CSE6331: Cloud Computing

Leonidas Fegaras

University of Texas at Arlington
©2019 by Leonidas Fegaras

Apache Spark

Based on:
Shannon Quinn slides at `http://cobweb.cs.uga.edu/~squinn/mmd_s15/lectures/lecture13_mar3.pdf`
Heather Miller slides at `https://www.epfl.ch/labs/lamp/teaching/page-118166-en-html/`

Alternatives to Map-Reduce

- MapReduce is not the only player in Big Data analytics any more
  - designed for batch processing
  - not well-suited for some big data workloads:
    - iterative algorithms, real-time analytics, continuous queries, ...
- Alternatives:
  - In-memory data parallel processing systems:

    Spark, Flink, Hama, ...
  - Vertex-centric graph processing systems:

    Pregel, Giraph, GraphLab, ...
  - Distributed stream processing engines (DSPEs):

    Storm, Spark Streaming, Flink Streaming, S4, Samza, ...
  - Query systems:

    Hive, Pig, Impala, Spark SQL, MRQL, DIQL, ...

Apache Spark http://spark.apache.org

- May succeed Map-Reduce as a general-purpose computation paradigm
- Fixes the weaknesses of MapReduce
- Does not use MapReduce as an execution engine
- Is compatible with the Hadoop ecosystem: it can run on YARN and works with Hadoop file formats
- Uses in-memory caching: keeps working datasets in memory between operations
- Outperforms equivalent MapReduce workflows in most cases
- Uses a general compute engine based on DAGs that can process arbitrary pipelines of operators
- Provides a rich API for performing many common data processing tasks, such as joins
- Supports iterative and stream data processing
- Supports many libraries, such as machine learning (MLlib), graph processing (GraphX), stream processing (Spark Streaming), and SQL (Spark SQL)

Need to learn Scala first!

To make a better use of Spark, you need to learn Scala

- Spark is written in Scala
- The Spark Scala API is easier to use than the Java Scala API
- The Spark Python API (pyspark) is not for high-performance workloads
- Spark mimics Scala's functional programming style and API
- The RDD API is similar to the Scala collection API
- Scala is an amazing language, which will hopefully replace Java

The Scala programming language

- Scala is object-oriented:
  - every value is an object
  - types and behavior of objects are described by classes and traits
- Scala is functional:
  - every function is a value (so, every function is an object)
  - provides a lightweight syntax for defining anonymous functions
  - supports higher-order functions
  - allows functions to be nested
  - supports currying
- Scala is statically typed:
  - ... but you are not required to annotate the program with redundant type information
  - you don't have to specify a type in most cases
- Scala runs on the the Java Virtual Machine (JVM)
  - it is compiled into Java Byte Code, which is executed by JVM
  - Scala and Java have a common run-time platform
  - you can easily move from Java to Scala
  - you can import Java methods in Scala, and vice versa

First example

```
object HelloWorld {
  def main(args: Array[String]) {
    println("Hello, world!")
  }
}
```

- *singleton object:* a class with a single instance
- the main method is not declared as static: no static members in Scala
- Compiling the example:

  ```
  scalac HelloWorld.scala
  ```

- Running the example:

  ```
  scala -classpath . HelloWorld
  ```

- You may execute scala expressions interactively using the scala interpreter:

  ```
  scala
  ```

Everything is an object

- **val** for immutable objects (values), **var** for mutable objects (variables)

```
val pi = 3.14
var x = 1
x = x+1
```

- Numbers are objects:

```
1+2
1.+(2)
(1).+(2)
```

- Tuples:

```
val x = (1,"a",(3.6,3))
x._1                                  -> 1
x._3._2                               -> 3
x match { case (_,_,(a,_)) => a }     -> 3.6
```

Everything is an object

- Immutable records:

```
case class R ( x: Int , y: String )
val rec = R(1,"a")
rec.x                                   -> 1
rec match { case R(_,s) => s }          -> "a"
```

- Mutable records:

```
case class R ( var x: Int , var y: String )
val rec = R(1,"a")
rec.x = 2
```

- Functions are objects:

```
val inc = (x:Int) => x+1
def apply(f: Int => Int, v: Int) = f(v)
apply(inc,3)                            -> 4
apply( x => x+1, 3 )                    -> 4
```

Classes

- The class attributes define a default constructor

```
class Complex(real: Double, imaginary: Double) {
  def re() = real
  def im() = imaginary
}
```

- An attribute is **var** by default, but you may declare it using **val**
- Creating an instance of the class Complex: new Complex(1.5, 2.3)
- You may also define secondary constructors

```
def this( real : Double) = this( real ,0.0)
```

- Methods without arguments:

```
def re = real
```

- Inheritance and overriding:

```
class Complex( real: Double, imaginary: Double ) extends Serializable {
  def re = real
  def im = imaginary
  override def toString() = "" + re + (if (im < 0) "" else "+") + im + "i"
}
```

Singleton Objects and Class Companions

- Methods and values that are not associated with a class instance belong in singleton objects

```
object MyLib {
  def sum(l: List[Int]): Int = l.sum
}
MyLib.sum(List(1,2))
```

- Scala does not have static methods nor static variables
- A class companion is a singleton object associated with a class of the same name
- The equivalent of a static method in Scala is a method in the class companion
- Class and companion object must be defined in the same file

```
case class Rectangle (length: Int = 0, width: Int = 0)

object Rectangle {
  def area ( r: Rectangle ) = r.length * r.width
}
val rect = Rectangle(10,20)
Rectangle.area(rect)
```

Case Classes and Pattern Matching

- A case class: is immutable (all attributes are **val**s), has one constructor, uses structural equality for ==, better toString

```
abstract class Tree
case class Sum(l: Tree, r: Tree) extends Tree
case class Var(n: String) extends Tree
case class Const(v: Int) extends Tree
```

- Constructors: can write Const(5) instead of **new** Const(5)

```
Sum(Var(x),Const(1))
```

- Pattern matching:

```
def eval ( t: Tree, env: Map[String,Int] ): Int =
  t match {
     case Sum(l, r) => eval(l, env) + eval(r, env)
     case Var(n) => env(n)
     case Const(v) => v
}
eval( Sum(Sum(Var("x"),Var("x")),Sum(Const(7),Var("y"))),
      Map( "x" => 5, "y" -> 7 ) )
```

Traits

- Similar to Java interfaces:

```
trait Ord {
    def < (that: Any): Boolean
    def <=(that: Any): Boolean = (this < that) || (this == that)
    def > (that: Any): Boolean = !(this <= that)
    def >=(that: Any): Boolean = !(this < that)
}
class Date(y: Int, m: Int, d: Int) extends Ord {
    def year = y
    def month = m
    def day = d

    override def < ( that: Any ): Boolean =
        that.isInstanceOf[Date] && {
            val o = that.asInstanceOf[Date]
            if (year != o.year) year < o.year
                else if (month != o.month) month < o.month
                        else day < o.day
```

Genericity

- Similar to Java generic classes (such as, Vector[Integer]):

```
class Stack[T] {
    var elems: List[T] = Nil
    def push(x: T) { elems = x :: elems }
    def top: T = elems.head
    def pop() { elems = elems.tail }
}

val stack = new Stack[Int]

stack.push(1)
stack.pop()                -> 1
```

Arrays

```
val a = Array(1,2,3,4,5,6)
a(0)                    -> 1
a.length                -> 6

a.map(_+1)              -> Array(2,3,4,5,6,7)
a.reduce(_+_)           -> 21

for (e <- a)
    println(e)

for (i <- 0 until a.length)
    a(i) = a(i)+1
```

Collections

```
val ls = List(1,2,3,4)
ls(0)                  -> 1
ls.head                -> 1
ls.last                -> 4
ls ++ List(5,6)        -> List(1,2,3,4,5,6)

ls.foreach( println )

for (elem <- ls)
    println(elem)

ls.map(_+1)            -> List(2,3,4,5)
ls. filter (_%2!=0)    -> List(1,3)

val m = Map( 1 -> "one", 3 -> "three", 2 -> "two")
m(2)                   -> "two"
m+(3 -> "3")           -> Map( 1 -> "one", 3 -> "3", 2 -> "two")
```

Anonymous Functions

- _+1 is the same as x =>x+1
- _+_ is the same as (x,y) =>x+y
- x =>x **match** { pattern =>exp }
  is the same as { pattern =>exp }

```
val l = List(1,2,3)
l.reduce((x,y) => x+y)
l.reduce(_+_)

val l = List((1,"a"),(2,"b"),(1,"c"))
l.map( x => x match { case (i,s) => i+1 } )
l.map { case (i,s) => i+1 }
l.map(_._1+1)
```

Sequence Comprehensions

- for–yield returns a sequence
- without a yield, it returns ()

```
for ( i <- 0 until n;
      j <- i until n if i + j == v )
    yield (i, j)

for ( i <- 0 until 20;
      j <- i until 20 if i + j == 32)
    println ("(" + i + ", " + j + ")")
```

Using Spark

- You can experiment with Spark using the Scala interpreter

  `spark-shell --master local`
- The variable sc is defined to be the new SparkContext

```
scala> val x = sc.textFile("simple.txt")
x: org.apache.spark.rdd.RDD[String] = MapPartitionsRDD[1] at textFile at <console>:27

scala> x.collect
res0: Array[String] = Array(1,2.3, 2,3.4, 1,4.5, 3,6.6, 2,3.0)

scala> val y = x.map( line => { val a = line.split(","); ( a(0).toInt, a(1).toDouble ) } )
y: org.apache.spark.rdd.RDD[(Int, Double)] = MapPartitionsRDD[2] at map

scala> y.collect
res1: Array[(Int, Double)] = Array((1,2.3), (2,3.4), (1,4.5), (3,6.6), (2,3.0))

scala> val z = y.reduceByKey(_+_)
z: org.apache.spark.rdd.RDD[(Int, Double)] = ShuffledRDD[3] at reduceByKey

scala> z.collect
res2: Array[(Int, Double)] = Array((1,6.8), (3,6.6), (2,6.4))
```

The Spark Programming Model

- The central Spark data abstraction is the **Resilient Distributed Dataset (RDD)**:
  - An **immutable** collection of values partitioned across multiple machines in a cluster
  - An RDD is **resilient**: a lost RDD partition can be reconstructed from its input RDD partitions (from the RDDs that were used to compute this RDD)
- Typical Spark **job** workflow:
  - **Creation:** Load input datasets into RDDs
  - **Transformation:** Transform and combine RDDs using a series of transformations, such as, map, filter, group-by, and join
  - **Action:** Bring some of the RDD data back to the application or write these data to persistent storage
- A Spark **application** may consist of multiple Spark jobs
- A Spark job is a DAG of **stages**
- Each stage is equivalent to a Map-Reduce job

Word Count

```
// Create an RDD
val rdd = spark.textFile("hdfs://...")

rdd.flatMap(line => line.split(" "))   // separate lines into words
   .map(word => (word, 1))             // include something to count
   .reduceByKey(_ + _)                 // sum up the 1s in the pairs
```

A Full Example

```
package edu.uta.cse6331

import org.apache.spark.SparkContext
import org.apache.spark.SparkConf

object SimpleSpark {
  def main ( args: Array[String] ) {
    val conf = new SparkConf().setAppName("simple")
    val sc = new SparkContext(conf)
    val x = sc.textFile ("simple.txt")
              .map( line => { val a = line.split (",")
                              ( a(0).toInt , a(1).toDouble ) } )
              .reduceByKey(_+_)
    x.collect ().foreach( println )
  }
}
```

The Simple Example in Java

```
import org.apache.spark.SparkConf;
import org.apache.spark.api.java.*;
import org.apache.spark.api.java.function.*;
import scala.Tuple2;

public class SimpleSpark {
    public static void main ( String[ ] args ) throws Exception {
        SparkConf conf = new SparkConf().setAppName("Join");
        JavaSparkContext sc = new JavaSparkContext(conf);
        JavaPairRDD<Integer,Double> x = sc.textFile("simple.txt")
            .mapToPair(new PairFunction<String,Integer,Double>() {
                    public Tuple2<Integer,Double> call ( String line ) {
                        Scanner s = new Scanner(line).useDelimiter(",");
                        return new Tuple2<Integer,Double>(s.nextInt(),s.nextDouble());
                    } })
            .reduceByKey(new Function2<Double,Double,Double>() {
                    public Double call ( Double x, Double y ) {
                        return new Double(x+y);
                    } });
        for ( Tuple2<Integer,Double> v: x.collect() )
            System.out.println(v);
        sc.stop(); sc.close();
    }
}
```

The Simple Example in Python (pyspark)

```
from pyspark import SparkContext
sc = SparkContext("local", "my job")

def parse( line ):
    a = line.split (",")
    return (int(a[0]), float(a[1]))

x = sc.textFile ("simple.txt")    \
      .map(parse)  \
      .reduceByKey(lambda x,y: x+y)

print(x.collect ())
```

RDD Operations

- **Creations:** these are SparkContext methods that create an RDD
  - Parallelize a collection:

```
sc.parallelize(1 to 10000)
```

  - Load a dataset from external storage:

```
sc.textFile(inputPath)
sc.sequenceFile[Int, String](inputPath)
```

- **Transformations:** these are methods from RDD[T] to RDD[T']
  - They generate a new RDD from one or more existing RDDs
  - Lazy evaluation: transformations are not executed immediately; their execution is triggered by an action
  - Class RDD[(K,V)] is coerced to PairRDDFunctions[K,V] implicitly
  - Class PairRDDFunctions[K,V] contains all transformation methods on key-value datasets (eg, groupByKey)
- **Actions:** these are methods from RDD[T] to a type other than RDD
  - They trigger a Spark job (they have an immediate effect)
  - They used to bring some of the RDD data back to the application or write these data to persistent storage

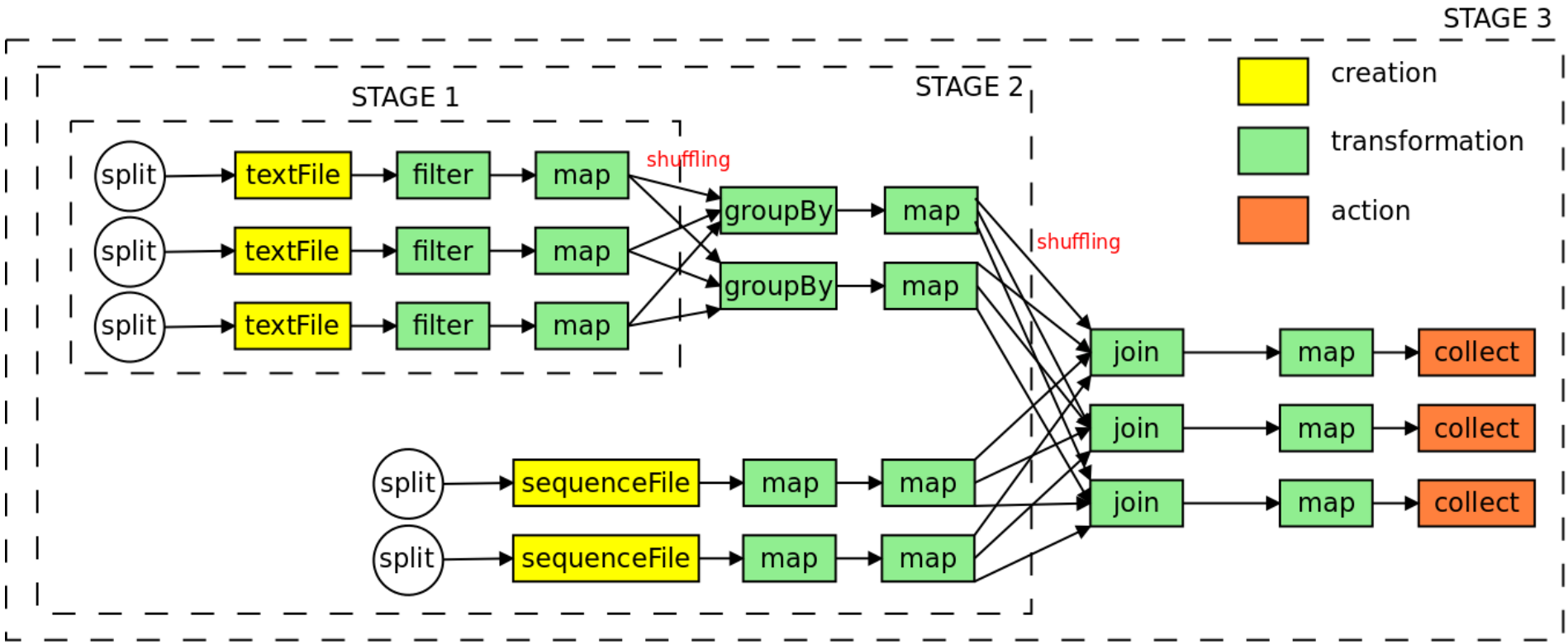
RDD Internals

- A Spark **application** may consist of multiple Spark jobs
- A Spark **job** consists of one or more RDD creations, followed by a series of tranformations, and ending with one action
- Transformations are not executed until an action is called
- Two types of transformations:
  - Those that do not cause data shuffling, such as, map and filter (transformations with **narrow dependencies**)
  - Those that cause data shuffling, such as, groupByKey and join (transformations with **wide dependencies**)
- Transformations with wide dependencies take an optional parameter numPartitions, which specifies the number of workers
  - default numPartitions is equal to the numPartitions of the previous stage
- ... or they take an optional parameter partitioner (of type Partitioner) that maps keys to partition numbers

RDD Internals (cont.)

- A Spark **stage** is a sub-series of transformations in a Spark job that do not cause data shuffling
- Data shuffling happens between stages
- The result of a stage is a new RDD that is cached in memory
- The RDD transformations inside a stage are evaluated lazily: they do not create any new RDD
- A stage corresponds to a Map-Reduce job
- A Spark job is a DAG of stages

A Spark Job with 3 Stages



Transformations with wide dependencies: groupBy and join

Specifying the Master Node

- The --**master** option specifies the master URL for a distributed cluster, or local to run locally
- Use --**master yarn-client** to run Spark on a Yarn cluster
- Other --master values:

master	*description*
`local`	run Spark locally with one worker thread (no parallelism)
`local[K]`	run Spark locally with K worker threads (ideally set to # cores)
`spark://HOST:PORT`	connect to a Spark standalone cluster; PORT depends on config (7077 by default)
`mesos://HOST:PORT`	connect to a Mesos cluster; PORT depends on config (5050 by default)

Transformations

transformation	*description*
map(*func***)**	return a new distributed dataset formed by passing each element of the source through a function *func*
filter(*func***)**	return a new dataset formed by selecting those elements of the source on which *func* returns true
flatMap(*func***)**	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)
sample(*withReplacement, fraction, seed***)**	sample a fraction *fraction* of the data, with or without replacement, using a given random number generator *seed*
union(*otherDataset***)**	return a new dataset that contains the union of the elements in the source dataset and the argument
distinct([*numTasks***]))**	return a new dataset that contains the distinct elements of the source dataset

Transformations

transformation	*description*
groupByKey([*numTasks*]**)**	when called on a dataset of `(K, V)` pairs, returns a dataset of `(K, Seq[V])` pairs
reduceByKey(*func*, [*numTasks*]**)**	when called on a dataset of `(K, V)` pairs, returns a dataset of `(K, V)` pairs where the values for each key are aggregated using the given reduce function
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
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
cogroup(*otherDataset*, [*numTasks*]**)**	when called on datasets of type `(K, V)` and `(K, W)`, returns a dataset of `(K, Seq[V], Seq[W])` tuples – also called `groupWith`
cartesian(*otherDataset***)**	when called on datasets of types `T` and `U`, returns a dataset of `(T, U)` pairs (all pairs of elements)

Actually, cogroup returns a dataset of (K, (Seq[V], Seq[W]))

Actions

action	*description*
reduce(*func***)**	aggregate the elements of the dataset using a function *func* (which takes two arguments and returns one), and should also be commutative and associative so that it can be computed correctly in parallel
collect()	return all the elements of the dataset as an array at the driver program – usually useful after a filter or other operation that returns a sufficiently small subset of the data
count()	return the number of elements in the dataset
first()	return the first element of the dataset – similar to *take(1)*
take(*n***)**	return an array with the first *n* elements of the dataset – currently not executed in parallel, instead the driver program computes all the elements
takeSample(*withReplacement, fraction, seed***)**	return an array with a random sample of *num* elements of the dataset, with or without replacement, using the given random number generator seed

Actions

action	*description*
`saveAsTextFile(`*`path`***`)`**	write the 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
`saveAsSequenceFile(`*`path`***`)`**	write the elements of the dataset as a Hadoop `SequenceFile` in a given path in the local filesystem, HDFS or any other Hadoop-supported file system. Only available on RDDs of key-value pairs that either implement Hadoop's `Writable` interface or are implicitly convertible to `Writable` (Spark includes conversions for basic types like `Int`, `Double`, `String`, etc).
`countByKey()`	only available on RDDs of type `(K, V)`. Returns a `Map` of `(K, Int)` pairs with the count of each key
`foreach(`*`func`***`)`**	run a function *func* on each element of the dataset – usually done for side effects such as updating an accumulator variable or interacting with external storage systems

Choosing the Right RDD Method

- map vs flatMap

```
val x = sc.textFile("simple.txt")
x.map(line => line.split(",")).collect
-> Array(Array("1","2.3"), Array("2","3.4"), Array("1","4.5"),
         Array("3","6.6"), Array("2","3.0"))
x.flatMap(line => line.split(",")).collect
  -> Array("1","2.3","2","3.4","1","4.5","3","6.6","2","3.0")
```

- join vs cogroup
  - join can be more efficient that cogroup
  - cogroup is more powerful than join (it can capture outer joins too)
  - x.join(y) is the same as:

```
x.cogroup(y)
 .map{ case (k,(xs,ys)) => xs.flatMap(x => ys.map(y => (k,(x,y)))) }
```

Choosing the Right RDD Method

- Instead of groupBy/groupByKey followed by aggregation over each key, use aggregateByKey or reduceByKey for better performance

Example:

```
case class Person ( name: String, dno: Int, Salary: Int )

val persons = sc.textFile("persons.txt")
                .map( _.split(",") )
                .map( p => Person(p(0),p(1).toInt,p(2).toInt) )
```

use this:

```
persons.map{ case Person(_,n,s) => (n,s) }.reduceByKey(_+_)
```

instead of this:

```
persons.map{ case Person(_,n,s) => (n,s) }.groupByKey()
       .map{ case (k,ss) => (k,ss.reduce(_+_)) }
```

Another Example

Instead of this:

```
rdd.map(p => (p.id, p.price))
    .groupByKey()
    .map{ case (id, prices ) => (id, ( prices . size ,  prices .sum)) }
    . collect ()
```

use this:

```
rdd.map(p => (p.id, (1, p. price )))
    .reduceByKey{ case ((c1,s1),(c2,s2)) => (c1+c2, s1+s2) }
    . collect ()
```

Using Range Partitioning

Range partitioning over 8 partitions.
Need to provide a key-value RDD with ordered keys, which is sampled to create a suitable set of sorted ranges.

```
val pairs = rdd.map(p => (p.id, p.price))
val partitioner = new RangePartitioner(8, pairs)

pairs.partitionBy( partitioner )
     .map(p => (p.id, (1, p.price )))
     .reduceByKey{ case ((c1,s1),(c2,s2)) => (c1+c2, s1+s2) }
     .collect ()
```

No shuffling during reduceByKey!
... but there was shuffling during partitionBy
The same happens if you use a HashPartitioner

Avoiding Network Shuffle By Pre-Partitioning

Examples:

1. reduceByKey running on a pre-partitioned RDD will cause the values to be computed locally, requiring only the final reduced values to be sent from the workers to the driver.
2. join called on two RDDs that are pre-partitioned with the same partitioner and cached on the same machine will cause the join to be computed locally, with no shuffling across the network.

Since pre-partitioning does the shuffling ahead of time, these techniques make sence when an RDD is used in joins or reduceByKey more that once

RDD Persistence

- You can mark an RDD to be persistent in storage (memory, disk, etc) using the persist() or cache() methods
- Set by passing a StorageLevel object to persist()
- The default storage level is to store deserialized objects in memory

```
rdd.persist(StorageLevel.MEMORY_ONLY)
```

- same as: rdd.cache()

Storage Level	Meaning
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 on the fly each time they're needed. This is the default level.
MEMORY_AND_DISK	Store RDD as deserialized Java objects in the JVM. 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.
MEMORY_ONLY_SER (Java and Scala)	Store RDD as *serialized* Java objects (one byte array per partition). This is generally more space-efficient than deserialized objects, especially when using a fast serializer, but more CPU-intensive to read.
MEMORY_AND_DISK_SER (Java and Scala)	Similar to MEMORY_ONLY_SER, but spill partitions that don't fit in memory to disk instead of recomputing them on the fly each time they're needed.
DISK_ONLY	Store the RDD partitions only on disk.

Data Serialization

- Spark's default serialization is Java serialization (class Serializable)
- You may use a custom serialization, such as Kryo, which is faster and more compact

```
conf.set("spark. serializer ","org.apache.spark. serializer . KryoSerializer ")
conf. registerKryoClasses (Array(classOf[MyClass1], classOf[MyClass2]))
```

- Non-local variables referenced by RDD methods are always serialized using the Serializable class
- OK for small array s, not OK for very large s:

```
val s = Array(1, 2, 3)
rdd. filter (x => s.member(x._1))
```

- Wrong to refer to an RDD within an RDD method closure:

```
val rdd1 = sc. textFile ( file )
rdd2. filter (x => x._1 < rdd1.count())     // wrong! - run-time error
```

- Correct:

```
val c = rdd1.count()
rdd2. filter (x => x._1 < c)
```

Data Serialization

If you refer to a class variable inside a closure, the entire object is serialized

```
class MyApp ( repos: RDD[Repository], team: Map[String,String] ) {
  def projects (): Array[Repository]
    = repos. filter {
        repo => team.exists( user => repo.contr.contains(user) )
      }. collect () }
```

You get a java.io.NotSerializableException error at run-time because team inside the closure is the same as this.team, so this needs to be serialized but class MyApp is not serializable. Instead do:

```
def projects (): Array[Repository] = {
  val t = team
  repos. filter {
      repo => t.exists( user => repo.contr.contains(user) )
    }. collect () }
```

Same if you call a class method inside a closure

Broadcast Variables

- Used for keeping a read-only variable cached on each machine
- Far more efficient than using non-local variables in closures
- Can be used to give every node a copy of a large input dataset in an efficient manner
- Can be used to implement a map-backed join
- Should not be modified after it is broadcast to ensure that all nodes get the same value

```
val broadcastVar = sc.broadcast(Array(1, 2, 3))
rdd. filter (x => broadcastVar.value.member(x._1))
```

Data Shuffling in Spark

- Done between stages
- Data shuffling is done in the same way as in Map-Reduce:
  - Spark initiates two sets of tasks: map tasks to sort the data, and reduce tasks to merge and aggregate the data
- Data shuffling generates a large number of intermediate files on the local disk of each node
- These files are preserved until the corresponding RDDs are garbage collected

Sharing RDDs

- Sharing RDDs within the same job is OK (a job is a DAG of RDDs)

```
val logData = sc.textFile(logFile)      // shared RDD
val As = logData.filter(line => line.contains("a"))
val Bs = logData.filter(line => line.contains("b"))
val nums = As.union(Bs).count()
```

- Sharing RDDs across two jobs may require some caching, otherwise the shared RDDs will be evaluated twice
  - Not needed if the shared RDD is a complete stage (ie, groupBy, join)
- Use rdd.cache() to cache the rdd in memory

```
val logData = sc.textFile(logFile).cache()
val numAs = logData.filter(line => line.contains("a")).count()
val numBs = logData.filter(line => line.contains("b")).count()
println("Lines with a: %s, Lines with b: %s".format(numAs, numBs))
```

  - There are two jobs here (because there are two count actions) that share the same RDD: logData
  - Without using cache, the logData would have been evaluated twice

A Join in Scala/Spark

```
object JoinSpark {
  def main(args: Array[ String ]) {
    val conf = new SparkConf().setAppName("Join")
    val sc = new SparkContext(conf)
    val emps = sc.textFile(args(0))
                 .map( line => { val a = line.split(",")
                                 (a(0),a(1).toInt,a(2)) } )
    val depts = sc.textFile(args(1))
                  .map( line => { val a = line.split(",")
                                  (a(0),a(1).toInt) } )
    val res = emps.map( e => (e._2,e) )
                  .join(depts.map( d => (d._2,d) ))
                  .map { case (k,(e,d)) => e._1+" "+d._1 }
    res.saveAsTextFile(args(2))
    sc.stop()
  }
}
```

The Same Join in Scala/Spark Using Serialized Classes

```
@SerialVersionUID(123L)
class Employee ( val name: String, val dno: Int, val address: String )
      extends Serializable {}

@SerialVersionUID(123L)
class Department ( val name: String, val dno: Int )
      extends Serializable {}

object JoinSpark {
  def main(args: Array[ String ]) {
    val conf = new SparkConf().setAppName("Join")
    val sc = new SparkContext(conf)
    val emps = sc.textFile(args(0)).map( line => { val a = line.split(",")
                                                   new Employee(a(0),a(1).toInt,a(2)) } )
    val depts = sc.textFile(args(1)).map( line => { val a = line.split(",")
                                                    new Department(a(0),a(1).toInt) } )
    val res = emps.map( e => (e.dno,e) )
                  .join(depts.map( d => (d.dno,d) ))
                  .map { case (k,(e,d)) => e.name+" "+d.name }
    res.saveAsTextFile(args(2))
    sc.stop()
  }
}
```

Join in Scala/Spark Using a Broadcast Variable

Assuming one of the join inputs (depts) is small:

```
...
// broadcast depts
val bv = sc.broadcast(depts.map( d => (d._1,d) ).collectAsMap())
// for each employee, find the corresponding department
val res = emps.flatMap( e => bv.value.get(e._2) match {
                                    case Some(d)
                                      => List(e._1+" "+d._1)
                                    case _ => List()
                                } )
...
```

Join without Shuffling using Pre-Partitioning

Re-partition the inputs with the same partitioner and cache:

```
val hp = new HashPartitioner(x.getNumPartitions)
val xx = x.partitionBy(hp).cache()
val yy = y.partitionBy(hp).cache()
xx.zipPartitions(yy,true)( ijoin(_,_))
```

where ijoin joins two iterators:

```
def ijoin[K,A,B](x: Iterator[(K,A)], y: Iterator[(K,B)]): Iterator[(K,(A,B))] = {
  val h = y.toList.groupBy(_._1)    // create a hash table
  x.flatMap{ case (k,x) => h.get(k) match { case Some(ys) => ys.map(y => (k,(x,y._2)))
                                           case _ => Nil } } }
```

Shuffling is done during partitionBy. Makes sense if one of the inputs is kept partitioned and used many times, such as x=x.join(y) in a loop:

```
x.partitioner match {
  case Some(p)
    => x = x.zipPartitions(y.partitionBy(p),true)( ijoin(_,_)).cache()
  case _ => x = x.join(y).cache()
}
```

Matrix Multiplication

```
val M = sc.textFile(args(0))
          .map( line => { val a = line.split(",")
                          ((a(0).toInt,a(1).toInt),a(2).toDouble) } )
val N = sc.textFile(args(1))
          .map( line => { val a = line.split(",")
                          ((a(0).toInt,a(1).toInt),a(2).toDouble) } )

M.map{ case ((i,j),m) => (j,(i,m)) }
 .join( N.map{ case ((i,j),n) => (i,(j,n)) } )
 .map{ case (k,((i,m),(j,n))) => ((i,j),m*n) }
 .reduceByKey(_+_)
```

PageRank

```
val E = sc.textFile(args(0))
        .map( line => { val a = line.split(",").toList
                        (a(0).toLong,a(1).toLong) } )

val links = E.groupByKey().cache()

var ranks = links.mapValues(v => 1.0/N)

for (i <- 1 to 10) {
    val contribs = links.join(ranks).values.flatMap {
                       case (urls, rank)
                        => val size = urls.size
                           urls.map(url => (url, rank / size))
                   }
    ranks = contribs.reduceByKey(_ + _)
                    .mapValues(0.15/N + 0.85 * _)
}
ranks.sortBy(_._2,false,1).take(30).foreach( println )
```