

# Trabajando con RDDs

**RDD:** *Resilient Distributed Dataset*: es una simple, RESILIENTE e INMUTABLE colección DISTRIBUIDA de objetos.

Es **RESILIENTE** (tolerante a fallos): Podemos recuperar los datos en memoria si se pierden.

Es **INMUTABLE**: no podemos realizar modificaciones sobre un mismo RDD. Si queremos modificarlo tendremos que crear uno nuevo.

Es **DISTRIBUIDO**: cada RDD es dividido en múltiples particiones automáticamente. El RDD puede ser ejecutado en diferentes nodos del clúster.

**DATASET**: Conjunto de datos. 2 tipos de fuentes: externas, datos en memoria.

Los usuarios pueden crear RDDs de dos maneras:

Transformando un RDD que ya existe.

Desde un objeto SparkContext distribuyendo una colección de objetos (ej: una lista) en su driver.

2 tipos de operaciones sobre los RDD:

**Transformaciones**: operaciones que crean un nuevo RDD.

Ejemplo: Filtrar un RDD por las líneas que contengan una cadena.

**Acciones**: operaciones que devuelven un resultado. Devuelven cualquier tipo de datos menos un RDD.

Ejemplo: contar el número de líneas de un RDD.

El Spark Context es la manera que tenemos de comunicarnos con el clúster.

El método Parallelize convierte una Scala Collection local en un RDD.

```
scala> val lines = sc.parallelize(List("pandas", "i like pandas"))
lines: org.apache.spark.rdd.RDD[String] = ParallelCollectionRDD[1] at paralleliz
e at <console>:24
```

El método textFile lee un fichero de texto desde HDFS o Local y lo transforma en un RDD de tipo.

```
scala> val textlines=sc.textFile("README.md")
textlines: org.apache.spark.rdd.RDD[String] = README.md MapPartitionsRDD[3] at t
extFile at <console>:24
```

## Transformaciones

Son perezosas: Spark no procesa las transformaciones sobre RDD's hasta que no se ejecuta una acción sobre ellos.

El RDD resultado no es inmediatamente computado

## Acciones

No son perezosas: Eager

El resultado es inmediatamente computado



Importante!

Esta es la forma que tiene Spark de reducir el tráfico de red. Una vez que Spark ve toda la cadena de transformaciones, puede procesar solo los datos que necesita para el resultado de la acción.

Ejemplo: Solo ejecutará las operaciones necesarias para obtener el valor del primer elemento del RDD.

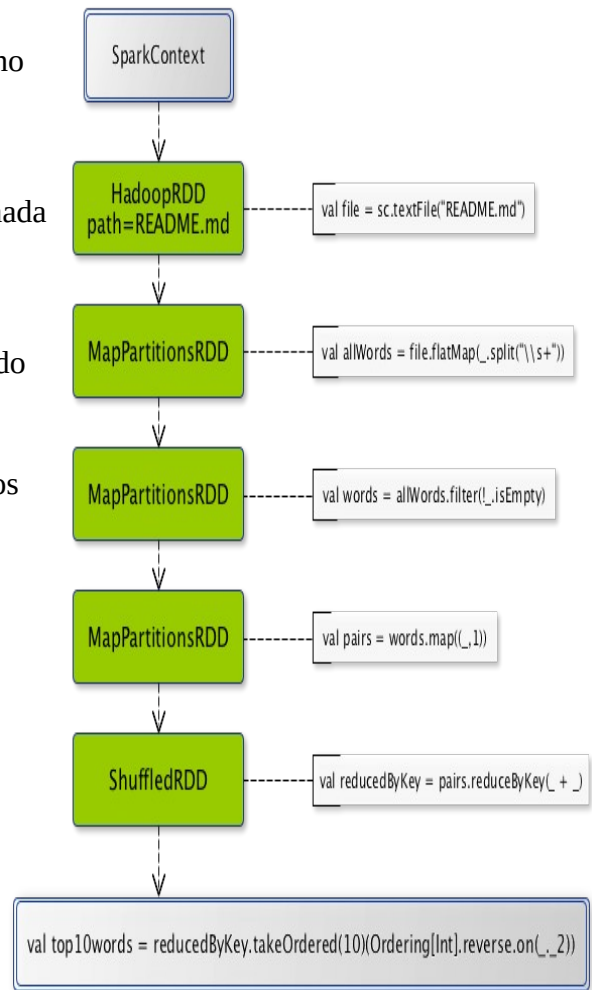
```
scala> println(lines.first())
pandas
```

Aunque pueda resultar sorprendente a primera vista, trabajar con transformaciones y acciones tiene mucho sentido al trabajar con Big Data.

Imaginemos, por ejemplo, que queremos cargar un gran dataset de URLs y posteriormente contar las veces que una determinada IP accede a una determinada URL.

Si ejecutásemos la primera transformación según se escribe el comando, perderíamos mucho tiempo y espacio al almacenar todo el dataset en memoria dado que solo queremos unas determinadas IPs y URLs.

En su lugar, Spark interpreta la cadena total de transformaciones y computa solamente aquellos datos que necesita para obtener el resultado

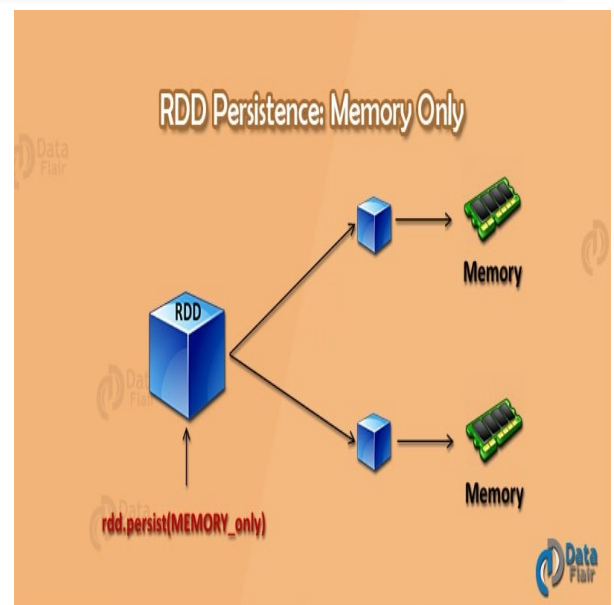


Como hemos visto, los RDDs son computados cada vez que se realiza una acción.

Si se va a utilizar el mismo RDD varias veces es interesante persistir los resultados para evitar la recomputación de los mismos datos, usando `RDD.persist()`

La persistencia hace que Spark almacene los resultados en memoria, particionados a lo largo del clúster para ser reutilizados.

Existen diferentes niveles de persistencia (memoria, disco, disco y memoria), pero su utilización no interesa para el objetivo del curso.



## Transformaciones

Aplican Lazy Evaluation

```
val inputRDD = sc.textFile("log.txt")
val errorsRDD = inputRDD.filter(line => line.contains("error"))
```

En este ejemplo, filter no cambia el contenido de inputRDD, simplemente crea punteros a un nuevo RDD, de manera que puede volver a ser reutilizado.

```
errorsRDD = inputRDD.filter(line => line.contains("error"))
warningsRDD = inputRDD.filter(line => line.contains("warning"))
badLinesRDD = errorsRDD.union(warningsRDD)
```

Esta forma de trabajar hace que, en caso de pérdida de datos, se pueda volver a procesar todo de nuevo, de manera que no se pierda nada.

### Transformaciones más usadas

#### map()

que genera un nuevo RDD aplicando alguna función sobre cada línea del RDD

#### filter()

Toma una función y la aplica a los elementos del RDD que cumplen el filtro. Básicamente hace lo mismo que el "where" en SQL

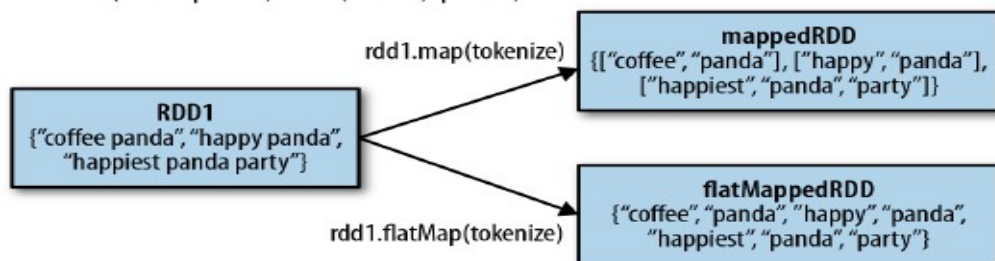
#### flatMap()

A veces queremos obtener como salida varios elementos dado uno de entrada.

Como map(), la función que le pasamos se aplica a cada elemento del RDD individualmente, pero en lugar de devolver un solo resultado compuesto por el resultado de aplicar dicha función a cada elemento, se itera sobre cada elemento inicial y se devuelve un valor por cada iteración.

Una forma sencilla de entenderlo es, teniendo como entrada una frase, devolver una lista de palabras.

```
tokenize("coffee panda") = List("coffee", "panda")
```



## ACCIONES

La acción más típica es

**reduce()** -- los tipos de entrada y salida son iguales.

Opera sobre dos elementos del RDD y devuelve un resultado.

El ejemplo más simple es la suma de elementos.

De este modo es sencillo realizar sumas, cuentas o agregaciones.

```
val sum = rdd.reduce((x, y) => x + y)
```

Todos los elementos del RDD deben ser del mismo tipo

Otro ejemplo, operación foldLeft (el tipo de datos de entrada es de un tipo y la salida puede ser de otro)

```
def sum(list: List[Int]): Int = list.foldLeft(0)((r,c) => r+c)
def sum(list: List[Int]): Int = list.foldLeft(0)(_+_)
```

Tenemos una lista de enteros

Valor inicial 0

R= resultado (acumulador)

C= current valor

Función: Suma secuencial, elemento actual más valor del siguiente. Hasta aquí todo correcto: se van sumando los valores secuencialmente.

Existe la función **collect()** para devolver el dataset completo, pero recordemos que el dataset será copiado al driver (nodo maestro) y una excepción será lanzada si es demasiado grande para caber en memoria, por lo que solo lo usaremos para pequeños datasets.

Para casos en los que queramos usar **collect()** en datasets muy grandes, podemos hacer que su resultado se almacene en algún sistema de almacenamiento, en lugar de en el driver, como Amazon s3 o HDFS.

Para ello podemos usar funciones como

**saveAsSequenceFile()**

Recordad que cada vez que ejecutamos una Acción, se debe ejecutar la secuencia completa de transformaciones sobre el RDD, por lo que conviene tener en mente la opción de persistir datos intermedios de vez en cuando.

### Acciones más usadas

**count()**, que cuenta el número de elementos (filas) de un RDD

**take(n)**, que devuelve un array de n elementos

**collect()**, que devuelve un array de todos los elementos


**saveAsTextfile(file)**, que guarda el RDD a un fichero de texto


### Trabajando con RDDs: foldLeft

\*No todo es paralelizable y distribuible: dividir el total en piezas y trabajar sobre cada pieza. Pero, y si en lugar de una suma de enteros tenemos algo como esto?

```
def foldLeft[B](z: B)(f: (B, A) => B): B  A, B representa el tipo de datos de
                                         entrada

val xs = List(1, 2, 3, 4)
val res = xs.foldLeft("")(str: String, i: Int) => str + i
```



Origen Lista de enteros  Resultado string

Qué ocurre si dividimos esta tarea en dos para paralelizar?

1ª iteración paralela: dividimos la lista de enteros en dos piezas y aplicamos la función en paralelo

2ª iteración: combinación de valores. Cada pieza es de tipo String (no int como la lista inicial), por lo que la función no aplica.

Si la lista fuera de strings, sería diferente.

Qué solución tenemos para este tipo de circunstancias?

Solución: utilizar la función **aggregate**

```
aggregate[B](z: => B)(seqop: (B, A) => B, combop: (B, B) => B): B
```

Contiene dos funciones

Seqop: operaciones secuenciales con datos de dos tipos diferentes A,B.

Combop: operaciones de combinación, siempre del mismo tipo B.

Esta es una de las funciones más usadas en Spark.

Es paralelizable.

Permite cambiar el tipo de los datos que devuelve.

# Listado de transformaciones

$RDD = \{1, 2, 3, 3\}$

Function name	Purpose	Example	Result
<code>map()</code>	Apply a function to each element in the RDD and return an RDD of the result.	<code>rdd.map(x =&gt; x + 1)</code>	<code>{2, 3, 4, 4}</code>
<code>flatMap()</code>	Apply a function to each element in the RDD and return an RDD of the contents of the iterators returned. Often used to extract words.	<code>rdd.flatMap(x =&gt; x.to(3))</code>	<code>{1, 2, 3, 2, 3, 3, 3}</code>
<code>filter()</code>	Return an RDD consisting of only elements that pass the condition passed to <code>filter()</code> .	<code>rdd.filter(x =&gt; x != 1)</code>	<code>{2, 3, 3}</code>
<code>distinct()</code>	Remove duplicates.	<code>rdd.distinct()</code>	<code>{1, 2, 3}</code>
<code>sample(withReplacement, fraction, [seed])</code>	Sample an RDD, with or without replacement.	<code>rdd.sample(false, 0.5)</code>	Nondeterministic
<code>union()</code>	Produce an RDD containing elements from both RDDs.	<code>rdd.union(other)</code>	<code>{1, 2, 3, 3, 4, 5}</code>
<code>intersection()</code>	RDD containing only elements found in both RDDs.	<code>rdd.intersection(other)</code>	<code>{3}</code>
<code>subtract()</code>	Remove the contents of one RDD (e.g., remove training data).	<code>rdd.subtract(other)</code>	<code>{1, 2}</code>
<code>cartesian()</code>	Cartesian product with the other RDD.	<code>rdd.cartesian(other)</code>	<code>{(1, 3), (1, 4), ... (3,5)}</code>

# Listado de acciones

Function name	Purpose	Example	Result
<code>collect()</code>	Return all elements from the RDD.	<code>rdd.collect()</code>	<code>{1, 2, 3, 3}</code>
<code>count()</code>	Number of elements in the RDD.	<code>rdd.count()</code>	4
<code>countByValue()</code>	Number of times each element occurs in the RDD.	<code>rdd.countByValue()</code>	<code>{(1, 1), (2, 1), (3, 2)}</code>
<code>take(num)</code>	Return num elements from the RDD.	<code>rdd.take(2)</code>	<code>{1, 2}</code>
<code>top(num)</code>	Return the top num elements the RDD.	<code>rdd.top(2)</code>	<code>{3, 3}</code>
<code>takeOrdered(num)(ordering)</code>	Return num elements based on provided ordering.	<code>rdd.takeOrdered(2)(nyOrdering)</code>	<code>{3, 3}</code>
<code>takeSample(withReplacement, num, [seed])</code>	Return num elements at random.	<code>rdd.takeSample(false, 1)</code>	Nondeterministic
<code>reduce(func)</code>	Combine the elements of the RDD together in parallel (e.g., sum).	<code>rdd.reduce((x, y) =&gt; x + y)</code>	9
<code>fold(zero)(func)</code>	Same as <code>reduce()</code> but with the provided zero value.	<code>rdd.fold(0)((x, y) =&gt; x + y)</code>	9