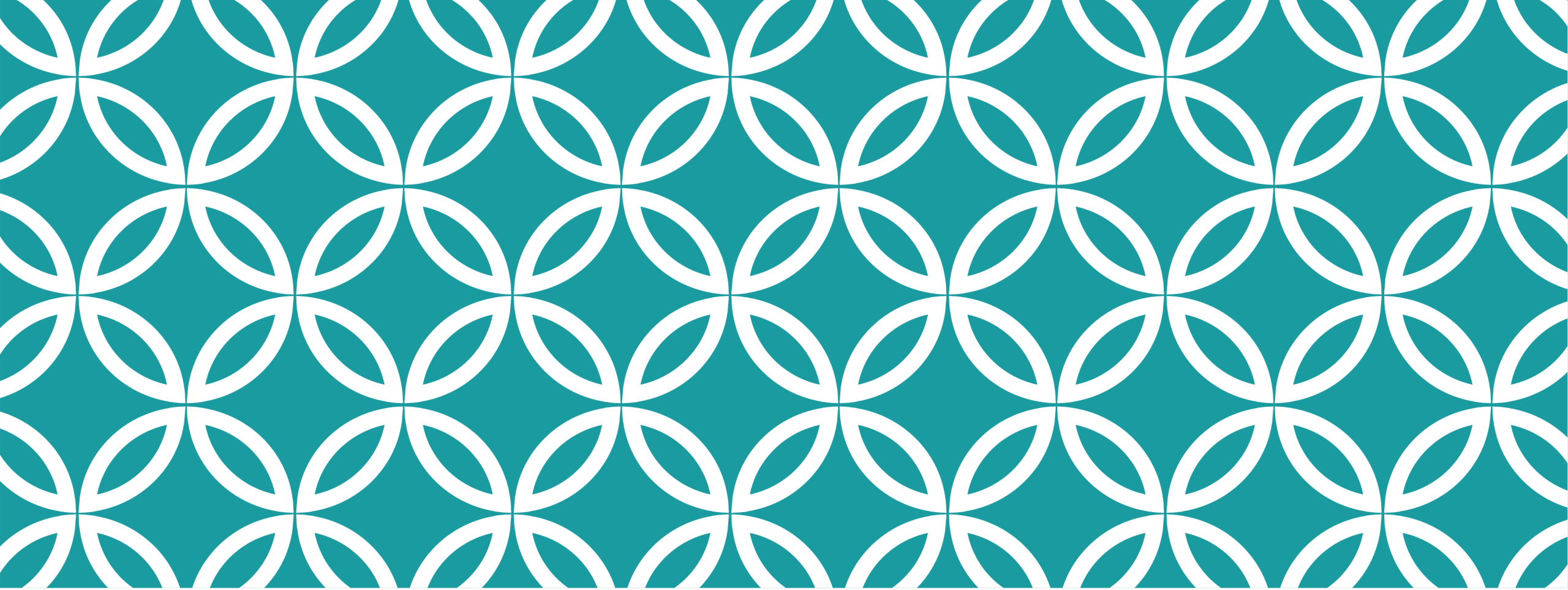


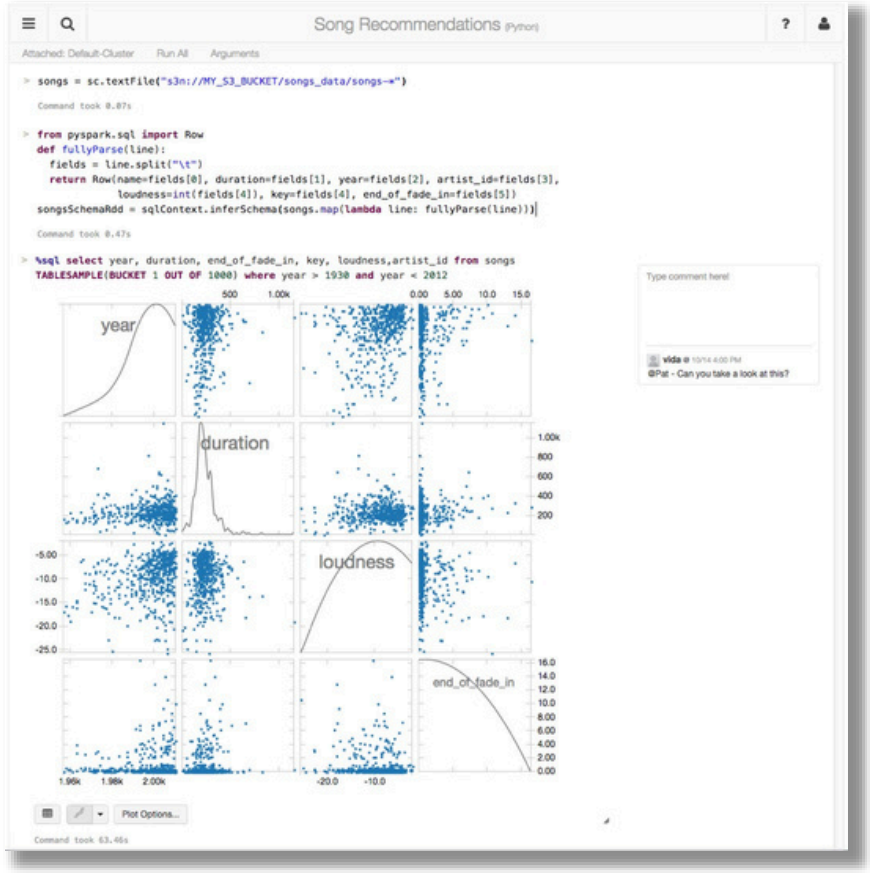
TRANSFORMATIONS AND ACTIONS

A Visual Guide of the API

databricks™

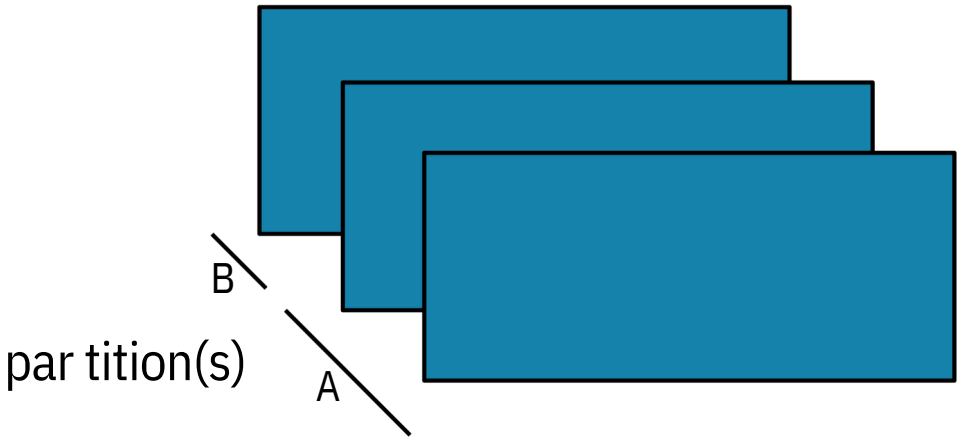
making big data simple

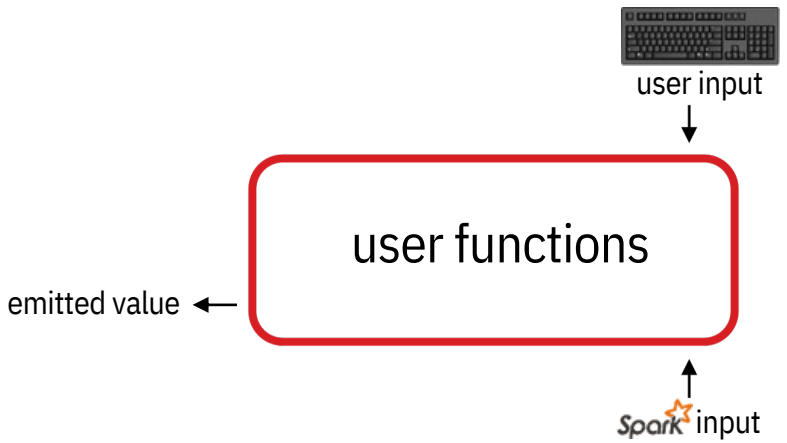
- Founded in late 2013
- by the creators of Apache Spark
- Original team from UC Berkeley AMPLab
- Raised $47 Million in 2 rounds
- ~55 employees
- We're hiring! (http://databricks.workable.com)
- Level 2/3 support partnerships with
  - Hortonworks
  - MapR
  - DataStax



Databricks Cloud: "A unified platform for building Big Data pipelines –from ETL to Exploration and Dashboards, to Advanced Analytics and Data Products."

RDD





Legend

RDD Elements

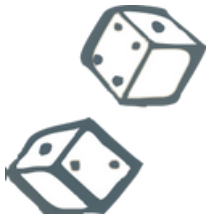
key

original item

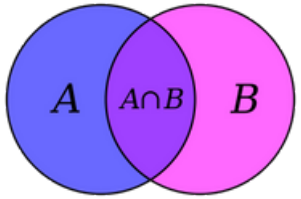
transformed type

object on driver

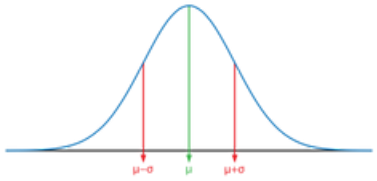
Legend



Randomized operation



Set Theory / Relational operation



Numeric calculation

Spark Operations =

TRANSFORMATIONS

+

ACTIONS

 = easy

 = medium

Essential Core & Intermediate Spark Operations

TRANSFORMATIONS

General

- map
- filter
- flatMap
- mapPartitions
- mapPartitionsWithIndex
- groupBy
- sortBy

Math / Statistical

- sample
- randomSplit

Set Theory / Relational

- union
- intersection
- subtract
- distinct
- cartesian
- zip

Data Structure / I/O

- keyBy
- zipWithIndex
- zipWithUniqueID
- zipPartitions
- coalesce
- repartition
- repartitionAndSortWithinPartitions
- pipe

- reduce
- collect
- aggregate
- fold
- first
- take
- forEach
- top
- treeAggregate
- treeReduce
- forEachPartition
- collectAsMap

- count takeSample
- max min sum
- histogram mean
- variance stdev
- sampleVariance
- countApprox
- countApproxDistinct

- takeOrdered

- saveAsTextFile
- saveAsSequenceFile
- saveAsObjectFile
- saveAsHadoopDataset
- saveAsHadoopFile
- saveAsNewAPIHadoopDataset
- saveAsNewAPIHadoopFile

 = easy

 = medium

Essential Core & Intermediate PairRDDOperations

TRANSFORMATIONS

General

- flatMapValues
- groupByKey
- reduceByKey
- reduceByKeyLocally
- foldByKey
- aggregateByKey
- sortByKey
- combineByKey

Math / Statistical

- sampleByKey

Set Theory / Relational

- cogroup (=groupWith)
- join subtractByKey
- fullOuterJoin
- leftOuterJoin
- rightOuterJoin
-

Data Structure

- partitionBy

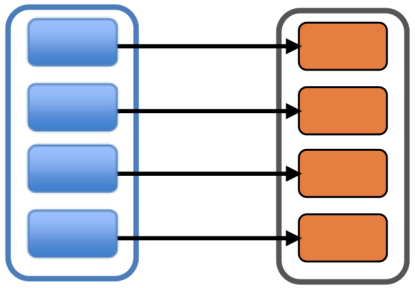
- keys
- values

- countByKey countByValue
- countByValueApprox
- countApproxDistinctByKey
- countApproxDistinctByKey
- countByKeyApprox
- sampleByKeyExact
-

vs

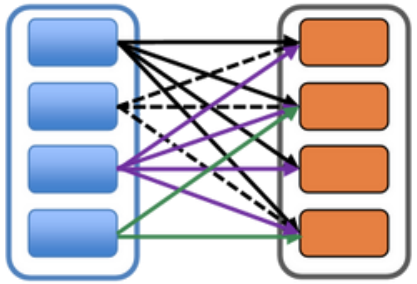
narrow

each partition of the parent RDD is used by at most one partition of the child RDD



wide

multiple child RDD partitions may depend on a single parent RDD partition



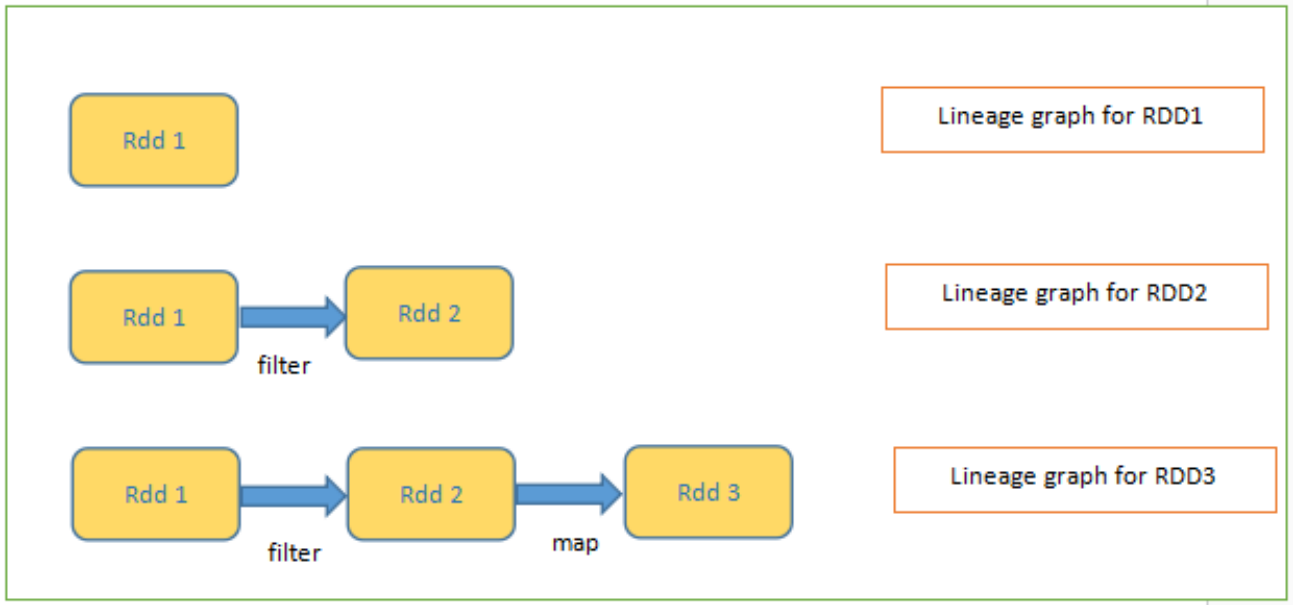
LINEAGE

"One of the challenges in providing RDDs as an abstraction is choosing a representation for them that can track lineage across a wide range of transformations."

"The most interesting question in designing this interface is how to represent dependencies between RDDs."

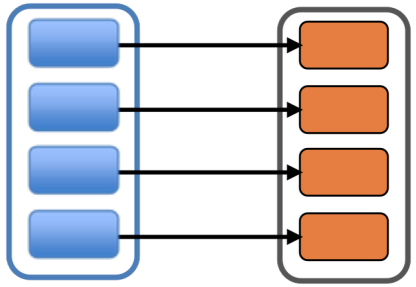
"We found it both sufficient and useful to classify dependencies into two types:

- narrow dependencies, where each partition of the parent RDD is used by at most one partition of the child RDD
- wide dependencies, where multiple child partitions may depend on it."

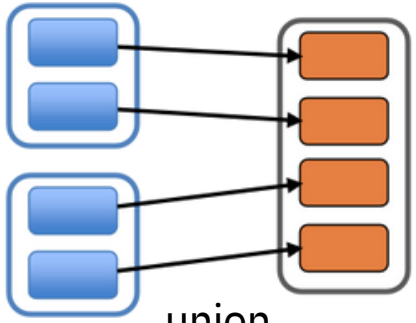


narrow

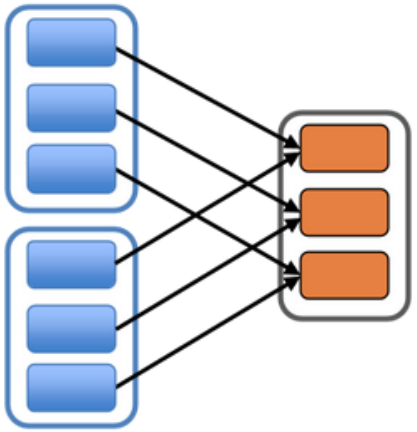
each partition of the parent RDD is used by at most one partition of the child RDD



map, filter



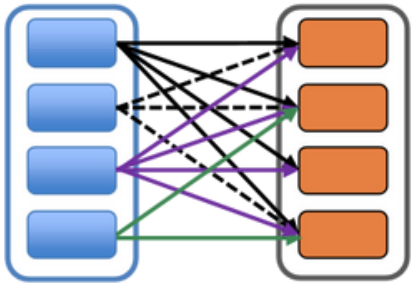
union



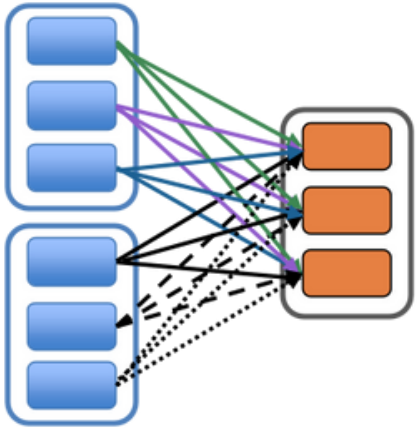
join w/ inputs co-partitioned

wide

multiple child RDD partitions may depend on a single parent RDD partition

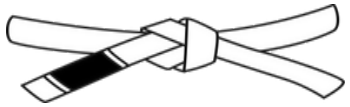


groupByKey



join w/ inputs not co-partitioned

TRANSFORMATIONS

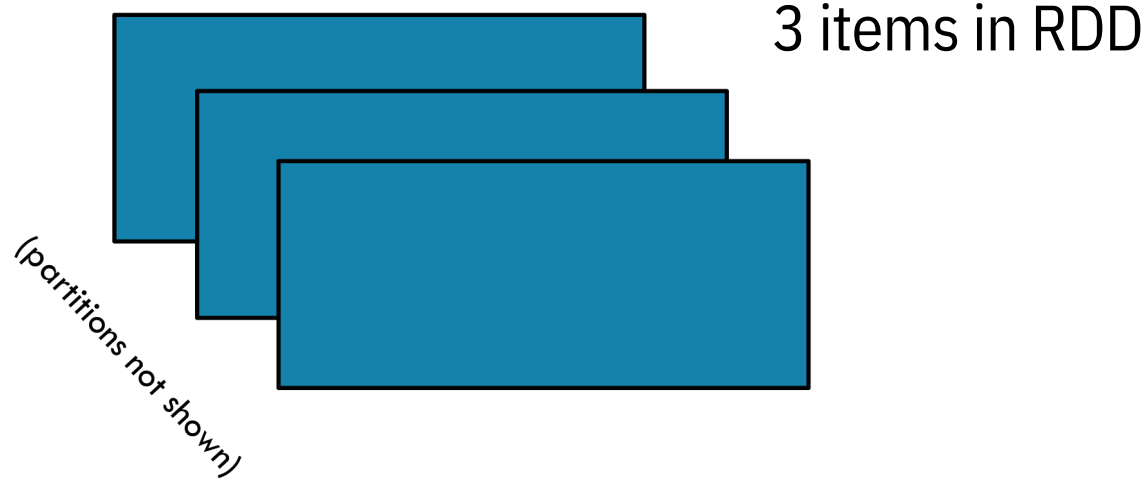


Core Operations

databricks™

MAP

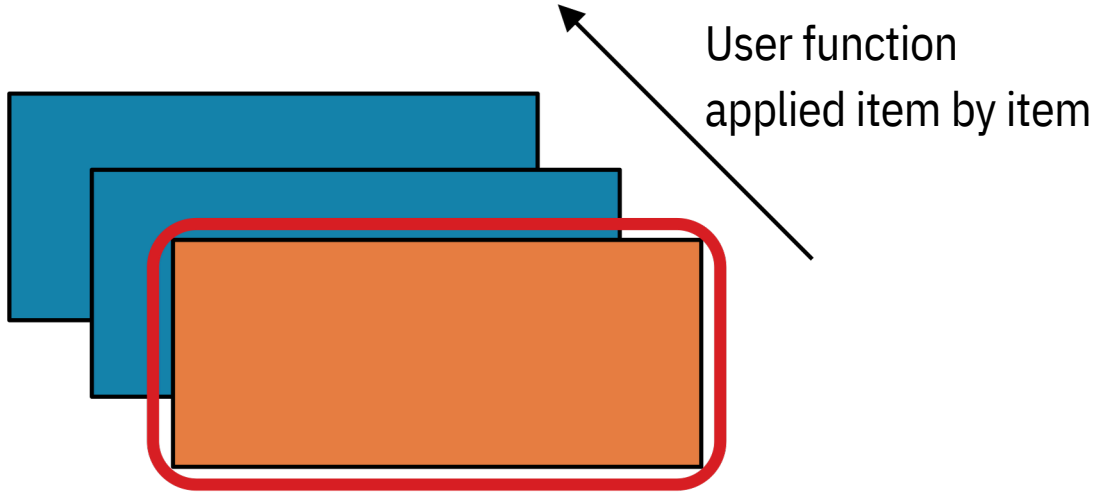
RDD: **x**



3 items in RDD

MAP

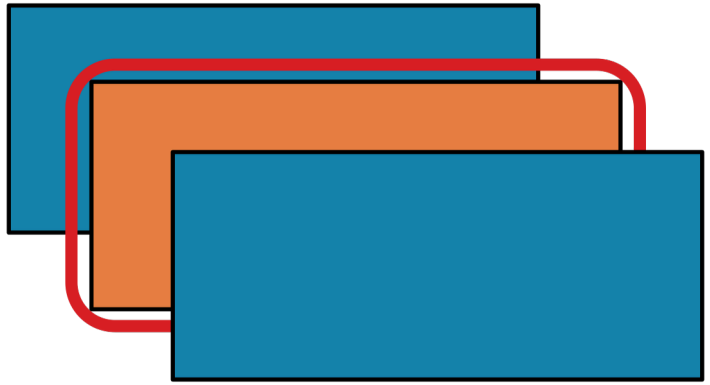
RDD: **x**



RDD: **y**

MAP

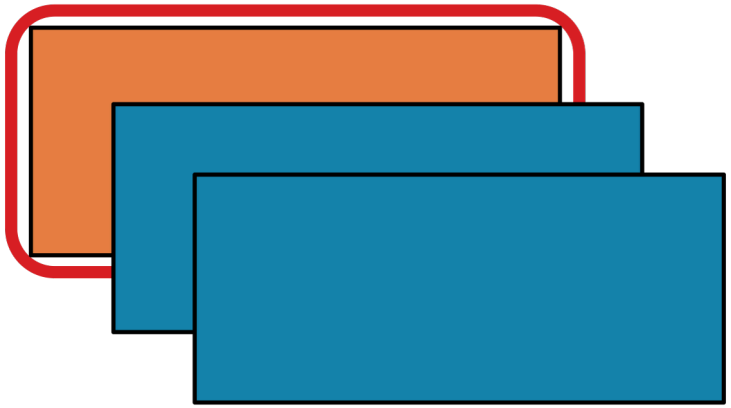
RDD: **x**



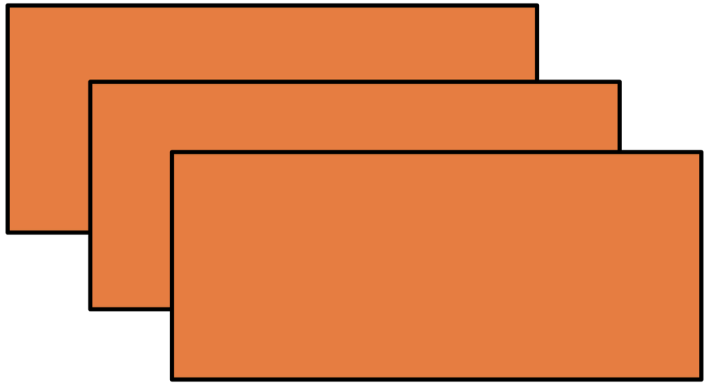
RDD: **y**

MAP

RDD: **x**



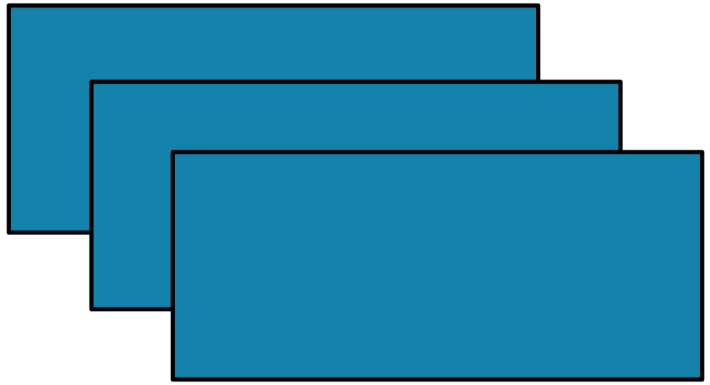
RDD: **y**



MAP

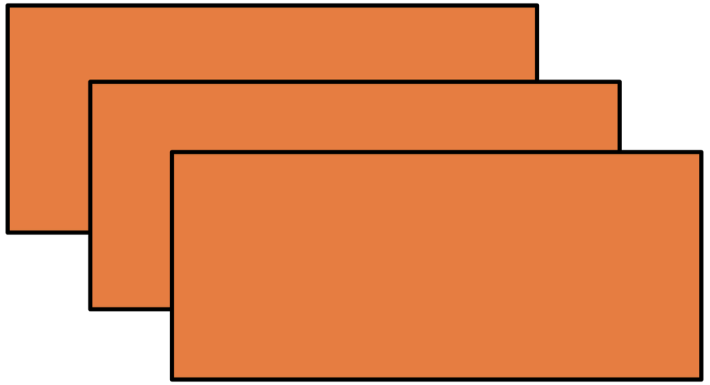
After `map()` has been applied...

RDD: **x**



before

RDD: **y**

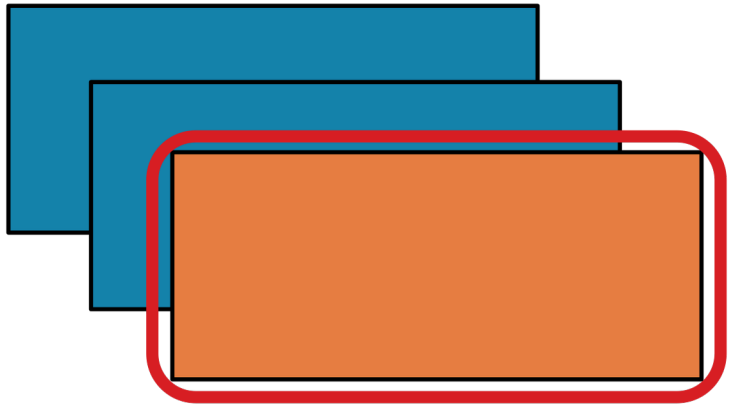


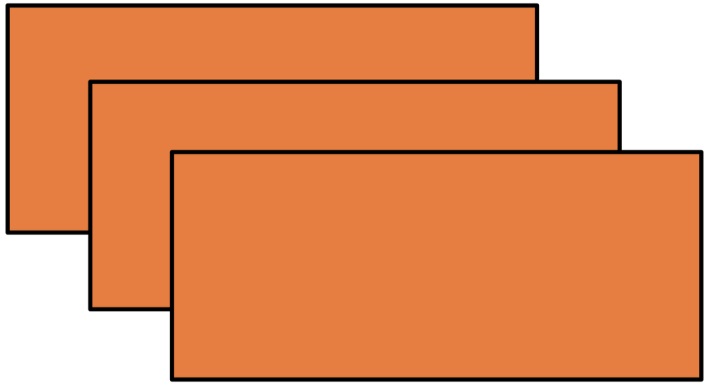
after

MAP

RDD: **x**

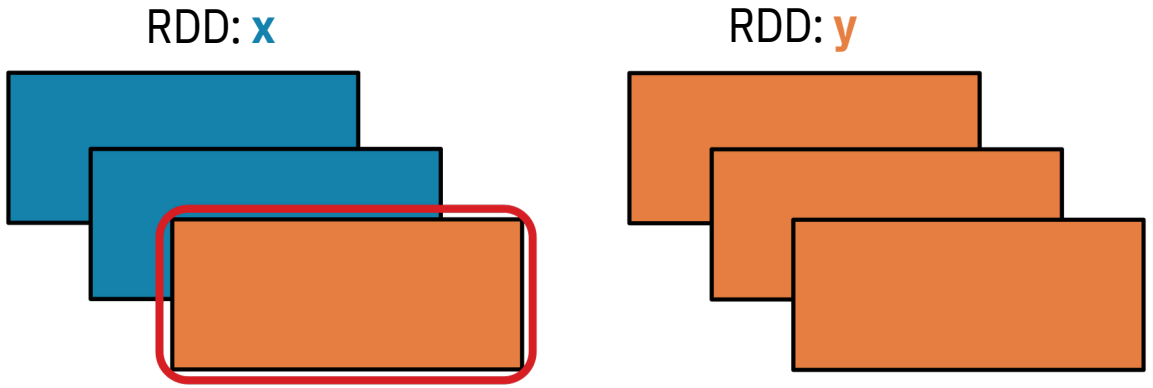
RDD: **y**





Return a new RDD by applying a function to each element of this RDD.

MAP




```
map(f, preservesPartitioning=False)
```

Return a new RDD by applying a function to each element of this RDD

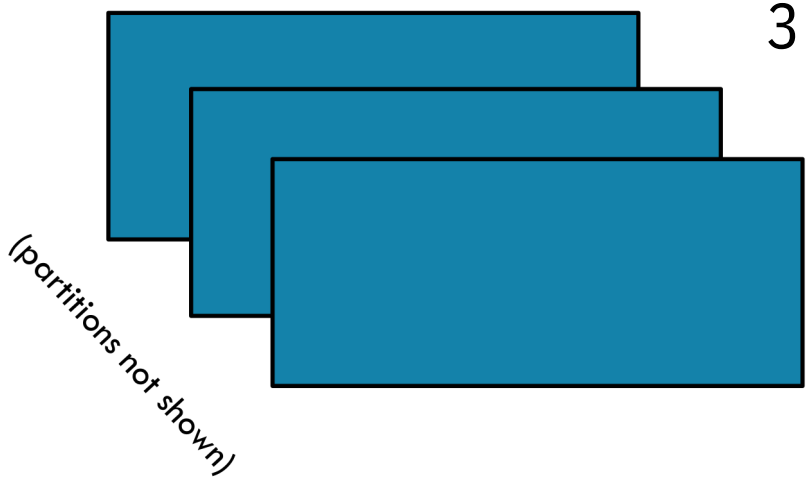
```
x = sc.parallelize(["b", "a", "c"])
y = x.map(lambda z: (z, 1))
print(x.collect())
print(y.collect())
```

```
val x = sc.parallelize(Array("b", "a", "c"))
val y = x.map(z => (z,1))
println(x.collect().mkString(", "))
println(y.collect().mkString(", "))
```

```
x: ['b', 'a', 'c']

y: [('b', 1), ('a', 1), ('c', 1)]
```

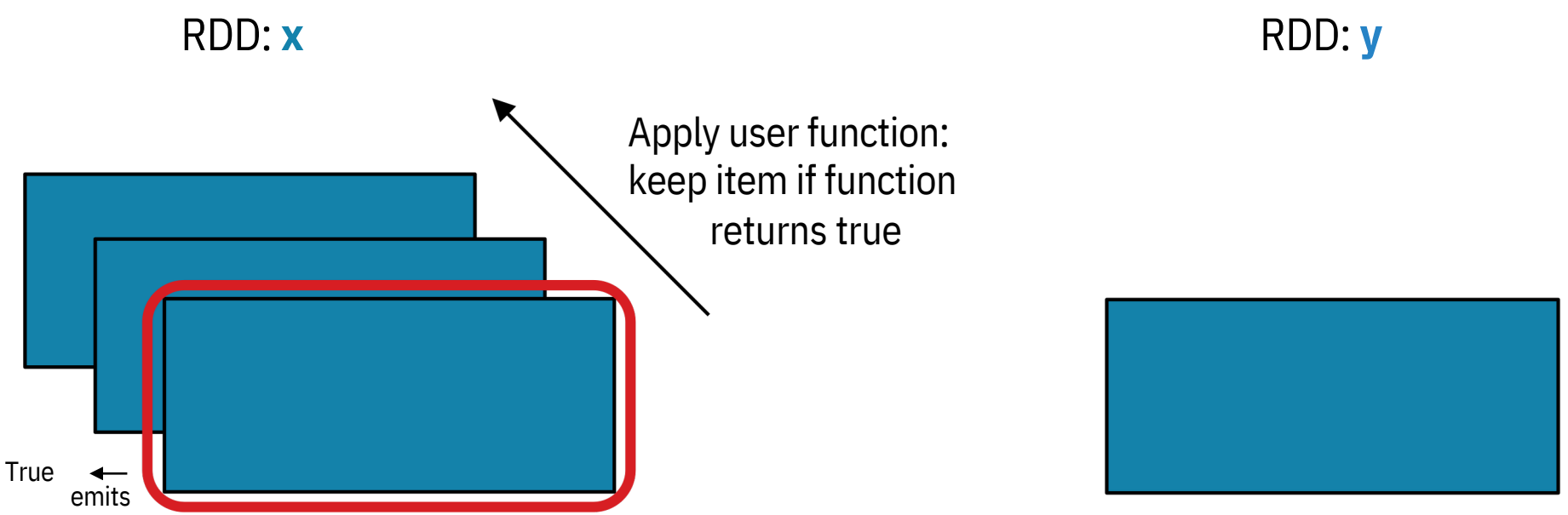
FILTER

RDD: **x**



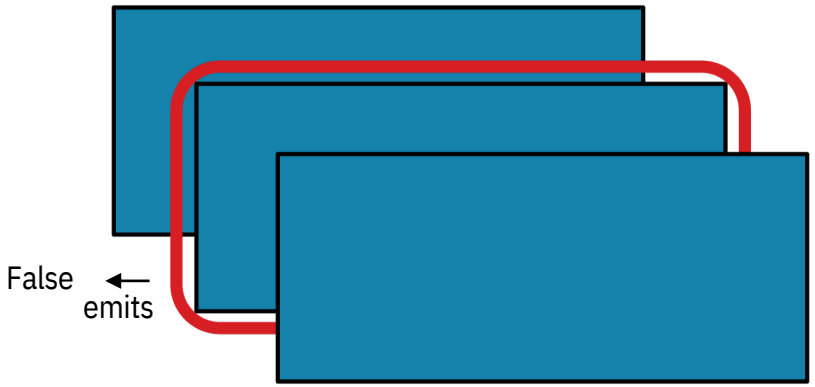
3 items in RDD

FILTER


RDD: x
Apply user function:
keep item if function
returns true
RDD: y
True
emits

FILTER

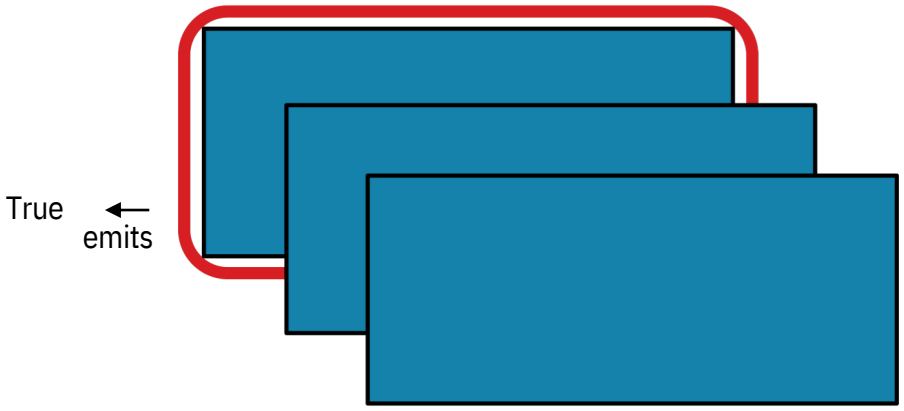
RDD: **x**



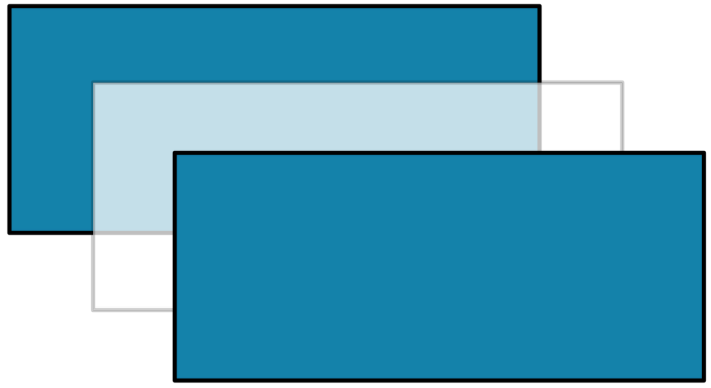
RDD: **y**

FILTER

RDD: **x**



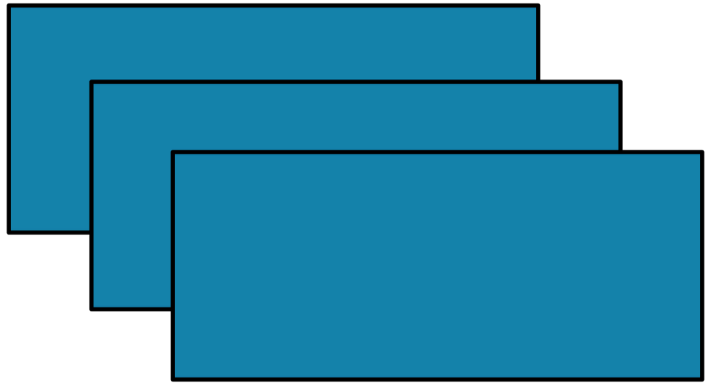
RDD: **y**



FILTER

After `filter()` has been applied...

RDD: **x**

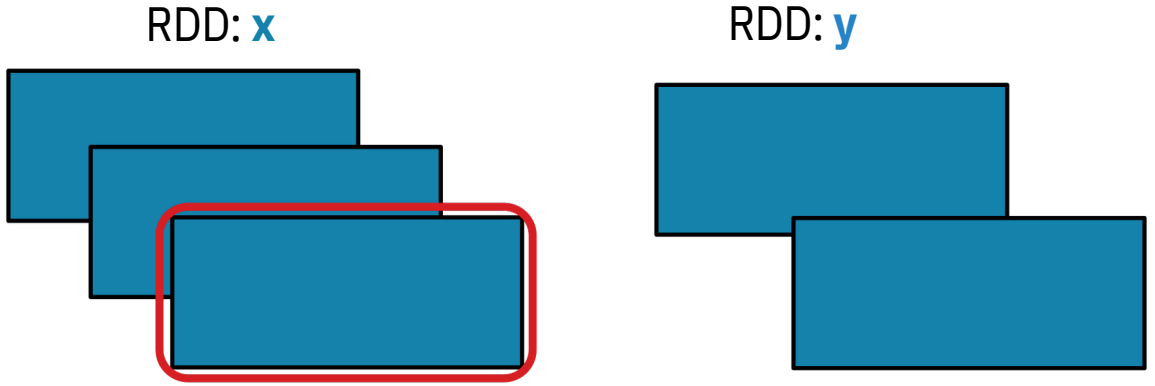


before

RDD: **y**

after

FILTER



Return a new RDD containing only the elements that satisfy a predicate

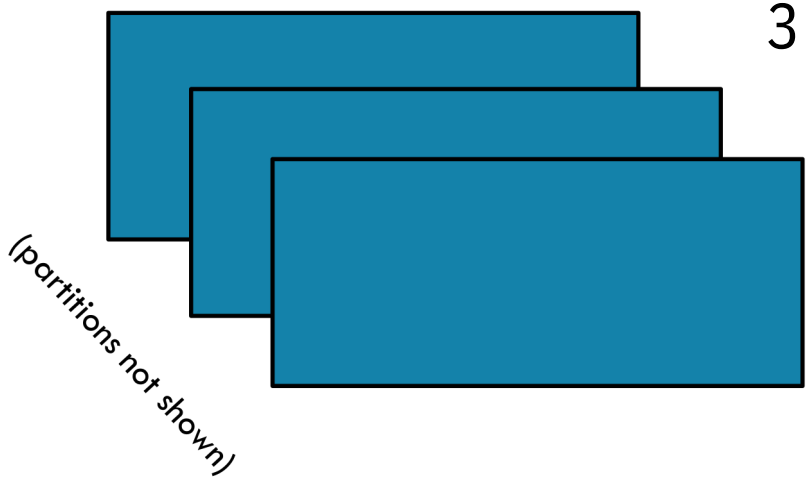
```python
x = sc.parallelize([1,2,3])
y = x.filter(lambda x: x%2 == 1) #keep odd values
print(x.collect())
print(y.collect())
```

```
x: [1, 2, 3]

y: [1, 3]
```

```scala
val x = sc.parallelize(Array(1,2,3))
val y = x.filter(n => n%2 == 1)
println(x.collect().mkString(",  "))
println(y.collect().mkString(", "))
```

FLATMAP

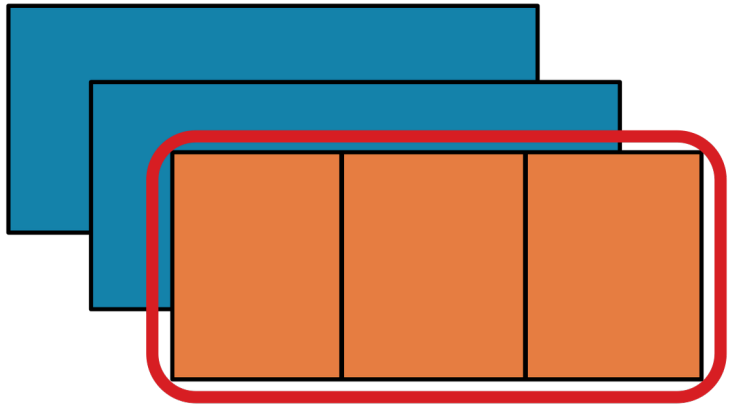
RDD: **x**



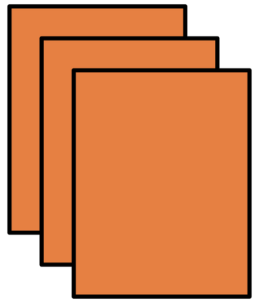
3 items in RDD

FLATMAP

RDD: **x**

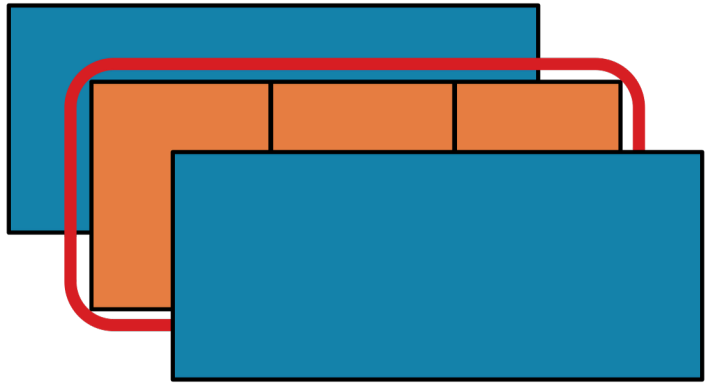


RDD: **y**

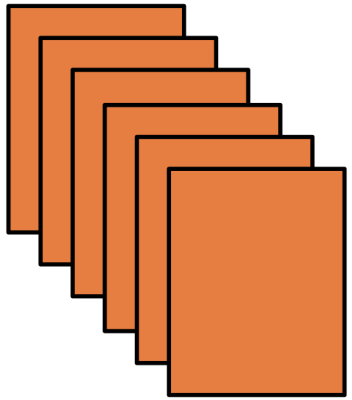


FLATMAP

RDD: **x**

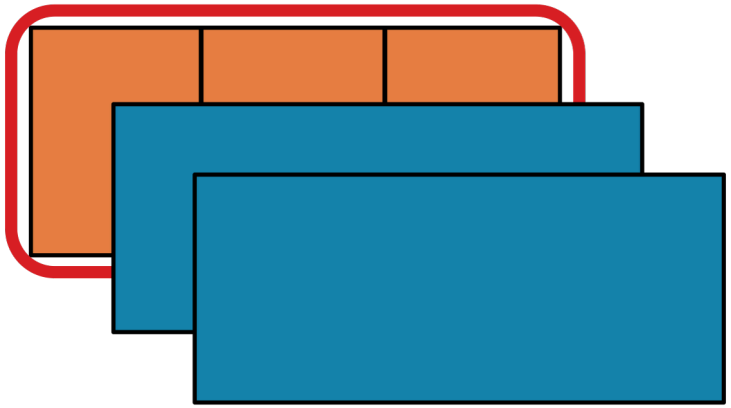


RDD: **y**

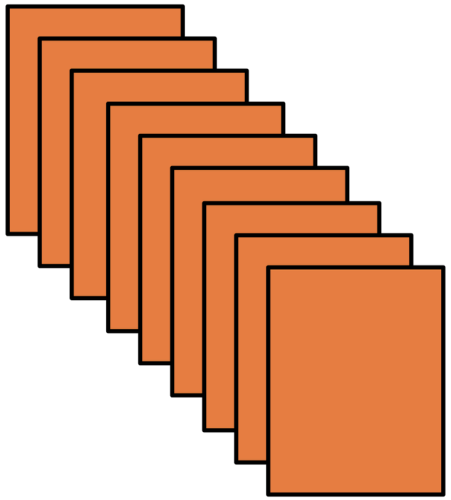


FLATMAP

RDD: **x**



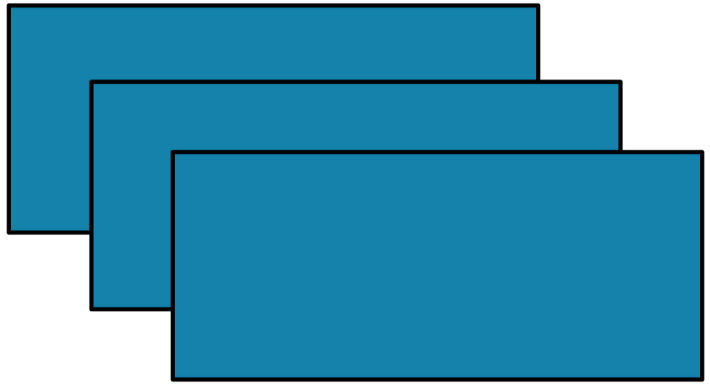
RDD: **y**



FLATMAP

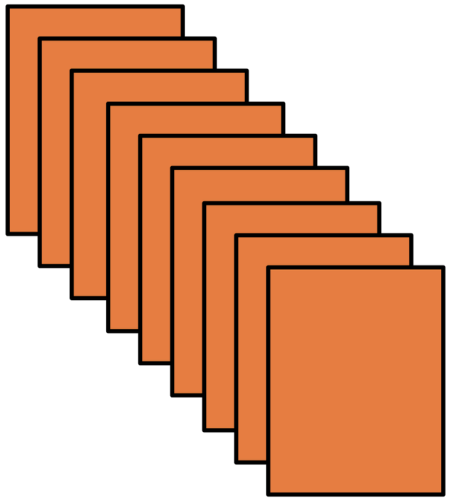
After `flatmap()` has been applied...

RDD: **x**



before

RDD: **y**

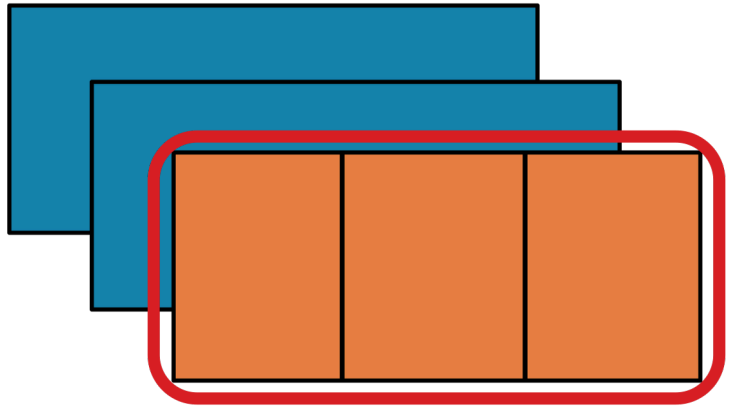


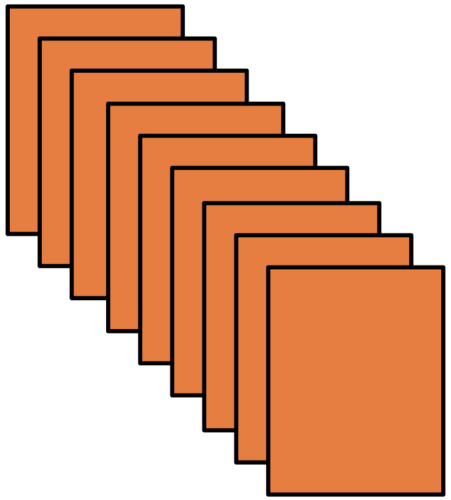
after

FLATMAP

RDD: **x**

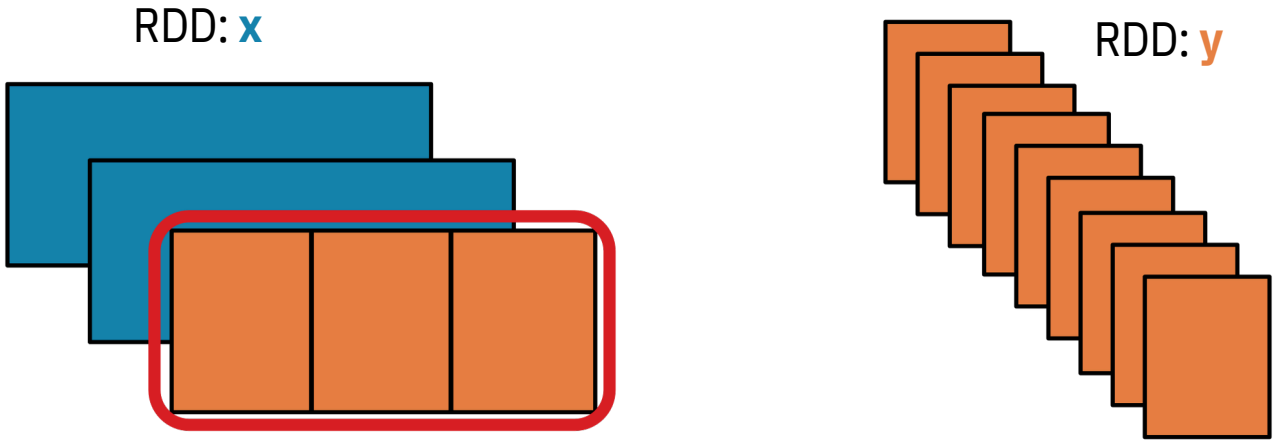
RDD: **y**





Return a new RDD by first applying a function to all elements of this RDD, and then flattening the results

FLATMAP




```
flatMap(f, preservesPartitioning=False)
```

Return a new RDD by first applying a function to all elements of this RDD, and then flattening the results

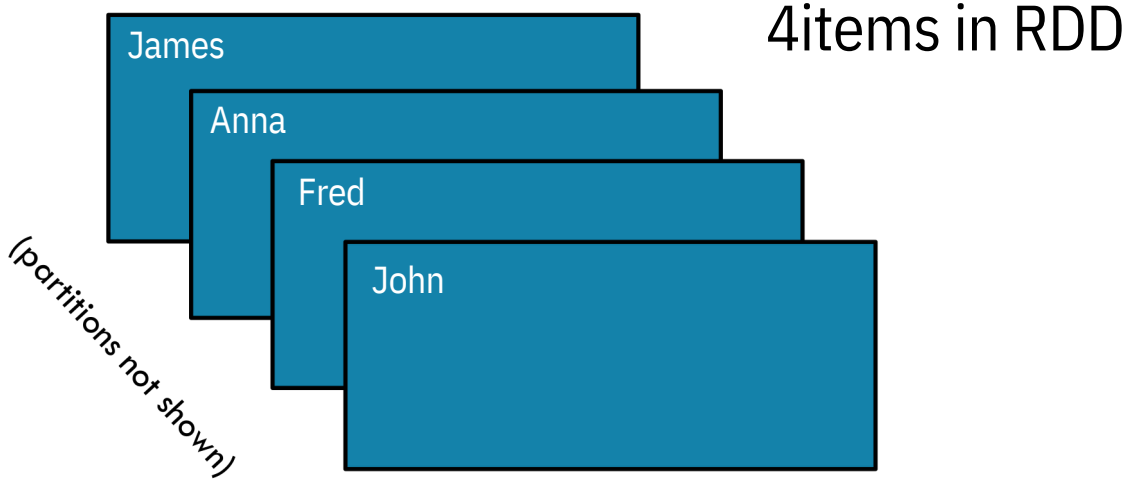
```
x = sc.parallelize([1,2,3])
y = x.flatMap(lambda x: (x, x*100, 42))
print(x.collect())
print(y.collect())
```

```
val x = sc.parallelize(Array(1,2,3))
val y = x.flatMap(n => Array(n, n*100, 42))
println(x.collect().mkString(",  "))
println(y.collect().mkString(", "))
```

```
x:  [1, 2, 3]
y:
    [1, 100, 42, 2, 200, 42, 3, 300, 42]
```

GROUPBY

RDD: **x**

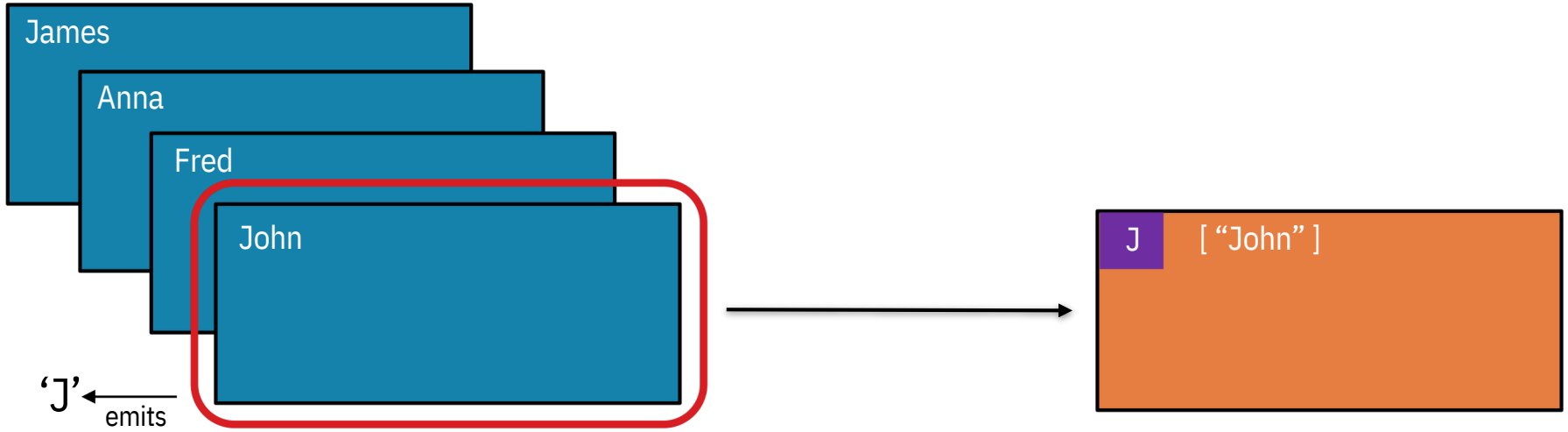


4items in RDD

GROUPBY

RDD: **x**

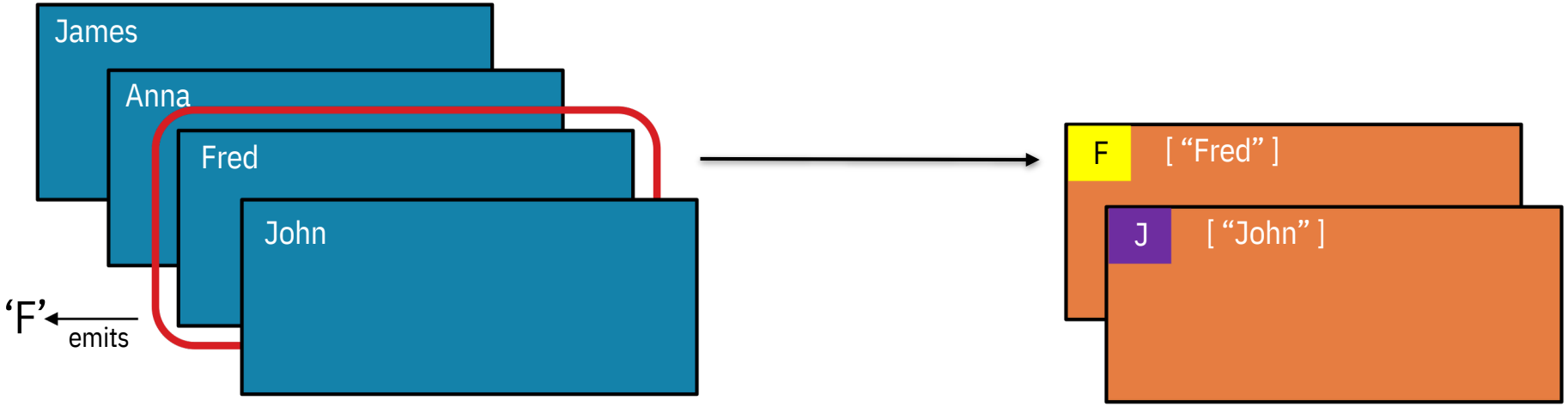
RDD: **y**



GROUPBY

RDD: **x**

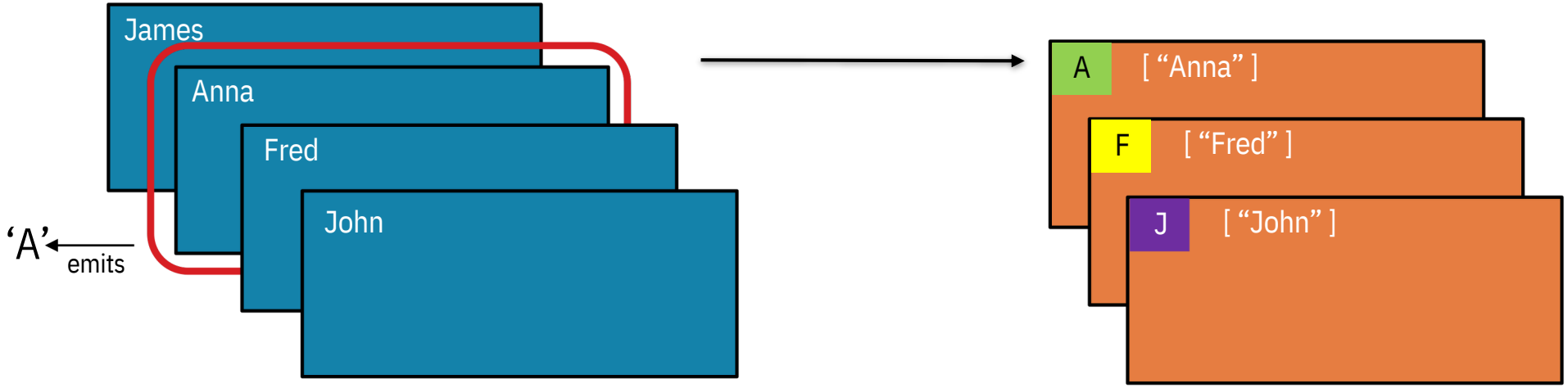
RDD: **y**



GROUPBY

RDD: **x**

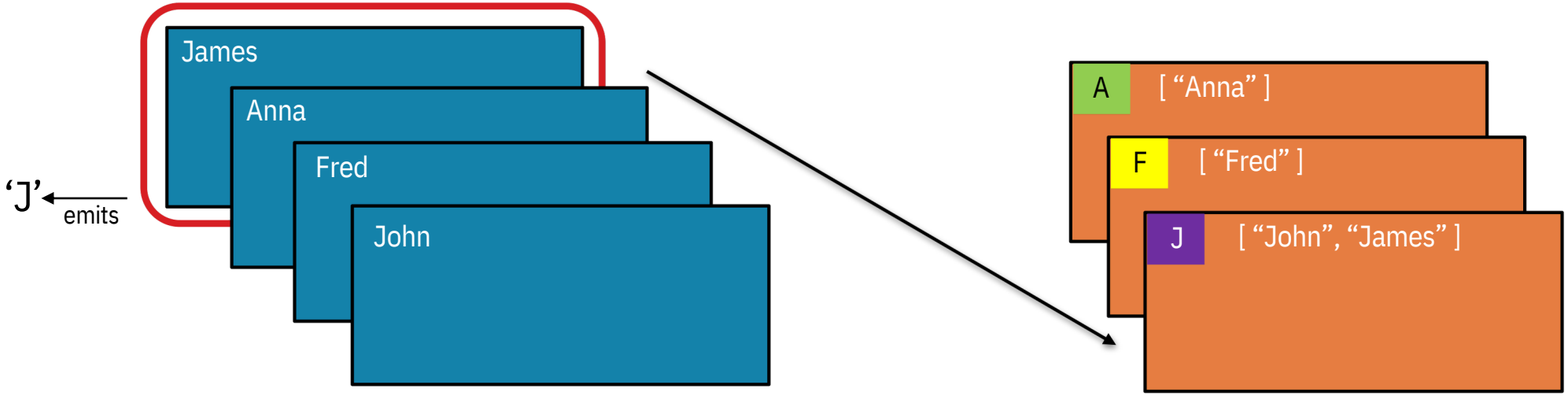
RDD: **y**



GROUPBY

RDD: **x**

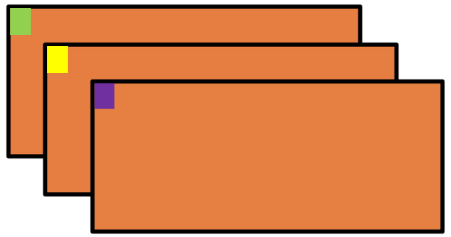
RDD: **y**



GROUPBY

RDD: x

RDD: y



```
groupBy(f, numPartitions=None)
```

Group the data in the original RDD. Create pairs where the key is the output of a user function, and the value is all items for which the function yields this key.

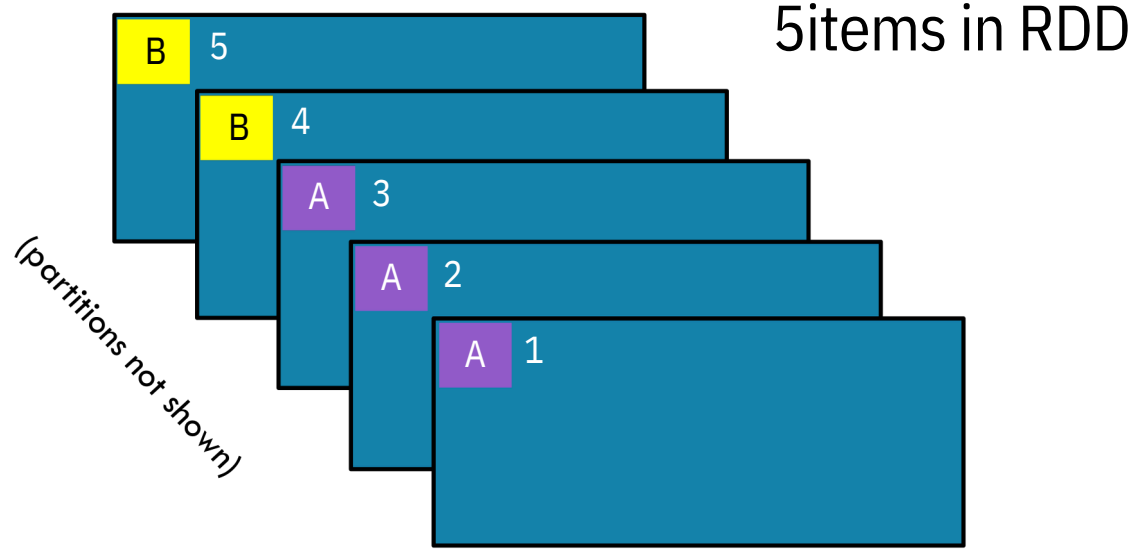
```
x = sc.parallelize(['John', 'Fred', 'Anna', 'James'])
y = x.groupBy(lambda w: w[0])
print [(k, list(v)) for (k, v) in y.collect()]
```

```
val x = sc.parallelize(
val  Array("John", "Fred", "Anna", "James"))
    y  = x.groupBy(w => w.charAt(0))
 println(y.collect().mkString(", "))
```

```
x: ['John', 'Fred', 'Anna', 'James']

y: [('A',['Anna']),('J',['John','James']),('F',['Fred'])]
```

GROUPBYKEY

Pair RDD: **x**

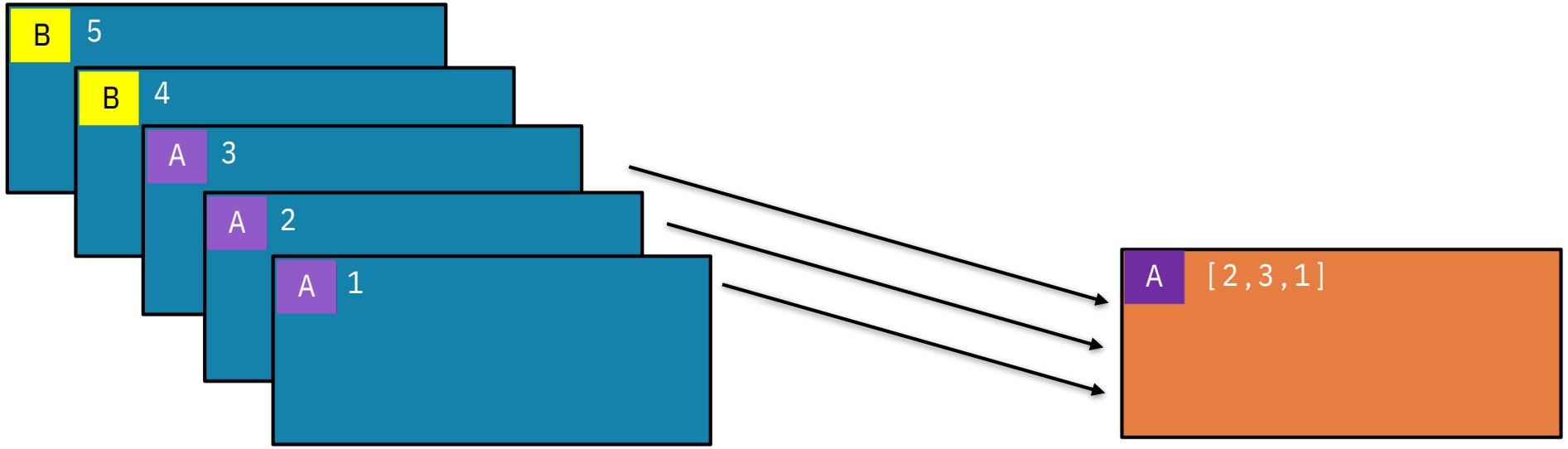


5items in RDD

GROUPBYKEY

Pair RDD: **x**

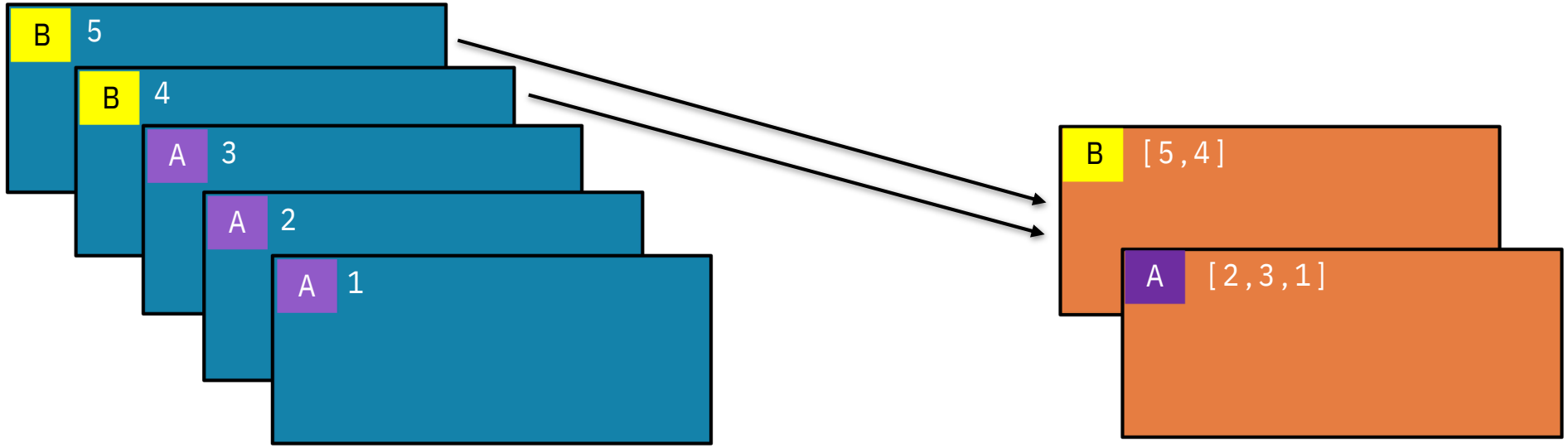
RDD: **y**



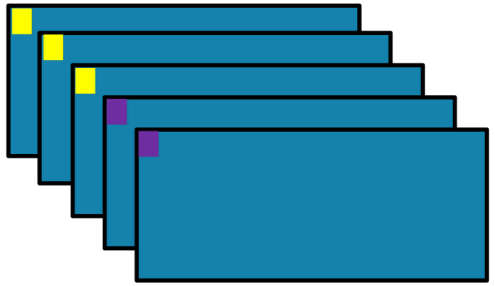
GROUPBYKEY

Pair RDD: **x**

RDD: **y**



GROUPBYKEY



```
groupByKey(numPartitions=None)
```

Group the values for each key in the original RDD. Create a new pair where the original key corresponds to this collected group of values.

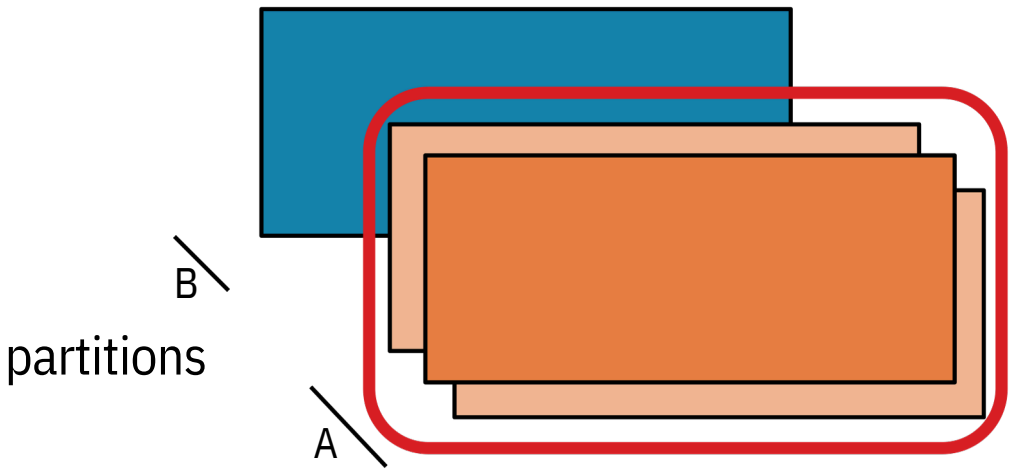
```python
x = sc.parallelize([('B',5),('B',4),('A',3),('A',2),('A',1)])
y = x.groupByKey()
print(x.collect())
print(list((j[0], list(j[1])) for j in y.collect()))
```

```
x: [('B', 5),('B', 4),('A', 3),('A', 2),('A', 1)]
y: [('A', [2, 3, 1]),('B',[5, 4])]
```

```scala
val x = sc.parallelize(
val y   Array(('B',5),('B',4),('A',3),('A',2),('A',1)))
      = x.groupByKey()
println(x.collect().mkString(",  "))
println(y.collect().mkString(", "))
```

MAPPARTITIONS

RDD: **x**

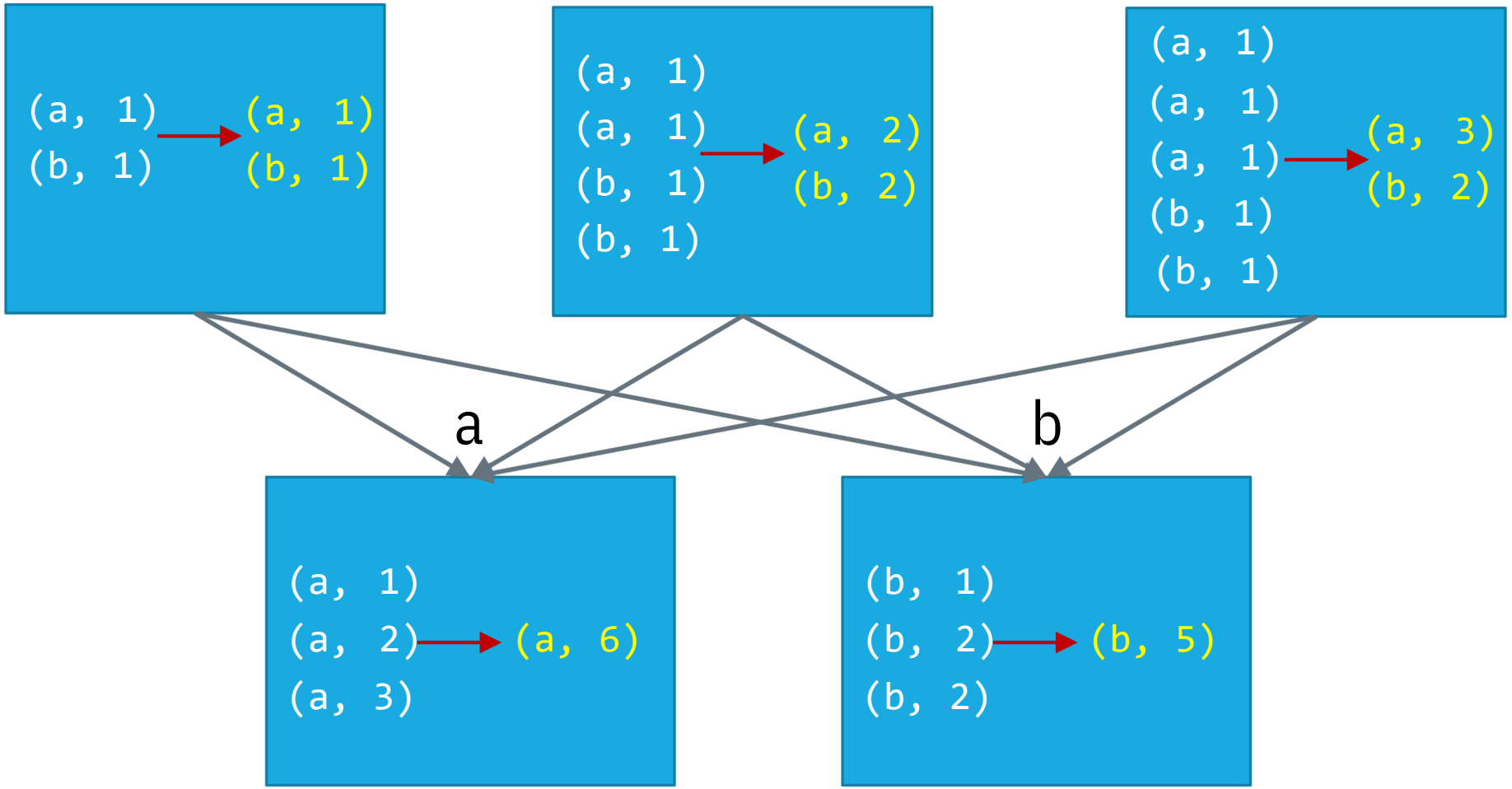


RDD: **y**

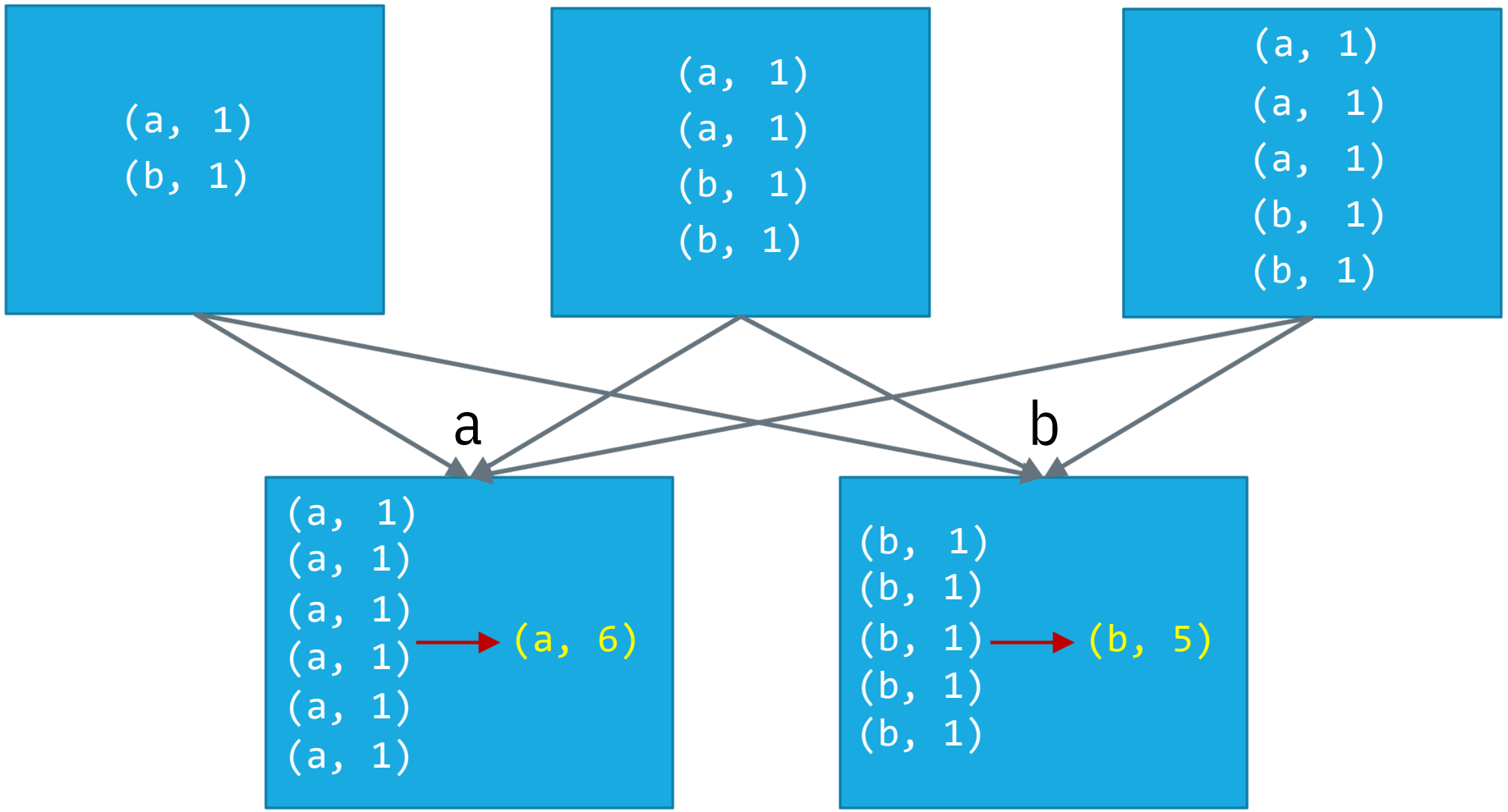
REDUCEBYKEY VS GROUPBYKEY

```
valwords = Array("one", "two", "two", "three", "three", "three")
valwordPairsRDD  = sc.parallelize(words).map(word => (word, 1))
valwordCountsWithReduce
                            = wordPairsRDD
  .reduceByKey(_ + _)
  .collect()

valwordCountsWithGroup  = wordPairsRDD
  .groupByKey()
  .map(t => (t._1, t._2.sum))
  .collect()
```

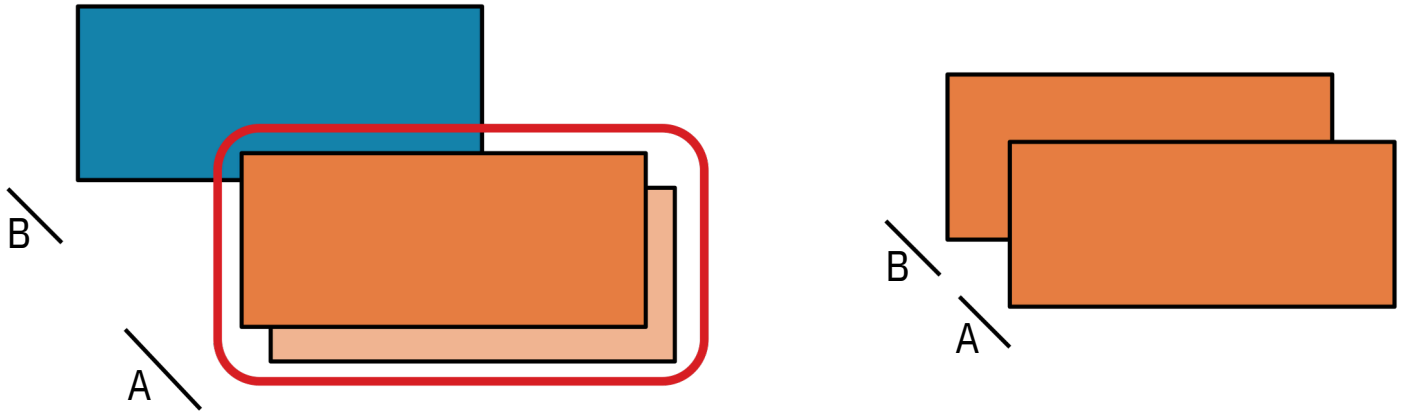
REDUCEBYKEY



GROUPBYKEY


(a, 1)
(b, 1)
(a, 1)
(a, 1)
(b, 1)
(b, 1)
(a, 1)
(a, 1)
(a, 1)
(b, 1)
(b, 1)
a
b
(a, 1)
(a, 1)
(a, 1)
(a, 1)
(a, 1)
(a, 1)
(a, 6)
(b, 1)
(b, 1)
(b, 1)
(b, 1)
(b, 1)
(b, 1)
(b, 5)

MAPPARTITIONS



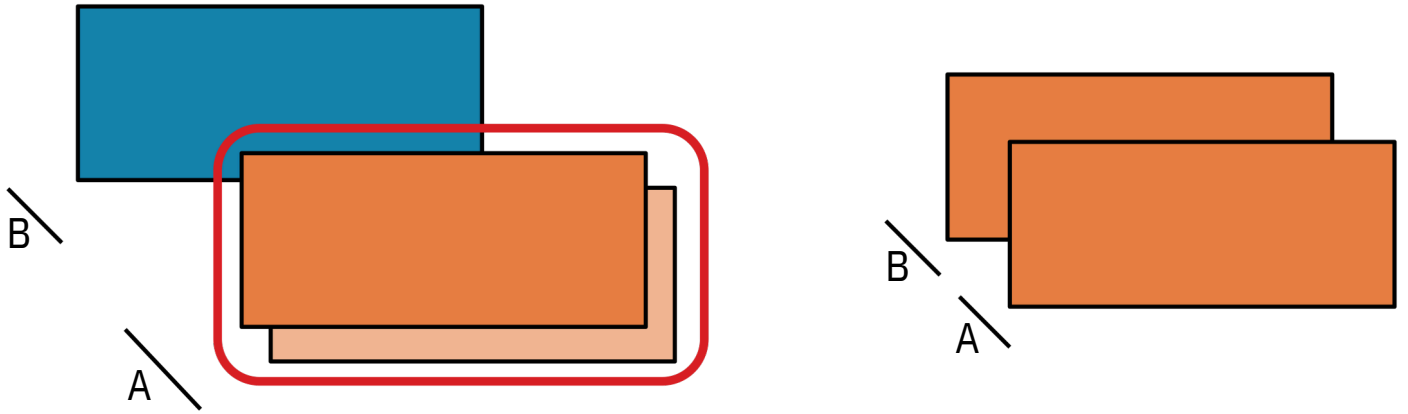

```
mapPartitions(f, preservesPartitioning=False)
```

Return a new RDD by applying a function to each partition of this RDD

```
x = sc.parallelize([1,2,3], 2)
def
    f(iterator): yield sum(iterator); yield 42

y = x.mapPartitions(f)

# glom() flattens elements on the same partition
print(x.glom().collect())
print(y.glom().collect())
```

```
x: [[1], [2, 3]]

y: [[1, 42], [5, 42]]
```

MAPPARTITIONS




```
mapPartitions(f, preservesPartitioning=False)
```

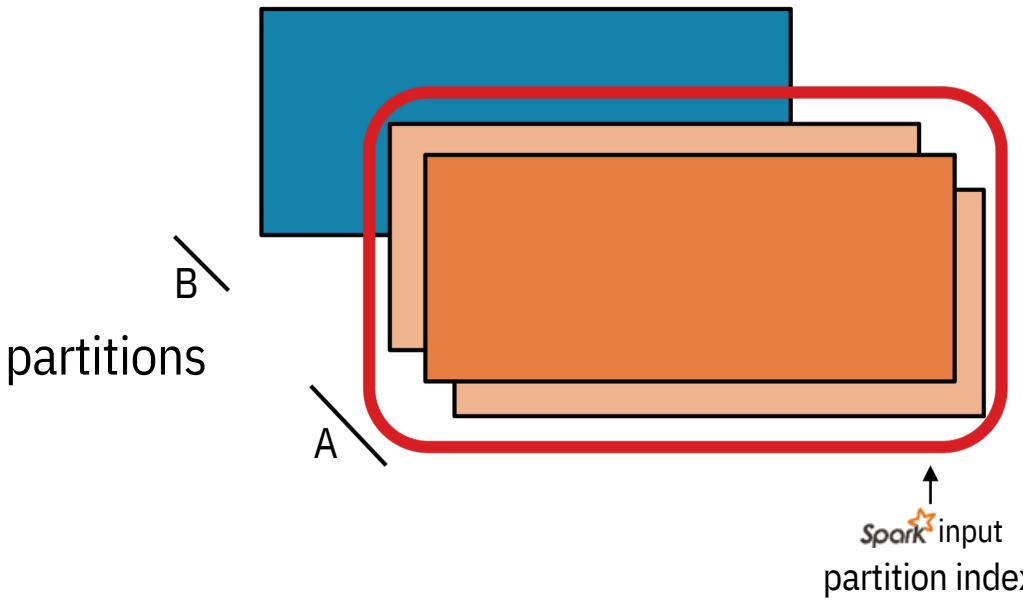
Return a new RDD by applying a function to each partition of this RDD

```
val x = sc.parallelize(Array(1,2,3), 2)
def f(i:Iterator[Int])={ (i.sum,42).productIterator
val                                                  }

    y = x.mapPartitions(f)

// glom() flattens elements on the same partition
val xOut =  x.glom().collect()
val yOut = y.glom().collect()
```

```
x:  Array(Array(1), Array(2, 3))

y:  Array(Array(1, 42), Array(5, 42))
```

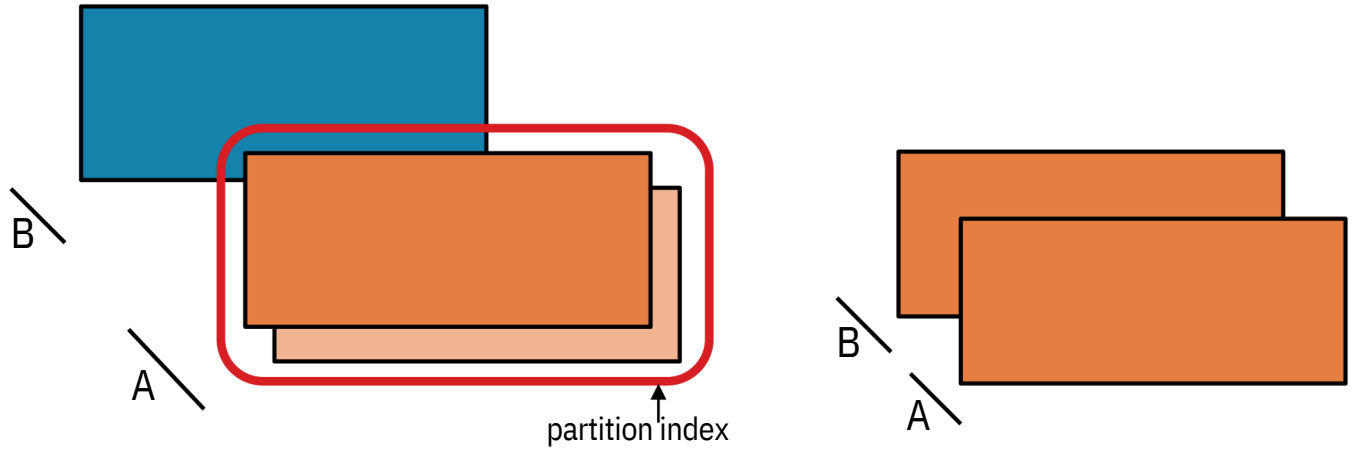
MAPPARTITIONSWITHINDEX

RDD: **x**



RDD: **y**

MAPPARTITIONSWITHINDEX



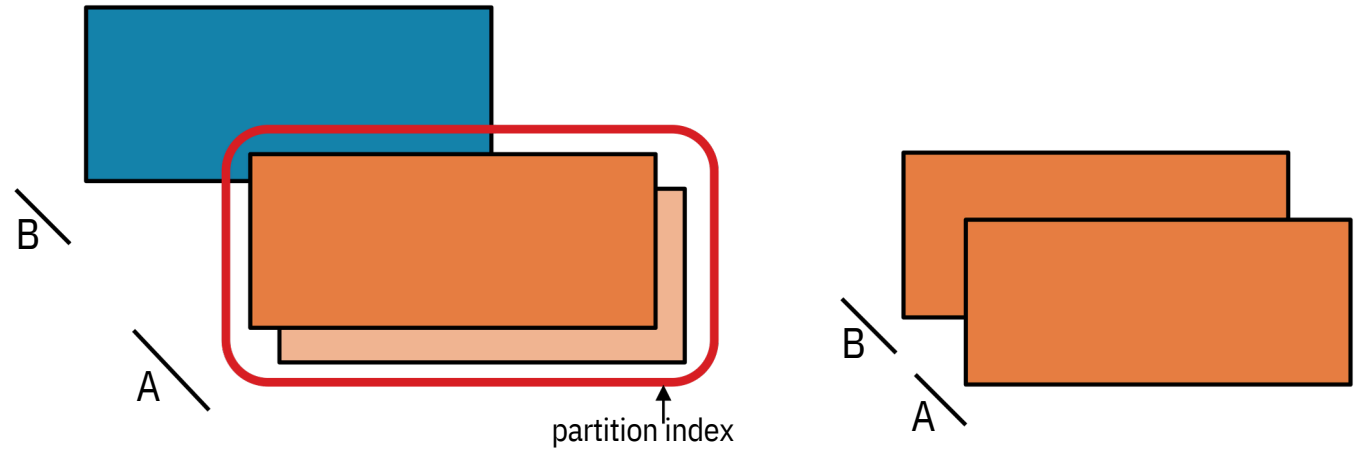

```
mapPartitionsWithIndex(f, preservesPartitioning=False)
```

Return a new RDD by applying a function to each partition of this RDD, while tracking the index of the original partition

```
x = sc.parallelize([1,2,3], 2)
def
    f(partitionIndex, iterator): yield (partitionIndex, sum(iterator))

y = x.mapPartitionsWithIndex(f)

# glom() flattens elements on the same partition
print(x.glom().collect())
print(y.glom().collect())
```

```
B       A
x: [[1], [2, 3]]

y: [[0, 1], [1, 5]]
```

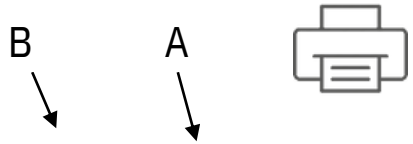
MAPPARTITIONSWITHINDEX




```
mapPartitionsWithIndex(f, preservesPartitioning=False)
```

Return a new RDD by applying a function to each partition of this RDD, while tracking the index of the original partition.

```
val x = sc.parallelize(Array(1,2,3), 2)
def
    f(partitionIndex:Int, i:Iterator[Int]) = {
 (partitionIndex, i.sum).productIterator
}
val
    y = x.mapPartitionsWithIndex(f)

// glom() flattens elements on the same partition
val xOut =  x.glom().collect()
val yOut = y.glom().collect()
```

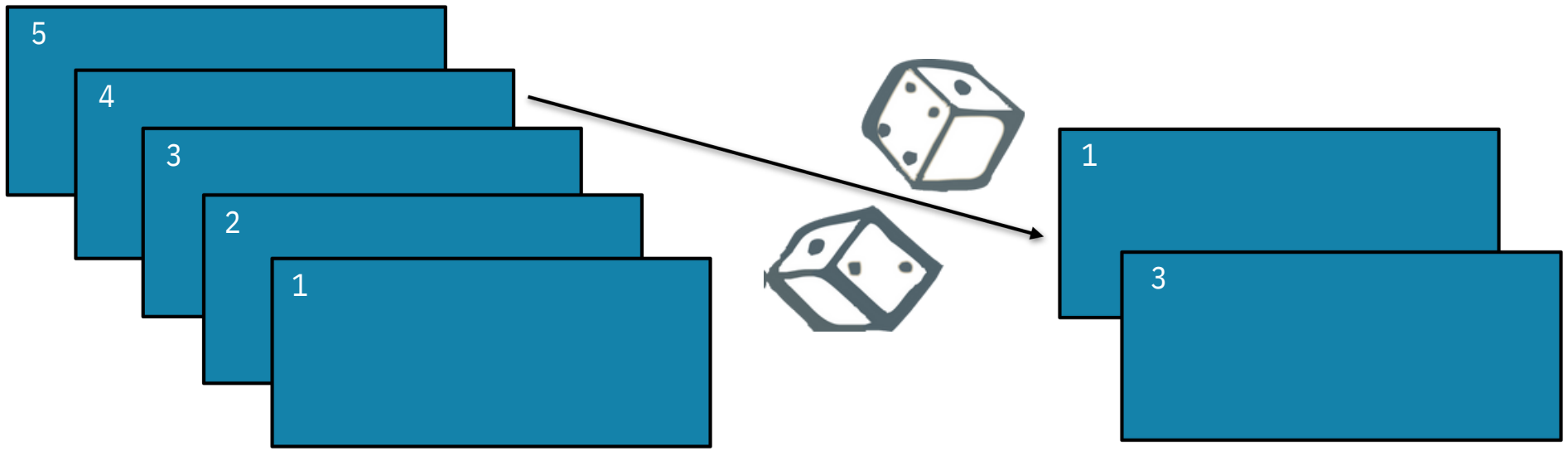



```
x:  Array(Array(1), Array(2, 3))

y:  Array(Array(0, 1), Array(1, 5))
```

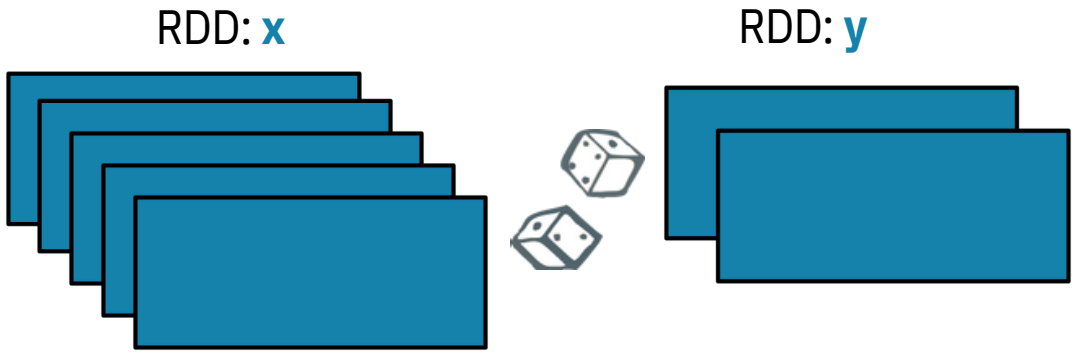
SAMPLE

RDD: **x**

RDD: **y**



SAMPLE




```
sample(withReplacement, fraction, seed=None)
```

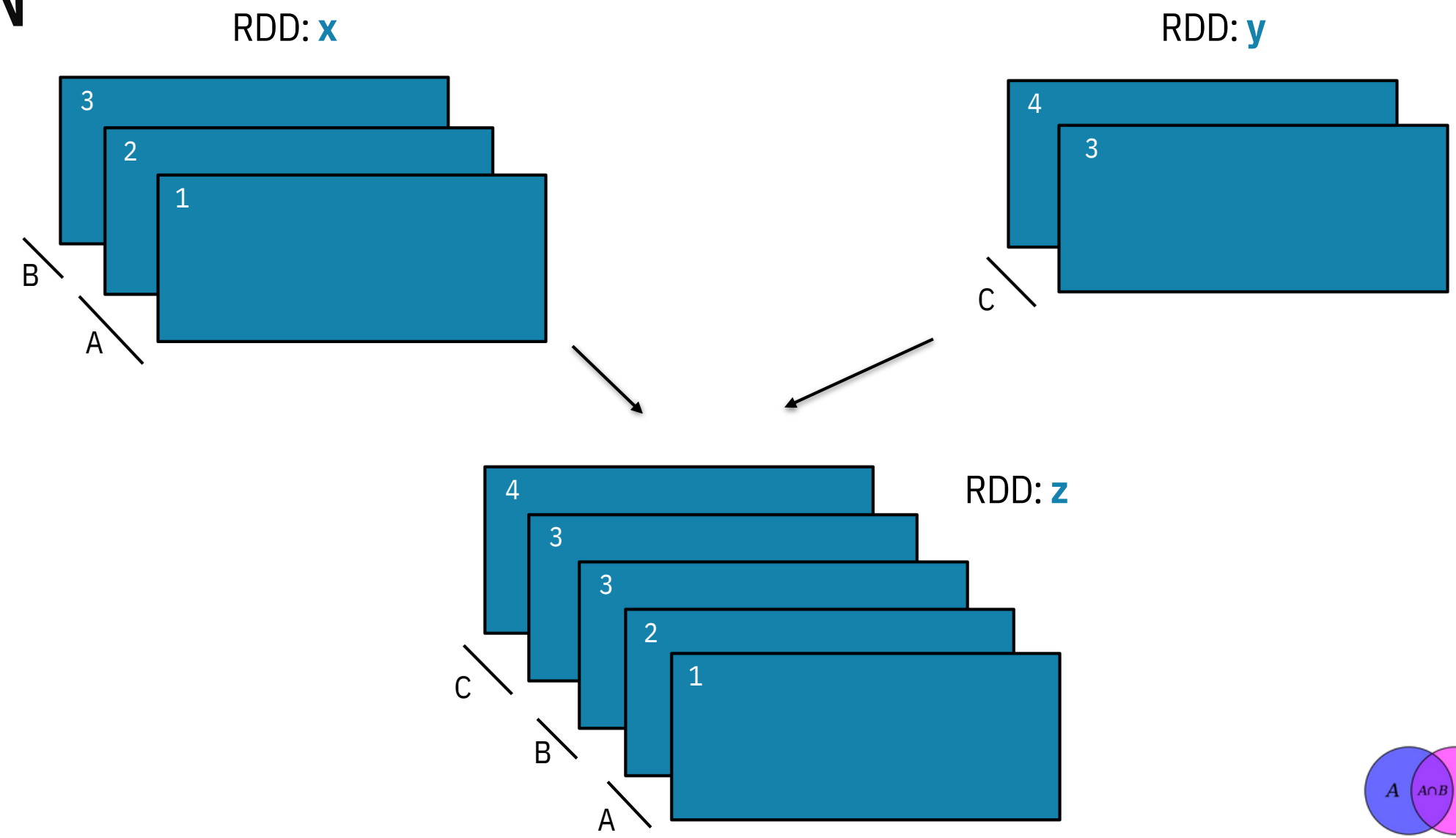
Return a new RDD containing a statistical sample of the original RDD

```
x = sc.parallelize([1, 2, 3, 4, 5])
y = x.sample(False, 0.4, 42)
print(x.collect())
print(y.collect())
```

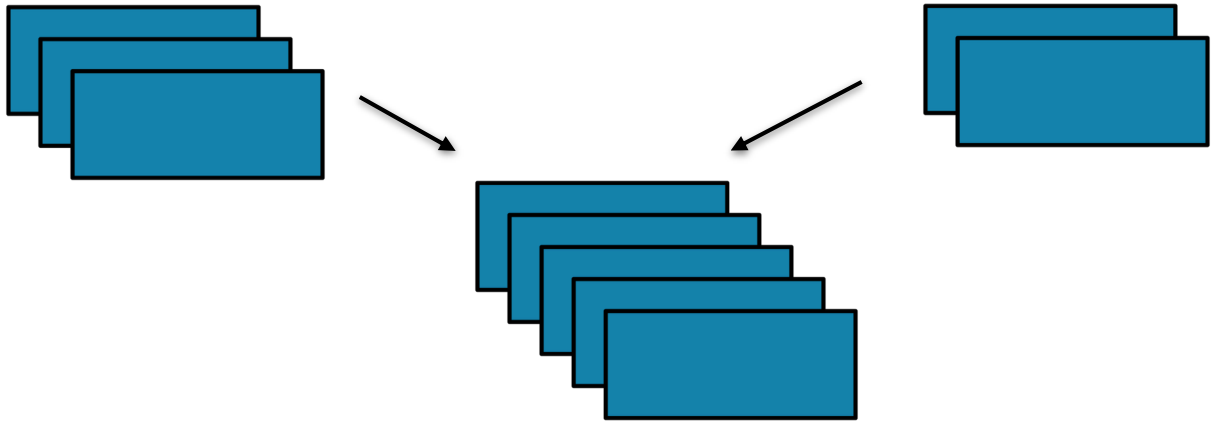
```
x:  [1, 2, 3, 4, 5]
y:
    [1, 3]
```

```
val x = sc.parallelize(Array(1, 2, 3, 4, 5))
val y = x.sample(false, 0.4)

// omitting seed will yield different output
println(y.collect().mkString(", "))
```

UNION



UNION



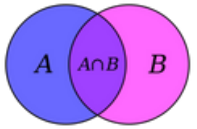
Return a new RDD containing all items from two original RDDs. Duplicates are *not* culled.

`union(otherRDD)`

```python
x = sc.parallelize([1,2,3], 2)
y = sc.parallelize([3,4], 1)
z = x.union(y)
print(z.glom().collect())
```

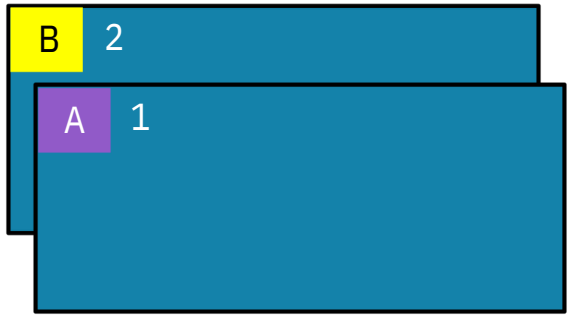
```
x: [1, 2, 3]

y: [3, 4]

z: [[1], [2, 3], [3, 4]]
```

```scala
val x =ysc.parallelize(Array(1,2,3), 2)
val z = sc.parallelize(Array(3,4), 1)
val zO=tx.union(y)
val      = z.glom().collect()
```

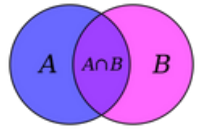


JOIN

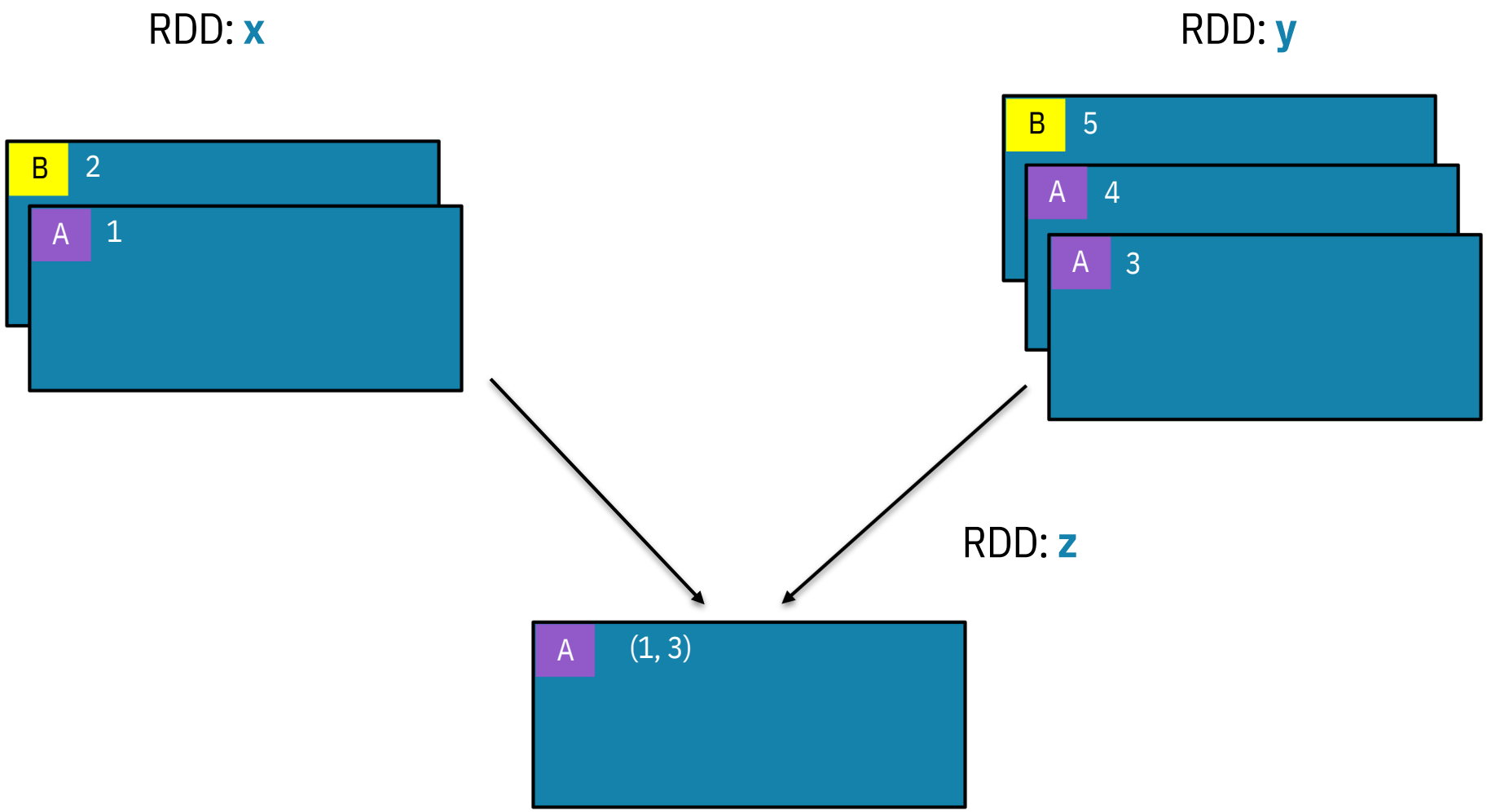
RDD: **x**

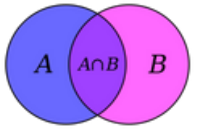


RDD: **y**



JOIN

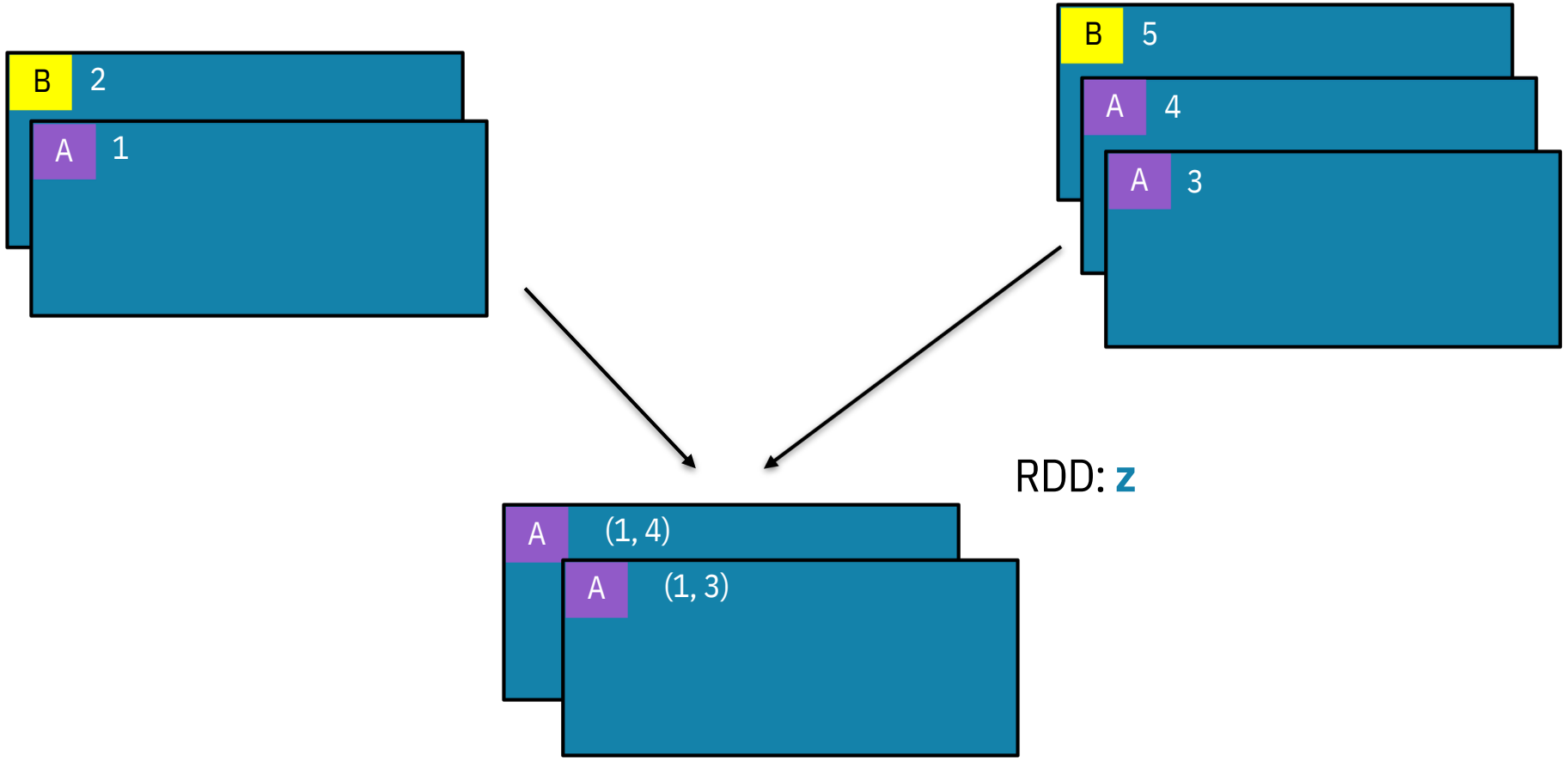


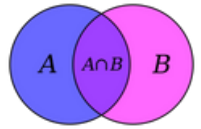


JOIN

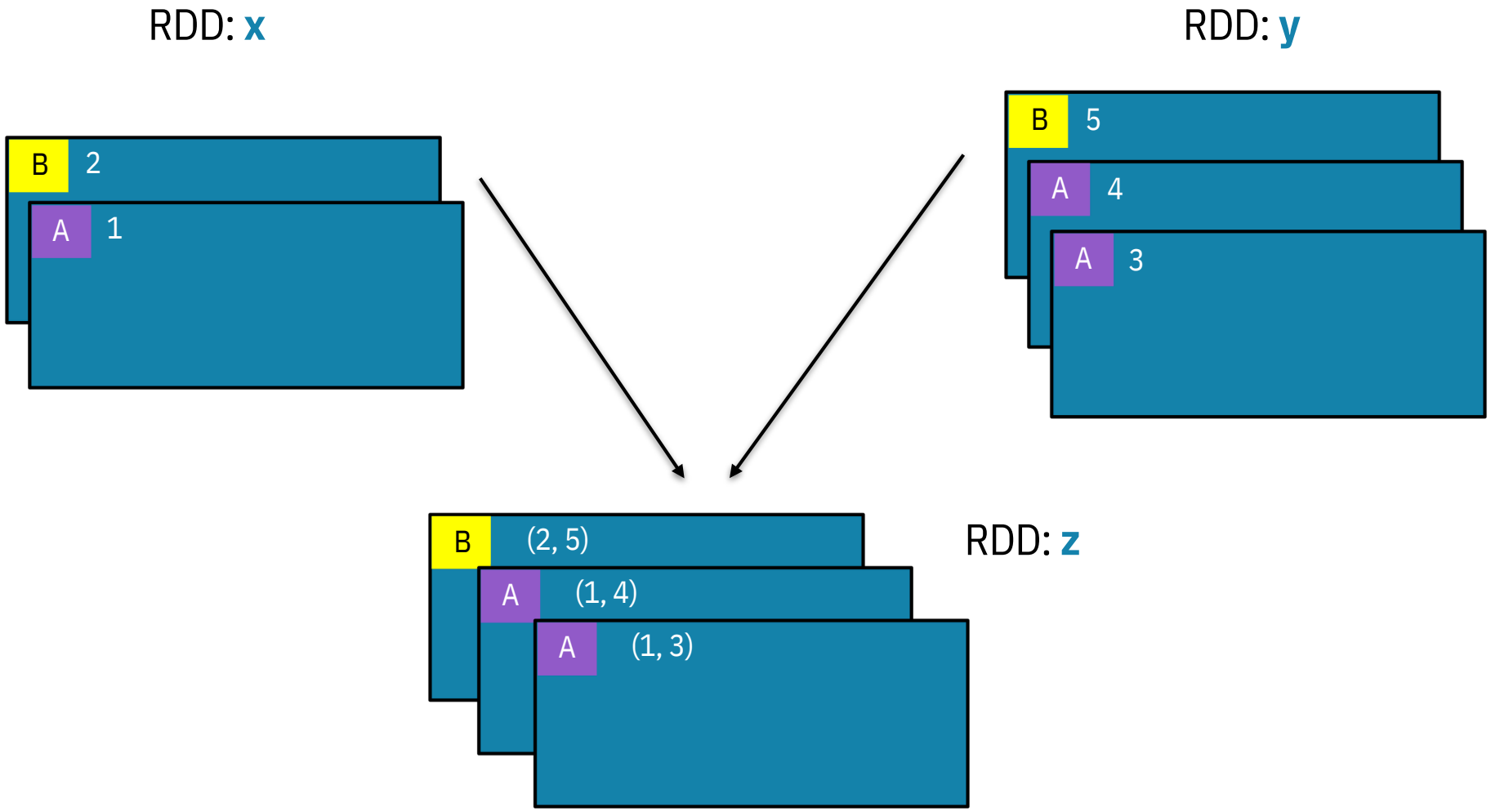
RDD: **x**

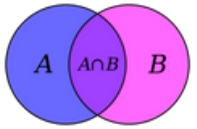
RDD: **y**



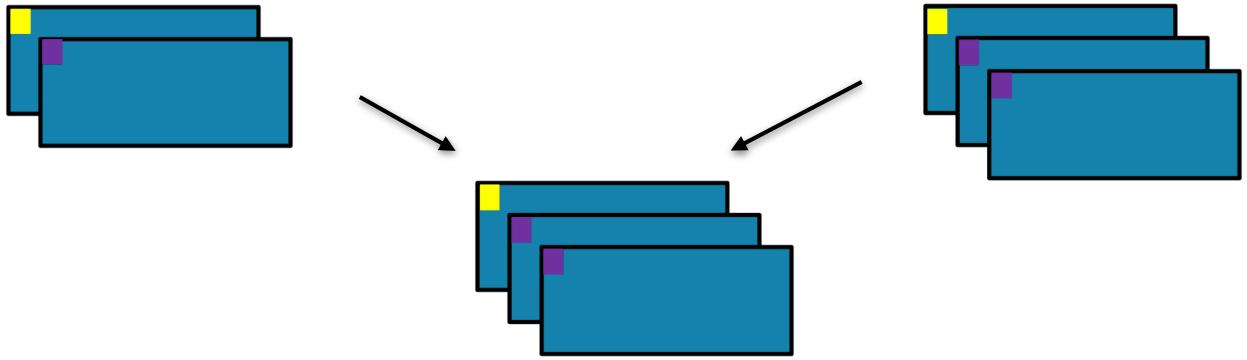


JOIN





JOIN



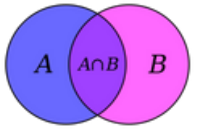
Return a new RDD containing all pairs of elements having the same key in the original RDDs

```
union(otherRDD, numPartitions=None)
```

```python
x = sc.parallelize([("a", 1), ("b", 2)])
y = sc.parallelize([("a", 3), ("a", 4), ("b", 5)])
z = x.join(y)
print(z.collect())
```

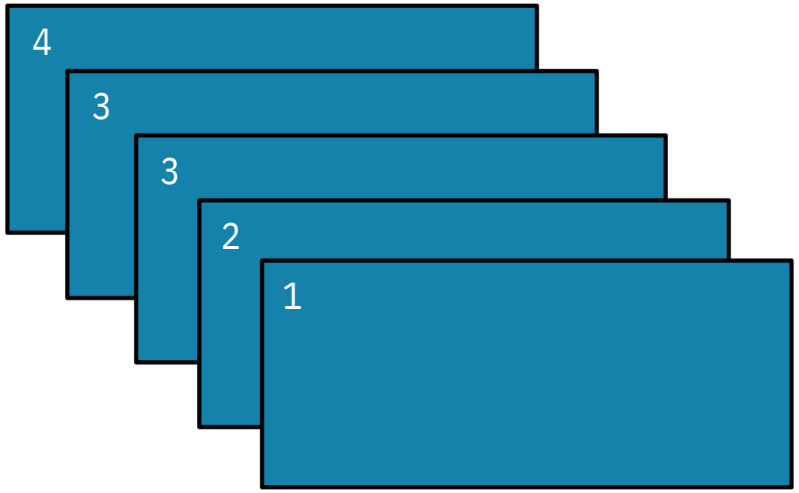
```
x: [("a", 1), ("b", 2)]

y: [("a", 3), ("a", 4), ("b", 5)]

z: [('a', (1, 3)), ('a', (1, 4)), ('b', (2, 5))]
```

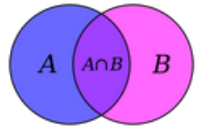
```scala
val x = sc.parallelize(Array(("a", 1), ("b", 2)))
val y = sc.parallelize(Array(("a", 3), ("a", 4), ("b", 5)))
val z = x.join(y)
println(z.collect().mkString(", "))
```



DISTINCT

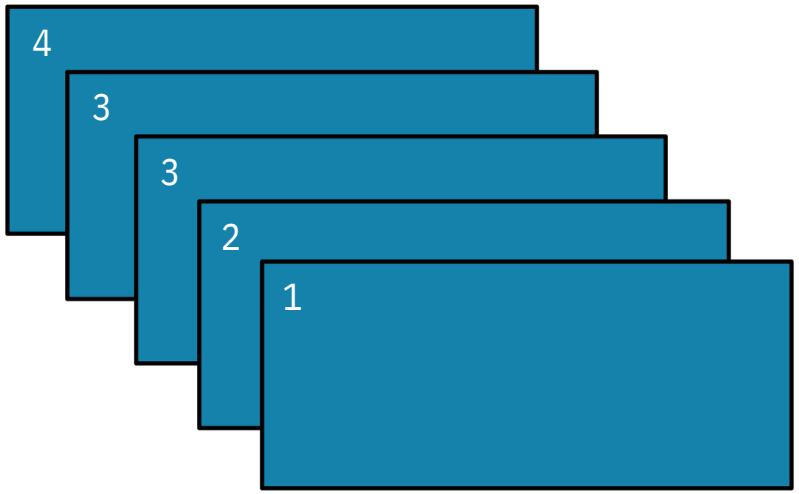
RDD: **x**



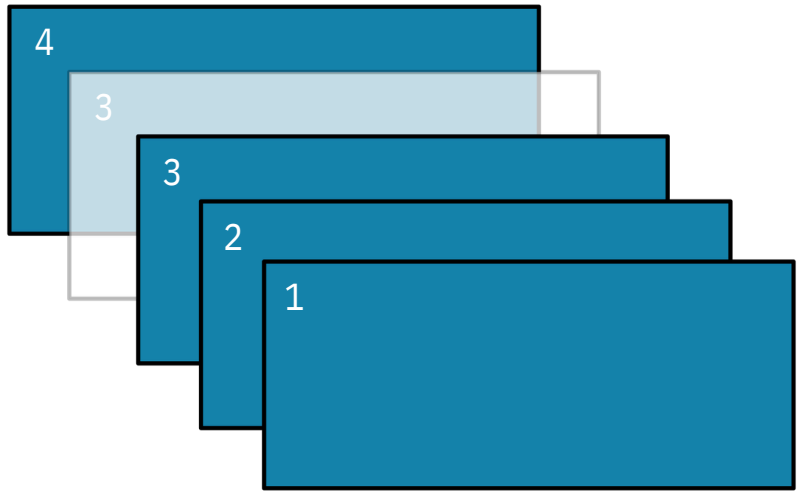


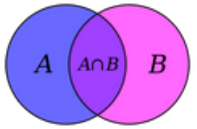
DISTINCT

RDD: **x**



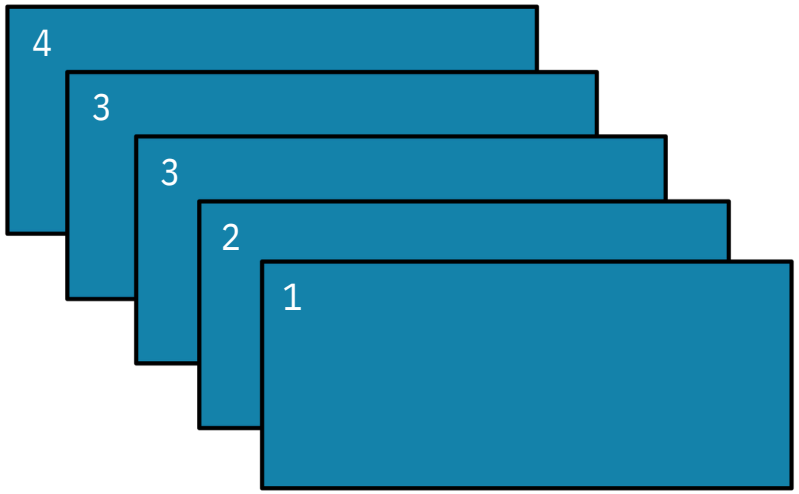
RDD: **y**



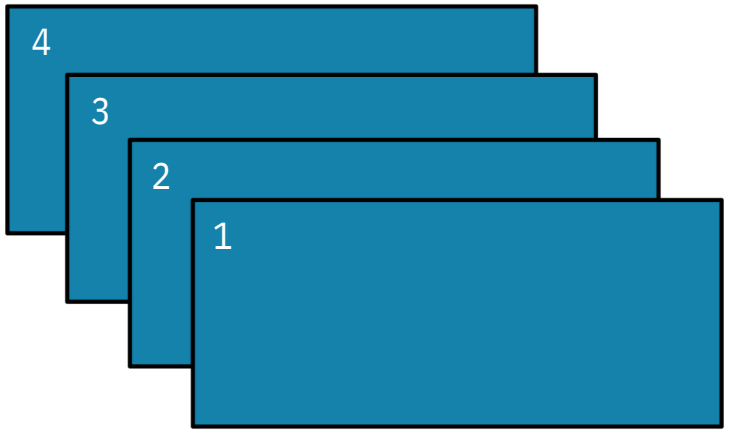


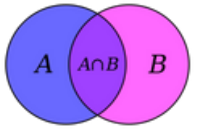
DISTINCT

RDD: **x**

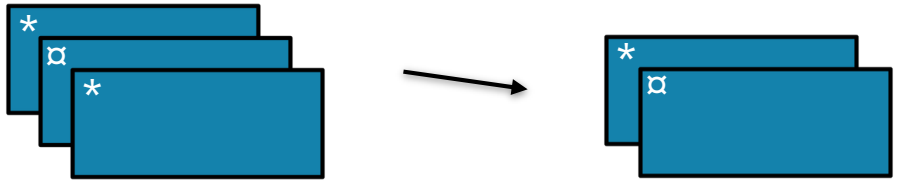


RDD: **y**





DISTINCT



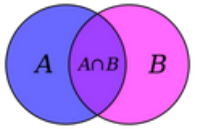
Return a new RDD containing distinct items from the original RDD (omitting all duplicates)

```
distinct(numPartitions=None)
```

```
x = sc.parallelize([1,2,3,3,4])
y = x.distinct()

print(y.collect())
```

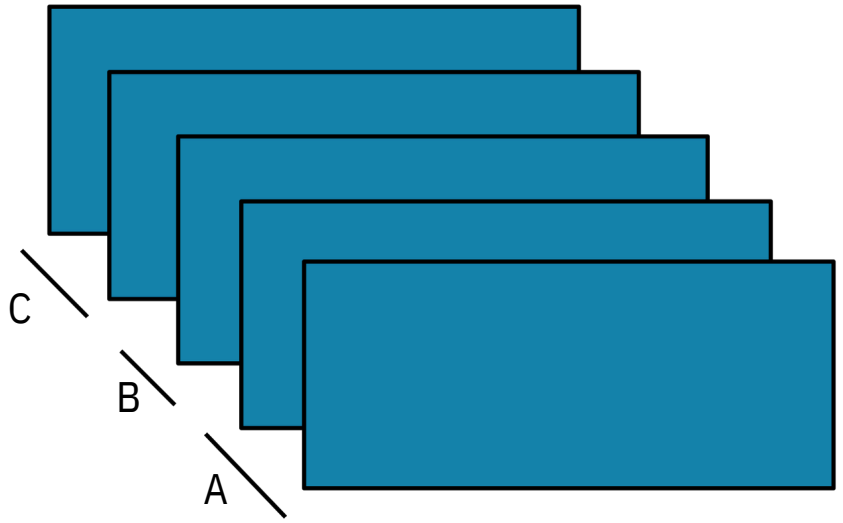
```
val x = sc.parallelize(Array(1,2,3,3,4))
val y = x.distinct()

println(y.collect().mkString(", "))
```

```
x:  [1, 2, 3, 3, 4]
y:
    [1, 2, 3, 4]
```



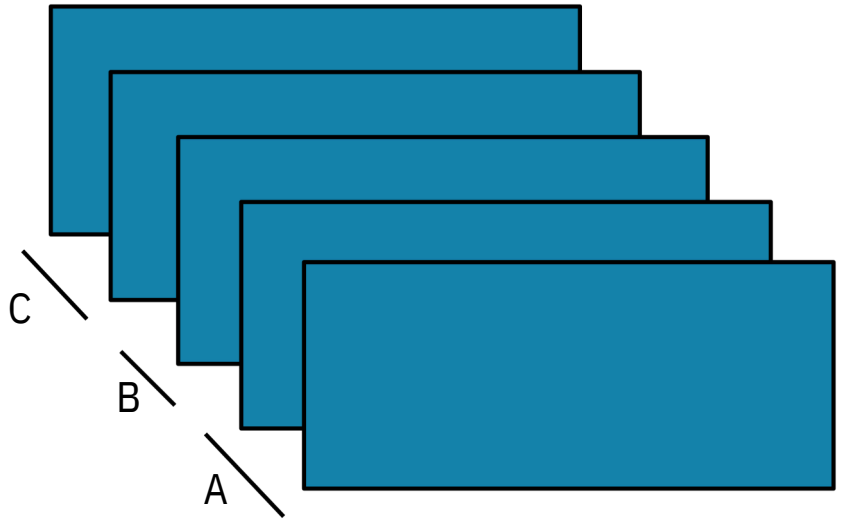
COALESCE

RDD: **x**

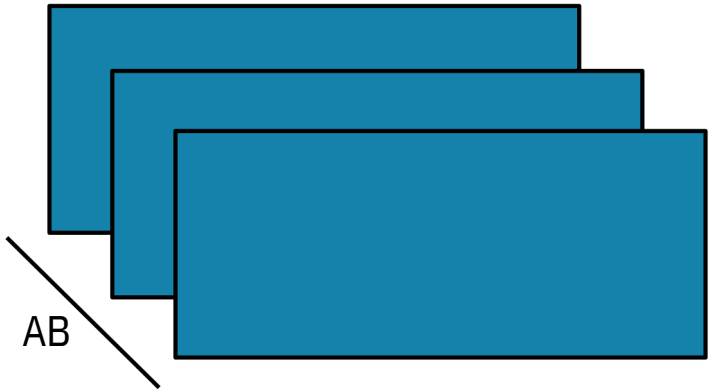


COALESCE

RDD: **x**

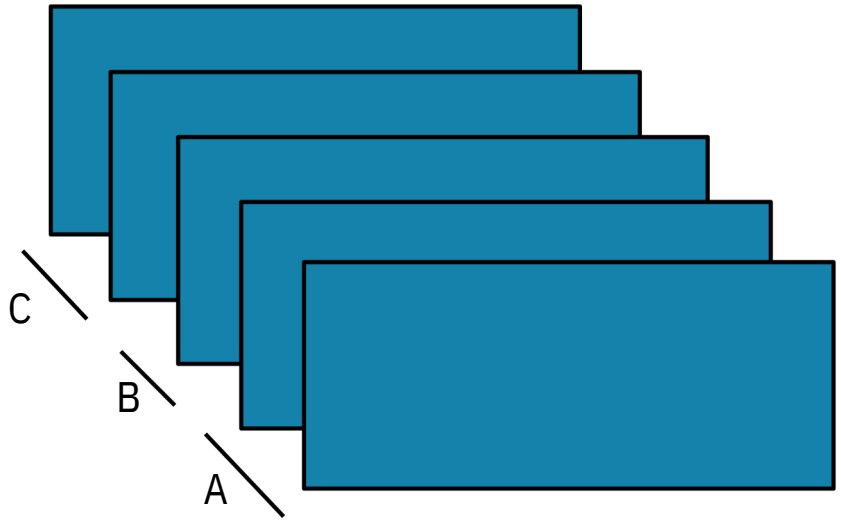


RDD: **y**

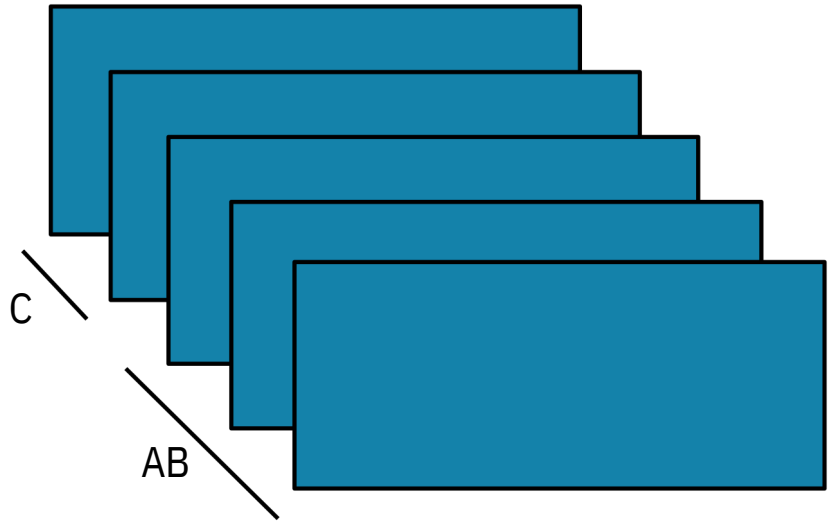


COALESCE

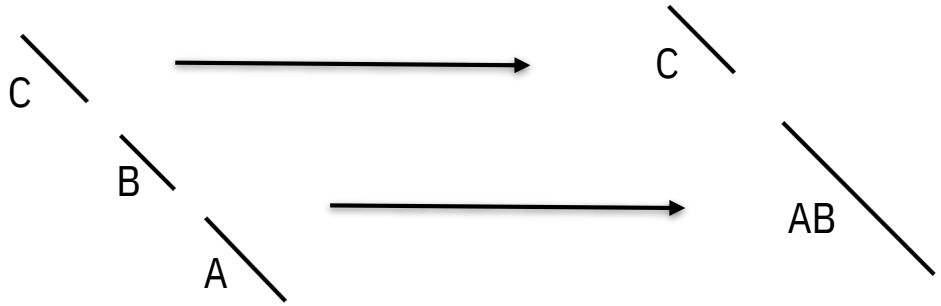
RDD: **x**



RDD: **y**



COALESCE



Return a new RDD which is reduced to a smaller number of partitions

```
coalesce(numPartitions, shuffle=False)
```

```python
x = sc.parallelize([1, 2, 3, 4, 5], 3)
y = x.coalesce(2)
print(x.glom().collect())
print(y.glom().collect())
```

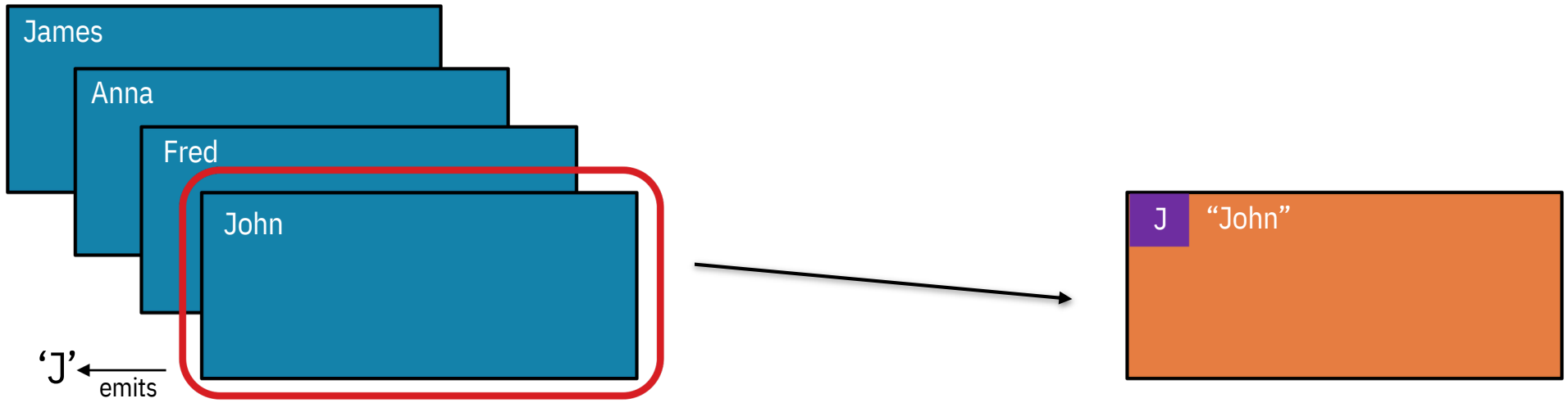
```
x: [[1], [2, 3], [4, 5]]

y: [[1], [2, 3, 4, 5]]
```

```scala
val x = sc.parallelize(Array(1, 2, 3, 4, 5), 3)
val y = x.coalesce(2)
val xOut =  x.glom().collect()
val yOut = y.glom().collect()
```

KEYBY

RDD: **x**

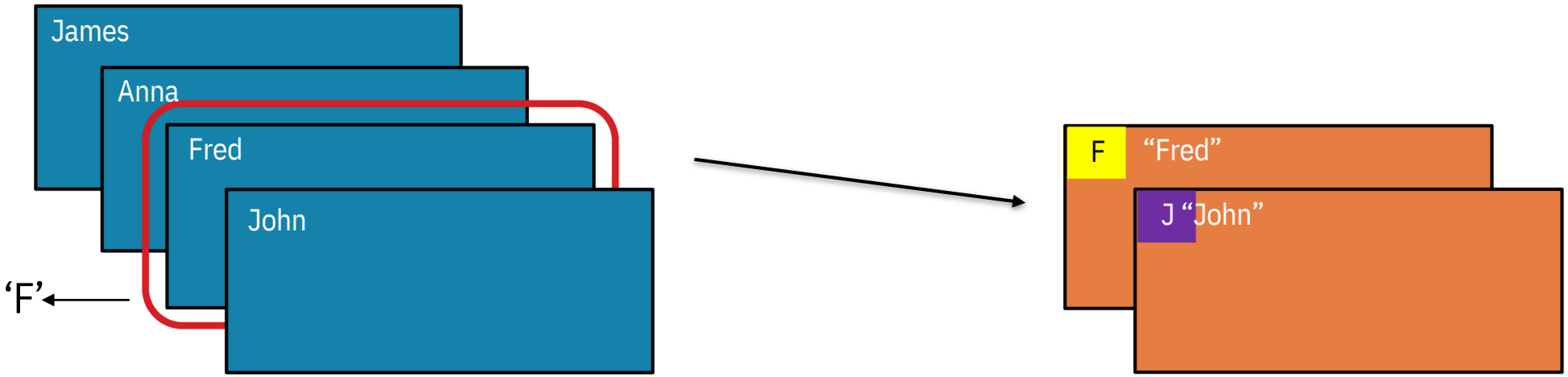
RDD: **y**



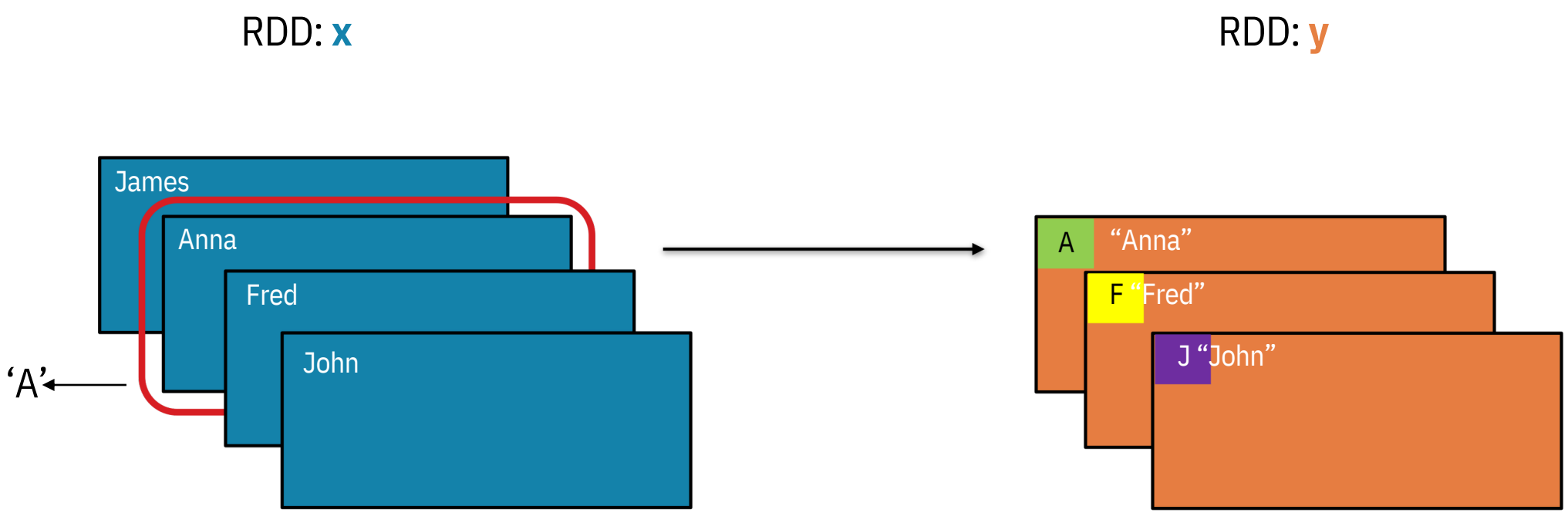
KEYBY

RDD: **x**

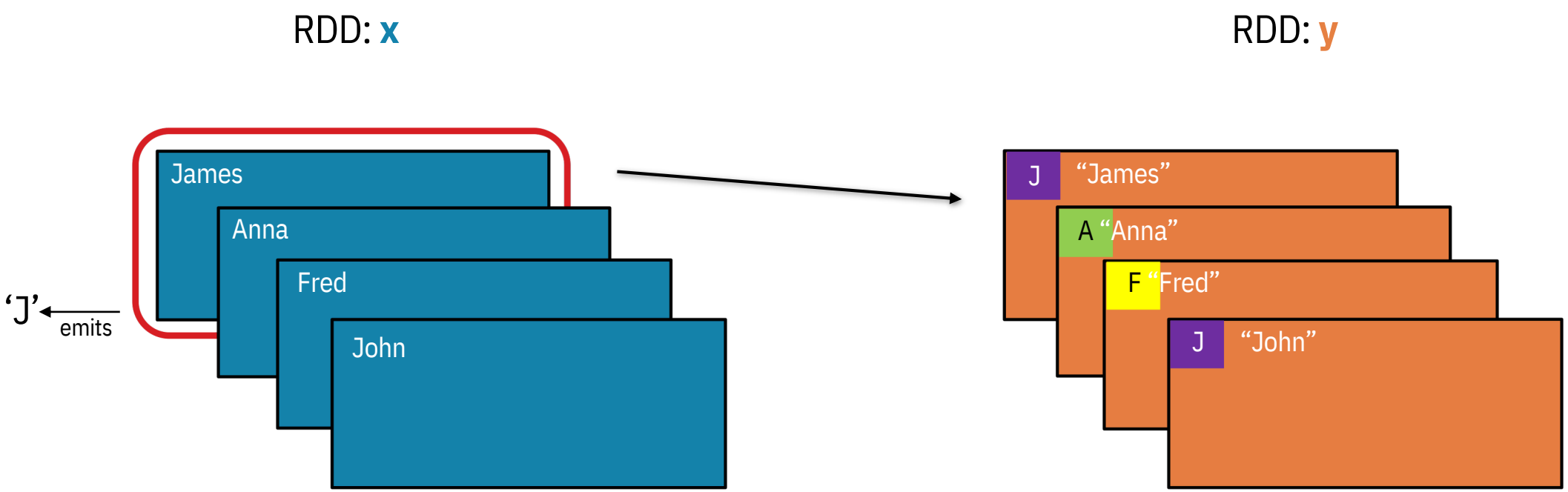
RDD: **y**



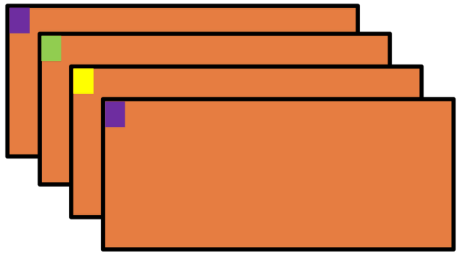
KEYBY



KEYBY


RDD: x
RDD: y
James
Anna
Fred
John
'J'
emits
J
"James"
A
"Anna"
F
"Fred"
J
"John"

KEYBY



keyBy(f)

Create a Pair RDD, forming one pair for each item in the original RDD. The pair's key is calculated from the value via a user-supplied function.

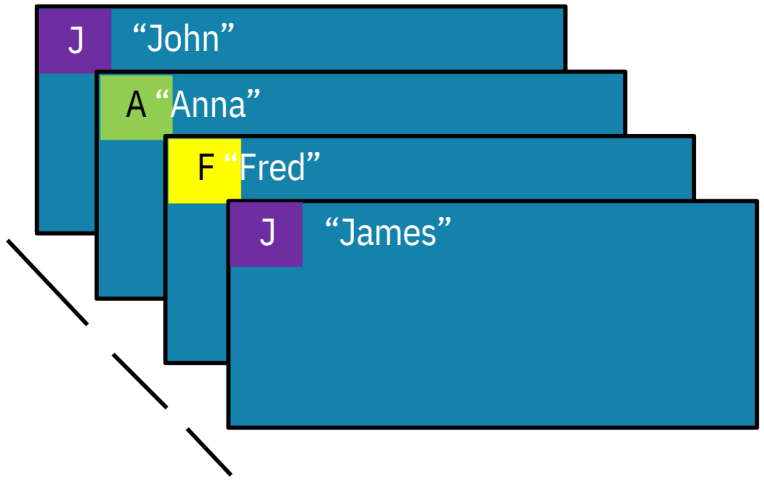
```
x = sc.parallelize(['John', 'Fred', 'Anna', 'James'])
y = x.keyBy(lambda w: w[0])
print y.collect()
```

```
val x = sc.parallelize(
val  Array("John", "Fred", "Anna", "James"))
    y = x.keyBy(w => w.charAt(0))
println(y.collect().mkString(", "))
```

```
x: ['John', 'Fred', 'Anna', 'James']

y: [('J','John'),('F','Fred'),('A','Anna'),('J','James')]
```

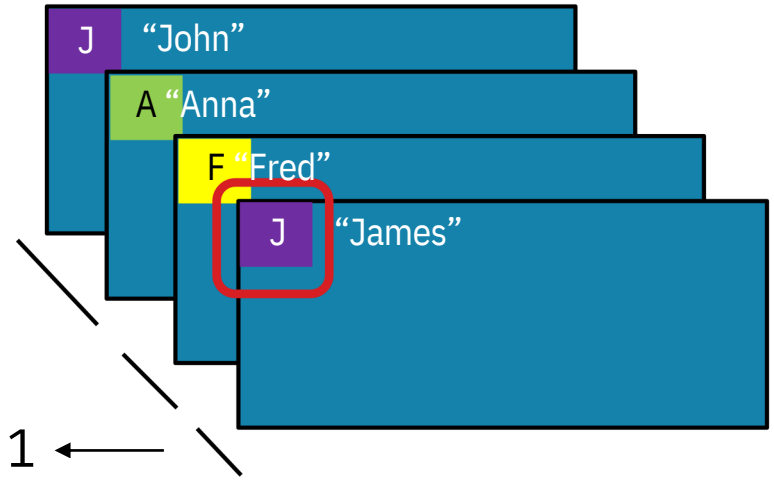
PARTITIONBY

RDD: **x**

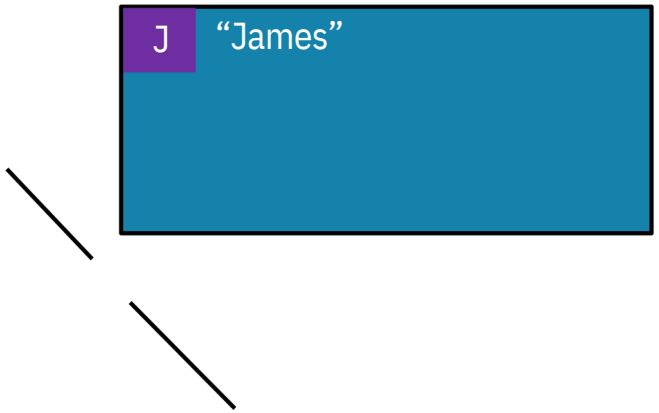


PARTITIONBY

RDD: **x**

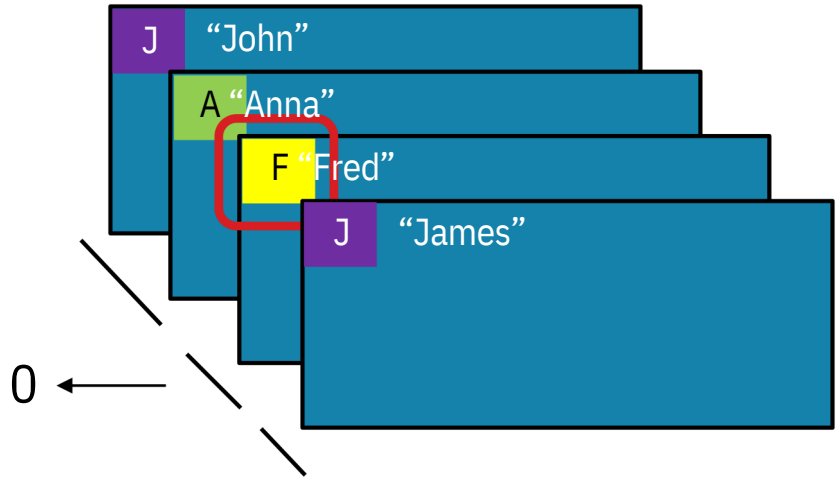


RDD: **y**

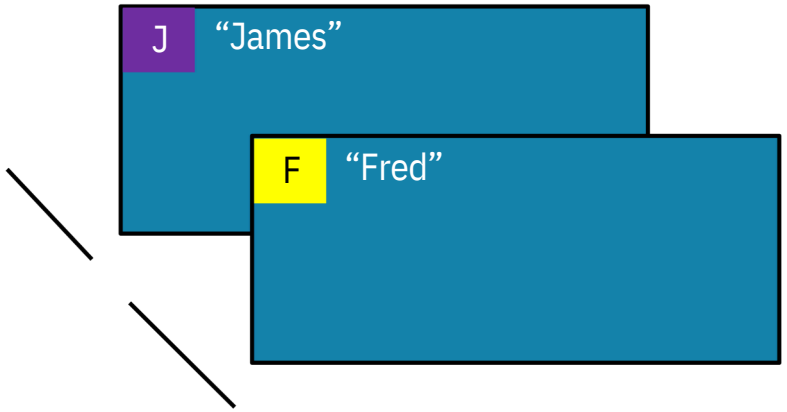


PARTITIONBY

RDD: **x**

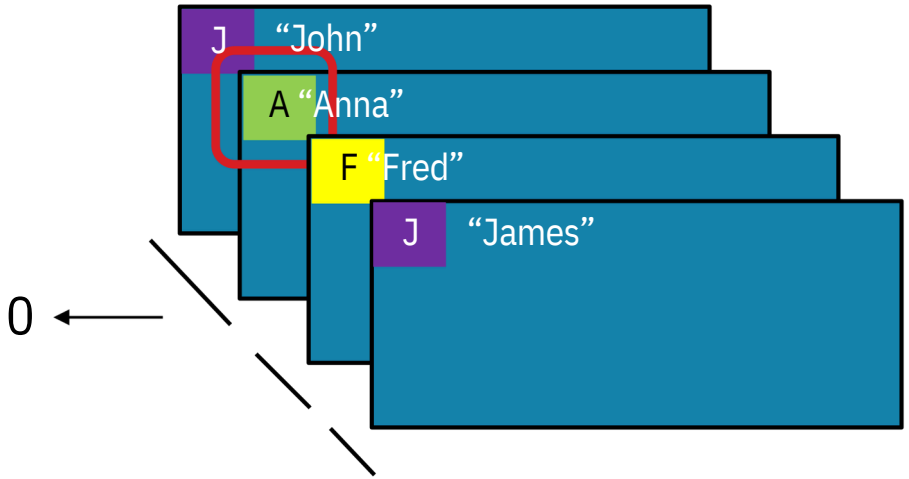


RDD: **y**

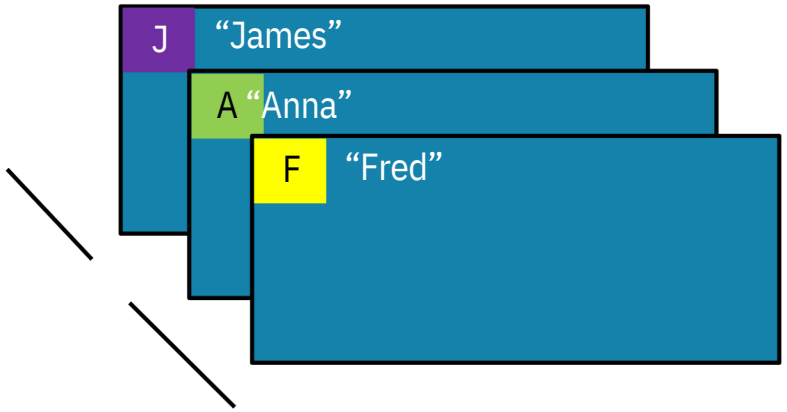


PARTITIONBY

RDD: **x**

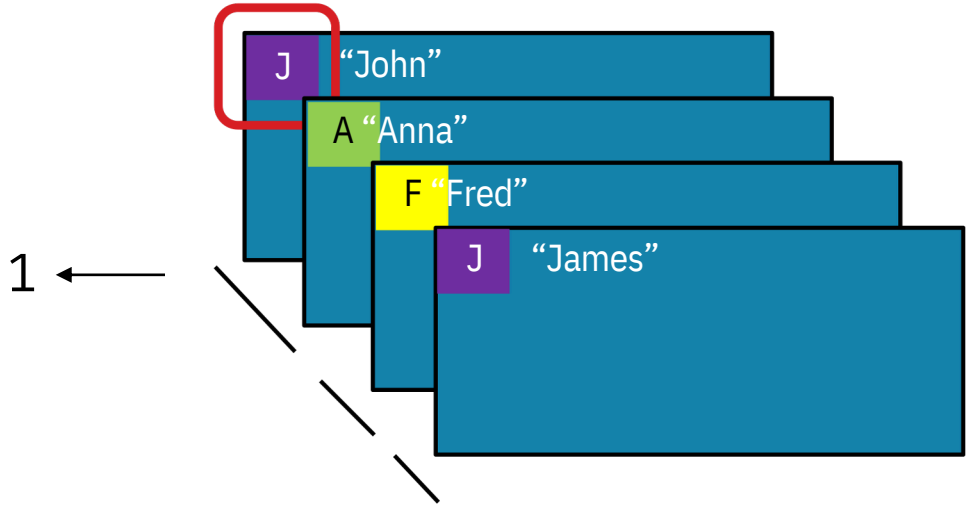


RDD: **y**

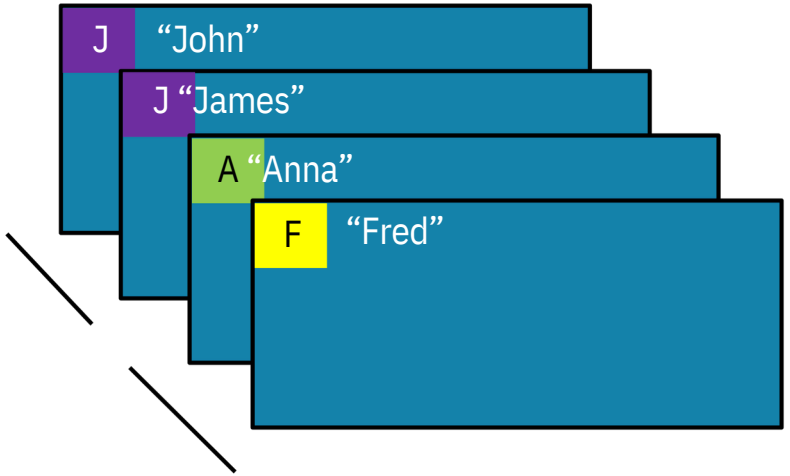


PARTITIONBY

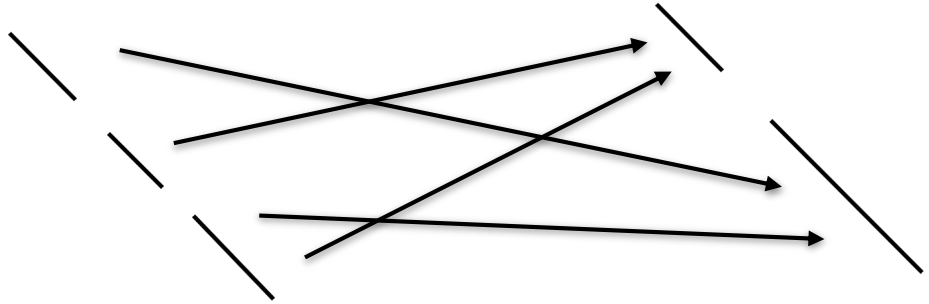
RDD: **x**



RDD: **y**



PARTITIONBY



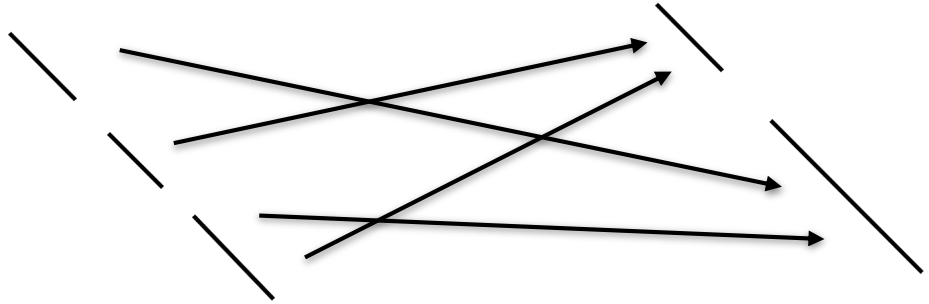
Return a new RDD with the specified number of partitions, placing original items into the partition returned by a user supplied function

```
partitionBy(numPartitions, partitioner=portable_hash)
```

```
x = sc.parallelize([('J','James'),('F','Fred'),
                    ('A','Anna'),('J','John')], 3)

y = x.partitionBy(2, lambda w: 0 if w[0] < 'H' else 1)
print  x.glom().collect()
print y.glom().collect()
```

```
x: [[('J', 'James')], [('F', 'Fred')],
    [('A', 'Anna'), ('J', 'John')]]
   [[('A', 'Anna'), ('F', 'Fred')],
y:
    [('J', 'James'), ('J', 'John')]]
```

PARTITIONBY



Return a new RDD with the specified number of partitions, placing original items into the partition returned by a user supplied function.

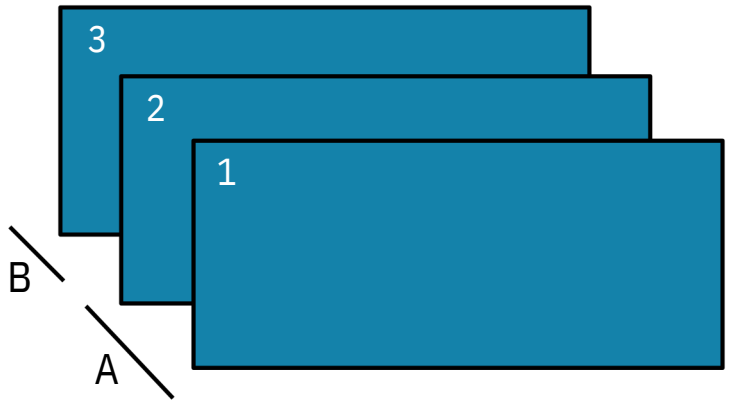
```
partitionBy(numPartitions, partitioner=portable_hash)
```

```
import org.apache.spark.Partitioner
val x = sc.parallelize(Array(('J',"James"),('F',"Fred"),
                             ('A',"Anna"),('J',"John")), 3)

val y = x.partitionBy(new Partitioner() {
  val numPartitions = 2
  def getPartition(k:Any) = {
     if (k.asInstanceOf[Char] < 'H') 0 else 1
  }
})
val
    yOut = y.glom().collect()
```

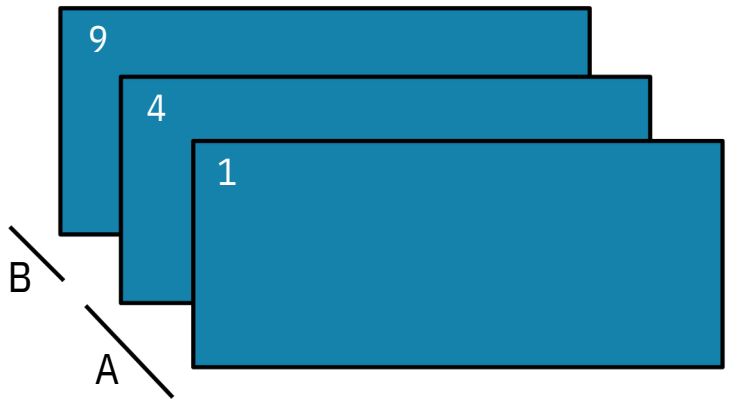
```
x: Array(Array((A,Anna), (F,Fred)),
         Array((J,John), (J,James)))

y: Array(Array((F,Fred), (A,Anna)),
         Array((J,John), (J,James)))
```

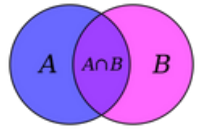
ZIP

RDD: **x**

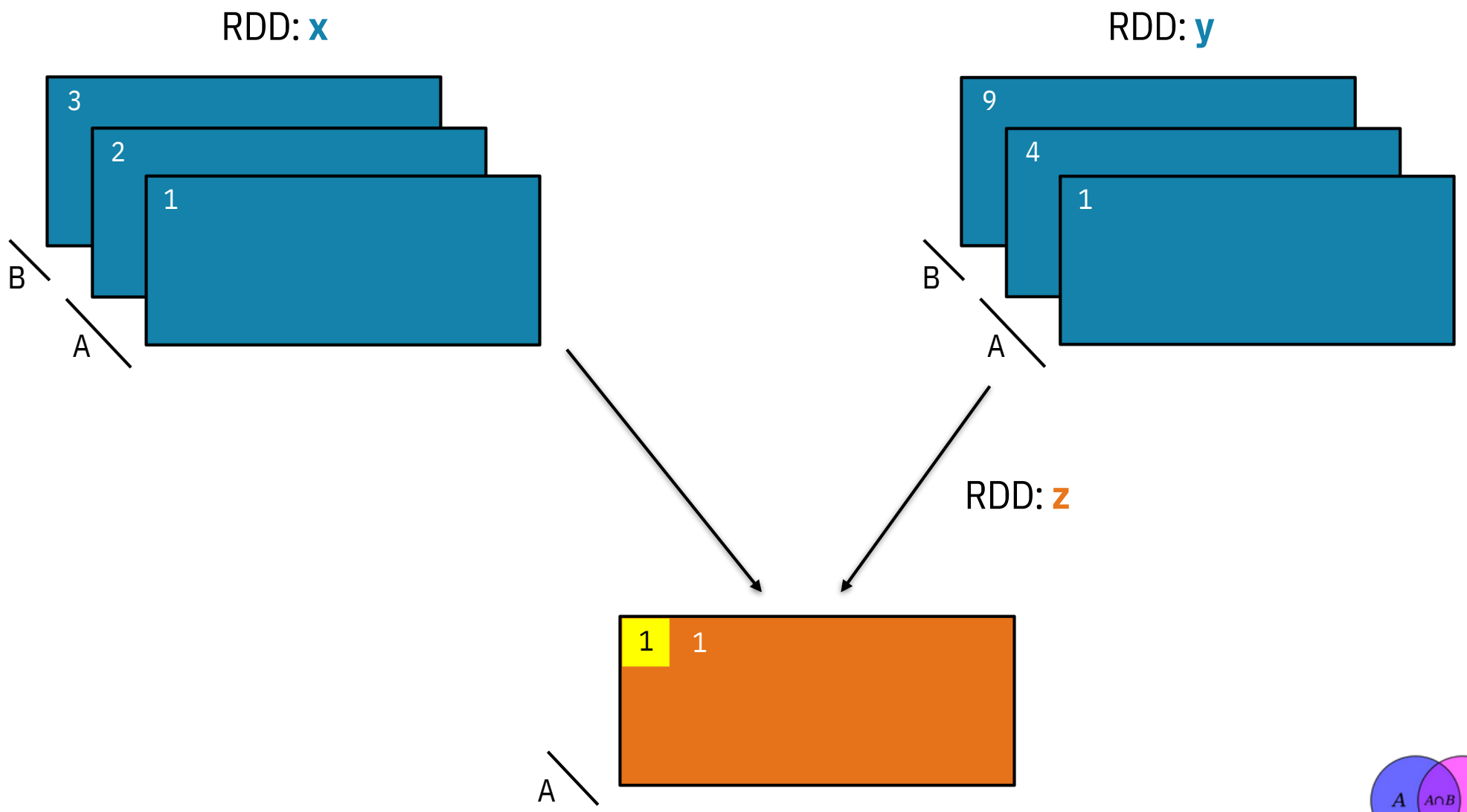


RDD: **y**

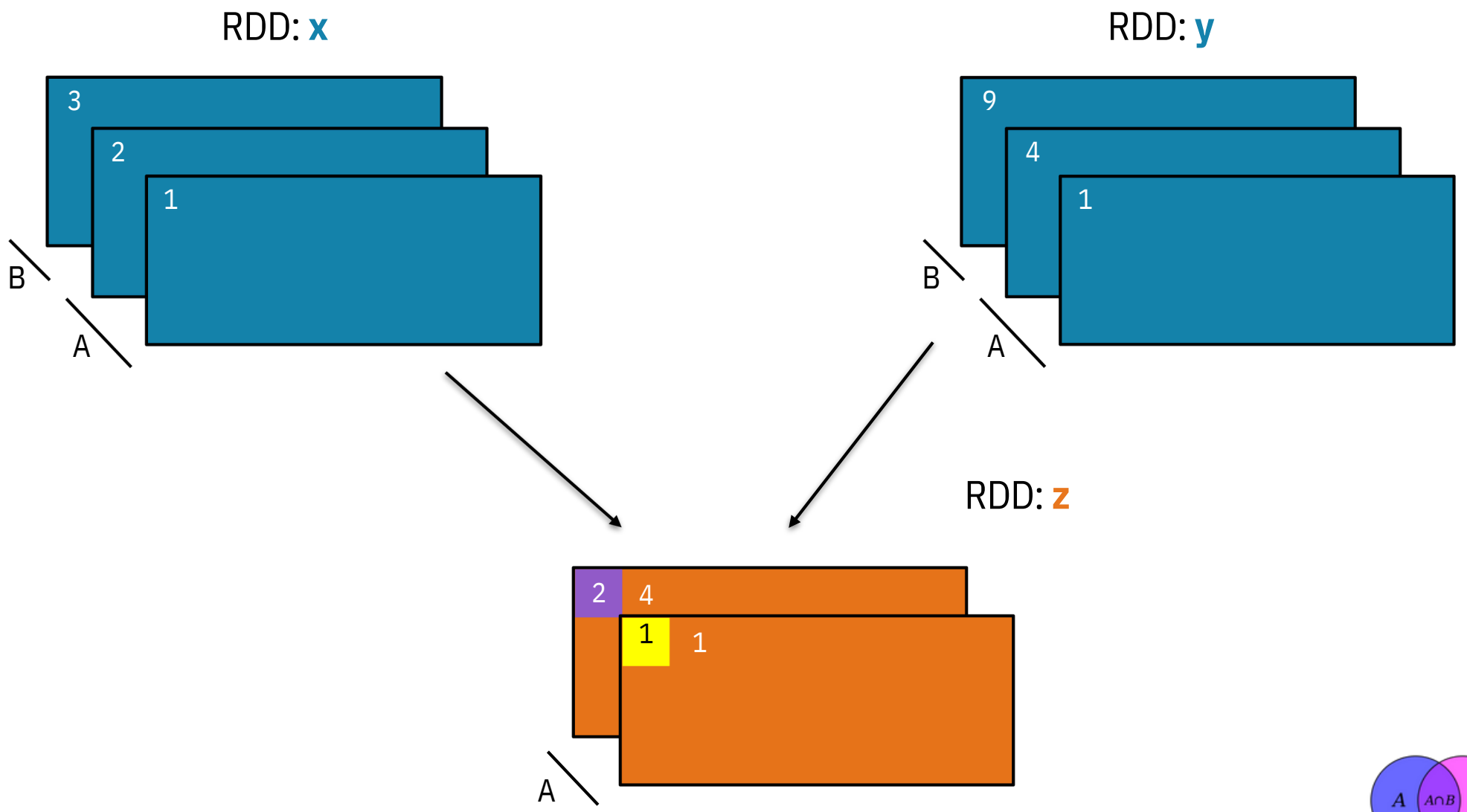


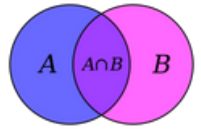


ZIP

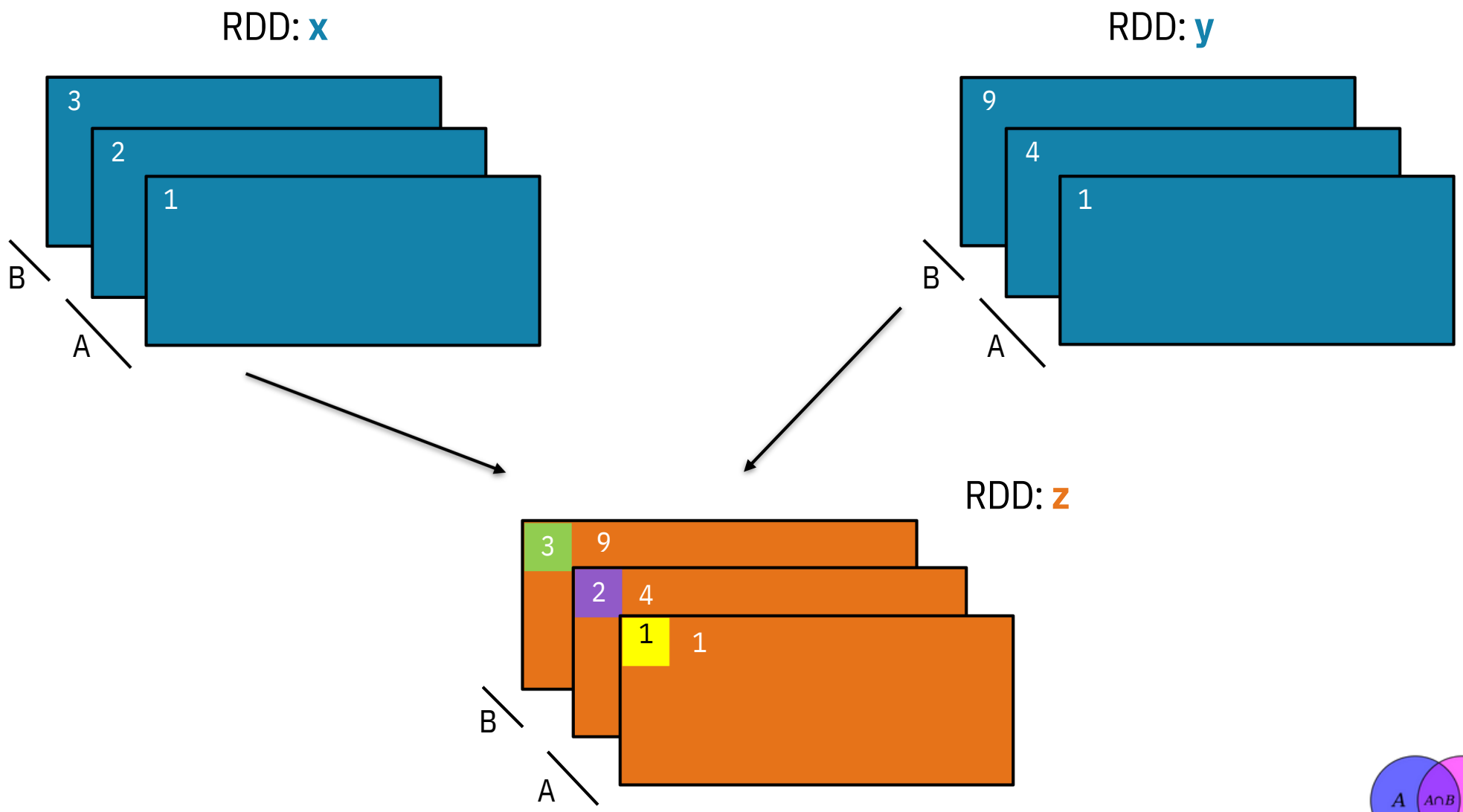


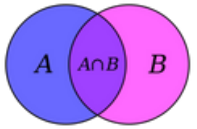
ZIP



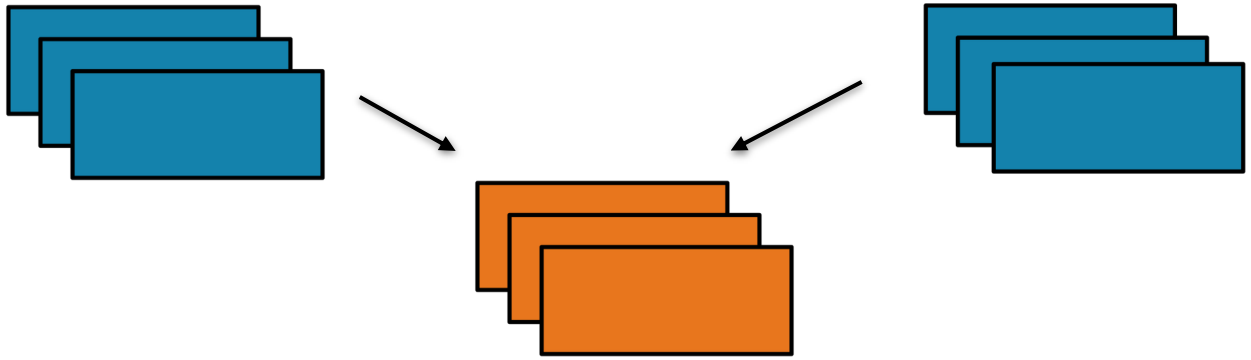


ZIP





ZIP



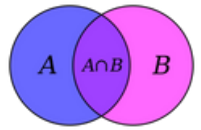
Return a new RDD containing pairs whose key is the item in the original RDD, and whose value is that item's corresponding element (same partition, same index) in a second RDD

```
zip(otherRDD)
```

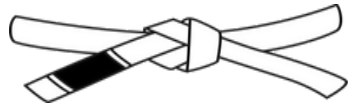
```
x = sc.parallelize([1, 2, 3])
y = x.map(lambda n:n*n)
z = x.zip(y)
```

```
val x = sc.parallelize(Array(1,2,3))
val y = x.map(n=>n*n)
val z = x.zip(y)

println(z.collect().mkString(", "))
```

```
x:  [1, 2, 3]

y:  [1, 4, 9]

z:  [(1, 1), (2, 4), (3, 9)]
```



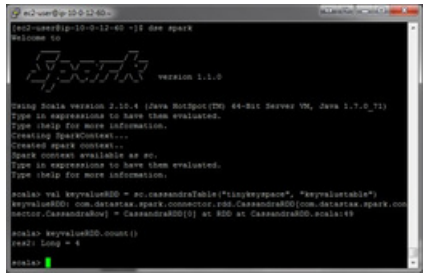
ACTIONS



Core Operations

databricks

vs



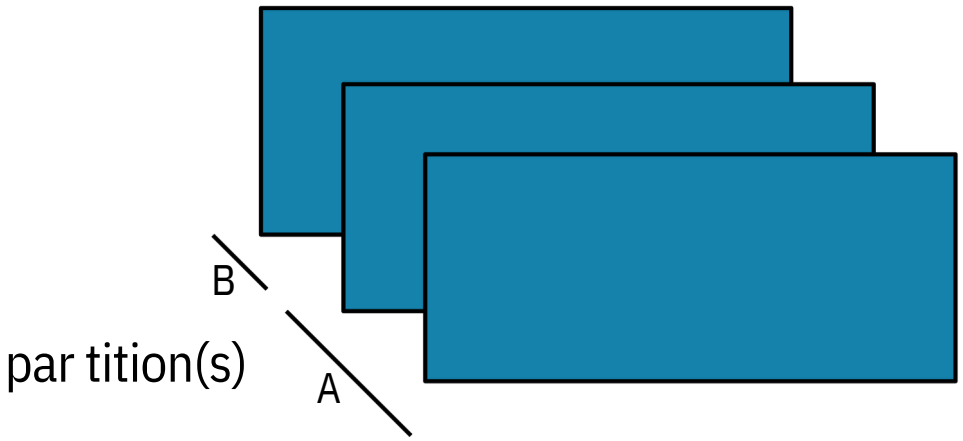
distributed

occurs across the cluster

driver

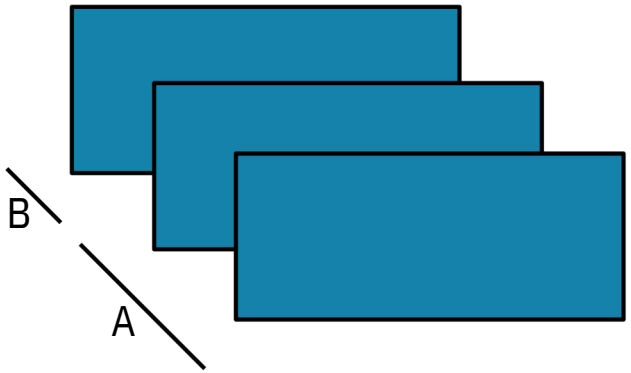
result must fit in driver JVM

GETNUMPARTITIONS



2

GETNUMPARTITIONS



`getNumPartitions()`

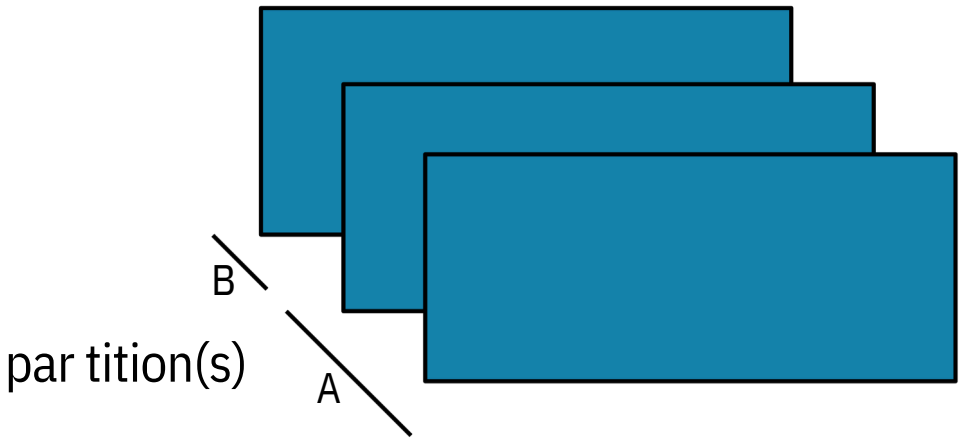
Return the number of partitions in RDD

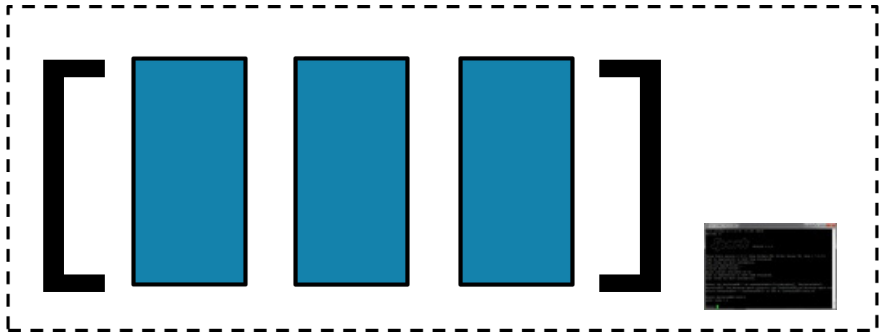
```python
x = sc.parallelize([1,2,3], 2)
y = x.getNumPartitions()

print(x.glom().collect())
print(y)
```

```scala
val x =ysc.parallelize(Array(1,2,3), 2)
val xO=tx.partitions.size
val      = x.glom().collect()
println(y)
```

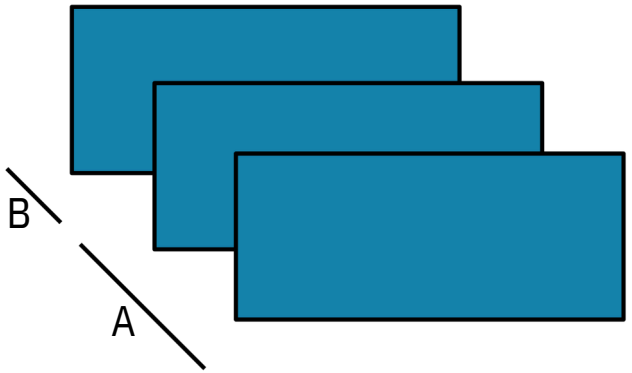
```
x: [[1], [2, 3]]

y: 2
```

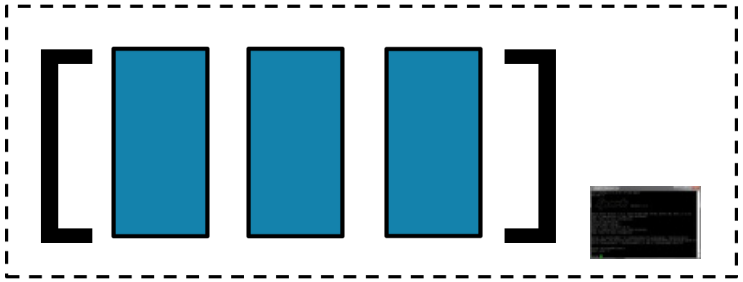
COLLECT





COLLECT





`collect()`

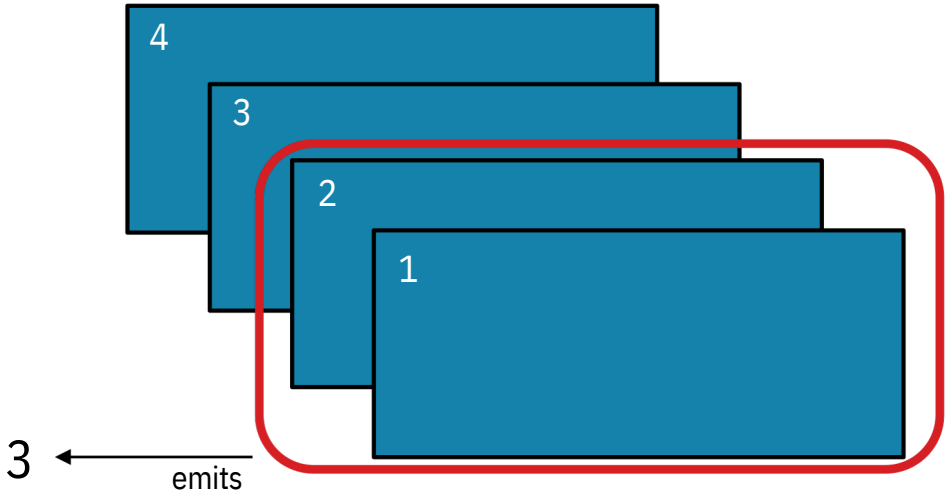
Return all items in the RDD to the driver in a single list

```python
x = sc.parallelize([1,2,3], 2)
y = x.collect()

print(x.glom().collect())
print(y)
```

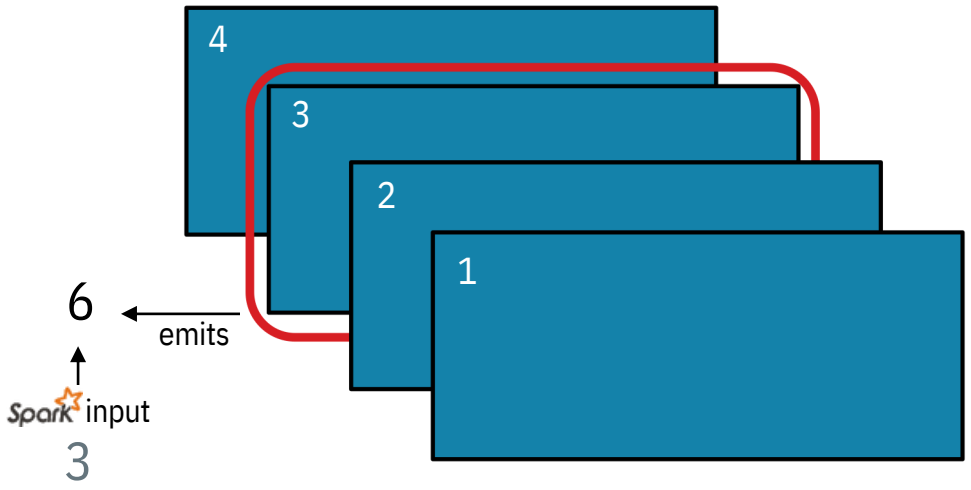
```scala
val x =ysc.parallelize(Array(1,2,3), 2)
val xOutx.collect()
val
         = x.glom().collect()
println(y)
```

```
x: [[1], [2, 3]]

y: [1, 2, 3]
```

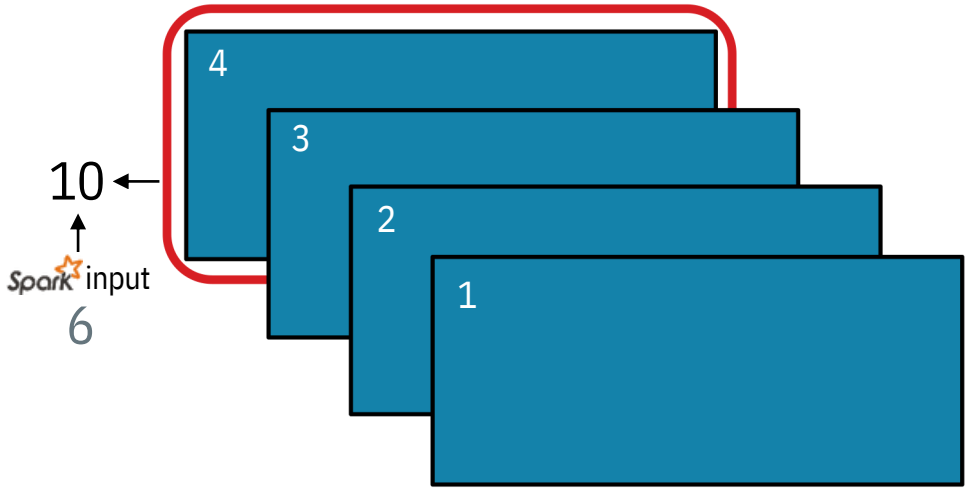
REDUCE



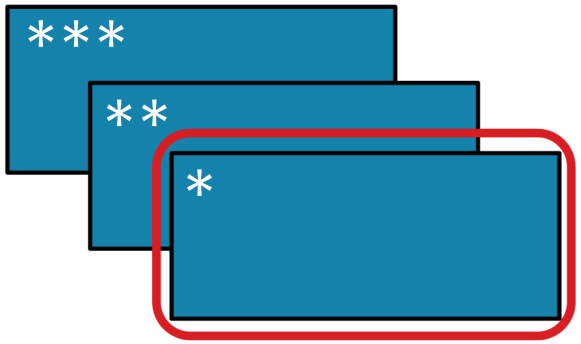
REDUCE



REDUCE



REDUCE



`reduce(f)`

Aggregate all the elements of the RDD by applying a user function pairwise to elements and partial results, and returns a result to the driver

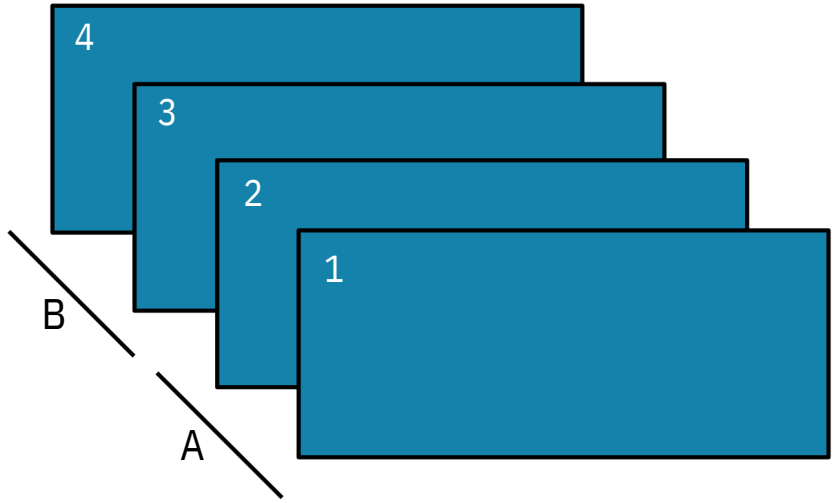
```python
x = sc.parallelize([1,2,3,4])
y = x.reduce(lambda a,b: a+b)

print(x.collect())
print(y)
```

```scala
val x = sc.parallelize(Array(1,2,3,4))
val y = x.reduce((a,b) => a+b)

println(x.collect.mkString(", "))
println(y)
```

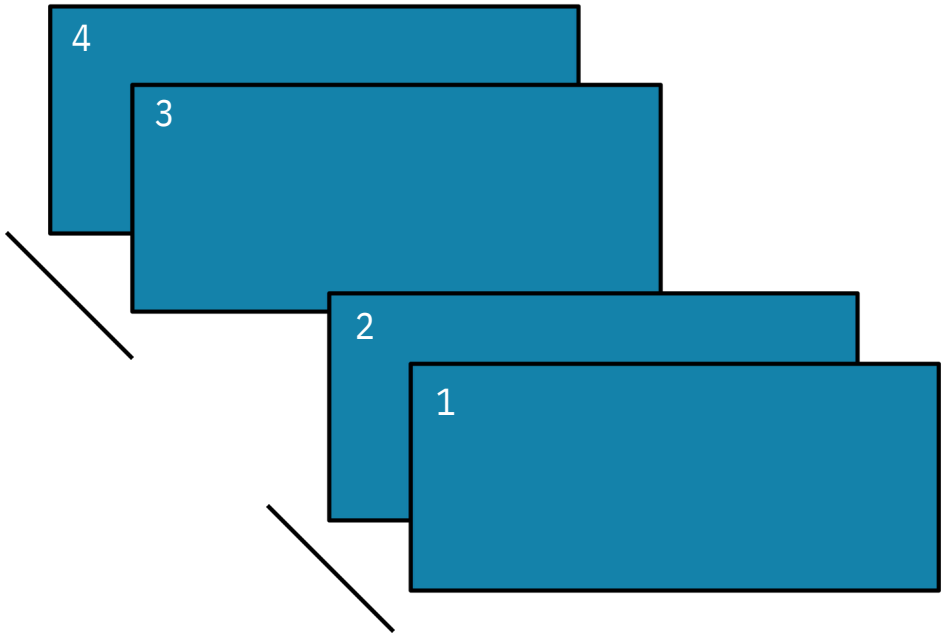
x: `[1, 2, 3, 4]`

y: `10`

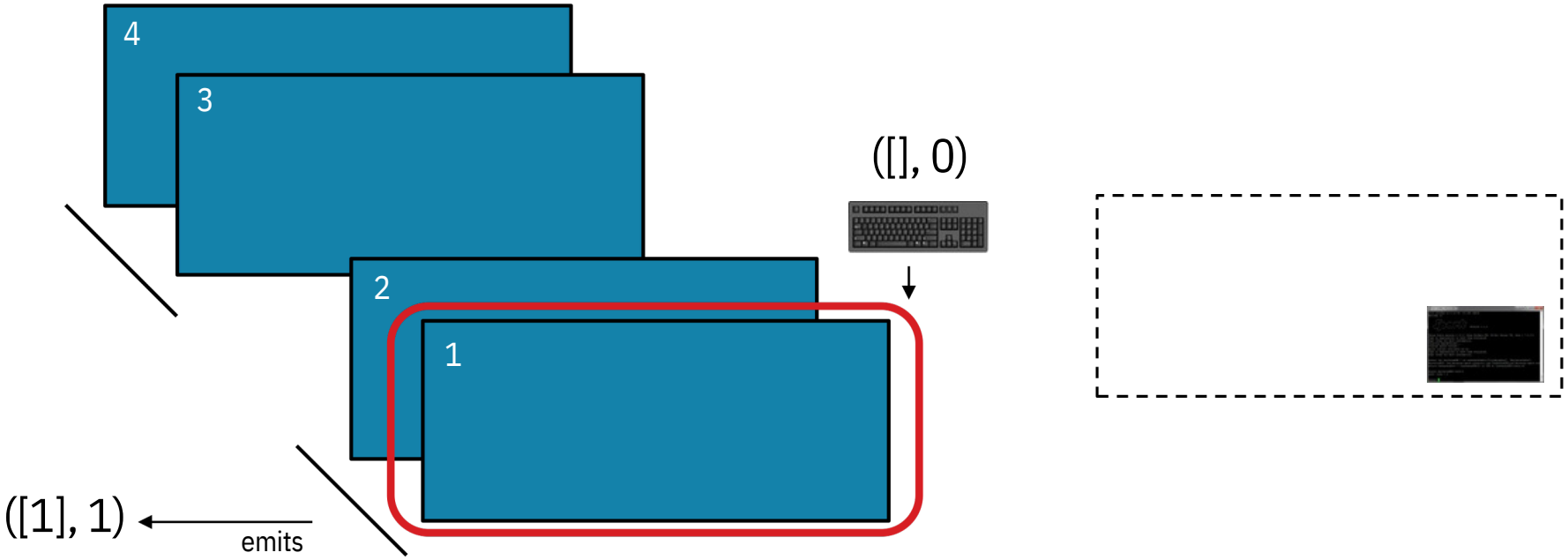
AGGREGATE



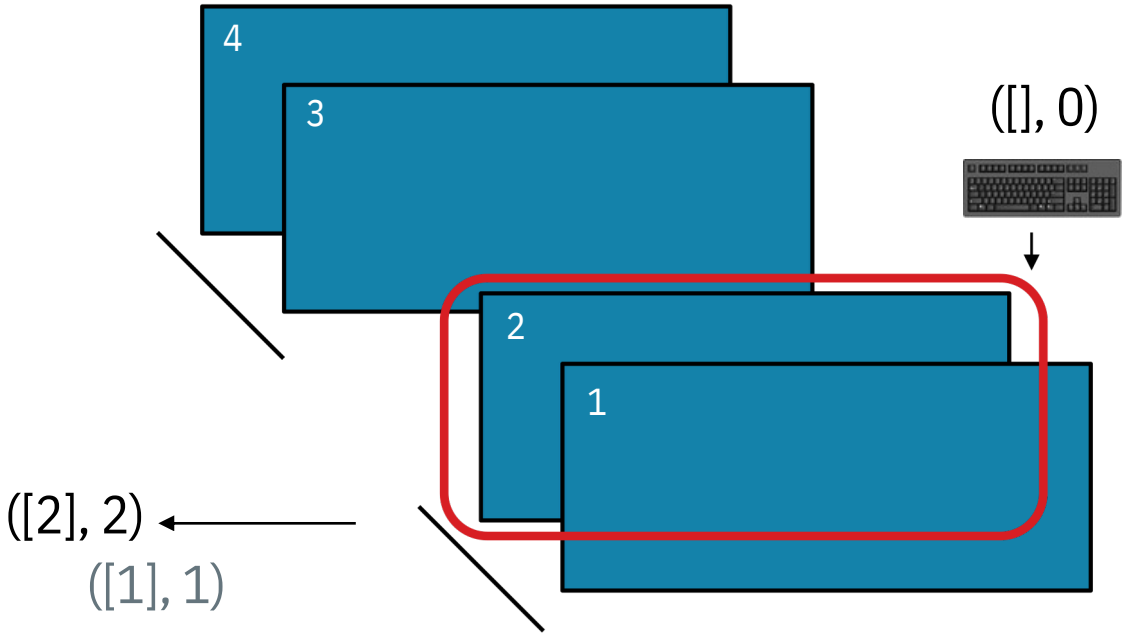
AGGREGATE



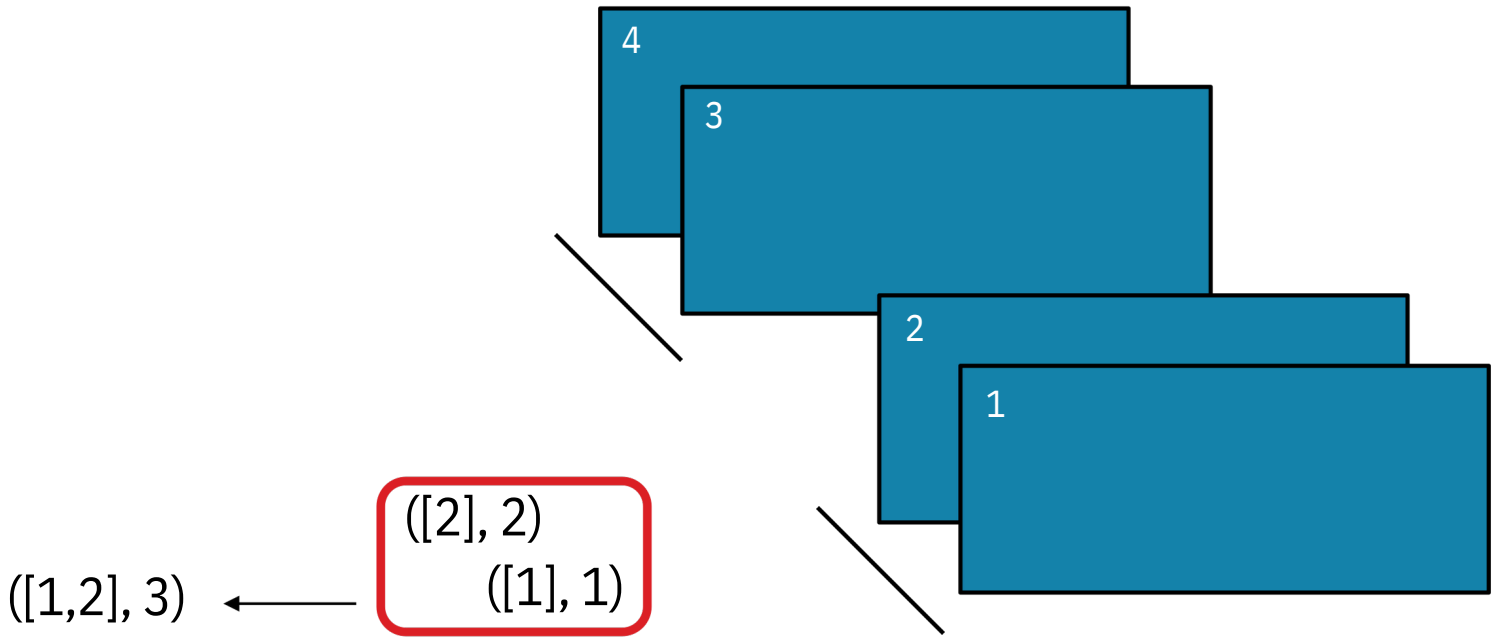
AGGREGATE



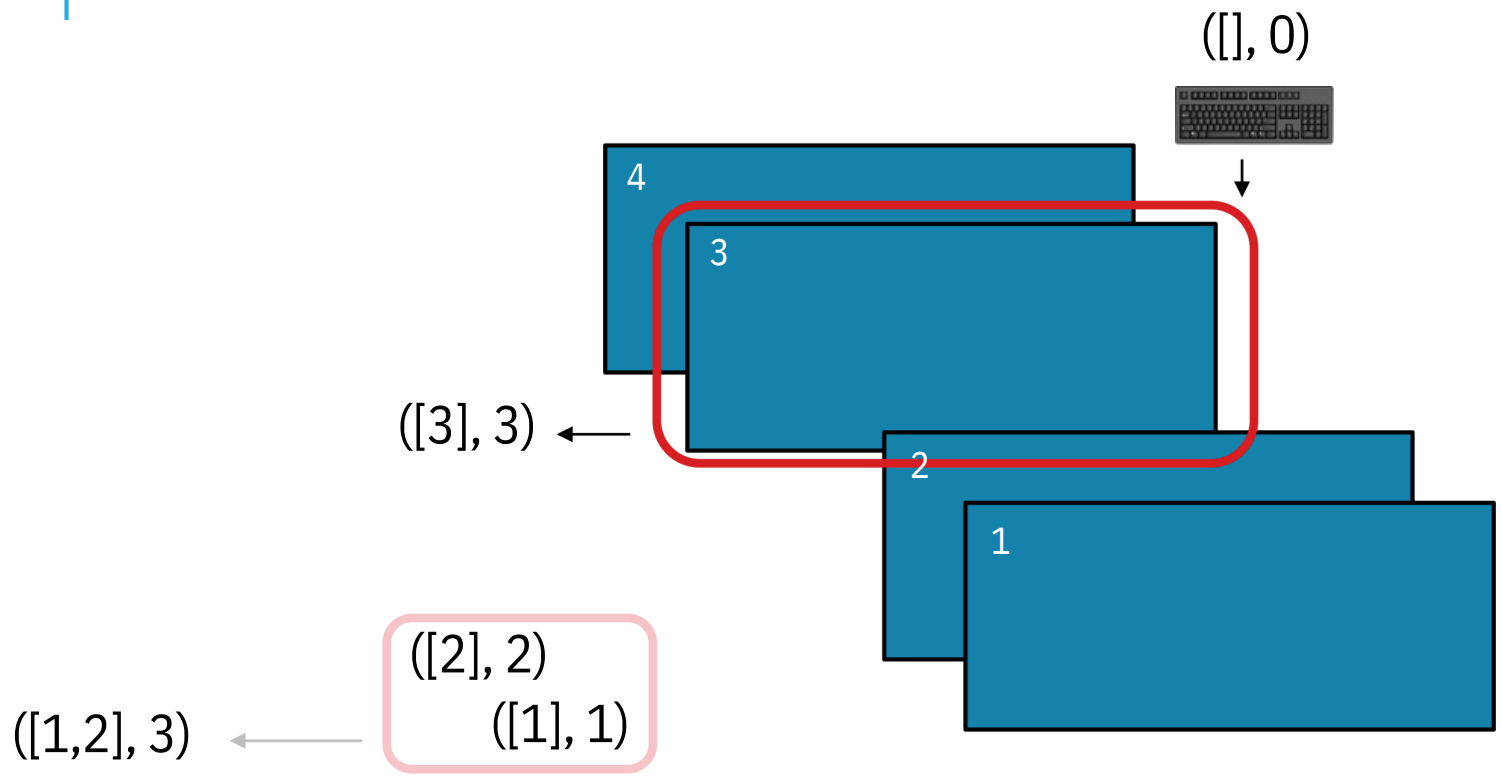
AGGREGATE



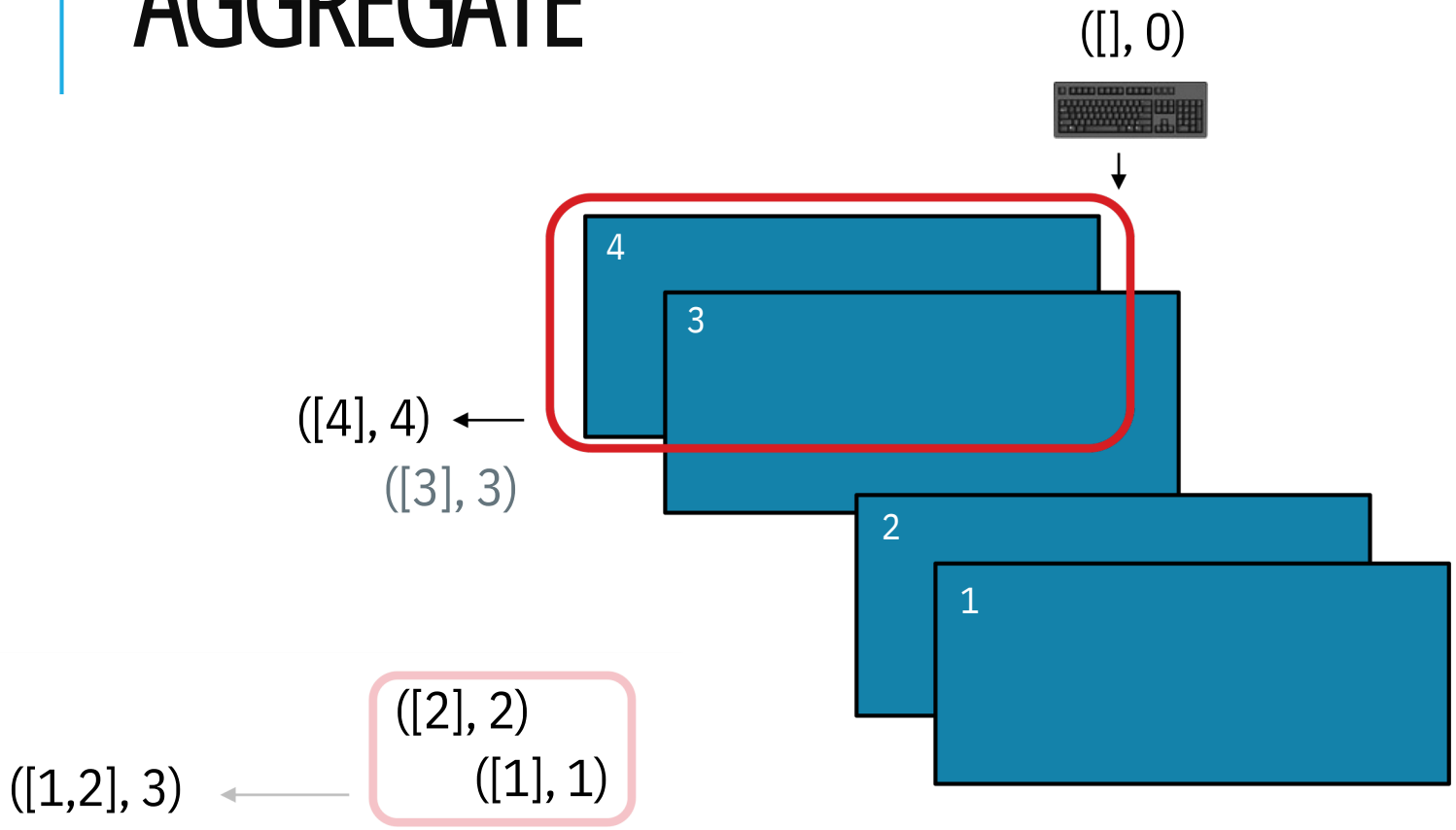
AGGREGATE



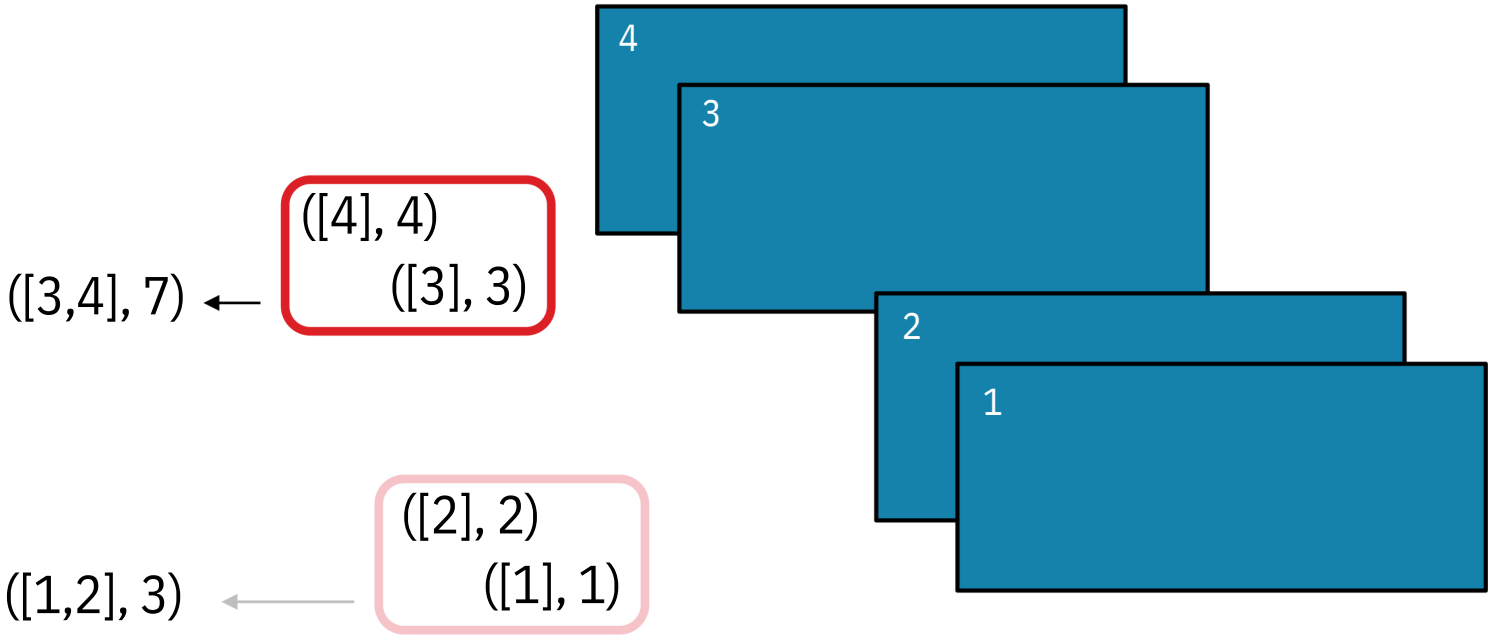
AGGREGATE



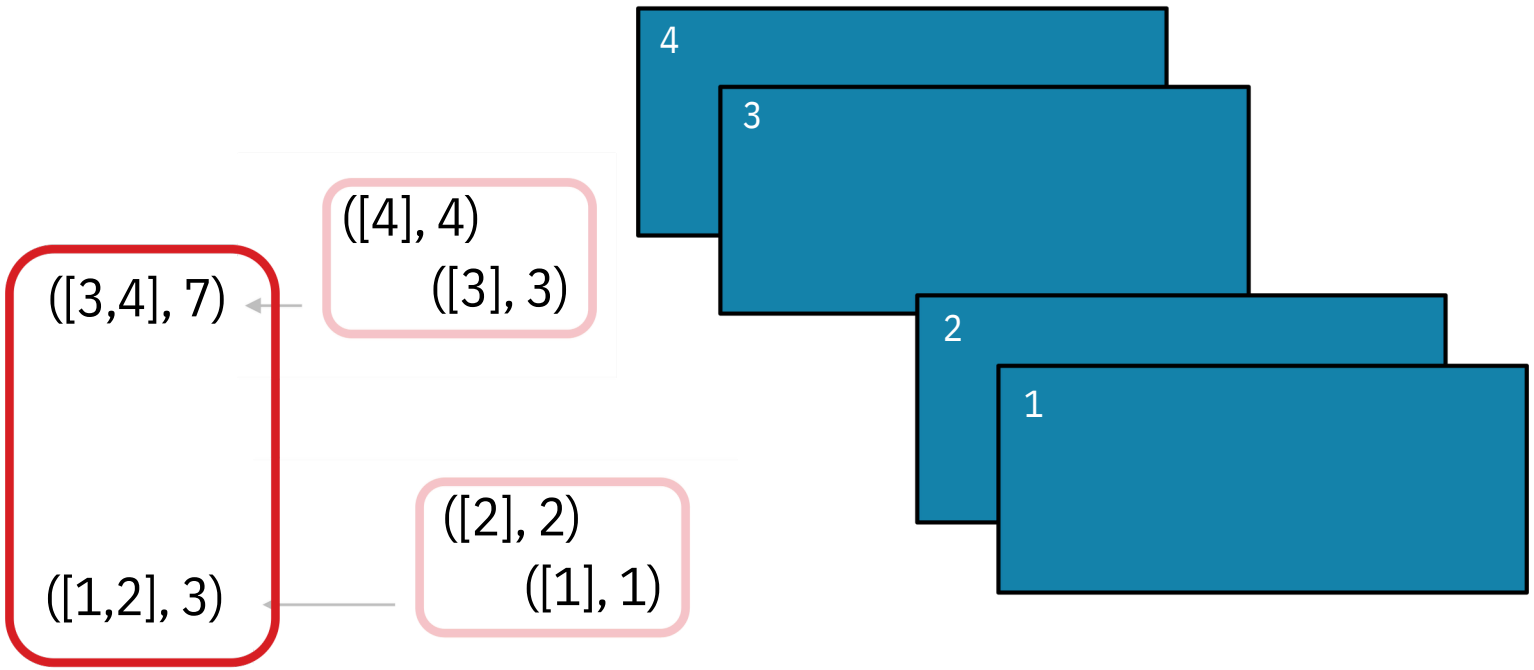
AGGREGATE



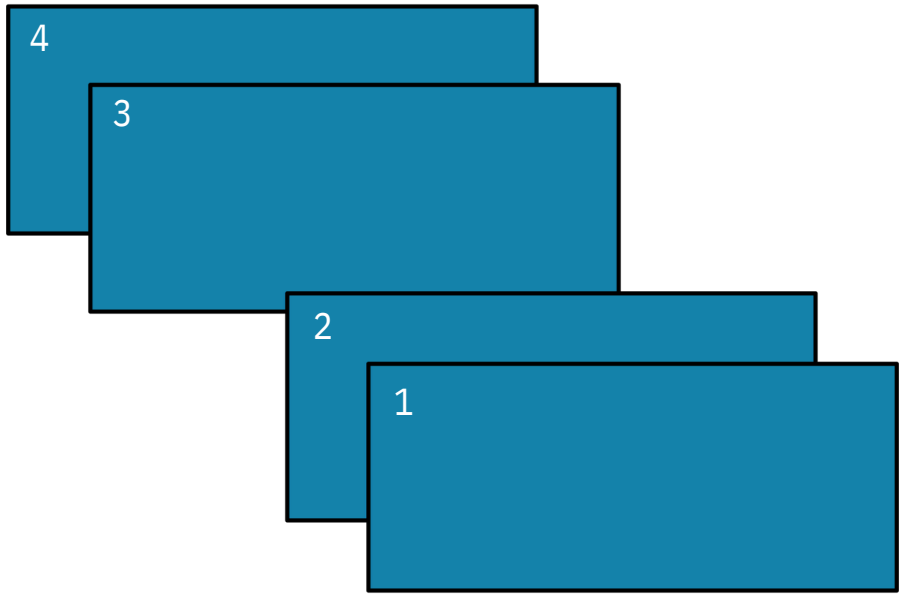
AGGREGATE



AGGREGATE



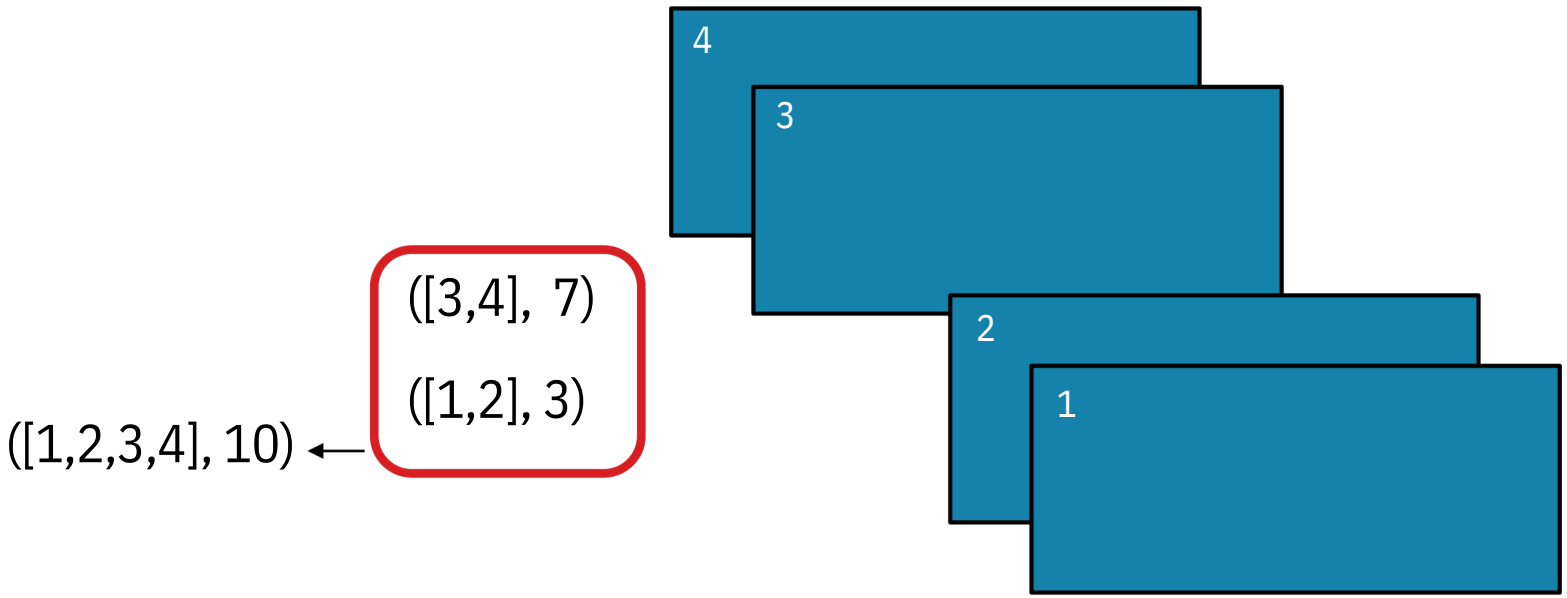
AGGREGATE



([3,4], 7)

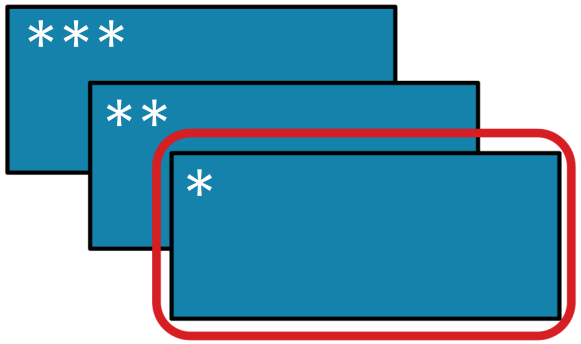
([1,2], 3)

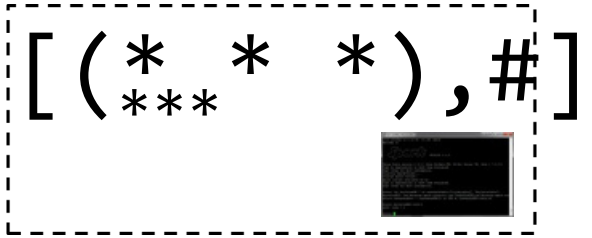
AGGREGATE



([1,2,3,4], 10)

AGGREGATE






```
aggregate(identity, seqOp, combOp)
```

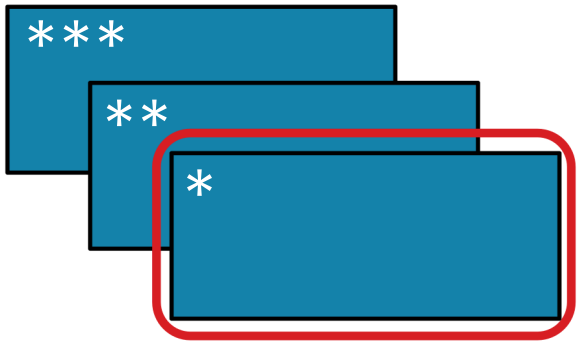
Aggregate all the elements of the RDD by:
-applying a user function to combine elements with user-supplied objects,
-then combining those user-defined results via a second user function,
-and finally returning a result to the driver.

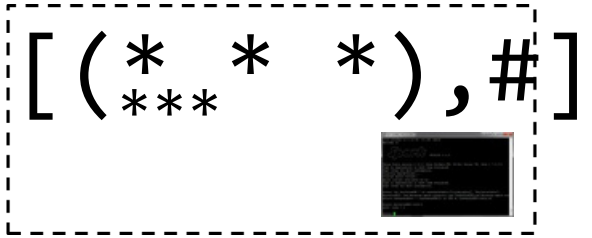
```
seqOp = lambda data, item: (data[0] + [item], data[1] + item)
combOp = lambda d1, d2: (d1[0] + d2[0], d1[1] + d2[1])

x = sc.parallelize([1,2,3,4])
y = x.aggregate(([], 0), seqOp, combOp)

print(y)
```

x: `[1, 2, 3, 4]`

y: `([1, 2, 3, 4], 10)`

AGGREGATE






```
aggregate(identity, seqOp, combOp)
```

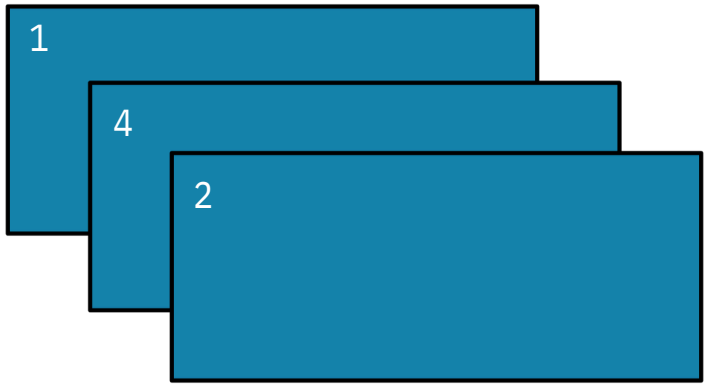
Aggregate all the elements of the RDD by:
-applying a user function to combine elements with user-supplied objects,
-then combining those user-defined results via a second user function,
-and finally returning a result to the driver.

```
def seqOp = (data:(Array[Int], Int), item:Int) =>
     (data._1 :+ item, data._2 + item)

def combOp = (d1:(Array[Int], Int), d2:(Array[Int], Int)) =>
     (d1._1.union(d2._1), d1._2 + d2._2)

val x = sc.parallelize(Array(1,2,3,4))
val y = x.aggregate((Array[Int](), 0))(seqOp, combOp)

println(y)
```

x: `[1, 2, 3, 4]`

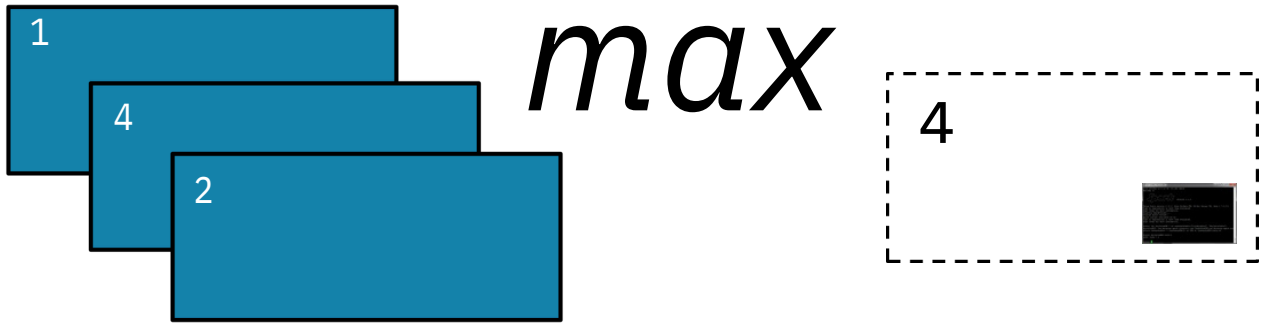
y: `(Array(3, 1, 2, 4),10)`

MAX



4

MAX



`max()`

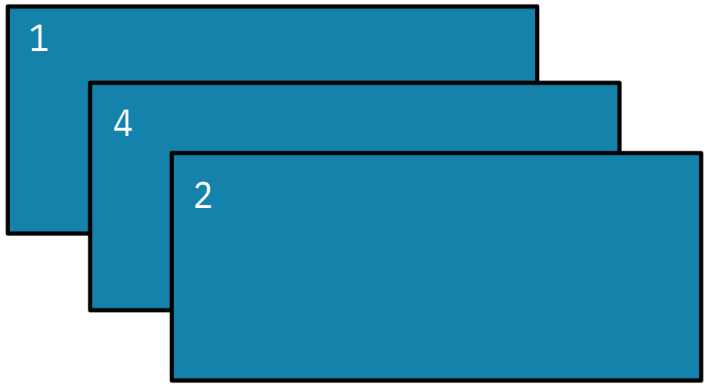
Return the maximum item in the RDD

```
x = sc.parallelize([2,4,1])
y = x.max()

print(x.collect())
print(y)
```

```
val x = sc.parallelize(Array(2,4,1))
val y = x.max

println(x.collect().mkString(", "))
println(y)
```

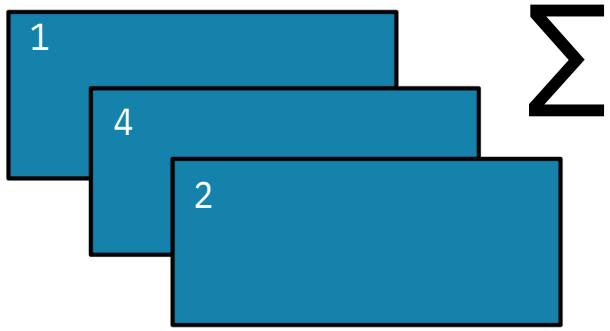
```
x: [2, 4, 1]

y: 4
```

SUM



7

SUM



`sum()`

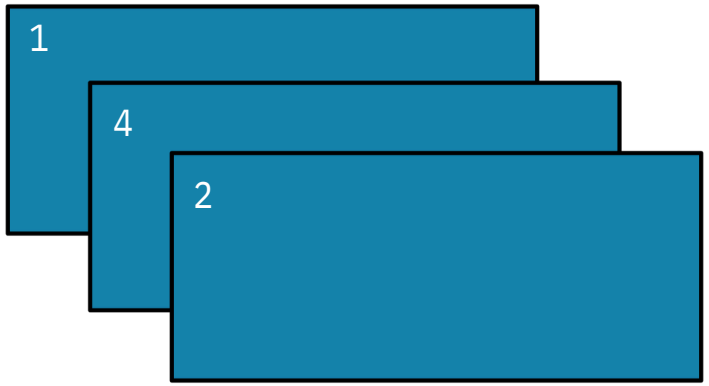
Return the sum of the items in the RDD

```python
x = sc.parallelize([2,4,1])
y = x.sum()

print(x.collect())
print(y)
```

```scala
val x = sc.parallelize(Array(2,4,1))
val y = x.sum

println(x.collect().mkString(", "))
println(y)
```

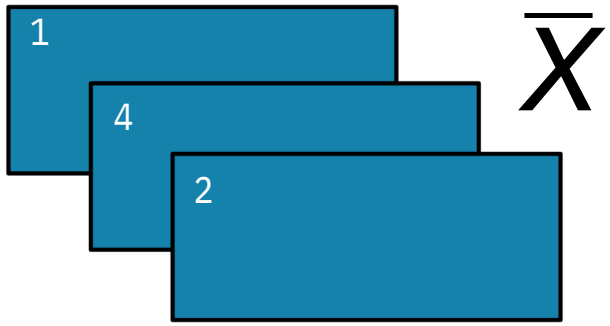
```
x: [2, 4, 1]

y: 7
```

MEAN



2.33333333

MEAN




```
2.3333333
```

`mean()`

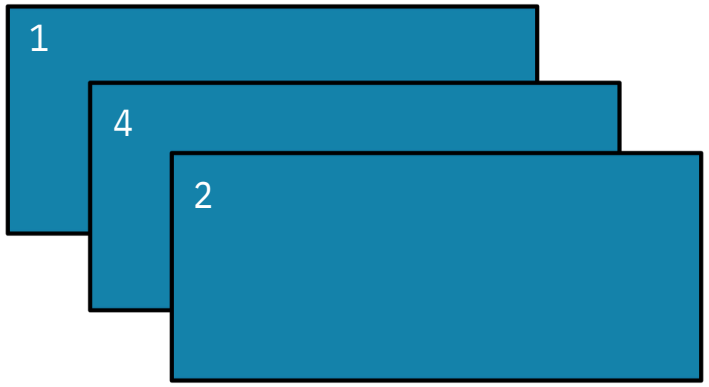
Return the mean of the items in the RDD

```python
x = sc.parallelize([2,4,1])
y = x.mean()

print(x.collect())
print(y)
```

```
x: [2, 4, 1]

y: 2.3333333
```

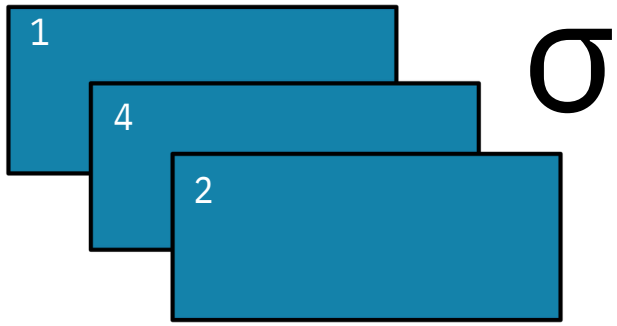
```scala
val x = sc.parallelize(Array(2,4,1))
val y = x.mean

println(x.collect().mkString(", "))
println(y)
```

STDEV



1.2472191

STDEV



1.2472191

stdev()

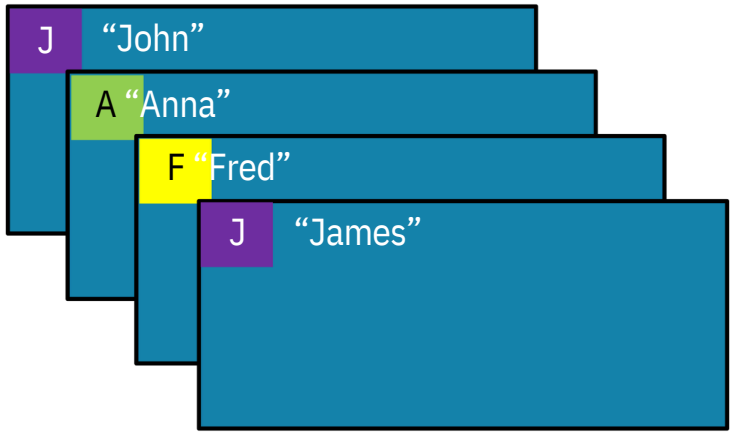
Return the standard deviation of the items in the RDD

```python
x = sc.parallelize([2,4,1])
y = x.stdev()

print(x.collect())
print(y)
```

```scala
val x = sc.parallelize(Array(2,4,1))
val y = x.stdev

println(x.collect().mkString(", "))
println(y)
```

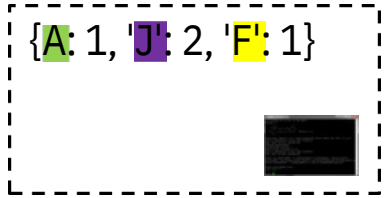
```
x: [2, 4, 1]

y: 1.2472191
```

COUNTBYKEY



{'A': 1, 'J': 2, 'F': 1}

COUNTBYKEY



countByKey()

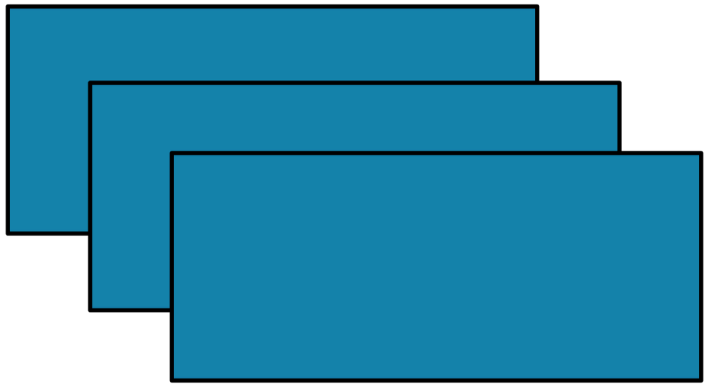
Return a map of keys and counts of their occurrences in the RDD

```
x = sc.parallelize([('J', 'James'), ('F','Fred'),
                    ('A','Anna'), ('J','John')])

y = x.countByKey()
print(y)
```

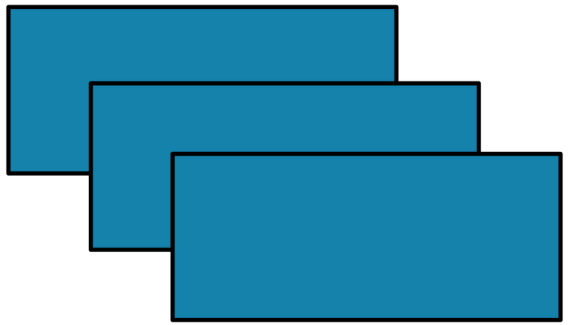
```
val x = sc.parallelize(Array(('J',"James"),('F',"Fred"),
                             ('A',"Anna"),('J',"John")))

val y = x.countByKey()
println(y)
```

```
x: [('J', 'James'), ('F','Fred'),
    ('A','Anna'), ('J','John')]

y: {'A': 1, 'J': 2, 'F': 1}
```

SAVEASTEXTFILE



SAVEASTEXTFILE



`saveAsTextFile(path, compressionCodecClass=None)`

Save the RDD to the filesystemindicated in the path

```python
dbutils.fs.rm("/temp/demo", True)
x = sc.parallelize([2,4,1])
x.saveAsTextFile("/temp/demo")

y = sc.textFile("/temp/demo")
print(y.collect())
```

```scala
dbutils.fs.rm("/temp/demo", true)
val x = sc.parallelize(Array(2,4,1))
x.saveAsTextFile("/temp/demo")

val y = sc.textFile("/temp/demo")
println(y.collect().mkString(", "))
```

```
x: [2, 4, 1]

y: [u'2', u'4', u'1']
```

databricks™