

Conceptos y Aplicaciones de Big Data

SPARK

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

Temario

Spark

- Acumuladores
- Variables broadcast
- Parallelize
- Más transformaciones

Algoritmos iterativos sobre Spark

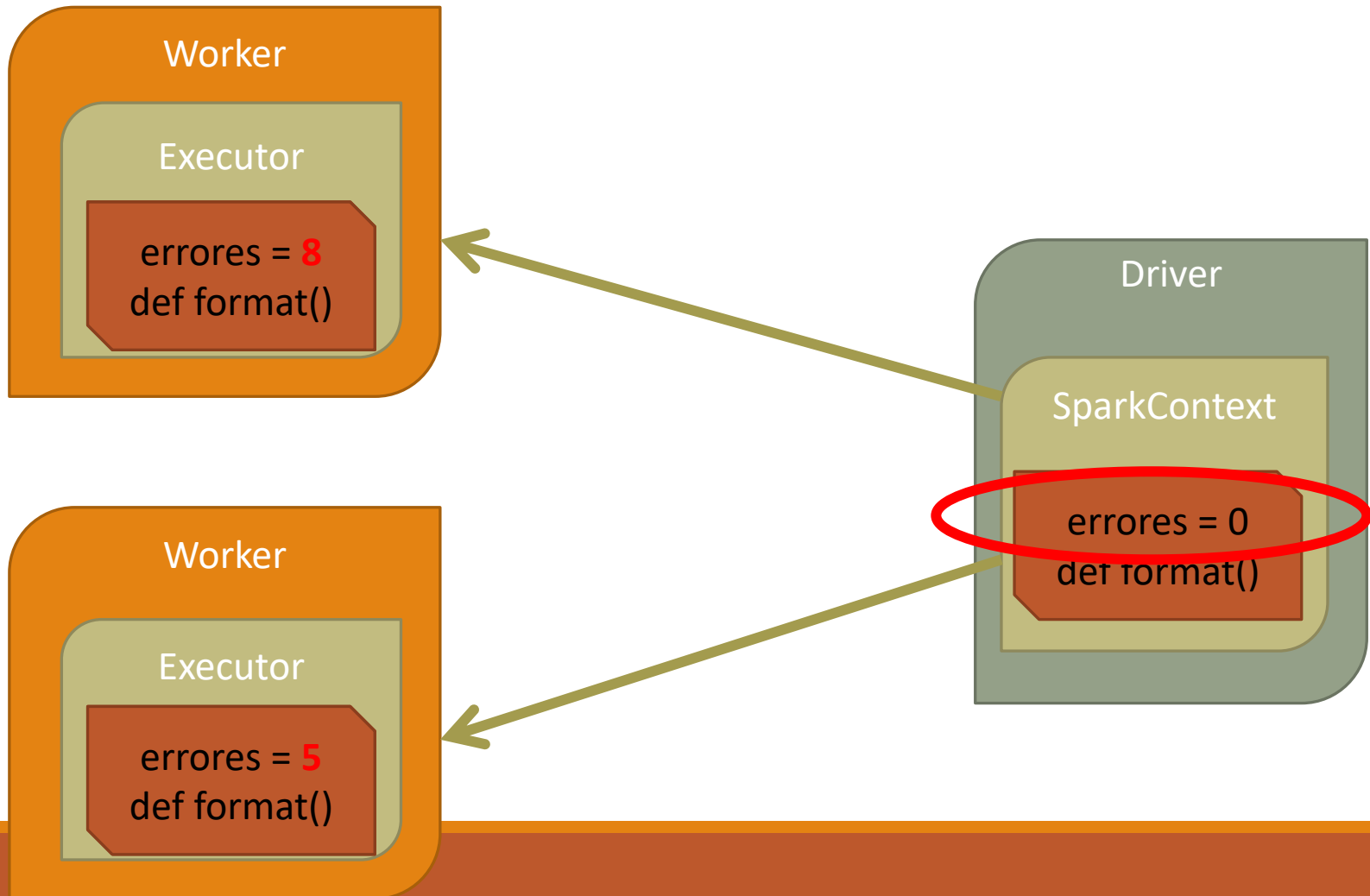
Ejemplo

```
errores = 0
lista_paises = ["ARG", "BOL", "BRA", "URU", ...]
def format(t):
    global lista_paises, errores
    campos = t.split("\t")
    if campos[5] in lista_paises:
        nac = campos[5]
    else:
        nac = "ERROR"
        errores = errores + 1
    return (int(campos[0]), campos[1] + " " + campos[2],
            int(campos[3]), campos[4], nac)

cli = clientes.textFile("Clientes").map(format)
cli.count()
print(errores)
```

El incremento de errores se ejecuta dentro de la función format en la llamada al map que es lazy. Se debe forzar la ejecución del DAG.

Variables globales en Spark



Acumuladores

Son variables especiales de Spark que permiten llevar a cabo un acumulado global entre todos los nodos del cluster

Cada nodo lleva a cabo su propio acumulado.

Cuando el driver solicita el valor del acumulador, se realiza el merge entre todos los acumulados parciales de todos los workers.

Su propósito es el de *debugging*

Acumuladores

Al SparkContext le pedimos la creación de un acumulador con su valor inicial

```
errores = sc.accumulator(0)
lista_paises = ["ARG", "BOL", "BRA", "URU", ...]
def format(t):
    global lista_paises, errores
    campos = t.split("\t")
    if campos[5] in lista_paises:
        nac = campos[5]
    else:
        nac = "ERROR"
        errores+= 1
    return (int(campos[0]), campos[1] + " " + campos[2],
            int(campos[3]), campos[4], nac)
```

En un acumulador solo está permitido sumarle valores mediante el operador de incremento

```
cli = clientes.textFile("Clientes").map(format)
cli.count()
print(errores.value)
```

Mediante la propiedad value del acumulador podremos saber cuantas líneas tenían errores

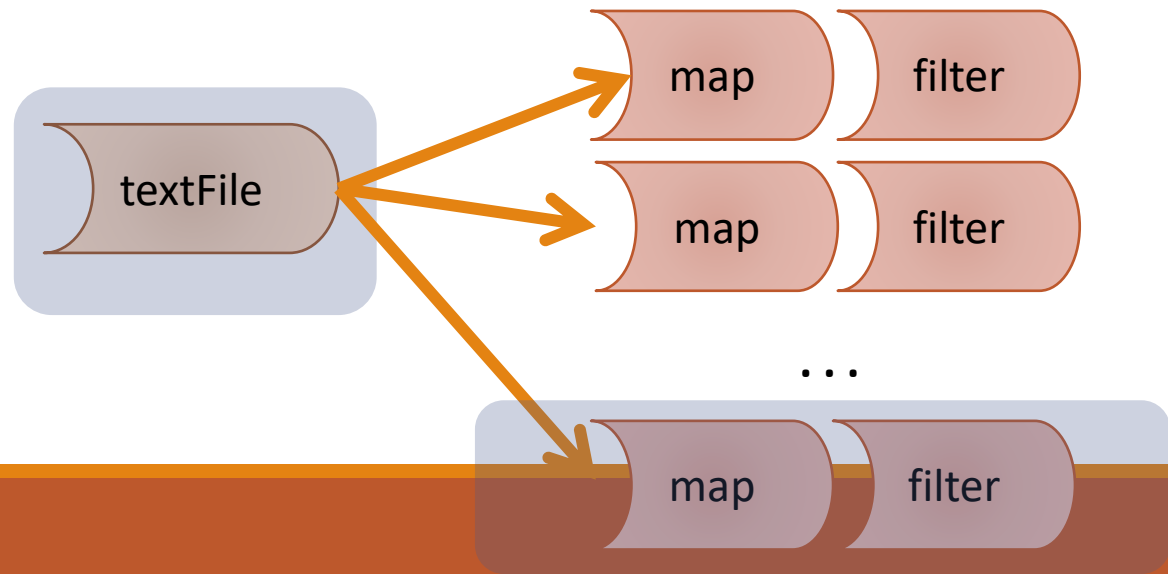
Algoritmos iterativos sobre Spark

```
data = sc.textFile(fileName)
for i in range(5):
    data = data.map(fMap)
    data = data.filter(fFilter)
result = data.collect()
```



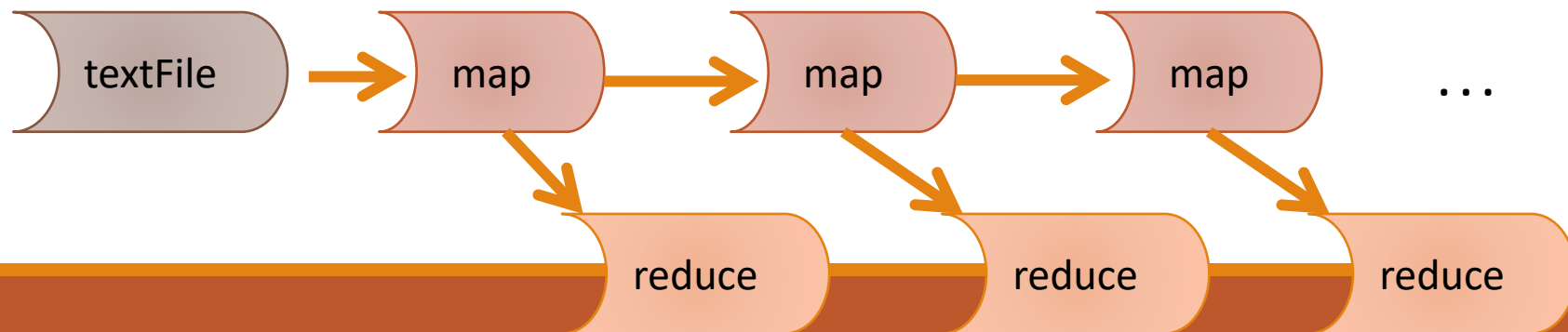
Algoritmos iterativos sobre Spark

```
data = sc.textFile(fileName)
for i in range(5):
    data2 = data.map(fMap)
    data2 = data2.filter(fFilter)
result = data2.collect()
```



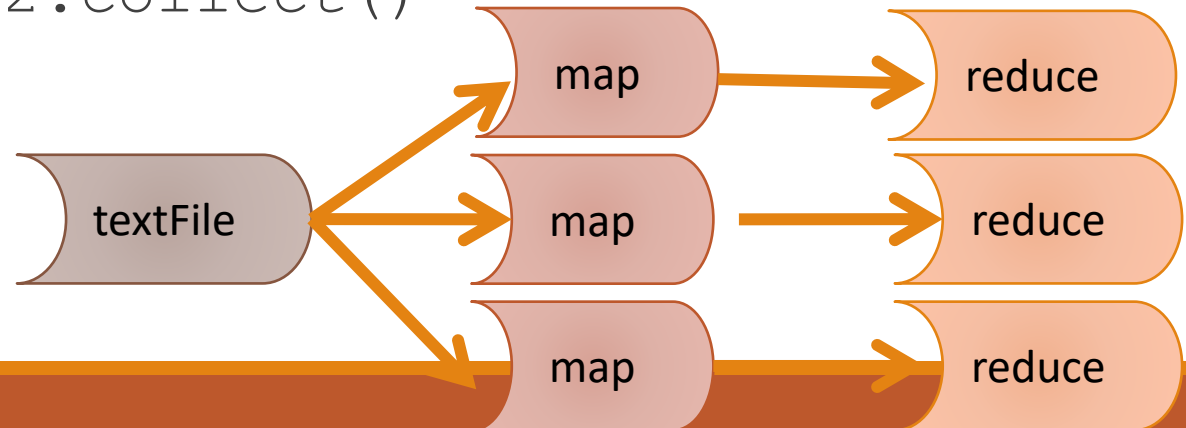
Algoritmos iterativos sobre Spark

```
data = sc.textFile(fileName)
error = 1
while error > 0.001:
    data = data.map(fMap)
    data.persist()
    error = data.reduce(fReduce)
result = data.collect()
```



Algoritmos iterativos sobre Spark

```
data = sc.textFile(fileName)
error = 1
while error > 0.001:
    data2 = data.map(fMap)
    error = data2.reduce(fReduce)
result = data2.collect()
```



parallelize

SparkContext tiene un método `parallelize` que se utiliza para generar una RDD desde una colección de valores (listas, tuplas, diccionarios, conjuntos).

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

Cada elemento de la lista se convierte en una tupla dentro de la nueva RDD.

parallelize - Ejemplo

```
values = [ ("ARG", "Buenos Aires"),  
           ("URU", "Montevideo"),  
           ("ECU", "Quito"),  
           ("PAR", "Asunción"),  
           ("PER", "Lima")      ]
```


```
rdd = sc.parallelize(values)
```



ARG	Buenos Aires
URU	Montevideo
ECU	Quito
PAR	Asunción
PER	Lima

Ejemplo – Cálculo de la mediana

```
def fmediana(v):  
    global mediana  
    if(v < mediana):  
        return 1  
    else:  
        return 0 if v == mediana else -1  
  
rdd = sc.parallelize([1,5,9,14,25,29,32,38,59])  
  
mediana = 0; cuantos = 1  
while cuantos != 0:  
    mediana = mediana + 1  
  
    rdd_temp = rdd.map(fmediana)  
    cuantos = rdd_temp.reduce(lambda x, y: x + y)  
print(mediana)
```



Ejemplo – Cálculo de la mediana

```
def fmediana(v):  
    global mediana  
    if(v < mediana):  
        return 1  
    else:  
        return 0 if v == mediana else -1
```

Es una variable
cuyo ámbito es el
driver

```
rdd = sc.parallelize([1,5,9,14,25,29,32,38,59])
```

```
mediana = 0; cuantos = 1  
while cuantos != 0:  
    mediana = mediana + 1  
  
    rdd_temp = rdd.map(fmediana)  
    cuantos = rdd_temp.reduce(lambda x, y: x + y)  
print(mediana)
```

Ejemplo – Cálculo de la mediana

```
def fmediana(v):  
    global bc_mediana  
    if(v < bc_mediana.value):  
        return 1  
    else:  
        return 0 if v == bc_mediana.value else  
  
rdd = sc.parallelize([1,5,9,14,25,29,32,37,59])  
  
mediana = 0; cuantos = 1  
while cuantos != 0:  
    mediana = mediana + 1  
    bc_mediana = sc.broadcast(mediana)  
    rdd_temp = rdd.map(fmediana)  
    cuantos = rdd_temp.reduce(lambda x, y: x + y)  
print(mediana)
```

Es más seguro
usar variables
broadcast

Variables broadcast

Una variable broadcast se crea a partir del SparkContext. Se envía de manera eficiente a todos los nodos del cluster. Se puede almacenar cualquier estructura: entero, string, lista, diccionario, etc.

```
bc = sc.broadcast(variable)
```

Al valor almacenado en una variable broadcast se accede mediante la propiedad *value*.

```
rdd.map(lambda t: t + bc.value))
```



Variables broadcast

Una variable broadcast se crea a partir del SparkContext. Se envía de manera eficiente a todos los nodos del cluster. Se puede almacenar cualquier estructura: entero, string, lista, diccionario, etc.

```
bc = sc.broadcast(variable)
```

Al valor almacenado en una variable broadcast se accede por la propiedad *value*.

```
rdd.map(lambda t: t + bc.value))
```



Spark garantiza que todos los nodos del cluster accedan al mismo valor

Problema

Cálculo de la sumatoria de 1 a 5: $1 + 2 + 3 + 4 + 5 = 15$

```
rdd = sc.parallelize([0])
```

```
for i in range(5):
```

```
    bc = sc.broadcast(i+1)
```

```
    rdd = rdd.map(lambda t: t + bc.value)
```

```
print(rdd.collect())
```



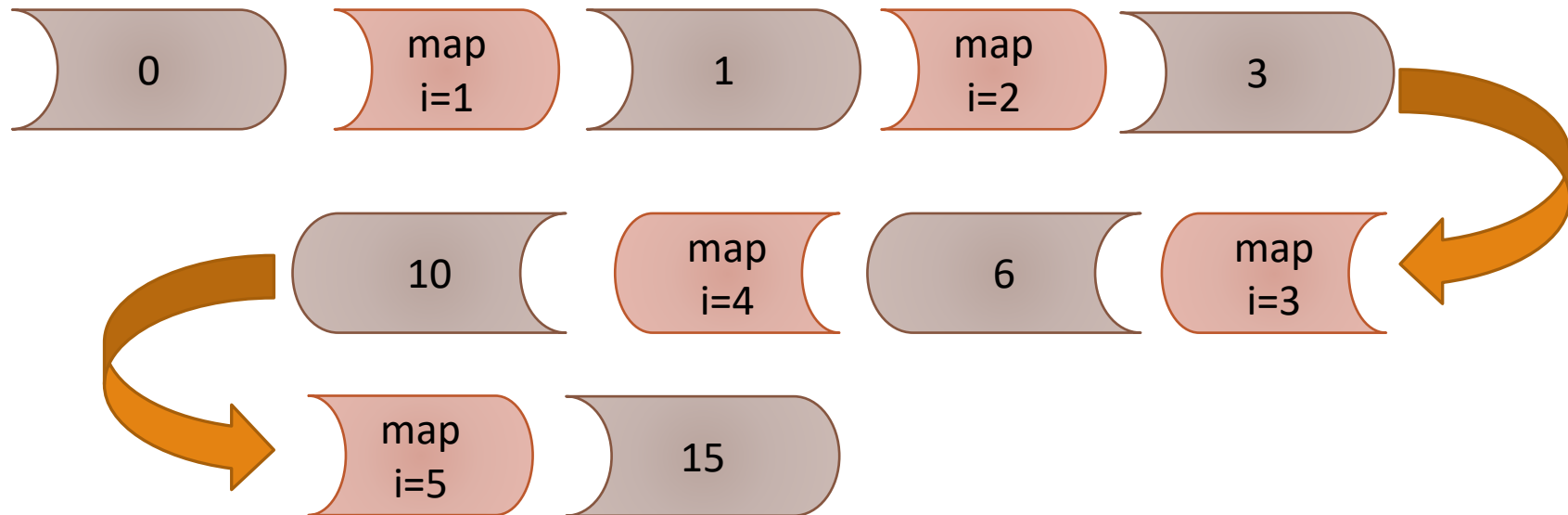
¿Qué imprime?

Problema

```
for i in range(5):
```

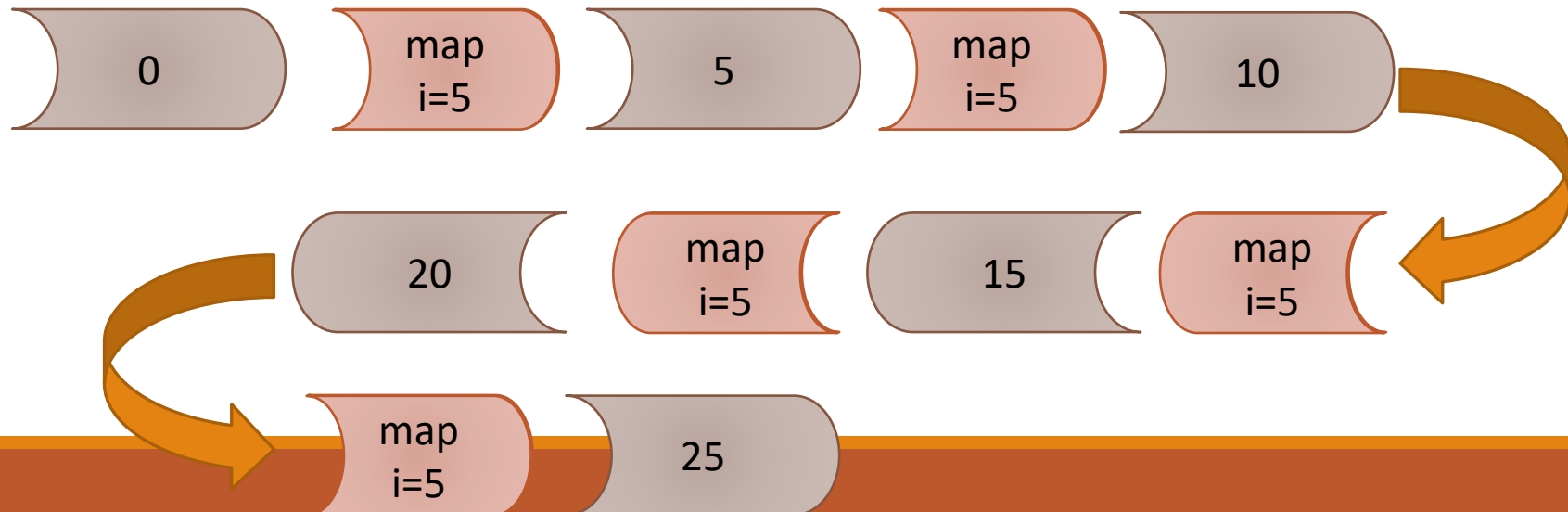
```
    bc = sc.broadcast(i+1)
```

```
    rdd = rdd.map(lambda t: t + bc.value)
```



Problema

```
for i in range(5):  
    bc = sc.broadcast(i+1)  
    rdd = rdd.map(lambda t: t + bc.value)  
    rdd.persist(); rdd.count()  
print(rdd.collect())
```



¿Problema?

```
rdd = sc.parallelize([0])
```

```
for i in range(5):
```

```
    rdd_temp = sc.parallelize([i+1])
```

```
    rdd_temp = rdd.cartesian(rdd_temp)
```

```
    rdd = rdd_temp.map(lambda v: v[0] + v[1])
```

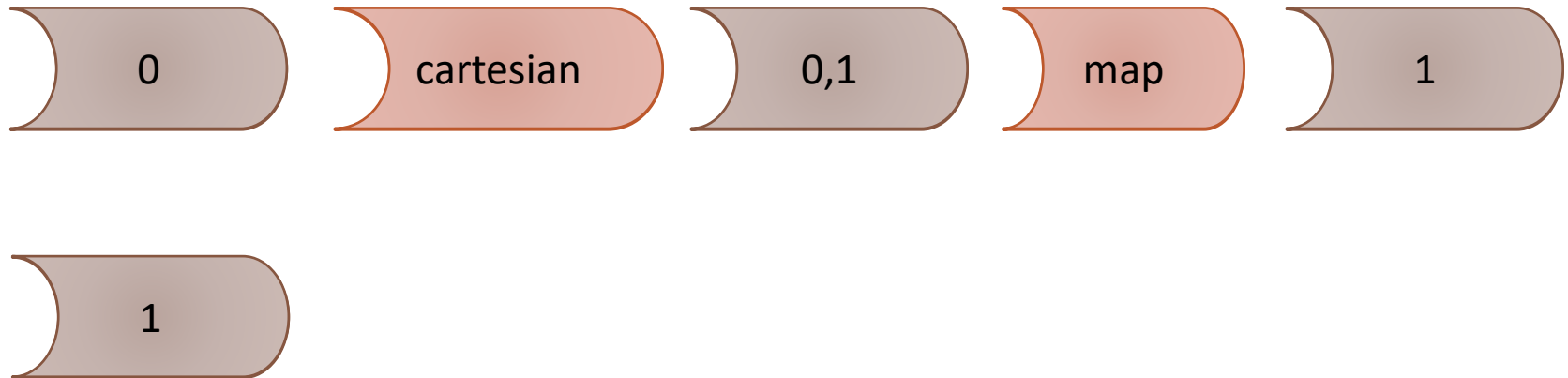
```
print(rdd.collect())
```



¿Qué imprime?

Solución

```
for i in range(5):  
    rdd_temp = sc.parallelize([i+1])  
    rdd_temp = rdd.cartesian(rdd_temp)  
    rdd = rdd_temp.map(lambda v: v[0] + v[1])  
print(rdd.collect())
```



Transformación mapPartitions

Permite trabajar con todas las tuplas que están en una partición.

$\text{mapPartitions}(\text{RDD}^n) \rightarrow \text{RDD}^m \quad n \geq m$

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

- La función pasada por parámetro recibirá un iterador de tuplas y debe devolver una lista de tuplas como salida (la partición de la RDD resultado):

$\text{fMapPartitions}(\text{ite}) \rightarrow [t_o]$

Transformación mapPartitions

```
def fMapPartitions(ite):
```

```
    cont=0; acum=0
```

```
    for t in ite:
```

```
        cont+= 1
```

```
        acum+= t
```

```
    return [(acum, cont)]
```

Es útil cuando se debe acceder a recursos como bases de datos, servicios web, etc.

```
rdd = sc.parallelize(list(range(50)), 4)
```

```
res = rdd.mapPartitions(fMapPartitions)
```

```
print(res.collect())
```


Transformación zip

Cuando se poseen dos RDDs que tienen el mismo número de particiones y el mismo número de tuplas en cada partición es posible juntarlas sin necesidad de hacer un join.

$$\text{zip}(\text{RDD}_1^n, \text{RDD}_2^n) \rightarrow \text{RDD}^n$$

Es útil usarla cuando se sabe la fuente de ambas RDDs: resultado de distintos maps a partir de la misma RDD o leyendo de fuentes que fueron generadas con el mismo criterio.

Transformación zip

54	("AA", 500)
23	("BB", 800)
14	("CC", 200)
6	("DD", 500)
33	("EE", 200)

54	(30, 800)
23	(10, 200)
14	(45, 500)
6	(8, 100)
33	(26, 900)



zip



(54, ("AA", 500))	(54, (30, 800))
(23, ("BB", 800))	(23, (10, 200))
(14, ("CC", 200))	(14, (45, 500))
(6, ("DD", 500))	(6, (8, 100))
(33, ("EE", 200))	(33, (26, 900))

Transformación zip

```
cli1 = clientes.map(lambda t: edad(t[4]))
cli2 = clientes.map(lambda t:
                    t[2] + ", " + t[1])

res = clientes.zip(cli1).zip(cli2)

print(res.take(10))
```

Transformaciones zip

zipWithIndex

- `res = rdd.zipWithIndex()`

zipWithUniqueId

- `res = rdd.zipWithUniqueId()`

Le agrega un número de fila a cada tupla
El primer elemento de la primer partición tendrá el índice 0. El último elemento de la última partición tendrá el índice n-1

Transformaciones zip

zipWithIndex

- `res = rdd.zipWithIndex()`

zipWithUniqueId

- `res = rdd.zipWithUniqueId()`

A cada tupla le asigna un número único, no hay garantía que dos tuplas consecutivas tengan ids consecutivos. Es más eficiente que `zipWithIndex`