TRANSFORMACIONES

Transformación	Definición	Ejemplo
map(*func*)	Devuelve un nuevo RDD tras pasar cada elemento del RDD original a través de una función.	val v1 = sc.parallelize(List(2, 4, 8)) **val v2 = v1.map(_ * 2)** v2.collect *res0: Array[Int] = Array(4, 8, 16)*
filter(*func*)	Realiza un filtrado de los elementos del RDD original para devolver un nuevo RDD con los datos filtrados.	val v1 = sc.parallelize(List("ABC", "BCD", "DEF")) **val v2 = v1.filter(_.contains("A"))** v2.collect *res0: Array[String] = Array(ABC)*
flatMap(*func*)	Parecido a la operación map, pero la función devuelve una secuencia de valores.	val x = sc.parallelize(List("Ejemplo proyecto Alejandro", "Hola mundo"), 2) val y = x.map(x => x.split(" ")) // split(" ") returns an array of words y.collect *res0: Array[Array[String]] = Array(Array(Ejemplo, proyecto, Alejandro), Array(Hola, mundo))* **val y = x.flatMap(x => x.split(" "))** y.collect *res1: Array[String] = Array(Ejemplo, proyecto, Alejandro, Hola, mundo)*
mapPartitions (*func*)	Similar a la operación *map*, pero se ejecuta por separado en cada partición del RDD.	val a = sc.parallelize(1 to 9, 3) def myfunc[T](iter: Iterator[T]) : Iterator[(T, T)] = { var res = List[(T, T)]() var pre = iter.next while (iter.hasNext) {val cur = iter.next; res .::= (pre, cur) pre = cur;} res.iterator} **a.mapPartitions(myfunc).collect** *res0: Array[(Int, Int)] = Array((2,3), (1,2), (5,6), (4,5), (8,9), (7,8))*
sample(*withReplacement*, *fraction*, *seed*)	Muestra una fracción de los datos, con o sin replazo, utilizando una semilla que genera número aleatorios.	val randRDD = sc.parallelize(List((7,"cat"), (6, "mouse"),(7, "cup"), (6, "book"), (7, "tv"), (6, "screen"), (7, "heater"))) val sampleMap = List((7, 0.4), (6, 0.6)).toMap **randRDD.sampleByKey(false, sampleMap,42).collect** *res0: Array[(Int, String)] = Array((6,book), (7,tv), (7,heater))*
union(*otherDataset*)	Devuelve un nuevo RDD con la unión de los elementos de los RDDs seleccionados.	val a = sc.parallelize(1 to 3, 1) val b = sc.parallelize(5 to 7, 1) ***a.union(b).collect()*** *res0: Array[Int] = Array(1, 2, 3, 5, 6, 7)*
intersection(*otherDataset*)	Devuelve los elementos de los RDDs que son iguales.	val x = sc.parallelize(1 to 20) val y = sc.parallelize(10 to 30) **val z = x.intersection(y)** z.collect

		res0: Array[Int] = Array(16, 14, 12, 18, 20, 10, 13, 19, 15, 11, 17)
distinct([*numTasks*]))	Devuelve los elementos de los RDDs que son distintos.	val c = sc.parallelize(List("Gnu", "Cat", "Rat", "Dog", "Gnu", "Rat"), 2) **c.distinct.collect** *res0: Array[String] = Array(Dog, Gnu, Cat, Rat)*
groupByKey([*numTasks*])	Similar al *grupoBy,* realiza el agrupamiento por clave de un conjunto de datos, pero en lugar de suministrar una función, el componente clave de cada par se presentará automáticamente al particionador.	val a = sc.parallelize(List("dog", "tiger", "lion", "cat", "spider", "eagle"), 2) val b = a.keyBy(_.length) **b.groupByKey.collect** *res0: Array[(Int, Seq[String])] = Array((4,ArrayBuffer(lion)), (6,ArrayBuffer(spider)), (3,ArrayBuffer(dog, cat)), (5,ArrayBuffer(tiger, eagle)))*
reduceByKey(*func*, [*numTasks*])	Devuelve un conjunto de datos de pares (K, V) donde los valores de cada clave son agregados usando la función de reducción dada.	val a = sc.parallelize(List("dog", "cat", "owl", "gnu", "ant"), 2) val b = a.map(x => (x.length, x)) **b.reduceByKey(_ + _).collect** *res0: Array[(Int, String)] = Array((3,dogcatowlgnuant))*
aggregateByKey(*zeroValue*)(*seqOp*, *combOp*, [*numTasks*])	Devuelve un conjunto de datos de pares (K, U) donde los valores de cada clave se agregan utilizando las funciones combinadas dadas y un valor por defecto de: "cero".	val nombres = sc.parallelize(List(("David", 6), ("Abby", 4), ("David", 5), ("Abby", 5))) **nombres.aggregateByKey(0)((k,v) => v.toInt+k, (v,k) => k+v).collect** *res0: Array[(String, Int)] = Array((Abby,9), (David,11))*
sortByKey([*ascending*], [*numTasks*])	Esta función ordena los datos del RDD de entrada y los almacena en un nuevo RDD.	val a = sc.parallelize(List("dog", "cat", "owl", "gnu", "ant"), 2) val b = sc.parallelize(1 to a.count.toInt, 2) val c = a.zip(b) **c.sortByKey(true).collect** *res0: Array[(String, Int)] = Array((ant,5), (cat,2), (dog,1), (gnu,4), (owl,3))* **c.sortByKey(false).collect** *res1: Array[(String, Int)] = Array((owl,3), (gnu,4), (dog,1), (cat,2), (ant,5))*
join(*otherDataset*, [*numTasks*])	Realiza una unión interna utilizando dos RDD de valor clave. Cuando se introduce conjuntos de datos de tipo (K, V) y (K, W), se devuelve un conjunto de datos de (K, (V, W)) para cada clave.	val a = sc.parallelize(List("dog", "salmon", "salmon", "rat", "elephant"), 3) val b = a.keyBy(_.length) val c = sc.parallelize(List("dog","cat","gnu","salmon","rabbit","turkey","wolf","bear","bee"), 3) val d = c.keyBy(_.length) **b.join(d).collect** *res0: Array[(Int, (String, String))] = Array((6, (salmon,salmon)), (6,(salmon,rabbit)), (6, (salmon,turkey)), (6,(salmon,salmon)), (6, (salmon,rabbit)), (6,(salmon,turkey)), (3,(dog,dog)), (3,(dog,cat)), (3,(dog,gnu)), (3,(dog,bee)), (3, (rat,dog)), (3,(rat,cat)), (3,(rat,gnu)), (3,(rat,bee)))*
cogroup(*otherDataset*, [*numTasks*])	Un conjunto muy potente de funciones que permiten agrupar hasta tres valores	val a = sc.parallelize(List(1, 2, 1, 3), 1) val b = a.map((_, "b")) val c = a.map((_, "c"))

	claves de RDDs utilizando sus claves.	val d = a.map((_, "d")) **b.cogroup(c, d).collect** *res0: Array[(Int, (Iterable[String], Iterable[String], Iterable[String]))] = Array((2, (ArrayBuffer(b),ArrayBuffer(c),ArrayBuffer(d))), (3, (ArrayBuffer(b),ArrayBuffer(c),ArrayBuffer(d))), (1, (ArrayBuffer(b, b),ArrayBuffer(c, c),ArrayBuffer(d, d)))*
cartesian(*otherDataset*)	Calcula el producto cartesiano entre dos RDD ,es decir cada elemento del primer RDD se une a cada elemento del segundo RDD, y los devuelve como un nuevo RDD.	val x = sc.parallelize(List(1,2,3,4,5)) val y = sc.parallelize(List(6,7,8,9,10)) **x.cartesian(y).collect** *res0: Array[(Int, Int)] = Array((1,6), (1,7), (1,8), (1,9), (1,10), (2,6), (2,7), (2,8), (2,9), (2,10), (3,6), (3,7), (3,8), (3,9), (3,10), (4,6), (5,6), (4,7), (5,7), (4,8), (5,8), (4,9), (4,10), (5,9), (5,10))*
pipe(*command*, *[envVars]*)	Toma los datos RDD de cada partición y los envía a través de stdin a un shell-command	val a = sc.parallelize(1 to 9, 3) **a.pipe("head -n 1").collect** *res0: Array[String] = Array(1, 4, 7)*
coalesce(numPartitions)	Disminuye el número de particiones en el RDD al número especificado (*numPartitions*)	val y = sc.parallelize(1 to 10, 10) **val z = y.coalesce(2, false)** z.partitions.length *res0: Int = 2*
repartition(*numPartitions*)	Reorganiza aleatoriamente los datos en el RDD para crear más o menos particiones.	val x = (1 to 12).toList val numbersDf = x.toDF("number") numbersDf.rdd.partitions.size *res0: Int = 4* *Partition 00000: 1, 2, 3* *Partition 00001: 4, 5, 6* *Partition 00002: 7, 8, 9* *Partition 00003: 10, 11, 12* **val numbersDfR = numbersDf.repartition(2)** *Partition A: 1, 3, 4, 6, 7, 9, 10, 12* *Partition B: 2, 5, 8, 11*
repartitionAndSortWithinPartitions(partitioner)	Reparte el RDD de acuerdo con el particionador dado y, dentro de cada partición resultante, clasifica los registros por sus claves. Como se puede observar en el ejemplo pedimos que los datos sean organizado en dos particiones: A y C como una particion y, B y D como otra.	>> pairs = sc.parallelize([["a",1], ["b",2], ["c",3], ["d",3]]) >> pairs.collect() # Output [['a', 1], ['b', 2], ['c', 3], ['d', 3]] >>**pairs.repartitionAndSortWithinPartitions(2).glom().collect()** # Output [[('a', 1), ('c', 3)], [('b', 2), ('d', 3)]] // Reorganización basado en cierta condición. >>**pairs.repartitionAndSortWithinPartitions(2,partitionFunc=lambda x: x == 'a').glom().collect()** # Output [[('b', 2), ('c', 3), ('d', 3)], [('a', 1)]]

ACCIONES

Transformación	Definición	Ejemplo
reduce(*func*)	Agrega los elementos del dataset usando una función. Esta función debe ser conmutativa y asociativa para que pueda calcularse correctamente en paralelo.	val a = sc.parallelize(1 to 100, 3) **a.reduce(_ + _)** *res0: Int = 5050*
collect()	Convierte un RDD en un array[8] y lo muestra por pantalla.	val c = sc.parallelize(List("Gnu", "Cat", "Rat", "Dog", "Gnu", "Rat"), 2) **c.collect** *res0: Array[String] = Array(Gnu, Cat, Rat, Dog, Gnu, Rat)*
count()	Devuelve el número de elementos del dataset.	val a = sc.parallelize(1 to 4) **a.count** *res0: Long = 4*
first()	Devuelve el primer elemento del conjunto de datos	val c = sc.parallelize(List("Gnu", "Cat", "Rat", "Dog"), 2) **c.first** *res0: String = Gnu*
take(*n*)	Devuelve un array con los primeros *n* elementos del dataset.	val b = sc.parallelize(List("dog", "cat", "ape", "salmon", "gnu"), 2) **b.take(2)** *res0: Array[String] = Array(dog, cat)*
takeSample(*withReplacement*, *num*, [*seed*])	Devuelve un array con una muestra aleatoria de elementos numéricos del dataset, con o sin sustitución, con la opción de especificar opcionalmente una semilla de generador de números aleatorios.	val x = sc.parallelize(1 to 200, 3) **x.takeSample(true, 20, 1)** *res0: Array[Int] = Array(74, 164, 160, 41, 123, 27, 134, 5, 22, 185, 129, 107, 140, 191, 187, 26, 55, 186, 181, 60)*
takeOrdered(*n*, *[ordering]*)	Devuelve los primeros *n* elementos del RDD usando su orden original o un comparador personalizado.	val b = sc.parallelize(List("dog", "cat", "ape", "salmon", "gnu"), 2) **b.takeOrdered(2)** *res0: Array[String] = Array(ape, cat)*
saveAsTextFile(*path*)	Guarda el RDD como un archivos de texto.	val a = sc.parallelize(1 to 10000, 3) **a.saveAsTextFile("/home/usuario/datos")** root@master:/home/usuario/datos # ls *part-00000 part-00001 part-00002 _SUCCESS* // Como se puede observar se han creado las 3 particiones, las cuales hemos especificado.

saveAsSequenceFile(*path*)	Guarda el RDD como un archivo de secuencia Hadoop.	val v = sc.parallelize(Array(("owl",3), ("gnu",4), ("dog",1), ("cat",2), ("ant",5)), 2) **v.saveAsSequenceFile("/home/usuario/seq_datos")** root@master:/home/usuario/seq_datos# ls -a *. .. part-00000 .part-00000.crc part-00001 .part-00001.crc _SUCCESS ._SUCCESS.crc*
saveAsObjectFile(*path*) (Java and Scala)	Guarda los elementos del conjunto de datos en un formato simple utilizando la serialización de Java	val x = sc.parallelize(Array(("owl",3), ("gnu",4), ("dog",1), ("cat",2), ("ant",5)), 2) **x.saveAsObjectFile("/home/usuario/objFile")** root@master:/home/usuario/objFile # ls -a *. .. part-00000 .part-00000.crc part-00001 .part-00001.crc _SUCCESS ._SUCCESS.crc*
countByKey()	Sólo disponible en RDD de tipo (K, V). Devuelve un hashmap de pares (K, Int) con el recuento de cada clave.	val c = sc.parallelize(List((3, "Gnu"), (3, "Yak"), (5, "Mouse"), (3, "Dog")), 2) **c.countByKey** res3: scala.collection.Map[Int,Long] = Map(3 -> 3, 5 -> 1)
foreach(*func*)	Ejecute una función func en cada elemento del dataset.	val c = sc.parallelize(List("cat", "dog", "tiger", "lion", "gnu", "crocodile", "ant", "whale", "dolphin", "spider"), 3) **c.foreach(x => println(x + "s are beautiful"))** *cats are beautiful dogs are beautiful tigers are beautiful ants are beautiful whales are beautiful dolphins are beautiful spiders are beautiful lions are beautiful gnus are beautiful crocodiles are beautiful*