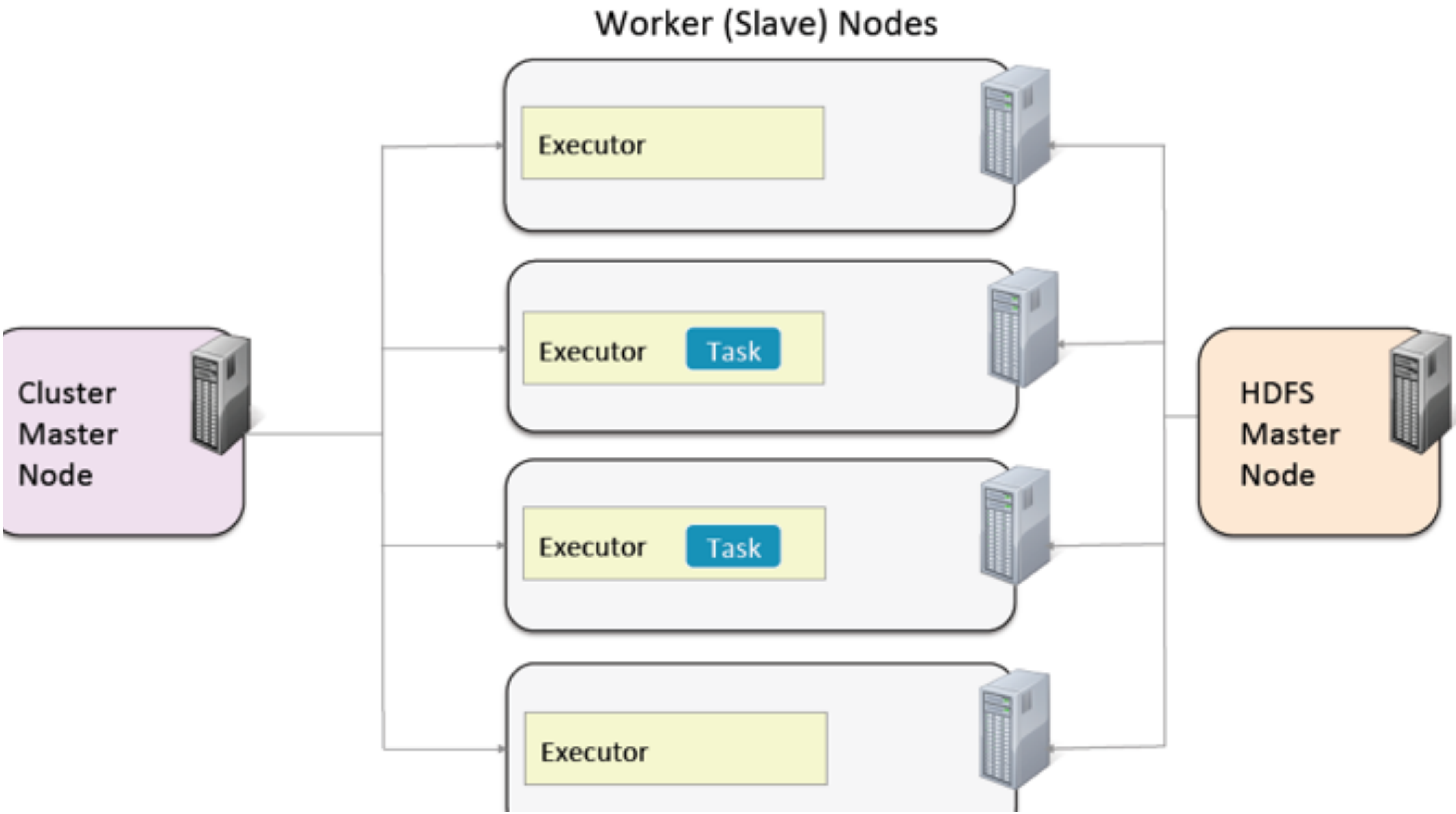
Chapter 6

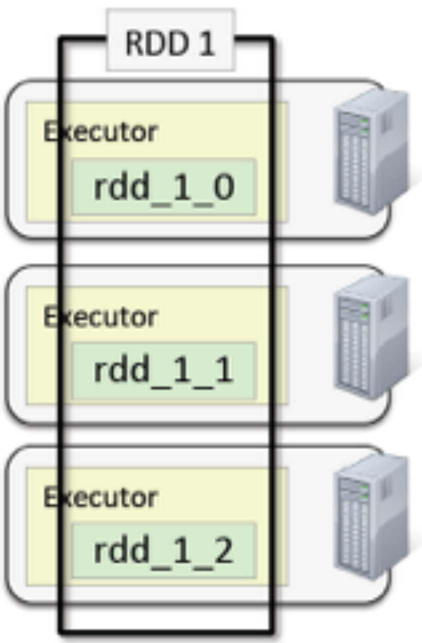
PARALLEL PROGRAMMING WITH SPARK

Spark Cluster Review



RDDs on a Cluster

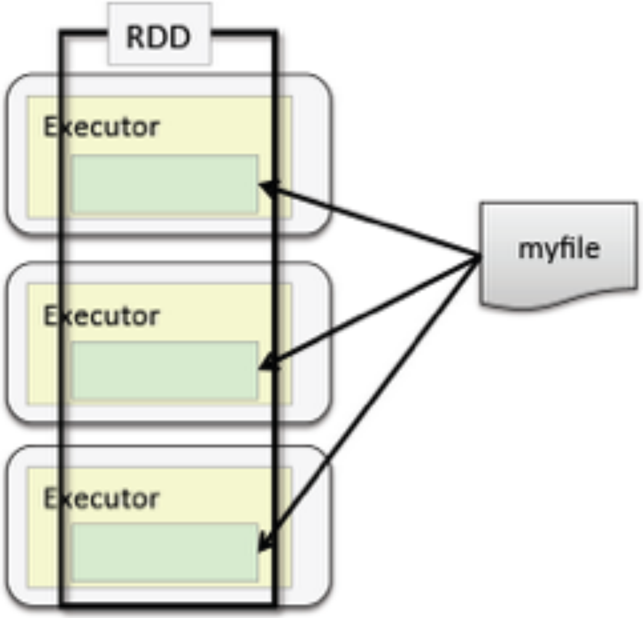
- Resilient Distributed Datasets
  - Data is partitioned across worker nodes
- Partitioning is done automatically by Spark
  - Optionally, you can control how many partitions are created



File Partitioning: Single Files

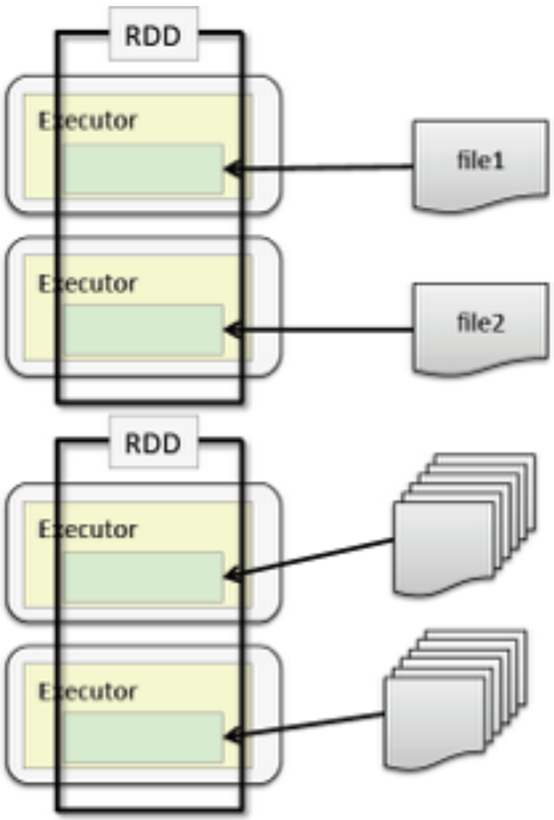
```
sc.textFile("myfile",3)
```

- Partitions from single files
  - Partitions based on size
  - You can optionally specify a minimum number of partitions textFile(file, minPartitions)
  - Default is 2
  - More partitions = more parallelization



File Partitioning: Multiple Files

- sc.textFile("mydir/*")
  – Each file becomes (at least) one partition
  – File-based operations can be done per-partition
- sc.wholeTextFiles("mydir")
  – For many small files
  – Creates a key-value PairRDD
    – key = file name
    – value = file contents



Operating on Partitions

- Most RDD operations work on each element of an RDD
- A few work on each partition
  - foreachPartition - call a function for each partition
  - mapPartitions - create a new RDD by executing a function on each partition in the current RDD
- Functions for partition operations take iterators

Example: Count JPGs Requests per File

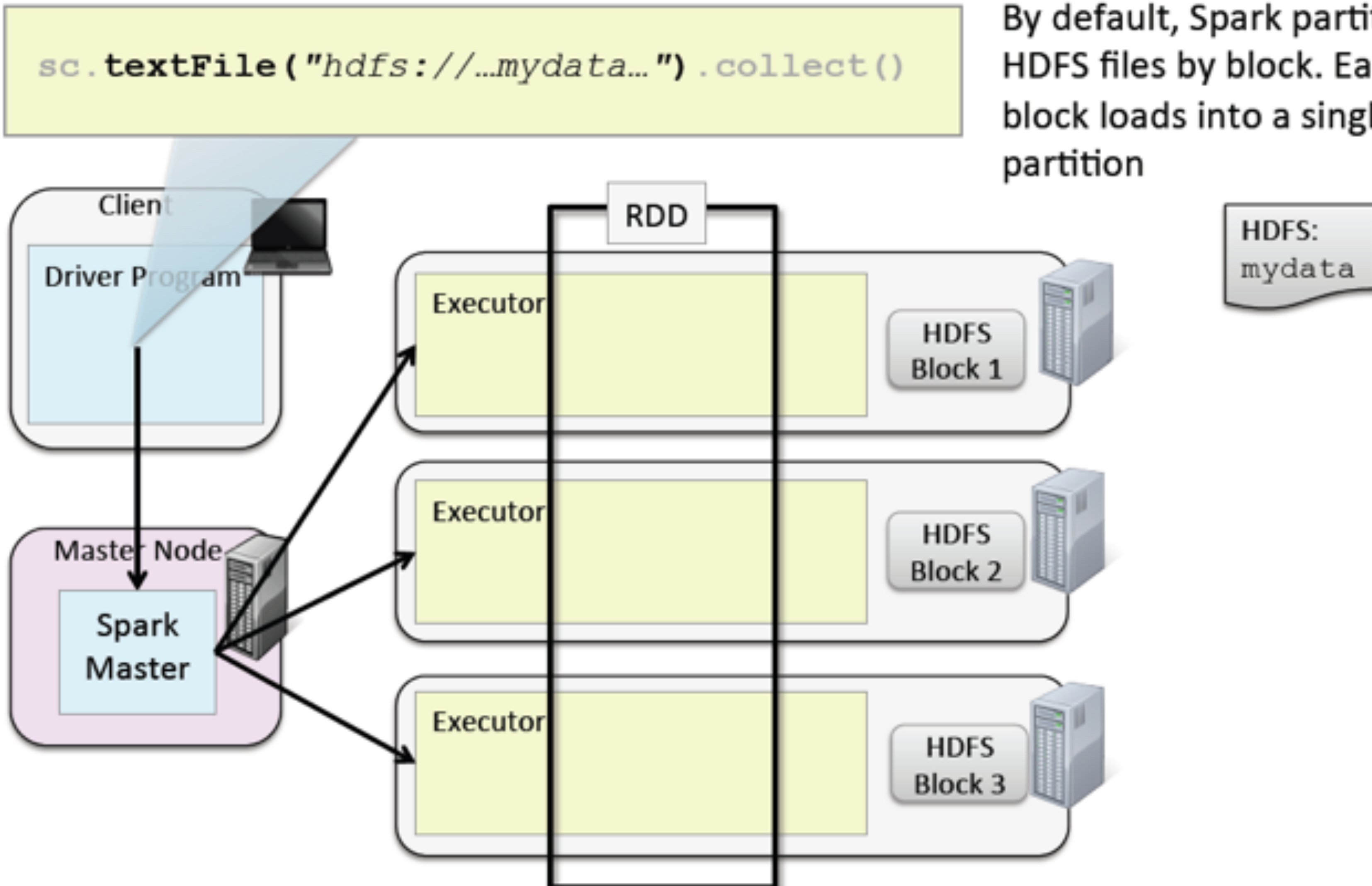
```python
def countJpgs(index,partIter):
  jpgcount = 0
  for line in partIter:
    if "jpg" in line: jpgcount += 1
  yield (index,jpgcount)

jpgcounts = sc.textFile("weblogs/*") \
  .mapPartitionsWithIndex(countJpgs)
```

```scala
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))
}
jpgcounts = sc.textFile("weblogs/*").
  mapPartitionsWithIndex(countJpgs)
```

Note: Works with small files that each fit in a single partition

jpgcounts
(0,237)
(1,132)
(2,188)
(3,193)
…

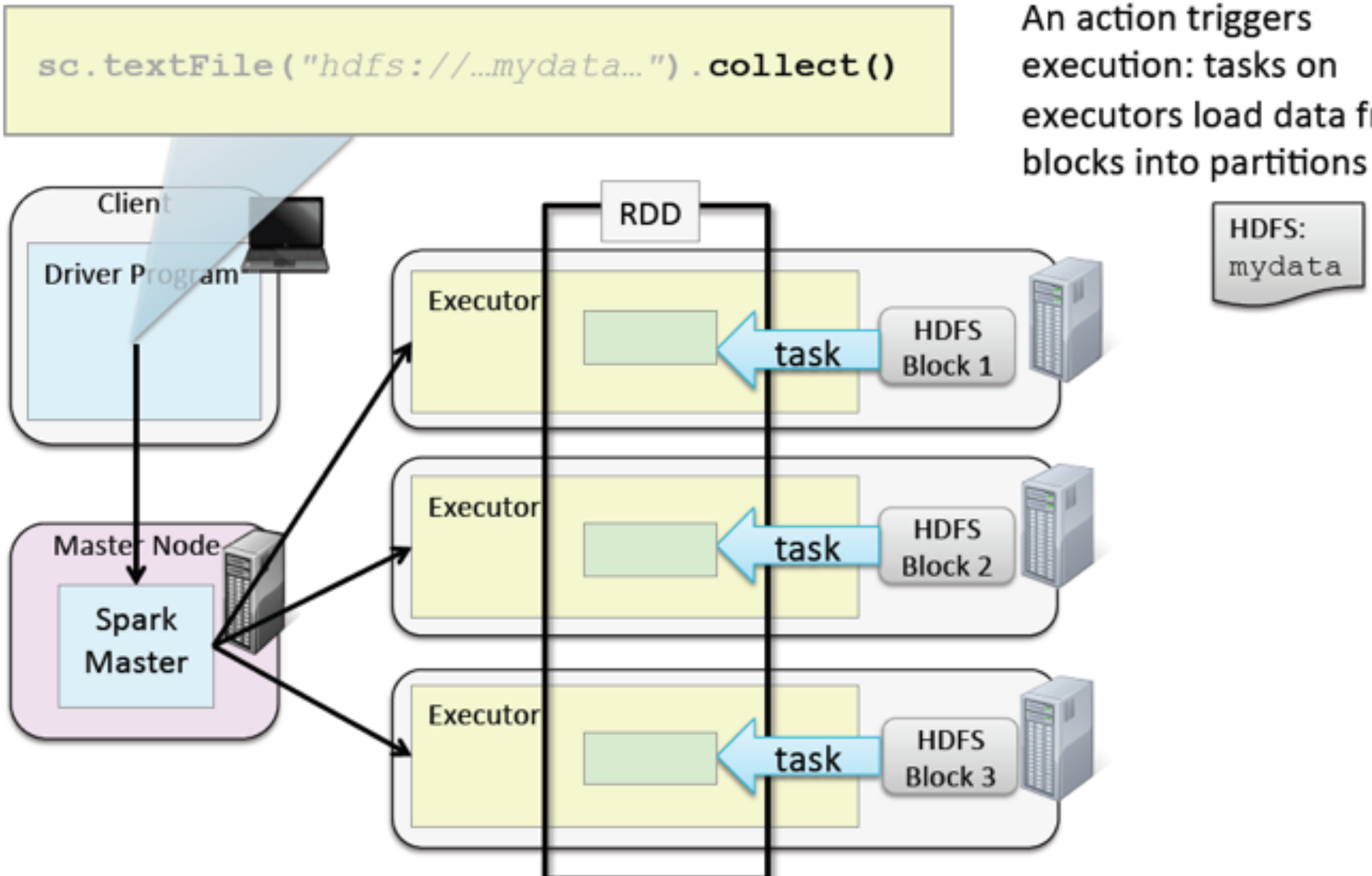
HDFS and Data Locality



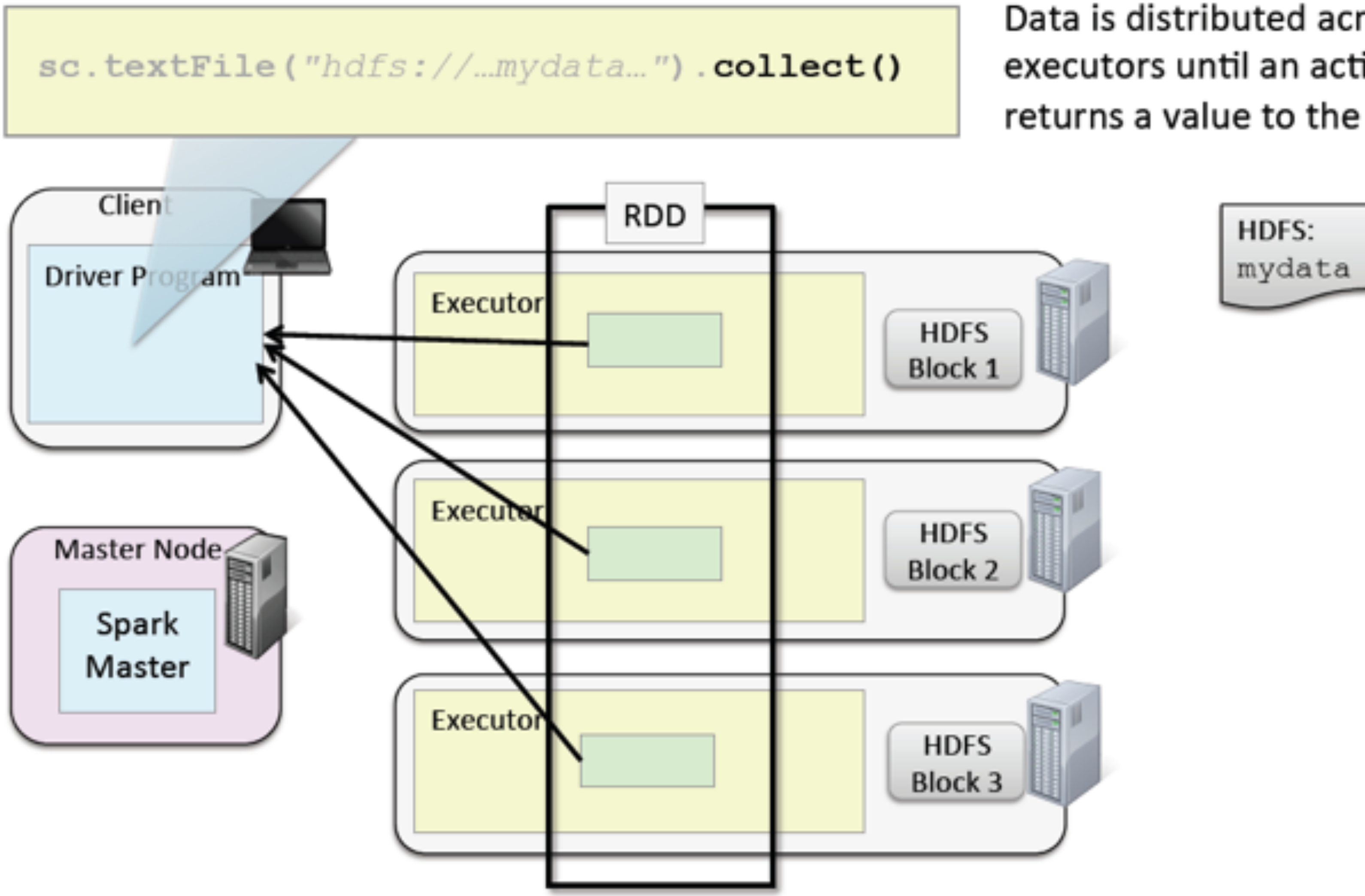
By default, Spark partitions HDFS files by block. Each block loads into a single partition

HDFS and Data Locality

An action triggers execution: tasks on executors load data from blocks into partitions



HDFS and Data Locality



Data is distributed across executors until an action returns a value to the driver

Hands On Exercise: Working With Partitions

HANDS-ON EXERCISE: WORKING WITH PARTITIONS

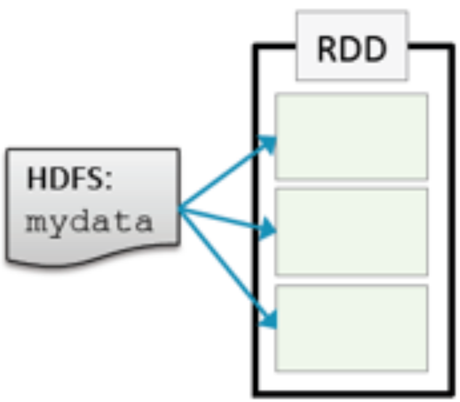
Parallel Operations on Partitions

- RDD operations are executed in parallel on each partition
  - When possible, tasks execute on the worker nodes where the data is in memory
- Some operations preserve partitioning
  - e.g., map, flatMap, filter
- Some operations repartition
  - e.g., reduce, sort, group

Example: Average Word Length by Letter

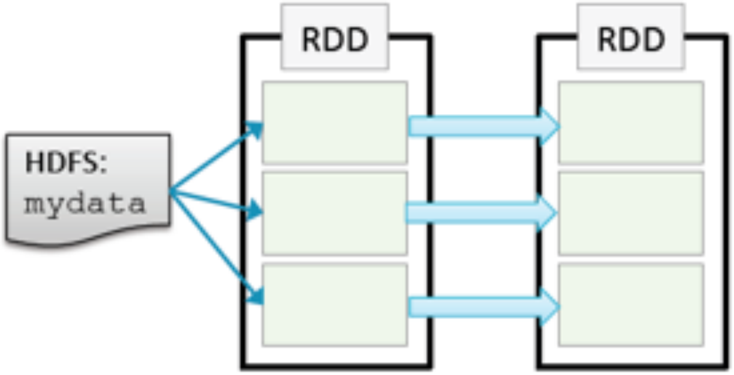
-

```
> avglens = sc.textFile(file)
```



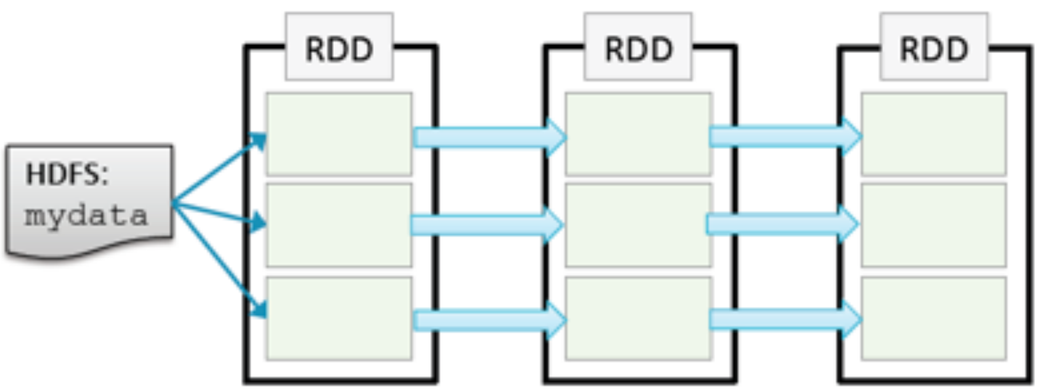
Example: Average Word Length by Letter

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



Example: Average Word Length by Letter

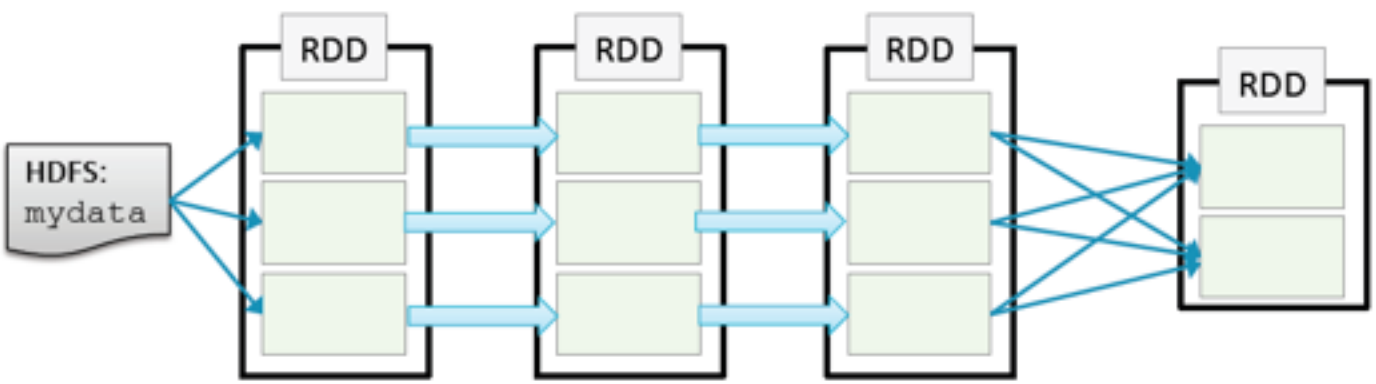
```
> avglens = sc.textFile(file) \
  .flatMap(lambda line: line.split()) \
  .map(lambda word: (word[0],len(word)))
```



Example: Average Word Length by Letter

-

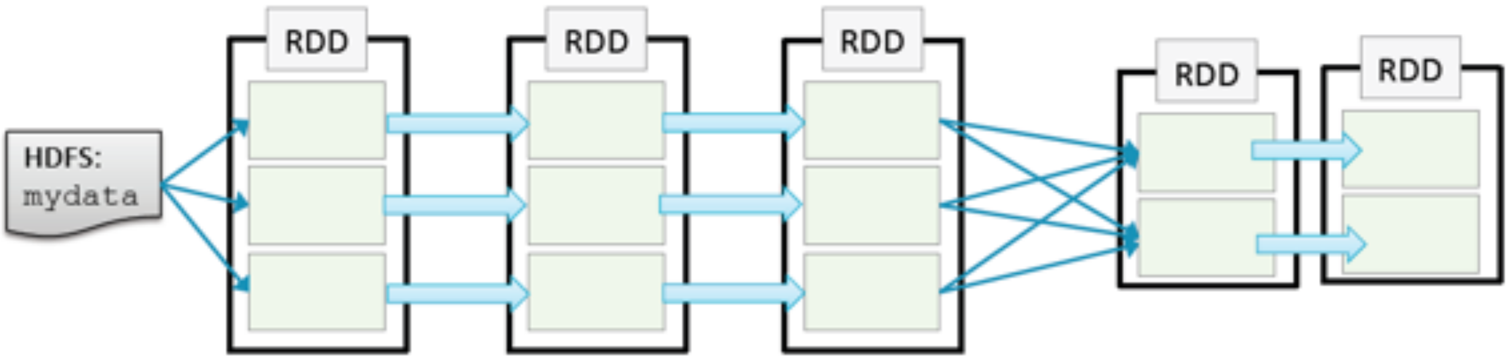
```
> avglens = sc.textFile(file) \
    .flatMap(lambda line: line.split()) \
    .map(lambda word: (word[0],len(word))) \
    .groupByKey()
```



Example: Average Word Length by Letter

•

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

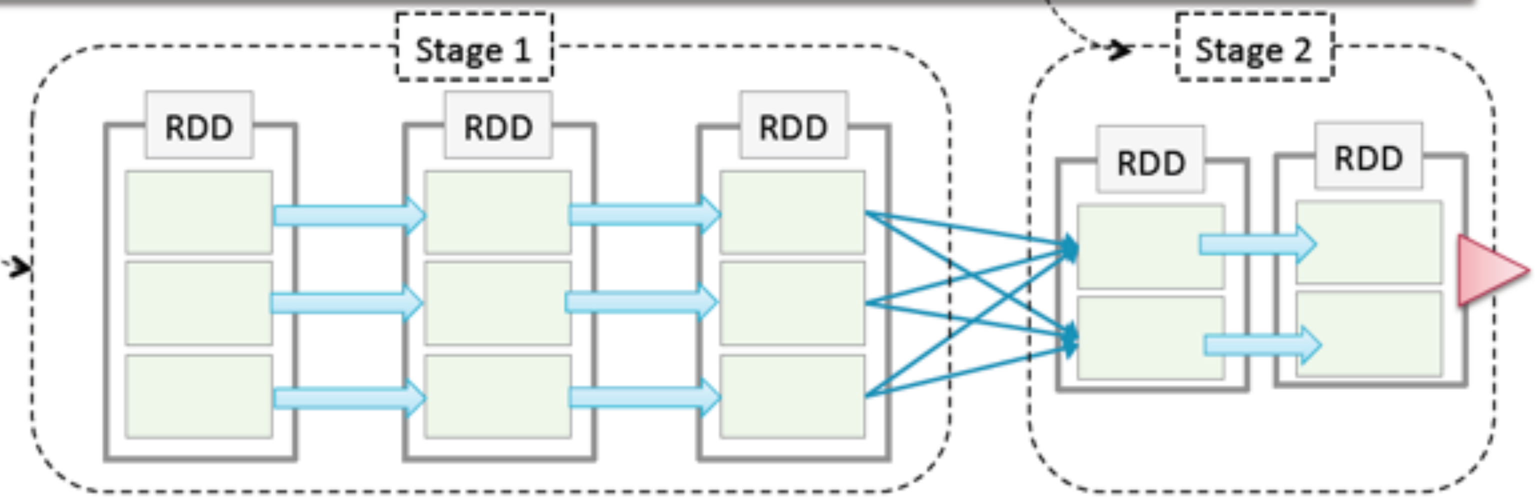


Stages

- **Operations that can run on the same partition are executed in stages**
- **Tasks within a stage are pipelined together**
- **Developers should be aware of stages to improve performance**

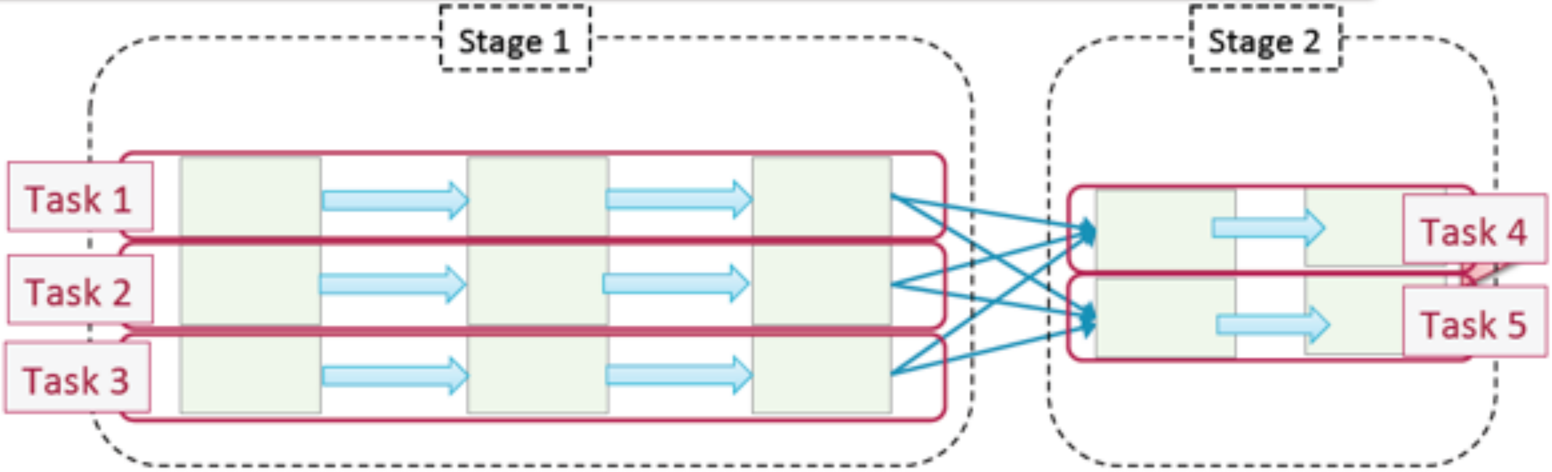
Spark Execution: Stages

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



Spark Execution: Stages

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


  ```

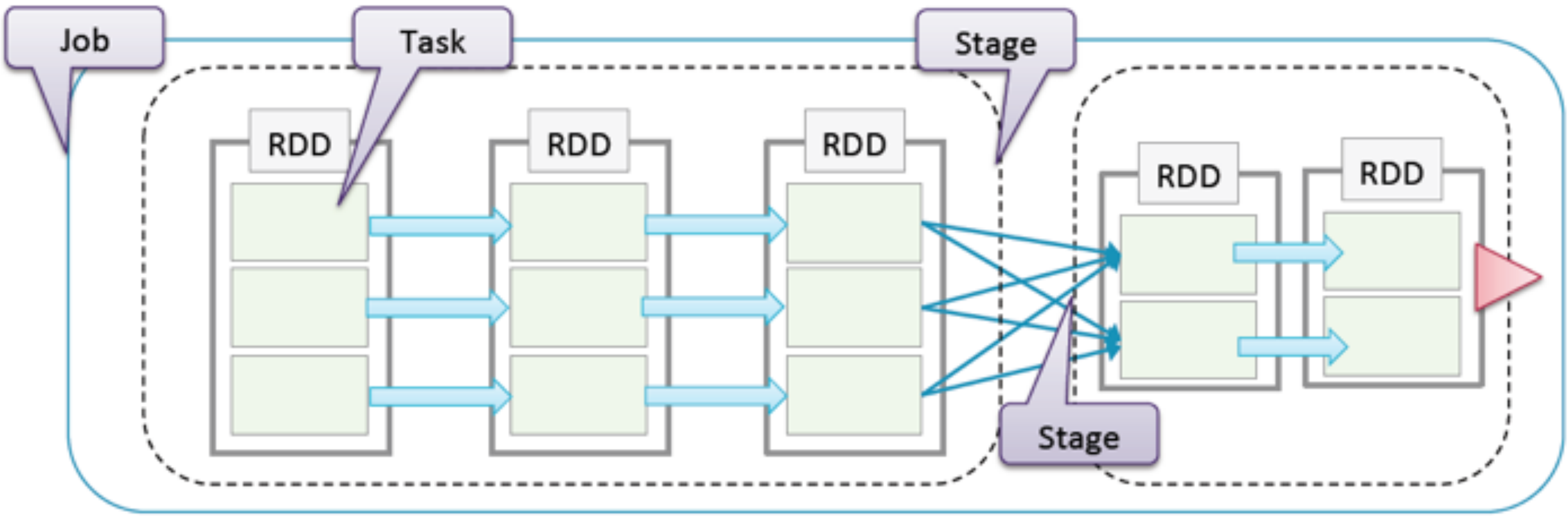
Controlling the Level of Parallelism

- **RDD operations that repartition data (e.g., reduceByKey) take an optional additional parameter for the number of partitions/tasks**
  - **partitions in the largest upstream RDD**
  - **Configure with the spark.default.parallelism property**

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

Summary of Spark Terminology

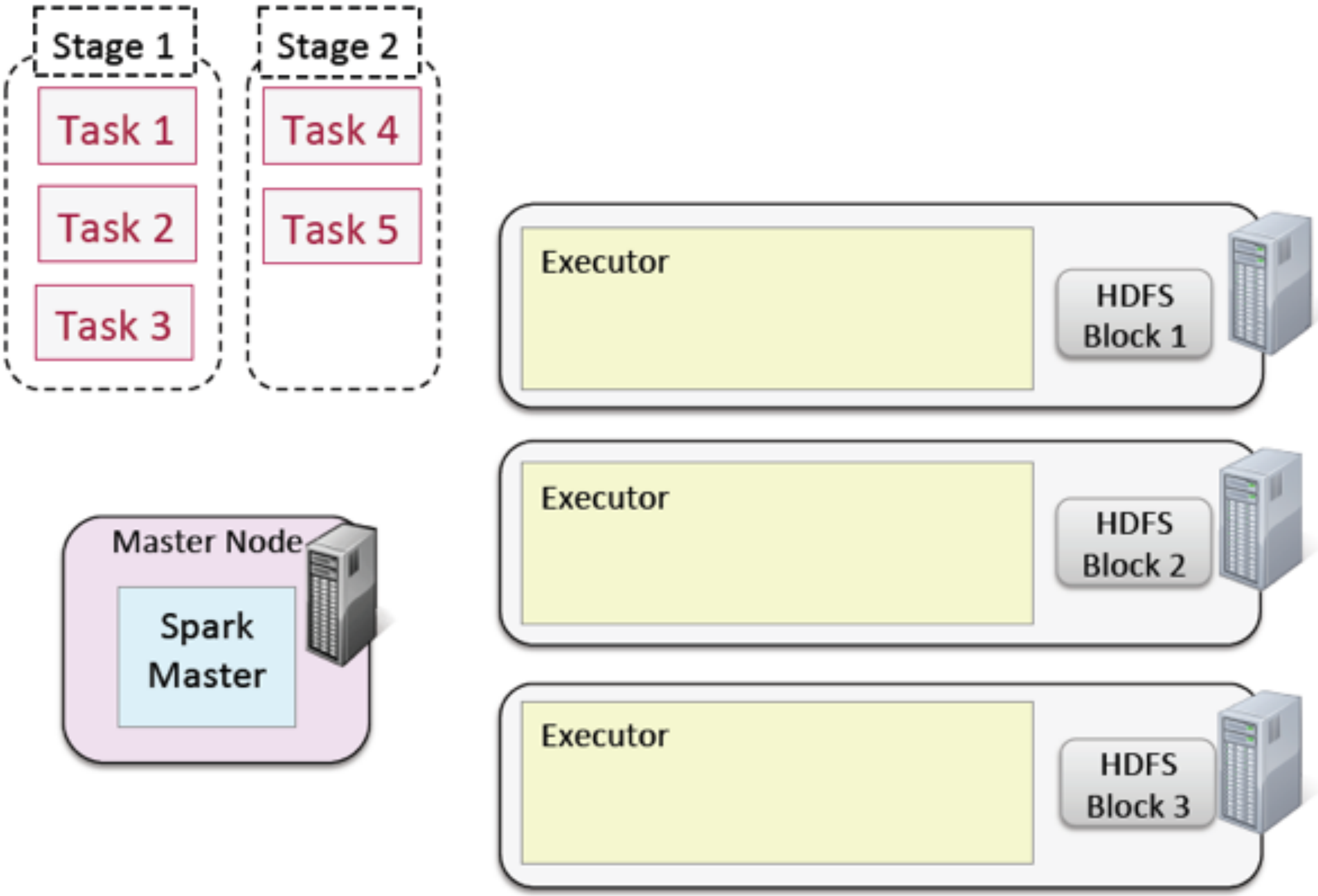
- **Job - a set of tasks executed as a result of an action**
- **Stage - a set of tasks in a job that can be executed in parallel**
- **Task - an individual unit of work sent to one executor**



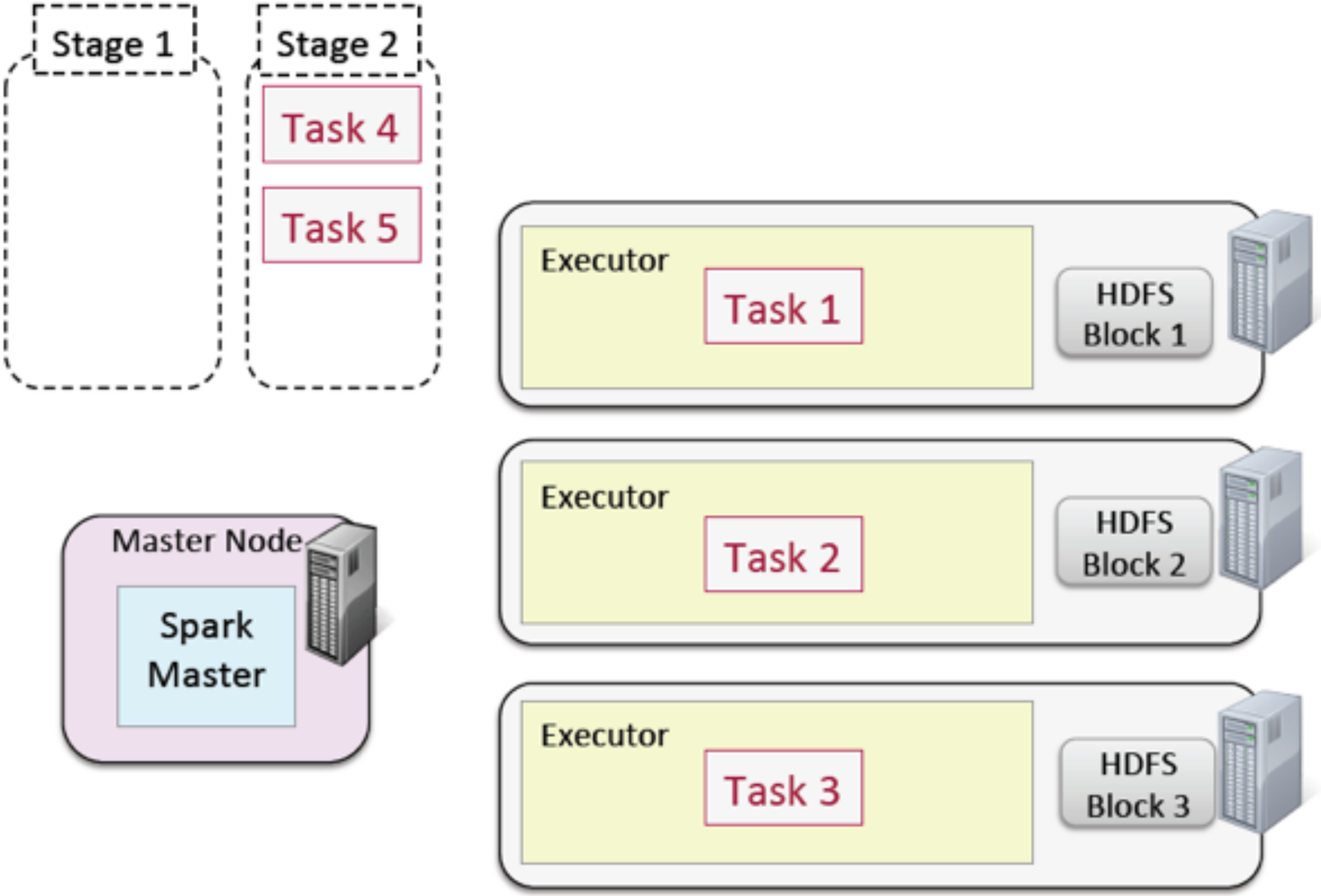
How Spark Calculates Stages

- Spark constructs a DAG (Directed Acyclic Graph) of RDD dependencies
- Narrow operations
  - Only one child depends on the RDD
  - 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
  - Defines a new stage
  - e.g., reduceByKey, join, groupByKey

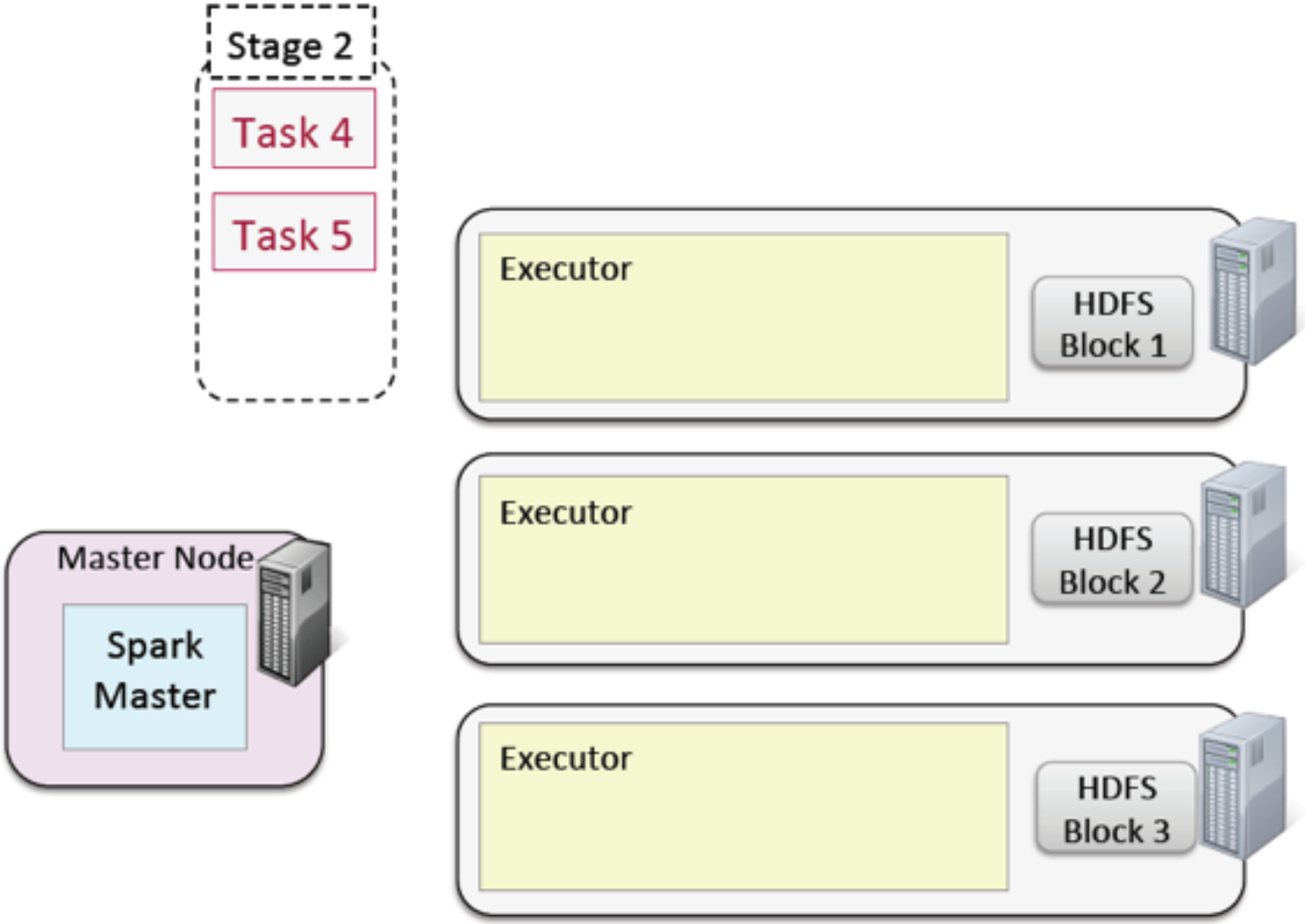
Spark Execution: Task Scheduling



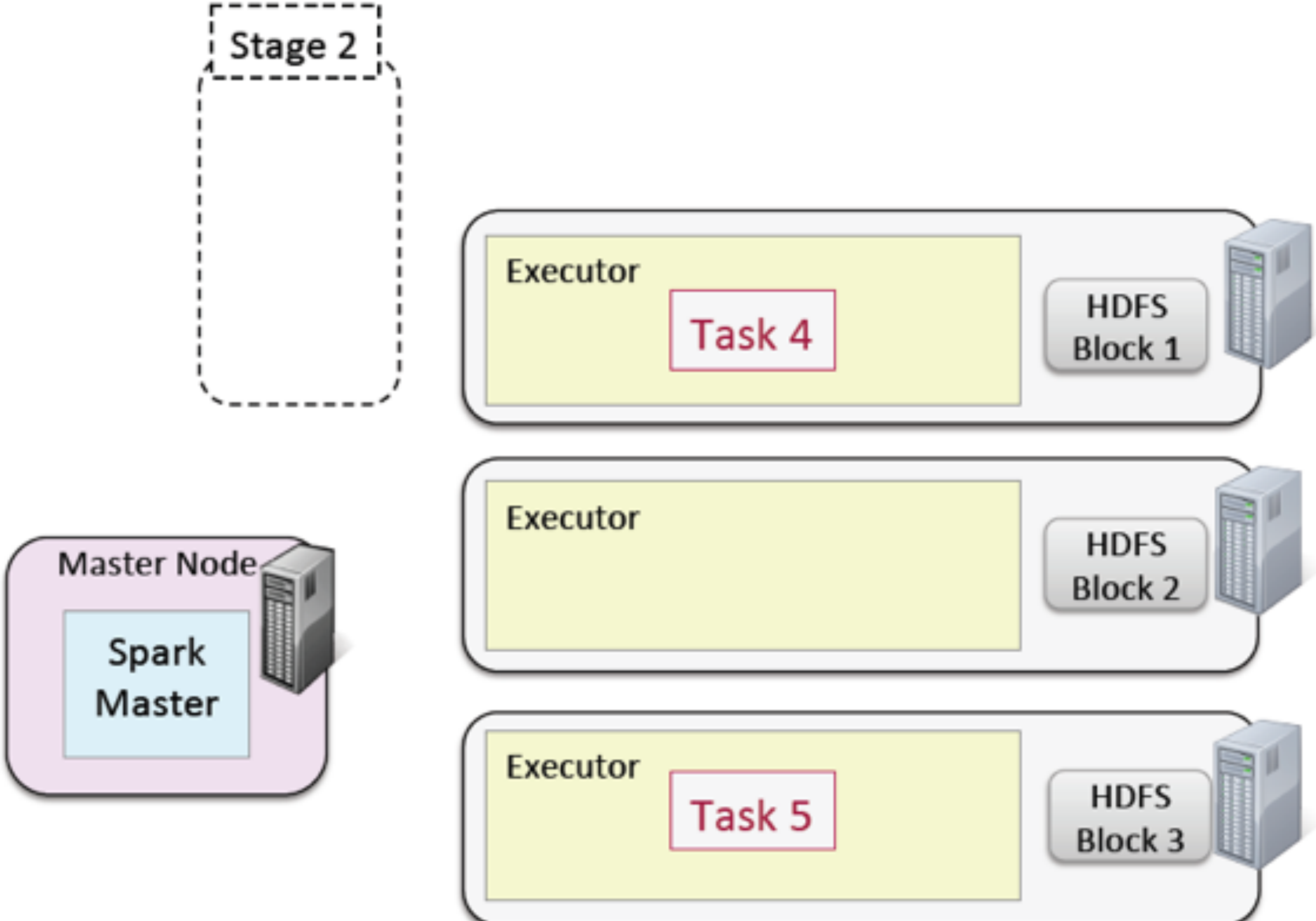
Spark Execution: Task Scheduling



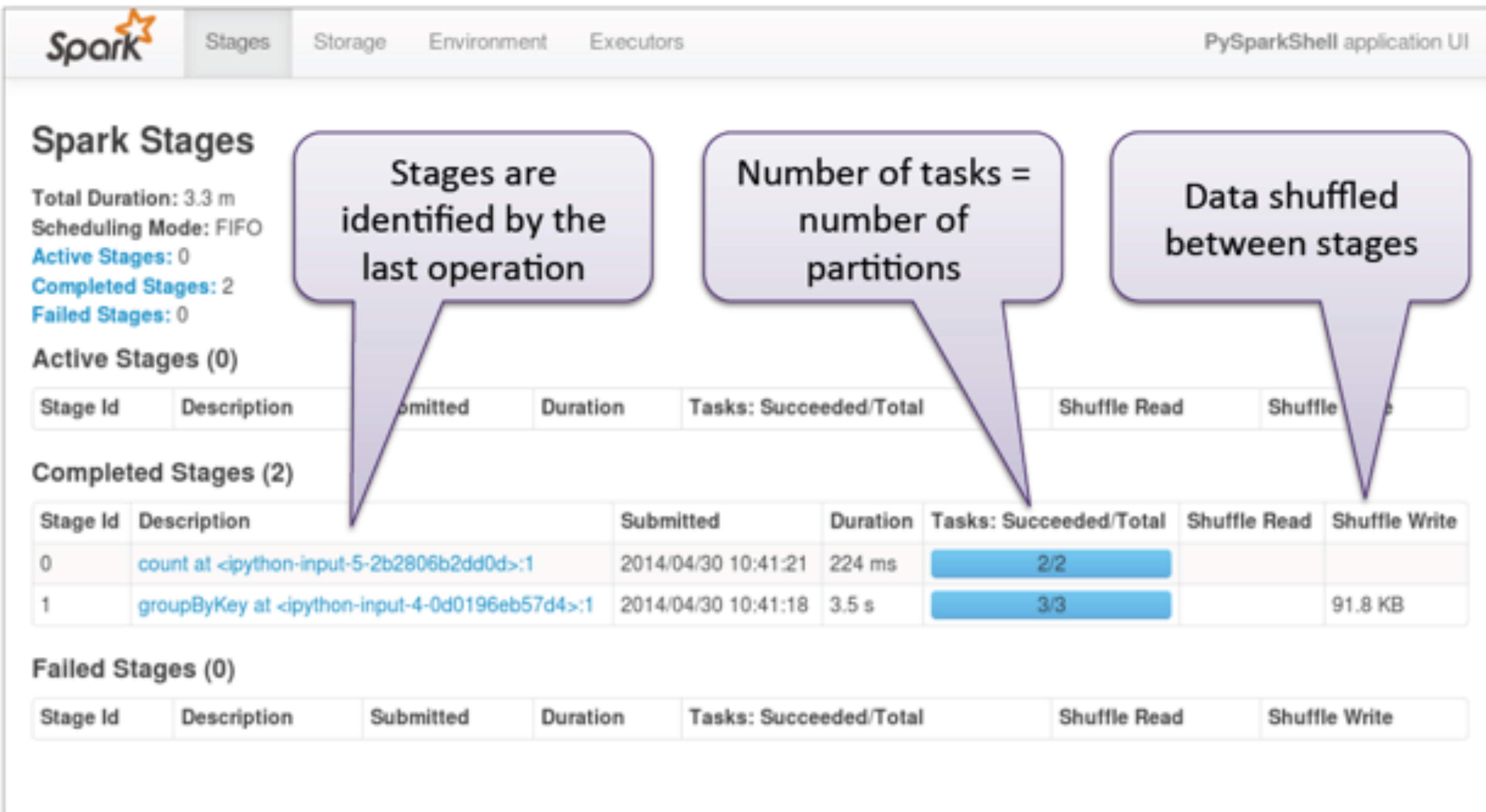
Spark Execution: Task Scheduling



Spark Execution: Task Scheduling



Viewing Stages in the Spark Application UI



Hands On Exercise: Viewing Stages and Tasks in the Spark Application UI

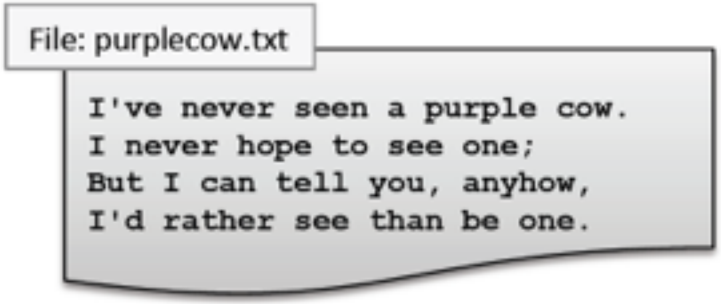
HANDS-ON EXERCISE: VIEWING STAGES AND TASKS IN THE SPARK APPLICATION UI

Chapter 7

CACHING AND PERSISTENCE

Example

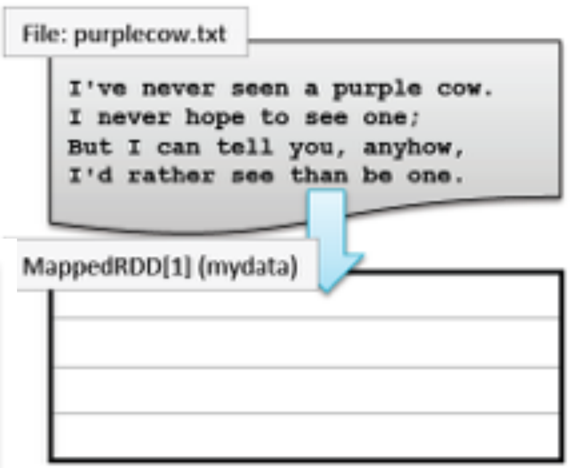
- **Each transformation operation**
  - **creates a new child RDD**



Example

- Each transformation operation
  - creates a new child RDD

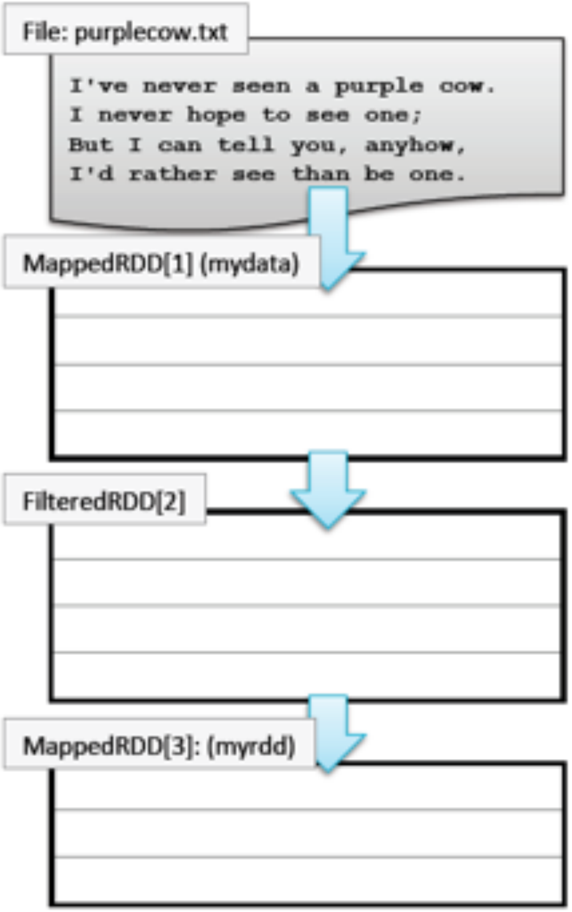
```
> mydata = sc.textFile("purplecow.txt")
```



Example

- Each transformation operation
  - creates a new child RDD

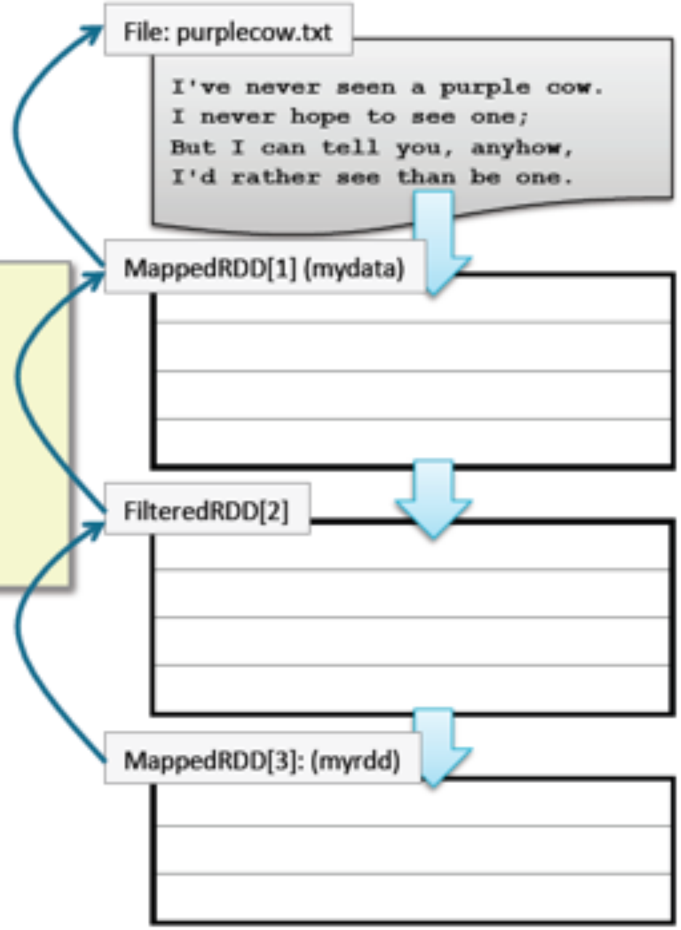
```
> mydata = sc.textFile("purplecow.txt")
> myrdd = mydata.map(lambda s: s.upper())\
  .filter(lambda s:s.startswith('I'))
```



Example

- Spark keeps track of the *parent* RDD for each new RDD
- Child RDDs *depend on* their parents

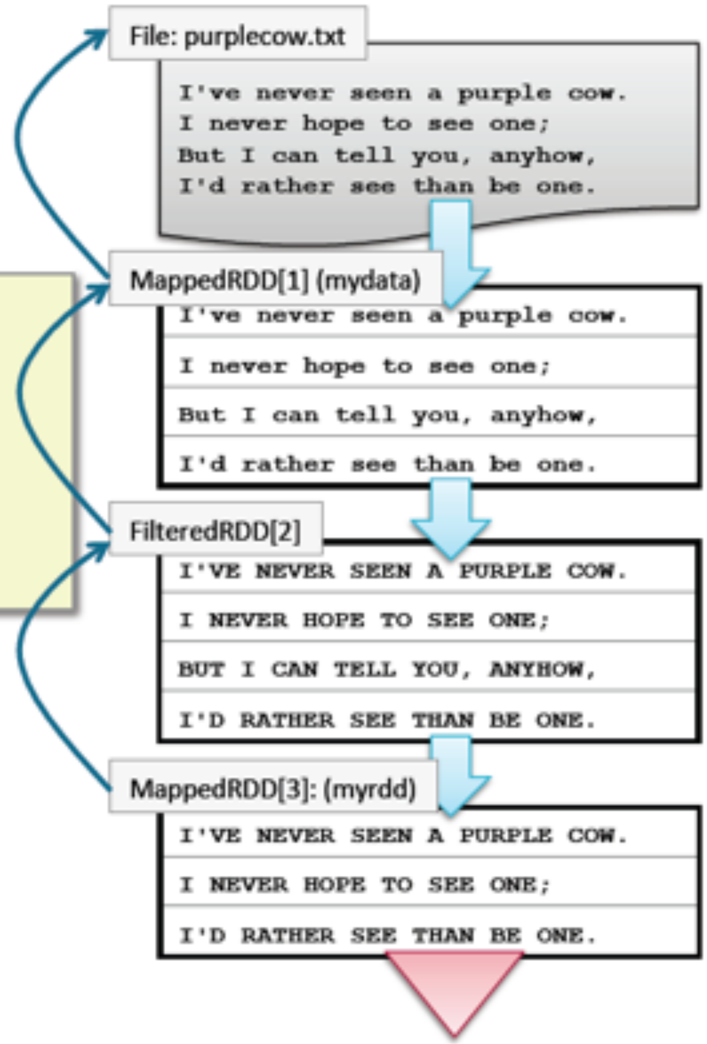
```
> mydata = sc.textFile("purplecow.txt")
> myrdd = mydata.map(lambda s: s.upper())\
  .filter(lambda s:s.startswith('I'))
```



Lineage Example

- *Action* operations execute the parent transformations

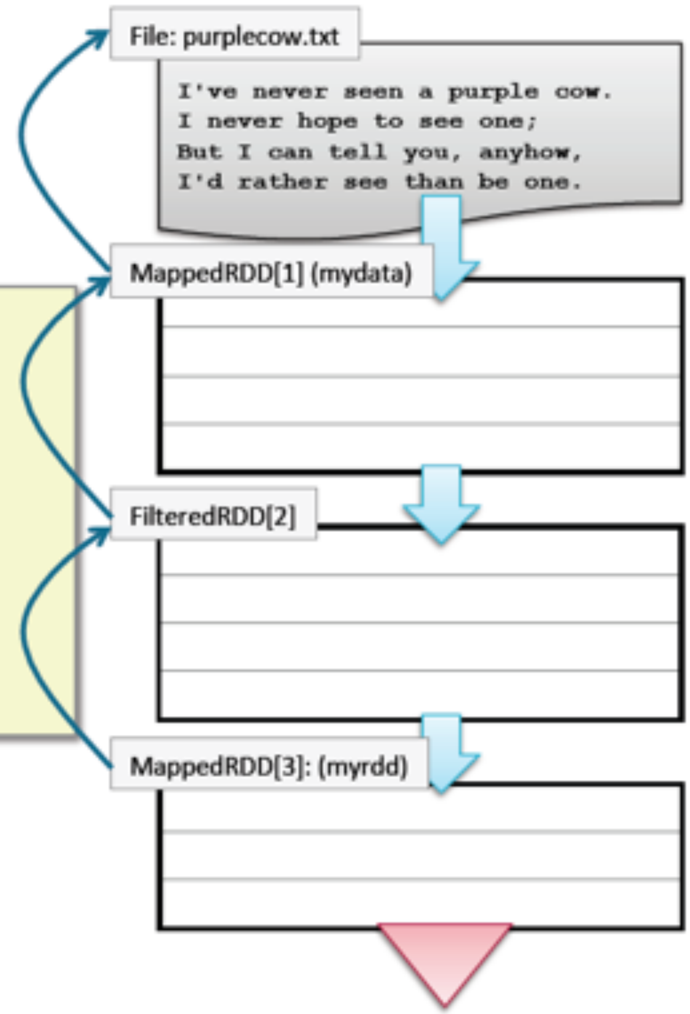
```
> mydata = sc.textFile("purplecow.txt")
> myrdd = mydata.map(lambda s: s.upper())\
  .filter(lambda s:s.startswith('I'))
> myrdd.count()
3
```



Lineage Example

- Each action re-executes the lineage transformations starting with the base
  - By default

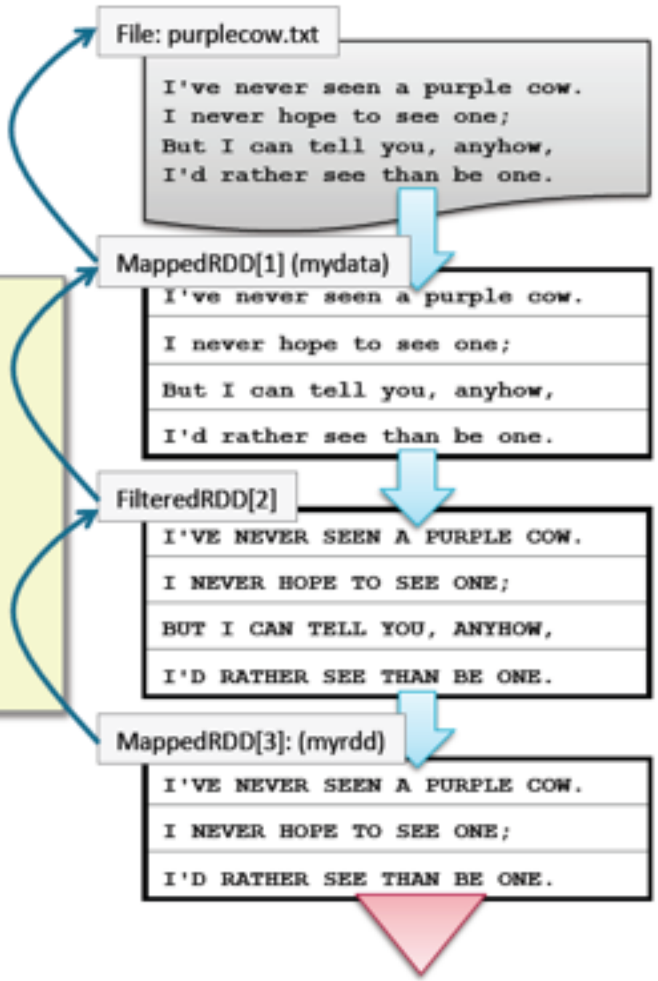
```
> mydata = sc.textFile("purplecow.txt")
> myrdd = mydata.map(lambda s: s.upper())\
  .filter(lambda s:s.startswith('I'))
> myrdd.count()
3
> myrdd.count()
```



Lineage Example

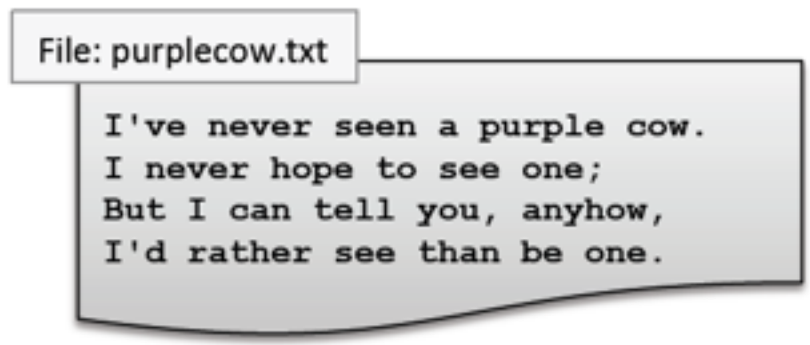
- Each action re-executes the lineage transformations starting with the base
  - By default

```
> mydata = sc.textFile("purplecow.txt")
> myrdd = mydata.map(lambda s: s.upper())\
  .filter(lambda s:s.startswith('I'))
> myrdd.count()
3
> myrdd.count()
3
```



Caching

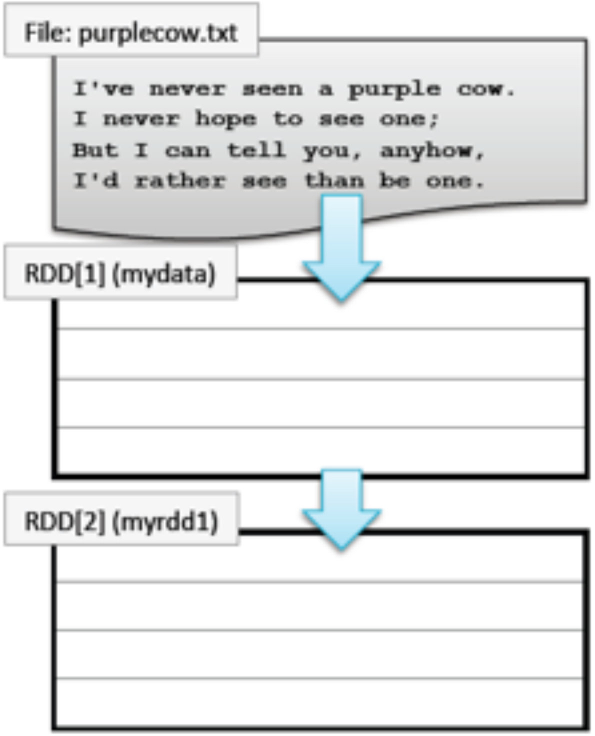
- **Caching an RDD saves the data in memory**



Caching

- Caching an RDD saves the data in memory

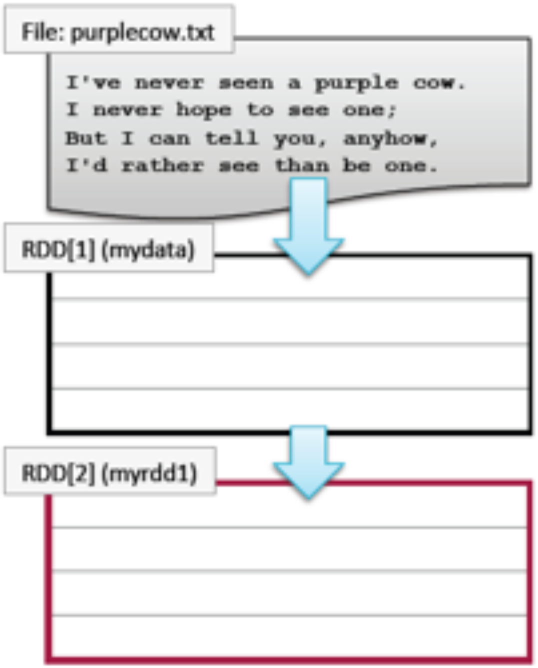
```
> mydata = sc.textFile("purplecow.txt")
> myrdd1 = mydata.map(lambda s:
   s.upper())
```



Caching

- Caching an RDD saves the data in memory

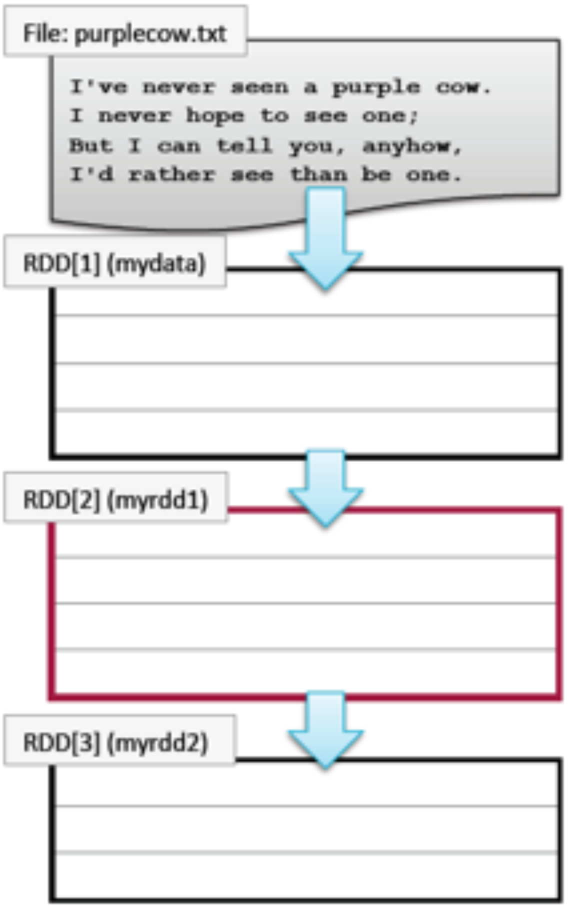
```
> mydata = sc.textFile("purplecow.txt")
> myrdd1 = mydata.map(lambda s:
   s.upper())
> myrdd1.cache()
```



Caching

- **Caching an RDD saves the data in memory**

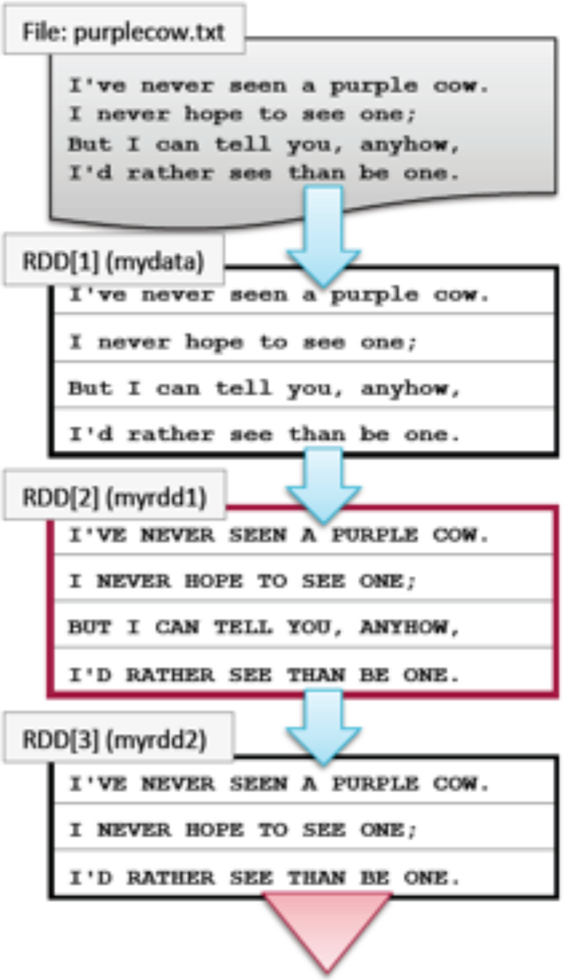
```
> mydata = sc.textFile("purplecow.txt")
> myrdd1 = mydata.map(lambda s:
   s.upper())
> myrdd1.cache()
> myrdd2 = myrdd1.filter(lambda \
   s:s.startswith('I'))
```



Caching

- **Caching an RDD saves the data in memory**

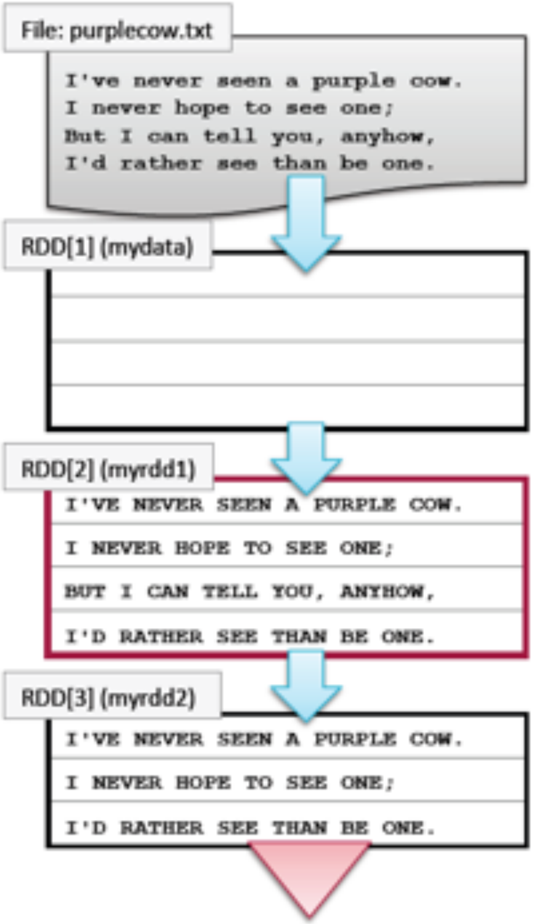
```
> mydata = sc.textFile("purplecow.txt")
> myrdd1 = mydata.map(lambda s:
   s.upper())
> myrdd1.cache()
> myrdd2 = myrdd1.filter(lambda \
   s:s.startswith('I'))
> myrdd2.count()
3
```



Caching

- **Subsequent operations use saved data**

```
> mydata = sc.textFile("purplecow.txt")
> myrdd1 = mydata.map(lambda s:
   s.upper())
> myrdd1.cache()
> myrdd2 = myrdd1.filter(lambda \
   s:s.startswith('I'))
> myrdd2.count()
3
> myrdd2.count()
3
```



Caching

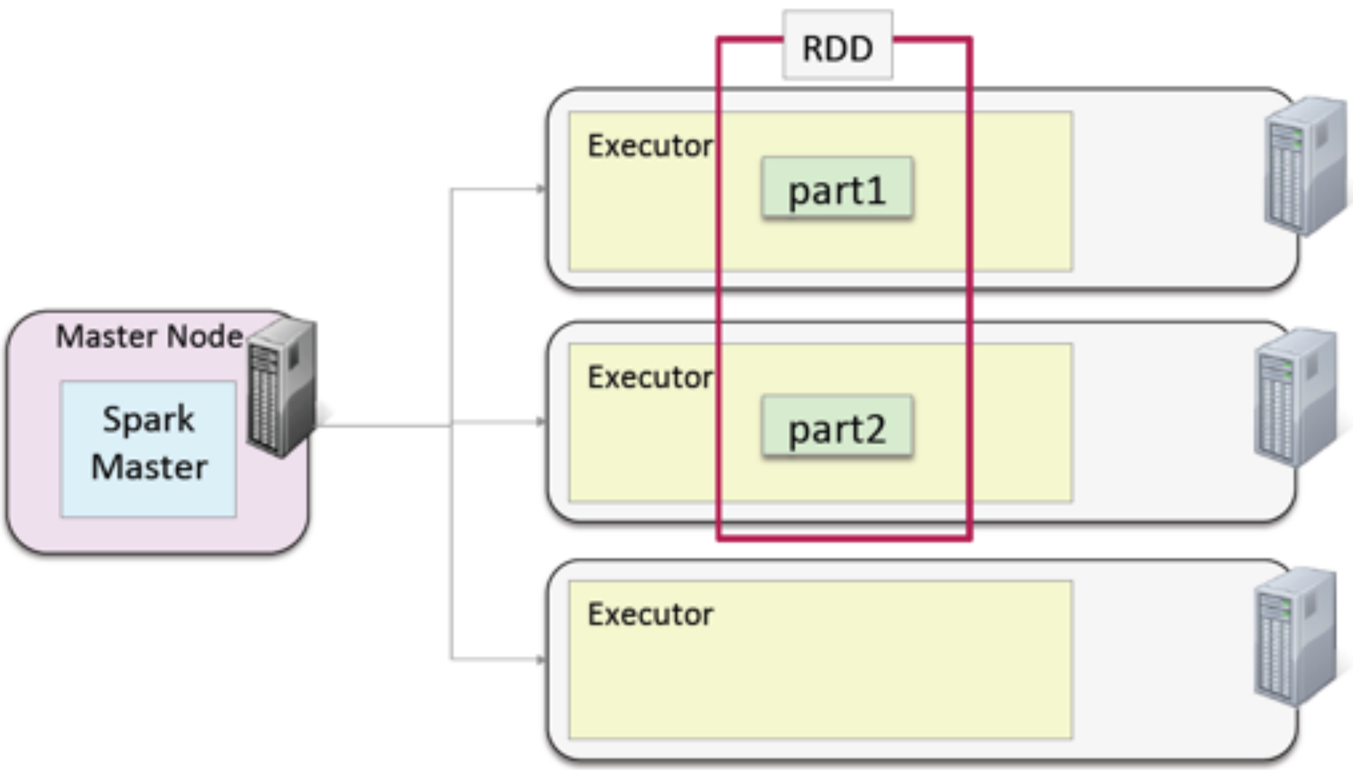
- **Caching is a suggestion to Spark**
  - **If not enough memory is available, transformations will be re-executed when needed**

Caching and Fault-Tolerance

- **RDD = Resilient Distributed Dataset**
  - **Resiliency is a product of tracking lineage**
  - **RDDs can always be recomputed from their base if needed**

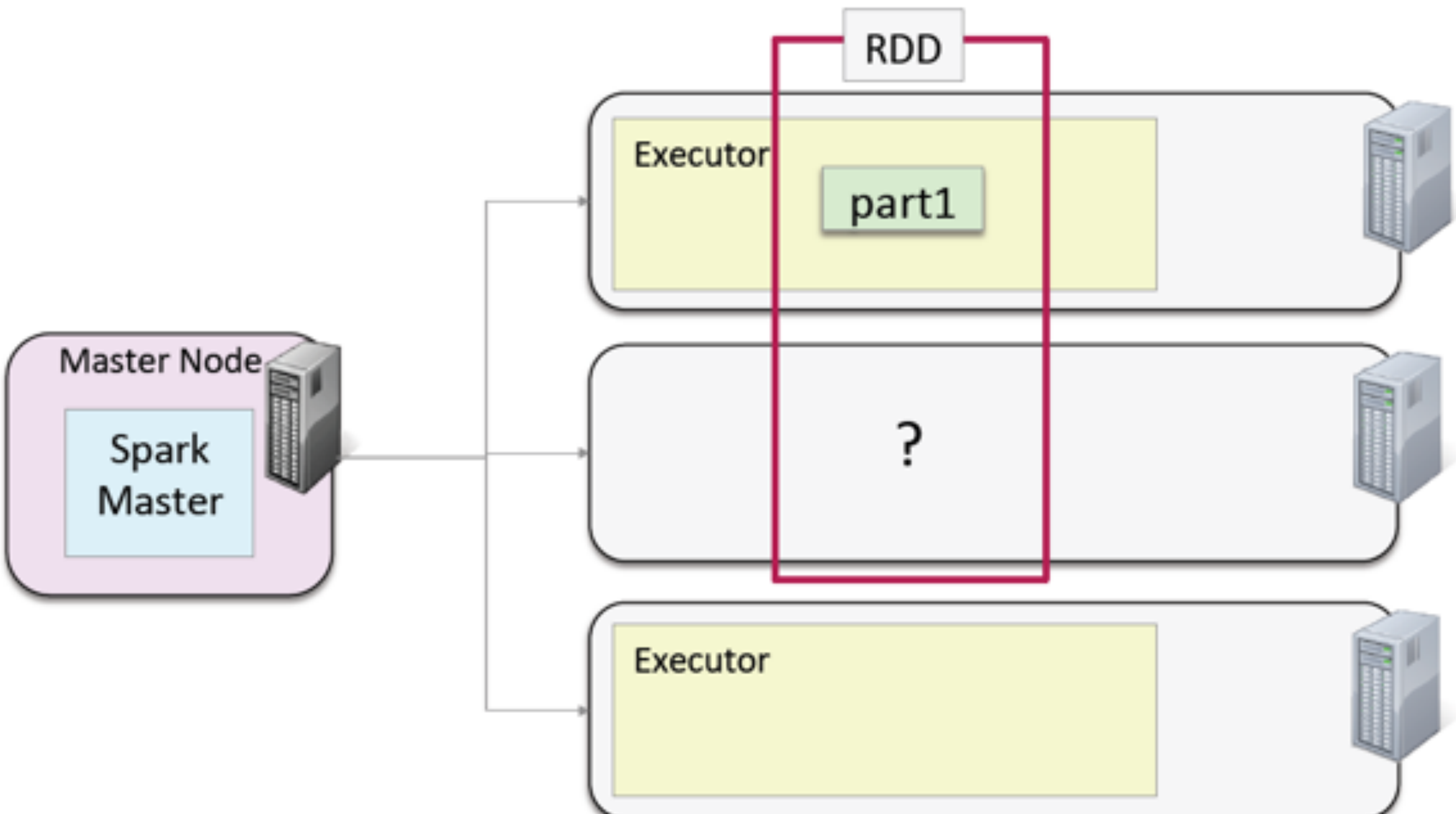
Distributed Cache

- RDD partitions are distributed across a cluster
- Cached partitions are stored in memory in Executor JVMs



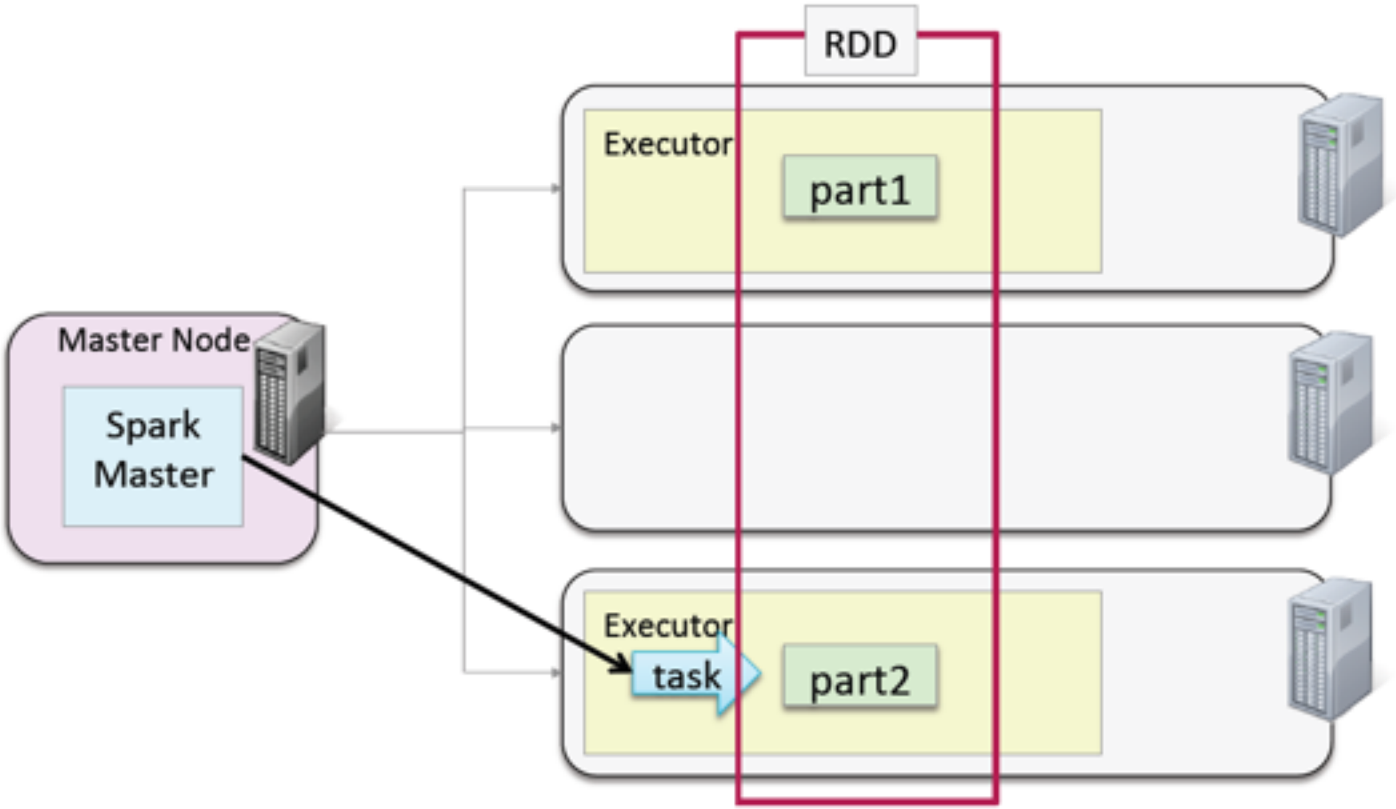
RDD Fault-Tolerance

- **What happens if a cached partition becomes unavailable?**



RDD Fault-Tolerance

- **The SparkMaster starts a new task to recompute the partition on a different node**



Persistence Levels

- The cache method stores data in memory only
- The persist method offers other Storage Levels
  - MEMORY_ONLY (default) - same as cache
  - MEMORY_AND_DISK - Store partitions on disk if they do not fit in memory
    - Called spilling
  - DISK_ONLY - Store all partitions on disk

```
> myrdd.persist(StorageLevel.DISK_ONLY)
```

Persistence Levels

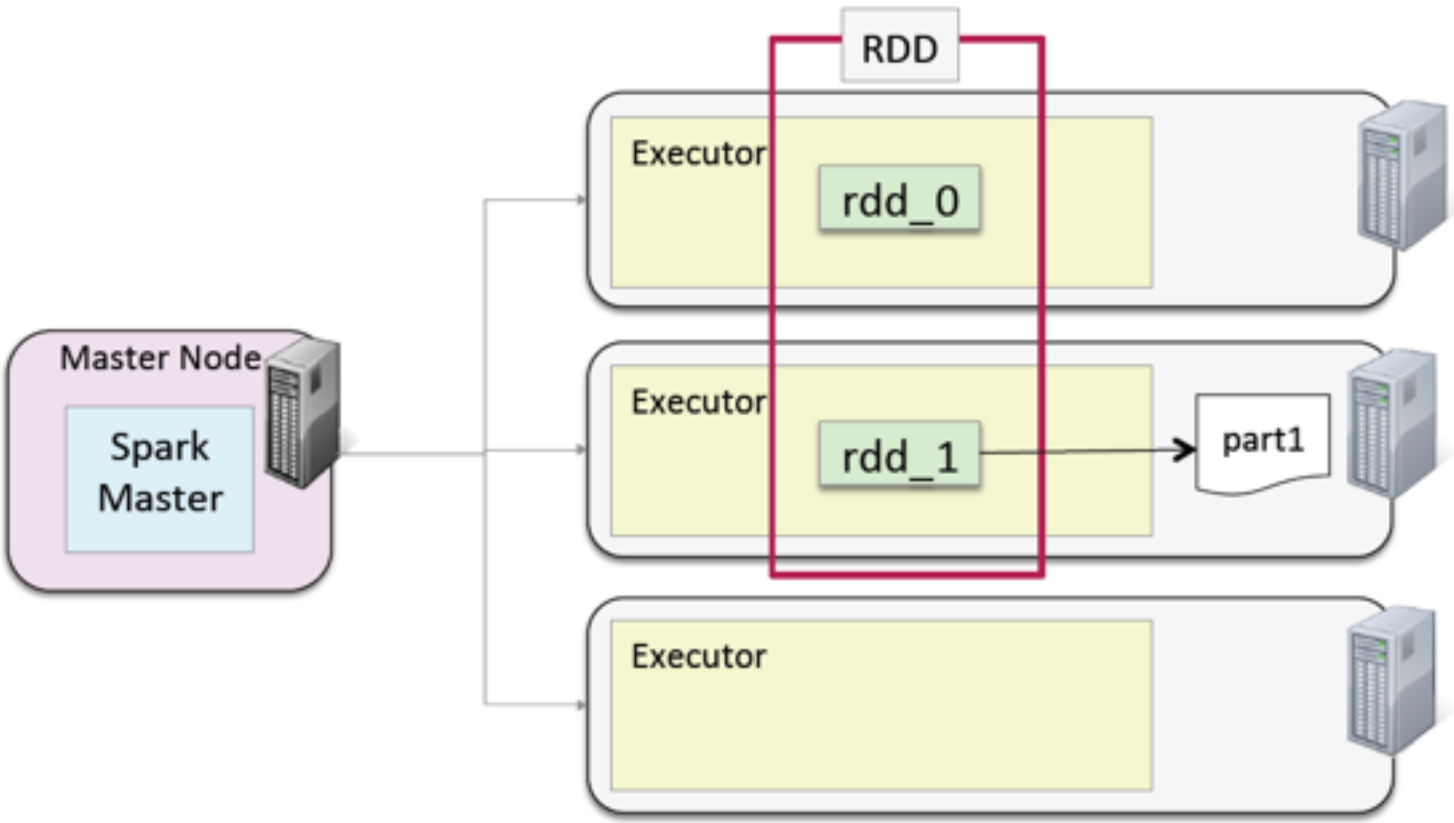
- You can choose to serialize the data in memory
  - MEMORY_ONLY_SER and MEMORY_AND_DISK_SER
  - Much more space efficient
  - Less time efficient
    - Choose a fast serialization library
- Replication - store partitions on two nodes
  - MEMORY_ONLY_2, MEMORY_AND_DISK_2, etc.

Changing Persistence Options

- To stop persisting and remove from memory and disk
  - rdd.unpersist()
- To change an RDD to a different persistence level
  - Unpersist first

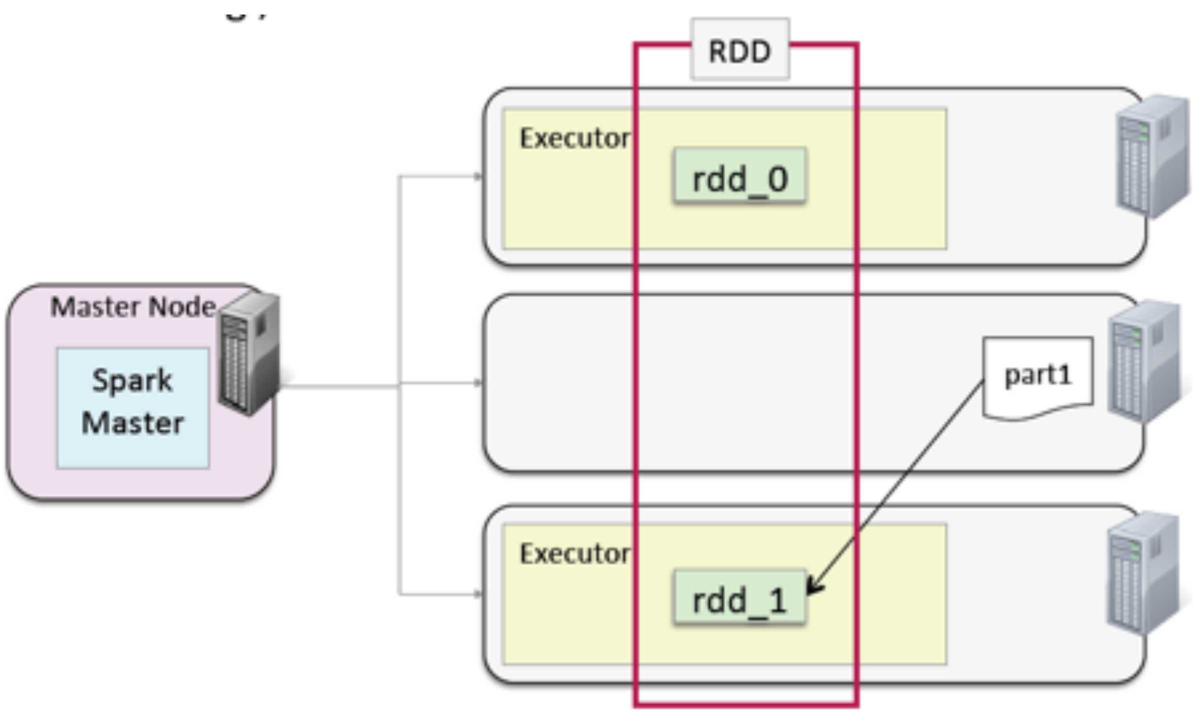
Distributed Disk Persistence

- **Disk persisted partitions are stored in local files**



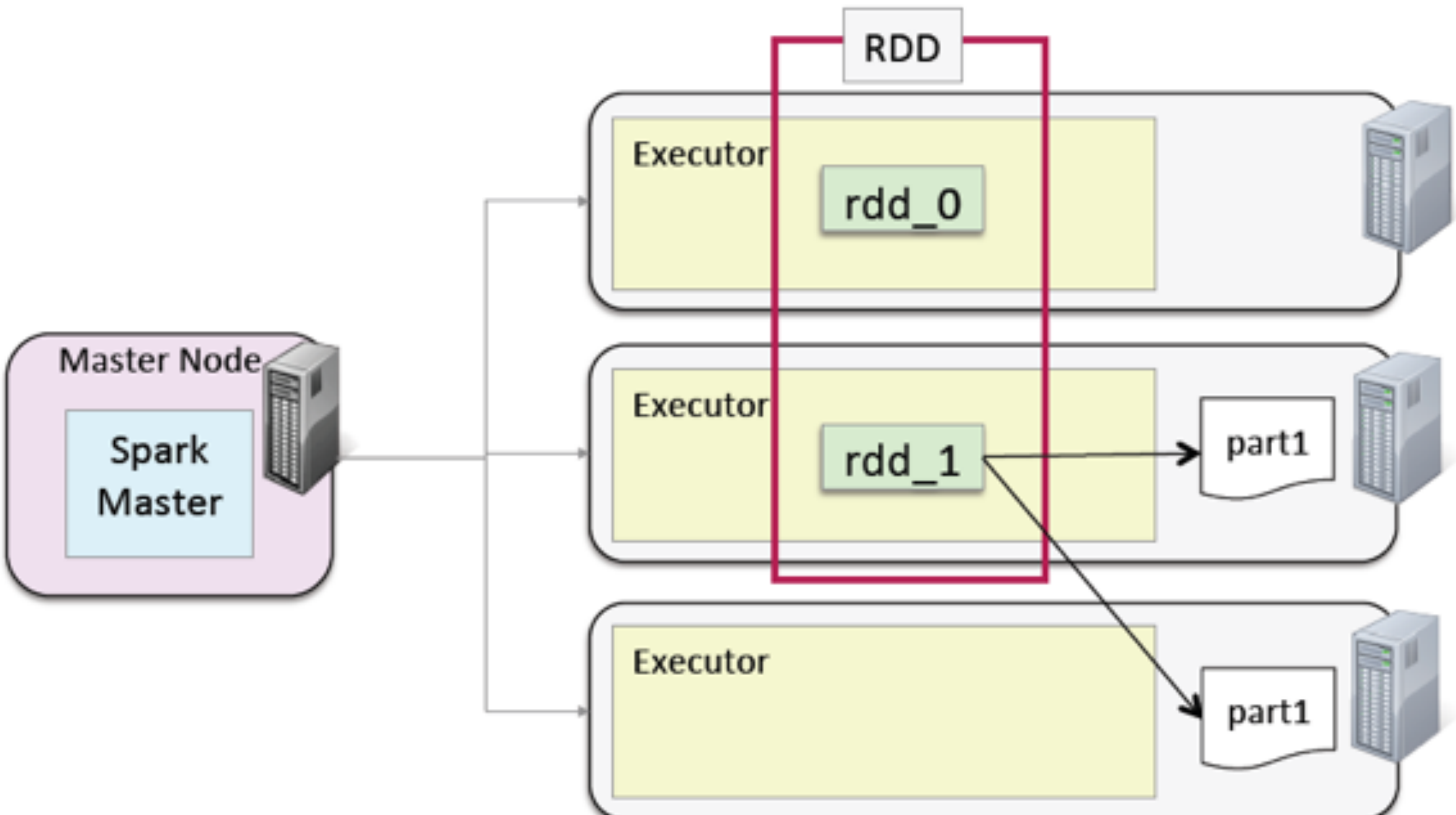
Distributed Disk Persistence

- Data on disk will be used to recreate the partition if possible
  - Will be recomputed if the data is unavailable
  - e.g., the node is down



Replication

- **Persistence replication makes recomputation less likely to be necessary**

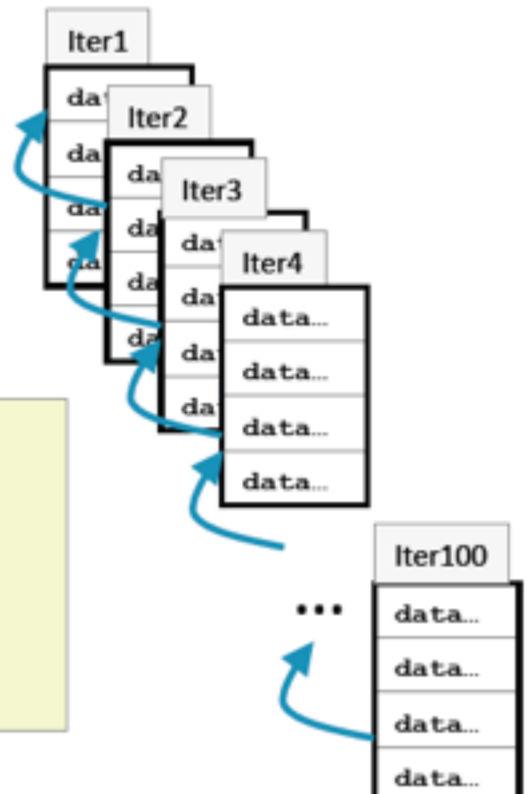


When and Where to Cache

- When should you cache a dataset?
  - When a dataset is likely to be re-used
    - e.g., iterative algorithms, machine learning
- How to choose a persistence level
  - Memory only - when possible, best performance
    - Save space by saving as serialized objects in memory if necessary
  - Disk - choose when recomputation is more expensive than disk read
    - e.g., expensive functions or filtering large datasets
  - Replication - choose when recomputation is more expensive than memory"

Checkpointing

- **Maintaining RDD lineage provides resilience but can also cause problems**
    - e.g., iterative algorithms, streaming
  - **Recovery can be very expensive**
  - **Potential stack overflow**

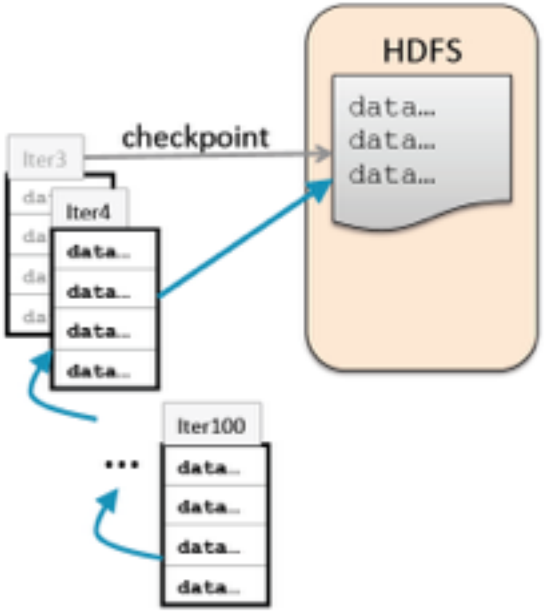



```
myrdd = ...initial-value...
while x in xrange(100):
   myrdd = myrdd.transform(...)
myrdd.saveAsTextFile()
```

Checkpointing

- Checkpointing saves the data to HDFS
- Provides fault-tolerant storage across nodes
- Lineage is not saved
- Must be checkpointed before any actions on the RDD

```
sc.setCheckpointDir(directory)
myrdd = ...initial-value...
while x in xrange(100):
  myrdd = myrdd.transform(...)
  if x % 3 == 0:
    myrdd.checkpoint()
    myrdd.count()
myrdd.saveAsTextFile()
```



Hands-On Exercise:

Caching RDDs

Checkpointing RDDs

HANDS-ON EXERCISES

Chapter 8

WRITING SPARK APPLICATIONS

Spark Shell vs. Spark Applications

- The Spark Shell allows interactive exploration and manipulation of data
  - REPL using Python or Scala
- Spark applications run as independent programs
  - Python, Scala, or Java
  - e.g., ETL processing, Streaming, and so on

The SparkContext

- Every Spark program needs a SparkContext
  - The interactive shell creates one for you
  - You create your own in a Spark application
  - Named sc by convention

Scala Example: WordCount

```
import org.apache.spark.SparkContext
import org.apache.spark.SparkContext._

object WordCount {
  def main(args: Array[String]) {
    if (args.length < 1) {
      System.err.println("Usage: WordCount <file>")
      System.exit(1)
    }

    val sc = new SparkContext()

    val counts = sc.textFile(args(0)).
       flatMap(line => line.split("\\W")).
       map(word => (word,1)).
       reduceByKey(_ + _)

    counts.take(5).foreach(println)
  }
}
```

Running a Spark Application

- **The easiest way to run a Spark Application is using the spark-submit script**

Python

```
$ spark-submit WordCount.py fileURL
```

Scala/
Java

```
$ spark-submit --class WordCount \
  MyJarFile.jar fileURL
```

Running a Spark Application

- Some key spark-submit options
  - --help - explain available options
  - --master - equivalent to MASTER environment variable for Spark Shell
    - local[*] - run locally with as many threads as cores (default)
    - local[n] - run locally with n threads
    - local - run locally with a single thread
    - master URL, e.g., spark://masternode:7077
  - --deploy-mode - either client or cluster
  - --name - application name to display in the UI (default is the Scala/Java class or Python program name)
  - --jars - additional JAR files (Scala and Java only)
  - --pyfiles - additional Python files (Python only)
  - --driver-java-options - parameters to pass to the driver JVM

Hands-On Exercise: Writing and Running a Spark Application

HANDS-ON EXERCISE: WRITING AND RUNNING A SPARK APPLICATION

Spark Application Configuration

- Spark provides numerous properties for configuring your application
- Some example properties
  - spark.master
  - spark.app.name
  - spark.local.dir - where to store local files such as shuffle output (default/tmp)
  - spark.ui.port - port to run the Spark Application UI (default 4040)
  - spark.executor.memory - how much memory to allocate to each Executor (default 512m)
- Most are more interesting to system administrators than developers
- Spark Applications can be configured
  - At run time or
  - Programmatically

Run-time Configuration Options

- spark-submit script
  - e.g., spark-submit --master spark://masternode:7077
- Properties file
  - Tab - or space-separated list of properties and values
  - Load with spark-submit --properties-file filename
    - Example:

```
spark.master       spark://masternode:7077
spark.local.dir    /tmp
spark.ui.port      4444
```

- Site defaults properties file
  - $SPARK_HOME/conf/spark-defaults.conf
  - Template file provided

Setting Configuration Properties Programmatically

- Spark configuration settings are part of the SparkContext
- Configure using a SparkConf object
- Some example functions
  - setAppName(name)
  - setMaster(master)
  - set(property-name, value)
- Set functions return a SparkConf object to support chaining

SparkConf Example (Scala)

```
import org.apache.spark.SparkContext
import org.apache.spark.SparkContext._
import org.apache.spark.SparkConf

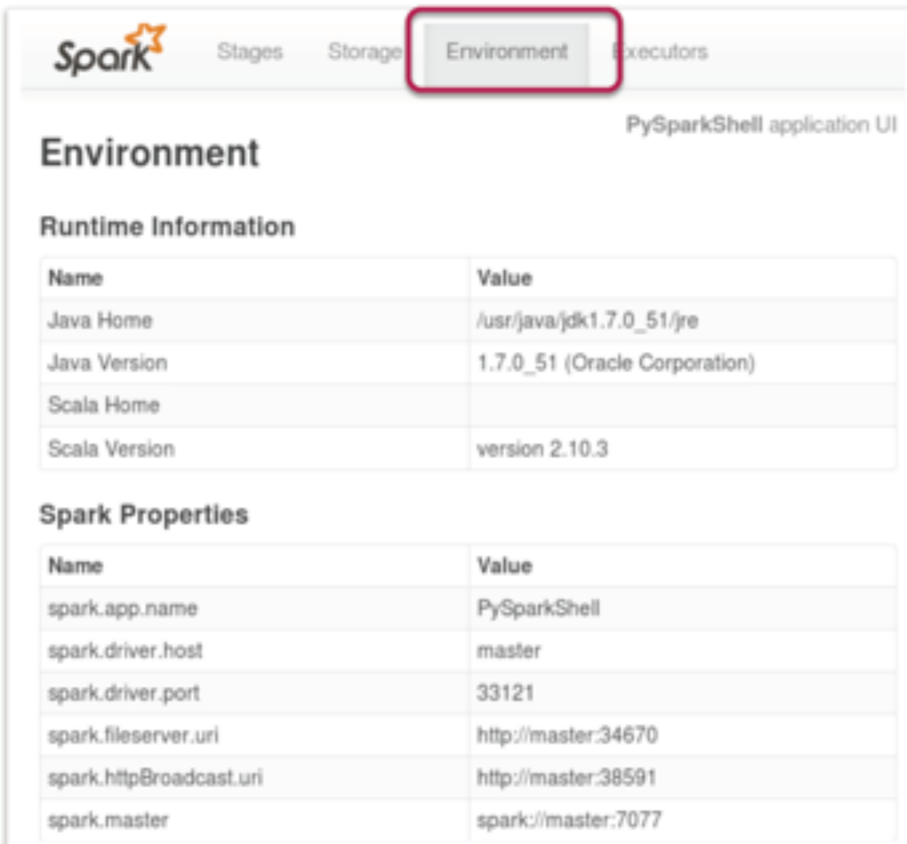
object WordCount {
  def main(args: Array[String]) {
    if (args.length < 1) {
      System.err.println("Usage: WordCount <file>")
      System.exit(1)
    }

    val sconf = new SparkConf()
      .setAppName("Word Count")
      .set("spark.ui.port","4141")
    val sc = new SparkContext(sconf)

    val counts = sc.textFile(args(0)).
       flatMap(line => line.split("\\W")).
       map(word => (word,1)).
       reduceByKey(_ + _)
    counts.take(5).foreach(println)
  }
}
```

Viewing Spark Properties

- You can view the Spark
- property setting in the Spark Application UI



Spark Logging

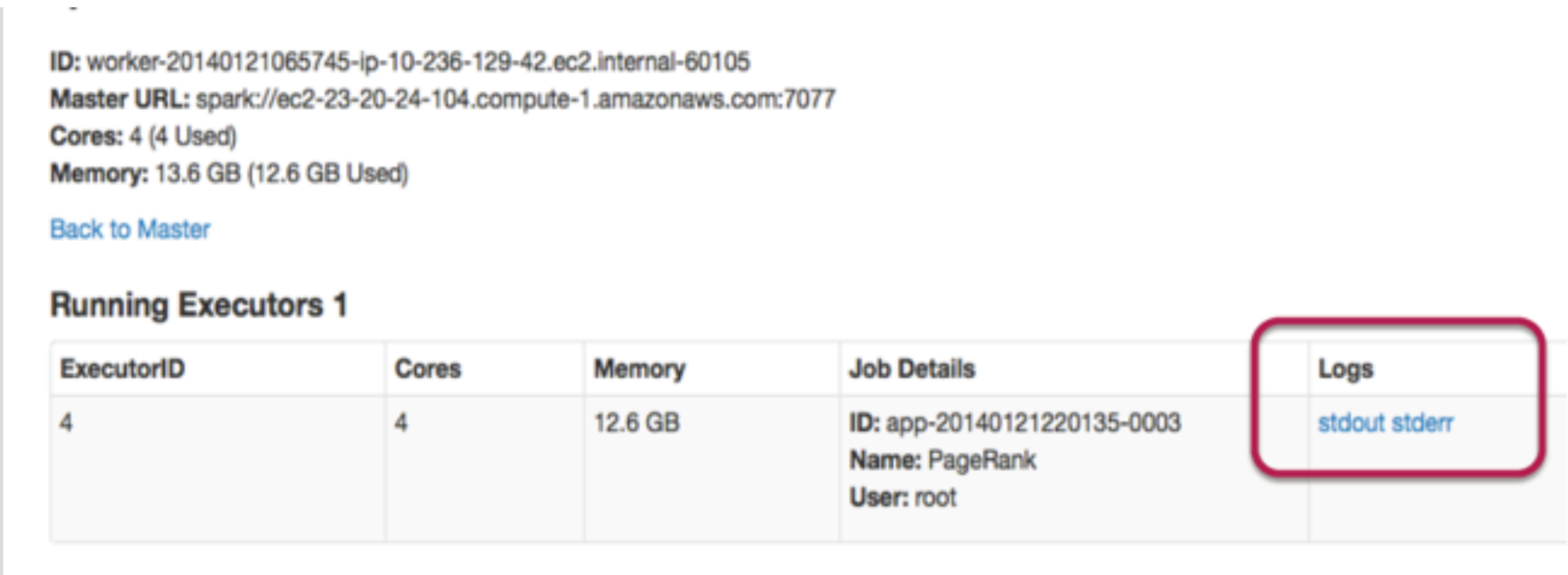
- Spark uses Apache Log4j for logging
  - Allows for controlling logging at runtime using a properties file
  - Enable or disable logging, set logging levels, select output destination
  - For more info see http://logging.apache.org/log4j/1.2/
- Log4j provides several logging levels
  - Fatal
  - Error
  - Warn
  - Info
  - Debug
  - Trace
  - Off

Spark Log Files

- Log file locations depend on your cluster management platform
- Spark Standalone defaults:
  - Spark daemons: /usr/hdp/current/spark/logs
  - Individual tasks: $SPARK_HOME/work on each worker node

Spark Worker UI – Log File Access

- Log file locations depend on your cluster management platform
- Spark Standalone defaults:
  - Spark daemons: /var/log/spark
  - Individual tasks: $SPARK_HOME/work on each worker node



Configuring Logging

- Logging levels can be set for the cluster, for individual applications, or even for specific components or subsystems
- Default for machine: $SPARK_HOME/conf/log4j.properties
  - Start by copying log4j.properties.template

log4j.properties.template

```
# Set everything to be logged to the console
log4j.rootCategory=INFO, console
log4j.appender.console=org.apache.log4j.ConsoleAppender
log4j.appender.console.target=System.err
…
```

Configuring Logging

- Spark will use the first log4j.properties file it finds in the Java classpath
- Spark Shell will read log4j.properties from the current directory
  - Copy log4j.properties to the working directory and edit

*…my-working-directory/*log4j.properties

```
# Set everything to be logged to the console
log4j.rootCategory=DEBUG, console
log4j.appender.console=org.apache.log4j.ConsoleAppender
log4j.appender.console.target=System.err
…
```

Hands-On Exercise: Configuring Spark Applications

HANDS-ON EXERCISE: CONFIGURING SPARK APPLICATIONS