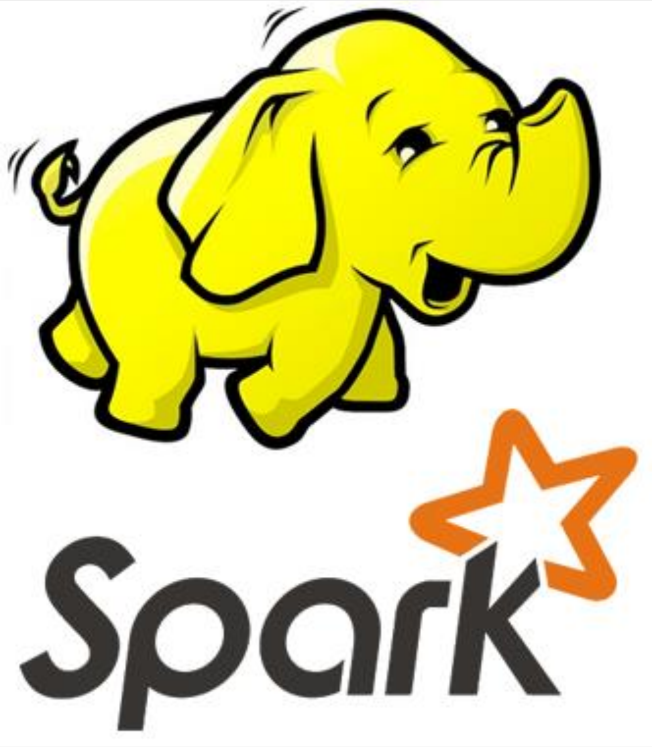
COMP9313: Big Data Management



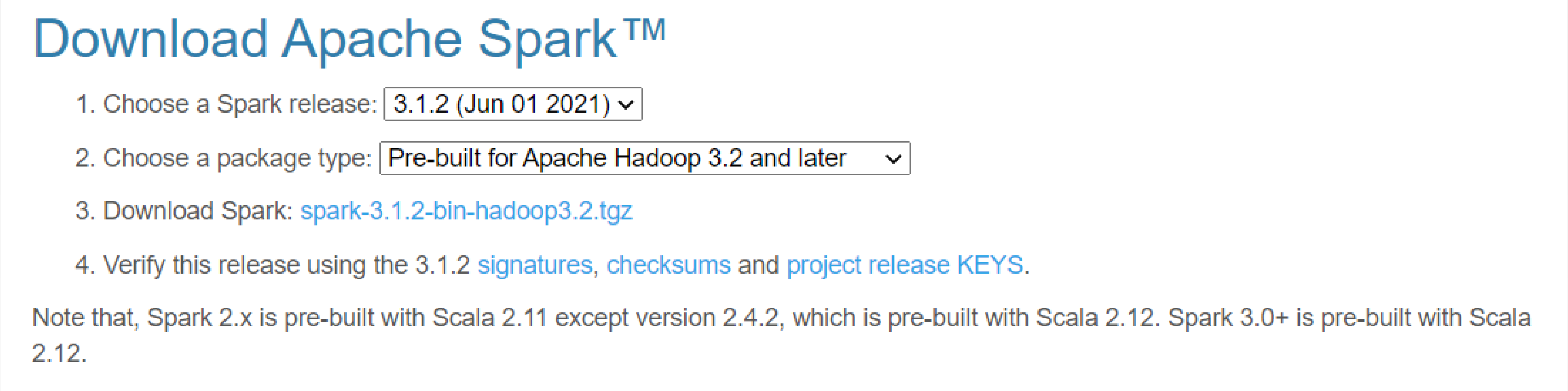
Lecturer: Xin Cao

Course web site: http://www.cse.unsw.edu.au/~cs9313/

Chapter 4.2: Spark II

Download and Configure Spark

- Current version: 3.1.2. https://spark.apache.org/downloads.html
  - You also need to install Java first



Download Apache Spark™

1. Choose a Spark release: 3.1.2 (Jun 01 2021)
2. Choose a package type: Pre-built for Apache Hadoop 3.2 and later
3. Download Spark: spark-3.1.2-bin-hadoop3.2.tgz
4. Verify this release using the 3.1.2 signatures, checksums and project release KEYS.

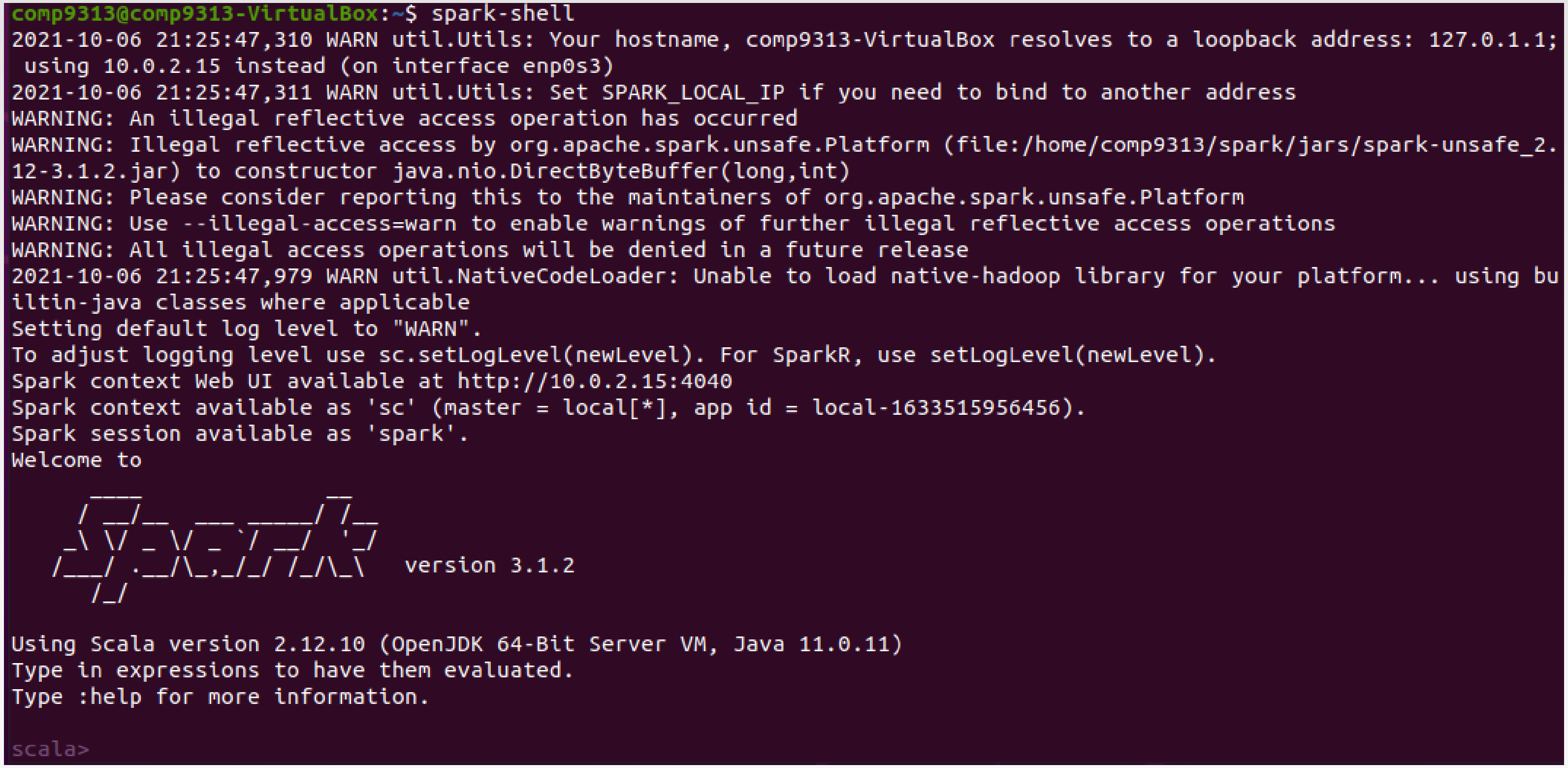
Note that, Spark 2.x is pre-built with Scala 2.11 except version 2.4.2, which is pre-built with Scala 2.12. Spark 3.0+ is pre-built with Scala 2.12.

- After downloading the package, unpack it and then configure the path variable in file ~/.bashrc

```
export SPARK_HOME=/home/comp9313/workdir/spark
export PATH=$SPARK_HOME/bin:$PATH
```

Spark Shell

- Spark comes with four widely used interpreters that act like interactive “shells” and enable ad hoc data analysis: pyspark, spark-shell, sparksql, and sparkR
- You can start the spark-shell by using the command “spark-shell”



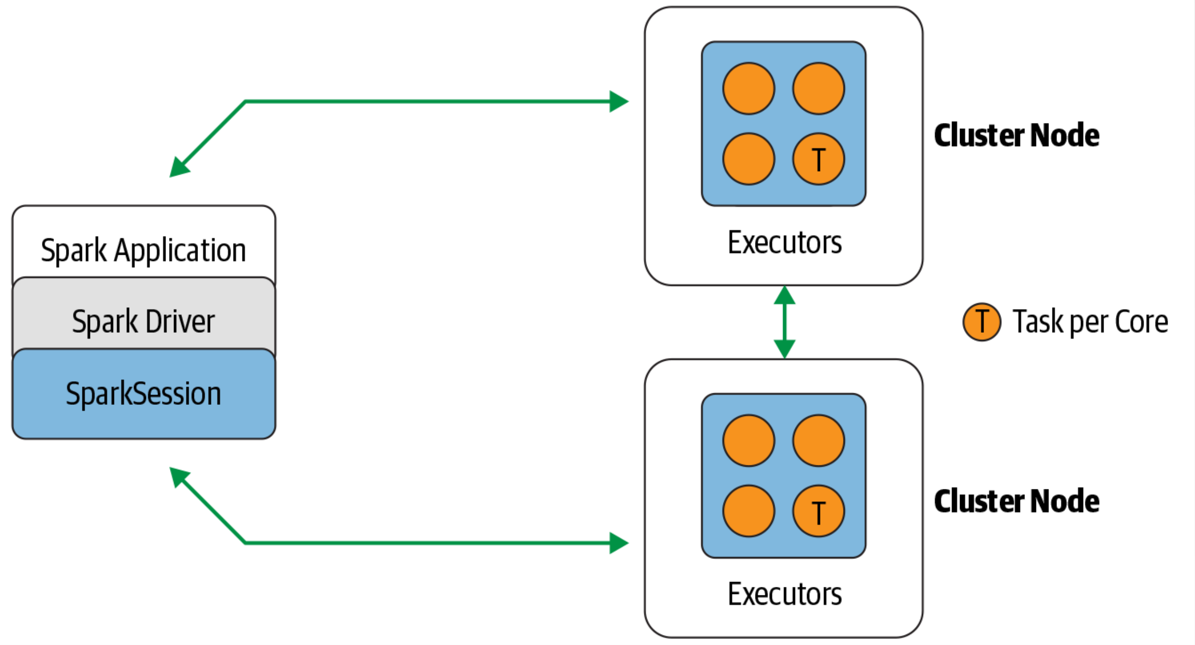
```
comp9313@comp9313-VirtualBox:~$ spark-shell
2021-10-06 21:25:47,310 WARN util.Utils: Your hostname, comp9313-VirtualBox resolves to a loopback address: 127.0.1.1;
 using 10.0.2.15 instead (on interface enp0s3)
2021-10-06 21:25:47,311 WARN util.Utils: Set SPARK_LOCAL_IP if you need to bind to another address
WARNING: An illegal reflective access operation has occurred
WARNING: Illegal reflective access by org.apache.spark.unsafe.Platform (file:/home/comp9313/spark/jars/spark-unsafe_2.
12-3.1.2.jar) to constructor java.nio.DirectByteBuffer(long,int)
WARNING: Please consider reporting this to the maintainers of org.apache.spark.unsafe.Platform
WARNING: Use --illegal-access=warn to enable warnings of further illegal reflective access operations
WARNING: All illegal access operations will be denied in a future release
2021-10-06 21:25:47,979 WARN util.NativeCodeLoader: Unable to load native-hadoop library for your platform... using bu
iltin-java classes where applicable
Setting default log level to "WARN".
To adjust logging level use sc.setLogLevel(newLevel). For SparkR, use setLogLevel(newLevel).
Spark context Web UI available at http://10.0.2.15:4040
Spark context available as 'sc' (master = local[*], app id = local-1633515956456).
Spark session available as 'spark'.
Welcome to
      ____              __
     / __/__  ___ _____/ /__
    _\ \/ _ \/ _ `/ __/  '_/
   /___/ .__/\_,_/_/ /_/\_\   version 3.1.2
      /_/

Using Scala version 2.12.10 (OpenJDK 64-Bit Server VM, Java 11.0.11)
Type in expressions to have them evaluated.
Type :help for more information.

scala>
```

Understanding Spark Application Concepts

- Application
  - A user program built on Spark using its APIs. It consists of a driver program and executors on the cluster
- SparkContext/SparkSession
  - An object that provides a point of entry to interact with underlying Spark functionality and allows programming Spark with its APIs
- Job
  - A parallel computation consisting of multiple tasks that gets spawned in response to a Spark action (e.g., save(), collect()).
- Stage
  - Each job gets divided into smaller sets of tasks called stages that depend on each other.
- Task
  - A single unit of work or execution that will be sent to a Spark executor.

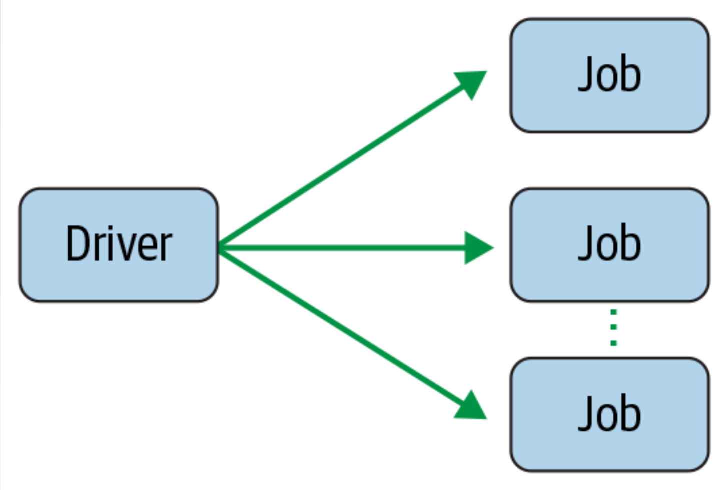
Spark Application and SparkSession

- The core of every Spark application is the Spark driver program, which creates a SparkSession (SparkContext in Spark 1.x) object.
  - When you're working with a Spark shell, the driver is part of the shell and the SparkSession/SparkContext object (accessible via the variable spark) is created for you
  - Once you have a SparkSession/ SparkContext, you can program Spark using the APIs to perform Spark operations.



Spark Jobs

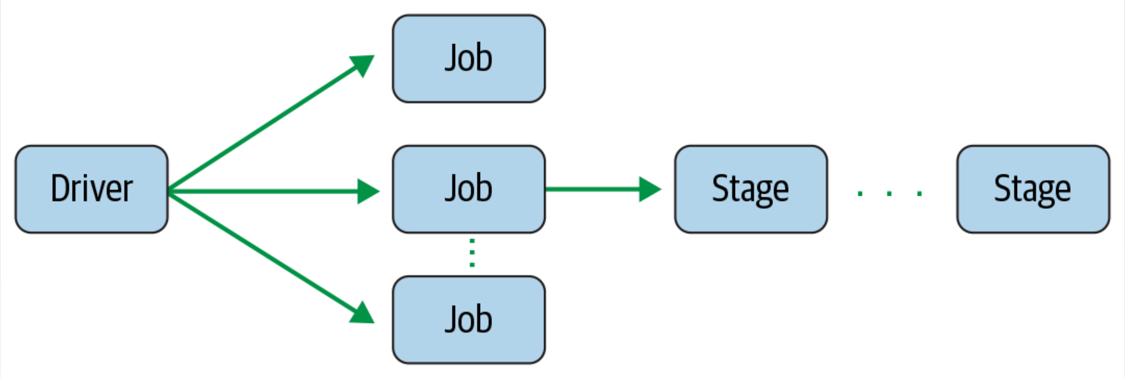
- During interactive sessions with Spark shells, the driver converts your Spark application into one or more Spark jobs



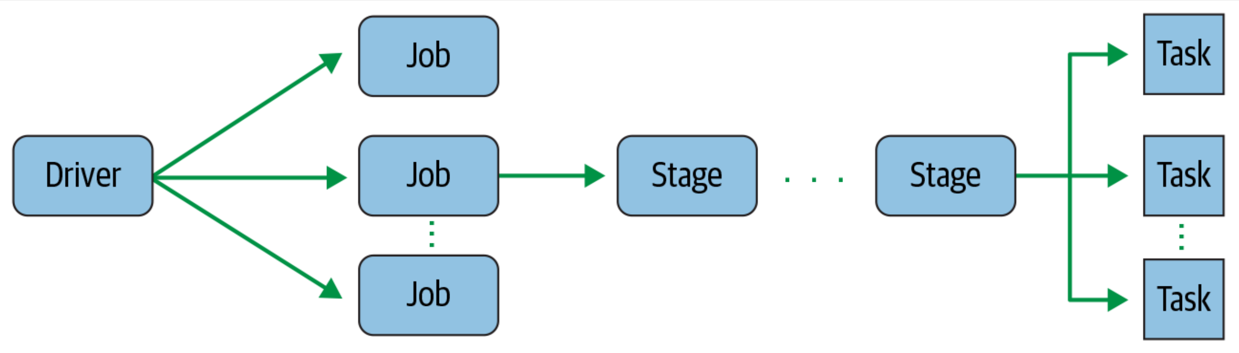
- It then transforms each job into a Spark's execution plan as a DAG, where each node within a DAG could be a single or multiple Spark stages.

Spark Stages and Tasks

- Stages are created based on what operations can be performed serially or in parallel.



- Each stage is comprised of Spark tasks (a unit of execution), which are then federated across each Spark executor; each task maps to a single core and works on a single partition of data

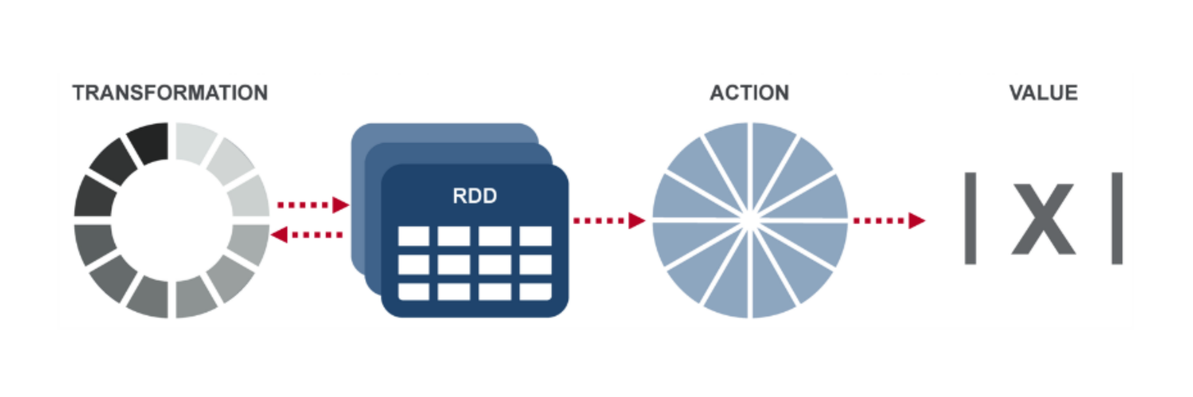


The Spark UI

- Spark includes a graphical user interface that you can use to inspect or monitor Spark applications in their various stages of decomposition—that is jobs, stages, and tasks.
- The driver launches a web UI, running by default on port 4040, where you can view metrics and details such as:
  - A list of scheduler stages and tasks
  - A summary of RDD sizes and memory usage
  - Information about the environment
  - Information about the running executors
  - All the Spark SQL queries
- In local mode, you can access this interface at http://localhost:4040 in a web browser.

Part 1: Programming with RDD

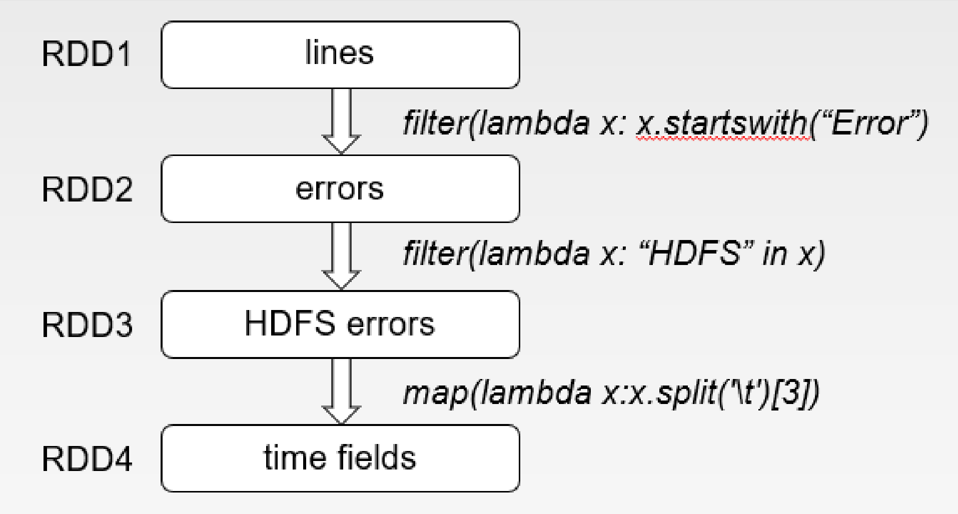
RDD Operations



- **Transformation:** returns a new RDD.
  - Nothing gets evaluated when you call a Transformation function, it just takes an RDD and return a new RDD.
  - Transformation functions include *map, filter, flatMap, groupByKey, reduceByKey, aggregateByKey, join, etc.*
- **Action:** evaluates and returns a new value.
  - When an Action function is called on a RDD object, all the data processing queries are computed at that time and the result value is returned.
  - Action operations include *reduce, collect, count, first, take, countByKey, foreach, saveAsTextFile, etc.*

Example

- Web service is experiencing errors and an operators want to search terabytes of logs in the Hadoop file system to find the cause.


RDD1
lines
filter(lambda x: x.startswith("Error")
RDD2
errors
filter(lambda x: "HDFS" in x)
RDD3
HDFS errors
map(lambda x:x.split('\t')[3])
RDD4
time fields

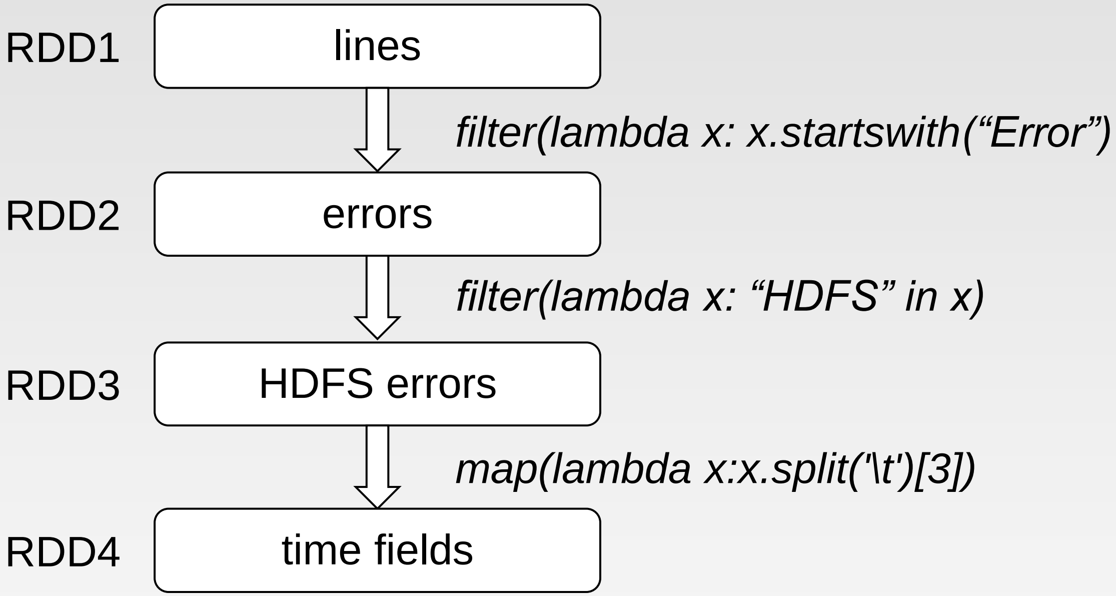
```
//base RDD
val lines = sc.textFile("hdfs://…")
//Transformed RDD
val errors = lines.filter(_.startsWith("Error"))
errors.persist()
errors.count()
errors.filter(_.contains("HDFS"))
          .map(_.split('\t')(3))
          .collect()
```

- Line1: RDD backed by an HDFS file (base RDD lines not loaded in memory)
- Line3: Asks for errors to persist in memory (errors are in RAM)

Lineage Graph

RDDs keep track of *lineage*

- RDD has enough information about how it was derived from to compute its partitions from data in stable storage.



RDD1 lines

filter(lambda x: x.startswith("Error")

RDD2 errors

filter(lambda x: "HDFS" in x)

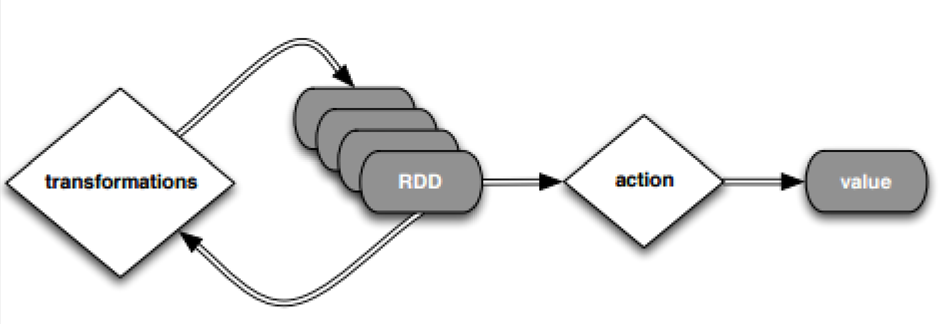
RDD3 HDFS errors

map(lambda x:x.split('\t')[3])

RDD4 time fields

- Example:
  - If a partition of errors is lost, Spark rebuilds it by applying a filter on only the corresponding partition of lines.
  - Partitions can be recomputed in parallel on different nodes, without having to roll back the whole program.

Deconstructed

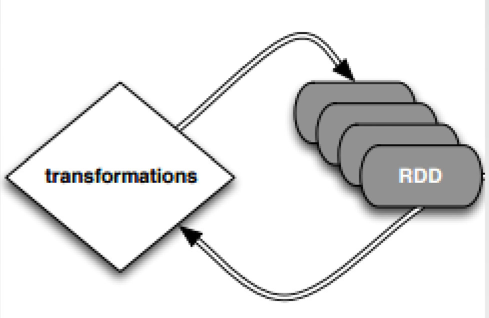


```
//base RDD
val lines = sc.textFile(“hdfs://…”)
//Transformed RDD
val errors = lines.filter(_.startsWith(“Error”))
errors.persist()
errors.count()
errors.filter(_.contains(“HDFS”))
          .map(_.split(‘\t’)(3))
          .collect()
```

Deconstructed

//base RDD

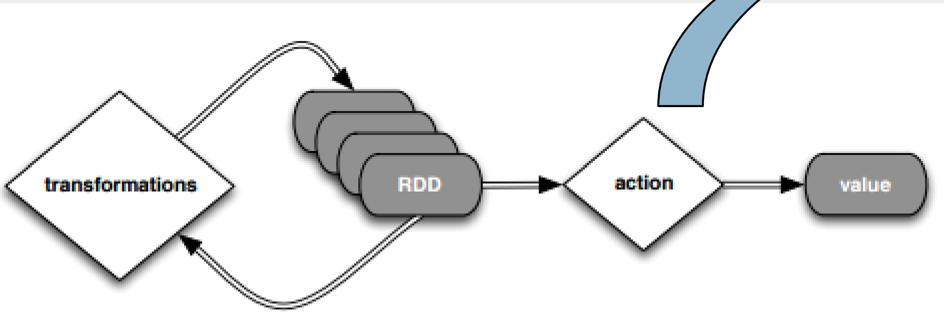
val lines = sc.textFile(“hdfs://…”)



//Transformed RDD

val errors = lines.filter(_.startsWith(“Error”))

errors.persist()



errors.count()

count() causes Spark to: 1) read data; 2) sum within partitions; 3) combine sums in driver

Put transform and action together:

errors.filter(_.contains(“HDFS”)).map(_split(‘\t’)(3)).collect()

SparkContext

- SparkContext is the entry point to Spark for a Spark application.
- Once a SparkContext instance is created you can use it to
  - Create RDDs
  - Create accumulators
  - Create broadcast variables
  - access Spark services and run jobs
- A Spark context is essentially a client of Spark’s execution environment and acts as the *master of your Spark application*
- The first thing a Spark program must do is to create a SparkContext object, which tells Spark how to access a cluster
- In the Spark shell, a special interpreter-aware SparkContext is already created for you, in the variable called *sc*

RDD Persistence: Cache/Persist

- One of the most important capabilities in Spark is *persisting* (or *caching*) a dataset in memory across operations.
- When you persist an RDD, each node stores any partitions of it. You can reuse it in other actions on that dataset
- Each persisted RDD can be stored using a different *storage level,* e.g.
  - MEMORY_ONLY:
    - Store RDD as deserialized Java objects in the JVM.
    - If the RDD does not fit in memory, some partitions will not be cached and will be recomputed when they're needed.
    - This is the default level.
  - MEMORY_AND_DISK:
    - If the RDD does not fit in memory, store the partitions that don't fit on disk, and read them from there when they're needed.
- cache() = persist(StorageLevel.MEMORY_ONLY)

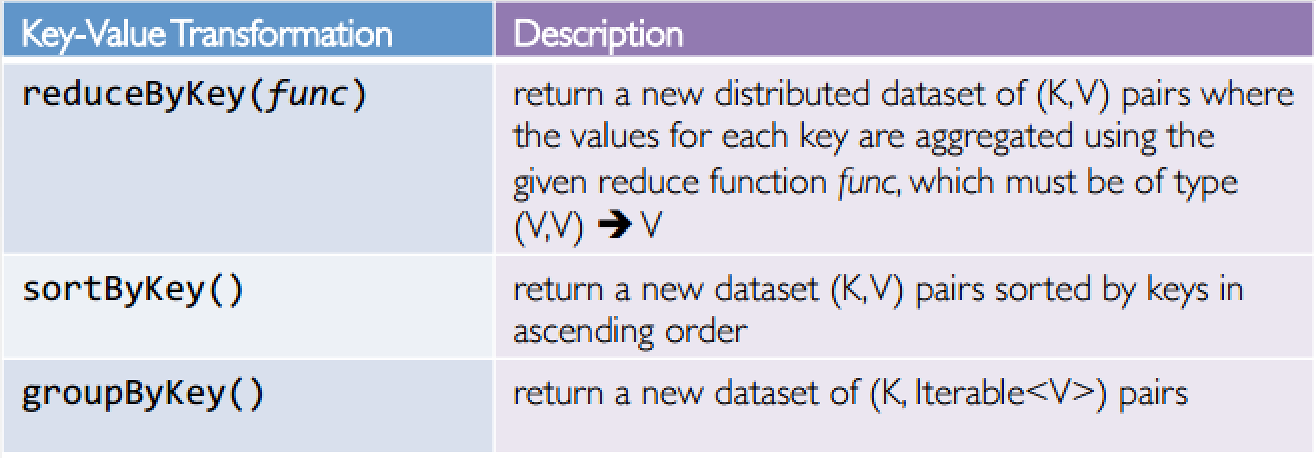
Why Persisting RDD?

```
val lines = sc.textFile(“hdfs://…”)
val errors = lines.filter(_.startsWith(“Error”))
errors.persist()
errors.count()
```

- If you do errors.count() again, the file will be loaded again and computed again.
- Persist will tell Spark to cache the data in memory, to reduce the data loading cost for further actions on the same data
- erros.persist() will do nothing. It is a lazy operation. But now the RDD says "read this file and then cache the contents". The action will trigger computation and data caching.

Spark Key-Value RDDs

- Similar to Map Reduce, Spark supports Key-Value pairs
- Each element of a *Pair RDD* is a pair tuple
- Spark supports data partitioning control for pair RDDs
- Some Key-Value transformation functions:



Key-Value Transformation	Description
`reduceByKey(`*`func`*`)`	return a new distributed dataset of (K,V) pairs where the values for each key are aggregated using the given reduce function *func*, which must be of type (V,V) ➔ V
`sortByKey()`	return a new dataset (K,V) pairs sorted by keys in ascending order
`groupByKey()`	return a new dataset of (K, Iterable<V>) pairs

Pair RDD Example (Transformation)

- Transformations on one pair RDD rdd = {(1, 2), (3, 4), (3, 6)}

Name	Purpose	Example	Result
reduceByKey(func)	Combine values with the same key	rdd.reduceByKey((x, y) => x + y)	{(1, 2)}
groupByKey()	Group values with the same key	rdd.groupByKey()	{(1, [2]), (3, [4, 6])}
mapValues(func)	Apply a function to each value of a pair RDD without changing the key	rdd.mapValues(x => x+1)	{(1, 3), (3, 5), (3, 7)}
keys()	Return an RDD of just the keys	rdd.keys()	{1, 3, 3}
values()	Return an RDD of just the values	rdd.values()	{2, 4, 6}
sortByKey()	Return an RDD sorted by the key	rdd.sortByKey()	{(1, 2), (3, 4), (3, 6)}

Pair RDD Example (Transformation)

- Transformations on two pair RDDs rdd1 = {(1, 2), (3, 4), (3, 6)} and rdd2 = {(3, 9)})

Name	Purpose	Example	Result
subtractByKey	Remove elements with a key present in the other RDD	rdd1.subtractByKey (rdd2)	{(1, 2), (3, 10)}
join	Perform an inner join between two RDDs	rdd1.join(rdd2)	{(3, (4, 9)), (3, (6, 9))}
cogroup	Group data from both RDDs sharing the same key	rdd1.cogroup(rdd2)	{(1,([2],[])), (3, ([4, 6],[9]))}

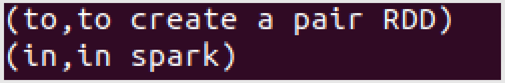
Pair RDD Example (Actions)

- Actions on one pair RDD rdd = ({(1, 2), (3, 4), (3, 6)})

Name	Purpose	Example	Result
countByKey()	Count the number of elements for each key	rdd.countByKey()	{(1, 1), (3, 2)}
collectAsMap()	Collect the result as a map to provide easy lookup	rdd.collectAsMap()	Map{(1, 2), (3, 4), (3, 6)}
lookup(key)	Return all values associated with the provided key	rdd.lookup(3)	[4, 6]

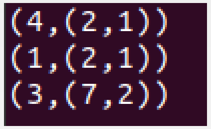
A Few Practices on Pair RDD

```
val lines = sc.parallelize(List("hello world", "this is a scala program", "to create a pair RDD", "in spark"))
val pairs = lines.map(x => (x.split(" ")(0), x))
pairs.filter {case (key, value) => key.length <3}.foreach(println)
```



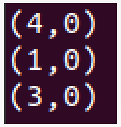
```
(to,to create a pair RDD)
(in,in spark)
```

```
val pairs = sc.parallelize(List((1, 2), (3, 1), (3, 6), (4,2)))
val pairs1 = pairs.mapValues(x=>(x, 1))
val pairs2 = pairs1.reduceByKey((x,y) => (x._1 + y._1, x._2+y._2))
pairs2.foreach(println)
```



```
(4,(2,1))
(1,(2,1))
(3,(7,2))
```

```
val pairs = sc.parallelize(List((1, 2), (3, 4), (3, 9), (4,2)))
val pairs1 = pairs.mapValues(x=>(x, 1)).reduceByKey((x,y) => (x._1 + y._1, x._2+y._2)).mapValues(x=>x._2/x._1)
pairs1.foreach(println)
```



```
(4,0)
(1,0)
(3,0)
```

Passing Functions to RDD

- Spark’s API relies heavily on passing functions in the driver program to run on the cluster.
  - Anonymous function. E.g.,
    - val words = input.flatMap(line => line.split(" "))
  - Static methods in a global singleton object. E.g,
    - object MyFunctions { def func1(s: String): String = { ... } }

      myRdd.map(MyFunctions.func1)

Understanding Closures

- RDD operations that modify variables outside of their scope can be a frequent source of confusion.
- Consider the naive RDD element sum below, which may behave differently depending on whether execution is happening within the same JVM. A common example of this is when running Spark in local mode (--master = local[n]) versus deploying a Spark application to a cluster (e.g. via spark-submit to YARN):

```
var counter = 0
var rdd = sc.parallelize(data)
rdd.foreach(x => counter += x)
println("Counter value: " + counter)
```

  - The behavior of the above code is undefined, and may not work as intended.
  - Spark sends the closure to each task containing variables must be visible to the executors. Thus “counter” in the executor is only a copy of the “counter” in the driver.

Load Your Data

- File formats range from unstructured, like text, to semi-structured, like JSON, to structured, like SequenceFiles.
- Text File:
  - input = sc.textFile("file:///home/holden/repos/spark/README.md")
- CSV File:
  - You can use csv libraries such as opencsv:

```
import Java.io.StringReader
import au.com.bytecode.opencsv.CSVReader

val input = sc.textFile(inputFile)
val result = input.map{ line =>
   val reader = new CSVReader(new StringReader(line));
   reader.readNext();
}
```

  - If you know the field separator in advance, you can also split each record into columns using the separator such as “,”

Save Your Data

- Text File:
  - result.saveAsTextFile(outputFile)
- CSV File:
  - You can use StringWriter/StringIO to allow us to put the result in our RDD

```
pandaLovers.map(person => List(person.name,
person.favoriteAnimal).toArray)
.mapPartitions{people =>
   val stringWriter = new StringWriter();
   val csvWriter = new CSVWriter(stringWriter);
   csvWriter.writeAll(people.toList)
   Iterator(stringWriter.toString)
}.saveAsTextFile(outFile)
```

  - You can also convert each record to a string with the fields separated by a separator such as “,”, and then save to a text file.

Setting the Level of Parallelism

- All the pair RDD operations take an optional second parameter for number of tasks

```
words.reduceByKey((x,y) => x + y, 5)
words.groupByKey(5)
```

Using Local Variables

- Any external variables you use in a closure will automatically be shipped to the cluster:

```
query = sys.stdin.readline()
pages.filter(x => x.contains(query)).count()
```

- Some caveats:
  - Each task gets a new copy (updates aren’t sent back)
  - Variable must be Serializable

Shared Variables

- When you perform transformations and actions that use functions (e.g., map(f: T=>U)), Spark will automatically push a closure containing that function to the workers so that it can run at the workers.

- Any variable or data within a closure or data structure will be distributed to the worker nodes along with the closure

- When a function (such as map or reduce) is executed on a cluster node, it works on **separate** copies of all the variables used in it.

- Usually these variables are just constants but they cannot be shared across workers efficiently.

Shared Variables

- Consider These Use Cases
  - Iterative or single jobs with large global variables
    - Sending large read-only lookup table to workers
    - Sending large feature vector in a ML algorithm to workers
    - Problems? Inefficient to send large data to each worker with each iteration
    - Solution: Broadcast variables
  - Counting events that occur during job execution
    - How many input lines were blank?
    - How many input records were corrupt?
    - Problems? Closures are one way: driver -> worker
    - Solution: Accumulators

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.
  - For example, to give every node a copy of a large input dataset efficiently
- Spark also attempts to distribute broadcast variables using efficient broadcast algorithms to reduce communication cost
- Broadcast variables are created from a variable **v** by calling **SparkContext.broadcast(v)**. Its value can be accessed by calling the **value** method.

```
scala > val broadcastVar =sc.broadcast(Array(1, 2, 3))
broadcastVar: org.apache.spark.broadcast.Broadcast[Array[Int]] = Broadcast(0)
scala > broadcastVar.value
res0: Array[Int] = Array(1, 2, 3)
```

- The broadcast variable should be used instead of the value **v** in any functions run on the cluster, so that **v** is not shipped to the nodes more than once.

Accumulators

- Accumulators are variables that are only “added” to through an associative and commutative operation and can therefore be efficiently supported in parallel.
- They can be used to implement counters (as in MapReduce) or sums.
- Spark natively supports accumulators of numeric types, and programmers can add support for new types.
- Only driver can read an accumulator’s value, not tasks
- An accumulator is created from an initial value **v** by calling **SparkContext.accumulator(v)**.

```
scala> val accum = sc.longAccumulator("My Accumulator")
accum: org.apache.spark.util.LongAccumulator = LongAccumulator(id: 0, name: Some(My Accumulator), value: 0)
scala> sc.parallelize(Array(1, 2, 3, 4)).foreach(x => accum.add(x))
... 10/09/29 18:41:08 INFO SparkContext: Tasks finished in 0.317106 s
scala> accum.value
res2: Long = 10
```

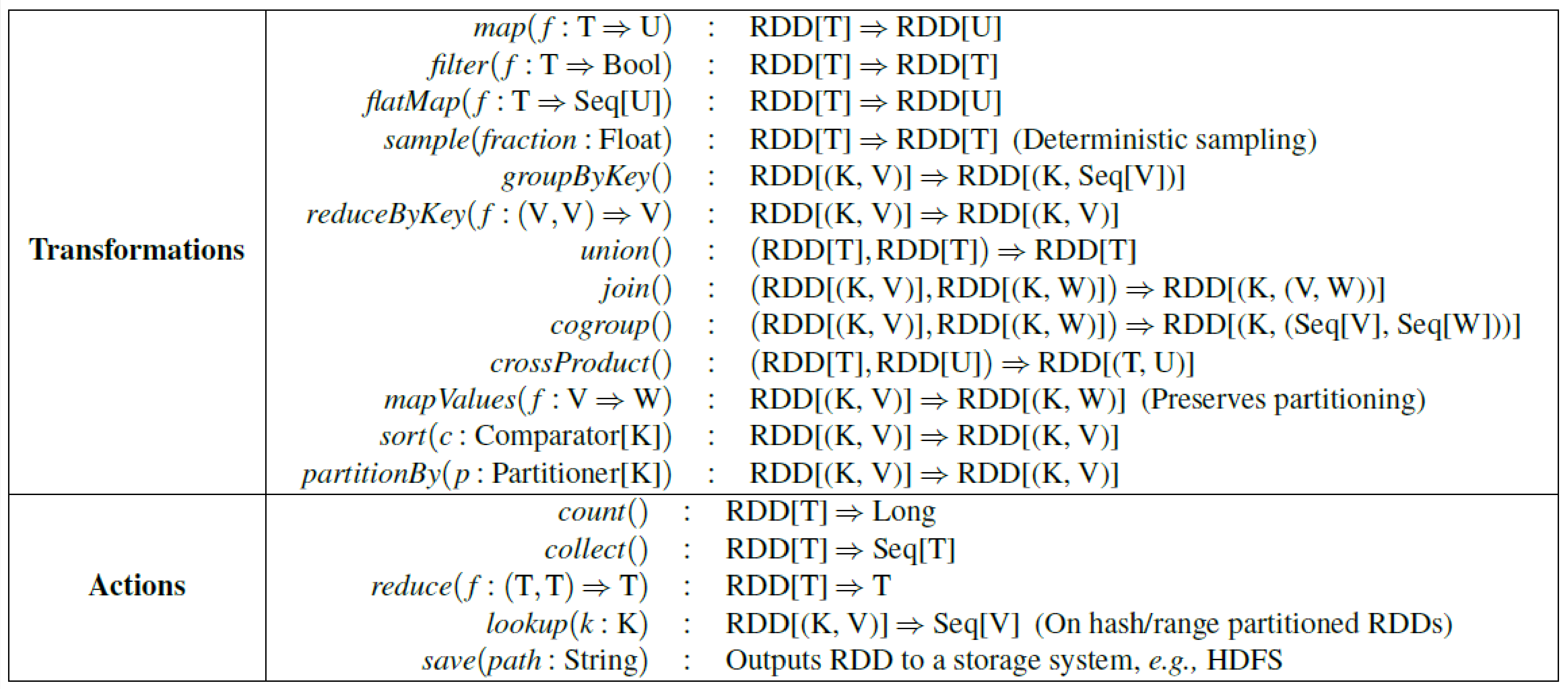
Accumulators Example (Python)

- Counting empty lines

```
file = sc.textFile(inputFile)
# Create Accumulator[Int] initialized to 0
blankLines = sc.accumulator(0)

def extractCallSigns(line):
        global blankLines # Make the global variable accessible
        if (line == ""):
                blankLines += 1
        return line.split(" ")

callSigns = file.flatMap(extractCallSigns)
print ("Blank lines: %d" % blankLines.value)
```

- blankLines is created in the driver, and shared among workers
- Each worker can access this variable

RDD Operations

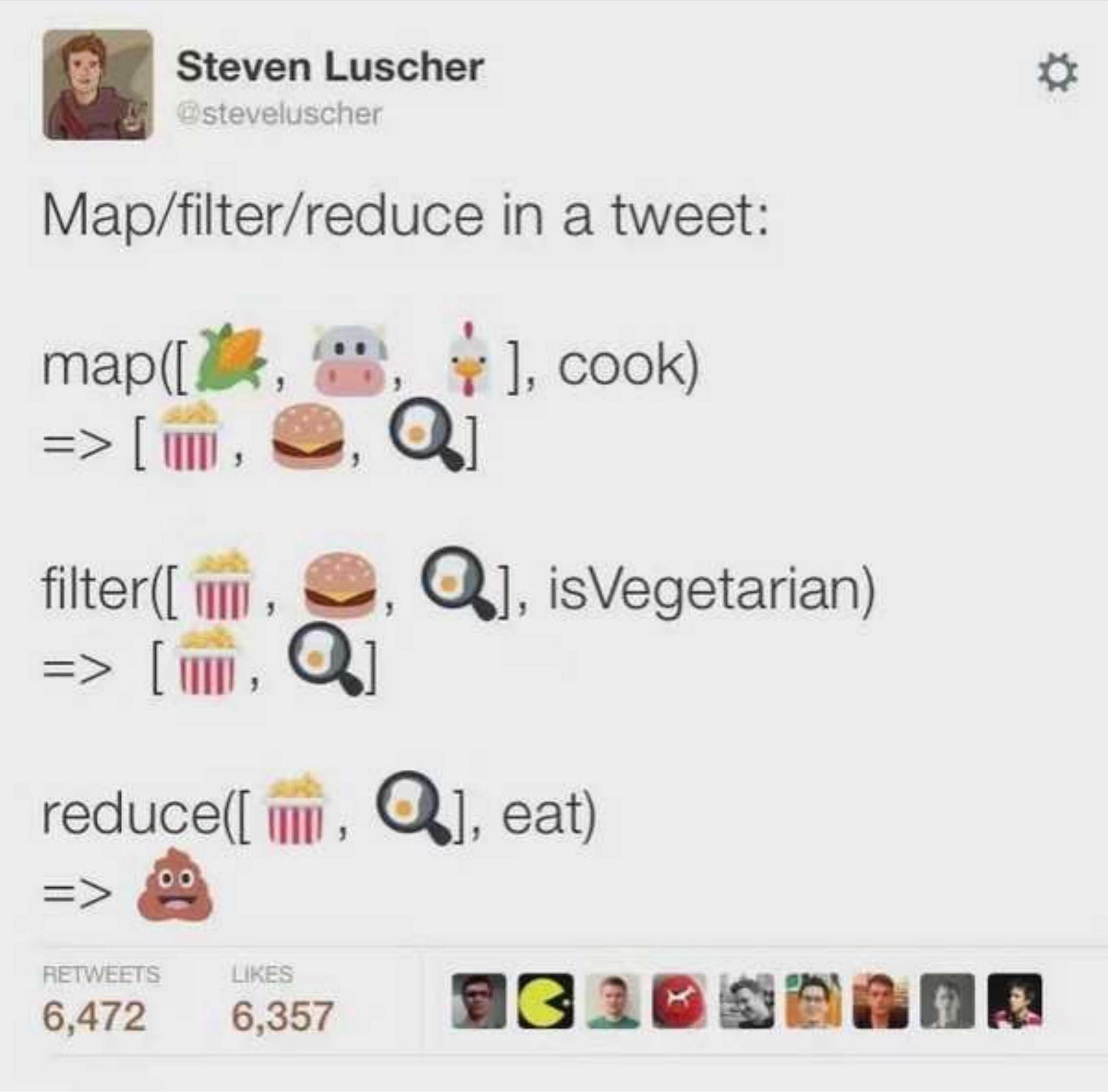


Transformations	$\mathit{map}(f : \mathrm{T} \Rightarrow \mathrm{U})$	:	$\mathrm{RDD[T]} \Rightarrow \mathrm{RDD[U]}$
	$\mathit{filter}(f : \mathrm{T} \Rightarrow \mathrm{Bool})$	:	$\mathrm{RDD[T]} \Rightarrow \mathrm{RDD[T]}$
	$\mathit{flatMap}(f : \mathrm{T} \Rightarrow \mathrm{Seq[U]})$	:	$\mathrm{RDD[T]} \Rightarrow \mathrm{RDD[U]}$
	$\mathit{sample}(\mathit{fraction} : \mathrm{Float})$	:	$\mathrm{RDD[T]} \Rightarrow \mathrm{RDD[T]}$ (Deterministic sampling)
	$\mathit{groupByKey}()$	:	$\mathrm{RDD[(K, V)]} \Rightarrow \mathrm{RDD[(K, Seq[V])]}$
	$\mathit{reduceByKey}(f : (\mathrm{V}, \mathrm{V}) \Rightarrow \mathrm{V})$	:	$\mathrm{RDD[(K, V)]} \Rightarrow \mathrm{RDD[(K, V)]}$
	$\mathit{union}()$	:	$(\mathrm{RDD[T]}, \mathrm{RDD[T]}) \Rightarrow \mathrm{RDD[T]}$
	$\mathit{join}()$	:	$(\mathrm{RDD[(K, V)]}, \mathrm{RDD[(K, W)]}) \Rightarrow \mathrm{RDD[(K, (V, W))]}$
	$\mathit{cogroup}()$	:	$(\mathrm{RDD[(K, V)]}, \mathrm{RDD[(K, W)]}) \Rightarrow \mathrm{RDD[(K, (Seq[V], Seq[W]))]}$
	$\mathit{crossProduct}()$	:	$(\mathrm{RDD[T]}, \mathrm{RDD[U]}) \Rightarrow \mathrm{RDD[(T, U)]}$
	$\mathit{mapValues}(f : \mathrm{V} \Rightarrow \mathrm{W})$	:	$\mathrm{RDD[(K, V)]} \Rightarrow \mathrm{RDD[(K, W)]}$ (Preserves partitioning)
	$\mathit{sort}(c : \mathrm{Comparator[K]})$	:	$\mathrm{RDD[(K, V)]} \Rightarrow \mathrm{RDD[(K, V)]}$
	$\mathit{partitionBy}(p : \mathrm{Partitioner[K]})$	:	$\mathrm{RDD[(K, V)]} \Rightarrow \mathrm{RDD[(K, V)]}$
Actions	$\mathit{count}()$	:	$\mathrm{RDD[T]} \Rightarrow \mathrm{Long}$
	$\mathit{collect}()$	:	$\mathrm{RDD[T]} \Rightarrow \mathrm{Seq[T]}$
	$\mathit{reduce}(f : (\mathrm{T}, \mathrm{T}) \Rightarrow \mathrm{T})$	:	$\mathrm{RDD[T]} \Rightarrow \mathrm{T}$
	$\mathit{lookup}(k : \mathrm{K})$	:	$\mathrm{RDD[(K, V)]} \Rightarrow \mathrm{Seq[V]}$ (On hash/range partitioned RDDs)
	$\mathit{save}(\mathit{path} : \mathrm{String})$	:	Outputs RDD to a storage system, *e.g.*, HDFS

Spark RDD API Examples:

http://homepage.cs.latrobe.edu.au/zhe/ZhenHeSparkRDDAPIExamples.html

Spark



Steven Luscher
@steveluscher

Map/filter/reduce in a tweet:

```
map([🌽, 🐮, 🐔], cook)
=> [🍿, 🍔, 🍳]

filter([🍿, 🍔, 🍳], isVegetarian)
=>  [🍿, 🍳]

reduce([🍿, 🍳], eat)
=> 💩
```

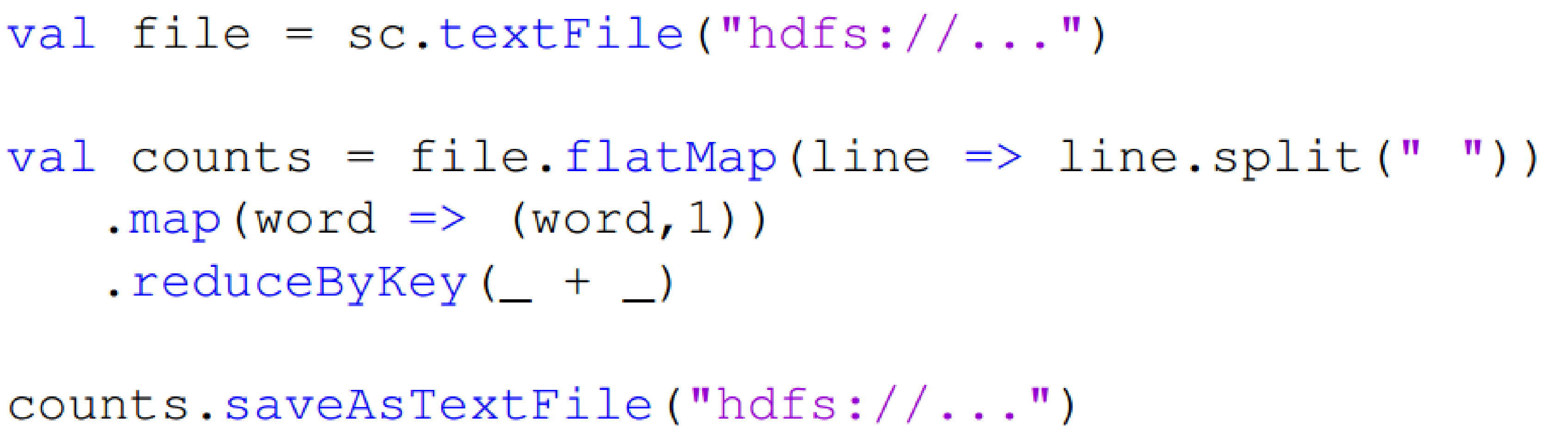
RETWEETS 6,472

LIKES 6,357

Part 2: Spark Programming Model (RDD)

How Spark Works

- User application create RDDs, transform them, and run actions.
- This results in a DAG (Directed Acyclic Graph) of operators.
- DAG is compiled into stages
- Each stage is executed as a series of Task (one Task for each Partition).

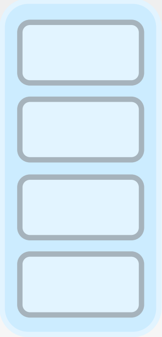


```
val file = sc.textFile("hdfs://...")

val counts = file.flatMap(line => line.split(" "))
    .map(word => (word,1))
    .reduceByKey(_ + _)

counts.saveAsTextFile("hdfs://...")
```

Word Count in Spark

```
val file = sc.textFile("hdfs://…", 4)
```

RDD[String]



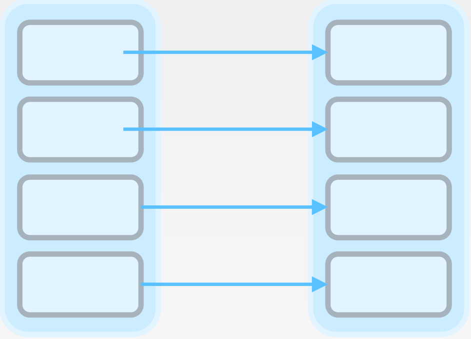
textFile

Word Count in Spark

```
val file = sc.textFile("hdfs://…", 4)
val words = file.flatMap(line =>
          line.split(" "))
```

RDD[String]

RDD[List[String]]



textFile

flatMap

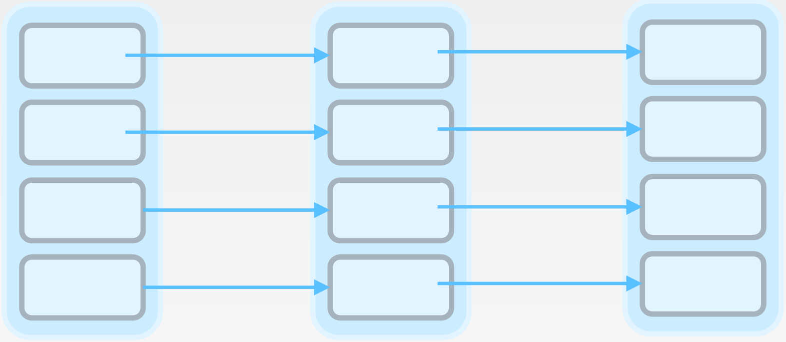
Word Count in Spark

```
val file = sc.textFile("hdfs://…", 4)
val words = file.flatMap(line =>
          line.split(" "))
val pairs = words.map(t => (t, 1))
```

RDD[String]

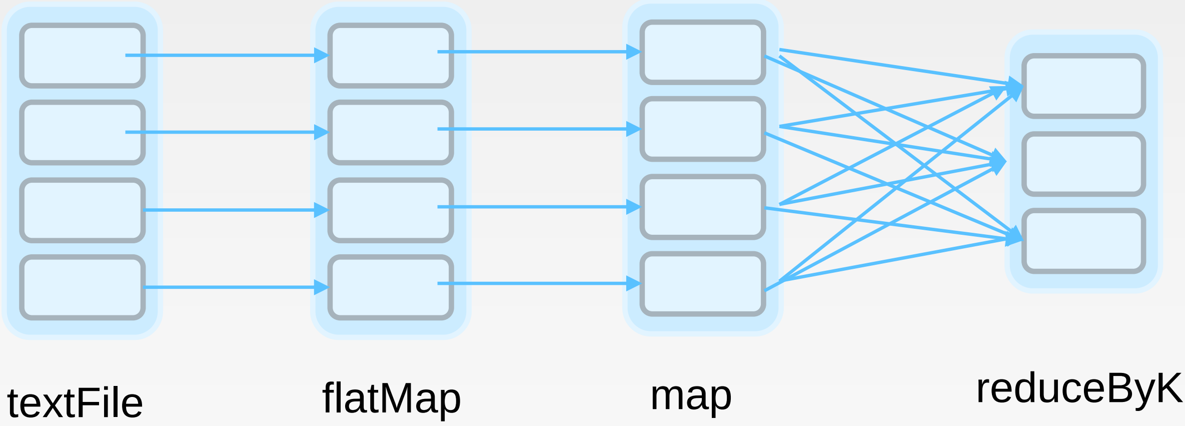
RDD[List[String]]

RDD[(String, Int)]



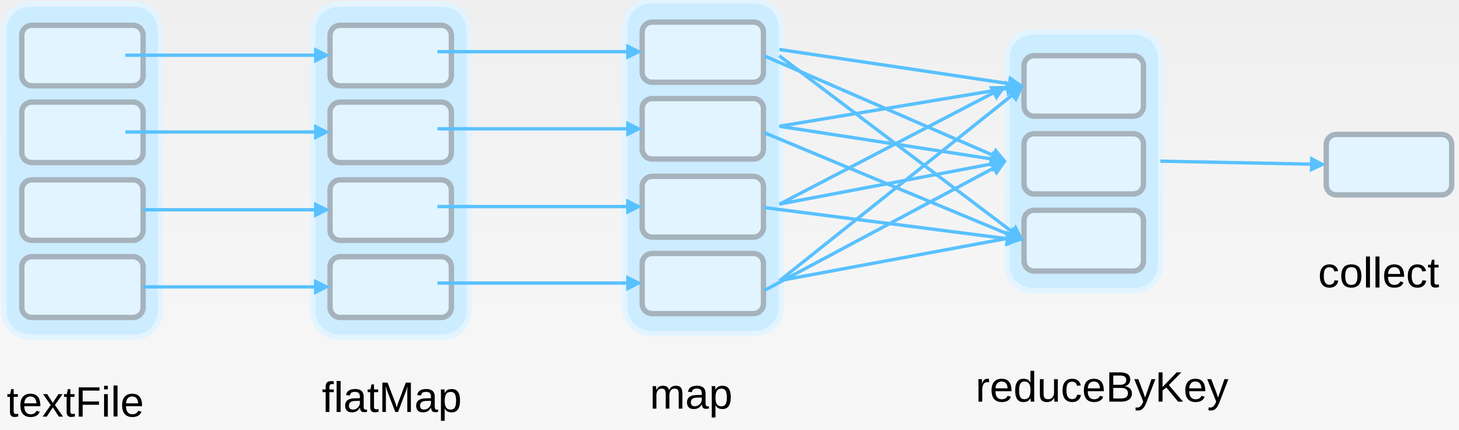
Word Count in Spark

```
val file = sc.textFile(“hdfs://…”, 4)          RDD[String]
val words = file.flatMap(line =>               RDD[List[String]]
          line.split(“ ”))
val pairs = words.map(t => (t, 1))             RDD[(String, Int)]
val count = pairs. reduceByKey(_+_)            RDD[(String, Int)]
```

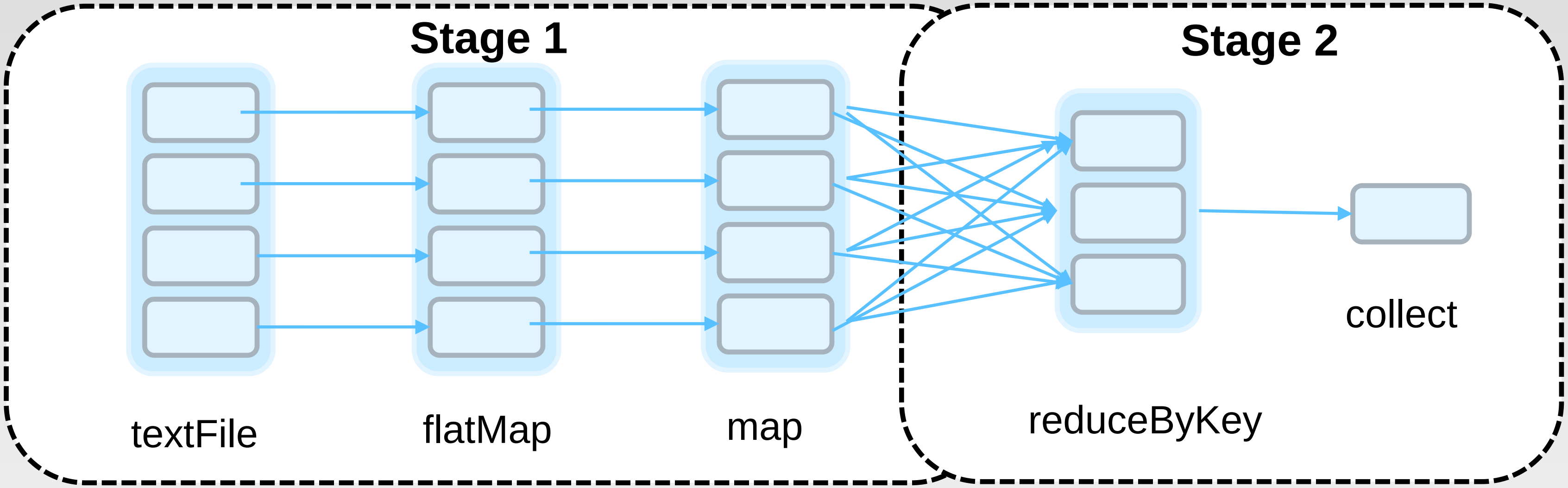


Word Count in Spark

```
val file = sc.textFile("hdfs://…", 4)           RDD[String]
val words = file.flatMap(line =>                RDD[List[String]]
          line.split(" "))
val pairs = words.map(t => (t, 1))              RDD[(String, Int)]
val count = pairs. reduceByKey(_+_)             RDD[(String, Int)]
count.collect()                                 Array[(String, Int)]
```

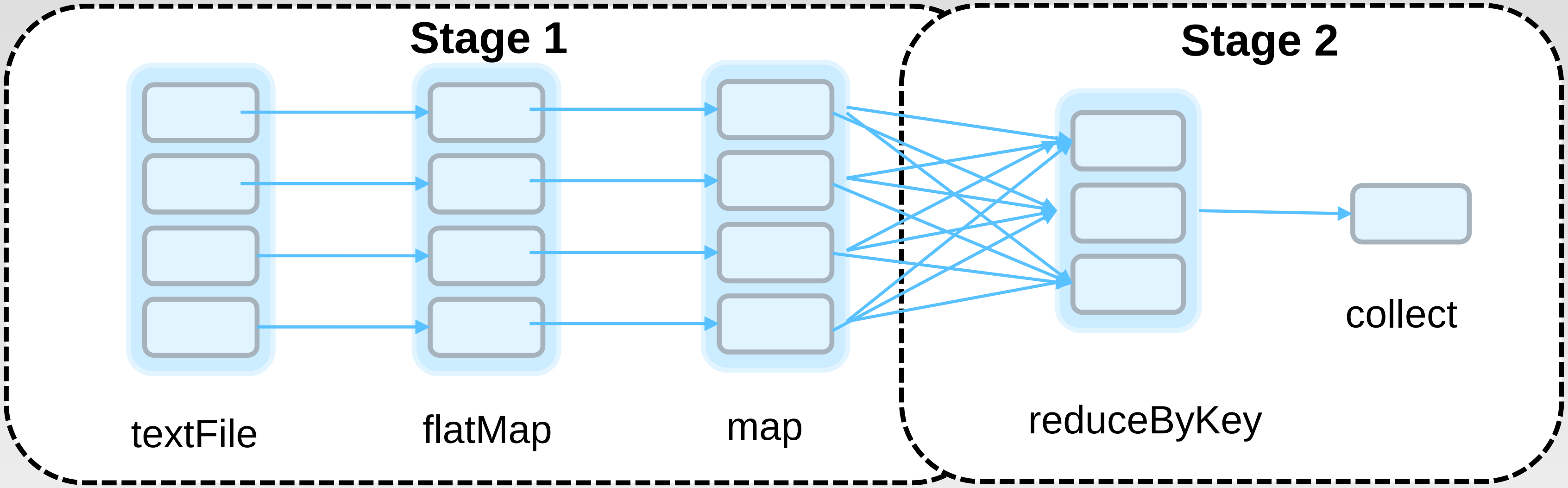


Execution Plan

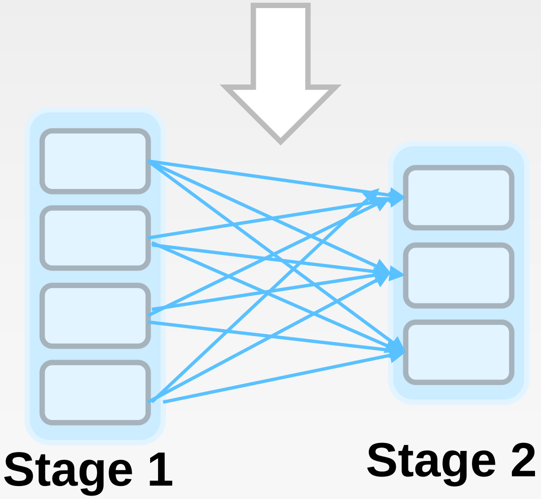


- The scheduler examines the RDD’s lineage graph to build a DAG of stages.
- Stages are sequences of RDDs, that don’t have a Shuffle in between
- The boundaries are the shuffle stages.

Execution Plan



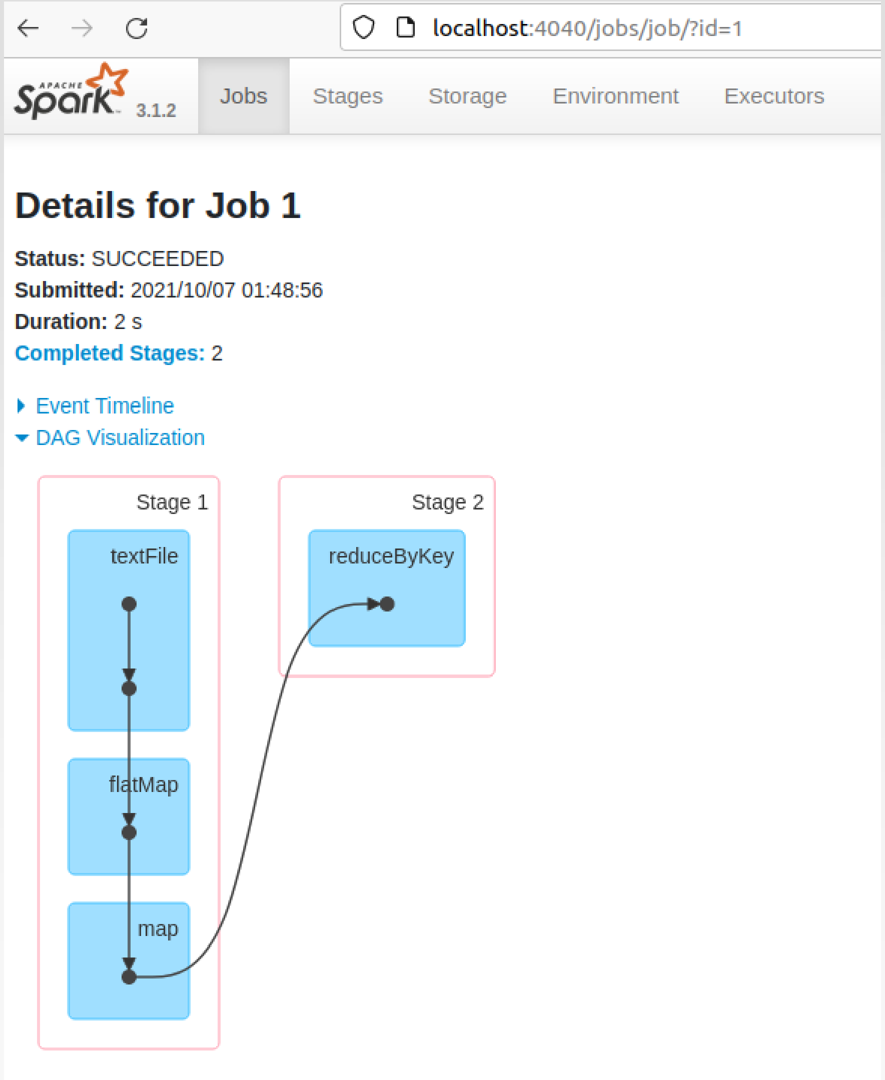
1. Read HDFS split
2. Apply both the maps
3. Start Partial reduce
4. Write shuffle data



1. Read shuffle data
2. Final reduce
3. Send result to driver program

Spark Web Console

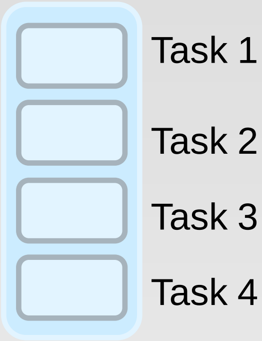
- You can browse the web interface for the information of Spark Jobs, storage, etc. at: **http://localhost:4040**



Details for Job 1

Status: SUCCEEDED
Submitted: 2021/10/07 01:48:56
Duration: 2 s
Completed Stages: 2

Stage Execution

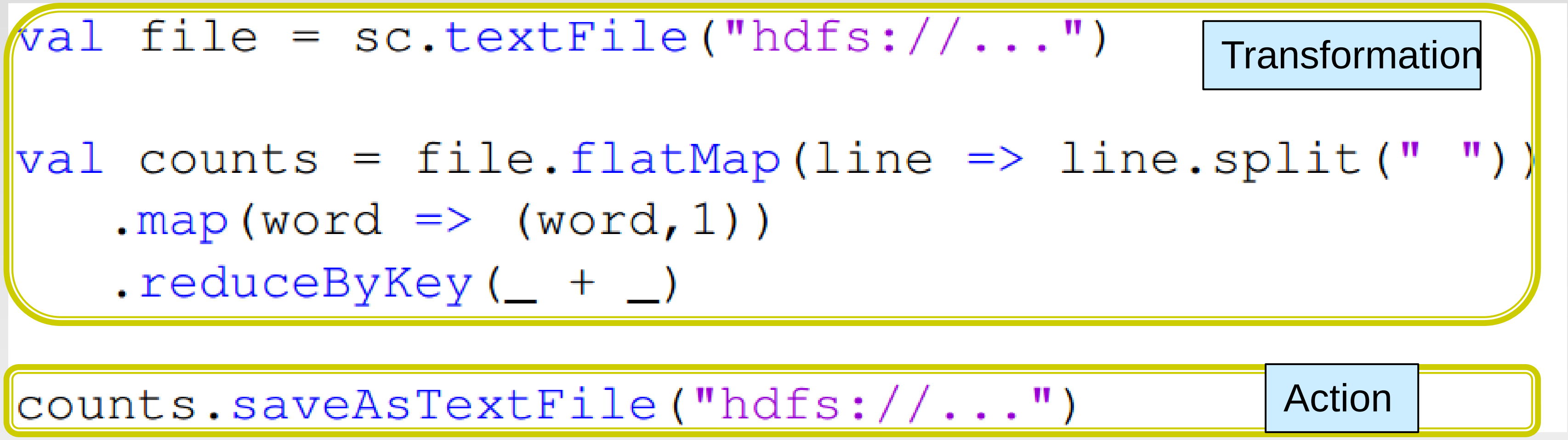


- Create a task for each Partition in the new RDD
- Serialize the Task
- Schedule and ship Tasks to Slaves

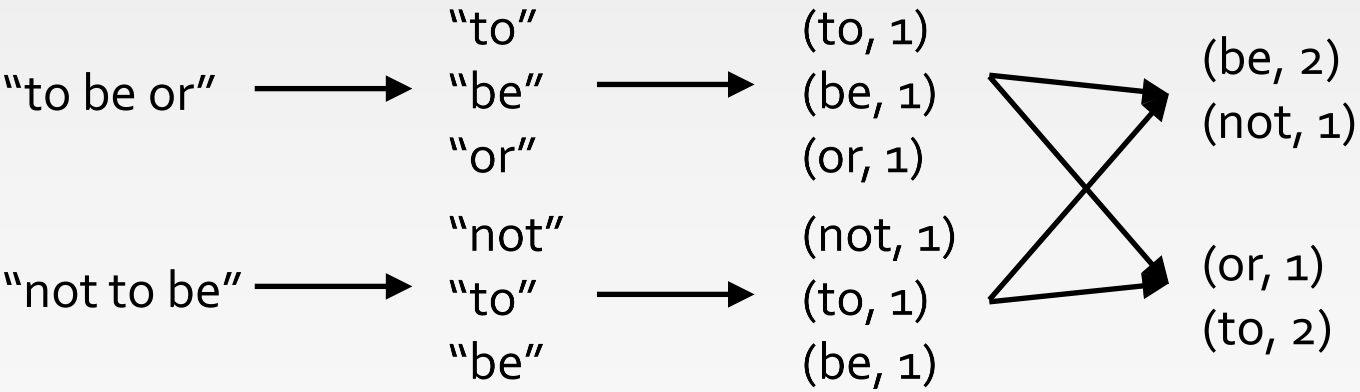
- All this happens internally

Word Count in Spark (As a Whole View)

- Word Count using Scala in Spark



```
val file = sc.textFile("hdfs://...")

val counts = file.flatMap(line => line.split(" "))
    .map(word => (word,1))
    .reduceByKey(_ + _)

counts.saveAsTextFile("hdfs://...")
```



"to be or" → "to" "be" "or" → (to, 1) (be, 1) (or, 1)

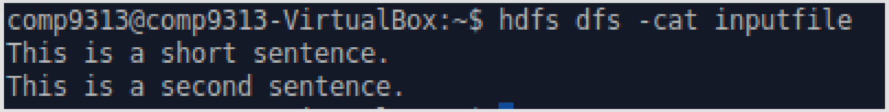
"not to be" → "not" "to" "be" → (not, 1) (to, 1) (be, 1)

(be, 2) (not, 1)

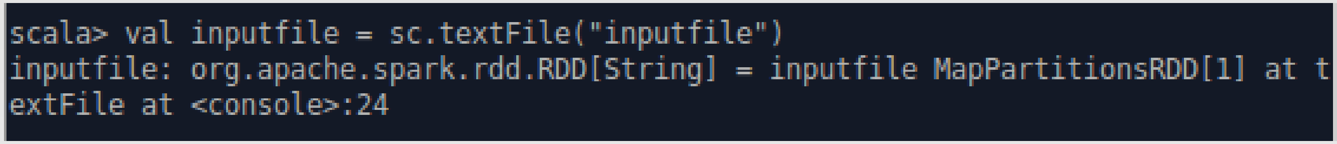
(or, 1) (to, 2)

map vs. flatMap

- Sample input file:

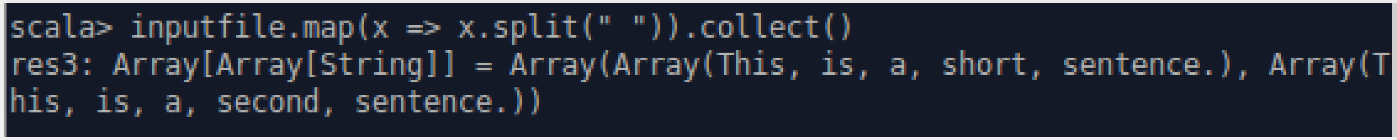


```
comp9313@comp9313-VirtualBox:~$ hdfs dfs -cat inputfile
This is a short sentence.
This is a second sentence.
```



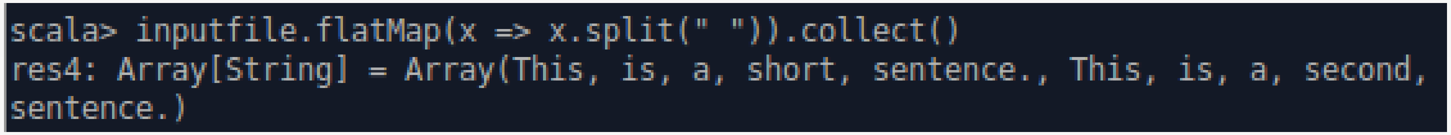
```
scala> val inputfile = sc.textFile("inputfile")
inputfile: org.apache.spark.rdd.RDD[String] = inputfile MapPartitionsRDD[1] at t
extFile at <console>:24
```

- map: Return a new distributed dataset formed by passing each element of the source through a function *func*.



```
scala> inputfile.map(x => x.split(" ")).collect()
res3: Array[Array[String]] = Array(Array(This, is, a, short, sentence.), Array(T
his, is, a, second, sentence.))
```

- flatMap: Similar to map, but each input item can be mapped to 0 or more output items (so *func* should return a Seq rather than a single item).



```
scala> inputfile.flatMap(x => x.split(" ")).collect()
res4: Array[String] = Array(This, is, a, short, sentence., This, is, a, second,
sentence.)
```

Part 3: Running on a Cluster

WordCount (RDD, Scala)

- Standalone code

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

object WordCount {
    def main(args: Array[String]) {
      val inputFile = args(0)
      val outputFolder = args(1)
      val conf = new SparkConf().setAppName(“wordCount”).setMaster(“local”)
      // Create a Scala Spark Context.
      val sc = new SparkContext(conf)
      // Load our input data.
      val input =  sc.textFile(inputFile)
      // Split up into words.
      val words = input.flatMap(line => line.split(" "))
      // Transform into word and count.
      val counts = words.map(word => (word, 1)).reduceByKey(_+_)
      counts.saveAsTextFile(outputFolder)
    }
}
```

WordCount (RDD, Scala)

- Linking with Apache Spark
  - The first step is to explicitly import the required spark classes into your Spark program

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

- Initializing Spark
  - Create a Spark context object with the desired spark configuration that tells Apache Spark on how to access a cluster

```
val conf = new SparkConf().setAppName(“wordCount”).setMaster(“local”)
val sc = new SparkContext(conf)
```

  - SparkConf: Spark configuration class
  - setAppName: set the name for your application
  - setMaster: set the cluster master URL

setMaster

- Set the cluster master URL to connect to
- Parameters for setMaster:
  - local(default) - run locally with only one worker thread (no parallel)
  - local[k] - run locally with k worker threads
  - spark://HOST:PORT - connect to Spark standalone cluster URL
  - mesos://HOST:PORT - connect to Mesos cluster URL
  - yarn - connect to Yarn cluster URL
    - Specified in SPARK_HOME/conf/yarn-site.xml
- setMaster parameters configurations:
  - In source code
    - SparkConf().setAppName(“wordCount”).setMaster(“local”)
  - spark-submit
    - spark-submit --master local
  - In SPARK_HOME/conf/spark-default.conf
    - Set value for spark.master

WordCount (RDD, Scala)

- Creating a Spark RDD
  - Create an input Spark RDD that reads the text file input.txt using the Spark Context created in the previous step

```
val input =  sc.textFile(inputFile)
```

- Spark RDD Transformations in Wordcount Example
  - flatMap() is used to tokenize the lines from input text file into words
  - map() method counts the frequency of each word
  - reduceByKey() method counts the repetitions of word in the text file
- Save the results to disk

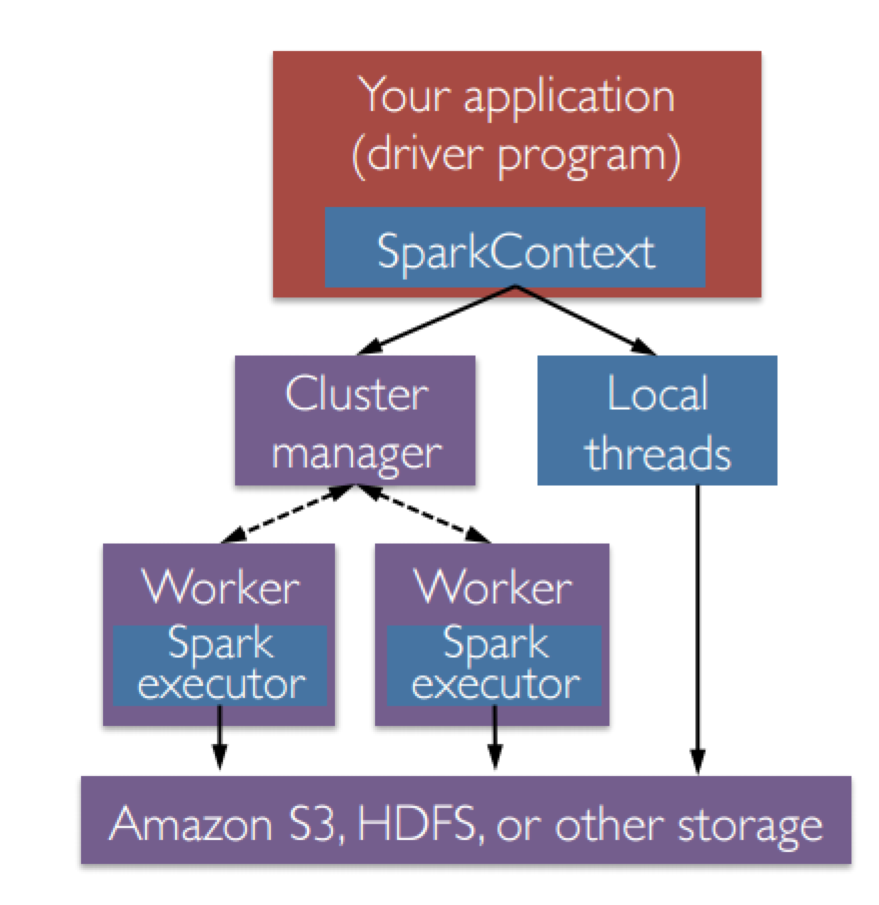
```
counts.saveAsTextFile(outputFolder)
```

Run the Application on a Cluster

- A Spark application is launched on a set of machines using an external service called a cluster manager
  - Local threads
  - Standalone
  - Mesos
  - Yarn

- Driver

- Executor



Launching a Program

- Spark provides a single script you can use to submit your program to it called spark-submit
  - The user submits an application using spark-submit
  - spark-submit launches the driver program and invokes the main() method specified by the user
  - The driver program contacts the cluster manager to ask for resources to launch executors
  - The cluster manager launches executors on behalf of the driver program
  - The driver process runs through the user application. Based on the RDD actions and transformations in the program, the driver sends work to executors in the form of tasks
  - Tasks are run on executor processes to compute and save results
  - If the driver's main() method exits or it calls SparkContext.stop(), it will terminate the executors and release resources from the cluster manager

Package Your Code and Dependencies

- Ensure that all your dependencies are present at the runtime of your Spark application
- Java Application (Maven)
- Scala Application (sbt)
  - a newer build tool most often used for Scala projects

```
name := "Simple Project"
version := "1.0"
scalaVersion := "2.12.10"
libraryDependencies += "org.apache.spark" %% "spark-
core" % “3.1.2"
```

  - libraryDependencies: list all dependent libraries (including third party libraries)
  - A jar file simple-project_2.12-1.0.jar will be created after compilation

Deploying Applications in Spark

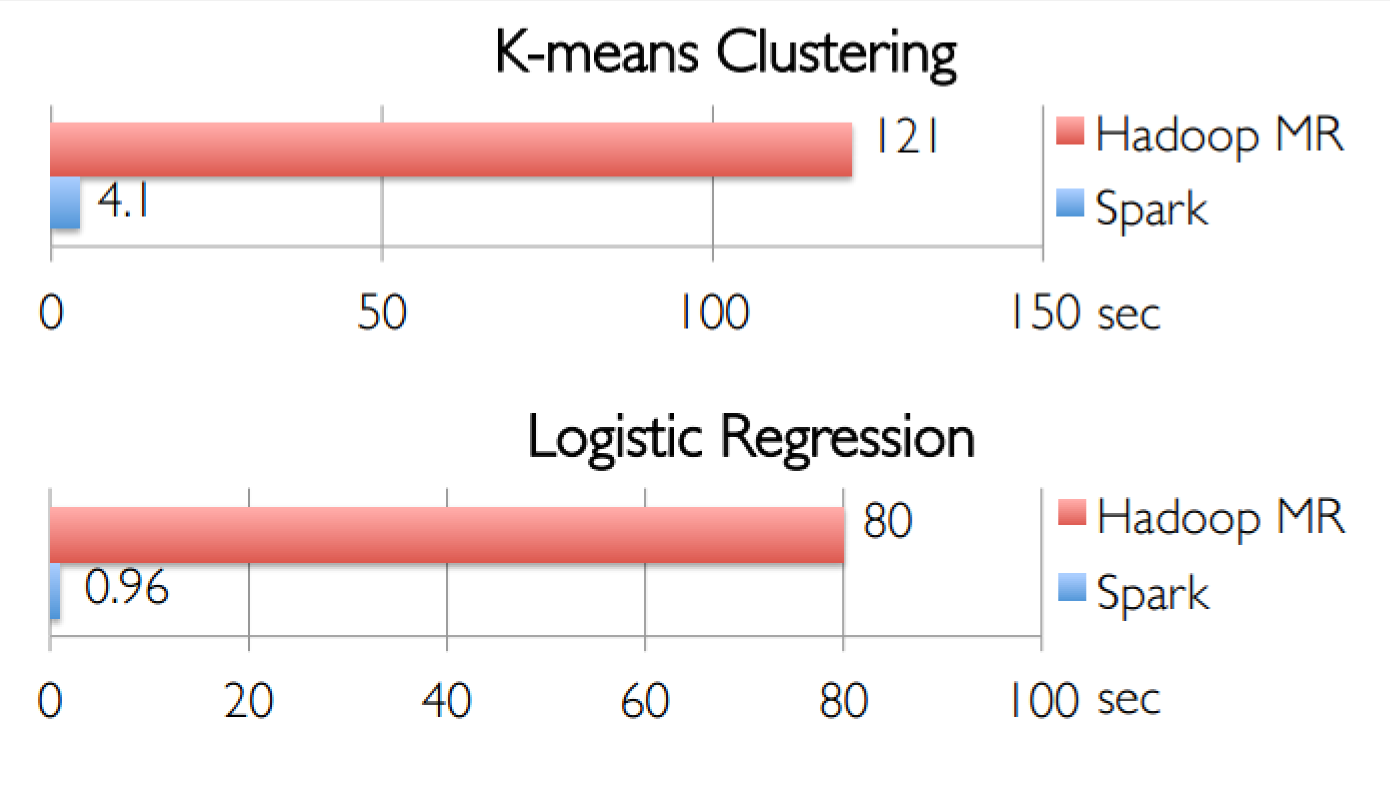
- spark-submit

Common flags	Explanation
--master	Indicates the cluster manager to connect to
--class	The “main” class of your application if you’re running a Java or Scala program
--name	A human-readable name for your application. This will be displayed in Spark’s web UI.
--executor-memory	The amount of memory to use for executors, in bytes. Suffixes can be used to specify larger quantities such as “512m” (512 megabytes) or “15g” (15 gigabytes)
--driver-memory	The amount of memory to use for the driver process, in bytes.

```
spark-submit --master spark://hostname:7077 \
                  --class YOURCLASS \
                   --executor-memory 2g \
                   YOURJAR "options" "to your application" "go here"
```

In-Memory Can Make a Big Difference

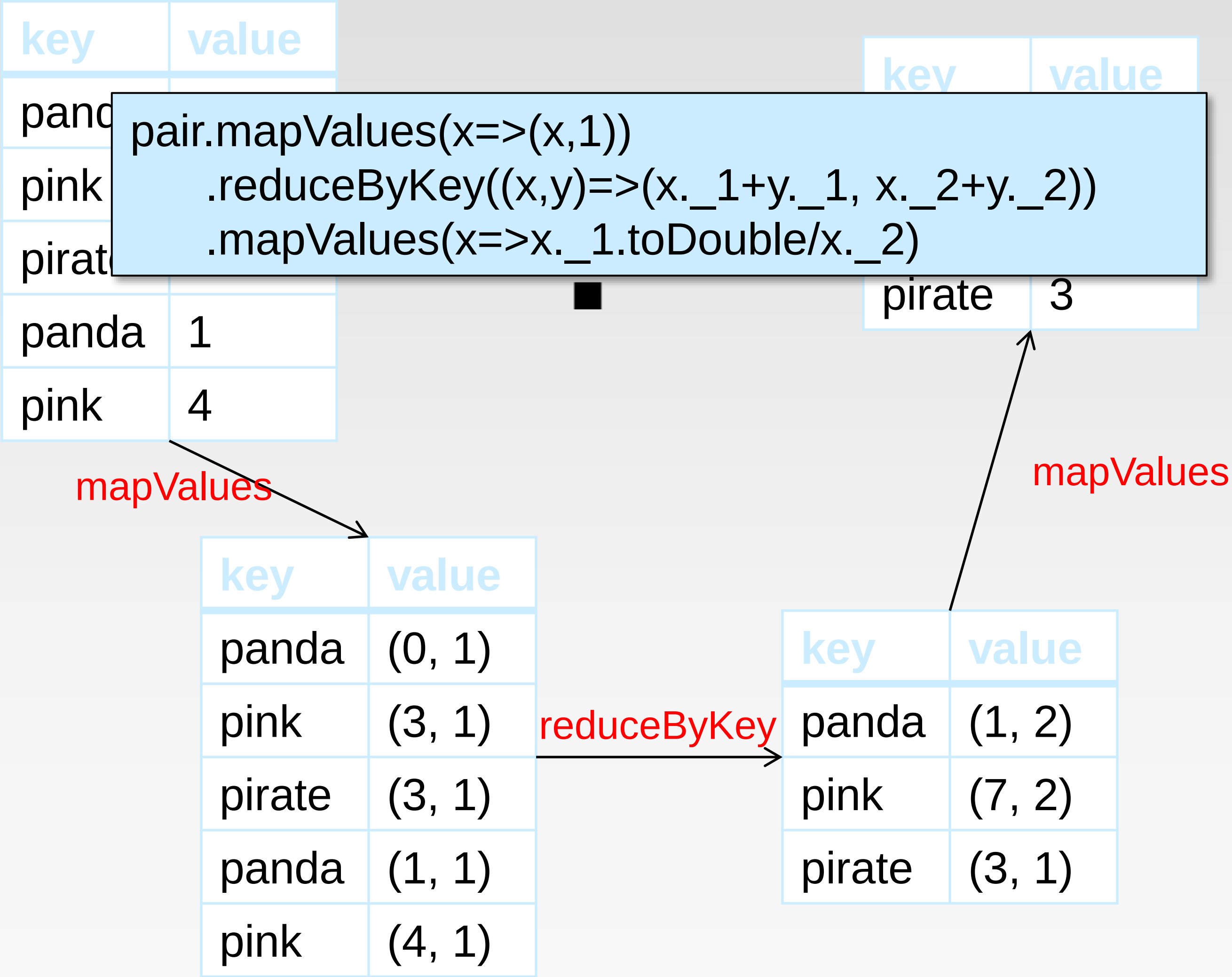
- Two iterative Machine Learning algorithms:



Spark Core Programming Practice

Practice

- Problem 1: Given a pair RDD of type [(String, Int)], compute the per-key average



key	value
panc	
pink	
pirat	
panda	1
pink	4

```
pair.mapValues(x=>(x,1))
      .reduceByKey((x,y)=>(x._1+y._1, x._2+y._2))
      .mapValues(x=>x._1.toDouble/x._2)
```

key	value
panda	(0, 1)
pink	(3, 1)
pirate	(3, 1)
panda	(1, 1)
pink	(4, 1)

key	value
panda	(1, 2)
pink	(7, 2)
pirate	(3, 1)

key	value
pirate	3

Practice

- Problem 2: Given the data in format of key-value pairs <Int, Int>, find the maximum value for each key across all values associated with that key.

```
val pairs = sc. Parallelize(List((1, 2), (3, 4),… …))

val resMax = pairs.reduceByKey( (a, b) => if(a > b) a else b )

resMax.foreach(x => println(x._1, x._2))
```

Practice

- Problem 3: Given a collection of documents, compute the average length of words starting with each letter.

```
val textFile = sc.textFile(inputFile)
val words = textFile.flatMap(_.split(“ “).toLowerCase)

val counts = words.filter(x=> x.length >=1 && x.charAt(0)<=’z’ &&
                x.charAt(0)>=’a’).map(x=>(x.charAt(0), (x.length, 1)))

val avgLen = counts.reduceByKey((a, b)=>(a._1+b._1, a._2+b._2)).foreach(x=>(x._1,
                x._2._1.toDouble/x._2._2))

avgLen.foreach(x => println(x._1, x._2))
```

References

- http://spark.apache.org/docs/latest/index.html
- Learning Spark. 1st and 2nd edition

End of Chapter 4.2