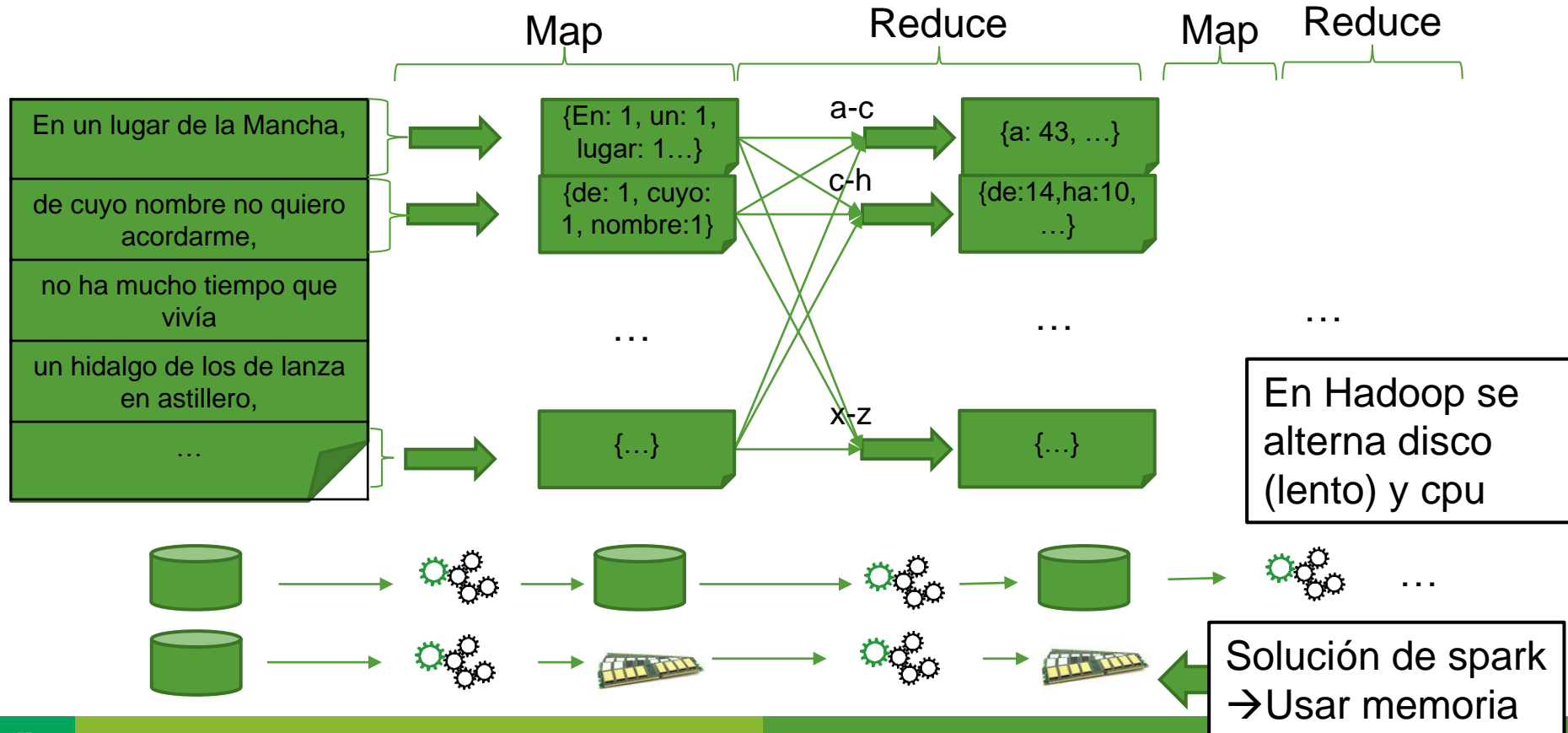
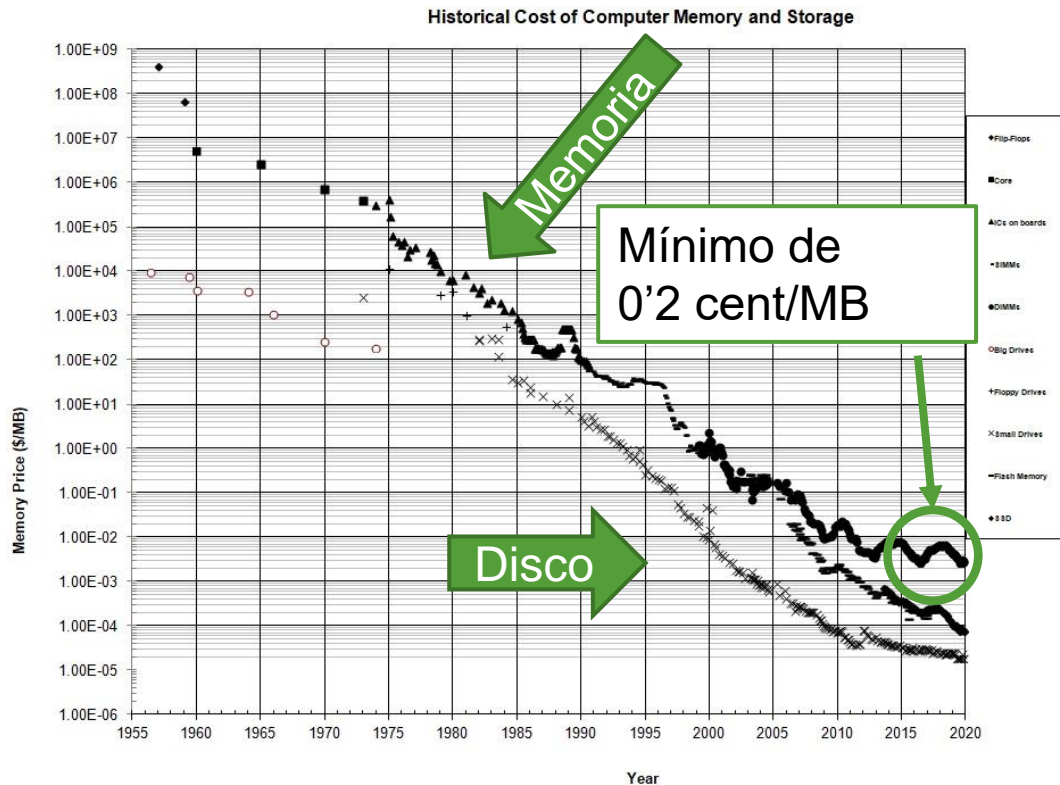


# Introducción a Apache Spark

# Hadoop Map-Reduce. Contar palabras



# Coste de la memoria



<http://www.jcmit.net/mem2015.htm>

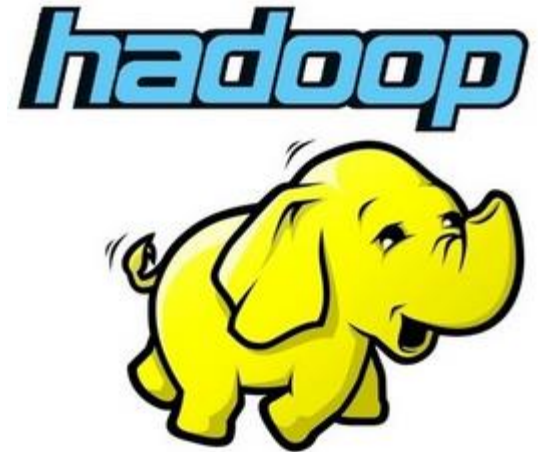
# ¿Qué es Apache Spark?

- Spark es una plataforma de computación para clústers
- Es de propósito general.
- Desarrollo simplificado
- Trabaja en memoria
- Rápido
- Permite trabajo interactivo, streaming...



# En qué mejora Spark a Hadoop

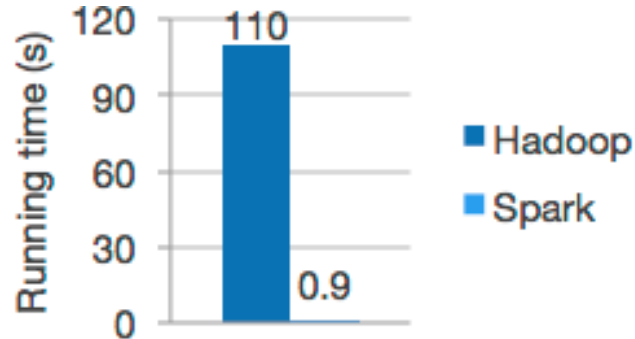
- Velocidad
- Simplicidad del API
- Ejecución Batch, interactiva, streaming vs solo batch en hadoop
- Integra varias herramientas: SQL, grafos, etc.
- Varias APIs: Java, scala, R, python. Hadoop solo java



# Velocidad

- Puede ser hasta 100x más rápido que Hadoop

Logistic regression in Hadoop and Spark



# Ordenando 1PB. Resultados de 2014

	Hadoop MR Record	Spark Record	Spark 1 PB
Data Size	102.5 TB	100 TB	1000 TB
Elapsed Time	72 mins	23 mins	234 mins
# Nodes	2100	206	190
# Cores	50400 physical	6592 virtualized	6080 virtualized
Cluster disk throughput	3150 GB/s (est.)	618 GB/s	570 GB/s
Sort Benchmark Daytona Rules	Yes	Yes	No
Network	dedicated data center, 10Gbps	virtualized (EC2) 10Gbps network	virtualized (EC2) 10Gbps network
<b>Sort rate</b>	<b>1.42 TB/min</b>	<b>4.27 TB/min</b>	<b>4.27 TB/min</b>
<b>Sort rate/node</b>	<b>0.67 GB/min</b>	<b>20.7 GB/min</b>	<b>22.5 GB/min</b>

<https://databricks.com/blog/2014/11/05/spark-officially-sets-a-new-record-in-large-scale-sorting.html>

# Resultados de 2016. Spark primero en Cloud

## Top Results <http://sortbenchmark.org>

	Daytona	Indy
Gray	<p>2016, 44.8 TB/min</p> <p><b>Tencent Sort</b> 100 TB in 134 Seconds 512 nodes x (2 OpenPOWER 10-core POWER8 2.926 GHz, 512 GB memory, 4x Huawei ES3600P V3 1.2TB NVMe SSD, 100Gb Mellanox ConnectX4-EN) Jie Jiang, Lixiong Zheng, Junfeng Pu, Xiong Cheng, Chongqing Zhao Tencent Corporation Mark R. Nutter, Jeremy D. Schaub</p>	<p>2016, 60.7 TB/min</p> <p><b>Tencent Sort</b> 100 TB in 98.8 Seconds 512 nodes x (2 OpenPOWER 10-core POWER8 2.926 GHz, 512 GB memory, 4x Huawei ES3600P V3 1.2TB NVMe SSD, 100Gb Mellanox ConnectX4-EN) Jie Jiang, Lixiong Zheng, Junfeng Pu, Xiong Cheng, Chongqing Zhao Tencent Corporation Mark R. Nutter, Jeremy D. Schaub</p>
Cloud	<p>2016, \$1.44 / TB</p> <p><b>NADSort</b> 100 TB for \$144 394 Alibaba Cloud ECS ecs.n1.large nodes x (Haswell E5-2680 v3, 8 GB memory, 40GB Ultra Cloud Disk, 4x 135GB SSD Cloud Disk) Qian Wang, Rong Gu, Yihua Huang Nanjing University Reynold Xin Databricks Inc. Wei Wu, Jun Song, Junluan Xia Alibaba Group Inc.</p>	<p>2016, \$1.44 / TB</p> <p><b>NADSort</b> 100 TB for \$144 394 Alibaba Cloud ECS ecs.n1.large nodes x (Haswell E5-2680 v3, 8 GB memory, 40GB Ultra Cloud Disk, 4x 135GB SSD Cloud Disk) Qian Wang, Rong Gu, Yihua Huang Nanjing University Reynold Xin Databricks Inc. Wei Wu, Jun Song, Junluan Xia Alibaba Group Inc.</p>



# Simplicidad

## Contar palabras en Hadoop

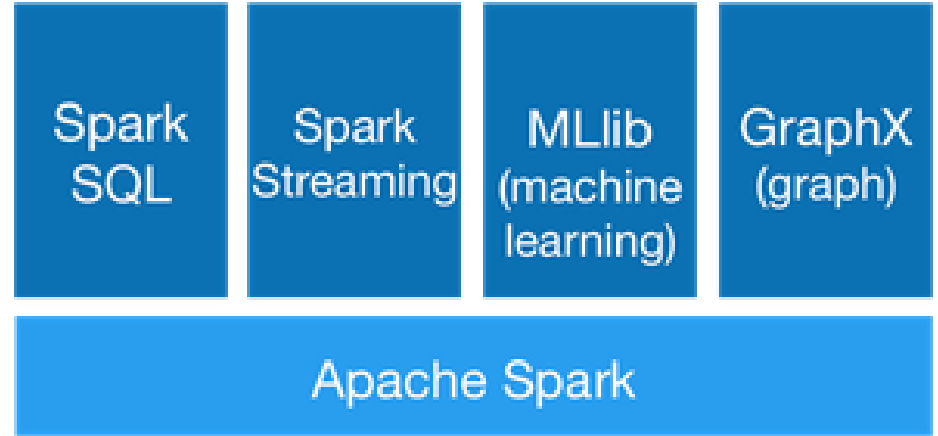
```
1 package org.myorg;
2
3 import java.io.IOException;
4 import java.util.*;
5
6 import org.apache.hadoop.fs.Path;
7 import org.apache.hadoop.conf.*;
8 import org.apache.hadoop.io.*;
9 import org.apache.hadoop.mapreduce.*;
10 import org.apache.hadoop.mapreduce.lib.input.FileInputFormat;
11 import org.apache.hadoop.mapreduce.lib.input.TextInputFormat;
12 import org.apache.hadoop.mapreduce.lib.output.FileOutputFormat;
13 import org.apache.hadoop.mapreduce.lib.output.TextOutputFormat;
14
15 public class WordCount {
16
17     public static class Map extends Mapper<LongWritable, Text, Text, IntWritable> {
18         private final static IntWritable one = new IntWritable(1);
19         private Text word = new Text();
20
21         public void map(LongWritable key, Text value, Context context) throws IOException {
22             String line = value.toString();
23             StringTokenizer tokenizer = new StringTokenizer(line);
24             while (tokenizer.hasMoreTokens()) {
25                 word.set(tokenizer.nextToken());
26                 context.write(word, one);
27             }
28         }
29     }
30
31     public static class Reduce extends Reducer<Text, IntWritable, Text, IntWritable> {
32
33         public void reduce(Text key, Iterable<IntWritable> values, Context context)
34             throws IOException, InterruptedException {
35             int sum = 0;
36             for (IntWritable val : values) {
37                 sum += val.get();
38             }
39             context.write(key, new IntWritable(sum));
40         }
41     }
42
43     public static void main(String[] args) throws Exception {
44         Configuration conf = new Configuration();
45
46         Job job = new Job(conf, "wordcount");
47
48         job.setOutputKeyClass(Text.class);
49         job.setOutputValueClass(IntWritable.class);
50
51         job.setMapperClass(Map.class);
52         job.setReducerClass(Reduce.class);
53
54         job.setInputFormatClass(TextInputFormat.class);
55         job.setOutputFormatClass(TextOutputFormat.class);
56
57         FileInputFormat.addInputPath(job, new Path(args[0]));
58         FileOutputFormat.setOutputPath(job, new Path(args[1]));
59
60         job.waitForCompletion(true);
61     }
62
63 }
```

## Contar palabras en Spark (python)

```
text_file = sc.textFile("hdfs://...")
counts = text_file.flatMap(lambda line: line.split(" ")) \
    .map(lambda word: (word, 1)) \
    .reduceByKey(lambda a, b: a + b)
counts.saveAsTextFile("hdfs://...")
```

# Aplicaciones

- SQL
- Streaming
- GraphX
- MLlib



# Historia

- En 2009 surge dentro de un proyecto de investigación en Berkeley
- La idea era hacer algo rápido para consultas interactivas. De aquí el utilizar datos en memoria
- En 2010 se hace de código abierto
- En 2013 se transfiere a la fundación Apache.
- Spin-off databricks
- Actualmente en versión 3.0



# Arquitectura

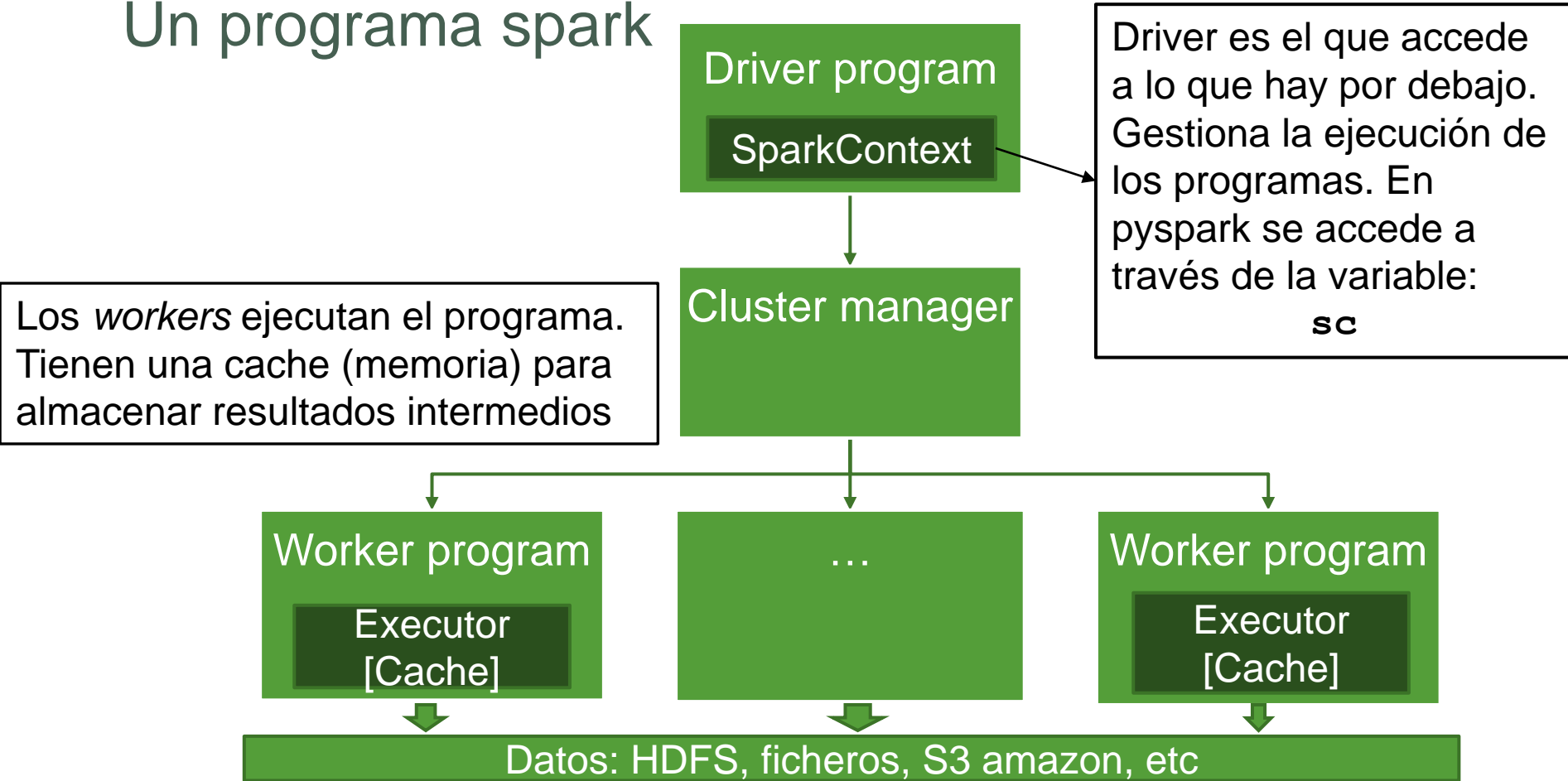
- Muy versátil. Puede trabajar:
  - Standalone.
  - Sobre la nube de Amazon
  - Sobre Hadoop
- Fuentes de datos:
  - Ficheros locales
  - HDFS
  - Cassandra
  - MongoDB
  - Hive
  - postgresQL, mySQL
  - S3 amazon
  - ...



# Aspectos básicos

- **pyspark**: interfaz python a Spark. Nos permite ejecutar tareas en paralelo de forma sencilla.
- A partir de unos datos, se definirá una secuencia de transformaciones y acciones que se ejecutan en paralelo.
- La gestión de la paralelización es transparente para el programador

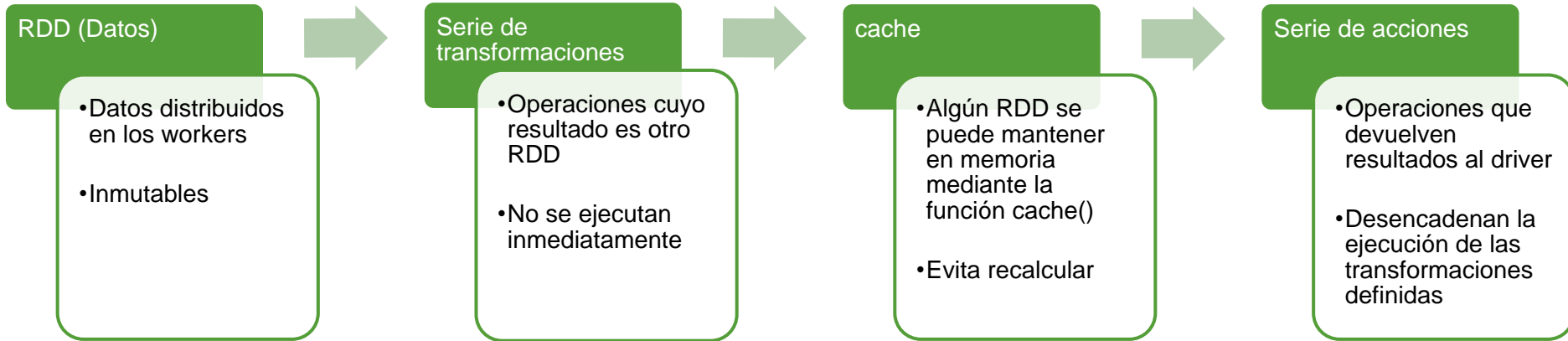
# Un programa spark



# Resilient Distributed Datasets (RDDs)

- Trabajaremos sobre colecciones de datos denominadas RDD:
  - Es el concepto básico de trabajo en Spark
  - Son inmutables. Es decir una vez creados no se pueden modificar.
  - Se pueden transformar para crear nuevos RDDs o realizar acciones sobre ellos pero no modificar.
  - Se guarda la secuencia de transformaciones para poder recuperar RDDs de forma eficiente si alguna máquina se cae
  - Están distribuidos en el clúster en los nodos workers

# Ciclo de vida de una aplicación en Spark





# Recordatorio funciones lambda de python

- Son funciones anónimas. Por ejemplo, para sumar dos números:

```
lambda a, b: a + b
```

- Se pueden usar cuando haya que pasar una función como parámetro
- Tienen una única instrucción cuyo valor corresponde al valor devuelto

# Creación de RDD - textFile

- Crea un RDD a partir del sistema local de archivos, HDFS, Cassandra, HBase, Amazon S3, etc.

```
lineas = sc.textFile('elquijote.txt', 8)
```

Fichero de datos

El número de particiones en que se dividirá el fichero (opcional)

- Las elementos del RDD son cada línea del fichero. Es decir, el RDD será una colección de cadenas
- Evaluación perezosa

# Creación de RDD - textFile

- Otras opciones: Directorio, con comodines, desde fichero comprimido,...:

```
lineas1 = sc.textFile("/my/directory")  
lineas2 = sc.textFile("/my/directory/*.txt")  
lineas3 = sc.textFile("/my/directory/*.gz")
```

- Otros protocolos, HDFS, S3,...:

```
lineas1 = sc.textFile("hdfs://...")  
lineas2 = sc.textFile("s3://...")
```

# Creación de RDD - parallelize

- Crea un RDD a partir de una lista python

```
numeros = sc.parallelize([1,2,3,4,5,6,7,8,9,10], 2)
```

Lista de python.  
Puede ser de  
números,  
cadenas...

Número de  
particiones en que  
se dividirá la lista  
(opcional)

- Evaluación perezosa

# Creación de RDD

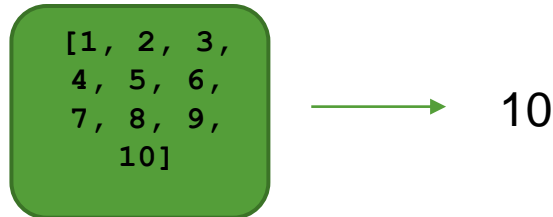
## ➤ Ejemplo evaluación perezosa



```
numeros = sc.parallelize([1,2,3,4,5,6,7,8,9,10])  
  
print(numeros.count())
```

- Spark “apunta” qué va a pasar
- No se calcula nada hasta que es necesario

RDD: numeros



# Transformaciones

- Crean un RDD a partir de otro u otros RDDs
- Evaluación perezosa. No se calculan los resultados inmediatamente. Spark apunta la serie de transformaciones que se deben aplicar para ejecutar después.
- Es como una receta

```
lineas.flatMap(...).filter(...).map(...).reduceByKey(...)
```

- ¿Os había dicho que no se evalúa directamente?  
Evaluación perezosa!!!

# Transformaciones generales

Transformación	Descripción
<code>map(func)</code>	Crea un nuevo RDD a partir de otro aplicando una transformación a cada elemento original
<code>filter(func)</code>	Crea un nuevo RDD a partir de otro manteniendo solo los elementos de la lista original que cumplan una condición
<code>flatMap(func)</code>	Como map pero cada elemento original se puede mapear a 0 o varios elementos de salida
<code>distinct()</code>	Crea un nuevo RDD a partir de otro eliminando duplicados
<code>union(otroRDD)</code>	Une dos RDD en uno
<code>sample()</code>	Obtiene un RDD con una muestra obtenida con reemplazamiento (o sin) a partir de otro RDD.

# Transformación - map

- Aplica una transformación a cada elemento del RDD original

```
numeros = sc.parallelize([1,2,3,4,5])  
  
num3 = numeros.map(lambda elemento: 3*elemento)
```

Función que se aplica a cada elemento del rdd números

- Resultado: [1,2,3,4,5] → [3,6,9,12,15]
- La función que se pasa a map debe:
  - Recibir un único parámetro, que serán elementos individuales del rdd de partida
  - Devolver el elemento transformado



# Transformación – cuestiones sobre el map

- ¿Cuál es el tamaño del rdd de salida?
  - El mismo que el tamaño de entrada



```
palabras = sc.parallelize(['HOLA', 'Que', 'TAL', 'Bien'])  
  
pal_minus = palabras.map(lambda elemento: elemento.lower())  
  
print(pal_minus.collect())
```

RDD: palabras

RDD: pal\_minus

['HOLA',  
'Que',  
'TAL',  
'Bien']



['hola',  
'que',  
'tal',  
'bien']



['hola', 'que', 'tal', 'bien']

# Transformación – cuestiones sobre el map

- ¿Podemos cambiar el tipo de los elementos de los RDDs con un map?
- No, los RDDs son inmutables!! Pero con map podemos crear nuevos RDDs



```
palabras = sc.parallelize(['HOLA', 'Que', 'TAL', 'Bien'])  
  
pal_long = palabras.map(lambda elemento: len(elemento))  
  
print(pal_long.collect())
```

RDD: palabras

['HOLA', 'Que',  
'TAL', 'Bien']

RDD: pal\_long

[4, 3, 3,  
4]



[4, 3, 3, 4]

# Transformación - filter

- Filtra un RDD manteniendo solo los elementos que cumplan una condición

```
numeros = sc.parallelize([1,2,3,4,5])  
  
rdd = numeros.filter(lambda elemento: elemento%2==0)
```

Función que se aplica a cada elemento para filtrarlo

- Resultado: [1,2,3,4,5] → [2,4]
- La función que se pasa a filter debe:
  - Recibir un único parámetro, que serán elementos individuales del rdd de partida
  - Devolver True o False para indicar si el elemento pasa o no el filtro

# Transformación – cuestiones sobre el filter

- ¿Cuál es el tamaño del rdd de salida?
  - Menor o igual que el original



```
log = sc.parallelize(['E: e21', 'W: w12', 'W: w13', 'E: e45'])

errors = log.filter(lambda elemento: elemento[0]=='E')

print(errors.collect())
```

RDD: log

['E: e21',  
'W: w12',  
'W: w13',  
'E: e45']



RDD: errors

['E: e21',  
'E: e45']

El RDD de salida es del mismo tipo que el de entrada



['E: e21', 'E: e45']

# Transformación - flatMap

- Como map pero cada elemento puede crear cero o más elementos


```
numeros = sc.parallelize([1,2,3,4,5])
```

```
rdd = numeros.flatMap(lambda elemento : [elemento, 10*elemento])
```

- Resultado → [1, 10, 2, 20, 3, 30, 4, 40, 5, 50]
- La función que se pasa a flatMap debe:
  - Recibir un único parámetro, que serán elementos individuales del rdd de partida
  - Devolver una lista de elementos

# Transformación – cuestiones sobre el flatMap

- ¿Cuántos elementos tendrá el RDD de salida?
  - Para cada elemento original se crean tantos elementos en el RDD de salida como elementos haya en la lista que devuelve la función



```
lineas = sc.parallelize(['', 'a', 'a b', 'a b c'])  
  
palabras = lineas.flatMap(lambda elemento: elemento.split())  
  
print(palabras.collect())
```

La función split() devuelve una lista con las palabras de una cadena

RDD: lineas

['', 'a', 'a  
b', 'a b c']

RDD: palabras

['a', 'a',  
'b', 'a',  
'b', 'c']



['a', 'a', 'b', 'a', 'b', 'c']

# Transformación – cuestiones sobre el flatMap

## ➤ Diferencias con map

```
lineas = sc.parallelize(['', 'a', 'a b', 'a b c'])  
  
palabras_flat = lineas.flatMap(lambda elemento: elemento.split())  
palabras_map = lineas.map(lambda elemento: elemento.split())
```

- Con flatMap → ['a', 'a', 'b', 'a', 'b', 'c']
- Con map → [[], ['a'], ['a', 'b'], ['a', 'b', 'c']]
- De aquí viene lo de *flat*, la lista de flatmap se ‘alisa’

# Transformación - distinct

- Crea un nuevo RDD eliminando duplicados

```
numeros = sc.parallelize([1,1,2,2,5])  
  
unicos = numeros.distinct()
```

- Resultado: [1,1,2,2,5] → [1, 2, 5]



# Transformación - union

- Une dos RDDs en uno

```
pares = sc.parallelize([2,4,6,8,10])  
impares = sc.parallelize([1,3,5,7,9])  
  
numeros = pares.union(impares)
```

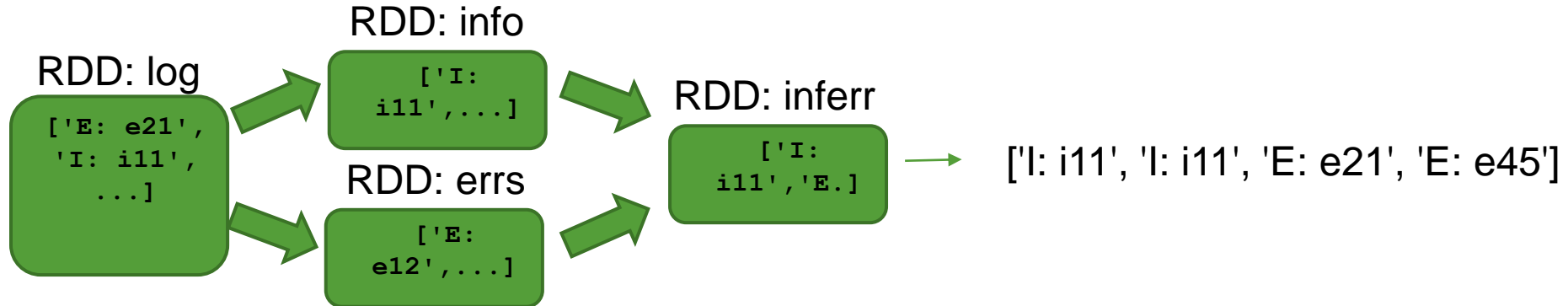
- Resultado: → [2, 4, 6, 8, 10, 1, 3, 5, 7, 9]

# Transformación – union otro ejemplo

```
log = sc.parallelize(['E: e21', 'I: i11', 'W: w12', 'I: i11', 'W: w13', 'E: e45'])

info = log.filter(lambda elemento: elemento[0]=='I')
errs = log.filter(lambda elemento: elemento[0]=='E')
inferr = info.union(errs)

print(inferr.collect())
```



# Transformación - sample

- Remuestrea el RDD de entrada con reemplazamiento o sin.
- El segundo parámetro indica la fracción de datos aproximados que se seleccionan.

```
numeros = sc.parallelize([1,2,3,4,5,6,7,8,9,10])  
  
rdd = numeros.sample(True, 1.0)
```

- Resultado -> [2,3,5,7,7,8,8,9,9,9]
- Cada ejecución da un resultado distinto
- Es útil cuando hay un número de datos demasiado elevado para poder trabajar con menos datos. Al menos en depuración

# Acciones

- Devuelven los resultados al driver program
- Desencadena la ejecución de toda la secuencia de RDD necesarios para calcular lo requerido.
- Ejecuta la receta

```
rdd = lineas.flatMap(...).filter(...).map(...).reduceByKey(...)  
print rdd.count()
```

# Transformación - union

- Une dos RDDs en uno

```
pares = sc.parallelize([2,4,6,8,10])  
impares = sc.parallelize([1,3,5,7,9])  
  
numeros = pares.union(impares)
```

- Resultado: → [2, 4, 6, 8, 10, 1, 3, 5, 7, 9]

# Acciones básicas

Acción	Descripción
<code>count()</code>	Devuelve el número de elementos del RDD
<code>reduce(func)</code>	Agrega los elementos del RDD usando <i>func</i>
<code>take(n)</code>	Devuelve una lista con los primeros n elementos del RDD
<code>collect()</code>	Devuelve una lista con todos los elementos del RDD
<code>takeOrdered(n[,key=func])</code>	Devuelve n elementos en orden ascendente. Opcionalmente se puede especificar la clave de ordenación

# Acción - count

- Devuelve el número de elementos del RDD



```
numeros = sc.parallelize([1,2,3,4,5,6,7,8,9,10])  
  
pares = numeros.filter(lambda elemento: elemento%2==0)  
  
print(pares.count())
```

RDD: numeros

[1, 2, 3,  
4, 5, 6,  
7, 8, 9,  
10]



RDD: pares

[2, 4,  
6, 8,  
10]



5

Debe calcular la secuencia de RDDs para saber cuántos elementos hay

# Acción - reduce

- Agrega todos los elementos del RDD por pares hasta obtener un único valor

```
numeros = sc.parallelize([1,2,3,4,5])  
  
print(numeros.reduce(lambda elem1,elem2: elem1+elem2))
```

- Resultado: → 15
- La función que se pasa a reduce debe:
  - Recibir dos argumentos y devolver uno **de tipo compatible**
  - Ser conmutativa y asociativa de forma que se pueda calcular bien el paralelo



# Acción - reduce

## ➤ Otro ejemplo

```
palabras = sc.parallelize(['HOLA', 'Que', 'TAL', 'Bien'])  
  
pal_minus = palabras.map(lambda elemento: elemento.lower())  
  
print(palabras.reduce(lambda elem1,elem2: elem1+ "-" + elem2))
```

- Resultado: “hola-que-tal-bien”
- ¿Tiene sentido esta operación?
  - No del todo. Aquí ha salido bien pero no es conmutativa
  - ¿Qué pasa si ponemos `elem2+ "-" + elem1`?

# Acción - take

- Devuelve una lista con los primeros n elementos del RDD

```
numeros = sc.parallelize([5,3,2,1,4])  
  
print(numeros.take(3))
```

- Resultado: → [5,3,2]

# Acción - collect

- Devuelve una lista con todos los elementos del RDD

```
numeros = sc.parallelize([5,3,2,1,4])  
  
print(numeros.collect())
```

- Resultado: → [5, 3, 2, 1, 4]
- Cuando se llama a collect todos los datos del RDD se envían al driver program  
¡¡Hay que estar seguros que caben en memoria!!

## Acción - takeOrdered

- Devuelve una lista con los primeros n elementos del RDD en orden

```
numeros = sc.parallelize([3,2,1,4,5])  
  
print(numeros.takeOrdered(3))
```

- Resultado: → [1,2,3]

## Acción - takeOrdered

- También podemos pasar una función para ordenar como creamos

```
numeros = sc.parallelize([3,2,1,4,5])  
  
print(numeros.takeOrdered(3, lambda elem: -elem))
```

- Resultado: → [5,4,3]
- ¿Cómo ordenarías para que primero aparezcan los pares ordenados y luego los impares?

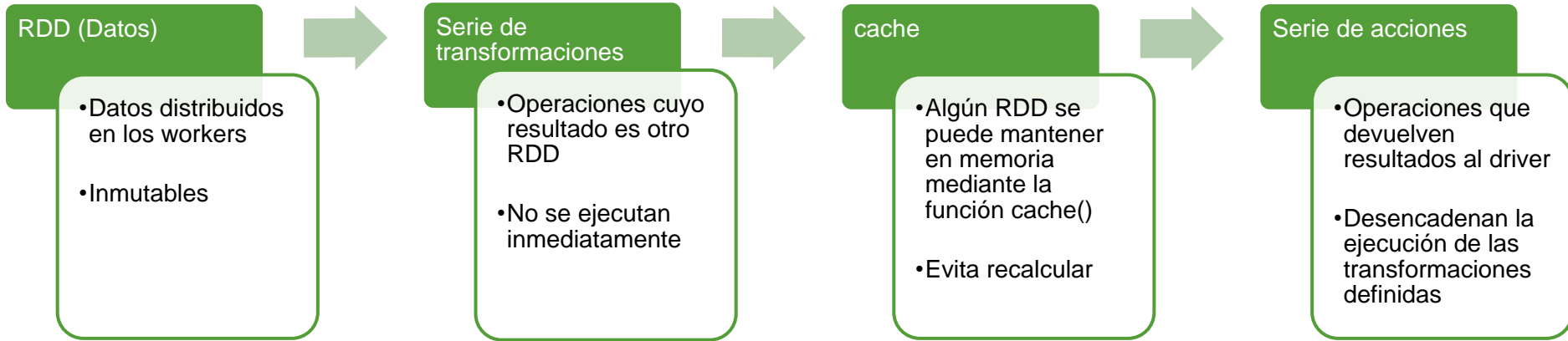
# Acción - foreach

- Ejecuta una función para cada elemento

```
def do_something(a):  
    ...  
  
numeros = sc.parallelize([3,2,1,4,5])  
  
numeros.foreach(do_something)
```

- Es una acción, no una transformación por lo que se ejecuta en el momento
- No devuelve ningún RDD

# Ciclo de vida de una aplicación en Spark



# Errores en spark – parte 1

-----  
Py4JJavaError Traceback (most recent call last)

<ipython-input-13-ea27ccfedbcl> in <module>()  
3 unicos = palabras\_map.distinct()

4  
----> 5 print unicos.collect()

6 # Prueba qué sucede si lo aplicamos a cadenas



Línea donde salta el error. Siempre es una acción aunque viene de alguna transformación previa.

/home/gonzalo/applications/spark/spark-1.4.1/python/pyspark/rdd.pyc

755 """  
756 with SCCallSiteSync(self.context) as css:  
--> 757 port = self.ctx.\_jvm.PythonRDD.collectAndServe(self.\_jrdd.rdd())  
758 return list(\_load\_from\_socket(port, self.\_jrdd\_deserializer))  
759

/home/gonzalo/applications/spark/spark-1.4.1/python/lib/py4j-0.8.2.1-src.zip/py4j/java\_gateway.py in \_\_call\_\_(self, \*args)

536 answer = self.gateway\_client.send\_command(command)  
537 return\_value = get\_return\_value(answer, self.gateway\_client,  
--> 538 self.target\_id, self.name)  
539  
540 for temp\_arg in temp\_args:

/home/gonzalo/applications/spark/spark-1.4.1/python/lib/py4j-0.8.2.1-src.zip/py4j/protocol.py in get\_return\_value(answer, gateway\_client, target\_id, name)

298 raise Py4JJavaError(  
299 'An error occurred while calling {0}{1}{2}.\n'.  
--> 300 format(target\_id, '.', name), value)  
301 else:  
302 raise Py4JError(



# Errores en spark – parte 2

```
Py4JJavaError: An error occurred while calling z:org.apache.spark.api.python.PythonRDD.collectAndServe.  
: org.apache.spark.SparkException: Job aborted due to stage failure: Task 1 in stage 15.0 failed 1 times, most recent failure: Lost task 1.0 in stage 15.0 (TID 113, localhost): org.apache.spark.api.python.PythonException: Traceback (most recent call last):
```

```
File "/home/gonzalo/applications/spark/spark-1.4.1/python/lib/pyspark.zip/pyspark/worker.py", line 111, in main  
    process()  
File "/home/gonzalo/applications/spark/spark-1.4.1/python/lib/pyspark.zip/pyspark/worker.py", line 106, in process  
    serializer.dump_stream(func(split_index, iterator), outfile)  
File "/home/gonzalo/applications/spark/spark-1.4.1/python/pyspark/rdd.py", line 2330, in pipeline_func  
    return func(split, prev_func(split, iterator))  
File "/home/gonzalo/applications/spark/spark-1.4.1/python/pyspark/rdd.py", line 2330, in pipeline_func  
    return func(split, prev_func(split, iterator))  
File "/home/gonzalo/applications/spark/spark-1.4.1/python/pyspark/rdd.py", line 316, in func  
    return f(iterator)  
File "/home/gonzalo/applications/spark/spark-1.4.1/python/pyspark/rdd.py", line 1758, in combineLocally  
    merger.mergeValues(iterator)  
File "/home/gonzalo/applications/spark/spark-1.4.1/python/lib/  
    d[k] = comb(d[k], v) if k in d else creator(v)  
TypeError: unhashable type: 'list'
```

```
at org.apache.spark.api.python.PythonRDD$$anon$1.read(Py  
at org.apache.spark.api.python.PythonRDD$$anon$1.<init>(  
at org.apache.spark.api.python.PythonRDD.compute(PythonRDD.scala:97)  
at org.apache.spark.rdd.RDD.computeOrReadCheckpoint(RDD.scala:277)  
at org.apache.spark.rdd.RDD.iterator(RDD.scala:244)  
at org.apache.spark.api.python.PairwiseRDD.compute(PythonRDD.scala:315)
```

La información del error está sepultada.  
Aparece justo antes del volcado de la  
pila de ejecución

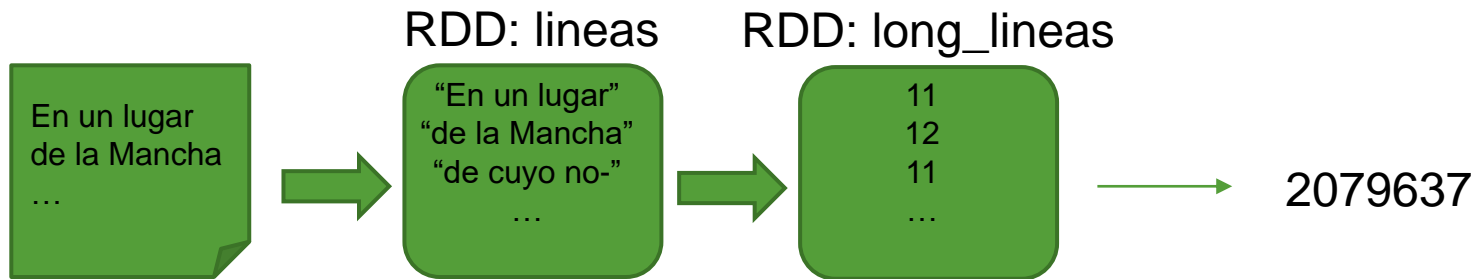
# Ejercicio 1: Contar caracteres de un fichero



```
lineas = sc.textFile('elquijote.txt', 8)

long_lineas = lineas.map(lambda elemento: len(elemento))

print long_lineas.reduce(lambda elem1,elem2: elem1 + elem2)
```



## Ejercicio 2: alturas\_v0.csv

- Objetivo: Calcular la media y la desviación típica de un fichero con alturas.
- Cada fila tiene una altura (en cm)
- Algunas filas tienen errores y pone -100 → hay que filtrarlas
- Algunas filas las alturas están en m → hay que corregirlas
- Herramientas: `textFile`, `map`, `reduce`, `float(str)` (Convierte una cadena a float), `filter` y `count`...

## Ejercicio 3: alturas.csv

- Objetivo: Calcular la media y la desviación típica de un fichero con alturas separadamente para mujeres y hombres.
- Cada fila tiene una genero y altura (en cm)
- Algunas filas tienen errores y pone -100 → hay que filtrarlas
- Algunas filas las alturas están en m → hay que corregirlas
- Herramientas: `textFile`, `map`, `reduce`, `float(str)` (Convierte una cadena a float), `filter`, `count`, `split()`...

# Más transformaciones

Transformación	Descripción
<code>reduceByKey(f)</code>	Al llamarlo sobre un RDD de pares clave-valor (K, V), devuelve otro de pares (K, V) donde los valores de cada clave se han agregado usando la función dada.
<code>groupByKey(f)</code>	Al llamarlo sobre un RDD de pares clave-valor (K, V), devuelve otro de pares (K, seq[V]) donde los valores de cada clave se han convertido a una secuencia.
<code>sortByKey()</code>	Ordena un RDD de pares clave-valor (K, V) por clave.
<code>join(rdd)</code>	Hace un join de dos rdd de pares (K, V1) y (K,V2) y devuelve otro RDD con claves (K, (V1, V2))

## RDD de pares clave-valor (K, V)

- Son RDD donde cada elemento de la colección es una tupla de dos elementos.
  - El primer elemento se interpreta como la clave
  - El segundo como el valor
- Se contruyen a partir de otras transformaciones:.

```
palabras = sc.parallelize(['HOLA', 'Que', 'TAL', 'Bien'])  
  
pal_long = palabras.map(lambda elem: (elem, len(elem)))
```

- Las palabras pasarían a ser las claves y los valores sus longitudes

# Transformación – reduceByKey()

- Agrega todos los elementos del RDD hasta obtener un único valor por clave
- El resultado sigue siendo una colección, esto en un RDD

```
r = sc.parallelize([('A', 1), ('C', 4), ('A', 1), ('B', 1), ('B', 4)])  
rr = r.reduceByKey(lambda v1,v2:v1+v2)  
print(rr.collect())
```

- Resultado: → [('A', 2), ('C', 4), ('B', 5)]
- La función que se pasa a reduce debe (como para reduce):
  - Recibir dos argumentos y devolver uno **de tipo compatible**
  - Ser conmutativa y asociativa de forma que se pueda calcular bien el paralelo
  - A la función se le van a pasar dos valores de elementos con la misma clave

# Transformación – cuestiones sobre el reduceByKey

- ¿De qué tamaño es el RDDs de salida?
  - Igual o menor que el RDD original
  - Exactamente, igual al número de claves distintas en el RDDs original

```
r = sc.parallelize([('A', 1), ('C', 4), ('A', 1), ('B', 1), ('B', 4)])  
rr1 = r.reduceByKey(lambda v1,v2:v1+v2)  
print rr1.collect()  
rr2 = rr1.reduceByKey(lambda v1,v2:v1+v2)  
print(rr2.collect())
```

- Resultado 1: → [('A', 2), ('C', 4), ('B', 5)]
- Resultado 2: → [('A', 2), ('C', 4), ('B', 5)]

Qué pasa si ponemos:  
`lambda v1,v2:'hola'`



# Ejemplo clasico: Contar palabras de un fichero



```
lineas = sc.textFile('elquijote.txt', 8)
pals = (lineas.flatMap(lambda linea: linea.lower().split())
        .map(lambda pal: (pal, 1))
        .reduceByKey(lambda elem1,elem2: elem1 + elem2))
print(pals.collect())
```

En un lugar  
de la Mancha  
...



RDD: lineas

“En un lugar”  
“de la Mancha”  
“de cuyo no-”  
...



RDD: pals

(en,1)  
(un, 1)  
...  
(mancha,1)

Modificalo para: Obtener  
histograma de caracteres y  
obtener la lista ordenada de  
mayor a menor

→ [(En, 1200),..., (mancha,12)]

# Transformación – groupByKey()

- Agrupa todos los elementos del RDD para obtener un único valor por clave con valor igual a la secuencia de valores
- El resultado sigue siendo una colección, esto en un RDD

```
r = sc.parallelize([('A', 1), ('C', 2), ('A', 3), ('B', 4), ('B', 5)])  
rr = r.groupByKey()  
print(rr.collect())
```

- Resultado: → [('A', (1,3)), ('C', (2,)), ('B', (4,5))]
- ¿De qué tamaño es el RDDs de salida?
  - Igual o menor que el RDD original
  - Exactamente, igual al número de claves distintas en el RDDs original
- ¿Qué operación se puede hacer tras un groupByKey para que el resultado sea equivalente a un reduceByKey()? ¿Y simular un group solo con un reduceByKey?

# Transformación – sortByKey()

- Ordena por clave un RDD de pares (K,V)
- Si le pasas False ordena de forma inversa

```
rdd = sc.parallelize([('A',1), ('B',2), ('C',3), ('A',4), ('A',5), ('B',6)])  
res = rdd.sortByKey(False)  
print(res.collect())
```

- Resultado: → [('C', 3), ('B', 2), ('B', 6), ('A', 1), ('A', 4), ('A', 5)]
- Las claves se tienen que poder ordenar

# Transformación – join()

- Realiza una operación join de dos RDD (K,V) y (K,W) por clave para dar un RDD (K,(V,W))

```
rdd1 = sc.parallelize([('A',1), ('B',2), ('C',3)])  
rdd2 = sc.parallelize([('A',4), ('B',5), ('C',6)])  
rddjoin = rdd1.join(rdd2)  
print(rddjoin.collect())
```

- Resultado: → [('A', (1, 4)), ('B', (2, 5)), ('C', (3, 6))]
- Prueba a cambiar las claves y ver cuantos elementos se crean

# Transformación – join()

- El join realiza el producto cartesiano

```
rdd1 = sc.parallelize([('A',1), ('B',2), ('C',3)])  
rdd2 = rdd2 = sc.parallelize([('A',4), ('A',5), ('B',6), ('D',7)])  
rddjoin = rdd1.join(rdd2)  
print(rddjoin.collect())
```

- Resultado: → [('A', (1, 4)), ('A', (1, 5)), ('B', (2, 6))]
- ¿Cuál es el tamaño del RDD de salida?
- Modifica join por leftOuterJoin, rightOuterJoin y fullOuterJoin ¿Qué sucede?

# Operaciones que generan traspiego de datos

- ¿Qué sucede cuando se hace un `reduceByKey`, `join`?
  - Hay que agrupar en un nodo los elementos con una misma clave
- Operaciones como esta generan traspiego (*Shuffle*) de datos.
- Esto puede ser muy costoso, pero es necesario
- Existen dos transformaciones que pueden gestionar/evitar este traspiego: `coalesce()` y `repartition()`

## coalesce(numPartitions)

- Reduce el número de particiones del RDD a numPartitions.
- Es útil para ejecutar operaciones de forma más eficiente por ejemplo después de filtrar un número elevado de datos.
- Evita el traspaso si se reduce el número de particiones
- No obtiene particiones homogéneas en número de datos

## repartition(numPartitions)

- Fuerza un trasiego de datos en el cluster (*Shuffle*).
- Se puede aumentar o reducir igual el número de particiones.
- Las particiones resultantes son de igual tamaño lo que permite ganar posteriormente en velocidad



# cache()

- Como hemos visto las transformaciones son de evaluación perezosa
- Pero además cuando se ejecutan son efímeras, no se guarda nada en memoria

```
textrdd = sc.textFile('ese_fichero_tan_largo.txt')  
print(textrdd.count()) # Desencadena la lectura del fichero  
print(textrdd.count()) # Vuelve a leer el fichero!!
```

- Si ponemos %time delante de los print ¿Qué tiempo de ejecución nos da?

# cache() - consideraciones

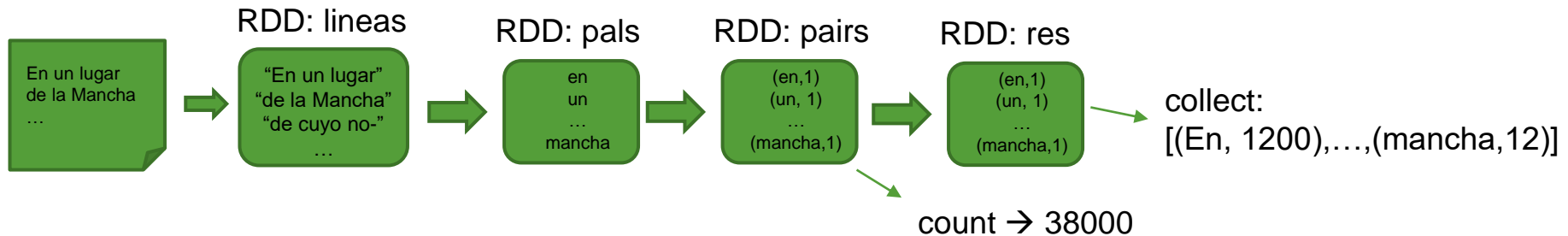
```
rdd = sc.textFile("sensors.txt")  
%time print(rdd.count())  
%time print(rdd.count())  
rdd.cache()  
%time print(rdd.count())  
%time print(rdd.count())
```

- ¿Cuál va a ser la primera línea que use datos en memoria?
- cache() es también de evaluación perezosa
- Solo tiene sentido usarlo si ese rdd se va a usar varias veces

# Ejemplo cache()



```
lineas = sc.textFile('elquijote.txt', 8)
pals   = lineas.flatMap(lambda linea: linea.lower().split())
pairs  = pals.map(lambda pal: (pal, 1))
pairs.cache()
res    = pairs.reduceByKey(lambda elem1,elem2: elem1 + elem2)
print(res.collect())
print(pairs.count())
```



# cache(), persist() y unpersist()

- rdd.persist() asigna un nivel de almacenamiento para el RDD. Sin parámetros funciona como cache() y hace almacenamiento en memoria

```
rdd.persist(StorageLevel)
```

Donde StorageLevel puede valer: MEMORY\_ONLY, DISK\_ONLY, ...

- cache() mantiene en memoria el RDD y usa MEMORY\_ONLY
- Spark, si necesita espacio, elimina automáticamente de memoria los RDDs utilizados hace más tiempo.
- También se puede usar rdd.unpersist() para quitar el RDD de memoria

## Persistencia en fichero: acción saveAsTextFile

- Escribe los elementos de un RDD en uno (o varios) fichero(s) de texto en el directorio del *worker*.
- Cada worker guarda su parte de los datos pero no en el mismo fichero
- Lo que escribes se puede leer mediante textFile

# Persistencia en fichero: acción saveAsTextFile

- Prueba este código y mira qué genera

```
if os.path.isdir('salida') :  
    n2 = sc.textFile('salida').map(lambda a:int(a))  
    print n2.reduce(lambda v1,v2: v1 + v2)  
else:  
    numeros = sc.parallelize(xrange(0,1000))  
    numeros.saveAsTextFile('salida')
```

- Borra la salida y cambia las particiones en parallelize ¿Qué sucede?
- Usa coalesce(1) antes de guardar ¿Qué sucede?

# ¿Qué resultado se obtiene?

```
counter = 0
rdd = sc.textFile('elquijote.txt')

def incrementar(x):
    global counter
    counter += x

rdd.map(lambda l:len(l)).foreach(incrementar)

print("Número de caracteres: {}".format(counter))
```

- → Número de caracteres: 0
- La operación está paralelizada, por lo que habrá un counter por JVM y el counter del driver no se incrementa nunca

# ¿Qué resultado se obtiene?

```
counter = 0
rdd = sc.textFile('elquijote.txt')

def incrementar(x):
    global counter
    counter += x

rdd.map(lambda l:len(l)).foreach(incrementar)

print("Número de caracteres: {}".format(counter))
```

- → Número de caracteres: 0
- La operación está paralelizada, por lo que habrá un counter por JVM y el counter del driver no se incrementa nunca



## Otro ejemplo

```
pals_a_eliminar = ['a', 'ante', 'bajo', 'segun', 'que', 'de']

def elimPalabras(p):
    global pals_a_eliminar
    return p not in pals_a_eliminar

lineas = sc.textFile('elquijote.txt', 8)
pals = (lineas.flatMap(lambda linea: linea.lower().split()).filter(elimPalabras)
        .map(lambda pal: (pal, 1)).reduceByKey(lambda elem1,elem2: elem1 + elem2))

print(pals.takeOrdered(5, key=lambda a:-a[1]))
```

- ¿Qué sucede aquí con pals\_a\_eliminar?

# Closures

- Las funciones que se ejecutan en las transformaciones se pasan a cada nodo junto con las variables necesarias. Esto es un *closure*
- No confundir con los valores propios del RDD que ya están en el nodo correspondiente.
- El closure se serializa y se envía a cada ejecutor.
- Las variables pasan a ser copias como el caso de counter en el ejemplo previo, donde se incrementa la copia local de la variable.

# Variables compartidas

- ¿Cómo hacemos si queremos contar el número de líneas corruptas de un fichero?
  - Variables compartidas de tipo accumulator
- ¿Cómo hacemos si queremos compartir cierta información con todos los *workers*?
  - Variables compartidas de tipo broadcast

## Variables *broadcast*

- Sirven para almacenar variables de lectura en cada *worker*.
- Pueden ser variables o listas de gran tamaño
- Solo se almacenan una vez por *worker*, no por tarea
- Evitan la sobrecarga de la red, que sí sucede si se envían en el *closure*.
- Utilizan algoritmos eficientes para hacer la distribución de la variable

# Broadcast

```
pals_a_eliminar = sc.broadcast(['a', 'ante', 'bajo', 'segun', 'que', 'de'])

def elimPalabras(p):
    return p not in pals_a_eliminar.value

lineas = sc.textFile('elquijote.txt', 8)
pals = (lineas.flatMap(lambda linea: linea.lower().split()).filter(elimPalabras)
        .map(lambda pal: (pal, 1)).reduceByKey(lambda elem1,elem2: elem1 + elem2))

print(pals.takeOrdered(5, key=lambda a:-a[1]))
```

- ¿Qué sucede aquí con pals\_a\_eliminar?

## Variables *accumulators*

- Sirven para acumular valores desde los *workers* al *driver*
- Para los *workers* las variables son de solo escritura
- Solo el *driver* puede leer las variables.

# Ejemplo: Accumulators

```
counter = sc.accumulator(0)
rdd = sc.textFile('elquijote.txt')

def incrementar(x):
    global counter
    counter += x

rdd.map(lambda l:len(l)).foreach(incrementar)

print("Número de caracteres: {}".format(counter.value))
```

➤ → Número de caracteres: 2079636

# Consola de spark

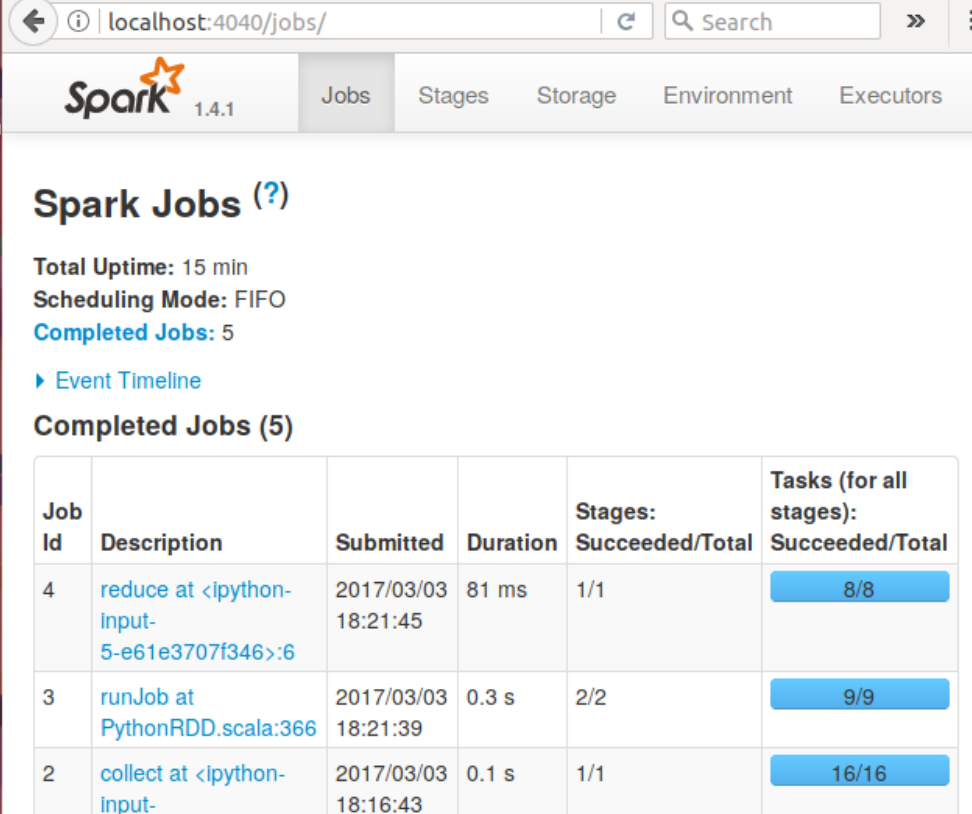
- En el siguiente enlace está la consola de spark para monitorear e inspeccionar los trabajos de spark  
[http://\[driver\]:4040/](http://[driver]:4040/)
- Esta dividida en:
  - jobs: Con el estado de todos los trabajos ejecutados en spark
  - stages: fases en las que se encuentran los trabajos
  - environment: variables del entorno
  - executors: Especifica los procesos que están ejecutando las tareas.
  - ...



# Web UI: jobs

- En el siguiente enlace se pueden ver los trabajos en ejecución y ejecutados.

[http://\[driver\]:4040/jobs](http://[driver]:4040/jobs)



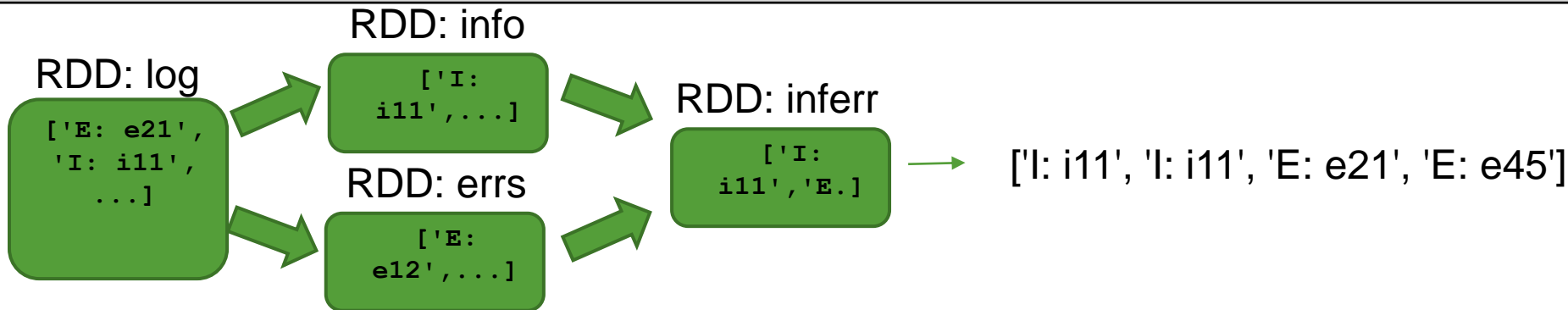
The screenshot shows the Spark Web UI at localhost:4040/jobs/. The interface includes a navigation bar with tabs for Jobs, Stages, Storage, Environment, and Executors. The main content area displays 'Spark Jobs (?)' with summary statistics: Total Uptime: 15 min, Scheduling Mode: FIFO, and Completed Jobs: 5. A link to the Event Timeline is also present. Below this, a table titled 'Completed Jobs (5)' lists the details of the jobs.

Job Id	Description	Submitted	Duration	Stages: Succeeded/Total	Tasks (for all stages): Succeeded/Total
4	<a href="#">reduce at &lt;python-input-5-e61e3707f346&gt;:6</a>	2017/03/03 18:21:45	81 ms	1/1	8/8
3	<a href="#">runJob at PythonRDD.scala:366</a>	2017/03/03 18:21:39	0.3 s	2/2	9/9
2	<a href="#">collect at &lt;python-input-</a>	2017/03/03 18:16:43	0.1 s	1/1	16/16

# Web UI: Stages

- Las transformaciones se pueden representar como un grafo acíclico dirigido

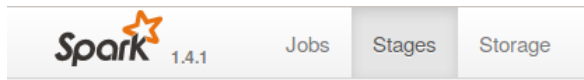
```
log = sc.parallelize(['E: e21', 'I: i11', 'W: w12', 'I: i11', 'W: w13', 'E: e45'])
info = log.filter(lambda elemento: elemento[0]=='I')
errs = log.filter(lambda elemento: elemento[0]=='E')
inferr = info.union(errs)
print inferr.collect()
```



# Web UI: Stages

- En esta pestaña se pueden ver los DAG de las ejecuciones:

[http://\[driver\]:4040/stages](http://[driver]:4040/stages)



## Details for Stage 2 (Attempt 0)

Total Time Across All Tasks: 0.5 s

▼ DAG Visualization

