COMP9313: Big Data Management

Lecturer: Xin Cao

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

Chapter 4.2: Spark II

Part 1: Programming with RDD

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*

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(lambda x, y: x+y)	{(1, 2), (3, 10)}
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(la mbda 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)}
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]

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)`

A Few Practices on Pair RDD

```
lines = sc.parallelize(["hello world", "this is a scala program", "to create a pair RDD", "in spark"])
pairs = lines.map(lambda x: (x.split(" ")[0], x))
pairs.filter(lambda x: len(x[0])<3).collect()
```

```
>>> pairs.filter(lambda x: len(x[0])<3).collect()
[('to', 'to create a pair RDD'), ('in', 'in spark')]
```

```
pairs = sc.parallelize([(1, 2), (3, 1), (3, 6), (4,2)])
pairs1 = pairs.mapValues(lambda x: (x, 1))
pairs2 = pairs1.reduceByKey(lambda x, y: (x[0] + y[0], x[1]+y[1]))
pairs2.foreach(lambda x: print(x))
```

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

```
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.,
    - (Scala) val words = input.flatMap(line => line.split(" "))
    - (Python) words = input.flatMap(lambda line: line.split(" "))
  - Static methods in a global singleton object. E.g,
    - (Scala) 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.
- The result could be different when running Spark in local mode (--master = local[n]) versus deploying a Spark application to a cluster (e.g. via spark-submit to YARN):

```
counter = 0
rdd = sc.parallelize(data)
rdd.foreach(lambda x: counter += x)
print("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.

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.

```
>>> broadcastVar = sc.broadcast([1, 2, 3])
<pyspark.broadcast.Broadcast object at 0x102789f10>
>>> broadcastVar.value
[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)**.

```
>>> accum = sc.accumulator(0)
>>> accum
Accumulator<id=0, value=0>
>>> sc.parallelize([1, 2, 3, 4]).foreach(lambda x: accum.add(x))...
10/09/29 18:41:08 INFO SparkContext: Tasks finished in 0.317106 s
>>> accum.value
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	*map*(*f* : T ⇒ U)	:	RDD[T] ⇒ RDD[U]
	filter(*f* : T ⇒ Bool)	:	RDD[T] ⇒ RDD[T]
	flatMap(*f* : T ⇒ Seq[U])	:	RDD[T] ⇒ RDD[U]
	sample(*fraction* : Float)	:	RDD[T] ⇒ RDD[T] (Deterministic sampling)
	groupByKey()	:	RDD[(K, V)] ⇒ RDD[(K, Seq[V])]
	reduceByKey(*f* : (V, V) ⇒ V)	:	RDD[(K, V)] ⇒ RDD[(K, V)]
	union()	:	(RDD[T], RDD[T]) ⇒ RDD[T]
	join()	:	(RDD[(K, V)], RDD[(K, W)]) ⇒ RDD[(K, (V, W))]
	cogroup()	:	(RDD[(K, V)], RDD[(K, W)]) ⇒ RDD[(K, (Seq[V], Seq[W]))]
	crossProduct()	:	(RDD[T], RDD[U]) ⇒ RDD[(T, U)]
	mapValues(*f* : V ⇒ W)	:	RDD[(K, V)] ⇒ RDD[(K, W)] (Preserves partitioning)
	sort(*c* : Comparator[K])	:	RDD[(K, V)] ⇒ RDD[(K, V)]
	partitionBy(*p* : Partitioner[K])	:	RDD[(K, V)] ⇒ RDD[(K, V)]
Actions	*count*()	:	RDD[T] ⇒ Long
	collect()	:	RDD[T] ⇒ Seq[T]
	reduce(*f* : (T, T) ⇒ T)	:	RDD[T] ⇒ T
	lookup(*k* : K)	:	RDD[(K, V)] ⇒ Seq[V] (On hash/range partitioned RDDs)
	save(*path* : String)	:	Outputs RDD to a storage system, *e.g.*, HDFS

Spark RDD API Examples（Scala):
http://homepage.cs.latrobe.edu.au/zhe/ZhenHeSparkRDDAPIExamples.html

Spark RDD API Reference（Python):
https://spark.apache.org/docs/latest/api/python/reference/api/pyspark.RDD.html?highlight=pyspark%20rdd

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).

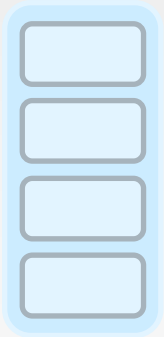
```
textfile = sc.textFile("hdfs://...", 4)

words = textfile.flatMap(lambda line: line.split(" "))
pairs = words.map(lambda word: (word, 1))
count = pairs.reduceByKey(lambda a, b: a + b)
count.collect()
```

Word Count in Spark

textfile = sc.textFile(“hdfs://…”, 4)

RDD[String]



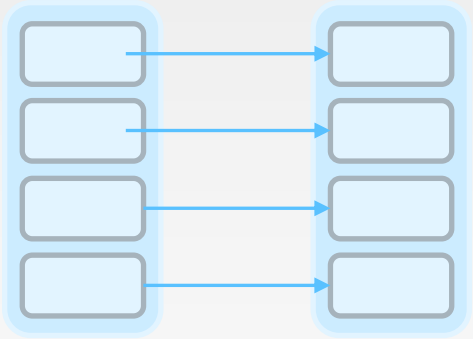
textFile

Word Count in Spark

```
textfile = sc.textFile("hdfs://…", 4)
words = text.flatMap(lambda line: line.split())
```

RDD[String]

RDD[List[String]]



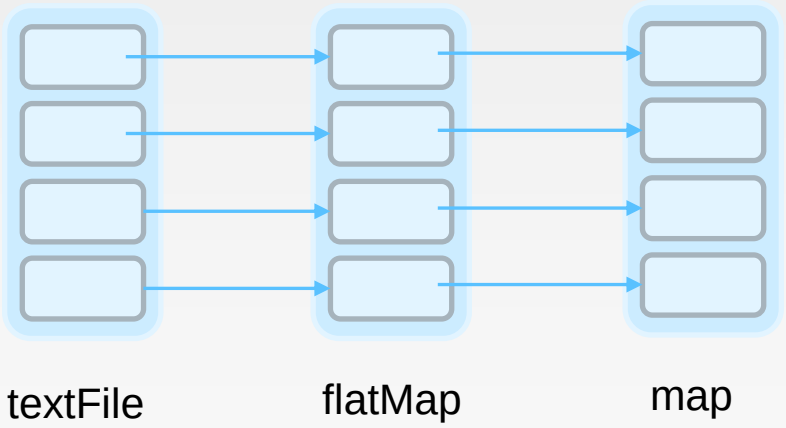
Word Count in Spark

```
textfile = sc.textFile("hdfs://…", 4)
words = text.flatMap(lambda line: line.split())
pairs = words.map(lambda word: (word, 1))
```

RDD[String]

RDD[String]

RDD[(String, Int)]



Word Count in Spark

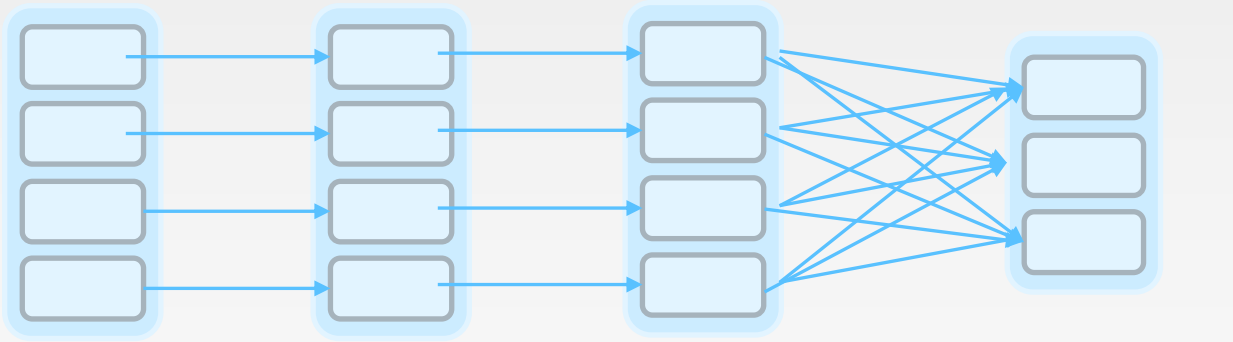
```
textfile = sc.textFile("hdfs://…", 4)
words = text.flatMap(lambda line: line.split())
pairs = words.map(lambda word: (word, 1))
count = pairs.reduceByKey(lambda a, b:
            a+b)
```

RDD[String]

RDD[String]

RDD[(String, Int)]

RDD[(String, Int)]



Word Count in Spark

```
textfile = sc.textFile("hdfs://…", 4)
words = text.flatMap(lambda line: line.split())
pairs = words.map(lambda word: (word, 1))
count = pairs.reduceByKey(lambda a, b:
            a+b)
count.collect()
```

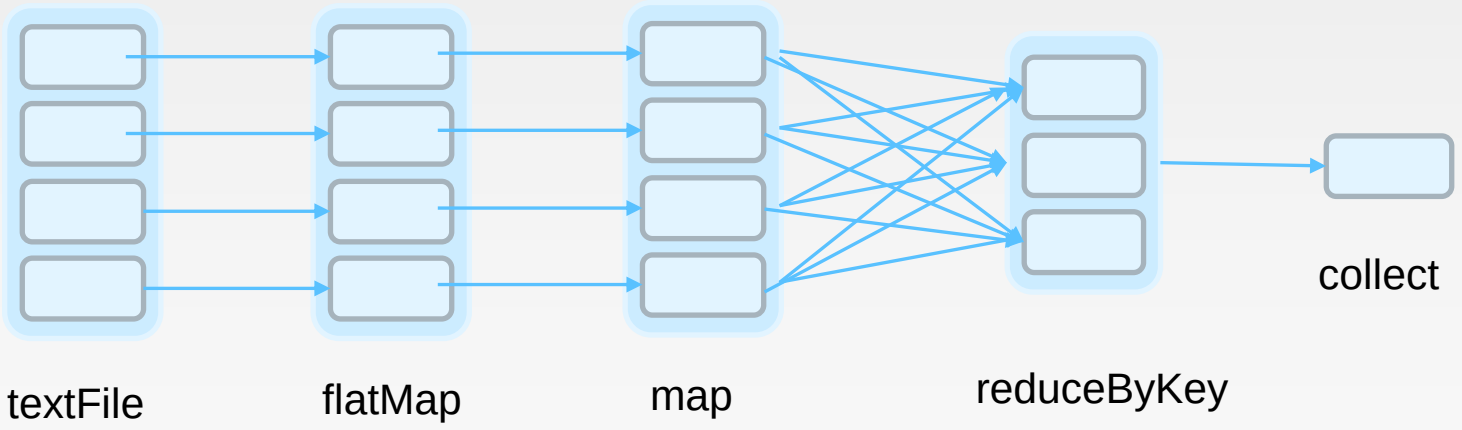
RDD[String]

RDD[String]

RDD[(String, Int)]

RDD[(String, Int)]

Array[(String, Int)]



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
```

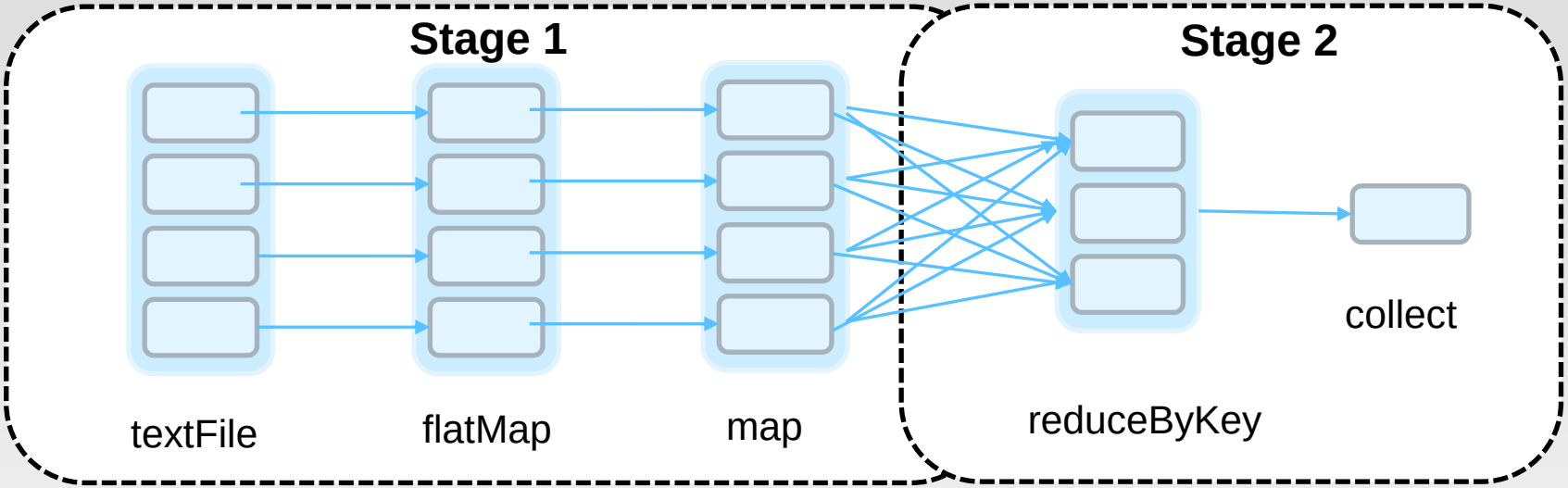
- map: Return a new distributed dataset formed by passing each element of the source through a function *func*.

```
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).

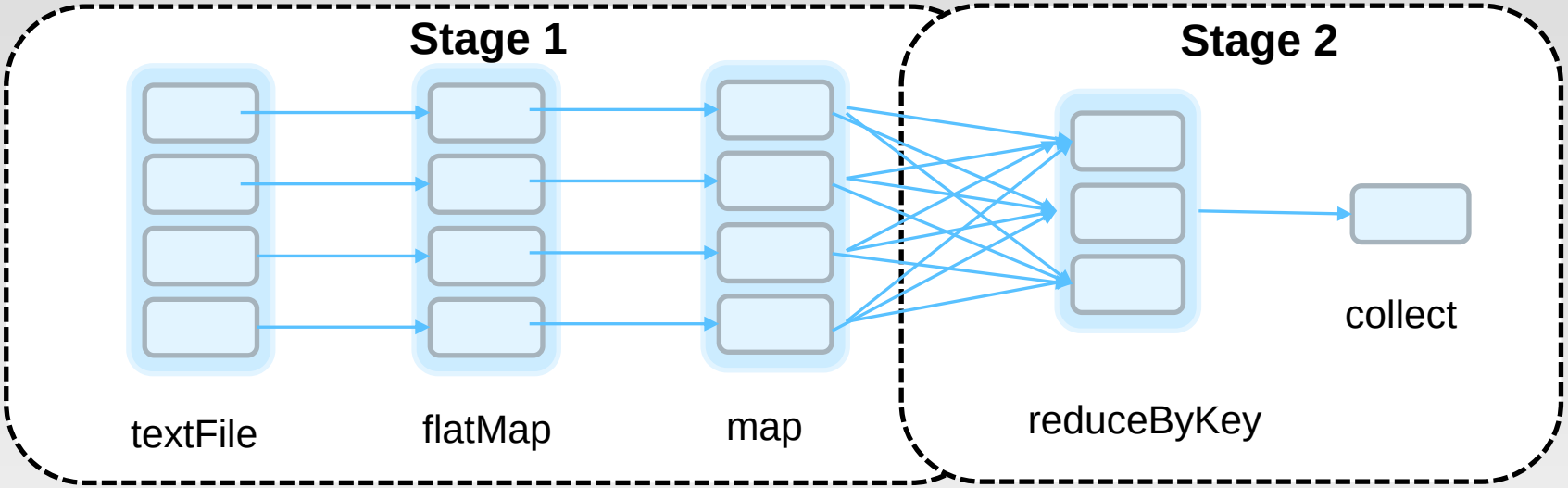
```
scala> inputfile.flatMap(x => x.split(" ")).collect()
res4: Array[String] = Array(This, is, a, short, sentence., This, is, a, second,
sentence.)
```

Execution Plan

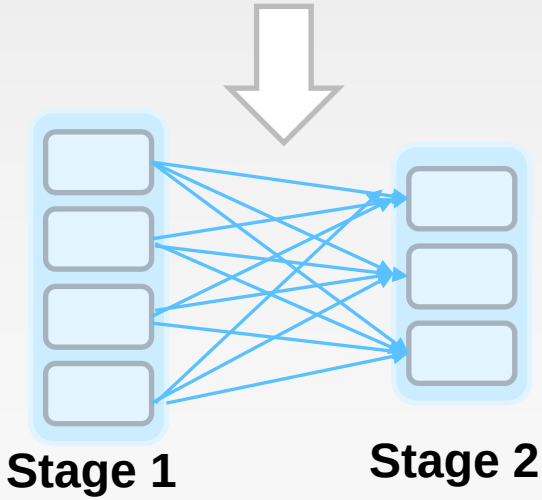


- The scheduler examines the RDD's lineage graph to build a DAG of stages.
- Stages are sequences of RDDs, that don't have a Shuffle in between
- The boundaries are the shuffle stages.

Execution Plan

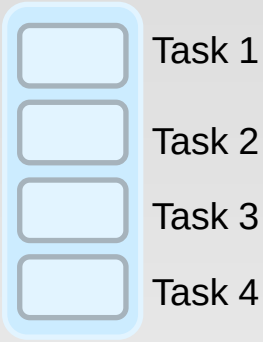


1. Read HDFS split
2. Apply both the maps
3. Start Partial reduce
4. Write shuffle data



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

Stage Execution



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

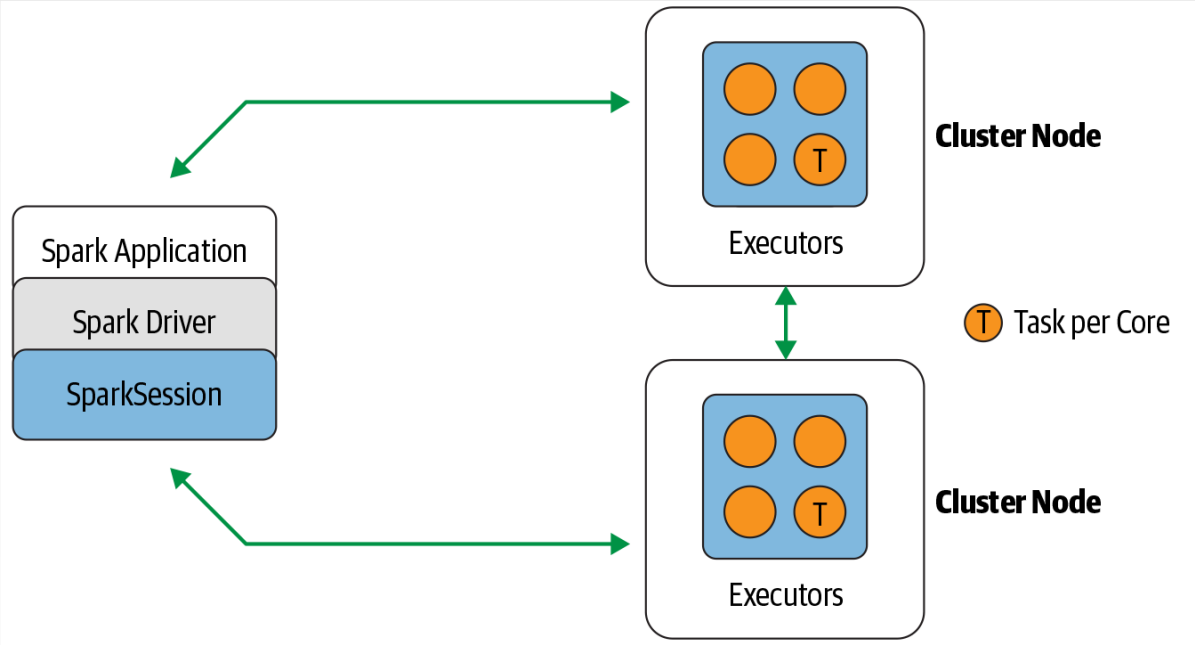
- All this happens internally

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.

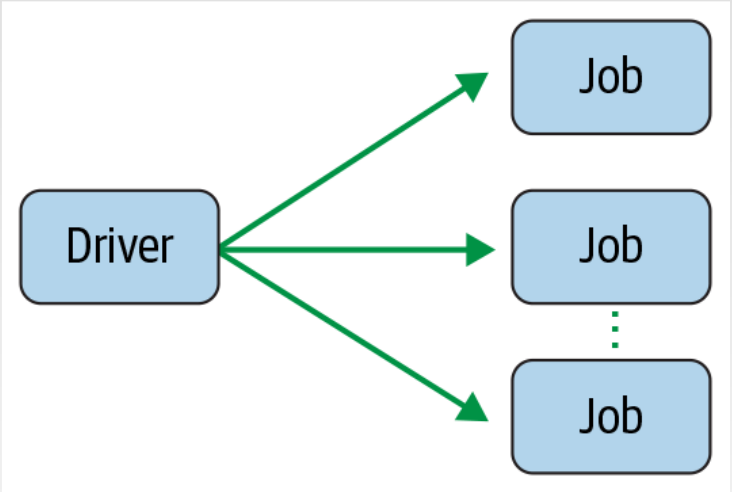
Spark Application and SparkSession

- 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

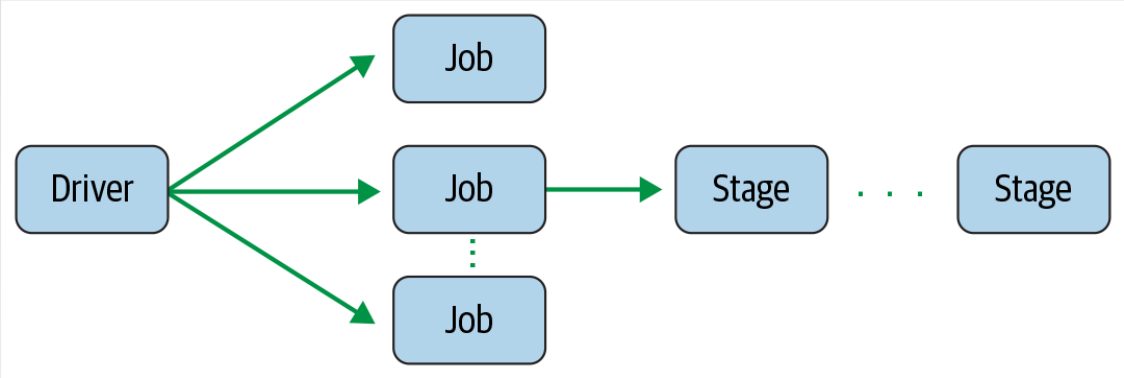
- During interactive sessions with Spark shells, the driver converts your Spark application into one or more Spark jobs



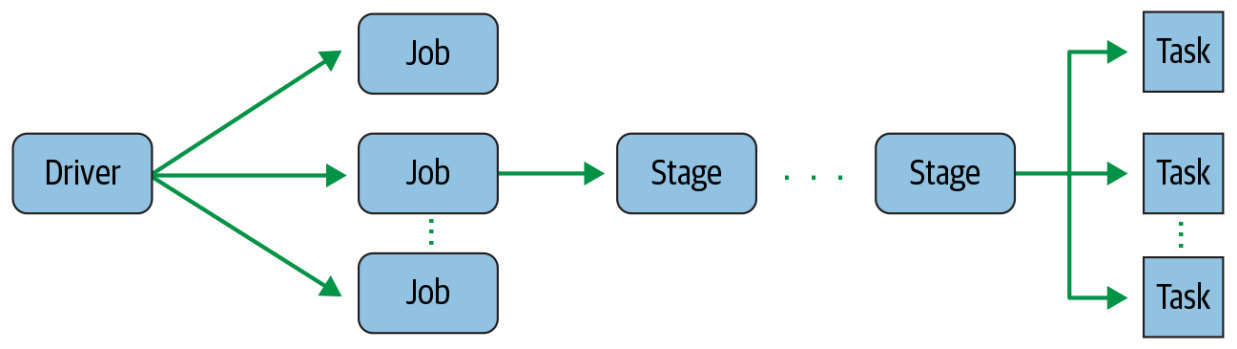
- It then transforms each job into a Spark's execution plan as a DAG, where each node within a DAG could be a single or multiple Spark stages.

Spark Stages and Tasks

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

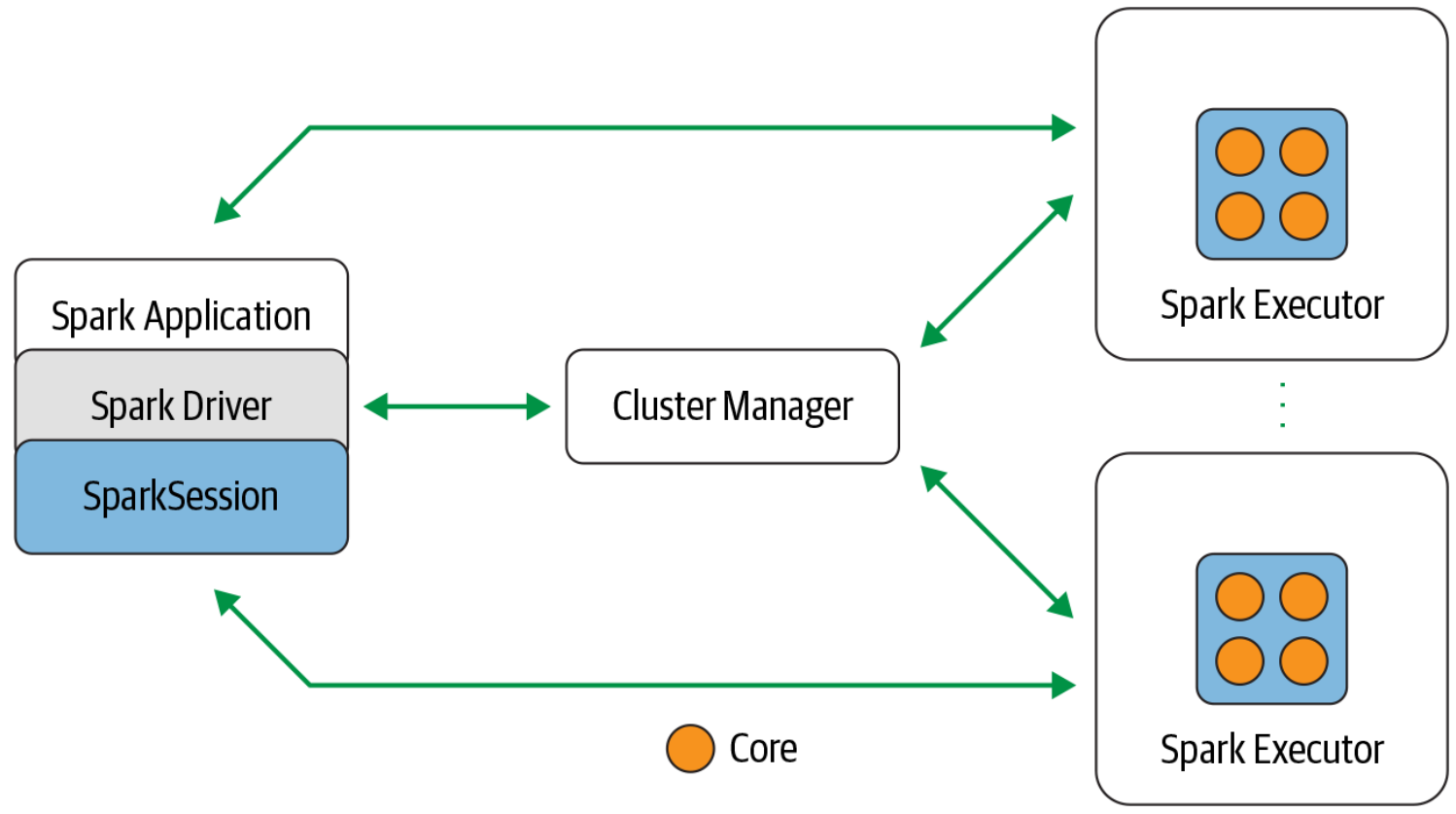


- Each stage is comprised of Spark tasks (a unit of execution), which are then federated across each Spark executor; each task maps to a single core and works on a single partition of data



Spark Architecture

- A Spark application consists of a driver program that is responsible for orchestrating parallel operations on the Spark cluster. The driver accesses the distributed components in the cluster—the Spark executors and cluster manager—through a SparkSession or SparkContext object.



Spark Components

- **Spark Driver**: part of the Spark application responsible for instantiating a SparkSession
  - Communicates with the cluster manager
  - Requests resources (CPU, memory, etc.) from the cluster manager for Spark's executors (JVMs)
  - Transforms all the Spark operations into DAG computations, schedules them, and distributes their execution as tasks across the Spark executors
  - Once the resources are allocated, it communicates directly with the executors.

Spark Components

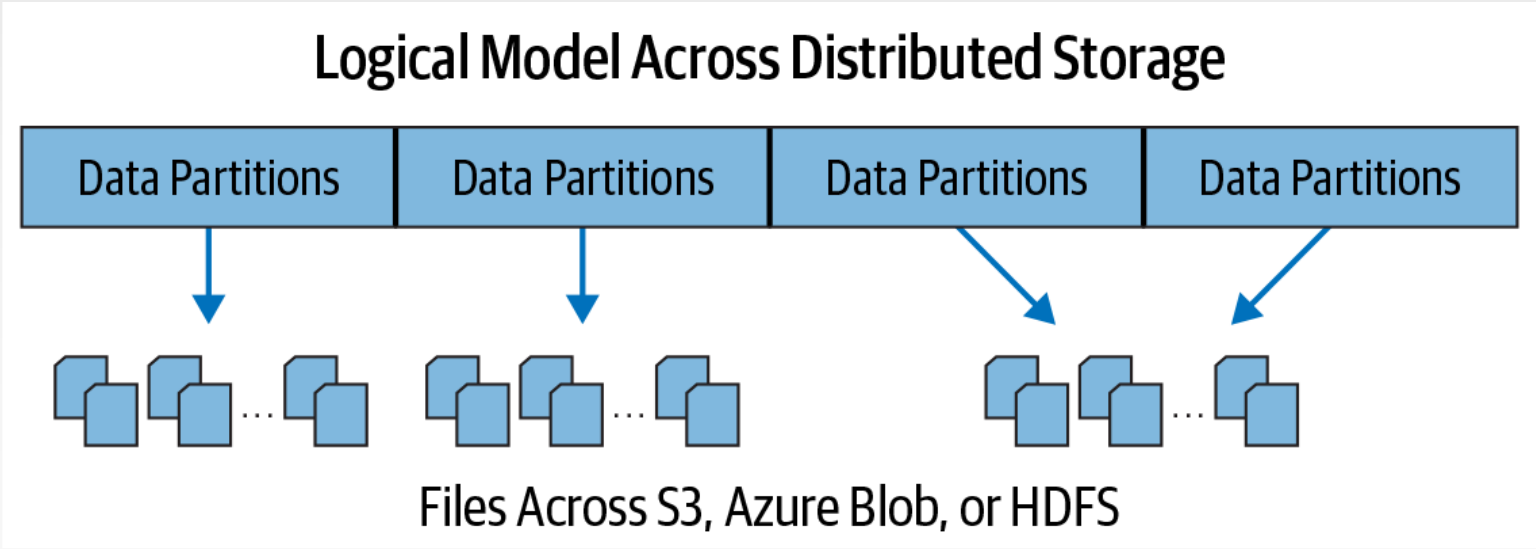
- Since Spark 2.x, the **SparkSession** became a unified conduit to all Spark operations and data (it subsumes previous entry points to Spark like the SparkContext)
- SparkSession provides a single unified entry point to all of Spark's functionality
  - Create JVM runtime parameters
  - Define DataFrames and Datasets
  - Read from Data Sources
  - Access catalog metadata
  - Issue Spark SQL queries

Spark Components

- Cluster manager
  - Responsible for managing and allocating resources for the cluster of nodes on which your Spark application runs.
  - Support four cluster managers: the built-in standalone cluster manager, Apache Hadoop YARN, Apache Mesos, and Kubernetes.
- Spark executor
  - Runs on each worker node in the cluster.
  - Communicate with the driver program and is responsible for executing tasks on the workers.
  - In most deployments modes, only a single executor runs per node.

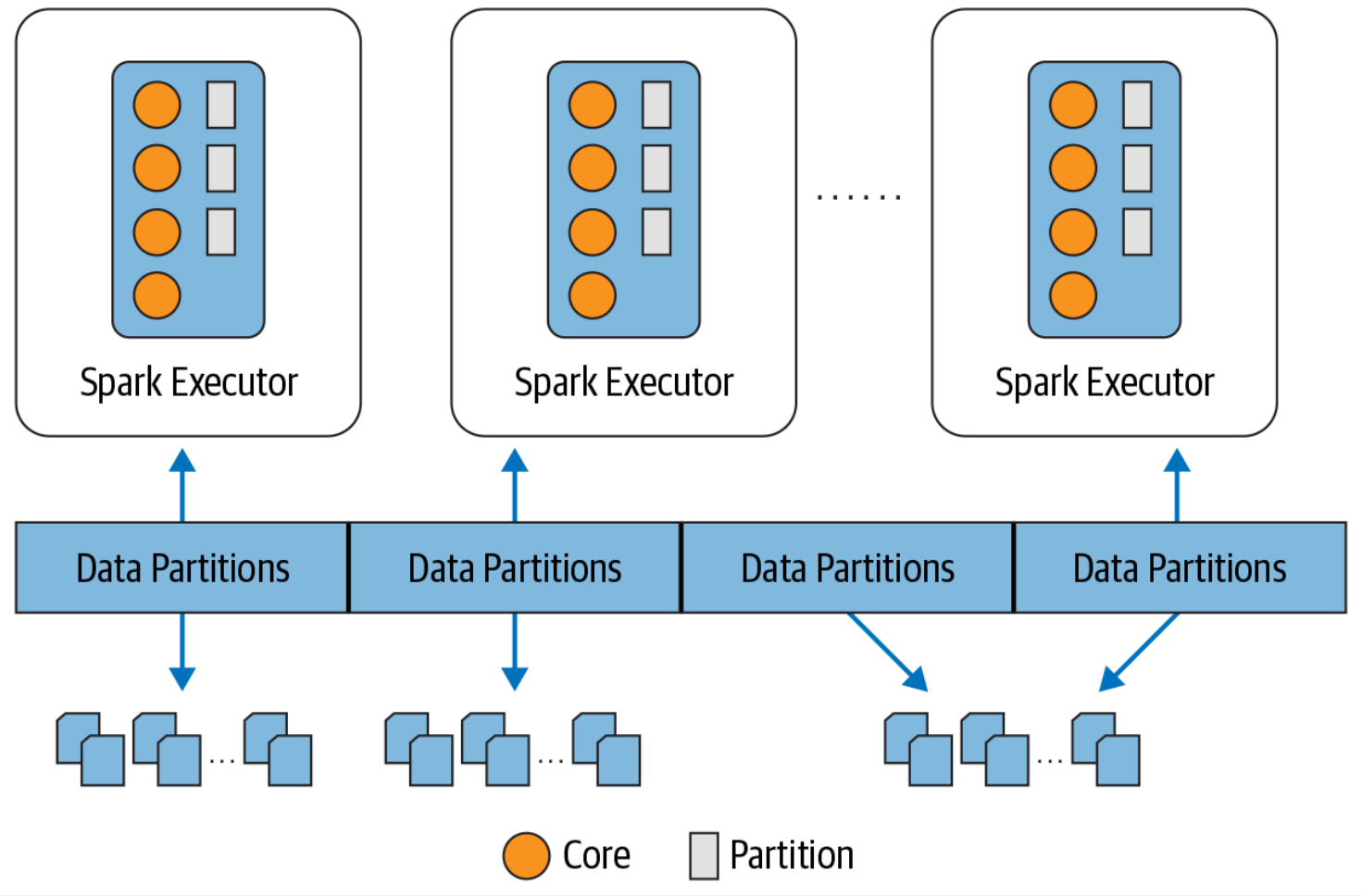
Distributed Data and Partitions

- Actual physical data is distributed across storage as partitions residing in either HDFS or other cloud storage.
- The data is distributed as partitions across the physical cluster
- Spark treats each partition as a high-level logical data abstraction in memory.
- Each Spark executor is preferably allocated a task that requires it to read the partition closest to it in the network, observing data locality.



Distributed Data and Partitions

- Each executor's core is assigned its own data partition to work on



Word Count in Spark (As a Whole View)

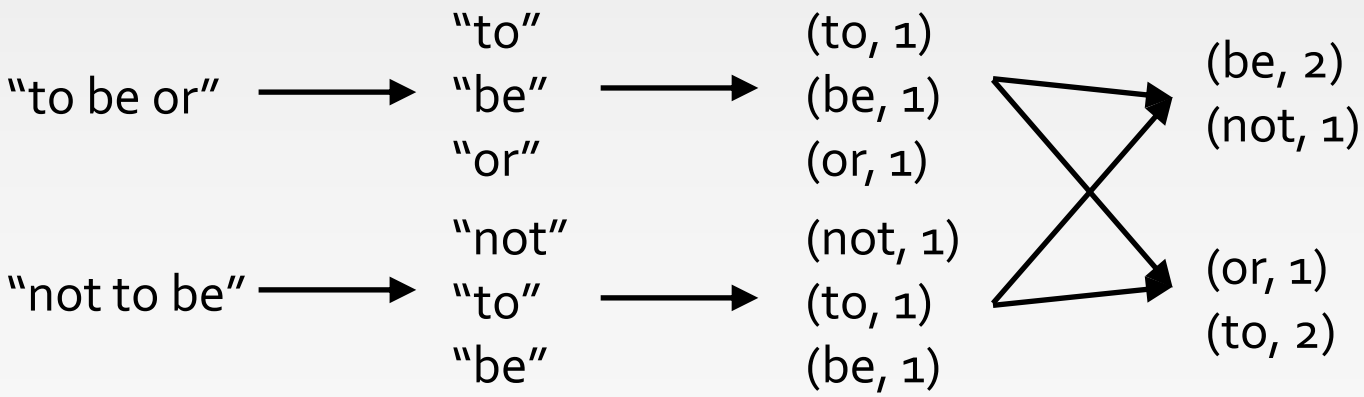
- Word Count using Scala in Spark

```
textfile = sc.textFile("hdfs://...", 4)

words = textfile.flatMap(lambda line: line.split(" "))
pairs = words.map(lambda word: (word, 1))
count = pairs.reduceByKey(lambda a, b: a + b)
count.collect()
```

Transformation

Action

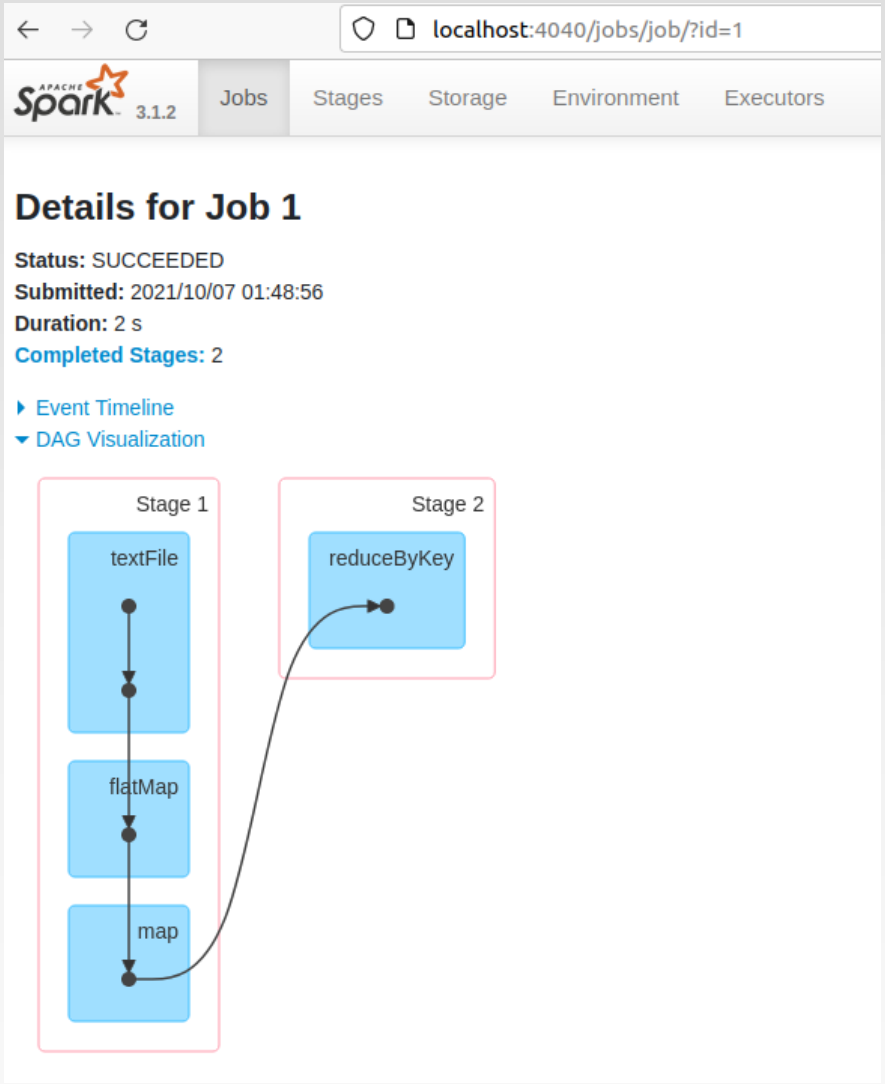


The Spark UI

- 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.

Spark Web Console

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



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)
```

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" % “3.3.0"
```

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

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

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"

WordCount (RDD, Python)

- Standalone code

```
from pyspark import SparkContext, SparkConf

conf = SparkConf()
conf.setMaster("local").setAppName("wordcount")
sc = SparkContext(conf=conf)

#Or you can do the below directly
#sc = SparkContext('local', wordcount')

text = sc.textFile("text.txt")
count = text.flatMap(lambda line: line.split()).map(lambda word: (word,
1)).reduceByKey(lambda a, b : a + b)
count.saveAsTextFile("results")
sc.stop()
```

  - Refer to the document of SparkConf and SparkContext
- Use spark-submit to run the code in a cluster:
  - spark-submit wordcount.py

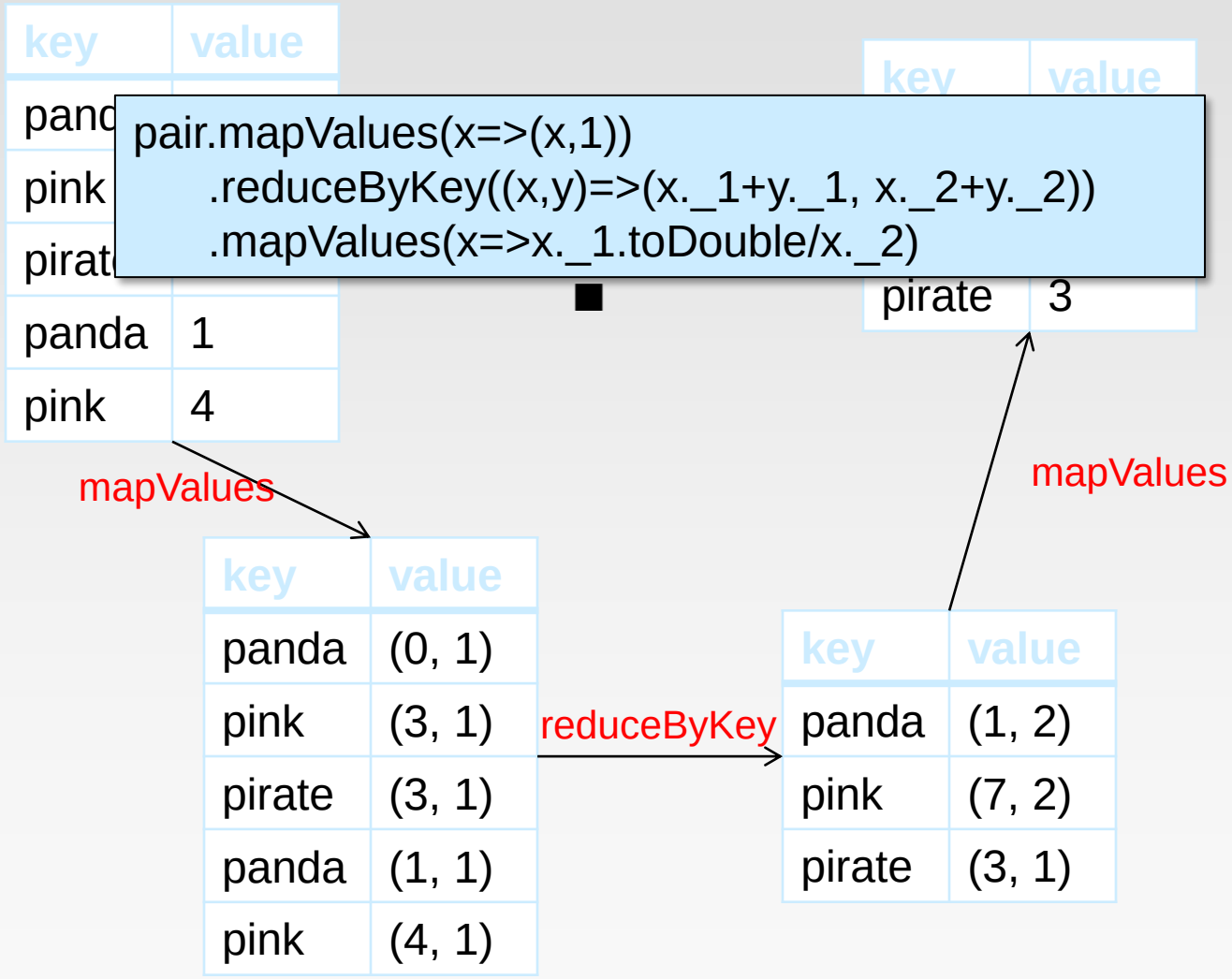
In-Memory Can Make a Big Difference

- 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



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.

```
textFile = sc.textFile(inputFile)
words = textFile.flatMap(lambda line: line.split(" ")).map(lambda x: x.lower())

counts = words.filter(lambda x: len(x) >=1 and x[0]<='z' and x[0]>='a').map(lambda x:
(x[0], (len(x), 1)))

avgLen = counts.reduceByKey(lambda a, b: (a[0]+b[0], a[1]+b[1])).map(lambda x:
(x[0], x[1][0]/x[1][1]))

avgLen.foreach(lambda x: print(x[0], x[1]))
```

```
val textFile = sc.textFile(inputFile)
val words = textFile.flatMap(_.split(" ")).map(_.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)).map(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 edition

End of Chapter 4.2