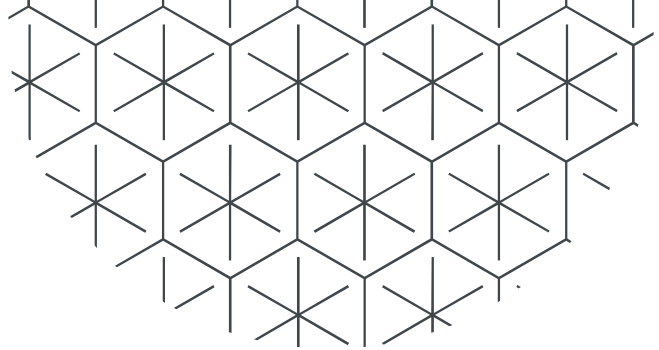
# Tecnologías de datos masivos

Doble Grado en Ingeniería en Tecnologías de Telecomunicación y Business Analytics



# SparkContext

# Spark - SparkContext

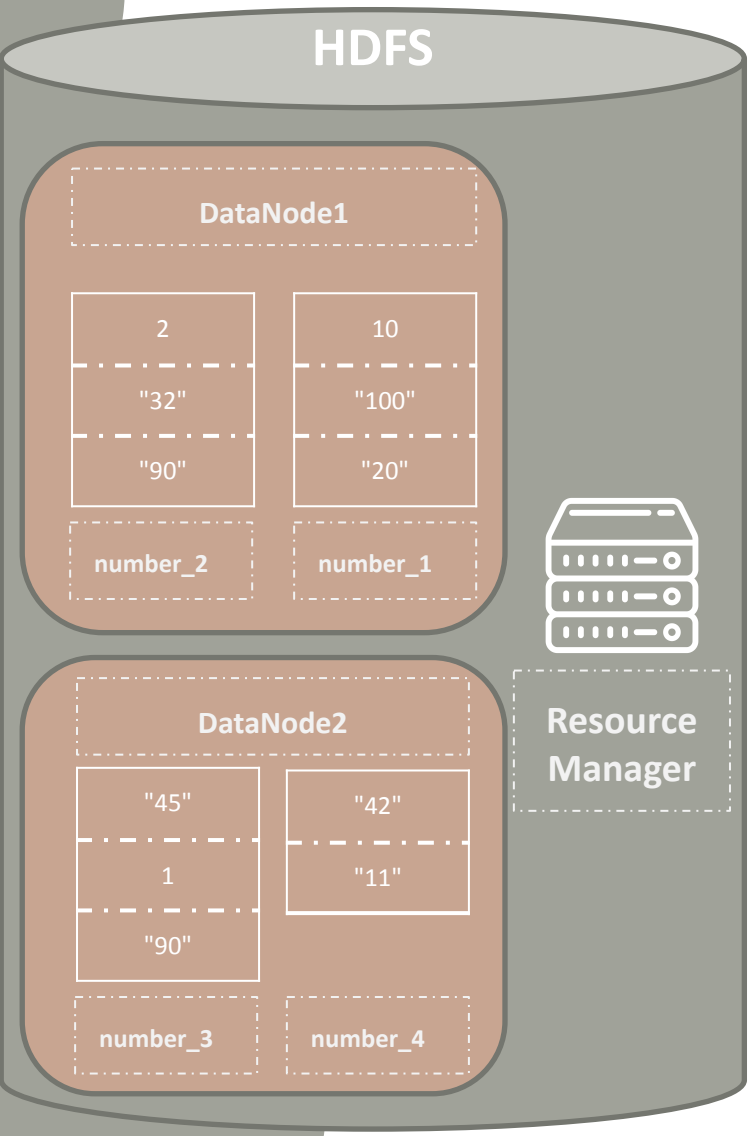
- La interacción de un cluster de Spark con su gestor de recursos se realizada mediante un objeto creado dentro de cada **application** llamando **SparkContext**
- Sólo puede haber un **SparkContext** por cada aplicación de Spark
- Se crea en el Driver de Spark
- Es el encargado de hacer la petición de recursos al gestor de recursos para su **application**
- Es el encargado de comunicarle al gestor de recursos que **baseRDD** es el que va a iniciar un **Job** de Spark
- Cuando acaba cada application de Spark se tiene que llamar a un método stop para que libere los recursos solicitados al gestor del recursos

# Spark - SparkContext

```
from pyspark import SparkConf
from pyspark import SparkContext
from pyspark.sql import SparkSession

conf = SparkConf()
conf.setMaster('yarn-client')
conf.setAppName('jlopezmalla-test')
conf.set("spark.executor.instances", "3")
conf.set("spark.executor.cores", "2")

sc = SparkContext(conf=conf)
```

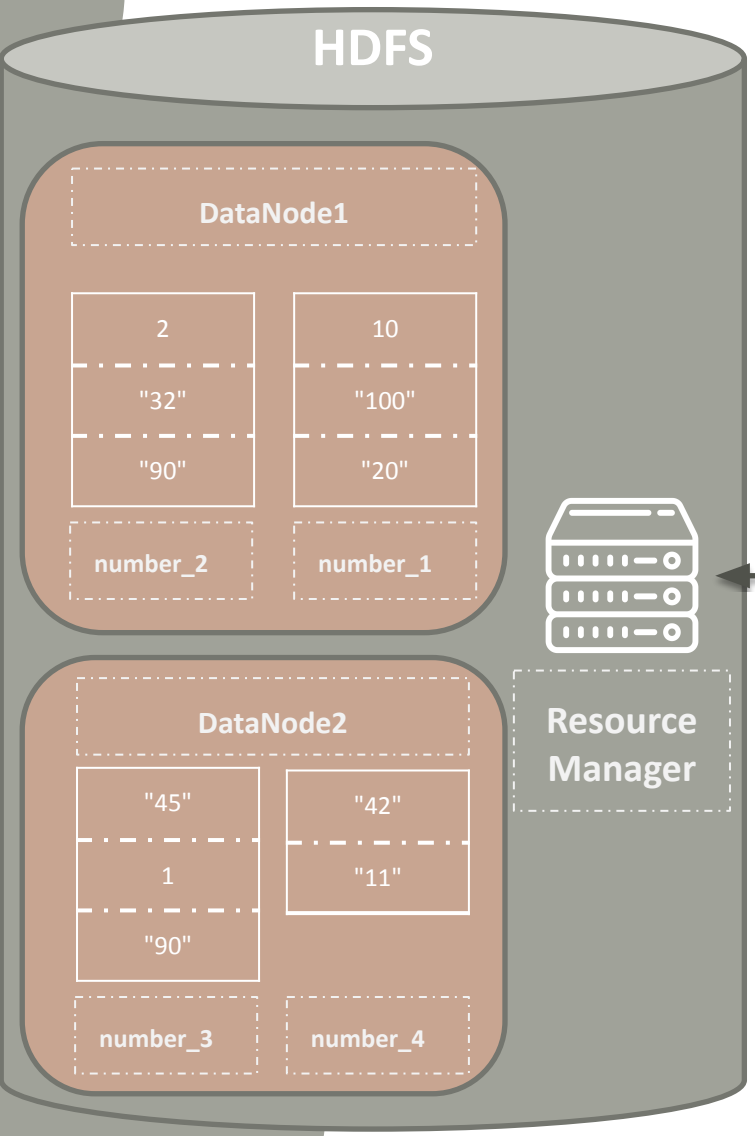
# Spark - SparkContext

```
>>>
```

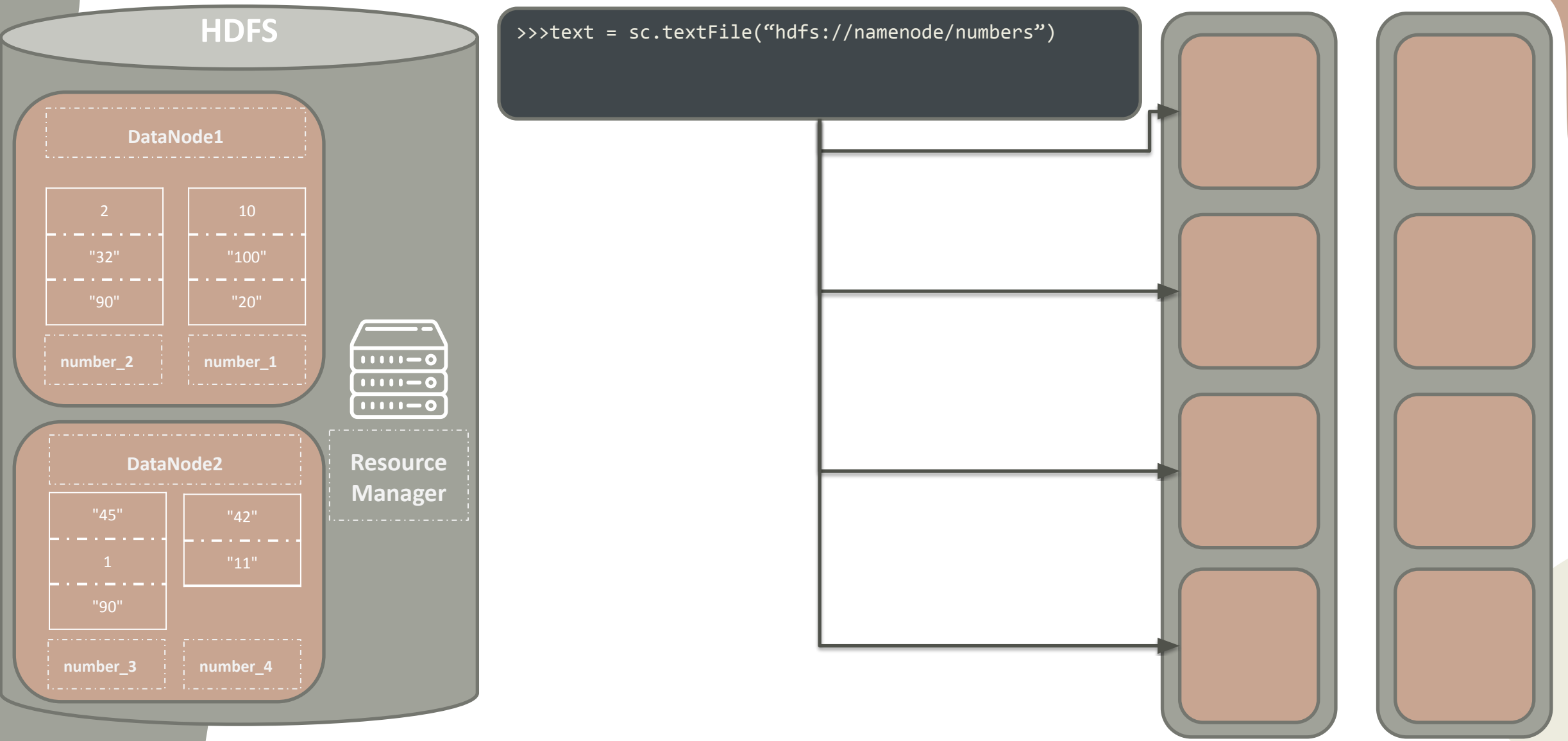


# Spark - SparkContext

```
>>>text = sc.textFile(“hdfs://namenode/numbers”)
MapPartitionsRDD[1] at textFile at
NativeMethodAccessorImpl.java:0
```

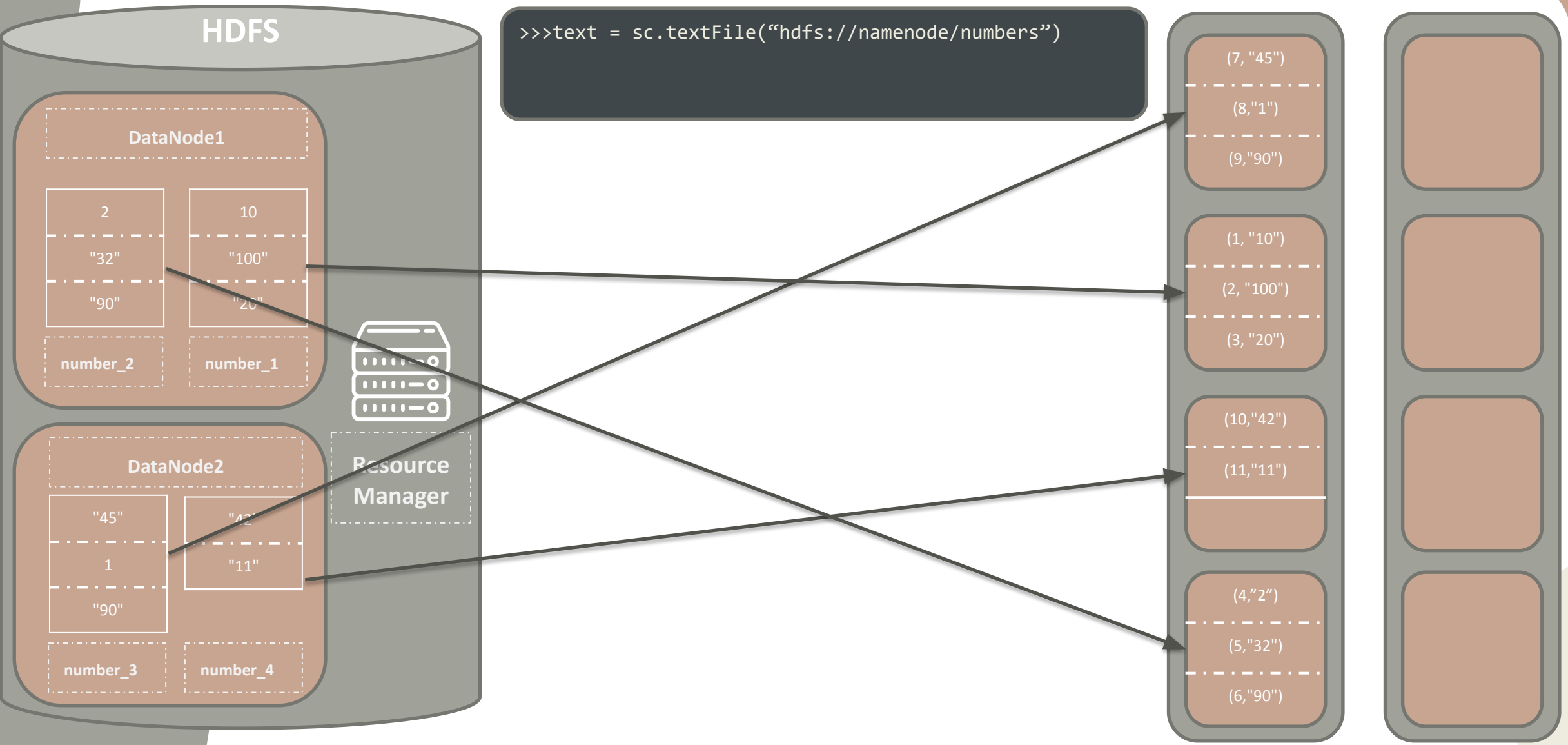


# Spark - SparkContext



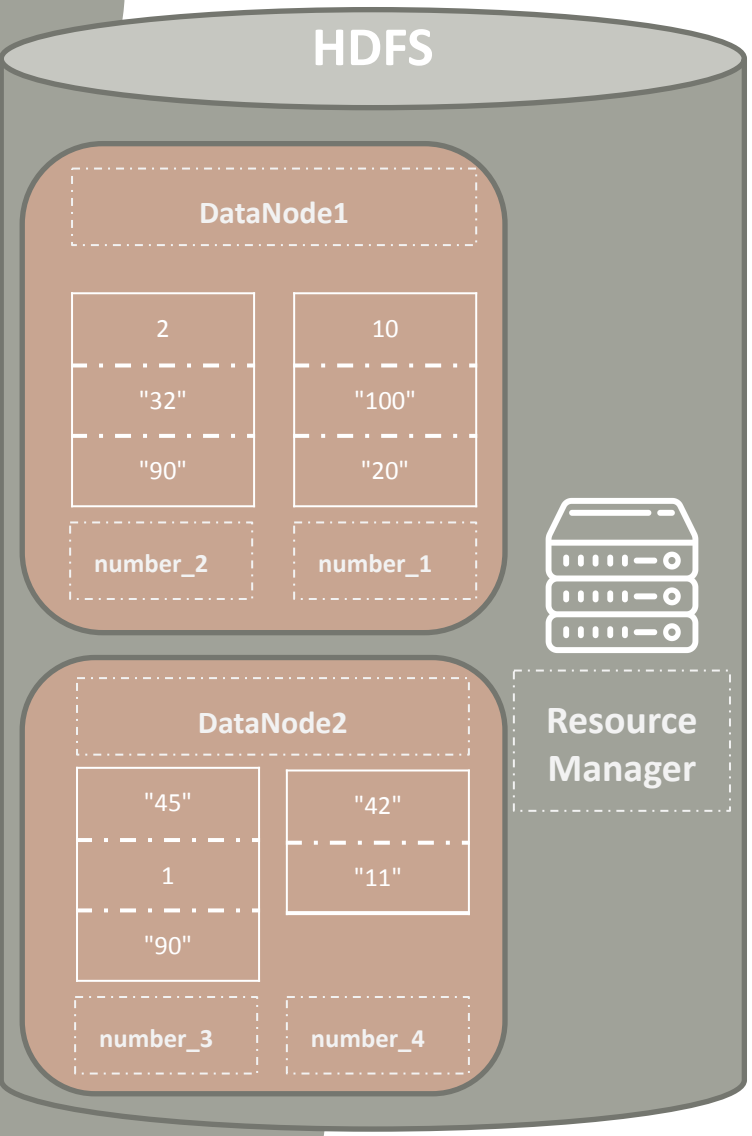
# Spark - SparkContext

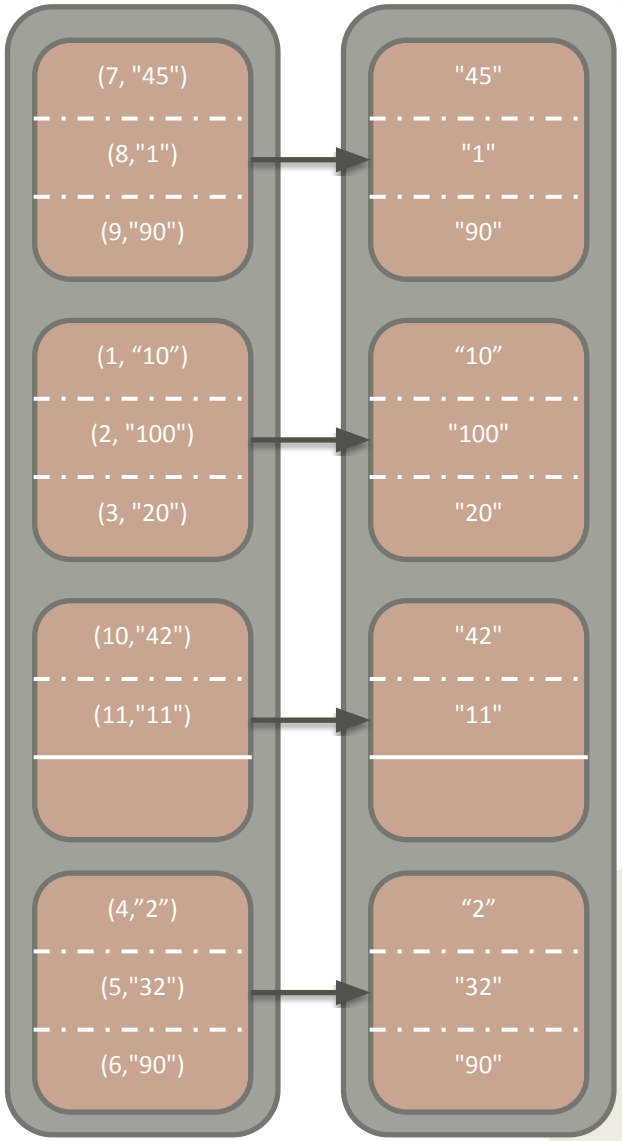
```
>>>text = sc.textFile("hdfs://namenode/numbers")
```



# Spark - SparkContext

```
>>>text = sc.textFile(“hdfs://namenode/numbers”)
```



# Spark - partitions

- Las particiones de todo un **job**, normalmente se determinan a partir del **baseRDD**
- Es de vital importancia que los datos estén bien particionados en el repositorio de información inicial para que todo el proceso sea óptimo
- Si este no fuera el caso, se pueden llamar a dos métodos llamados **coalesce** y **repartition** que modifican la cantidad de **partitions** de un RDD

# Transformaciones y acciones

# Spark - Transformaciones y acciones

- En Spark hay dos tipos de operaciones que se pueden realizar sobre los **RDD**
    - **Transformaciones**
        - Son todas aquellas operaciones que se pueden realizar sobre un **RDD** que devuelven otro **RDD**
        - Dentro de estas operaciones hay de dos tipos
            - **Narrow** que formarán parte de un **stage** y no llamarán a la fase de shuffle
            - **Wide:** que marcarán el fin de un **stage** y el inicio del siguiente llamando a la fase de shuffle
        - Todo lo que programamos en una transformación se ejecuta en los **executor** y tiene como entrada **un único elemento de cualquier tipo, incluidos lista o una tupla, de nuestro RDD**

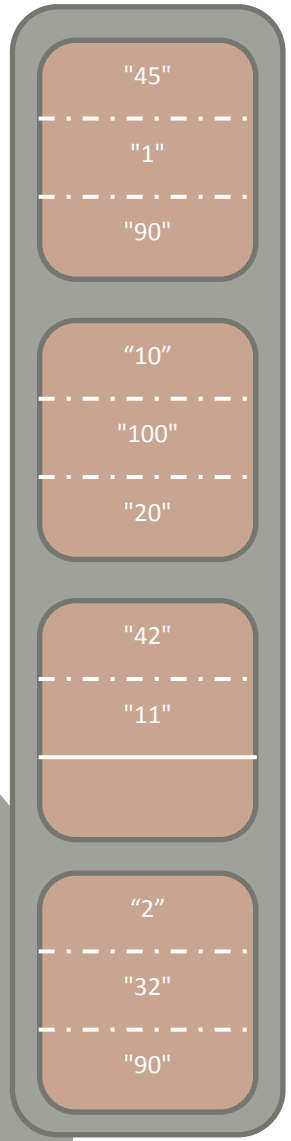
# Spark - Transformaciones y acciones

- En Spark hay dos tipos de operaciones que se pueden realizar sobre los **RDD**
    - **Acciones**
        - Son todas aquellas operaciones que se pueden realizar sobre un **RDD** que NO devuelven otro **RDD**
        - Son las operaciones que determinan el fin de un **job**.
        - Son las operaciones que hacen que se lance la ejecución de las transformaciones
        - Pueden ser potencialmente peligrosas dado que tienen la capacidad de enviar todos los datos de un **RDD** al **Driver**

# Spark - map

- La función Map es similar a las operaciones Map de Map & Reduce
- Es del tipo **Narrow** por lo que no acaba un stage ni llama a Shuffle
- Los datos de salida no tiene porque ser del mismo tupo de los de entrada, ni ser clave valor
- Se realiza la misma operación a todos los elementos del **RDD**
- El resultado tendrá exactamente el mismo número de resultados
- map(f: (V) ⇒ U) : RDD[U]

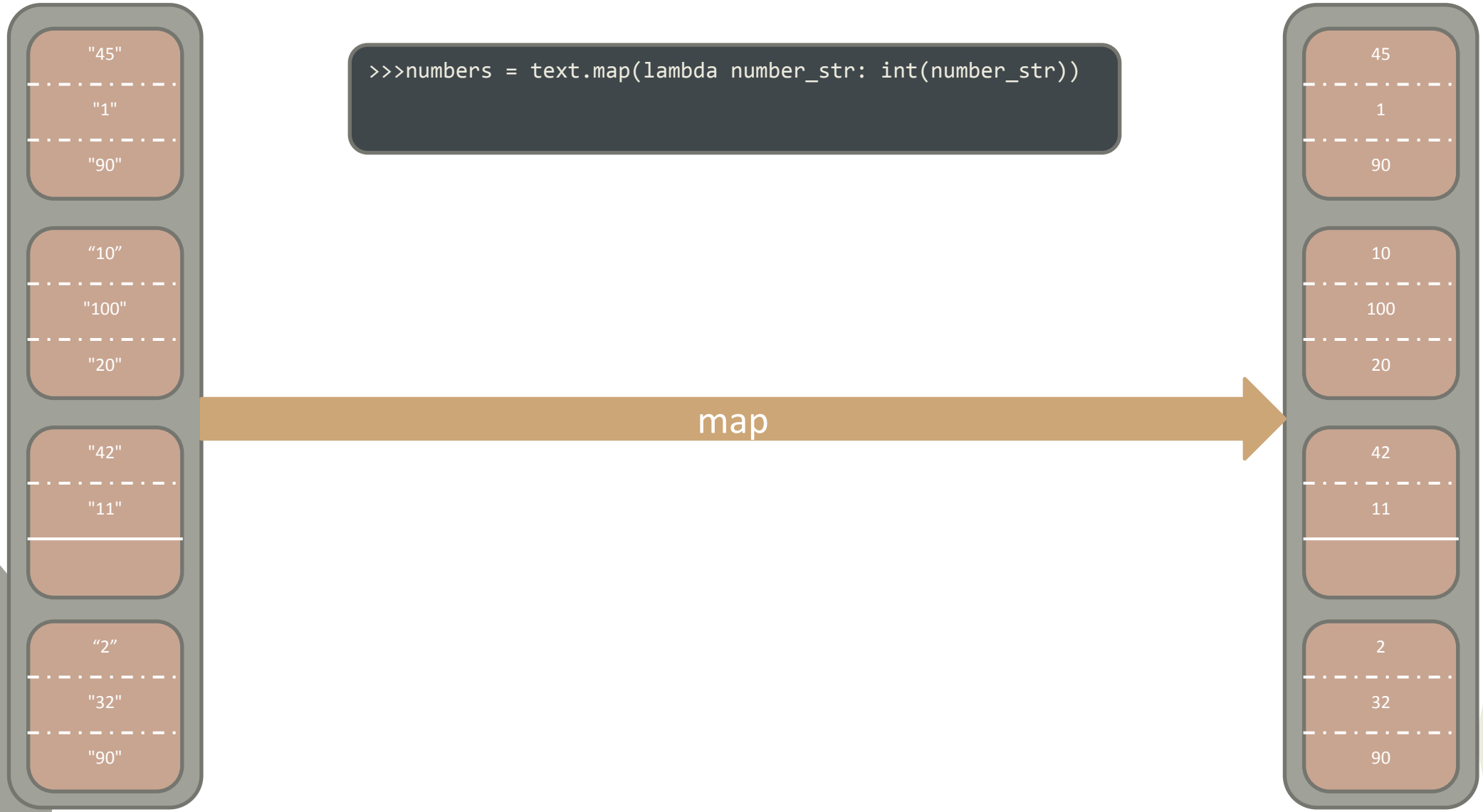
# Spark - map



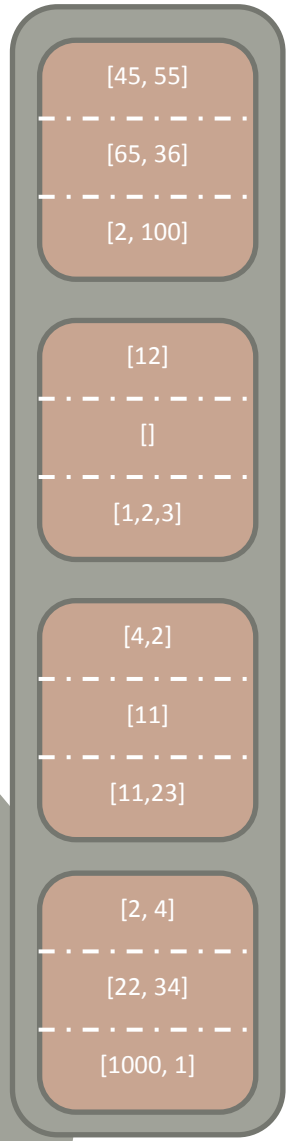

```
>>>numbers = text.map(lambda number_str: int(number_str))
```

# Spark - map

```
>>>numbers = text.map(lambda number_str: int(number_str))
```



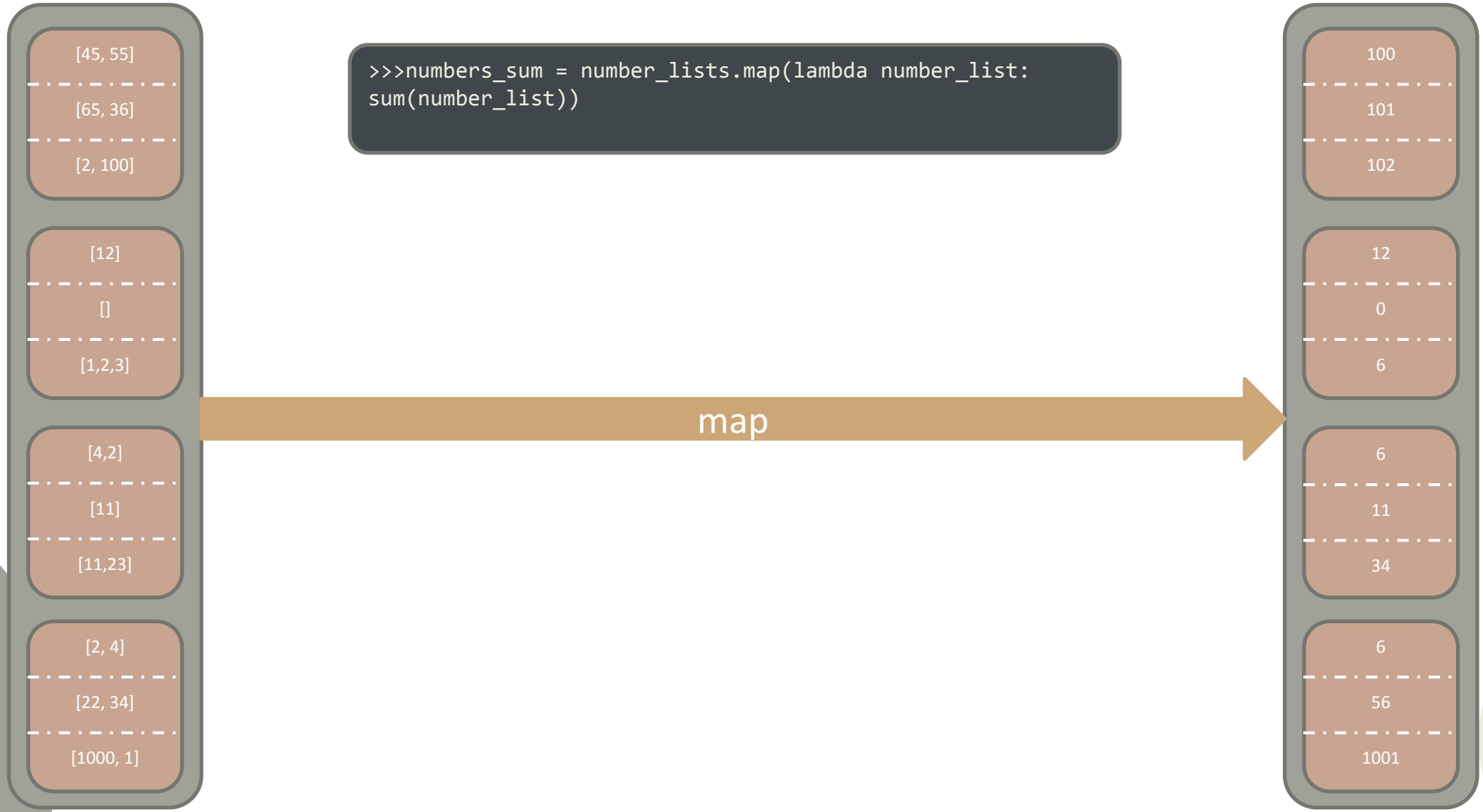
# Spark - map




```
>>>numbers_sum = number_lists.map(lambda number_list:
sum(number_list))
```

# Spark - map

```
>>>numbers_sum = number_lists.map(lambda number_list:
sum(number_list))
```

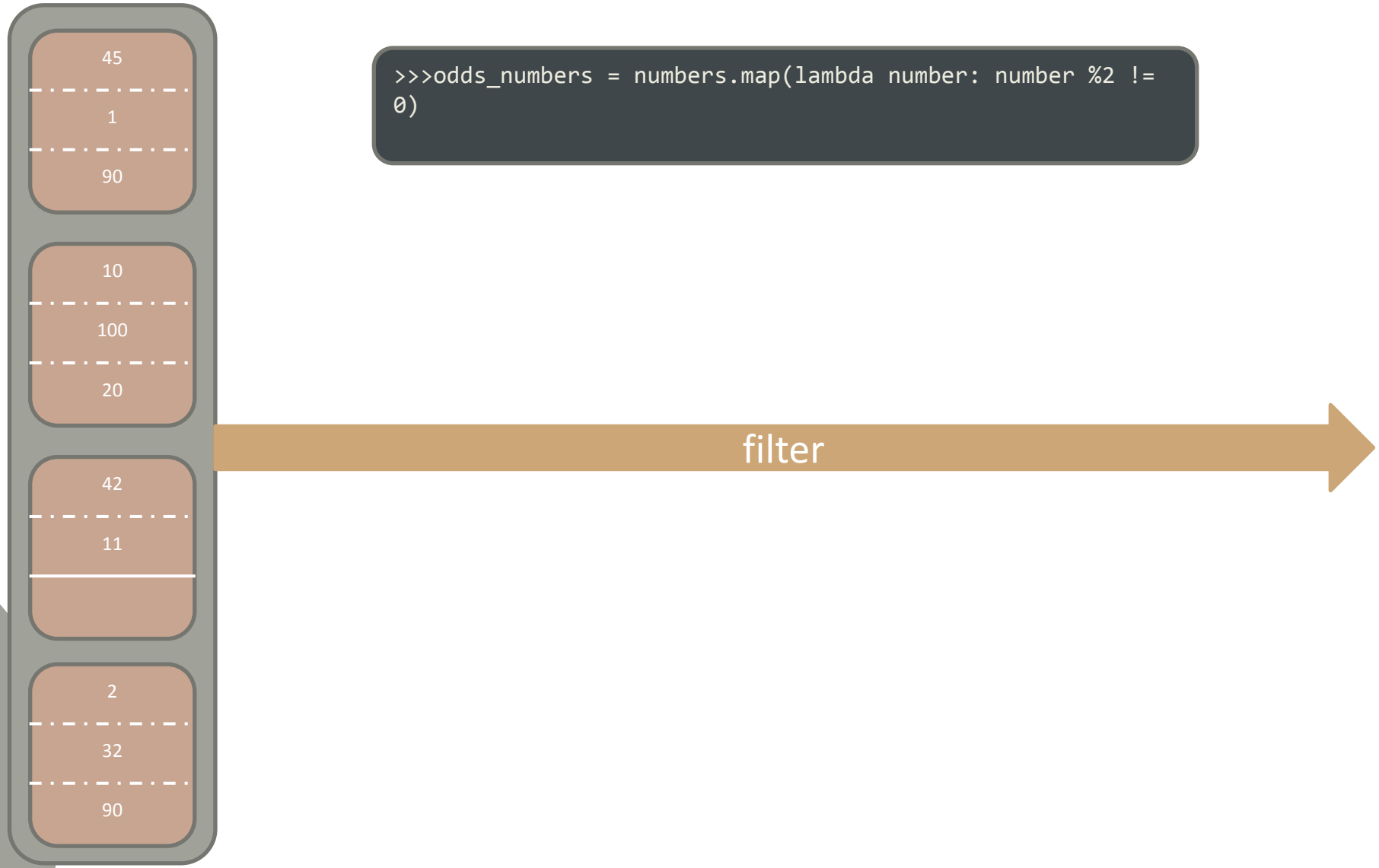


# Spark - filter

- La función filter filtra aquellos registros que cumplan la condición de la función pasada por parámetro
- Es del tipo **Narrow** por lo que no acaba un stage ni llama a Shuffle
- Se realiza la misma operación a todos los elementos del **RDD**
- El resultado tendrá, como máximo, el mismo número de resultados
- filter(f: (U) ⇒ Boolean) : RDD[U]

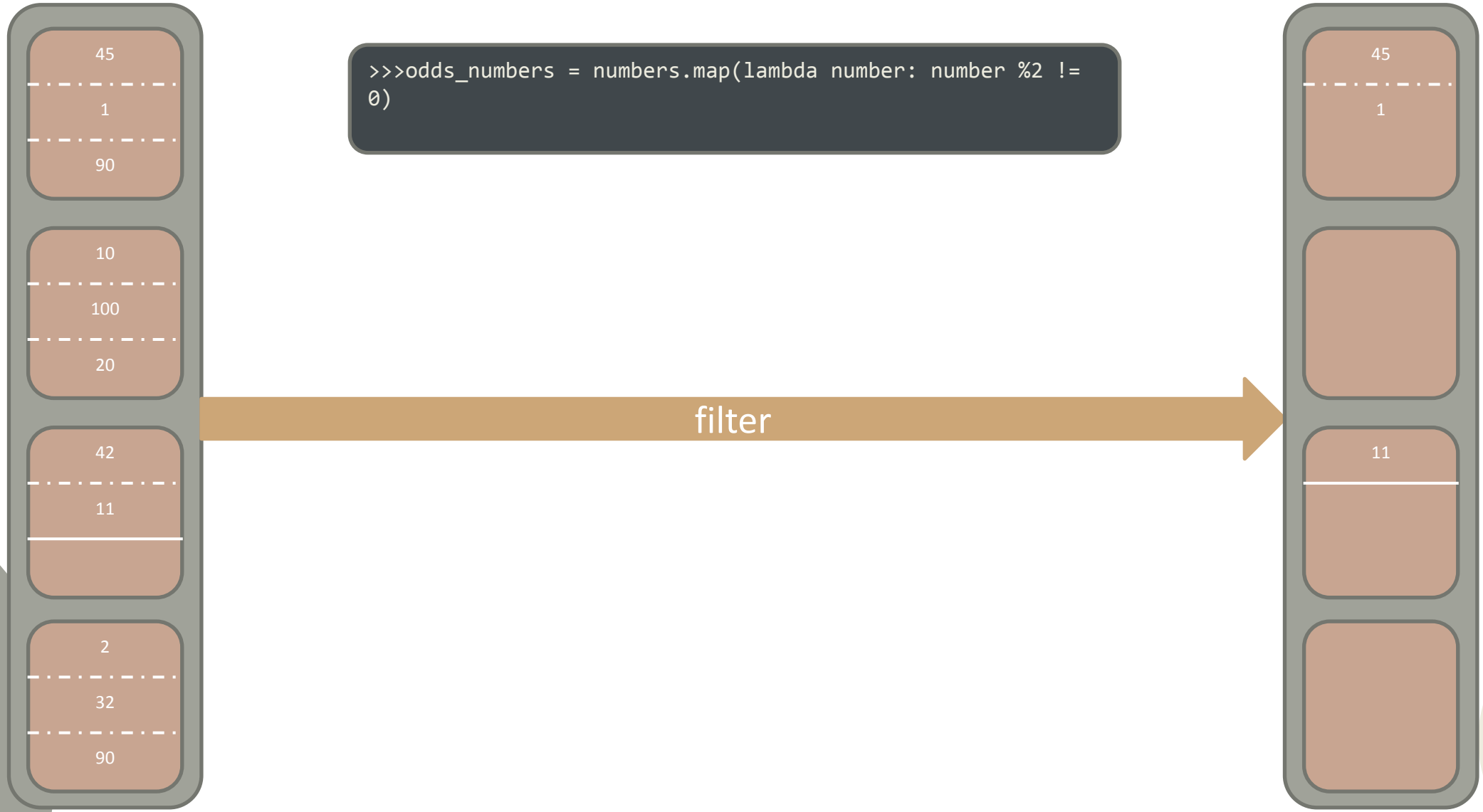
# Spark - filter

```
>>>odds_numbers = numbers.map(lambda number: number %2 != 0)
```



# Spark - filter

```
>>>odds_numbers = numbers.map(lambda number: number %2 != 0)
```



# Spark - filter

```
>>>three_elemnts_array = number_lists.filter(lambda
number_list: len(number_lists) == 3)
```

# Spark - filter

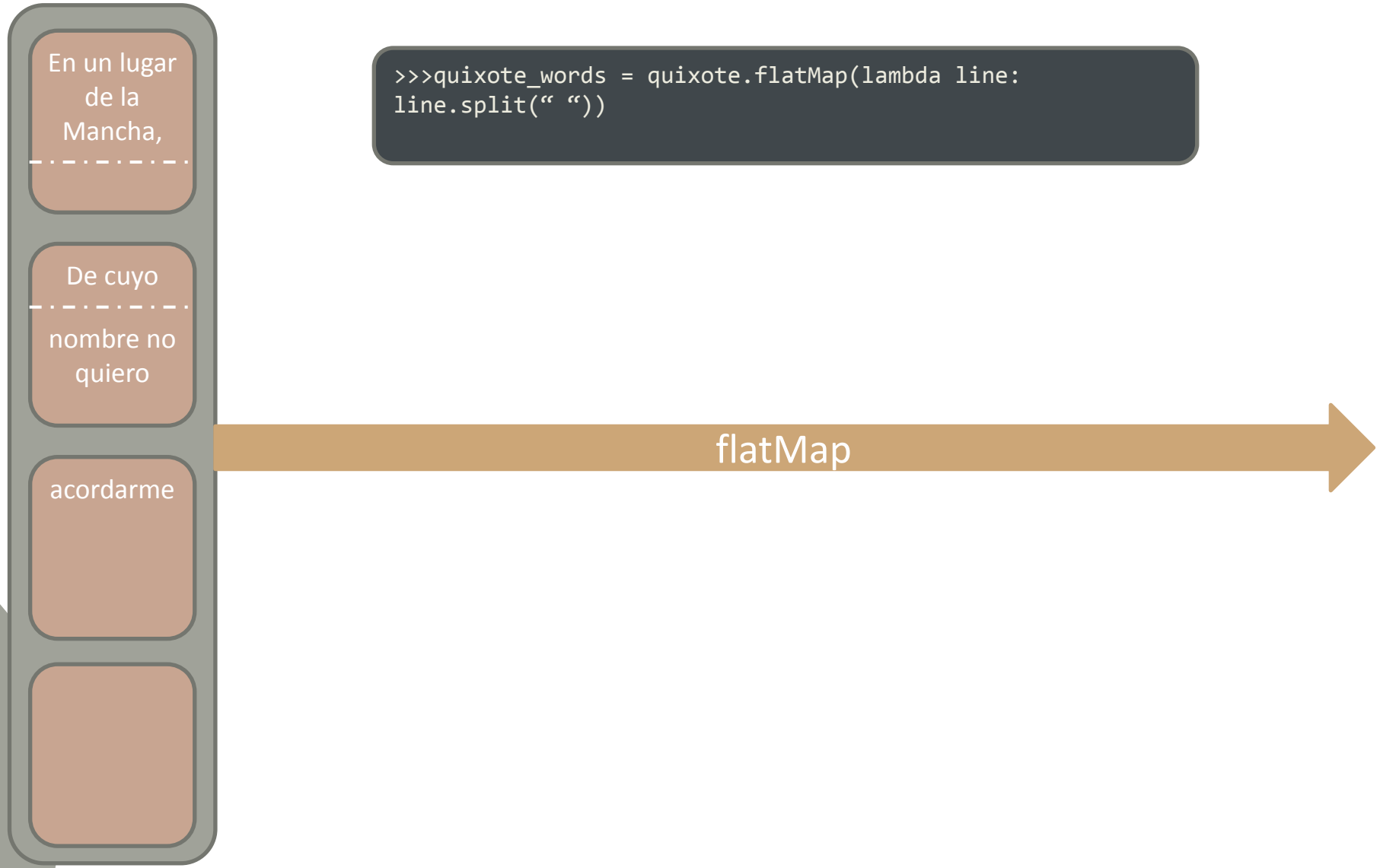
```
>>>three_elemnts_array = number_lists.filter(lambda
number_list: len(number_lists) == 3)
```

# Spark - flatMap

- La función flatMap es similar a las operaciones Map de Map & Reduce
- Es del tipo **Narrow** por lo que no acaba un stage ni llama a Shuffle
- Los datos de salida tienen que ser una lista, el **RDD** resultante tendrá todos los elementos de todas las listas resultantes.
- Se realiza la misma operación a todos los elementos del **RDD**
- El resultado tendrá más, menos o el mismo número de elementos que el **RDD** de entrada
- flatMap(f: (U) ⇒ List[U]) : RDD[U]

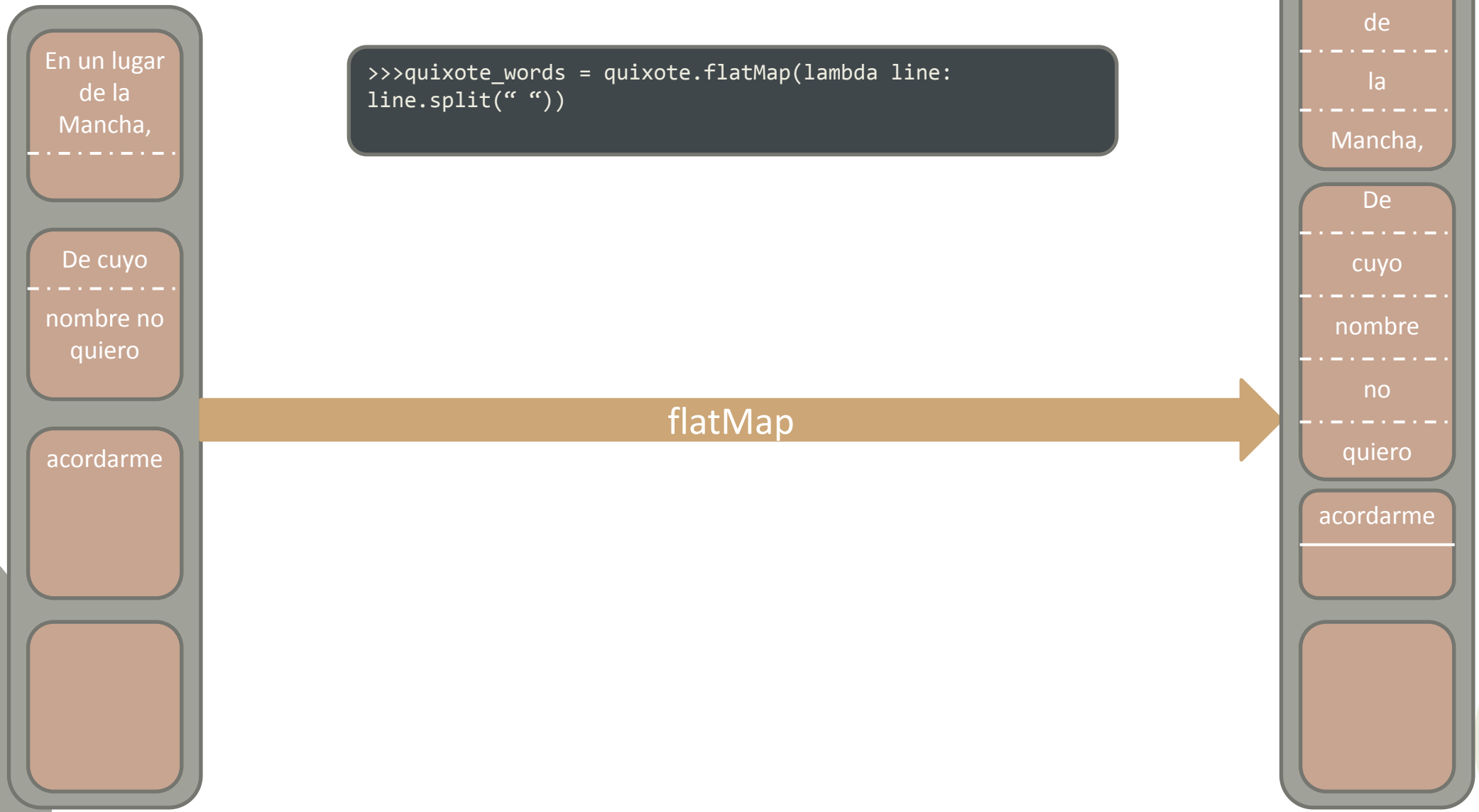
# Spark - flatMap

```
>>>quixote_words = quixote.flatMap(lambda line:
line.split(“ “))
```

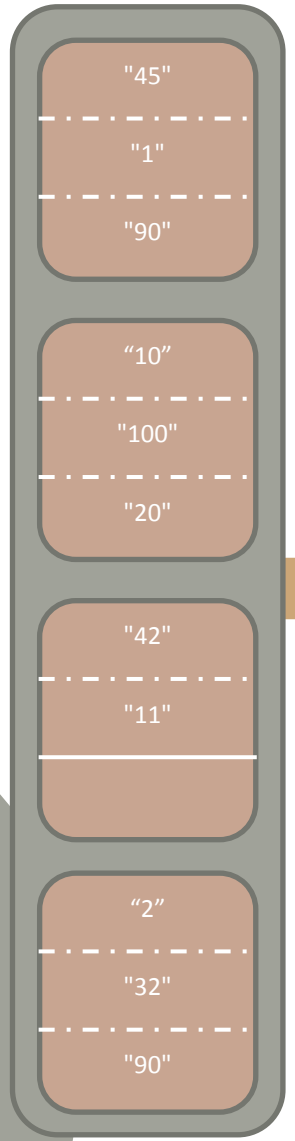


# Spark - flatMap

```
>>>quixote_words = quixote.flatMap(lambda line:
line.split(“ “))
```



# Spark - flatMap




```
>>>def convert_and_filter_odd(number_str):
    number = int(number_str)
    if number % 2 != 0:
       return [number]
    else:
       return []

>>>odd_numbers = text.flatMap(lambda number_str:
convert_and_filter_odd(number_str))
```

flatMap

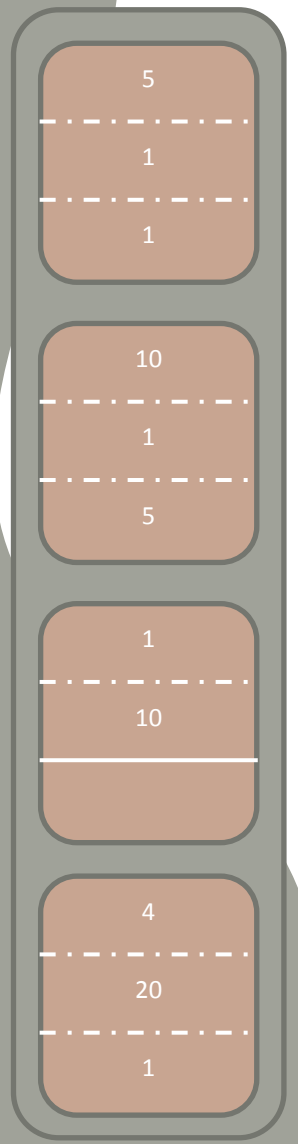
# Spark - flatMap

# Spark - distinct

- La función **disintct** obtiene los distintos elementos del **RDD**
- Es del tipo **Wide** por lo que acaba un stage y llama a Shuffle
- Los datos de salida tendrán el mismo tipo que los de entrada.
- Se realiza la misma operación a todos los elementos del **RDD**
- El resultado tendrá como máximo el mismo número de elementos que el **RDD** de entrada y como mínimo 1
- distinct() : RDD[U]

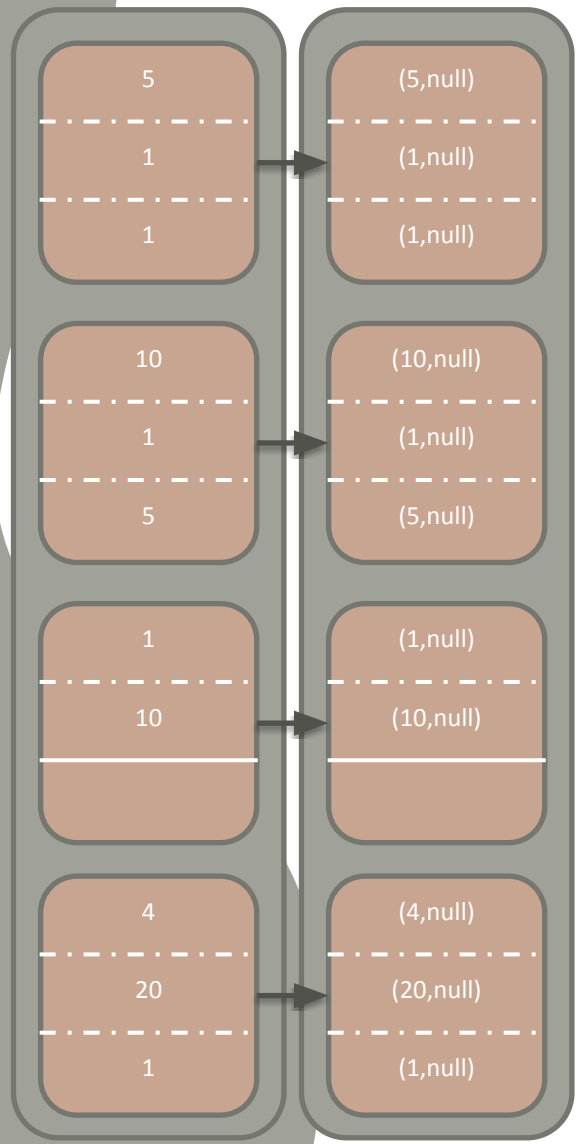
# Spark - distinct

```
>>>distinct_numbers = numbers.distinct()
```

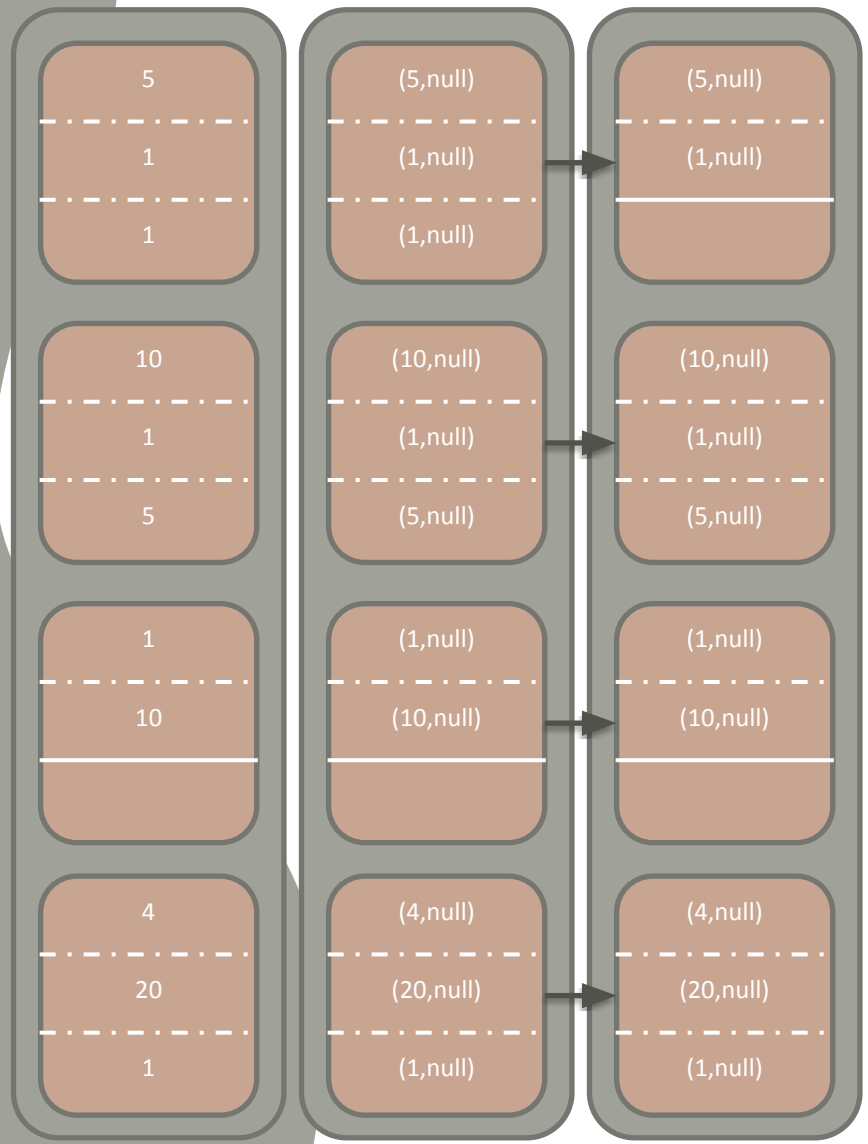


# Spark - distinct

```
>>>distinct_numbers = numbers.distinct()
```

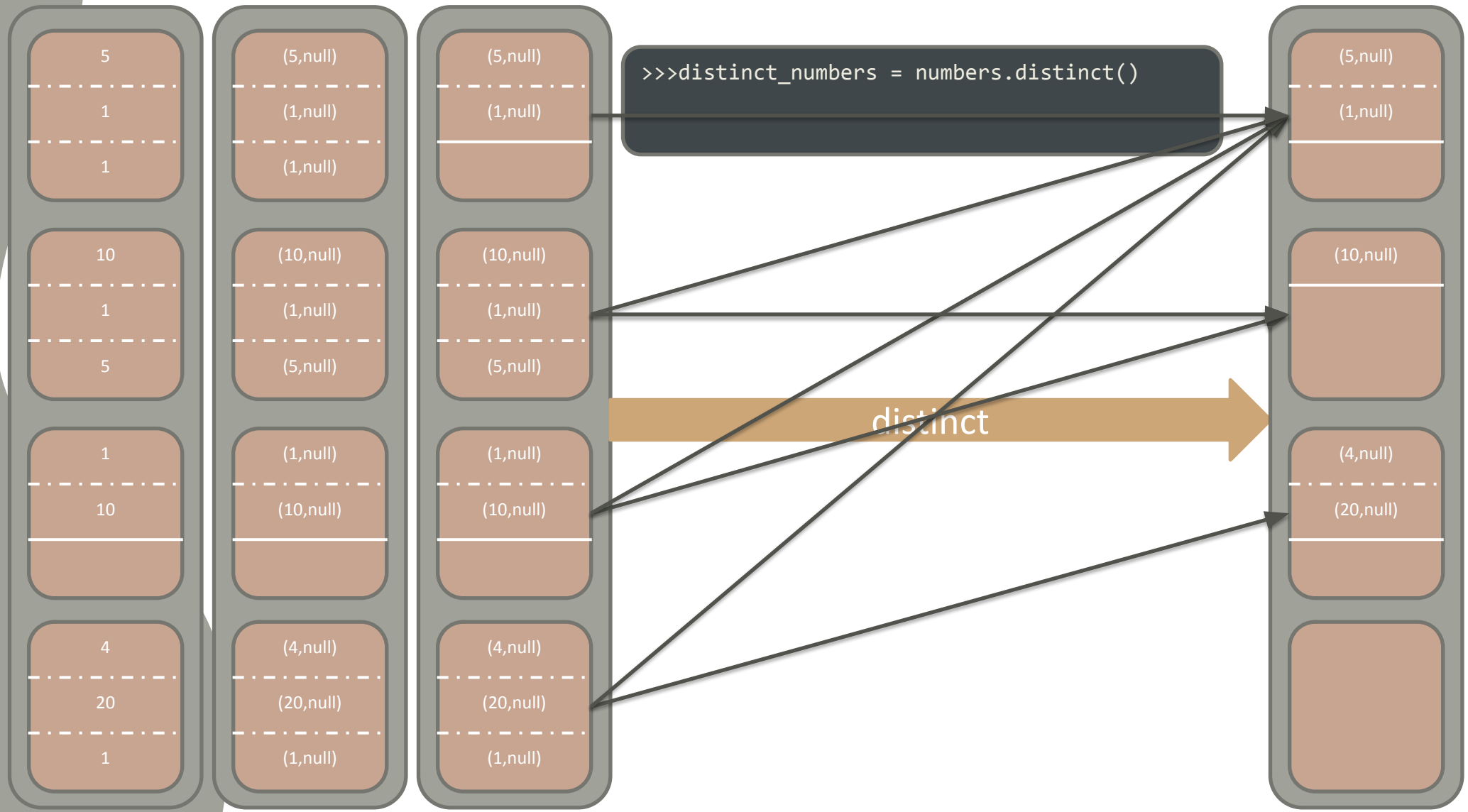


# Spark - distinct



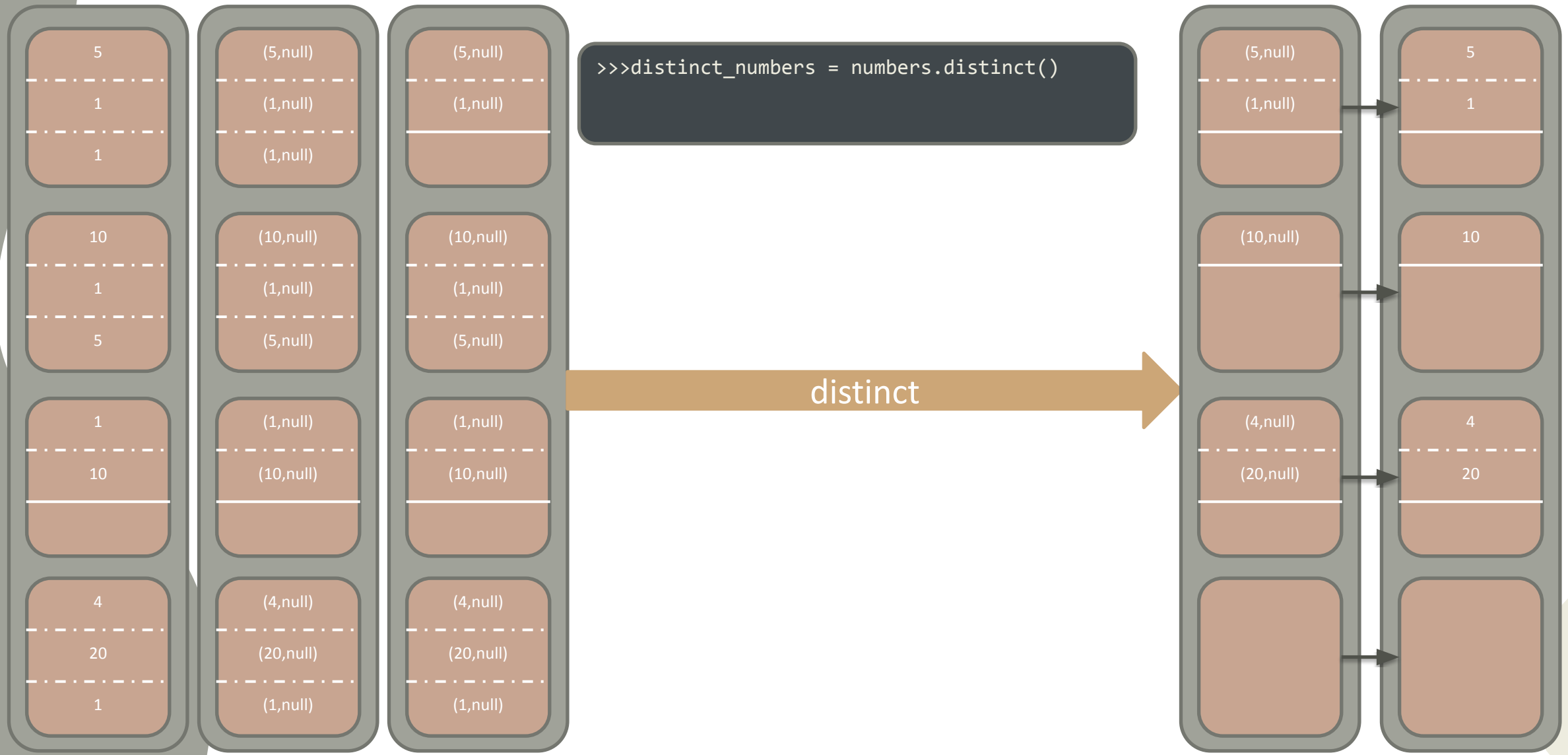

```
>>>distinct_numbers = numbers.distinct()
```

# Spark - distinct



# Spark - distinct

```
>>>distinct_numbers = numbers.distinct()
```

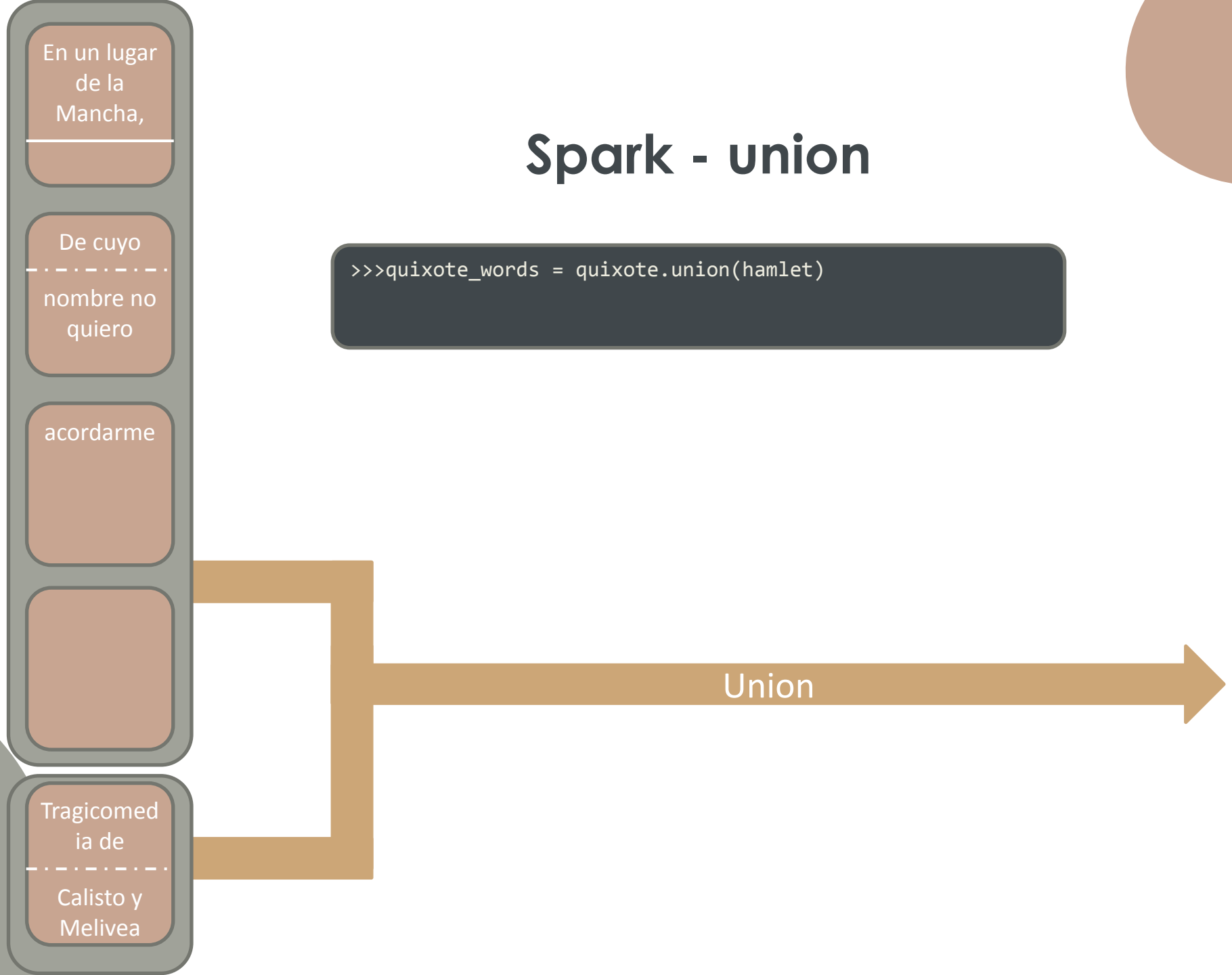


# Spark - union

- La función union une los registros de dos **RDD** en un mismo **RDD**
- Es del tipo **Wide** por lo que acaba un stage y llama a Shuffle
- El **RDD** resultante tiene la suma de los elementos de los RDD y tambien la suma de sus particiones
- union(other:RDD[u]) : RDD[U]

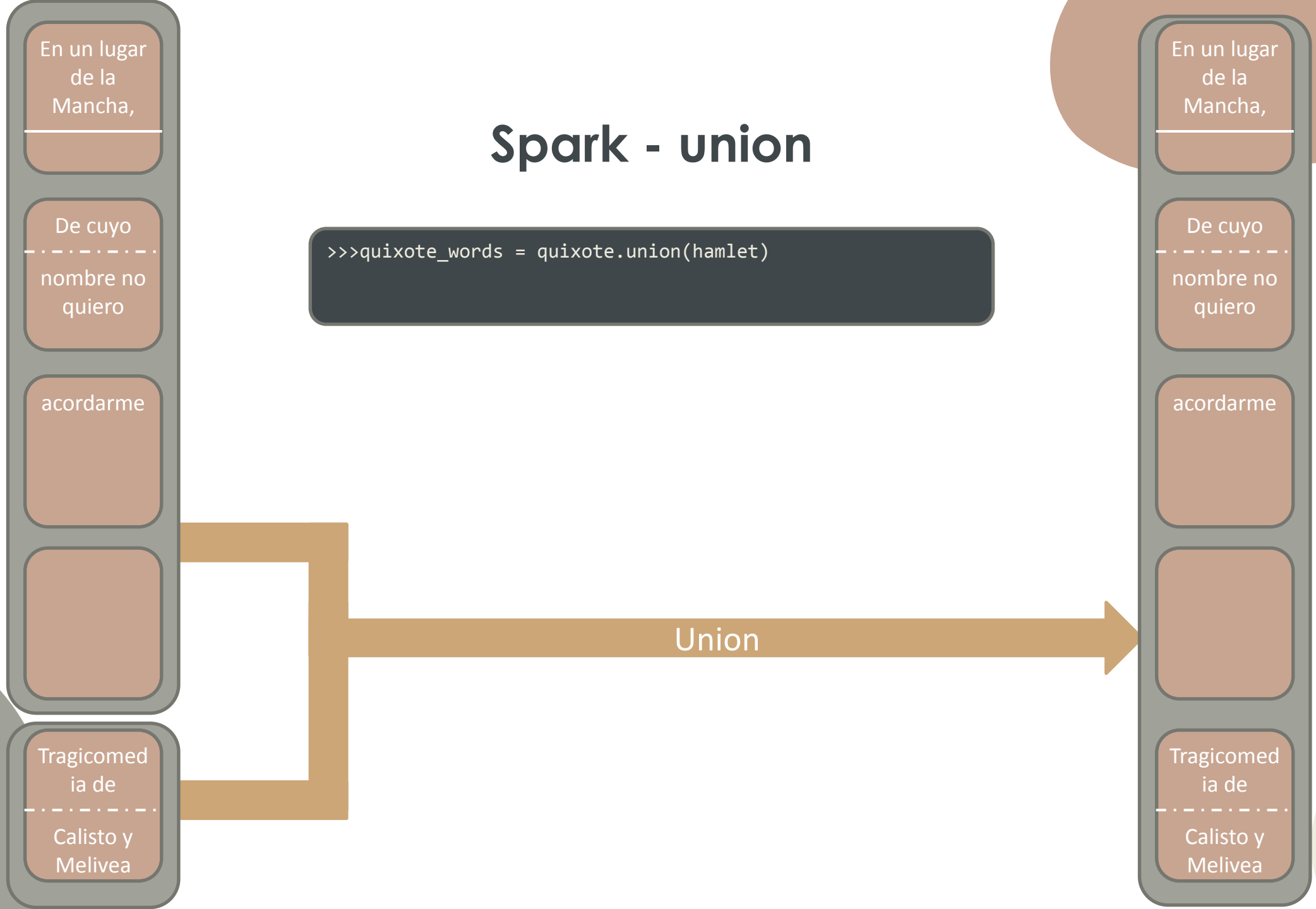
# Spark - union

```
>>>quixote_words = quixote.union(hamlet)
```



# Spark - union

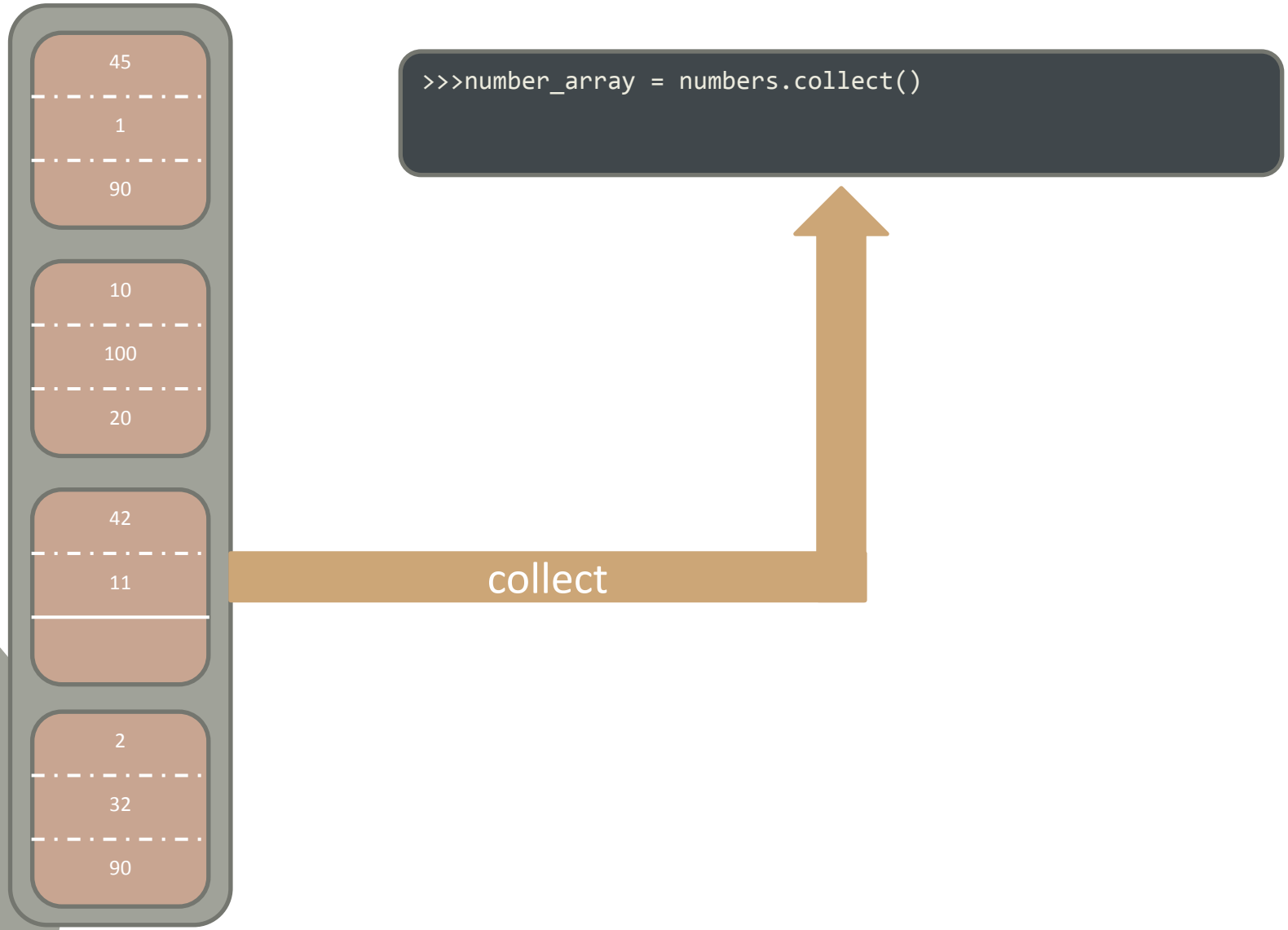
```
>>>quixote_words = quixote.union(hamlet)
```



# Spark - collect

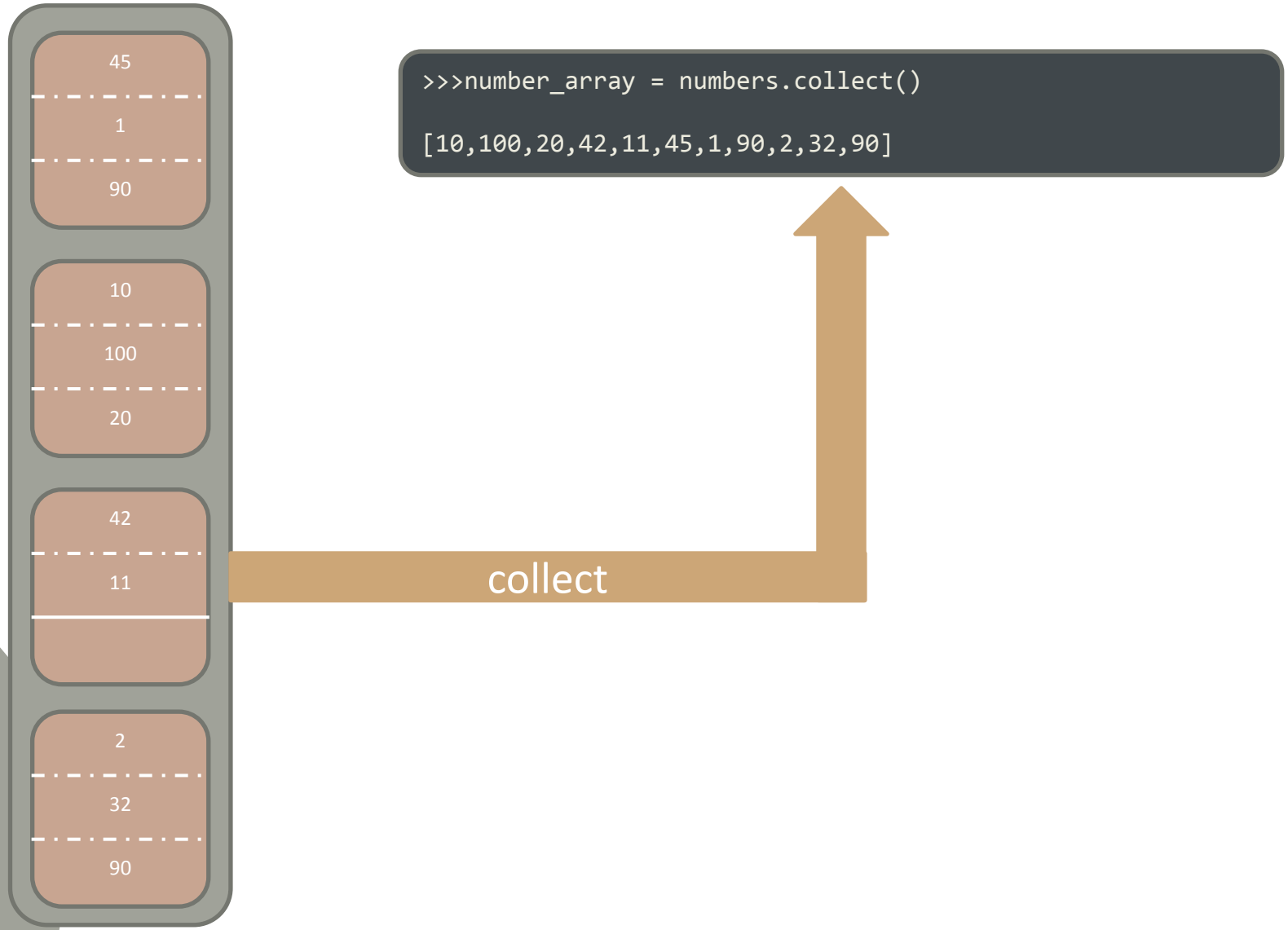
- Es una acción por lo que terminará un **job**
- El resultado será un Array con el mismo número de resultados que el **RDD**
- El **driver** tendrá que tener espacio en memoria RAM suficiente para todos los elementos del **RDD**
- Las **partition** pueden llegar en cualquier orden pero los elementos de una particion siempre tendrán el mismo orden que tenían en el **partition**
- collect() : List[U]

# Spark - collect




```
>>>number_array = numbers.collect()
```

# Spark - collect

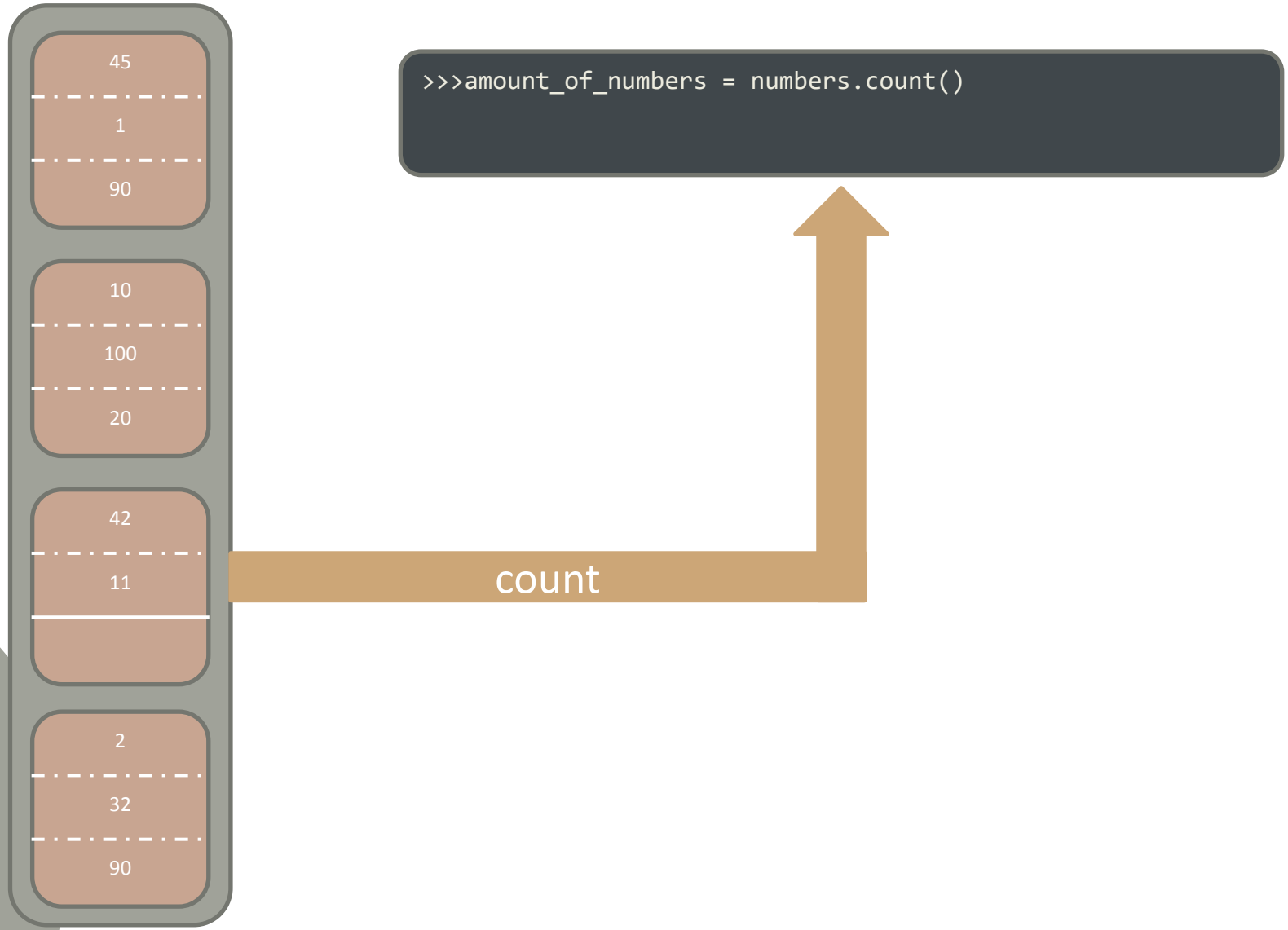



```
>>>number_array = numbers.collect()

[10,100,20,42,11,45,1,90,2,32,90]
```

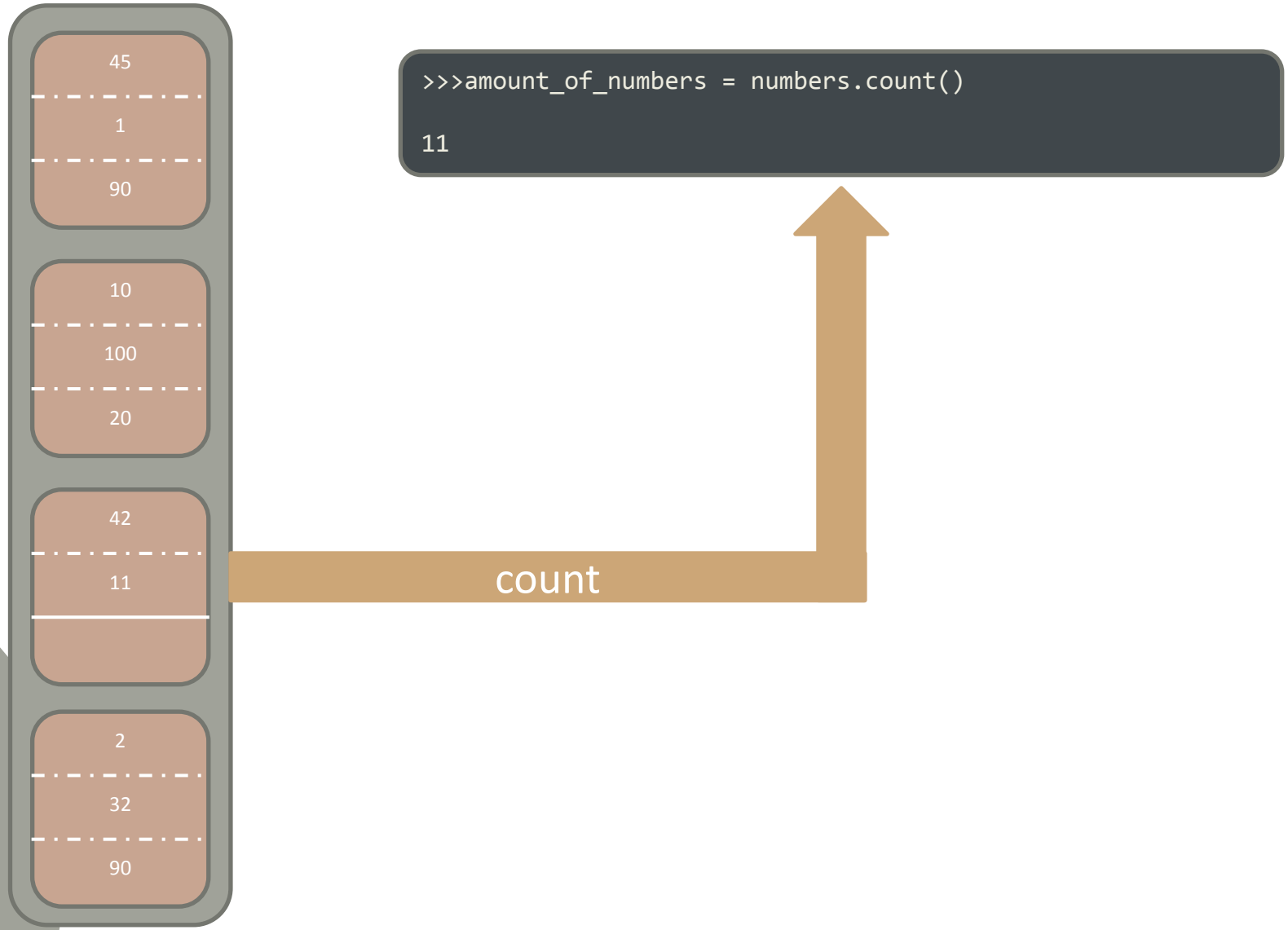
# Spark - count

- Es una acción por lo que terminará un **job**
- Cuenta el número de registros que tiene un **RDD**
- count() : Int

# Spark - count


45
1
90
10
100
20
42
11
2
32
90
>>>amount_of_numbers = numbers.count()
count

# Spark - count

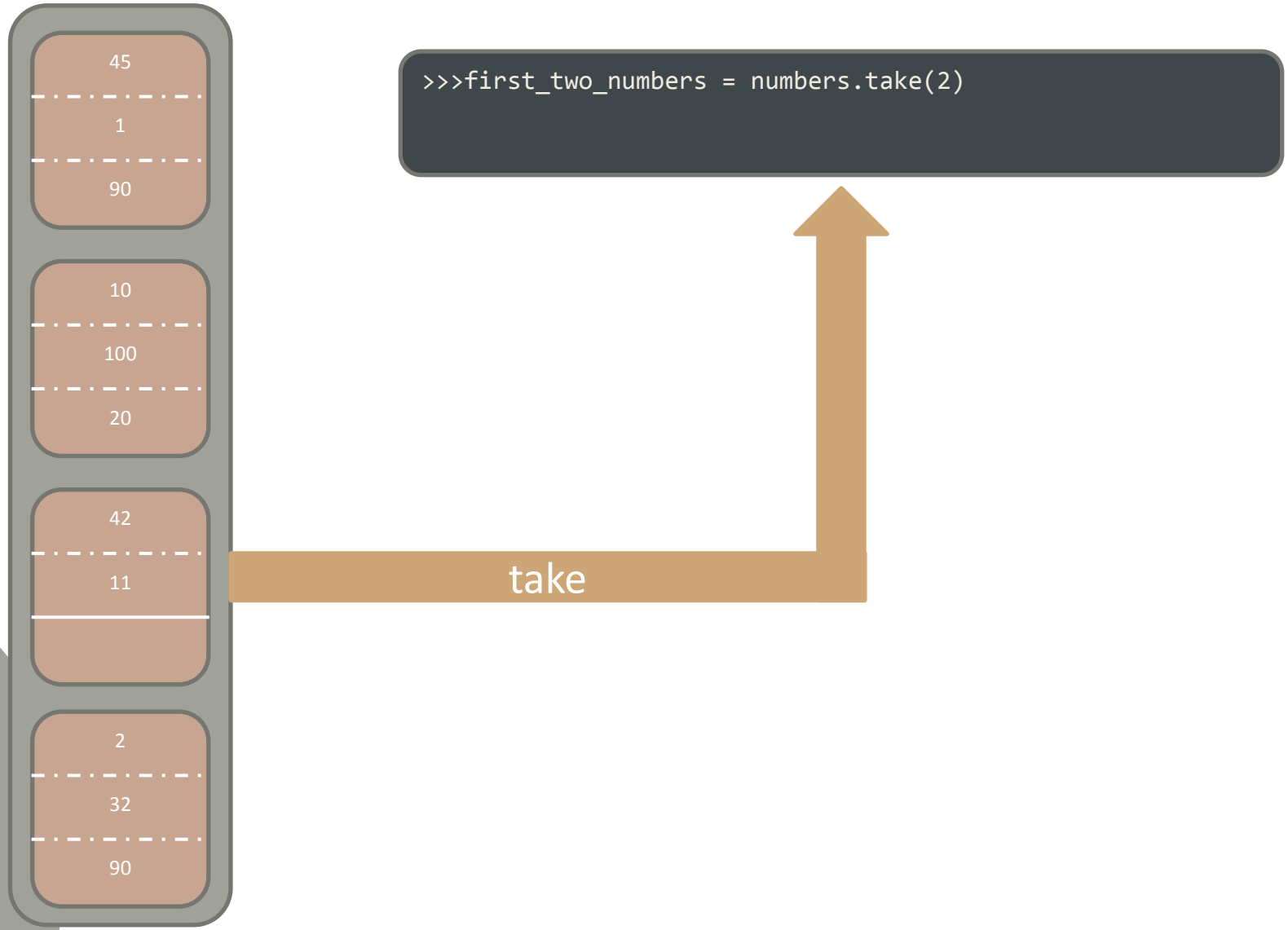



```
>>>amount_of_numbers = numbers.count()

11
```

# Spark - take(n)

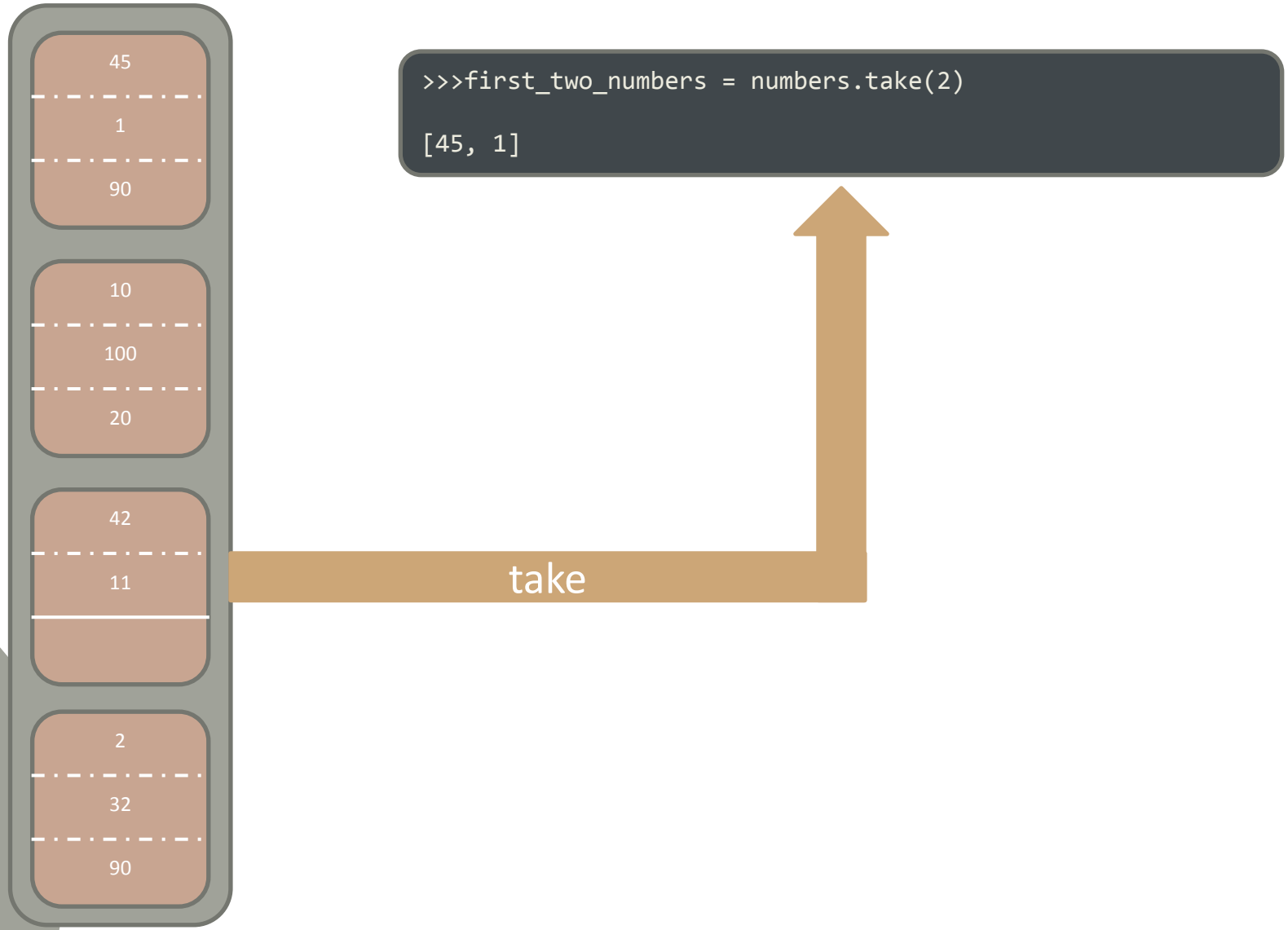
- Es una acción por lo que terminará un **job**
- El resultado será un Array con, como máximo los N primeros elementos del **RDD**
- El **driver** tendrá que tener espacio en memoria RAM suficiente para todos los elementos del **RDD**
- Se irán recogiendo los elementos de los **partitions hasta llegar a n o hasta que se acaben los elementos**
- take(n) : List[U]

# Spark - take(n)



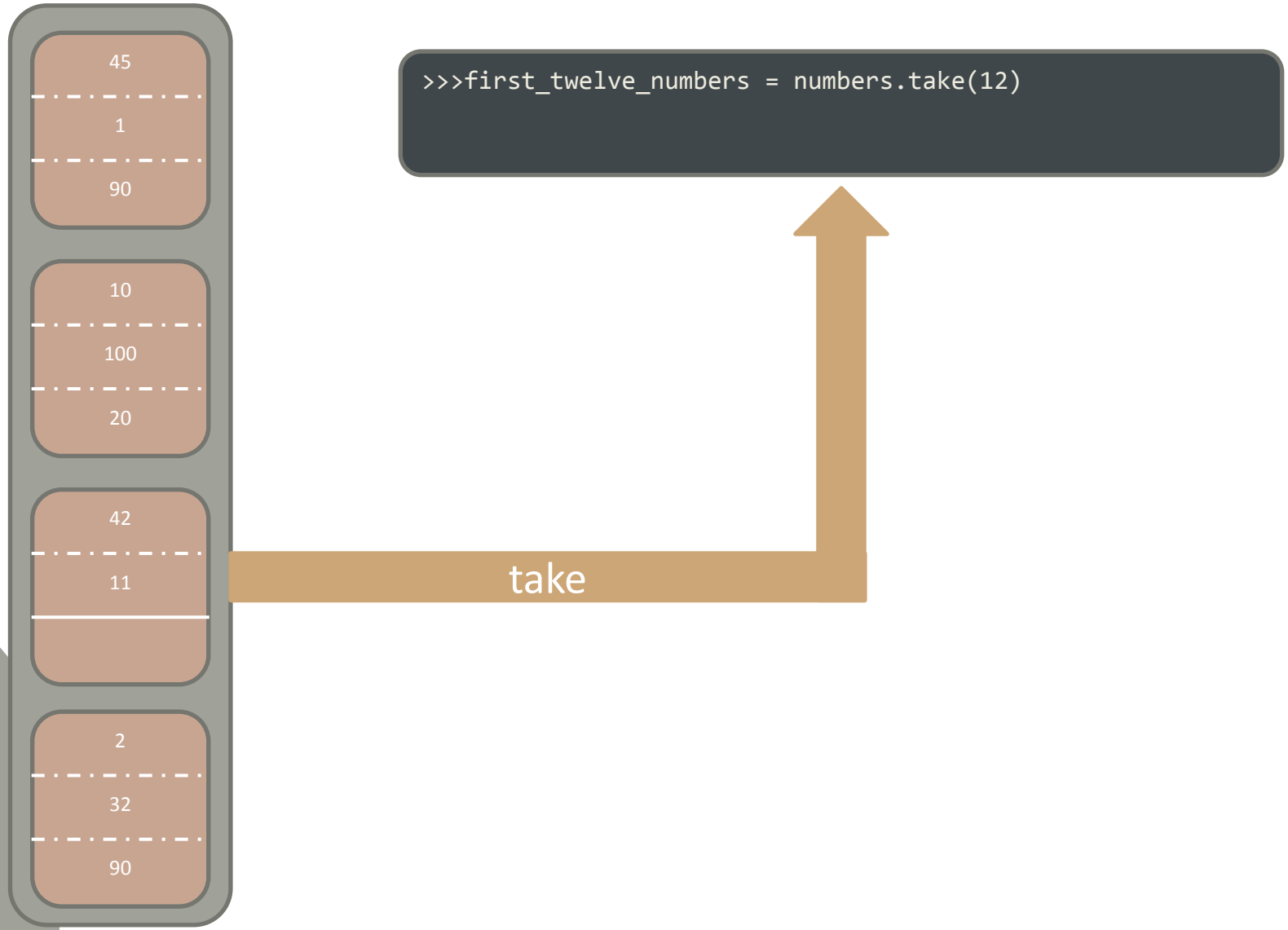

```
>>>first_two_numbers = numbers.take(2)
```

# Spark - take(n)



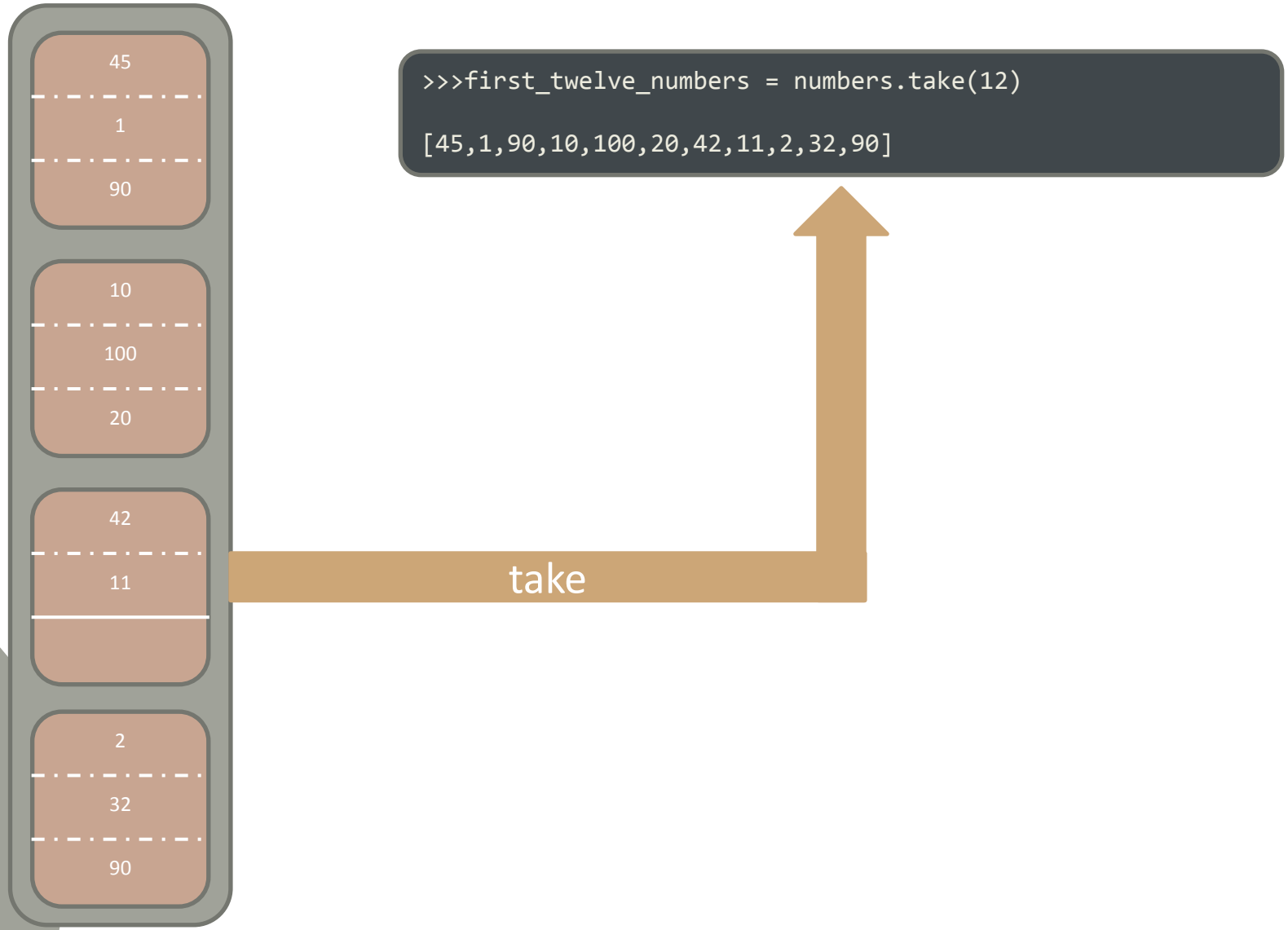

```
>>>first_two_numbers = numbers.take(2)

[45, 1]
```

# Spark - take(n)



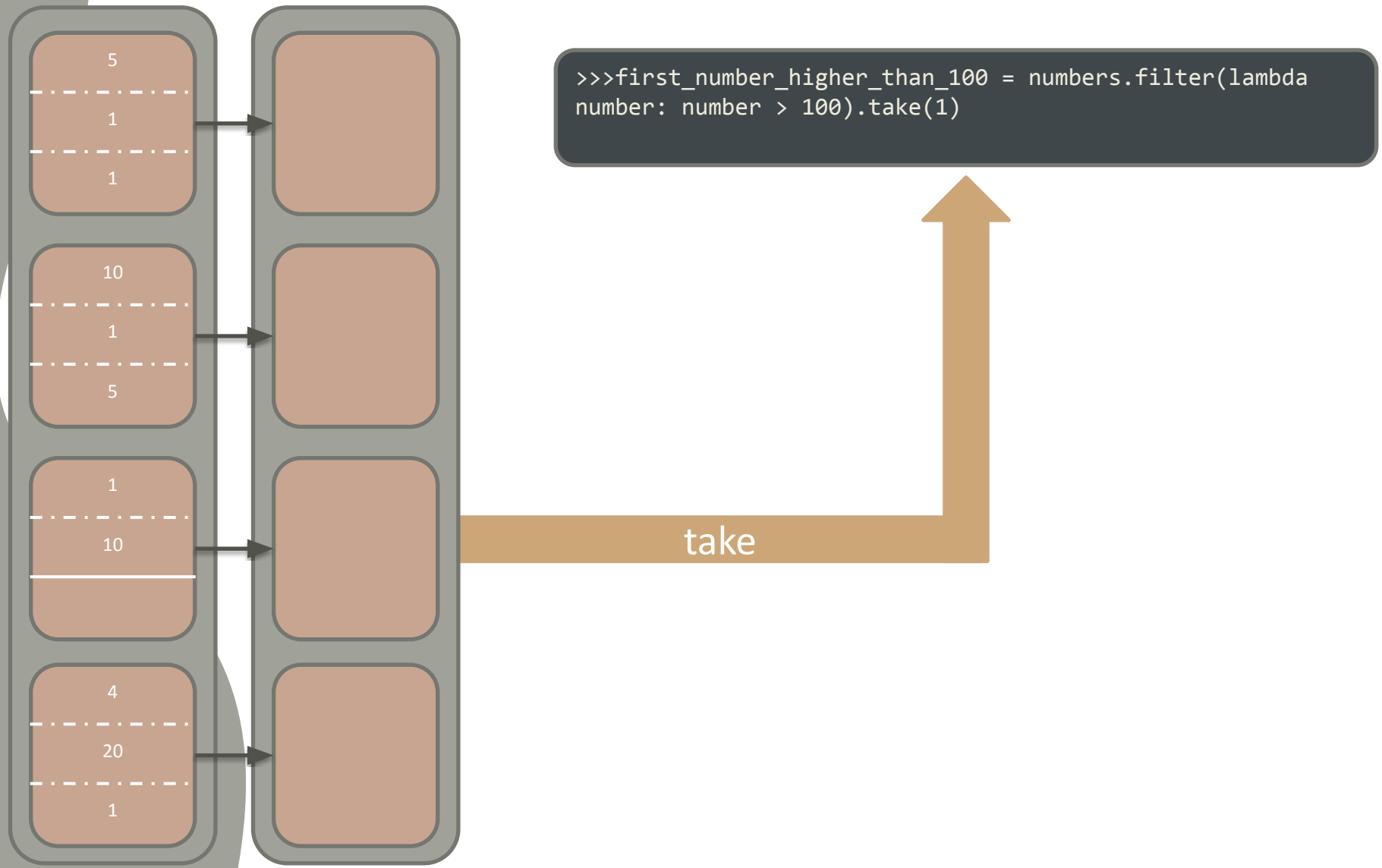

```
>>>first_twelve_numbers = numbers.take(12)
```

# Spark - take(n)



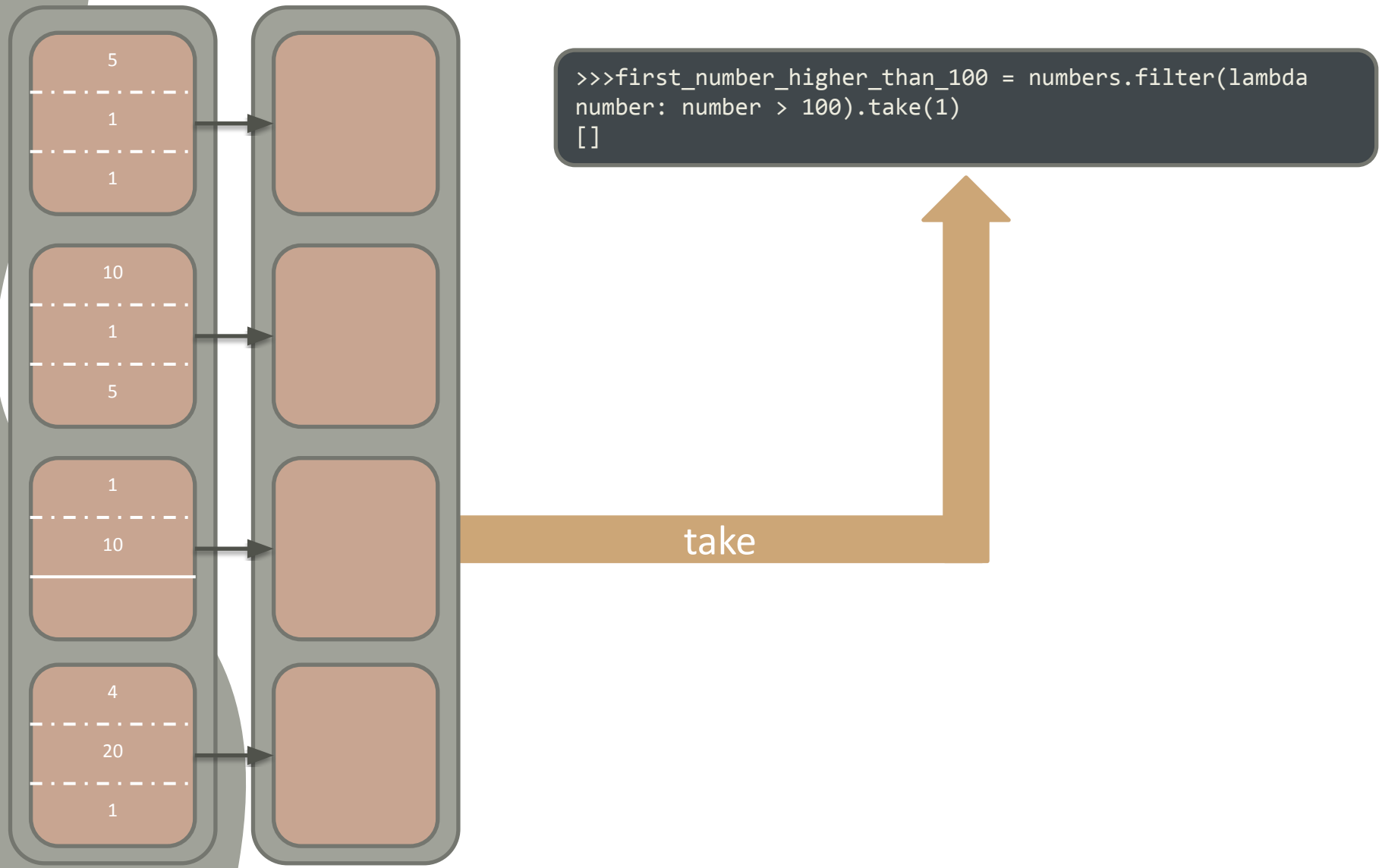

```
>>>first_twelve_numbers = numbers.take(12)

[45,1,90,10,100,20,42,11,2,32,90]
```

# Spark - take(n)




```
>>>first_number_higher_than_100 = numbers.filter(lambda
number: number > 100).take(1)
```

# Spark - take(n)

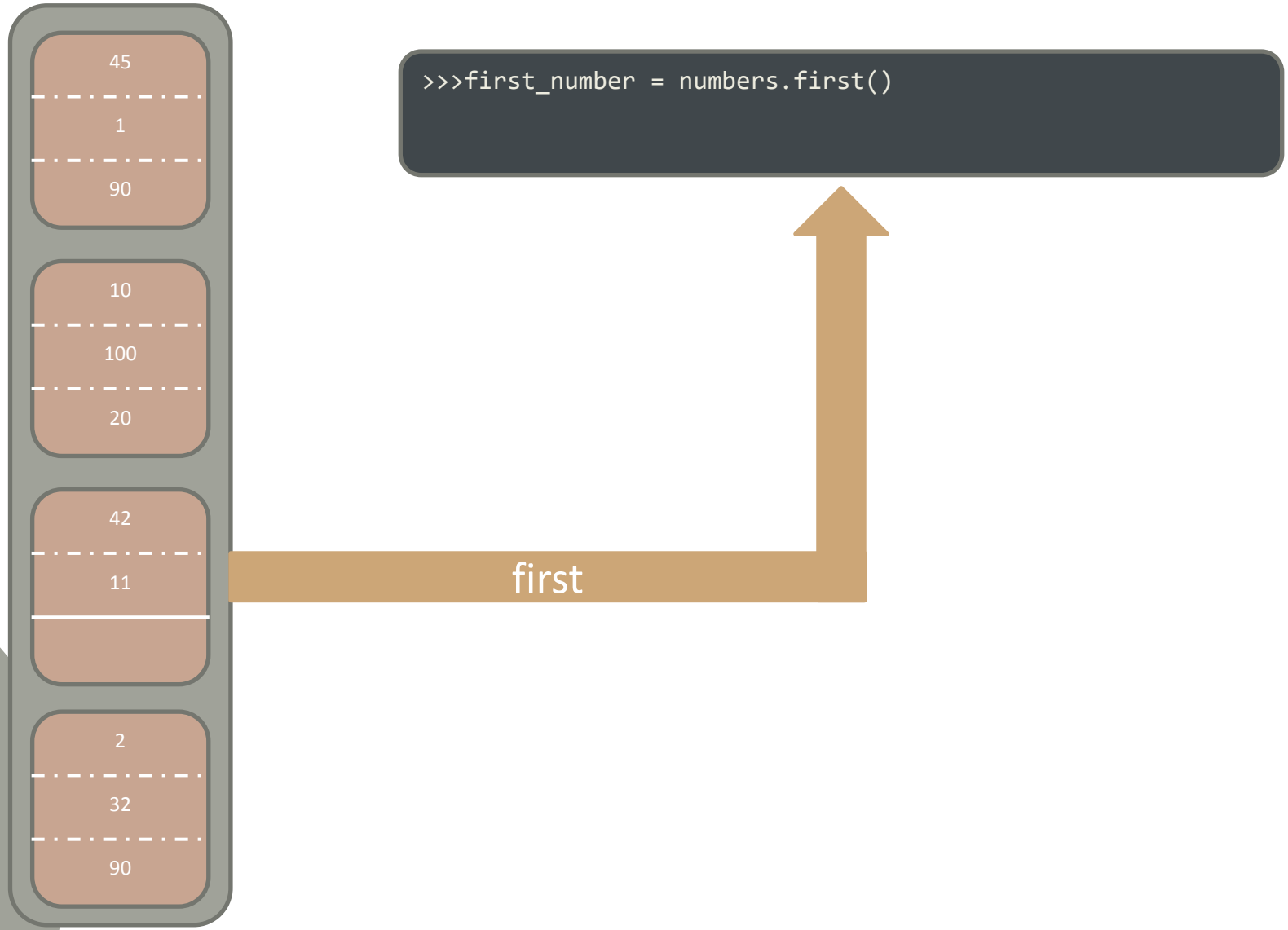



```
>>>first_number_higher_than_100 = numbers.filter(lambda
number: number > 100).take(1)
[]
```

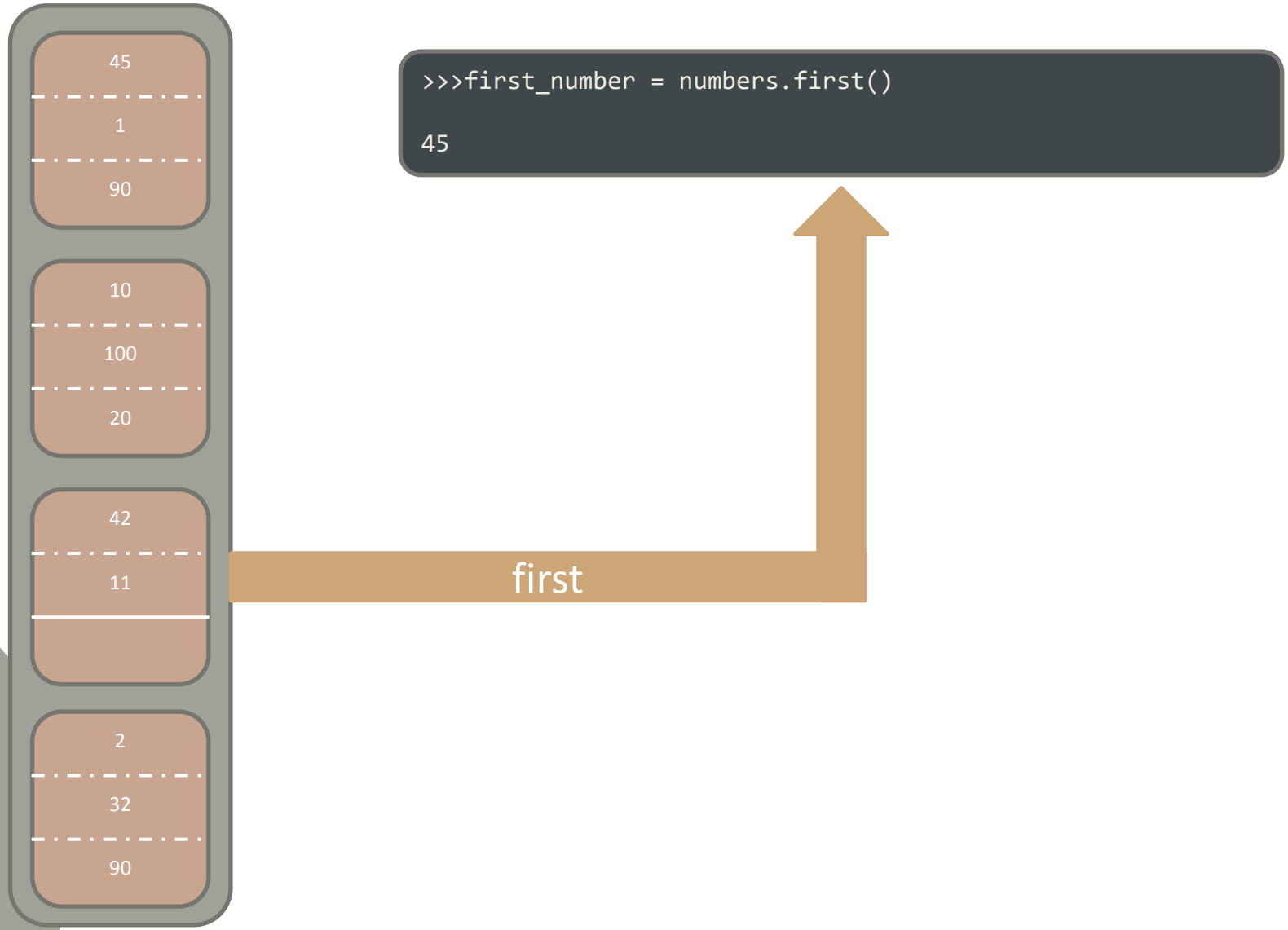
# Spark - first

- Es una acción por lo que terminará un **job**
- El resultado será el primer elemento del **RDD**
- Se enviará al driver el primer elemento del primer partition del RDD
- Si el RDD está vacío, obtendremos un error
- first() : U

# Spark - first

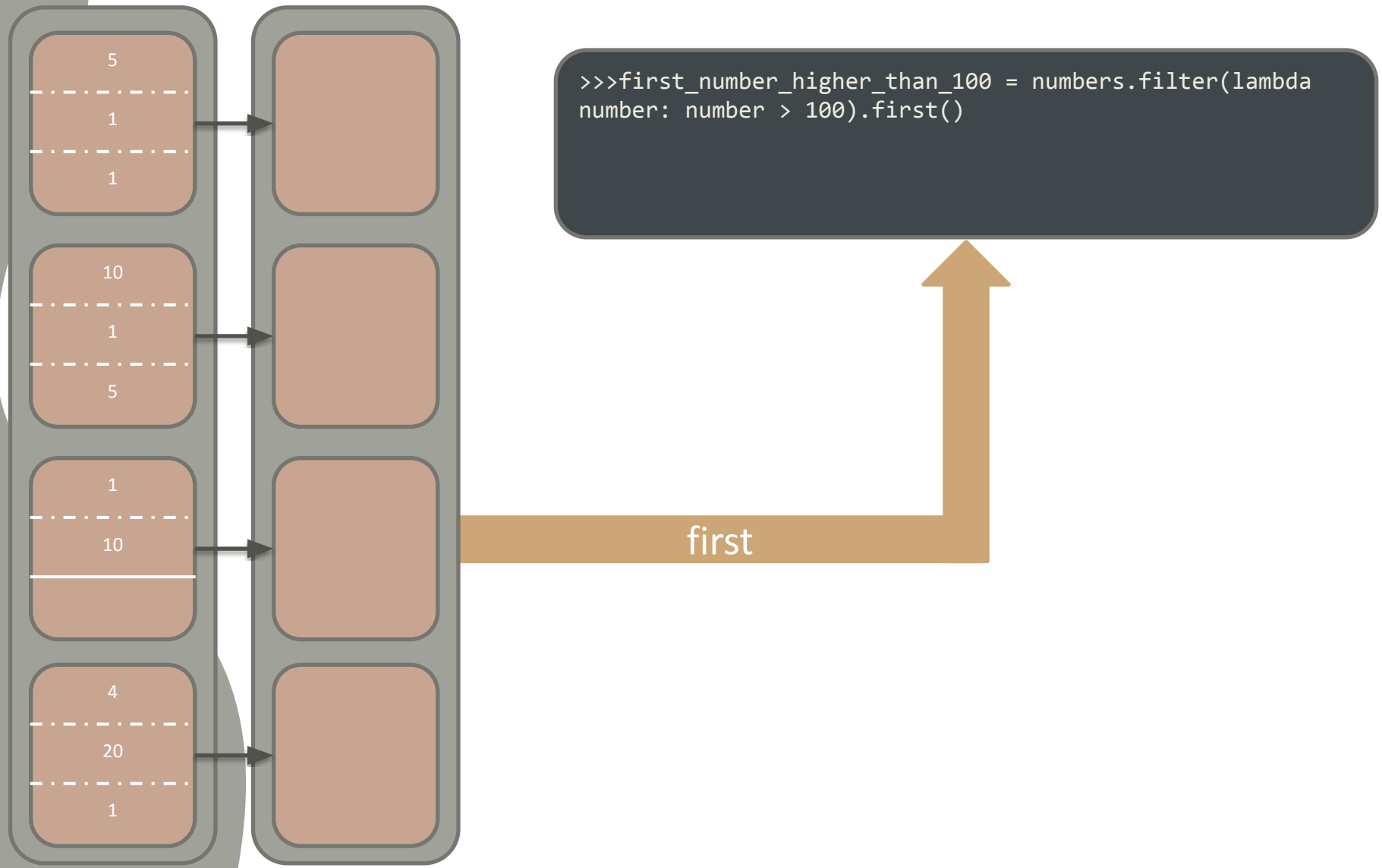


# Spark - first



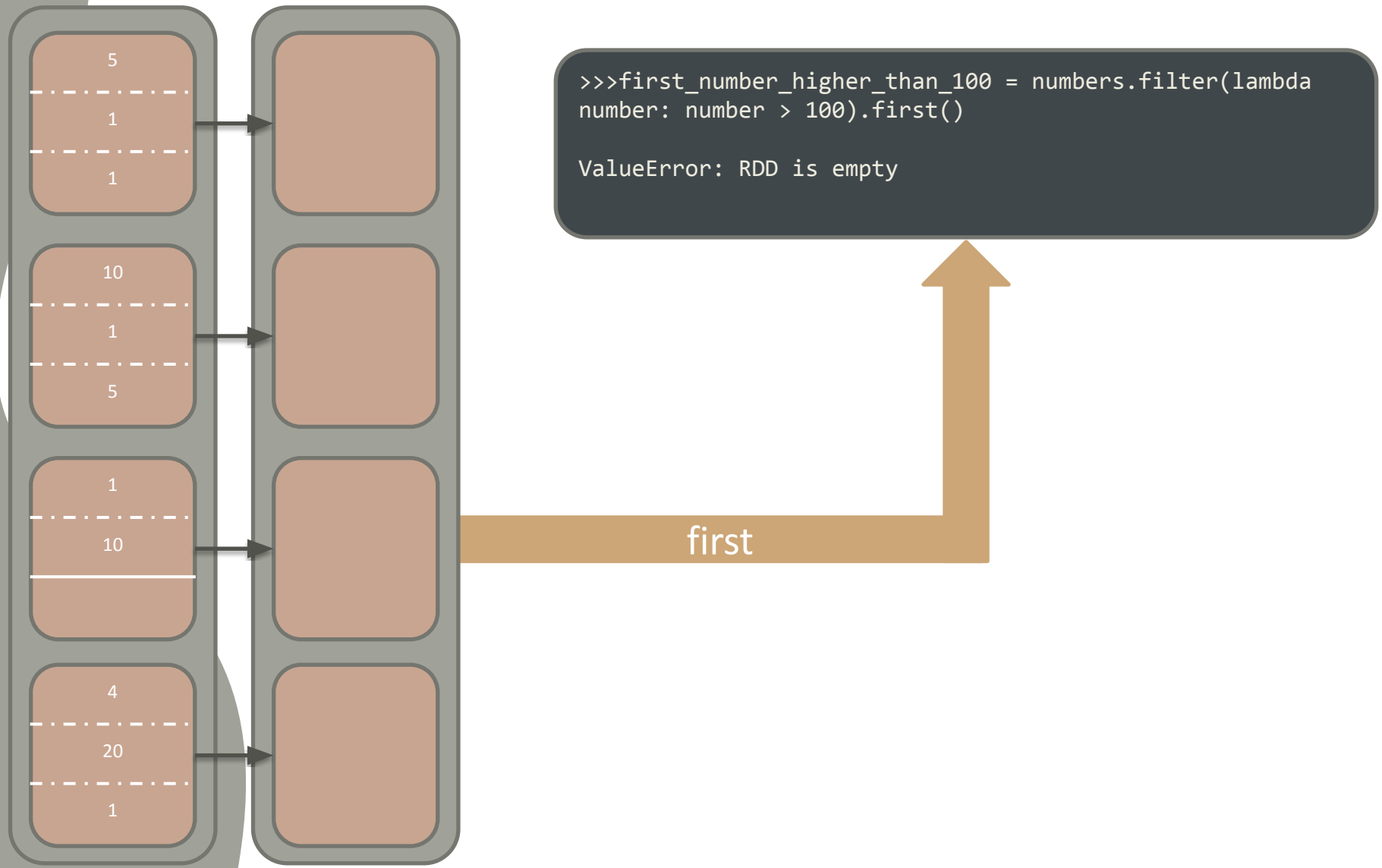

```
>>>first_number = numbers.first()

45
```

# Spark - first




```
>>>first_number_higher_than_100 = numbers.filter(lambda
number: number > 100).first()
```

# Spark - first

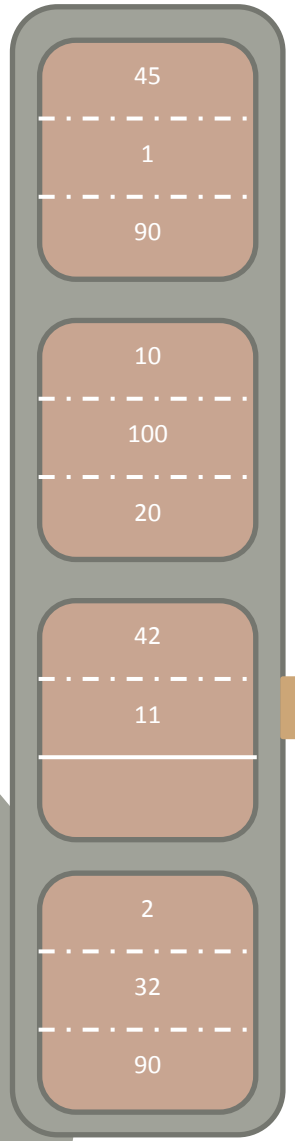



```
>>>first_number_higher_than_100 = numbers.filter(lambda
number: number > 100).first()

ValueError: RDD is empty
```

# Spark - reduce

- Es una acción por lo que terminará un **job**
- El resultado será el resultado que ejecutar la operación definida de dos en dos elementos
- A diferencia del Reduce de Map&Reduce los elementos no tienen que ser del tipo clave valor y el resultado es devuelto al Driver
-
- reduce(f: (U, U) ⇒ U) : U

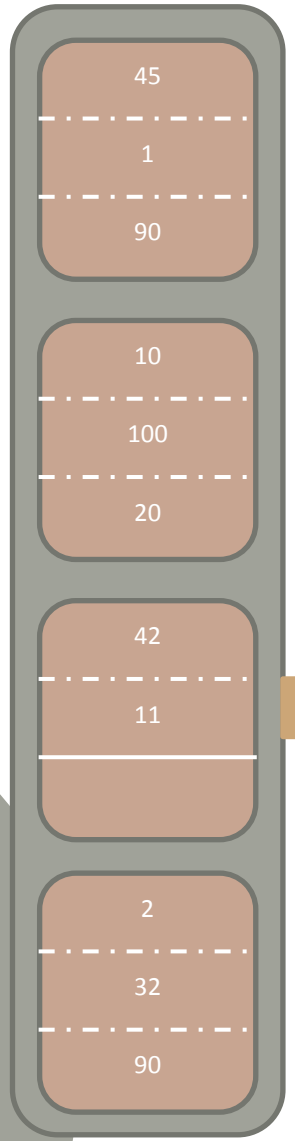
# Spark - reduce




```
>>>def calculate_max(number1, number2):
  if number1 >= number2:
    return number1
  else:
    return number2

>>>max_number = numbers.reduce(lambda number1, number2:
calculate_max(number1, number2))
```

reduce

# Spark - reduce




```
>>>def calculate_max(number1, number2):
  if number1 >= number2:
    return number1
  else:
    return number2

>>>max_number = numbers.reduce(lambda number1, number2:
calculate_max(number1, number2))

100
```

reduce

# Apache Spark