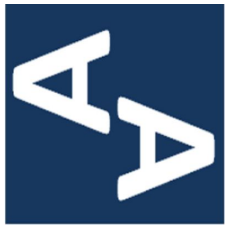
Programación funcional en Scala

4. Introducción a Spark



Habla Computing
info@hablapps.com
@hablapps

Objetivos

- Conocer los fundamentos de Apache Spark:
  - *Resilient Distributed Datasets* (**RDD**s)
  - Arquitectura y API

- Entender cómo Apache Spark aplica conceptos de PF
  - Separación entre lenguaje e intérprete
  - Funciones de orden superior
  - Type classes

Spark y la programación funcional

- **Sección 0 - Introducción a Apache Spark**
- Sección 1 - Core RDDs
- Sección 2 - Double RDDs
  - Ejercicio 1 - Regresión lineal
- Sección 3 - Pair RDDs
- Sección 4 - Ordered RDDs
  - Ejercicio 2 - Quijote
  - Ejercicio 3 - Logs
- Sección 5 - Lenguaje de procesado de datos
- Sección 6 - Conclusión del curso

¿Qué es Apache Spark?

- Un framework para el procesado de datos
- Cantidades masivas de datos (*bigdata*)
- Proceso distribuido: el código se manda donde están los datos

Resilient Distributed Datasets (RDDs)

- Colecciones *inmutables* y *tipadas* `RDD[T]`

- RDDs se crean:
  - A partir de datos existentes
  - O como resultado de operaciones en otros RDDs

- Todo el trabajo se realiza mediante operaciones sobre RDDs

Lectura/escritura datos en RDDs

- Volcado datos en RDDs:
  - Local (*hardcoded* o lectura de fichero local)
  - Lectura de sistema remoto (HDFS, Cassandra...)

- Volcado de datos de RDDs:
  - Local: ‘traer’ a mem o escribir en fichero local
  - Escritura en sistema remoto (HDFS, Cassandra...)

Ejemplo RDD

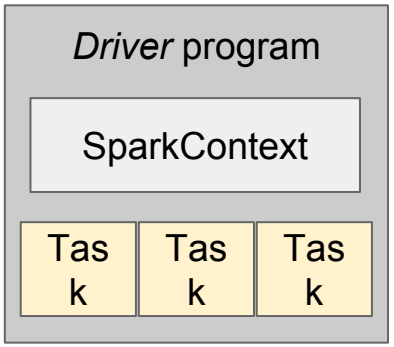
```scala
// Creando un SparkContext
val conf = new SparkConf()
  .setMaster("local")
  .setAppName("Example")
val sc = new SparkContext(conf)

// Creamos un RDD
val dice: RDD[Int] = sc.parallelize(1 to 6)
// Transformamos los datos
val oddDice: RDD[Int] = dice.filter(_ % 2 != 0)
// Materializamos los datos
val res: Array[Int] = oddDice.collect()
// res: Array[Int] = Array(1, 3, 5)
```

PF en API Spark

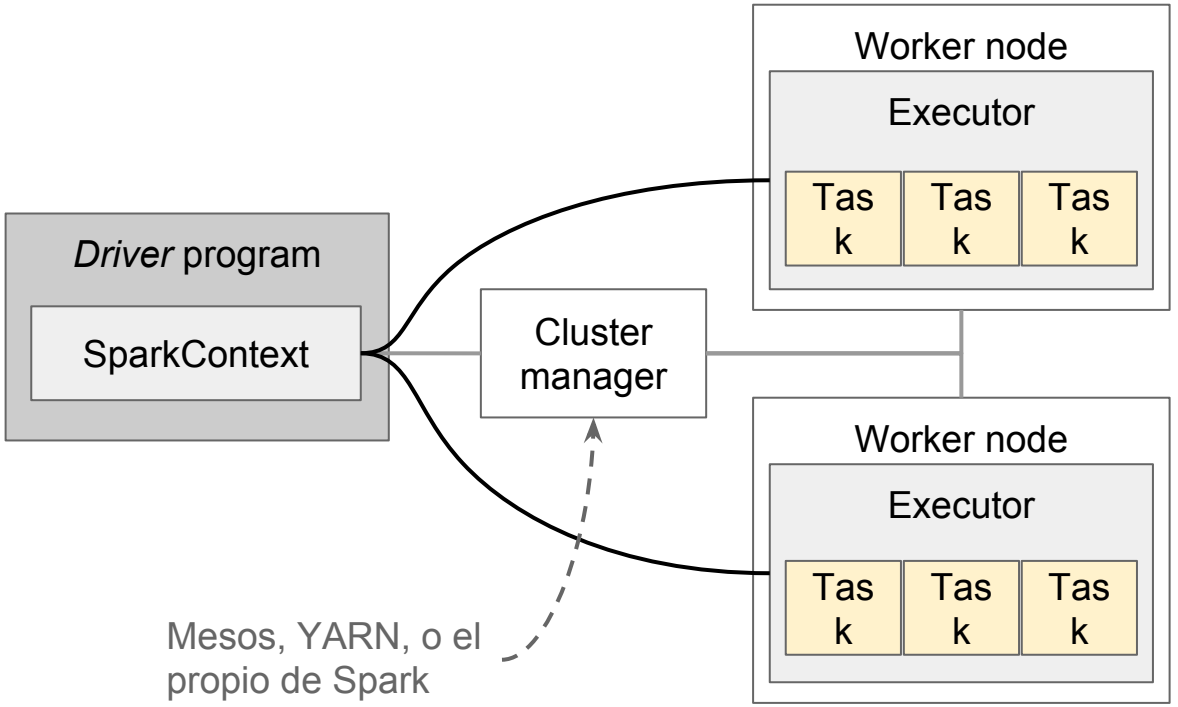
- Las **transformaciones** son *funciones puras*
  - Definidas como *lazy*, definen qué hacer pero no lo ejecutan en el momento
- Los **RDDs** con sus transformaciones definen un *lenguaje* de transformación de datos
  - Un **RDD** es un *programa* en ese lenguaje
- La *interpretación* del lenguaje se realiza cada vez que se ejecuta una **acción**

Arquitectura Spark

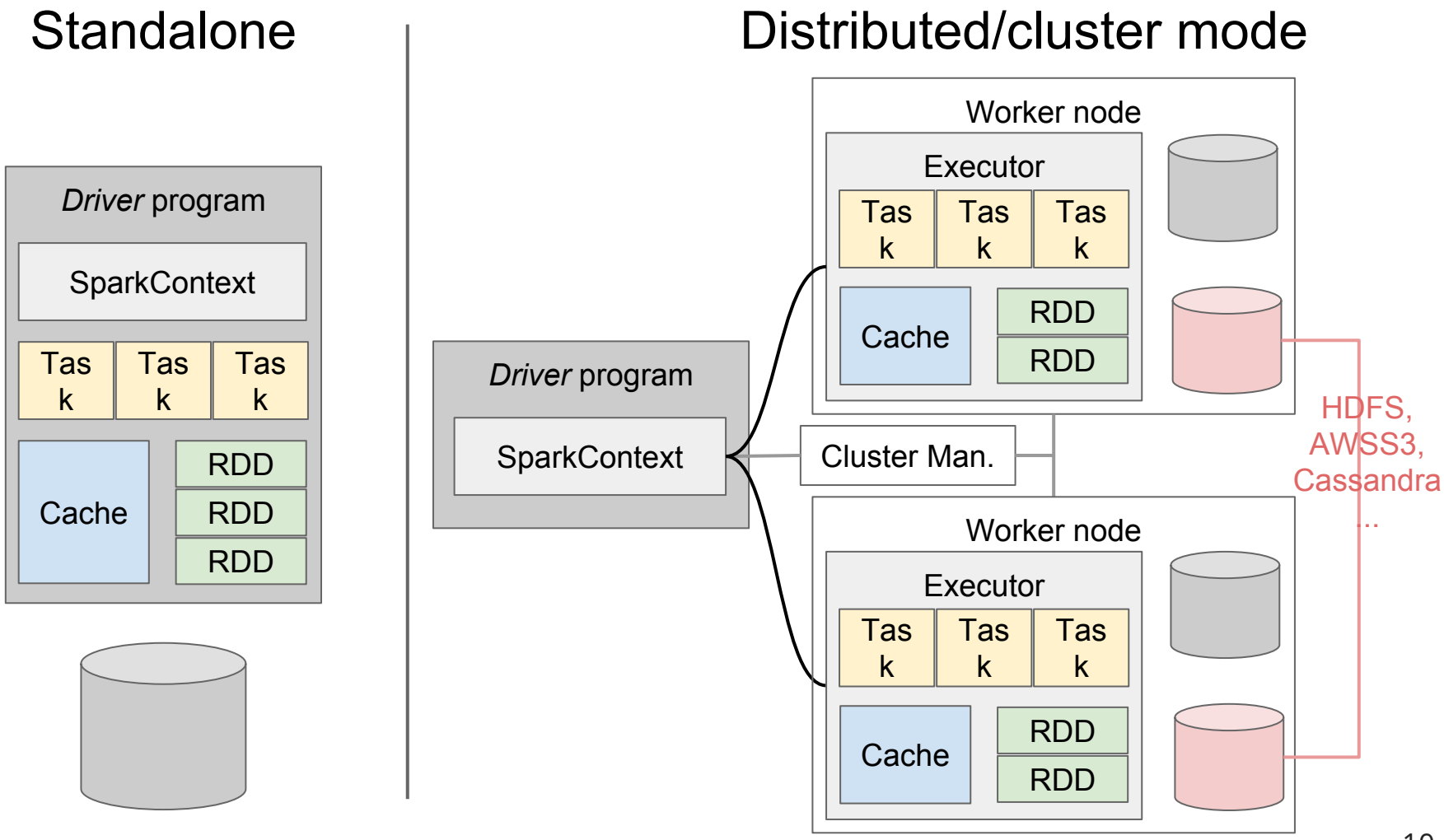
Standalone



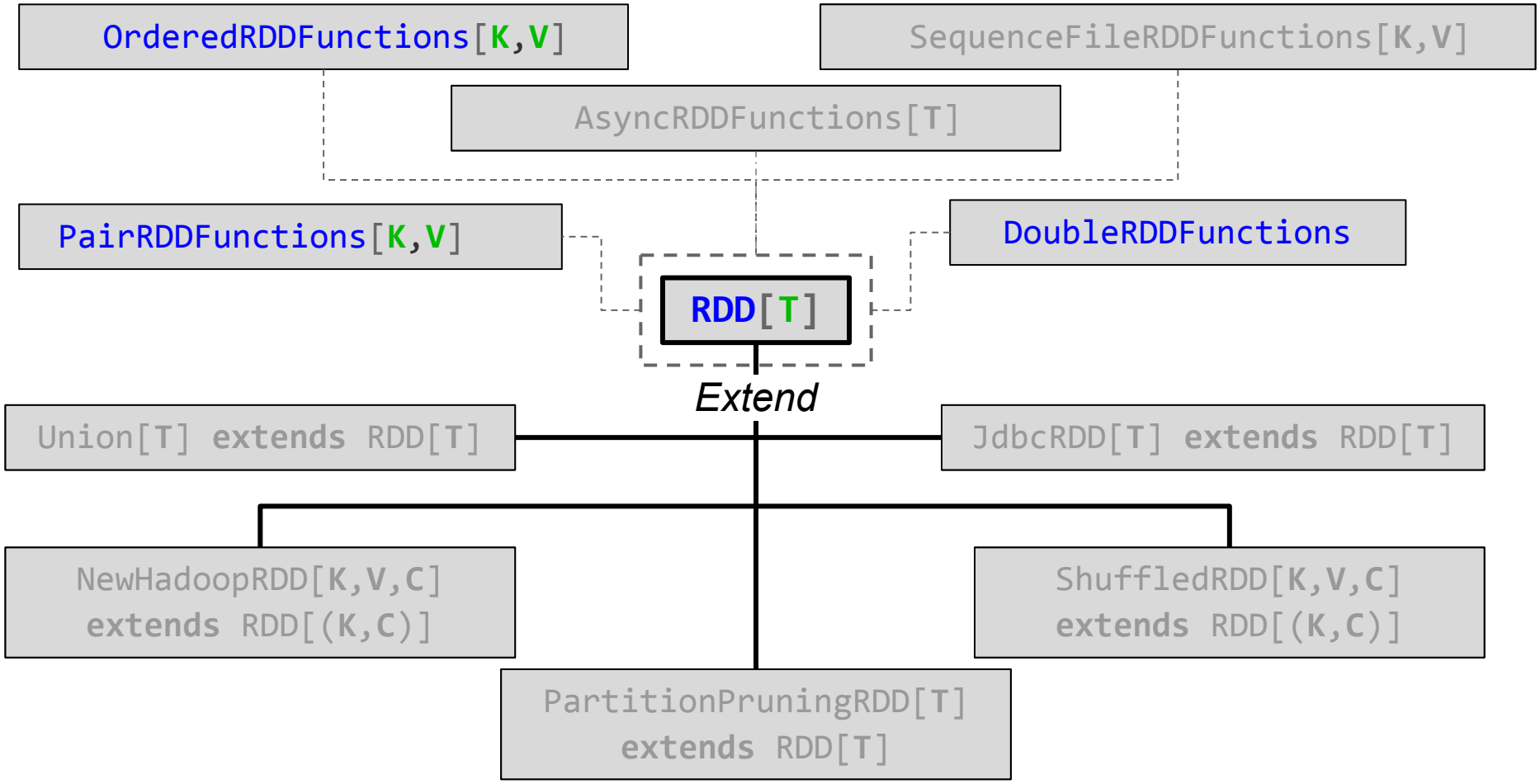
Distributed/cluster mode



Arquitectura Spark II , RDDs



API RDDs



- Todas las ops sobre datos son o bien ***transformaciones*** o bien ***acciones***

Spark y la programación funcional

- Sección 0 - Introducción a Apache Spark
- **Sección 1 - Core RDDs**
- Sección 2 - Double RDDs
  - Ejercicio 1 - Regresión lineal
- Sección 3 - Pair RDDs
- Sección 4 - Ordered RDDs
  - Ejercicio 2 - Quijote
  - Ejercicio 3 - Logs
- Sección 5 - Lenguaje de procesado de datos
- Sección 6 - Conclusión del curso

Trans. y acciones en RDDs

RDD[T]

Transformaciones

```
filter(f:(T)⇒Boolean): RDD[T]
map[U](f:(T)⇒U): RDD[U]
flatMap[U](f:(T)⇒TraversableOnce[U]):RDD[U]

union(other: RDD[T]): RDD[T]
subtract(other: RDD[T]): RDD[T]
intersection(other: RDD[T]): RDD[T]

cartesian[U](other:RDD[U]): RDD[(T,U)]
distinct(): RDD[T]
groupBy[K](f: (T)⇒K): RDD[(K,Iterable[T])]
sortBy[K](f:(T)⇒K)
         (implicit o:Ordering[K]):RDD[T]

persist(level:StorageLevel): RDD.this.type
cache(): RDD.this.type
```

Acciones

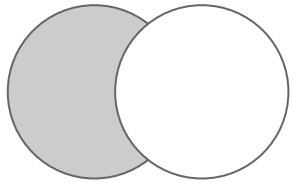
```
aggregate[U](z:U)(o:(U,T)⇒U,o2:(U,U)⇒U):U
reduce(f: (T, T) ⇒ T): T

isEmpty(): Boolean
count(): Long
countByValue(): Map[T, Long]

first(): T
take(num: Int): Array[T]
collect(): Array[T]
foreach(f: (T) ⇒ Unit): Unit
```

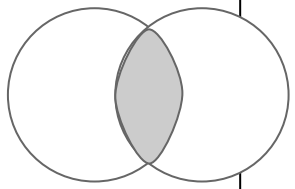
Transformaciones RDDs

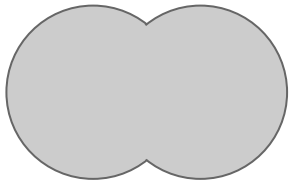
```scala
// Leemos las líneas de un fichero
val lines: RDD[String] = sc.textFile("quijote.txt")

// Separamos las palabras
val words: RDD[String] =
  lines.flatMap(_.split(" "))

// Filtramos las palabras vacías
val nonEmptyWords: RDD[String] =
  words.filter(!_.empty)

// Nos quedamos con la longitud de las palabras
val wordsLength: RDD[Int] =
  nonEmptyWords.map(_.length)
```

Transformaciones RDDs II

```scala
// RDDs como 'conjuntos'
val dice: RDD[Int] = sc.parallelize(1 to 6)
val oddDice: RDD[Int] = dice.filter(_ % 2 == 1)

// SUBSTRACT
val evenDice: RDD[Int] = dice subtract oddDice

// INTERSECTION
val empty: RDD[Int] = evenDice insersection oddDice

// UNION
val dice2: RDD[Int] = evenNums union oddNums
```







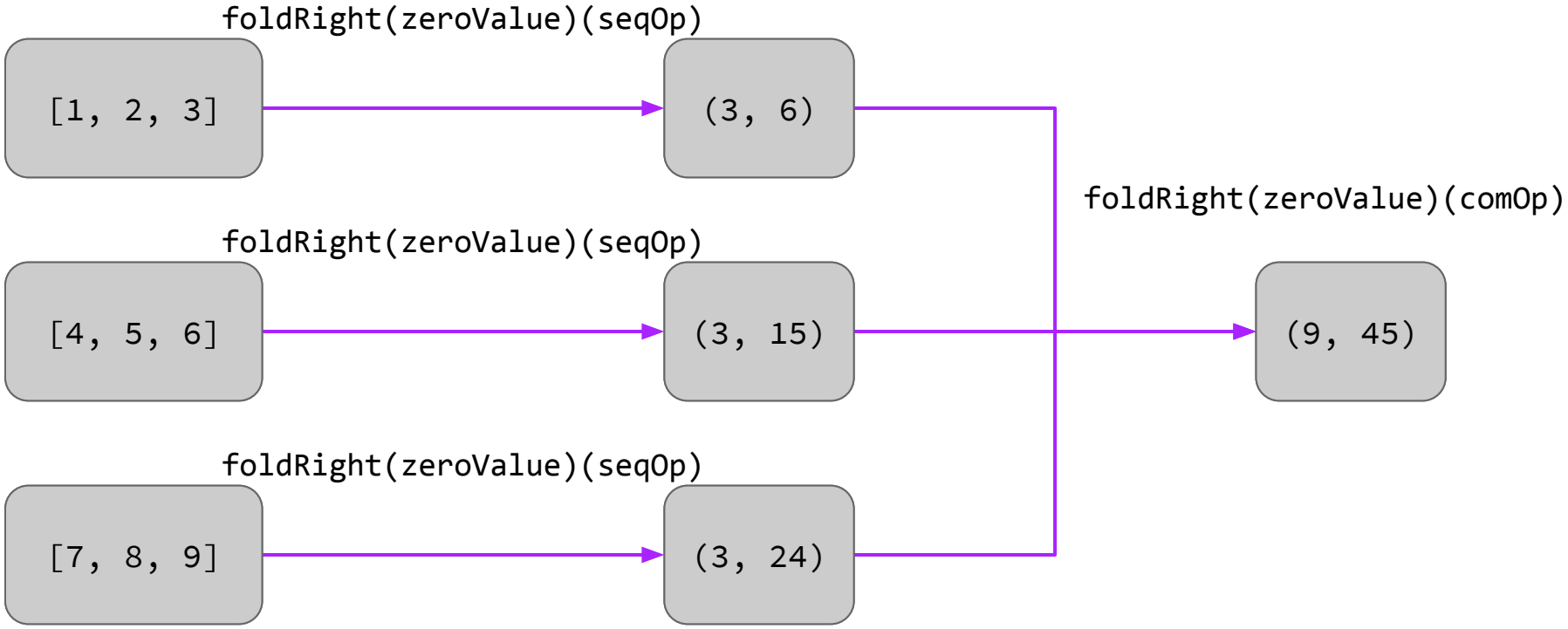
Acciones RDDs

```
dice.count
// 6

dice.collect
// Array(1, 2, 3, 4, 5, 6)

dice take 4
// Array(1, 2, 3, 4)

dice.first
// 1
```

```
dice.isEmpty
// false

dice.reduce(_ + _)
// 21

dice.fold(0)(_ + _)
// 21
```

Aggregate

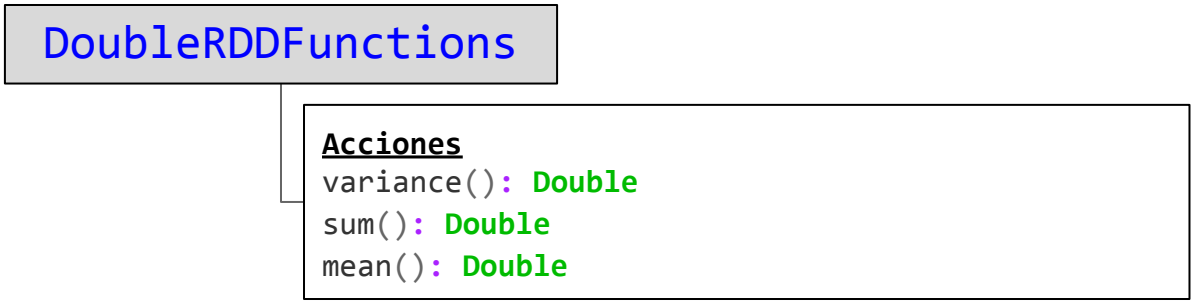
```
zeroValue = (0, 0)
seqOp = (x, acc) =>
  (acc._1 + 1, acc._2 + x)
comOp = (p1, p2) =>
  (p1._1 + p2._1, p1._2 + p2._2)
```



Spark y la programación funcional

- Sección 0 - Introducción a Apache Spark
- Sección 1 - Core RDDs
- **Sección 2 - Double RDDs**
  - Ejercicio 1 - Regresión lineal
- Sección 3 - Pair RDDs
- Sección 4 - Ordered RDDs
  - Ejercicio 2 - Quijote
  - Ejercicio 3 - Logs
- Sección 5 - Lenguaje de procesado de datos
- Sección 6 - Conclusión del curso

DoubleRDDFunctions



Transformaciones DoubleRDDs

```
val randomNumbers: RDD[Int] = ???
val numsLocal: RDD[String] = ???

randomNumbers.sum
// 477082.0

randomNumbers.variance
// 82486.14327599997

randomNumbers.mean
// 477.08199999999977

numsLocal.map(_.toInt).sum
// 55.0
```

Spark y la programación funcional

- Sección 0 - Introducción a Apache Spark
- Sección 1 - Core RDDs
- Sección 2 - Double RDDs
  - **Ejercicio 1 - Regresión lineal**
- Sección 3 - Pair RDDs
- Sección 4 - Ordered RDDs
  - Ejercicio 2 - Quijote
  - Ejercicio 3 - Logs
- Sección 5 - Lenguaje de procesado de datos
- Sección 6 - Conclusión del curso

Ejercicio1: Regresión Lineal Simple

- Dadas dos series de valores $\boldsymbol{x}$ e $\boldsymbol{y}$, se busca la función lineal que los relacione

$$y = f(x) = a + bx$$

- La *regresión lineal simple* nos dice cómo calcular a y b:

$$b = \frac{\sum_{i=1}^{n}(x_i - \overline{x})(y_i - \overline{y})}{\sum_{i=1}^{n}(x_i - \overline{x})^2} \qquad a = \overline{y} - b\overline{x}$$

Spark y la programación funcional

- Sección 0 - Introducción a Apache Spark
- Sección 1 - Core RDDs
- Sección 2 - Double RDDs
  - Ejercicio 1 - Regresión lineal
- **Sección 3 - Pair RDDs**
- Sección 4 - Ordered RDDs
  - Ejercicio 2 - Quijote
  - Ejercicio 3 - Logs
- Sección 5 - Lenguaje de procesado de datos
- Sección 6 - Conclusión del curso

PairRDDFunctions

```
PairRDDFunctions[K,V]
```

Transformaciones

```
aggregateByKey[U](zero: U)
  (seqOp:(U,V)⇒U,coOp:(U,U)⇒U): RDD[(K,U)]

foldByKey(zero: V)
  (func: (V, V) ⇒ V): RDD[(K, V)]

combineByKey[C](
  create: (V) ⇒ C,
  merge: (C, V) ⇒ C,
  merge2: (C, C)⇒ C): RDD[(K,C)]

groupByKey(): RDD[(K, Iterable[V])]

cogroup[W](other: RDD[(K,W)])
  :RDD[(K,(Iterable[V],Iterable[W]))]

join[W](other: RDD[(K, W)]): RDD[(K, (V, W))]
flatMapValues[U](f: (V) ⇒ TraversableOnce[U]):
  RDD[(K, U)]
```

Acciones

```
collectAsMap(): Map[K, V]
countByKey(): Map[K, Long]
lookup(key: K): Seq[V]
```

Explicación *cogroup*

`PairRDD[Int, (String, Int)]`

1	(“0.0.0.1”, 1)
2	(“0.0.0.2”, 2)
4	(“0.0.0.4”, 4)

`PairRDD[Int, (String, String)]`

1	(“ERROR,” “aaa”)
1	(“ERROR”, “bbb”)
2	(“ERROR”, “ccc”)
2	(“INFO”, “ddd”)
3	(“INFO”, “eee”)

1	(((“0.0.0.1”,1)), ((“ERROR”,“aaa”), (“ERROR”,“bbb”)))
2	(((“0.0.0.2”,2)), ((“ERROR”,“ccc”), (“INFO”,“ddd”)))
3	((), ((“ERROR”, “eee”)))
4	(((“0.0.0.4”,4)), ())

`PairRDD[Int, (Iterable[(String,Int)],Iterable[(String,String)])]`

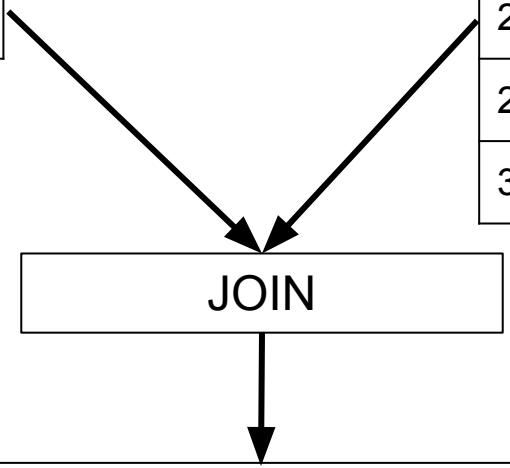
Explicación *join*

```
PairRDD[Int, (String, Int)]
```

1	(“0.0.0.1”, 1)
2	(“0.0.0.2”, 2)
4	(“0.0.0.4”, 4)

```
PairRDD[Int, (String, String)]
```

1	(“ERROR,” “aaa”)
1	(“ERROR”, “bbb”)
2	(“ERROR”, “ccc”)
2	(“INFO”, “ddd”)
3	(“INFO”, “eee”)



1	((“0.0.0.1”, 1), (“ERROR,” “aaa”))
1	((“0.0.0.1”, 1), (“ERROR”, “bbb”))
2	((“0.0.0.2”, 2), (“ERROR”, “ccc”))
2	((“0.0.0.2”, 2), (“INFO”, “ddd”))

```
PairRDD[Int, ((String, Int), (String, String))]
```

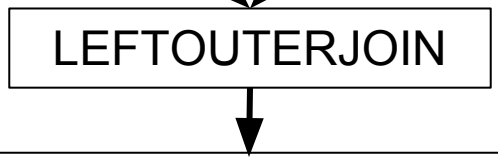
Explicación *leftOuterJoin*

`PairRDD[Int, (String, Int)]`

1	(“0.0.0.1”, 1)
2	(“0.0.0.2”, 2)
4	(“0.0.0.4”, 4)

`PairRDD[Int, (String, String)]`

1	(“ERROR,” “aaa”)
1	(“ERROR”, “bbb”)
2	(“ERROR”, “ccc”)
2	(“INFO”, “ddd”)
3	(“INFO”, “eee”)



1	((“0.0.0.1”,1), Some((“ERROR”,“aaa”)))
1	((“0.0.0.1”,1), Some((“ERROR”,“bbb”)))
2	((“0.0.0.2”,2), Some((“ERROR”,“ccc”)))
2	((“0.0.0.2”,2), Some((“INFO”,“ddd”)))
4	((“0.0.0.4”,4), None)

`PairRDD[Int, ((String,Int),Option[(String,String)])]`

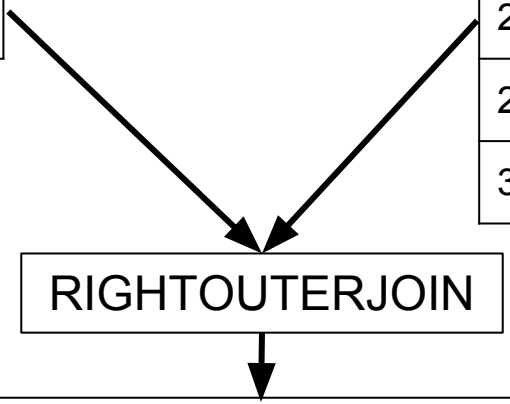
Explicación *rightOuterJoin*

`PairRDD[Int, (String, Int)]`

1	(“0.0.0.1”, 1)
2	(“0.0.0.2”, 2)
4	(“0.0.0.4”, 4)

`PairRDD[Int, (String, String)]`

1	(“ERROR,” “aaa”)
1	(“ERROR”, “bbb”)
2	(“ERROR”, “ccc”)
2	(“INFO”, “ddd”)
3	(“INFO”, “eee”)



1	(Some((“0.0.0.1”,1)), (“ERROR”,“aaa”))
1	(Some((“0.0.0.1”,1)), (“ERROR”,“bbb”))
2	(Some((“0.0.0.2”,2)), (“ERROR”,“ccc”))
2	(Some((“0.0.0.2”,2)), (“INFO”,“ddd”))
3	(None, (“INFO”, “eee”))

`PairRDD[Int, (Option[(String,Int)],(String,String))]`

Transformaciones PairRDDs

```
// PairRDDFunctions
val bills: RDD[(String,Int)] =
  sc.parallelize(("Luis", 5) :: ("Javi", 3) :: ... :: Nil)

val amountsWithVAT: RDD[(String, Double)] =
  bills mapValues (_ * 1.2) // VAT 20%
// Array((“Luis”, 5.1), (“Javi”, 3.x), ...)

val amountsPerUser: RDD[(String, Iterable[Int])] =
  bills.groupByKey
// Array((“Luis”, [5, 2, ...]), (“Javi”, [3]), ...)

val fullAmountPerUser: RDD[(String, Int)] =
  bills reduceByKey (_ + _)
// Array((“Luis”, 32.14), (“Javi”, 10.58), ...)
```

Transformaciones PairRDDs II

```
// PairRDDFunctions

val addresses: RDD[(String, String)] =
  sc.parallelize(("Luis", "Avd Alamo 5") :: … )

val usersWithNoAddress: RDD[(String, Int)] =
  bills subtractByKey addresses

val usersBillsAndAddress:
  RDD[(String,(Iterable[Int], Iterable[String]))] =
  bills cogroup addresses

val billsWithAddress: RDD[(String,(Int,String))] =
  bills join addresses
```

Acciones Pair RDDs

```scala
// PairRDDFunctions

val billsPerUser: scala.collection.Map[String, Long] =
  bills countByKey()

val joseBillsValues: Seq[Int] =
  bills lookup ("Jose")
```

Spark y la programación funcional

- Sección 0 - Introducción a Apache Spark
- Sección 1 - Core RDDs
- Sección 2 - Double RDDs
  - Ejercicio 1 - Regresión lineal
- Sección 3 - Pair RDDs
- **Sección 4 - Ordered RDDs**
  - Ejercicio 2 - Quijote
  - Ejercicio 3 - Logs
- Sección 5 - Lenguaje de procesado de datos
- Sección 6 - Conclusión del curso

OrderedRDDFunctions

```
OrderedRDDFunctions[K,V]
```

Transformaciones

```
filterByRange(lower: K, upper: K): RDD[P]
sortByKey(ascending: Boolean = true): RDD[(K, V)]
```

Transformaciones OrderedRDDs

```
// OrderedRDDFunctions

val users: RDD[(Int,String)] =
  sc.parallelize(List((3,"Ana"), (5,"Pepe"), (1,"Rosa"),
(2,"Javier"), (4,"Maria")))

val users1To3: RDD[(Int,String)] =
  users.filterByRange(2,3)
// Array((2, “Javier”), (3, “Ana”))

val usersSorted: RDD[(Int,String)] =
  users.sortByKey()
// Array((1, “Rosa”), (2, “Javier”), (3, “Ana”),
// (4, “Maria”), (5, “Pepe”))
```

Spark y la programación funcional

- Sección 0 - Introducción a Apache Spark
- Sección 1 - Core RDDs
- Sección 2 - Double RDDs
  - Ejercicio 1 - Regresión lineal
- Sección 3 - Pair RDDs
- Sección 4 - Ordered RDDs
  - **Ejercicio 2 - Quijote**
  - Ejercicio 3 - Logs
- Sección 5 - Lenguaje de procesado de datos
- Sección 6 - Conclusión del curso

Ejercicio2: Quijote

- Se proporciona un extracto de *El Quijote*
  - `“tema4-spark/data/quijote.txt”`

- Se pide analizar el texto y responder a las siguientes preguntas:
  - *¿Cuáles son las 10 palabras más usadas?*
  - *¿Cuáles son las 10 palabras más usadas que tienen más de 3 letras?*

Spark y la programación funcional

- Sección 0 - Introducción a Apache Spark
- Sección 1 - Core RDDs
- Sección 2 - Double RDDs
  - Ejercicio 1 - Regresión lineal
- Sección 3 - Pair RDDs
- Sección 4 - Ordered RDDs
  - Ejercicio 2 - Quijote
  - **Ejercicio 3 - Logs**
- Sección 5 - Lenguaje de procesado de datos
- Sección 6 - Conclusión del curso

Ejercicio3: Procesar logs

- Tenemos unas máquinas y sus logs:

Logs (ejemplo)		
MachineId	Error Type	Msg
1	ERROR	“aaa”
1	ERROR	“bbb”
2	ERROR	“ccc”
2	INFO	“ddd”

Listado máquinas (ejemplo)		
MachineId	IP	# Cores
1	0.0.0.1	1
2	0.0.0.2	2

  - Muestra la máquina con más msgs de error
  - Muestra el # de errores por # de cores. Para el ejemplo anterior tu código debería mostrar algo como lo siguiente:

```
MachineInfo(1,/0.0.0.1,1)
Map(1 -> 2, 2 -> 1)
```

Spark y la programación funcional

- Sección 0 - Introducción a Apache Spark
- Sección 1 - Core RDDs
- Sección 2 - Double RDDs
  - Ejercicio 1 - Regresión lineal
- Sección 3 - Pair RDDs
- Sección 4 - Ordered RDDs
  - Ejercicio 2 - Quijote
  - Ejercicio 3 - Logs
- **Sección 5 - Lenguaje de procesado de datos**
- Sección 6 - Conclusión del curso

Lenguaje de transformaciones

- Los RDDs forman un lenguaje de transformación de datos
- Podemos crearnos una clase de lenguajes para abstraernos de la representación
  - RDDs
  - Lists
  - ...

Quijote (RDD)

```scala
val quijoteText: RDD[String] = ???
val quijote: RDD[(Int, String)] = quijoteText
  .flatMap(_.split(" "))
  .filter(!_.isEmpty)
  .map((_, 1))
  .reduceByKey(_ + _)
  .map(_.swap)
  .sortByKey(false)

val res: List[(Int, String)] = quijote take 10
```

Quijote (List)

```
val quijoteText: List[String] = ???
val quijote: List[(Int, String)] = quijoteText
  .flatMap(_.split(" "))
  .filter(!_.isEmpty)
  .map((_, 1))
  .groupBy(_._1)
  .mapValues(_.map(_._2).reduce(f))
  .toList
  .map(_.swap)
  .sortBy(_._1)
  .reverse

val res: List[(Int, String)] = quijote take 10
```

Lenguaje

```scala
trait TransformationLanguage[T[_, _]] {
  def expand[A, B](f: A => TraversableOnce[B]): T[A, B]
  def filter[A](f: A => Boolean): T[A, A]
  def andThen[A, B, C](t1: T[A, B])(t2: T[B, C]): T[A, C]
  def apply[A, B](f: A => B): T[A, B]
  def reduceByKey[K, V](f: (V, V) => V): T[(K, V), (K, V)]
  def sortBy[A, B](
    f: A => B,
    asc: Boolean = true)(implicit O: Ordering[B]): T[A, A]

  // Derivadas
  def sortByKey[K, V](asc: Boolean = true)
      (implicit O: Ordering[K]): T[(K, V), (K, V)] =
    sortBy(_._1, asc)
}
```

Programa genérico

```
def wordCount[T[_, _]](implicit T: TLang[T])
    : T[String, (Int, String)] =
  T.expand(_.split(" ")) andThen
  T.filter(!_.isEmpty)   andThen
  T((_, 1))              andThen
  T.reduceByKey(_ + _)   andThen
  T(_.swap)              andThen
  T.sortByKey(false)
```

Conclusiones Spark

- Framework para análisis distribuido de grandes cantidades de datos
  - Basado en el concepto de RDDs
  - Muy eficiente gracias a la posibilidad de ‘cachear’ resultados (RDDs) intermedios
- Aplica conceptos fundamentales de la PF:
  - Las transformaciones de RDDs forman un lenguaje
  - Las acciones sobre RDDs disparan la interpretación de ese lenguaje
  - El lenguaje utiliza funciones de órden superior
  - Las type classes nos permiten abstraer el lenguaje de transformaciones de Spark

Ejercicios para casa

- **`Ejercicio 1`**
  `tema4-spark/homework/EjercicioHomework_Spark.scala`

  Se analizarán visitas a páginas web. El input serán dos ficheros en formato CSV. El primero, **`data/PagesKeywords.csv`** contiene para cada página una serie de palabras clave. El segundo, **`data/PagesVisits.csv`**, contiene las visitas que cada usuario ha hecho a cada página. Estas visitas no están ordenadas y el mismo usuario puede visitar la misma página varias veces. El ejercicio consta de las siguientes partes:

  - Leer los ficheros con la información como **RDD**s
  - Crear un **RDD** que a cada usuario le asocie un **Map** que diga, para cada *keyword*, la cantidad de veces que el usuario ha visitado alguna página con esa *keyword*
  - A partir del **RDD** anterior decir para cada usuario qué tres *keywords* le interesan más

Spark y la programación funcional

- Sección 0 - Introducción a Apache Spark
- Sección 1 - Core RDDs
- Sección 2 - Double RDDs
  - Ejercicio 1 - Regresión lineal
- Sección 3 - Pair RDDs
- Sección 4 - Ordered RDDs
  - Ejercicio 2 - Quijote
  - Ejercicio 3 - Logs
- Sección 5 - Lenguaje de procesado de datos
- **Sección 6 - Conclusión del curso**

Conclusiones: ¿Por qué PF?

- La PF es simplemente un (paradigma/conjunto de técnicas) que nos permiten escribir código con garantías:
  - Re-usabilidad, mantenibilidad, testabilidad, ...
- Estas garantías se consiguen mediante técnicas de modularidad:
  - HOFs
  - Type classes
  - Lenguajes
  - ...

Conclusiones: ¿Por qué PF?

- Una arquitectura funcional está definida por la modularización que inducen los lenguajes y las funciones puras definidas sobre ellos
- Hemos visto las técnicas fundamentales de la PF, en el curso avanzado se ahonda más en estas técnicas:
  - Combinación de efectos, lenguajes aplicativos, ...
  - Y se hace uso del ecosistema de librerías funcionales de Scala (Scalaz, Shapeless, ...)

Conclusiones: ¿Por qué Scala?

- Soporte de técnicas de PF:
  - Genericidad *Higher-kind*
  - Implícitos
  - Lambdas
  - Sintaxis (*for-comprehension*, *context bound*, ...)
- Ecosistema
  - Compatible con Java
  - Spark, Shapeless, Scalaz, Play, Akka, ...