

# CS595—BIG DATA TECHNOLOGIES

---

Module 06

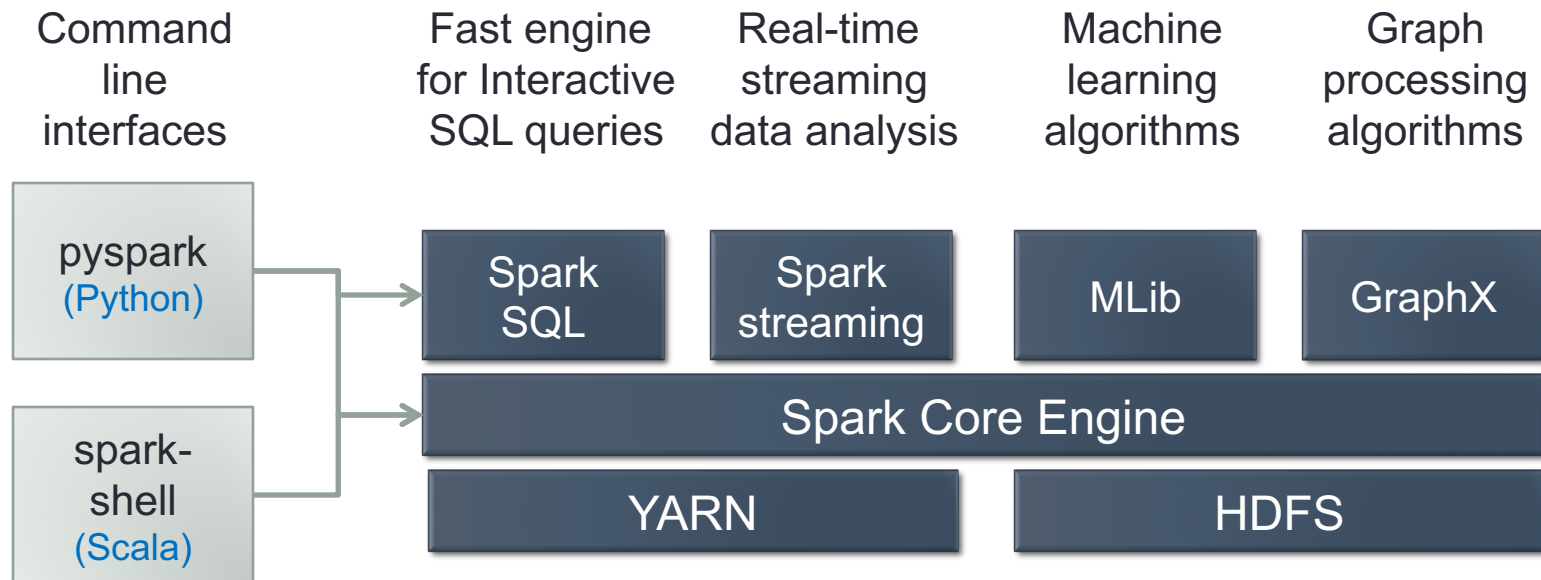
Spark Core

# Apache Spark

- Apache Spark is an open source, Hadoop-compatible, fast and general purpose cluster-computing platform
- It was created at AMPLabs in UC Berkeley as part of Berkeley Data Analytics Stack
- Includes SPARK SQL, SPARK Streaming, MLlib (Machine Learning) and GraphX (graph processing)
- Spark capable to run programs up to 100x faster than Hadoop MapReduce in memory, or 10x faster on disk
- Spark can run on a Hadoop YARN cluster manager, and can read any existing Hadoop data
- Spark itself is written in the language Scala, and runs on the Java Virtual Machine (JVM)

# Apache Spark

## Architecture Landscape



# Apache Spark

## On the speed side...

- Spark extends the MapReduce model to support more types of computations...
- Including interactive queries and stream processing
- Speed is important in processing large datasets...
- It means the difference between exploring data interactively and waiting minutes or hours
- One of the main features Spark offers for speed is the ability to run computations in memory
- But the system is also more efficient than MapReduce for complex applications running on disk
- Spark is capable to run programs up to 100x faster than Hadoop MapReduce in memory or 10x faster on disk

# Apache Spark

## On the general purpose side...

- Spark is designed to cover a wide range of workloads that previously needed separate distributed systems...
- Including batch applications, iterative algorithms, interactive queries, and streaming
- By supporting these workloads in the same engine, Spark makes it easy to *combine* different processing types...  
Which is often necessary in big data analysis pipelines

# Apache Spark Components

## Spark Core

- Spark Core contains the basic functionality of Spark
  - Including components for task scheduling, memory management, fault recovery, interacting with storage systems, and more
- Spark Core is also home to the API that defines *resilient distributed datasets* (RDDs)
  - RDDs are Spark's main programming abstraction
  - RDDs represent a collection of items distributed across many compute nodes that can be manipulated in parallel
- Spark Core provides Java, Scala, R and Python APIs for ease of development, and for building and manipulating these collections

# Apache Spark Components

## Spark Streaming

- Spark Streaming is a Spark component that enables processing of live streams of data...
  - In the form of micro-batches of logs, messages or events
- Spark Streaming provides an API for manipulating data streams that closely matches the Spark Core's RDD API
- Underneath its API, Spark Streaming was designed to provide the same degree of fault tolerance, throughput, and scalability as Spark Core

# Apache Spark Components

## Spark SQL

- Spark SQL is Spark's package for working with structured data
- It allows querying data via SQL as well as the Hive Query Language (HQL)
- It supports many sources of data, including Hive tables and JSON
- Beyond providing a SQL interface to Spark, Spark SQL allows developers to mix SQL queries with use of data manipulations supported by RDDs
- Allows combining SQL with complex analytics



# Apache Spark Components

## MLib

- Spark comes with a library containing common machine learning (ML) functionality, called Mllib
- MLib provides multiple types of machine learning algorithms...
- Including classification, regression, clustering, and collaborative filtering...
- As well as supporting functionality such as model evaluation and data import
- It also provides some lower-level ML primitives, including a generic gradient descent optimization algorithm
- All of these methods are designed to scale across a cluster

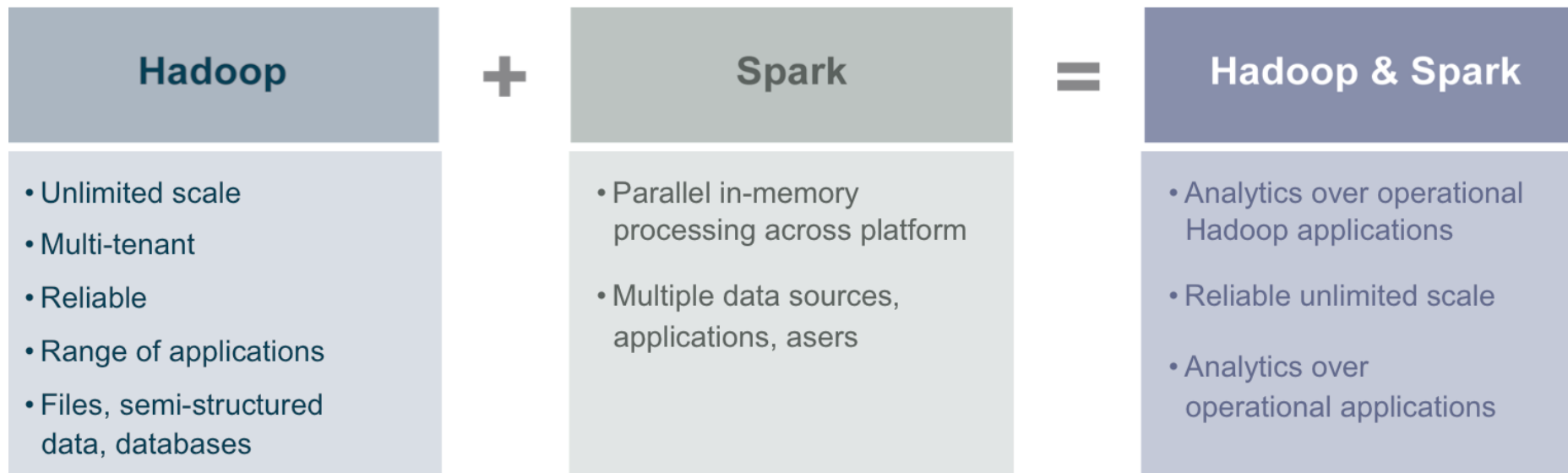
# Apache Spark Components

## GraphX

- GraphX is a library for manipulating graphs and also performing graph-parallel computations
- Like Spark Streaming and Spark SQL, GraphX extends the Spark RDD API
- Allows us to create a directed graph with arbitrary properties attached to each vertex and edge
- GraphX also provides various operators for manipulating graphs (e.g., subgraph and mapVertices) and a library of common graph algorithms (e.g., PageRank and triangle counting)

# Hadoop and Spark

- Spark can execute in standalone mode or using a cluster manager other than YARN (MESOS)
- But using SPARK on a Hadoop cluster with YARMS as its cluster manager offers numerous benefits



# Apache Spark and MapReduce

| Aspects          | MapReduce  | Spark   |
|------------------|--|---|
| Difficulty       | MapReduce is difficult to program and needs abstractions.                    | Spark is easy to program and does not require any abstractions.   |
| Interactive Mode | There is no in-built interactive mode, except Pig and Hive.                  | It has interactive mode.  |
| Streaming        | Hadoop MapReduce just get to process a batch of large stored data.           | Spark can be used to modify in real time through Spark Streaming.   |
| Performance      | MapReduce does not leverage the memory of the Hadoop cluster to the maximum. | Spark has been said to execute batch processing jobs about 10 to 100 times faster than Hadoop MapReduce.          |
| Latency          | MapReduce is disk oriented completely.                                       | Spark ensures lower latency computations by caching the partial results across its memory of distributed workers. |
| Ease of coding   | Writing Hadoop MapReduce pipelines is complex and lengthy process.           | Writing Spark code is always more compact.  |

# Using Apache Spark

## Interactive Shells

- Spark comes with interactive shells that enable ad hoc data analysis
- Spark's shells will feel familiar if you have used other shells...
- Such as those in R, Python, and Scala, or operating system shells like Bash
- Unlike most other shells which let you manipulate data using the disk and memory on a single machine...
- Spark's shells allow you to interact with data distributed on disk or in memory across many machines...
- And Spark takes care of automatically distributing this processing

# Using Apache Spark

## Interactive Shells

- Because Spark can load data into memory on worker nodes, many distributed computations, even ones that process terabytes of data across dozens of machines, can run in a few seconds
- This makes the sort of iterative, ad hoc, and exploratory analysis commonly done in shells a good fit for Spark
- Spark provides both Python and Scala shells that have been augmented to support connecting to a cluster
- Spark shells provide a simple way to learn the Spark API as well as a powerful tool to analyze data interactively

# Using Apache Spark

## Interactive Shells

- The easiest way to demonstrate the power of Spark's shells is to start using one of them for some simple data analysis
- The first step is to open up one of Spark's shells
- To open the Python version of the Spark shell type `pyspark` ← this is the one we shall use in our course
- To open the Scala version of the Spark shell, type `spark-shell`
- The shell prompt should appear within a few seconds
- When the shell starts you will see a lot of log messages
- You may need to press Enter once to clear the log output and get to a shell prompt

# Using Apache Spark

## Interactive Shells

- pyspark defines a special variable 'sc' which is initialized to reference an instance of the SparkContext
- A SparkContext provides the Spark client app an interface to the Spark execution engine
- Once pyspark is started you can type lines of Spark code in manually
- Or you can execute a file of Spark statements by using the 'execfile('somefilepath')' function
  - For example, `execfile('/home.maria_dev/myfile.py')`
  - Unlike when you enter actions manually, you need to include print statements to see the results of such actions when using `execfile`



# Using Apache Spark

## Self-Contained Applications

- Suppose we wish to write a self-contained application using the Spark API
- As an example, we'll create a simple Spark application, SimpleApp.py:

```
"""SimpleApp.py"""
from pyspark import SparkContext

logFile = "YOUR_SPARK_HOME/README.md" # Should be some file on your system
sc = SparkContext("local", "Simple App")
logData = sc.textFile(logFile).cache()

numAs = logData.filter(lambda s: 'a' in s).count()
numBs = logData.filter(lambda s: 'b' in s).count()

print("Lines with a: %i, lines with b: %i" % (numAs, numBs))
```

# Using Apache Spark

## Self-Contained Applications

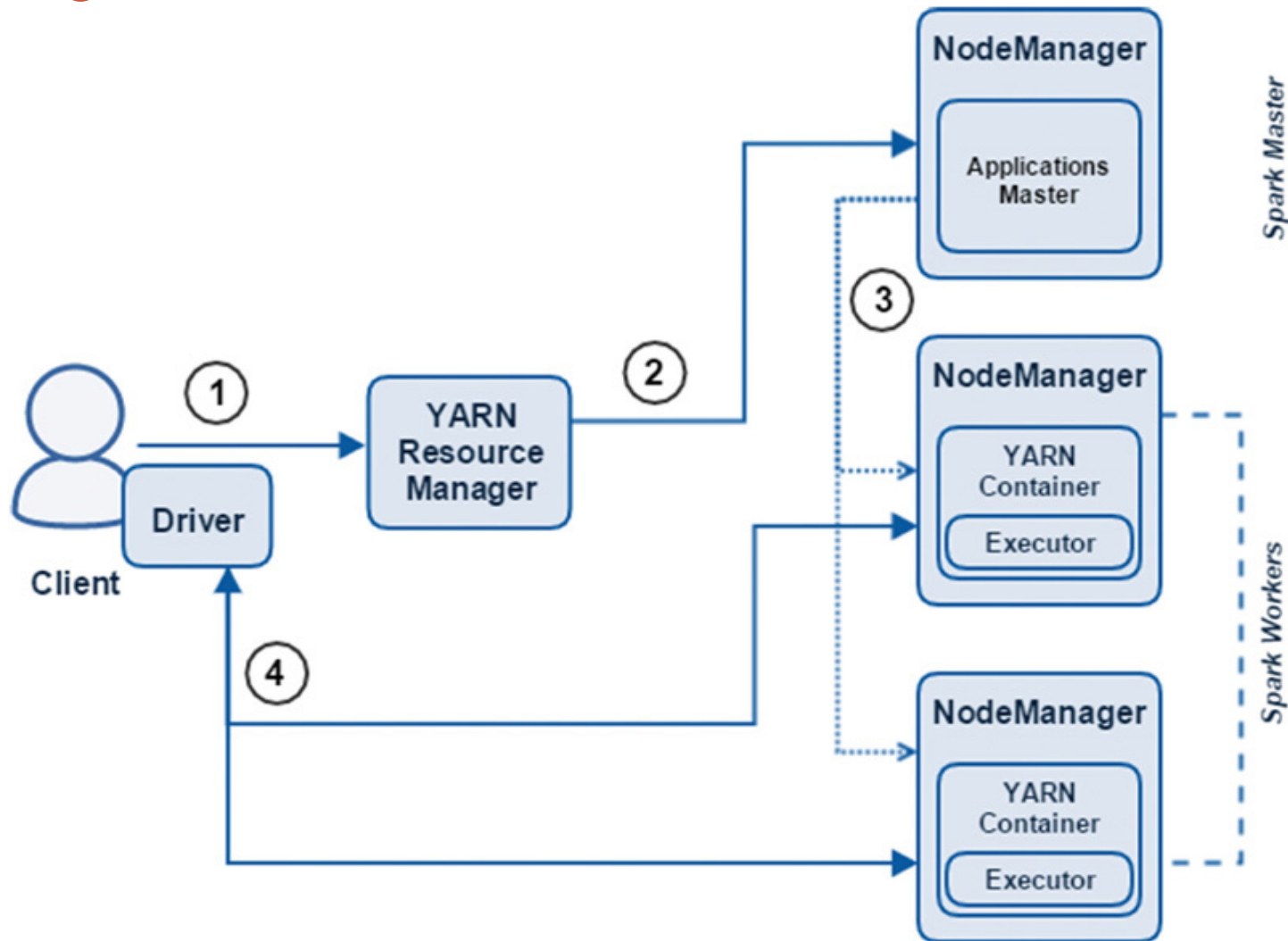
- This program just counts the number of lines containing 'a' and the number containing 'b' in a text file
- We can pass Python functions to Spark, which are automatically serialized along with any variables that they reference
- For applications that use custom classes or third-party libraries, we can also add code dependencies to spark-submit through its --py-files argument by packaging them into a .zip file
- SimpleApp is simple enough that we do not need to specify code dependencies.
- We can run this application using the bin/spark-submit script:  
# Use spark-submit to run your application  
\$ spark-submit --master yarn -deploy-mode cluster SimpleApp.py  
...  
Lines with a: 46, Lines with b: 23

# Spark On Hadoop

- A Spark application contains several components...
- All of which exist whether you are running Spark on a single machine or across a cluster of hundreds or thousands of nodes
- Each component has a specific role in executing a Spark program
- Some of these roles are passive during execution such as the client components...
- And other roles are active in the execution of the program, including components executing computation functions
- The components of a Spark application are the driver, the application master, the YARN cluster manager, and the executors that run on worker nodes
- All of the Spark components, including the driver, master, and executor processes, run in Java virtual machines (JVMs)

# Spark On Hadoop

## Using an Interactive CLI



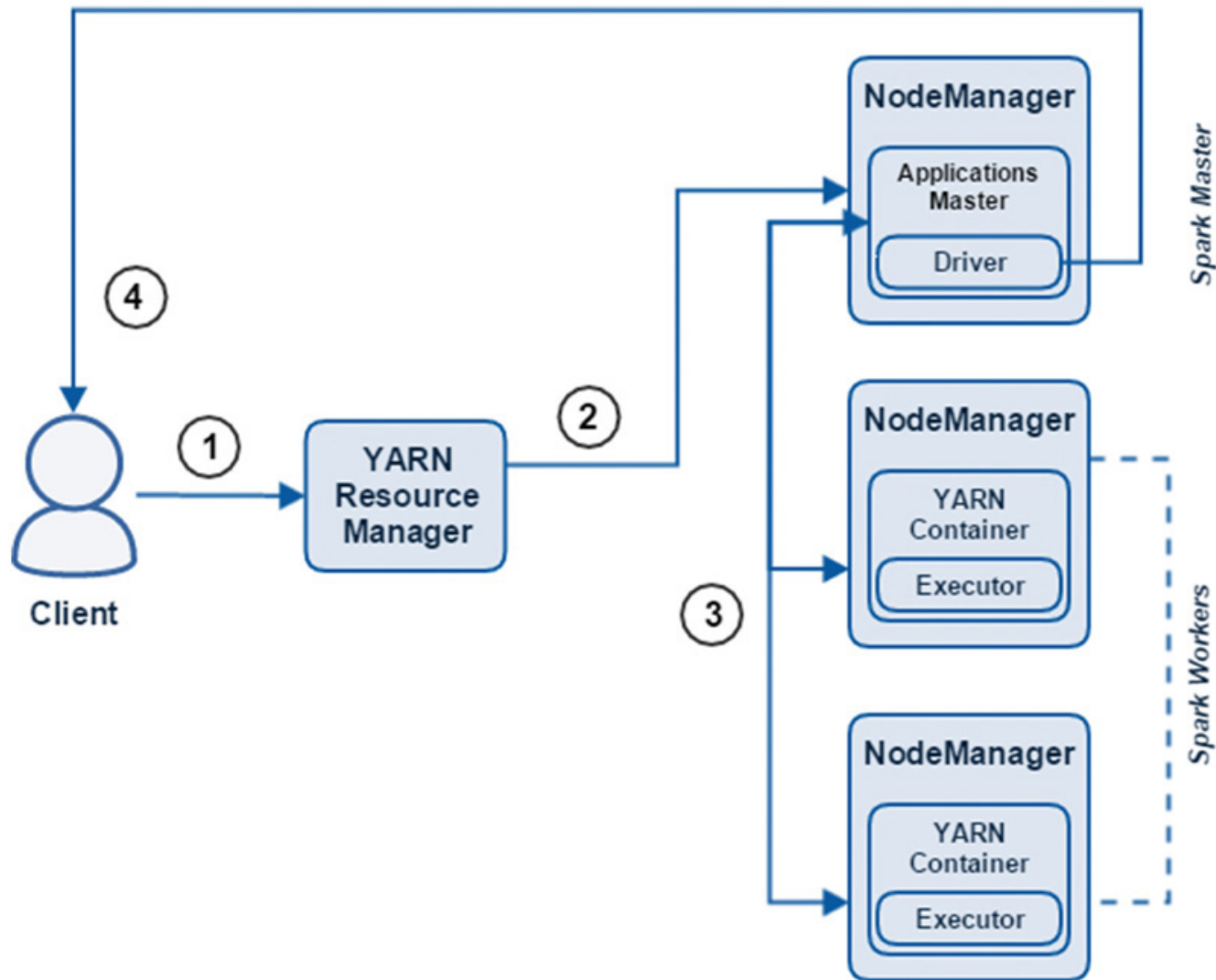
# Spark On Hadoop

## Using an Interactive CLI

1. The client submits a Spark application to the cluster manager (the YARN ResourceManager). The driver process is created and runs on the client.
2. The ResourceManager assigns an ApplicationsMaster (the Spark master) for the application.
3. The ApplicationsMaster requests containers for executors from the ResourceManager. The containers are assigned and the executors are spawned.
4. The driver (located on the client) then communicates with the executors to marshal processing of tasks and stages of the Spark program. The driver returns progress, results, and status to the client.

# Spark On Hadoop

## Using a Self-Contained Program



# Spark On Hadoop

## Using a Self-Contained Program

1. The client (a user process invoking `spark-submit`) submits a Spark application to the cluster manager (the YARN ResourceManager)
2. The ResourceManager assigns an ApplicationsMaster (the Spark master) for the application. The driver process is created on the same node
3. The ApplicationsMaster requests containers for executors from the ResourceManager. The containers are assigned and the executors are spawned. The driver then communicates with the executors to marshal processing of tasks and stages of the Spark program.
4. The driver returns progress, results, and status to the client

# Relating Spark to Other Hadoop Systems

- Data Representation
  - MapReduce → key value pairs
  - Hive → table (with schema)
  - Pig → relation (without schema)  
→ relation (with schema)
  - Spark → Resilient Distributed Datasets (RDDs)  
*like a Pig relation without schema.*  
*like MapReduce can hold key value pairs*  
→ DataFrames  
*like a Hive table with schema*



# Relating Spark to Other Hadoop Systems

- Data Operations
  - Hive
    - HQL (SQL-like language)
    - Hive CLI (or execute from .hql file)
  - Pig
    - Pig Latin (scripting language)
    - Pig CLI (or execute from .pig file)
  - Spark
    - Java, Scala, Python, R (programming languages)
    - Scala or Python CLI (or execute from program file)

# Relating Spark to Other Hadoop Systems

- Data Operations
  - MapReduce → map, reduce, sort
  - Hive → SQL-like operators
  - Pig → Relational operators  
*takes a relation as input and produces another relation as output.*  
→ Output operators  
*a DUMP or STORE statement is required to generate output*

# Relating Spark to Other Hadoop Systems

- Data Operations

- Spark
  - SQL-like operators (on DataFrames)
  - Transformation operators (on RDDs)
    - like Pig relational operators.*
    - takes an RDD as input and produces another relation as output.*
    - includes MR map-like, reduce-like and sort-like operations.*
  - Action operators (on RDDs)
    - like Pig output operators.*
    - takes an RDD as input and produces some result or stores into a file.*

# Resilient Distributed Datasets (RDD)

- Spark's core abstraction for working with data, is the resilient distributed dataset (RDD)
- An RDD is simply a distributed collection of elements
- In Spark all work is expressed as either creating new RDDs, transforming existing RDDs, or calling operations on RDDs to compute a result
- Under the hood, Spark automatically distributes the data contained in RDDs across your cluster...
- And parallelizes the operations you perform on them

# Resilient Distributed Datasets (RDD)

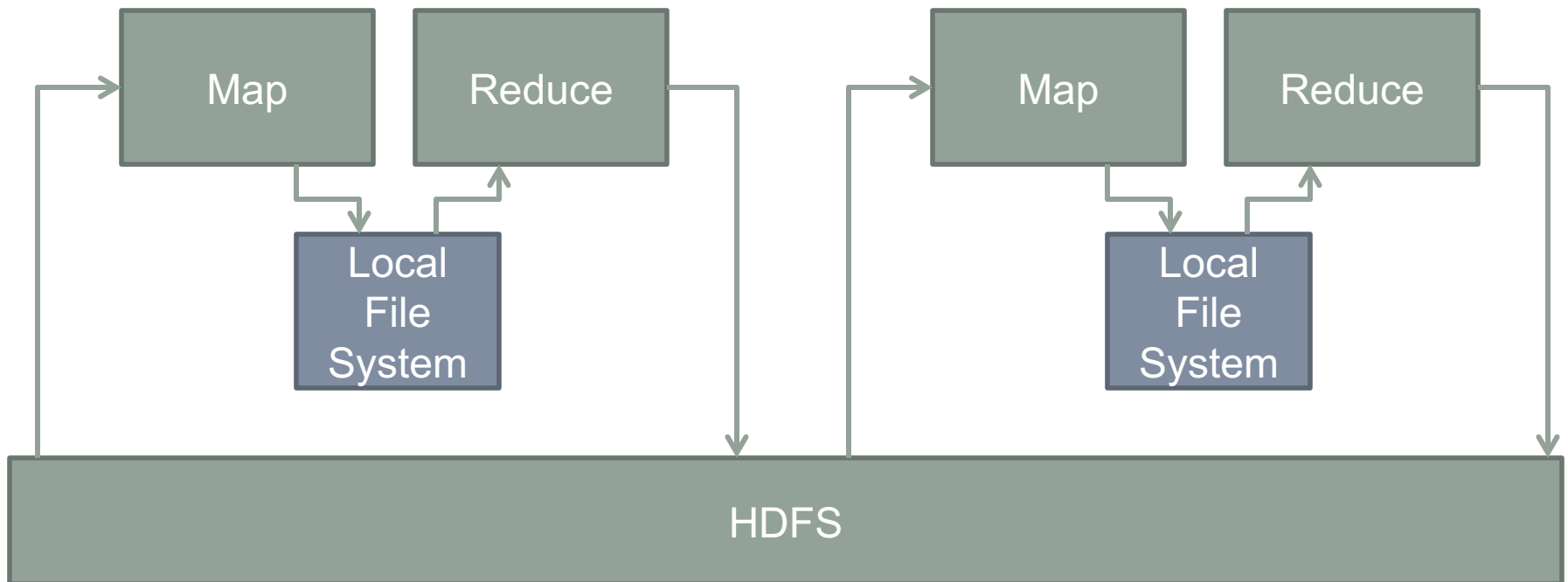
- An RDD in Spark is simply an immutable distributed collection of objects
- Each RDD is split into multiple *partitions*, which may be computed on different nodes of the cluster
- RDDs can contain any type of Python, Java, or Scala objects, including user-defined classes
- Users create RDDs in two ways: by loading an external dataset, or by distributing a collection of objects (e.g., a list or set) in their driver program

# Resilient Distributed Datasets (RDD)

- As one of the initial intended uses for Spark was to support machine learning, Spark's RDDs provided a form of shared memory that could make efficient reuse of data for iterative operations
- One of the main downsides of Hadoop's implementation of MapReduce was its persistence of intermediate data to disk and the copying of this data between nodes at runtime
- This distributed processing method of sharing data did provide resiliency and fault tolerance but it was at the cost of latency
- As data volumes increased along with the necessity for real-time data processing and insights...
- Spark's (mainly) in-memory processing framework based on the RDD has grown in popularity

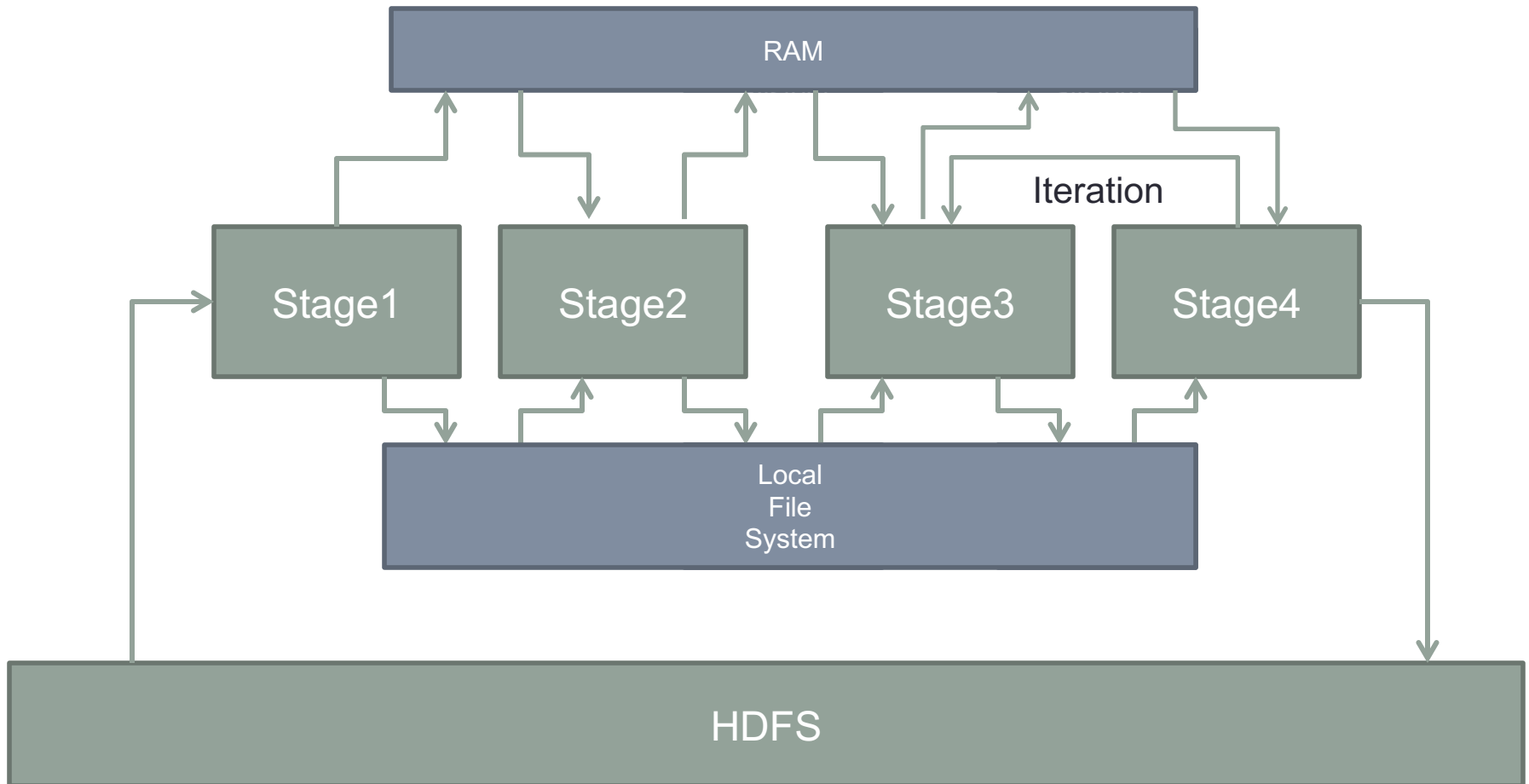
# MapReduce

## File Based Processing



# Spark

## Adaptive (In Memory and File Based Processing)





# Resilient Distributed Dataset (RDD)

- The term *Resilient Distributed Dataset* is an accurate and succinct descriptor for the concept. Here's how it breaks down:
  - **Resilient**
    - RDDs are *resilient*, meaning that if a node performing an operation in Spark is lost, the dataset can be reconstructed
    - This is because Spark knows the *lineage* of each RDD, which is the sequence of steps to create the RDD.
  - **Distributed**
    - RDDs are *distributed*, meaning the data in RDDs is divided into one or many *partitions* and distributed as in-memory collections of objects across worker nodes in the cluster
    - RDDs provide an effective form of shared memory to exchange data between processes (executors) on different nodes (workers)

# Resilient Distributed Dataset (RDD)

- **Dataset**
  - RDDs are *datasets* that consist of *records*
  - Records are uniquely identifiable data collections within a dataset
  - Records could be a collection of fields similar to a row in a table in a relational database, a line of text in a file, or multiple other formats.
  - RDDs are partitioned such that each partition contains a unique set of records and can be operated on independently

# Resilient Distributed Datasets (RDD)

- Another key property of RDDs is their immutability
- This means that after they are instantiated and populated with data, they cannot be updated
- Rather, new RDDs are created by performing one or more *transformations* such as `map()` or `filter()` on existing RDDs

# Data Locality with RDDs

- By default, Spark tries to read data into an RDD from the nodes that are close to it.
- As Spark usually accesses distributed partitioned data to optimize transformation operations...
- It creates partitions to hold the underlying blocks from HDFS


# Resilient Distributed Dataset (RDD)

- Because Spark can load data into memory on the worker nodes, many distributed computations, even ones that process terabytes of data across dozens of machines, can run in a few seconds
- This makes the sort of iterative, ad hoc, and exploratory analysis commonly done in shells a good fit for Spark
- Spark provides both Python and Scala shells that have been augmented to support connecting to a cluster

# Resilient Distributed Dataset (RDD)

## Transformation and Actions

- Once created, RDDs offer two types of operations *transformations* and *actions*
- *Transformations* construct a new RDD from a previous one.
- For example, one common transformation is filtering data that matches a predicate
- In our example, we use this to create a new RDD holding just the strings that contain the word *Python*



Source RDD holding lines of text. Some lines contain the word Python

- `pythonLines = textLines.filter(lambda line: "Python" in line)`


New RDD holding lines of text each of which contain the word Python

# Resilient Distributed Dataset (RDD)

## Transformation and Actions

- *Actions*, on the other hand, compute a result based on an RDD, and either return it to the driver program or save it to an external storage system (e.g., HDFS)
- One example of an action is `first()`, which returns the first element in an RDD as demonstrated

```
someString = pythonLines.first()
```



Returns the first record of the RDD as a string. An action does not create an RDD.

# RDD Fault Tolerance

RDDs track the transformations used to build them (their *lineage*) to recompute lost data

E.g:

```
messages = textFile(...).filter(lambda s: s.contains("ERROR"))  
                        .map(lambda s: s.split('\t')[2])
```



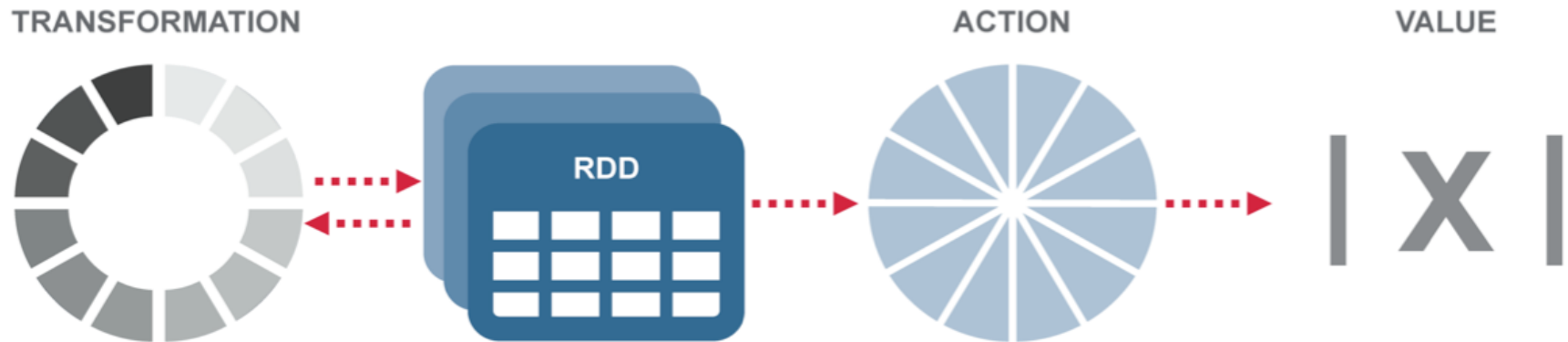


# Resilient Distributed Dataset (RDD)

## Transformation and Actions

- Transformations and actions are different because of the way Spark computes RDDs
- Although you can define new RDDs any time, Spark computes them only in a *lazy* fashion—that is, the first time they are used in an action
- Once Spark sees a whole chain of transformations, it can compute just the data needed for its result

# Transformations And Actions



# Transformations and Actions

## Summary



### Transformations

- Creates new dataset from existing one
- Transformed RDD executed only when action runs on it
- Examples: filter(), map(), flatMap()



### Actions

- Return a value to driver program after computation on dataset
- Examples: count(), reduce(), take(), collect()



# Resilient Distributed Dataset (RDD)

## Caching

- Finally, Spark's RDDs are by default recomputed each time you run an action on them
- If you would like to reuse an RDD in multiple actions, you can ask Spark to *persist* it using `RDD.persist()/RDD.cache()`
- We can ask Spark to persist our data in a number of different places
- After computing it the first time, Spark will store the RDD contents in memory (partitioned across the machines in your cluster), and reuse them in future actions
- Persisting RDDs on disk instead of memory is also possible
- The behavior of not persisting by default may again seem unusual, but it makes a lot of sense for big datasets
- If you will not reuse the RDD, there's no reason to waste storage space when Spark could instead stream through the data once and just compute the result

# Resilient Distributed Dataset (RDD)

## Caching

- Caching is very useful when data is accessed repeatedly, such as when querying a small “hot” dataset, or when running an iterative algorithm like PageRank
- As a simple example, let's mark our linesWithSpark dataset to be cached:

```
>>> linesWithSpark.cache()
```

```
>>> linesWithSpark.count()  
19
```

```
>>> linesWithSpark.count()  
19
```

- These same functions can be used on very large data sets, even when they are striped across tens or hundreds of nodes

# Resilient Distributed Dataset (RDD)

## Processing

- To summarize, every Spark program and shell session will work as follows:
- Create some input RDDs from external data.
- Transform them to define new RDDs using transformations like `filter()`.
- Ask Spark to `persist()/cache()` any intermediate RDDs that will need to be reused.
- Launch actions such as `count()` and `first()` to kick off a parallel computation, which is then optimized and executed by Spark

# Apache Spark

## Lambda Functions

- In Python, an anonymous function is a function that is defined without a name
- Normal Python functions are defined using the “def” keyword
- Anonymous functions are defined using the lambda keyword
  - Hence, anonymous functions are also called lambda functions
- A lambda function has the following syntax...  
lambda arguments: expression
  - Lambda functions can have any number of arguments but only one expression
  - The expression is evaluated and returned
- Lambda functions can be used wherever function objects are required

# Apache Spark

## Lambda Functions

- So for Spark it is correct to code either of the following
- Using a named function

```
def g (x):  
    x < 25
```

```
# include in otherRdd only those records of someRdd with value less  
# than 25
```

```
otherRdd = someRdd.filter(g)
```

- Using a lambda function
- ```
otherRdd = someRdd.filter(lambda x: x < 25)
```

- The lambda approach is often more readable in Scala and Python, but less so when using Java



# Apache Spark

## WordCount Example

- Load a file from HDFS into an RDD. Each record is a line from the original file
- Here 'sc' represents the system context—a connection to the Hadoop cluster

```
lines = sc.textFile("/user/jrosen/document.txt")
```

- Split apart the words in each record of the lines RDD
- Make each word its own record in a new words RDD using the flatMap() transformation
- Note from one record "line" flatMap produces a list of records each one a word

```
words = lines.flatMap(lambda line: line.split(" "))
```

- For each word in the words RDD create a key value pair of the form (word, 1) using the map transformation

```
wordPairs = words.map(lambda word: (word, 1))
```

# Apache Spark

## WordCount Example

- Now group each of the values in the wordPairs RDD by key of the form (key, (1, 1, ...)) using the reduceByKey() transformation
- And add all the 1's for a key together to get the count of times a word appears

```
wordCounts = wordPairs.reduceByKey(lambda a, b: a + b)
```

- Now store the word count RDD to HDFS using the saveAsTextFile() action

```
wordCount.saveAsTextFile("/user/jrosen/wordCounts.txt")
```

# Tips

- Recall that transformations execute in a lazy fashion...
  - Only upon execution of an action
- So if you provide a wrong argument to a transformation that can only be checked at run time...
  - This will only become apparent when you execute an action
- For example, let's say you want to load a text file named `"/user/test123.txt"`
- But you enter the following incorrect file name in your load command
  - `someRdd = sc.textFile("/user/test123")`
  - Notice that you left off the `".txt"` suffix
- This will be accepted by the spark shell without complaint at entry time
- But if you try to employ the following you will get an error, not from the action, but from the prior transformation
  - `someRdd.take(5)`

# Tips

- So as you are starting to explore spark just execute a simple action after you enter each transformation
  - A good choice is the “take(n)” transformation where n is some usually some small number, say 5
- If that works you are assured your transformation arguments are basically correct
- It also provides some idea of what the form and content of your transformation might be

# Tips

- Sometimes after executing a spark statement interactively the prompt does not display again until you hit “enter” one or two times

# API Overview

- Spark's capabilities can all be accessed and controlled using a rich API
- This supports Spark's four principal development environments
  - Scala, Java, Python, R
- Extensive documentation is provided regarding the API's instantiation in each of these languages
- The **Spark Programming Guide** provides further detail, with comprehensive code snippets in Scala, Java and Python
- The Spark API was optimized for manipulating data, with a design that reduced some common data science tasks from 100's or 1000's of lines of code to only a few

# Example Transformations

|                    |                                                                                                  |
|--------------------|--------------------------------------------------------------------------------------------------|
| <b>map</b>         | Returns new RDD by applying func to each element of source                                       |
| <b>filter</b>      | Returns new RDD consisting of elements from source on which function is true                     |
| <b>groupByKey</b>  | Returns dataset (K, Iterable<V>) pairs on dataset of (K,V)                                       |
| <b>reduceByKey</b> | Returns dataset (K, V) pairs where value for each key aggregated using the given reduce function |
| <b>flatMap</b>     | Similar to map, but function should return a sequence rather than a single item                  |
| <b>distinct</b>    | Returns new dataset containing distinct elements of source                                       |

# Transformations

- There are two kinds of transformations:
  - narrow transformations
  - wide transformations
- Narrow Transformations (single node, single partition)
  - Narrow transformations are the result of map, filter and similar, that is, using the data from a single partition only.
  - An output RDD has partitions with records that originate from a single partition in the parent RDD
  - Only a limited subset of partitions used to calculate the result.
  - Spark groups narrow transformations as a stage which is called pipelining



# Transformations

- Wide Transformations (multiple node, multiple partitions)
  - Wide transformations are the result of groupByKey and reduceByKey.
  - The data required to compute the records in a single partition may reside in many partitions of the parent RDD.
  - Note, wide transformations are also called shuffle transformations as they may or may not depend on a shuffle.
  - All of the tuples with the same key must end up in the same partition, processed by the same task
  - To satisfy these operations, Spark must execute RDD shuffle, which transfers data across cluster and results in a new stage with a new set of partitions

# Transformations

## `filter(func)`

- Description
  - Return a new dataset formed by selecting those elements of the source on which *func* returns true
  - Filter operator is somewhat equivalent to the WHERE clause in SQL
  - The function expression must always evaluate to (and return) a boolean type

# Transformations

## filter(func)

- Example
  - Input RDD
    - u'Joe,44,33,41,1'
    - u'Mel,13,33,30,50'
    - u'Mel,12,40,30,42'
    - u'Sam,15,28,28,39'
    - u'Mel,9,23,17,2'
    - ...
  - Operation
    - inRdd = sc.textFile("user/test123.txt")
    - filterRdd = inRdd.filter(lambda line: "Jill" in line)

# Transformations

## filter(func)

- Example
  - Output RDD
    - u'Jill,32,25,4,6'
    - u'Jill,3,37,15,4'
    - u'Jill,14,41,7,25',
    - u'Jill,38,20,47,46'
    - u'Jill,12,45,40,8
    - ...

# Transformations

## map(func)

- Return a new distributed dataset formed by passing each element of the source through a function *func*
- If *func* returns a collection of results then all those results are still grouped into one record

# Transformations

## map(func)

- *Example*

- Input RDD

- u'Joe,44,33,41,1'

- u'Mel,13,33,30,50'

- u'Mel,12,40,30,42'

- u'Sam,15,28,28,39'

- u'Mel,9,23,17,2'

- ...

- Operation

- inRdd = sc.textFile("user/test123.txt")

- upperRdd = inRdd.map(lambda line: line.upper())

- Note, the upper function accepts a string as input and returns a single string as output

# Transformations

## map(func)

- *Example*
  - Output RDD
    - u'JOE', u'44', u'33', u'41', u'1'
    - u'MELI', u'13', u'33', u'30', u'50'
    - u'MELI', u'12', u'40', u'30', u'42'
    - u'SAM', u'15', u'28', u'28', u'39'
    - u'MEL', u'9', u'23', u'17', u'2'
    - ...
- Note we still take one record as input to the map function and produce one record as output

# Transformations

## map(func)

- *Example*

- Input RDD

- u'Joe,44,33,41,1'

- u'Mel,13,33,30,50'

- u'Mel,12,40,30,42'

- u'Sam,15,28,28,39'

- u'Mel,9,23,17,2'

- ...

- Operation

- inRdd = sc.textFile("user/test123.txt")

- splitRdd = inRdd.map(lambda line: line.split(","))

- Note, the split function returns a list (collection) of words



# Transformations

## map(func)

- *Example*

- Output RDD

u'Joe', u'44', u'33', u'41', u'1' ← each record is composed of a list of items  
u'Mel', u'13', u'33', u'30', u'50'  
u'Mel', u'12', u'40', u'30', u'42'  
u'Sam', u'15', u'28', u'28', u'39'  
u'Mel', u'9', u'23', u'17', u'2'  
...

- Note we still take one record as input to the map function and produce one record as output
- Here each output record, rather than being a single string, is split into a collection of strings after using “,” as the delimiter
- This collection of strings is then take to be a single output record and not a record per collection member

# Transformations

## flatMap(func)

- Similar to map, but each input record can be mapped to 0 or more output records...
- So if *func* should return a collection rather than a single item each collection member becomes a new record
  - Here one record at a time is passed to *func* which then outputs zero or more results as a collection
  - Each member of the collection becomes a new record in the resulting RDD

# Transformations

## flatMap(func)

- *Example*

- Input RDD

- u'Joe,44,33,41,1'

- u'Mel,13,33,30,50'

- u'Mel,12,40,30,42'

- u'Sam,15,28,28,39'

- u'Mel,9,23,17,2'

- ...

- Operation

- inRdd = sc.textFile("user/test123.txt")

- flatRdd = inRdd.flatMap(lambda line: line.upper())

- The function upper returns a single string

# Transformations

## flatMap(func)

- *Example*

- Output RDD

- u'JOE,44,33,41,1'

- u'MEL,13,33,30,50'

- u'MEL,12,40,30,42'

- u'SAM,15,28,28,39'

- u'MEL,9,23,17,2'

- ...

- Note we still take one record as input to the map function and produce one record as output

# Transformations

## flatMap(func)

- *Example*

- Input RDD

- u'Joe,44,33,41,1'

- u'Mel,13,33,30,50'

- u'Mel,12,40,30,42'

- u'Sam,15,28,28,39'

- u'Mel,9,23,17,2'

- ...

- Operation

- inRdd = sc.textFile("user/test123.txt")

- flatRdd = inRdd.flatMap(lambda line: line.split(","))

- The function split returns a list (collection) of strings

# Transformations

## map(func)

- *Example*

- Output RDD

- u'Joe'

- u'44'

- u'33'

- u'41'

- u'1'

- u'Mel'

- u'13'

- u'33'

- ...

- Again each output record, rather than being a single string, is split into a collection of strings after using “,” as the delimiter
- Here each collection member becomes a new output record

# Transformations

## Working with Key-Value Pairs

- Most Spark operations work on RDDs containing any type of objects
- But there are a few special operations only available on RDDs holding key-value pairs
  - These RDDs are called “pair RDDs”
- Pair RDDs are useful as they allow operations that allow you to act on each key in parallel or group data across the network
- The most common operations are distributed “shuffle” operations...
  - These include grouping or aggregating the elements by a key
- In Python, these operations work on RDDs containing built-in Python tuples such as (1, 2)
  - Simply create such tuples and then call your desired operation

# Creating a Pair RDD

- Pair RDDs can be created by running a `map()` function that returns key-value pairs
- The procedure to build the key-value RDDs differs by language
- In Python language, for the functions on keyed data to work we need to return an RDD composed of tuples
- Creating a pair RDD using the first word as the key in Python programming language:

```
pairs = lines.map(lambda x: (x.split(" ")[0], x))
```



# Creating a Pair RDD

- Example

- Input RDD

- u'Joe'

- u'44'

- u'33'

- u'41'

- u'1'

- u'Mel'

- u'13'

- u'33'

- ...

- Operation

- wordsRdd = inRdd.map(lambda word: (word, 1))

- For each word (record) in the input RDD form a pair (word, 1)

# Creating a Pair RDD

- Example
  - Output RDD
    - (u'Joe', 1)
    - (u'44', 1)
    - (u'33', 1)
    - (u'41', 1)
    - (u'1', 1)]
    - ...

# Transformations on Pair RDDs

- Aggregation
- Grouping
- Joining
- Sorting

# Aggregations

- When datasets are described in terms of key or value pairs...
- It is common need is aggregate statistics across all elements with the same key
- Spark has a set of operations that process values associated with the same key
- These operations return RDDs and are transformations rather than actions
  - `reduceByKey()`
  - `foldByKey()` [not discussed further]
  - `combineByKey()` [not discussed further]

# Transformations

## `reduceByKey(func, [numTasks])`

- We start with a dataset of (K, V) pairs
- The values for each key are aggregated using the given reduce function *func*, which must be of type  $(V,V) \Rightarrow V$
- The number of reduce tasks is configurable through an optional second argument

# Transformations

## `reduceByKey(func, [numTasks])`

- Let's begin with a simple function to be used as an argument to `reduceByKey()`
- Now let's assume we have a pair RDD that looks as follows...
  - ("jill", 1)
  - ("jill", 1)
  - ("Jack", 1)
  - ("Jack", 1)
  - ("Jack", 1)
- What we want to do is create an output RDD with all the 1's associated with each name summed up follows:
  - ("Jill", 2)
  - ("Jack", 3)

# Transformations

## reduceByKey(func, [numTasks])

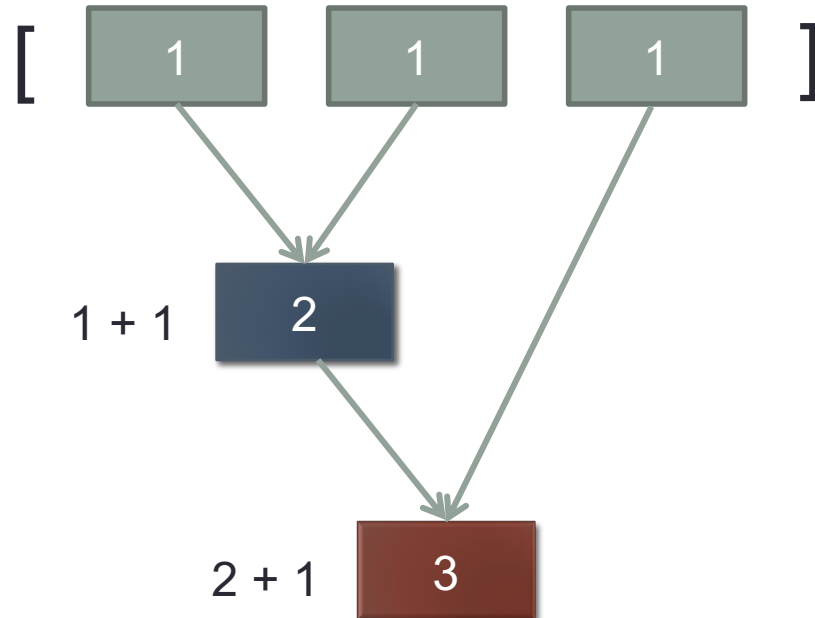
- Conceptually this can be done as a two step process
  - First group all the values associated with a single key into a collection
    - (“jill”, [1, 1])
    - (“Jack”, [1, 1, 1])
    - Note this step is conceptually the same as groupByKey()
  - Next sum up the 1’s in the list associated with each key together
    - The way this is represented in a spark aggregation function is as follows  
 $\text{lambda } (x, x) : x + x$
  - But how is this function applied by spark to arrive at the following?
    - (“Jill”, 2)
    - (“Jack”, 3)

# Transformations

## `reduceByKey(func, [numTasks])`

- It works as follows for key “Jack”

`lambda (x, y) : x + y`





# Transformations

## reduceByKey(func, [numTasks])

- Example

- Input RDD

- (u'Joe', 1)

- (u'44', 1)

- (u'12', 1)

- (u'41', 1)

- (u'1', 1)]

- (u'Joe', 1)

- (u'12', 1)

- (u'33', 1)

- (u'21', 1)

- (u'11', 1)

- ...

- Operation

- wordsRdd = inRdd.reduce(lambda (x, y) : x + y)

- Sum the 1's associated with each key

# Transformations

## reduceByKey(func, [numTasks])

- Example
  - Output RDD
    - (u'Joe', 2)
    - (u'44', 1)
    - (u'12', 2)
    - (u'41', 1)
    - (u'1', 1)]
    - (u'33', 1)
    - (u'21', 1)
    - (u'11', 1)
    - ...

# Grouping Data

- With key value data a common use case is grouping our data sets with respect to predefined key value
- For example, viewing all of a customer's orders together as one unit
- If our customer data is organized as key value pairs where each key is as name and each value is an order
- We can use `groupByKey()` to our order data by customer name into an RDD
- On an RDD consisting of keys of type `K` and values of type `V`...
- We get back an RDD operation of type `[K, Iterable[V]]`

# Transformations

## groupByKey([numTasks])

- When called on a dataset of (K, V) pairs, returns a dataset of (K, Iterable<V>) pairs

# Transformations

## groupByKey([numTasks])

- Example
  - Input RDD
    - (u'Joe', 1)
    - (u'44', 1)
    - (u'12', 1)
    - (u'41', 1)
    - (u'1', 1)]
    - (u'Joe', 1)
    - (u'12', 1)
    - (u'33', 1)
    - (u'21', 1)
    - (u'11', 1)
    - ...
  - Operation
    - wordsRdd = inRdd.groupByKey()
    - Collects the 1's associated with each key

# Transformations

## groupByKey([numTasks])

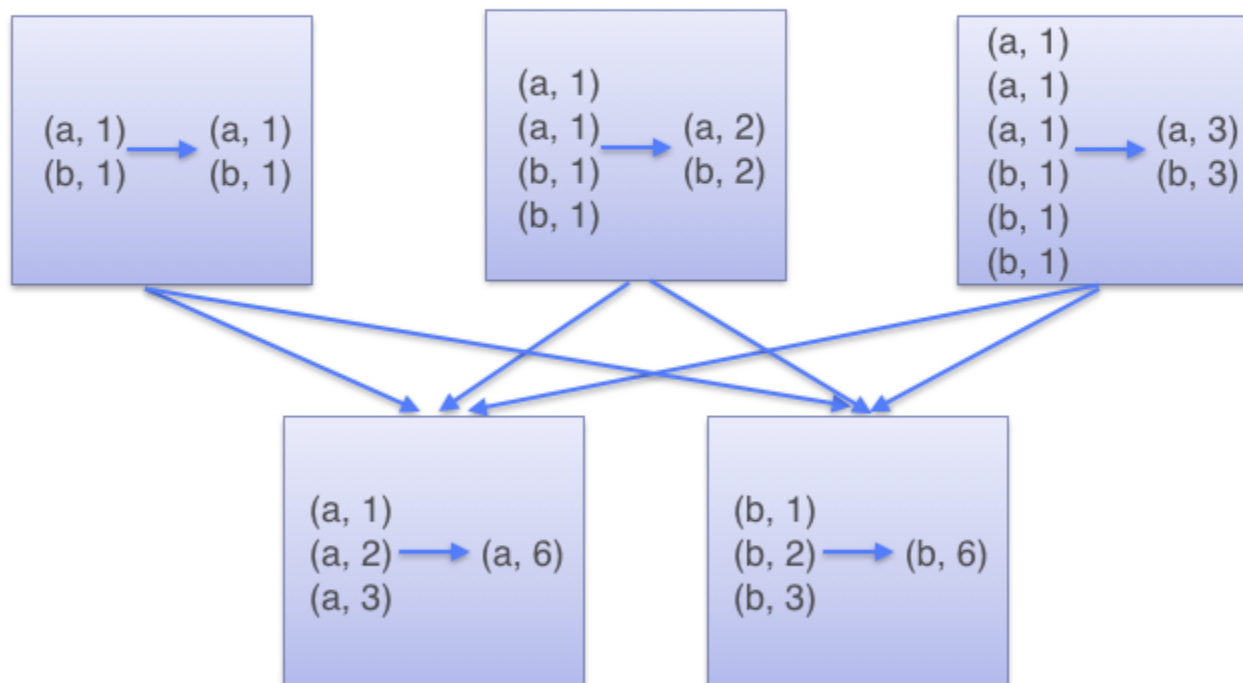
- Example
  - Output RDD
    - (u'Joe', [1, 1])
    - (u'44', 1)
    - (u'12', [1, 1])
    - (u'41', 1)
    - (u'1', 1)]
    - (u'33', 1)
    - (u'21', 1)
    - (u'11', 1)
    - ...

## ReduceByKey() Versus GroupByKey()

- While both of these functions will produce the correct answer, `reduceByKey` works much better on a large dataset
- That's because Spark knows it can combine output with a common key on each partition before shuffling the data.
- Look at the diagram below to understand what happens with `reduceByKey`. Notice how pairs on the same machine with the same key are combined (by using the `lambda` function passed into `reduceByKey`) before the data is shuffled. Then the `lambda` function is called again to reduce all the values from each partition to produce one final result.

# ReduceByKey() Versus GroupByKey()

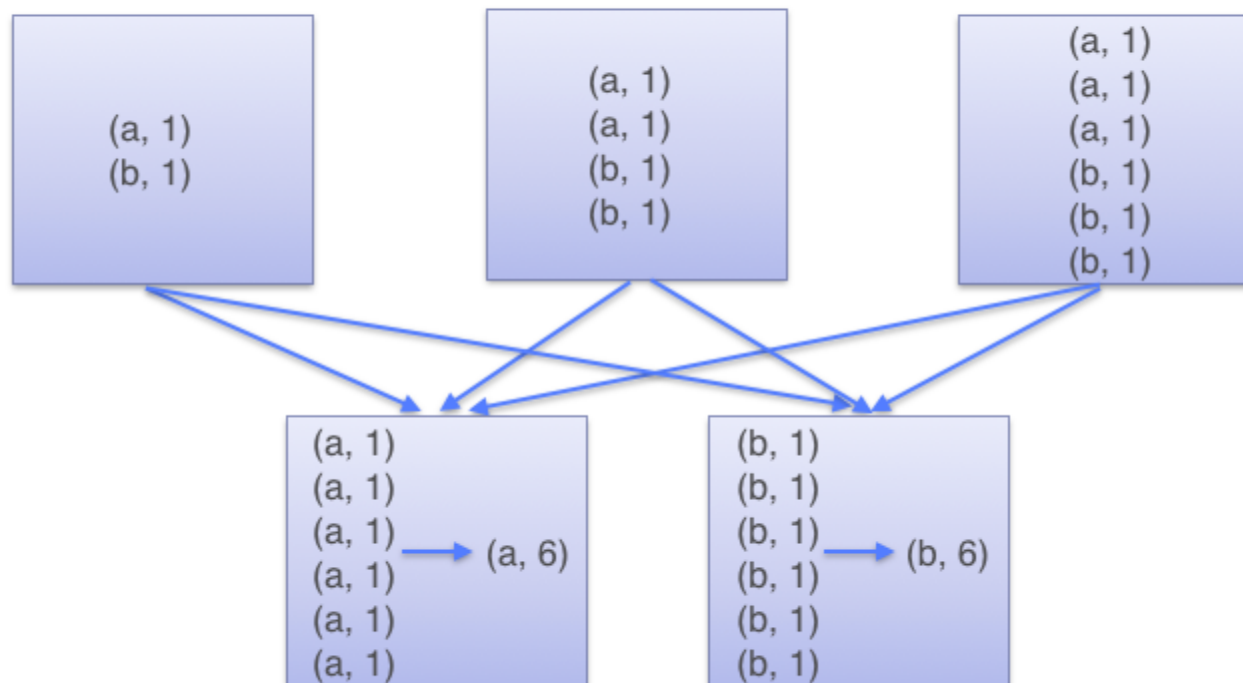
- Look at the diagram below to see what happens with reduceByKey.
- Pairs on the same machine with the same key are combined (by using lambda function passed into reduceByKey) before the data is shuffled
- Then the lambda function is called again to reduce all the values from each partition to produce one final result.





# ReduceByKey() Versus GroupByKey()

- On the other hand, when calling groupByKey - all the key-value pairs are shuffled around
- This is a lot of unnecessary data to being transferred over the network.



# Joining

- The most useful and effective operations we get with keyed data values comes from using it with other keyed data
- Joining datasets together is one of the most common type of operations you can find out on a pair RDD
  - Inner Join() : Only keys that are present in both pair RDDs have values that in the output RDD
  - leftOuterJoin() : The resulting pair RDD has entries for each key in the source RDD
  - rightOuterJoin() : is almost identical functioning to leftOuterJoin () except the key must be present in the other RDD

# Transformations

## *join(otherDataset, [numTasks])*

- When called on datasets of type  $(K, V)$  and  $(K, W)$ , returns a dataset of  $(K, (V, W))$  pairs, with all pairs of elements for each key

# Transformations

## *join(otherDataset, [numTasks])*

- Example

- Input RDD1 (in1Rdd)

- (u'Joe', 1)

- (u'44', 1)

- (u'12', 1)

- (u'41', 1)

- (u'1', 1)]

- (u'33', 1)

- (u'21', 1)

- (u'11', 1)

- ...

- Operation

- joinRdd = in1Rdd.join(in2Rdd)

- Input RDD2 (in2Rdd)

- (u'Joe', 5)

- (u'44', 5)

- (u'12', 5)

- (u'41', 5)

- (u'1', 1)]

- (u'33', 5)

- (u'21', 5)

- (u'11', 5)

- ...

# Transformations

*join(otherDataset, [numTasks])*

- Example

- Output RDD

- (u'Joe', (1, 5))

- (u'44', (1, 5))

- (u'12', (1, 5))

- (u'41', (1, 5))

- (u'1', (1, 5))

- (u'33', (1, 5))

- (u'21', (1, 5))

- (u'11', (1, 5))

- ...

- Note that the join of values creates a list of the value parts of each pair RDD

# Sorting Data

- We can sort an RDD with key or value pairs provided that there is an ordering defined on the key set
- Once we have sorted our data elements, any following call on the sorted data to `collect()` or `save()` will result in ordered dataset

# Transformations

## `sortByKey([ascending], [numTasks])`

- When called on a dataset of (K, V) pairs where K implements Ordered...
- Returns a dataset of (K, V) pairs sorted by keys in ascending or descending order as specified in the boolean ascending argument.

# Transformations

## sortByKey(ascending, [numTasks])

- Example
  - Input RDD (inRdd)
    - (u'Joe', 1)
    - (u'44', 1)
    - (u'12', 1)
    - (u'41', 1)
    - (u'1', 1)]
    - (u'Joe', 1)
    - (u'12', 1)
    - (u'33', 1)
    - (u'21', 1)
    - (u'11', 1)]
    - ...
  - Operation
    - sortRdd = inRdd.sortByKey(False)
    - Sort keys in descending order



# Transformations

## sortByKey(ascending, [numTasks])

- Example
  - Output RDD
    - (u'Joe', 1)
    - (u'Joe', 1)
    - (u'44', 1)
    - (u'41', 1)
    - (u'33', 1)
    - (u'21', 1)
    - (u'12', 1)
    - (u'12', 1)
    - (u'11', 1)
    - (u'1', 1)]
    - ...

# Example Actions

|                     |                                                               |
|---------------------|---------------------------------------------------------------|
| <b>count()</b>      | Returns the number of elements in the dataset                 |
| <b>reduce(func)</b> | Aggregate elements of dataset using function func             |
| <b>collect()</b>    | Returns all elements of dataset as an array to driver program |
| <b>take(n)</b>      | Returns an array with first n elements                        |
| <b>first()</b>      | Returns the first element of the dataset                      |

# Actions

## count()

- Return the number of elements in the dataset

# Actions

## count()

- Example
  - Input RDD (inRdd)
    - (u'Joe', 1)
    - (u'Joe', 1)
    - (u'44', 1)
    - (u'41', 1)
    - (u'33', 1)
    - (u'21', 1)
    - (u'12', 1)
    - (u'12', 1)
    - (u'11', 1)
    - (u'1', 1)]
    - ...
  - Operation
    - inRdd.count()

# Actions

## count()

- Example
  - Output
    - 5500 (count of the number of records in the RDD)

# Actions

## collect()

- Return all the elements of the dataset as an array at the driver program
- This is usually useful after a filter or other operation that returns a sufficiently small subset of the data

# Actions

## collect()

- Example

- Input RDD (inRdd)

- u'Joe,44,33,41,1'

- u'Mel,13,33,30,50'

- u'Mel,12,40,30,42'

- u'Sam,15,28,28,39'

- u'Mel,9,23,17,2'

- ...

- Operation

- someArray = inRdd.collect()

- All data of inRdd is provided to the client program

- This could exhaust memory if the result is too large

- Note the result is not an RDD but an array

# Actions

## collect()

- Example
  - Output Array
    - [ u'Joe,44,33,41,1'
    - u'Mel,13,33,30,50'
    - u'Mel,12,40,30,42'
    - u'Sam,15,28,28,39'
    - u'Mel,9,23,17,2'
    - ... ]



# Actions

## `reduce()`

- Aggregate the elements of the dataset using a function *func* which takes two arguments and returns one
- The function should be commutative and associative so that it can be computed correctly in parallel

# Actions

## reduce()

- Example
  - Input RDD (inRdd)
    - u'Joe,44,33,41,1'
    - u'Mel,13,33,30,50'
    - u'Mel,12,40,30,42'
    - u'Sam,15,28,28,39'
    - u'Mel,9,23,17,2'
    - ...
  - Operation
    - recLenRdd = inRdd.map(lambda s: len(s))
    - totalLen = recLenRdd.reduce(lambda a, b: a + b)
- The first line defines recLenRdd as the result of a map transformation
- We then run reduce, which is an action
- Spark breaks the computation into tasks to run on separate machines...
- And each machine runs its part of the map and a local reduction, returning only its answer to the driver program

# Actions

## reduce()

- Example

- Input RDD (inRdd)

- u'Joe,44,33,41,1'

- u'Mel,13,33,30,50'

- u'Mel,12,40,30,42'

- u'Sam,15,28,28,39'

- u'Mel,9,23,17,2'

- ...

- Operation

- recLenRdd = inRdd.map(lambda s: len(s))

- totalLen = recLenRdd.reduce(lambda a, b: a + b)

- The first line defines recLenRdd as the result of a map transformation
    - We then run reduce, which is an action
    - Spark breaks the computation into tasks to run on separate machines...
    - And each machine runs its part of the map and a local reduction, returning only its answer to the driver program

# Actions

## reduce()

- Example
  - Output  
print totalLen  
24375

# Actions

## take( $n$ )

- Return an array with the first  $n$  elements of the dataset

# Actions

`takeSample(withReplacement, num, [seed])`

- Return an array with a random sample of *num* elements of the dataset, with or without replacement (boolean)
- Optionally pre-specify a random number generator seed

# Actions

## first()

- Return the first element of the dataset (similar to take(1))

# Actions

## `foreach(func)`

- Run a function *func* on each element of the dataset
- Unlike other actions, `foreach` do not return any value
- It simply operates on all the elements in the RDD
- `foreach()` can be used in situations, where we do not want to return any result, but want to initiate a computation



# Actions

## `foreach(func)`

- Example
    - Input RDD (inRdd)
      - `u'Joe,44,33,41,1'`
      - `u'Mel,13,33,30,50'`
      - `u'Mel,12,40,30,42'`
      - `u'Sam,15,28,28,39'`
      - `u'Mel,9,23,17,2'`
      - `...`
    - Operation
      - `def p(x)`
        - `print x`
- `inRdd.foreach(p);`

# Actions

## `foreach(func)`

- Example

- Output

Joe,44,33,41,1

Mel,13,33,30,50

Mel,12,40,30,42

Sam,15,28,28,39

Mel,9,23,17,2

...

# Actions

## `saveAsTextFile(path)`

- Write elements of the dataset as a text file (or set of text files) in a given directory in the local filesystem, HDFS or any other Hadoop-supported file system
- Spark will call `toString()` on each element to convert it to a line of text in the file

# Shared Variables

- Normally, when a function passed to a Spark operation (such as map or reduce) is executed on a remote cluster node, it works on separate copies of all variables used in the function
- These variables are copied to each machine, and no updates to the variables on the remote machine are propagated back to the driver program
- Supporting general, read-write shared variables across tasks would be inefficient
- However, Spark does provide two limited types of *shared variables* for two common usage patterns...
- Broadcast variables and accumulators (not discussed)

# Broadcast Variables

- Broadcast variables allow the programmer to keep a read-only variable cached on each machine rather than shipping a copy of it with tasks
- They can be used, for example, to give every node a copy of a large input dataset once in an efficient manner, rather than each time a spark function (closure) is executed
- The best examples of use of broadcast variables are for distributing large reference data or lookup tables
- Spark attempts to distribute broadcast variables using efficient broadcast algorithms to reduce communication cost

# Broadcast Variables

- Broadcast variables are created from a variable `v` by calling `SparkContext.broadcast(v)`
- The broadcast variable is a wrapper around `v`, and its value can be accessed by calling the `value` method
- The Python code below shows this

```
data = [1, 2, 3]
```

```
broadcastVar = sc.broadcast(data)
```

```
# The array 'data' is now available on every cluster node to be
```

```
# accessed as follows
```

```
someRdd = otherRdd.filter (lambda x : x < broadcastVar.value[0])
```