Conceptos y Aplicaciones de Big Data

SPARK

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

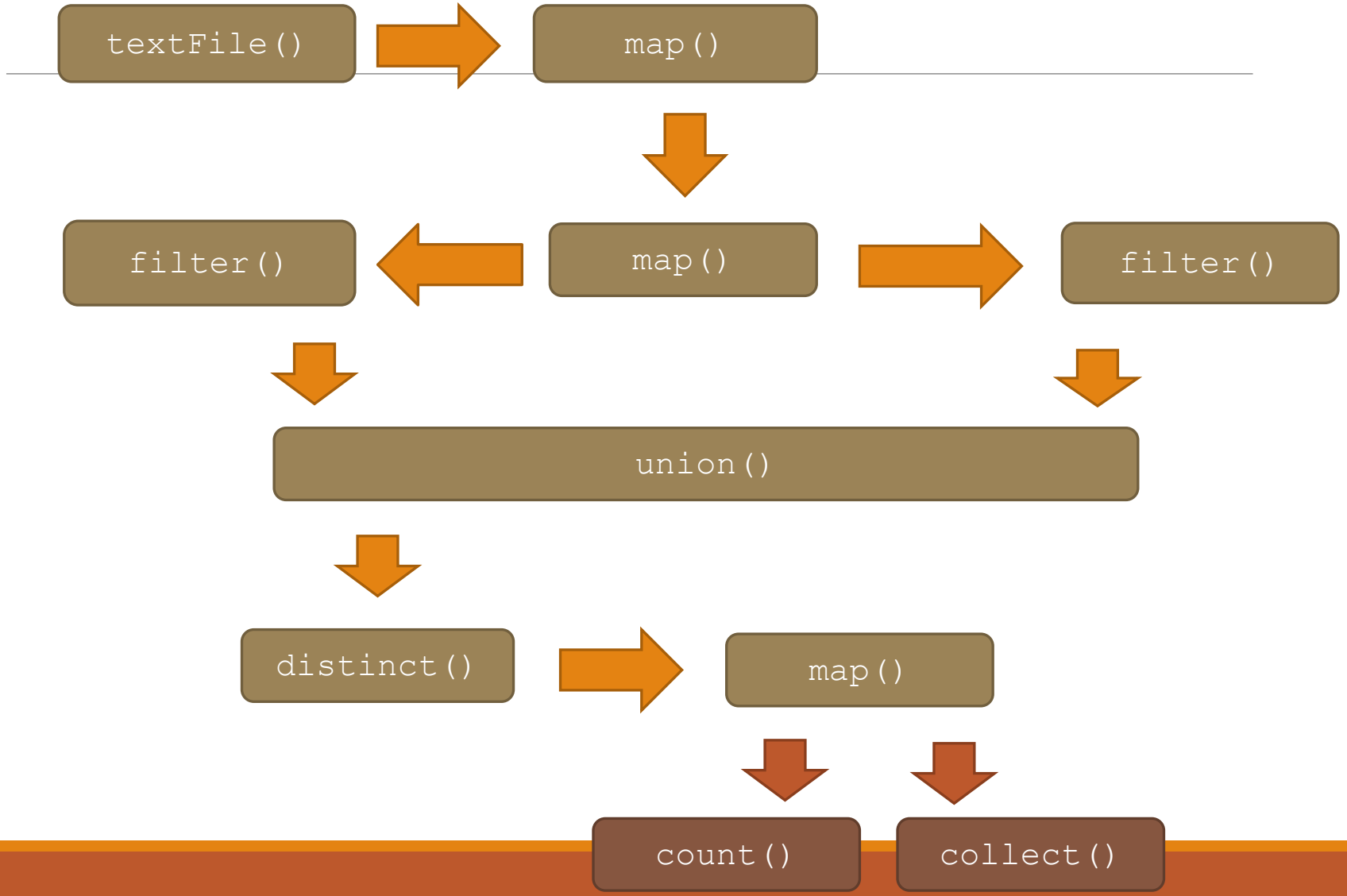
Temario

API - Spark

- Persistencia
- Más transformaciones y acciones
- Trabajando con pares clave-valor
  - WordCount

Optimización de particiones

Repaso


textFile()
map()
filter()
map()
filter()
union()
distinct()
map()
count()
collect()

Persistencia

Cuando una RDD (que depende de muchas otras) se va a utilizar en más de una operación (más de una acción) es conveniente persistirla.

Spark posee diferentes tipos de persistencia según las necesidades: en memoria, en disco, ambas.

Para persistir una RDD:

```
rdd = rdd.persist()
```

Persistencia

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

print(cliens.count())

print(cliens.collect())
```

StorageLevel.DISK_ONLY
StorageLevel.MEMORY_AND_DISK
y más …

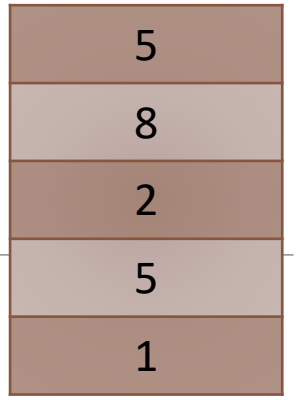
Clase StorageLevel(disk, memory, offheap, deserialized, replication)

Problema

Dada la RDD:

¿Cómo calcular máximo, mínimo y promedio de todos sus valores?

Problema



Solución 1

```
max = rdd.reduce(lambda x,y:
                        x if x > y else y)

min = rdd.reduce(lambda x,y:
                        x if x < y else y)

acum = rdd.reduce(lambda x,y:
                        x+y)

prom = acum / rdd.count()
```

Problema

5	5	5	1
8	8	8	1
2	2	2	1
5	5	5	1
1	1	1	1

Solución 2

```
rddTmp = rdd.map(lambda x: (x, x, x, 1))
all = rddTmp.reduce(lambda x, y:
      ( x[0] if x[0] > y[0] else y[0],
        x[1] if x[1] < y[1] else y[1],
        x[2] + y[2],
        x[3] + y[3] ) )
prom = all[2] / all[3]
```

Acción aggregate

La acción aggregate se utiliza para reducir/resumir una RDD pero cambiando la estructura de la RDD.

- La reducción con aggregate se hace por pares de tuplas.

Recibe como parámetro un valor "cero", una función *f* para operar con una tupla original y otra "resumida" y una segunda función *g* que sabe operar con dos tuplas "resumidas".

- La función *f* pasada por parámetro recibe una tupla t_1 con el formato "resumido", una tupla v con el formato original de la RDD y debe devolver una tupla t_3 con el mismo formato que t_1:

$$f(t_1, v) \rightarrow t_3$$

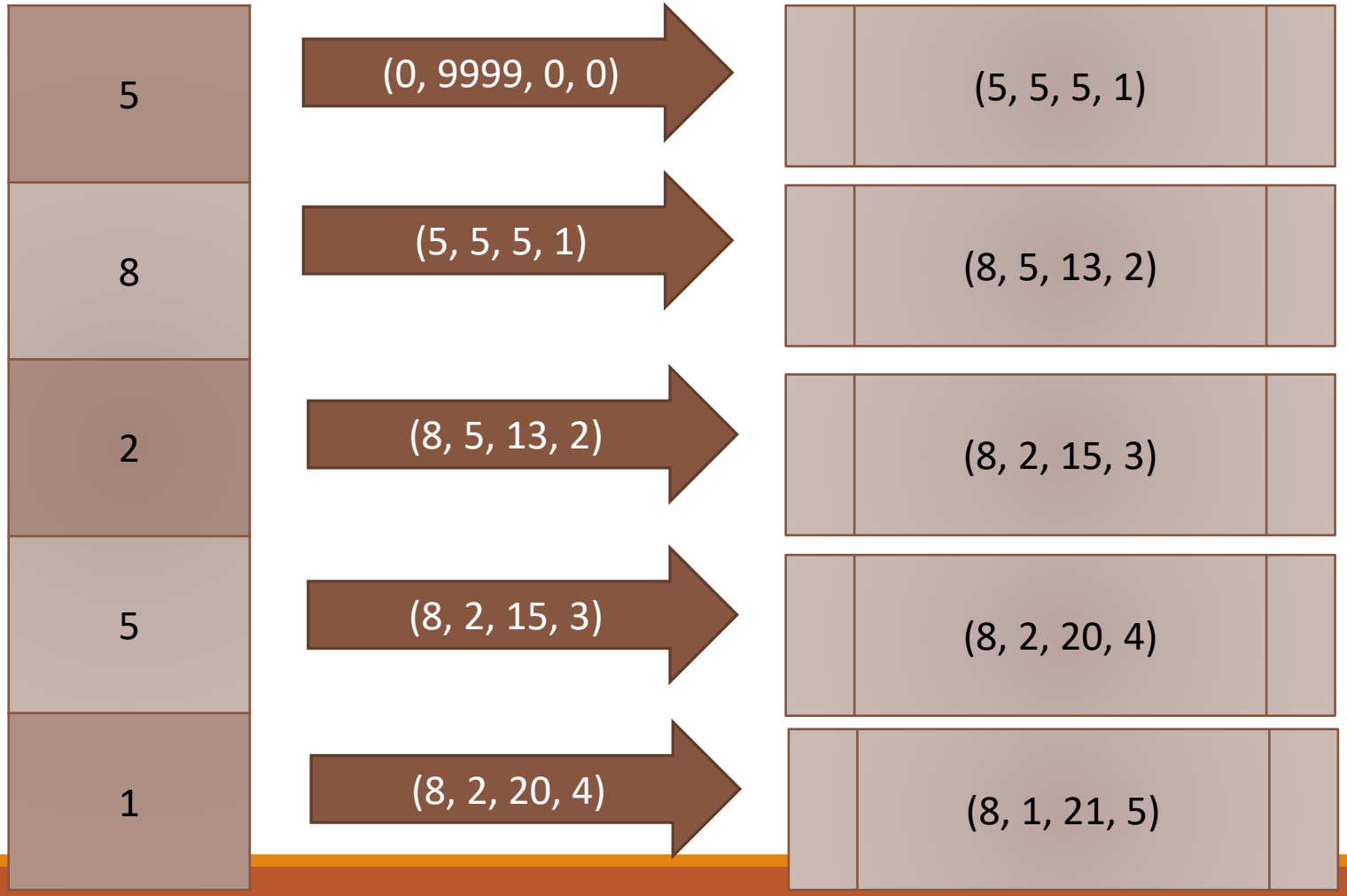
- La función *g* pasada por parámetro recibe dos tuplas t_1 y t_2 con el formato "resumido" y debe devolver una tupla t_3 con el mismo formato que t_1 y t_2:

$$g(t_1, t_2) \rightarrow t_3$$

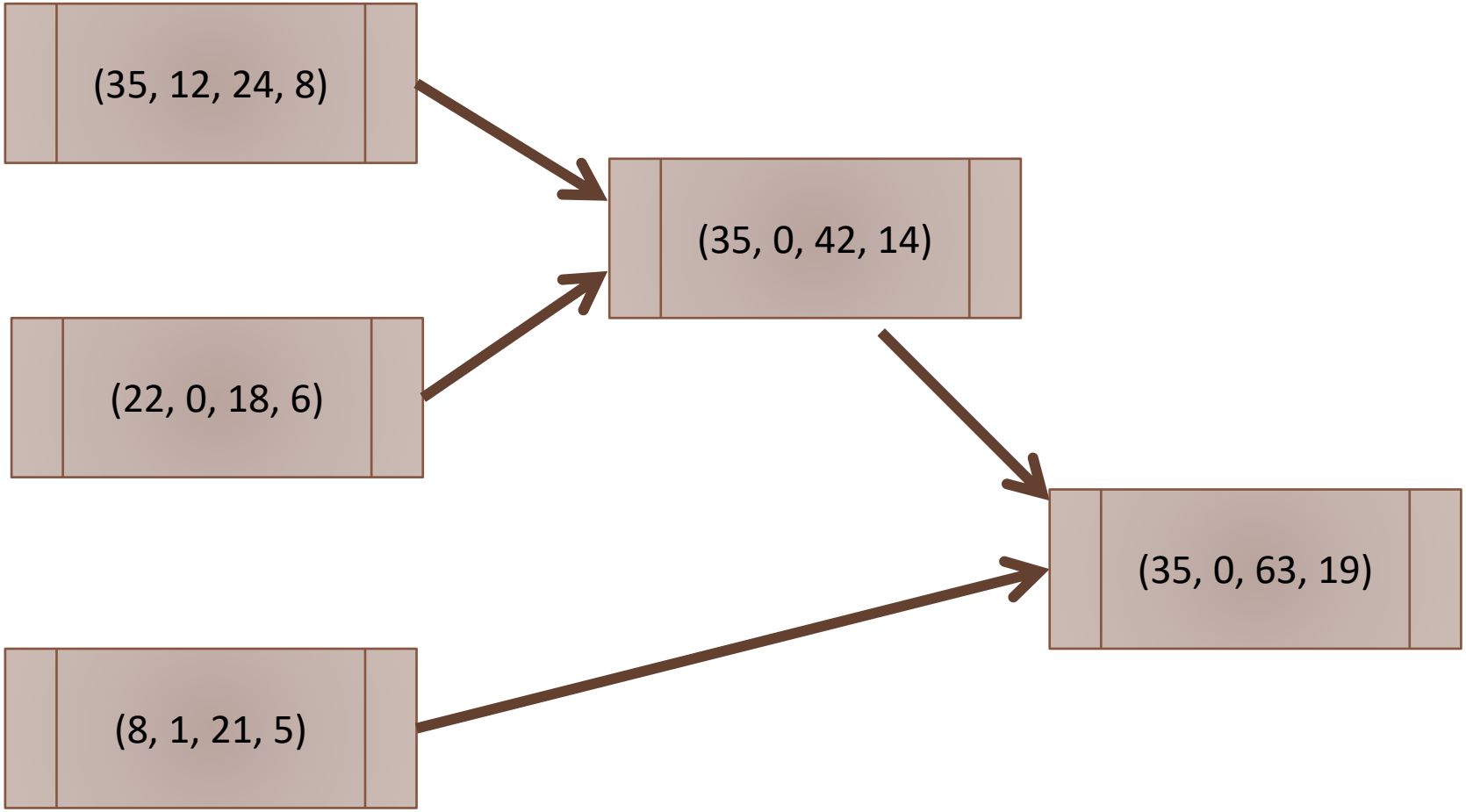
Acción aggregate

```
all = rdd.aggregate(
      (0, 9999, 0, 0),
      (lambda res, ori:
            (res[0] if res[0] > ori else ori,
             res[1] if res[1] < ori else ori,
             res[2] + ori, res[3] + 1) ),
      (lambda r1, r2:
            (r1[0] if r1[0] > r2[0] else r2[0],
             r1[1] if r1[1] < r2[1] else r2[1],
             r1[2] + r2[2], r1[3] + r2[3] ) ) )
```

Acción aggregate



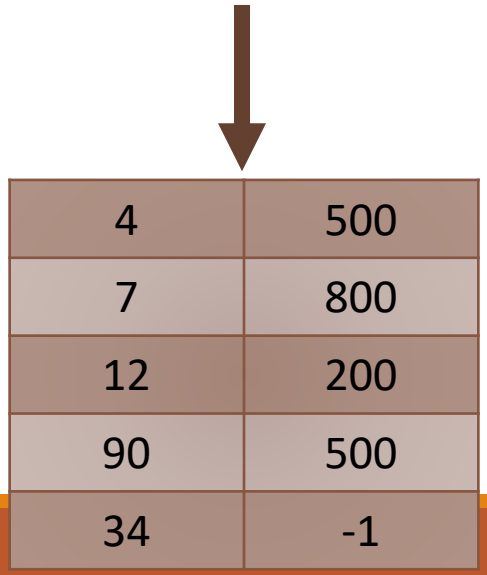
Acción aggregate



Trabajando con RDDs clave/valor

La función map puede ser utilizada para crear tuplas de la forma clave-valor (*Pair RDD*).

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



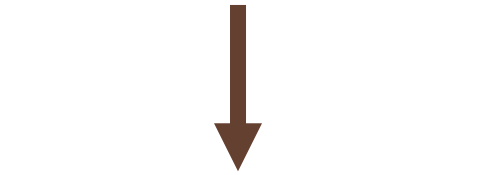
4	500
7	800
12	200
90	500
34	-1

El objetivo de este map es que devuelva una tupla (clave, valor)
En este ejemplo, la tupla devuelta es (ID_Cliente, saldo)

Trabajando con RDDs clave/valor

La función map puede ser utilizada para crear tuplas de la forma clave-valor.

```
kv = prestamos.map(lambda t:
                   (t[0], (t[1], t[2]) ) )
```



54	(12, 500)
23	(24, 800)
129	(6, 200)
6	(36, 500)
31	(18, 200)

El "valor" puede tener más de un dato, pero la RDD kv siempre tiene que tener dos campos
En este ejemplo, la tupla devuelta es (ID_Caja, (cuotas, monto))

Transformaciones "ByKey"

Spark posee varias funciones de transformación que operan con RDDs que tienen la forma (clave, valor)

- reduceByKey
- aggregateByKey
- groupByKey
- combineByKey
- sortByKey
- …

Transformación reduceByKey

La transformación reduceByKey se utiliza para reducir/resumir una RDD.

- La reducción de una RDD se hace por pares de tuplas.
- Para cada clave distinta se reducen los valores asociados a dicha clave

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

- La función pasada por parámetro recibe por parámetro dos "valores" asociados a la misma clave y debe devolver un "valor" del mismo tipo:

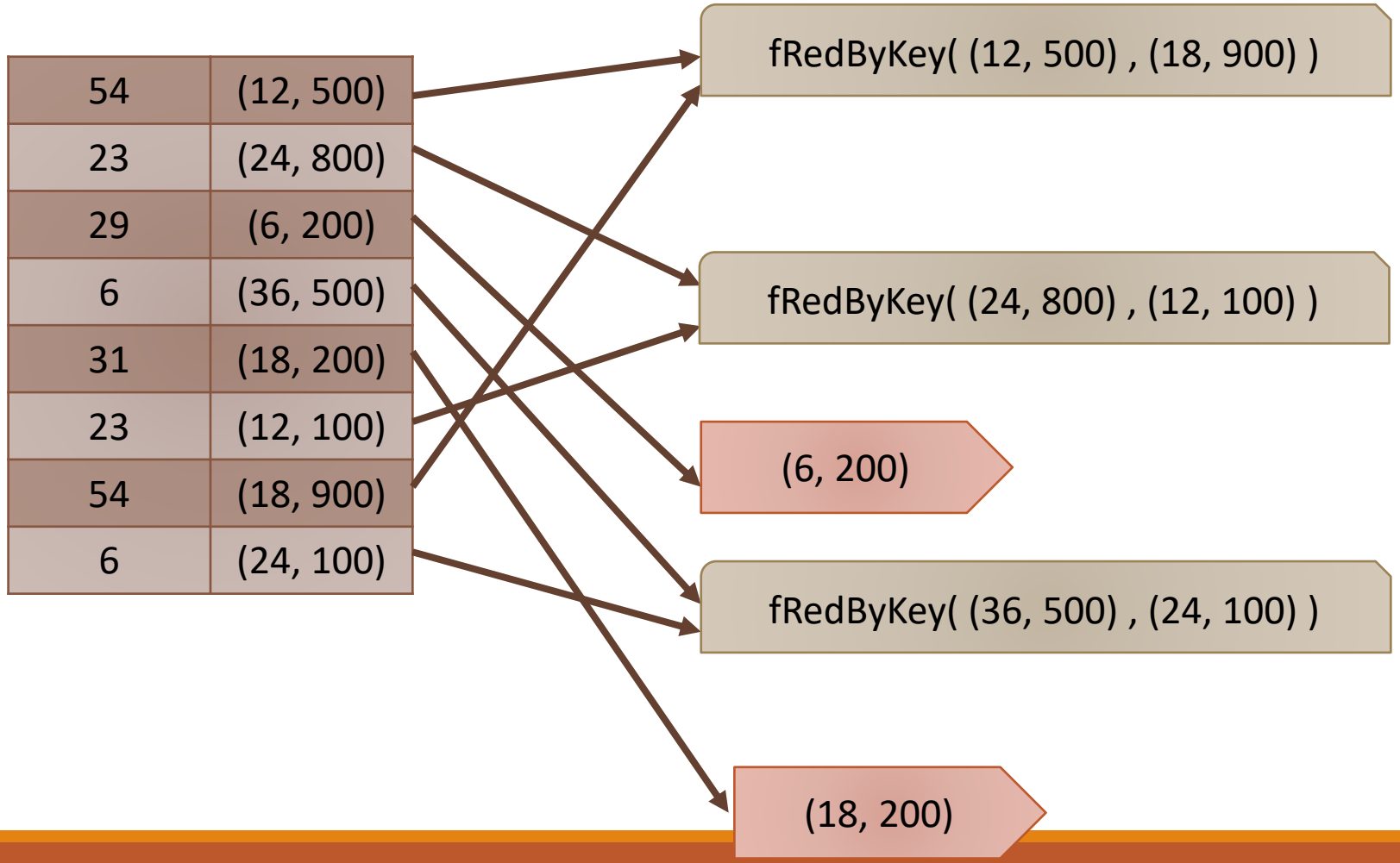
$$\text{reduceByKey}(v_1, v_2) \rightarrow v_3$$

Transformación reduceByKey

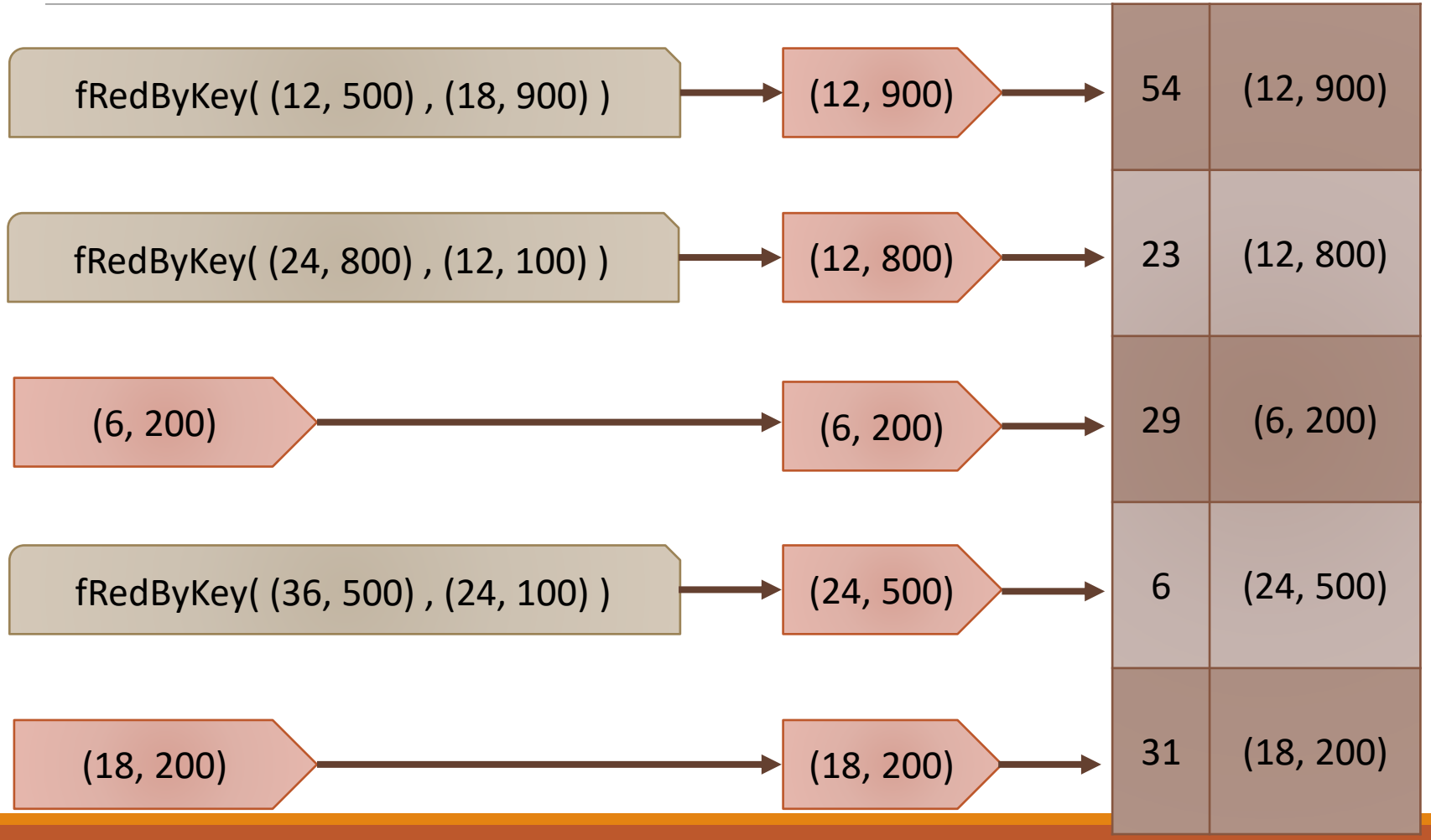
```
def freductionByKey(v1, v2):
      return (v1[0] if v1[0] < v2[0] else v2[0],
              v1[1] if v1[1] > v2[1] else v2[1])

minmax = prestamos.reduceByKey(freductionByKey)
```

```
minmax = rdd.reduceByKey(lambda v1, v2:
            (v1[0] if v1[0] < v2[0] else v2[0],
             v1[1] if v1[1] > v2[1] else v2[1]) )
```

Transformación reduceByKey



Transformación reduceByKey



Transformaciones/Acciones "ByKey"

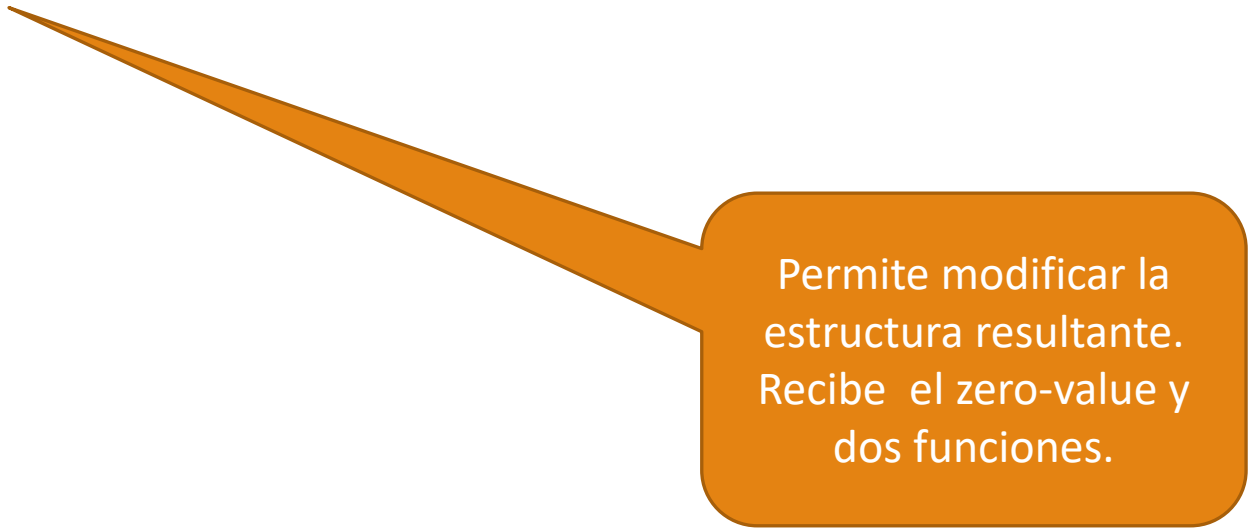
aggregateByKey

groupByKey

sortByKey

mapValues

countByKey



Transformaciones/Acciones "ByKey"

aggregateByKey

groupByKey

sortByKey

mapValues

countByKey

La RDD resultante tiene para cada key, la lista de valores asociados. Se debe tener cuidado, ya que su ejecución es muy costosa.

```
rdd = clientes.groupByKey()
res = rdd.collect()
for elem in res:
  key = elem[0]
  values = elem[1]
  for v in values:
    print(key, v)
```

Transformaciones/Acciones "ByKey"

aggregateByKey

groupByKey

sortByKey

mapValues

countByKey

Esta función no hace tarea de reducción, solo ordena por clave.
Resulta útil su aplicación luego de una reducción.

Transformaciones/Acciones "ByKey"

aggregateByKey

groupByKey

sortByKey

mapValues

countByKey

Transformaciones/Acciones "ByKey"

aggregateByKey

groupByKey

sortByKey

mapValues

countByKey

Devuelve la cantidad de tuplas que tiene cada clave.
countByKey es una acción.

Transformación join

Para poder hacer la operación de join entre dos RDDs, ambas deben tener la forma clave-valor.

- Input
  - rdd1 = (key, (field1_1, field1_2, field1_3, …))
  - rdd2 = (key, (field2_1, field2_2, field2_3, …))
- Output
  - rdd3 = (key, **(** (field1_1, field1_2, field1_3, …) , (field2_1, field2_2, field2_3, …) **)**)

Esta transformación actúa como un *"inner join"*.

```
rdd3 = rdd1.join(rdd2)
```

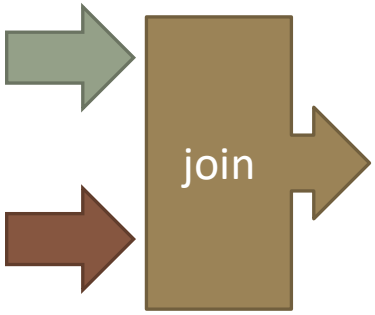
Transformación join

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

cajas = sc.textFile("Cajas de ahorro")
cajas = cajas.map(lambda line: line.split("\t"))

cli_j = clientes.map(lambda t: (t[0], t[1:]))
caj_j = cajas.map(lambda t: (t[1], (t[0), t[2])))

res = cli_j.join(caj_j)
```

Transformación join

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

23	(30, 800)
54	(10, 200)
16	(45, 500)
23	(8, 100)
54	(26, 900)
6	(87, 100)



54	("AA", 500), (10, 200)
54	("AA", 500), (26, 900)
54	("CC", 200), (10, 200)
54	("CC", 200), (26, 900)
23	("BB", 800), (30, 800)
23	("BB", 800), (8, 100)
23	("EE", 200), (30, 800)
23	("EE", 200), (8, 100)
6	("DD", 500), (87, 100)

Transformación cogroup

Similar a la operación de join existe cogroup, la cual trabaja sobre dos RDDs con forma clave-valor

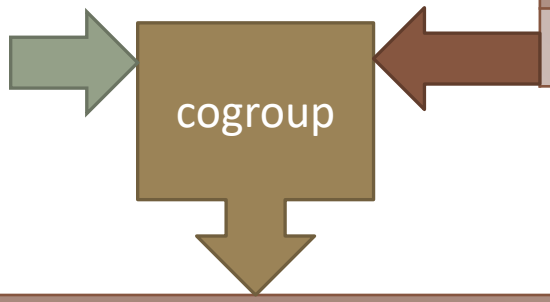
Cogroup también actúa como groupBy, en el sentido que devolverá para cada clave todos sus valores asociados

```
rdd3 = rdd1.cogroup(rdd2)
```

Transformación cogroup

54	("AA", 500)
23	("BB", 800)
54	("CC", 200)
6	("DD", 500)
23	("EE", 200)
23	("FF", 100)
54	("GG", 900)
6	("HH", 100)

54	(12, 500),
23	(30, 800)
54	(10, 200)
16	(45, 500)
23	(29, 200)
23	(8, 100)
54	(26, 900)
16	(87, 100)



54	(("AA", 500), ("CC", 200), ("GG", 900)), ((12, 500), (10, 200), (26, 900))
23	(("BB", 800), ("EE", 200), ("FF", 100)), ((30, 800), (29, 200), (8, 100))
16	( ) , ((45, 500) , (45, 500), (87, 100))
6	(("DD", 500), ("HH", 100)) , ( )

Transformaciones tipo join

rightOuterJoin

leftOuterJoin

cartesian

…

Transformación flatMap

La función flatMap permite que de cada tupla procesada, se generen una o más tuplas en la RDD resultante.

$$flatMap(RDD^n) \rightarrow RDD^m \qquad m \geq n$$

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

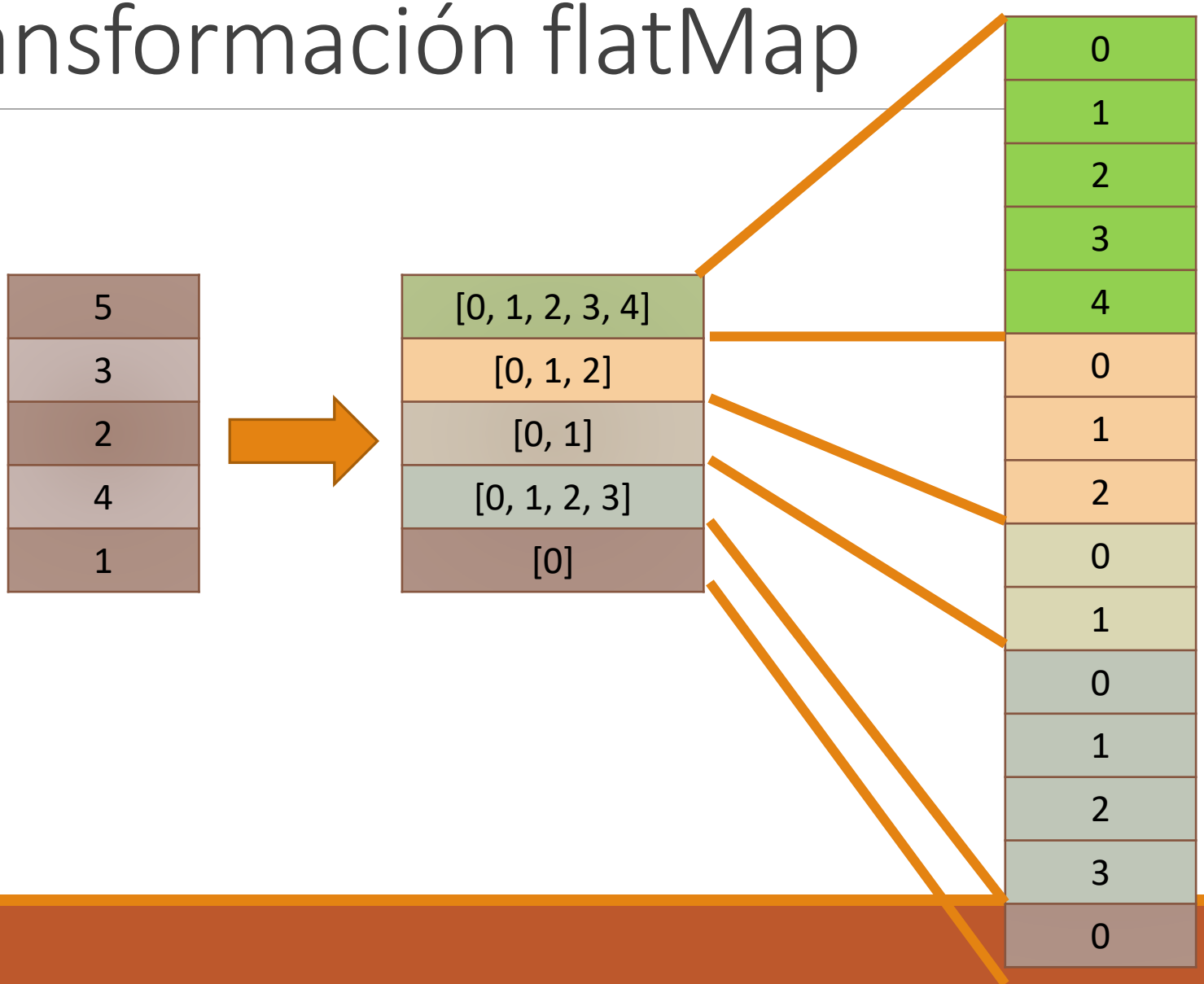
- La función pasada por parámetro recibirá por parámetro un tupla y debe devolver una lista de valores como salida (cada valor devuelto será parte de una nueva tupla):

$$fflatMap(t_i) \rightarrow [\ t_o\ ]$$

Transformación flatMap

```
def fflatMap(tupla):
     return list(range(tupla))

res = rdd.flatMap(fflatMap)



res = rdd.map(lambda tupla: list(range(t))
```

Transformación flatMap



WordCount

```
wordcount = sc.textFile("WordCount")

wordcount = wordcount.flatMap(lambda t:
                              t.split())

wordcount = wordcount.map(lambda t: (t,1))

wordcount = wordcount.reduceByKey(
                    lambda w1, w2: w1 + w2)
```

WordCount

```
wordcount = sc.textFile("WordCount")  \
            .flatMap(lambda t: t.split()) \
            .map(lambda t: (t,1)) \
            .reduceByKey(lambda w1, w2: w1+w2) \
           .map(lambda t: (t[1], t[0])) \
            .top(20)
```

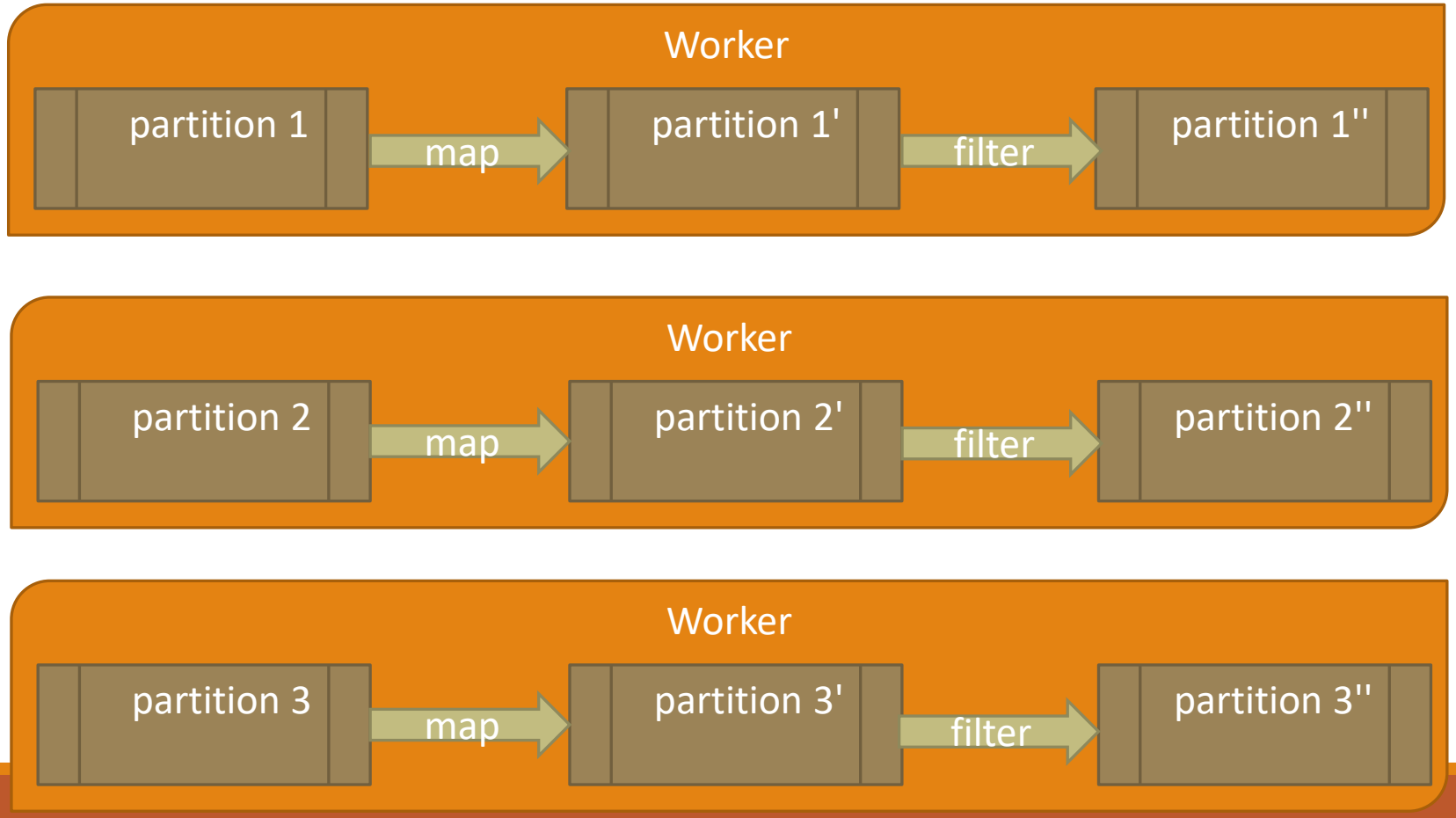
Transformaciones

Existen dos tipos de transformaciones

- Transformaciones "estrechas" (*Narrow transformations*)
  - Pueden ser ejecutadas sobre una partición sin necesidad de transferir datos entre nodos
- Transformaciones "amplias" (*Wide transformations*)
  - Necesitan transferir datos entre nodos para poder realizar la transformación

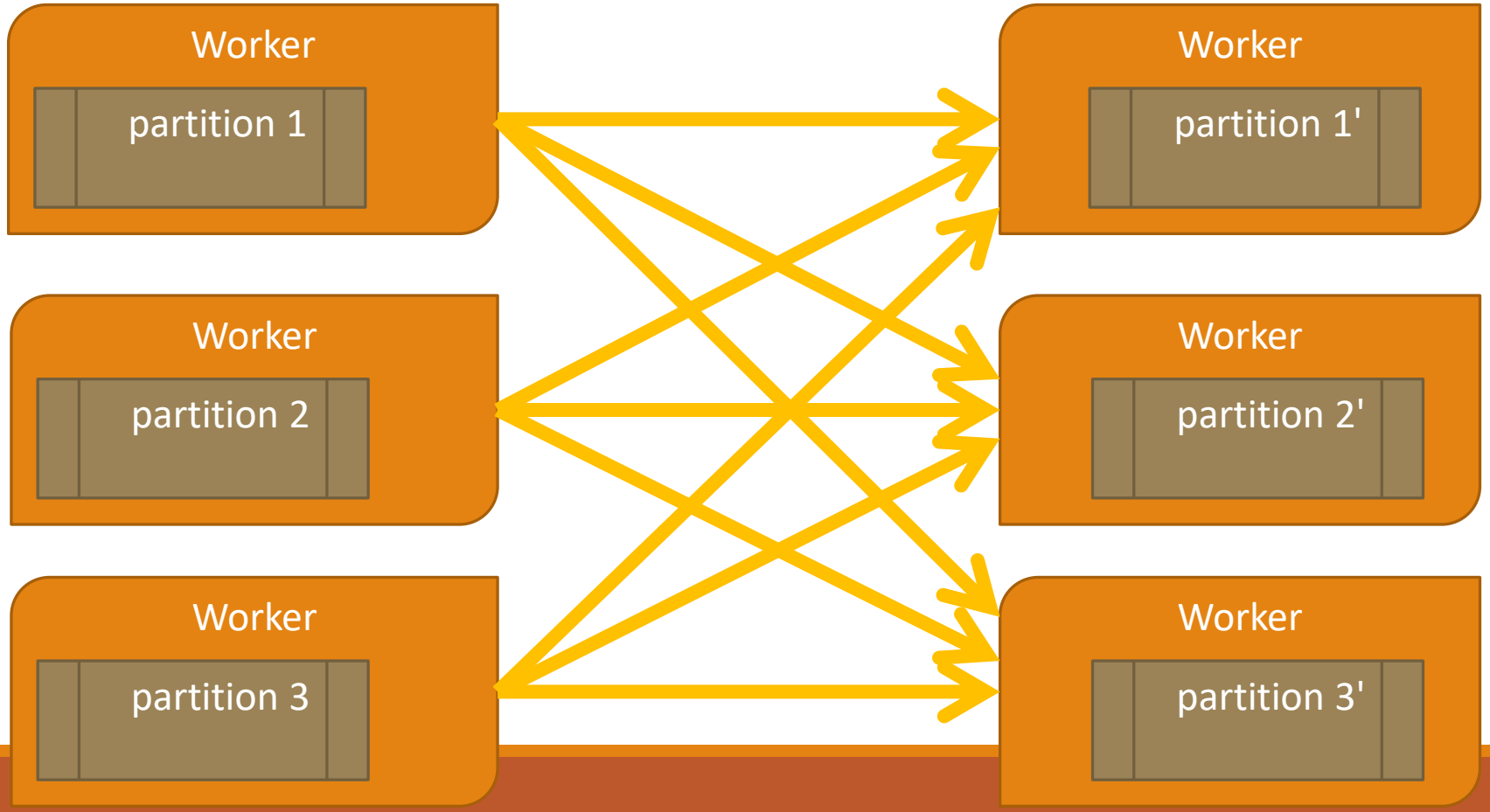
Transformaciones *narrow*

map, filter, …

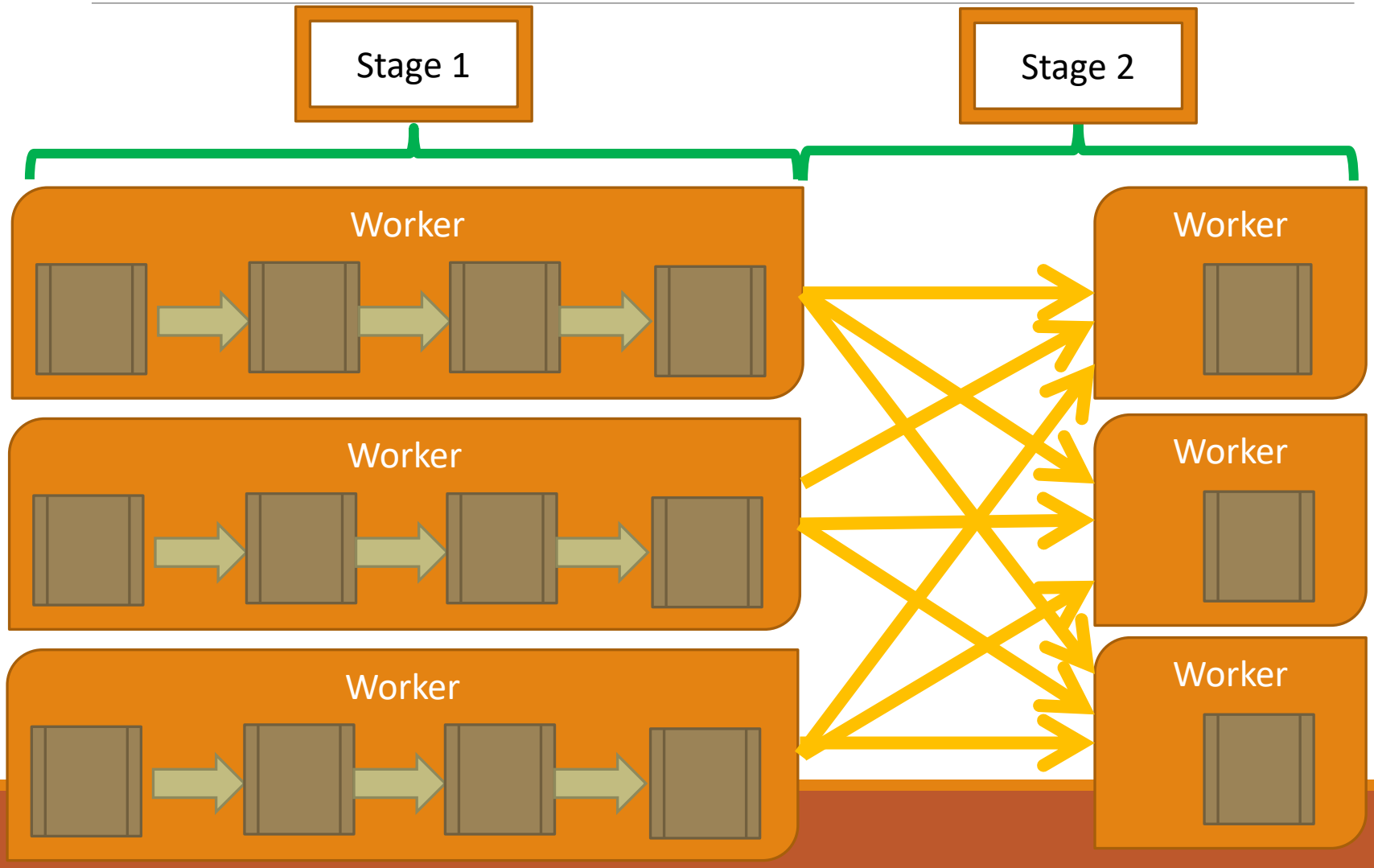


Transformaciones *wide*

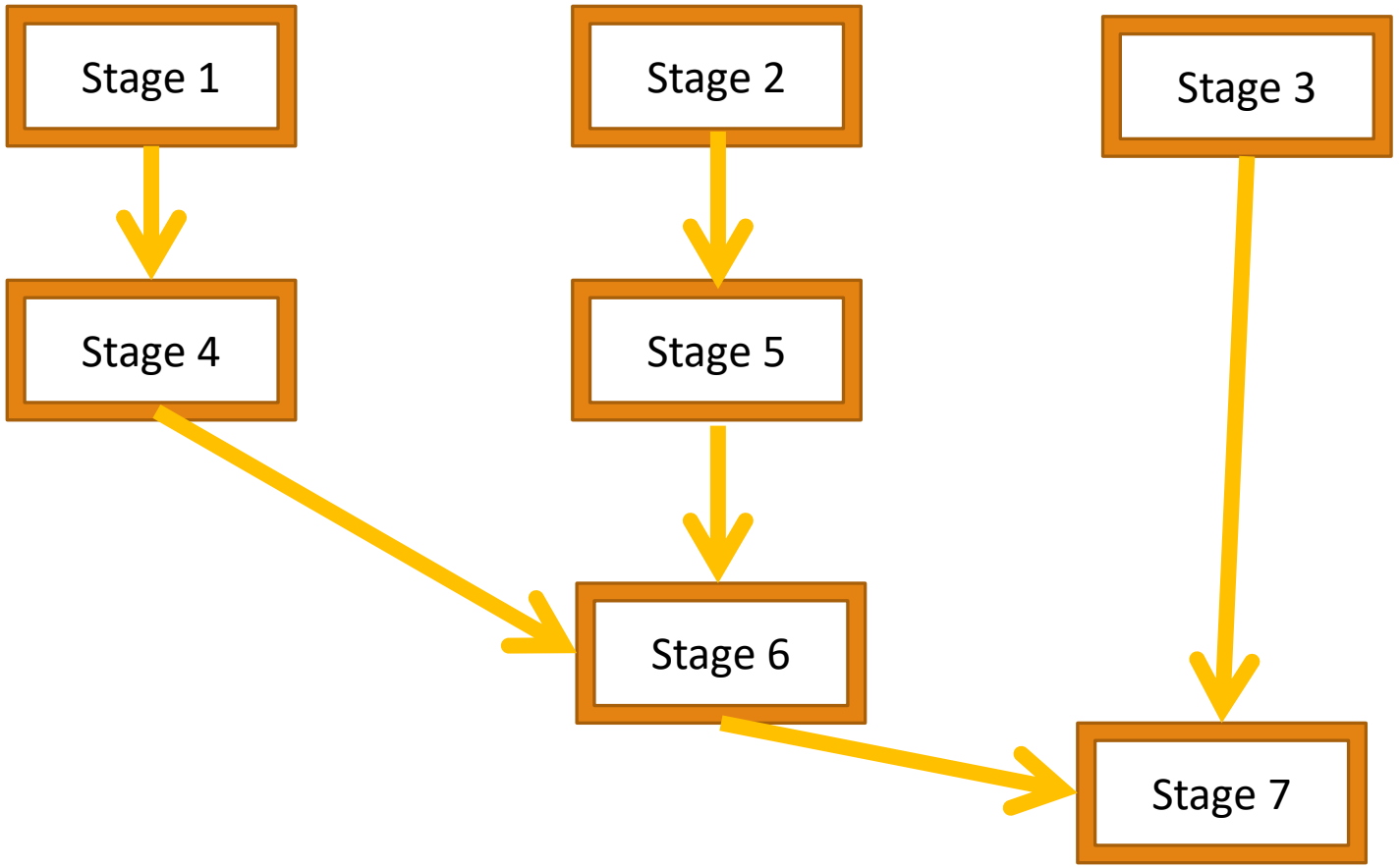
groupBy, join, …



Stages



Stages



Particionado de datos

La tarea de "shuffle" en Spark es la más costosa.

El DAGScheduler hará una optimización.

Es posible diagramar las etapas para garantizar una ejecución rápida de un trabajo en Spark

- Maximizar transformaciones narrow
- Minimizar transformaciones wide
  - Co-particionar las RDDS

Particionado de datos

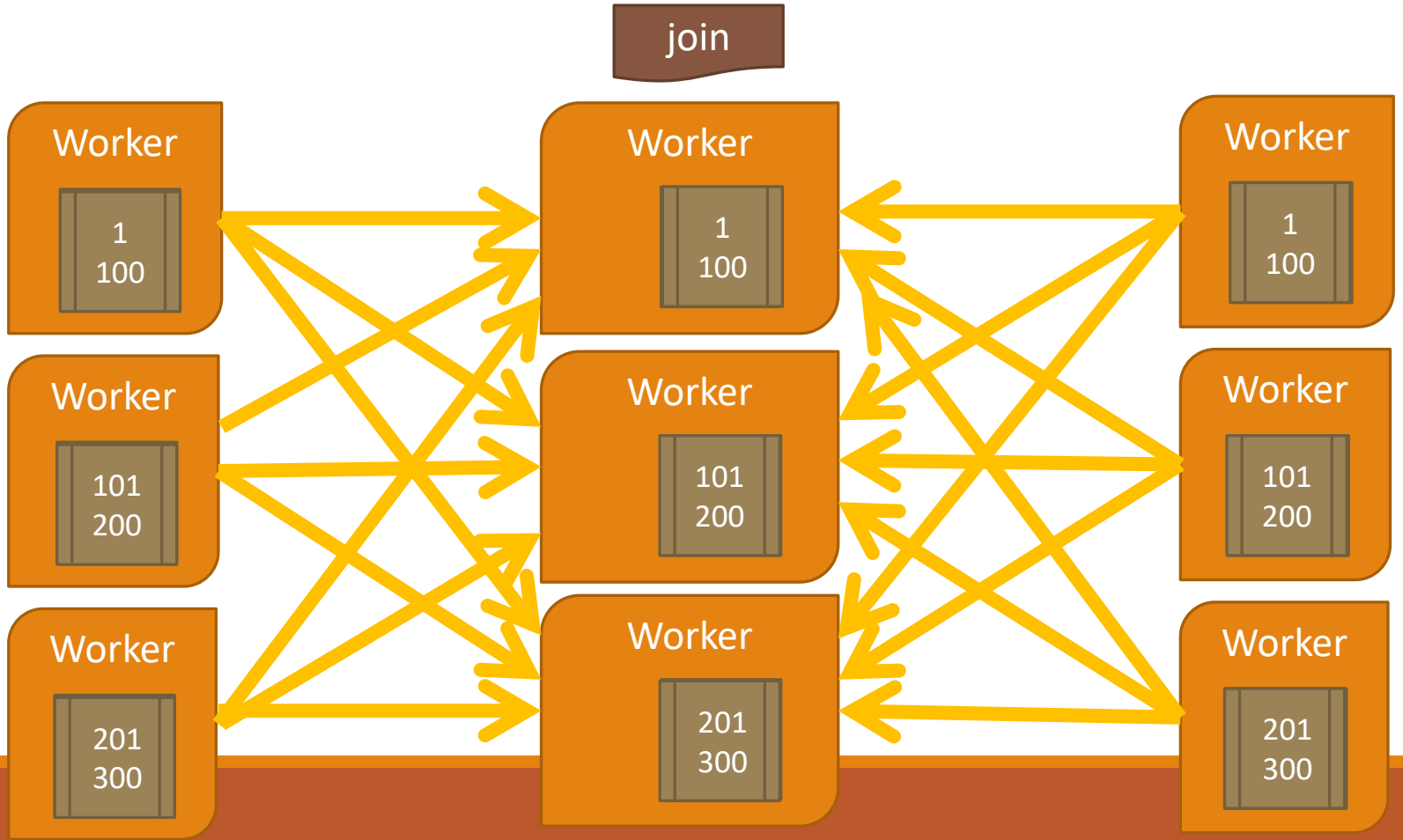
El particionado de RDDs de la forma clave-valor puede ser realizado en función de cada clave.

Aunque no hay manera de decirle a Spark que clave tiene que procesar un determinado nodo, si es posible almacenar subconjuntos de claves en el mismo nodo.

- Particionado por hash (HashPartitioner)
- Particionado por intervalos (RangePartitioner)
- Particionando de manera personalizada (CustomPartitioner)

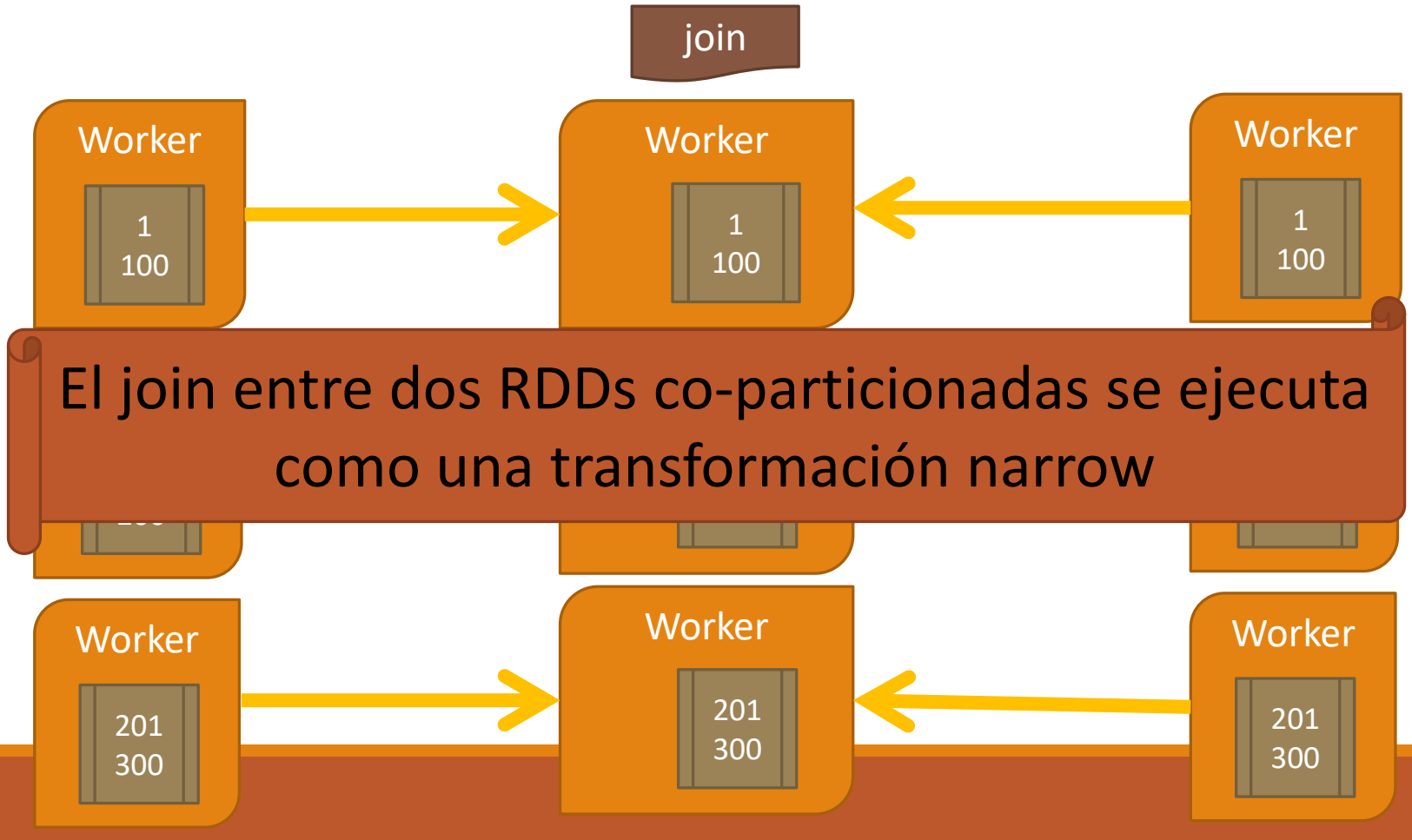
Particionado de datos

Minimiza el tráfico de datos en el cluster



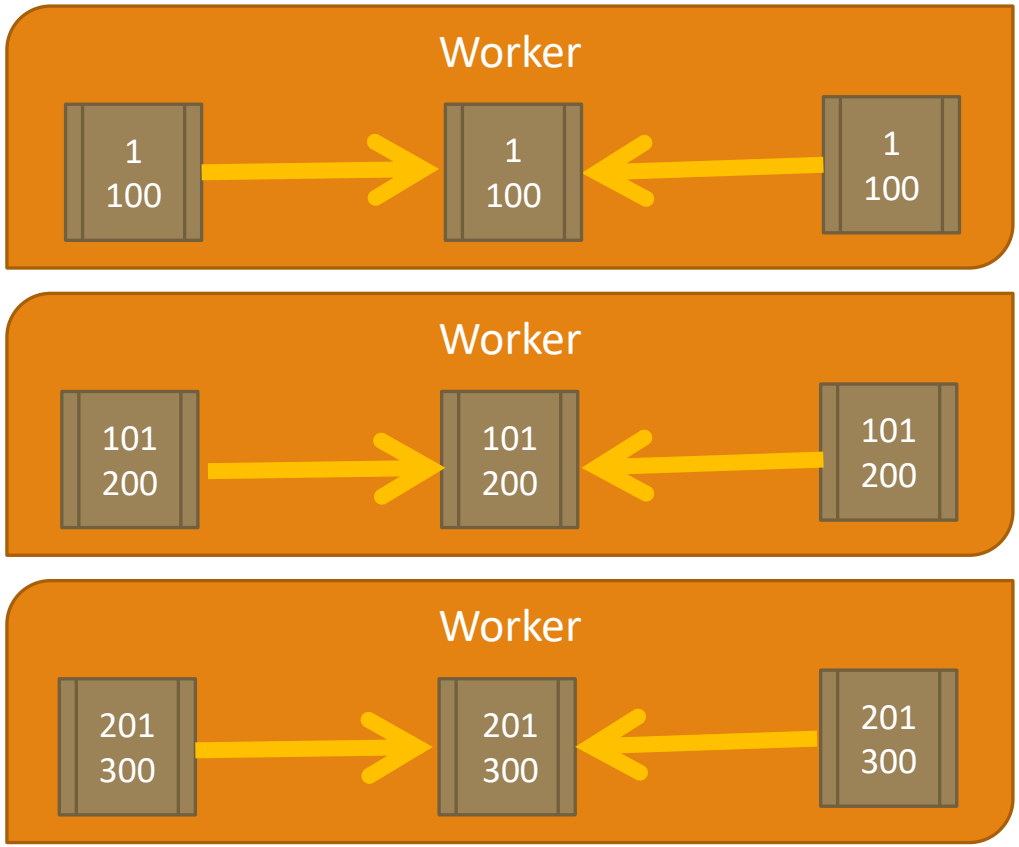
Particionado de datos

Minimiza el tráfico de datos en el cluster



Particionado de datos

Minimiza el tráfico de datos en el cluster



Transformación partitionBy

Brinda información a Spark sobre como almacenar juntas un subconjunto de claves en el mismo nodo.

Recibe como parámetro la cantidad de particiones a crear y una función (opcional) que dice como agrupar las claves.

Transformación partitionBy

```
cli = clientes.partitionBy(3)


parts = cli.keys().glom().collect()
for p in parts:
  print(set(p))
```

{'ARG', 'ITA', 'ESP', 'VEN'}
{'PER', 'BRA', 'BOL'}
{'COL', 'PAR', 'CHI', 'URU', 'ECU'}

Transformación partitionBy

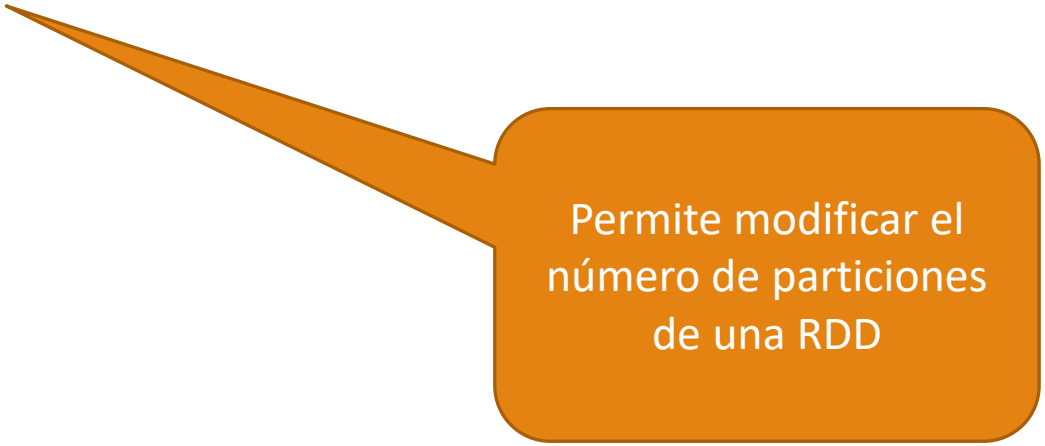
```
def porZonas(k):
  if(k in ["ESP", "ITA"]):
    return 1
  elif(k in ["ARG", "URU", "CHI", "BRA", "PAR", "BOL"]):
    return 2
  else:
    return 3


cli = clientes.partitionBy(3, p


cli = cli.persist()
```

{'ECU', 'PER', 'COL', 'VEN'}
{'ITA', 'ESP'}
{'PAR', 'CHI', 'URU', 'BOL', 'BRA', 'ARG'}

Particionado de datos

repartition

coalesce



Particionado de datos

repartition

coalesce