

Conceptos y Aplicaciones de Big Data

SPARK

Prof. Waldo Hasperué
whasperue@lidi.info.unlp.edu.ar

Temario

Spark

- Arquitectura
- RDDs
- Transformaciones
- Acciones

Spark

Spark nace en 2009 en los laboratorios de la Universidad de California. Desde 2013 pertenece a la fundación Apache.

Spark soporta los dos modos de trabajo en Big Data:

- Consultas en grandes volúmenes de datos (batch processing)
- Procesamiento de streaming

Incluye MLlib (librería de machine learning)

Soporta distintos lenguajes

- Java, Python, R y Scala

Spark

Spark fue desarrollado para que trabaje con el HDFS de hadoop, pero también permite la integración con otros medios de almacenamiento:

- HBase
- Cassandra
- MapR-DB
- MongoDB
- Amazon's S3.

Características de Spark

Simple

- Una gran cantidad funciones (API) desarrolladas para trabajar con grande volúmenes de datos

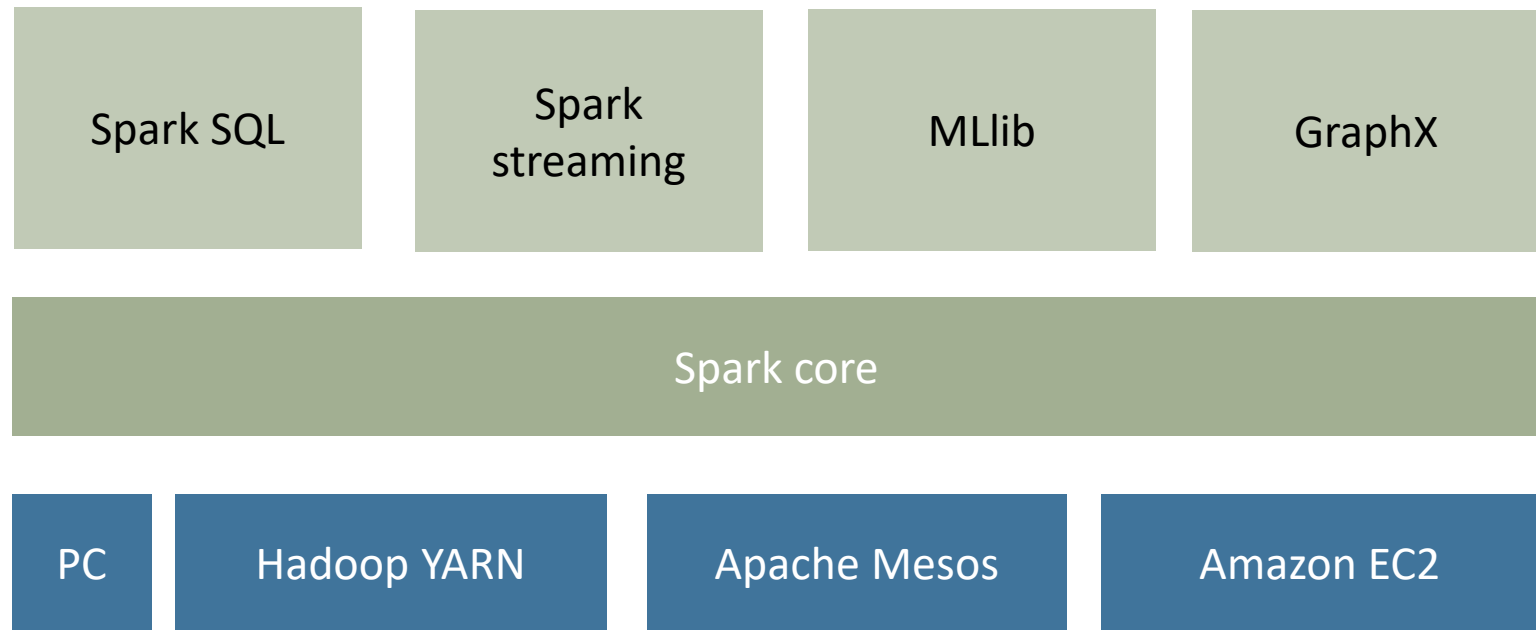
Velocidad

- Optimizado para trabajar en RAM
- Según benchmarks, spark es 100 veces más rápido que Hadoop MapReduce

Soporte

- Permite trabajar en varios lenguajes
- Permite la integración con varios medios de almacenamiento.

Arquitectura de Spark



Arquitectura de Spark

Spark core

- Manejo de funciones, tareas de scheduling

Spark SQL

- Módulo de trabajo con datos estructurados, soporta Hive, conexiones ODBC, JDBC

Spark streaming

- Optimizado para trabajar con Flume (gestor de logs), Kafka (mensajería distribuída)

MLlib

- Librería de machine learning. Tiene algoritmos de clasificación, regresión, clustering

GraphX

- Librería para el análisis sobre grafos de datos

Por lo general una aplicación Spark solo requiere, además de Spark core, una única librería, pero se pueden desarrollar aplicaciones más potentes usando más de una librería.

Resilient Distributed Datasets (RDDs)

Las bases de datos distribuídas y resilientes (RDDs) son parte del núcleo de Spark.

Están diseñadas para el almacenamiento de los datos en la RAM de manera distribuida en un cluster

- Eficiencia: paralelización usando varios nodos; permite el replicado de datos
- Tolerante a fallas: se realiza el tracking de las diferentes operaciones realizadas.

Resilient Distributed Datasets (RDDs)

Son inmutables.

Son divididas en particiones las cuales se distribuyen entre los nodos de un cluster.



Se construyen a partir de la lectura del contenido de un archivo o de una base de datos, como así también mediante la *paralelización* de colecciones de datos.

Spark context

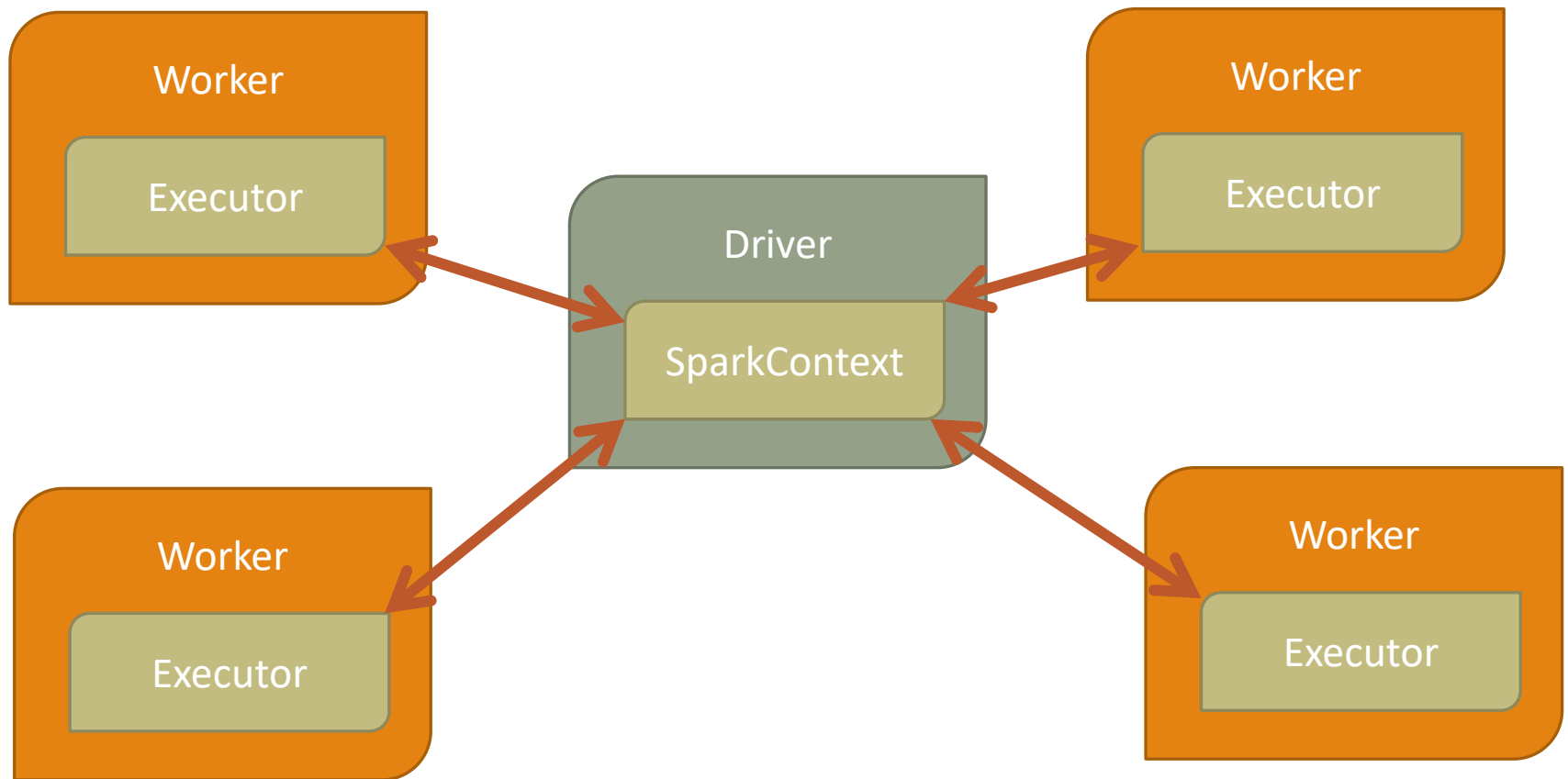
Spark trabaja con el modelo master-slave.

En el master se ejecuta el módulo principal denominado *driver* que es el encargado de enviar las operaciones a realizar sobre las RDDs a cada nodo del cluster.

La conexión con el cluster se realiza mediante un objeto denominado *SparkContext*.

El *SparkContext* crea y maneja las RDDs.

Spark context



Spark shell

De manera nativa las aplicaciones en Spark se programan en Java.

Spark además posee dos shells para trabajar con Python y con Scala.

- Python:
 - Pyspark
 - El shell crea automáticamente un spark context y lo deja disponible en una variable llamada **sc**.
- Scala:
 - spark-shell

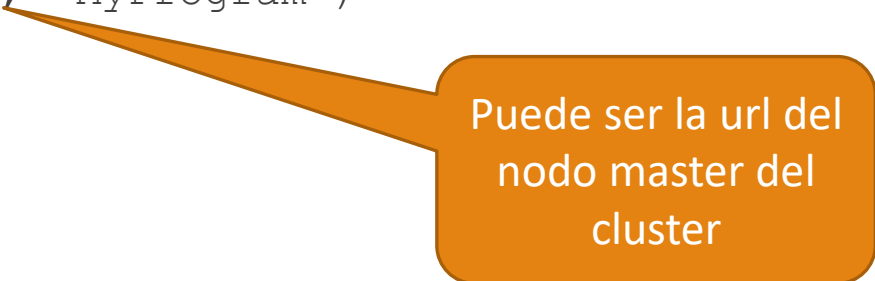
Creando el spark context

Si se ejecuta un script de python fuera del shell entonces se debe incluir la creación del Spark Context de manera explícita:

```
from pyspark import SparkContext  
  
sc = SparkContext("local", "MyProgram")
```

Se puede ejecutar con el comando spark-submit:

```
spark-submit MyProgram.py
```



Puede ser la url del
nodo master del
cluster

Se escribe un script que se ejecuta de manera secuencial


- Las operaciones que llamaremos dentro de ese script se ejecutan en paralelo (transparente para el usuario)

Lectura de archivos


```
clientes = sc.textFile("Cientes")
```

```
cajaDeAhorro = sc.textFile("CajasDeAhorro")
```

```
prestamos = sc.textFile("Prestamos")
```

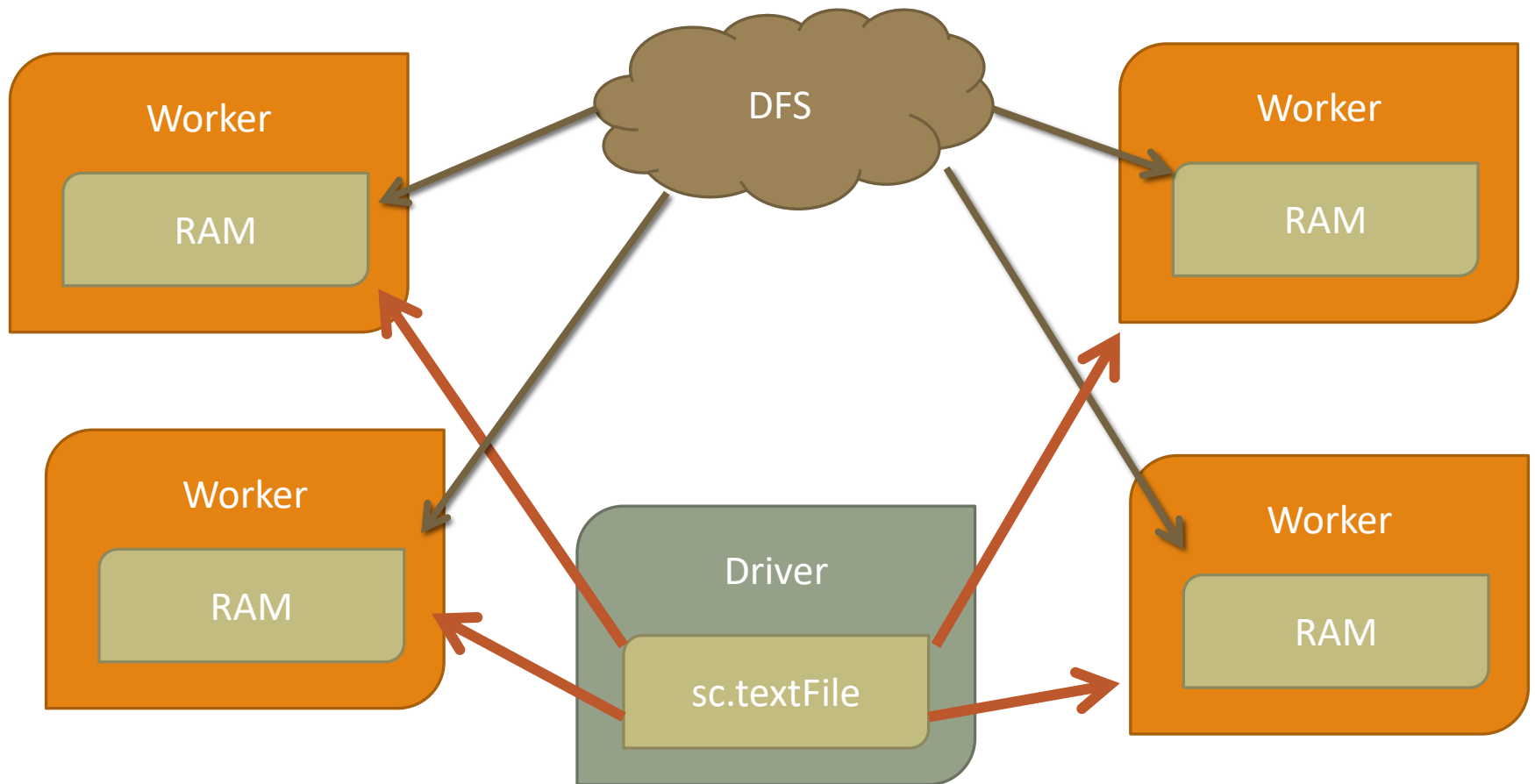


En variables nos guardamos la referencia a la RDD leída



Los directorios pueden ser locales o de un DFS

Lectura de archivos



Lectura de archivos

Worker

RAM

RDD partition

```
<'9750\tPjlkvohq\tBazyv\t20450941\t1959-02-22\tPAR'>  
<'71024\tEiqmq\tNcvsh\t31536997\t1992-07-13\tITA'>  
<'63012\tGcmofvu\tKemwq\t33904855\t1989-09-10\tESP'>  
<'90512\tMceaho\tZcrnf\t35051278\t1983-09-24\tVEN'>  
<'84254\tlxqramo\tQmomhey\t11398586\t1996-10-06\tITA'>  
<'79678\tVosyfur\tTgdzv\t16653090\t1994-07-29\tPAR'>  
<'74595\tMwrzf\tQxzyry\t26642873\t1997-12-02\tECU'>  
<'32199\tOggrw\tNzmildx\t24188150\t1969-06-08\tCOL'>  
<'34942\tRhfbkd\tUzssjspy\t11302880\t1962-07-20\tURU'>  
<'89188\tSbojxtw\tJzylfbab\t28009825\t1953-06-17\tCOL'>
```


Resilient Distributed Datasets (RDDs)

Cargado los datos en una RDD se pueden realizar dos operaciones básicas:

- Transformaciones: cambia el RDD original a través de un proceso de mapeo, filtrado, etc.
- Acciones: tales como el conteo, sumas, los cuales se calculan sobre una RDD sin modificarla

Las transformaciones y acciones son parte de la API de Spark

Transformaciones

Hay distintas funciones que realizan la tarea de transformaciones

- map, filter, flatMap, union, intersection, subtract, cartesian, distinct, sample, ...

Toda transformación se aplica sobre una RDD_i y el resultado es otra RDD_o .

Transformación(RDD^m) \rightarrow RDD^n

Transformación map

La función map se utiliza para realizar transformaciones sobre las tuplas de una RDD.

- También se utilizan para la generación de pares clave-valor

$\text{map}(\text{RDD}^n) \rightarrow \text{RDD}^n$

Recibe como parámetro una función: la que tiene implementada la tarea de que hacer sobre cada tupla

- La función pasada por parámetro recibirá por parámetro una tupla y debe devolver otra tupla como salida:

$\text{fmap}(t_i) \rightarrow t_o$

Transformación map

```
def fmap(tupl):  
    return tupl * 2
```

```
res = rdd.map(fmap)
```


```
res = rdd.map(lambda tupl: tupl*2)
```


Transformación map

< '9750\tPjlkvohq\tBazyv\t20450941\t1959-02-22\tPAR' >


`clientes = clientes.map(lambda line:
 line.split("\t"))`


< ['9750', 'Pjlkvohq', 'Bazyv', '20450941', '1959-02-22', 'PAR'] >


`clientes = clientes.map(lambda t :
 (int(t[0]), t[1] + " " + t[2],
 int(t[3]), t[4], t[5]))`


<(9750, 'Pjlkvohq Bazyv', 20450941, '1959-02-22', 'PAR')>

Transformación filter

La función filter se utiliza para filtrar tuplas en una RDD.

$$\text{filter}(\text{RDD}^n) \rightarrow \text{RDD}^m \quad m \leq n$$

Recibe como parámetro una función que debe devolver un boolean.

- La función pasada por parámetro recibirá por parámetro una tupla y debe devolver True si la tupla "pasa" el filtro o False en caso contrario:

$$\text{ffilter}(t) \rightarrow \text{boolean}$$

Transformación filter

```
def ffilter(t):  
    if(t > 0):  
        return True  
    else:  
        return False
```

```
res = rdd.filter(ffilter)
```


```
res = rdd.filter(lambda t: t > 0)
```

Transformación filter

	<(9750, 'Pjlkvohq 'Bazyv', 20450941, '1959-02-22', 'PAR')> <(1024, 'Eiqmq Ncvsh', 31536997, '1952-07-13', 'ITA')> <(84254, 'Ixqramo Qmomhey', 11398586, '1996-10-06', 'ITA')>	
--	---	--

 `clienITA = clientes.filter(lambda t:`
`t[4] == "ITA")`

	<(1024, 'Eiqmq Ncvsh', 31536997, '1952-07-13', 'ITA')> <(84254, 'Ixqramo Qmomhey', 11398586, '1996-10-06', 'ITA')>	
--	---	--

 `clienMayor = clientes.filter(lambda t:`
`int(t[3][0:4]) < 1960)`

	<(9750, 'Pjlkvohq 'Bazyv', 20450941, '1959-02-22', 'PAR')> <(1024, 'Eiqmq Ncvsh', 31536997, '1952-07-13', 'ITA')>	
--	--	--

Transformación union

La función union se utiliza para realizar la unión entre dos RDDs.

$$\text{union}(\text{RDD}^n, \text{RDD}^m) \rightarrow \text{RDD}^{n+m}$$

Se unen todas las tuplas, sin importar que haya repetidos
(NO es la unión de conjuntos)

Transformación union

	<(1024, 'Eiqmq Ncvsh', 31536997, '1952-07-13', 'ITA')> <(84254, 'Ixqramo Qmomhey', 11398586, '1996-10-06', 'ITA')>	
--	---	--

	<(9750, 'Pjlkvohq 'Bazyv', 20450941, '1959-02-22', 'PAR')> <(1024, 'Eiqmq Ncvsh', 31536997, '1952-07-13', 'ITA')>	
--	--	--

`cliens = clienITA.union(clienMayor)`

Están todas las operaciones de conjuntos: union, intersection, substract, cartesian

	<(1024, 'Eiqmq Ncvsh', 31536997, '1952-07-13', 'ITA')> <(84254, 'Ixqramo Qmomhey', 11398586, '1996-10-06', 'ITA')> <(9750, 'Pjlkvohq 'Bazyv', 20450941, '1959-02-22', 'PAR')> <(1024, 'Eiqmq Ncvsh', 31536997, '1952-07-13', 'ITA')>	
--	---	--

Transformación distinct

La función `distinct` se utiliza para eliminar tuplas duplicadas dentro de una RDD.

$$\text{distinct}(\text{RDD}^n) \rightarrow \text{RDD}^m \quad m \leq n$$

La comparación para eliminar tuplas repetidas es la del propio lenguaje

Transformación distinct

	<p><(1024, 'Eiqmq Ncvsh', 31536997, '1952-07-13', 'ITA')> <(84254, 'Ixqramo Qmomhey', 11398586, '1996-10-06', 'ITA')> <(9750, 'Pjlkvohq 'Bazyv', 20450941, '1959-02-22', 'PAR')> <(1024, 'Eiqmq Ncvsh', 31536997, '1952-07-13', 'ITA')></p>	
--	---	--



```
cliens = cliens.distinct()
```



	<p><(1024, 'Eiqmq Ncvsh', 31536997, '1952-07-13', 'ITA')> <(84254, 'Ixqramo Qmomhey', 11398586, '1996-10-06', 'ITA')> <(9750, 'Pjlkvohq 'Bazyv', 20450941, '1959-02-22', 'PAR')></p>	
--	--	--

Spark el "perezoso"

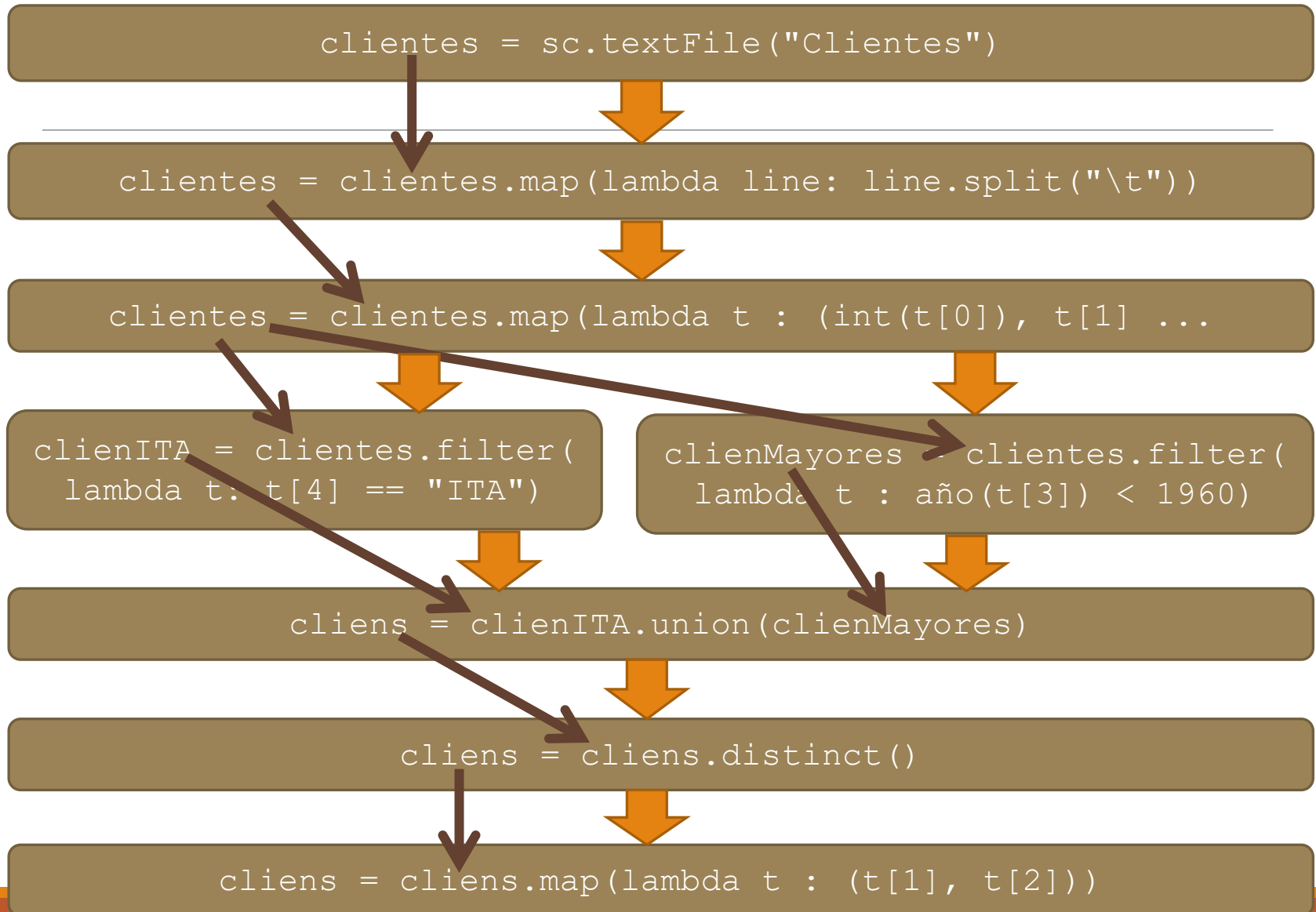
```
from pyspark import SparkContext

sc = SparkContext("local", "MyProgram")

clientes = sc.textFile("Clientes")
clientes = clientes.map(lambda line: line.split("\t"))
clientes = clientes.map(lambda t: (int(t[0]),
                                   t[1] + " " + t[2], int(t[3]), t[4], t[5]))
clienITA = clientes.filter(lambda t: t[4] == "ITA")
clienMayores = clientes.filter(lambda t:
                                int(t[3][0:4]) < 1960)

cliens = clienITA.union(clienMayores)
cliens = cliens.distinct()
cliens = cliens.map(lambda t: (t[1], t[2]))
```

DAG (RDD lineage)



DAG (RDD lineage)

```
clientes = sc.textFile("Clientes")
```

```
clientes = clientes.map(lambda line: line.split("\t"))
```

```
clientes = clientes.map(lambda t : (int(t[0]), t[1] ...
```

```
clienITA = clientes.filter(  
    lambda t: t[4] == "ITA")
```

```
clienMayores = clientes.filter(  
    lambda t : año(t[3]) < 1960)
```

```
cliens = clienITA.union(clienMayores)
```

```
cliens = cliens.distinct()
```

```
cliens = cliens.map(lambda t : (t[1], t[2]))
```

Ejecución de un DAG

Las transformaciones van armando el DAG vinculando las operaciones entre las diferentes RDD que son resultado de tales transformaciones

El DAG (todas las transformaciones) se ejecuta al momento de invocar una acción.

Al ejecutar una acción *SparkContext*, invoca a un proceso interno llamado *DAGScheduler* quien se encarga de transformar el DAG en un plan de ejecución físico y distribuirlo a todos los nodos del cluster.

Acciones

Hay distintas funciones que realizan las acciones

- count, first, take, top, collect, reduce, ...

Toda acción se aplica sobre una RDD y el resultado es un valor que es, o bien devuelto al driver, o bien guardar la RDD en un medio de almacenamiento.

Acción(RDD) ➔ valor al driver / almacenar RDD

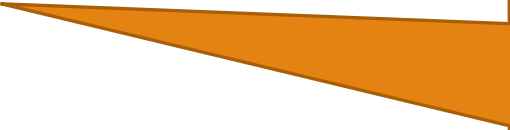
Uso: `rdd.action()`

Acciones

`cliens.count()`

`cliens.first()`

`cliens.take(5)`



Muestra aleatoria, toma
datos de cualquier
partición (minimizando
el número de particiones
a las que se accede)

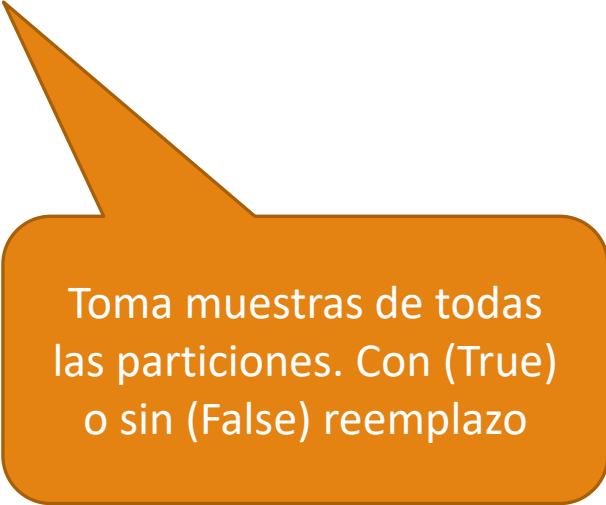
Acciones

```
cliens.count()
```

```
cliens.first()
```

```
cliens.take(5)
```

```
cliens.takeSample(True, 100)
```



Toma muestras de todas las particiones. Con (True) o sin (False) reemplazo

Acciones

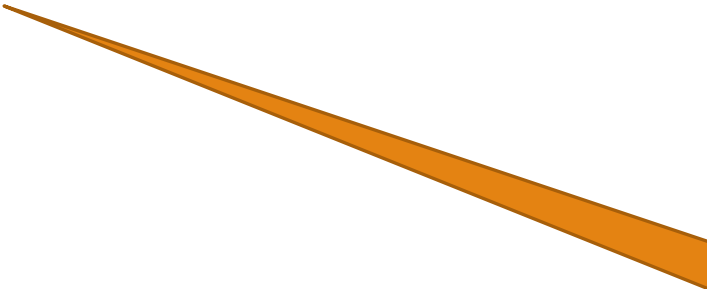
```
cliens.count()
```

```
cliens.first()
```

```
cliens.take(5)
```

```
cliens.takeSample(True, 100)
```

```
cliens.top(5)
```



Incluye una
ordenación interna

Acciones

`cliens.count()`

`cliens.first()`

`cliens.take(5)`

`cliens.takeSample(True, 100)`

`cliens.top(5)`

`cliens.collect()`

Devuelve todo el contenido de la RDD para trabajarlo en el driver.

OJO: Podría ocupar mucha memoria.

Acciones

```
cliens.count()
```

```
cliens.first()
```

```
cliens.take(5)
```

```
cliens.takeSample(True, 100)
```

```
cliens.top(5)
```

```
cliens.collect()
```

```
cliens.saveAsTextFile("output")
```



Directorio local o del DFS.

Acción reduce

La acción reduce se utiliza para reducir/resumir una RDD.

- La reducción de una RDD se hace por pares de tuplas.

Recibe como parámetro una función: la que tiene implementada la tarea de como reducir la RDD.

- La función pasada por parámetro recibe dos tuplas por parámetro y debe devolver otra:

$$\text{freduction}(t_1, t_2) \rightarrow t_3$$

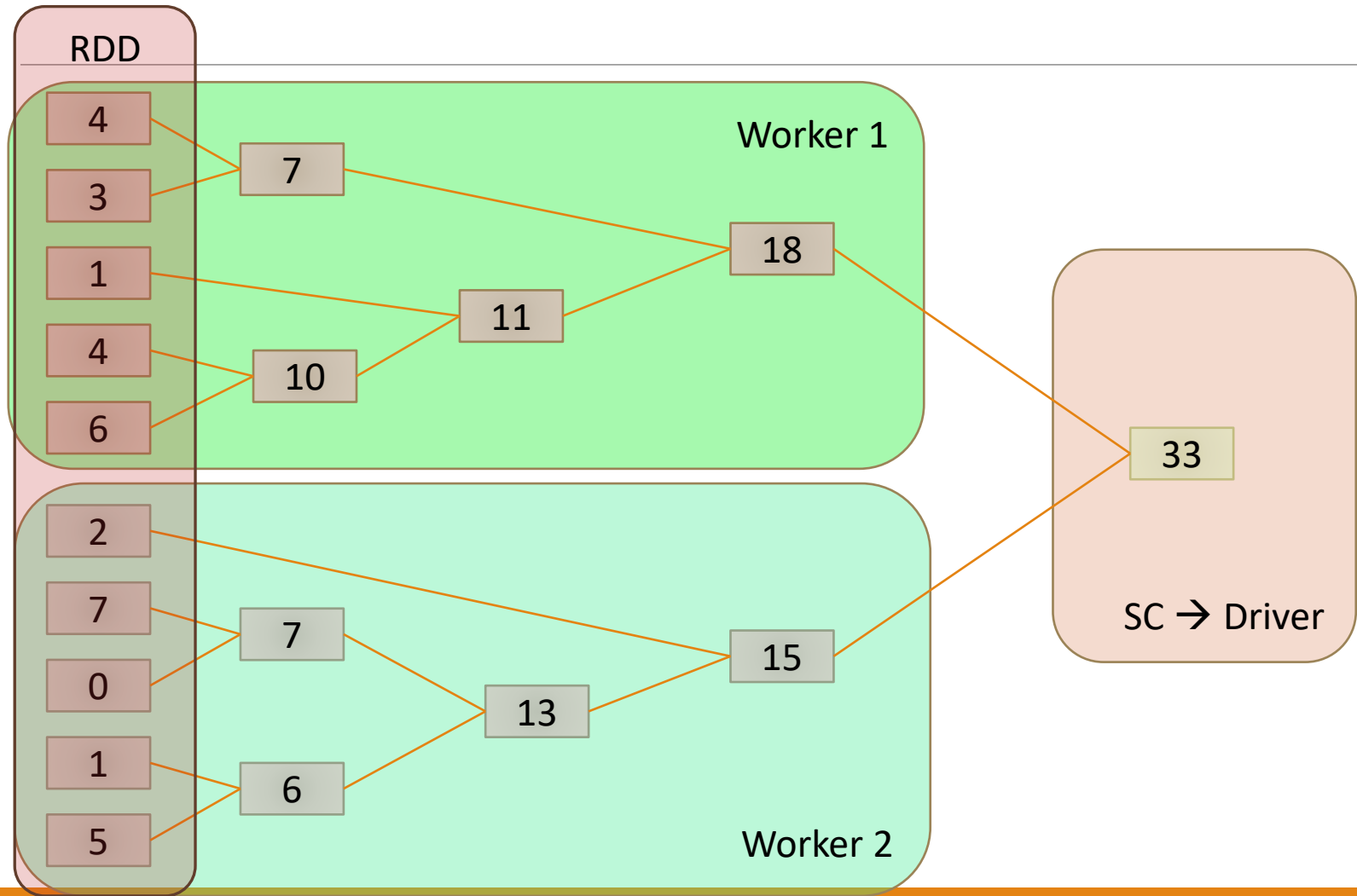
Acción reduce

```
def freduction(t1, t2):  
    return t1 + t2
```

```
total = rdd.reduce(freduction)
```

```
total = rdd.reduce(lambda t1, t2: t1 + t2)
```


Acción reduce



Acción reduce

Cálculo del máximo en RDDs de un campo

RDD
5
8
2
5
1

```
maximo = rdd.reduce(lambda t1, t2:  
                    t1 if t1 > t2 else t2)
```

Acción reduce

Cálculo del máximo en RDDs de más de un campo

R	5
S	8
K	2
X	5
Z	1

```
maximo = rdd.reduce(lambda t1, t2:  
    t1 if t1[1] > t2[1] else t2)
```

Se debe devolver el mismo
tipo de tupla (dos campos
(string, número))

La comparación se hace
por un campo

Acción reduce

Cálculo del máximo en más de un campo

R	5	3	0
S	8	7	2
K	2	1	6
X	5	8	3
Z	1	9	7

```
maximos = rdd.reduce(lambda t1, t2:
    ( "",
      t1[1] if t1[1] > t2[1] else t2[1],
      t1[2] if t1[2] > t2[2] else t2[2],
      t1[3] if t1[3] > t2[3] else t2[3] ) )
```

Se debe devolver el mismo tipo de tupla (cuatro campos (string, número, número, número))