DEVELOPER TRAINING FOR APACHE SPARK

Chapter 1

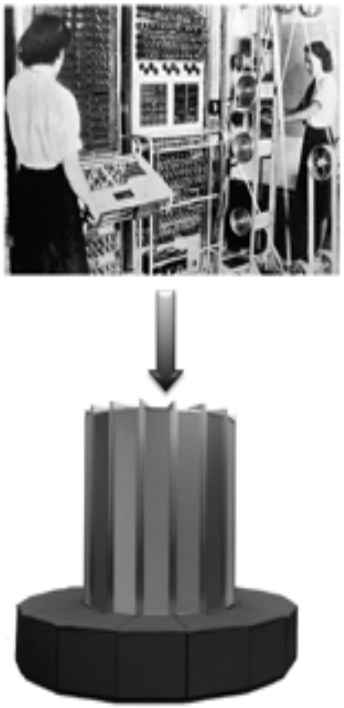
WHY SPARK?

Why Spark?

- In this chapter you will learn
- What problems exist with traditional large-scale computing systems
- How Spark addresses those issues
- Some typical big data questions Spark can be used to answer

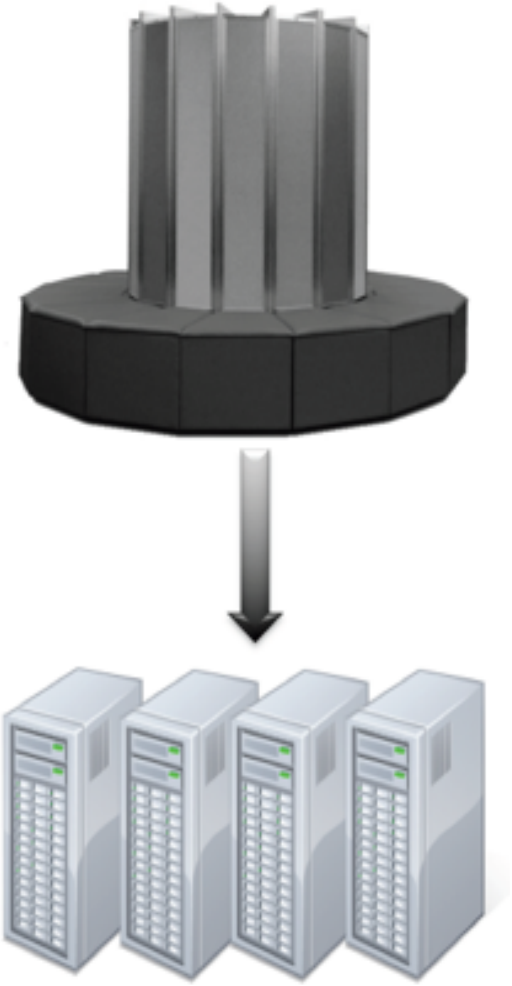
Traditional Large-Scale Computation

- Traditionally, computation has been processor bound
  - Relatively small amounts of data
  - Lots of complex processing
- The early solution: bigger computers
  - Faster processor, more memory
  - But even this couldn't keep up



Distributed Systems

- **The better solution: more computers**
  - **Distributed systems - use multiple machines for a single job**



"In pioneer days they used oxen for heavy pulling, and when one ox couldn't budge a log, we didn't try to grow a larger ox. We shouldn't be trying for bigger computers, but for more systems of computers."

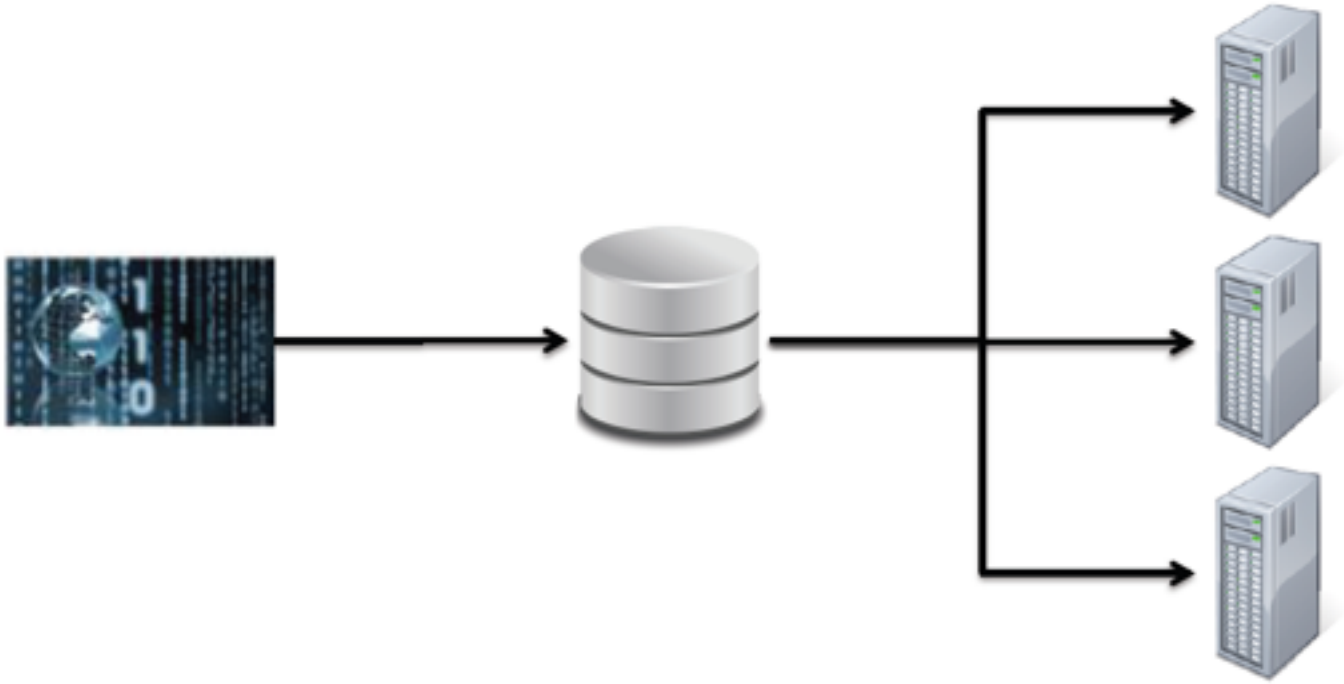
-Grace Hopper

Distributed Systems: Challenges

- **Challenges with distributed systems**
  - **Programming complexity**
  - **Keeping data and processes in sync**
  - **Finite bandwidth**
  - **Partial failures**

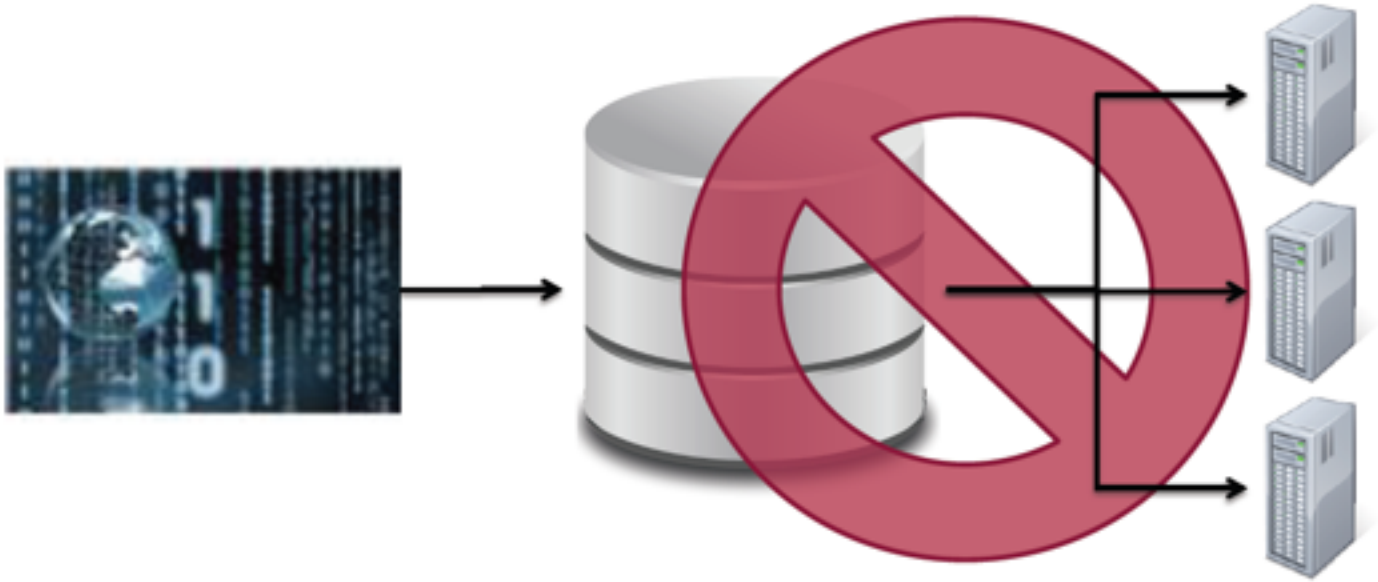
Distributed Systems: The Data Bottleneck

- **Traditionally, data is stored in a central location**
- **Data is copied to processors at runtime**
- **Fine for limited amounts of data**



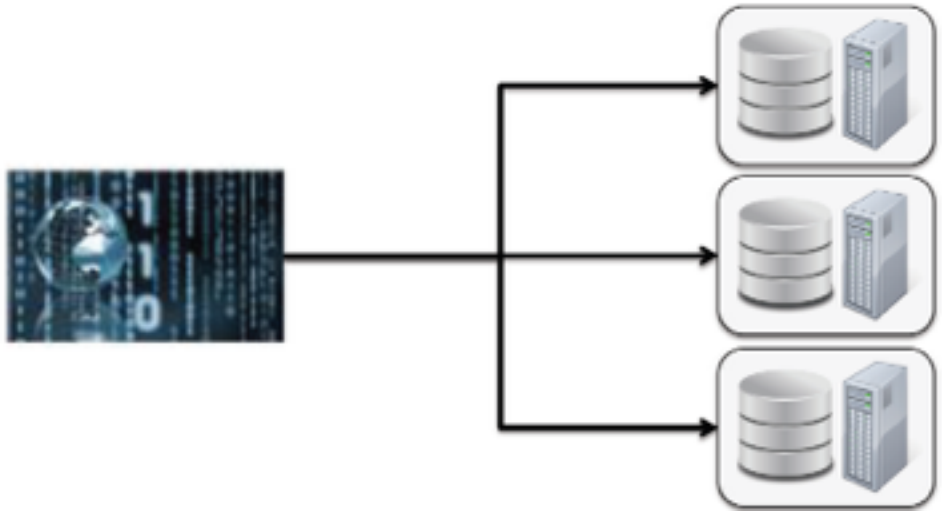
Distributed Systems: The Data Bottleneck

- **Modern systems have much more data**
  - **terabytes+ a day**
  - **petabytes+ total**
- **We need a new approach…**

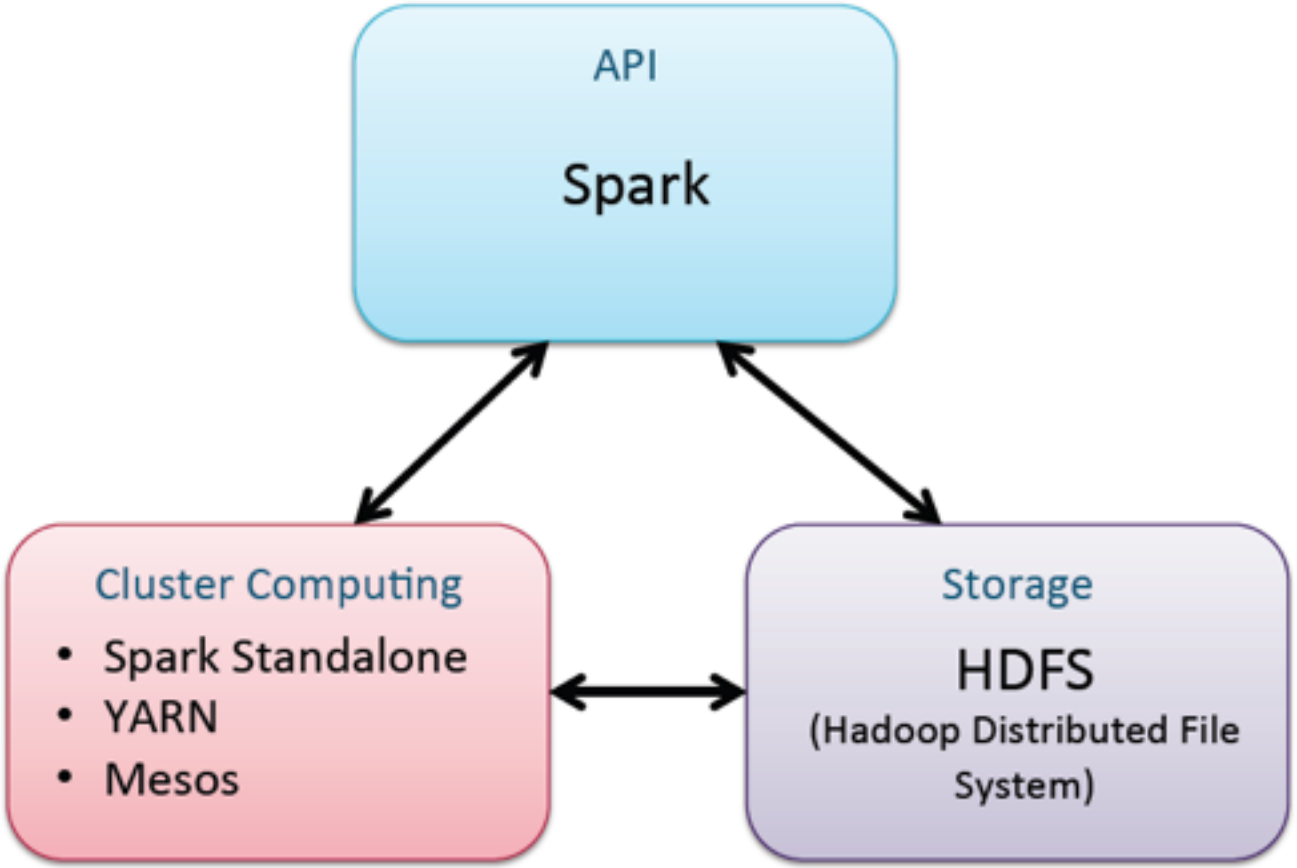


Big Data Processing

- Hadoop introduced a radical new approach based on two key concepts
  - Distribute data when the data is stored
  - Run computation where the data is
- Spark takes this new approach to the next level
  - Data is distributed in memory



Distributed Processing with the Spark Framework

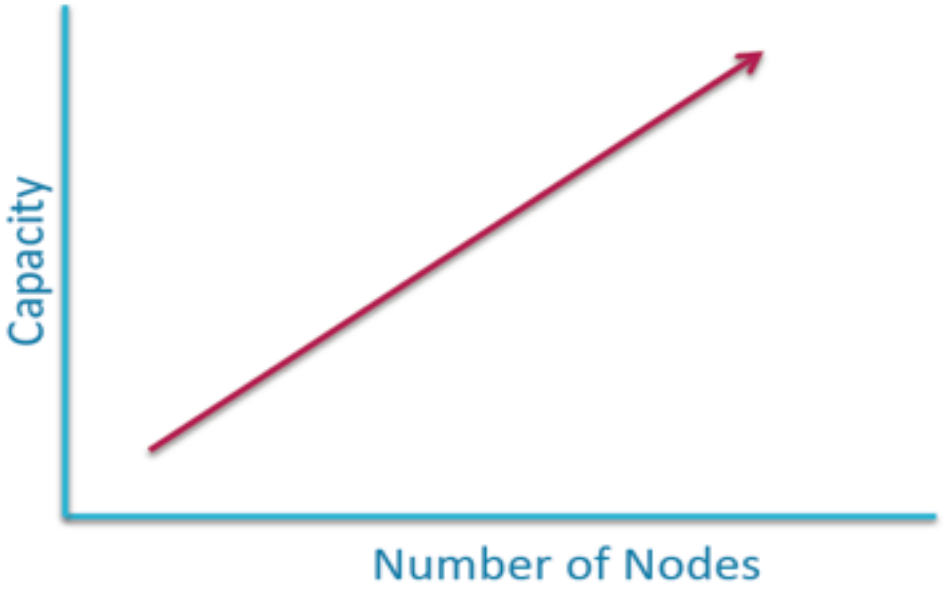


Advantages of Spark

- High level programming framework
  - Programmers can focus on logic, not plumbing
- Cluster computing
  - Application processes are distributed across a cluster of worker nodes
  - Managed by a single “master”
  - Scalable and fault tolerant
- Distributed storage
  - Data is distributed when it is stored
  - Replicated for efficiency and fault tolerance
  - “Bring the computation to the data”
- Data in memory
  - Configurable caching for efficient iteration

Scalability

- Increasing load results in a graceful decline in performance
  - Not failure of the system
- Adding nodes adds capacity proportionally



Fault Tolerance

- Node failure is inevitable
- What happens?
  - System continues to function
  - Master re-assigns tasks to a different node
  - Data replication= no loss of data
  - Nodes which recover rejoin the cluster automatically

Who Uses Spark?

- Yahoo!
  - Personalization and ad analytics
- Conviva
  - Real-time video stream optimization
- Technicolor
  - Real-time analytics for telco clients
- Ooyala
  - Cross-device personalized video experience
- Plus…
  - Intel, Groupon, TrendMicro, Autodesk, Nokia, Shopify, ClearStory, Technicolor, and many more…

Common Spark Use Cases

- **Extract/Transform/Load (ETL)**
- **Text mining**
- **Index building**
- **Graph creation and analysis**
- **Pattern recognition**
- **Collaborative filtering**
- **Prediction models**
- **Sentiment analysis**
- **Risk assessment**

Benefits of Spark

- **Previously impossible or impractical analysis**
- **Lower cost**
- **Less time**
- **Greater flexibility**
- **Near-linear scalability**

Spark v. Hadoop MapReduce

- Spark takes the concepts of MapReduce to the next level
- Higher level API = faster, easier development
  - Low latency = near real-time processing
  - In-memory data storage = up to 100x performance improvement


```
sc.textFile(file) \
  .flatMap(lambda s: s.split()) \
  .map(lambda w: (w,1)) \
  .reduceByKey(lambda v1,v2: v1+v2)
  .saveAsTextFile(output)
```

```
public class WordCount {
  public static void main(String[] args) throws
    Job job = new Job();
    job.setJarByClass(WordCount.class);
    job.setJobName("Word Count");
    FileInputFormat.setInputPaths(job, new Path(args[0]));
    FileOutputFormat.setOutputPath(job, new Path(args[1]));
    job.setMapperClass(WordMapper.class);
    job.setReducerClass(SumReducer.class);
    job.setMapOutputKeyClass(Text.class);
    job.setMapOutputValueClass(IntWritable.class);
    job.setOutputKeyClass(Text.class);
    job.setOutputValueClass(IntWritable.class);
    boolean success = job.waitForCompletion(true);
    System.exit(success ? 0 : 1);
  }
}

public class WordMapper extends Mapper<LongWritable, Text, Text,
IntWritable> {
public void map(LongWritable key, Text value,
Context context) throws IOException, InterruptedException {
    String line = value.toString();
    for (String word : line.split("\\W+")) {
      if (word.length() > 0)
        context.write(new Text(word), new IntWritable(1));
      }
    }
  }
}

public class SumReducer extends Reducer<Text, IntWritable, Text,
IntWritable> {
  public void reduce(Text key, Iterable<IntWritable>
  values, Context context) throws IOException, InterruptedException {
    int wordCount = 0;
    for (IntWritable value : values) {
      wordCount += value.get();
    }
    context.write(key, new IntWritable(wordCount));
  }
}
```

Chapter 2

SPARK BASICS

What is Apache Spark?

- Apache Spark is a fast and general engine for large scale data processing
- Written in Scala
  - Functional programming language that runs in a JVM
- Spark Shell
  - Interactive- for learning or data exploration
  - Python or Scala
- Spark Applications
  - For large scale data processing
  - Python, Scala , or Java

Spark Shell

- The Spark Shell provides interactive data exploration (REPL)
- Writing Spark applications without the shell will be covered later
-
- Python Shell: pyspark

```
$ pyspark

Welcome to
      ____              __
     / __/__  ___ _____/ /__
    _\ \/ _ \/ _ `/ __/  '_/
   /__ / .__/\_,_/_/ /_/\_\   version 0.9.1
      /_/

Using Python version 2.6.6 (r266:84292, Jan
22 2014 09:42:36)
Spark context available as sc.

>>>
```

Scala Shell: spark-shell

```
$ spark-shell

Welcome to
      ____              __
     / __/__  ___ _____/ /__
    _\ \/ _ \/ _ `/ __/  '_/
   /___/ .__/\_,_/_/ /_/\_\   version 0.9.1
      /_/

Using Scala version 2.10.3 (Java HotSpot(TM)
64-Bit Server VM, Java 1.7.0_51)
Created spark context..
Spark context available as sc.

scala>
```

REPL: Read/Evaluate/Print Loop

Spark Context

- Every Spark application requires a Spark Context
  - The main entry point to the Spark API
- Spark Shell provides a preconfigured Spark Context called sc

```
Using Python version 2.6.6 (r266:84292, Jan 22 2014 09:42:36)
Spark context available as sc.

>>> sc.appName
u'PySparkShell'
```

```
Using Scala version 2.10.3 (Java HotSpot(TM) 64-Bit Server VM, Java
1.7.0_51)
Created spark context..
Spark context available as sc.

scala> sc.appName
res0: String = Spark shell
```

RDD (Resilient Distributed Dataset)

- **RDD (Resilient Distributed Dataset)**
  - **Resilient – if data in memory is lost, it can be recreated**
  - **Distributed – stored in memory across the cluster**
  - **Dataset – initial data can come from a file or be created programmatically**
- **RDDs are the fundamental unit of data in Spark**
- **Most Spark programming consists of performing operations on RDDs**

Creating an RDD

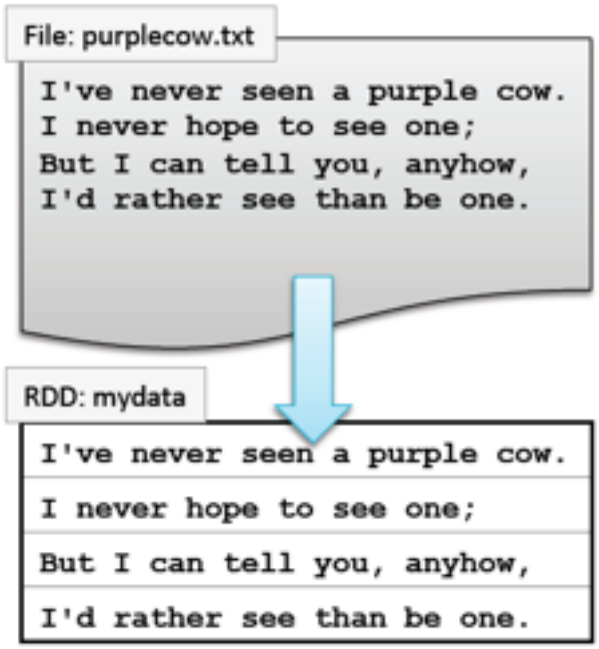
- **Three ways to create an RDD**
  - **From a file or set of files**
  - **From data in memory**
  - **From another RDD**

File-Based RDDs

- For file-based RDDS, use Spark Context.textFile
  - Accepts a single file, a wildcard list of files, or a comma-separated list of files
  - Examples
    - sc.textFile("myfile.txt")
    - sc.textFile("mydata/*.log")
    - sc.textFile("myfile1.txt,myfile2.txt")
    - Each line in the file(s) is a separate record in the RDD
- Files are referenced by absolute or relative URI
  - Absolute URI: file:/home/training/myfile.txt
  - Relative URI (uses default file system): myfile.txt

Example: A File-based RDD

```
> mydata = sc.textFile("purplecow.txt")
…
14/01/29 06:20:37 INFO storage.MemoryStore:
   Block broadcast_0 stored as values to
   memory (estimated size 151.4 KB, free 296.8
   MB)

> mydata.count()

…
14/01/29 06:27:37 INFO spark.SparkContext: Job
   finished: take at <stdin>:1, took
   0.160482078 s
4
```



RDD Operations

- Two types of RDD operations
  - Actions - return values
  - Transformations- define a new RDD based on the current one(s)
- Pop quiz:
  - Which type of operation is count()?

RDD Operations: Actions

- Some common actions
  - count() - return the number of elements
  - take(n) - return an array of the first n elements
  - collect()- return an array of all elements
  - saveAsTextFile(file) - save to text file(s)

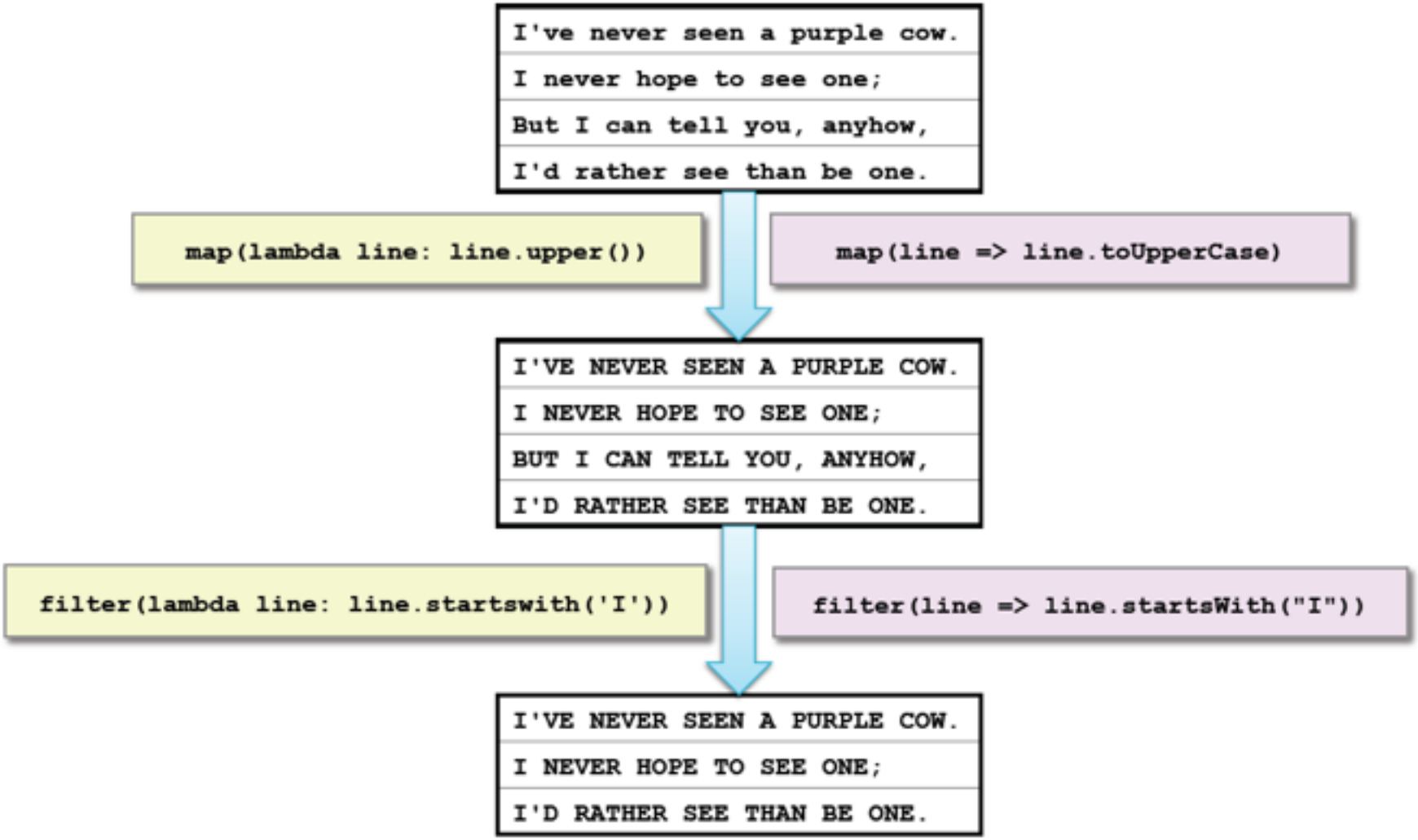
```
> mydata =
  sc.textFile("purplecow.txt")

> mydata.count()
4

> for line in mydata.take(2):
    print line
I've never seen a purple cow.
I never hope to see one;
```

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

> mydata.count
4

> for (line <- mydata.take(2))
    println(line)
I've never seen a purple cow.
I never hope to see one;
```

RDD Operations: Transformations

- Transformations create a new RDD from an existing one
- RDDs are immutable
  - Data in an RDD is never changed
  - Transform in sequence to modify the data as needed
- Some common transformations
  - map(function) - creates a new RDD by performing a function on each record in the base RDD
  - filter(function) - creates a new RDD by including or excluding each record in the base RDD according to a boolean function

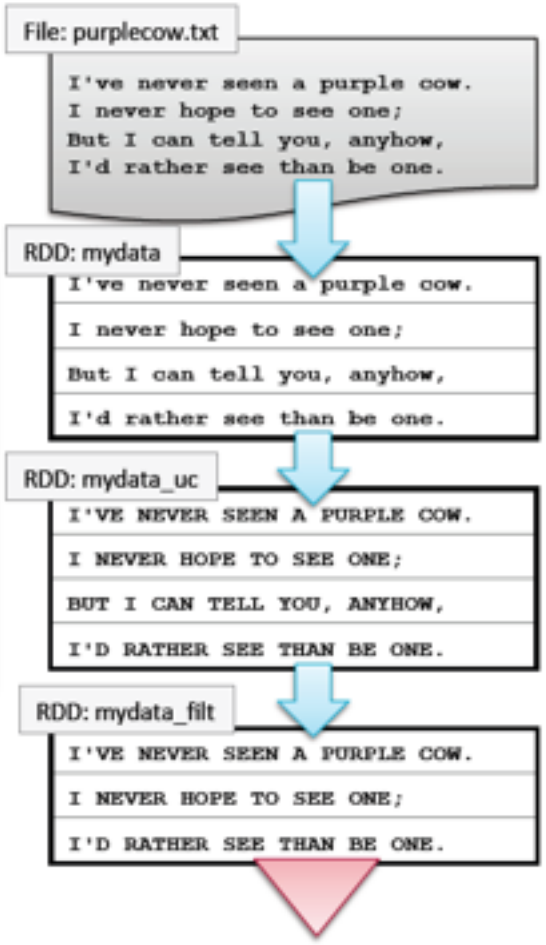
Example: map and filter Transformations



Lazy Execution

- **Data in RDDs is not processed until an action is performed**

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



Chaining Transformations

- **Transformations may be chained together**

```
> mydata = sc.textFile("purplecow.txt")
> mydata_uc = mydata.map(lambda line: line.upper())
>           = mydata_uc.filter(lambda line: line.startswith('I'))
> mydata_filt.count()
3
```

is exactly equivalent to

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

Functional Programming in Spark

- **Spark depends heavily on the concepts of functional programming**
  - **Functions are the fundamental unit of programming**
  - **Functions have input and output only**
  - **No state or side effects**
- **Key concepts**
  - **Passing functions as input to other functions**
  - **xAnonymous functions**

Passing Functions as Parameters

- Many RDD operations take functions as parameters
- Pseudocode for the RDD map operation
  - Applies function fn to each record in the RDD

```
RDD {
    map(fn(x)) {
        foreach record in rdd
        emit fn(record)
    }
}
```

Example: Passing Named Functions

```
> def toUpper(s: String): String =
    { s.toUpperCase }
> val mydata = sc.textFile("purplecow.txt")
> mydata.map(toUpper).take(2)
```

Anonymous Functions

- **Functions defined in-line without an identifier**
  - **Best for short, one-off functions**
- **Supported in many programming languages**

Example: Passing Anonymous Functions

```
> mydata.map(line => line.toUpperCase()).take(2)
```

OR

```
> mydata.map(_.toUpperCase()).take(2)
```

Example: Java

Java 7

```
...
  JavaRDD<String> lines = sc.textFile("file");
  JavaRDD<String> lines_uc = lines.map(
    new MapFunction<String, String>() {
      public String call(String line) {
        return line.toUpperCase();
      }
    }
...
```

Java 8

```
...
 JavaRDD<String> lines = sc.textFile("file");
 JavaRDD<String> lines_uc = lines.map(
    line -> line.toUpperCase());
...
```

Now, please do the following three Hands-On Exercises

1. Viewing the Spark Documentation
2. Using the Spark Shell
3. Getting Started with RDDs

HANDS-ON EXERCISES

Chapter 3

WORKING WITH RDDS

RDDs

- RDDs can hold any type of element
  - Primitive types: integers, characters, booleans, etc.
  - Sequence types: strings, lists, arrays, tuples, dicts, etc. (including nested data types)
  - Scala/Java Objects (if serializable)
  - Mixed types
- Some types of RDDs have additional functionality
  - Pair RDDs
  - RDDs consisting of Key-Value pairs
  - Double RDDs
  - RDDs consisting of numeric data

Creating RDDs From Collections

- **You can create RDDs from collections instead of files**

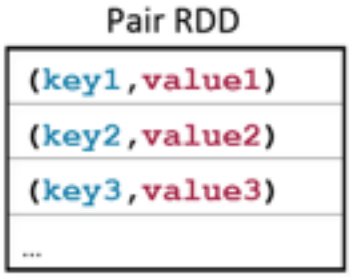
```
> randomnumlist = \
    [random.uniform(0,10) for _ in xrange(10000)]
> randomrdd = sc.parallelize(randomnumlist)
> print "Mean: %f" % randomrdd.mean()
```

Some Other General RDD Operations

- Transformations
  - distinct - filter out duplicates
  - union - add all elements of two RDDs into a single new RDD
  - zip - pair the elements of one RDD with another
- Other RDD operations
  - first - return the first element of the RDD
  - for each - apply a function to each element in an RDD
  - top(n) - return the largest n elements using natural ordering
- Sampling operations
  - sample(percent) - create a new RDD with a sampling of elements
  - takeSample (percent) - return an array of sampled elements
- Double RDD operations
  - Statistical functions, e.g. mean, sum, variance, stdev

Pair RDDs

- Pair RDDs are a special form of RDD
  - Each element must be a key-value pair (a two-element tuple)
  - Keys and values can be any type
- Why?
  - Use with Map Reduce algorithms
  - Many additional functions are available for common data processing needs
    - e.g., sorting, joining, grouping, counting, etc.



Creating Pair RDDs

- **The first step in most workflows is to get the data into key/value form**
  - **What should the RDD should be keyed on?**
  - **What is the value?**
- **Commonly used functions to create Pair RDDs**
  - **map**
  - **flatMap**
  - **keyBy**

Example: A Simple Pair RDD

```
> users = sc.textFile(file) \
    .map(lambda line: line.split('\t')) \
    .map(lambda fields: (fields[0],fields[1]))
```

```
> val users = sc.textFile(file) \
    .map(line => line.split('\t')) \
    .map(fields => (fields(0),fields(1)))
```

```
user001   Fred Flintstone
user090   Bugs Bunny
user111   Harry Potter
…
```

```
(user001,Fred Flintstone)
(user090,Bugs Bunny)
(user111,Harry Potter)
…
```

Example: Keying Web Logs by User ID

```
> sc.textFile(logfile) \
    .keyBy(lambda line: line.split(' ')[2])
```

```
> sc.textFile(logfile) \
    .keyBy(line => line.split(' ')(2))
```

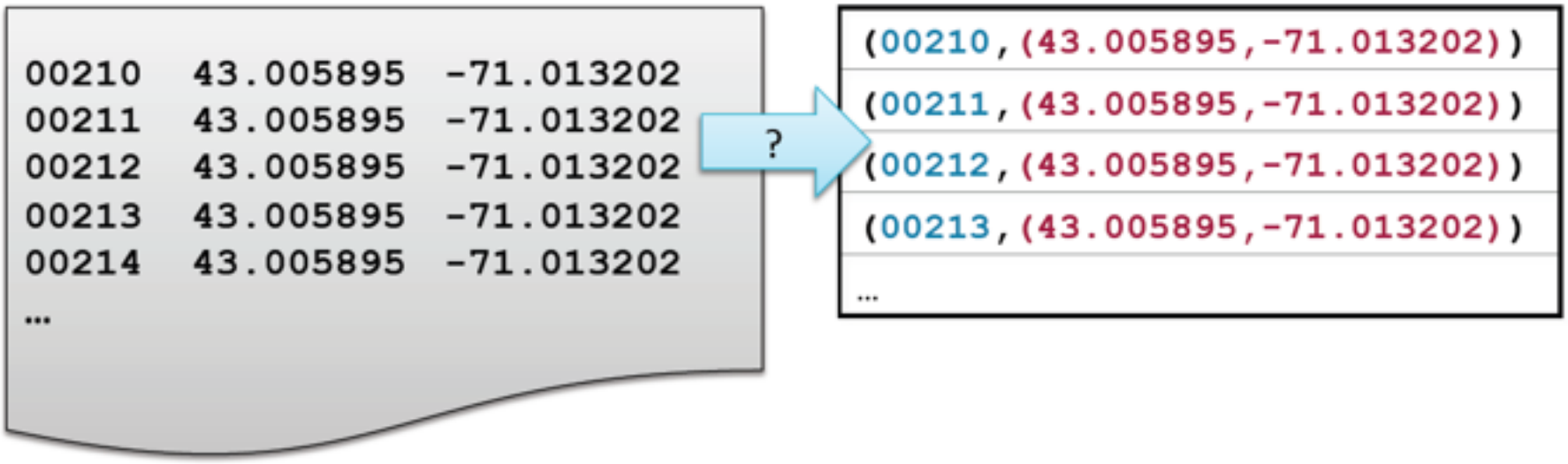
User ID

```
56.38.234.188 - 99788 "GET /KBDOC-00157.html HTTP/1.0" …
56.38.234.188 - 99788 "GET /theme.css HTTP/1.0" …
203.146.17.59 - 25254 "GET /KBDOC-00230.html HTTP/1.0" …
…
```

```
(99788,56.38.234.188 - 99788 "GET /KBDOC-00157.html…)
(99788,56.38.234.188 - 99788 "GET /theme.css…)
(25254,203.146.17.59 - 25254 "GET /KBDOC-00230.html…)
…
```

Pairs With Complex Values

- How would you do this?
  - Input: a list of postal codes with latitude and longitude
  - Output: postal code (key) and lat/long pair (value)



Pairs With Complex Values

```
> sc.textFile(file) \
  .map(lambda line: line.split()) \
  .map(lambda fields: (fields[0],(fields[1],fields[2])))
```

```
00210   43.005895  -71.013202
00211   43.005895  -71.013202
00212   43.005895  -71.013202
00213   43.005895  -71.013202
00214   43.005895  -71.013202
…
```

```
(00210,(43.005895,-71.013202))
(00211,(43.005895,-71.013202))
(00212,(43.005895,-71.013202))
(00213,(43.005895,-71.013202))
…
```

Mapping Single Rows to Multiple Pairs

- How would you do this?
  - Input: order numbers with a list of SKUs in the order
  - Output: order (key) and sku (value)

Mapping Single Rows to Multiple Pairs

- **Hint: map alone won't work**

```
00001    sku010:sku933:sku022
00002    sku912:sku331
00003    sku888:sku022:sku010:sku594
00004    sku411
```

```
(00001,(sku010,sku933,sku022))
(00002,(sku912,sku331))
(00003,(sku888,sku022,sku010,sku594))
(00004,(sku411))
```

Answer 2: Creating a Pair RDD

```
> sc.textFile(file) \
  .map(lambda line: line.split('\t')) \
  .flatMap(lambda fields:(fields[0],fields[1].split(':'))
```

00001	sku010:sku933:sku022
00002	sku912:sku331
00003	sku888:sku022:sku010:sku594
00004	sku411

(00001,sku010:sku933:sku022)
(00002,sku912:sku331)
(00003,sku888:sku022:sku010:sku594)
(00004,sku411)

(00001,sku010)
(00001,sku933)
(00001,sku022)
(00002,sku912)
(00002,sku331)
(00003,sku888)
…

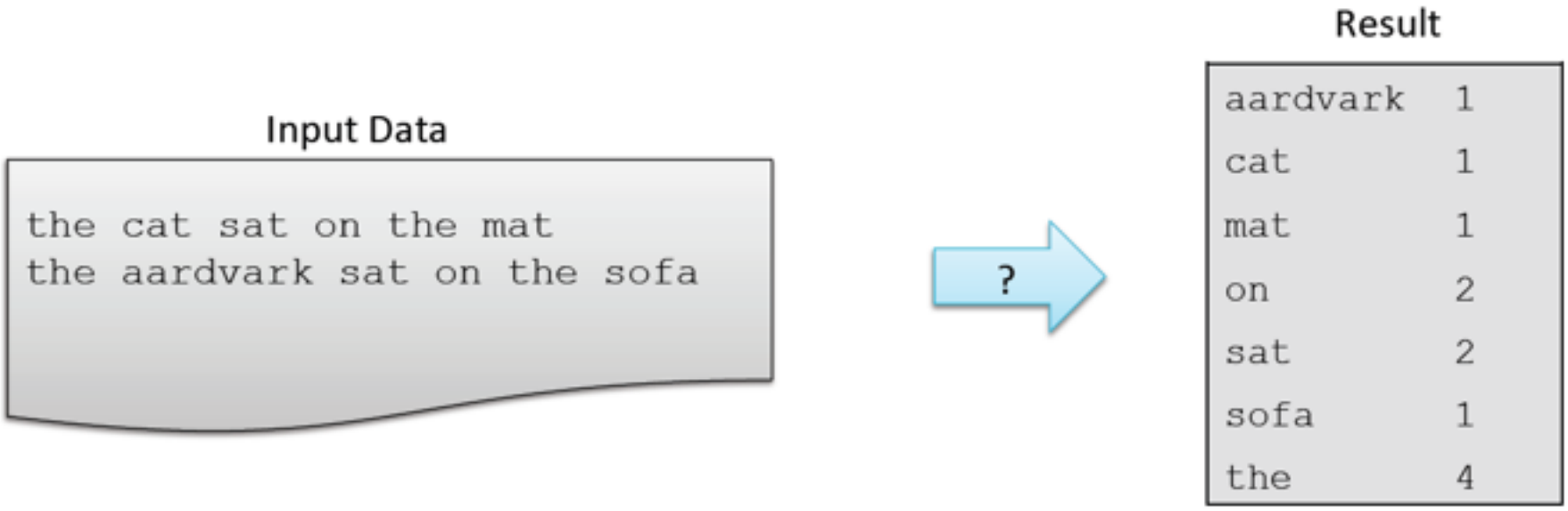
MapReduce

- MapReduce is a common programming model
  - Easily applicable to distributed processing of large data sets
- Hadoop MapReduce is the major implementation
  - Somewhat limited
    - Each job has one Map phase, one Reduce phase
    - Job output is saved to files
- Spark implements MapReduce with much greater flexibility
  - Map and Reduce functions can be interspersed
  - Results stored in memory
    - Operations can easily be chained

MapReduce in Spark

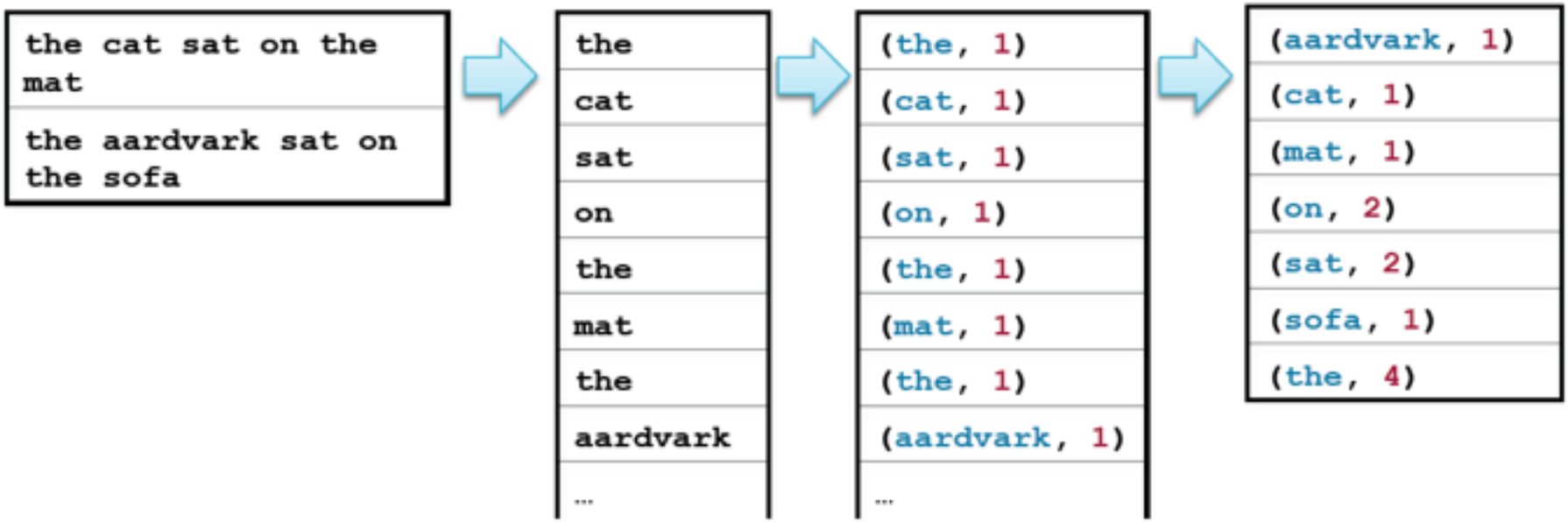
- MapReduce in Spark works on Pair RDDs
- Map
  - Operates on one record at a time
  - "Maps" each record to one or more new records
  - map and flatMap
- Reduce
  - Works on Map output
  - Consolidates multiple records
  - reduceByKey

MapReduce Example: Word Count



Example: Word Count

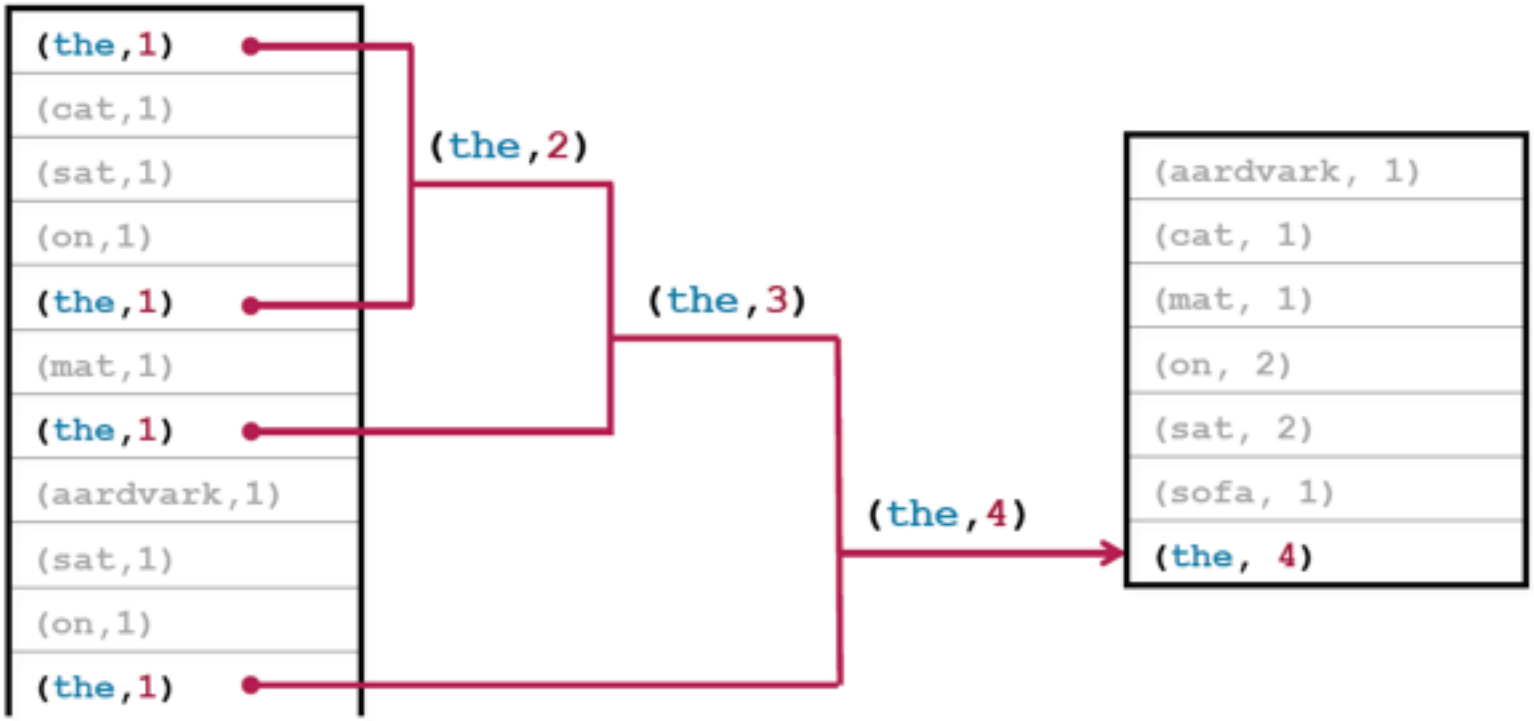
```
> counts = sc.textFile(file) \
    .flatMap(lambda line: line.split()) \
    .map(lambda word: (word,1)) \
    .reduceByKey(lambda v1,v2: v1+v2)
```



ReduceByKey

ReduceByKey functions must be
- Binary – combines values from two keys
- Commutative – x+y = y+x
- Associative – (x+y)+z = x+(y+z)

```
> counts = sc.textFile(file) \
  .flatMap(lambda line: line.split()) \
  .map(lambda word: (word,1)) \
  .reduceByKey(lambda v1,v2: v1+v2)
```



Word Count Recap (the Scala Version)

```
> val counts = sc.textFile(file) \
  .flatMap(line => line.split("\\W")) \
  .map(word => (word,1)) \
  .reduceByKey((v1,v2) => v1+v2)
```

OR

```
> val counts = sc.textFile(file) \
  .flatMap(_.split("\\W")) \
  .map(_,1)) \
  .reduceByKey(_+_)
```

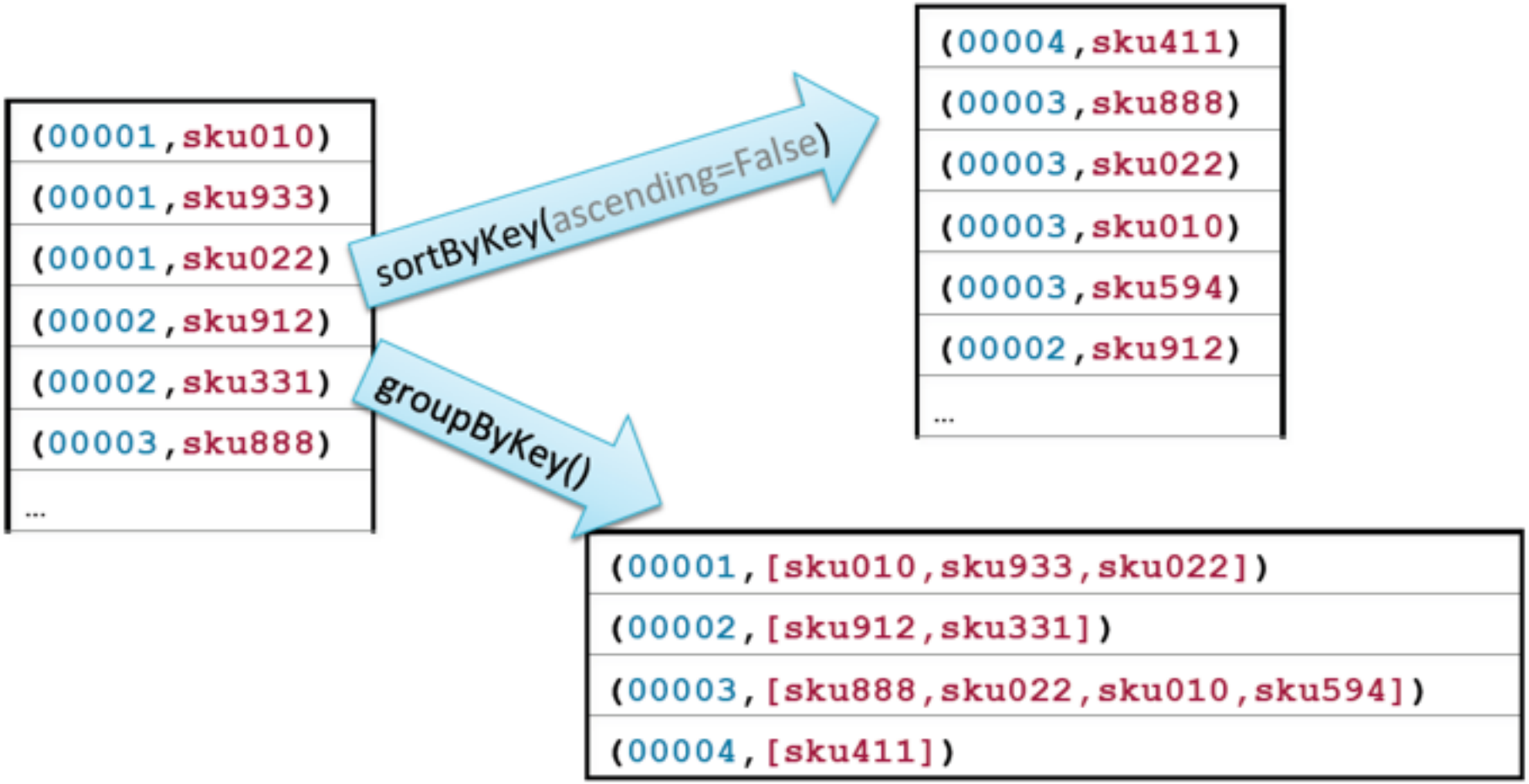
Why Do We Care About Counting Words?

- Word count is challenging over massive amounts of data
  - Using a single compute node would be too time-consuming
  - Number of unique words could exceed available memory
- Statistics are often simple aggregate functions
  - Distributive in nature
  - e.g., max, min, sum, count
- MapReduce breaks complex tasks down into smaller elements which can be executed in parallel
- Many common tasks are very similar to word count
  - e.g., log file analysis

Pair RDD Operations

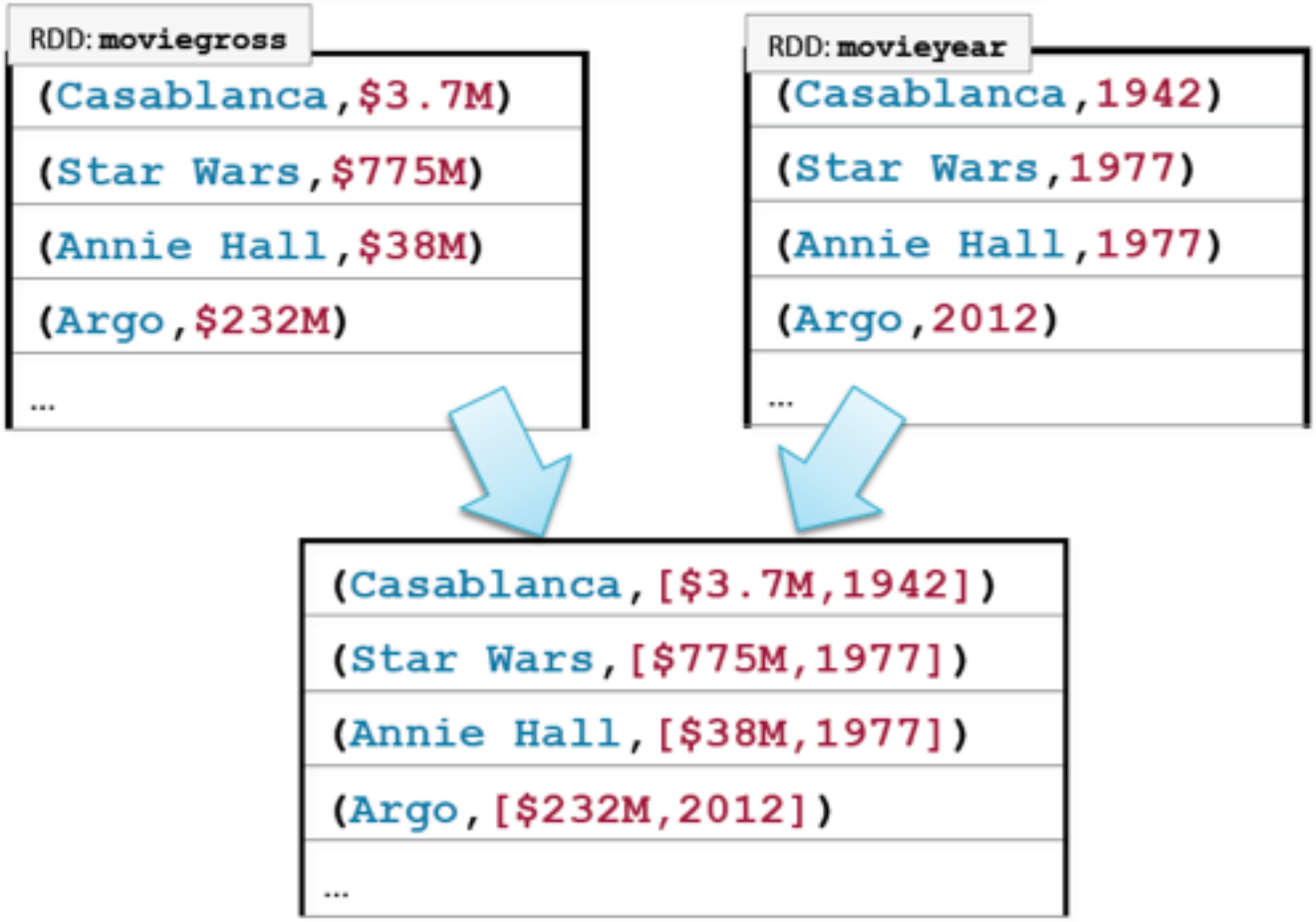
- In addition to map and reduce functions, Spark has several operations specific to Pair RDDs
- Examples
  - countByKey-return a map with the count of occurrences of each key
  - groupByKey - group all the values for each key in an RDD
  - sortByKey - sort in ascending or descending order
  - join - return an RDD containing all pairs with matching keys in two RDDs

Example: Pair RDD Operations



Example: Joining by Key

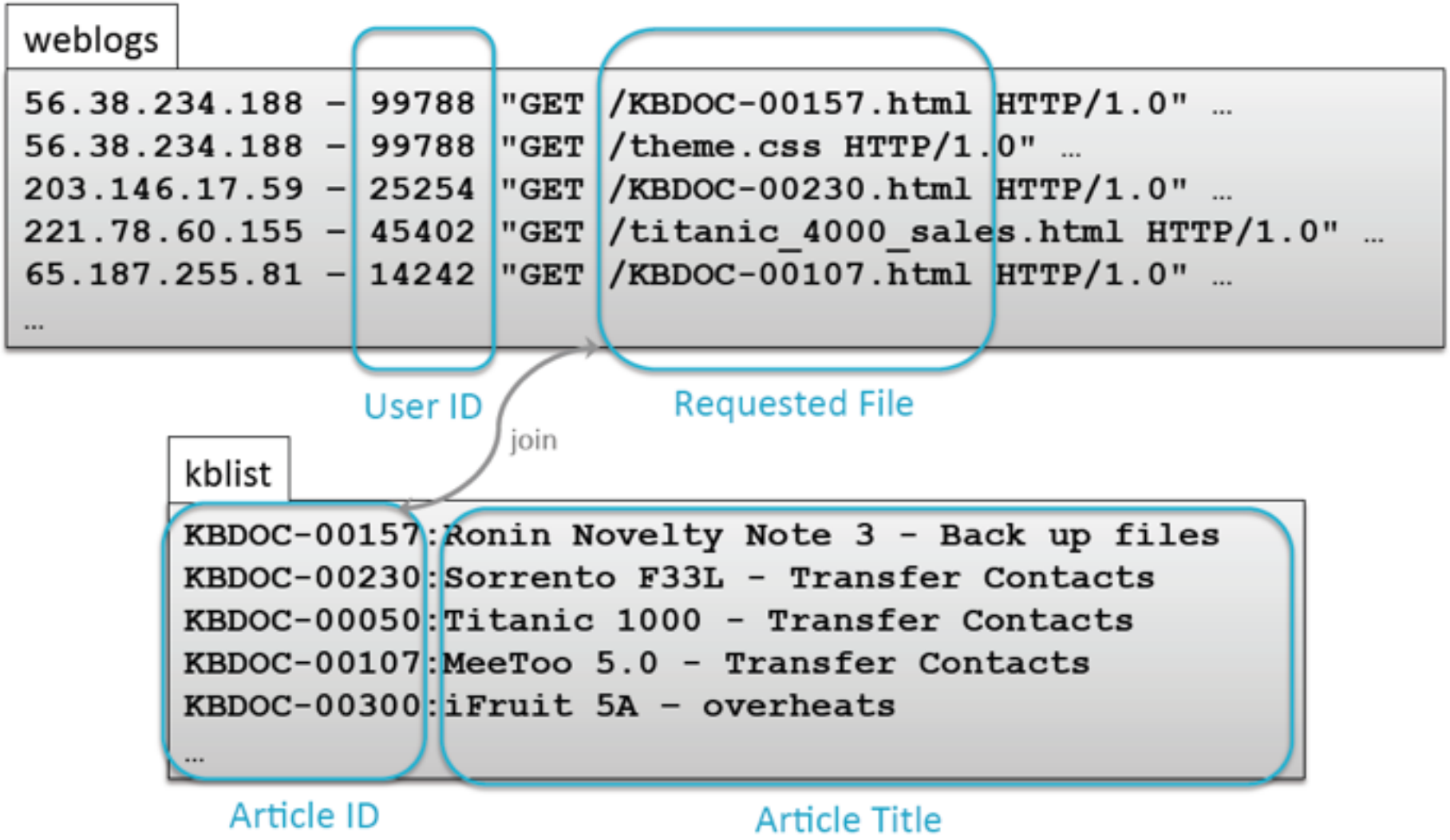
```
> movies = moviegross.join(movieyear)
```



Using Join

- **A common programming pattern**
  - **1. Map separate datasets into key-value Pair RDDs**
  - **2. Join by key**
  - **3. Map joined data into the desired format**
  - **4. Save, display, or continue processing…**

Example: Join Web Log With Knowledge Base Articles (1)



Example: Join Web Log With Knowledge Base Articles (2)

- **Steps**
  - **1. Map separate datasets into key-value Pair RDDs**
    - **a. Map web log requests to (docid,userid)**
    - **b. Map KB Doc index to (docid,title)**
  - **2. Join by key: docid**
  - **3. Map joined data into the desired format: (userid,title)**
  - **4. Further processing: Group titles by userid**

Step1a: Map Web Log Requests to (docid,userid)

```
> import re
> def getRequestDoc(s):
    return re.search(r'KBDOC-[0-9]*',s).group()

> kbreqs = sc.textFile(logfile) \
    .filter(lambda line: 'KBDOC-' in line) \
    .map(lambda line: (getRequestDoc(line),line.split(' ')[2])) \
    .distinct()
```

```
56.38.234.188 - 99788 "GET /KBDOC-00157.html HTTP/1.0" …
56.38.234.188 - 99788 "GET /theme.css HTTP/1.0" …
203.146.17.59 - 25254 "GET /KBDOC-00230.html HTTP/1.0"
221.78.60.155 - 45402 "GET /titanic_4000_sales.html
65.187.255.81 - 14242 "GET /KBDOC-00107.html HTTP/1.
…
```

kbreqs

(KBDOC-00157,99788)
(KBDOC-00203,25254)
(KBDOC-00107,14242)
…

Step1b: Map KB Index to (docid,userid)

```
> kblist = sc.textFile(kblistfile) \
   .map(lambda line: line.split(':')) \
   .map(lambda fields: (fields[0],fields[1]))
```

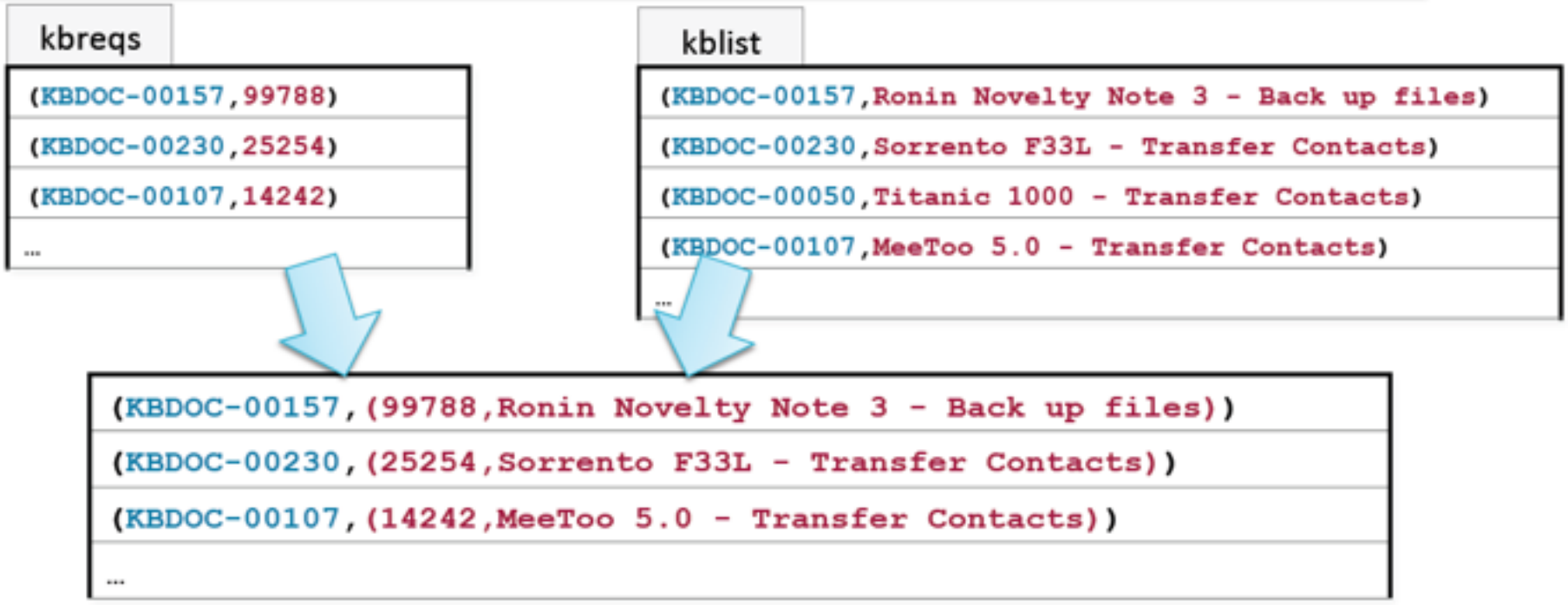
```
KBDOC-00157:Ronin Novelty Note 3 - Back up files
KBDOC-00230:Sorrento F33L - Transfer Contacts
KBDOC-00050:Titanic 1000 - Transfer Contacts
KBDOC-00107:MeeToo 5.0 - Transfer Contacts
KBDOC-00206:iFruit 5A – overheats
…
```

kblist

(KBDOC-00157,Ronin Novelty Note 3 - Back up files)
(KBDOC-00230,Sorrento F33L - Transfer Contacts)
(KBDOC-00050,Titanic 1000 - Transfer Contacts)
(KBDOC-00107,MeeToo 5.0 - Transfer Contacts)
…

Step 2: Join By Key docid

```
> titlereqs = kbreqs.join(kblist)
```



Step 3: Map Result to Desired Format (userid,title)

```
> titlereqs = kbreqs.join(kblist) \
    .map(lambda (docid,(userid,title)): (userid,title))
```

(KBDOC-00157,(99788,Ronin Novelty Note 3 - Back up files))
(KBDOC-00230,(25254,Sorrento F33L - Transfer Contacts))
(KBDOC-00107,(14242,MeeToo 5.0 - Transfer Contacts))
…

(99788,Ronin Novelty Note 3 - Back up files)
(25254,Sorrento F33L - Transfer Contacts)
(14242,MeeToo 5.0 - Transfer Contacts)
…

Step 4: Continue Processing- Group by User ID

```
> titlereqs = kbreqs.join(kblist) \
    .map(lambda (docid,(userid,title)): (userid,title)) \
    .groupByKey()
```

(99788,Ronin Novelty Note 3 - Back up files)
(25254,Sorrento F33L - Transfer Contacts)
(14242,MeeToo 5.0 - Transfer Contacts)
…

(99788,[Ronin Novelty Note 3 - Back up files, Ronin S3 - overheating])
(25254,[Sorrento F33L - Transfer Contacts])
(14242,[MeeToo 5.0 - Transfer Contacts, MeeToo 5.1 - Back up files, iFruit 1 - Back up files, MeeToo 3.1 - Transfer Contacts])
…

Example Output

```
> for (userid,titles) in titlereqs.take(10):
    print 'user id: ',userid
    for title in titles: print '\t',title

user id: 99788
  Ronin Novelty Note 3 - Back up files
  Ronin S3 - overheating
user id: 25254
  Sorrento F33L - Transfer Contacts
user id: 14242
  MeeToo 5.0 - Transfer Contacts
  MeeToo 5.1 - Back up files
  iFruit 1 - Back up files
  MeeToo 3.1 - Transfer Contacts
…
```

```
(99788,[Ronin Novelty Note 3 - Back up files,
        Ronin S3 - overheating])
(25254,[Sorrento F33L - Transfer Contacts])
(14242,[MeeToo 5.0 - Transfer Contacts,
        MeeToo 5.1 - Back up files,
        iFruit 1 - Back up files,
        MeeToo 3.1 - Transfer Contacts])
…
```

Aside: Anonymous Function Parameters

Python

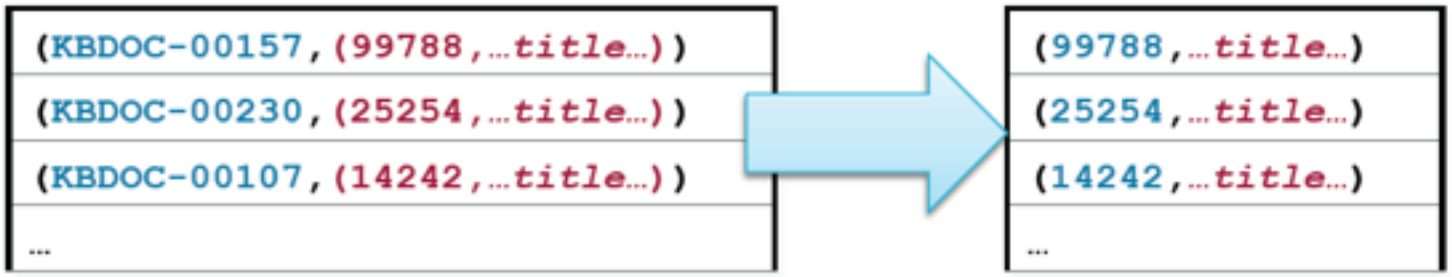
```
> map(lambda (docid,(userid,title)): (userid,title))
```

Scala

```
> map(pair => (pair._2._1,pair._2._2))
```

OR

```
> map{case (docid,(userid,title)) => (userid,title)}
```



Other Pair Operations

- Some other pair operations
  - keys - return an RDD of just the keys, without the values
  - values - return an RDD of just the values, without keys
  - lookup(key)
  - leftOuterJoin, rightOuterJoin - join, including keys defined only in the left or right RDDs respectively
  - mapValues, flatMapValues - execute a function on just the values, keeping the key the same

- See the PairRDDFunctions class Scaladoc for a full list

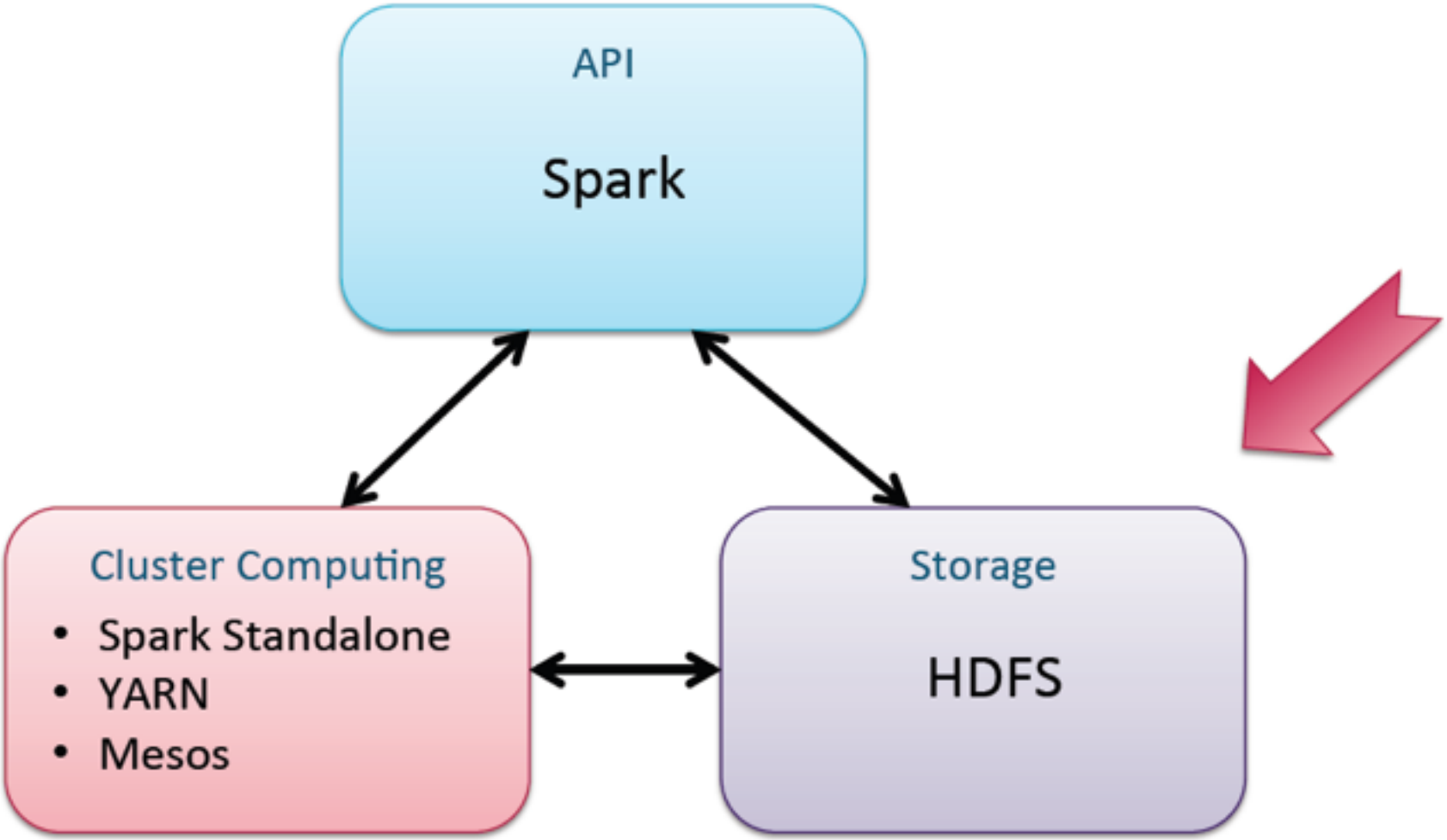
Hands-On Exercise: Working with Pair RDDs

HANDS-ON EXERCISE: WORKING WITH PAIR RDDS

Chapter 4

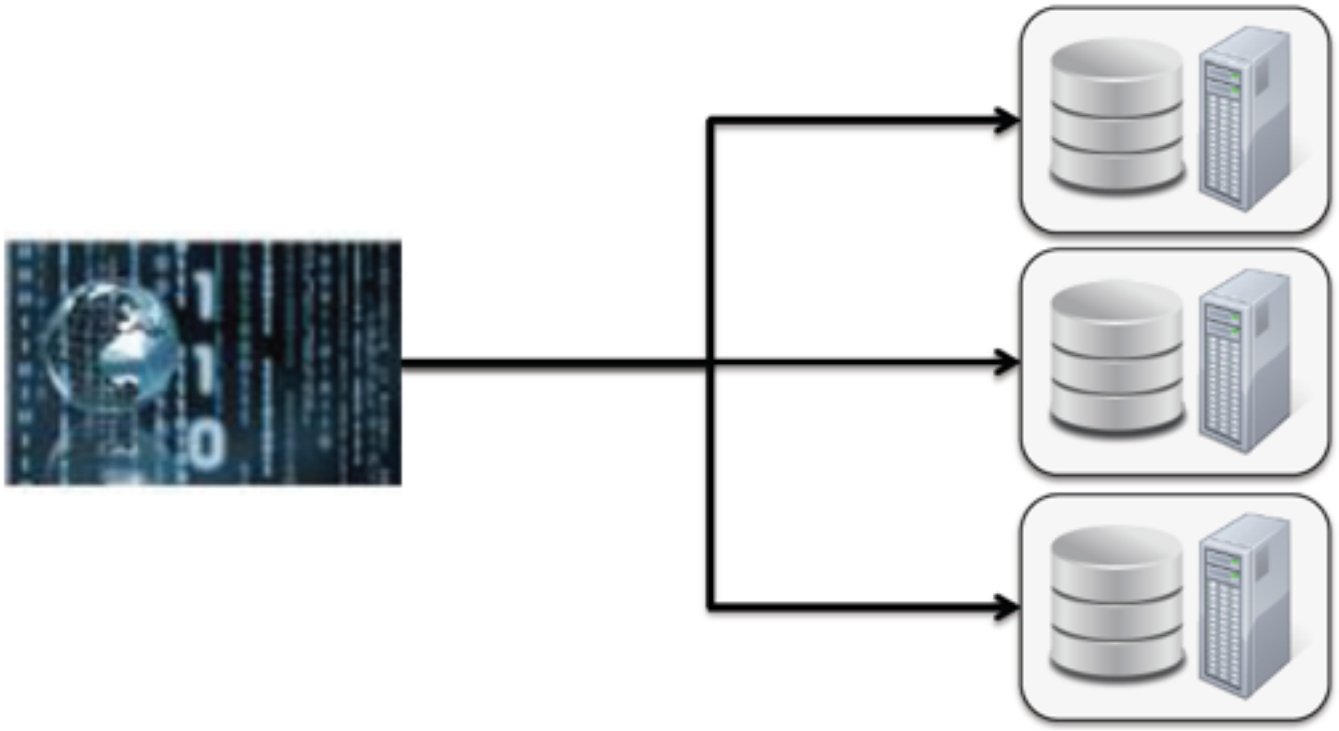
THE HADOOP DISTRIBUTED FILE SYSTEM (HDFS)

Distributed Processing with the Spark Framework



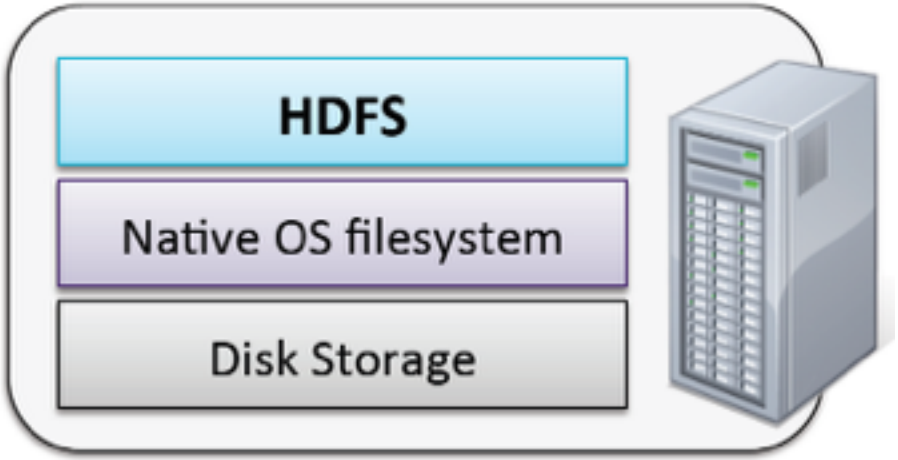
Big Data Processing with Spark

- **Three key concepts**
  - **Distribute data when the data is stored - HDFS**
  - **Run computation where the data is - HDFS and Spark**
  - **Cache data in memory - Spark**



HDFS Basic Concepts

- HDFS is a filesystem written in Java
  - Based on Google's GFS
- Sits on top of a native filesystem
- Provides redundant storage for massive amounts of data
  - Using readily-available, industry-standard computers

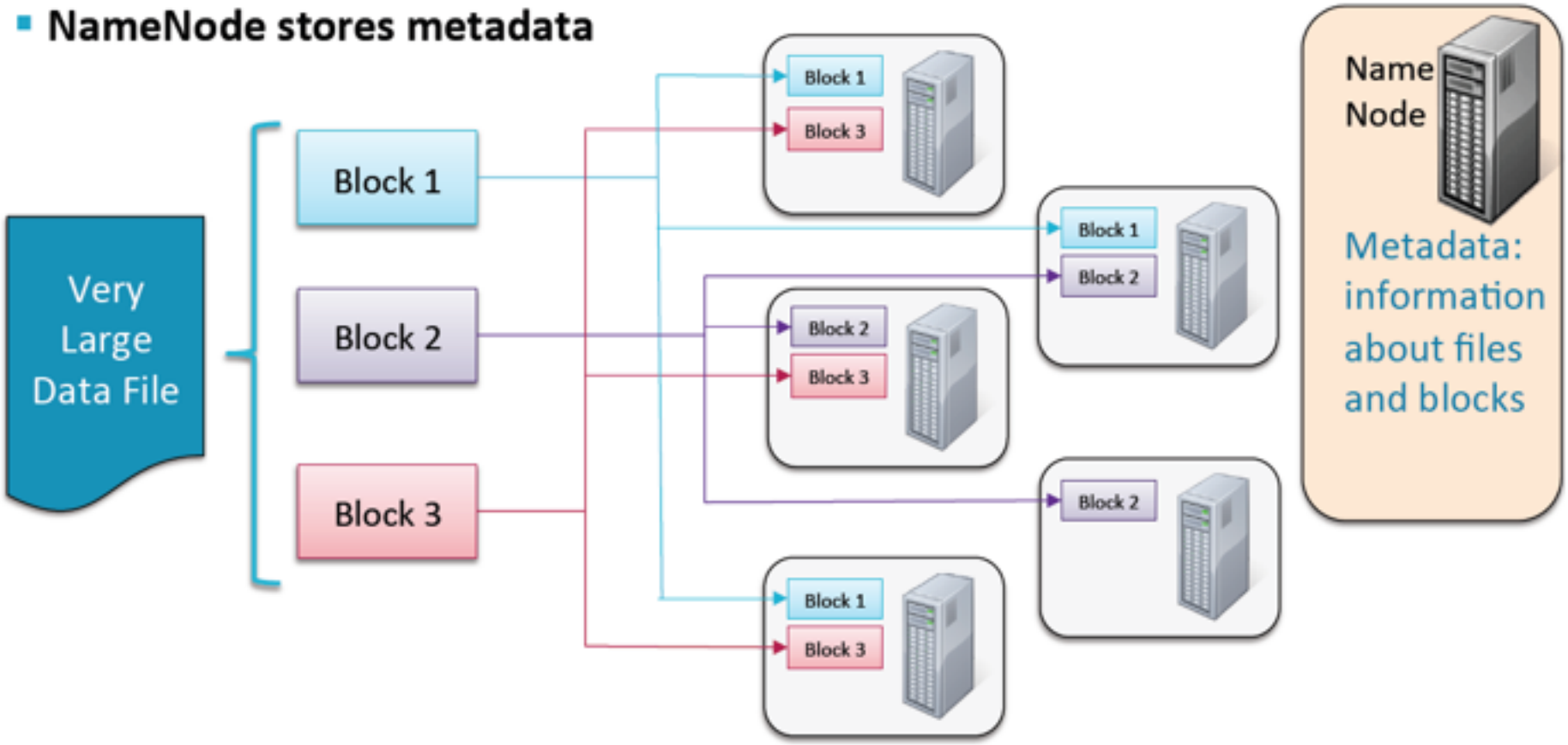


HDFS Basic Concepts

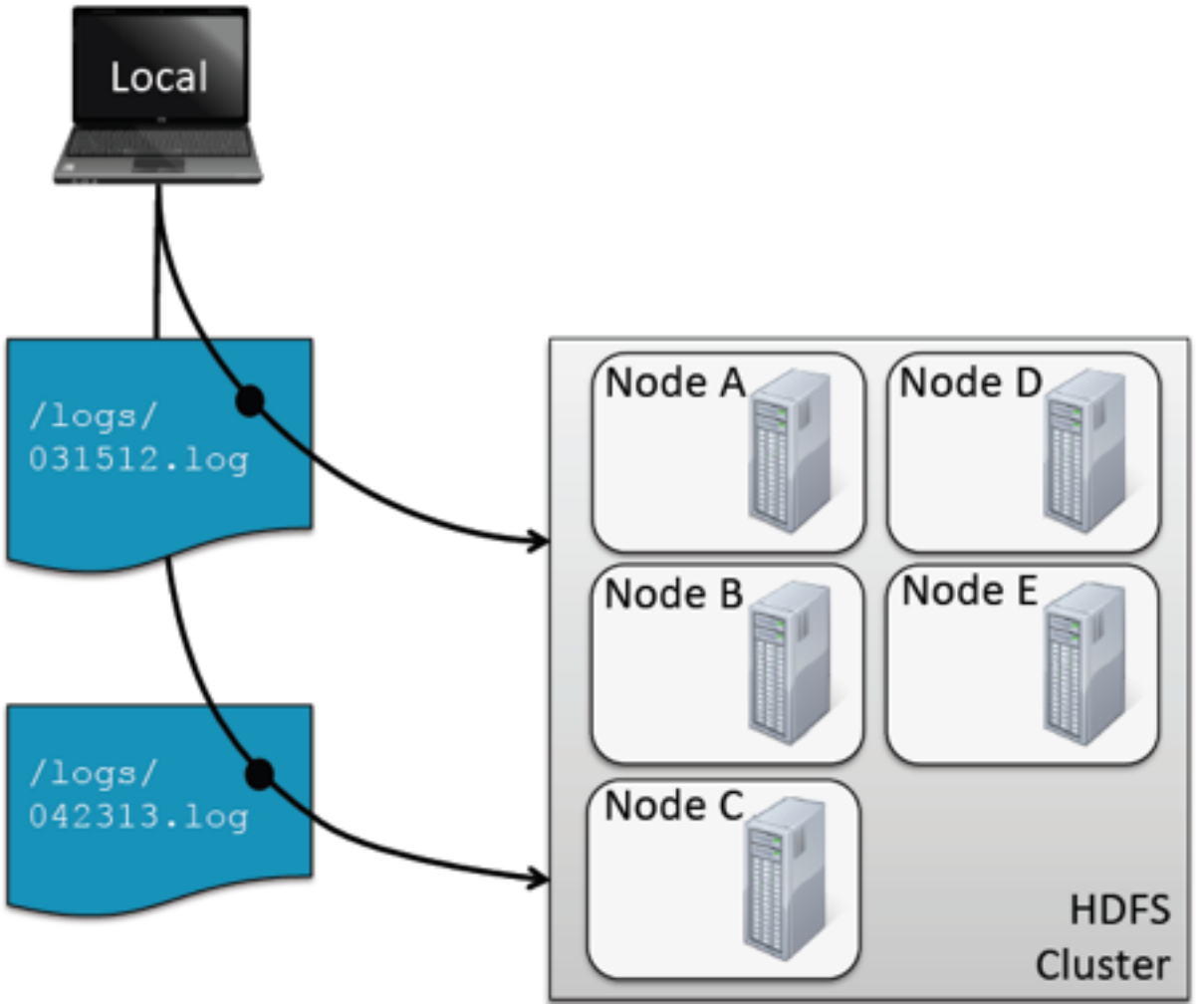
- HDFS performs best with a 'modest' number of large files
  - Millions, rather than billions, of files
  - Each file typically 100MB or more
- Files in HDFS are 'write once'
  - No random writes to files are allowed
- HDFS is optimized for large, streaming reads of files
  - Rather than random reads

How Files Are Stored

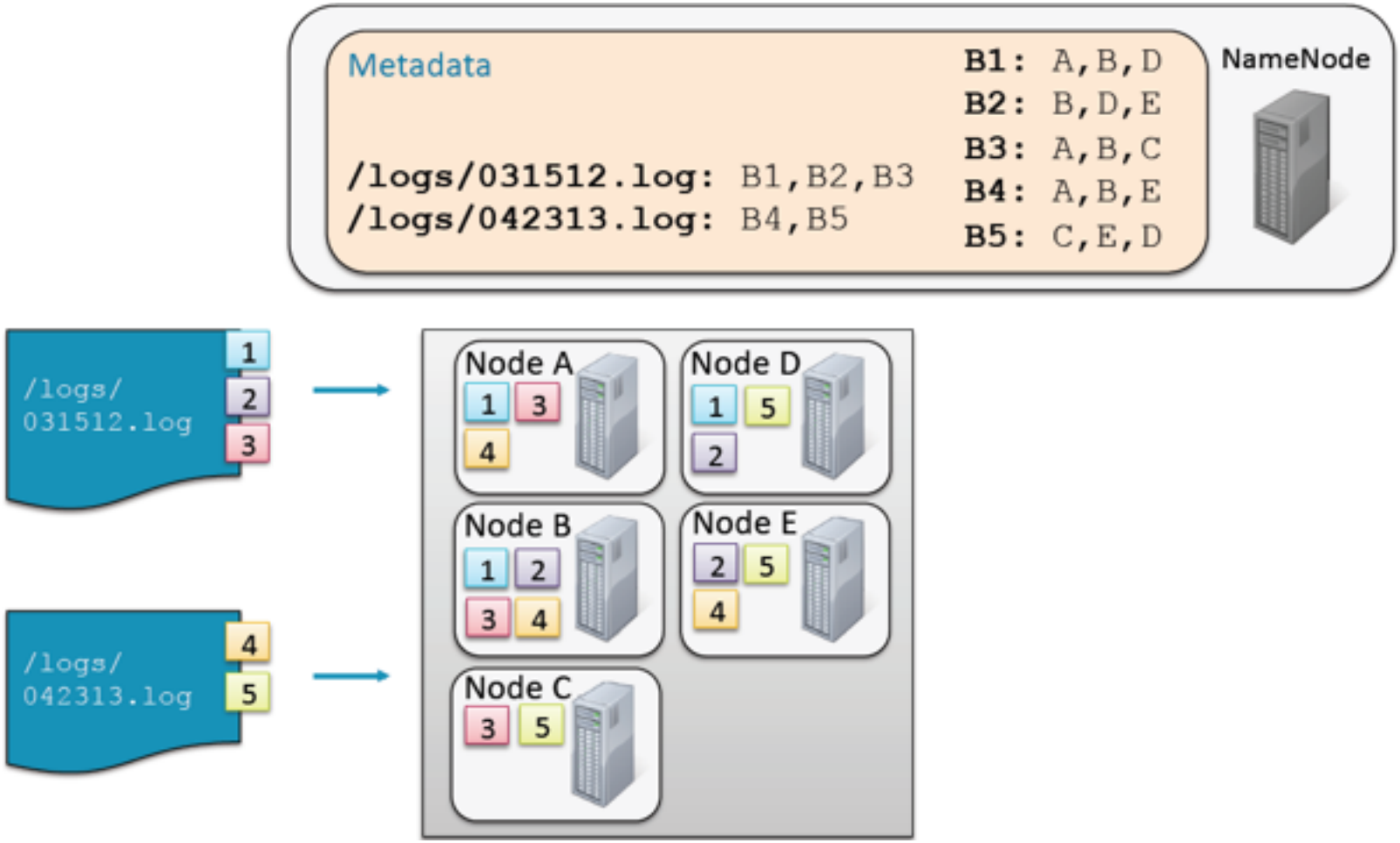
- **NameNode stores metadata**



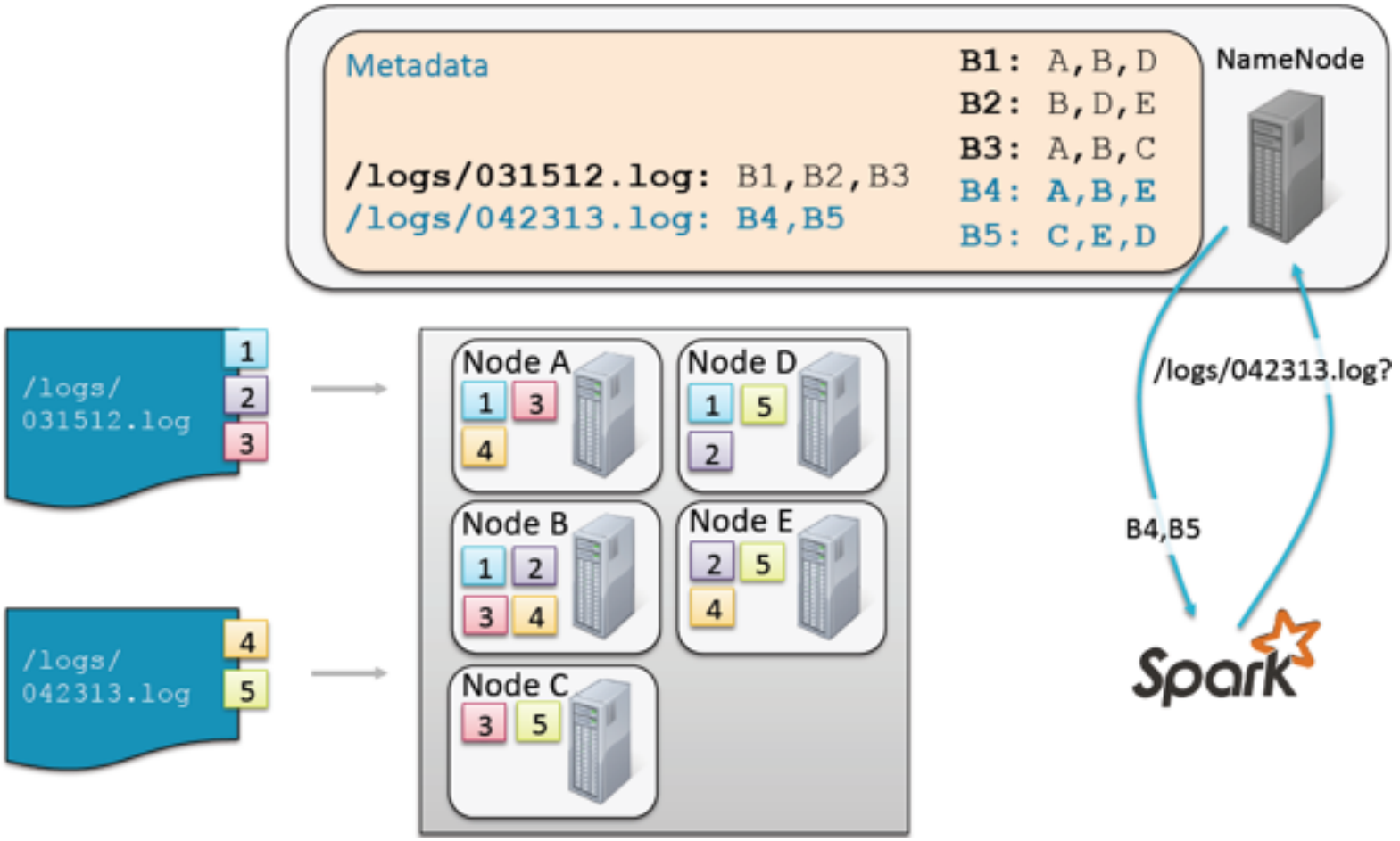
Storing and Retrieving Files



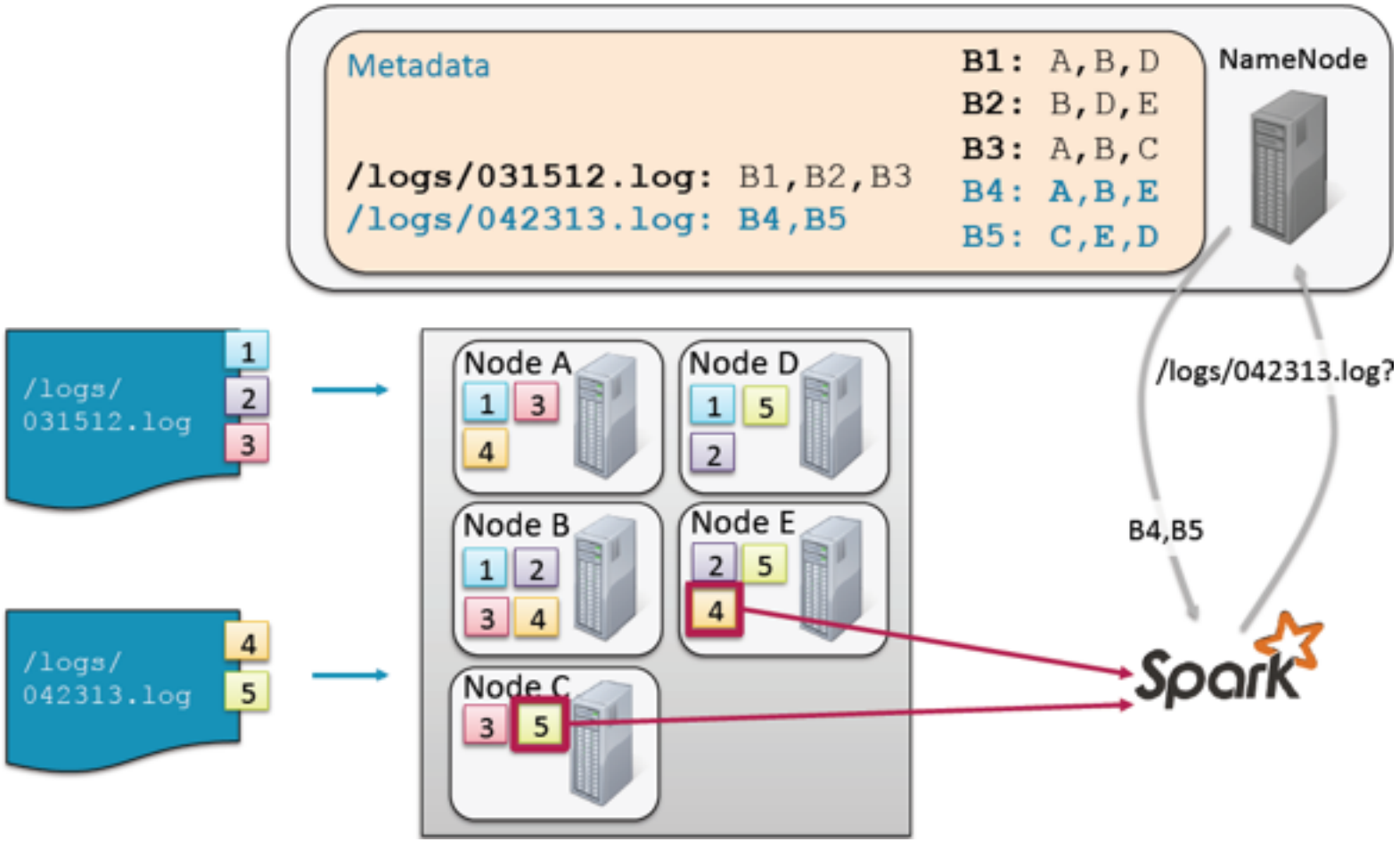
Storing and Retrieving Files



Storing and Retrieving Files



Storing and Retrieving Files



HDFS NameNode Availability

- The NameNode daemon must be running at all times
  - If the NameNode stops, the cluster becomes inaccessible
- HDFS is typically set up for High Availability
  - Two NameNodes: Active and Standby"
- Small clusters may use ‘Classic mode’
  - One NameNode
  - One “helper” node called the Secondary NameNode
  - Bookkeeping, not backup

Options for Accessing HDFS

- From the command line
  - FsShell: hdfs dfs"
- In Spark
  - By URI, e.g. hdfs://host:port/file…

- Other programs
  - Java API
  - Used by Hadoop MapReduce,
  - Hue, Sqoop,
  - Flume, etc.
  - RESTful interface

hdfs dfs Examples

```
$ hdfs dfs -put foo.txt foo.txt
```

- This will copy the file to `/user/username/foo.txt`

- **Get a directory listing of the user's home directory in HDFS**

```
$ hdfs dfs -ls
```

- **Get a directory listing of the HDFS root directory**

```
$ hdfs dfs -ls /
```

hdfs dfs Examples

```
$ hdfs dfs -cat /user/fred/bar.txt
```

- **Copy that file to the local disk, named as `baz.txt`**

```
$ hdfs dfs -get /user/fred/bar.txt baz.txt
```

- **Create a directory called `input` under the user's home directory**

```
$ hdfs dfs -mkdir input
```

Note: copyFromLocal is a synonym for put; copyToLocal is a synonym for get

Example: HDFS in Spark

- Specify HDFS files in Spark by URI
  - hdfs://hdfs-host[:port]/path
  - Default port is 8020

```
> mydata = sc.textFile \
  ("hdfs://hdfs-host:port/user/training/purplecow.txt")


> mydata.map(lambda s: s.upper()).\
  saveAsTextFile \
  ("hdfs://hdfs-host:port/user/training/purplecowuc")
```

Using HDFS By Default

- If Hadoop configuraraon files are on Spark's classpath, Spark will use HDFS by default
  - e.g. /etc/hadoop/conf
- Paths are relative to user's home HDFS directory

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

hdfs://*hdfs-host:port*/user/training/purplecow.txt

Hands On Exercise: Using HDFS

HANDS-ON EXERCISE: USING HDFS

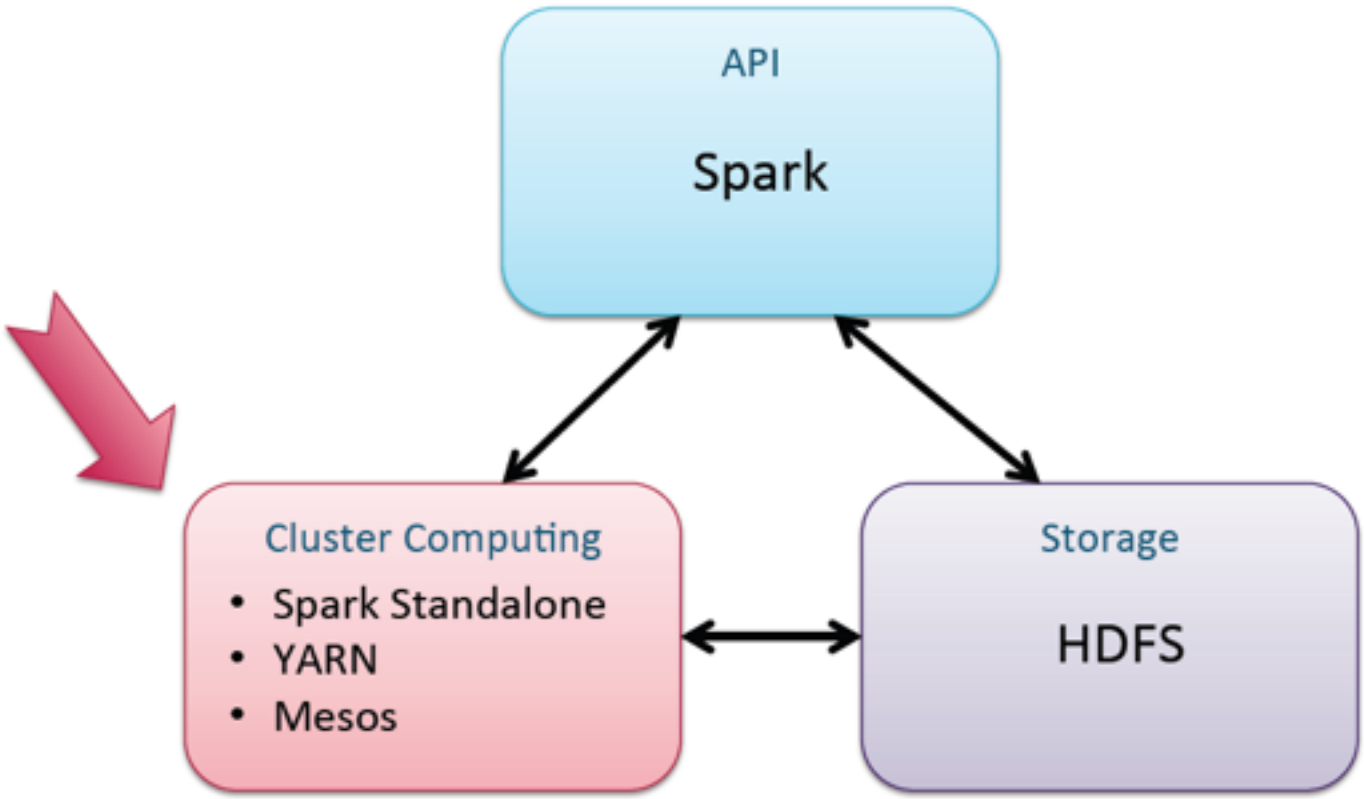
Chapter 5

SPARK ON A CLUSTER

Spark Cluster Options

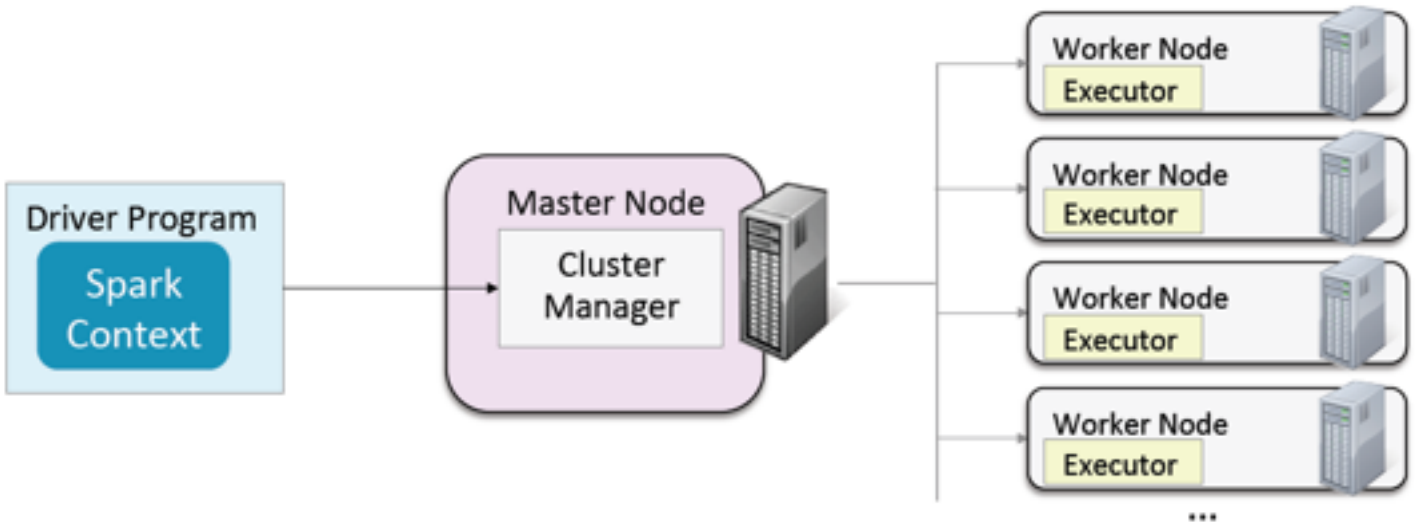
- Spark can run
  - Locally
    - No distributed processing
    - Locally with multiple worker threads
  - On a cluster
    - Spark Standalone
    - Apache Hadoop YARN (Yet Another Resource Negotiator)

Distributed Processing with the Spark Framework



The Spark Driver Program

- A Spark Driver
  - The "main" program
    - Either the Spark Shell or a Spark application
  - Creates a Spark Context configured for the cluster
  - Communicates with Cluster Manager to distribute tasks to executors



Starting the Spark Shell on a Cluster

- Set the Spark Shell master to
  - url - the URL of the cluster manager
  - local[*] - run with as many threads as cores (default)
  - local[n] - run locally with n worker threads
  - local - run locally without distributed processing
- This configures the SparkContext.master property

Python

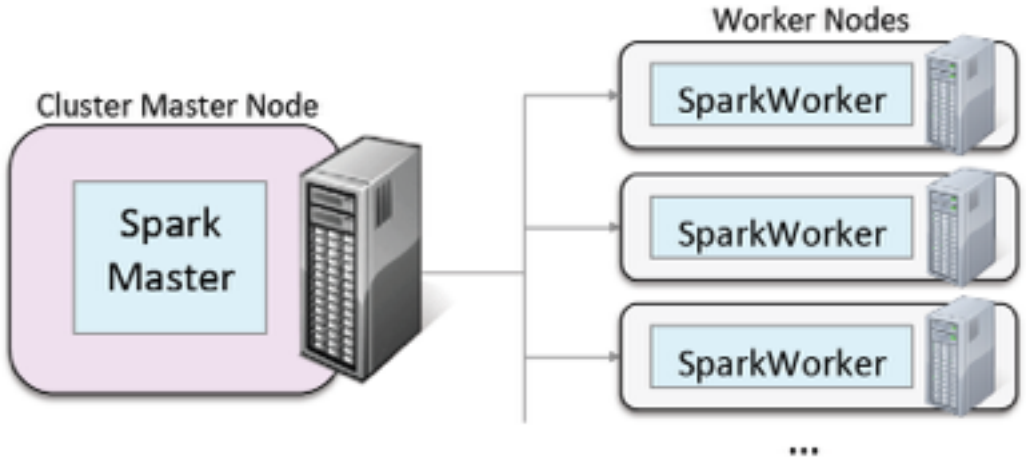
```
$ MASTER=spark://masternode:7077 pyspark
```

Scala

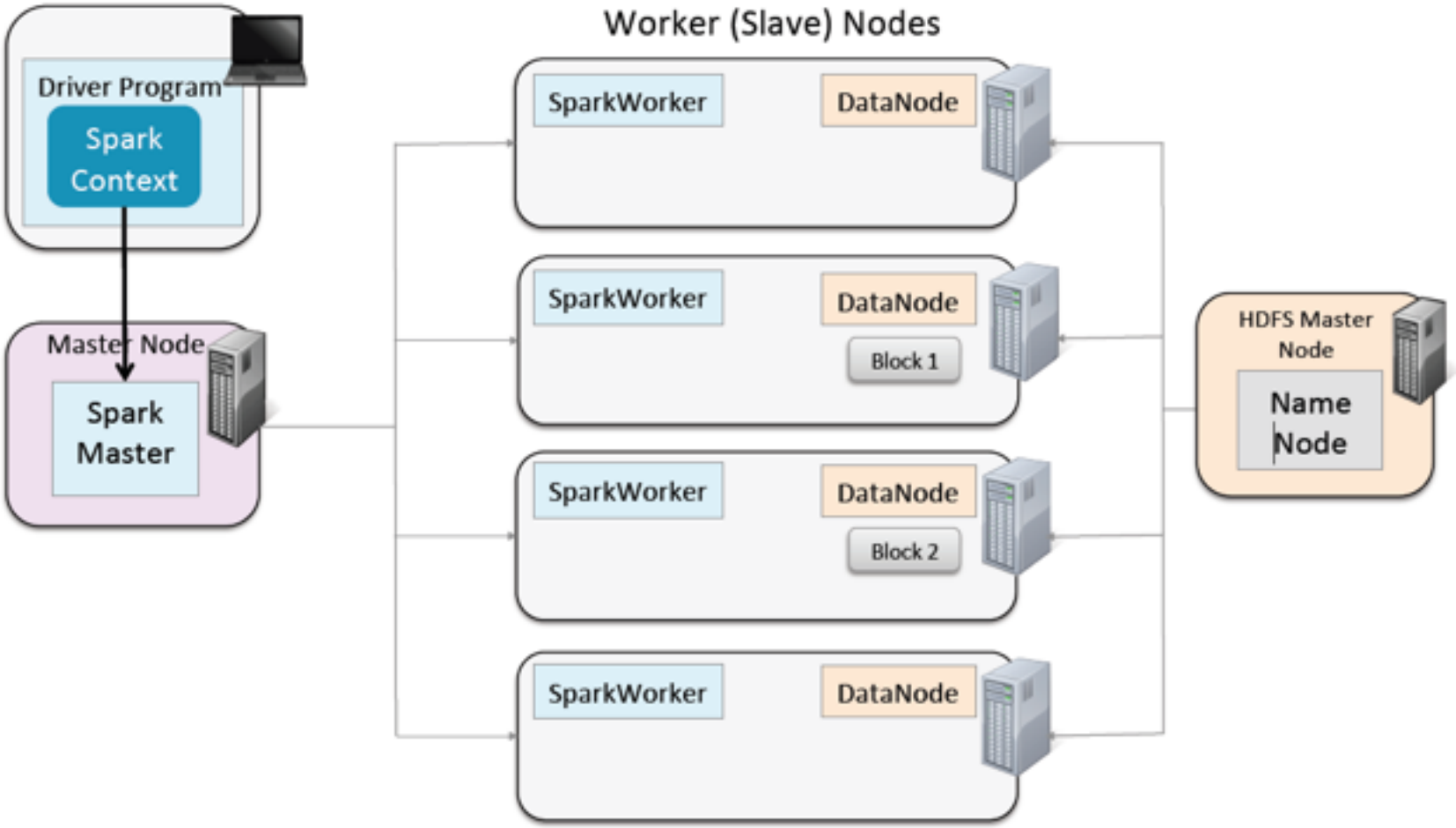
```
$ spark-shell --master spark://masternode:7077
```

Spark Standalone Daemons

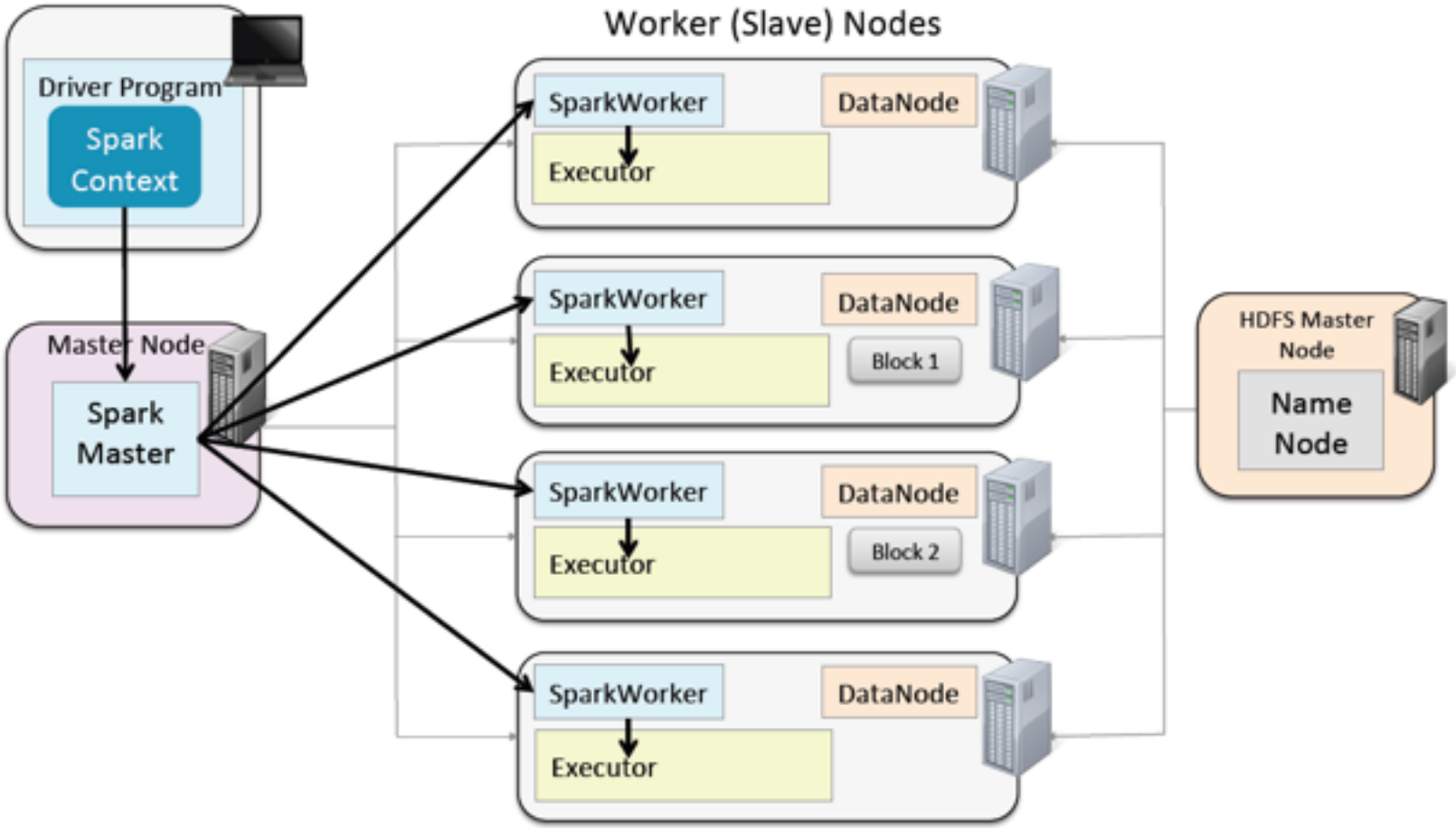
- Spark Standalone daemons
  - Spark Master
    - One per cluster
    - Manages applications, distributes individual tasks to Spark Workers
  - Spark Worker
    - One per worker node
    - Starts and monitors Executors for applications



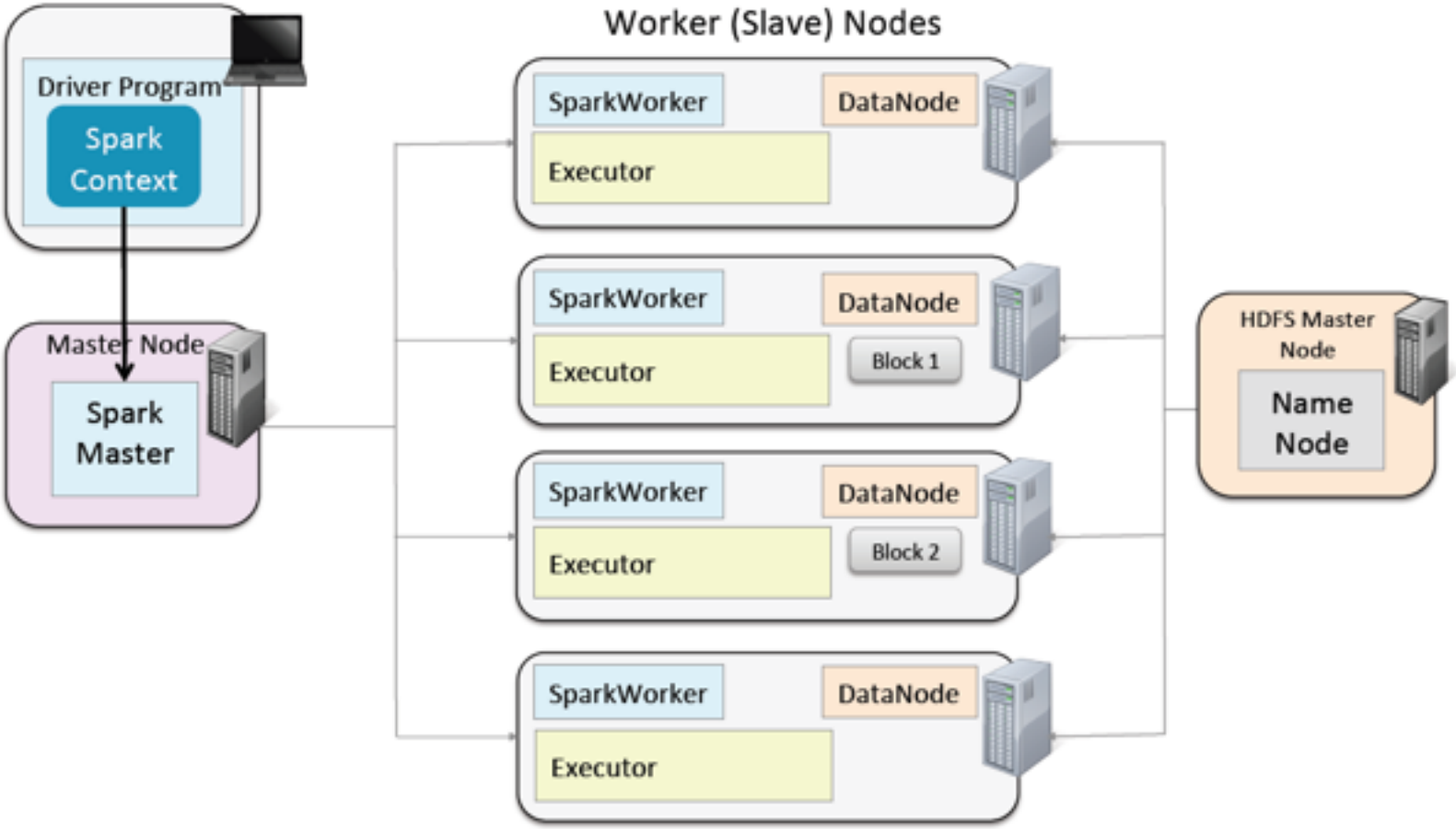
Running Spark on a Standalone Cluster



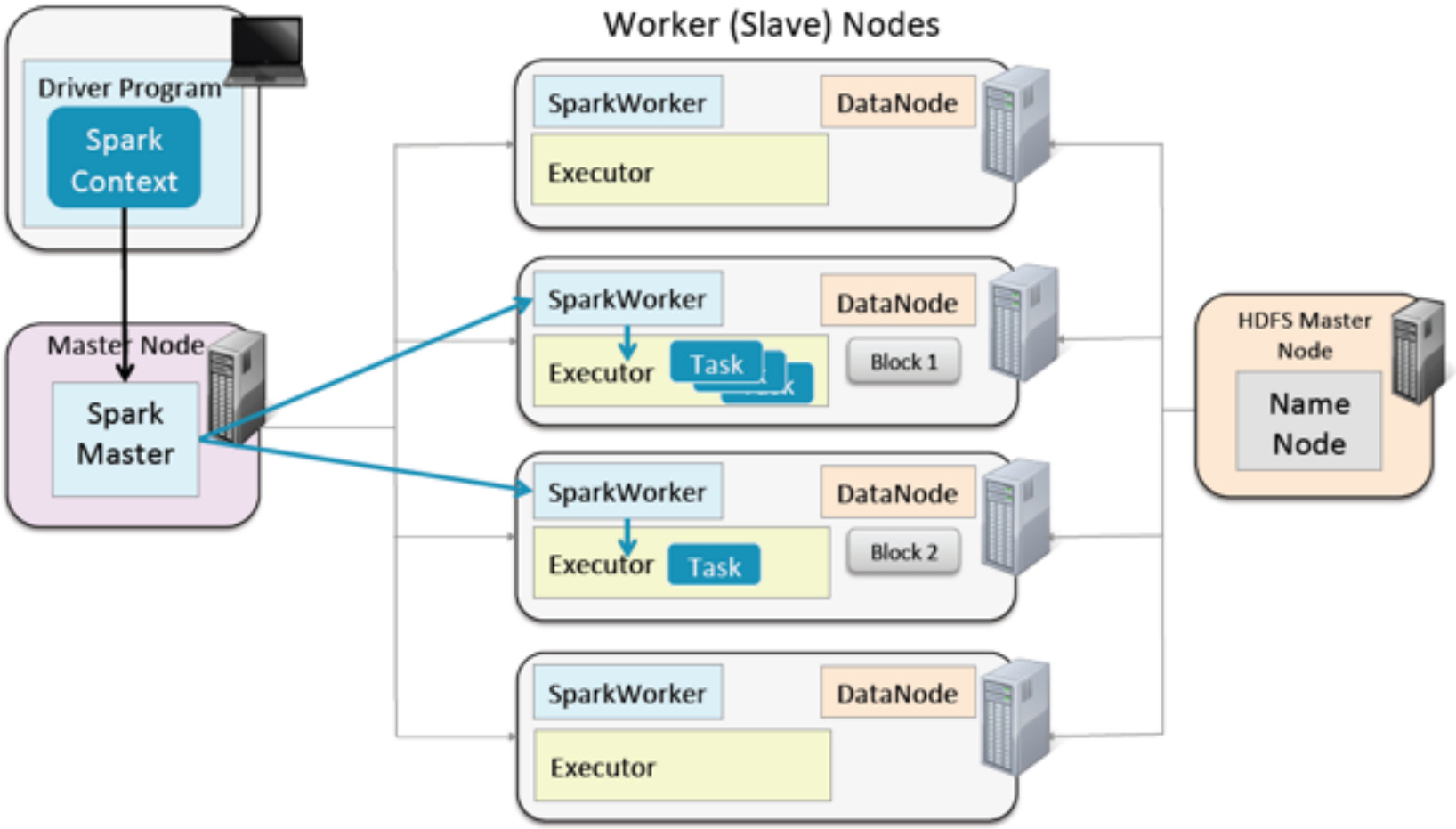
Running Spark on a Standalone Cluster



Running Spark on a Standalone Cluster



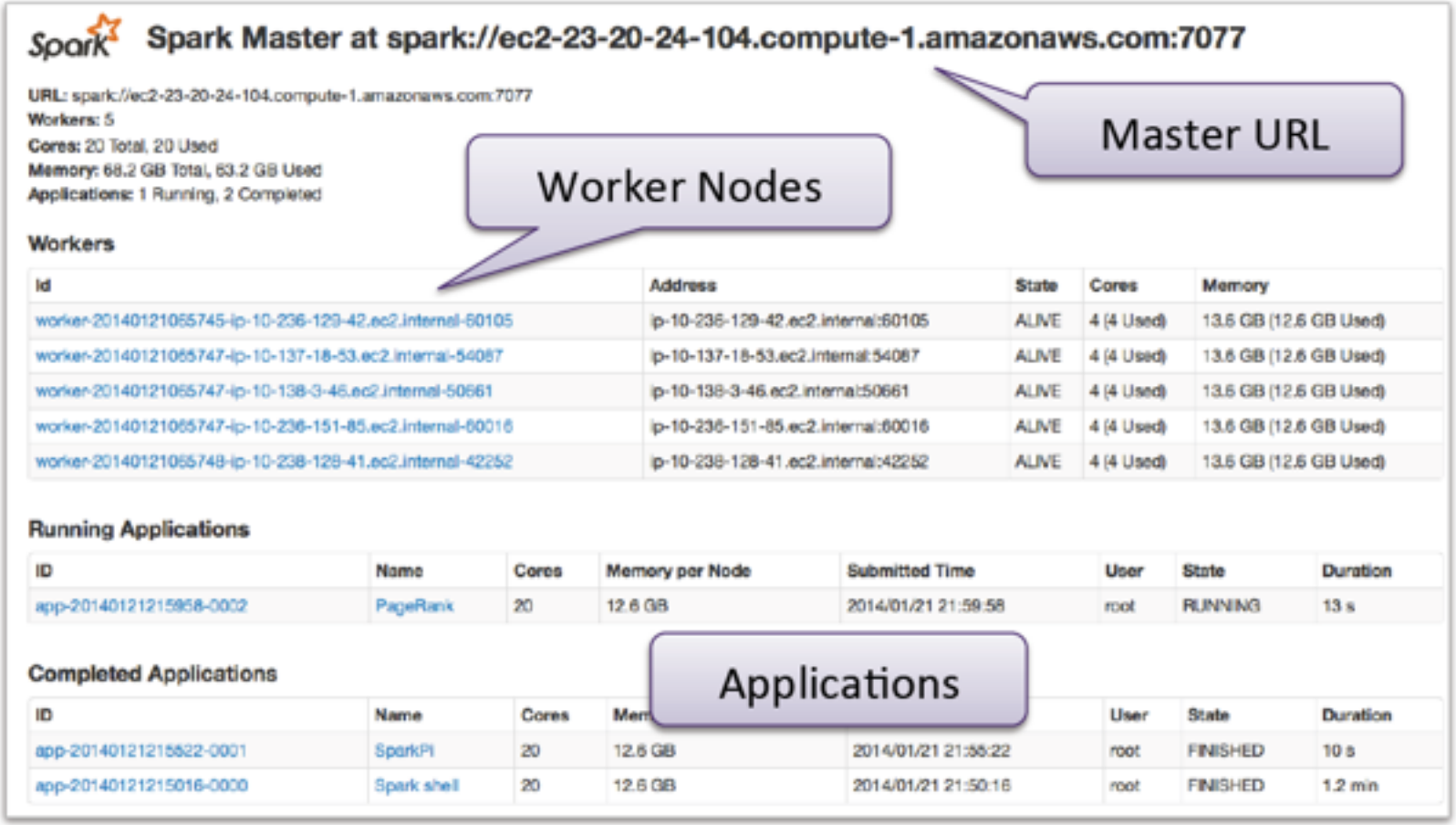
Running Spark on a Standalone Cluster



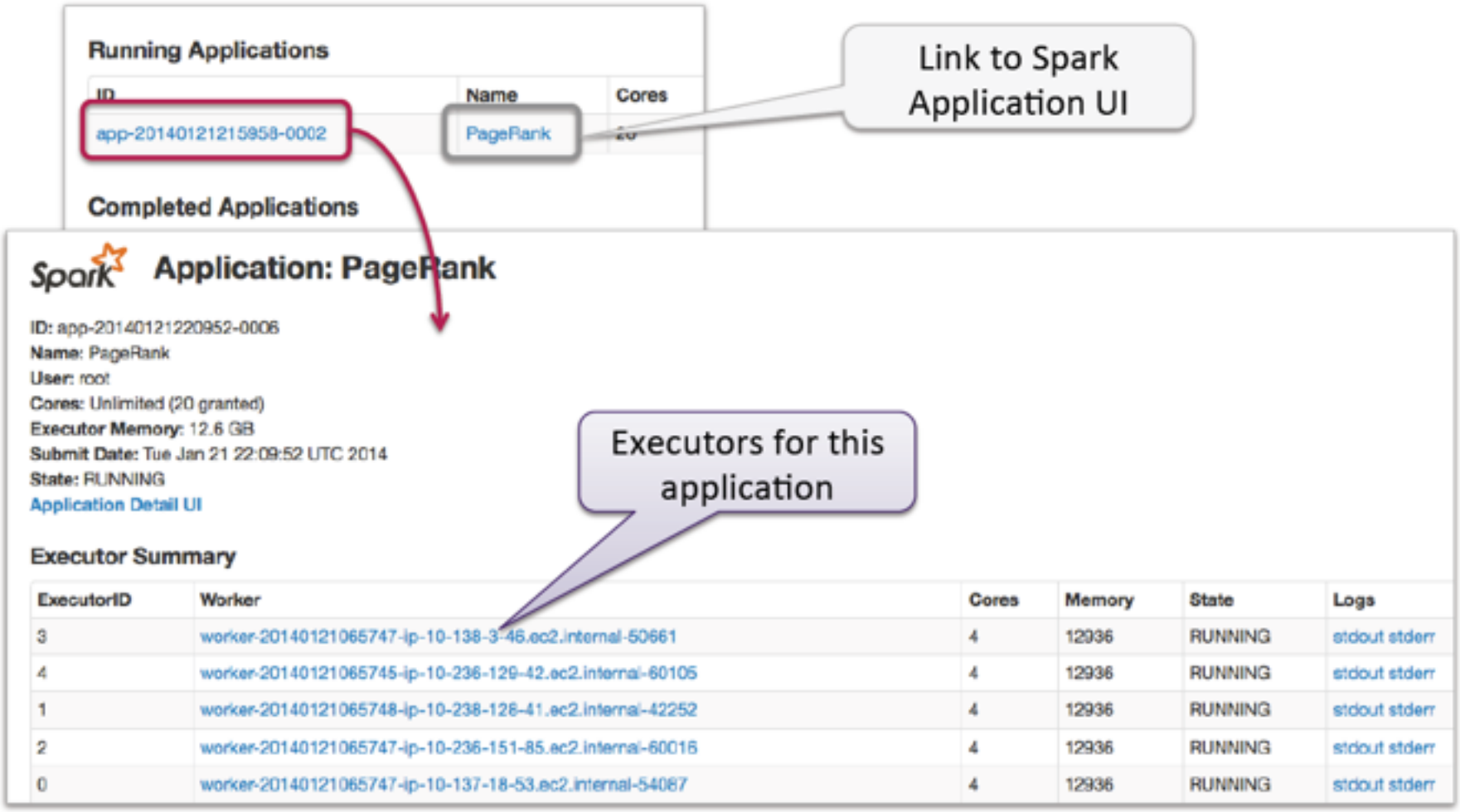
Spark Standalone Web UI

Spark Standalone clusters offer a Web UI to monitor the cluster

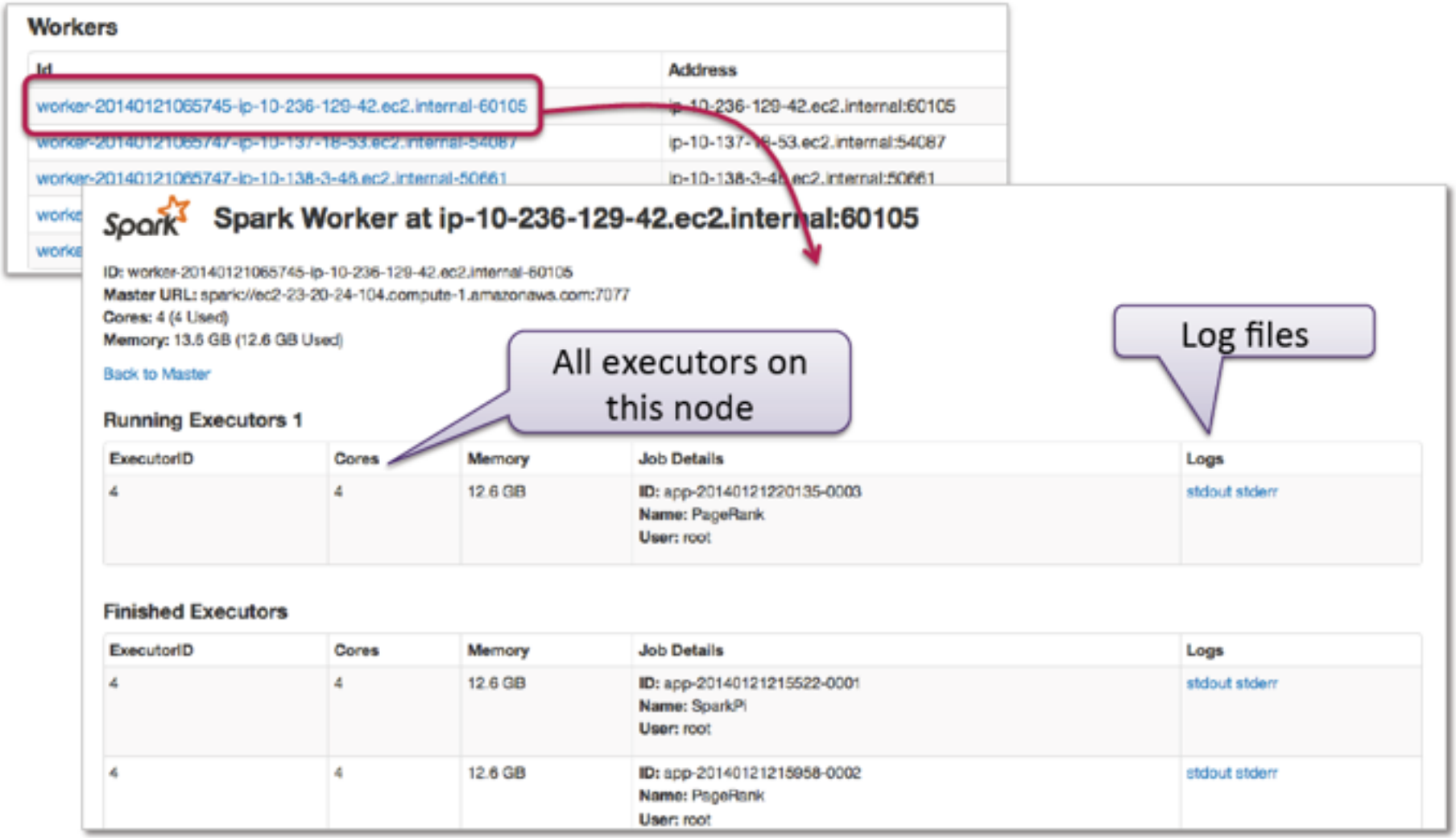
- http://masternode:uiport
- e.g., in our class environment, http://localhost:18080



Spark Standalone Web UI: Application Overview



Spark Standalone Web UI: Worker Detail

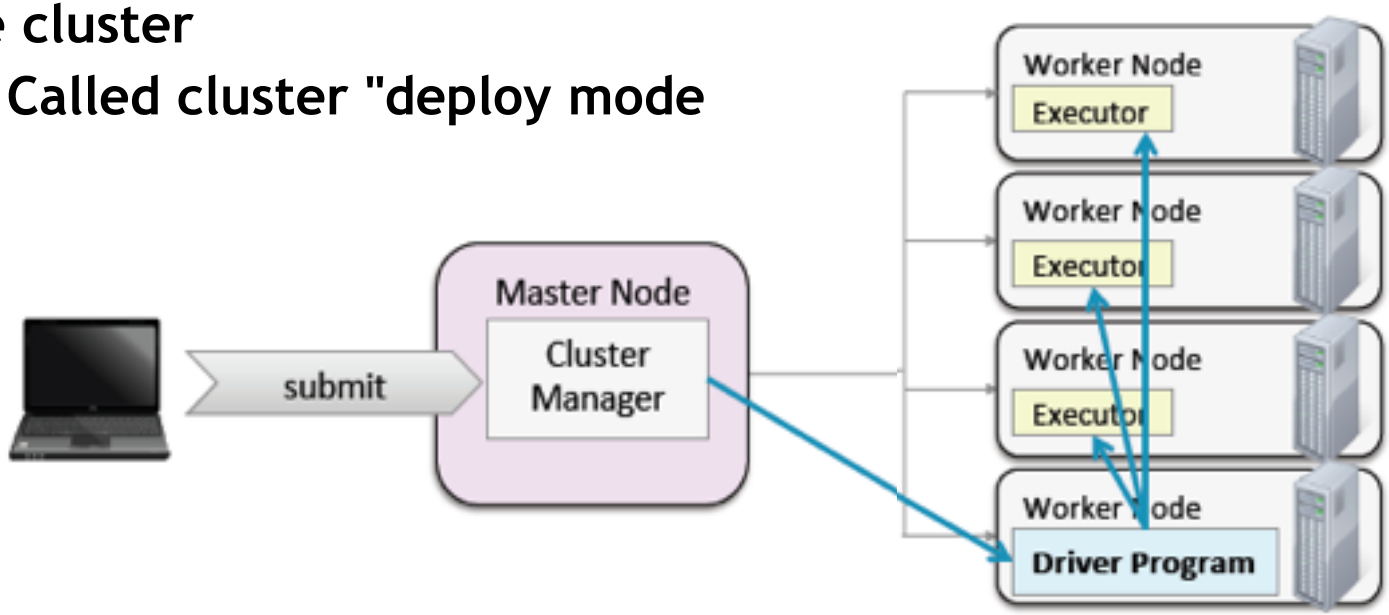


Supported Cluster Resource Managers

- Spark Standalone
  - Included with Spark
  - Easy to install and run
  - Limited configurability and scalability
  - Useful for testing, development, or small systems
- Hadoop YARN
  - Included in HDP
  - Most common for production sites
  - Allows sharing cluster resources with other applications (MapReduce, etc.)

Client Mode and Cluster Mode

- By default, the driver program runs outside the cluster
  - Called “client” deploy mode
  - Most common
  - Required for interactive use (e.g., the Spark Shell)
- It is also possible to run the driver program on a worker node in the cluster
  - Called cluster "deploy mode



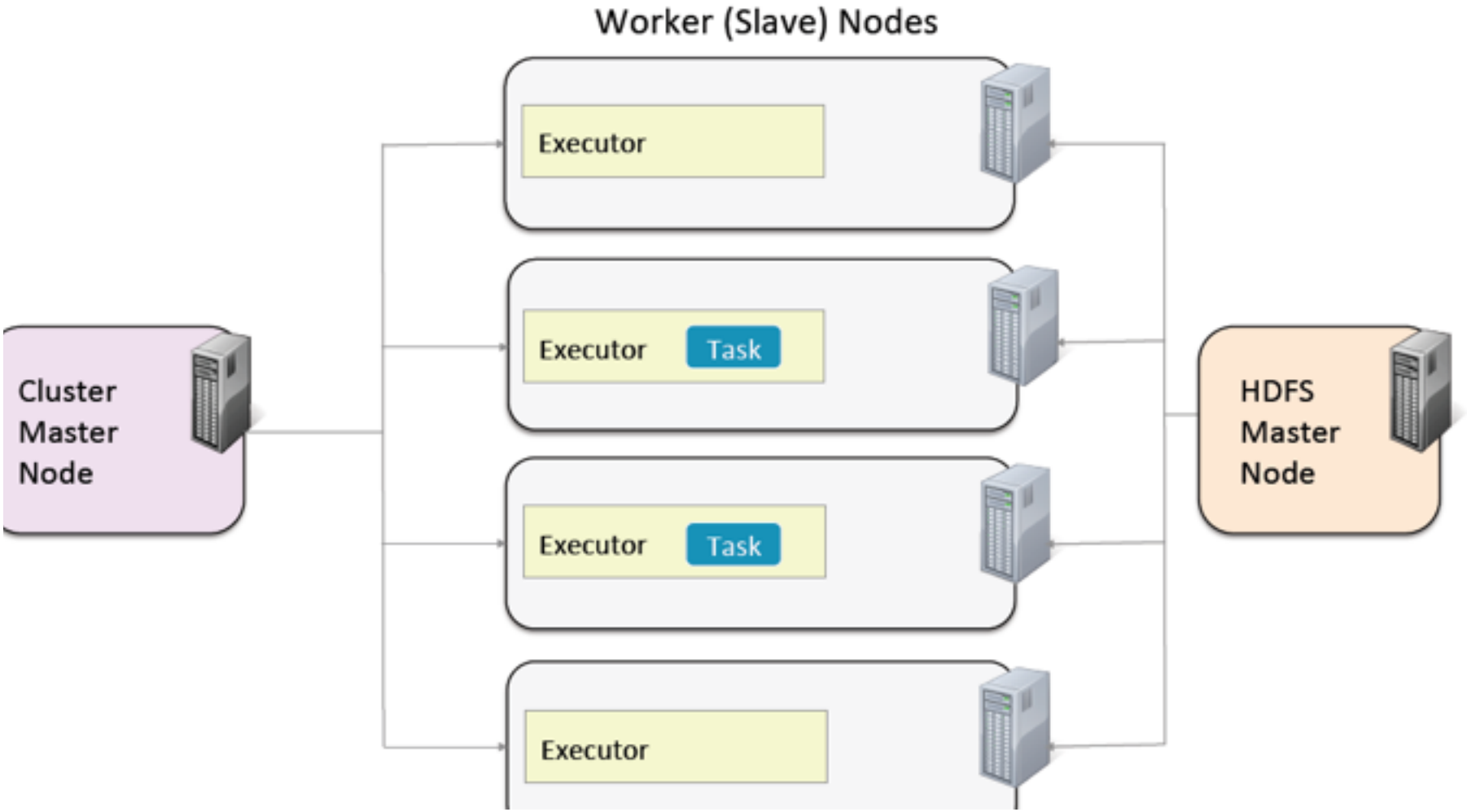
Hands On Exercise: Running Spark on a Cluster

HANDS-ON EXERCISE: RUNNING SPARK ON A CLUSTER

Chapter 6

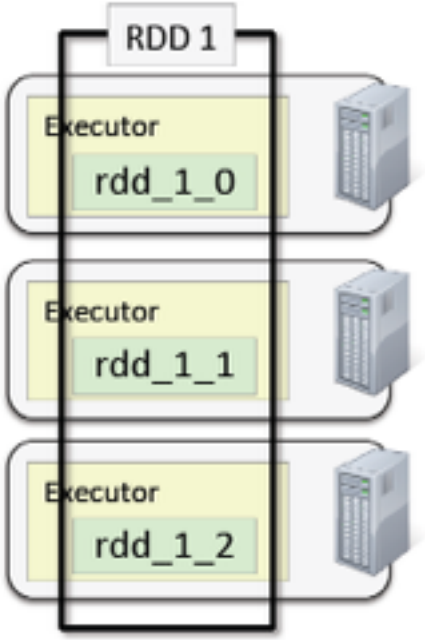
PARALLEL PROGRAMMING WITH SPARK

Spark Cluster Review



RDDs on a Cluster

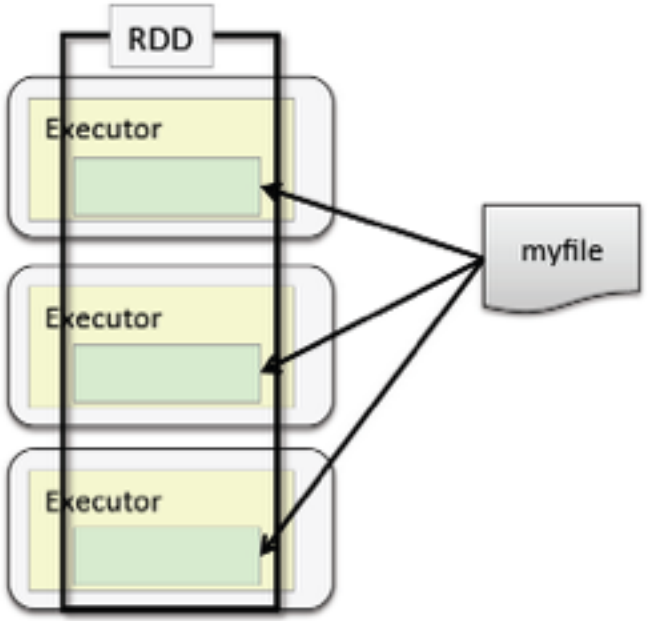
- **Resilient Distributed Datasets**
  - **Data is partitioned across worker nodes**
- **Partitioning is done automatically by Spark**
  - **Optionally, you can control how many partitions are created**



File Partitioning: Single Files

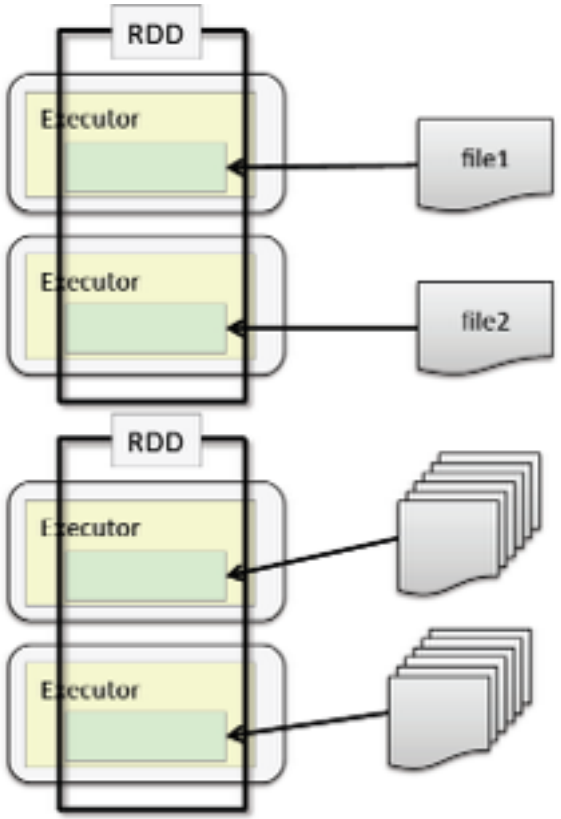
```
sc.textFile("myfile",3)
```

- 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

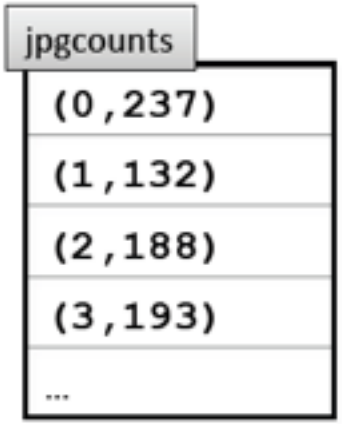
- Most RDD operations work on each element of an RDD
- A few work on each partition
  - foreachPartition - call a function for each partition
  - mapPartitions - create a new RDD by executing a function on each partition in the current RDD
  - mapPartitionsWithIndex - same as mapPartitions but includes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

Example: Count JPGs Requests per File

```
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)
```

```
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)
```

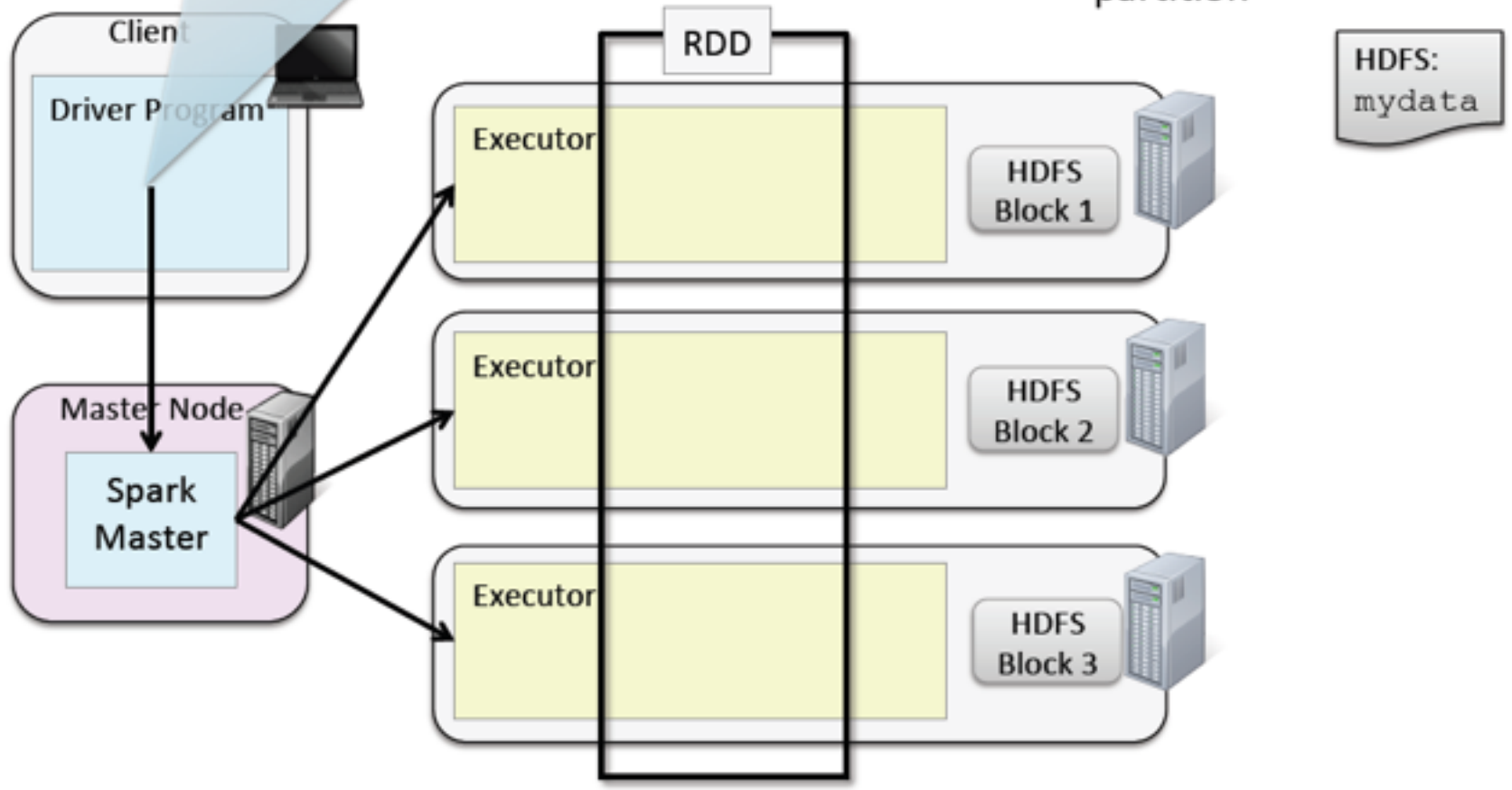
Note: Works with small files that each fit in a single partition



HDFS and Data Locality

```
sc.textFile("hdfs://…mydata…").collect()
```

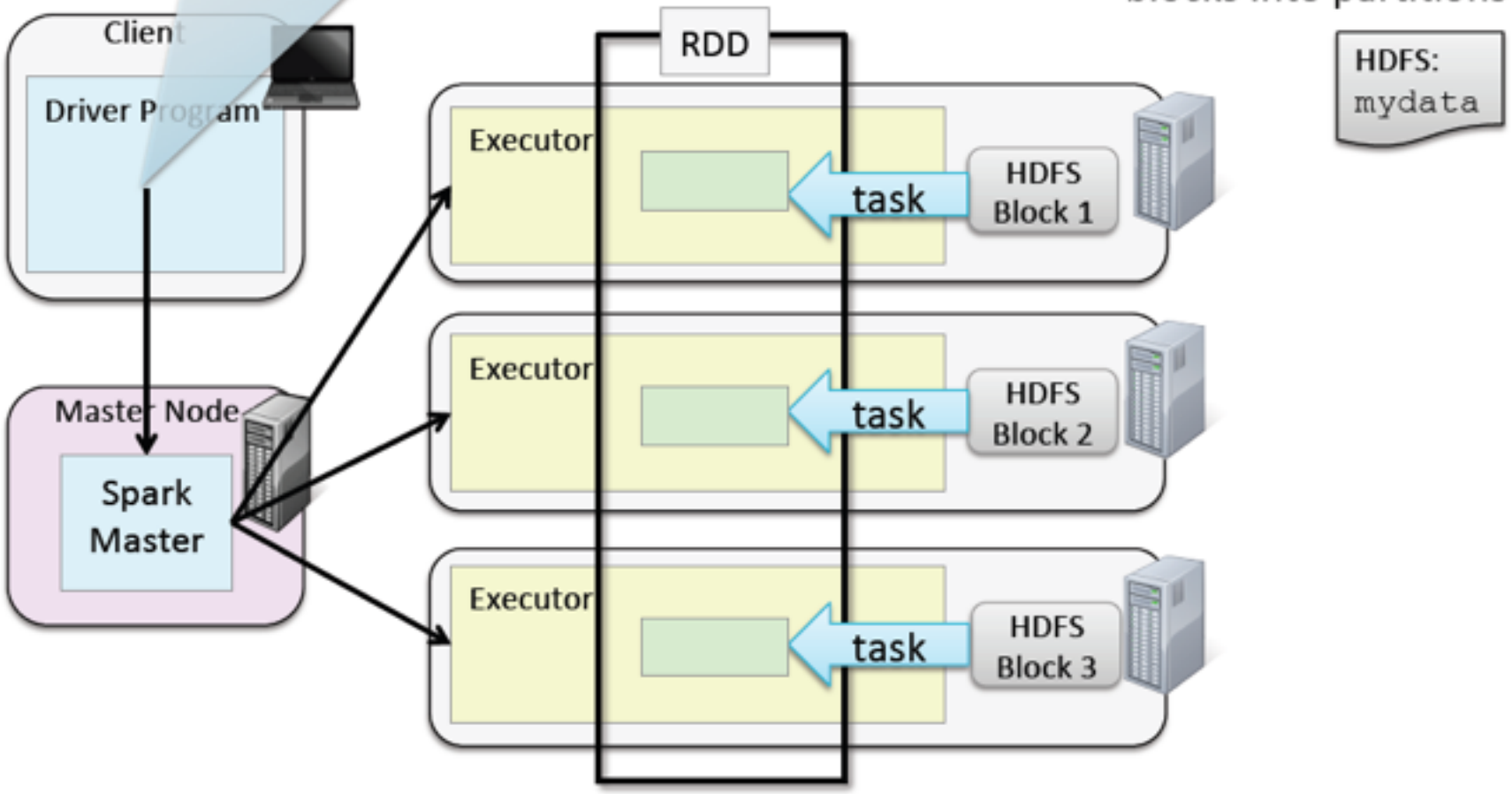
By default, Spark partitions HDFS files by block. Each block loads into a single partition



HDFS and Data Locality

```
sc.textFile("hdfs://…mydata…").collect()
```

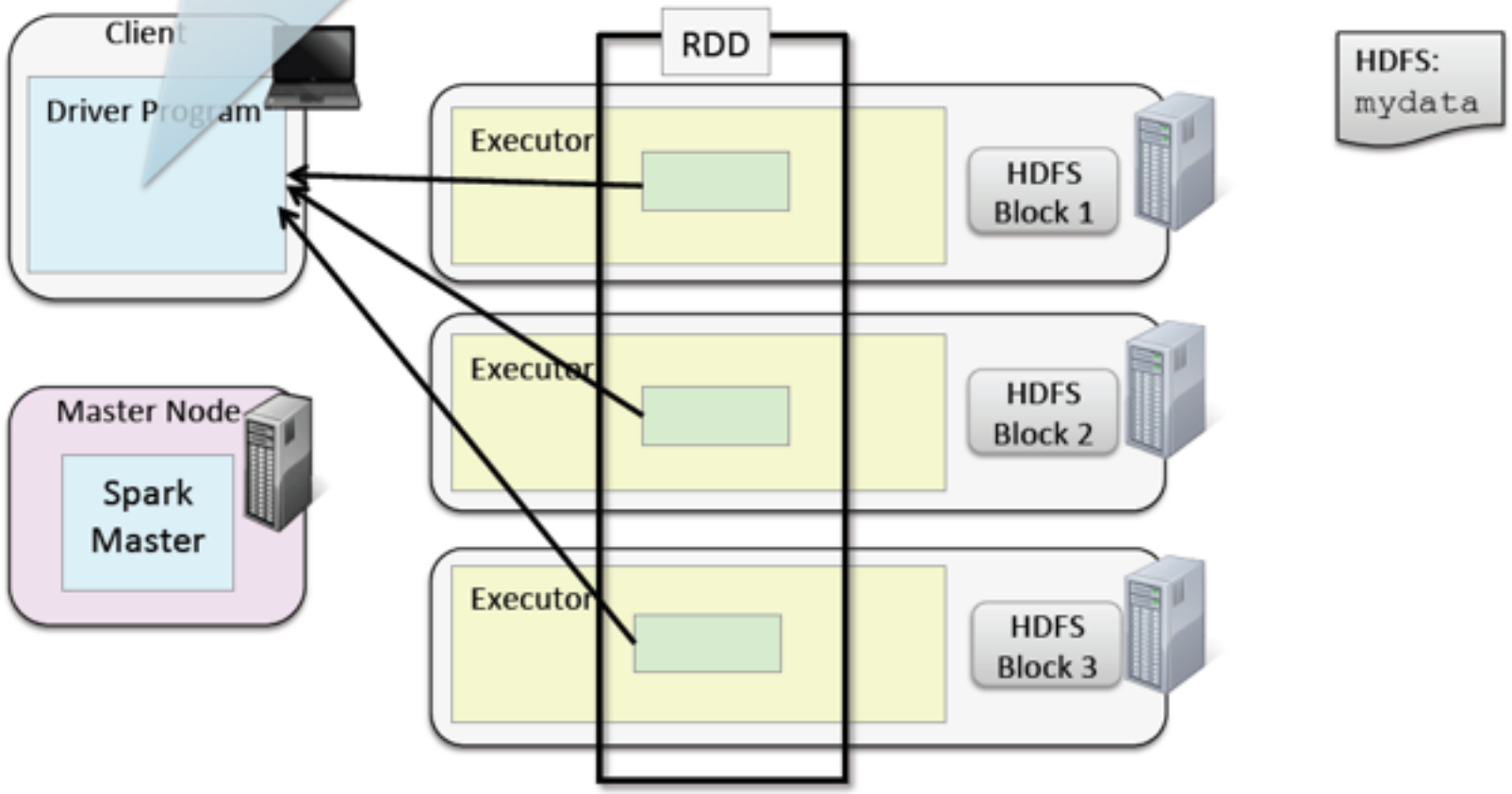
An action triggers execution: tasks on executors load data from blocks into partitions



HDFS and Data Locality

```
sc.textFile("hdfs://…mydata…").collect()
```

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

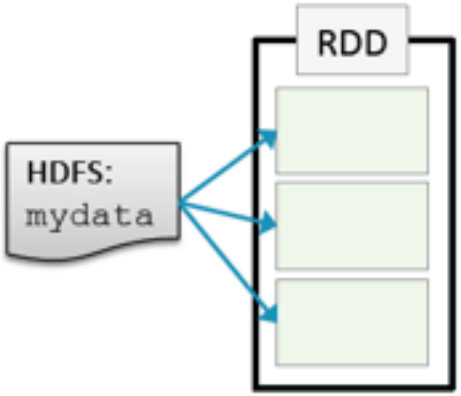
Parallel Operations on Partitions

- 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

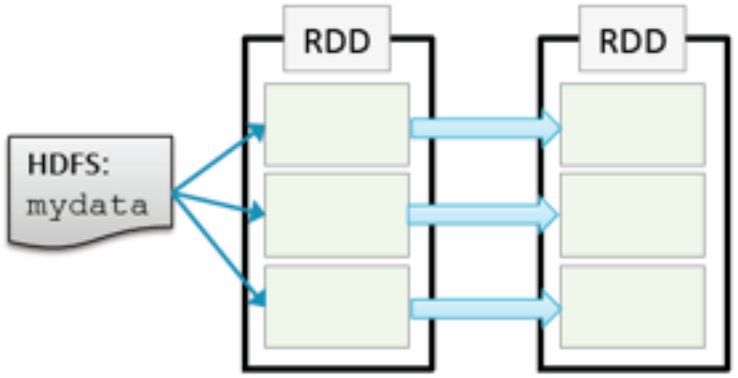
-

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



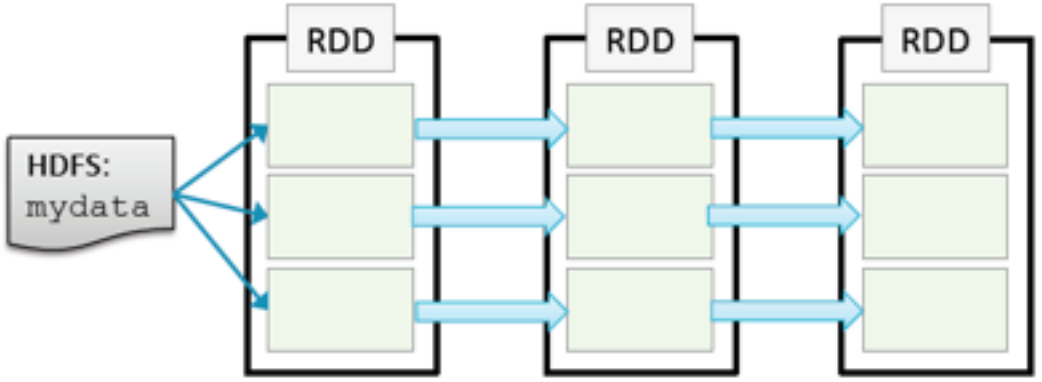
Example: Average Word Length by Letter

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



Example: Average Word Length by Letter

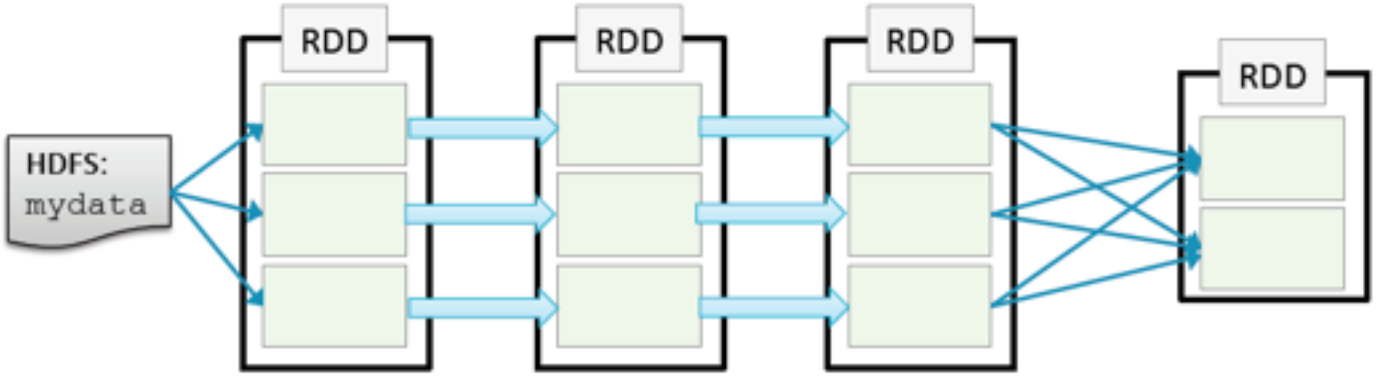
```
> avglens = sc.textFile(file) \
  .flatMap(lambda line: line.split()) \
  .map(lambda word: (word[0],len(word)))
```



Example: Average Word Length by Letter

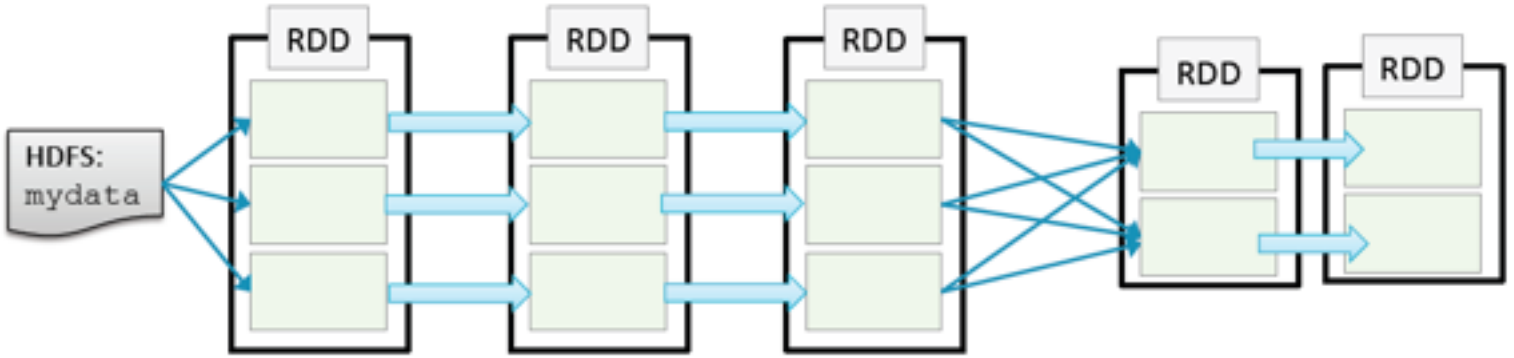
-

```
> 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)))
```

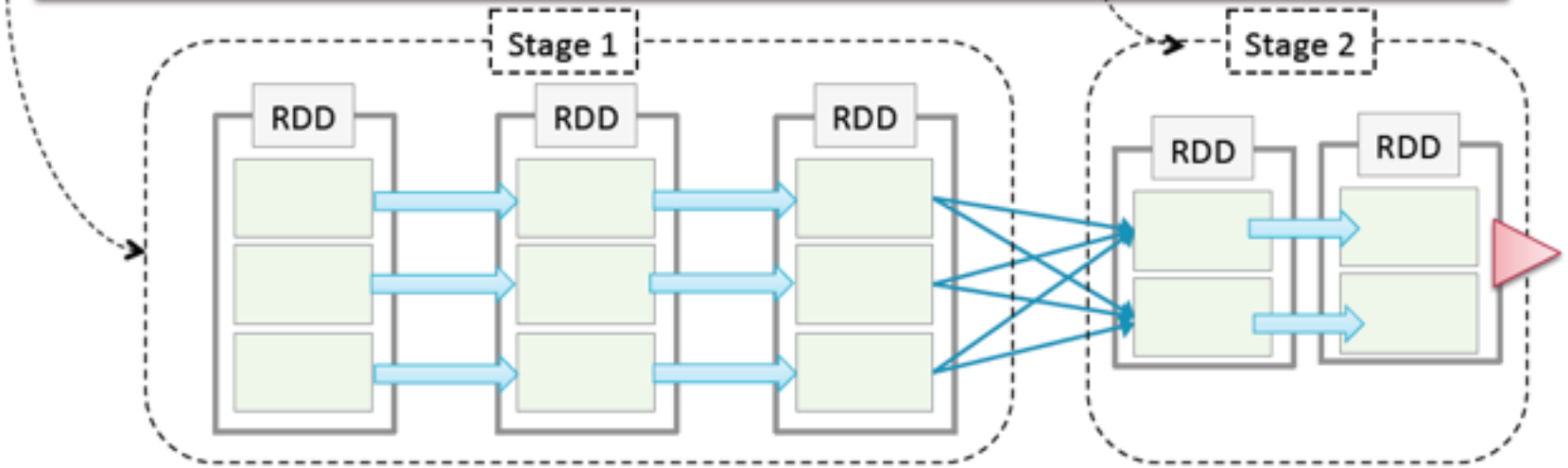


Stages

- **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**

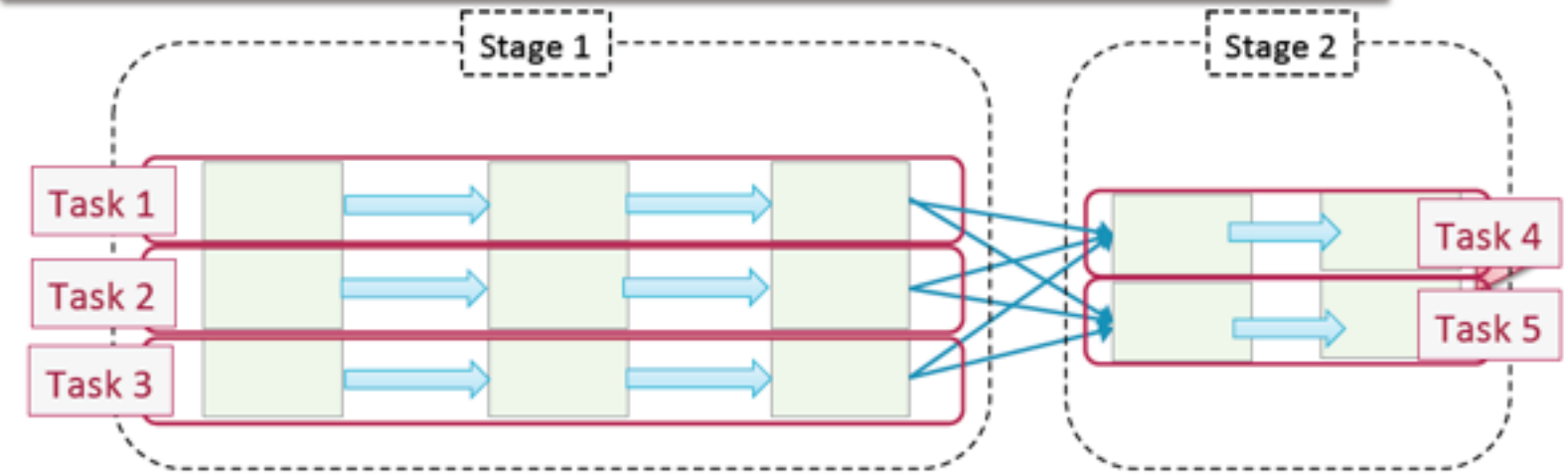
Spark Execution: Stages

```
> 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()
```



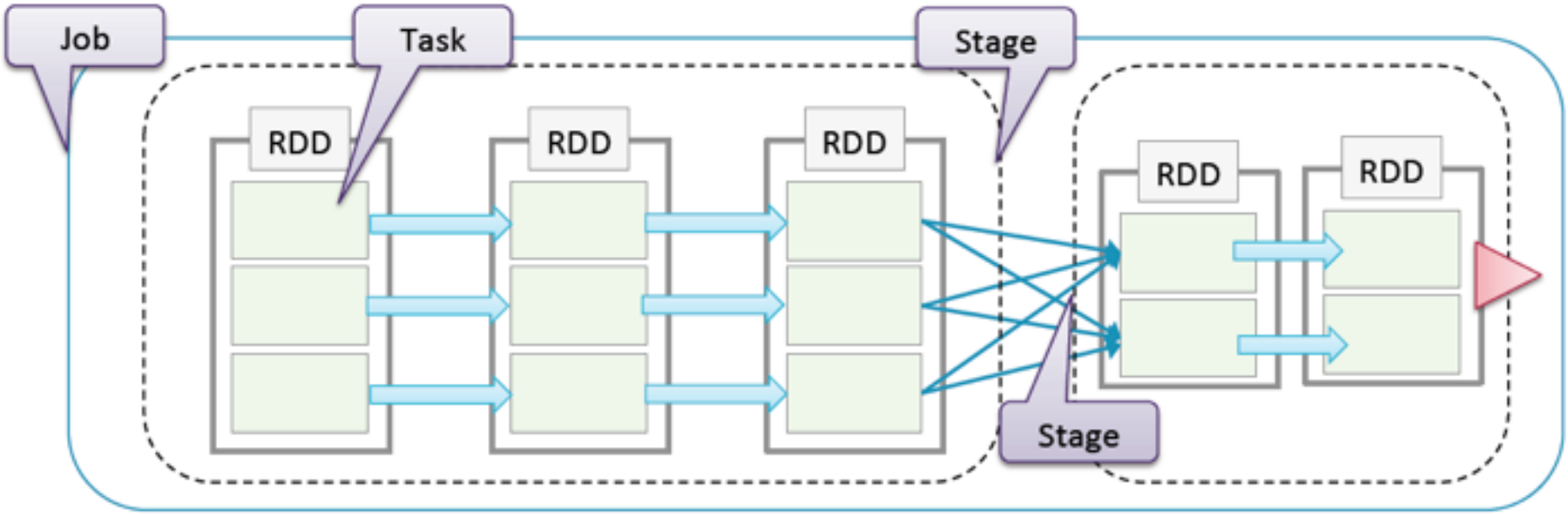
Controlling the Level of Parallelism

- RDD operations that repartition data (e.g., reduceByKey) take an optional additional parameter for the number of partitions/tasks
  - By default the number of cores in the cluster, or local threads
  - Configure with the spark.default.parallelism property

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

Summary of Spark Terminology

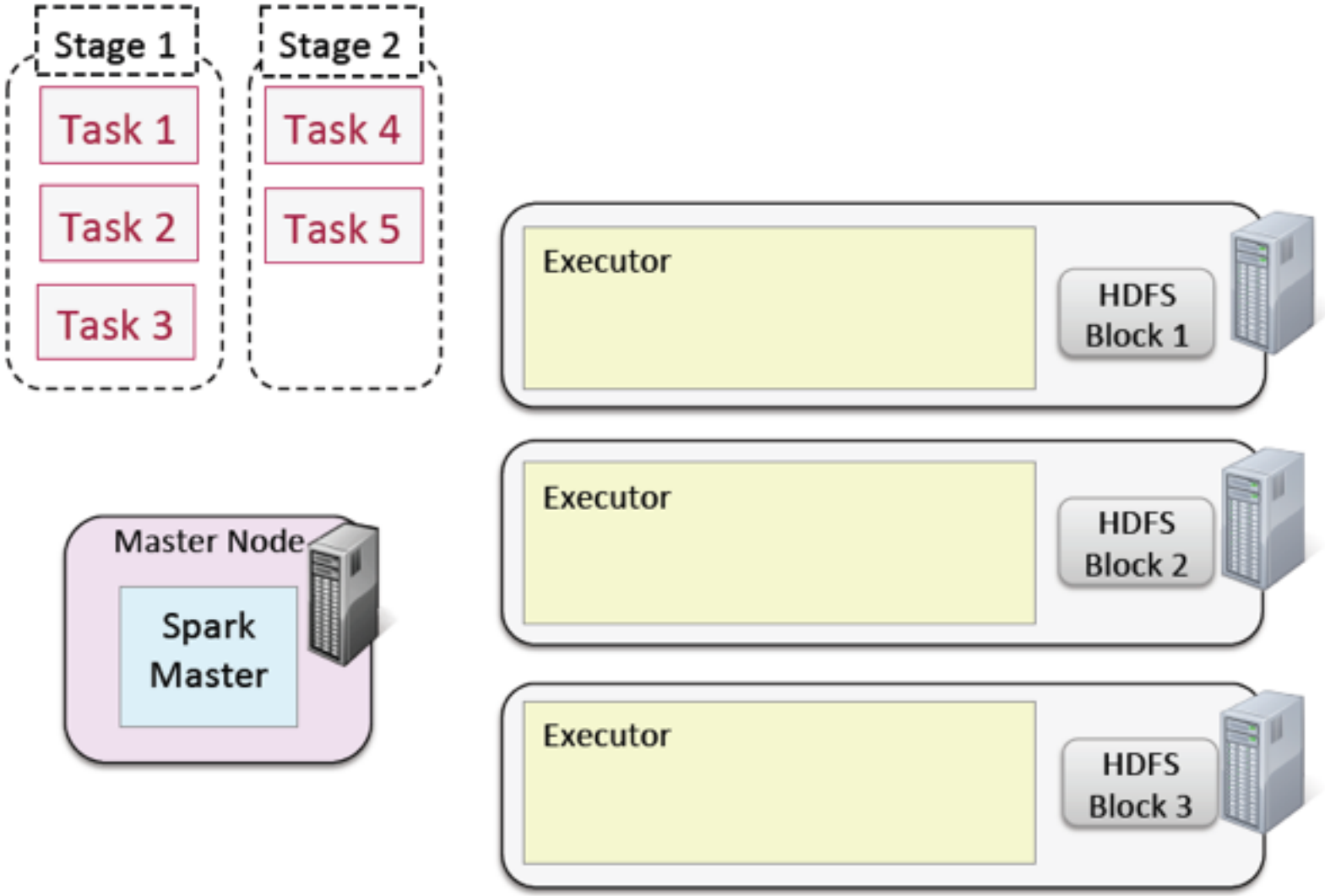
- Job – a set of tasks executed as a result of an action
- Stage – a set of tasks in a job that can be executed in parallel
- Task – an individual unit of work sent to one executor



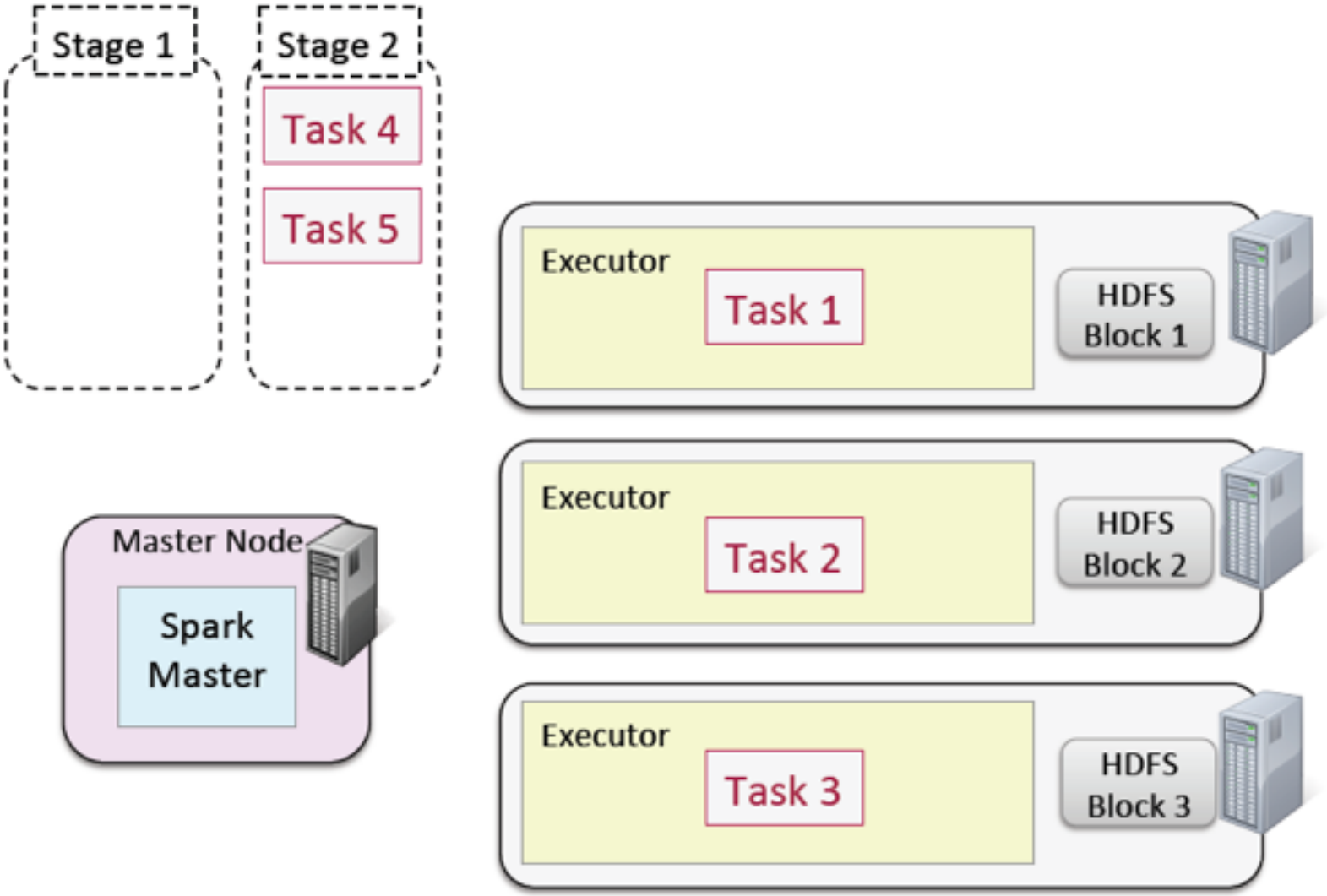
How Spark Calculates Stages

- 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

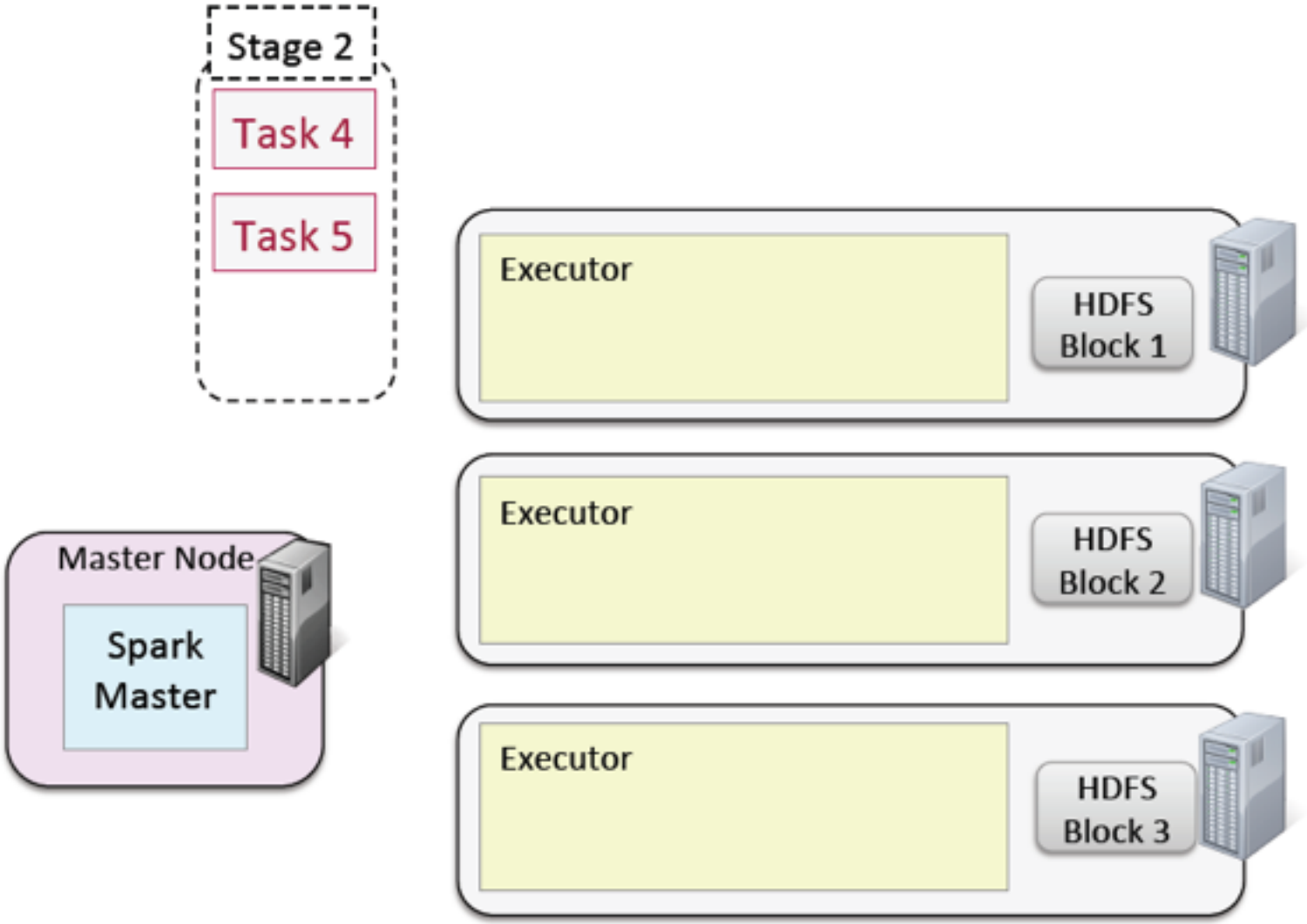
Spark Execution: Task Scheduling



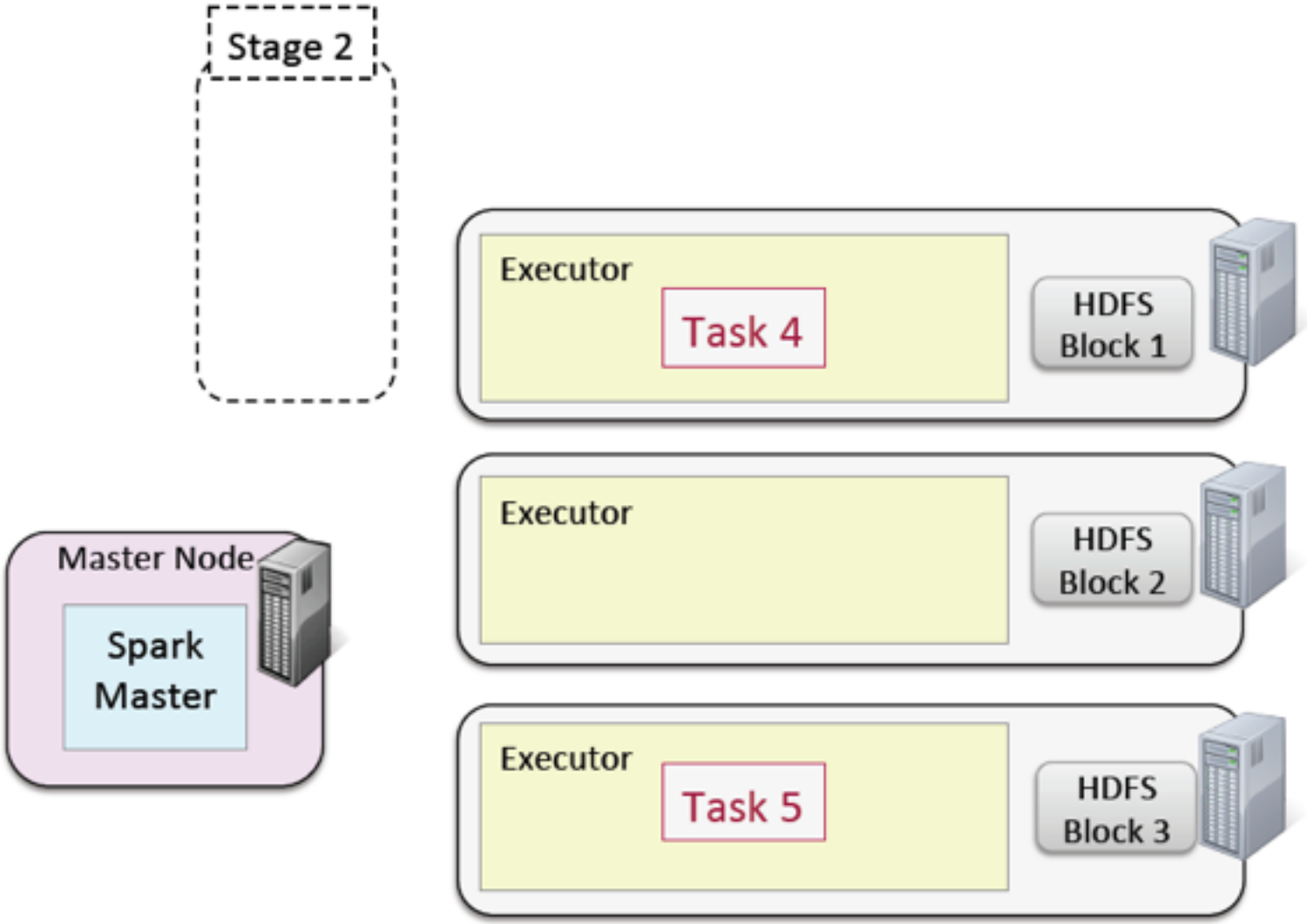
Spark Execution: Task Scheduling



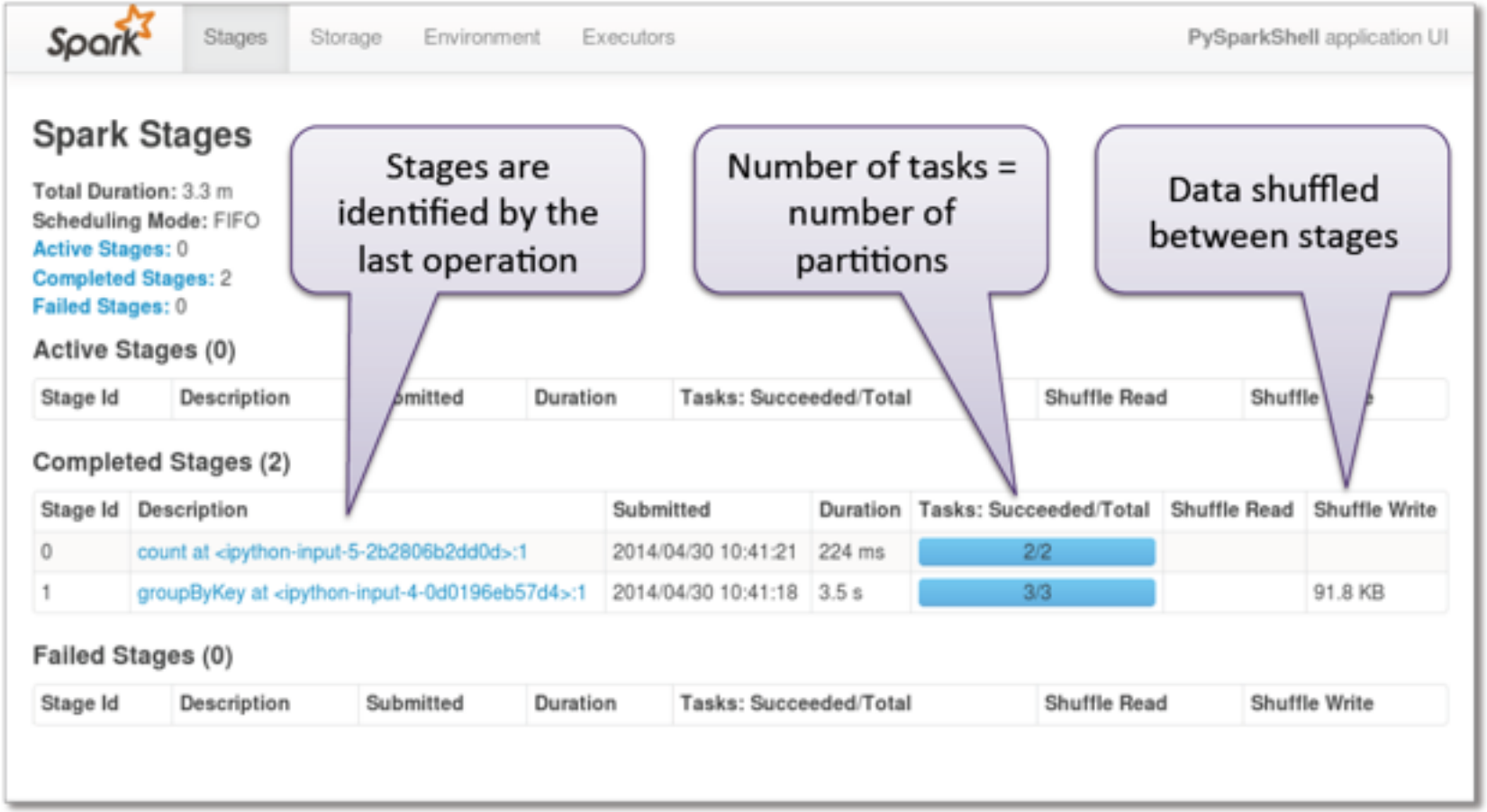
Spark Execution: Task Scheduling



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**

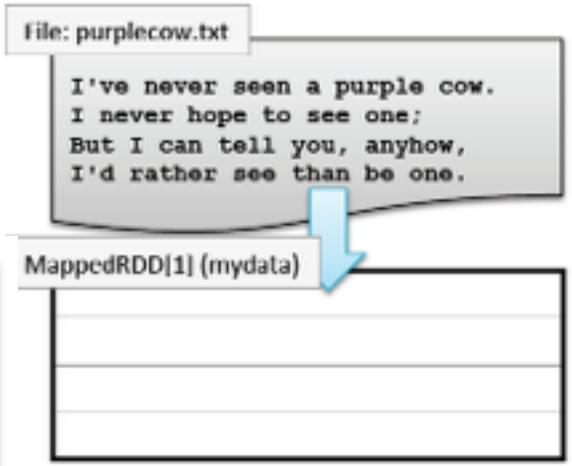
File: purplecow.txt

```
I've never seen a purple cow.
I never hope to see one;
But I can tell you, anyhow,
I'd rather see than be one.
```

Example

- Each transformation operation
  - creates a new child RDD

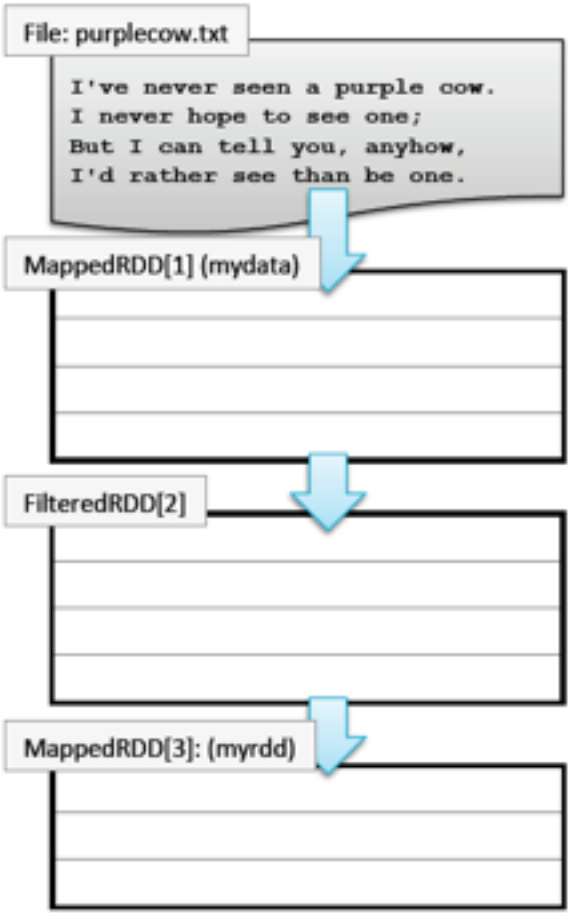
```
> mydata = sc.textFile("purplecow.txt")
```



Example

- Each transformation operation
  - creates a new child RDD

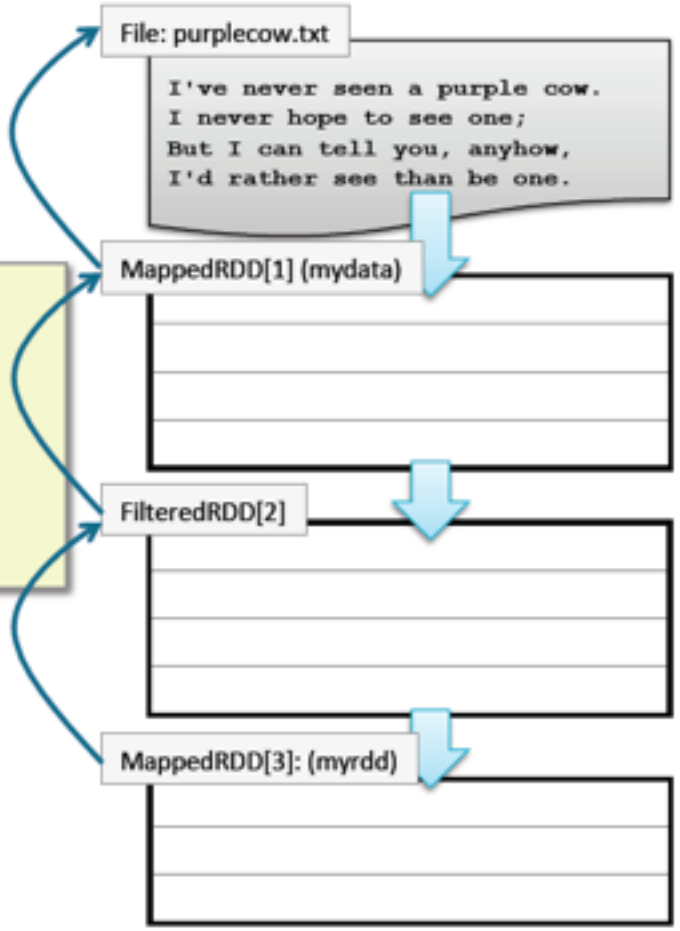
```
> mydata = sc.textFile("purplecow.txt")
> myrdd = mydata.map(lambda s: s.upper())\
.filter(lambda s:s.startswith('I'))
```



Example

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

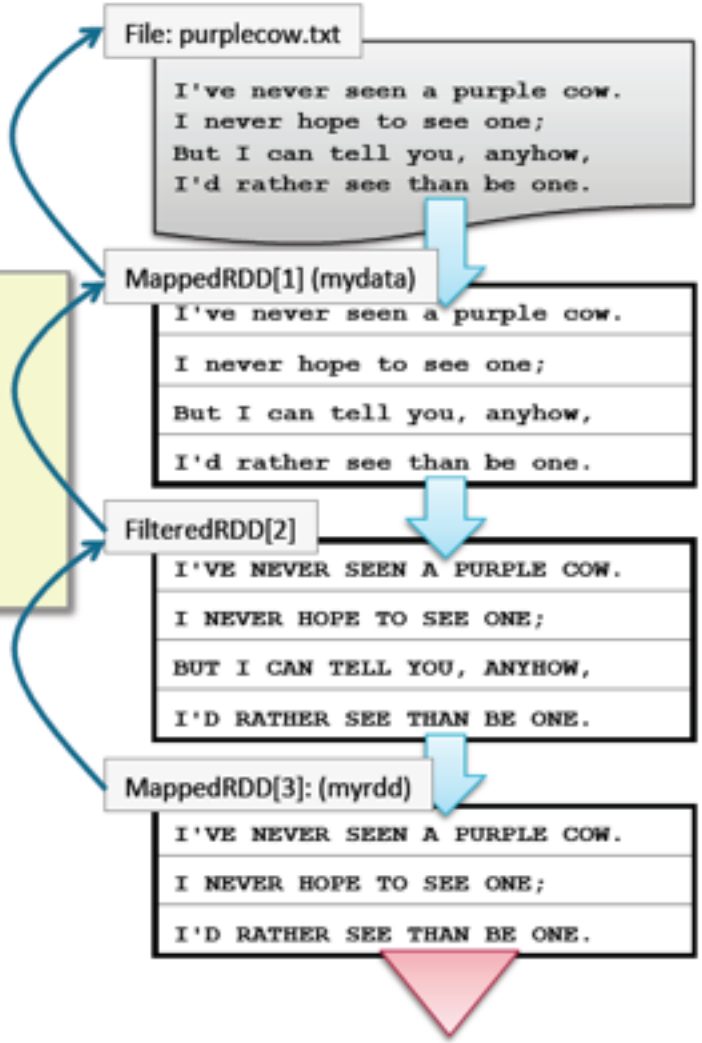
```
> mydata = sc.textFile("purplecow.txt")
> myrdd = mydata.map(lambda s: s.upper())\
  .filter(lambda s:s.startswith('I'))
```



Lineage Example

- *Action* operations execute the parent transformations

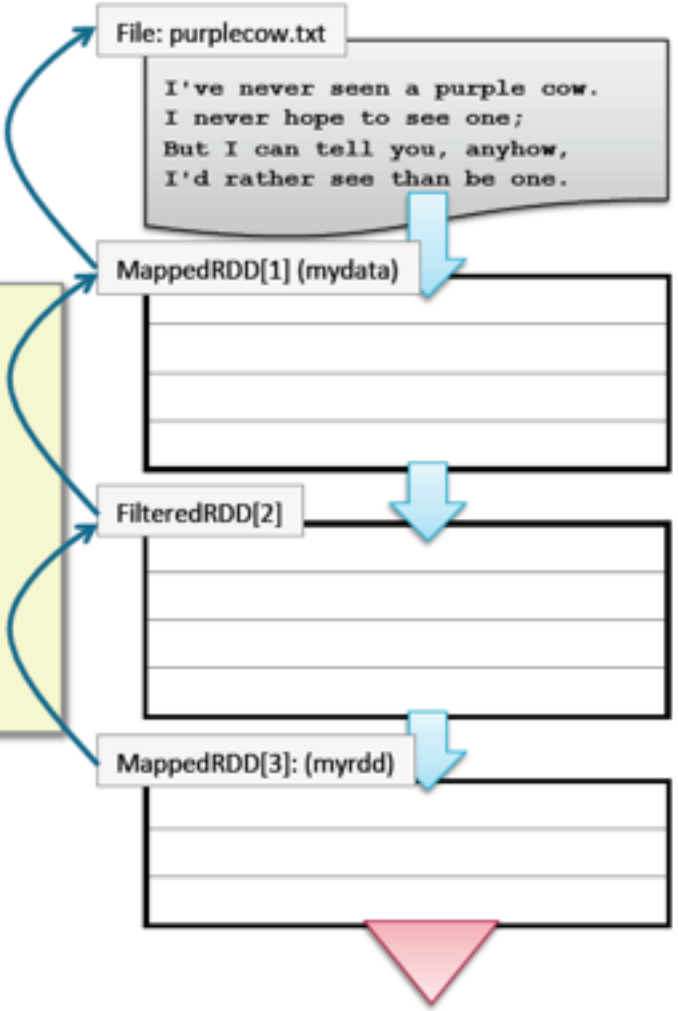
```
> mydata = sc.textFile("purplecow.txt")
> myrdd = mydata.map(lambda s: s.upper())\
  .filter(lambda s:s.startswith('I'))
> myrdd.count()
3
```



Lineage Example

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

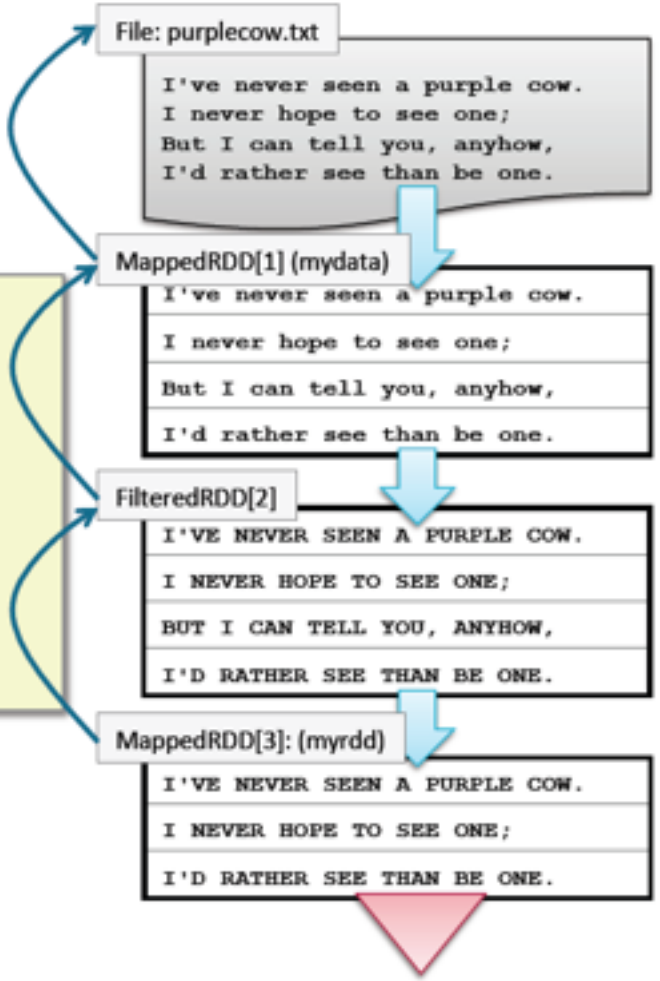
```
> mydata = sc.textFile("purplecow.txt")
> myrdd = mydata.map(lambda s: s.upper())\
  .filter(lambda s:s.startswith('I'))
> myrdd.count()
3
> myrdd.count()
```



Lineage Example

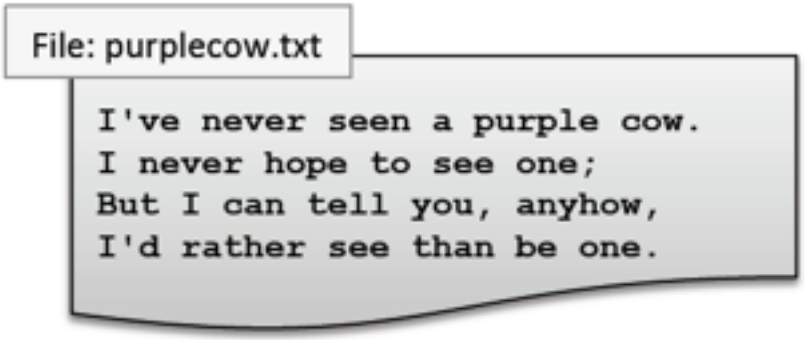
- Each action re-executes the lineage transformations starting with the base
  - By default

```
> 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

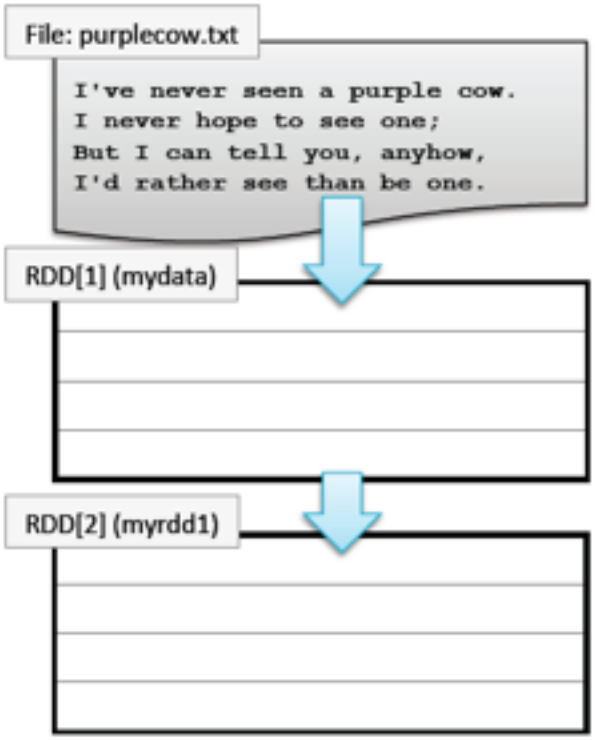
- **Caching an RDD saves the data in memory**



Caching

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

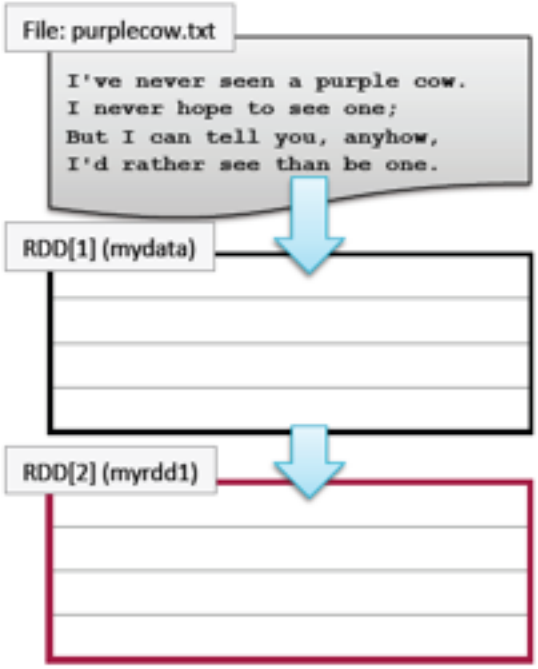
```
> mydata = sc.textFile("purplecow.txt")
> myrdd1 = mydata.map(lambda s:
  s.upper())
```



Caching

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

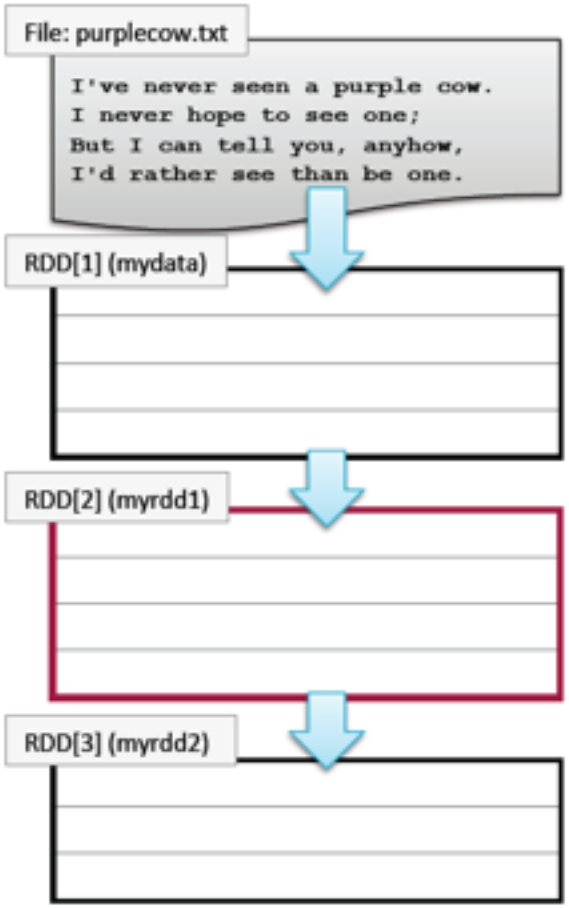
```
> mydata = sc.textFile("purplecow.txt")
> myrdd1 = mydata.map(lambda s:
  s.upper())
> myrdd1.cache()
```



Caching

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

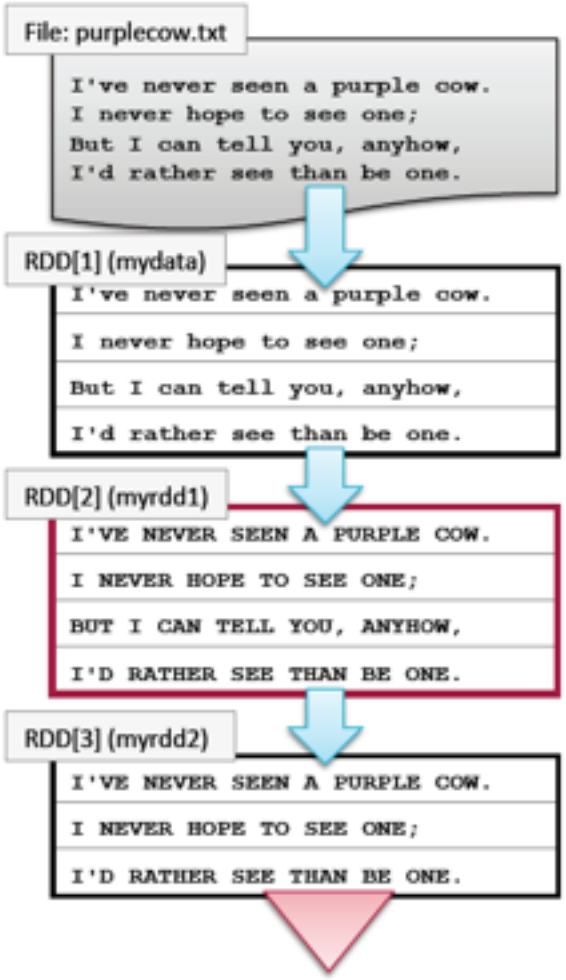
```
> mydata = sc.textFile("purplecow.txt")
> myrdd1 = mydata.map(lambda s:
  s.upper())
> myrdd1.cache()
> myrdd2 = myrdd1.filter(lambda \
  s:s.startswith('I'))
```



Caching

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

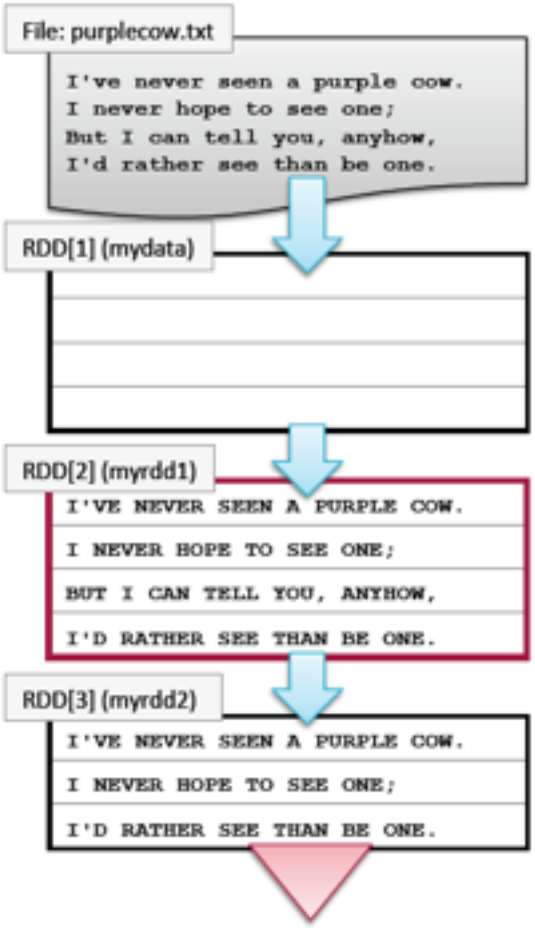
```
> mydata = sc.textFile("purplecow.txt")
> myrdd1 = mydata.map(lambda s:
  s.upper())
> myrdd1.cache()
> myrdd2 = myrdd1.filter(lambda \
  s:s.startswith('I'))
> myrdd2.count()
3
```



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

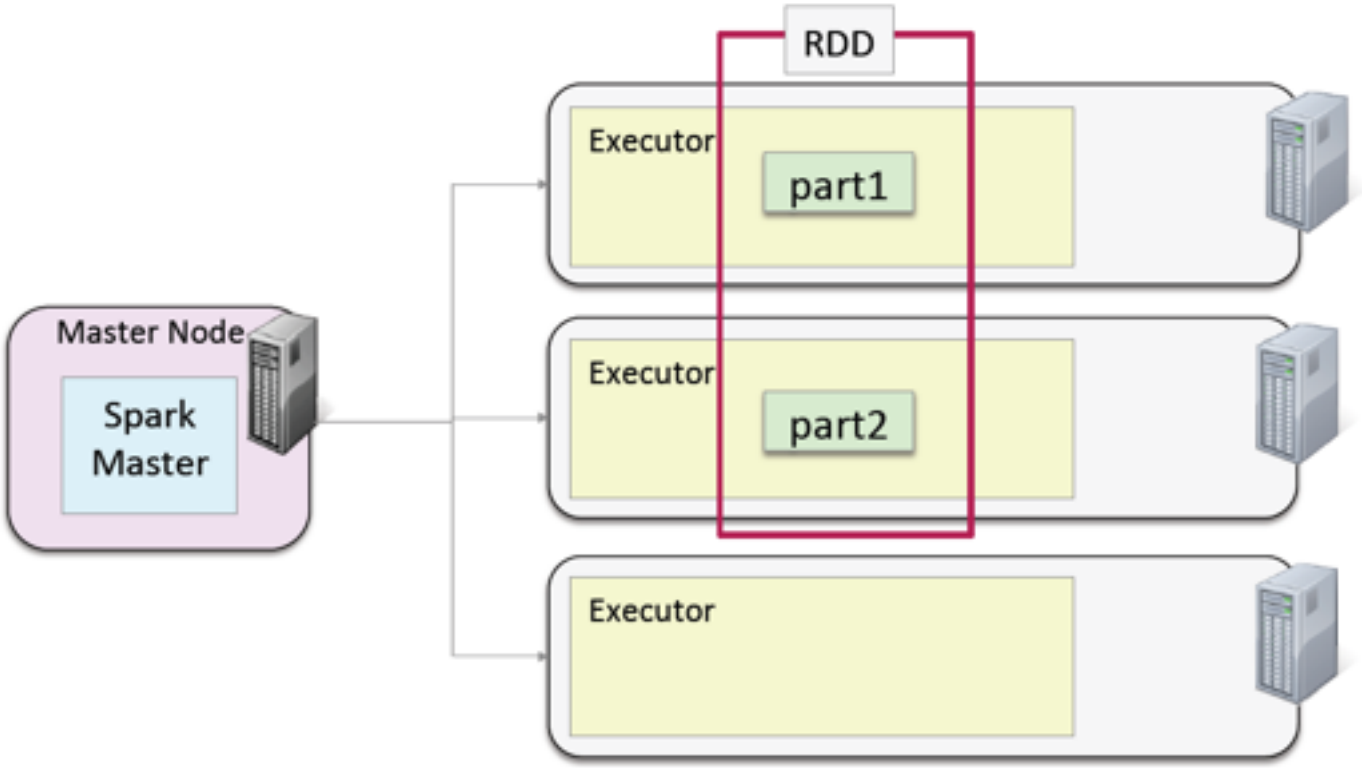
- **Caching is a suggestion to Spark**
  - **If not enough memory is available, transformations will be re-executed when needed**

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**

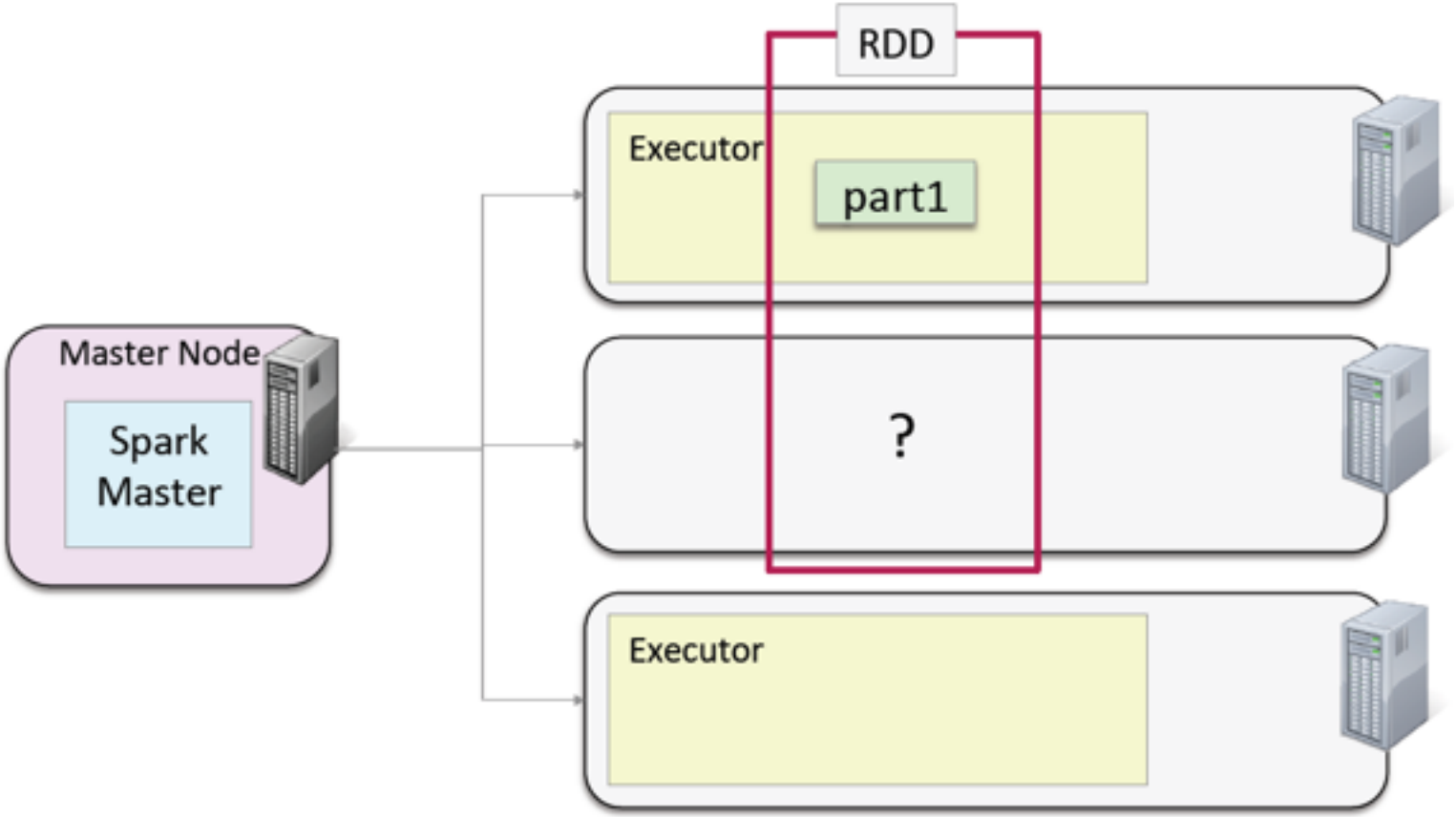
Distributed Cache

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



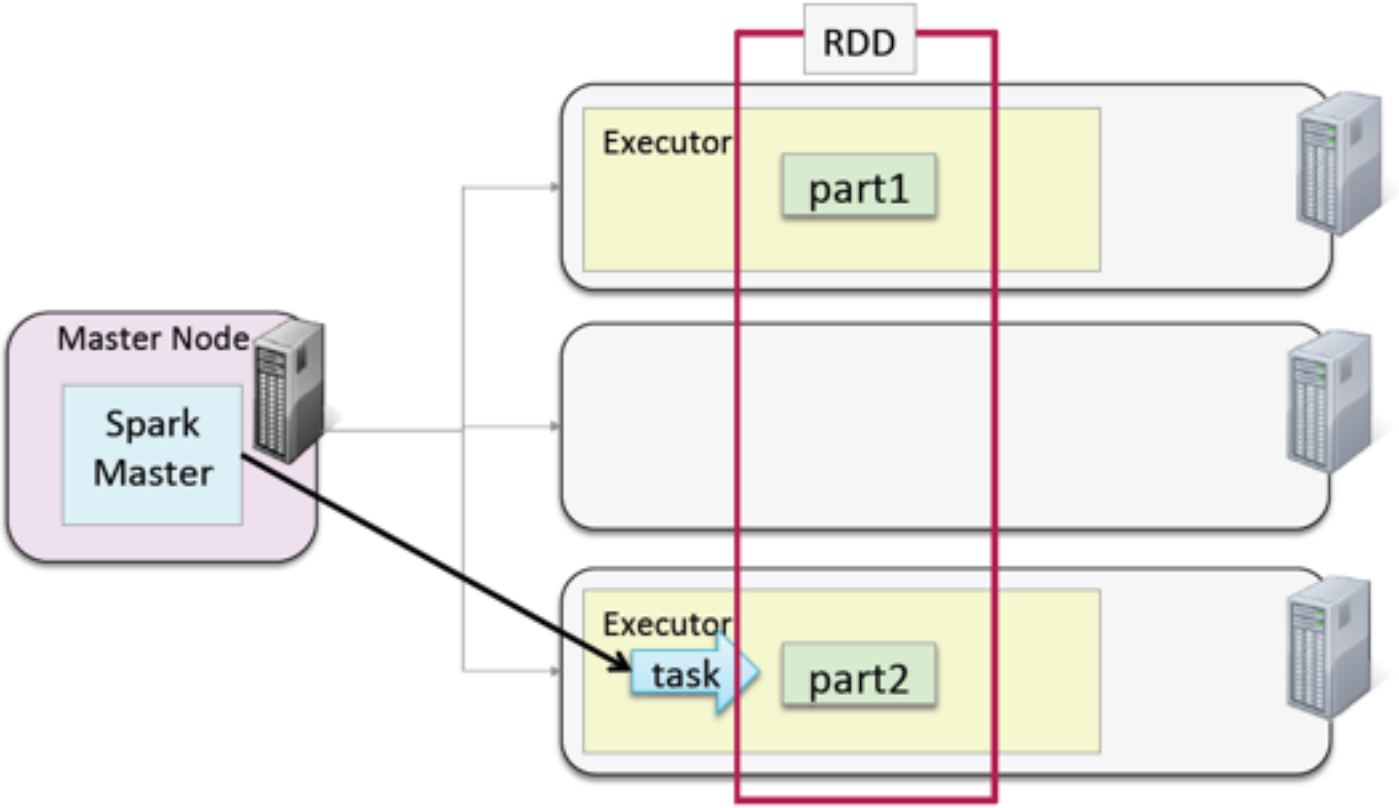
RDD Fault-Tolerance

- **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

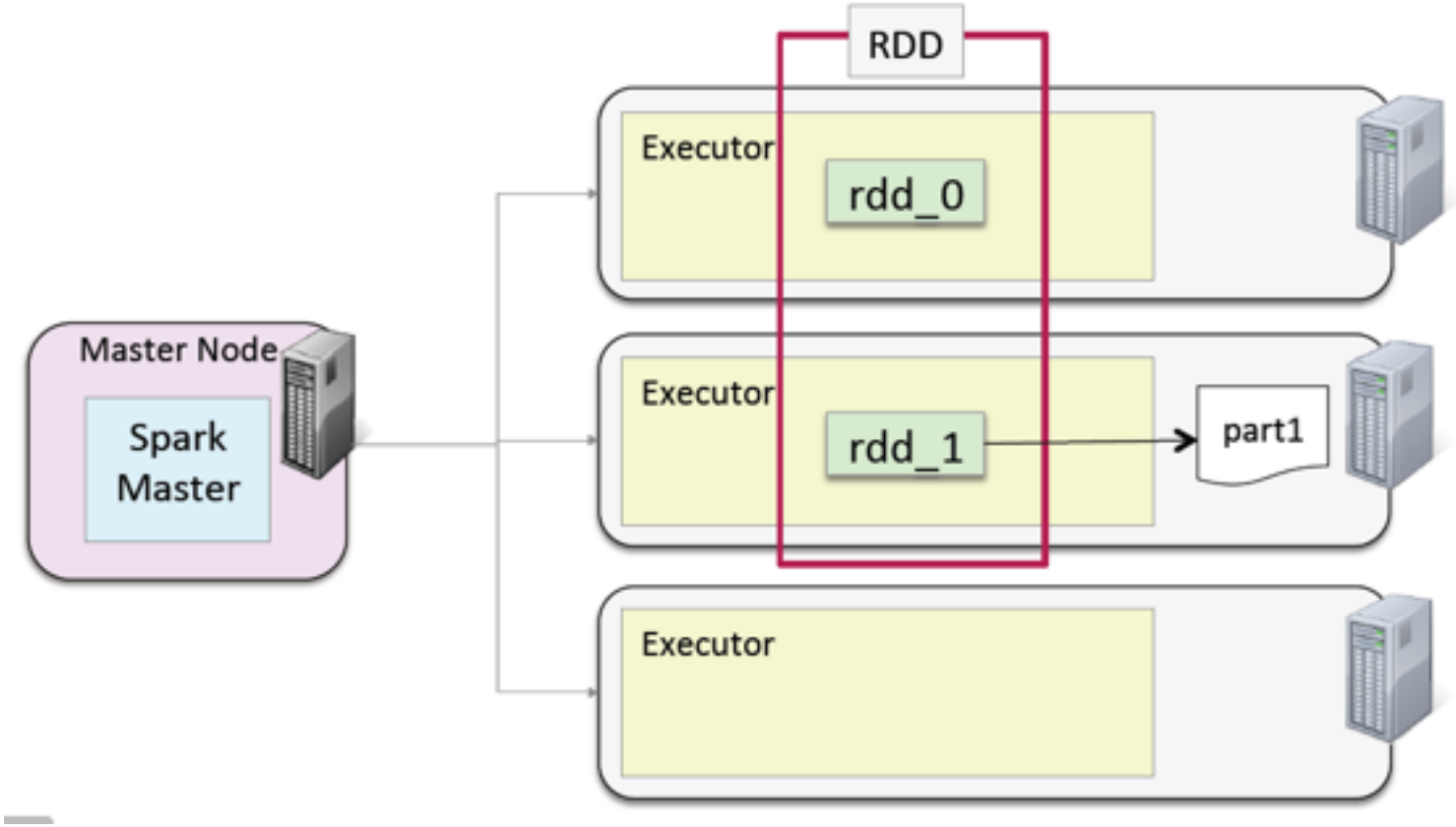
- You can choose to serialize the data in memory
  - MEMORY_ONLY_SER and MEMORY_AND_DISK_SER
  - Much more space efficient
  - Less time efficient
    - Choose a fast serialization library
- Replication - store partitions on two nodes
  - MEMORY_ONLY_2, MEMORY_AND_DISK_2, etc.

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**

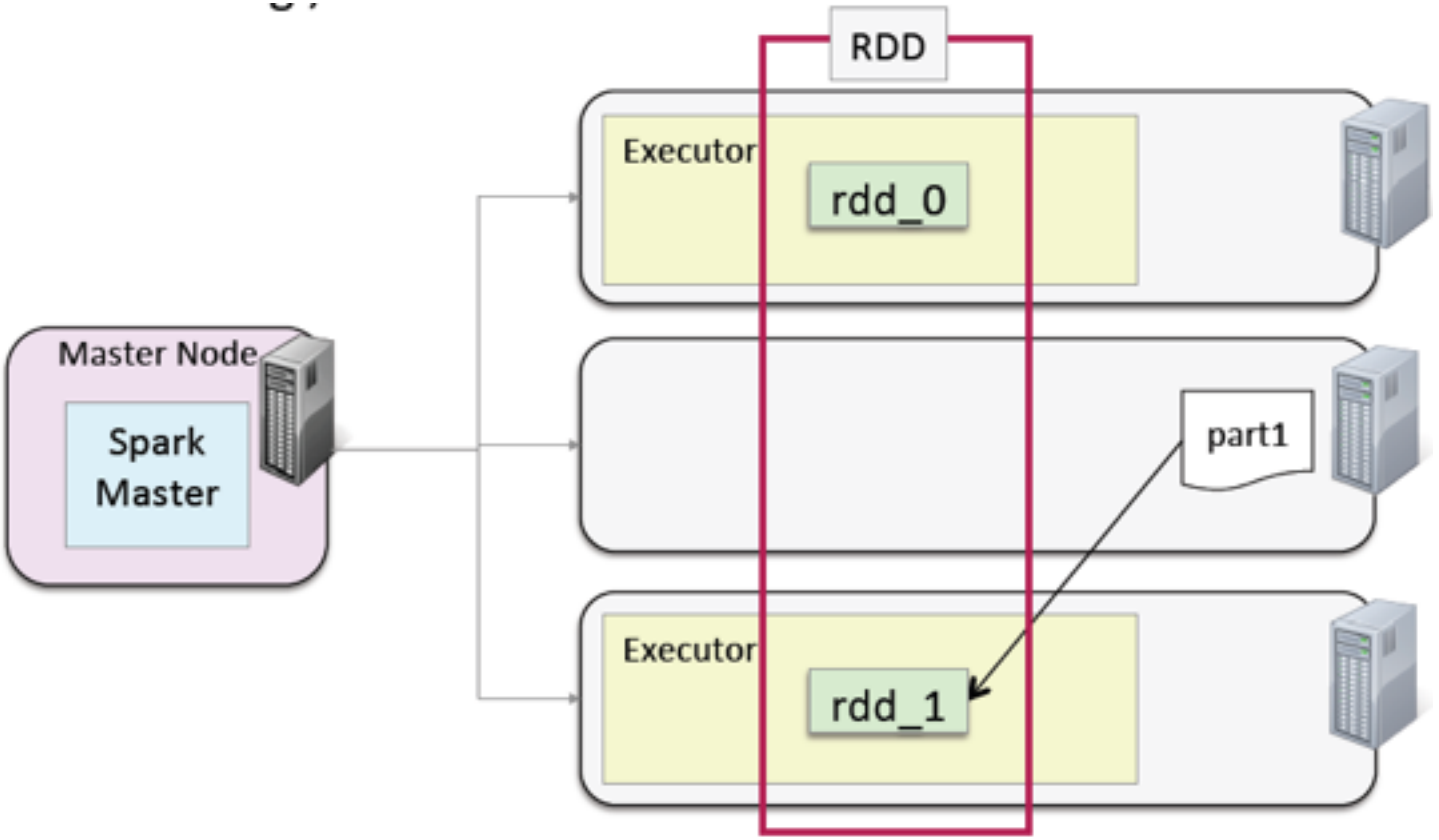
Distributed Disk Persistence

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



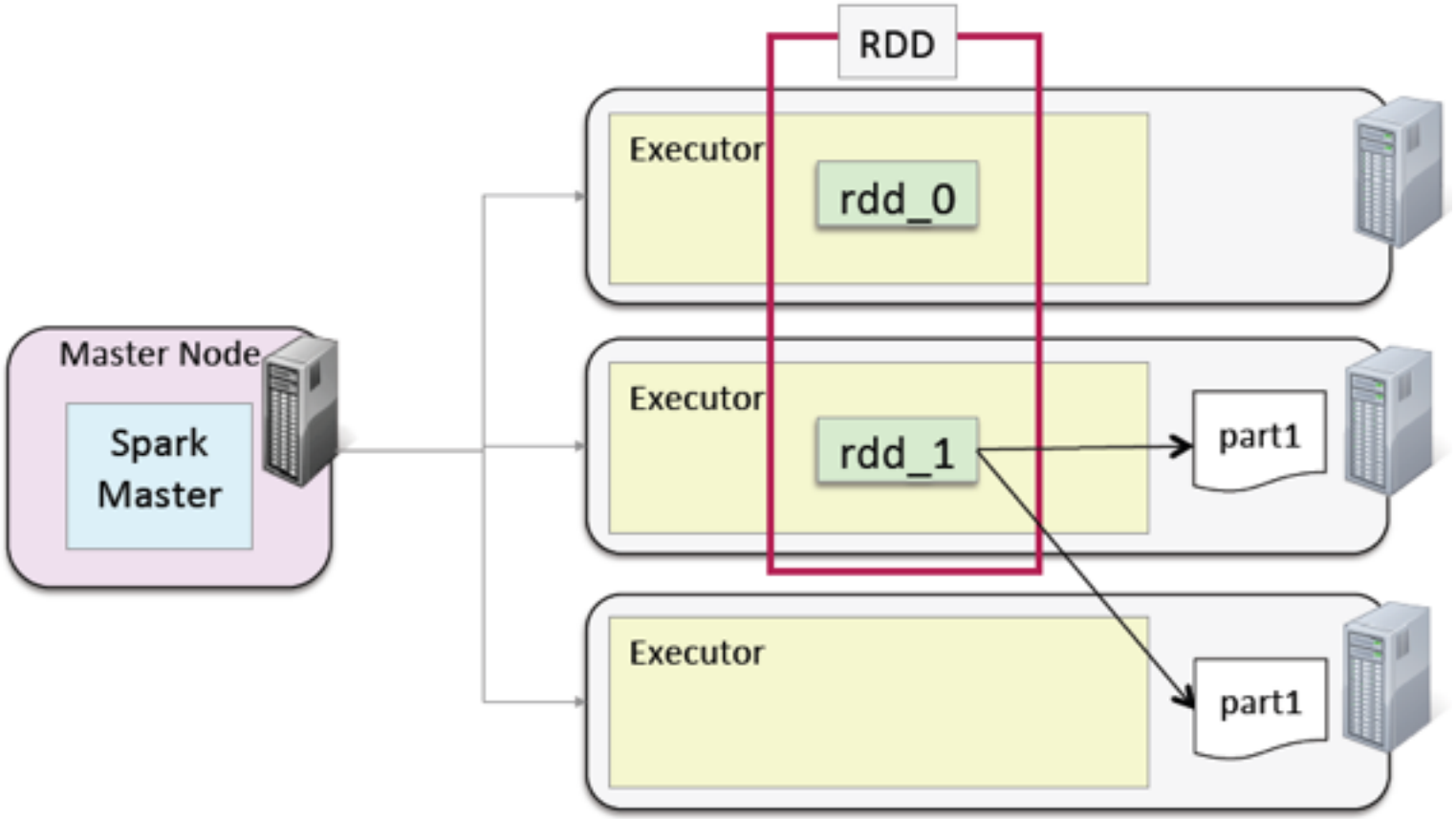
Distributed Disk Persistence

- **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**

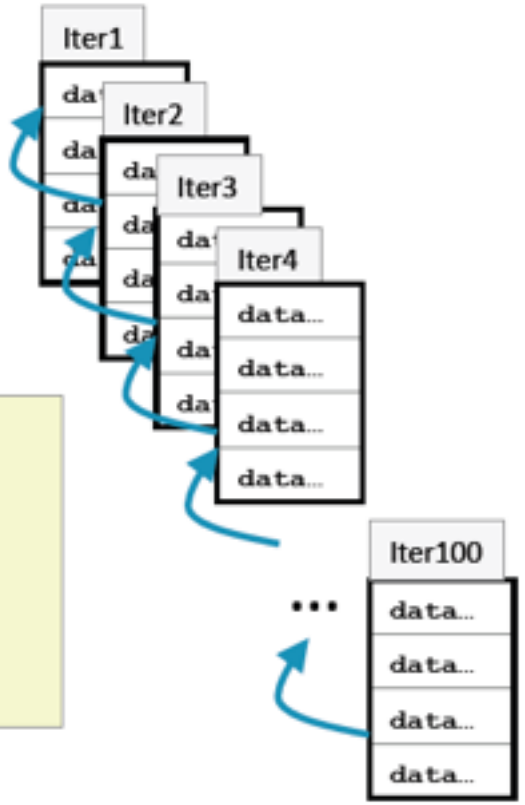


When and Where to Cache

- 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**

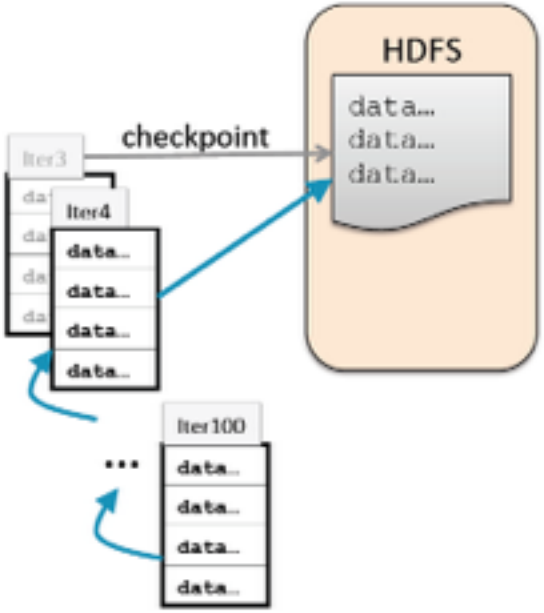



```
myrdd = …initial-value…
while x in xrange(100):
   myrdd = myrdd.transform(…)
myrdd.saveAsTextFile()
```

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

Building and Running Scala Applications in the Hands-On Exercises

- **Basic Maven projects are provided in the exercises/projects directory with two packages**
  - **stubs - starter Scala file, do exercises here**
  - **solution - final exercise solution**

```
$ mvn package

$ spark-submit \
  --class stubs.CountJPGs \
  target/countjpgs-1.0.jar \
  weblogs.*
```

Project Directory Structure

```
+countjpgs
  -pom.xml
  +src
    +main
      +scala
        +solution
          -CountJPGs.scala
        +stubs
          -CountJPGs.scala
  +target
    -countjpgs-1.0.jar
```

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

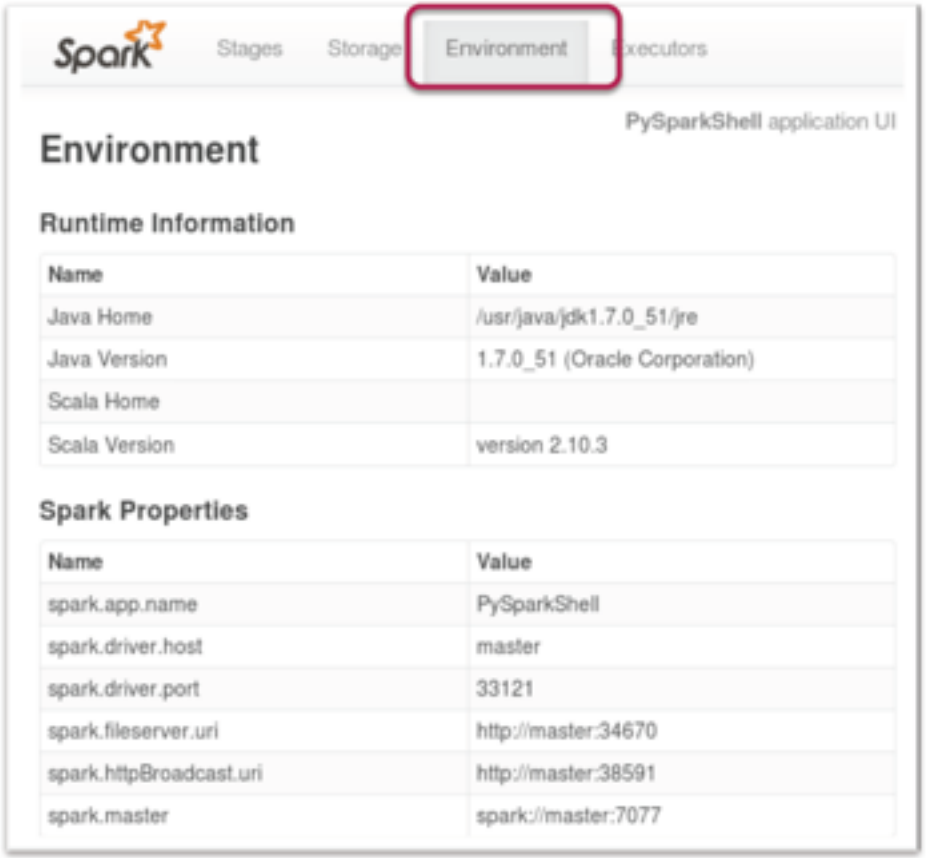
- Spark configuration settings are part of the SparkContext
- Configure using a SparkConf object
- Some example functions
  - setAppName(name)
  - setMaster(master)
  - set(property-name, value)
- Set functions return a SparkConf object to support chaining

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

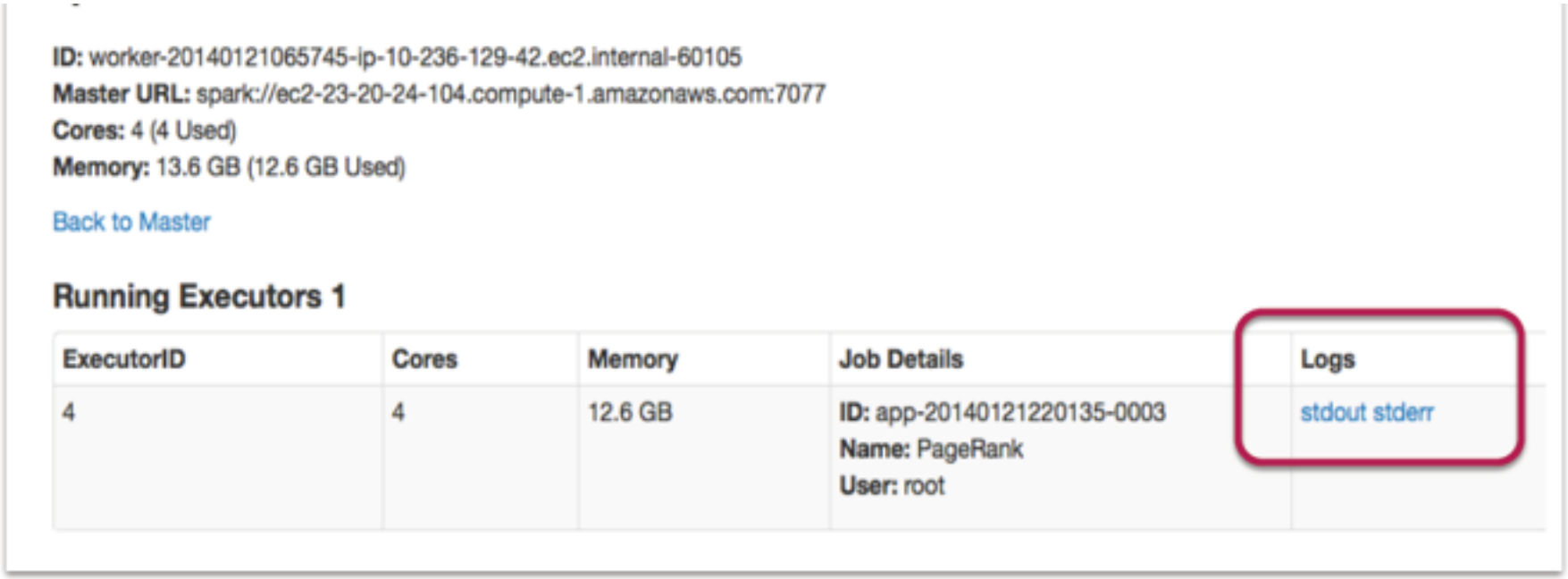
- Spark uses Apache Log4j for logging
  - Allows for controlling logging at runtime using a properties file
  - Enable or disable logging, set logging levels, select output destination
  - For more info see http://logging.apache.org/log4j/1.2/
- Log4j provides several logging levels
  - Fatal
  - Error
  - 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"Spark"Logging"(1)"

- 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 Spark Logging (2)

- 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

Chapter 09

SPARK STREAMING

What is Spark Streaming?

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

Why Spark Streaming?

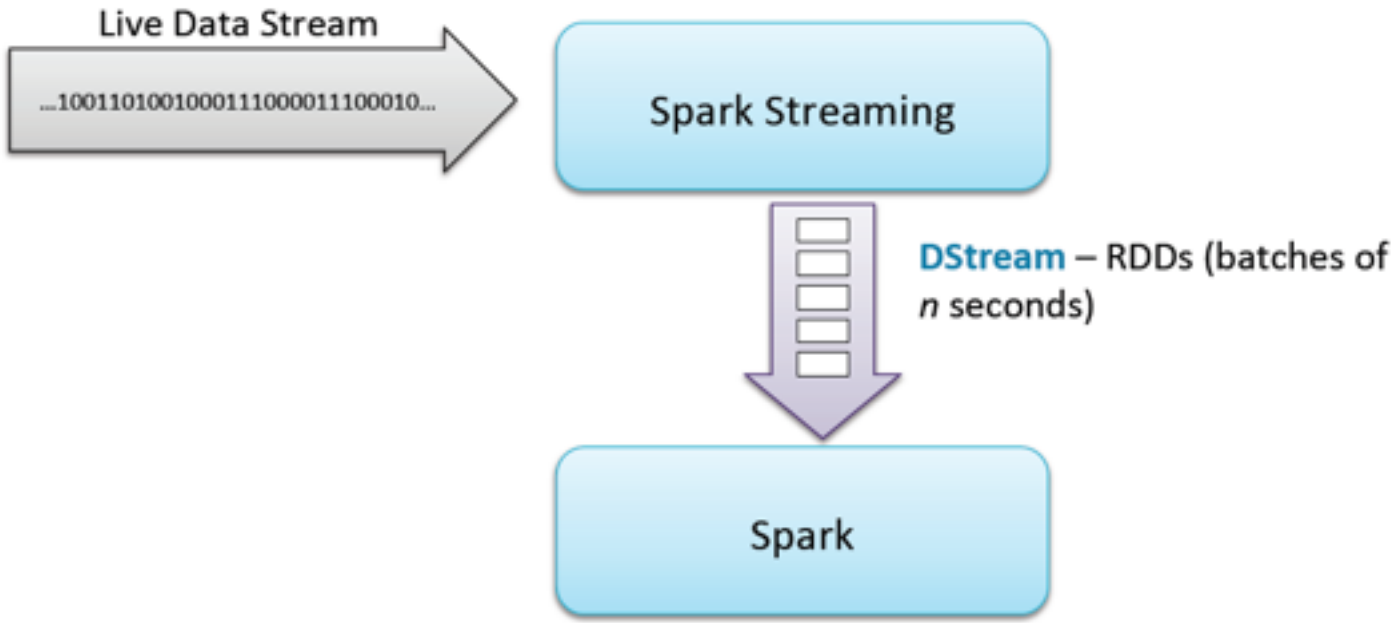
- **Many big data applications need to process large data streams in real time**
  - **Website monitoring**
  - **Fraud detection**
  - **Ad monetization**

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()

    s
    s
  }
}
```

- 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: Streaming Request Count (Recap)

```
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()

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

Streaming Example Output

```
-------------------------------------------
Time: 1401219545000 ms
-------------------------------------------
(23713,2)
(53,2)
(24444,2)
(127,2)
(93,2)
...
```

Starts 2 seconds after `ssc.start`

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)
...
```

2 seconds later...

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)
…
```

2 seconds later…

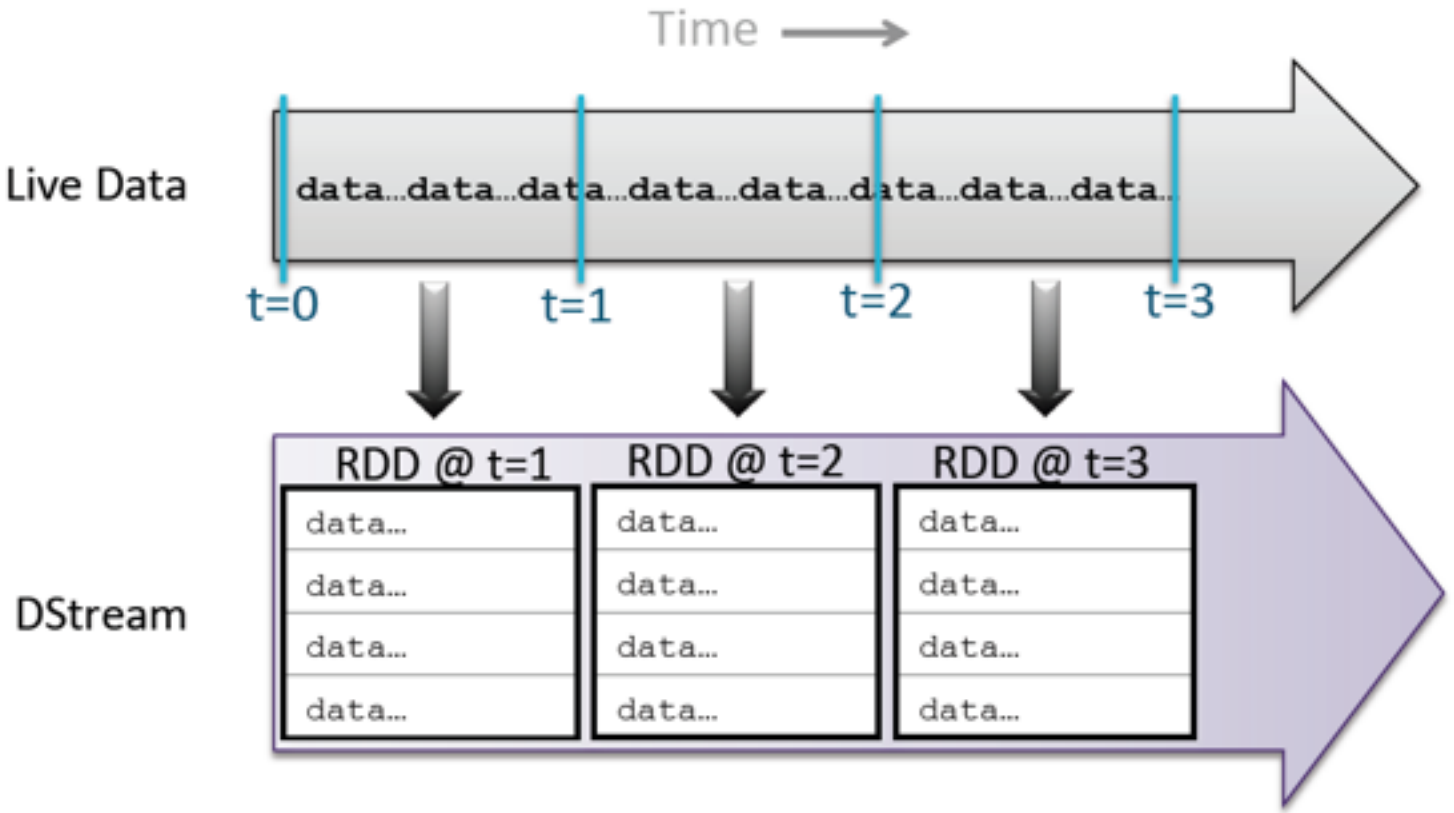
Streaming Example Output

```
-------------------------------------------
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

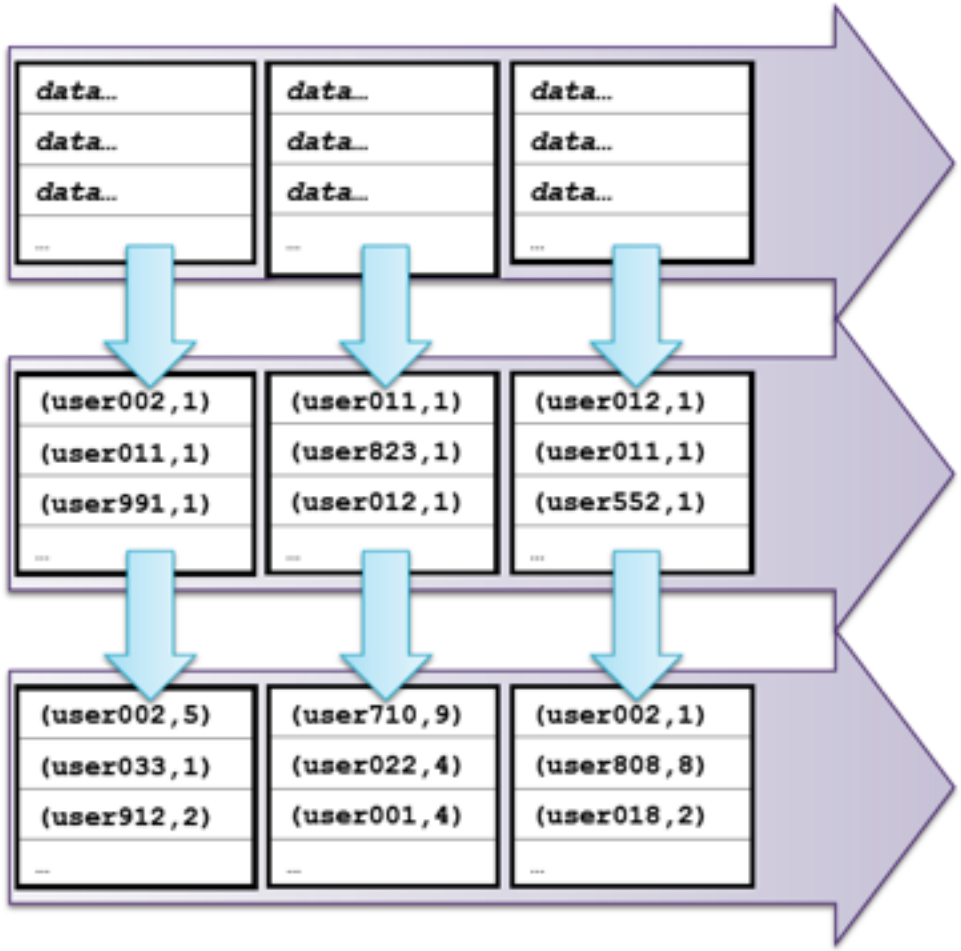
Dstream Transformations

- 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))
```

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

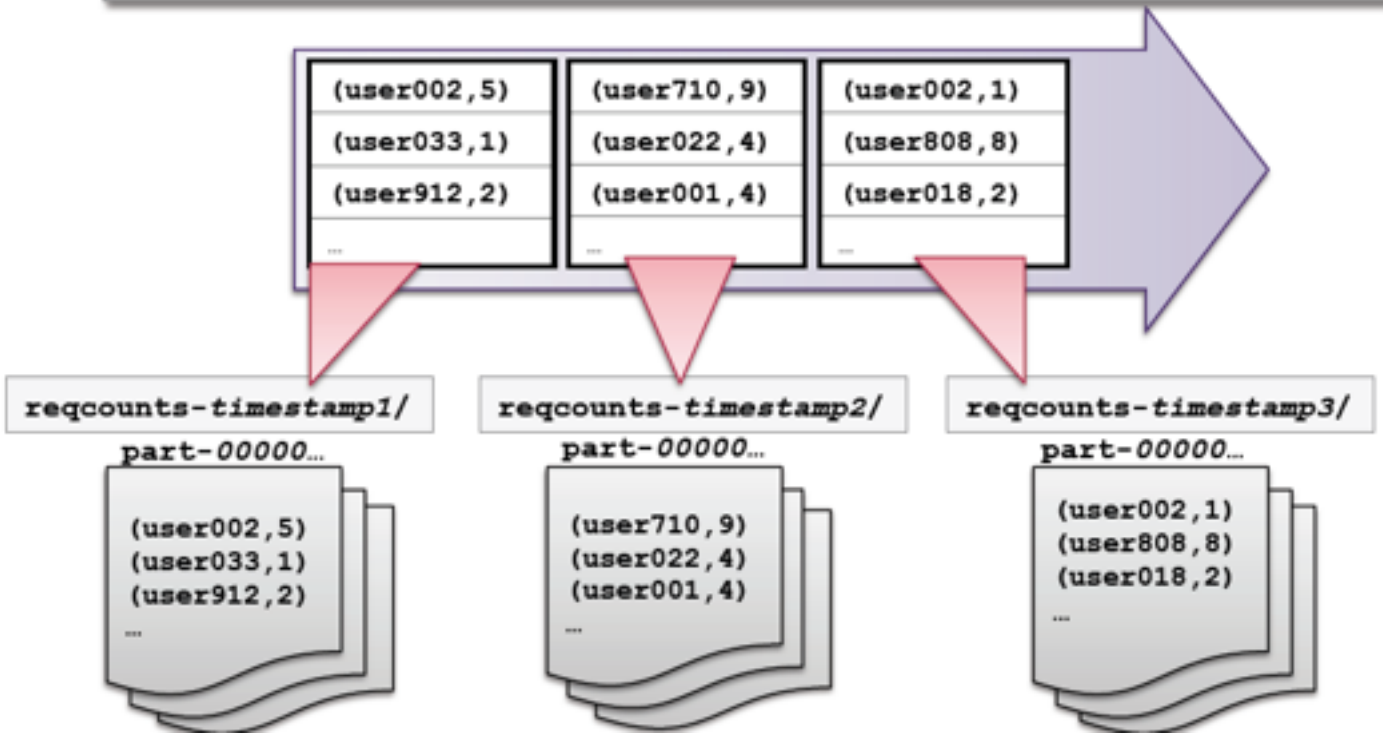


Dstream Output Operations

- 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(

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()
…
```

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

Find Top Users – Output

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

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

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)
```

t =1
(2 seconds later)

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]
```

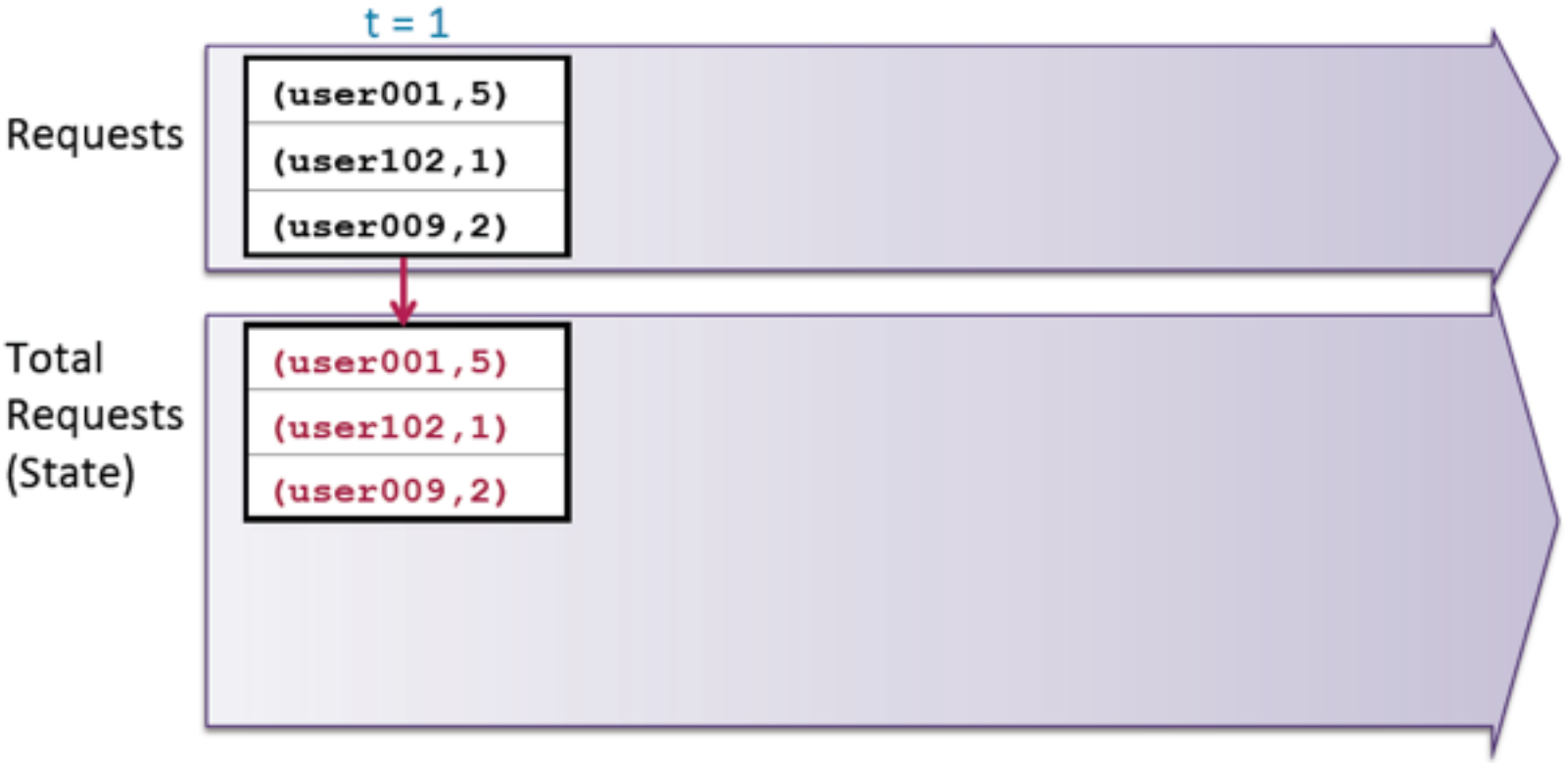
  - Stopping and restarting the Streaming Context is unsupported

Hands-On Exercise: Exploring Spark Streaming

HANDS-ON EXERCISE: EXPLORING SPARK STREAMING

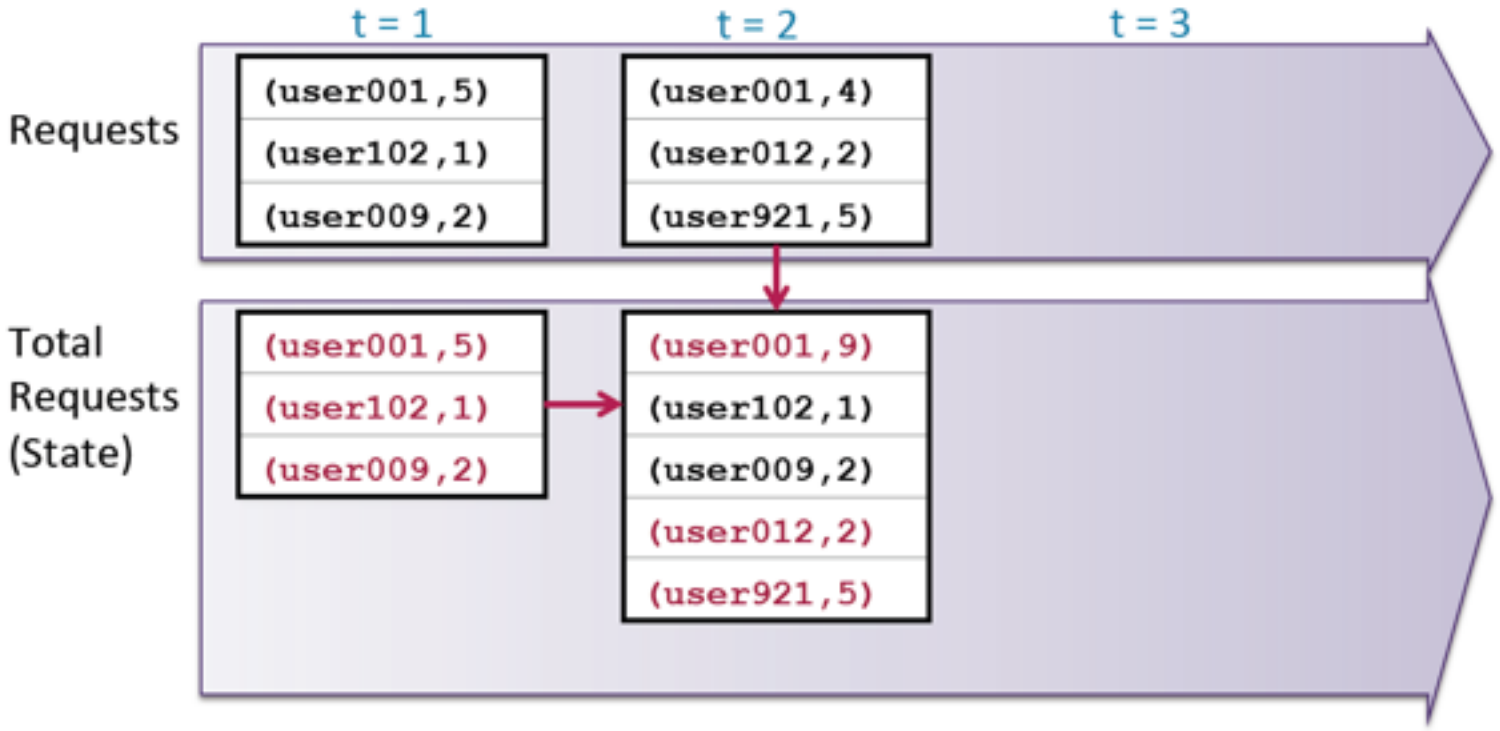
State Dstreams

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



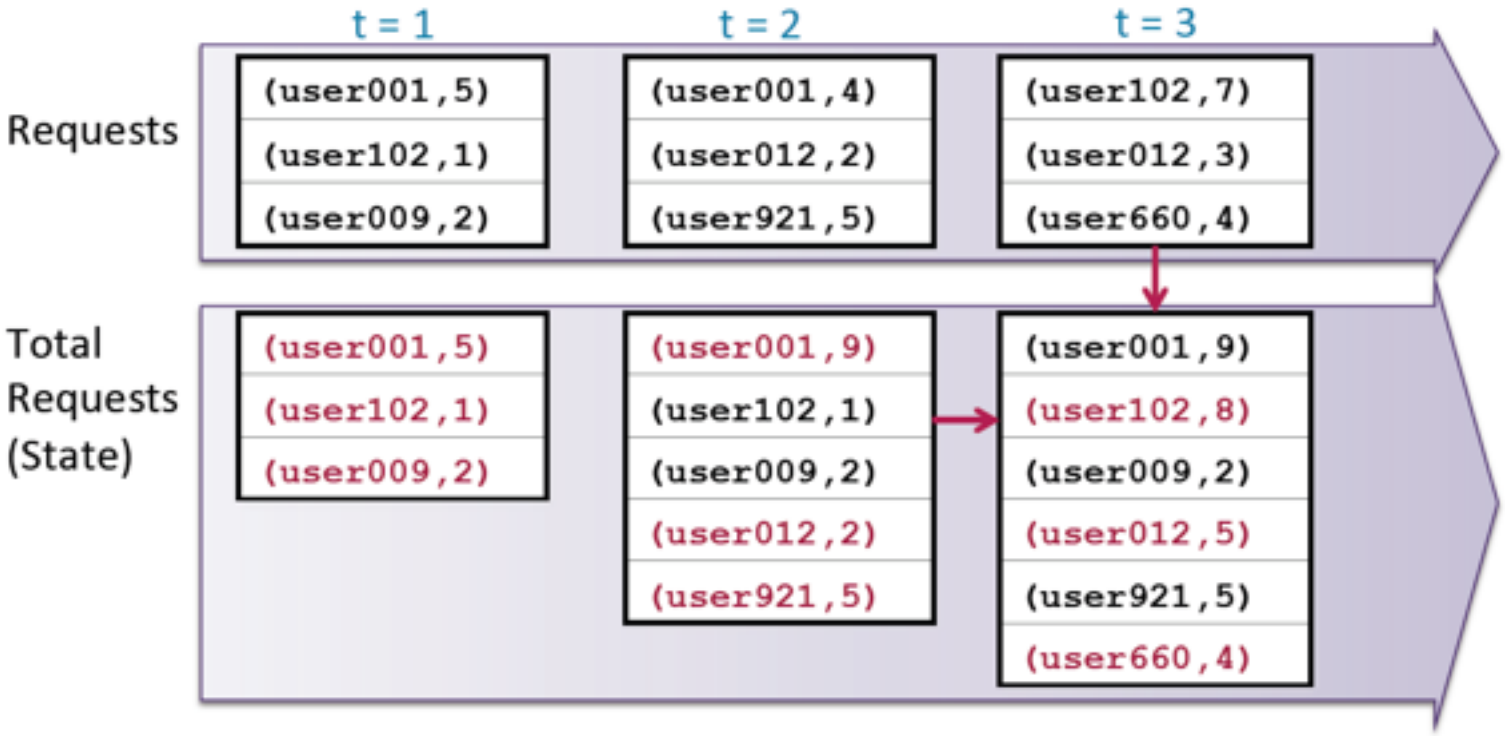
State Dstreams

- **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.

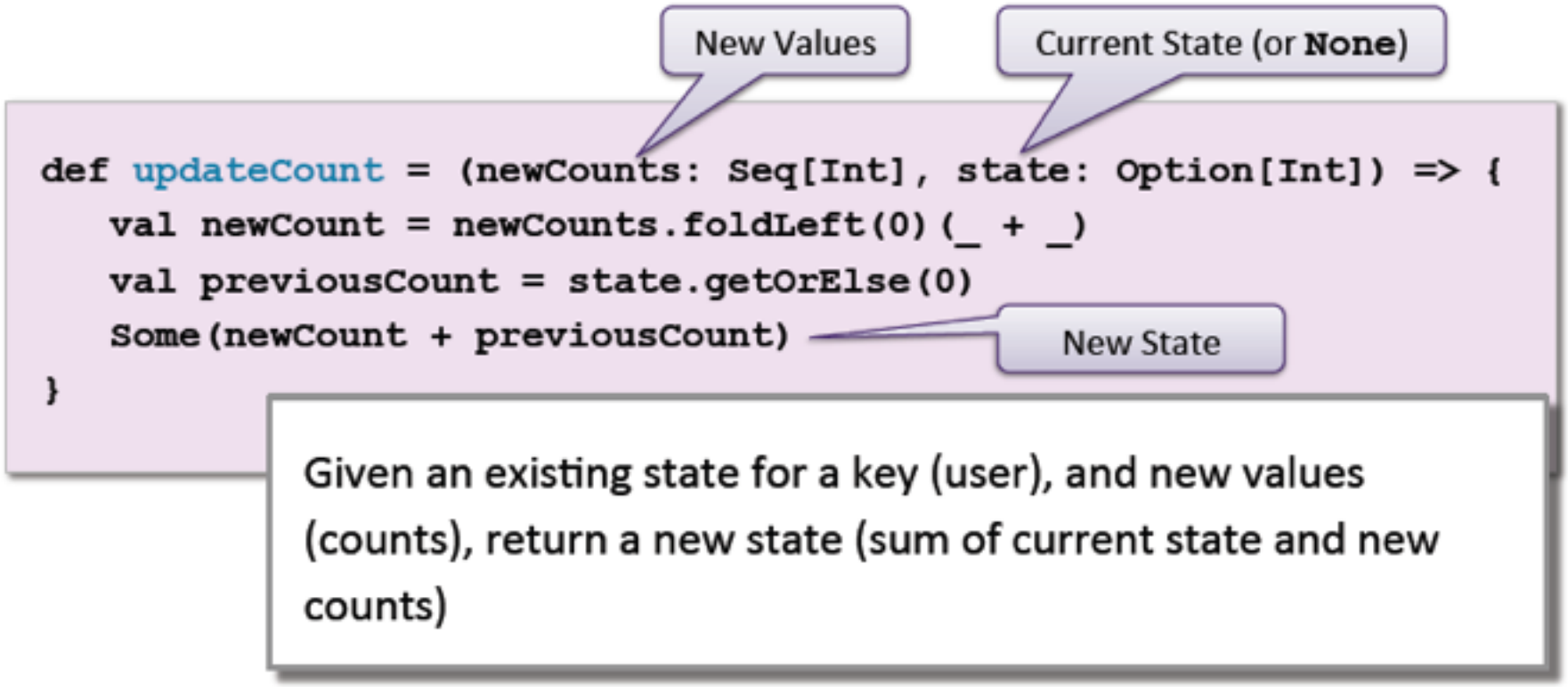
Total User Request Count

```
…
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...

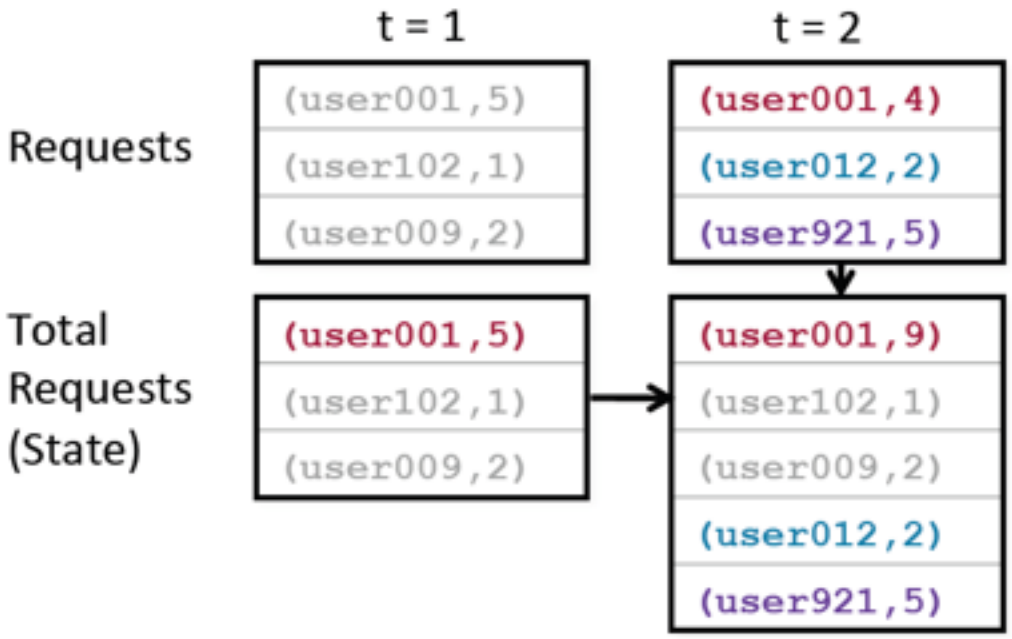
Compute a state DStream based on the previous states updated with the values from the current batch of request counts

Total User Request Count – Update Function

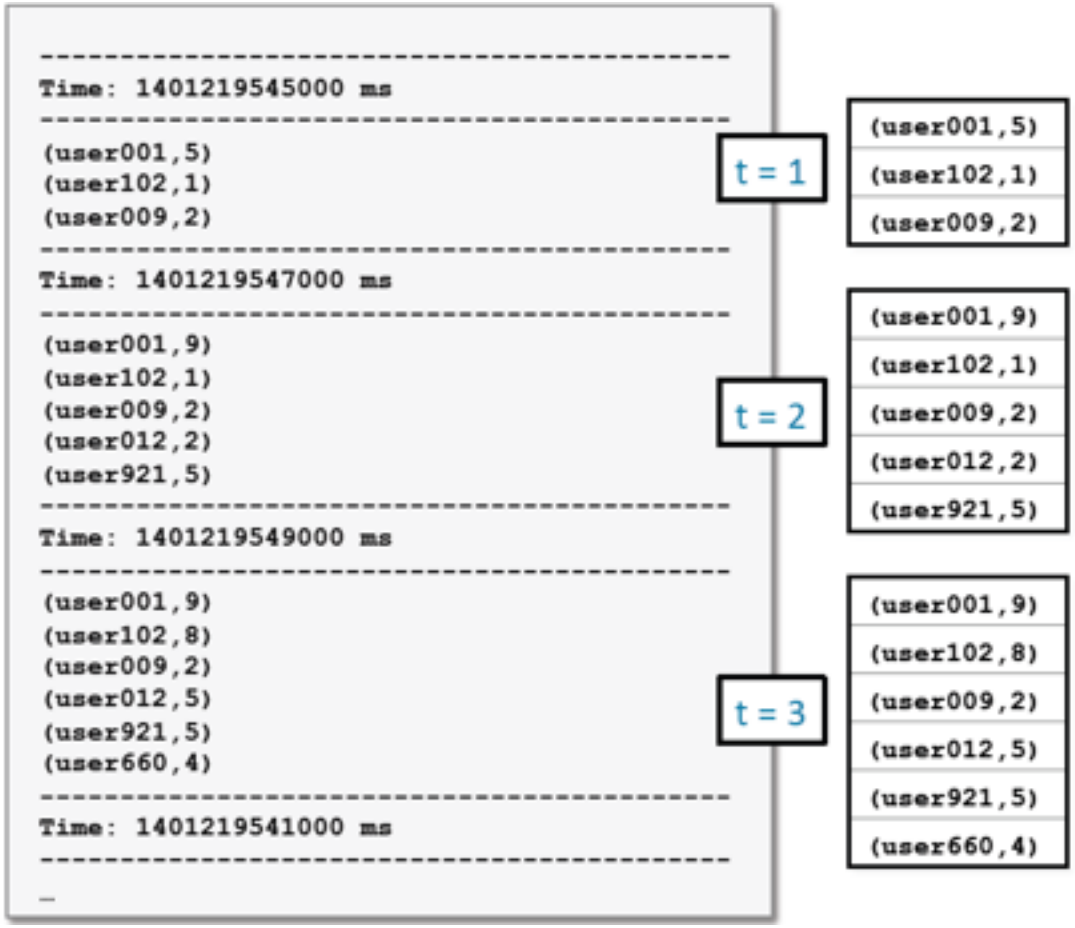


Total User Request Count – Update Function

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

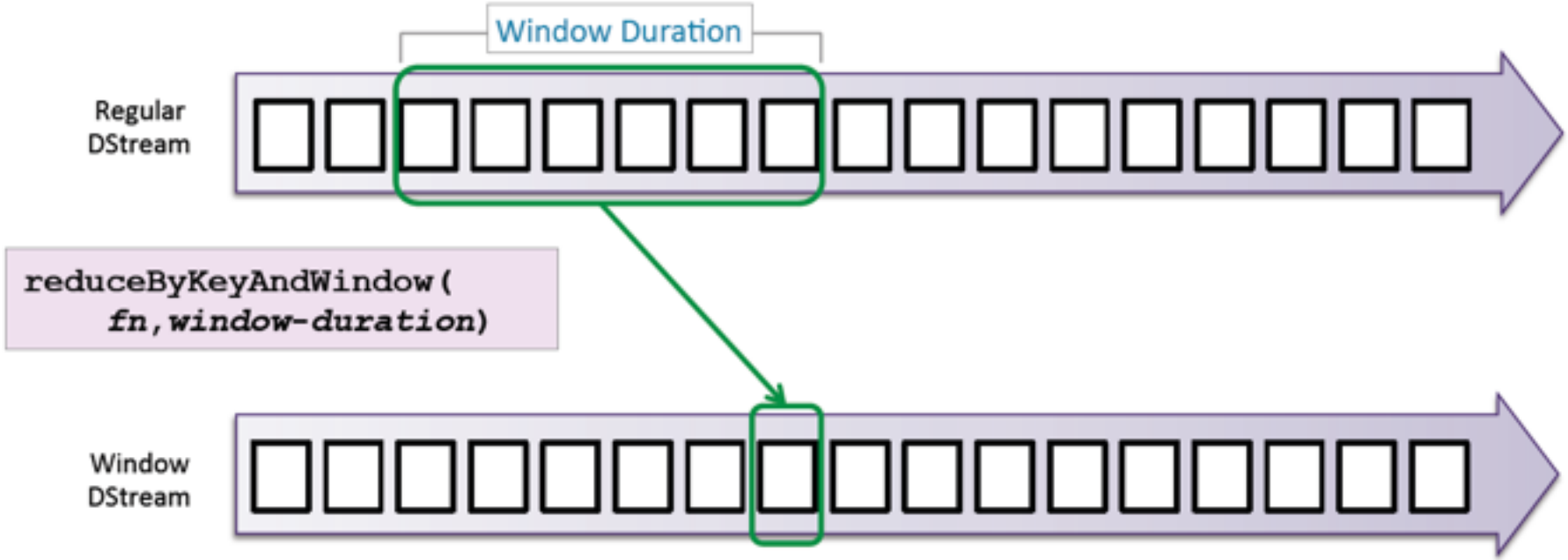


Maintaining State – Output



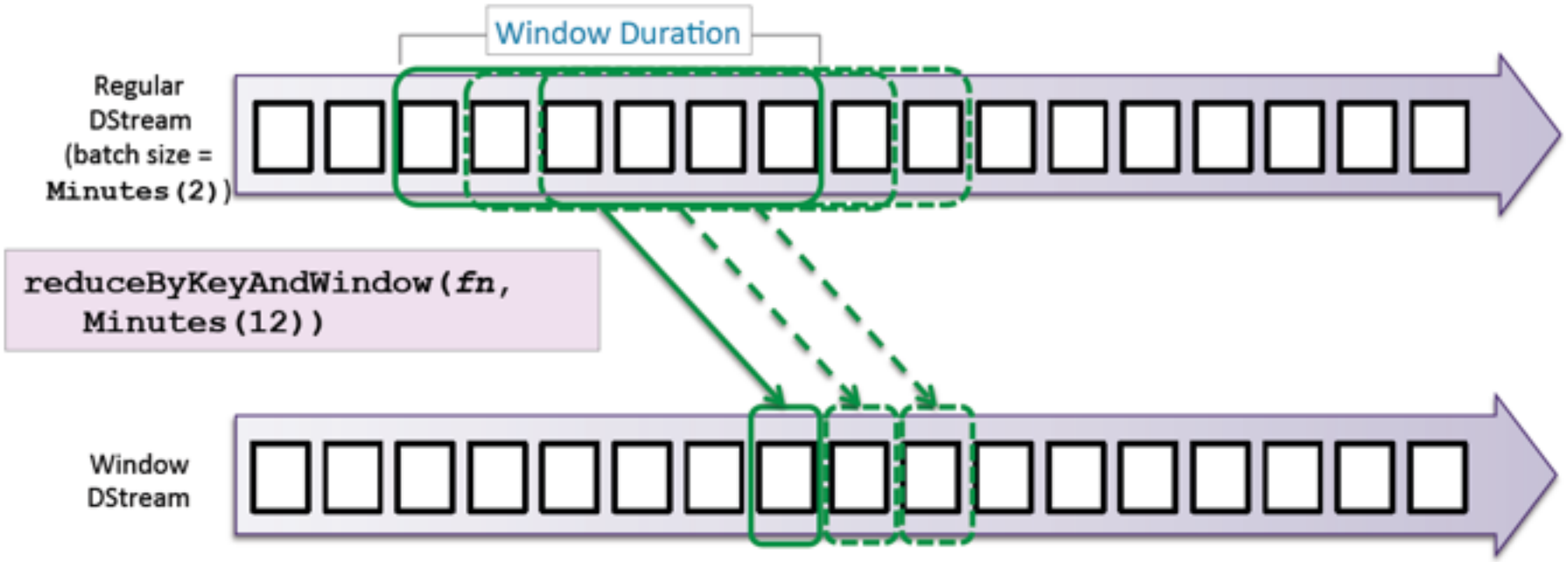
Sliding Window Operations

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



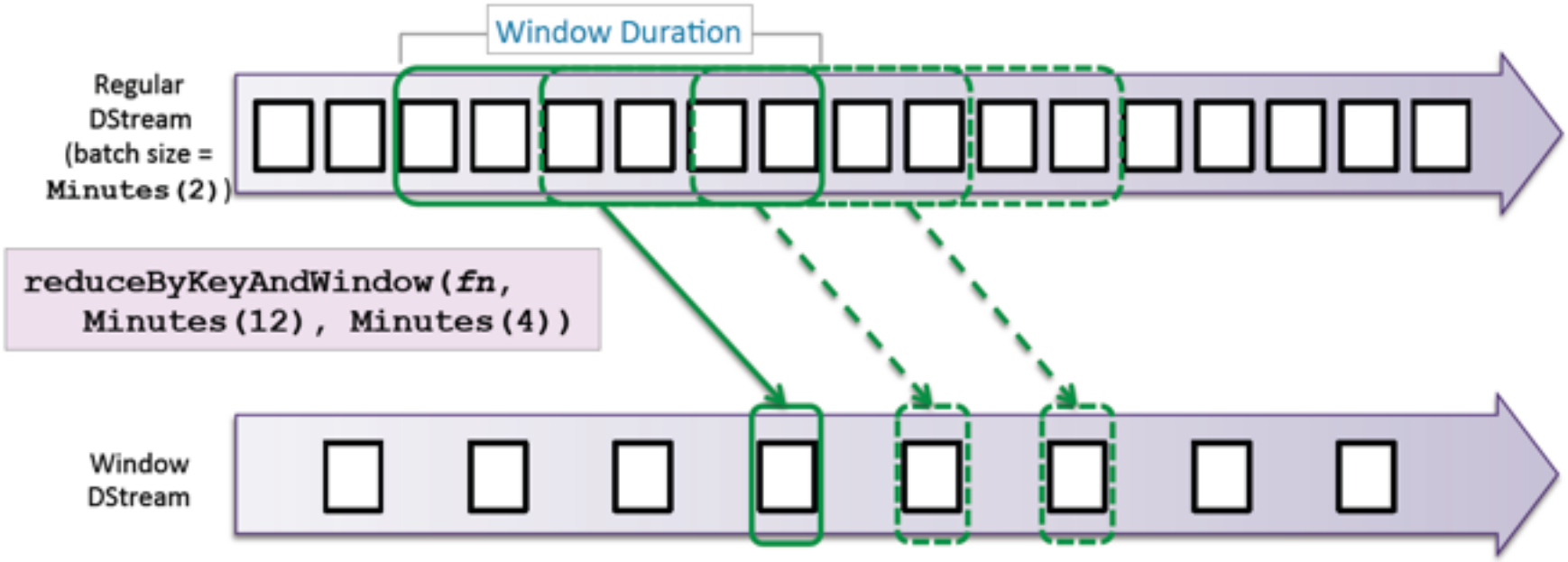
Sliding Window Operations

- 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

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

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

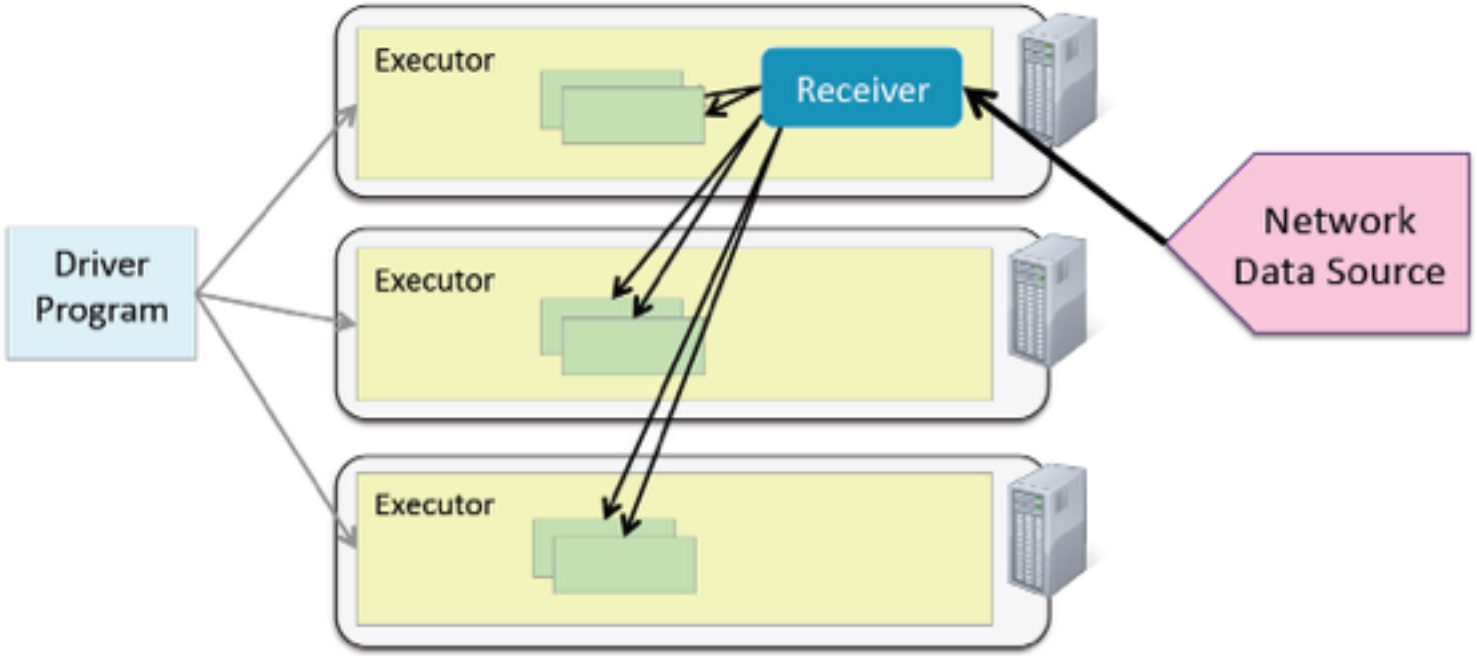
Sort and print the top users for every RDD (every 30 seconds)

Special Considerations for Streaming Applications

- Spark Streaming applications are by definition long-running
  - Require some different approaches than typical Spark applications
- Metadata accumulates over timme
  - 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

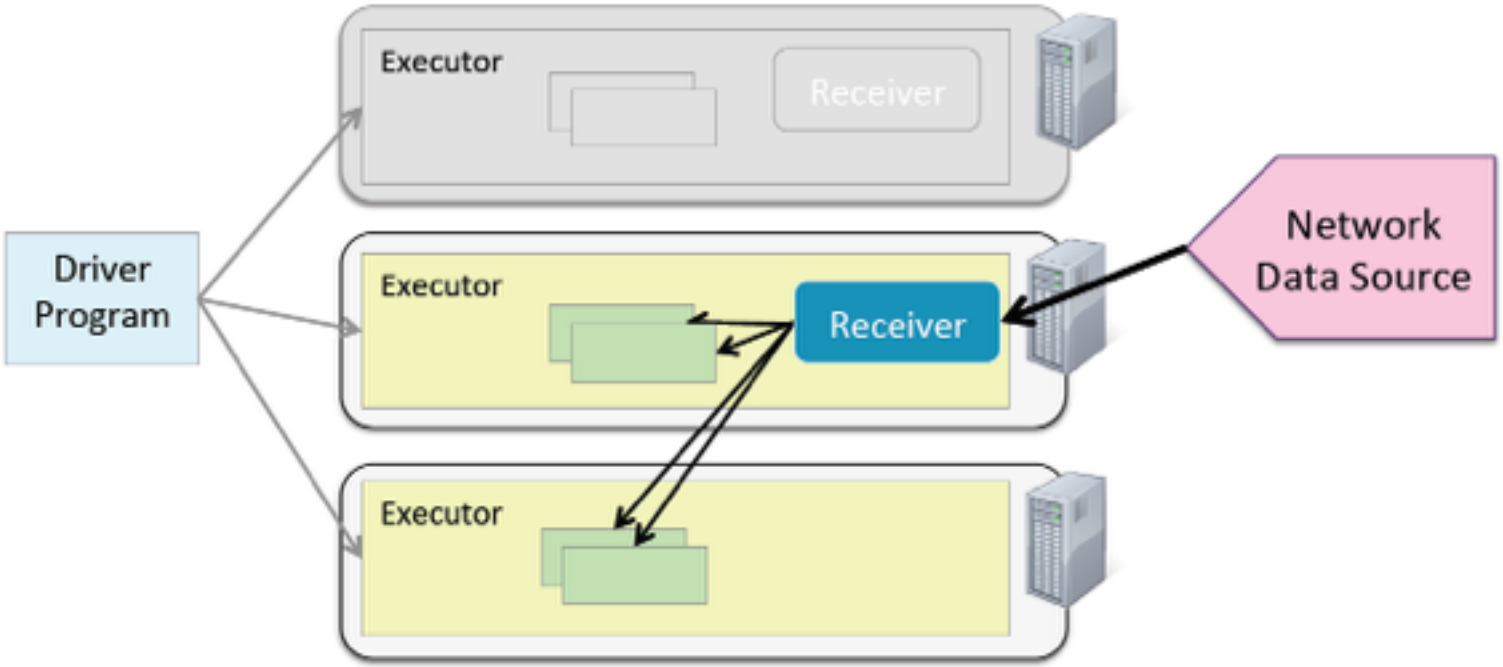
Spark Fault Tolerance

- **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**

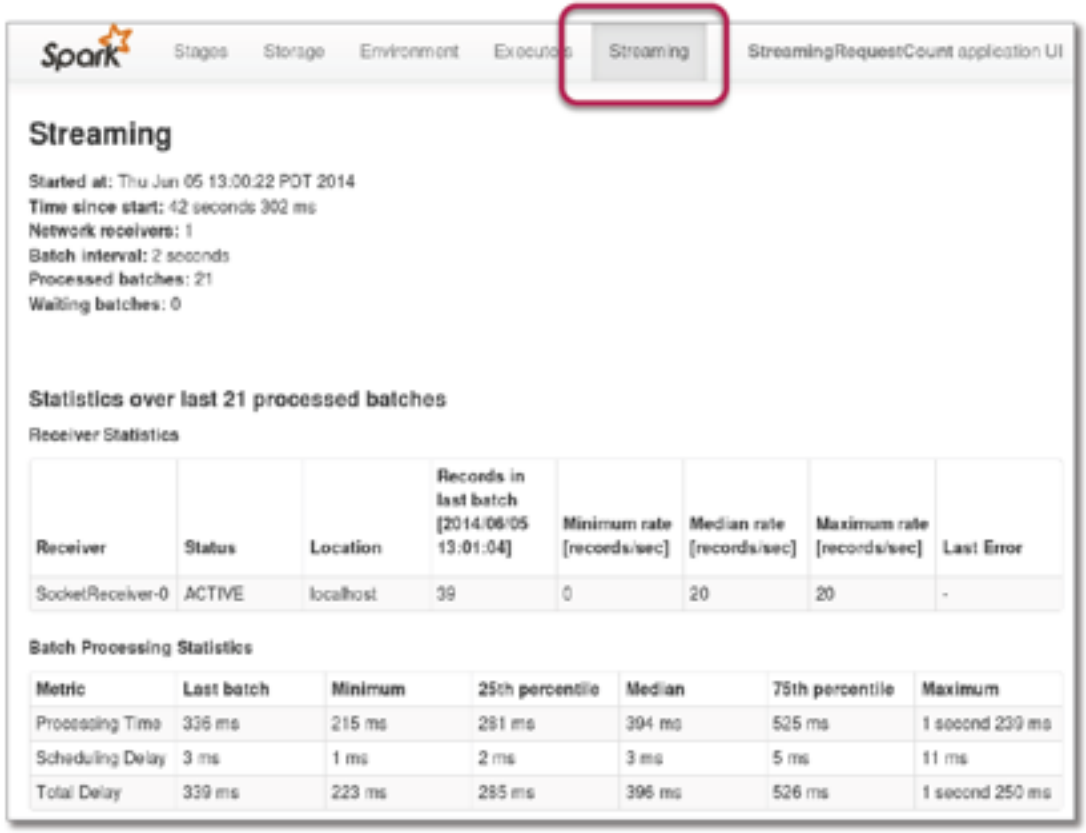


Building and Running Spark Streaming Applications

- **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

HANDS-ON EXERCISE: WRITING A SPARK STREAMING APPLICATION

Chapter 10

COMMON PATTERNS IN SPARK PROGRAMMING

Common Spark Use Cases

- **Spark is especially useful when working with any combination of:**
  - **Large amounts of data**
    - **Distributed storage**
  - **Intensive computations**
    - **Distributed computing**
  - **Iterative algorithms**
    - **In-memory processing and pipelining**

Common Spark Use Cases

- Examples
  - Risk analysis
    - "How likely is this borrower to pay back a loan?
  - Recommendations
    - "Which products will this customer enjoy?"
  - Predictions
    - "How can we prevent service outages instead of simply reacting to them?"
  - Classification
    - "How can we tell which mail is spam and which is legitimate?"

"PageRank"

- PageRank gives web pages a ranking score based on links from other pages
  - Higher scores given for more links, and links from other high ranking pages
- Why do we care?
  - PageRank is a classic example of big data analysis (like WordCount)
  - Lots of data- needs an algorithm that is distributable and scalable
  - Iterative - the more iterations, the better than answer

Graph Analytics

- Many data analytics problems work with “data parallel” algorithms
  - Records can be processed independently of each other
  - Very well suited to parallelizing
- Some problems focus on the relationships between the individual data items. For example:
  - Social networks
  - Web page hyperlinks
  - Roadmaps
- These relationships can be represented by graphs
  - Requires “graph parallel” algorithms

Graph Analysis Challenges at Scale

- Graph Creation
  - Extracting relationship information from a data source
    - For example, extracting links from web pages
- Graph Representation
  - e.g., adjacency lists in a table
- Graph Analysis
  - Inherently iterative, hard to parallelize
  - This is the focus of specialized libraries like Pregel, GraphLab
- Post analysis processing
  - e.g., incorporating product recommendations into a retail site

Graph Analysis in Spark

- Spark is very well suited to graph parallel algorithms
- GraphX
  - UC Berkeley AMPLab project on top of Spark
  - Unifies optimized graph computation with Spark's fast data parallelism and interactive abilities

Machine Learning

- Most programs tell computers exactly what to do
  - Database transactions and queries
  - Controllers
    - Phone systems, manufacturing processes, transport, weaponry, etc.
  - Media delivery
  - Simple search
  - Social systems
    - Chat, blogs, email, etc.
- An alternative technique is to have computers learn what to do
- Machine Learning refers to programs that leverage collected data to drive future program behavior
- This represents another major opportunity to gain value from data

The ‘Three Cs’

- **Machine Learning is an active area of research and new applications**
- **There are three well-established categories of techniques for exploiting data:**
  - **Collaborative filtering (recommendations)**
  - **Clustering**
  - **Classification**

Collaborative Filtering

- Collaborative Filtering is a technique for recommendations
- Example application: given people who each like certain books, learn to suggest what someone may like in the future based on what they already like
- Helps users navigate data by expanding to topics that have affinity with their established interests
- Collaborative Filtering algorithms are agnostic to the different types of data items involved
  - Useful in many different domains

Clustering

- Clustering algorithms discover structure in collections of data
  - Where no formal structure previously existed
- They discover what clusters, or groupings, naturally occur in data
- Examples:
  - Finding related news articles
  - Computer vision (groups of pixels that cohere into objects)

Classification

- The previous two techniques are considered ‘unsupervised’ learning
  - The algorithm discovers groups or recommendations itself
- Classification is a form of ‘supervised’ learning
- A classification system takes a set of data records with known labels
  - Learns how to label new records based on that information
- Example:
  - Given a set of e-mails identified as spam/not spam, label new e-mails as spam/not spam
  - Given tumors identified as benign or malignant, classify new tumors

Machine Learning Challenges

- **Highly computation intensive and iterative**
- **Many traditional numerical processing systems do not scale to very large datasets**
  - **e.g., MatLab**

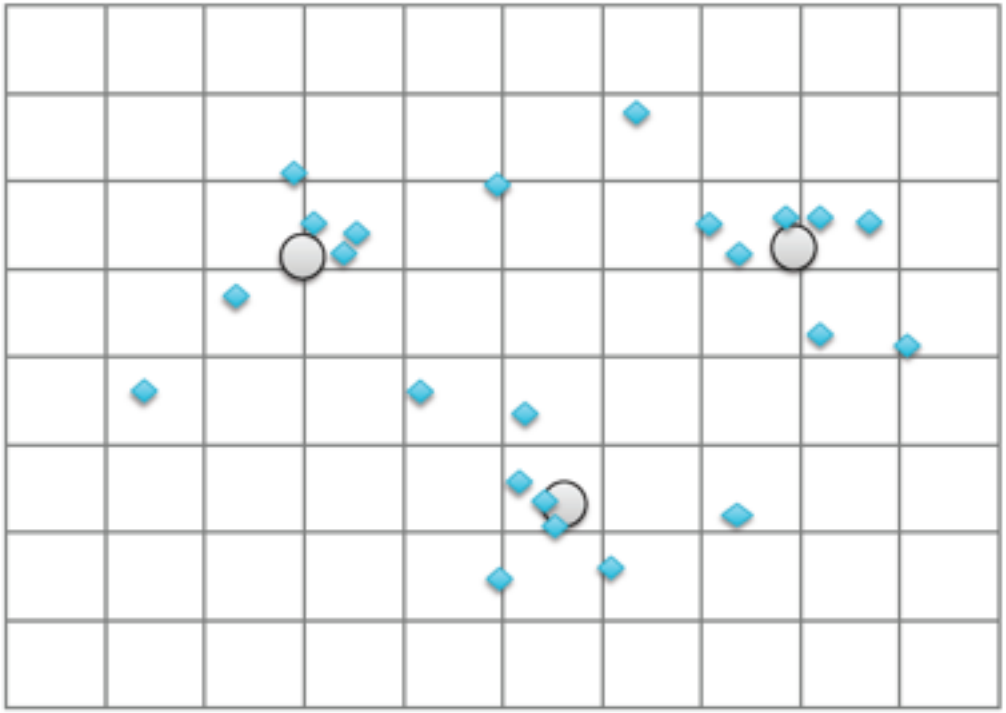
MLlib: Machine Learning on Spark

- MLlib is part of Apache Spark
- Includes many common ML functions
  - ALS (alternating least squares)
  - k-means
  - Logistic Regression
  - Linear Regression
  - Gradient Descent
- Still a ‘work in progress’

K-means Clustering

- **K-means Clustering**
  - **A common iterative algorithm used in graph analysis and machine learning**
  - **You will implement a simplified version in the Hands-On Exercises**

Approximate k-means Clustering



1. Choose K random points as starting centers
2. Find all points closest to each center
3. Find the center (mean) of each cluster
4. If the centers changed by more Than c, iterate again

…

5. Close enough!

Hands-On Exercise: Iterative Processing in Spark

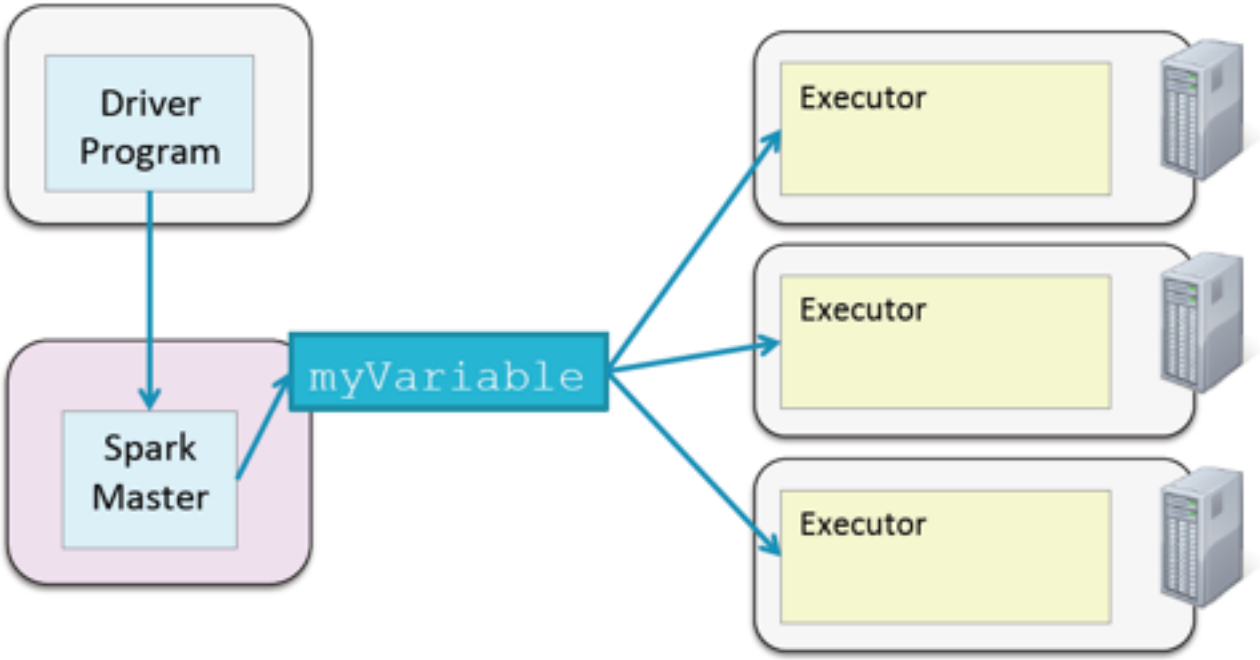
HANDS-ON EXERCISE

Chapter 11

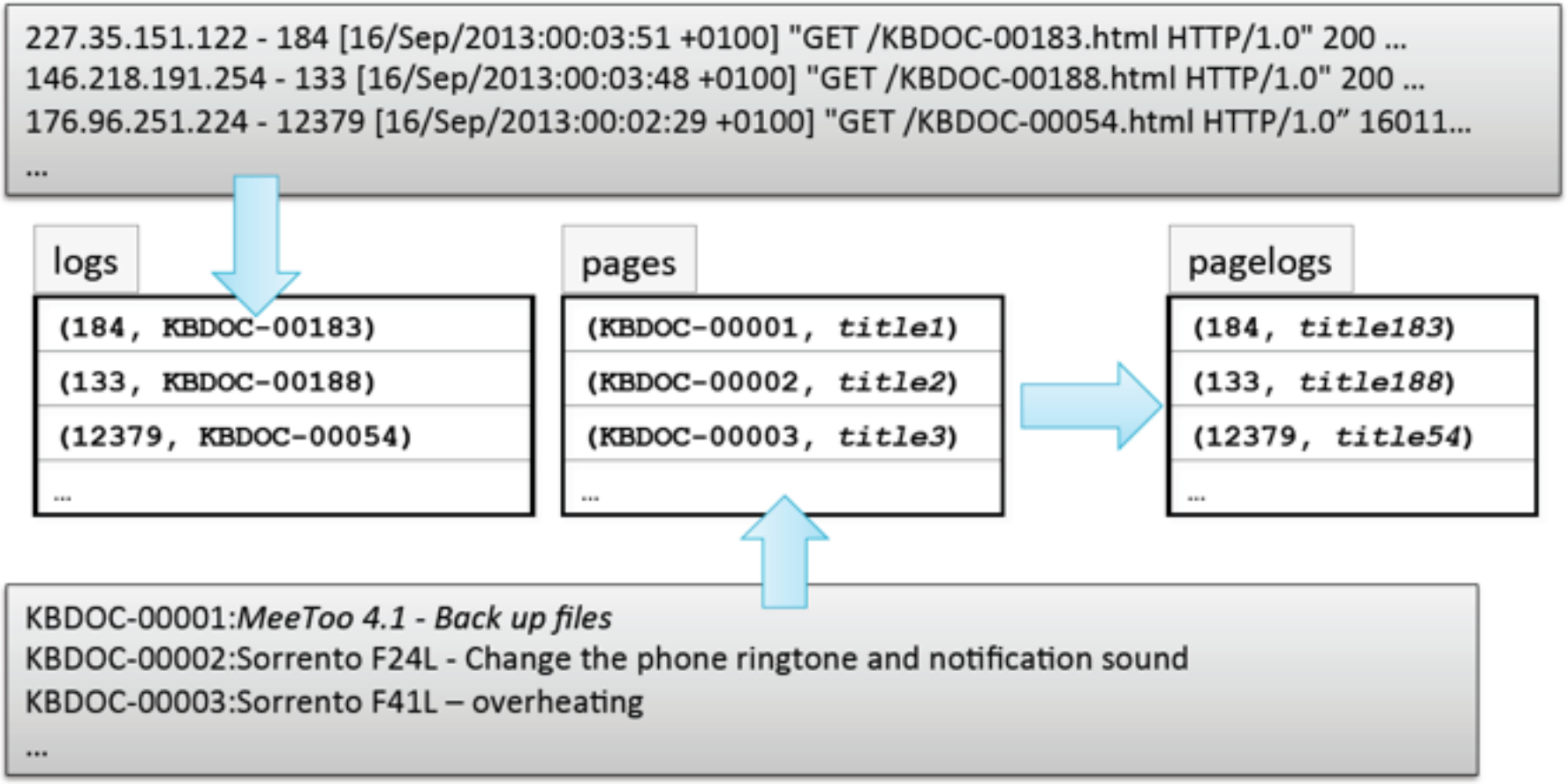
IMPROVING SPARK PERFORMANCE

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**

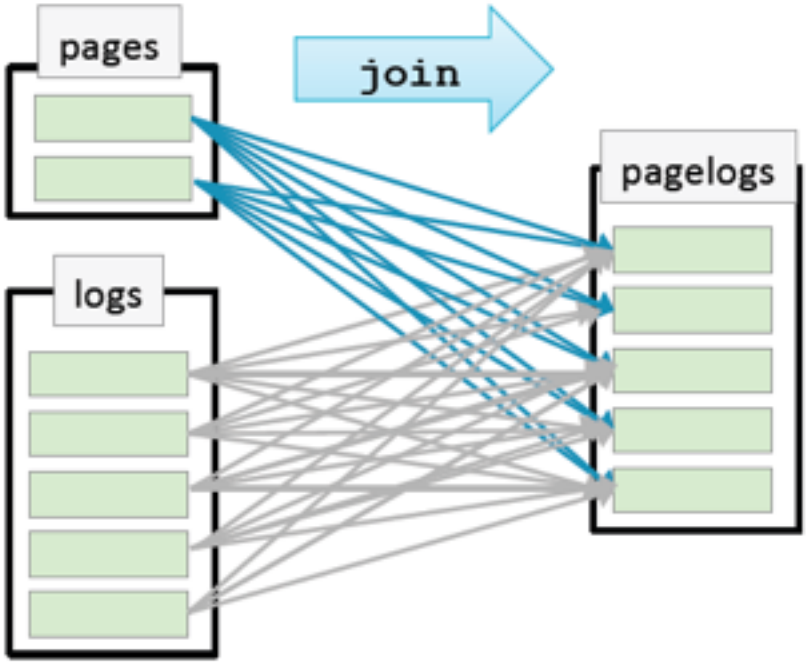


Match User IDs with Requested Page Titles



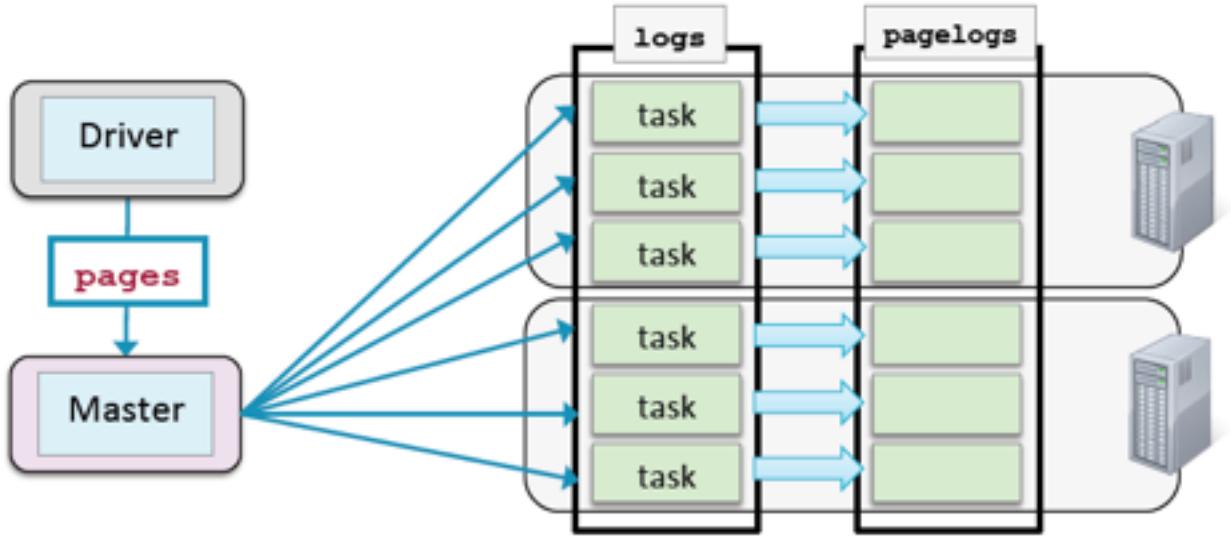
Join a Web Server Log with Page Titles

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



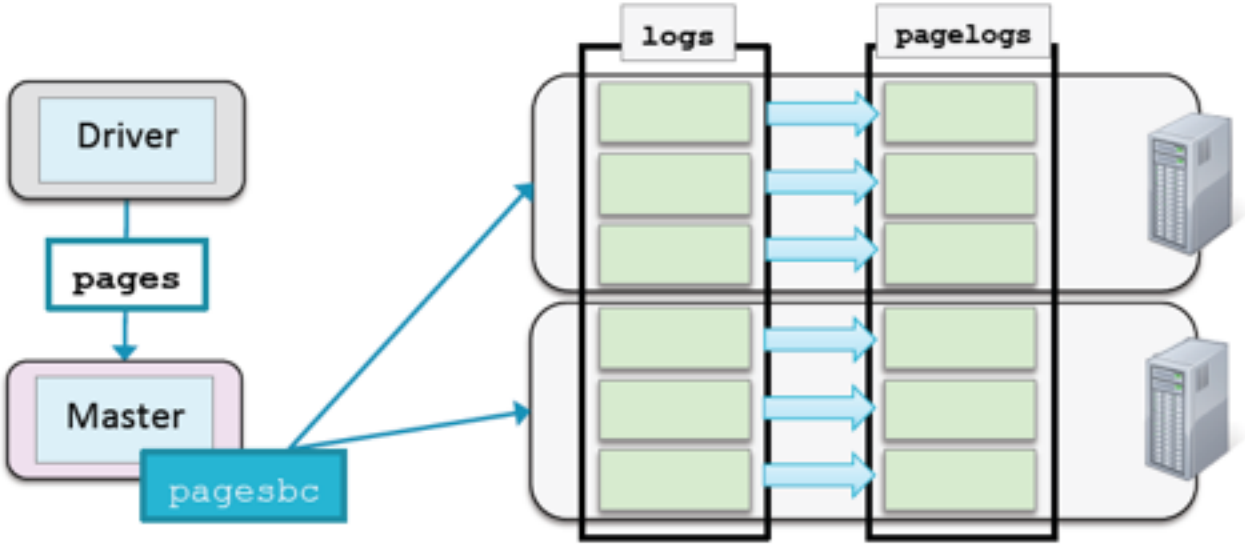
Pass a Small Table as a Parameter

```
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

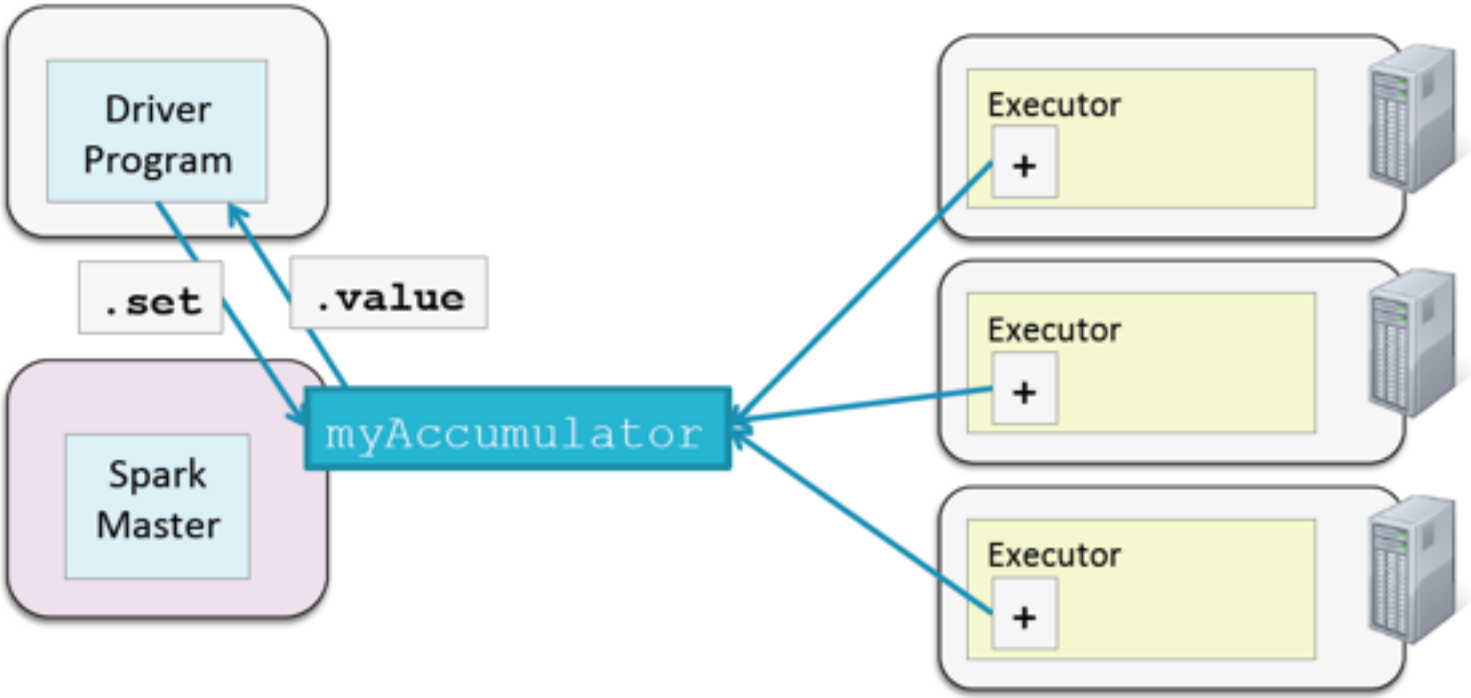
- **Why use Broadcast variables?**
  - **Use to minimize transfer of data over the network, which is usually the biggest bottleneck**
  - **Spark Broadcast variables are distributed to worker nodes using a very efficient peer-to-peer algorithm**

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

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

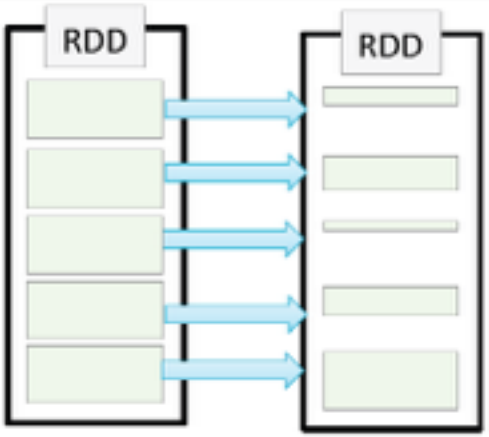
- Use Kryo serialization for Scala and Java
  - To enable, set spark.serializer = spark.KryoSerializer
- To enable Kryo for your custom classes
  - Create a KryoRegistrar class and set spark.kryo.registrator=MyRegistrator
  - Register your classes with Kryo

```
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**

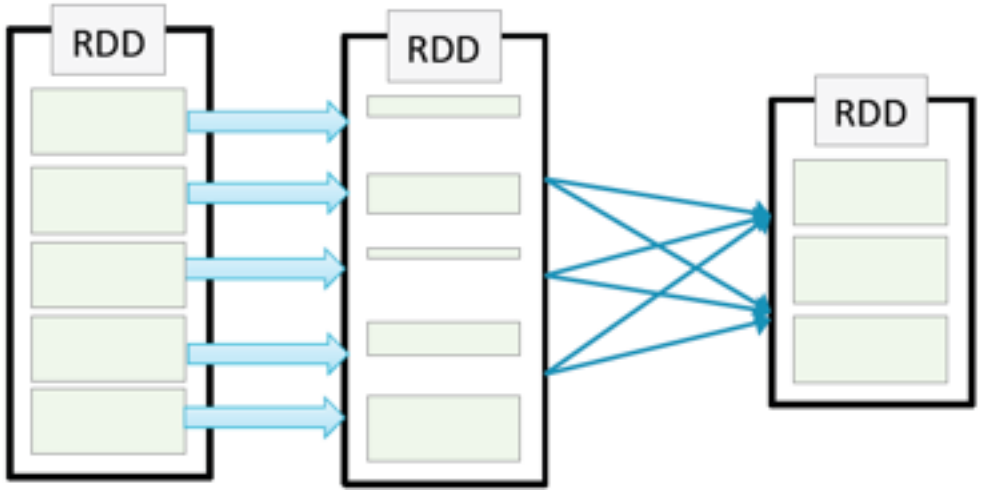
```
sc.textFile(file) \
  .filter(lambda s: s.startswith('I')) \
  .map(lambda s: \
     (s.split()[0],(s.split()[1],s.split()[2])))
```



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**

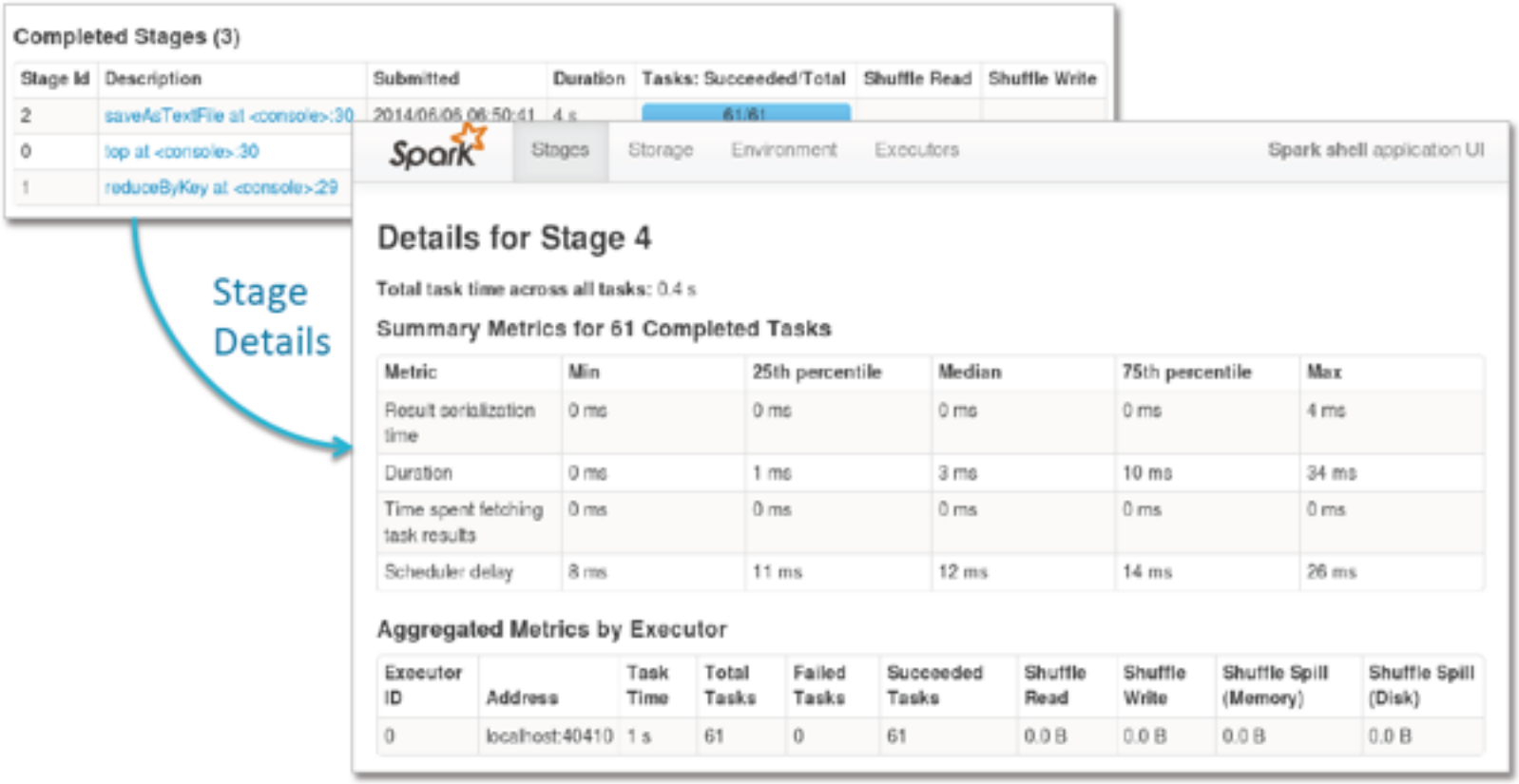
```
hashmap = some_massive_hash_map()
bhashmap = sc.broadcast(hashmap)
…
myrdd.map(lambda x: bhashmap(x)).countByValue()
```

  - 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

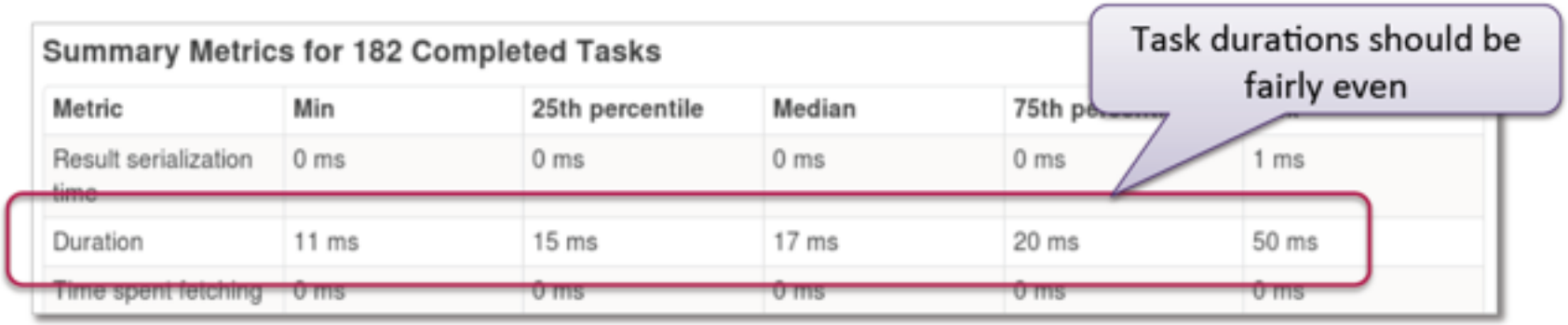
Metric	Min	25th percentile	Median	75th percentile	Max
Result serialization time	0 ms	0 ms	0 ms	0 ms	4 ms
Duration	20 ms	23 ms	30 ms	44 ms	0.3 s
Time spent fetching task results	0 ms	0 ms	0 ms	0 ms	0 ms
Scheduler delay	11 s	11 s	12 s	12 s	15 s

Task Execution Issues

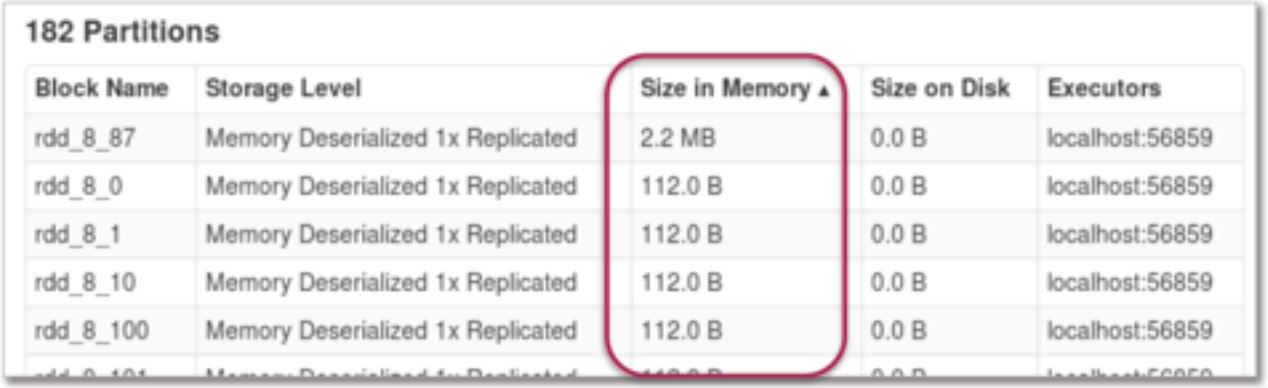
- **Task execution taking too long?**
  - **Are there tasks with a very high per-record overhead?**
    - **e.g., mydata.map (dbLookup)**
    - **Each lookup call opens a connection to the DB, reads, and closes**
  - **Try mapPartitions**

Task Execution Issues

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

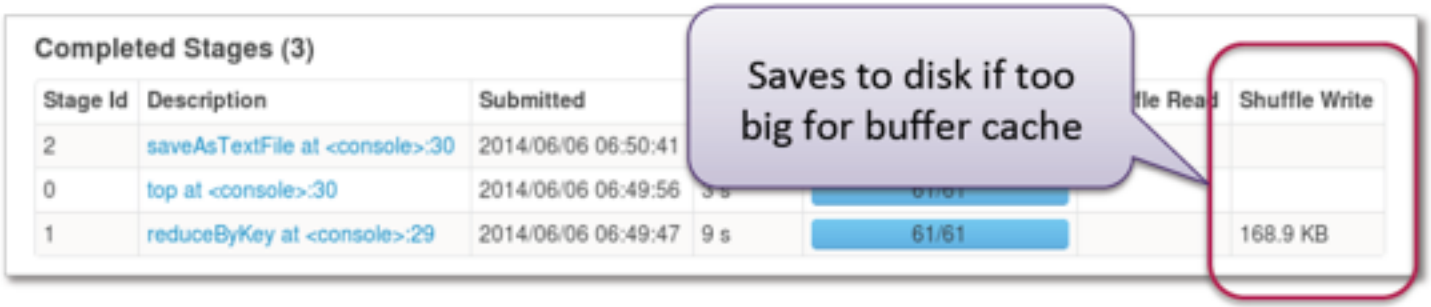


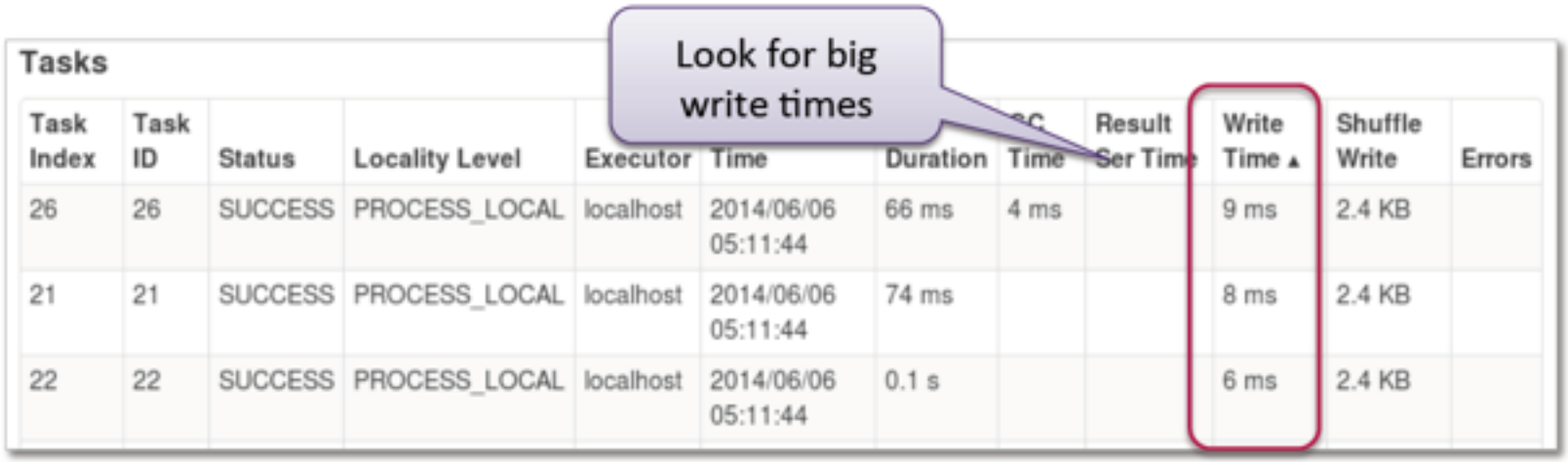
Example: empty partitions due to filtering



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**





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

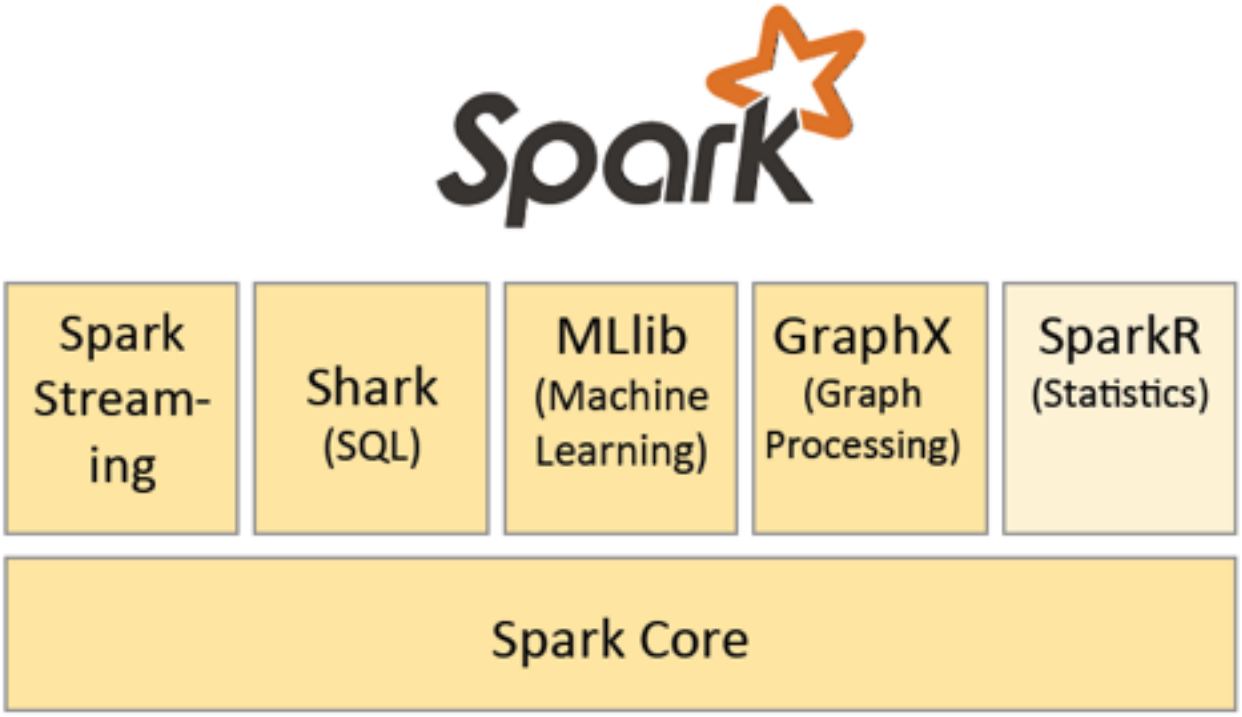
- Spark supports integration with other performance tools
  - Configurable metrics system built on the Coda Hale Metrics Library
  - Metrics can be
    - Saved to files
    - Output to the console
    - Viewed in the JMX console
    - Sent to reporting tools like Graphite or Ganglia

Chapter 12

SPARK, HADOOP, AND THE ENTERPRISE DATA CENTER

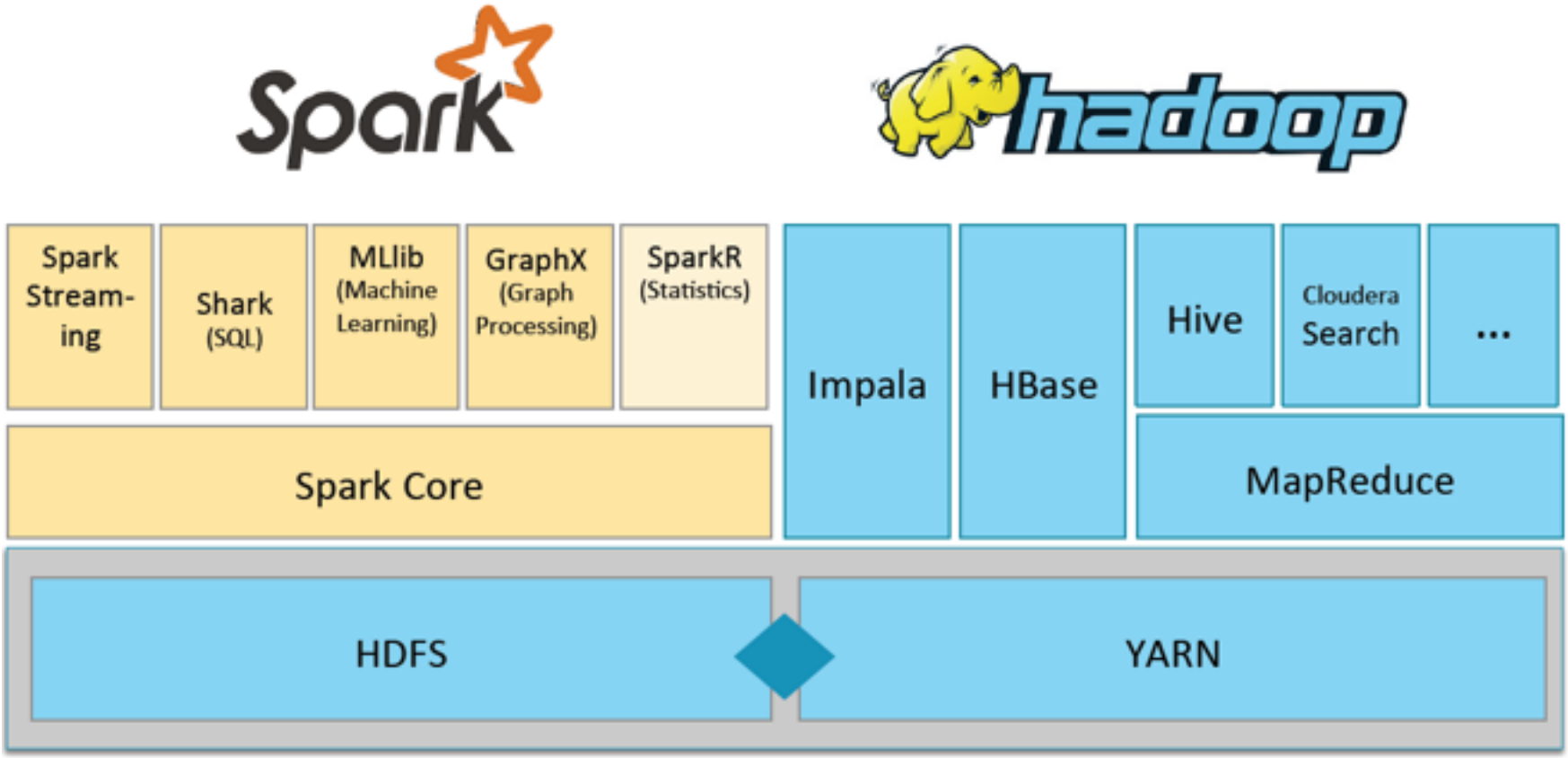
The Spark Stack

- **In addition to the core Spark engine, there are an ever-growing number of related projects**
- **Sometimes called the Berkeley Data Analytics Stack (BDAS)**



Spark and Hadoop

- **Spark was created to complement, not replace, Hadoop**

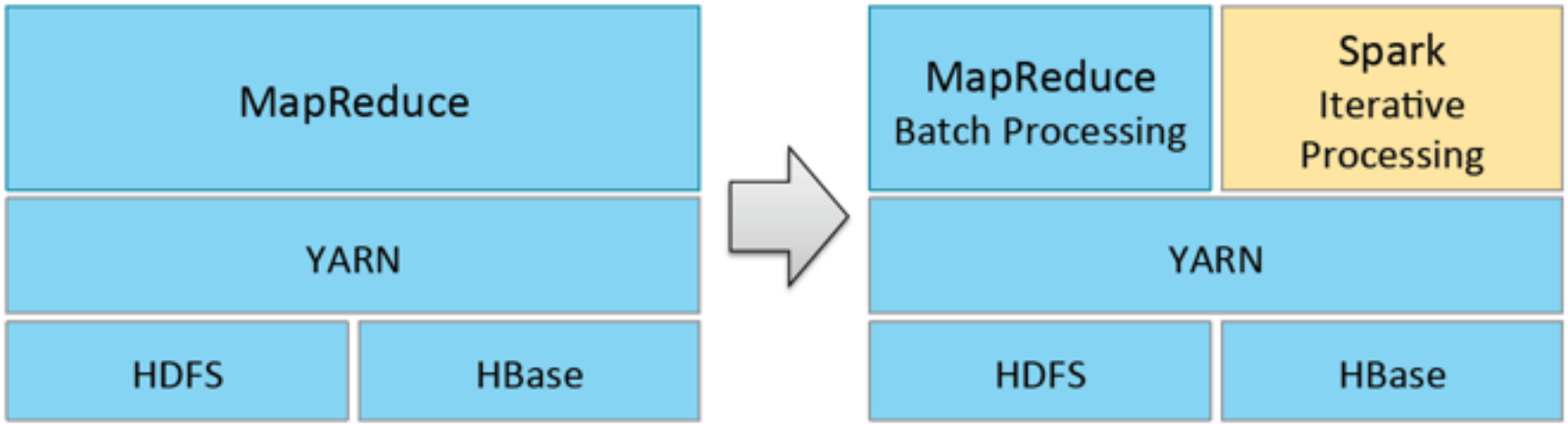


Spark and Hadoop (2)

- Spark uses HDFS
  - Can use any Hadoop data source
  - Uses Hadoop InputFormats and OutputFormats
    - This means it can manipulate e.g., Avro files and SequenceFiles
- Spark runs on YARN
  - Can run on the same cluster with MapReduce jobs, Impala, etc.
- Spark works with the Hadoop ecosystem
  - Flume
  - Sqoop
  - HBase

Yahoo

- Example use-case: Yahoo is a major user of Hadoop
  - Uses Hadoop for personalization, collaborative filtering, ad analytics…
- MapReduce couldn't keep up
  - Highly iterative machine learning algorithms
- Moved iterative processing to Spark

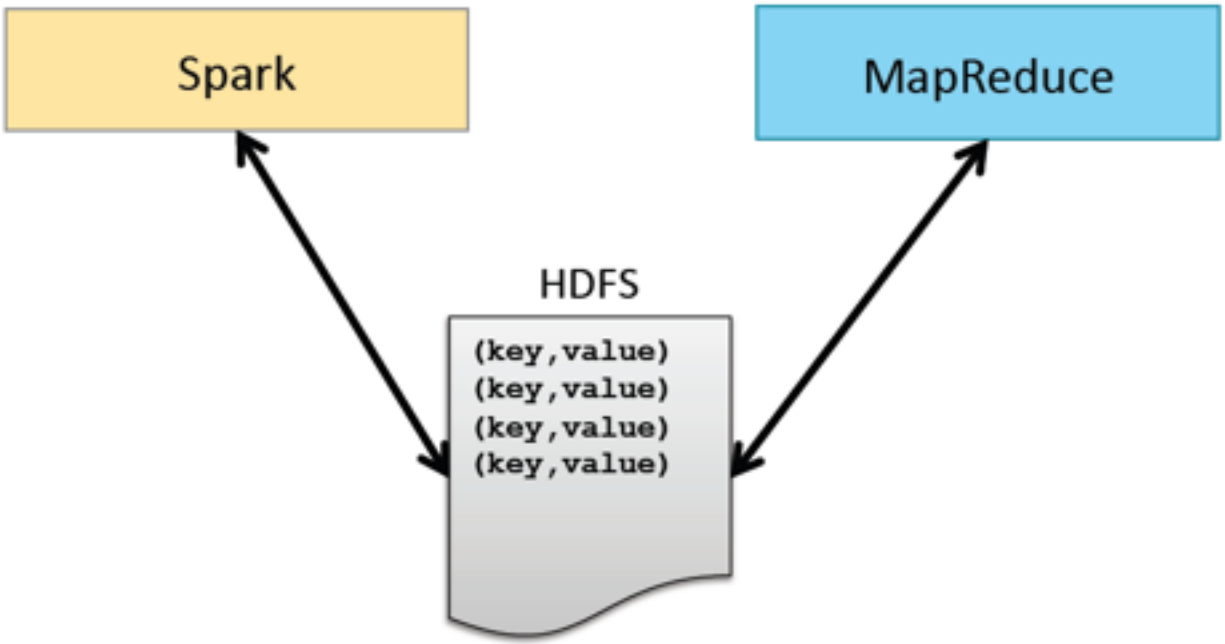


Spark vs Hadoop MapReduce

- Hadoop MapReduce
  - Widely used, huge investment already made
  - Supports and supported by many complementary tools
  - Mature, stable, well-tested technology
  - Skilled developers available
- Spark
  - Flexible
  - Elegant
  - Fast
  - Changing rapidly

Sharing Data Between Spark and MapReduce Jobs

- Apache Avro is a binary file format for saving datasets
- Hadoop SequenceFiles are similar; used by many existing Hadoop data centers
- Both are supported by Spark (Scala now, Python soon)



The Hadoop Ecosystem

- In addition to HDFS and MapReduce, the Hadoop Ecosystem includes many additional components
- Some that may be of particular interest to Spark developers
  - Data Storage: HBase
  - Data Analysis: Hive and Impala
  - Data Integration: Flume and Sqoop

Data Storage: Hbase – The Hadoop Database

- HBase: database layered on top of HDFS
  - Provides interactive access to data
- Stores massive amounts of data
  - Petabytes+
- High throughput
  - Thousands of writes per second (per node)
- Handles sparse data well
  - No wasted space for a row with empty columns
- Limited access model
  - Optimized for lookup of a row by key rather than full queries
    - No transactions: single row operations only

Data Analysis: Hive

- **What is Hive?**
  - Open source Apache project
  - Built on Hadoop MapReduce
  - **HiveQL**: An SQL-like interface to Hadoop

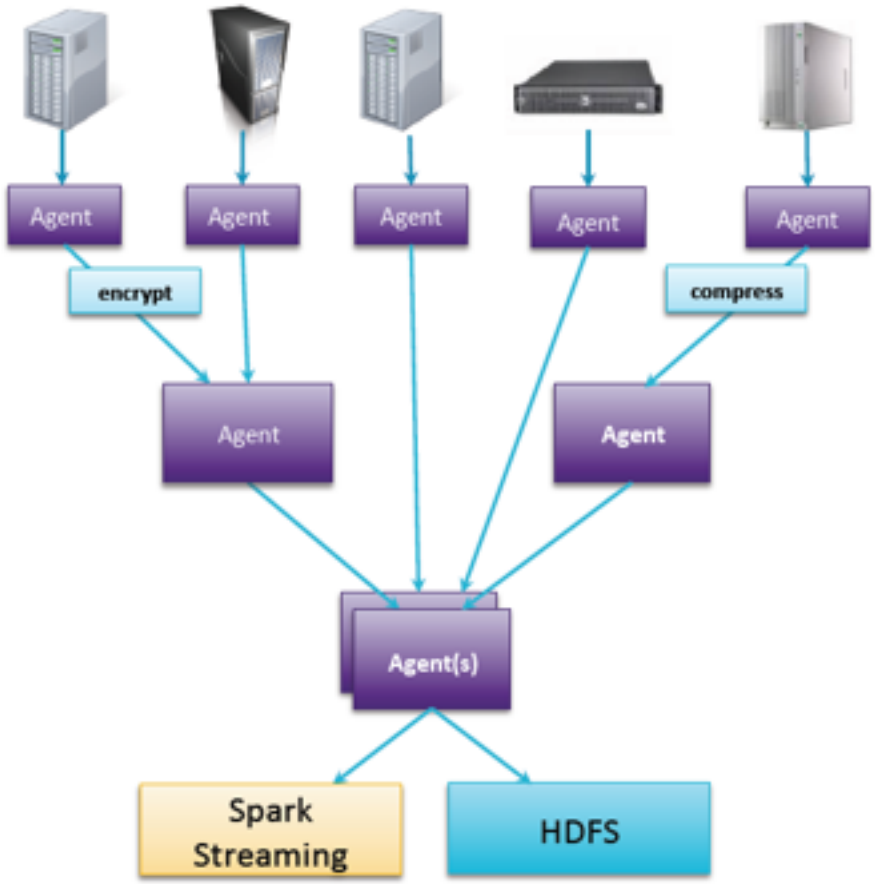
```
SELECT * FROM purchases WHERE price > 10000 ORDER BY
storeid
```

Data Analysis: Flume

- What is Flume?
  - A service to move large amounts of data in real
  - Example: storing log files in HDFS
- Flume is
  - Distributed
  - Reliable and available
  - Horizontally scalable
  - Extensible
- Spark Streaming is integrated with Flume

Data Analysis: Flume

- Collect data as it is produced
  - Files, syslogs, stdout or custom source
- Process in place
  - e.g., encrypt, compress
- Pre-process data before storing
  - e.g., transform, scrub, enrich
- Write in parallel
  - Scalable throughput
- Store in any format
  - Text, compressed, binary, or custom sink



Data Integration: Sqoop – SQL to Hadoop

- **Typical scenario: data stored in an RDBMS is needed in a Spark application**
  - **Lookup tables**

- **Possible to read directly from an RDBMS in your Spark application**
  - Can lead to the equivalent of a distributed denial of service (DDoS) attack on your RDBMS
  - In practice – don't do it!

- **Better idea: use Sqoop to import the data into HDFS beforehand**

Data Integration: Sqoop – SQL to Hadoop

- Sqoop: open source tool
  - Now a top-level Apache Software Foundation project
- Imports tables from an RDBMS into HDFS
  - Just one table, all tables, or portions of a table
  - Uses MapReduce to actually import the data
- Uses a JDBC interface
  - Works with virtually any JDBC-compatable database
- Imports data to HDFS as delimited text files or SequenceFiles
  - Default is comma-delimited text files
- Can be used for incremental data imports
  - First import retrieves all rows in a table
  - Subsequent imports retrieve just rows created since the last import

Sqoop: Basic Syntax

- **Standard syntax:**

```
$ sqoop tool-name [tool-options]
```

- **Tools include:**

```
import
import-all-tables
list-tables
```

- **Options include:**

```
--connect
--username
--password
```

Sqoop: Example

- **Example: import a table called employees from a database called**

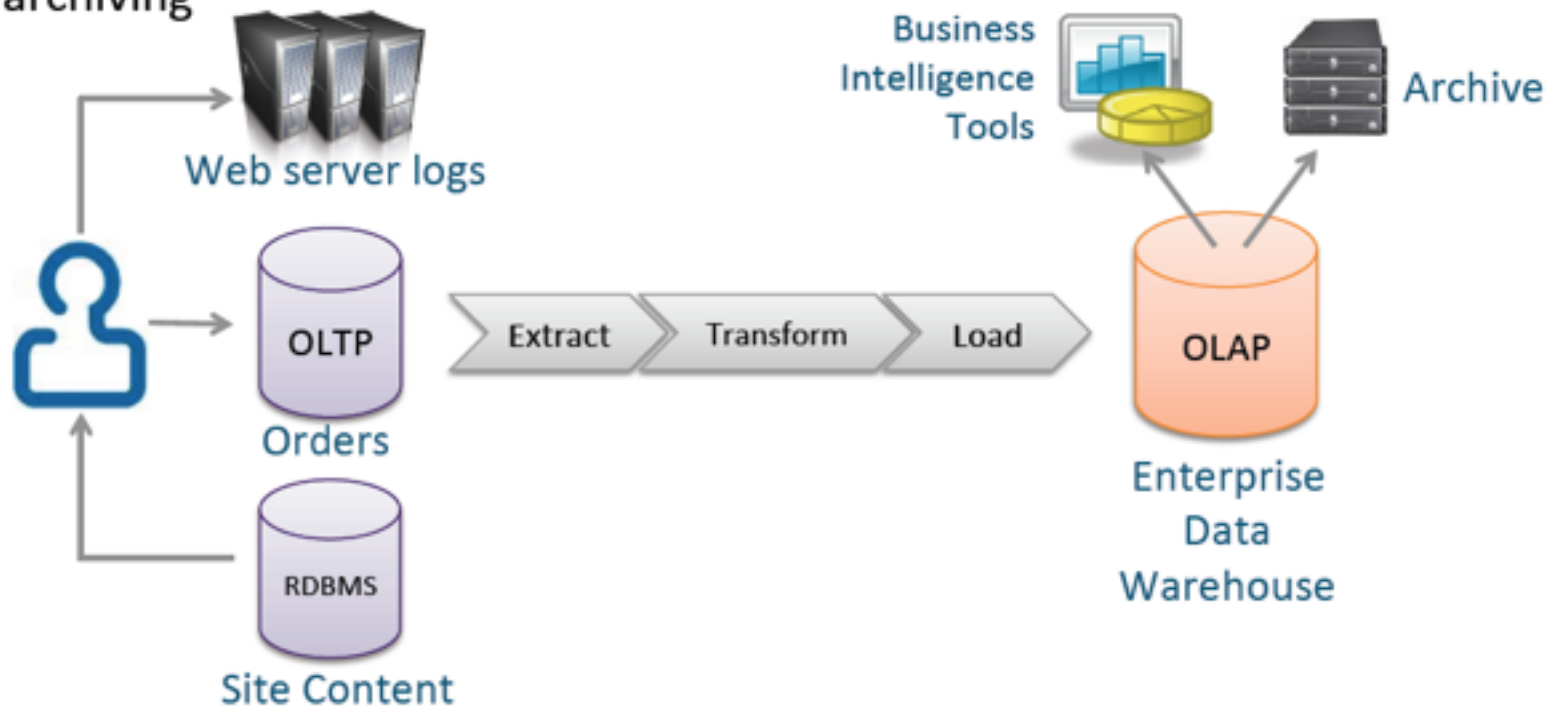
```
$ sqoop import --username fred --password derf \
    --connect jdbc:mysql://database.example.com/personnel \
    --table employees
```

- **Example: as above, but only records with an ID greater than 1000**

```
$ sqoop import --username fred --password derf \
    --connect jdbc:mysql://database.example.com/personnel \
    --table employees \
    --where "id > 1000"
```

Typical RDBMS Scenario

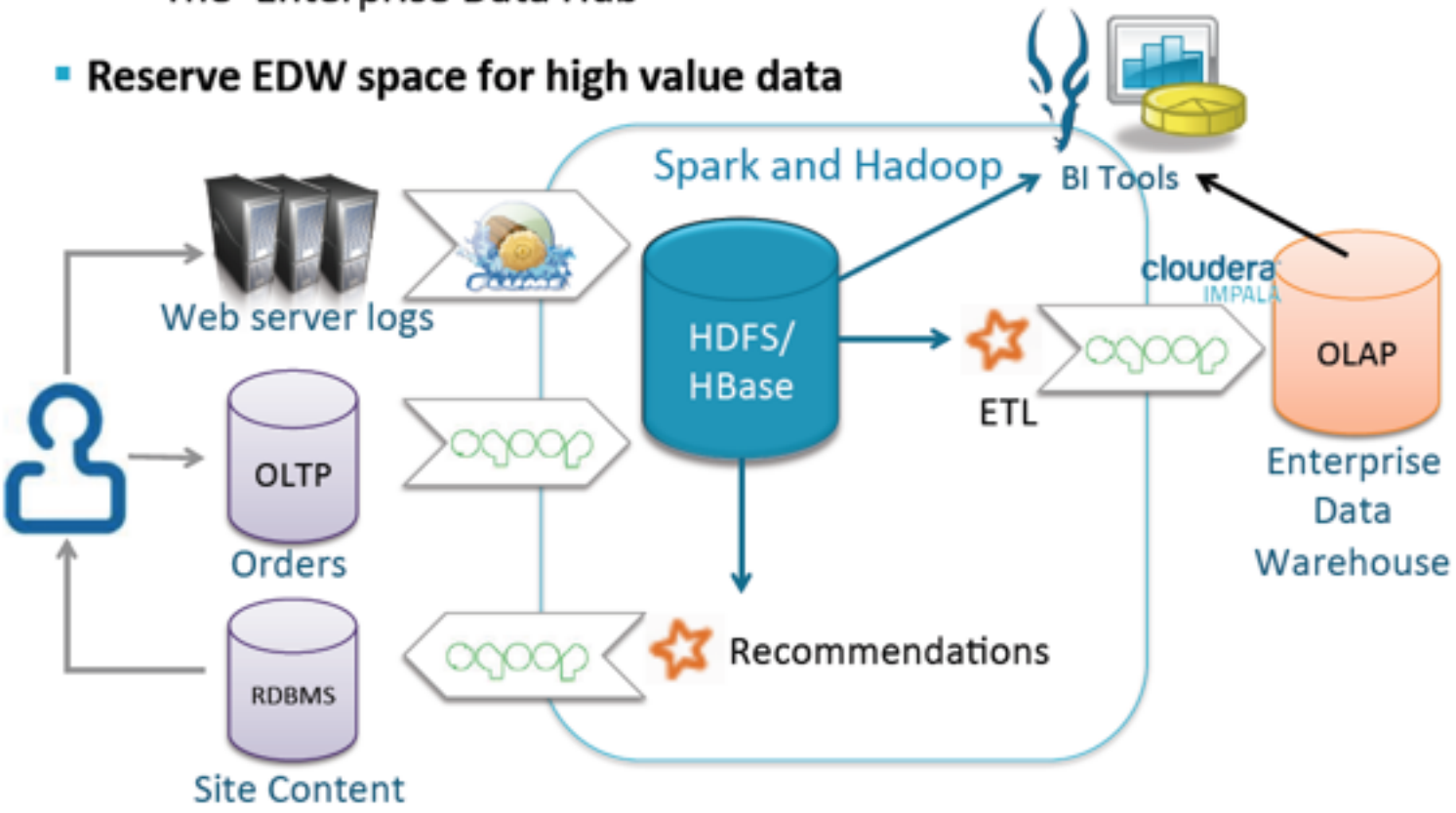
- **Typical scenario:**
  - Interactive RDBMS serves queries from a web site
  - Data is extracted and loaded into a data warehouse for processing and archiving



OLTP: Online Transaction Processing
OLAP: Online Analytical Processing

Using Spark and Hadoop to Augment Existing Databases

- **With Spark and Hadoop you can store and process *all* your data**
  - The 'Enterprise Data Hub'
- **Reserve EDW space for high value data**



Benefits of Spark and Hadoop Over RDBMSs

- **Processing power scales with data storage**
  - **As you add more nodes for storage, you get more processing power 'for free'**
- **Views do not need prematerialization**
  - **Ad-hoc full or partial dataset queries are possible**
- **Total query size can be multiple petabytes**

Traditional High-Performance File Servers

- Enterprise data is often held on large fileservers, such as products from
  - NetApp
  - EMC
- Advantages
  - Fast random access
  - Many concurrent clients
- Disadvantages
  - High cost per terabyte of storage

File Servers and HDFS

- Choice of storage depends on the expected access paberns
  - Sequentially read, append-only data: HDFS
  - Random access: file server
- HDFS can crunch sequential data faster
- Offloading data to HDFS leaves more room on file servers for 'interactive' data
- Use the right tool for the job!

Hands-On Exercise: Importing RDBMS Data Into Spark

HANDS-ON EXERCISE: IMPORTING RDBMS DATA INTO SPARK

THANK YOU!