
UNIVERSIDAD COMPLUTENSE MADRID

# Apache Spark

Enrique Martín (emartinm@ucm.es)
Sistemas de Gestión de Datos y de la Información
Master Ing. Informática
Fac. Informática

# Contenidos

# Bibliografía

# Bibliografía


- *Learning Spark, 2nd Edition*. Jules S. Damji, Brooke Wenig, Tathagata Das, Denny Lee. O'Reilly, 2020
- *Fast Data Processing with Spark*. Holden Karau. Pakct, 2013.
- *Resilient Distributed Datasets: A Fault-Tolerant Abstraction for In-Memory Cluster Computing*. Matei Zaharia et. al. Proceedings NSDI'12. Disponible en `https://www.usenix.org/system/files/conference/nsdi12/nsdi12-final138.pdf`

# Introducción

# Apache Spark

- Apache Spark es un sistema de **cómputo distribuido** alternativo a MapReduce
- Surgió en 2009 en el AMPLab de la Universidad de California en Berkeley
- En 2013 pasó a ser un proyecto Apache, donde actualmente es uno de los proyectos más activos

# Características de Apache Spark

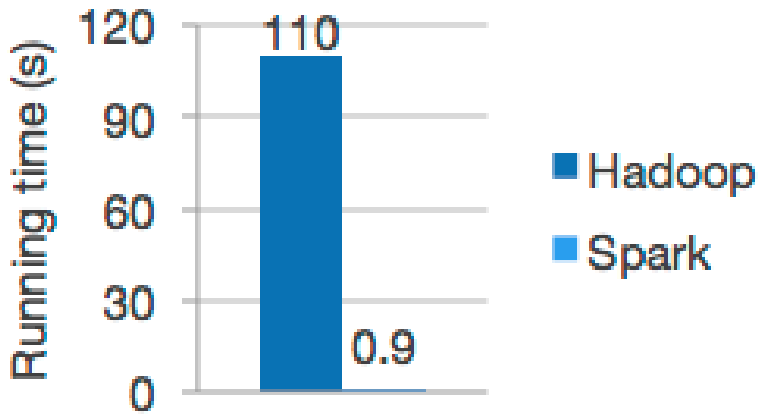
- Al igual que MapReduce:
  - Es **escalable**: se pueden añadir nuevos equipos y la capacidad de cómputo aumenta
  - Es **resistente a fallos**: los equipos pueden caer y el cómputo no se aborta sino que recupera los datos perdidos
- Sin embargo, **mejora a MapReduce** en varios aspectos:
  - La organización del cómputo es más **flexible**, con una gran cantidad de **operaciones** disponibles.
  - Trabaja en memoria, por lo que es **más eficiente**

# Flexibilidad de Spark

- Apache Spark es más **flexible** ya que proporciona una gran cantidad de operaciones para transformar nuestro conjunto de datos: `map`, `filter`, `flatMap`, `aggregateByKey`, etc.
- Estas operaciones se pueden **combinar** de cualquier manera compleja, sin necesidad de volcar resultados intermedios a disco

# Eficiencia de Spark

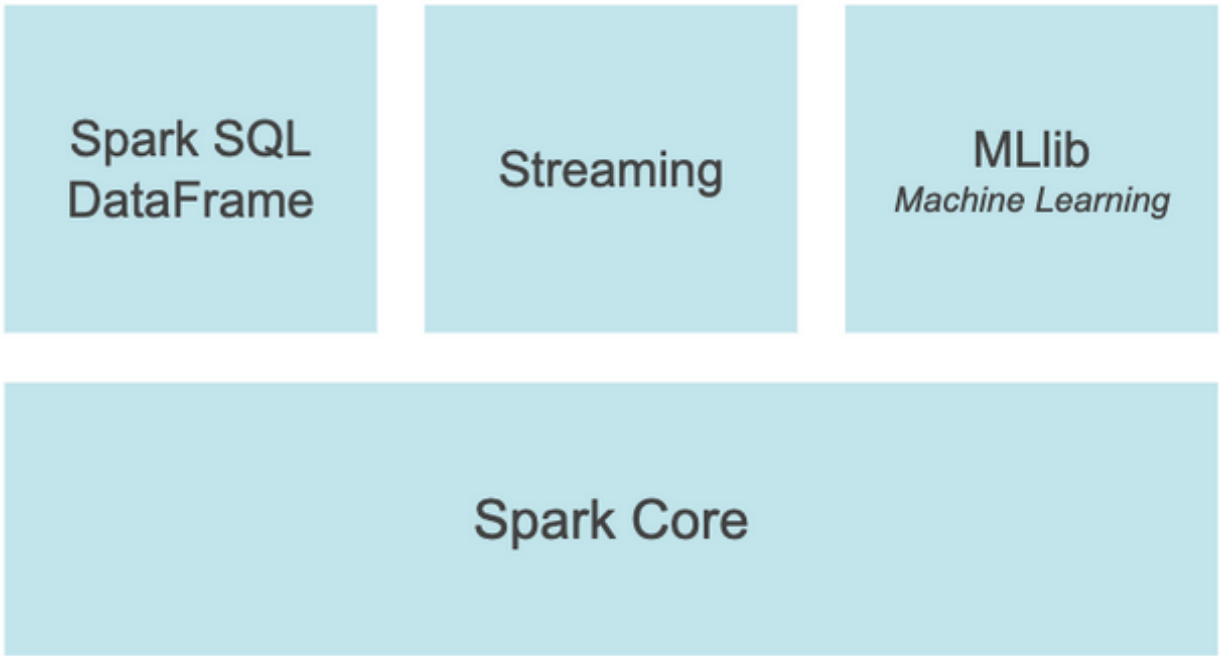
- En Spark los datos residen principalmente en memoria, por lo que la velocidad de cómputo es mucho mayor
- Puede conseguir mejoras cercanas al 100x frente a MapReduce




**Fuente:** `https://spark.apache.org/`

# Bibliotecas de Spark

- Además del núcleo, dispone de varias **bibliotecas** con utilidades para Big Data:
  - **SQL y DataFrames**: consultas SQL
  - **Spark Streaming**: proceso de flujos
  - **MLlib**: algoritmos de aprendizaje automático
  - **GraphX**: procesamiento de grafos



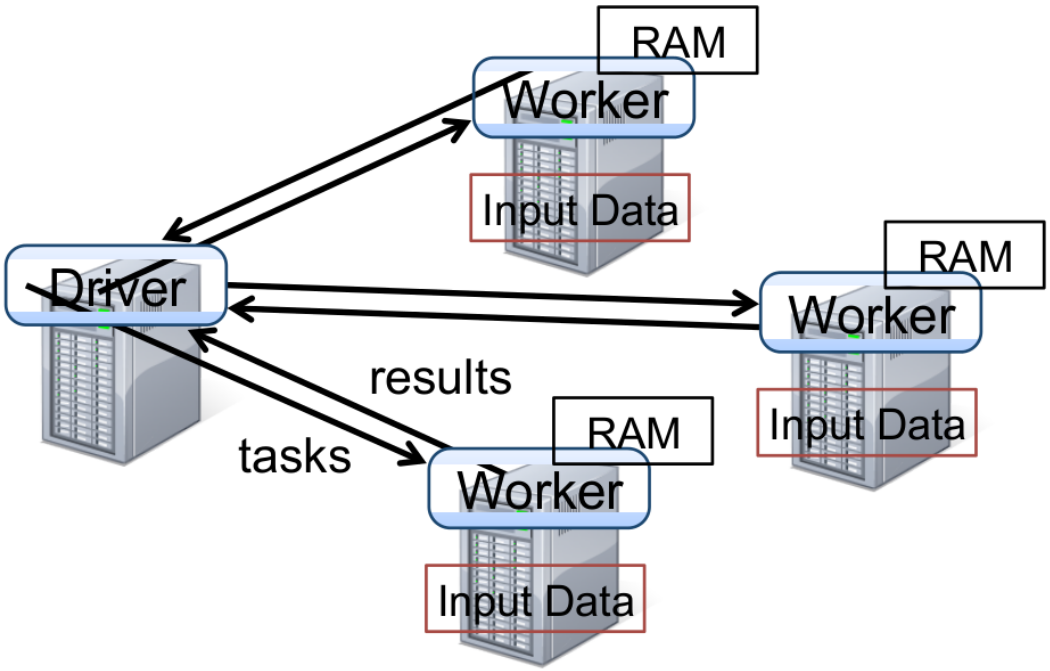

**Fuente:** `https://spark.apache.org/`

# Programación en Spark

- El lenguaje oficial de Spark es Scala:
  - Funcional + imperativo
  - Con tipos
  - Funciones anónimas y orden superior
- Sin embargo también se puede utilizar desde Java, **Python** y R

# Organización de un cómputo en Spark

- El programa *driver* se conecta a un clúster de *workers*
- El *driver* define uno o varios RDDs e invoca acciones en ellos
- Los *workers* pueden almacenar fragmentos de RDDs entre operaciones en su memoria (y también en disco)




**Fuente:** Resilient Distributed Datasets: A Fault-Tolerant Abstraction for In-Memory Cluster Computing. Matei Zaharia et. al. Proc. NSDI'12.

# Resilient Distributed Dataset (RDD)

# RDD

- Es el **concepto fundamental** de Spark, y en el que expresan todos los cómputos
- Un Resilient Distributed Dataset (RDD) es una **colección** de elementos que se **reparte** entre la memoria de los equipos del clúster
- Un RDD es **resiliente** porque si un equipo deja de funcionar, el fragmento de RDD perdido se vuelve a recalcular de manera transparente
- Spark calcula automáticamente las particiones de un RDD
- El programador puede indicar explícitamente cuántas particiones quiere o cómo quiere realizar el particionado (hash, función personalizada)

# RDD

- Sobre los RDDs se construyen interfaces de cómputo de alto nivel como los **DataFrames** o los **DataSets**
- Se programa realizando operaciones en paralelo, que afectan a todos los fragmentos del RDD
- Un RDD puede ser creado desde **ficheros**, o ser el resultado de **transformar otro RDD**
- Un RDD es **inmutable**: una vez creado nunca se modifica

# Operaciones de RDD

- Los RDDs admiten dos tipos de operaciones:
  - **Transformaciones**: realizan algún cálculo sobre los datos y crean un nuevo RDD
  - **Acciones**: devuelven un valor al controlador (*driver*) o vuelcan los datos al almacenamiento
- Las transformaciones no se ejecutan en el momento, sino que se **planifican**. Cuando se realiza una acción es cuando se llevan a cabo todas las transformaciones.
- Retrasar las transformaciones permite a Spark **optimizar** transformaciones (p.ej. fusionando varias etapas en una sola)

# Tipos de RDD

- Los RDDs son colecciones de elementos iguales. Estos elementos pueden ser:
  - De un mismo tipo: `int`, `str`, `float`, ...
  - Parejas (clave,valor)
- Los RDDs clave-valor proporcionan operaciones específicas que tienen en cuenta la clave, que veremos más adelante

# Usando Spark desde Python

# Spark desde Python

- A continuación veremos distintos ejemplos de transformaciones en Spark usando Python:
  - `pyspark` lanza el intérprete interactivo
  - `spark-submit` permite lanzar un fichero Python (Scala o Java) como programa completo
- Normalmente se usa `pyspark` para realizar pruebas intermedias e ir escribiendo poco a poco el programa final que se lanzará con `spark-submit`

# Funciones como parámetros

- La mayoría de operaciones aceptan funciones como parámetros
- Si la función está definida, se puede usar directamente su nombre. Por ejemplo duplicar todos los elementos de un RDD con `rdd.map(duplica)`
- Normalmente se suele usar funciones anónimas, por ejemplo

```
rdd.map(lambda x: x*2)
```

## Ejemplos de funciones anónimas

- `lambda x: x+2 #int -> int`
- `lambda x: x>0 #int -> bool`
- `lambda x,y: (len(x), y*3) #str -> int -> (int,int)`

# Creación de RDD

- Para la creación de RDD se utilizan métodos del `SparkContext`
- Desde la versión 2.0 el punto de entrada principal es `SparkSession`, que incluye a `SparkContext`
- `pySpark` carga por defecto la `SparkSession` en la variable `spark` y su `SparkContext` asociado en la variable `sc`
- En los programas completos lanzados con `spark-submit` tendremos que construirlo al inicio (ver referencias).

```
from pyspark.sql import SparkSession

spark = SparkSession.builder.getOrCreate()
sc = spark.sparkContext
```

# Elección del clúster

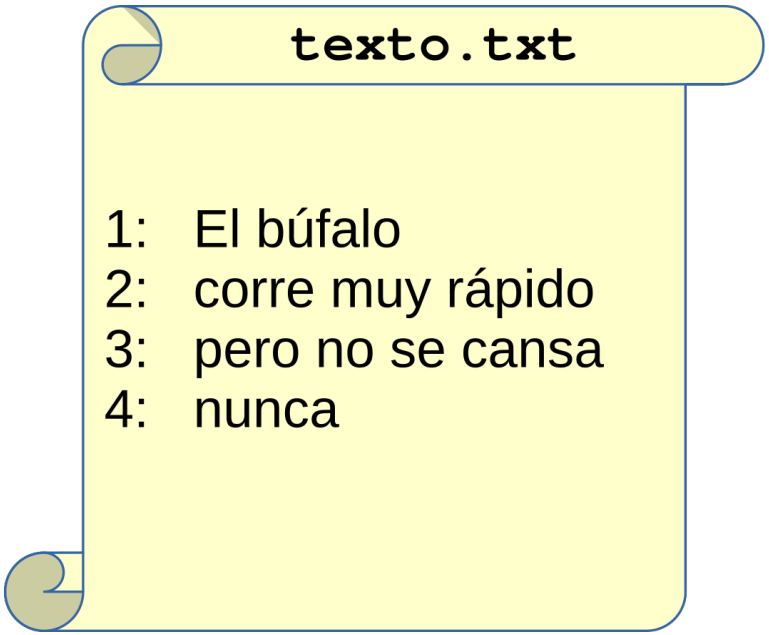
- El lanzador `spark-submit` admite varios parámetros, siendo `master` el más importante pues indica la URL del clúster:
  - `spark://host:port` → Clúster Spark específico
  - `local` → Local con un solo núcleo
  - `local[N]` → Local con N núcleos
  - `local[*]` → Local con tantos núcleos como tenga la máquina
- También es posible configurar Spark sobre Mesos y YARN

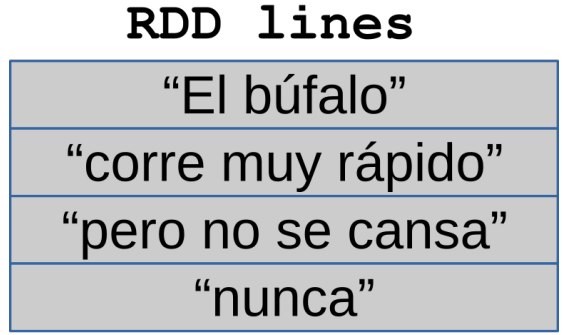
# Creación de RDDs: sc.textFile()

- Crea un RDD con los datos de uno o varios ficheros. Pueden ser locales o remotos (como HDFS)
- Se puede usar un nombre de fichero, de directorio o usar comodines:
  - `lines = sc.textFile("texto.txt")`
  - `lines = sc.textFile("logs/")`
  - `lines = sc.textFile("logs/2015*.log")`

# sc.textFile()

```
lines = sc.textFile("texto.txt")
```

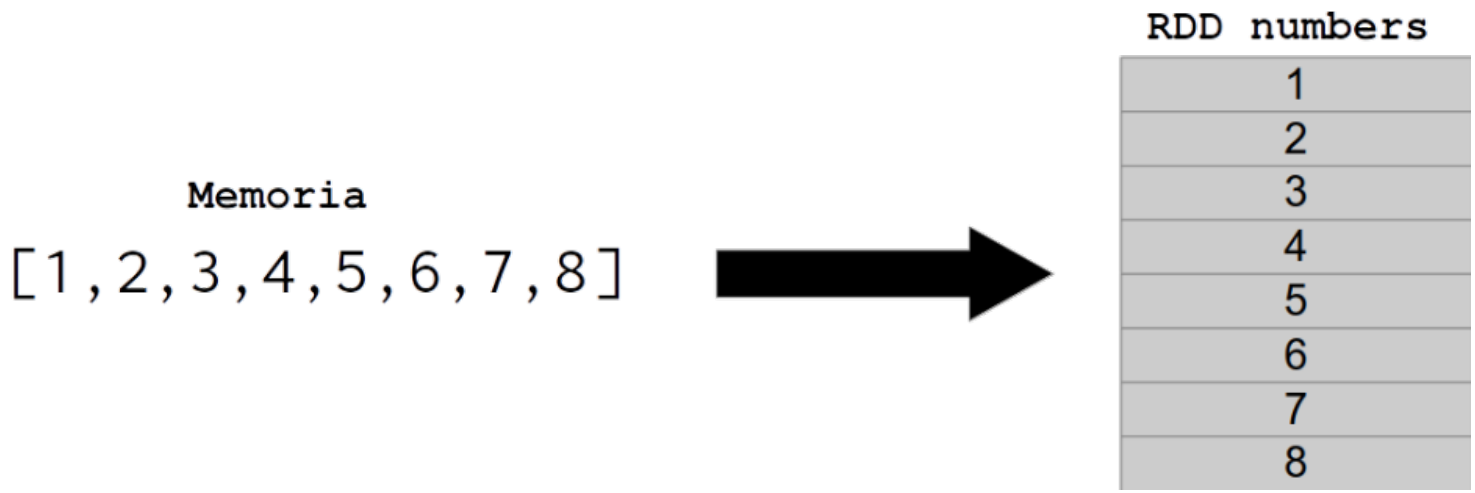


# Creación de RDD: parallelize()

- Crea un RDD a partir de una colección existente en el programa:
  - `sc.parallelize([1, 2, 3, 4])`
  - `sc.parallelize(['Hello', 'Sam']`
  - `sc.parallelize(('soy', 'una', 'tupla'))`
  - `sc.parallelize({1:'hola', 4:'hi'}) # solo incluirá las claves`

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



# Almacenar RDD

- Los RDDs se pueden volcar a disco local o distribuido usando diferentes formatos. Por ejemplo:
  - `saveAsTextFile(path)`
  - `SaveAsNewAPIHadoopDataset(...)`
  - `saveAsPickleFile(path)`
  - `saveAsSequenceFile(path)`
- Estas operaciones son **acciones**, así que lanzan el cómputo del RDD

# Acciones

- Veamos algunas acciones importantes:
  - `count()`
  - `countByValue()`
  - `reduce()`
  - `collect()`
- Consultar la documentación de la clase `pyspark.RDD` para más información.

# collect()

- Devuelve los elementos del RDD como una colección (en Python es una lista)
- Solo debe usarse con RDDs pequeños, ya que los datos deben caber en la memoria del programa *driver*.

```
>>> a = sc.parallelize([1, 2, 3, 4])
>>> a.collect()
[1, 2, 3, 4]
```

# collect()

```
lista = numbers.collect()
```

| RDD numbers |
|---|
| 1 |
| 2 |
| 3 |
| 4 |
| 5 |
| 6 |
| 7 |
| 8 |

**Memoria**

[1,2,3,4,5,6,7,8]

# count()

- Cuenta el número de elementos de un RDD

```
>>> a = sc.parallelize([1,2,3,4])
>>> a.count()
4
```

# countByValue()

- Cuenta el número de veces que aparece cada elemento del RDD
- Devuelve un **diccionario Python**

```
>>> rdd = sc.parallelize(['A', 'B', 'A', 'A'])
>>> rdd.countByValue()
{'A': 3, 'B': 1}
```

# reduce()

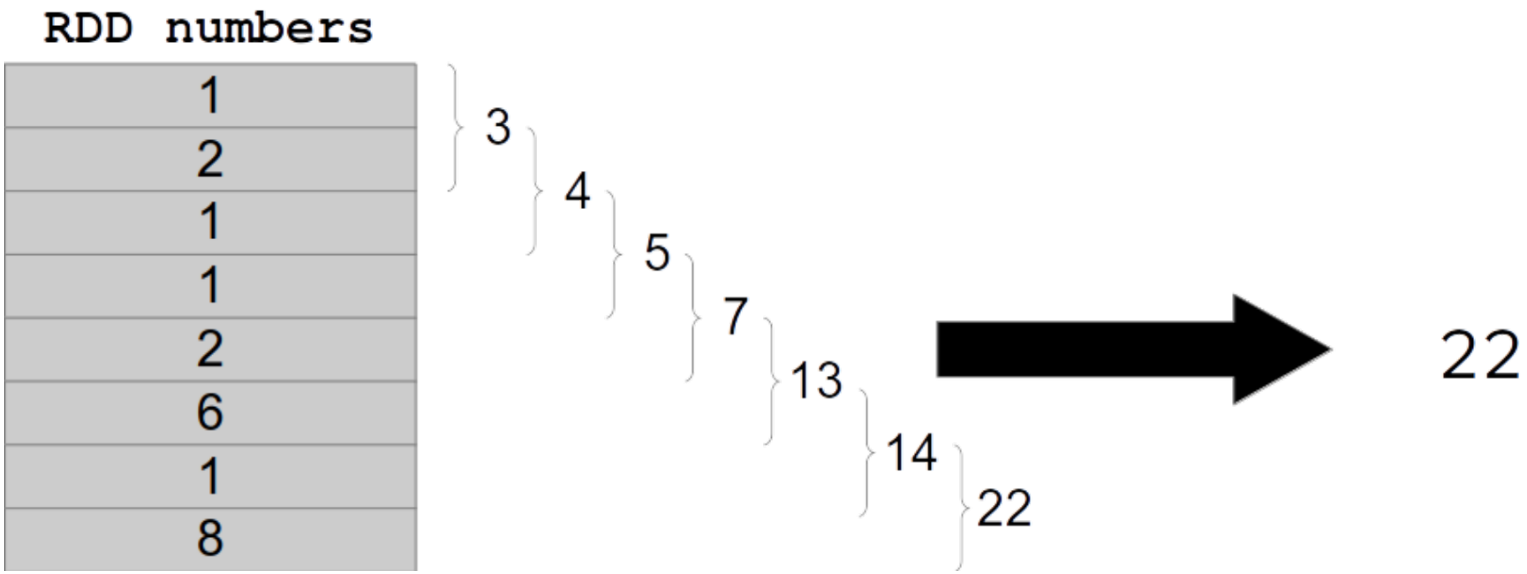
- Combina los elementos del RDD utilizando la función proporcionada

```
>>> rdd = sc.parallelize([1,2,3,4])
>>> rdd.reduce(lambda x,y: x+y)
10
>>> rdd.reduce(lambda x,y: x*y)
24
>>> rdd.reduce(lambda x,y: max(x,y))
4
```

- La función debe ser asociativa y conmutativa, y tiene que producir elementos del mismo tipo que el RDD

# reduce()

```
sum_elem = numbers.reduce(lambda x,y: x+y)
```



# Transformaciones

- Existe un gran número de transformaciones disponibles en los RDD (ver referencias)
- Veremos con detalle algunas de ellas:
  - `filter()`
  - `map()`
  - `flatMap()`
  - `union()`

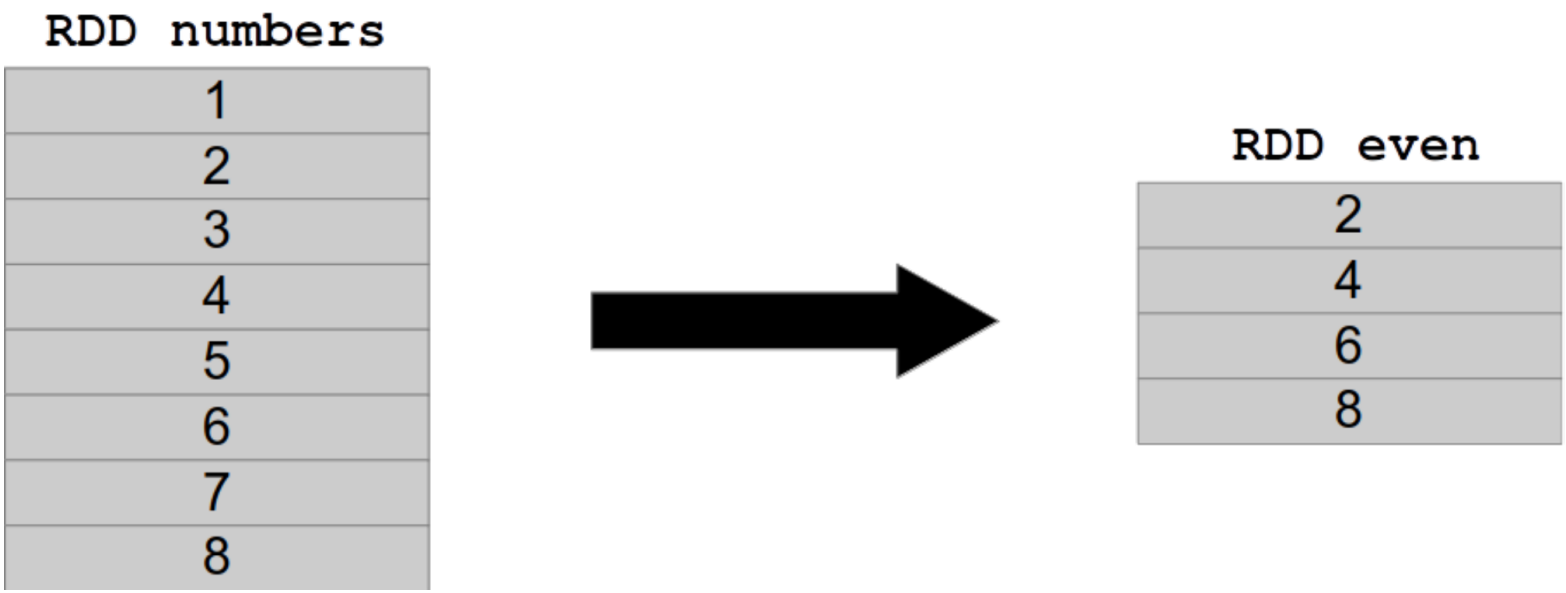
# filter()

- Crea un nuevo RDD únicamente con aquellos elementos que cumplen la propiedad
- La propiedad se representa como una función que toma un elemento y devuelve un `bool`

```
>>> numbers = sc.parallelize([1,2,3,4])
>>> # imaginemos que existe una función es_numero_par(x)
    # que devuelve True cuando x es par
    even = numbers.filter(es_numero_par)
>>> even.collect()
[2,4]
>>> # Ahora con función anónima
    odd = numbers.filter(lambda x: x%2==1)
[1,4]
```

# filter()

```
even = numbers.filter(lambda x: x%2==0)
```



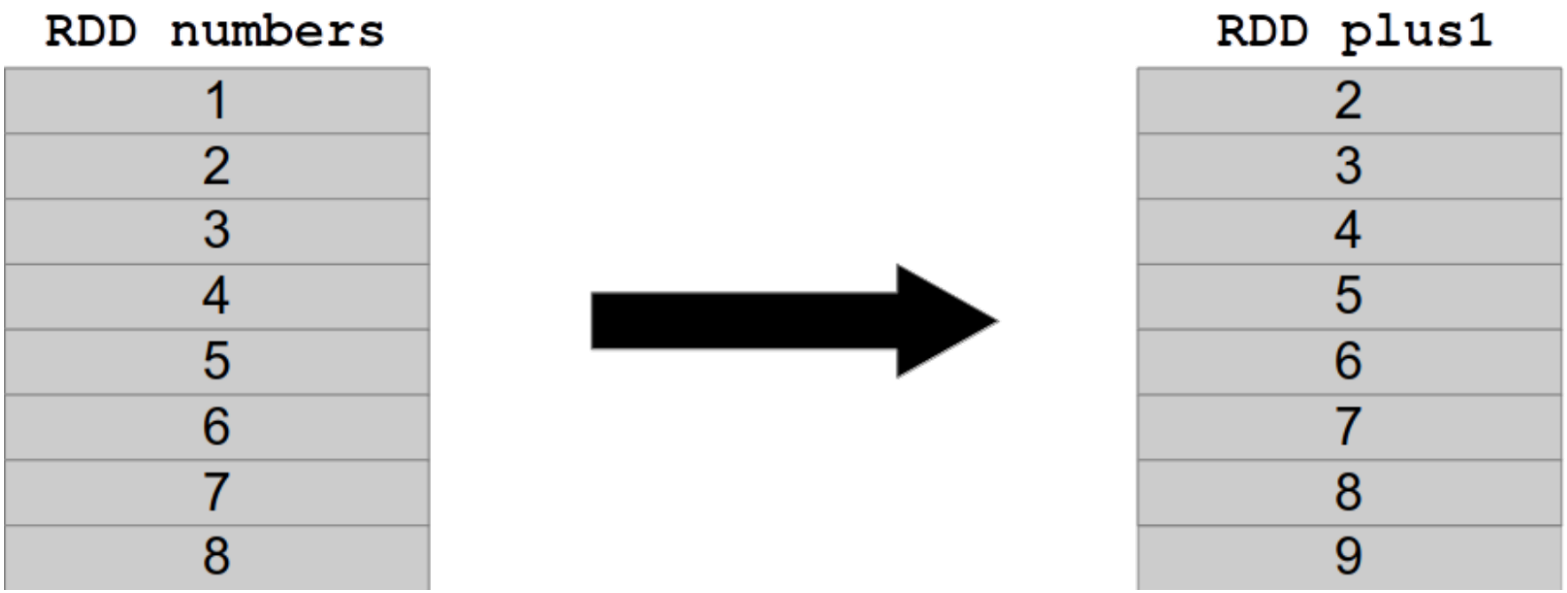
# map()

- Aplica una función a cada elemento del RDD y crea un RDD con todos los resultados

```
>>> numbers = sc.parallelize([1,2,3,4])
>>> plus1 = numbers.map(lambda x: x+1)
>>> plus1.collect()
[2,3,4,5]
```

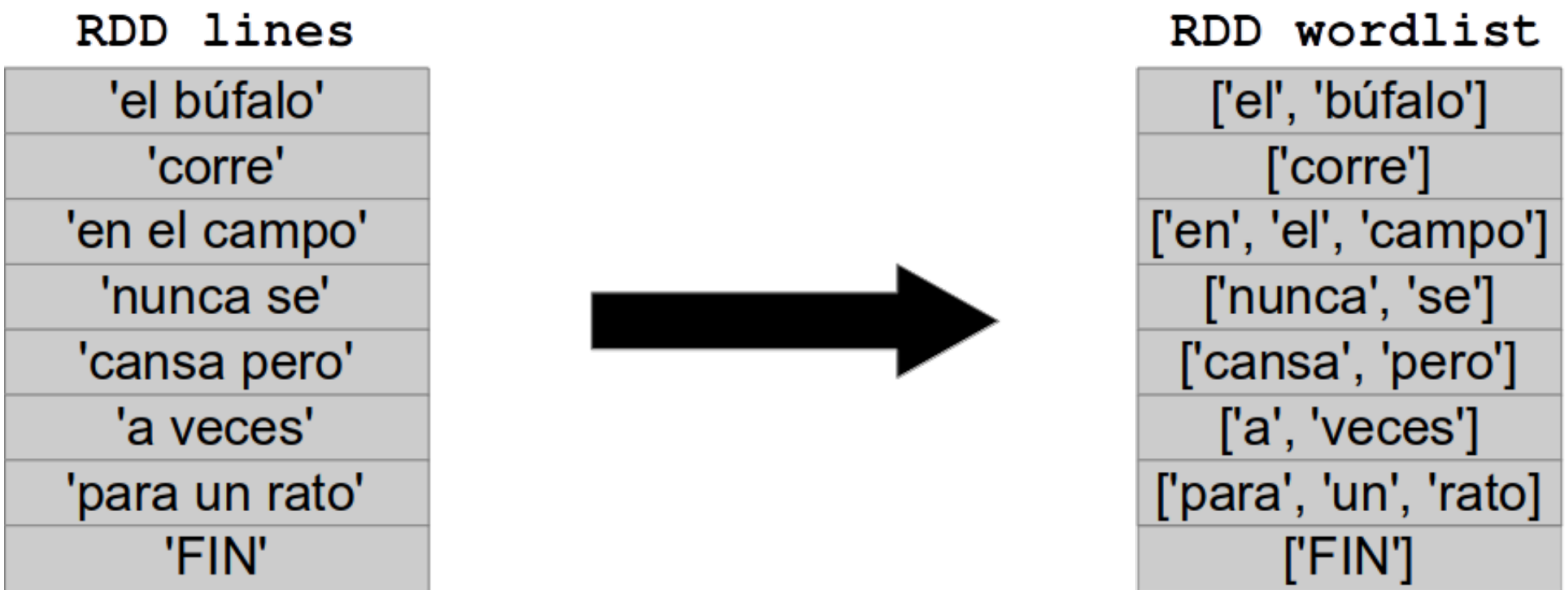
# map()

```
plus1 = numbers.map(lambda x: x+1)
```



# map()

```
wordlist = lines.map(lambda x: x.split())
```



# flatMap()

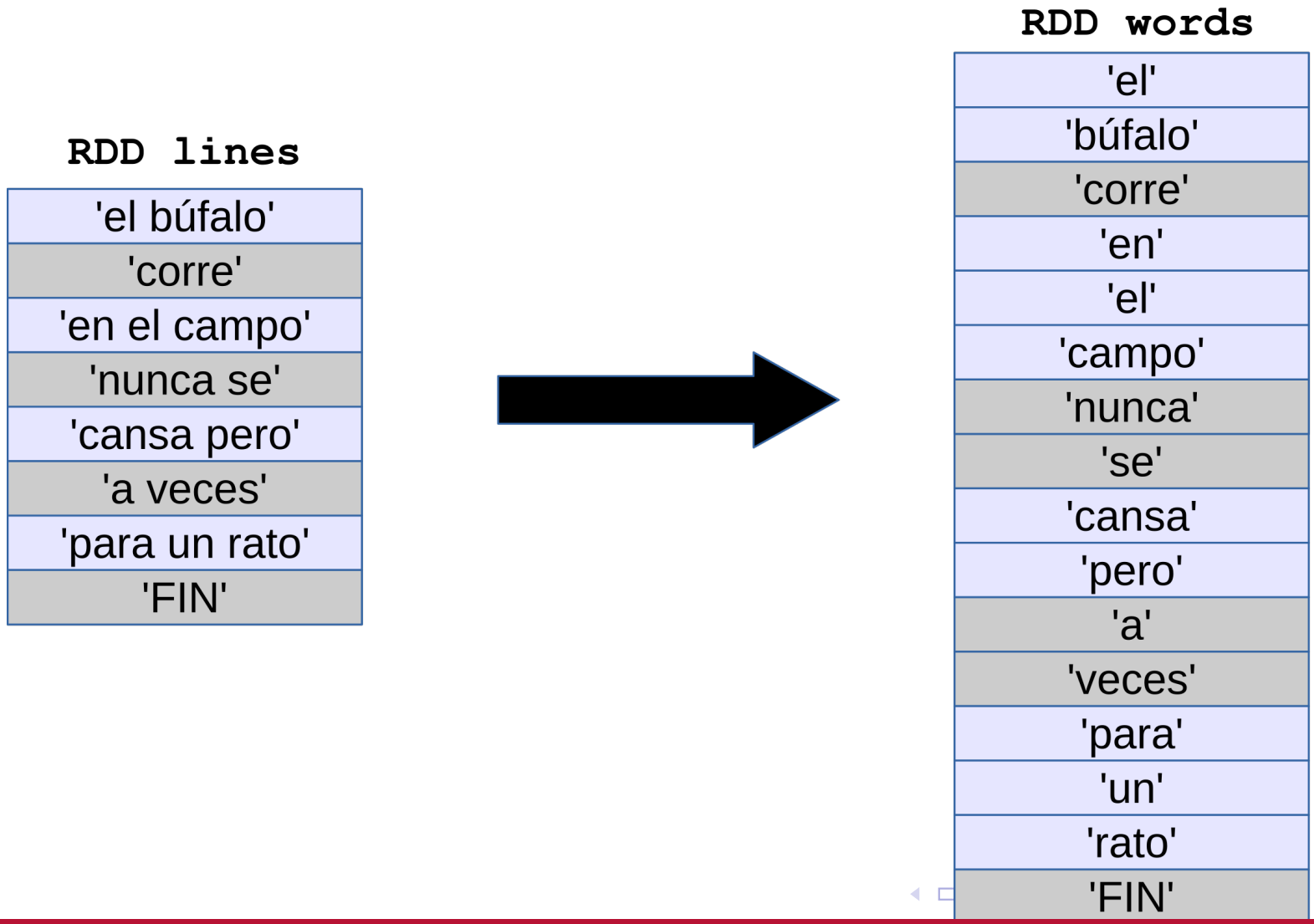
- Aplica una función a cada elemento del RDD para finalmente crear un RDD con el **aplanamiento de todos los resultados**

```
>>> text = ["quien es", "soy yo"]
>>> lines = sc.parallelize(text)
>>> words = lines.flatMap(lambda x:x.split())
>>> words.collect()
['quien', 'es', 'soy', 'yo'] # 4 elementos
```

- **Importante**: la función `lambda x:x.split()` crea listas que normalmente queremos aplanar con `flatMap` si buscamos procesar cada palabra de manera independiente

# flatMap()

```
words = lines.flatMap(lambda x: x.split())
```



# map() frente a flatMap()

- Ambas transformaciones aplican una función a cada elemento del RDD original
- La diferencia está en el aplanado posterior que realiza `flatMap`:

```
>>> text = ["quien es", "soy yo"]
>>> lines = sc.parallelize(text)
>>> words = lines.map(lambda x: x.split())
>>> words.collect()
[['quien', 'es'], ['soy', 'yo']]
# 2 elementos, cada elemento es una lista producida
# por split
```

# union()

- Crea un RDD uniendo los elementos de dos RDDs

```
>>> a = sc.parallelize([0, 2, 4])
>>> b = sc.parallelize([1, 3])
>>> all = a.union(b)
>>> all.collect()
[0, 2, 4, 1, 3]
```

- En Python los elementos de los dos RDD pueden ser de tipos diferentes (p.ej. `int` y `str`). Al usar Spark en otros lenguajes fuertemente tipados como Scala, la unión está restringida a RDD con elementos del mismo tipo

# RDD clave-valor

- Los RDDs clave-valor almacenan parejas (clave, valor) en sus elementos, y Spark nos ofrece algunas transformaciones especiales
- Veremos solo unas pocas:
  - `reduceByKey()`
  - `keys()`
  - `join()`

# reduceByKey()

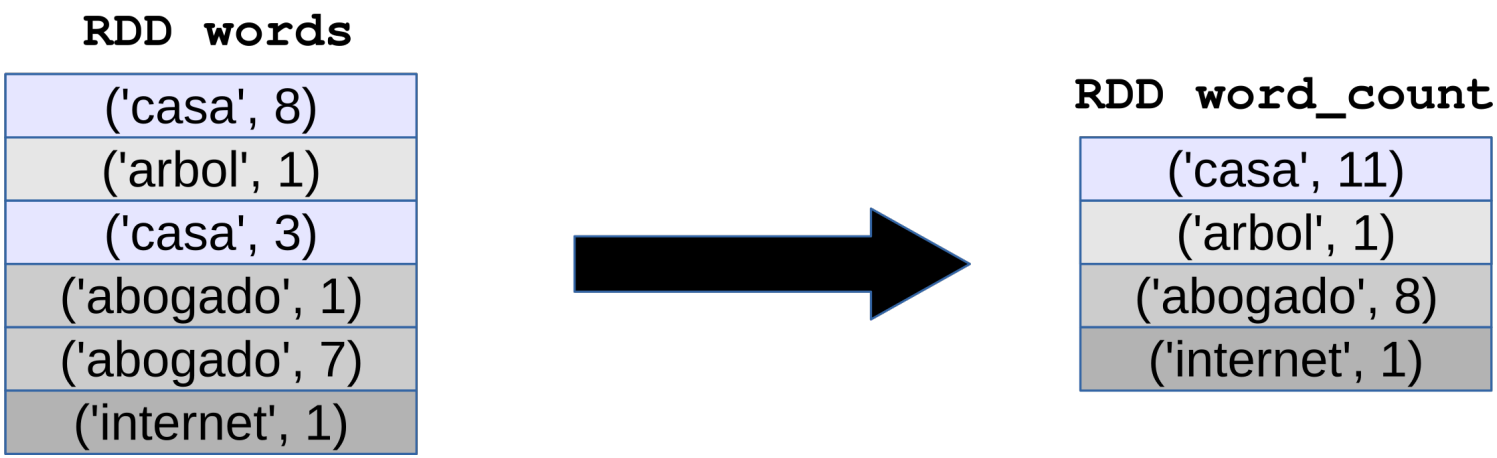
- Combina todos los **valores asociados a la misma clave** usando la **función proporcionada**

```
>>> a = sc.parallelize([(1,2),(3,4),(3,9)]
>>> b = a.reduceByKey(lambda x,y: x+y)
>>> b.collect()
[(1, 2), (3, 13)]
```

- Esta transformación es similar a la fase Reduce de MapReduce

# reduceByKey()

```
word_count = words.reduceByKey(lambda x,y: x+y)
```



# keys()

- Devuelve un RDD con todas las claves
- No elimina duplicados
- Es equivalente a `map(lambda x: x[0])`

```
>>> a = sc.parallelize([('ana',2),('pep',4),('pep',9)])
>>> b = a.keys()
>>> b.collect()
['ana', 'pep', 'pep']
```

# join()

- Combina dos RDDs uniendo los valores de las claves comunes y omitiendo el resto
- Mismo funcionamiento que una reunión intern (*inner join*) en BD relacionales

```
>>> es = sc.parallelize([(1,'hola'), (2,'adiós'),
                         (3,'gracias')])
>>> eng = sc.parallelize([(1,'hi'), (1,'hello'),
                          (2,'bye'), (4,'sorry')])
>>> rdd = es.join(eng)
>>> rdd.collect()
[(1, ('hola', 'hi')), (1, ('hola', 'hello')),
 (2, ('adiós', 'bye'))]
```

- También existen las reuniones externas `rightOuterJoin()`, `leftOuterJoin()` y `fullOuterJoin()`

# join()

```
combined = names.join(age)
```

**RDD names**

| |
|---|
| (1,'pepe') |
| (2,'juan') |
| (3,'eva') |
| (4,'ana') |
| (5,'loli') |

**RDD combined**

| |
|---|
| (1, ('pepe',36)) |
| (1, ('pepe',27)) |
| (5, ('loli',34)) |

**RDD age**

| |
|---|
| (1,36) |
| (1, 27) |
| (5, 34) |

# Ejemplos con pySpark

# Contar apariciones de cada palabra

```
lines = sc.textFile(filename)
counts = (lines.flatMap(lambda x: x.split())
               .map(lambda x: (x, 1))
               .reduceByKey(lambda x,y: x+y)
         )
output = counts.collect()
print(output)
```

# Contar líneas, palabras y caracteres

```
def len_words(l):
# Dada una cadena de texto, suma la longitud de sus palabras
  nchars = 0
  for w in l.split():
    nchars += len(w)
  return nchars


lines = sc.textFile(filename)
nwords = (lines.map(lambda x: len(x.split()))
               .reduce(lambda x,y: x + y)
         )
nchars = (lines.map(len_words)
               .reduce(lambda x,y: x + y)
         )

print('#lines ', lines.count())
print('#words ', nwords)
print('#chars ', nchars)
```

# Ejercicios en Spark

- Filtrar un log para obtener el número de páginas web servidas a navegadores Chrome cada hora de cada día.
- Formato de log (CSV):

```
fecha,hora,recurso,navegador
2012/12/03,10:30,/index.html,Chrome
2012/12/03,17:31,/perro.png,Chrome
2012/12/03,18:59,/index.html,Safari
...
```

- Resultado:

```
2012/12/03-17    1254
2012/12/03-18    58476
2012/12/03-19    258
...
```

# Ejercicios en Spark

- Longitud mínima y máxima de los mensajes de Twitter por horas (de cualquier día)
- Formato del fichero: JSON con un mensaje **en cada línea**:

```
{
  "user": {"name": "pepe", "location": "Magaluf, ES"},
  "date": "2018/08/29",
  "time": "5:25:34",
  "text": "viva el vino @juanpedro!"
}
```

- Resultado:

```
...
9   {min:50, max:140}
10  {min:5, max:120}
11  {min:17, max:110}
...
```

# Ejercicios en Spark

- Índice invertido de menciones en tuits, es decir, asociar a cada mención el conjunto de usuarios que las han realizado
- Formato del fichero: JSON **en cada línea** (como antes):

```
{ "user": {"name": "pepe", "location": "Magaluf, ES"},
  "date": "2018/08/29",
  "time": "5:25:34",
  "text": "viva el vino @juanpedro!"
}
```

- Resultado:

```
@juanpedro  {pepe, ana}
@ana    {pepe,eva,ana,evaristo}
@luis   {eva,evaristo,julian}
...
```

# Referencias

# Referencias

- `pyspark.RDD`
  `https://spark.apache.org/docs/latest/api/python/reference/api/pyspark.RDD.html`
- `pyspark.SparkContext`
  `https://spark.apache.org/docs/latest/api/python/reference/api/pyspark.SparkContext.html`
- `SparkSession`
  `https://spark.apache.org/docs/latest/api/python/reference/pyspark.sql/spark_session.html`