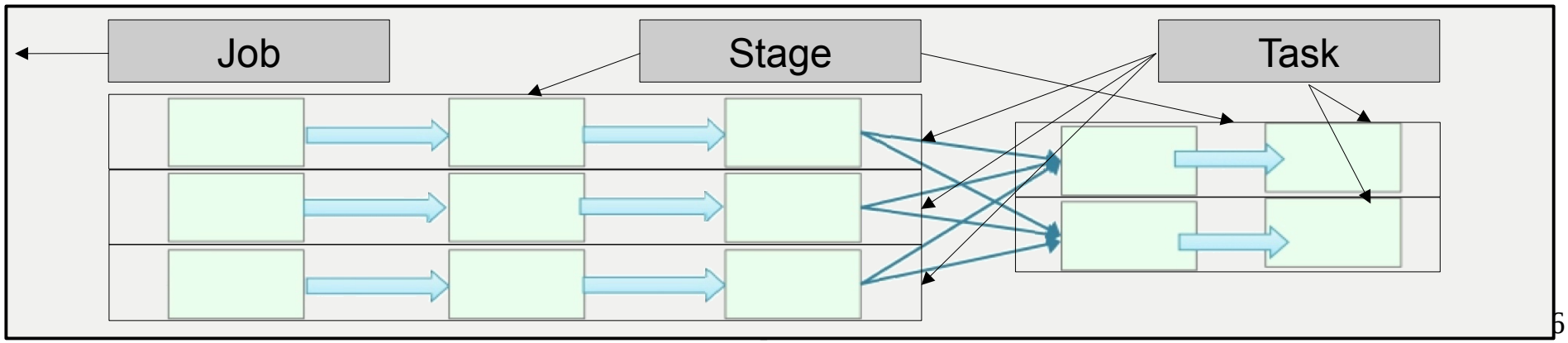
- When possible, Spark will perform sequences of transformations by row so no data is stored

```
> val mydata_filt =
      sc.textFile("purplecow.txt")
> val mydata_uc =
      mydata.map(line => line.toUpperCase())
> val mydata_filt = mydata_uc
      .filter(line => line.startswith('I'))
> mydata_filt.take(2)
I'VE NEVER SEEN A PURPLE COW.
I NEVER HOPE TO SEE ONE;
```



Summary Of Spark Terminology

- Application
  - A program that creates a Spark context, then performs transformations and actions on RDDs
- Job
  - A set of tasks, executed in response to an action
- Stage
  - A set of tasks in a job that can be executed in parallel
  - Partitioning of RDDs is preserved within a stage
- Task
  - An individual unit of work sent to a single executor



Contrasting Spark Terminology With Hadoop Terminology

- Application
  - In Hadoop an application consists of a single Job
  - In Spark an application consists of 1+ Jobs
- Job
  - In Hadoop, a Job is an instance of an Application, typically consisting of a single map phase and 0-1 reduce phases
  - In Spark, a Job is the set of stages/tasks required to return a value to the driver
    - In other words, a Job in Spark is an Action, possibly consisting of several map and reduce phases

How Spark Calculates Stages

- Spark constructs a DAG (Directed Acyclic Graph) (Tree) of RDD dependencies that fall into two categories
  - Narrow operations
    - Only one child depends on the RDD
      - Preserve partitioning
    - No shuffle required between nodes
    - Can be collapsed into a single stage
    - e.g. map, filter, union
  - Wide operations
    - Multiple children depend on the RDD
      - Do not preserve partitioning
    - Require shuffling between nodes
    - Define a new stage
    - e.g., reduceByKey, join, groupByKey

Viewing Stages Using toDebugString (Scala)

```
> val avglens = sc.textFile(myfile)
    .flatMap(line => line.split("\\W"))
    .map(word => (word(0),word.length))
    .groupByKey()
    .map(pair => (pair._1, pair._2.sum/pair._2.size.toDouble))

> avglens.toDebugString()

(2) MappedRDD[5] at map at …
 |  ShuffledRDD[4] at groupByKey at …
 +-(4) MappedRDD[3] at map at …
    |  FlatMappedRDD[2] at flatMap at …
    |  myfile MappedRDD[1] at textFile at …
    |  myfile HadoopRDD[0] at textFile at …
```

Stage 2

Stage 1

Indents indicate stages (shuffle boundaries)

Viewing Stages Using toDebugString (Python)

```
> avglens = sc.textFile(myfile) \
  .flatMap(lambda line: line.split()) \
  .map(lambda word: (word[0],len(word))) \
  .groupByKey() \
  .map(lambda (k, values): \
     (k, sum(values)/len(values)))

> print avglens.toDebugString()
(2) PythonRDD[13] at RDD at …
 |  MappedRDD[12] at values at …
 |  ShuffledRDD[11] at partitionBy at …
 +-(4) PairwiseRDD[10] at groupByKey at …
    |  PythonRDD[9] at groupByKey at …
    |  myfile MappedRDD[7] at textFile at …
    |  myfile HadoopRDD[6] at textFile at …
```

Stage 2

Stage 1

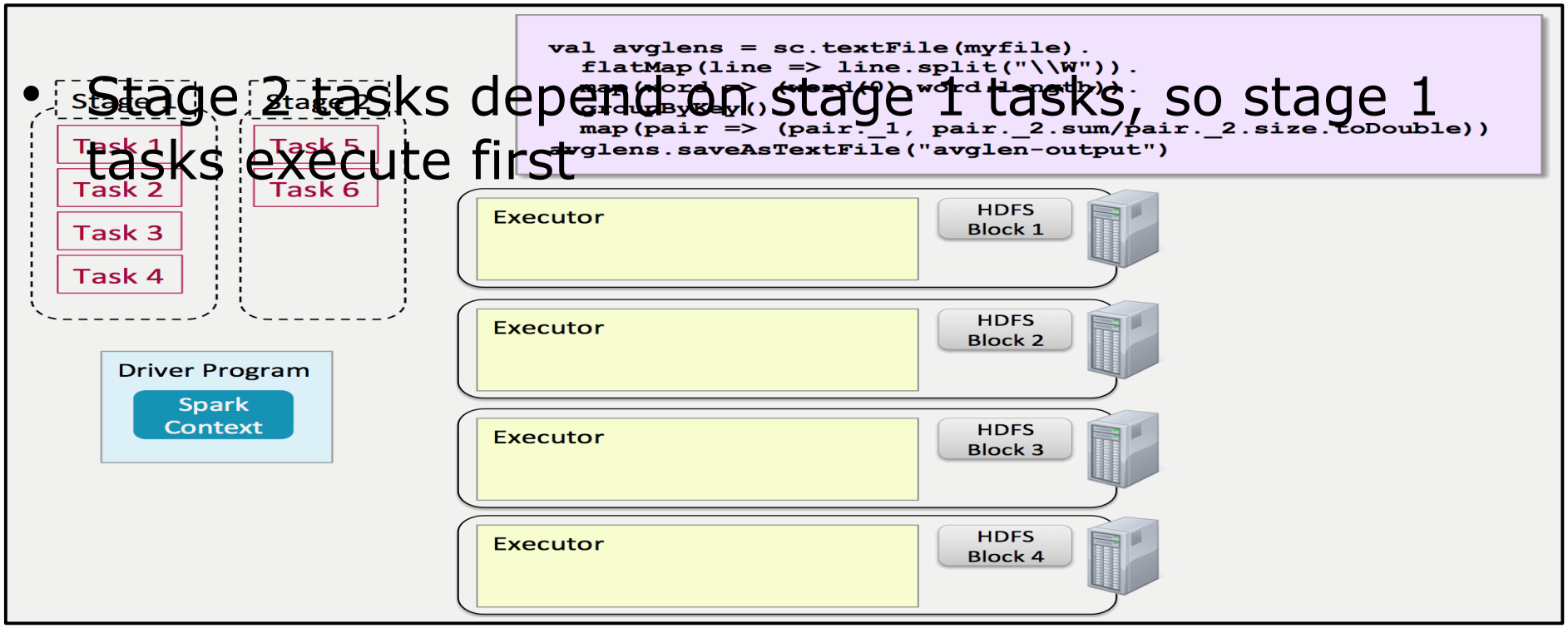
Indents indicate stages (shuffle boundaries)

The Big Idea (Optimization)

- Other things being equal, try to structure tasks to avoid the generation of new stages (which are more expensive computationally)
  - With the exception being when the outcome of the new stage reduces the size of the resulting RDD so much that subsequent processing improves substantially
- Or adjust the parallelism for Wide operations to improve throughput

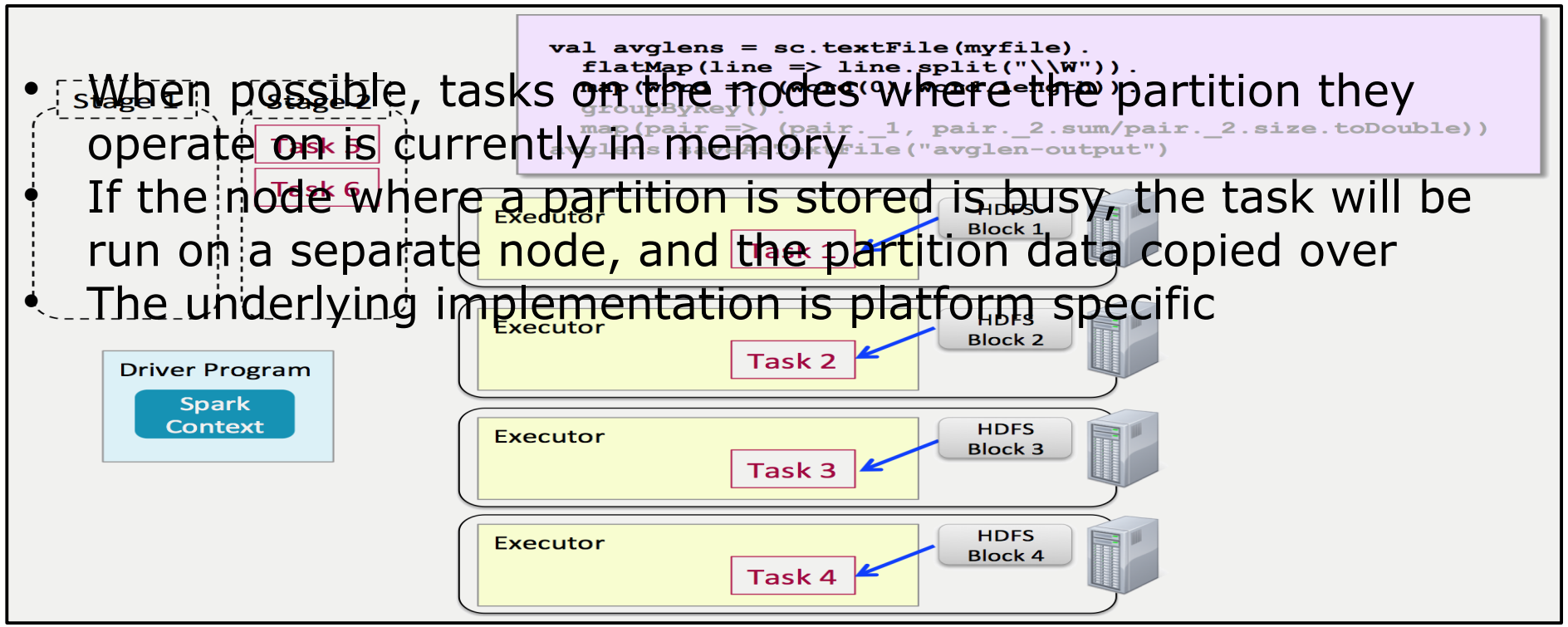
Spark Task Execution (1 Of 7)

- Stage 2 tasks depend on stage 1 tasks, so stage 1 tasks execute first



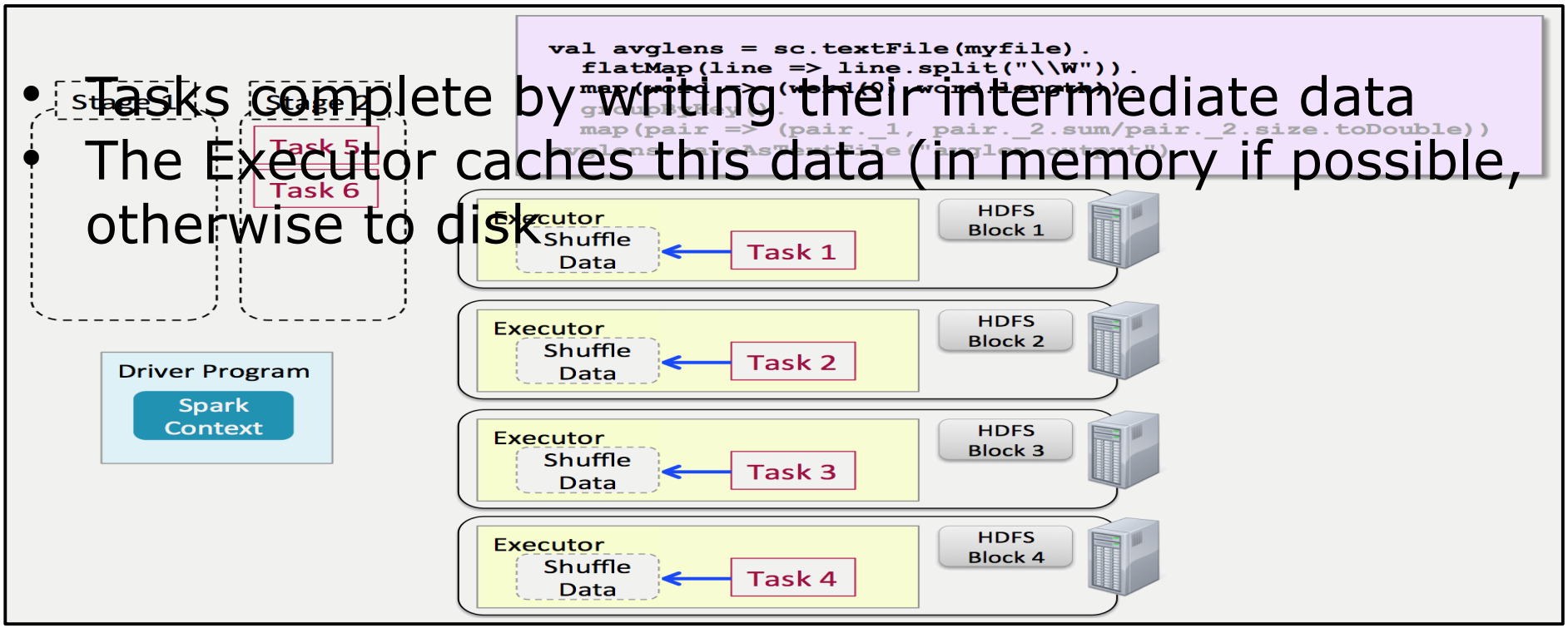
Spark Task Execution (2 Of 7)

- When possible, tasks on the nodes where the partition they operate on is currently in memory
- If the node where a partition is stored is busy, the task will be run on a separate node, and the partition data copied over
- The underlying implementation is platform specific



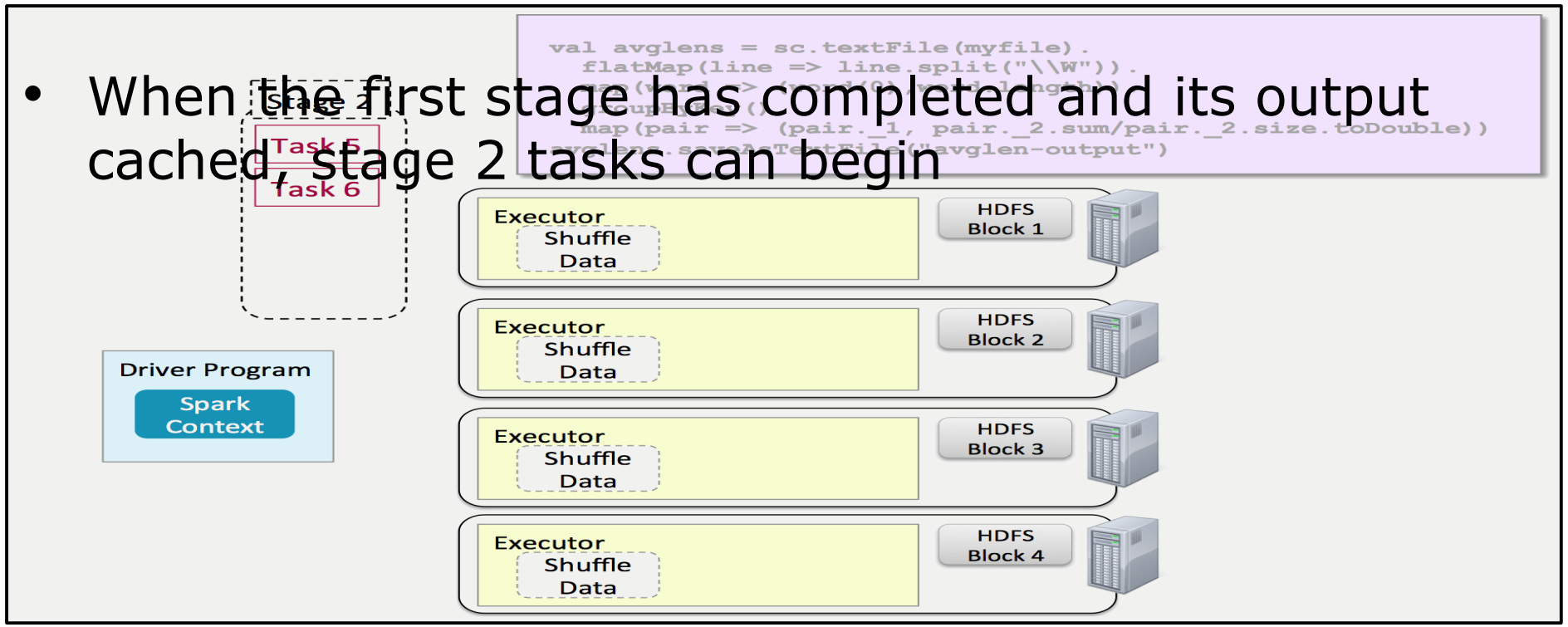
Spark Task Execution (3 Of 7)

- Tasks complete by writing their intermediate data
- The Executor caches this data (in memory if possible, otherwise to disk)



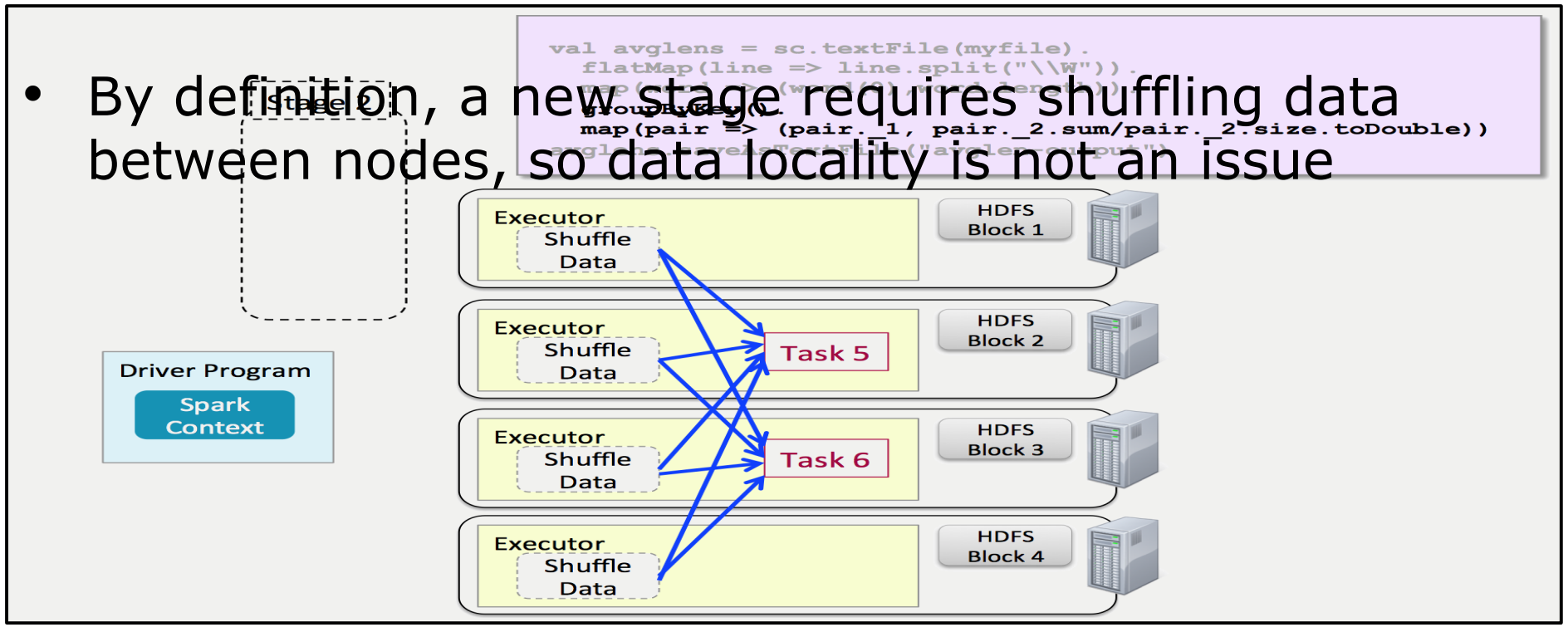
Spark Task Execution (4 Of 7)

- When the first stage has completed and its output cached, stage 2 tasks can begin



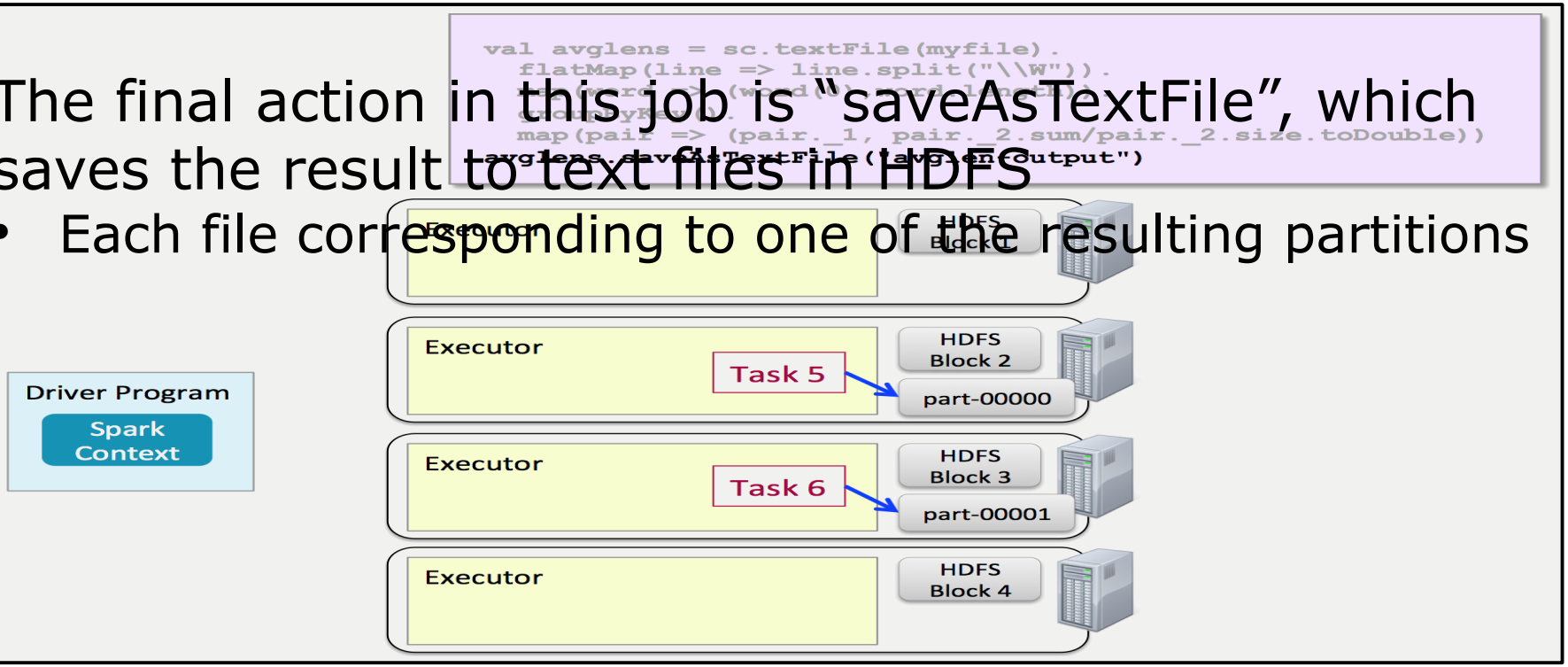
Spark Task Execution (5 Of 7)

- By definition, a new stage requires shuffling data between nodes, so data locality is not an issue



Spark Task Execution (6 Of 7)

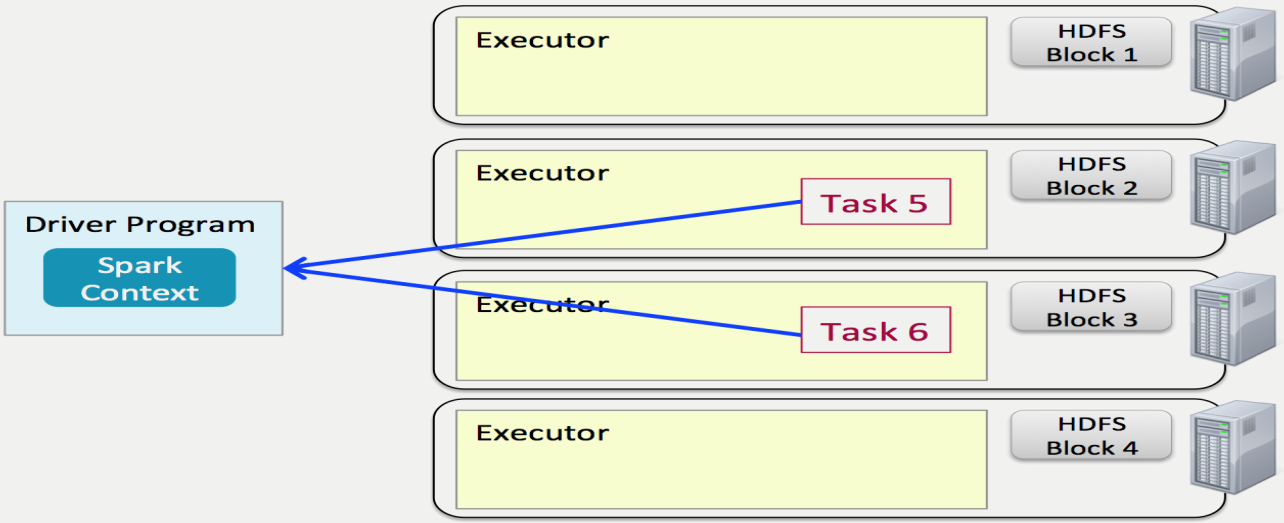
- The final action in this job is "saveAsTextFile", which saves the result to text files in HDFS
  - Each file corresponding to one of the resulting partitions



Spark Task Execution (7 Of 7)

- As an alternative, you could send your result back to the Driver

```
val avglens = sc.textFile(myfile).
  flatMap(line => line.split("\\W")).
  map(pair => (pair._1, pair._2.sum/pair._2.size.toDouble))
avglens.collect()
```



Controlling The Level Of Parallelism (1 Of 2)

- Wide operations (e.g., reduceByKey()) partition the resulting RDDs
  - The more partitions, the more parallel tasks
  - The cluster will be underutilized if there are too few partitions
    - Tasks that could benefit from running in parallel will instead run consecutively

Controlling The Level Of Parallelism (2 Of 2)

- Spark figures out how many partitions to create in a reduce operation in one of 3 ways
  - You can configure a global setting with the spark.default.parallelism property

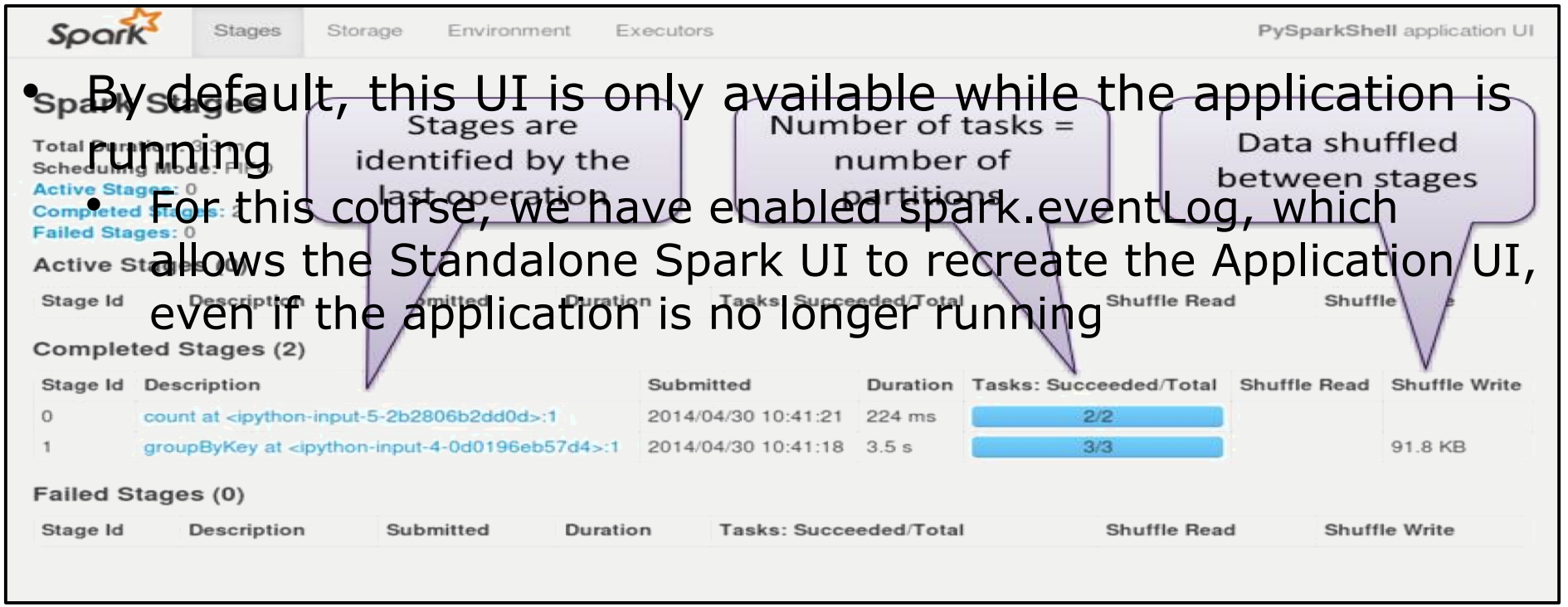
```
spark.default.parallelism : 10
```

    a reduce method call (overriding the global setting)
  - Otherwise, Spark's behavior is platform specific
    - Python sets the default parallelism to the number of

```
words.reduceByKey(lambda v1, v2: v1 + v2, 15)
```

    - Scala sets the default parallelism to the number of partitions belonging to the parent (or the number of partitions belonging to the larger parent in the event of a join)

Viewing Stages In The Spark Application UI



- By default, this UI is only available while the application is running
- For this course, we have enabled spark.eventLog, which allows the Standalone Spark UI to recreate the Application UI, even if the application is no longer running

Chapter Topics

- Review: Spark On A Cluster
- RDD Partitions
- Partitioning Of File-Based RDDS
- HDFS And Data Locality
- Hands-On Exercise: Working With Partitions
- Executing Parallel Operations
- Stages And Tasks
- Summary
- Review
- References
- Hands-On Exercise: Viewing Stages And Tasks In The Spark Application UI

Summary

- RDDs run in the memory of Spark executor JVMs
- Data is split into partitions, each running in a separate executor
- The benefits of data locality in Spark extend beyond those in Hadoop
  - In Hadoop the benefits of data locality are limited to loading data from HDFS
  - In Spark they also apply to narrow operations
- RDD operations are executed on partitions in parallel
- Operations that depend on the same partition are pipelined together in stages
  - e.g., map, filter
- Operations that depend on multiple partitions are executed in separate stages
  - e.g., join, reduceByKey

Chapter Topics

- Review: Spark On A Cluster
- RDD Partitions
- Partitioning Of File-Based RDDS
- HDFS And Data Locality
- Hands-On Exercise: Working With Partitions
- Executing Parallel Operations
- Stages And Tasks
- Summary
- Review
- References
- Hands-On Exercise: Viewing Stages And Tasks In The Spark Application UI

Review

- List 2 uses of the sc.wholeTextFiles method

Review Answer

- List 2 uses of the sc.wholeTextFiles method
  - Can be used to gather multiple files into a single RDD, one per record
  - Can be used to process multiline content (since the entire content of the file is in a single element of the RDD)

Review

- The number of partitions into which Spark breaks a single file is based on _____. (Choose one)
  - The number of cores on the node in question
  - The number of records in the file
  - The size of the file in bytes

Review Answer

- The number of partitions into which Spark breaks a single file is based on _____. (Choose one)
  - The number of cores on the node in question
  - The number of records in the file
  - The size of the file in bytes

Review

- Data locality applies to:
  - Shuffle and Sort
  - Narrowing operations
  - Widening operations

Review Answer

- Data locality applies to:
  - Shuffle and Sort
  - Narrowing operations
  - Widening operations

Review

- Under what conditions does Spark not apply data locality?

Review Answer

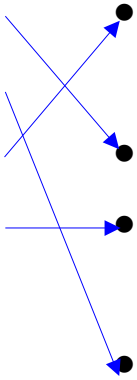
- Under what conditions does Spark not apply data locality?
  - When the node on which data resides is busy
  - On widening operations
  - On the shuffle and sort

Review

- Match the term on the left with its definition on the right
  - Application
  - Job
  - Stage
  - Task
  -
    - A set of tasks in a job that can be executed in parallel
    - A set of jobs
    - An individual unit of work sent to a single executor
    - A set of tasks, executed in response to an action

Review Answer

- Match the term on the left with its definition on the right



• Application	• A set of tasks in a job that can be executed in parallel
• Job	• A set of jobs
• Stage	• An individual unit of work sent to a single executor
• Task	• A set of tasks, executed in response to an action

Review

- Spark uses 2 strategies for determining the level of parallelism required by an application. What are these strategies?

Review Answer

- Spark uses 2 strategies for determining the level of parallelism required by an application. What are these strategies?
  - Pass a number representing the desired level of parallelism as a function argument
    - This strategy overrides others
  - Use the value of the spark.default.parallelism property
    - If specified

Review

- What does it mean to say that an operation "preserves partitioning"? (Choose one)
  - All of the input data comes from a single partition
  - The input RDD and output RDD are identical
  - The output RDD is a reference to the input RDD (whose contents may have been altered as a side effect of the operation)
  - None of the above

Review Answer

- What does it mean to say that an operation "preserves partitioning"? (Choose one)
  - All of the input data comes from a single partition
  - The input RDD and output RDD are identical
  - The output RDD is a reference to the input RDD (whose contents may have been altered as a side effect of the operation)
  - None of the above

Review

- What triggers the end of a stage?

Review Answer

- What triggers the end of a stage?
  - The end of a stage is triggered by an operation that does not preserve partitioning, or by an action

Review

- What does it mean to say that the tasks in a stage are pipelined?

Review Answer

- What does it mean to say that the tasks in a stage are pipelined?
  - Rather than executing as a chain of batch operations, the tasks within an operation will be applied consecutively to each record
    - Providing the benefits of data locality, as well as eliminating the generation of lots of intermediate data streams

Chapter Topics

- Review: Spark On A Cluster
- RDD Partitions
- Partitioning Of File-Based RDDS
- HDFS And Data Locality
- Hands-On Exercise: Working With Partitions
- Executing Parallel Operations
- Stages And Tasks
- Summary
- Review
- References
- Hands-On Exercise: Viewing Stages And Tasks In The Spark Application UI

References

- https://spark.apache.org/docs/latest/configuration.html#spark-properties
- https://stackoverflow.com/questions/23127329/how-to-define-custom-partitioner-for-spark-rdds-of-equally-sized-partition-where

Chapter Topics

- Review: Spark On A Cluster
- RDD Partitions
- Partitioning Of File-Based RDDS
- HDFS And Data Locality
- Hands-On Exercise: Working With Partitions
- Executing Parallel Operations
- Stages And Tasks
- Summary
- Review
- References
- Hands-On Exercise: Viewing Stages And Tasks In The Spark Application UI

Hands-On Exercise

- Viewing Stages And Tasks In The Spark Application UI
  - Use the Spark Application UI to view how stages and tasks are executed in a job
- Please refer to the Hands-On Exercise Manual