Procesamiento de datos masivos

Jesús Morán

viu | Universidad Internacional de Valencia

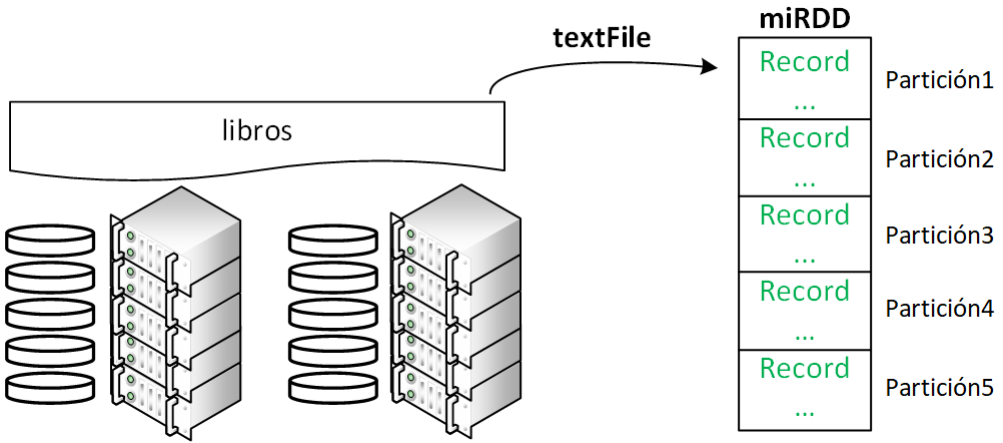
20.12.21

- Cualquier fuente de datos soportada por Hadoop
  - HDFS
  - S3
  - Sequence files
  - Bases de datos relacionales
  - Bases de datos NoSQL
  - …

- textFile
  - Crea un RDD de un archivo o carpeta
  - Cada registro es una línea
  - Puede leer de diferentes sistemas de archivos

```
miRDD = sc.textFile("file:///home/hadmin/libros")
```

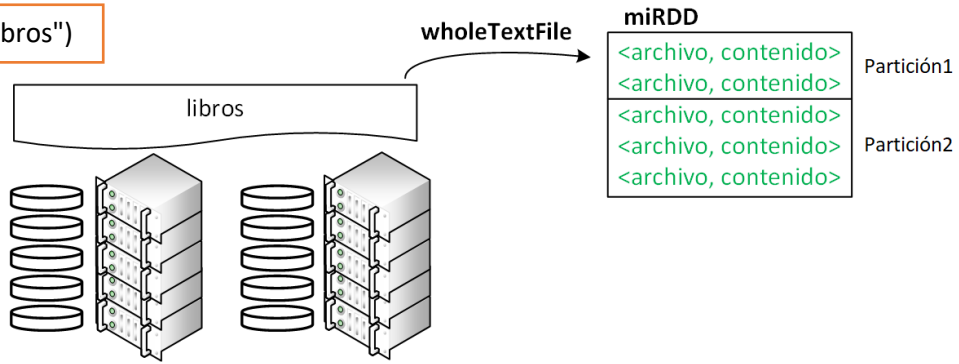
```
miRDD = sc.textFile("hdfs:///user/hadmin/libros")
```



- textFile
- wholeTextFile
  - Crea un RDD con pares <archivo, contenido del archivo>

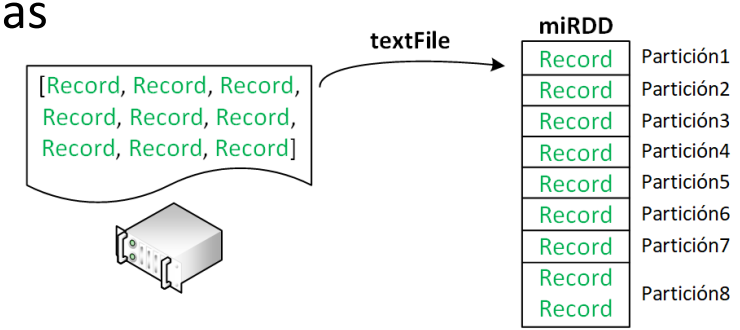
```
miRDD = sc.wholeTextFiles("file:///home/hadmin/libros")
```

```
miRDD = sc.wholeTextFiles("hdfs:///home/hadmin/libros")
```



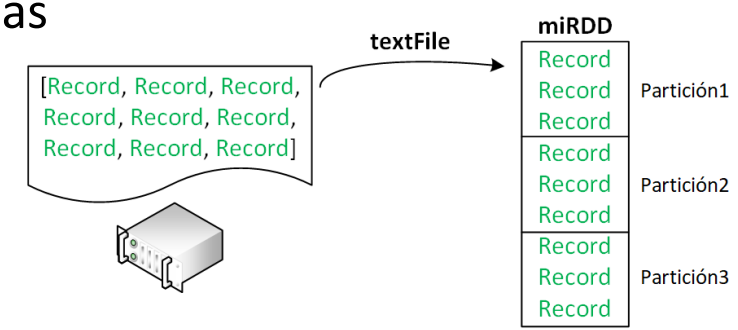
- textFile
- wholeTextFile
- parallelize
  - Crea un RDD a partir de una distribución de datos
  - Se puede utilizar para realizar pruebas

```
miRDD = sc.parallelize(["uno", "dos", "tres",
                        "cuatro", "cinco", "seis",
                        "siete", "ocho", "nueve"])
```



- textFile
- wholeTextFile
- parallelize
  - Crea un RDD a partir de una distribución de datos
  - Se puede utilizar para realizar pruebas
  - Se puede indicar num particiones

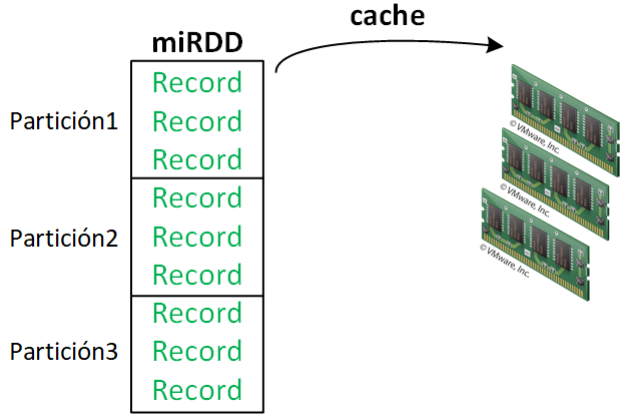
```
miRDD = sc.parallelize(["uno", "dos", "tres",
                        "cuatro", "cinco", "seis",
                        "siete", "ocho", "nueve"],3)
```



- textFile
- wholeTextFile
- parallelize
- newAPIHadoopRDD
  - Crea un RDD a partir de un InputFormat de Hadoop

- cache
  - Cachea el RDD en memoria
  - Las partes que no quepan en memoria se recalculan

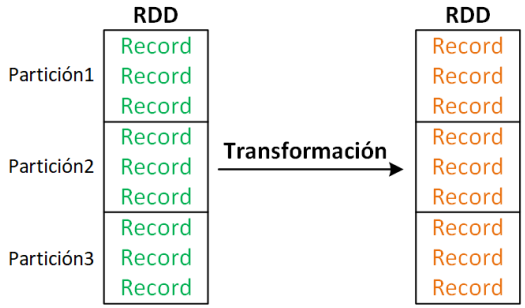
```
miRDD = sc.parallelize(["uno", "dos", "tres",
                        "cuatro", "cinco", "seis",
                        "siete", "ocho", "nueve"], 3)
miRDD.cache()
```



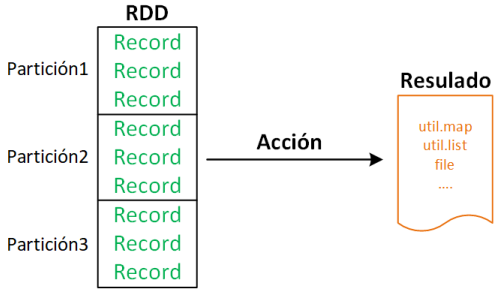
- cache
- persist
  - Similar a cache
  - Permite más niveles:
    - memoria
    - disco
    - Con replicaciones
    - memoria + disco
    - off-heap
    - …

- cache
- persist
- unpersist
  - Descacheamos manualmente la información
  - Spark puede descachear con política Least Recently Used

- Transformaciones: “generan” nuevos RDD



- Acciones: generan resultados



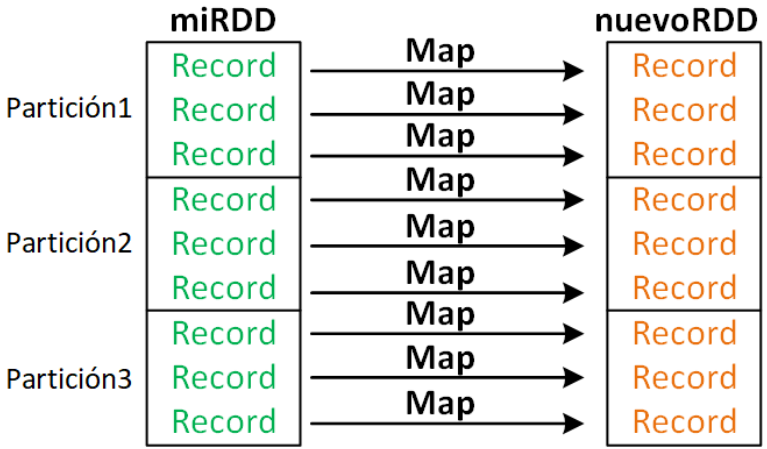
RDD transformaciones

- Transformaciones sencillas:
  - map
  - flatMap
  - mapPartitions
  - filter
  - sample
  - union
  - keyBy
  - sortByKey
  - pipe
- Transformaciones complejas:
  - intersection
  - distinct
  - reduceByKey
  - groupByKey
  - aggregateByKey
  - join
  - cartesian
  - repartition
  - coalesce
  - cogroup

- map
  - Por cada registro siempre emite un registro
  - Es diferente al Map de MapReduce

map(*func*) Return a new distributed dataset formed by passing each element of the source through a function *func*.

```
def miFuncion(record):
            return len(record)

miRDD = sc.parallelize(["uno", "dos", "tres",
                        "cuatro", "cinco", "seis",
                        "siete", "ocho", "nueve"],3)

nuevoRDD = miRDD.map(miFuncion)
```

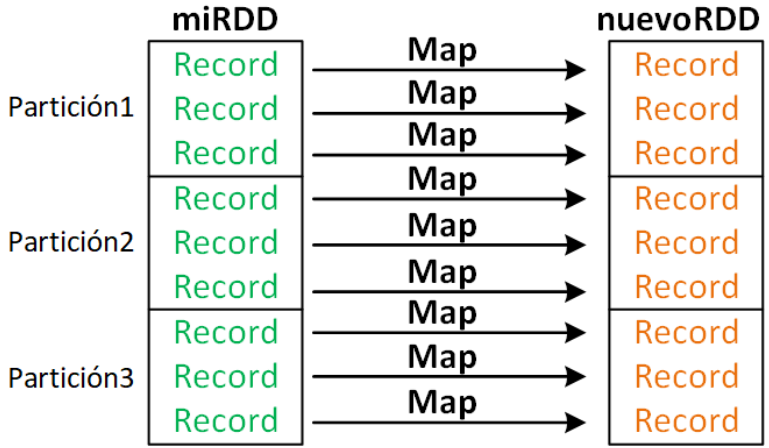



```
miRDD: ['uno', 'dos', 'tres', 'cuatro', 'cinco', 'seis', 'siete', 'ocho', 'nueve']
nuevoRDD: [3, 3, 4, 6, 5, 4, 5, 4, 5]
```

- map
  - Por cada registro siempre emite un registro
  - Es diferente al Map de MapReduce

map(*func*) Return a new distributed dataset formed by passing each element of the source through a function *func*.

```
miRDD = sc.parallelize(["uno", "dos", "tres",
                        "cuatro", "cinco", "seis",
                        "siete", "ocho", "nueve"],3)

nuevoRDD = miRDD.map(lambda record: len(record))
```

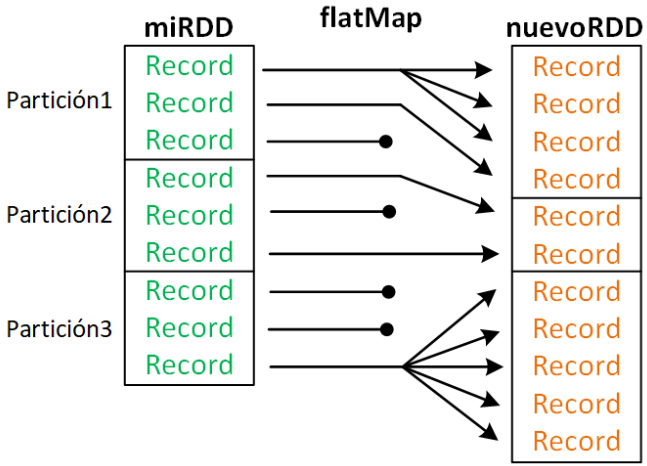



```
miRDD: ['uno', 'dos', 'tres', 'cuatro', 'cinco', 'seis', 'siete', 'ocho', 'nueve']
nuevoRDD: [3, 3, 4, 6, 5, 4, 5, 4, 5]
```

flatMap

- Por cada registro puede emitir 0, 1 o varios registros
- Es parecida al Map de MapReduce

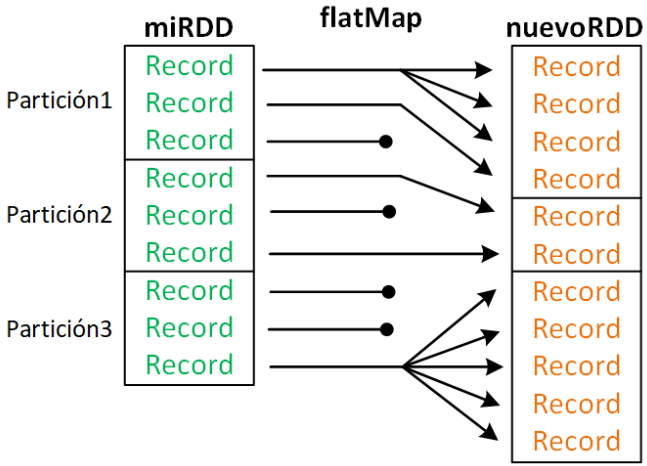
```
def miFuncion(record):
    if record != "-1":
        words = record.split(" ")
            return words
     else:
            return []

miRDD = sc.parallelize(["son tres palabras", "cuatro", "-1",
                        "uno", "-1", "otra",
                        "-1", "-1", "tercer particion con cinco palabras"], 3)
nuevoRDD = miRDD.flatMap(miFuncion)
```




```
miRDD: ['son tres palabras', 'cuatro', '-1', 'uno', '-1', 'otra', '-1', '-1', 'tercer particion con cinco palabras']
nuevoRDD: ['son', 'tres', 'palabras', 'cuatro', 'uno', 'otra', 'tercer', 'particion', 'con', 'cinco', 'palabras']
```

- flatMap
  - Por cada registro puede emitir 0, 1 o varios registros
  - Es parecida al Map de MapReduce

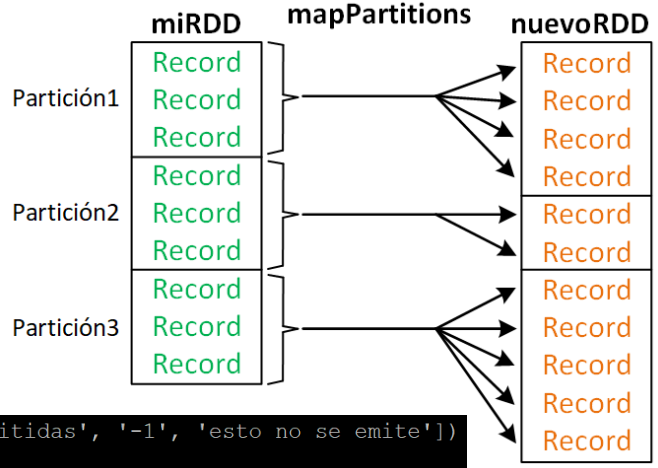
```
miRDD = sc.parallelize(["son tres palabras", "cuatro", "-1",
                        "uno", "-1", "otra",
                        "-1", "-1", "tercer particion con cinco palabras"], 3)
nuevoRDD = miRDD.flatMap(lambda record: [] if record == "-1" else record.split(" "))
```




```
miRDD: ['son tres palabras', 'cuatro', '-1', 'uno', '-1', 'otra', '-1', '-1', 'tercer particion con cinco palabras']
nuevoRDD: ['son', 'tres', 'palabras', 'cuatro', 'uno', 'otra', 'tercer', 'particion', 'con', 'cinco', 'palabras']
```

- mapPartitions
  - Por cada partición puede emitir 0, 1 o n registros
  - Es parecida a Mapper de MapReduce

```
def miFuncion(particion):
    emision = []
    for record in particion:
        if record == "-1":
            return emision
        else:
            emision.append(record)
    return emision

miRDD = sc.parallelize(["las cuatro palabras emitidas","-1", "esto no se emite",
                        "se", "emite", "-1",
                        "lo ultimo que se emite", "-1", "esto tampoco se emite"], 3)

nuevoRDD = miRDD.mapPartitions(miFuncion)
```

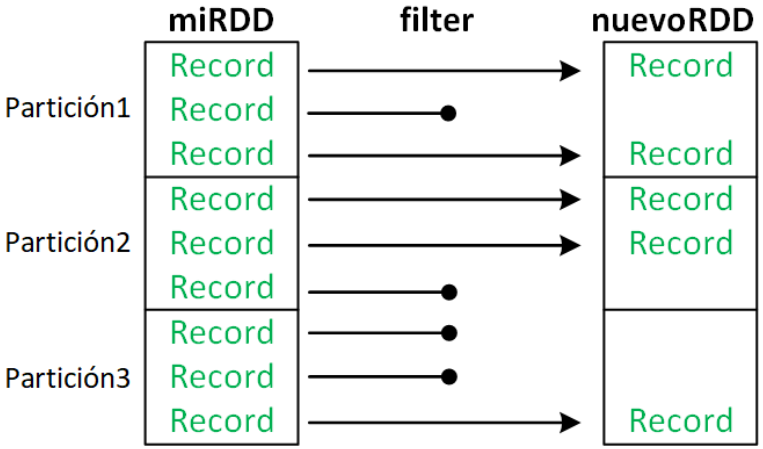



```
miRDD:
('Particion: ', ['las cuatro palabras emitidas', '-1', 'esto no se emite'])
('Particion: ', ['se', 'emite', '-1'])
('Particion: ', ['lo ultimo que se emite', '-1', 'esto tampoco se emite'])
```

```
nuevoRDD:
('Particion: ', ['las cuatro palabras emitidas'])
('Particion: ', ['se', 'emite'])
('Particion: ', ['lo ultimo que se emite'])
```

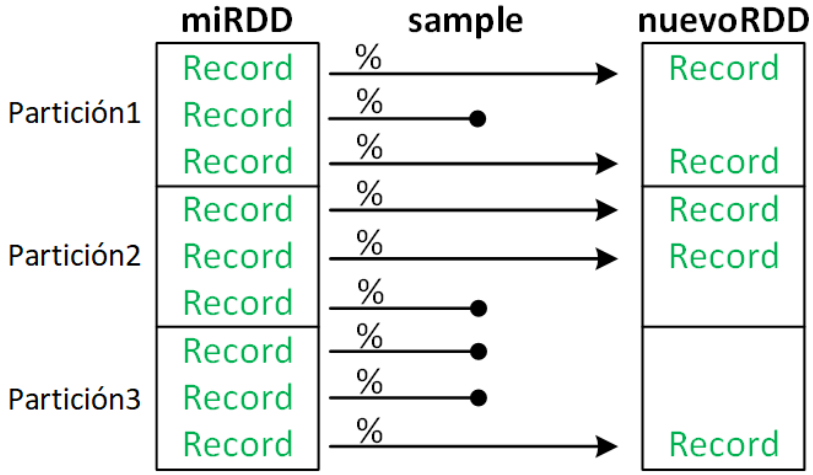
- filter
  - Por cada registro o lo emite o no

```
def miFuncion(record):
    if record != "-1":
       return True
    else:
       return False

miRDD = sc.parallelize(["1","-1", "3",
                        "4", "5", "-1",
                        "-1", "-1", "9"], 3)

nuevoRDD = miRDD.filter(miFuncion)
```




```
miRDD: ['1', '-1', '3', '4', '5', '-1', '-1', '-1', '9']
nuevoRDD: ['1', '3', '4', '5', '9']
```

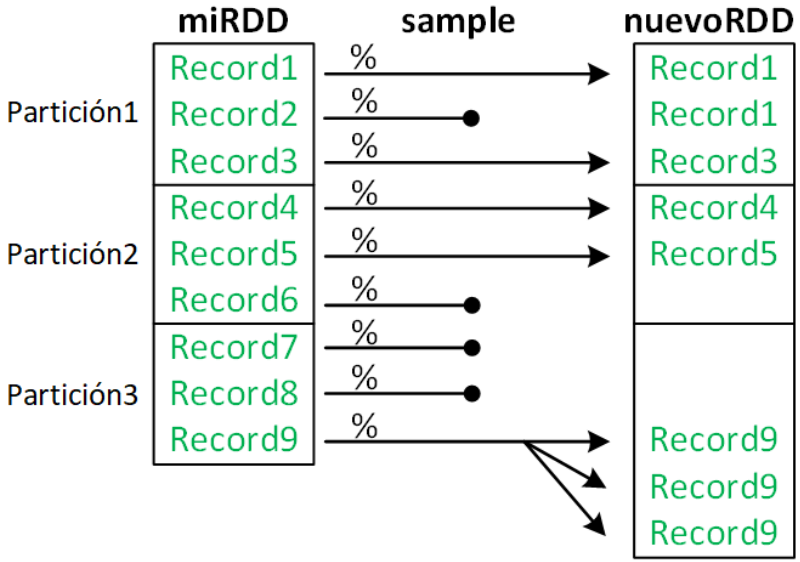
- sample
  - Hace un muestreo de los datos
  - **Sin**/con reemplazamiento




```
miRDD = sc.parallelize(["1","2", "3",
                        "4", "5", "6",
                        "7", "8", "9"], 3)

nuevoRDD = miRDD.sample(False, 0.6)
```

```
miRDD: ['1', '2', '3', '4', '5', '6', '7', '8', '9']
nuevoRDD: ['1', '3', '4', '5', '9']
```

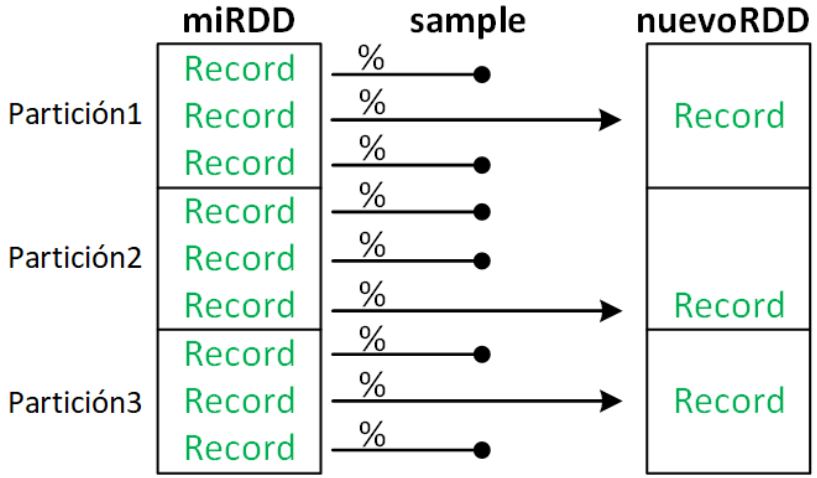
- sample
  - Hace un muestreo de los datos
  - Sin/**con** reemplazamiento




```
miRDD = sc.parallelize(["1","2", "3",
                        "4", "5", "6",
                        "7", "8", "9"], 3)

nuevoRDD = miRDD.sample(True, 0.6)
```

```
miRDD: ['1', '2', '3', '4', '5', '6', '7', '8', '9']
nuevoRDD: ['1', '1', '3', '4', '5', '9', '9', '9', '9']
```

- sample
  - Hace un muestreo de los datos
  - Sin/con reemplazamiento
  - **Seed**



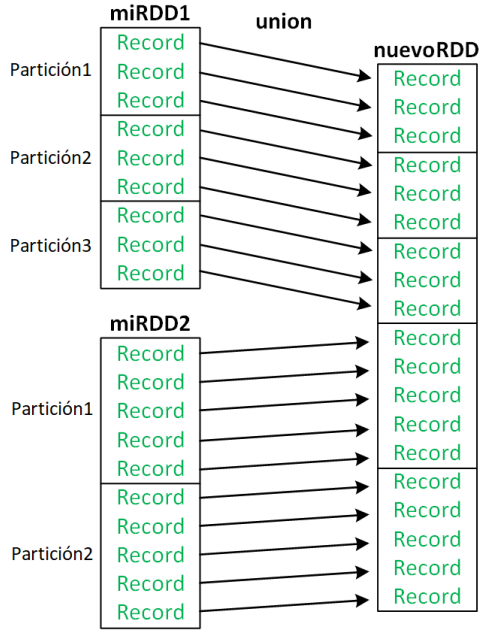

```
miRDD = sc.parallelize(["1","2", "3",
                        "4", "5", "6",
                        "7", "8", "9"], 3)

nuevoRDD = miRDD.sample(False, 0.6, 1)
```

```
miRDD: ['1', '2', '3', '4', '5', '6', '7', '8', '9']
nuevoRDD: ['2', '5', '6', '8']
```

union

- Une dos RDD

```
miRDD1 = sc.parallelize(["1","2", "3",
 "4", "5", "6",
 "7", "8", "9"], 3)

miRDD2 = sc.parallelize(["10","11", "12", "13", "14",
                         "15", "16", "17", "18", "19"], 2)


nuevoRDD = miRDD1.union(miRDD2)
```

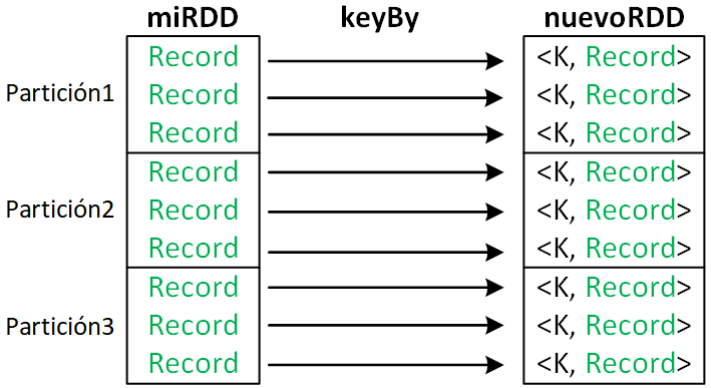



```
miRDD1: ['1', '2', '3', '4', '5', '6', '7', '8', '9']
miRDD2: ['10', '11', '12', '13', '14', '15', '16', '17', '18', '19']
nuevoRDD: ['1', '2', '3', '4', '5', '6', '7', '8', '9', '10', '11', '12', '13', '14
', '15', '16', '17', '18', '19']
```

keyBy

- Obtiene la clave de cada registro

```
def miFuncion(linea):
                anyo, temp = linea.split(";", 1)
                return anyo


miRDD = sc.parallelize([("1999;7"),("1999;5"), ("2000;10"),
                        ("1999;4"), ("2000;3"), ("2000;7"),
                        ("2000;10"), ("2001;3"), ("1999;5")], 3)

nuevoRDD = miRDD.keyBy(miFuncion)
```




```
miRDD: ['1999;7', '1999;5', '2000;10', '1999;4', '2000;3', '2000;7', '2000;10', '2001;
3', '1999;5']
nuevoRDD: [('1999', '1999;7'), ('1999', '1999;5'), ('2000', '2000;10'), ('1999', '1999
;4'), ('2000', '2000;3'), ('2000', '2000;7'), ('2000', '2000;10'), ('2001', '2001;3'),
 ('1999', '1999;5')]
```

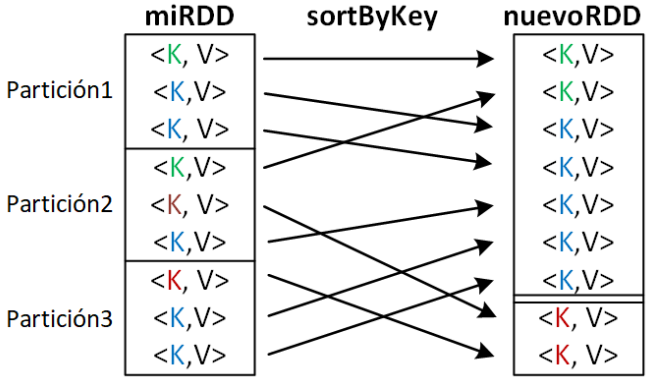
- sortByKey
  - Ordena un RDD según la clave
  - Orden ascendente/descendente
  - Se puede cambiar el número de particiones

sortByKey([*ascending*], [*numPartitions*])

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.

```
miRDD = sc.parallelize([("1", "v1_1"),("2", "v2_1"), ("2", "v2_2"),
                        ("1", "v1_2"), ("3", "v3_1"), ("2", "v2_3"),
                        ("3", "v3_2"), ("2", "v2_4"), ("2", "v2_5")], 3)

nuevoRDD = miRDD.sortByKey()
```

```
miRDD: [('1', 'v1_1'), ('2', 'v2_1'), ('2', 'v2_2'), ('1', 'v1_2'), ('3', 'v3_1'
), ('2', 'v2_3'), ('3', 'v3_2'), ('2', 'v2_4'), ('2', 'v2_5')]
nuevoRDD: [('1', 'v1_1'), ('1', 'v1_2'), ('2', 'v2_1'), ('2', 'v2_2'), ('2', 'v2
_3'), ('2', 'v2_4'), ('2', 'v2_5'), ('3', 'v3_1'), ('3', 'v3_2')]
```



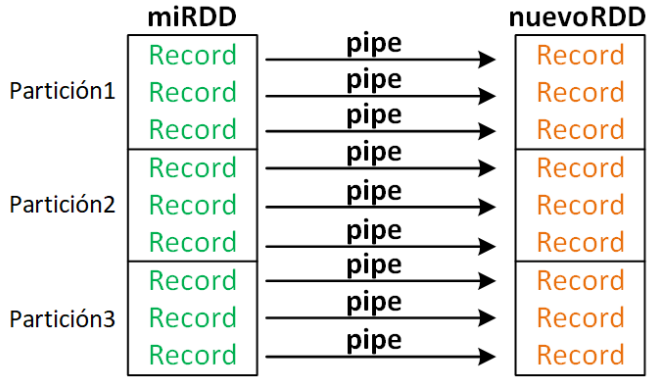
pipe

- Recibe un registro y permite ejecutar un comando externo

pipe(*command*, *[envVars]*)

Pipe each partition of the RDD through a shell command, e.g. a Perl or bash script. RDD elements are written to the process's stdin and lines output to its stdout are returned as an RDD of strings.

```
miRDD = sc.parallelize([("1999;7"),("1999;5"), ("2000;10"),
                        ("1999;4"), ("2000;3"), ("2000;7"),
                        ("2000;10"), ("2001;3"), ("1999;5")], 3)

nuevoRDD = miRDD.pipe("cut -d "";"" -f 1")
```

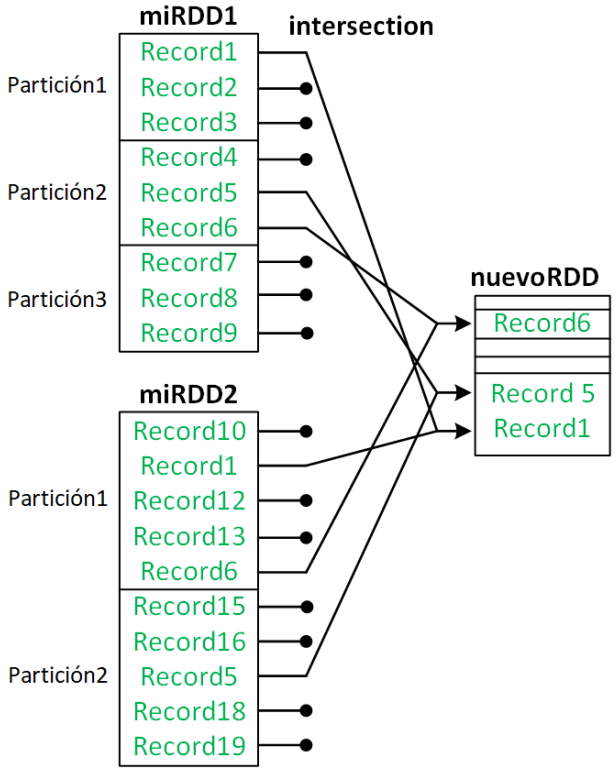



```
miRDD: ['1999;7', '1999;5', '2000;10', '1999;4', '2000;3', '2000;7', '2000;10', '2001;3', '1999;5']
nuevoRDD: [u'1999', u'1999', u'2000', u'1999', u'2000', u'2000', u'2000', u'2001', u'1999']
```

- intersection
  - Obtiene la intersección entre dos RDD

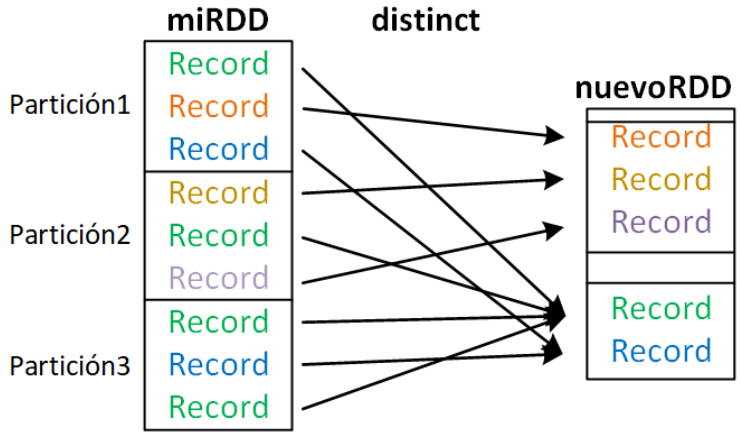
```
miRDD1 = sc.parallelize(["1","2", "3",
                         "4", "5", "6",
                         "7", "8", "9"], 3)

miRDD2 = sc.parallelize(["10","1", "12", "13", "6",
                         "15", "16", "5", "18", "19"], 2)


nuevoRDD = miRDD1.intersection(miRDD2)
```

```
miRDD1: ['1', '2', '3', '4', '5', '6', '7', '8', '9']
miRDD2: ['10', '1', '12', '13', '6', '15', '16', '5', '18', '19']
nuevoRDD: ['6', '1', '5']
```



- distinct
  - Elimina los duplicados de un RDD

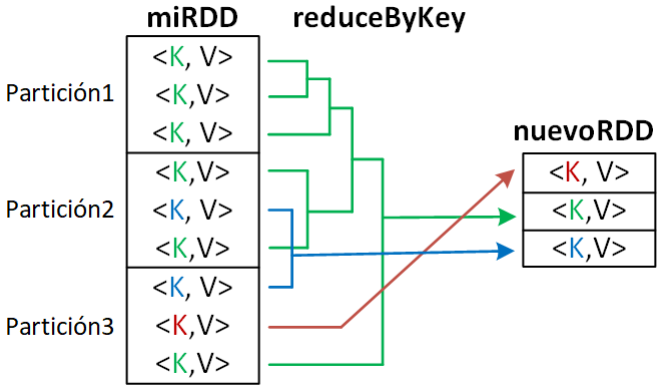
```
miRDD = sc.parallelize(["1","2", "3",
                        "4", "1", "6",
                        "1", "3", "1"], 3)

nuevoRDD = miRDD.distinct()
```




```
miRDD: ['1', '2', '3', '4', '1', '6', '1', '3', '1']
nuevoRDD: ['2', '4', '6', '1', '3']
```

- reduceByKey
  - Parecido al Reduce de Hadoop MapReduce
  - Compara un valor con el resultado de los anteriores (parecido a Combiner)

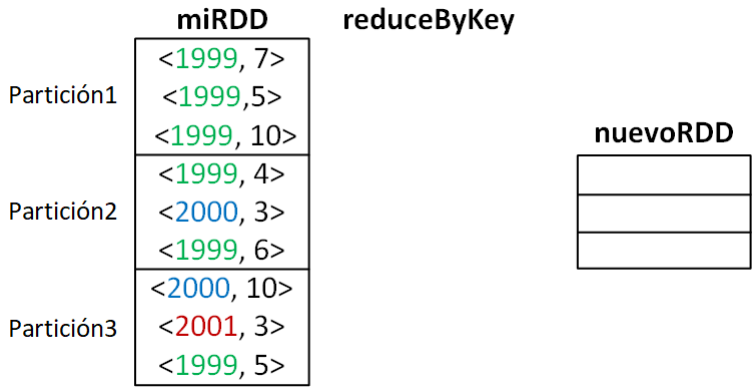
```
def miFuncion(accum, value):
    max = accum
    if value > accum:
       max = value
    return max

miRDD = sc.parallelize([(1999,7),(1999,5), (1999,10),
                        (1999,4), (2000,3), (1999,6),
                        (2000,10), (2001,3), (1999,5)], 3)

nuevoRDD = miRDD.reduceByKey(miFuncion)
```




```
miRDD: [(1999, 7), (1999, 5), (1999, 10), (1999, 4), (2000, 3),
(1999, 6), (2000, 10), (2001, 3), (1999, 5)]
nuevoRDD: [(2001, 3), (1999, 10), (2000, 10)]
```

- reduceByKey
  - Parecido al Reduce de Hadoop MapReduce
  - Compara un valor con el resultado de los anteriores (parecido a Combiner)

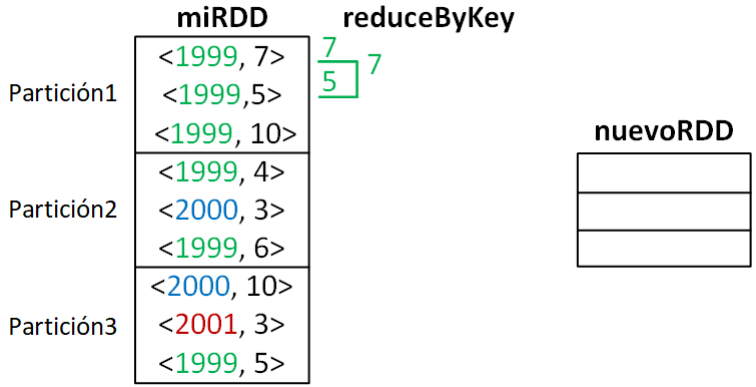
```
def miFuncion(accum, value):
    max = accum
    if value > accum:
       max = value
    return max

miRDD = sc.parallelize([(1999,7),(1999,5), (1999,10),
                        (1999,4), (2000,3), (1999,6),
                        (2000,10), (2001,3), (1999,5)], 3)

nuevoRDD = miRDD.reduceByKey(miFuncion)
```




```
miRDD: [(1999, 7), (1999, 5), (1999, 10), (1999, 4), (2000, 3),
(1999, 6), (2000, 10), (2001, 3), (1999, 5)]
nuevoRDD: [(2001, 3), (1999, 10), (2000, 10)]
```

- reduceByKey
  - Parecido al Reduce de Hadoop MapReduce
  - Compara un valor con el resultado de los anteriores (parecido a Combiner)

```
def miFuncion(accum, value):
    max = accum
    if value > accum:
       max = value
    return max

miRDD = sc.parallelize([(1999,7),(1999,5), (1999,10),
                        (1999,4), (2000,3), (1999,6),
                        (2000,10), (2001,3), (1999,5)], 3)

nuevoRDD = miRDD.reduceByKey(miFuncion)
```

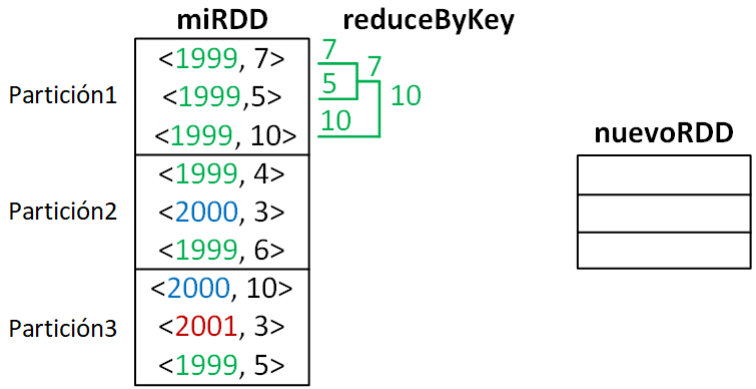



```
miRDD: [(1999, 7), (1999, 5), (1999, 10), (1999, 4), (2000, 3),
(1999, 6), (2000, 10), (2001, 3), (1999, 5)]
nuevoRDD: [(2001, 3), (1999, 10), (2000, 10)]
```

- reduceByKey
  - Parecido al Reduce de Hadoop MapReduce
  - Compara un valor con el resultado de los anteriores (parecido a Combiner)

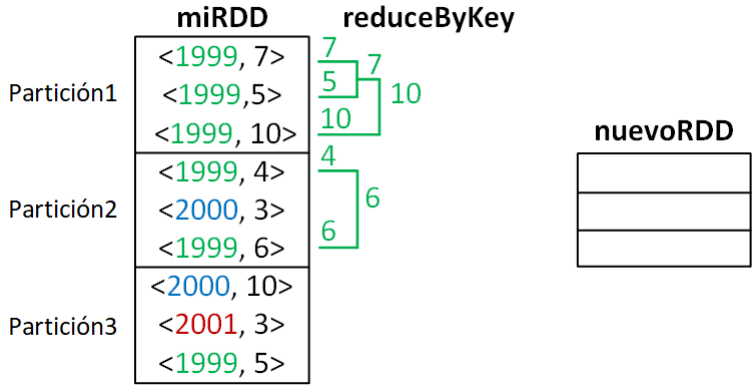
```
def miFuncion(accum, value):
    max = accum
    if value > accum:
       max = value
    return max

miRDD = sc.parallelize([(1999,7),(1999,5), (1999,10),
                        (1999,4), (2000,3), (1999,6),
                        (2000,10), (2001,3), (1999,5)], 3)

nuevoRDD = miRDD.reduceByKey(miFuncion)
```

```
miRDD: [(1999, 7), (1999, 5), (1999, 10), (1999, 4), (2000, 3),
(1999, 6), (2000, 10), (2001, 3), (1999, 5)]
nuevoRDD: [(2001, 3), (1999, 10), (2000, 10)]
```



- reduceByKey
  - Parecido al Reduce de Hadoop MapReduce
  - Compara un valor con el resultado de los anteriores (parecido a Combiner)

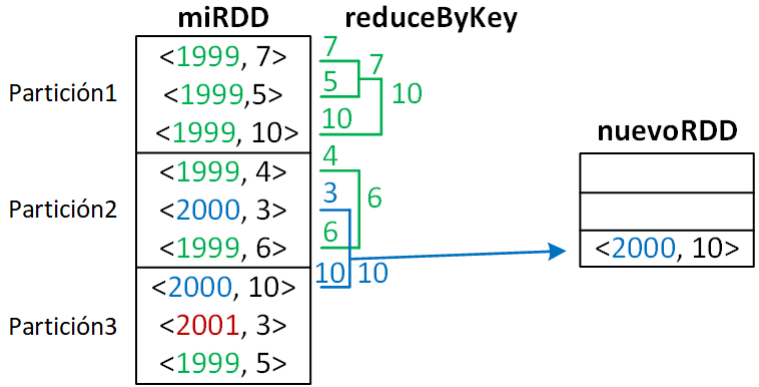
```
def miFuncion(accum, value):
    max = accum
    if value > accum:
       max = value
    return max

miRDD = sc.parallelize([(1999,7),(1999,5), (1999,10),
                        (1999,4), (2000,3), (1999,6),
                        (2000,10), (2001,3), (1999,5)], 3)

nuevoRDD = miRDD.reduceByKey(miFuncion)
```




```
miRDD: [(1999, 7), (1999, 5), (1999, 10), (1999, 4), (2000, 3),
(1999, 6), (2000, 10), (2001, 3), (1999, 5)]
nuevoRDD: [(2001, 3), (1999, 10), (2000, 10)]
```

- reduceByKey
  - Parecido al Reduce de Hadoop MapReduce
  - Compara un valor con el resultado de los anteriores (parecido a Combiner)

```
def miFuncion(accum, value):
    max = accum
    if value > accum:
       max = value
    return max

miRDD = sc.parallelize([(1999,7),(1999,5), (1999,10),
                        (1999,4), (2000,3), (1999,6),
                        (2000,10), (2001,3), (1999,5)], 3)

nuevoRDD = miRDD.reduceByKey(miFuncion)
```

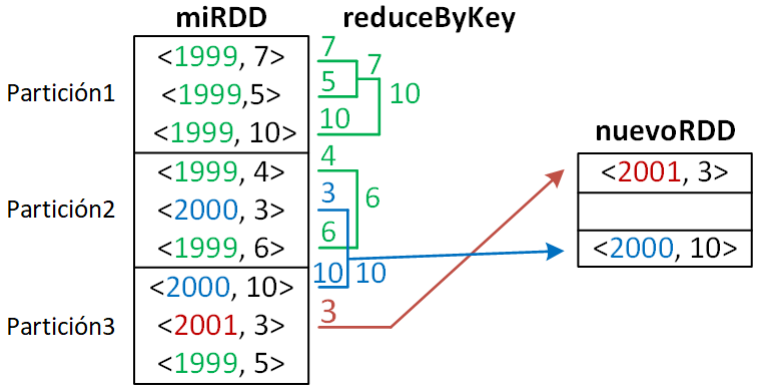



```
miRDD: [(1999, 7), (1999, 5), (1999, 10), (1999, 4), (2000, 3),
(1999, 6), (2000, 10), (2001, 3), (1999, 5)]
nuevoRDD: [(2001, 3), (1999, 10), (2000, 10)]
```

- reduceByKey
  - Parecido al Reduce de Hadoop MapReduce
  - Compara un valor con el resultado de los anteriores (parecido a Combiner)

```
def miFuncion(accum, value):
    max = accum
    if value > accum:
       max = value
    return max

miRDD = sc.parallelize([(1999,7),(1999,5), (1999,10),
                        (1999,4), (2000,3), (1999,6),
                        (2000,10), (2001,3), (1999,5)], 3)

nuevoRDD = miRDD.reduceByKey(miFuncion)
```

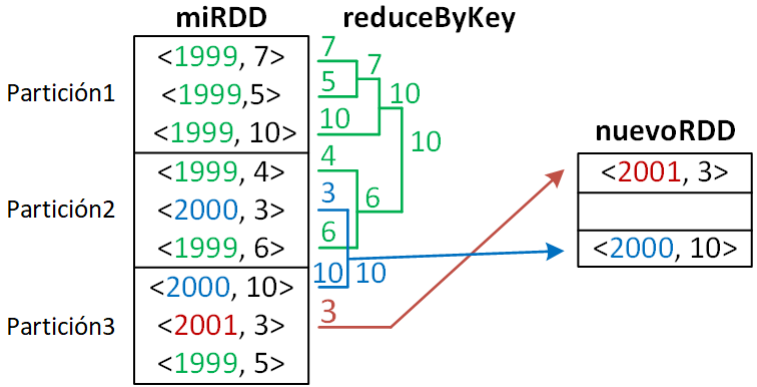
```
miRDD: [(1999, 7), (1999, 5), (1999, 10), (1999, 4), (2000, 3),
(1999, 6), (2000, 10), (2001, 3), (1999, 5)]
nuevoRDD: [(2001, 3), (1999, 10), (2000, 10)]
```



- reduceByKey
  - Parecido al Reduce de Hadoop MapReduce
  - Compara un valor con el resultado de los anteriores (parecido a Combiner)

```
def miFuncion(accum, value):
    max = accum
    if value > accum:
       max = value
    return max

miRDD = sc.parallelize([(1999,7),(1999,5), (1999,10),
                        (1999,4), (2000,3), (1999,6),
                        (2000,10), (2001,3), (1999,5)], 3)

nuevoRDD = miRDD.reduceByKey(miFuncion)
```

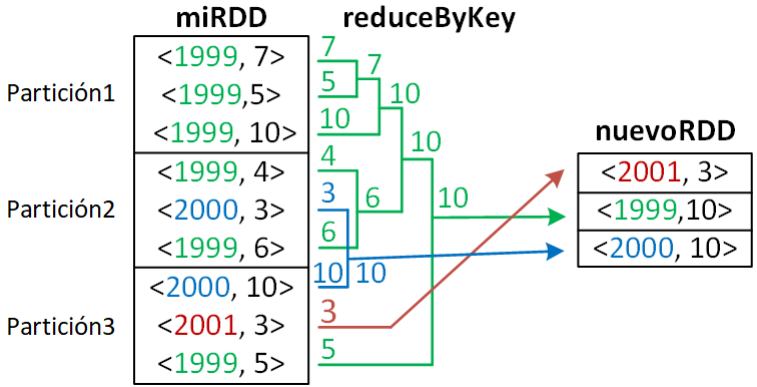
```
miRDD: [(1999, 7), (1999, 5), (1999, 10), (1999, 4), (2000, 3),
(1999, 6), (2000, 10), (2001, 3), (1999, 5)]
nuevoRDD: [(2001, 3), (1999, 10), (2000, 10)]
```



- reduceByKey
  - Parecido al Reduce de Hadoop MapReduce
  - Compara un valor con el resultado de los anteriores (parecido a Combiner)

```
def miFuncion(accum, value):
    max = accum
    if value > accum:
       max = value
    return max

miRDD = sc.parallelize([(1999,7),(1999,5), (1999,10),
                        (1999,4), (2000,3), (1999,6),
                        (2000,10), (2001,3), (1999,5)], 3)

nuevoRDD = miRDD.reduceByKey(miFuncion)
```

```
miRDD: [(1999, 7), (1999, 5), (1999, 10), (1999, 4), (2000, 3),
(1999, 6), (2000, 10), (2001, 3), (1999, 5)]
nuevoRDD: [(2001, 3), (1999, 10), (2000, 10)]
```



groupByKey

- Agrupa los valores por la clave

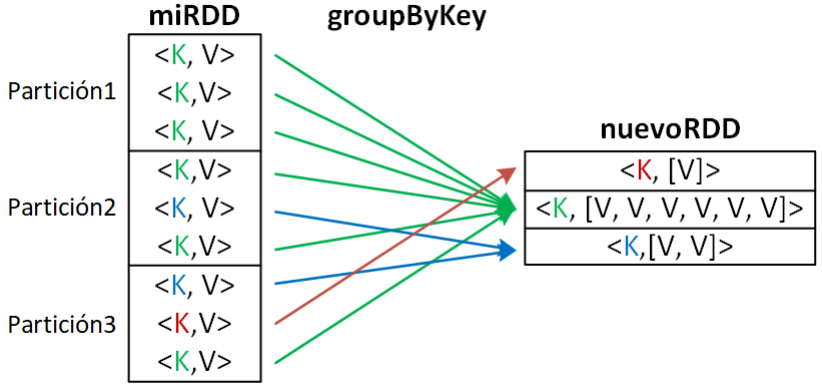
groupByKey([*numPartitions*])

When called on a dataset of (K, V) pairs, returns a dataset of (K, Iterable<V>) pairs.
Note: If you are grouping in order to perform an aggregation (such as a sum or average) over each key, using `reduceByKey` or `aggregateByKey` will yield much better performance.
Note: By default, the level of parallelism in the output depends on the number of partitions of the parent RDD. You can pass an optional `numPartitions` argument to set a different number of tasks.

```
miRDD = sc.parallelize([(1999,7),(1999,5), (1999,10),
                        (1999,4), (2000,3), (1999,6),
                        (2000,10), (2001,3), (1999,5)], 3)

nuevoRDD = miRDD.groupByKey()
```

```
miRDD: [(1999, 7), (1999, 5), (1999, 10), (1999, 4), (2000, 3),
(1999, 6), (2000, 10), (2001, 3), (1999, 5)]
```

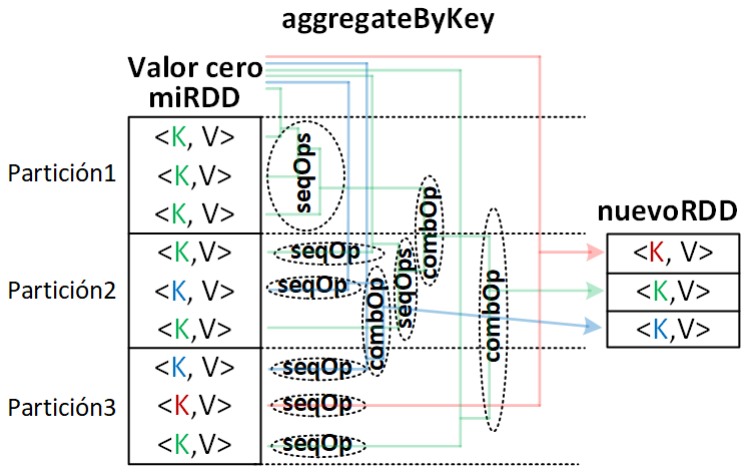
```
nuevoRDD: < 2001 , [3] >
          < 1999 , [7, 5, 10, 4, 6, 5] >
          < 2000 , [3, 10] >
```



- aggregateByKey (3 parámetros)
  - Un valor cero/inicial con el que se empezará hacer la agregación
  - seqOP: Una función parecida a Combine que se ejecuta con los datos locales de las particiones
  - combOp: Una función parecida a Reduce que se ejecuta con los datos resultantes de las particiones

```
def miSeqFuncion(accum, value):
                    max = accum
                    if value > accum:
                                        max = value
                    return max

def miCombFuncion(accum, value):
                    max = accum
                    if value > accum:
                                        max = value
                    return max

miRDD = sc.parallelize([(1999,7),(1999,5), (1999,10),
                        (1999,4), (2000,3), (1999,6),
                        (2000,10), (2001,3), (1999,5)], 3)

nuevoRDD = miRDD.aggregateByKey(- float("inf"), miSeqFuncion, miCombFuncion)
```

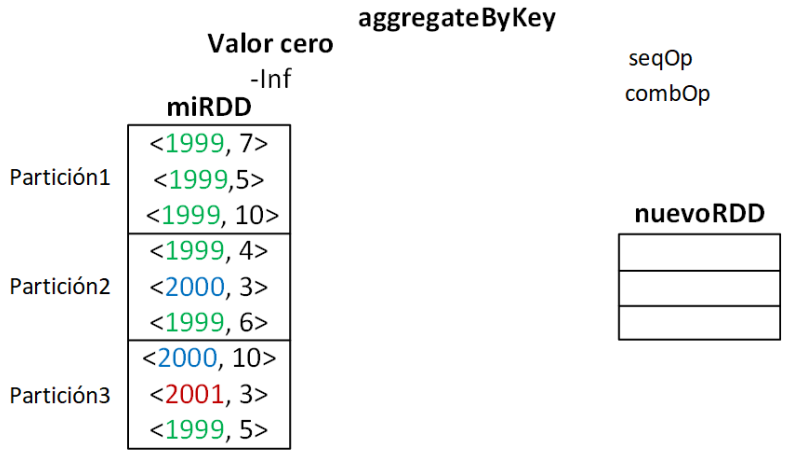
```
miRDD: [(1999, 7), (1999, 5), (1999, 10), (1999, 4), (2000,
3), (1999, 6), (2000, 10), (2001, 3), (1999, 5)]
nuevoRDD: [(2001, 3), (1999, 10), (2000, 10)]
```



- aggregateByKey (3 parámetros)
  - Un valor cero/inicial con el que se empezará hacer la agregación
  - seqOP: Una función parecida a Combine que se ejecuta con los datos locales de las particiones
  - combOp: Una función parecida a Reduce que se ejecuta con los datos resultantes de las particiones

```
def miSeqFuncion(accum, value):
                    max = accum
                    if value > accum:
                                        max = value
                    return max

def miCombFuncion(accum, value):
                    max = accum
                    if value > accum:
                                        max = value
                    return max

miRDD = sc.parallelize([(1999,7),(1999,5), (1999,10),
                        (1999,4), (2000,3), (1999,6),
                        (2000,10), (2001,3), (1999,5)], 3)

nuevoRDD = miRDD.aggregateByKey(- float("inf"), miSeqFuncion, miCombFuncion)
```

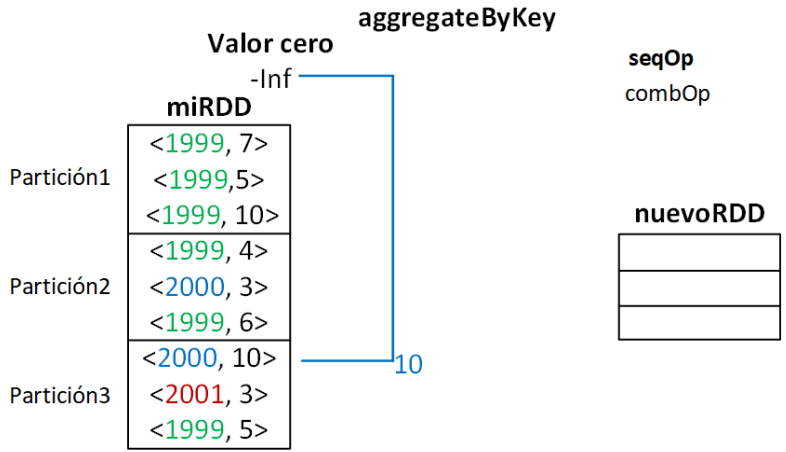
```
miRDD: [(1999, 7), (1999, 5), (1999, 10), (1999, 4), (2000,
3), (1999, 6), (2000, 10), (2001, 3), (1999, 5)]
nuevoRDD: [(2001, 3), (1999, 10), (2000, 10)]
```



- aggregateByKey (3 parámetros)
  - Un valor cero/inicial con el que se empezará hacer la agregación
  - seqOP: Una función parecida a Combine que se ejecuta con los datos locales de las particiones
  - combOp: Una función parecida a Reduce que se ejecuta con los datos resultantes de las particiones

```
def miSeqFuncion(accum, value):
                    max = accum
                    if value > accum:
                                        max = value
                    return max

def miCombFuncion(accum, value):
                    max = accum
                    if value > accum:
                                        max = value
                    return max

miRDD = sc.parallelize([(1999,7),(1999,5), (1999,10),
                        (1999,4), (2000,3), (1999,6),
                        (2000,10), (2001,3), (1999,5)], 3)

nuevoRDD = miRDD.aggregateByKey(- float("inf"), miSeqFuncion, miCombFuncion)
```

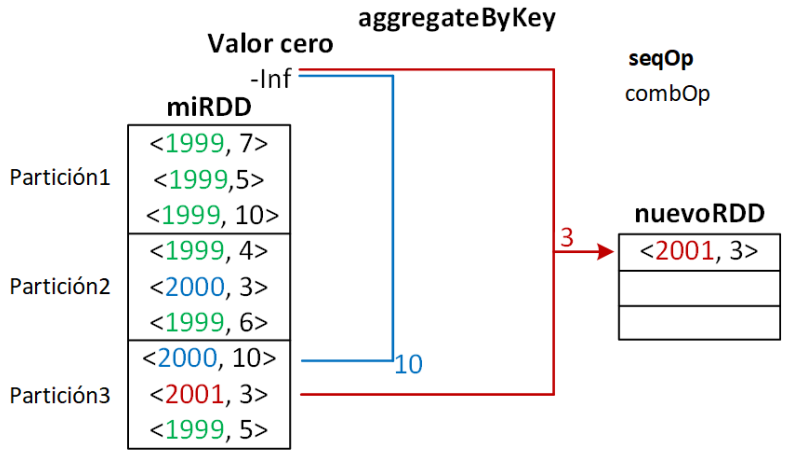
```
miRDD: [(1999, 7), (1999, 5), (1999, 10), (1999, 4), (2000,
3), (1999, 6), (2000, 10), (2001, 3), (1999, 5)]
nuevoRDD: [(2001, 3), (1999, 10), (2000, 10)]
```



- aggregateByKey (3 parámetros)
  - Un valor cero/inicial con el que se empezará hacer la agregación
  - seqOP: Una función parecida a Combine que se ejecuta con los datos locales de las particiones
  - combOp: Una función parecida a Reduce que se ejecuta con los datos resultantes de las particiones

```
def miSeqFuncion(accum, value):
                    max = accum
                    if value > accum:
                                        max = value
                    return max

def miCombFuncion(accum, value):
                    max = accum
                    if value > accum:
                                        max = value
                    return max

miRDD = sc.parallelize([(1999,7),(1999,5), (1999,10),
                        (1999,4), (2000,3), (1999,6),
                        (2000,10), (2001,3), (1999,5)], 3)

nuevoRDD = miRDD.aggregateByKey(- float("inf"), miSeqFuncion, miCombFuncion)
```

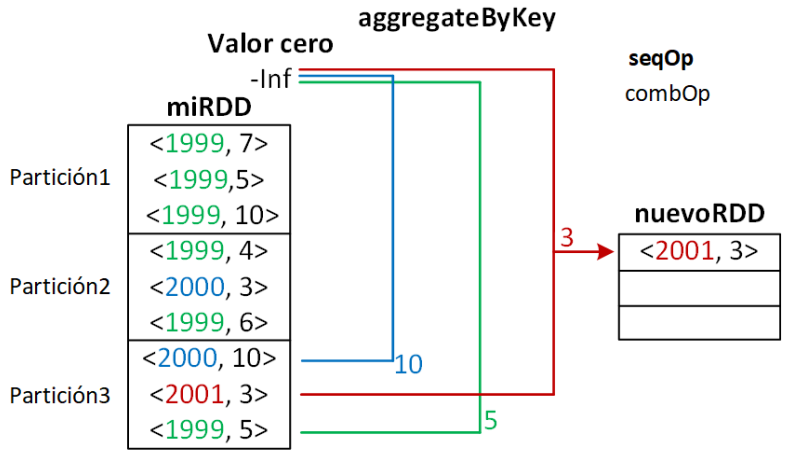
```
miRDD: [(1999, 7), (1999, 5), (1999, 10), (1999, 4), (2000,
3), (1999, 6), (2000, 10), (2001, 3), (1999, 5)]
nuevoRDD: [(2001, 3), (1999, 10), (2000, 10)]
```



- aggregateByKey (3 parámetros)
  - Un valor cero/inicial con el que se empezará hacer la agregación
  - seqOP: Una función parecida a Combine que se ejecuta con los datos locales de las particiones
  - combOp: Una función parecida a Reduce que se ejecuta con los datos resultantes de las particiones

```
def miSeqFuncion(accum, value):
                    max = accum
                    if value > accum:
                                        max = value
                    return max

def miCombFuncion(accum, value):
                    max = accum
                    if value > accum:
                                        max = value
                    return max

miRDD = sc.parallelize([(1999,7),(1999,5), (1999,10),
                        (1999,4), (2000,3), (1999,6),
                        (2000,10), (2001,3), (1999,5)], 3)

nuevoRDD = miRDD.aggregateByKey(- float("inf"), miSeqFuncion, miCombFuncion)
```

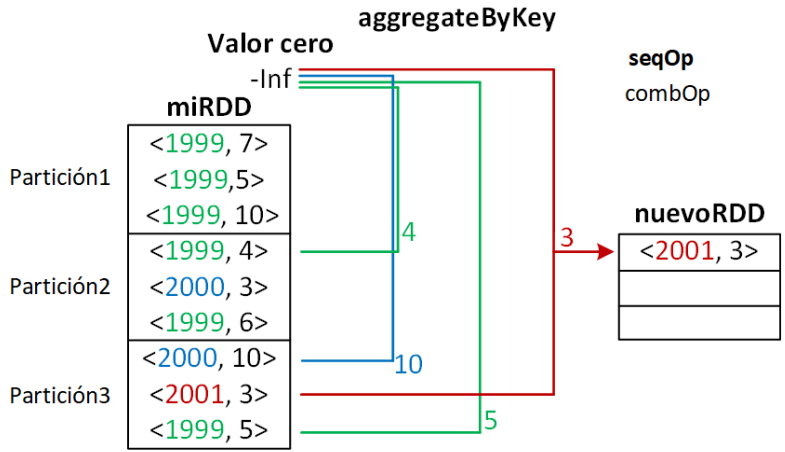
```
miRDD: [(1999, 7), (1999, 5), (1999, 10), (1999, 4), (2000,
3), (1999, 6), (2000, 10), (2001, 3), (1999, 5)]
nuevoRDD: [(2001, 3), (1999, 10), (2000, 10)]
```



- aggregateByKey (3 parámetros)
  - Un valor cero/inicial con el que se empezará hacer la agregación
  - seqOP: Una función parecida a Combine que se ejecuta con los datos locales de las particiones
  - combOp: Una función parecida a Reduce que se ejecuta con los datos resultantes de las particiones

```
def miSeqFuncion(accum, value):
                    max = accum
                    if value > accum:
                                        max = value
                    return max

def miCombFuncion(accum, value):
                    max = accum
                    if value > accum:
                                        max = value
                    return max

miRDD = sc.parallelize([(1999,7),(1999,5), (1999,10),
                        (1999,4), (2000,3), (1999,6),
                        (2000,10), (2001,3), (1999,5)], 3)

nuevoRDD = miRDD.aggregateByKey(- float("inf"), miSeqFuncion, miCombFuncion)
```

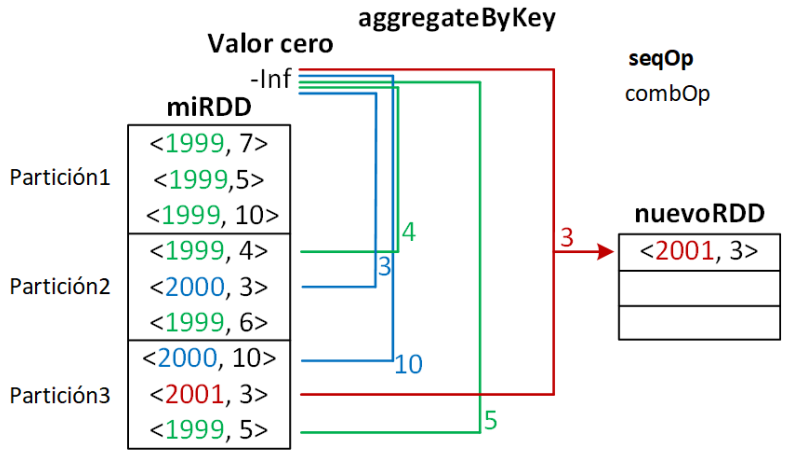
```
miRDD: [(1999, 7), (1999, 5), (1999, 10), (1999, 4), (2000,
3), (1999, 6), (2000, 10), (2001, 3), (1999, 5)]
nuevoRDD: [(2001, 3), (1999, 10), (2000, 10)]
```



- aggregateByKey (3 parámetros)
  - Un valor cero/inicial con el que se empezará hacer la agregación
  - seqOP: Una función parecida a Combine que se ejecuta con los datos locales de las particiones
  - combOp: Una función parecida a Reduce que se ejecuta con los datos resultantes de las particiones

```
def miSeqFuncion(accum, value):
                    max = accum
                    if value > accum:
                                        max = value
                    return max

def miCombFuncion(accum, value):
                    max = accum
                    if value > accum:
                                        max = value
                    return max

miRDD = sc.parallelize([(1999,7),(1999,5), (1999,10),
                        (1999,4), (2000,3), (1999,6),
                        (2000,10), (2001,3), (1999,5)], 3)

nuevoRDD = miRDD.aggregateByKey(- float("inf"), miSeqFuncion, miCombFuncion)
```

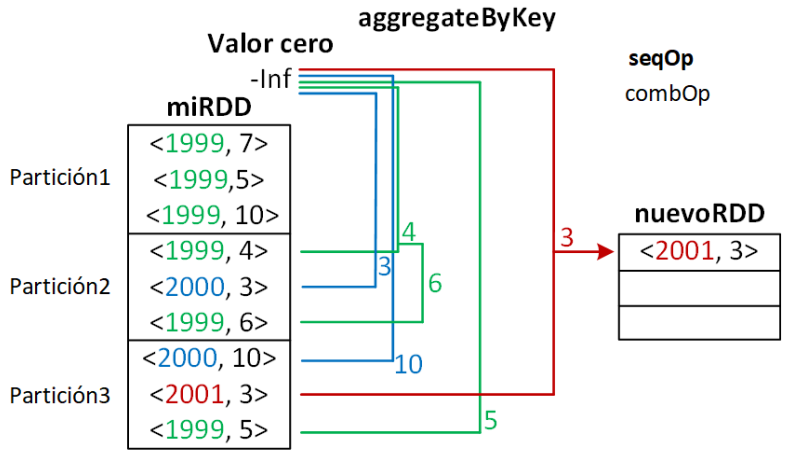
```
miRDD: [(1999, 7), (1999, 5), (1999, 10), (1999, 4), (2000,
3), (1999, 6), (2000, 10), (2001, 3), (1999, 5)]
nuevoRDD: [(2001, 3), (1999, 10), (2000, 10)]
```



- aggregateByKey (3 parámetros)
  - Un valor cero/inicial con el que se empezará hacer la agregación
  - seqOP: Una función parecida a Combine que se ejecuta con los datos locales de las particiones
  - combOp: Una función parecida a Reduce que se ejecuta con los datos resultantes de las particiones

```
def miSeqFuncion(accum, value):
                    max = accum
                    if value > accum:
                                        max = value
                    return max

def miCombFuncion(accum, value):
                    max = accum
                    if value > accum:
                                        max = value
                    return max

miRDD = sc.parallelize([(1999,7),(1999,5), (1999,10),
                        (1999,4), (2000,3), (1999,6),
                        (2000,10), (2001,3), (1999,5)], 3)

nuevoRDD = miRDD.aggregateByKey(- float("inf"), miSeqFuncion, miCombFuncion)
```

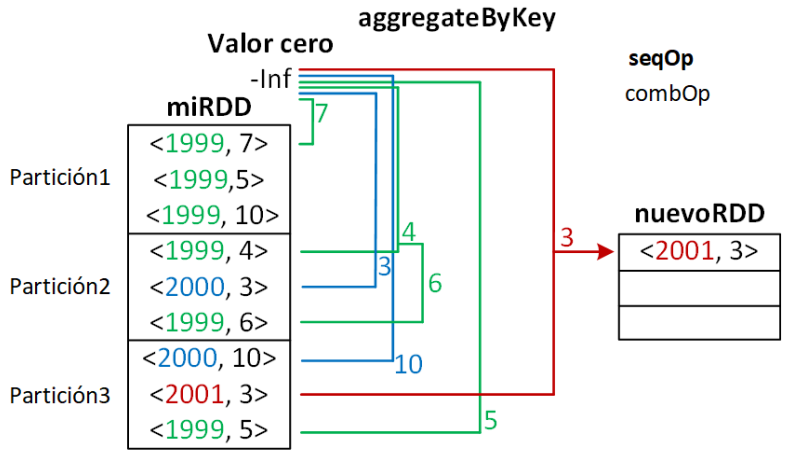
```
miRDD: [(1999, 7), (1999, 5), (1999, 10), (1999, 4), (2000,
3), (1999, 6), (2000, 10), (2001, 3), (1999, 5)]
nuevoRDD: [(2001, 3), (1999, 10), (2000, 10)]
```



- aggregateByKey (3 parámetros)
  - Un valor cero/inicial con el que se empezará hacer la agregación
  - seqOP: Una función parecida a Combine que se ejecuta con los datos locales de las particiones
  - combOp: Una función parecida a Reduce que se ejecuta con los datos resultantes de las particiones

```
def miSeqFuncion(accum, value):
                    max = accum
                    if value > accum:
                                        max = value
                    return max

def miCombFuncion(accum, value):
                    max = accum
                    if value > accum:
                                        max = value
                    return max

miRDD = sc.parallelize([(1999,7),(1999,5), (1999,10),
                        (1999,4), (2000,3), (1999,6),
                        (2000,10), (2001,3), (1999,5)], 3)

nuevoRDD = miRDD.aggregateByKey(- float("inf"), miSeqFuncion, miCombFuncion)
```

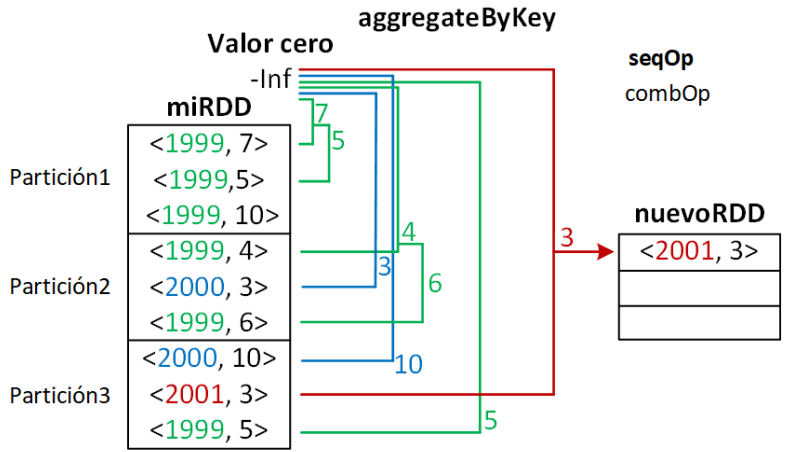
```
miRDD: [(1999, 7), (1999, 5), (1999, 10), (1999, 4), (2000,
3), (1999, 6), (2000, 10), (2001, 3), (1999, 5)]
nuevoRDD: [(2001, 3), (1999, 10), (2000, 10)]
```



- aggregateByKey (3 parámetros)
  - Un valor cero/inicial con el que se empezará hacer la agregación
  - seqOP: Una función parecida a Combine que se ejecuta con los datos locales de las particiones
  - combOp: Una función parecida a Reduce que se ejecuta con los datos resultantes de las particiones

```
def miSeqFuncion(accum, value):
                    max = accum
                    if value > accum:
                                        max = value
                    return max

def miCombFuncion(accum, value):
                    max = accum
                    if value > accum:
                                        max = value
                    return max

miRDD = sc.parallelize([(1999,7),(1999,5), (1999,10),
                        (1999,4), (2000,3), (1999,6),
                        (2000,10), (2001,3), (1999,5)], 3)

nuevoRDD = miRDD.aggregateByKey(- float("inf"), miSeqFuncion, miCombFuncion)
```

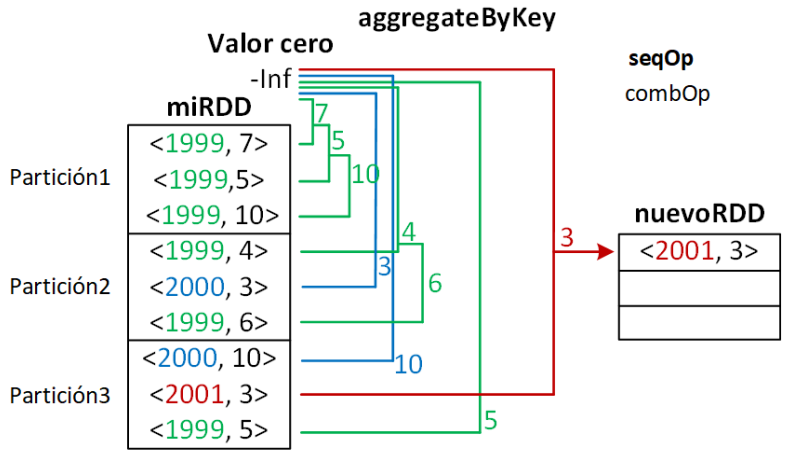
```
miRDD: [(1999, 7), (1999, 5), (1999, 10), (1999, 4), (2000,
3), (1999, 6), (2000, 10), (2001, 3), (1999, 5)]
nuevoRDD: [(2001, 3), (1999, 10), (2000, 10)]
```



- aggregateByKey (3 parámetros)
  - Un valor cero/inicial con el que se empezará hacer la agregación
  - seqOP: Una función parecida a Combine que se ejecuta con los datos locales de las particiones
  - combOp: Una función parecida a Reduce que se ejecuta con los datos resultantes de las particiones

```
def miSeqFuncion(accum, value):
                    max = accum
                    if value > accum:
                                        max = value
                    return max

def miCombFuncion(accum, value):
                    max = accum
                    if value > accum:
                                        max = value
                    return max

miRDD = sc.parallelize([(1999,7),(1999,5), (1999,10),
                        (1999,4), (2000,3), (1999,6),
                        (2000,10), (2001,3), (1999,5)], 3)

nuevoRDD = miRDD.aggregateByKey(- float("inf"), miSeqFuncion, miCombFuncion)
```

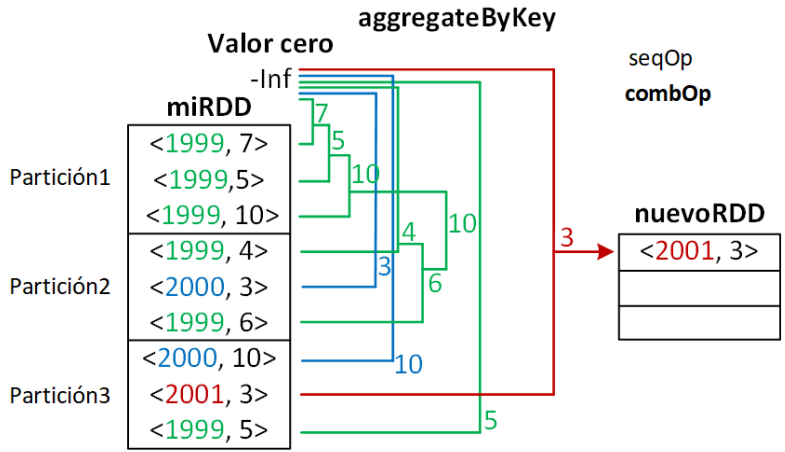
```
miRDD: [(1999, 7), (1999, 5), (1999, 10), (1999, 4), (2000,
3), (1999, 6), (2000, 10), (2001, 3), (1999, 5)]
nuevoRDD: [(2001, 3), (1999, 10), (2000, 10)]
```



- aggregateByKey (3 parámetros)
  - Un valor cero/inicial con el que se empezará hacer la agregación
  - seqOP: Una función parecida a Combine que se ejecuta con los datos locales de las particiones
  - combOp: Una función parecida a Reduce que se ejecuta con los datos resultantes de las particiones

```
def miSeqFuncion(accum, value):
                    max = accum
                    if value > accum:
                                        max = value
                    return max

def miCombFuncion(accum, value):
                    max = accum
                    if value > accum:
                                        max = value
                    return max

miRDD = sc.parallelize([(1999,7),(1999,5), (1999,10),
                        (1999,4), (2000,3), (1999,6),
                        (2000,10), (2001,3), (1999,5)], 3)

nuevoRDD = miRDD.aggregateByKey(- float("inf"), miSeqFuncion, miCombFuncion)
```

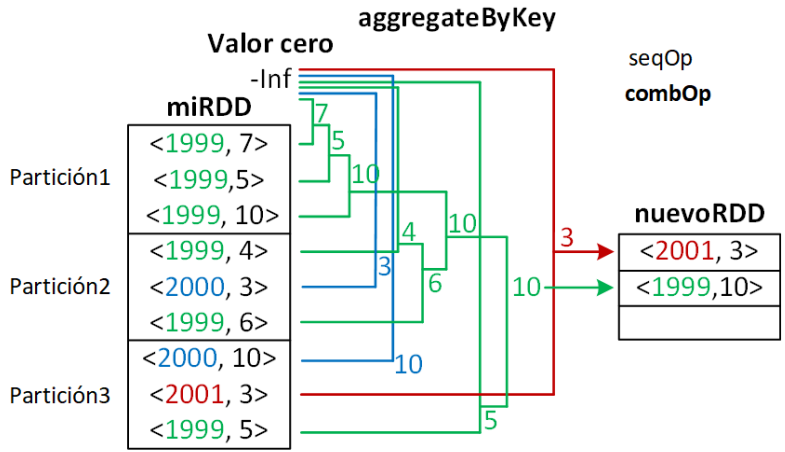
```
miRDD: [(1999, 7), (1999, 5), (1999, 10), (1999, 4), (2000,
3), (1999, 6), (2000, 10), (2001, 3), (1999, 5)]
nuevoRDD: [(2001, 3), (1999, 10), (2000, 10)]
```



- aggregateByKey (3 parámetros)
  - Un valor cero/inicial con el que se empezará hacer la agregación
  - seqOP: Una función parecida a Combine que se ejecuta con los datos locales de las particiones
  - combOp: Una función parecida a Reduce que se ejecuta con los datos resultantes de las particiones

```
def miSeqFuncion(accum, value):
                    max = accum
                    if value > accum:
                                        max = value
                    return max

def miCombFuncion(accum, value):
                    max = accum
                    if value > accum:
                                        max = value
                    return max

miRDD = sc.parallelize([(1999,7),(1999,5), (1999,10),
                        (1999,4), (2000,3), (1999,6),
                        (2000,10), (2001,3), (1999,5)], 3)

nuevoRDD = miRDD.aggregateByKey(- float("inf"), miSeqFuncion, miCombFuncion)
```

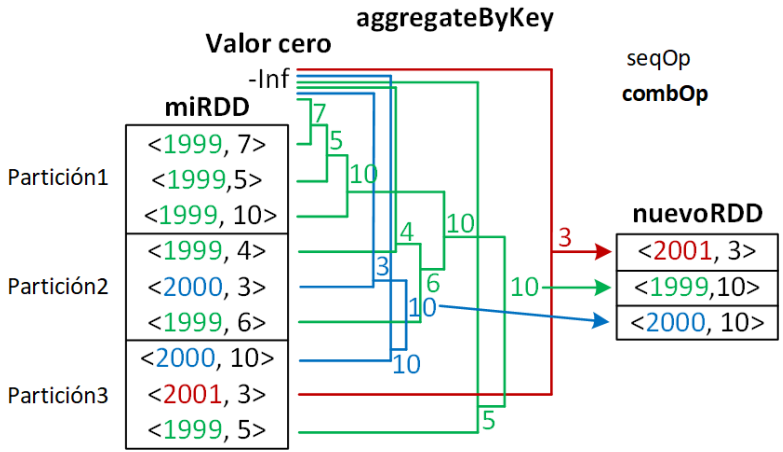
```
miRDD: [(1999, 7), (1999, 5), (1999, 10), (1999, 4), (2000,
3), (1999, 6), (2000, 10), (2001, 3), (1999, 5)]
nuevoRDD: [(2001, 3), (1999, 10), (2000, 10)]
```



- aggregateByKey (3 parámetros)
  - Un valor cero/inicial con el que se empezará hacer la agregación
  - seqOP: Una función parecida a Combine que se ejecuta con los datos locales de las particiones
  - combOp: Una función parecida a Reduce que se ejecuta con los datos resultantes de las particiones

```
def miSeqFuncion(accum, value):
                    max = accum
                    if value > accum:
                                        max = value
                    return max

def miCombFuncion(accum, value):
                    max = accum
                    if value > accum:
                                        max = value
                    return max

miRDD = sc.parallelize([(1999,7),(1999,5), (1999,10),
                        (1999,4), (2000,3), (1999,6),
                        (2000,10), (2001,3), (1999,5)], 3)

nuevoRDD = miRDD.aggregateByKey(- float("inf"), miSeqFuncion, miCombFuncion)
```

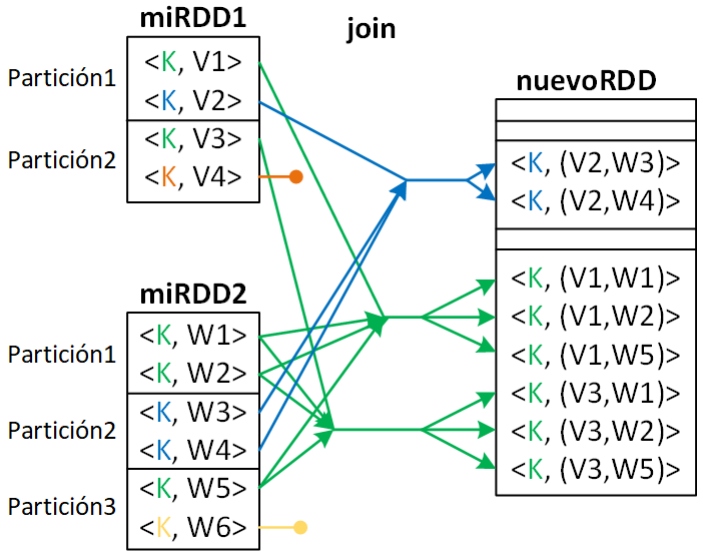
```
miRDD: [(1999, 7), (1999, 5), (1999, 10), (1999, 4), (2000,
3), (1999, 6), (2000, 10), (2001, 3), (1999, 5)]
nuevoRDD: [(2001, 3), (1999, 10), (2000, 10)]
```



- join
  - Ejecuta un inner join
  - Oros joins soportados por spark: leftOuterJoin y rightOuterJoin

```
miRDD1 = sc.parallelize([("compra1", "tarjeta1"),("compra2", "tarjeta1"),
                         ("compra1", "tarjeta2"), ("compra3", "tarjeta1")], 2)

miRDD2 = sc.parallelize([("compra1", "Ordenador") , ("compra1", "altavoces"),
                         ("compra2", "coche"), ("compra2", "ruedas"),
                         ("compra1", "teclado"), ("compra4", "otro")], 3)


nuevoRDD = miRDD1.join(miRDD2)
```

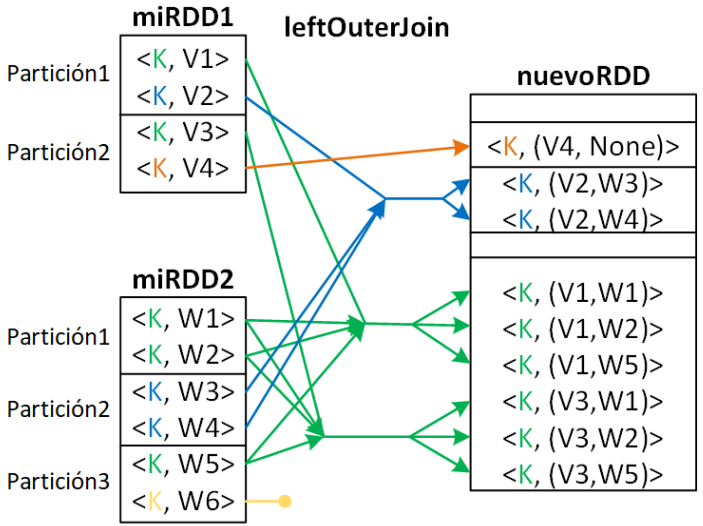
```
miRDD1: [('compra1', 'tarjeta1'), ('compra2', 'tarjeta1'), ('compra1', 'tarjeta2
'), ('compra3', 'tarjeta1')]
miRDD2: [('compra1', 'Ordenador'), ('compra1', 'altavoces'), ('compra2', 'coche'
), ('compra2', 'ruedas'), ('compra1', 'teclado'), ('compra4', 'otro')]
nuevoRDD: [('compra2', ('tarjeta1', 'coche')), ('compra2', ('tarjeta1', 'ruedas'
)), ('compra1', ('tarjeta1', 'Ordenador')), ('compra1', ('tarjeta1', 'altavoces'
)), ('compra1', ('tarjeta1', 'teclado')), ('compra1', ('tarjeta2', 'Ordenador'))
, ('compra1', ('tarjeta2', 'altavoces')), ('compra1', ('tarjeta2', 'teclado'))]
```



- join
  - Ejecuta un inner join
  - Oros joins soportados por spark: **leftOuterJoin** y rightOuterJoin

```
miRDD1 = sc.parallelize([("compra1", "tarjeta1"),("compra2", "tarjeta1"),
                         ("compra1", "tarjeta2"), ("compra3", "tarjeta1")], 2)

miRDD2 = sc.parallelize([("compra1", "Ordenador") , ("compra1", "altavoces"),
                         ("compra2", "coche"), ("compra2", "ruedas"),
                         ("compra1", "teclado"), ("compra4", "otro")], 3)


nuevoRDD = miRDD1.leftOuterJoin(miRDD2)
```

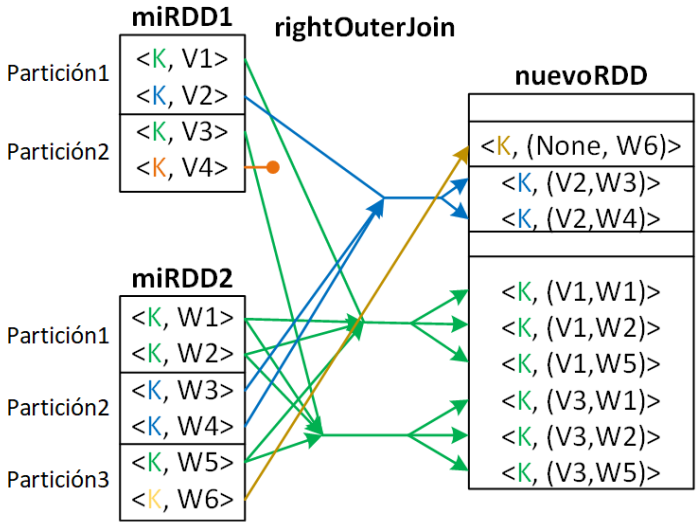
```
miRDD1: [('compra1', 'tarjeta1'), ('compra2', 'tarjeta1'), ('compra1', 'tarjeta2
'), ('compra3', 'tarjeta1')]
miRDD2: [('compra1', 'Ordenador'), ('compra1', 'altavoces'), ('compra2', 'coche'
), ('compra2', 'ruedas'), ('compra1', 'teclado'), ('compra4', 'otro')]
nuevoRDD: [('compra3', ('tarjeta1', None)), ('compra2', ('tarjeta1', 'coche')),
('compra2', ('tarjeta1', 'ruedas')), ('compra1', ('tarjeta1', 'Ordenador')), ('c
ompra1', ('tarjeta1', 'altavoces')), ('compra1', ('tarjeta1', 'teclado')), ('com
pra1', ('tarjeta2', 'Ordenador')), ('compra1', ('tarjeta2', 'altavoces')), ('com
pra1', ('tarjeta2', 'teclado'))]
```



- join
  - Ejecuta un inner join
  - Oros joins soportados por spark: leftOuterJoin y **rightOuterJoin**

```
miRDD1 = sc.parallelize([("compra1", "tarjeta1"),("compra2", "tarjeta1"),
                         ("compra1", "tarjeta2"), ("compra3", "tarjeta1")], 2)

miRDD2 = sc.parallelize([("compra1", "Ordenador") , ("compra1", "altavoces"),
                         ("compra2", "coche"), ("compra2", "ruedas"),
                         ("compra1", "teclado"), ("compra4", "otro")], 3)


nuevoRDD = miRDD1.rightOuterJoin(miRDD2)
```

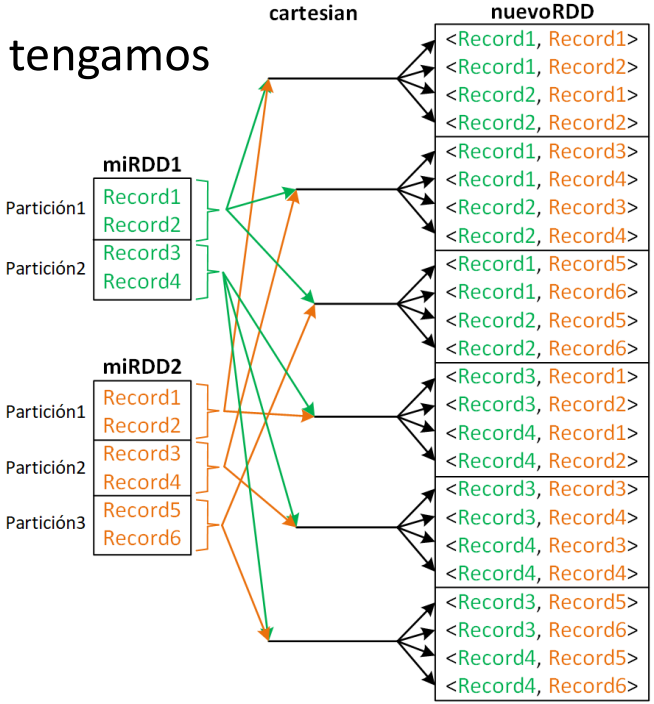
```
miRDD1: [('compra1', 'tarjeta1'), ('compra2', 'tarjeta1'), ('compra1', 'tarjeta2
'), ('compra3', 'tarjeta1')]
miRDD2: [('compra1', 'Ordenador'), ('compra1', 'altavoces'), ('compra2', 'coche'
), ('compra2', 'ruedas'), ('compra1', 'teclado'), ('compra4', 'otro')]
nuevoRDD: [('compra4', (None, 'otro')), ('compra2', ('tarjeta1', 'coche')), ('co
mpra2', ('tarjeta1', 'ruedas')), ('compra1', ('tarjeta1', 'Ordenador')), ('compr
a1', ('tarjeta1', 'altavoces')), ('compra1', ('tarjeta1', 'teclado')), ('compra1
', ('tarjeta2', 'Ordenador')), ('compra1', ('tarjeta2', 'altavoces')), ('compra1
', ('tarjeta2', 'teclado'))]
```



- cartesian
  - Crea todos los pares de todos los datos que tengamos

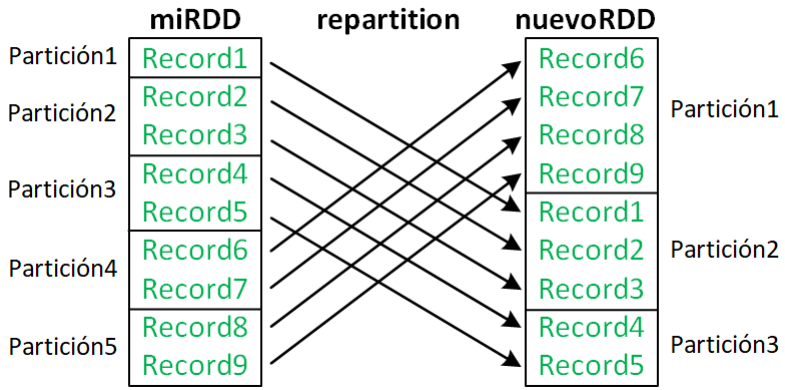
```
miRDD1 = sc.parallelize(["tarjeta1", "tarjeta2",
                         "tarjeta3", "tarjeta4"], 2)

miRDD2 = sc.parallelize(["Ordenador", "altavoces",
                         "coche", "ruedas",
                         "teclado", "otro"], 3)


nuevoRDD = miRDD1.cartesian(miRDD2)
```

```
miRDD1: ['tarjeta1', 'tarjeta2', 'tarjeta3', 'tarjeta4']
miRDD2: ['Ordenador', 'altavoces', 'coche', 'ruedas', 'teclado', 'otro']
nuevoRDD: [('tarjeta1', 'Ordenador'), ('tarjeta1', 'altavoces'), ('tarjeta2', 'O
rdenador'), ('tarjeta2', 'altavoces'), ('tarjeta1', 'coche'), ('tarjeta1', 'rued
as'), ('tarjeta2', 'coche'), ('tarjeta2', 'ruedas'), ('tarjeta1', 'teclado'), ('
tarjeta1', 'otro'), ('tarjeta2', 'teclado'), ('tarjeta2', 'otro'), ('tarjeta3',
'Ordenador'), ('tarjeta3', 'altavoces'), ('tarjeta4', 'Ordenador'), ('tarjeta4',
 'altavoces'), ('tarjeta3', 'coche'), ('tarjeta3', 'ruedas'), ('tarjeta4', 'coch
e'), ('tarjeta4', 'ruedas'), ('tarjeta3', 'teclado'), ('tarjeta3', 'otro'), ('ta
rjeta4', 'teclado'), ('tarjeta4', 'otro')]
```



- repartition
  - Reduce/aumenta el número de particiones
  - Intenta balancear la carga
  - Es compleja porque se tiene que hacer un shuffle

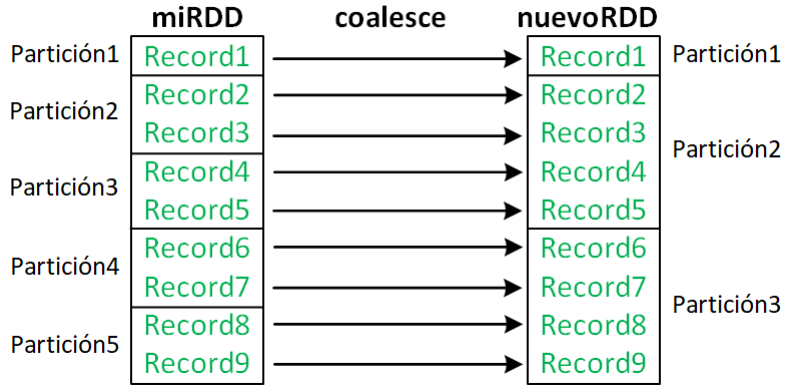
```
miRDD = sc.parallelize(["1",
                        "2", "3",
                        "4", "5",
                        "6", "7",
                        "8", "9"], 5)

nuevoRDD = miRDD.repartition(3)
```




```
miRDD: ['1', '2', '3', '4', '5', '6', '7', '8', '9']
nuevoRDD: ['6', '7', '8', '9', '1', '2', '3', '4', '5']
```

- coalesce
  - Reduce el número de particiones
  - Suele ser más óptima que repartition (no hace un shuffle completo)

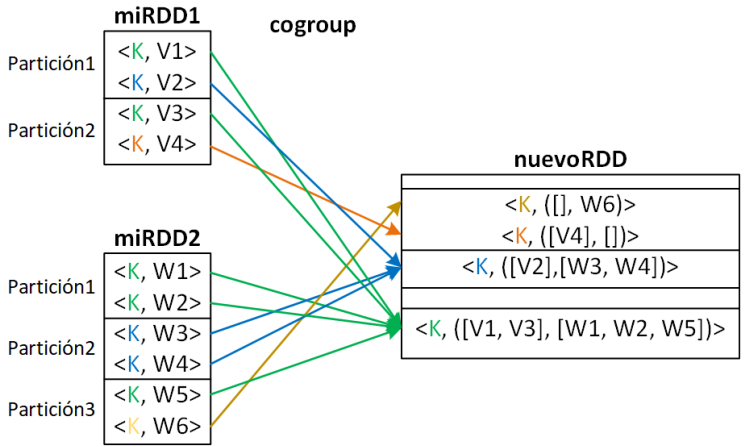
```
miRDD = sc.parallelize(["1",
                        "2", "3",
                        "4", "5",
                        "6", "7",
                        "8", "9"], 5)

nuevoRDD = miRDD.coalesce(3)
```




```
miRDD: ['1', '2', '3', '4', '5', '6', '7', '8', '9']
nuevoRDD: ['1', '2', '3', '4', '5', '6', '7', '8', '9']
```

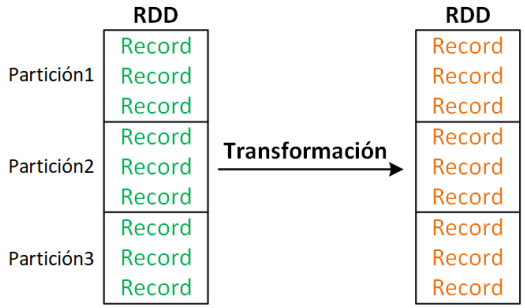
- cogroup
  - Se agrupan los valores de dos RDD por la clave
  - Parecido a un full outer join de los dos RDD

```
miRDD1 = sc.parallelize([("compra1", "tarjeta1"),("compra2", "tarjeta1"),
                         ("compra1", "tarjeta2"), ("compra3", "tarjeta1")], 2)

miRDD2 = sc.parallelize([("compra1", "Ordenador") , ("compra1", "altavoces"),
                         ("compra2", "coche"), ("compra2", "ruedas"),
                         ("compra1", "teclado"), ("compra4", "otro")], 3)

nuevoRDD = miRDD1.cogroup(miRDD2)
```

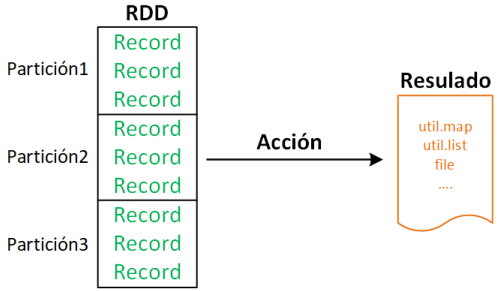



```
< compra4 , [] ['otro'] >
< compra3 , ['tarjeta1'] [] >
< compra2 , ['tarjeta1'] ['coche', 'ruedas'] >
< compra1 , ['tarjeta1', 'tarjeta2'] ['Ordenador', 'altavoces', 'teclado'] >
```

- Transformaciones: “generan” nuevos RDD



- Acciones: generan resultados

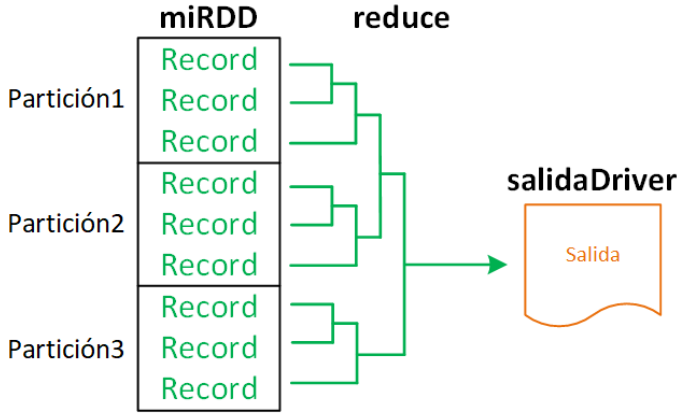


- Acciones:
  - reduce
  - collect
  - count
  - first
  - take
  - takeSample

- Acciones:
  - takeOrdered
  - saveAsTextFile
  - saveAsSequenceFile (java/scala)
  - saveAsObject (java/scala)
  - countByKey
  - Foreach
  - getNumPartitions

- reduce
  - Aplica una función a todos los registros del RDD
  - Devuelve el resultado al driver

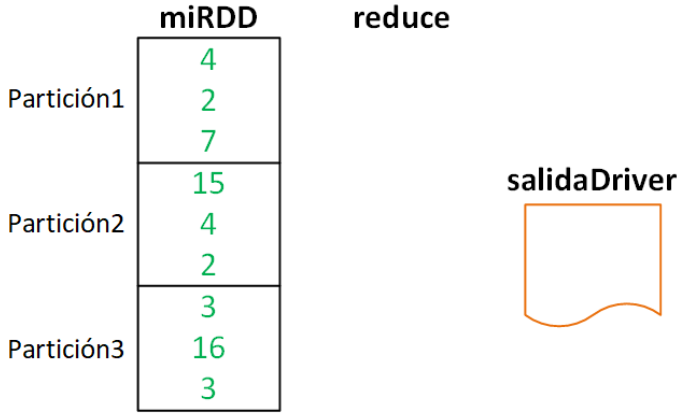
```
def miFuncion(acumulado, valor):
    max = acumulado
    if valor > max:
        max = valor
    return max

miRDD = sc.parallelize([4, 2, 7,
                        15, 4, 2,
                        3, 16, 3],3)
salidaDriver = miRDD.reduce(miFuncion)
```




```
miRDD: [4, 2, 7, 15, 4, 2, 3, 16, 3]
Salida en driver: 16
```

- reduce
  - Aplica una función a todos los registros del RDD
  - Devuelve el resultado al driver

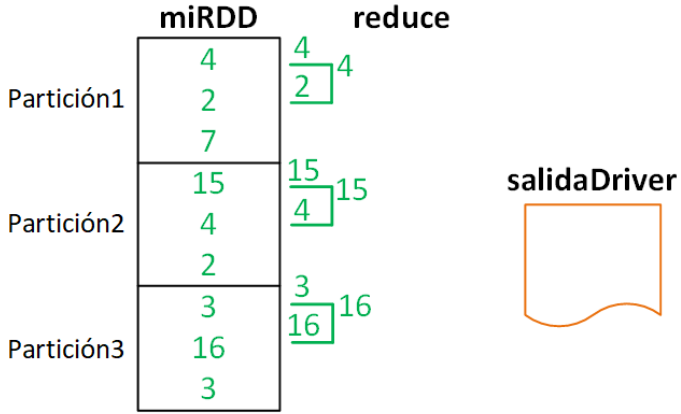
```
def miFuncion(acumulado, valor):
    max = acumulado
    if valor > max:
        max = valor
    return max

miRDD = sc.parallelize([4, 2, 7,
                        15, 4, 2,
                        3, 16, 3],3)
salidaDriver = miRDD.reduce(miFuncion)
```




```
miRDD: [4, 2, 7, 15, 4, 2, 3, 16, 3]
Salida en driver: 16
```

- reduce
  - Aplica una función a todos los registros del RDD
  - Devuelve el resultado al driver

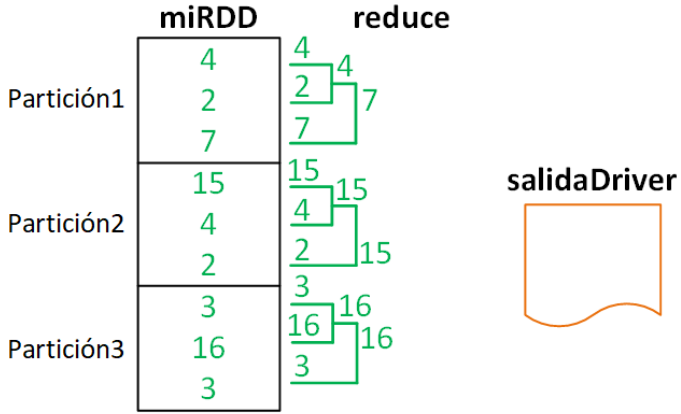
```
def miFuncion(acumulado, valor):
    max = acumulado
    if valor > max:
        max = valor
    return max

miRDD = sc.parallelize([4, 2, 7,
                        15, 4, 2,
                        3, 16, 3],3)
salidaDriver = miRDD.reduce(miFuncion)
```




```
miRDD: [4, 2, 7, 15, 4, 2, 3, 16, 3]
Salida en driver: 16
```

- reduce
  - Aplica una función a todos los registros del RDD
  - Devuelve el resultado al driver

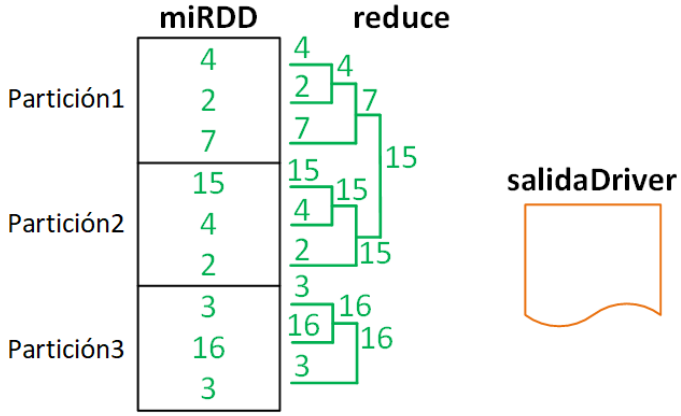
```
def miFuncion(acumulado, valor):
    max = acumulado
    if valor > max:
        max = valor
    return max

miRDD = sc.parallelize([4, 2, 7,
                        15, 4, 2,
                        3, 16, 3],3)
salidaDriver = miRDD.reduce(miFuncion)
```




```
miRDD: [4, 2, 7, 15, 4, 2, 3, 16, 3]
Salida en driver: 16
```

- reduce
  - Aplica una función a todos los registros del RDD
  - Devuelve el resultado al driver

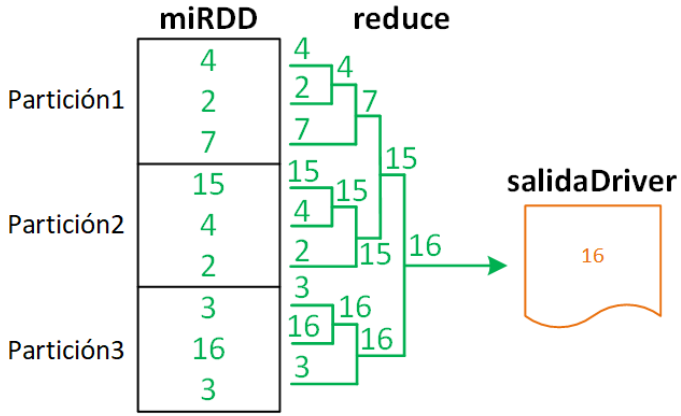
```
def miFuncion(acumulado, valor):
    max = acumulado
    if valor > max:
        max = valor
    return max

miRDD = sc.parallelize([4, 2, 7,
                        15, 4, 2,
                        3, 16, 3],3)
salidaDriver = miRDD.reduce(miFuncion)
```




```
miRDD: [4, 2, 7, 15, 4, 2, 3, 16, 3]
Salida en driver: 16
```

- reduce
  - Aplica una función a todos los registros del RDD
  - Devuelve el resultado al driver

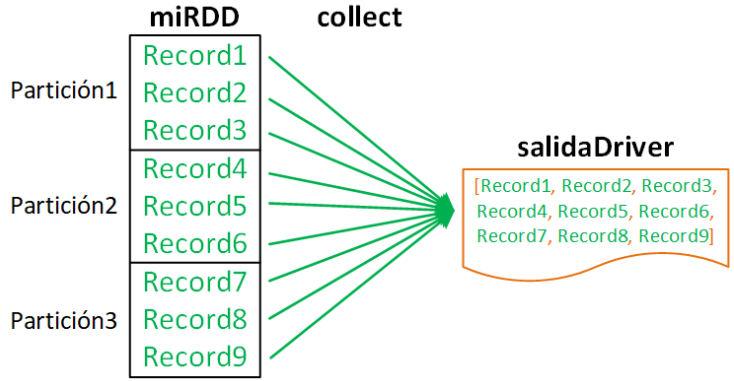
```
def miFuncion(acumulado, valor):
    max = acumulado
    if valor > max:
        max = valor
    return max

miRDD = sc.parallelize([4, 2, 7,
                        15, 4, 2,
                        3, 16, 3],3)
salidaDriver = miRDD.reduce(miFuncion)
```




```
miRDD: [4, 2, 7, 15, 4, 2, 3, 16, 3]
Salida en driver: 16
```

- collect
  - Devuelve al driver un array con el RDD
  - Se utiliza para pruebas con pocos datos

```
miRDD = sc.parallelize([4, 2, 7,
                        15, 4, 2,
                        3, 16, 3],3)

salidaDriver = miRDD.collect()
```

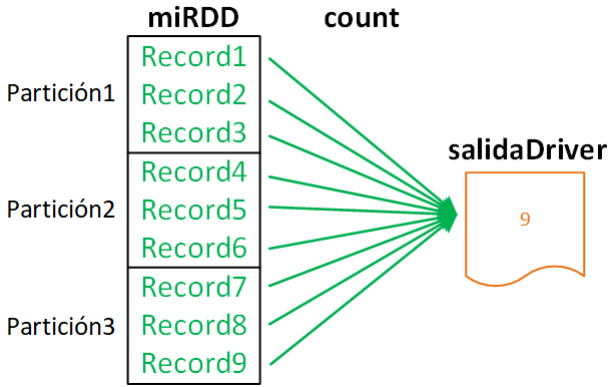



```
miRDD: [4, 2, 7, 15, 4, 2, 3, 16, 3]
Salida en driver: [4, 2, 7, 15, 4, 2, 3, 16, 3]
```

- count
  - Devuelve al driver el número de elementos del RDD

```
miRDD = sc.parallelize([4, 2, 7,
                        15, 4, 2,
                        3, 16, 3],3)

salidaDriver = miRDD.count()
```

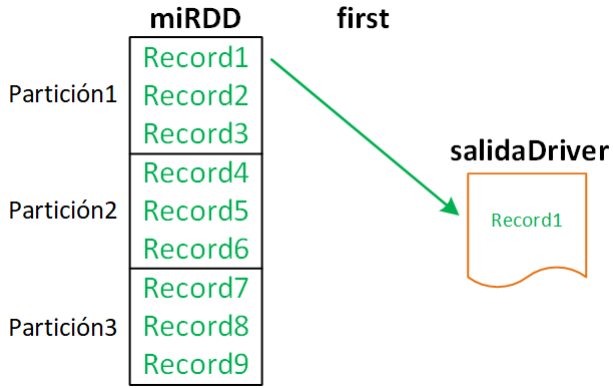
```
miRDD: [4, 2, 7, 15, 4, 2, 3, 16, 3]
Salida en driver: 9
```



- first
  - Devuelve al driver el primer elemento del RDD

```
miRDD = sc.parallelize([4, 2, 7,
                        15, 4, 2,
                        3, 16, 3],3)

salidaDriver = miRDD.first()
```

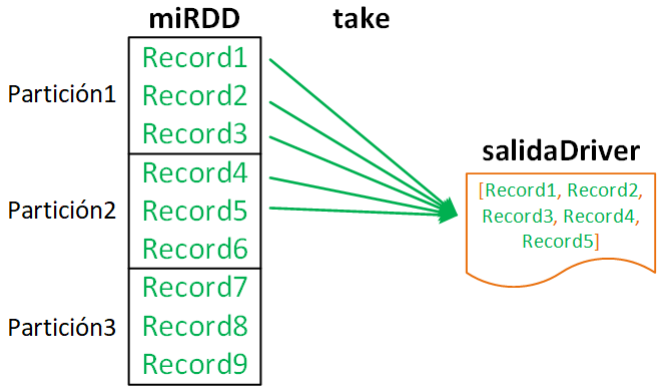
```
miRDD: [4, 2, 7, 15, 4, 2, 3, 16, 3]
Salida en driver: 4
```



- take
  - Devuelve al driver los primeros N elementos del RDD

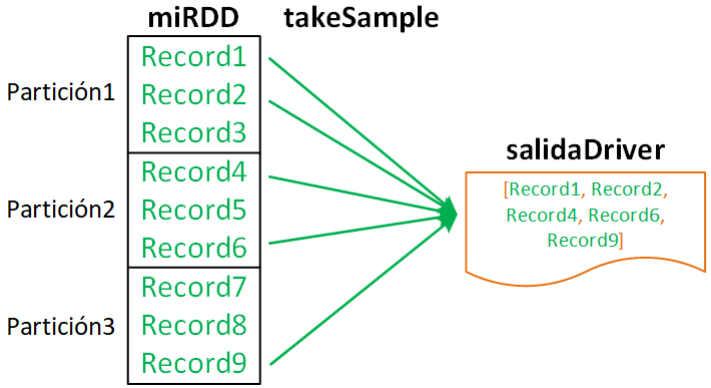
```
miRDD = sc.parallelize([4, 2, 7,
                        15, 4, 2,
                        3, 16, 3],3)

salidaDriver = miRDD.take(5)
```

```
miRDD: [4, 2, 7, 15, 4, 2, 3, 16, 3]
Salida en driver: [4, 2, 7, 15, 4]
```



- takeSample
  - Devuelve al driver un array de N elementos aleatorios
  - Con/Sin reemplazamiento
  - Seed

```
miRDD = sc.parallelize([4, 2, 7,
                        15, 4, 2,
                        3, 16, 3],3)

salidaDriver = miRDD.takeSample(False, 5)
```

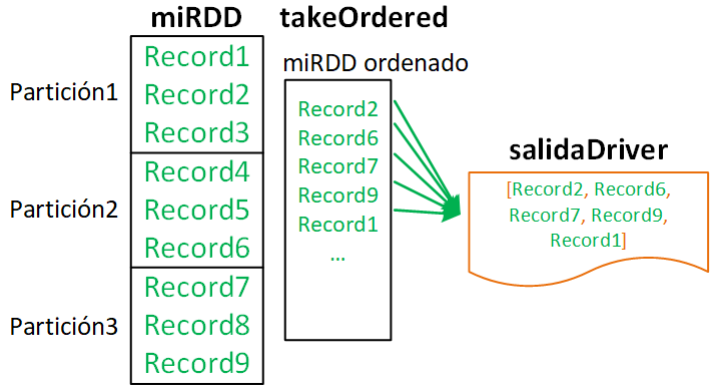



```
miRDD: [4, 2, 7, 15, 4, 2, 3, 16, 3]
Salida en driver: [2, 15, 2, 4, 3]
```

- takeOrdered
  - Devuelve al driver un array de N elementos (según un ordenamiento)
  - Se puede cambiar la función de ordenación

```
miRDD = sc.parallelize([4, 2, 7,
                        15, 4, 2,
                        3, 16, 3],3)
salidaDriver = miRDD.takeOrdered(5)
```

```
miRDD: [4, 2, 7, 15, 4, 2, 3, 16, 3]
Salida en driver: [2, 2, 3, 3, 4]
```



saveAsTextFile

Guarda el RDD en local o en un DFS

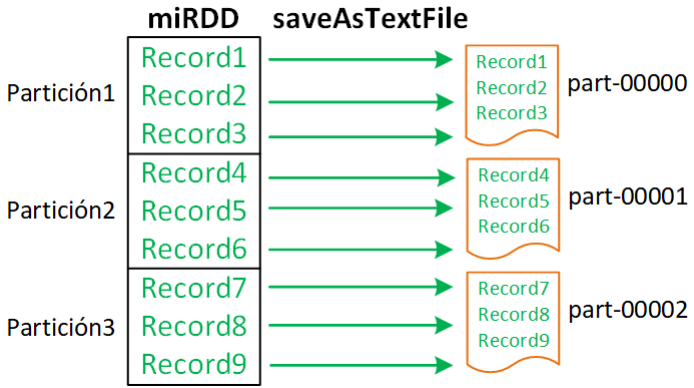
saveAsTextFile(*path*)

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.

```
miRDD = sc.parallelize([4, 2, 7,
                        15, 4, 2,
                        3, 16, 3],3)

miRDD.saveAsTextFile("datasetGuardado")
```

```
[hadmin@INHTEST ~]$ hdfs dfs -cat datasetGuardado/*
4
2
7
15
4
2
3
16
3
```

```
[hadmin@INHTEST ~]$ hdfs dfs -ls
Found 84 items
drwxr-xr-x   - hadmin hadoop          0 2018-08-22 14:21 .sparkStaging
drwxr-xr-x   - hadmin hadoop          0 2018-11-25 22:28 datasetGuardado
drwxr-xr-x   - hadmin hadoop          0 2016-12-07 16:30 ej_sampling_bor
```



saveAsTextFile

Guarda el RDD en local o en un DFS

saveAsTextFile(*path*)

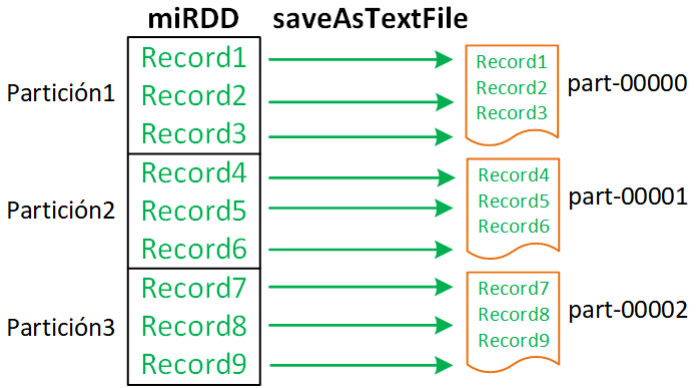
Write the elements of the dataset as a text file (or set of text files) in a given directory in the local filesystem, HDFS or any other Hadoop-supported file system. Spark will call toString on each element to convert it to a line of text in the file.

```
miRDD = sc.parallelize([4, 2, 7,
                        15, 4, 2,
                        3, 16, 3],3)

miRDD.saveAsTextFile("file:///home/hadmin/datasetGuardado")
```

```
[hadmin@INHTEST ~]$ ls -l
drwxr-xr-x.  2 hadmin hadoop     4096 nov 25 22:49 datasetGuardado
-rw-r--r--.  1 hadmin hadoop      680 nov 25 22:25 derby.log
```

```
[hadmin@INHTEST ~]$ ls -l /home/hadmin/datasetGuardado/
total 12
-rw-r--r--. 1 hadmin hadoop 6 nov 25 22:49 part-00000
-rw-r--r--. 1 hadmin hadoop 7 nov 25 22:49 part-00001
-rw-r--r--. 1 hadmin hadoop 7 nov 25 22:49 part-00002
-rw-r--r--. 1 hadmin hadoop 0 nov 25 22:49 _SUCCESS
```

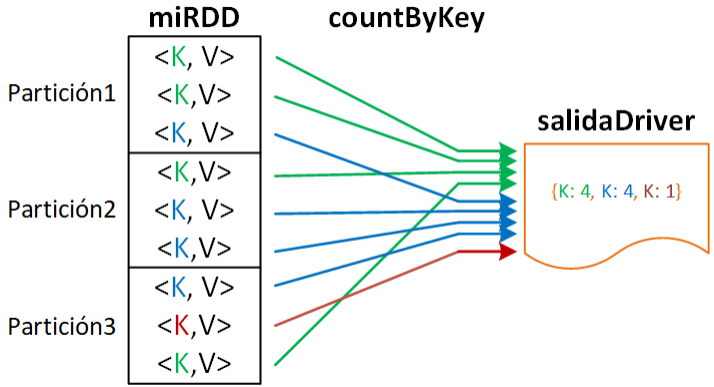
```
[hadmin@INHTEST ~]$ cat /home/hadmin/datasetGuardado/*
4
2
7
15
4
2
3
16
3
```



countByKey

- Devuelve al driver un diccionario con <clave, número registros>

```
miRDD = sc.parallelize([("1999", 7),("1999", 5), ("2000", 10),
                        ("1999", 4), ("2000", 3), ("2000", 7),
                        ("2000", 10), ("2001", 3), ("1999", 5)], 3)

salidaDriver = miRDD.countByKey()
```

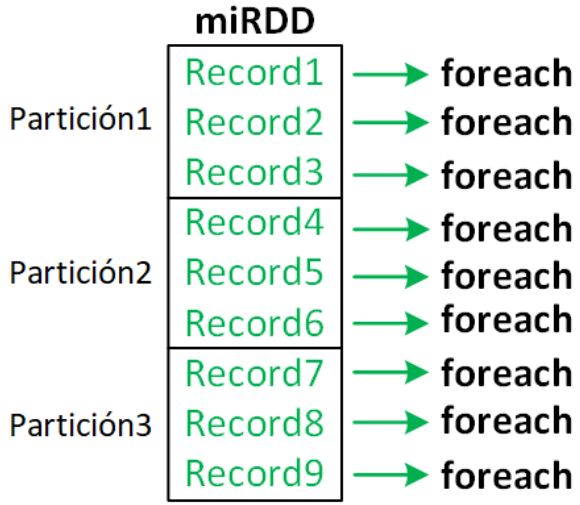



```
miRDD: [('1999', 7), ('1999', 5), ('2000', 10), ('1999', 4), ('2000', 3), ('2000
', 7), ('2000', 10), ('2001', 3), ('1999', 5)]
Salida en driver: defaultdict(<type 'int'>, {'1999': 4, '2000': 4, '2001': 1})
```

- foreach
  - Aplica una función a cada registro
  - Se ejecuta distribuido en el cluster

```
def miFuncion(record):
    print len(record)

miRDD = sc.parallelize(["uno", "dos", "tres",
                        "cuatro", "cinco", "seis",
                        "siete", "ocho", "nueve"],3)

miRDD.foreach(miFuncion)
```

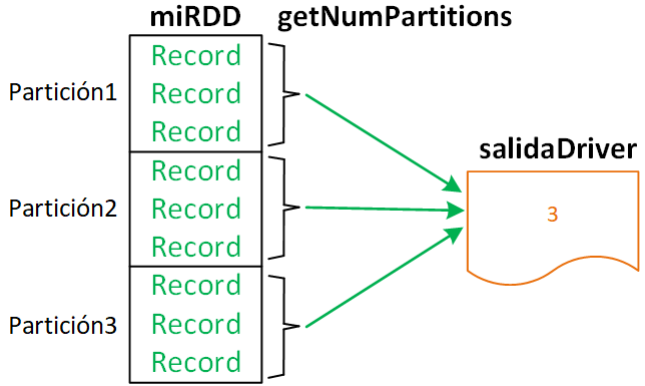
```
>>> miRDD.foreach(miFuncion)
6
5
4
3
3
4
5
4
5
```




```
miRDD: ['uno', 'dos', 'tres', 'cuatro', 'cinco', 'sei
s', 'siete', 'ocho', 'nueve']
```

- getNumPartitions
  - Obtiene el número de particiones

```
miRDD = sc.parallelize(["uno", "dos", "tres",
                        "cuatro", "cinco", "seis",
                        "siete", "ocho", "nueve"],3)

salidaDriver = miRDD.getNumPartitions()
```




```
miRDD: ['uno', 'dos', 'tres', 'cuatro', 'cinco', 'seis', 'siete', 'ocho', 'nueve']
Salida en driver: 3
```

- Consultar particiones
  - Permite imprimir por pantalla las particiones y su contenido
  - Es útil para realizar pruebas

```
miRDD = sc.parallelize(["uno", "dos", "tres",
                        "cuatro", "cinco", "seis",
                        "siete", "ocho", "nueve"],3)

#imprimir particiones de miRDD
print "\nmiRDD:"
partitions = miRDD.glom().collect()
for partition in partitions:
    print("Particion: ", partition)
```

```
miRDD:
('Particion: ', ['uno', 'dos', 'tres'])
('Particion: ', ['cuatro', 'cinco', 'seis'])
('Particion: ', ['siete', 'ocho', 'nueve'])
```

Gracias

viu | Universidad Internacional de Valencia