Day 2: S3 – Spark-Scala-Session

By Keshav Thakur ☺

Spark – Under the Hood | RDD's → Reduction && Distributed Key-Value Pairs

Reduction Operation's in Spark RDD's → Fold, Reduce, Aggregate
reduceByKey, groupByKey, aggregateByKey, combineByKey --- ..... ??? Avoid GroupByKey as Network Latency

groupByKey:

Syntax:

```
sparkContext.textFile("hdfs://")
                    .flatMap(line => line.split(" ") )
                    .map(word => (word,1))
                    .groupByKey()
                    .map((x,y) => (x,sum(y)))
```

groupByKey can cause out of disk problems as data is sent over the network and collected on the reduce workers.

reduceByKey:

Syntax:

```
sparkContext.textFile("hdfs://")
                    .flatMap(line => line.split(" "))
                    .map(word => (word,1))
                    .reduceByKey((x,y)=> (x+y))
```

Data are combined at each partition, only one output for one key at each partition to send over the network. reduceByKey required combining all your values into another value with the exact same type.

aggregateByKey:

same as reduceByKey, which takes an initial value.

3 parameters as input i. initial value ii. Combiner logic iii. sequence op logic

Example:

```
val keysWithValuesList = Array("foo=A", "foo=A", "foo=A", "foo=A", "foo=B", "bar=C", "bar=D
    val data = sc.parallelize(keysWithValuesList)
    //Create key value pairs
    val kv = data.map(_.split("=")).map(v => (v(0), v(1))).cache()
    val initialCount = 0;
    val addToCounts = (n: Int, v: String) => n + 1
    val sumPartitionCounts = (p1: Int, p2: Int) => p1 + p2
    val countByKey = kv.aggregateByKey(initialCount)(addToCounts, sumPartitionCounts)
```

ouput: Aggregate By Key sum Results bar -> 3 foo -> 5

combineByKey:

3 parameters as input

1. Initial value: unlike aggregateByKey, need not pass constant always, we can pass a function that will return a new value.
2. merging function
3. combine function

Example:

```
val result = rdd.combineByKey(
                        (v) => (v,1),
                        ( (acc:(Int,Int),v) => acc._1 +v , acc._2 +1 ) ,
                        ( acc1:(Int,Int),acc2:(Int,Int) => (acc1._1+acc2._1) , (acc1._2+acc
                        ).map( { case (k,v) => (k,v._1/v._2.toDouble) })
        result.collect.foreach(println)
```

reduceByKey,aggregateByKey,combineByKey *preferred over* groupByKey

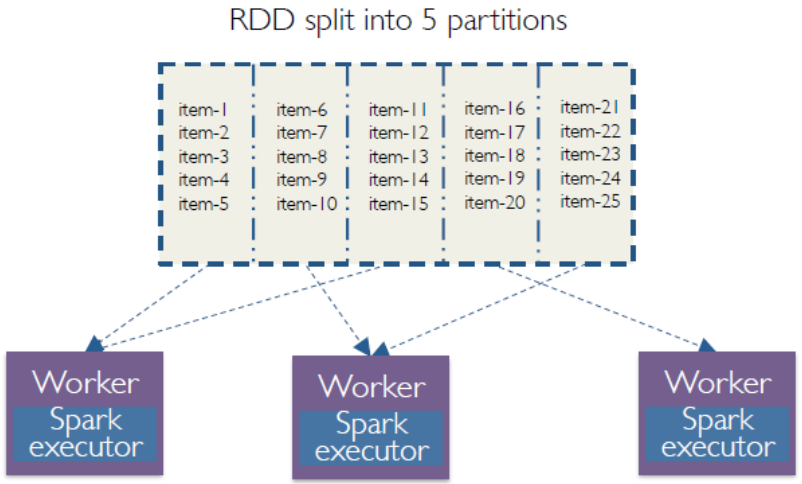
reduceBykey	aggregateByKey	groupByKey
Uses Combiner	Uses Combiner	Do not uses Combiner
Take one parameter as function – for seqOp and combOp	Take 2 parameters as functions – one for seqOp and other for combOp	No parameters as functions. Generally followed by map or flatMap
Implicit Combiner	Explicit Combiner	No combiner
Performance is high for aggregations	Performance is high for aggregations	Relatively slow for aggregations

Joins

Inner Joins
Outer Joins(Left/Right)

https://backtobazics.com/big-data/spark/apache-spark-aggregatebykey-example/

Spark – Under the Hood | Shuffling && Partitioning



shuffle
/ˈʃʌf(ə)l/

verb
gerund or present participle: **shuffling**

1. walk by dragging one's feet along or without lifting them fully from the ground.
   "I stepped into my skis and shuffled to the edge of the steep slope"
   Similar: shamble | drag one's feet | stumble | lumber | stagger | teeter
   - shift one's position while sitting or move one's feet while standing, typically because of boredom, nervousness, or embarrassment.
     "Christine shuffled uneasily in her chair"
     Similar: scrape | drag | scratch | grind | scuffle | scuff
   - put part of one's body into (an item of clothing), typically in a clumsy way.
     "she shuffled her feet into a pair of shoes"
2. rearrange (a pack of cards) by sliding them over each other quickly.
   "he shuffled the cards and cut the deck"
   Similar: mix | mix up | mingle | intermix | shift about | rearrange
   - move (people or things) around so as to occupy different positions or to be in a different order.
     "my father shuffled his money around various building societies"
   - play or arrange (tracks on a music player) in a random order.
     "the control lets you shuffle or skip songs by flicking the phone with your wrist"
   - sort or look through (a number of things) hurriedly.
     "he shuffled through the papers"
3. get out of or avoid a responsibility or obligation.
   "some hospitals can shuffle off their responsibilities by claiming to have no suitable facilities"
   - ARCHAIC
     behave in a shifty or evasive manner.
     "Mr Milles did not frankly own it, but seem'd to shuffle about it"
   - ARCHAIC
     get out of (a difficult situation) in an underhand way.
     "he shuffles out of the consequences by vague charges of undue influence"

We typically have to move data from one node to another when group by action is called.

→ **Doing this is Shuffling**

→ **Problematic if this happens a lot with huge data sets.**

Partitioning --- When I shuffle, I get more or less partitions

Two type of Partitioning: Otherwise how would the driver knows?

Hash Partitioning

Range partitioning

By Default, it's the no of core's available on every executors node.

how many tasks ? The number of tasks should be equal to

Sum of (Stage * #Partitions in the stage)

- load two datasources
- perform some map operation on both of the data sources separately
- join them
- perform some map and filter operations on the result
- save the result

```
val sfi  = sc.textFile("/data/blah/input").map{ x => val xi = x.toInt; (xi,xi*xi) }
val sp = sc.parallelize{ (0 until 1000).map{ x => (x,x * x+1) }}
val spj = sfi.join(sp)
val sm = spj.mapPartitions{ iter => iter.map{ case (k,(v1,v2)) => (k, v1+v2) }}
val sf = sm.filter{ case (k,v) => v % 10 == 0 }
sf.saveAsTextFile("/data/blah/out")
```

Completed Stages (3)

Stage Id	Description		Submitted
2	saveAsTextFile at <console>:35	+details	2016/05/30 08:07:10
1	parallelize at <console>:24	+details	2016/05/30 08:07:09
0	map at <console>:24	+details	2016/05/30 08:07:09

Spark – Under the Hood | Shuffling && Partitioning

Winowing Function

Name	Dept	salary	lag	lead
abc	it	100	0	50
bcd	hr	150	-50	20
efg	it	170	-20	-70
ghf	sales	100	70	0

Pivot

name	it	hr	sales	salary
abc	1	0	0	100
bcd	0	1	0	150
efg	1	0	0	170
ghf	0	0	1	100

count(*)	dept
2	it
1	hr
1	sales

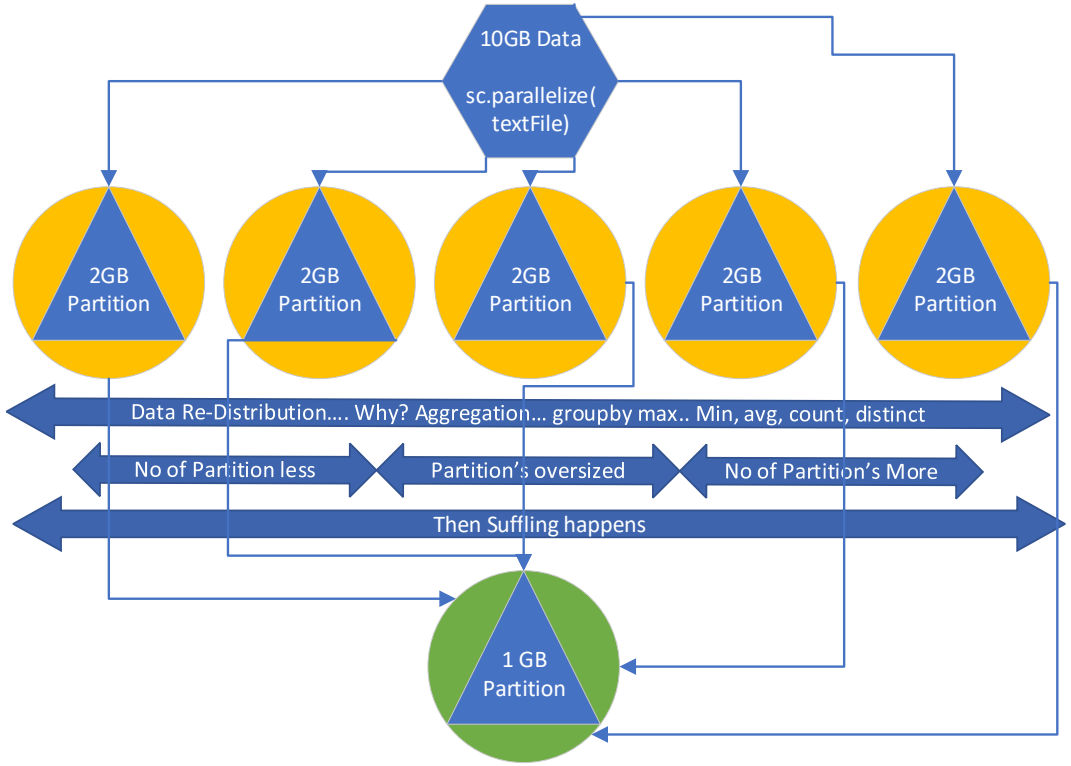
Max(salary)	dept
170	it
150	hr
100	sales

Utilisation
Efficiency
Occupancy
Sales per counter
Counter Usage

- Application Properties
- Runtime Environment
- Shuffle Behavior
- Spark UI
- Compression and Serialization
- Memory Management
- Execution Behavior
- Executor Metrics
- Networking
- Scheduling
- Barrier Execution Mode
- Dynamic Allocation
- Thread Configurations

To Avoid: Dataframe function *repartition* is costly because it will Cause shuffle.

- Query Planning
- Logical Plan
- Physical Plan
- EnsureRequirements
- Bucketing
- Repartition
- Example I: One-side shuffle-free join
- Example II: Aggregation followed by a join
- Example III: Union of two aggregations



Prefer reduced shuffle than total shuffle: Shuffle after reduceByKey

=Scala=

CHEAT SHEET v.0.1

"Every value is an object & every operation is a message send."

PACKAGE

Java style:

```
package com.mycompany.mypkg
```

applies across the entire file scope

Package "scoping" approach: curly brace delimited

```
package com
{
  package mycompany
  {
    package scala
    {
      package demo
      {
        object HelloWorld
        {
          import java.math.BigInteger
          // just to show nested importing
          def main(args : Array[String]) :
          Unit =
          { Console.println("Hello there!")
          }
        }
      }
    }
  }
}
```

IMPORT

```
import p._          // imports all members of p
// (this is analogous to import p.* in Java)

import p.x          // the member x of p
import p.{x => a}   // the member x of p renamed
                    // as a
import p.{x, y}     // the members x and y of p
import p1.p2.z      // the member z of p2,
                    // itself member of p1
import p1._, p2._   // is a shorthand for import
                    // p1._; import p2._
```

implicit imports:

the package `java.lang`
the package `scala`
and the object `scala.Predef`

Import anywhere inside the client Scala file, not just at the top of the file, for scoped relevance, see example in Package section

VARIABLE

```
var var_name: type = init_value;
```

eg. `var i : int = 0;`

default values:

```
private var myvar: T = _ // "_" is a default
value
```

`scala.Unit` is similar to `void` in Java, except Unit can be assigned the () value.

```
unnamed2: Unit = ()
```

default values:

- 0 for numeric types
- false for the Boolean type
- () for the Unit type
- null for all object types

CONSTANT

Prefer val over var.

form: `val var_name: type = init_value;`

```
val i : int = 0;
```

STATIC

No static members, use Singleton, see Object

CLASS

Every class inherits from `scala.Any`

2 subclass categories:

`scala.AnyVal` (maps to java.lang.Object)
`scala.AnyRef`

form: `abstract class(pName: PType1, pName2: PType2...) extends SuperClass`

with optional constructor in the class definition:

```
class Person(name: String, age: int) extends
Mammal {
  // secondary constructor
  def this(name: String) {
    // calls to the "primary" constructor
    this(name, 1);
  }
  // members here
}
```

predefined function `classOf[T]` returns Scala class type T

OBJECT

A concrete class instance and is a singleton.

```
object RunRational extends Application
{
  // members here
}
```

MIXIN CLASS COMPOSITION

Mixin:

```
trait RichIterator extends AbsIterator {
  def foreach(f: T => Unit) {
    while (hasNext) f(next)
  }
}
```

Mixin Class Composition:

The first parent is called the superclass of Iter, whereas the second (and every other, if present) parent is called a mixin.

```
object StringIteratorTest {
  def main(args: Array[String]) {
    class Iter extends StringIterator(args(0))
      with RichIterator
    val iter = new Iter
    iter foreach println
  }
}
```

note the keyword "with" used to create a mixin composition of the parents StringIterator and RichIterator.

TRAITS

Like Java interfaces, defines object types by specifying method signatures, can be partially implemented. See example in Mixin.

GENERIC CLASS

```
class Stack[T] {
  // members here
}
```

Usage:

```
object GenericsTest extends Application {
  val stack = new Stack[Int]
  // do stuff here
}
```

note: can also define generic methods

INNER CLASS

example:

```
class Graph {
  class Node {
    var connectedNodes: List[Node] = Nil
    def connectTo(node: Node) {
      if
(connectedNodes.find(node.equals).isEmpty) {
        connectedNodes = node :: connectedNodes
      }
    }
  }
  // members here
}
```

usage:

```
object GraphTest extends Application {
  val g: Graph = new Graph
  val n1: g.Node = g.newNode
  val n2: g.Node = g.newNode
  n1.connectTo(n2)       // legal
  val h: Graph = new Graph
  val n3: h.Node = h.newNode
  n1.connectTo(n3)       // illegal!
}
```

Inner classes are bound to the outer object, so a node type is prefixed with its outer instance and can't mix instances.

CASE CLASSES

See http://www.scala-lang.org/node/107 for info.

METHODS/FUNCTIONS

Methods are Functional Values and Functions are Objects

form: `def name(pName: PType1, pName2: PType2...) : RetType`

use `override` to override a method

```
override def toString() = "" + re + (if (im < 0) "" else "+") + im + "i"
```

Can override for different return type.

"=>" separates the function's argument list from its body

```
def re = real // method without arguments
```

Anonymous:

(function params) | rt. arrow | function body

```
(x : int, y : int) => x + y
```

OPERATORS

All operators are functions on a class.

Have fixed precedences and associativities:

```
(all letters)
|
^
&
< >
= !
:
+ -
/ %
*
(all other special characters)
```

Operators are usually left-associative, i.e. x + y + z is interpreted as (x + y) + z,

except operators ending in colon ':' are treated as right-associative.

An example is the list-consing operator "::". where, `x :: y :: zs` is interpreted as `x :: (y :: zs)`.

eg.

```
def + (other: Complex) : Complex = {
  //....
}
```

Infix Operator:

Any single parameter method can be used :

```
System exit 0
Thread sleep 10
```

unary operators - prefix the operator name with "unary_"

```
def unary_~ : Rational = new Rational(denom, numer)
```

The Scala compiler will try to infer some meaning out of the "operators" that have some predetermined meaning, such as the += operator.

ARRAYS

arrays are classes

```
Array[T]
```

access as function:

```
a(i)
```

parameterize with a type

```
val hellos = new Array[String](3)
```

MAIN

```
def main(args: Array[String])
```

return type is `Unit`

ANNOTATIONS

See http://www.scala-lang.org/node/106

ASSIGNMENT

=

```
protected var x = 0
```

<-

`val x <- xs` is a generator which produces a sequence of values

SELECTION

The else must be present and must result in the same kind of value that the if block does

```
val filename =
  if (options.contains("configFile"))
    options.get("configFile")
  else
    "default.properties"
```

ITERATION

Prefer recursion over looping.

while loop: similar to Java

for loop:

```
// to is a method in Int that produces a Range object
for (i <- 1 to 10; i % 2 == 0) // the left-arrow means "assignment" in Scala
  System.out.println("Counting " + i)
```

`i <- 1 to 10` is equivalent to:

```
for (i <- 1.to(10))
```

`i % 2 == 0` is a filter, optional

```
for (val arg <- args)
```

maps to `args foreach (arg => ...)`

More to come...

REFERENCES

The Busy Developers' Guide to Scala series:

- "Don't Get Thrown for a Loop", IBM developerWorks
- "Class action", IBM developerWorks
- "Functional programming for the object oriented", IBM developerWorks

Scala Reference Manuals:

- "An Overview of the Scala Programming Language" (2. Edition, 20 pages), scala-lang.org
- A Brief Scala Tutorial, scala-lang.org
- "A Tour of Scala", scala-lang.org

"Scala for Java programmers", A. Sundararajan's Weblog, blogs.sun.com