# Spark Actions and Transformations

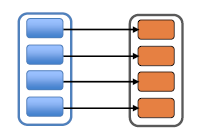
**Spark Transformation** is a function that produces new RDD from the existing RDDs. It takes RDD as input and produces one or more RDD as output.

Each time it creates new RDD when we apply any transformation. The so input RDDs, cannot be changed since RDD are immutable in nature.

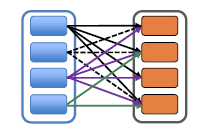
Transformations are lazy in nature i.e., they get execute when we call an action. They are not executed immediately.

There are two types of transformations:

* Narrow transformation – In Narrow transformation, all the elements that are required to compute the records in single partition live in the single partition of parent RDD. A limited subset of partition is used to calculate the result. Narrow transformations are the result of map(), filter().



* Wide transformation – In wide transformation, all the elements that are required to compute the records in the single partition may live in many partitions of parent RDD. The partition may live in many partitions of parent RDD. Wide transformations are the result of groupbyKey and reducebyKey.



**Actions** are RDD operations that produce non-RDD values. They materialize a value in a Spark program. In other words, a RDD operation that returns a value of any type but RDD[T] is an action.

They trigger execution of RDD transformations to return values. Simply put, an action evaluates the RDD lineage graph.

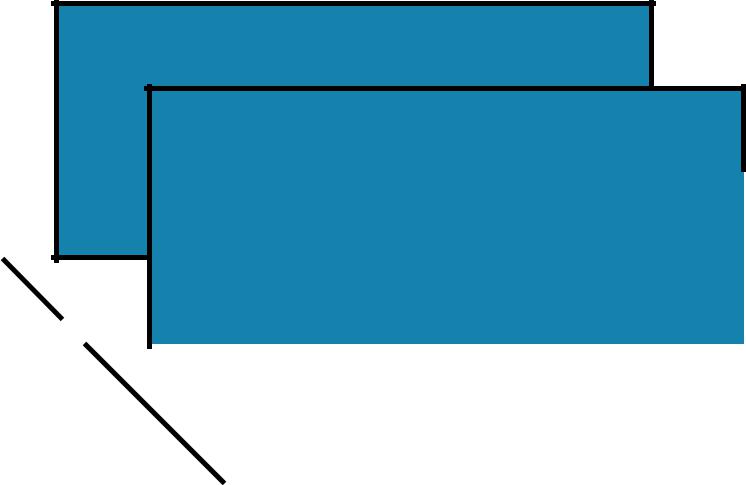
Below link will have the latest Scala API’s available in Spark:

<http://spark.apache.org/docs/latest/api/scala/index.html#org.apache.spark.rdd.RDD>

RDD RDD Elements



Legend key



original item

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| B |  |  |  |  |  |  |
|  |  |  |  | transformed |  |
| partition(s) |  |  |  |  |  |
|  |  |  |  |  |
| A | |  |  | type |  |
|  |  |  |
|  |  |  |  |  |  |  |



object on driver

user input



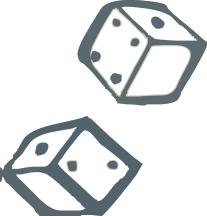
user functions

emitted value

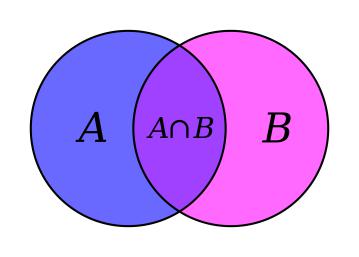
input



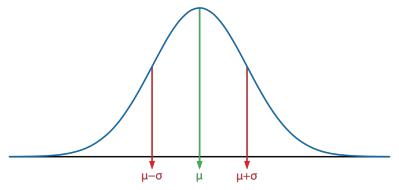
Legend



Randomized operation



Set Theory / Relational operation



Numeric calculation



TRANSFORMATIONS

Operations

=

+



ACTIONS

|  |
| --- |
| ACTIONS TRANSFORMATIONS |



= easy  = medium



Essential Core & Intermediate Spark Operations

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
|  | **General** | **Math / Statistical** | | **Set Theory / Relational** | | **Data Structure / I/O** | |
| • | map | • | sample | • | union | • | keyBy |
| • | filter | • | randomSplit | • | intersection | • | zipWithIndex |
| • | flatMap |  |  | • | subtract | • | zipWithUniqueID |
| • | mapPartitions |  |  | • | distinct | • | zipPartitions |
| • | mapPartitionsWithIndex |  |  | • | cartesian | • | coalesce |
| • | groupBy |  |  | • | zip | • | repartition |
| • | sortBy |  |  |  |  | • | repartitionAndSortWithinPartitions |
|  |  |  |  |  |  | • | pipe |



|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| • | reduce | • | count | • takeOrdered | • | saveAsTextFile |
| • | collect | • | takeSample |  | • | saveAsSequenceFile |
| • | aggregate | • | max |  | • | saveAsObjectFile |
| • | fold | • | min |  | • | saveAsHadoopDataset |
| • | first | • | sum |  | • | saveAsHadoopFile |
| • | take | • | histogram |  | • | saveAsNewAPIHadoopDataset |
| • | forEach | • | mean |  | • | saveAsNewAPIHadoopFile |
| • | top | • | variance |  |  |  |
| • | treeAggregate | • | stdev |  |  |  |
| • | treeReduce | • | sampleVariance |  |  |  |
| • | forEachPartition | • | countApprox |  |  |  |
| • | collectAsMap | • | countApproxDistinct |  |  |  |



|  |
| --- |
| ACTIONS TRANSFORMATIONS |

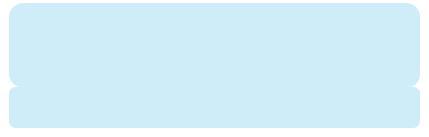


= easy  = medium



Essential Core & Intermediate PairRDD Operations

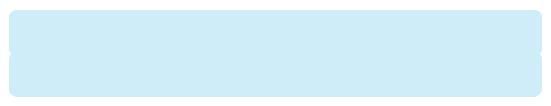
|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
|  | **General** | **Math / Statistical** | **Set Theory / Relational** | | **Data Structure** |
| • | flatMapValues | • sampleByKey | • | cogroup (=groupWith) | • partitionBy |
| • | groupByKey |  | • | join |  |
| • | reduceByKey |  | • | subtractByKey |  |
| • | reduceByKeyLocally |  | • | fullOuterJoin |  |
| • | foldByKey |  | • | leftOuterJoin |  |
| • | aggregateByKey |  | • | rightOuterJoin |  |



* sortByKey
* combineByKey



|  |  |  |  |
| --- | --- | --- | --- |
| • | keys | • | countByKey |
| • | values | • | countByValue |
|  |  | • | countByValueApprox |
|  |  | • | countApproxDistinctByKey |
|  |  | • | countApproxDistinctByKey |
|  |  | • | countByKeyApprox |
|  |  | • | sampleByKeyExact |

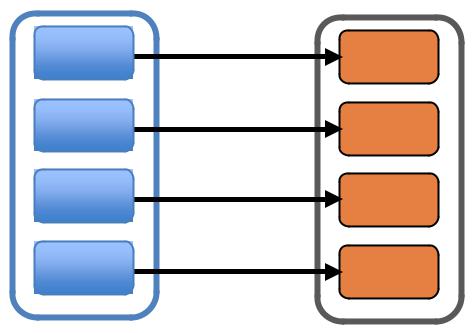




vs

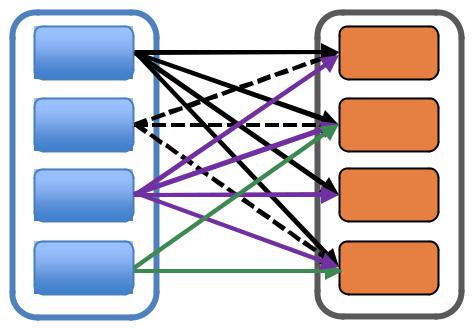
narrow

*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*





“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.”

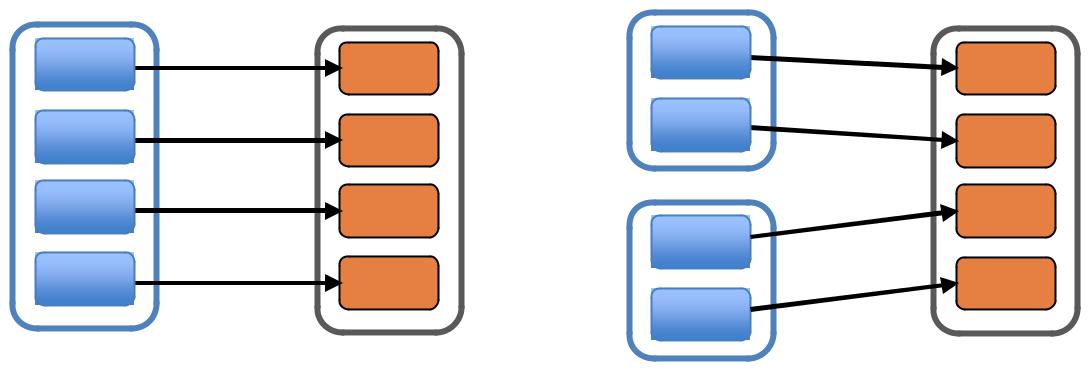
“We found it both sufficient and useful to classify dependencies into two types:

* 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.”

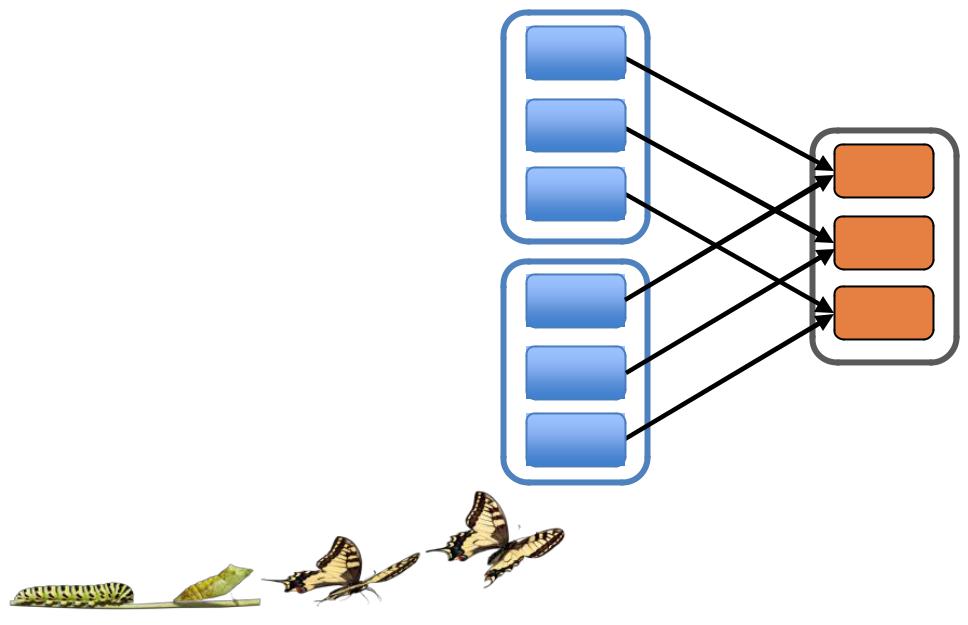


narrow

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



map, filter union

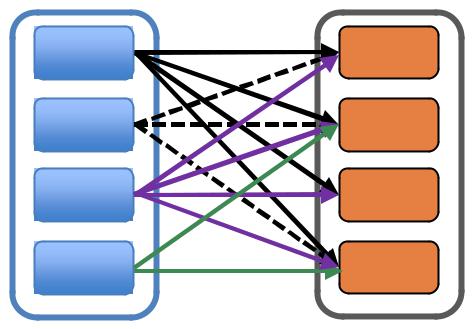


join w/ inputs

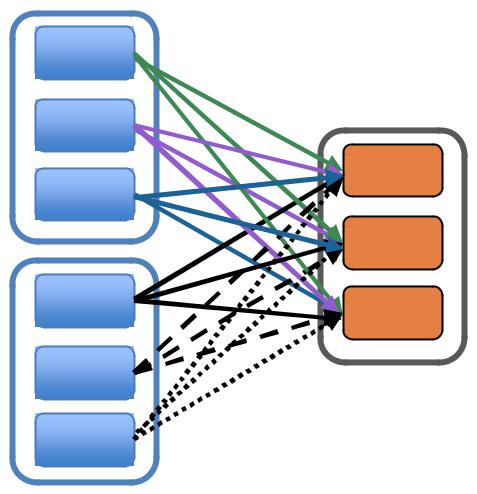
co-partitioned

wide

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



groupByKey



join w/ inputs not

co-partitioned

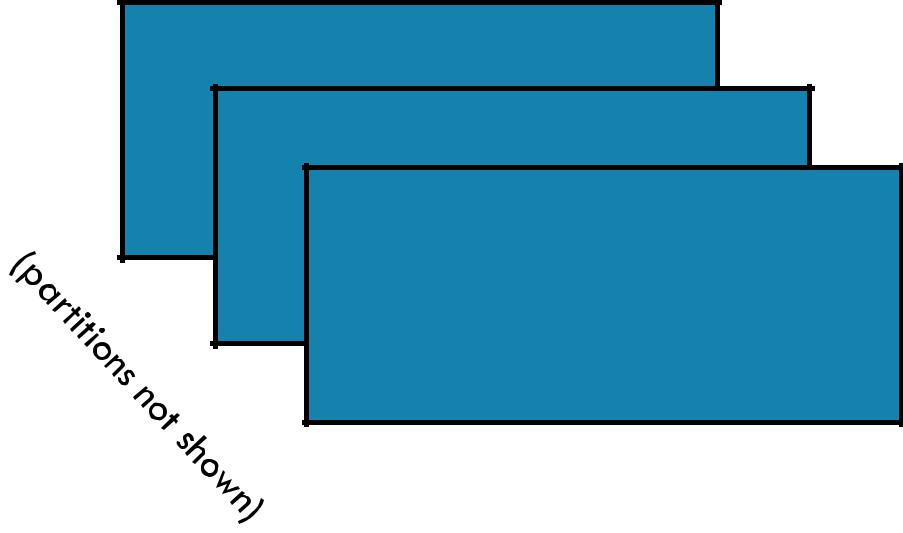
|  |  |  |
| --- | --- | --- |
| Transformations | Core Operations |  |



MAP

RDD: **x**

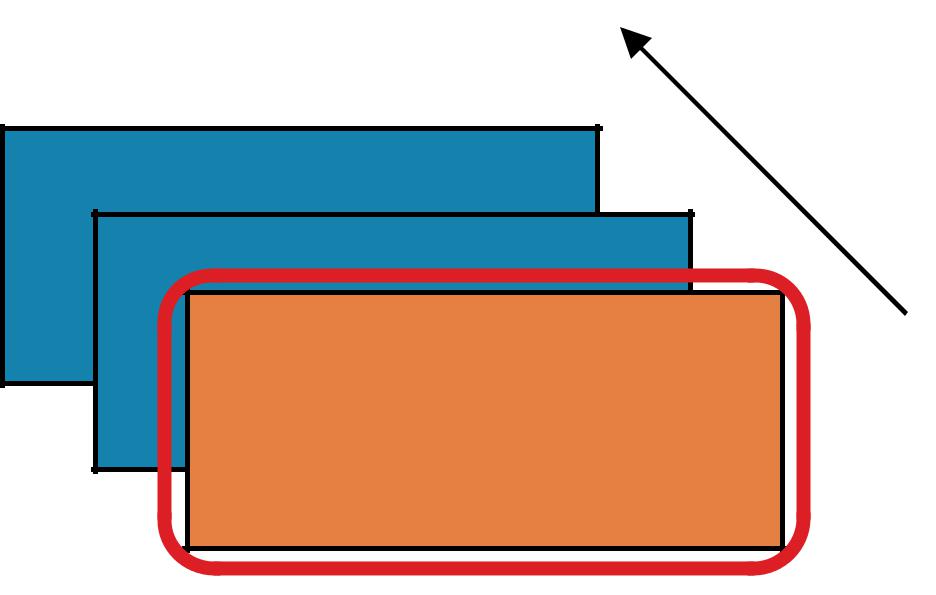
3 items in RDD





MAP

RDD: **x** RDD: **y**



User function

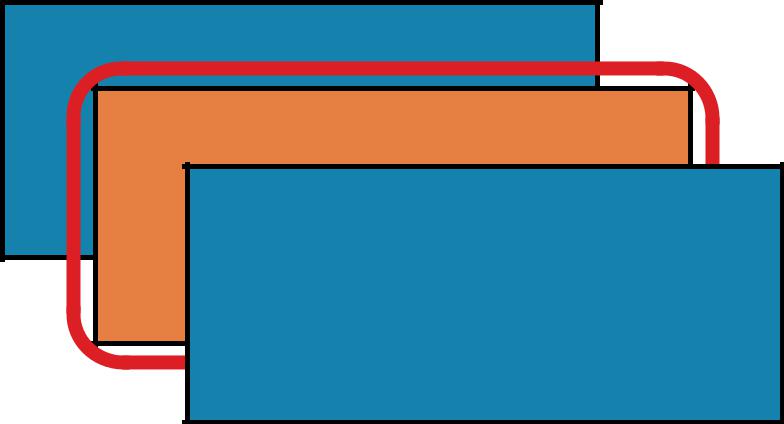
applied item by item





MAP

RDD: **x** RDD: **y**





MAP

RDD: **x** RDD: **y**



MAP

After map() has been applied…

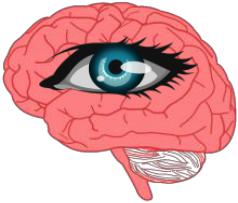


RDD: **x**

RDD: **y**

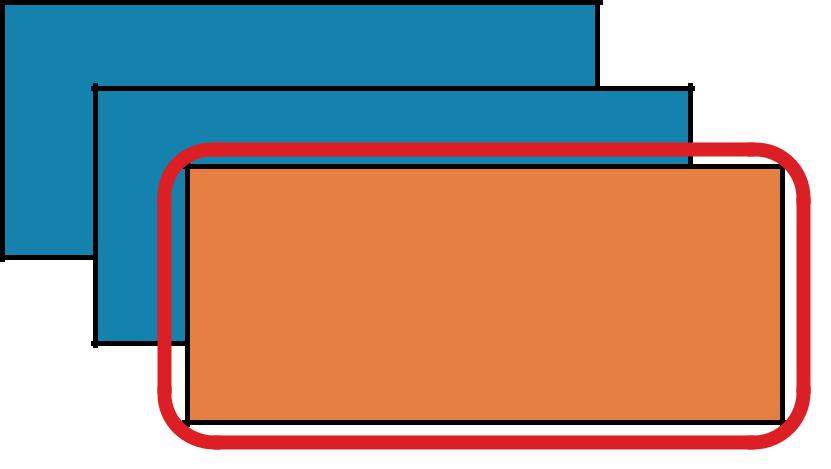
before after





MAP

RDD: **x**

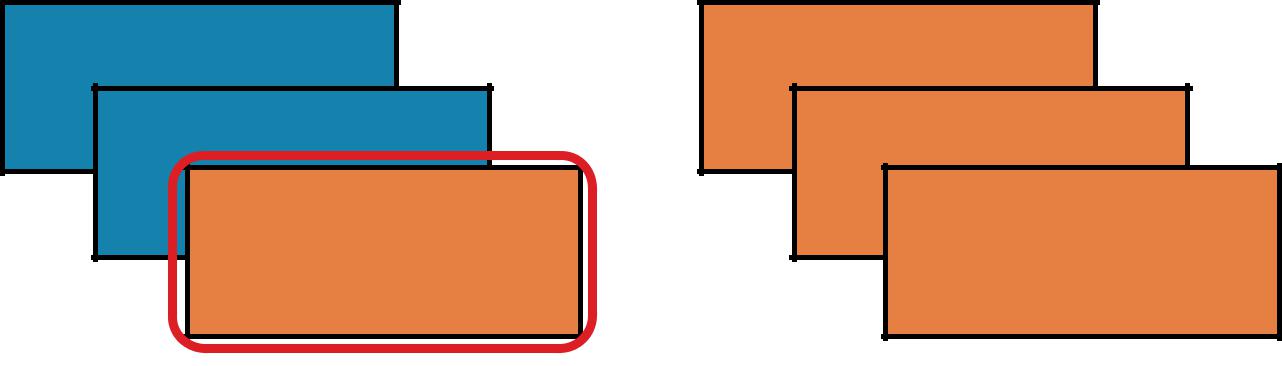


RDD: **y**

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



RDD: **x** RDD: **y**



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"])



1. = **x**.map(lambda z: (**z**, 1)) print(**x**.collect()) print(**y**.collect())



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

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

val **x** = sc.parallelize(Array("b", "a", "c"))



val **y** = **x**.map(z => (z,1))

println(**x**.collect().mkString(", "))

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

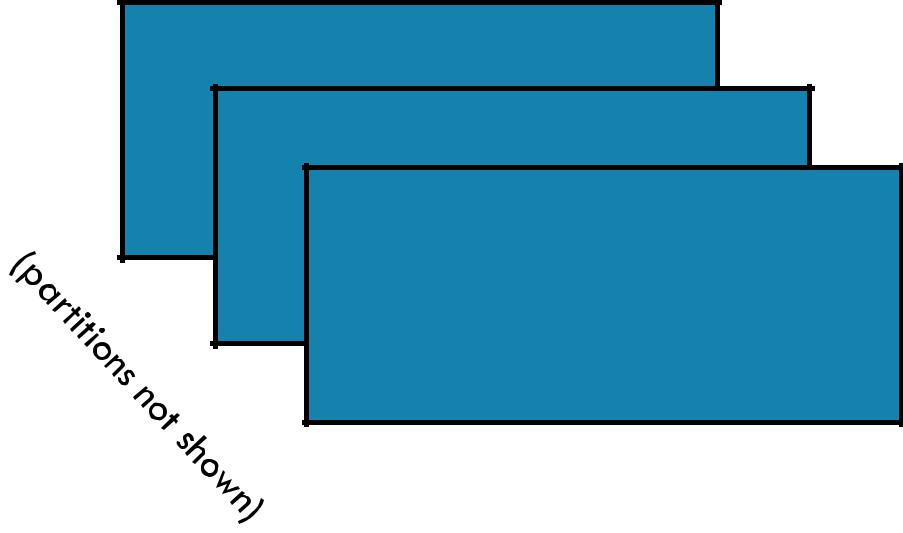




FILTER

RDD: **x**

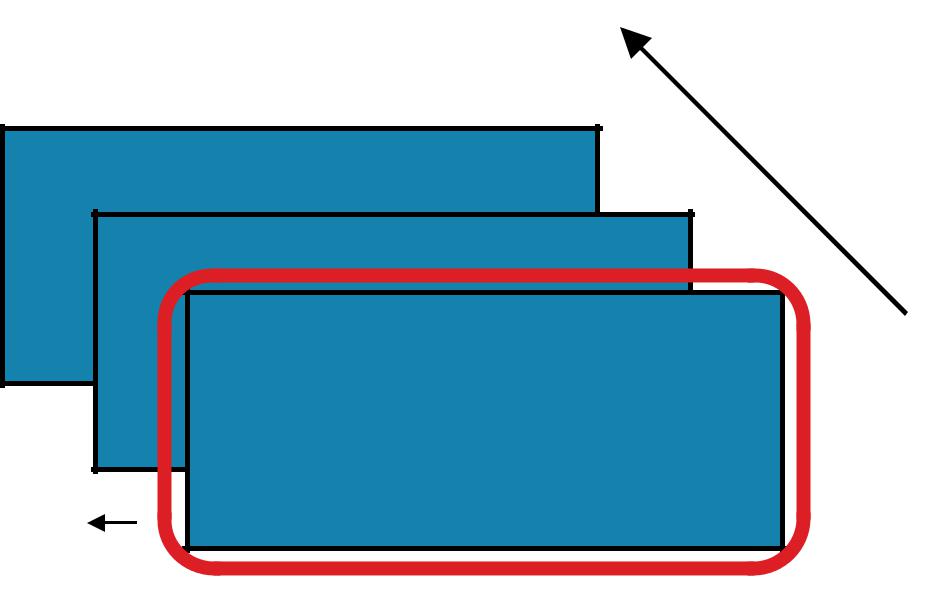
3 items in RDD





FILTER

RDD: **x** RDD: **y**



Apply user function:

keep item if function

returns true

True

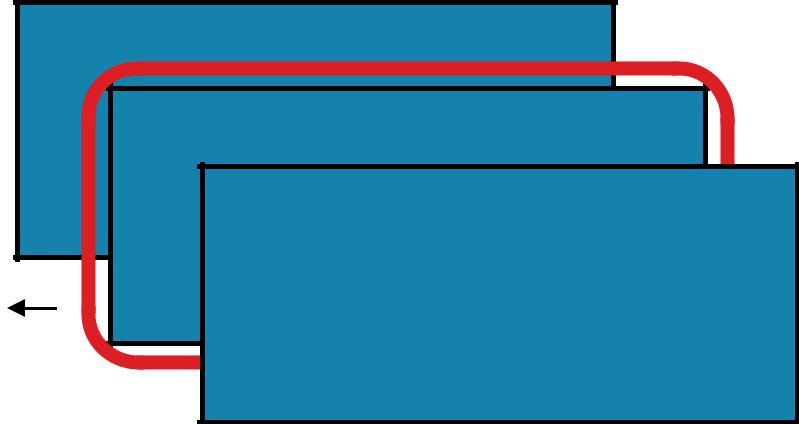
emits





FILTER

RDD: **x** RDD: **y**



False

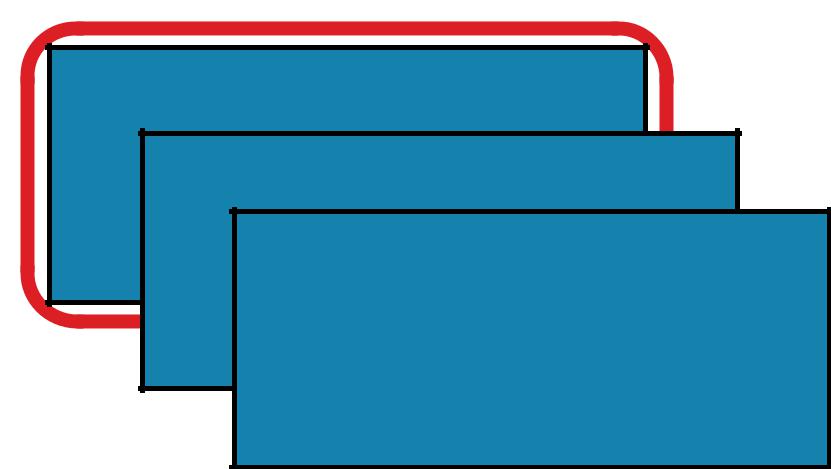
emits





FILTER

RDD: **x**



RDD: **y**

True



emits



FILTER

RDD: **x**

After filter() has been applied…

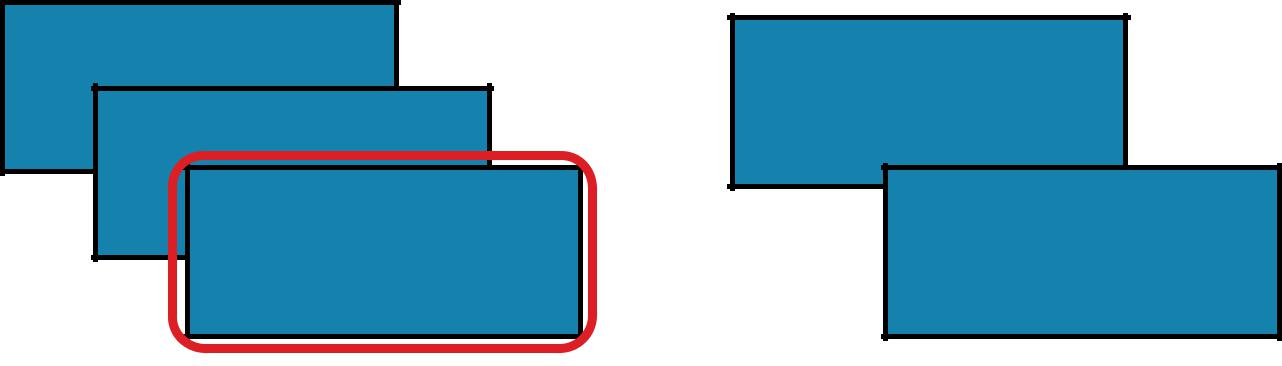


RDD: **y**

before after



RDD: **x** RDD: **y**



FILTER

**filter(*f*)**

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

**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]

val **x** = sc.parallelize(Array(1,2,3))



val **y** = **x**.filter(n => n%2 == 1)

println(**x**.collect().mkString(", "))

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

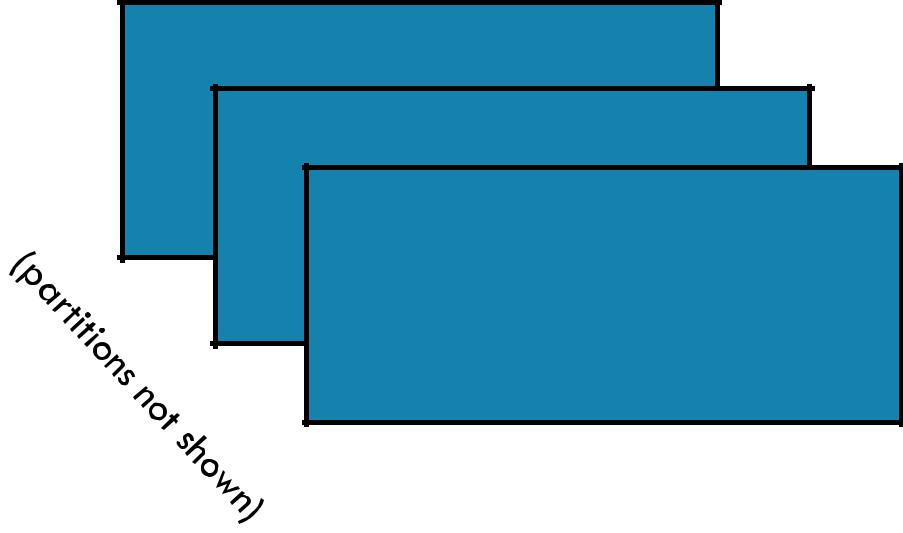




FLATMAP

RDD: **x**

3 items in RDD





FLATMAP

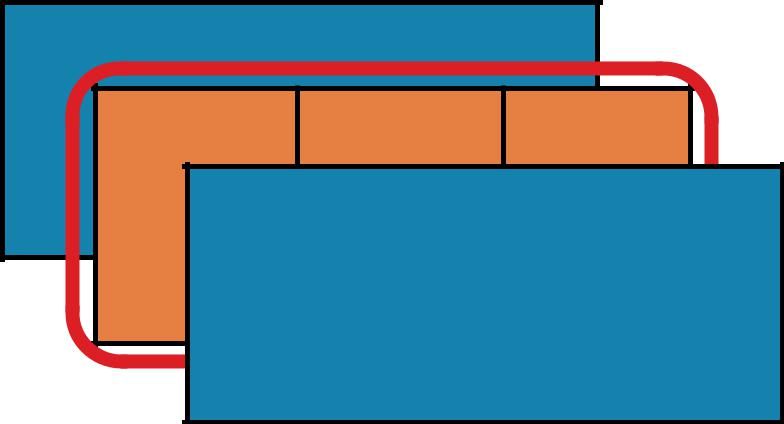
RDD: **x** RDD: **y**





FLATMAP

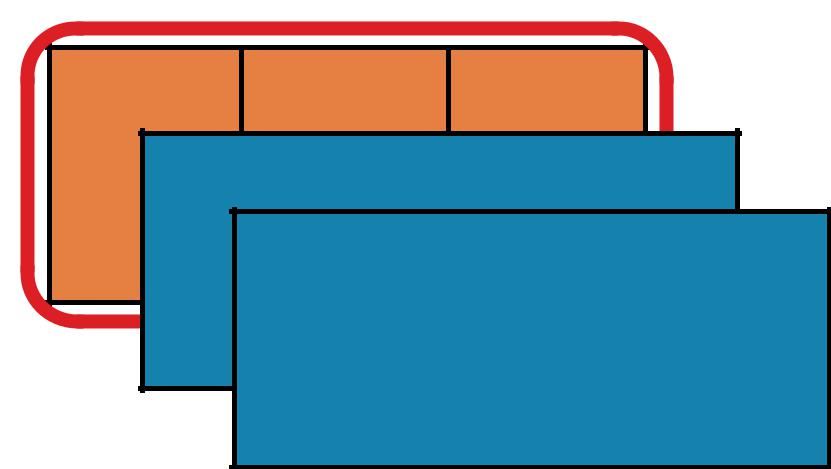
RDD: **x** RDD: **y**





FLATMAP

RDD: **x** RDD: **y**



FLATMAP

RDD: **x**

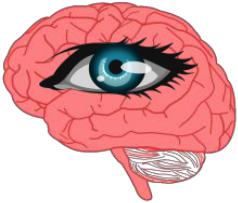
After flatmap() has been applied…



RDD: **y**

before after





FLATMAP

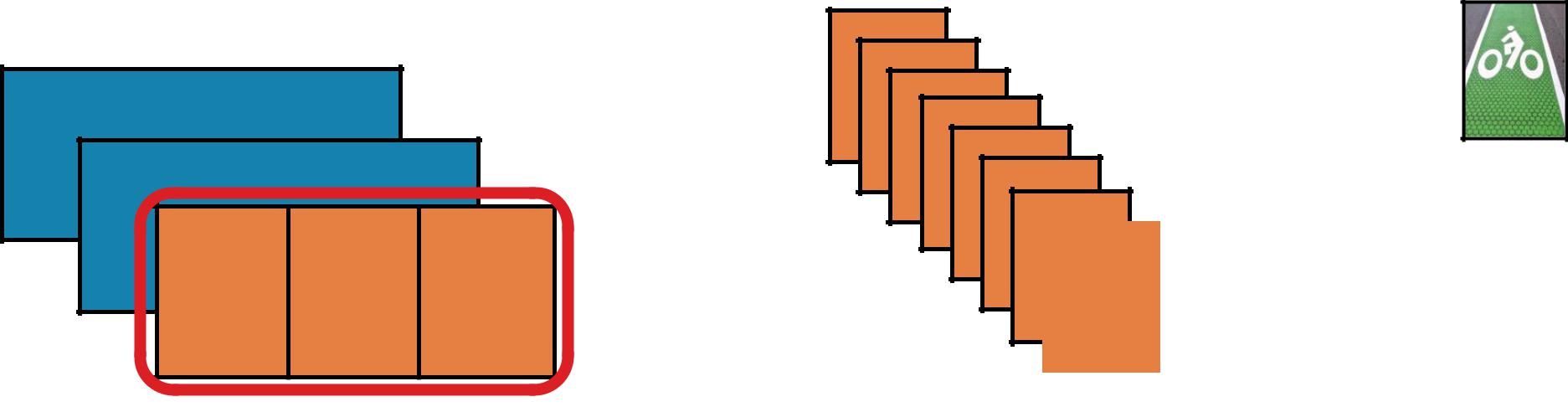
RDD: **x** RDD: **y**



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



|  |  |  |
| --- | --- | --- |
| RDD: **x** | RDD: **y** |  |
|  |  |



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])



1. = **x**.flatMap(lambda **x**: (**x**, **x**\*100, 42)) print(**x**.collect()) print(**y**.collect())

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

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

val **x** = sc.parallelize(Array(1,2,3))



val **y** = **x**.flatMap(n => Array(n, n\*100, 42))

println(**x**.collect().mkString(", "))

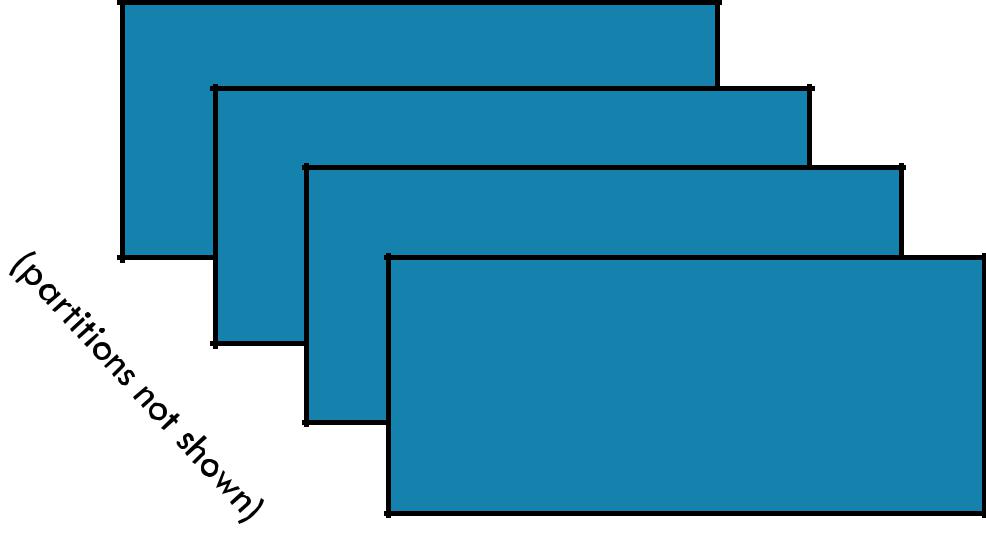
println(**y**.collect().mkString(", "))





GROUPBY

RDD: **x**



James

Anna

Fred

John

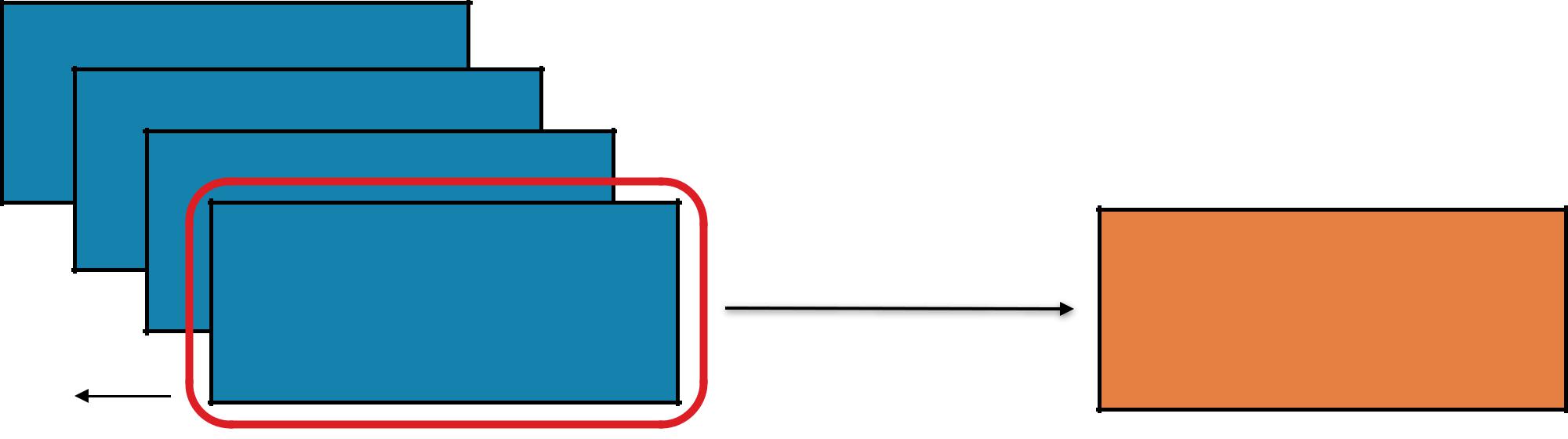


4 items in RDD



GROUPBY

RDD: **x** RDD: **y**



James

Anna

Fred

|  |  |  |
| --- | --- | --- |
| John | J | [ “John” ] |
|  |  |  |

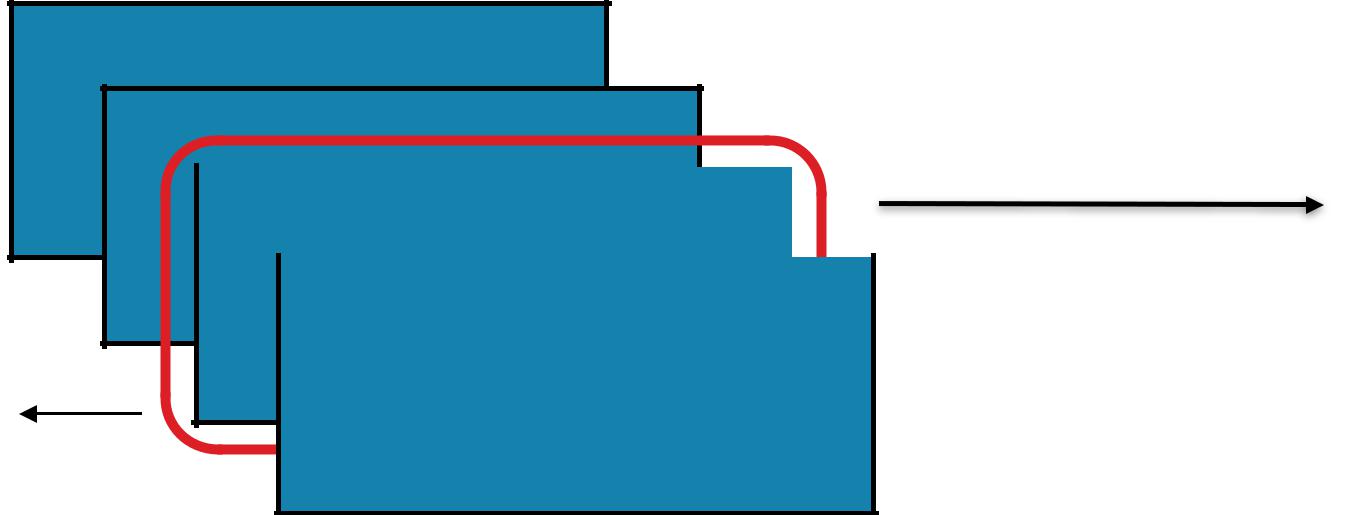
‘J’ emits





GROUPBY

RDD: **x** RDD: **y**



James

Anna

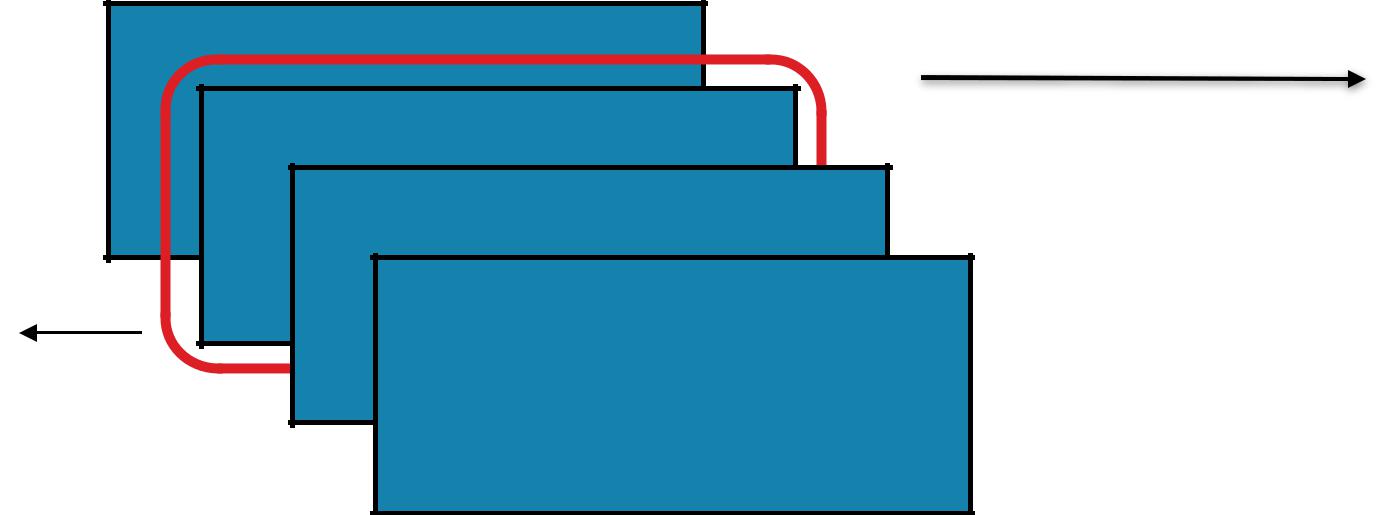
|  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  |  |  |  |  | F | | |  | [ “Fred” ] | |  |  |
| Fred | |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |  |  |  |
|  |  | |  |  |  |  |  |  |  |  |  |  |
|  | John | | | |  |  |  | J |  | [ “John” ] | |  |
|  |  |  |  |  |  |  |  |  |  |  |  |  |

‘F’ emits



GROUPBY

RDD: **x**



James

Anna

Fred

John

‘A’ emits



RDD: **y**



A [ “Anna” ]

F [ “Fred” ]

J [ “John” ]

GROUPBY

RDD: **x**

James

Anna

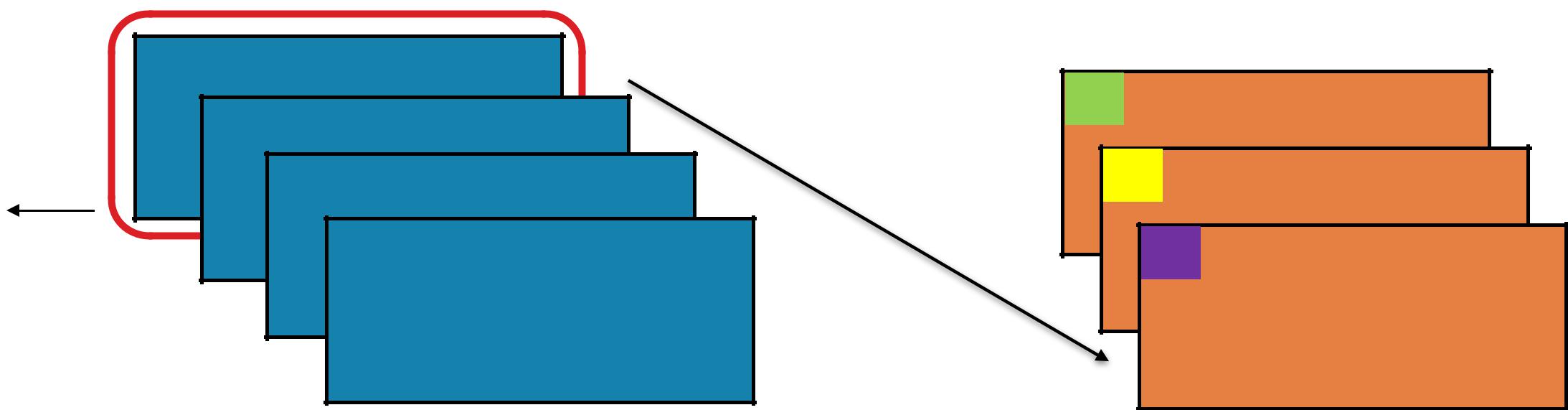
Fred

‘J’ emits

John



RDD: **y**



1. [ “Anna” ]
2. [ “Fred” ]
3. [ “John”, “James” ]

RDD: **x** RDD: **y**

GROUPBY

**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()]

1. ['John', 'Fred', 'Anna', 'James']
   1. [('A',['Anna']),('J',['John','James']),('F',['Fred'])]

val **x** = sc.parallelize(



Array("John", "Fred", "Anna", "James"))

val **y** = **x**.groupBy(w => w.charAt(0))

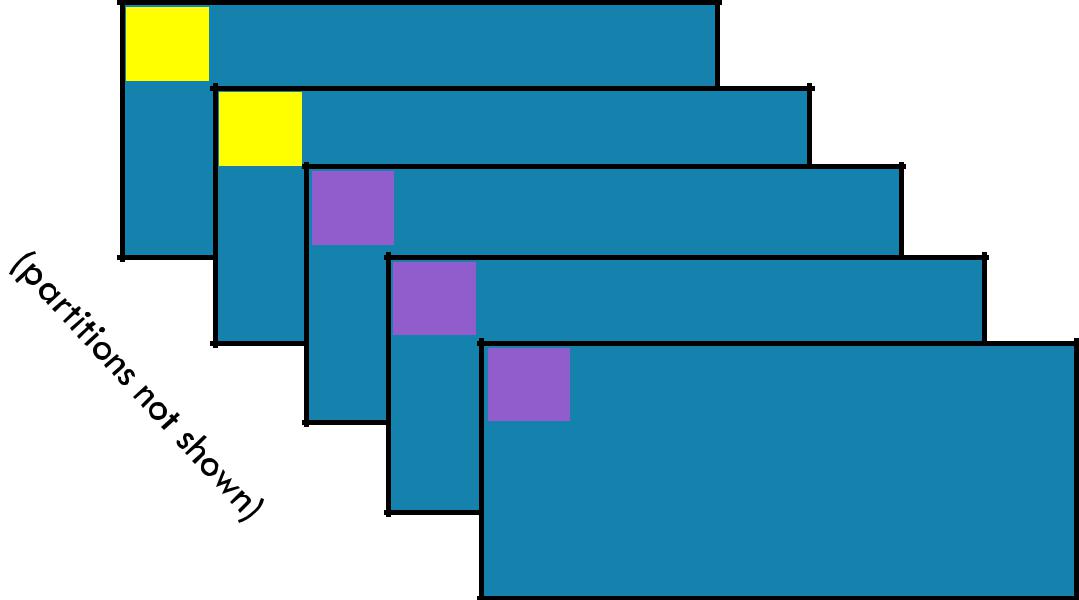
println(**y**.collect().mkString(", "))





GROUPBYKEY

Pair RDD: **x**



1. 5
2. 4

A 3

A 2

A 1

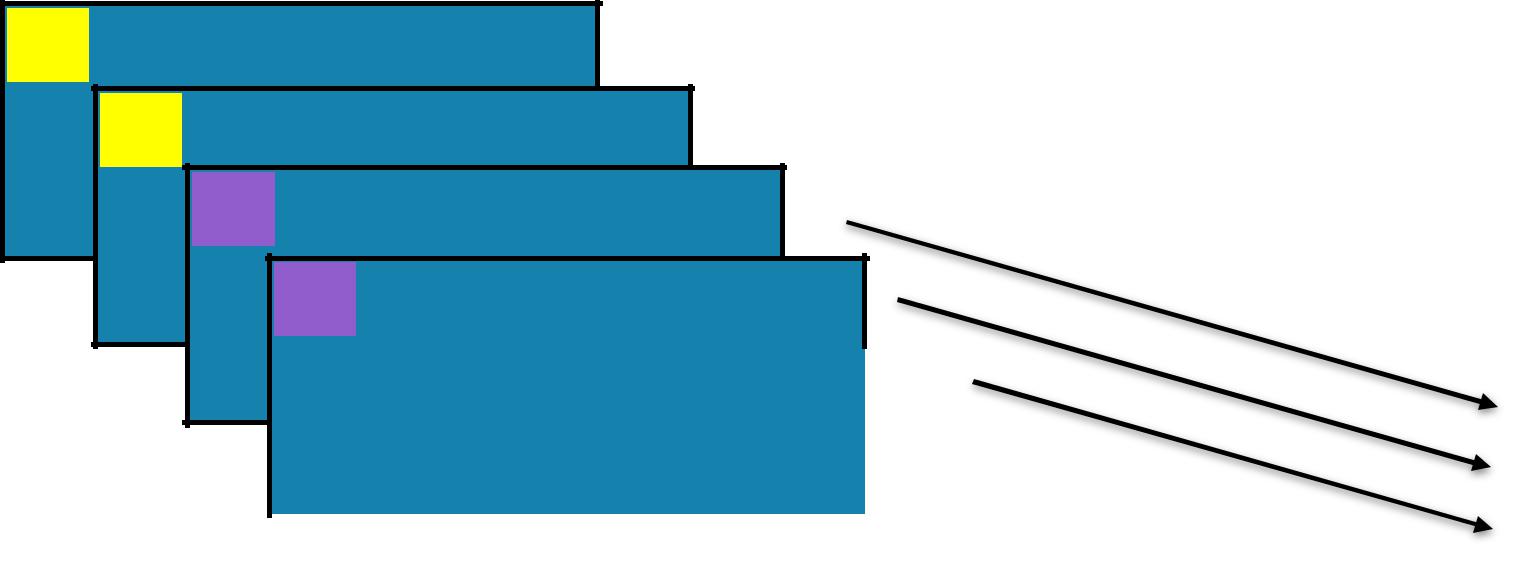


5 items in RDD



GROUPBYKEY

Pair RDD: **x** RDD: **y**



1. 5
2. 4

A 3

A 2

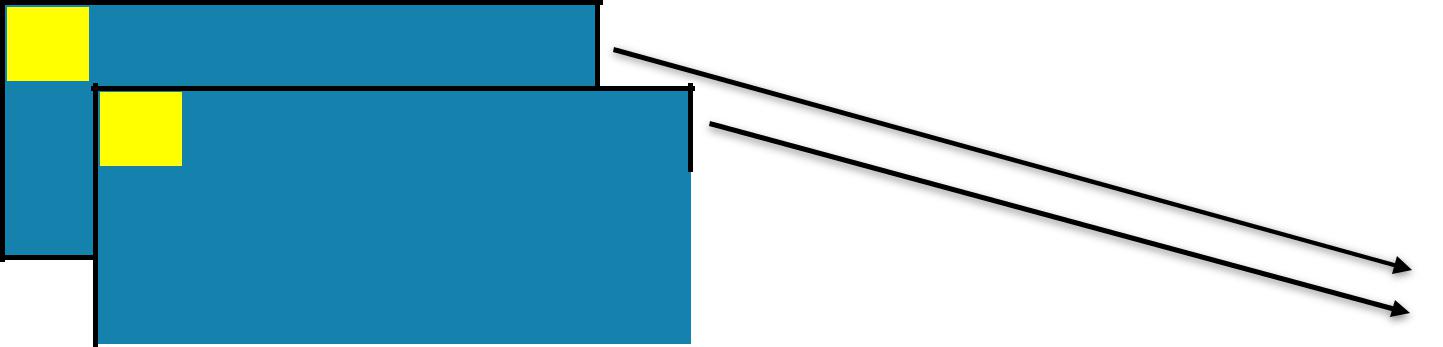
|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
|  |  |  |  | A | [ 2 , 3 , 1 ] |  |
|  | A | 1 |  |  |
|  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |





GROUPBYKEY

Pair RDD: **x** RDD: **y**



1. 5
2. 4

|  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | A | 3 |  | |  |  |  | B | [ 5 , 4 ] | |  |
|  |  |  |  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |  |  |
|  |  | A | 2 | |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |  |  |
|  |  |  | A | 1 |  |  |  |  | A | [ 2 , 3 , 1 ] |  |
|  |  |  |  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |  |  |



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.

**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])]

val **x** = sc.parallelize(



Array(('B',5),('B',4),('A',3),('A',2),('A',1)))

val **y** = **x**.groupByKey()

println(**x**.collect().mkString(", "))

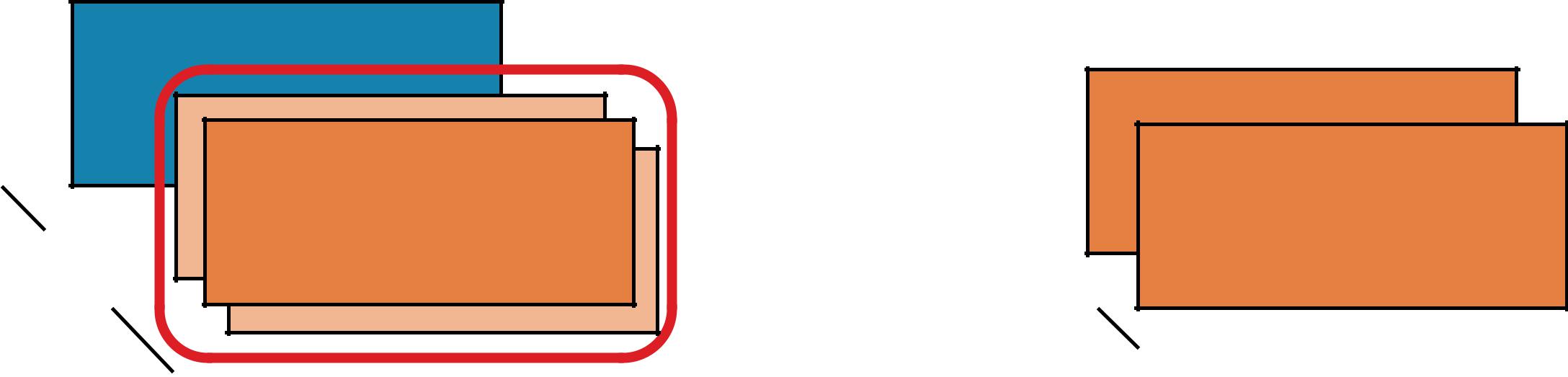
println(**y**.collect().mkString(", "))





MAPPARTITIONS

RDD: **x** RDD: **y**



B



|  |  |  |  |
| --- | --- | --- | --- |
| partitions |  | B |  |
|  | A | A |  |
|  |  |  |



REDUCEBYKEY VS GROUPBYKEY

val words = Array("one", "two", "two", "three", "three", "three")

val wordPairsRDD = sc.parallelize(words).map(word => (word, 1))

val wordCountsWithReduce = wordPairsRDD

.reduceByKey(\_ + \_)

.collect()

val wordCountsWithGroup = wordPairsRDD

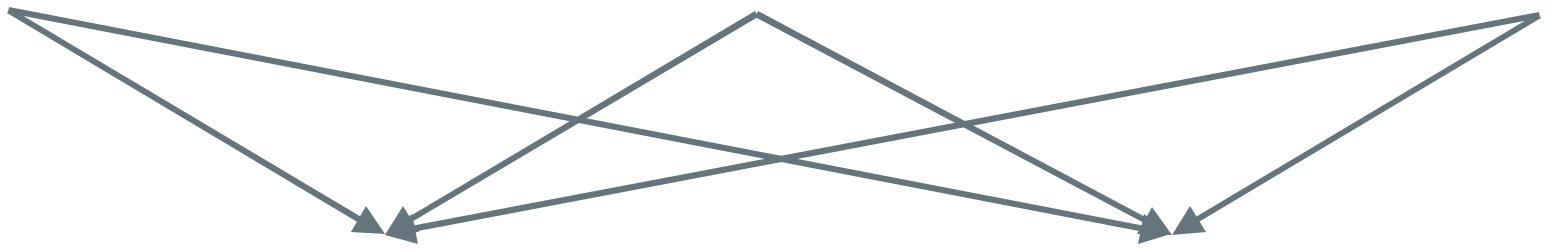
.groupByKey()

.map(t => (t.\_1, t.\_2.sum))

.collect()

REDUCEBYKEY

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
|  |  |  | (a, 1) |  |  | (a, 1) |  |
| (a, 1) |  |  |  |  | (a, 1) |  |
| (a, 1) |  | (a, 1) | (a, 2) |  |  |
|  |  |  |  | (a, 3) |  |
| (b, 1) | (b, 1) |  | (b, 1) | (b, 2) |  | (a, 1) |  |
|  |  |  |  | (b, 2) |  |
|  |  |  | (b, 1) |  |  | (b, 1) |  |
|  |  |  |  |  | (b, 1) |  |
|  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |



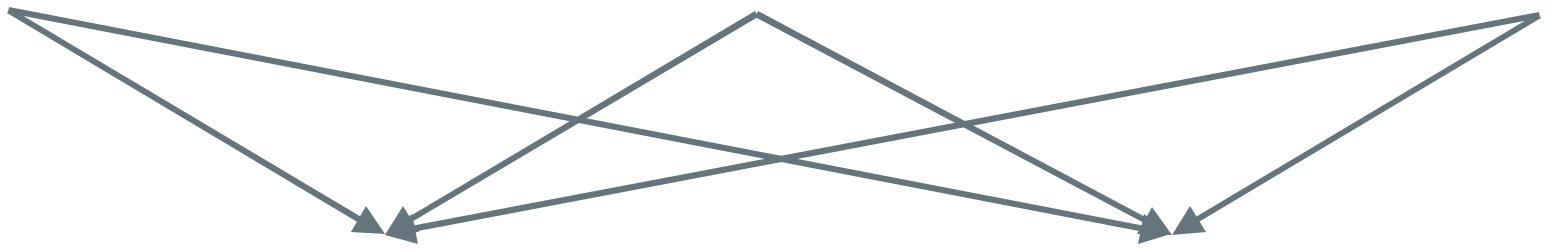
a b

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| (a, 1) |  |  | (b, 1) |  |
| (a, 2) | (a, 6) |  | (b, 2) | (b, 5) |
| (a, 3) |  |  | (b, 2) |  |
|  |  |  |  |  |



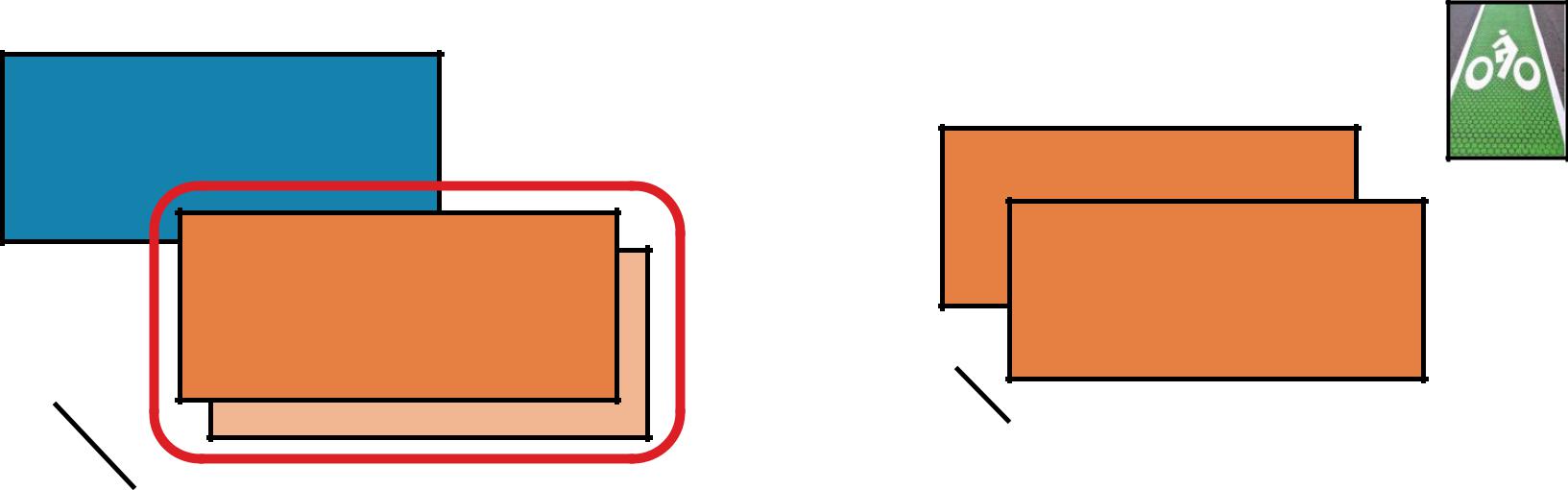
GROUPBYKEY

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
|  |  | (a, 1) |  | (a, 1) |  |
|  |  |  | (a, 1) |  |
| (a, 1) |  | (a, 1) |  |  |
|  |  | (a, 1) |  |
| (b, 1) |  | (b, 1) |  |  |
|  |  | (b, 1) |  |
|  |  | (b, 1) |  |  |
|  |  |  | (b, 1) |  |
|  |  |  |  |  |
|  |  |  |  |  |  |
|  |  |  |  |  |  |



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



MAPPARTITIONS B



B

A

A

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

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

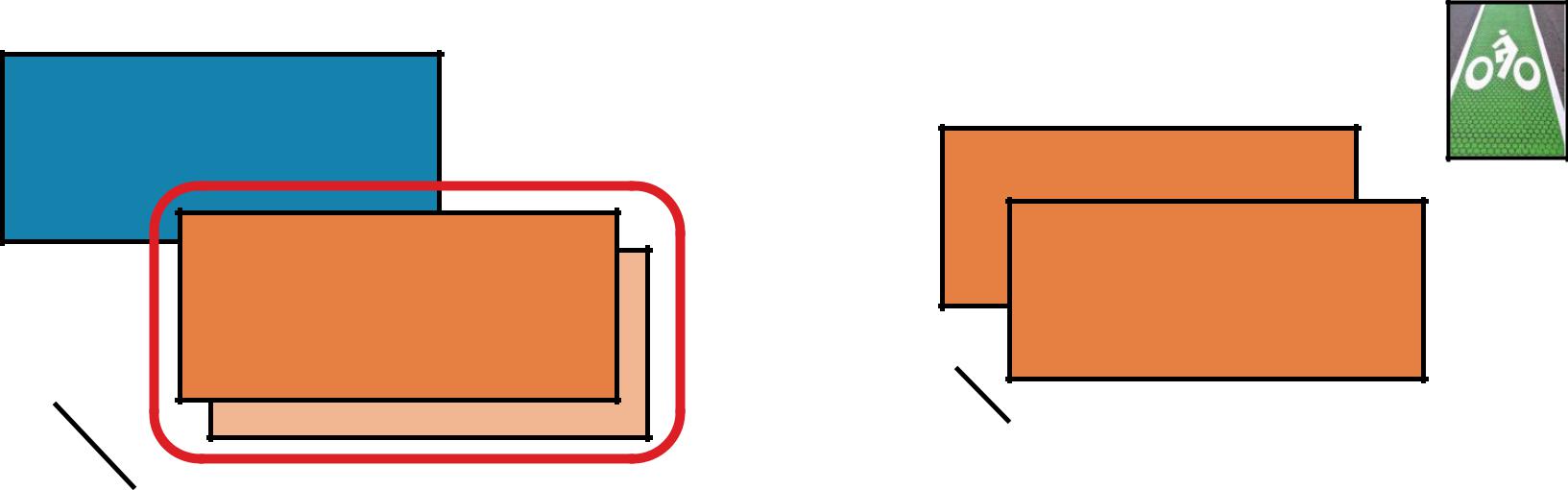


|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **x** | = sc.parallelize([1,2,3], 2) | | |  |
| def | | **f(**iterator**):** yield | sum(iterator); yield 42 | **x:** [[1], [2, 3]] |
| **y** | = | **x**.mapPartitions(**f**) |  | **y:** [[1, 42], [5, 42]] |



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



MAPPARTITIONS B



B

A

A

**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 } | | | **x:** Array(Array(1), Array(2, 3)) |
| val | **y** | = | **x**.mapPartitions(f) | **y:** Array(Array(1, 42), Array(5, 42)) |



* glom() flattens elements on the same partition val **xOut** = **x**.glom().collect()

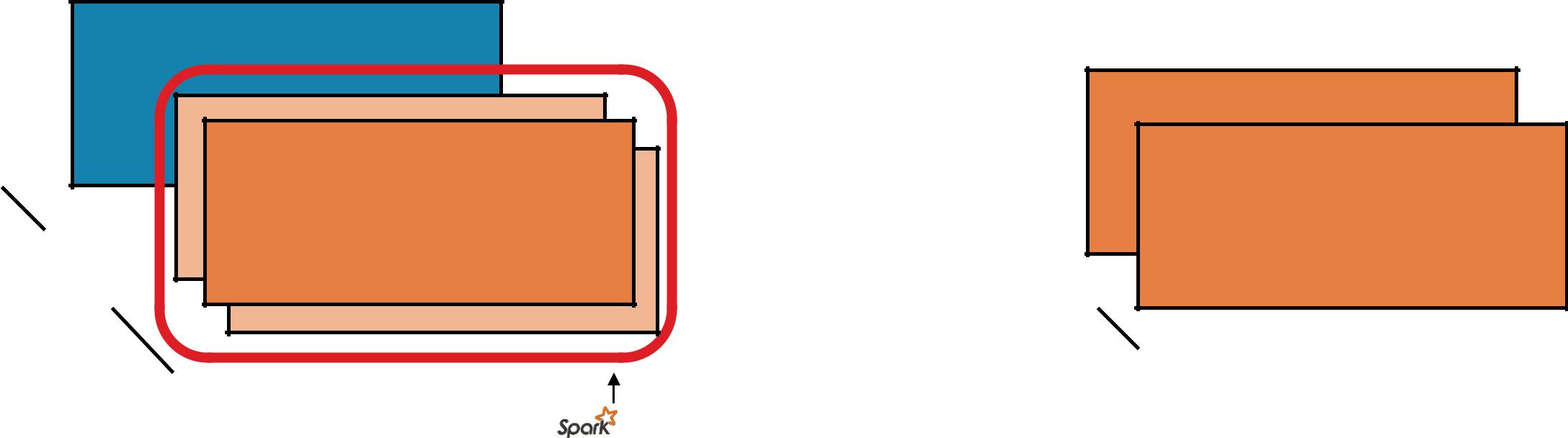
val **yOut** = **y**.glom().collect()





MAPPARTITIONSWITHINDEX

RDD: **x** RDD: **y**



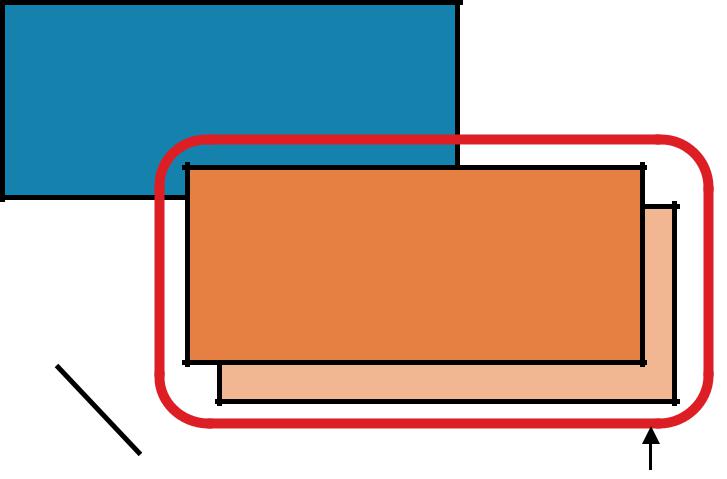
B



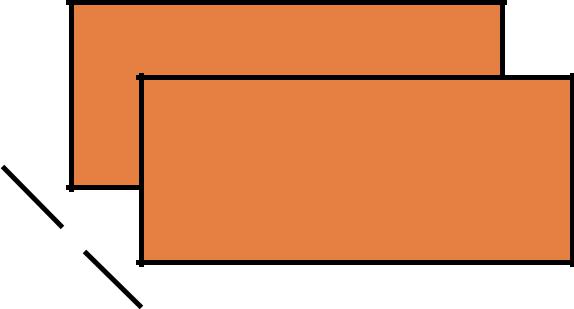
|  |  |  |  |
| --- | --- | --- | --- |
| partitions |  | B |  |
|  | A | A |  |
|  |  |  |
|  |  | input |  |

partition index





MAPPARTITIONSWITHINDEX B



B

|  |  |  |  |
| --- | --- | --- | --- |
| A |  | A |  |
|  | partition index |  |
|  |  |  |

**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)

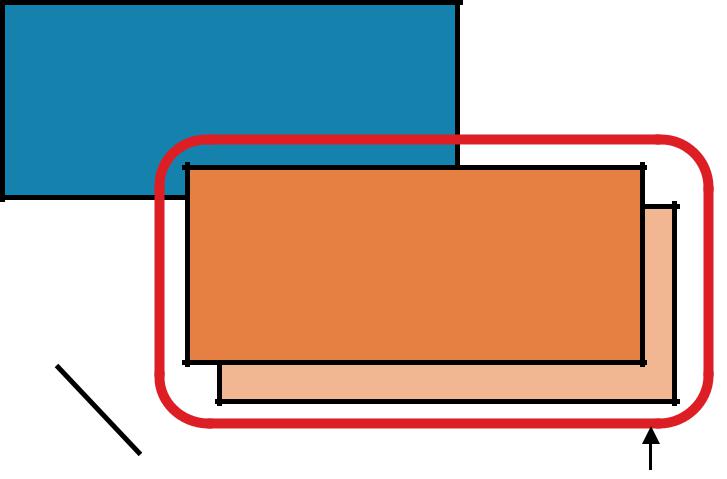
* glom() flattens elements on the same partition print(**x**.glom().collect()) print(**y**.glom().collect())



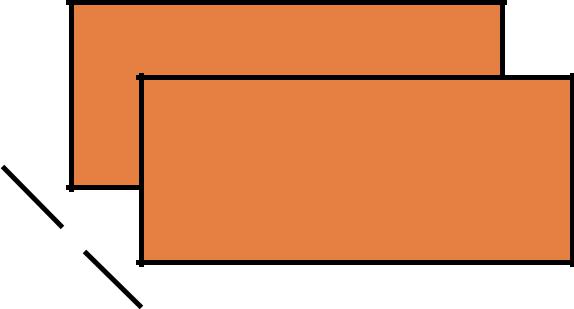
B A



1. [[1], [2, 3]]
   1. [[0, 1], [1, 5]]



MAPPARTITIONSWITHINDEX B



B

|  |  |  |  |
| --- | --- | --- | --- |
| A |  | A |  |
|  | partition index |  |
|  |  |  |

**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()



B A



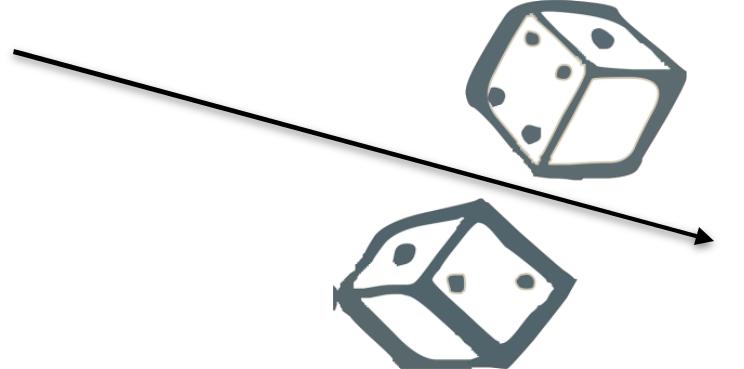
1. Array(Array(1), Array(2, 3))
   1. Array(Array(0, 1), Array(1, 5))



SAMPLE

RDD: **x** RDD: **y**

5

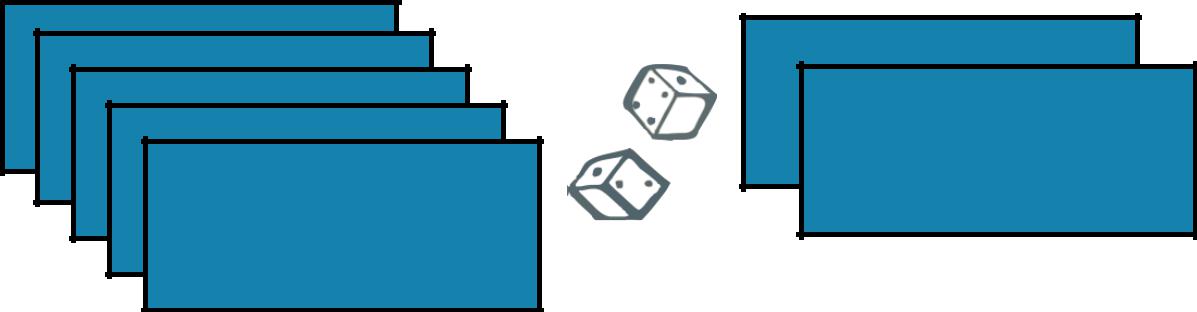


4

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | 3 |  |  |  |  |  | 1 |  |  |
|  |  |  |  |  |  |  |  |  |  |
|  |  | 2 |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |
|  |  |  | 1 |  |  |  |  | 3 |  |
|  |  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |



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())

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(", "))

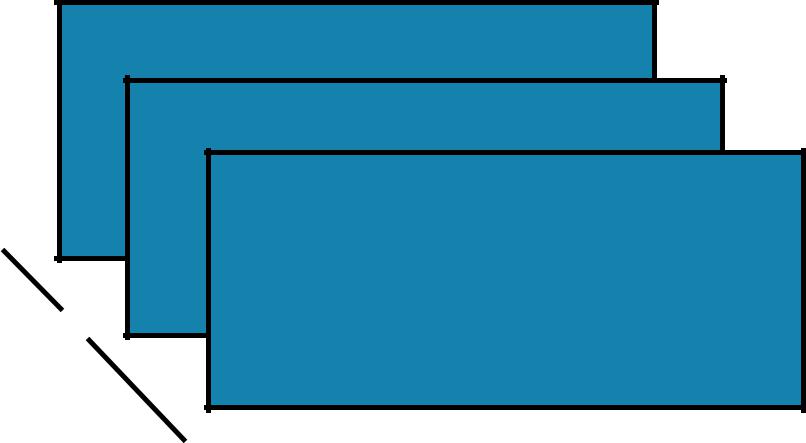


1. [1, 2, 3, 4, 5]
   1. [1, 3]



UNION

RDD: **x** RDD: **y**



3

2

1

B

A



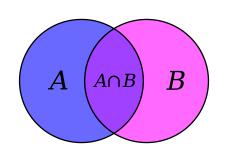
4

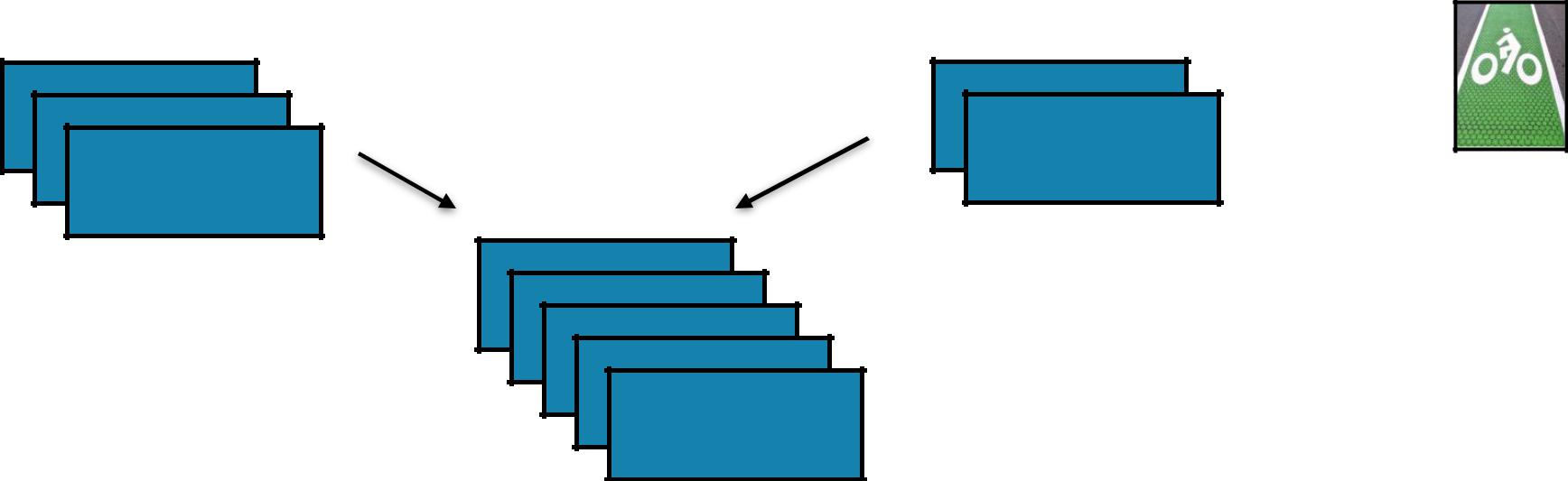
3

C



|  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | 4 |  |  |  |  |  | RDD: **z** | | | |  |
|  |  | 3 |  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |  |  |
|  |  |  | 3 |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |  |  |
|  |  |  |  | 2 |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |
|  |  |  |  |  | 1 |  |  |  |  |  |  |
| C | |  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  | | |  |  |  |  |
|  |  | B | |  |  | | | |  |  |  |
|  |  |  | A |  |  | | | | |  |  |
|  |  |  |  |  |  |  |  |  |  |



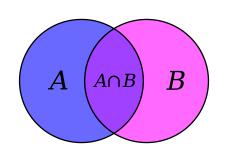


UNION

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

**union(*otherRDD*)**

|  |  |  |  |
| --- | --- | --- | --- |
| **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] |  |
|  |  |  |
| val | **x** = sc.parallelize(Array(1,2,3), 2) |  |  |
| val | **y** = sc.parallelize(Array(3,4), 1) | **z:** [[1], [2, 3], [3, 4]] |  |
| val | **z** = **x**.union(**y**) |  |  |
| val | **zOut** = **z**.glom().collect() |  |  |



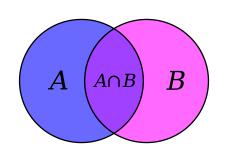
JOIN

RDD: **x** RDD: **y**



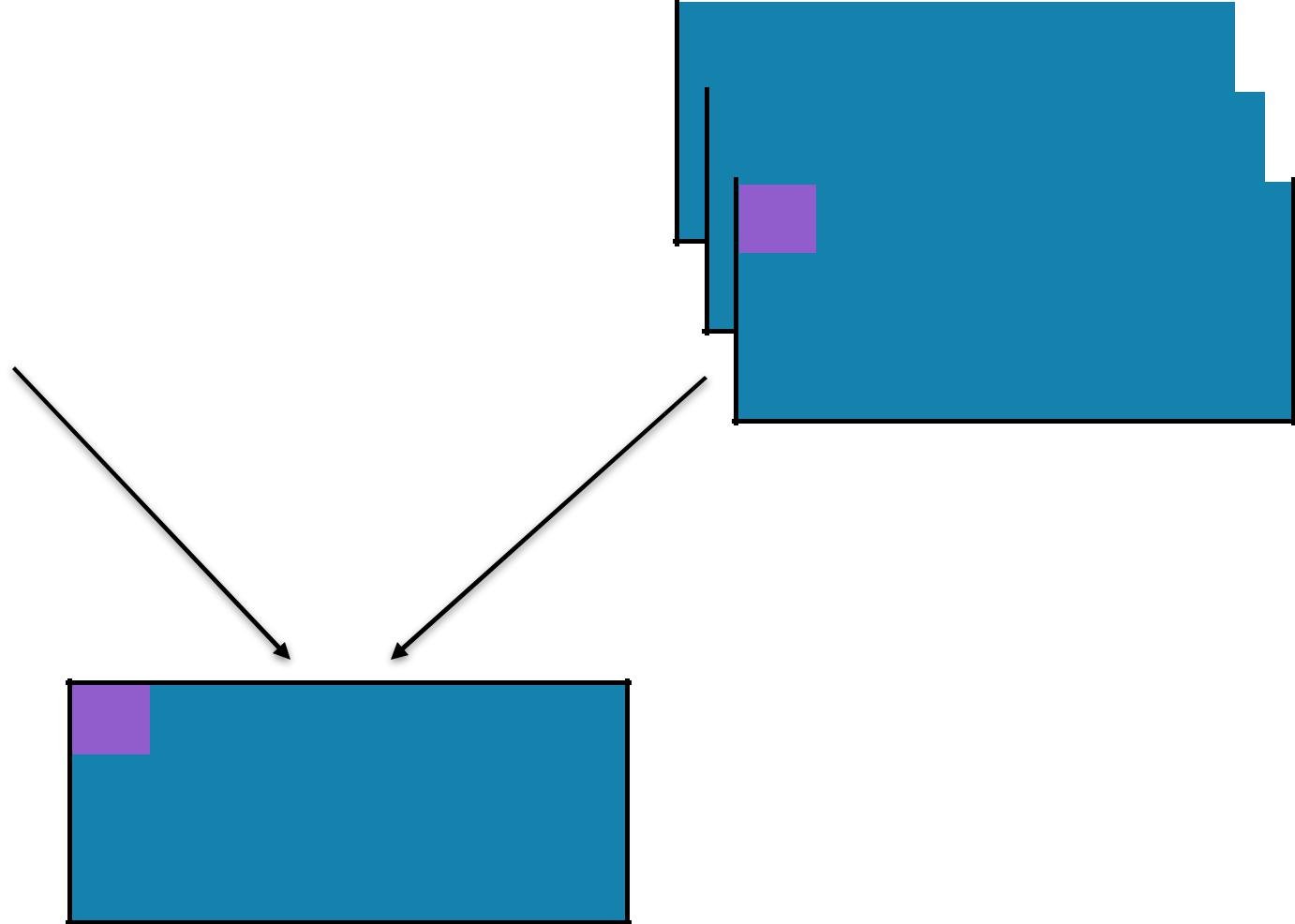
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  |  |  |  |  |  |  |  |  |  |  |  |  | 5 | |  |  |  |
|  |  |  |  |  |  |  |  |  |  | B | |  |  |  |  |
|  |  | B |  | 2 |  |  |  |  |  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  | A | |  | 4 |  |  |  |
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |  |
|  |  | A | | 1 |  |  |  |  |  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |

A 3



JOIN

RDD: **x** RDD: **y**

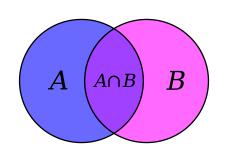


|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  |  |  |  |  |  |  |  |  |  |  |  |  | 5 | |  |  |  |
|  |  |  |  |  |  |  |  |  |  | B | |  |  |  |  |
|  |  | B |  | 2 |  |  |  |  |  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  | A | |  | 4 |  |  |  |
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |  |
|  |  | A | | 1 |  |  |  |  |  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |

A 3

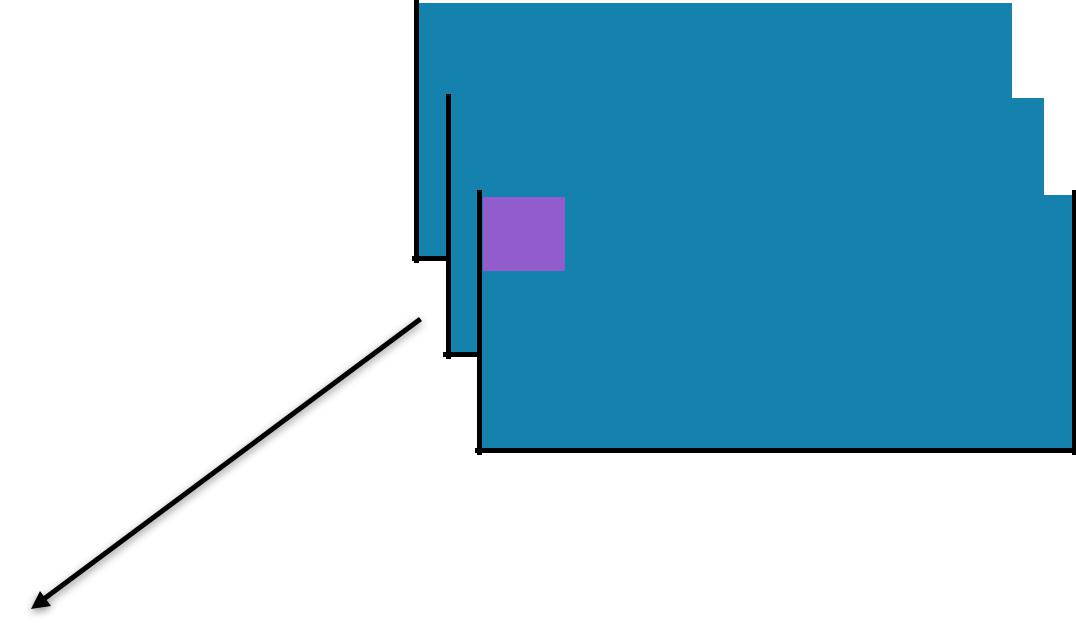
RDD: **z**

A (1, 3)



JOIN

RDD: **x** RDD: **y**



|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  |  |  |  |  |  |  |  |  |  |  |  |  | 5 | |  |  |  |
|  |  |  |  |  |  |  |  |  |  | B | |  |  |  |  |
|  |  | B |  | 2 |  |  |  |  |  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  | A | |  | 4 |  |  |  |
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |  |
|  |  | A | | 1 |  |  |  |  |  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |

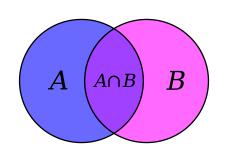
A 3



RDD: **z**

A (1, 4)

A (1, 3)

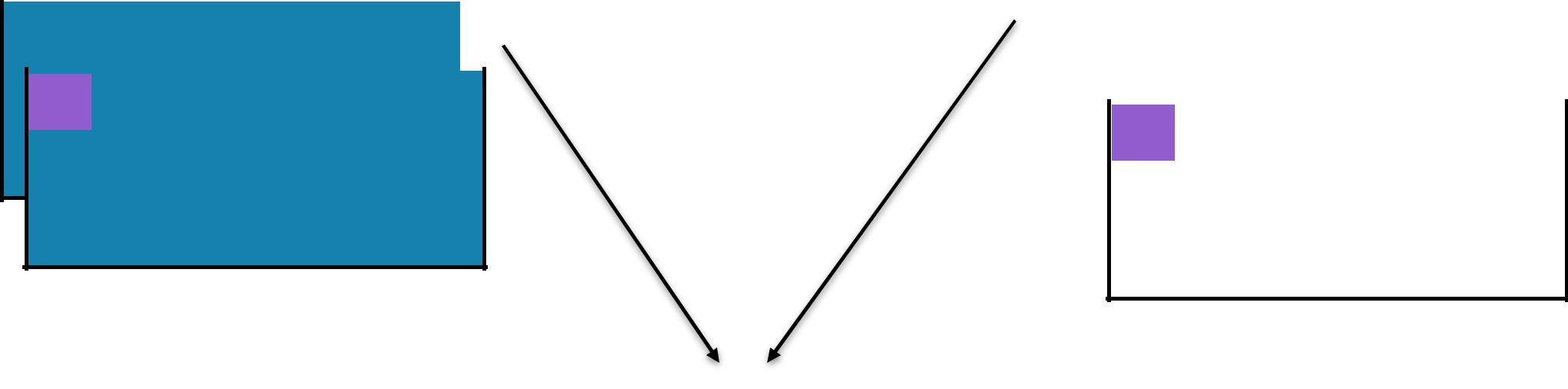
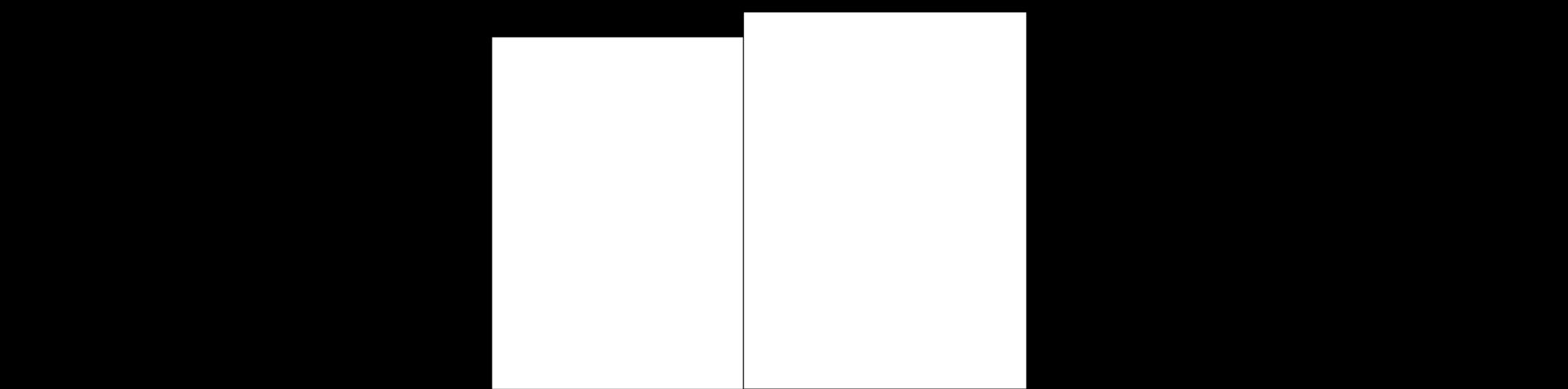


JOIN

RDD: **x** RDD: **y**

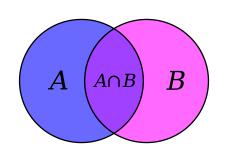


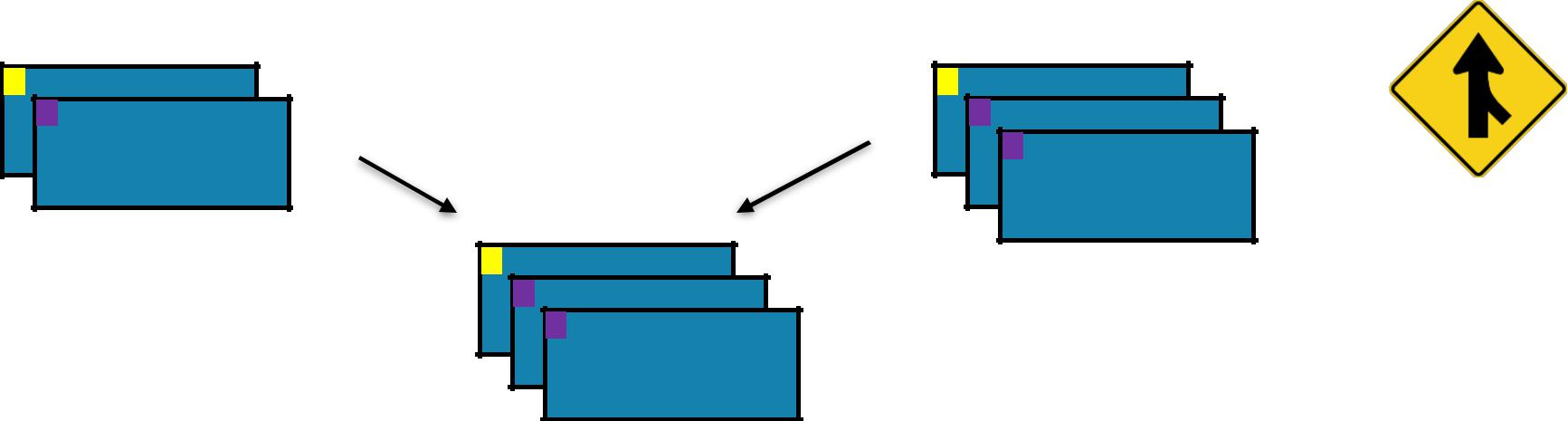
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  |  |  |  |  |  |  |  |  |  |  |  |  | 5 | |  |  |  |
|  |  |  |  |  |  |  |  |  |  | B | |  |  |  |  |
|  |  | B |  | 2 |  |  |  |  |  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  | A | |  | 4 |  |  |  |
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |  |
|  |  | A | | 1 |  |  |  |  |  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |



A 3

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| B | |  | (2, 5) | |  |  | RDD: **z** | |
|  |  | A |  | (1, 4) | |  |  |  |
|  |  |  |  | A | (1, 3) |  |  |  |
|  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |





JOIN

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

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



**x** = sc.parallelize([("a", 1), ("b", 2)])



**y** = sc.parallelize([("a", 3), ("a", 4), ("b", 5)])

1. = **x**.join(**y**) print(**z**.collect())

val **x** = sc.parallelize(Array(("a", 1), ("b", 2)))



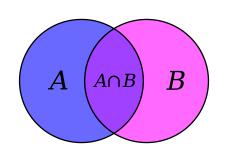
val **y** = sc.parallelize(Array(("a", 3), ("a", 4), ("b", 5)))

val **z** = **x**.join(**y**)

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



1. [("a", 1), ("b", 2)]
   1. [("a", 3), ("a", 4), ("b", 5)]
   2. [('a', (1, 3)), ('a', (1, 4)), ('b', (2, 5))]





DISTINCT

RDD: **x**

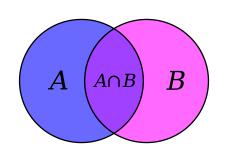
4

3

3

2

1



DISTINCT

RDD: **x**

4

3

3

2

1



RDD: **y**



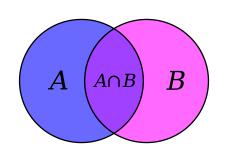
4

3

3

2

1



DISTINCT

RDD: **x**

4

3

3

2

1



RDD: **y**

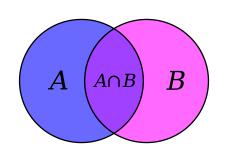


4

3

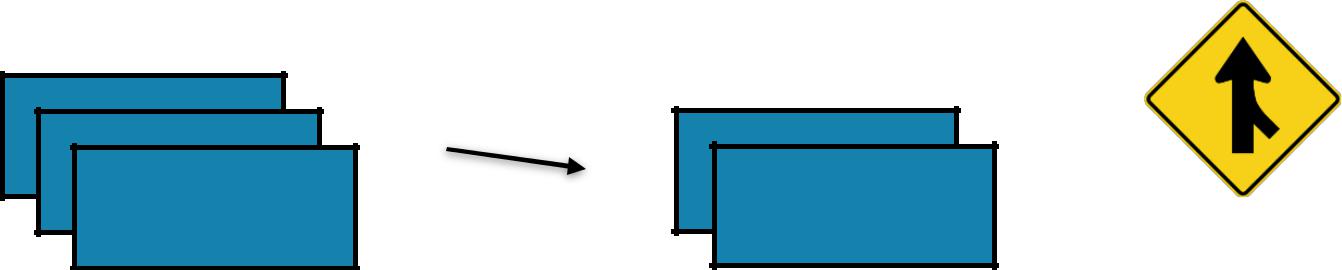
2

1



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())

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

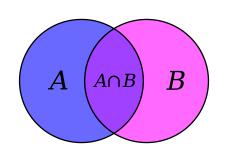
**y:** [1, 2, 3, 4]

val **x** = sc.parallelize(Array(1,2,3,3,4))



val **y** = **x**.distinct()

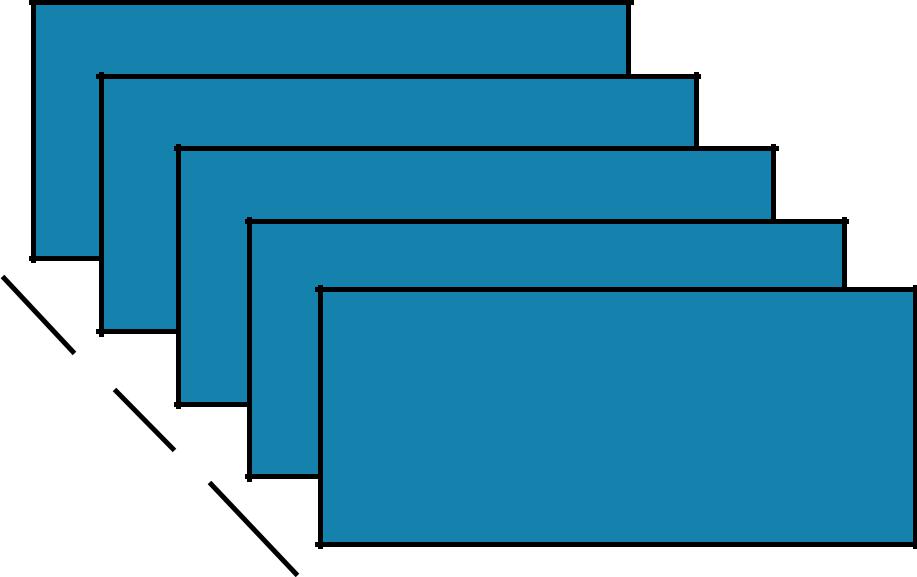
println(**y**.collect().mkString(", "))





COALESCE

RDD: **x**



C

B

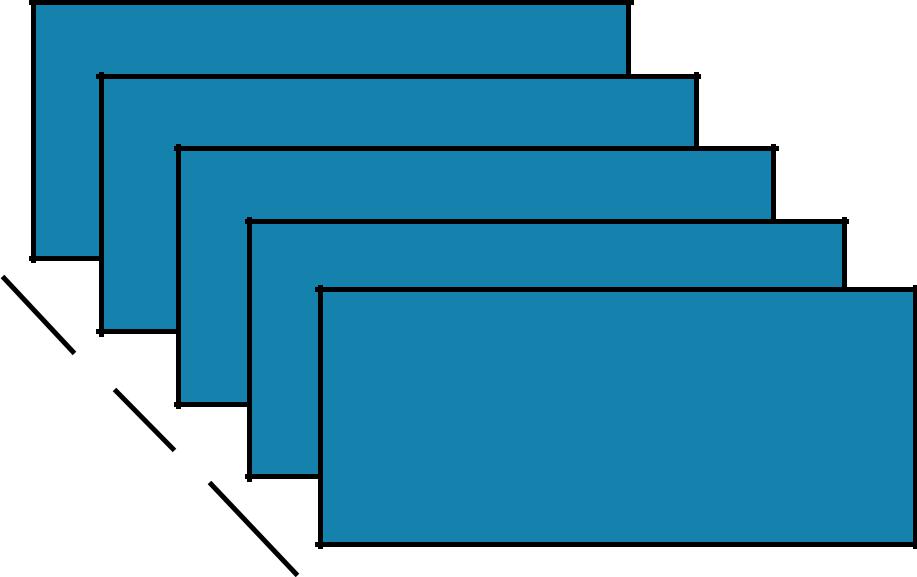
A





COALESCE

RDD: **x**



RDD: **y**



C

B

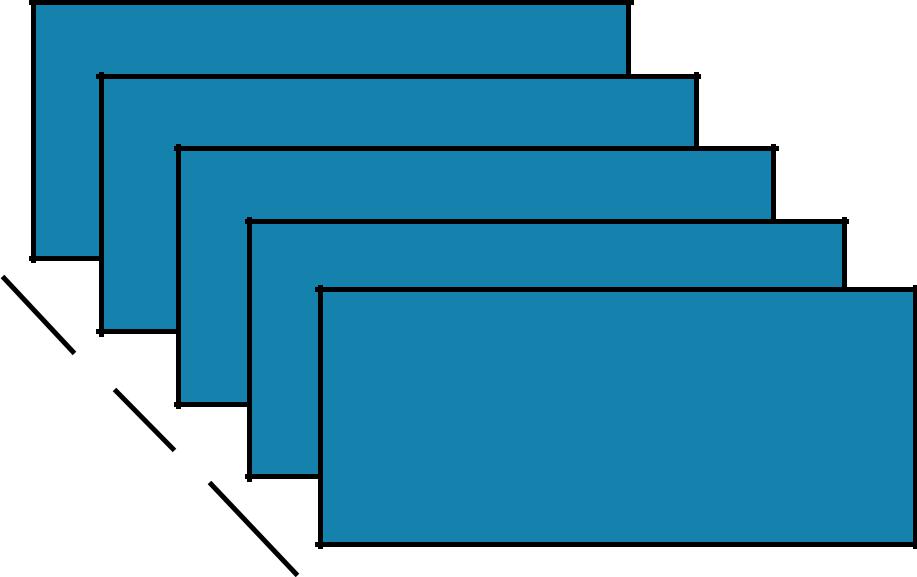
|  |  |  |
| --- | --- | --- |
| A | AB |  |
|  |  |



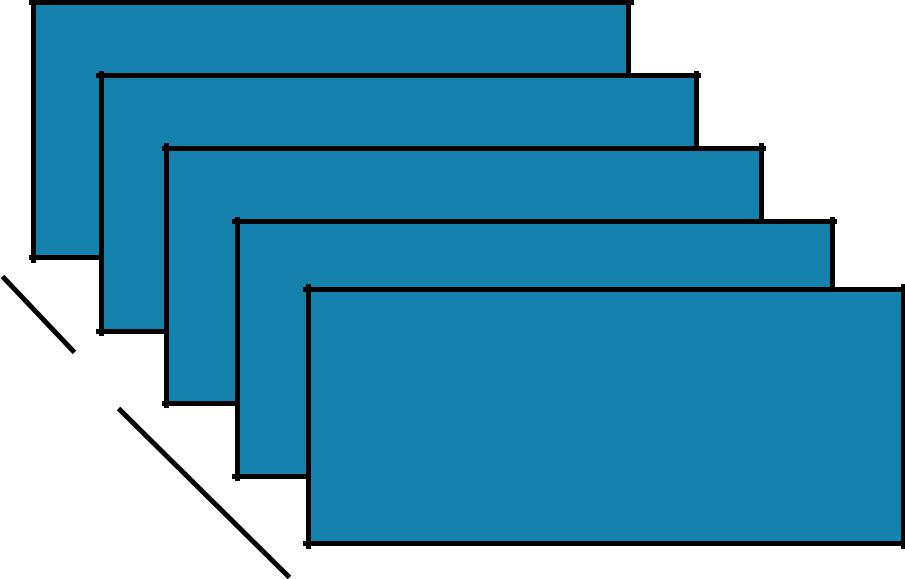


COALESCE

RDD: **x**



RDD: **y**



|  |  |  |  |
| --- | --- | --- | --- |
| C |  | C |  |
|  | B |  |  |
|  | A | AB |  |
|  |  |  |

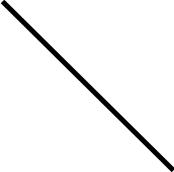


COALESCE

C



C



B



AB



A

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

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

**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]]

val **x** = sc.parallelize(Array(1, 2, 3, 4, 5), 3)



val **y** = **x**.coalesce(2)

val **xOut** = **x**.glom().collect()

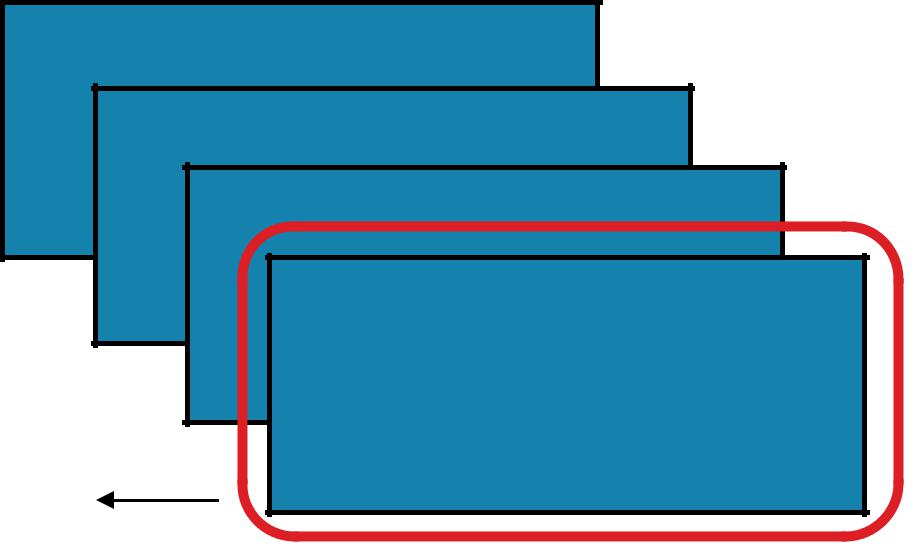
val **yOut** = **y**.glom().collect()





KEYBY

RDD: **x** RDD: **y**



James

Anna

Fred

|  |  |
| --- | --- |
| John | J “John” |



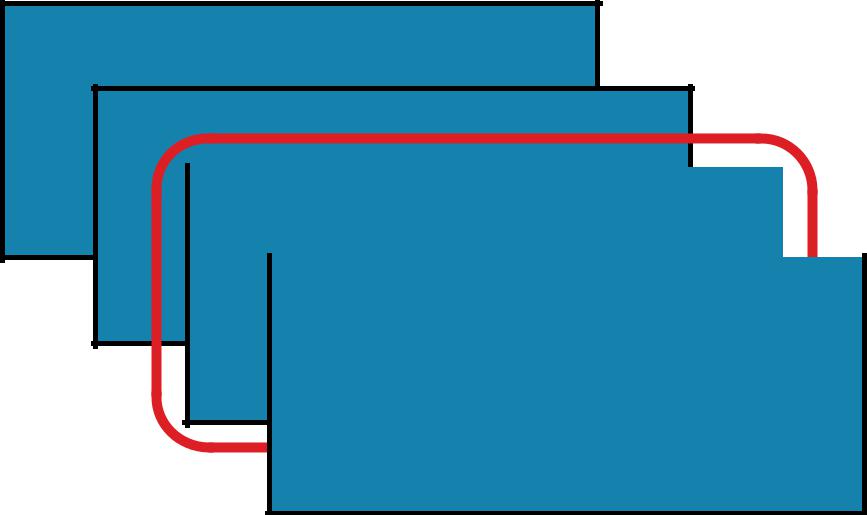
‘J’ emits





KEYBY

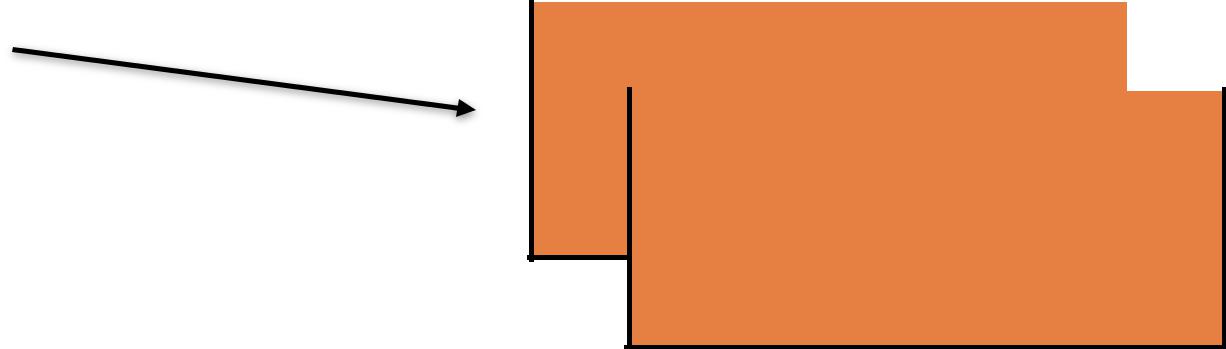
RDD: **x** RDD: **y**



James

Anna

|  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Fred | |  |  |  | F |  |  | “Fred” | |  |
|  |  |  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |  |
|  | John |  |  |  |  |  |  | J | “John” |  |
|  |  |  |  |  |  |  |  |  |  |  |



‘F’

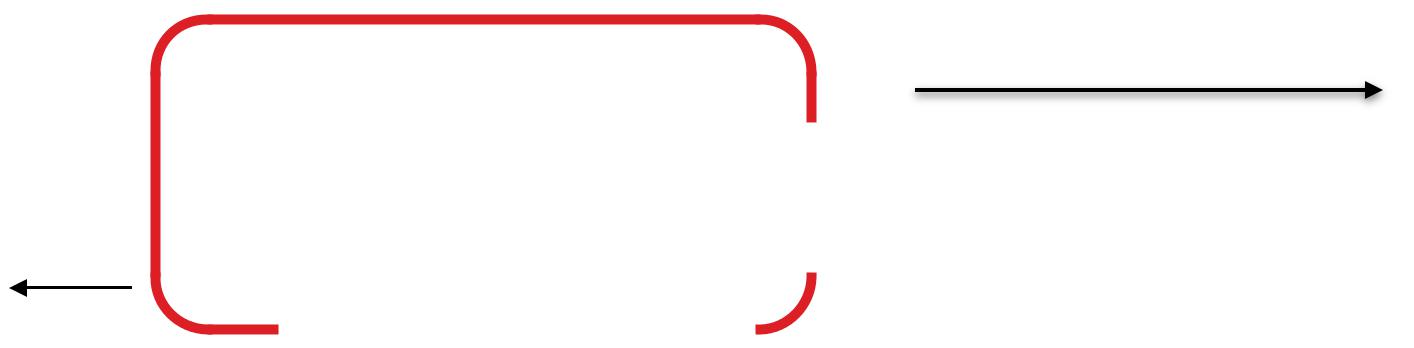




KEYBY

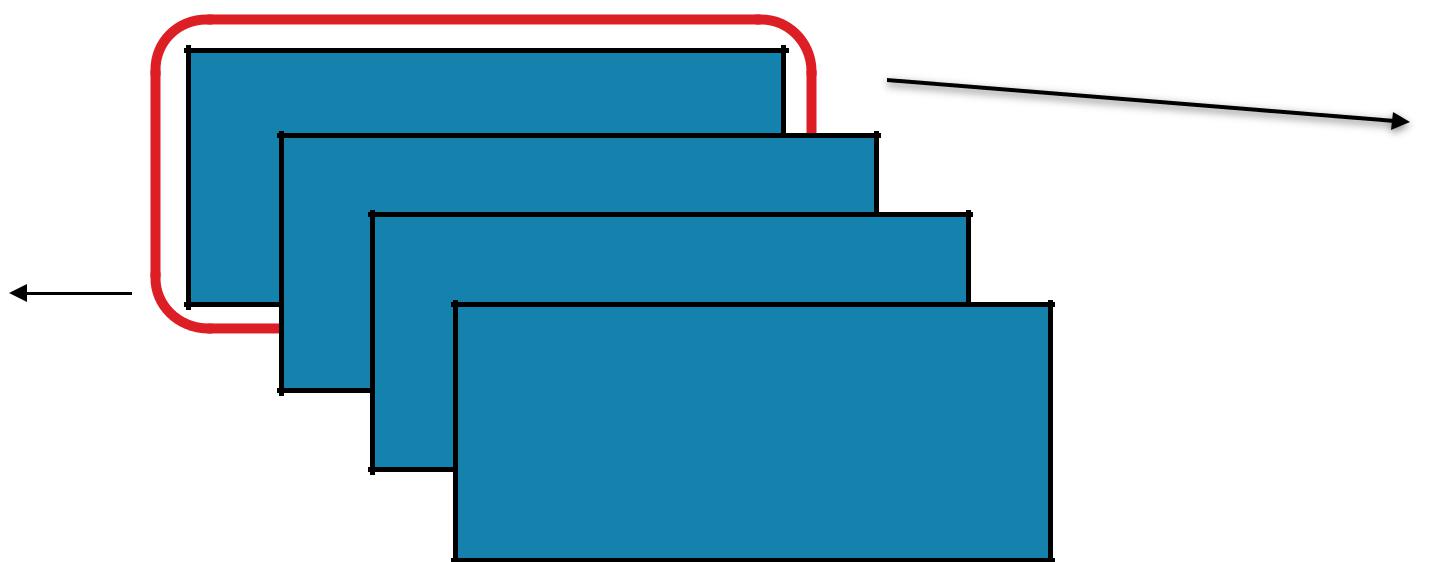
RDD: **x** RDD: **y**

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | James | | | |  |  |  |  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
|  |  | Anna | | |  |  |  |  | A | “Anna” | | |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |  |  |  | |  |  |  |
|  |  |  | Fred | |  |  |  |  |  | F |  | “Fred” | |  |  |  |
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| ‘A’ |  |  |  | John |  |  |  |  |  |  |  | J | “John” |  |  |  |
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |



KEYBY

RDD: **x**



James

Anna

Fred

‘J’ emits

John



RDD: **y**



1. “James”
2. “Anna”

F “Fred”

J “John”

RDD: **x**

RDD: **y**



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()

1. ['John', 'Fred', 'Anna', 'James']
   1. [('J','John'),('F','Fred'),('A','Anna'),('J','James')]

val **x** = sc.parallelize(



Array("John", "Fred", "Anna", "James"))

val **y** = **x**.keyBy(w => w.charAt(0))

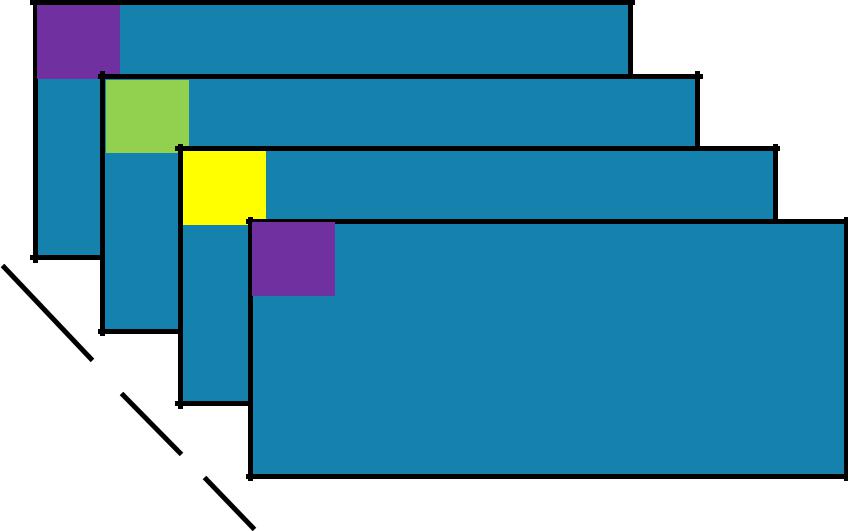
println(**y**.collect().mkString(", "))





PARTITIONBY

RDD: **x**



J “John”

A “Anna”

F “Fred”

J “James”





PARTITIONBY

|  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  |  |  |  | RDD: **x** | |  |  |  |  |  | RDD: **y** |  |
|  |  |  |  |  |  |  |  |  |  |  |  |  |
| J | | “John” | | | |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |  |  |  |
|  |  | A | “Anna” | | |  |  |  |  | J | “James” |  |
|  |  |  | F | “Fred” | |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |  |  |  |
|  |  |  |  | J | “James” |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |  |  |  |



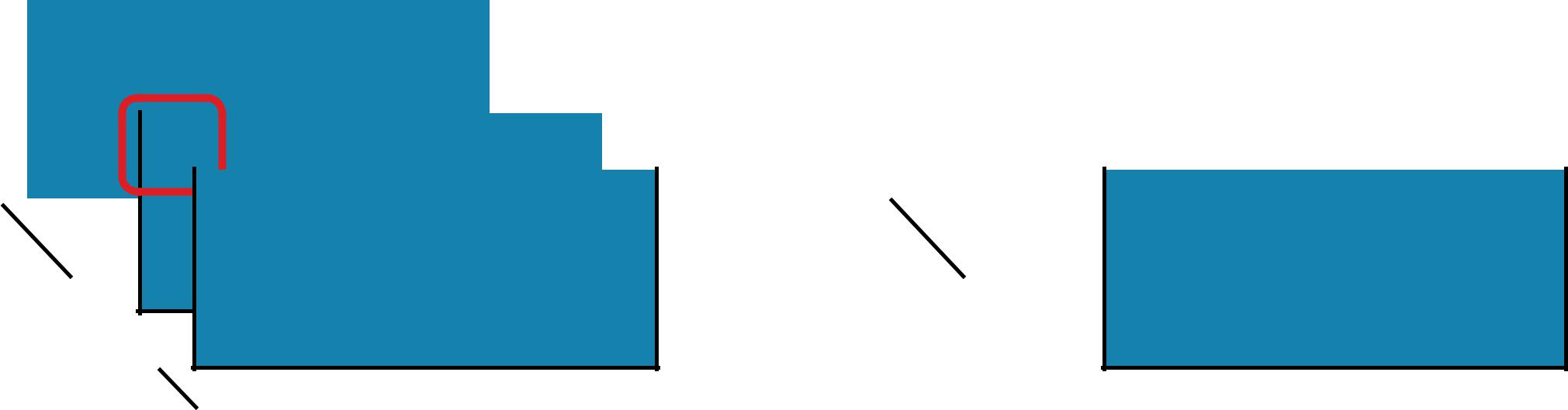
1



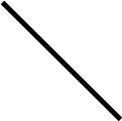


PARTITIONBY

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  |  |  |  |  | RDD: **x** | | |  |  |  |  |  |  | RDD: **y** | |  |  |
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| J | | “John” | | | | | |  |  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
|  |  | A | | “Anna” | | | |  |  |  |  | J | “James” | | |  |  |
|  |  |  |  | F | | “Fred” | |  |  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  | J | “James” |  |  |  |  |  |  | F | “Fred” |  |  |
|  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |



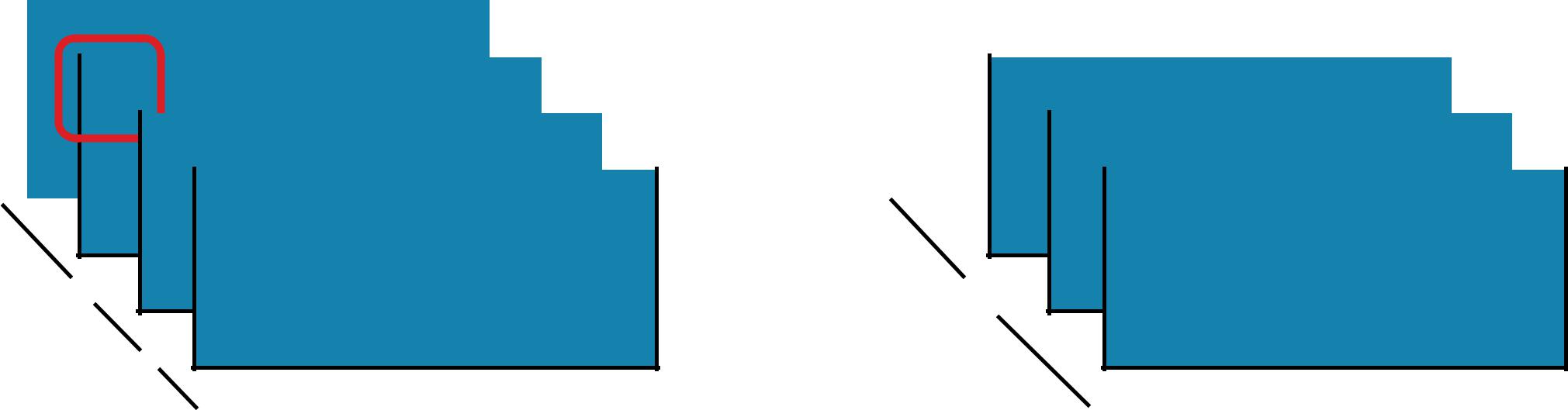
0





PARTITIONBY

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  |  |  |  |  |  |  | RDD: **x** | | | |  |  |  |  |  |  |  |  |  | RDD: **y** | |  |  |  |
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| J | | | | “John” | | | | | | |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
|  |  |  |  | A | | “Anna” | | | | |  |  |  |  |  | J | | “James” | | | |  |  |  |
|  |  |  |  |  |  | F | | | “Fred” | |  |  |  |  |  |  |  | A | | “Anna” | |  |  |  |
|  |  |  |  |  |  |  |  |  | J | “James” |  |  |  |  |  |  |  |  |  | F | “Fred” |  |  |  |
|  |  |  |  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |



0

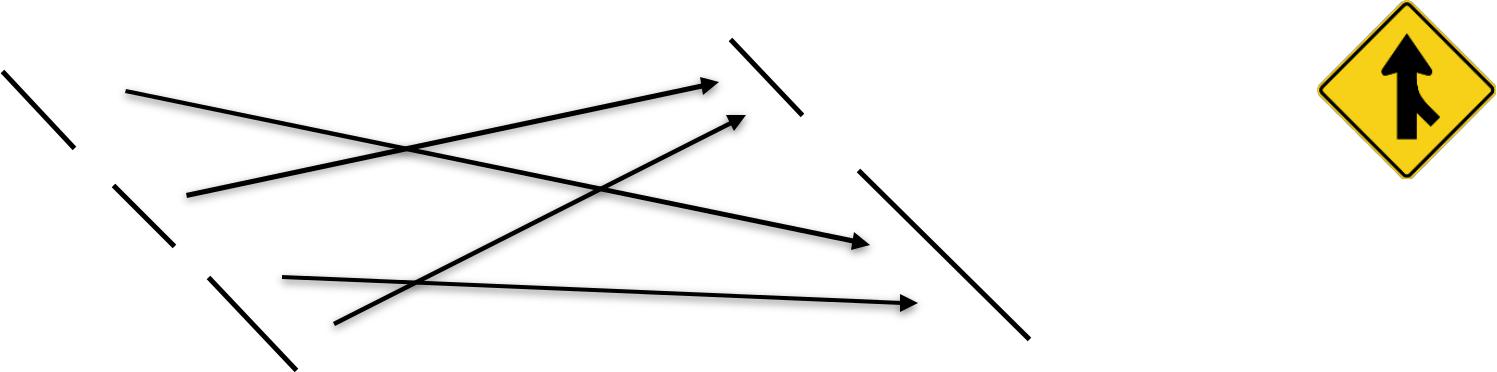




PARTITIONBY

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  |  |  |  |  | RDD: **x** | | |  |  |  |  |  |  |  |  | RDD: **y** | |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
|  | J | | “John” | | | | |  |  |  |  | J | | “John” | | | |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
|  |  |  | A | “Anna” | | | |  |  |  |  |  |  | J | “James” | | |  |  |  |  |
|  |  |  |  | F | | “Fred” | |  |  |  |  |  |  |  | A | “Anna” | |  |  |  |  |
| 1 |  |  |  |  |  | J | “James” |  |  |  |  |  |  |  |  | F | “Fred” |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |





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)



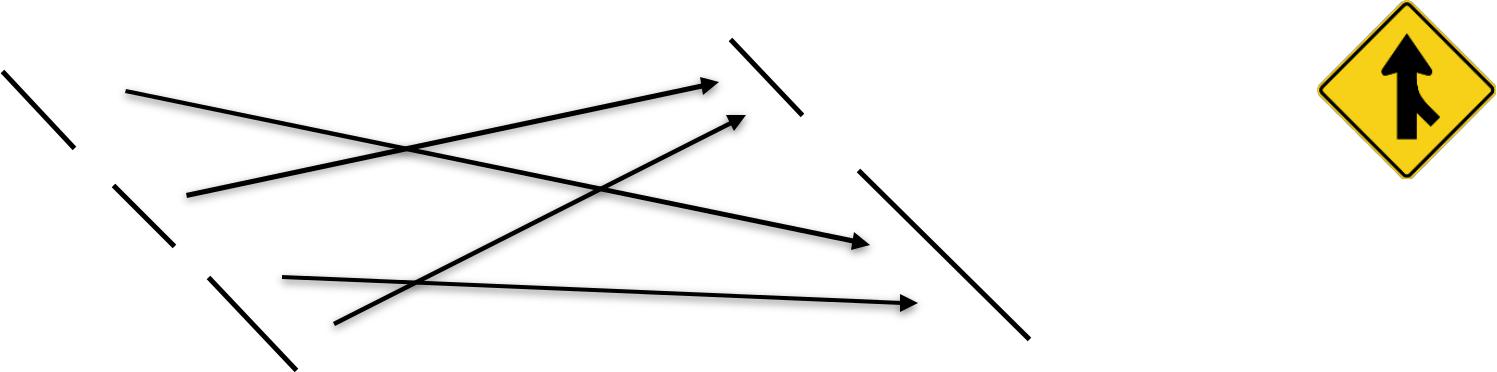
**y** = **x**.partitionBy(2, lambda w: 0 if w[0] < 'H' else 1)

print **x**.glom().collect()

print **y**.glom().collect()



1. [[('J', 'James')], [('F', 'Fred')], [('A', 'Anna'), ('J', 'John')]]
   1. [[('A', 'Anna'), ('F', 'Fred')], [('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() { | **x:** Array(Array((A,Anna), (F,Fred)), |
| val | numPartitions = 2 | Array((J,John), (J,James))) |
| def | getPartition(k:Any) = { |  |
| if (k.asInstanceOf[Char] < 'H') 0 else 1 | | **y:** Array(Array((F,Fred), (A,Anna)), |
| } |  | Array((J,John), (J,James))) |
| }) |  |  |



val **yOut** = **y**.glom().collect()

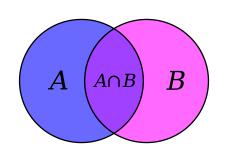


ZIP

RDD: **x** RDD: **y**



|  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | 3 |  |  |  | | | 9 |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |  |  |  |
|  |  | 2 |  |  |  | |  | 4 |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |  |  |  |
|  |  |  | 1 |  |  |  |  |  | 1 |  |  |  |
|  |  |  |  |  |  |  |  |  |  |  |  |  |
| B | |  |  |  |  | B | |  |  |  |  |  |
|  | A | |  |  |  |  | A | |  |  |  |  |
|  |  |  |  |  |  |  |  |

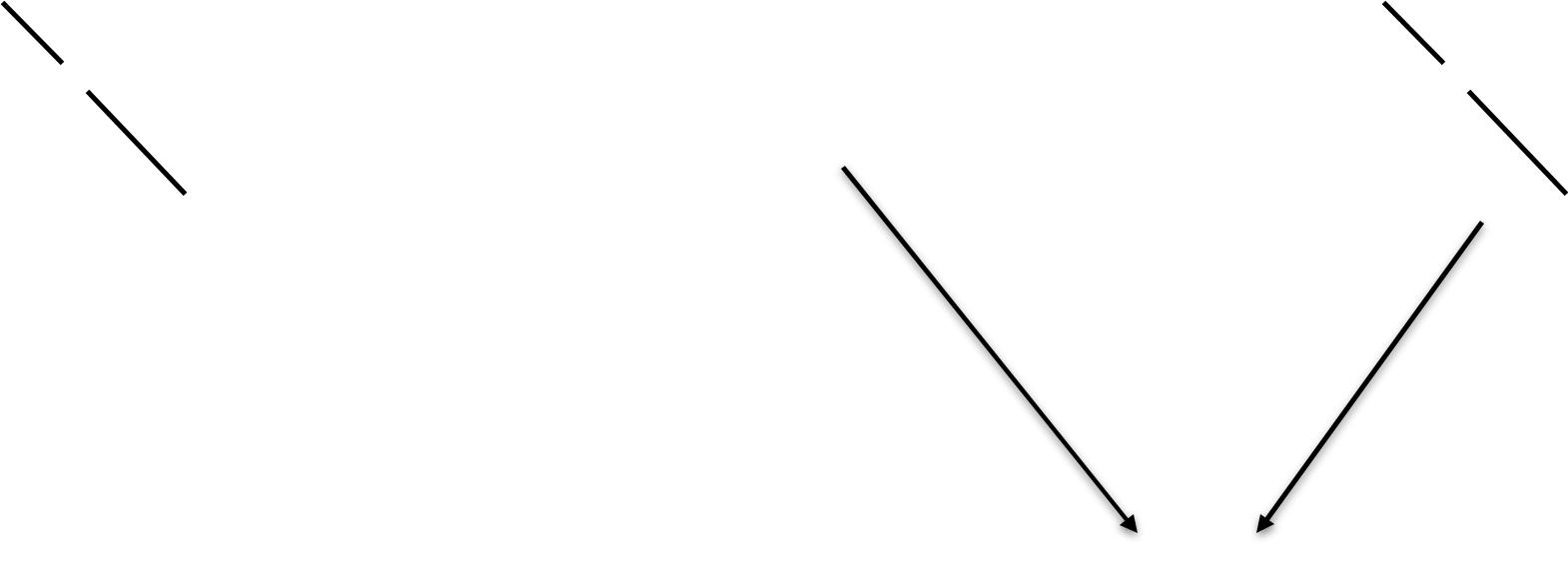


ZIP

RDD: **x** RDD: **y**

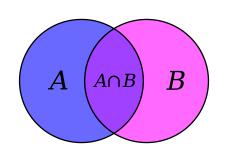


|  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | 3 |  |  |  | | | 9 |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |  |  |  |
|  |  | 2 |  |  |  | |  | 4 |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |  |  |  |
|  |  |  | 1 |  |  |  |  |  | 1 |  |  |  |
|  |  |  |  |  |  |  |  |  |  |  |  |  |
| B | |  |  |  |  | B | |  |  |  |  |  |
|  | A | |  |  |  |  | A | |  |  |  |  |
|  |  |  |  |  |  |  |  |



4 RDD: **z**

1 1



A

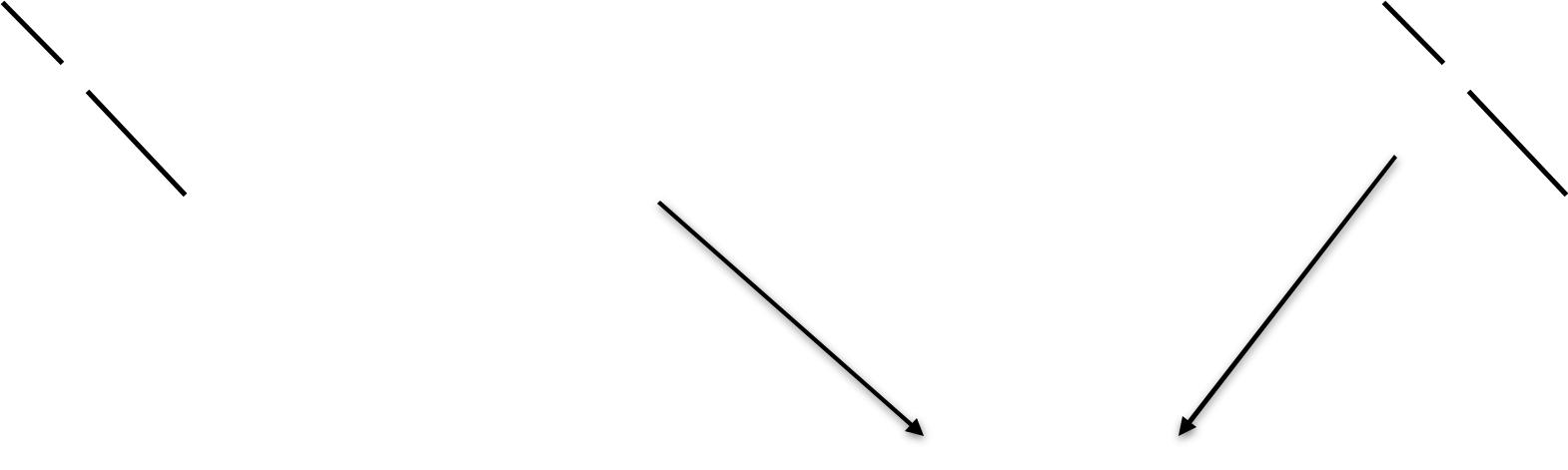


ZIP

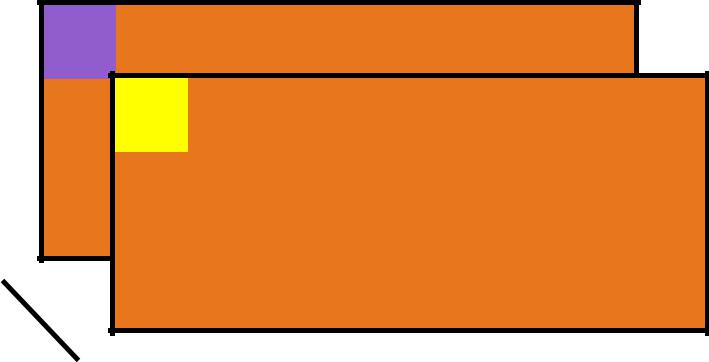
RDD: **x** RDD: **y**



|  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | 3 |  |  |  | | | 9 |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |  |  |  |
|  |  | 2 |  |  |  | |  | 4 |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |  |  |  |
|  |  |  | 1 |  |  |  |  |  | 1 |  |  |  |
|  |  |  |  |  |  |  |  |  |  |  |  |  |
| B | |  |  |  |  | B | |  |  |  |  |  |
|  | A | |  |  |  |  | A | |  |  |  |  |
|  |  |  |  |  |  |  |  |

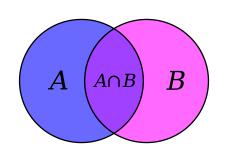


4 RDD: **z**



2 4

1 1



A

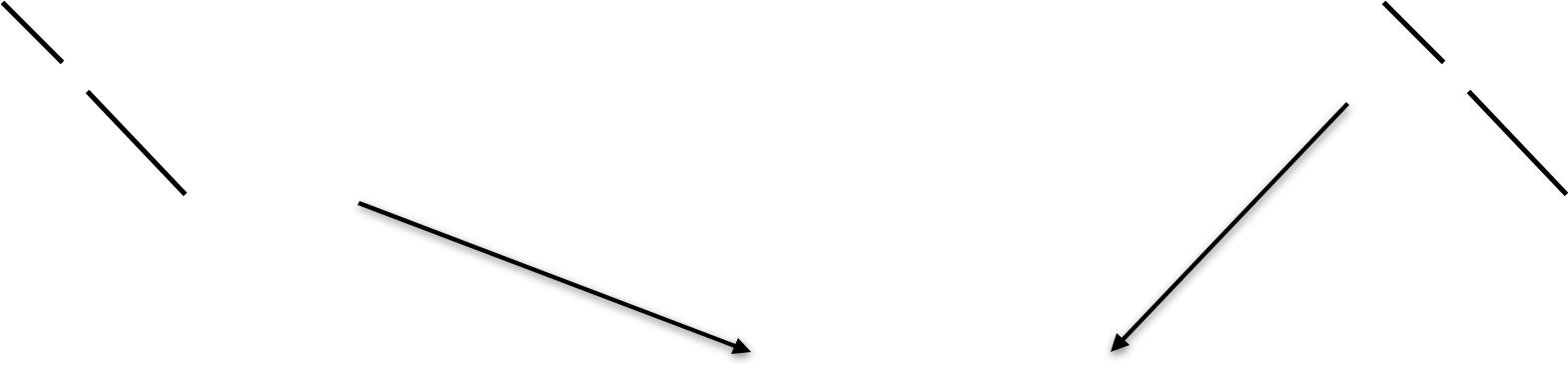


ZIP

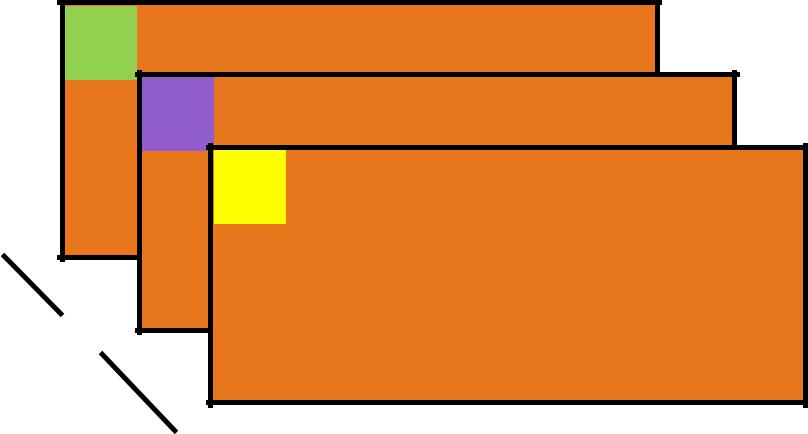
RDD: **x** RDD: **y**



|  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | 3 |  |  |  | | | 9 |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |  |  |  |
|  |  | 2 |  |  |  | |  | 4 |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |  |  |  |
|  |  |  | 1 |  |  |  |  |  | 1 |  |  |  |
|  |  |  |  |  |  |  |  |  |  |  |  |  |
| B | |  |  |  |  | B | |  |  |  |  |  |
|  | A | |  |  |  |  | A | |  |  |  |  |
|  |  |  |  |  |  |  |  |



4 RDD: **z**

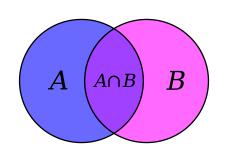


3 9

2 4

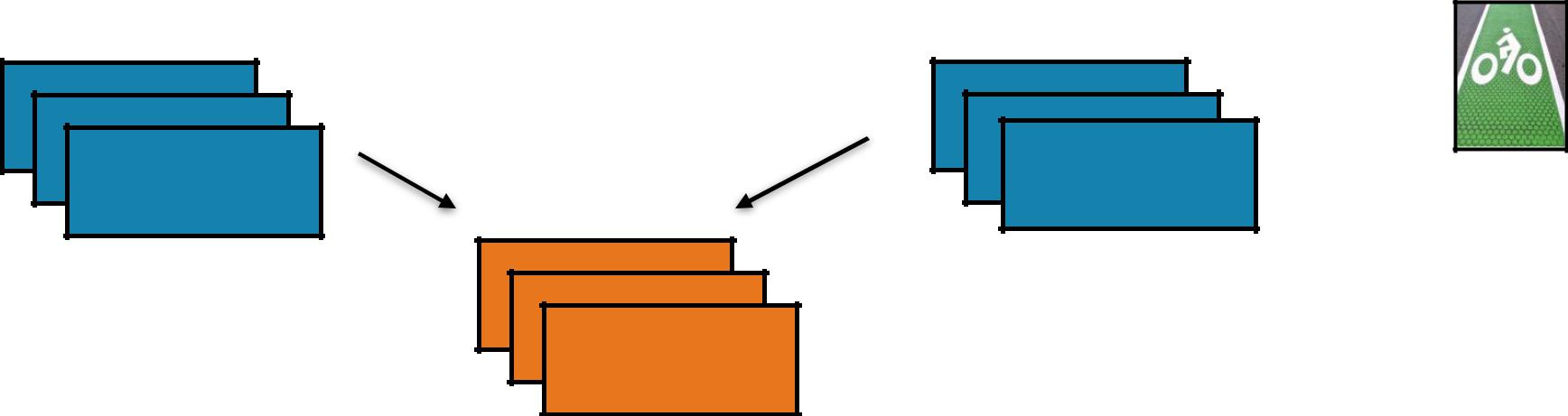
1 1

B



A





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])



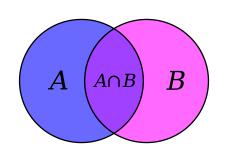
**y** = **x**.map(lambda n:n\*n)

1. = **x**.zip(**y**)

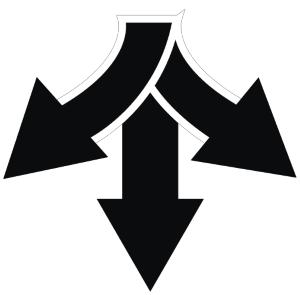
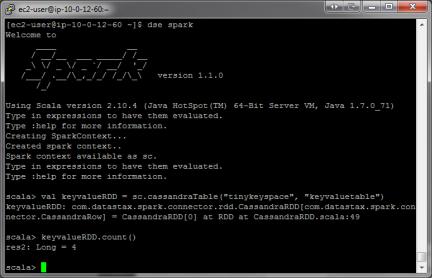
|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| print(**z**.collect()) | | | | **x:** [1, 2, 3] |  |
|  |  |  |  | **y:** [1, 4, 9] |  |
| val **x** = sc.parallelize(Array(1,2,3)) | | | |  |
|  |  |
| val | **y** | = | **x**.map(n=>n\*n) | **z:** [(1, 1), (2, 4), (3, 9)] |  |
| val | **z** | = | **x**.zip(**y**) |  |  |



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



|  |  |  |
| --- | --- | --- |
| ACTIONS | Core Operations |  |
|  |  |



vs

distributed

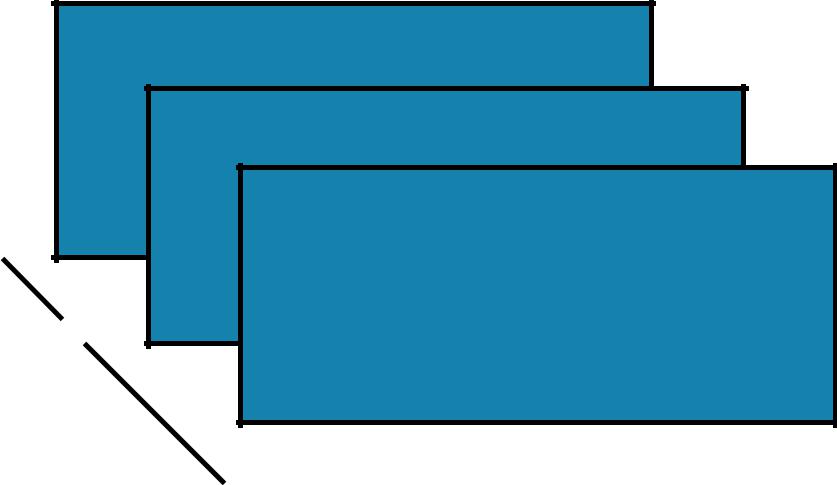
driver

occurs across the cluster



result must fit in driver JVM

GETNUMPARTITIONS



2

B

partition(s) A



GETNUMPARTITIONS B



A

**x** = sc.parallelize([1,2,3], 2)



**y** = **x**.getNumPartitions()

print(**x**.glom().collect())

print(**y**)

val **x** = sc.parallelize(Array(1,2,3), 2)



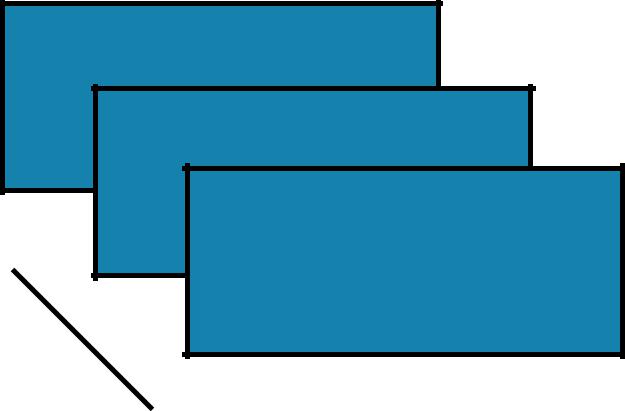
val **y** = **x**.partitions.size

val **xOut** = **x**.glom().collect()

println(**y**)



2



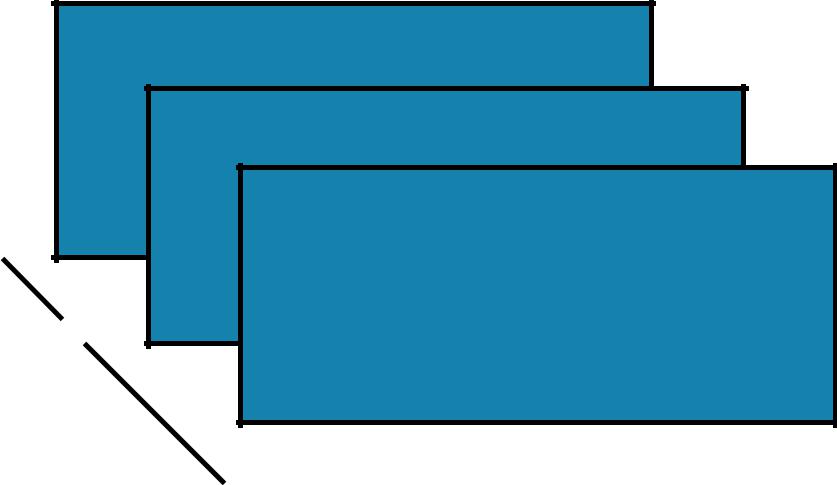
**getNumPartitions()**

Return the number of partitions in RDD

1. [[1], [2, 3]]
   1. 2



COLLECT

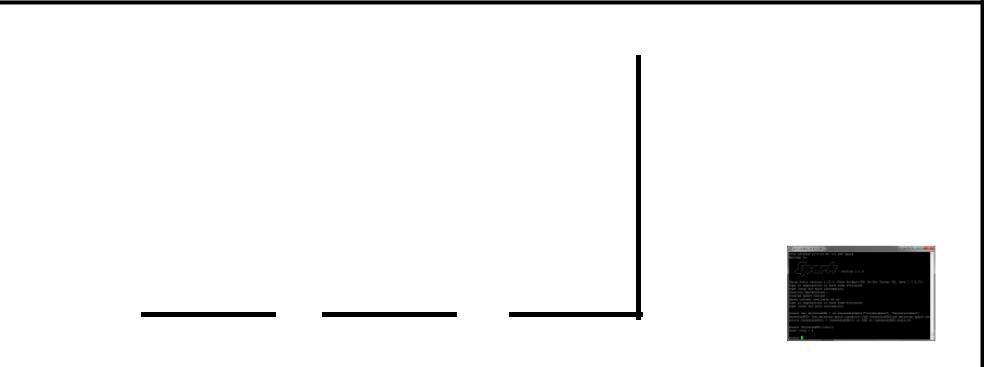
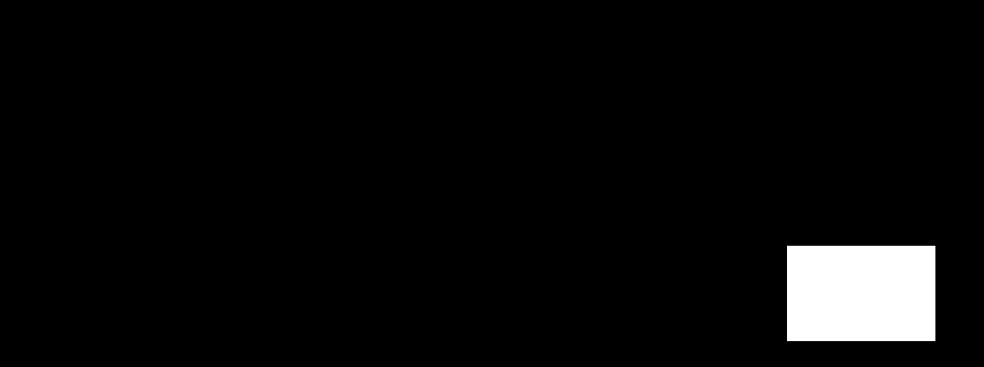


B

partition(s) A

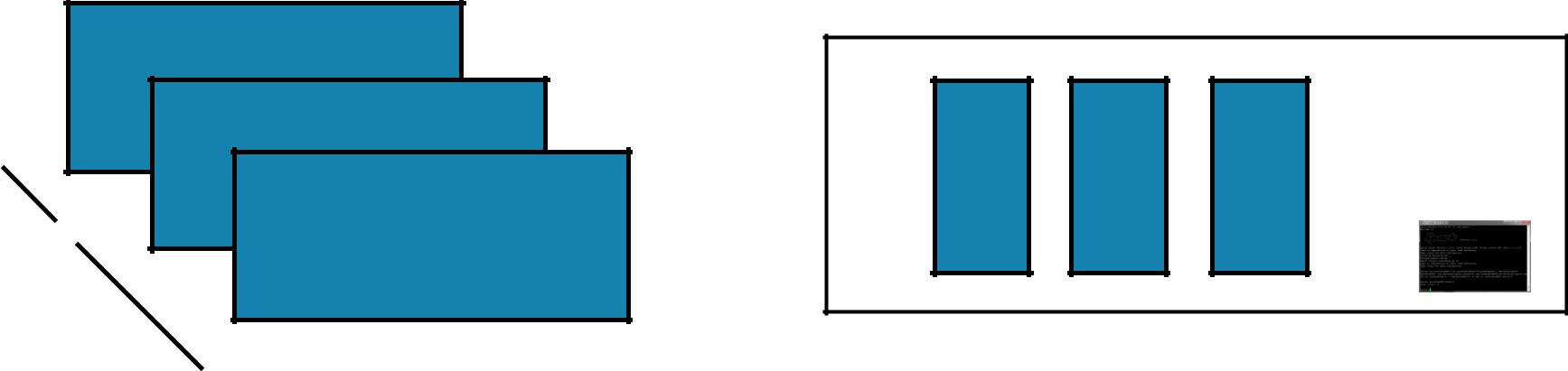


[]



COLLECT

|  |  |  |  |
| --- | --- | --- | --- |
| B | [ | ] |  |
|  |  |  |
|  | A |  |  |



**collect()**

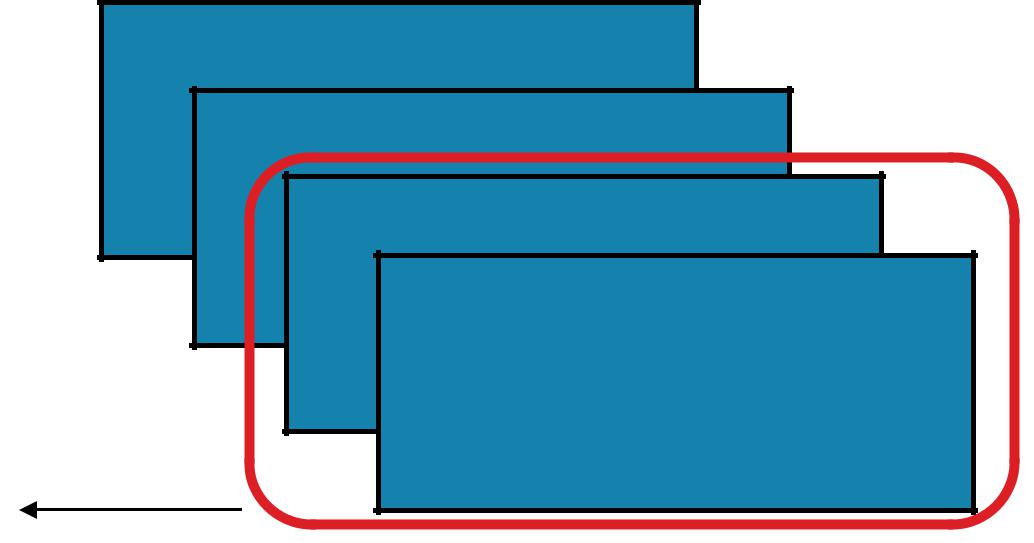
Return all items in the RDD to the driver in a single list

|  |  |  |  |
| --- | --- | --- | --- |
| **x** = | sc.parallelize([1,2,3], 2) |  |  |
| **y** = | **x**.collect() |  |  |
| print(**x**.glom().collect()) | |  |  |
| print(**y**) | |  |  |
|  |  | **x:** [[1], [2, 3]] |  |
|  |  |  |
| val | **x** = sc.parallelize(Array(1,2,3), 2) | **y:** [1, 2, 3] |  |
|  |  |
| val | **y** = **x**.collect() |  |  |
| val | **xOut** = **x**.glom().collect() |  |  |



println(**y**)

REDUCE



4

3



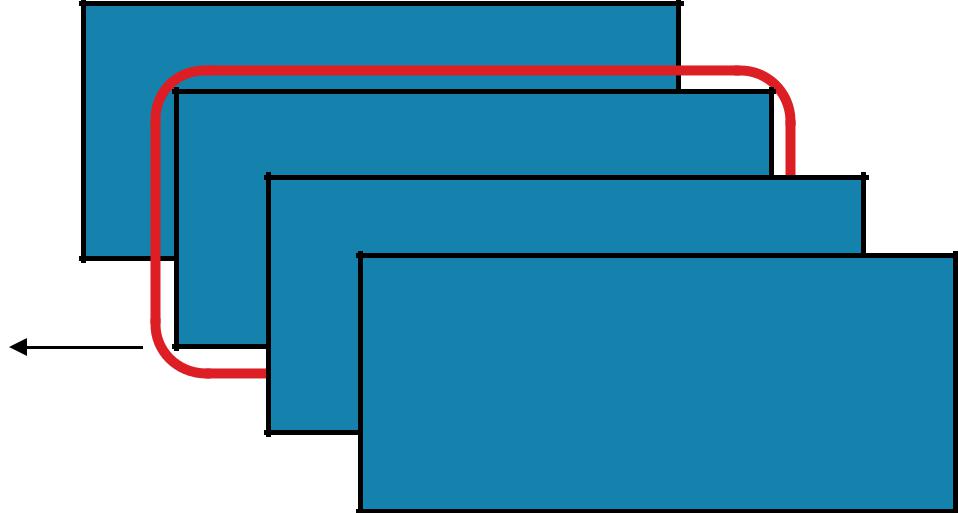
2

1

3 emits



REDUCE



4

3



2

1

6

emits



input

3



REDUCE

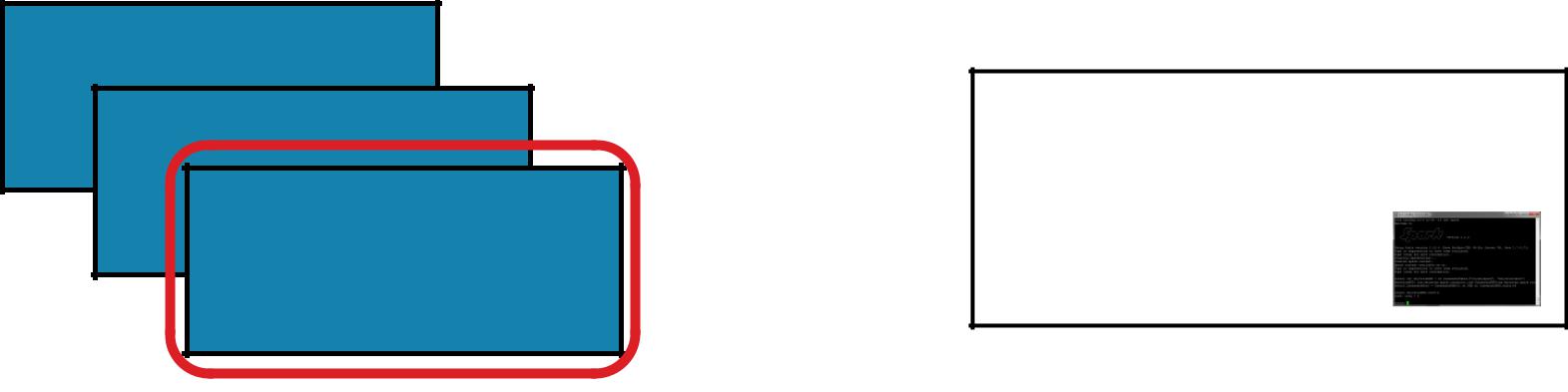


|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | 4 |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |
| 10 |  | 3 |  |  |  |  |  |  |  |
|  |  |  |  |  |  | 10 | |  |
|  |  | 2 |  |  |  |  |
|  |  |  |  |  |  |  |
| input |  |  |  | 1 |  |  |  |  |  |
| 6 |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |



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

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



**y** = x.reduce(lambda a,b: a+b)

print(x.collect())

print(**y**)

val **x** = sc.parallelize(Array(1,2,3,4))



val **y** = **x**.reduce((a,b) => a+b)

println(**x**.collect.mkString(", "))

println(**y**)



**x:**

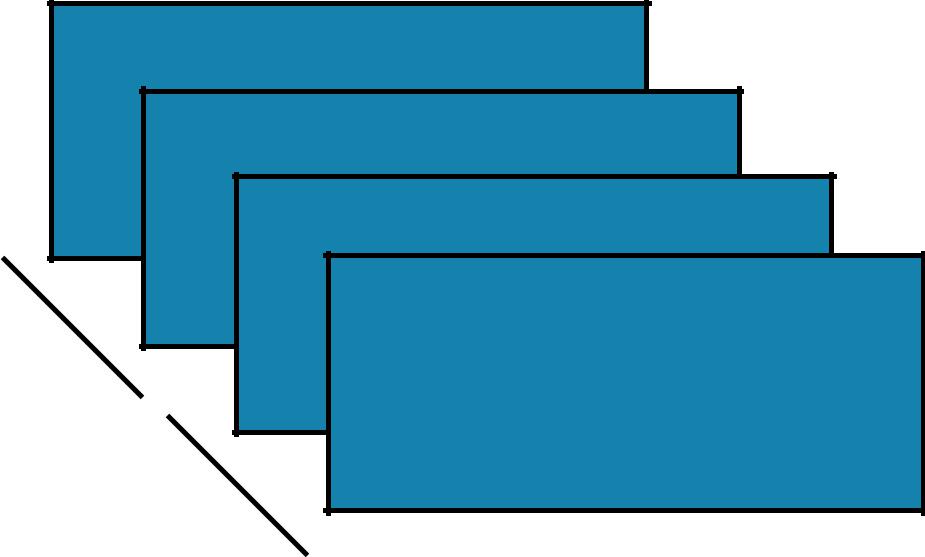


**y:**

[1, 2, 3, 4]

10

AGGREGATE



4

3



2

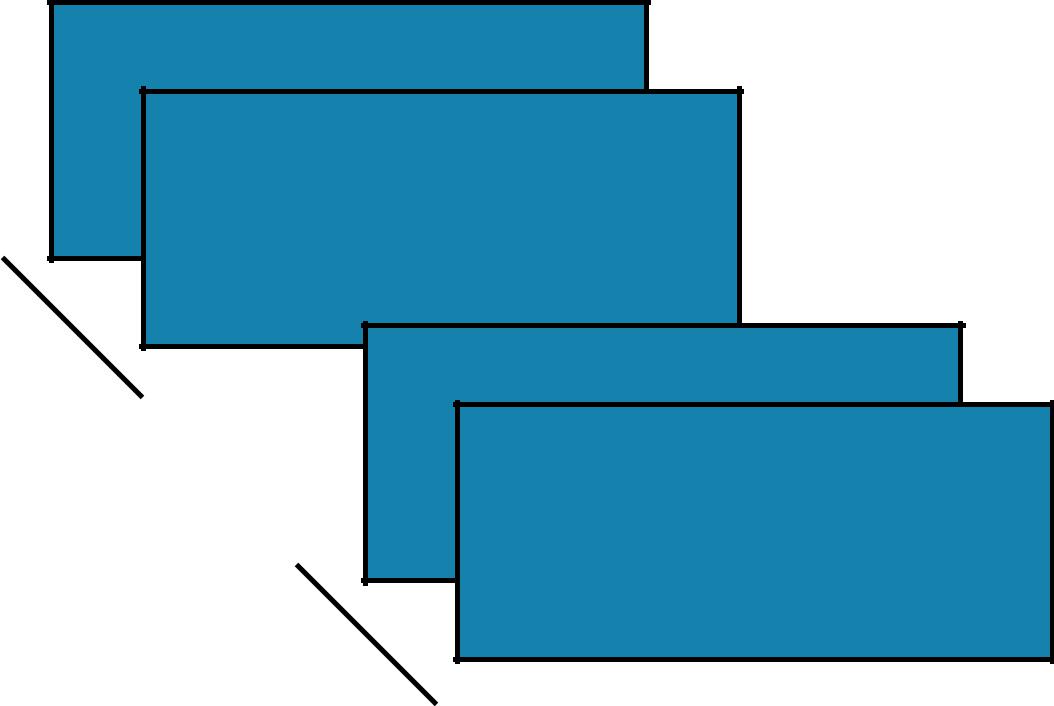
1

B

A



AGGREGATE



4

3

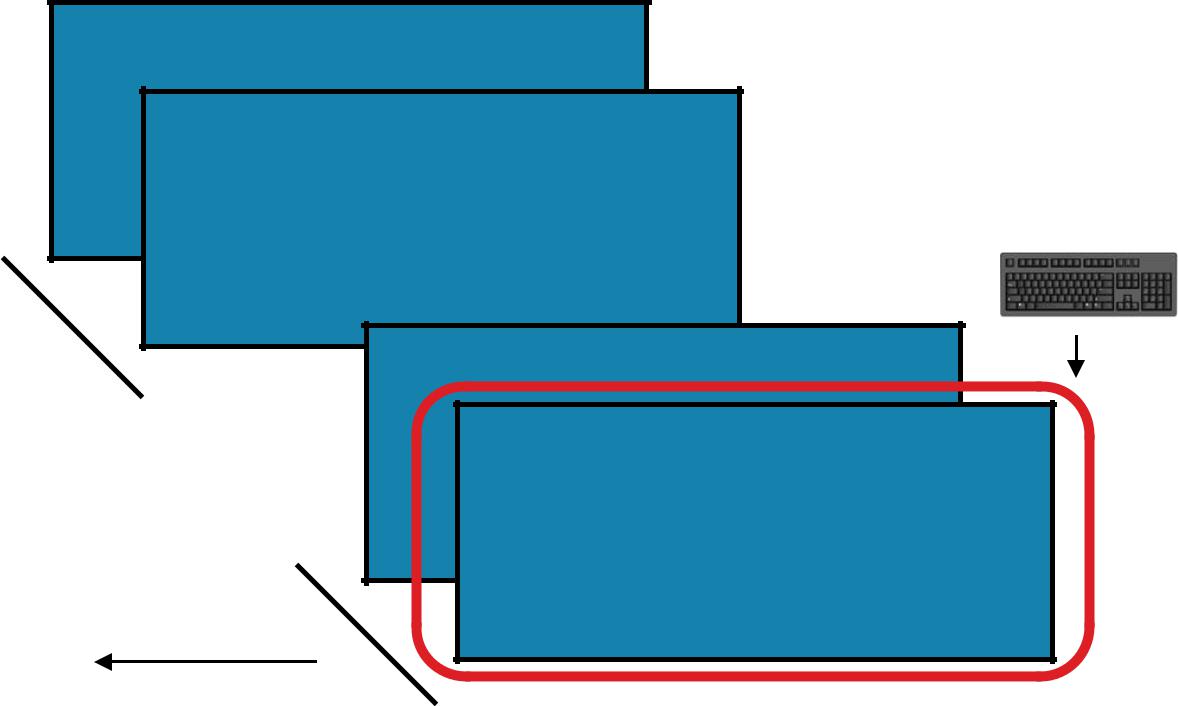


2

1



AGGREGATE



4

3

([1], 1)



([], 0)

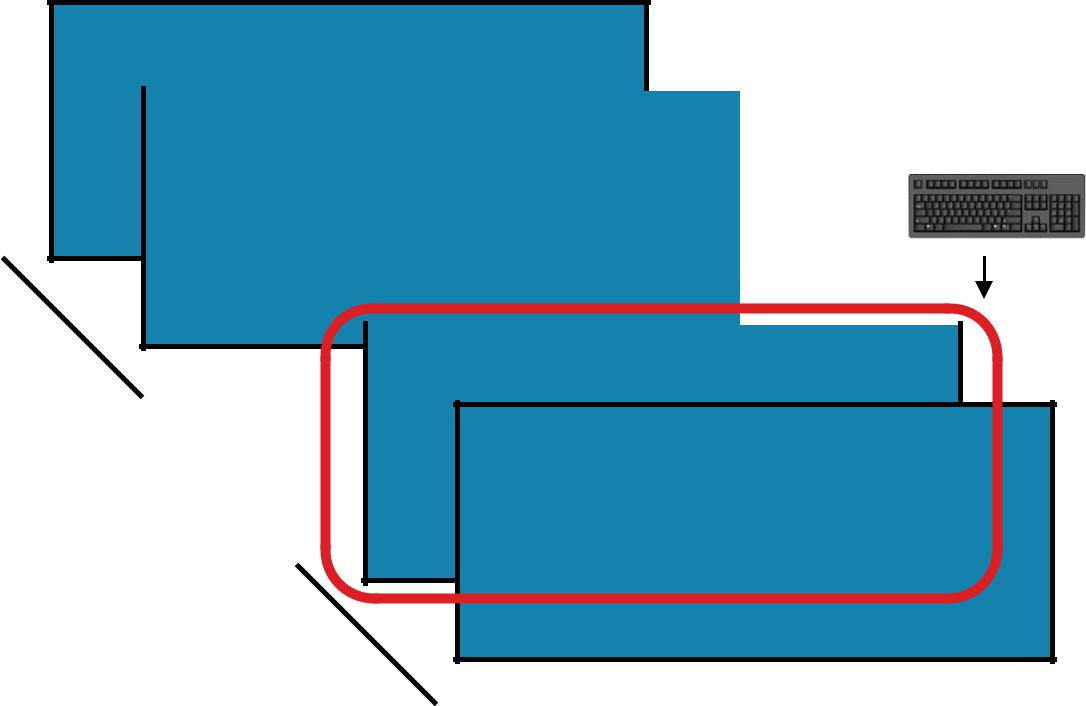


2

1

emits

AGGREGATE



4

|  |  |  |  |
| --- | --- | --- | --- |
| 3 |  | ([], 0) | |
|  |  |  |  |



2

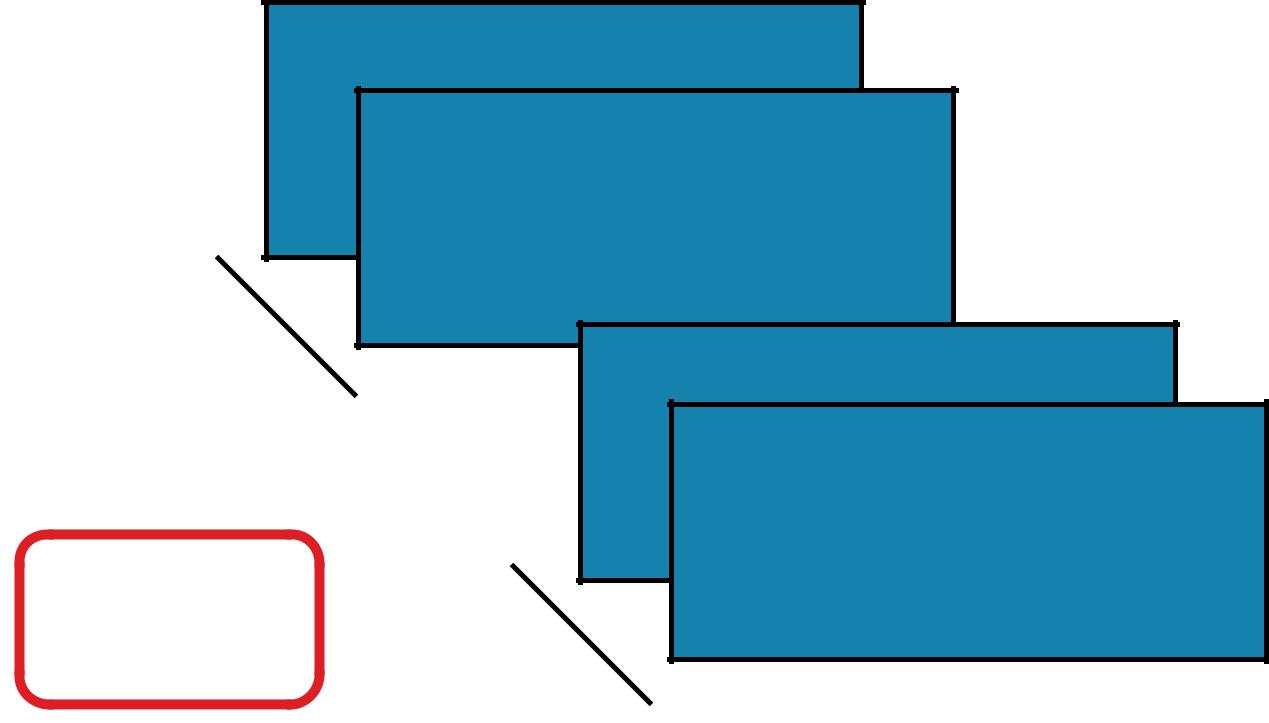
1

([2], 2)

([1], 1)



AGGREGATE



4

3



2

1

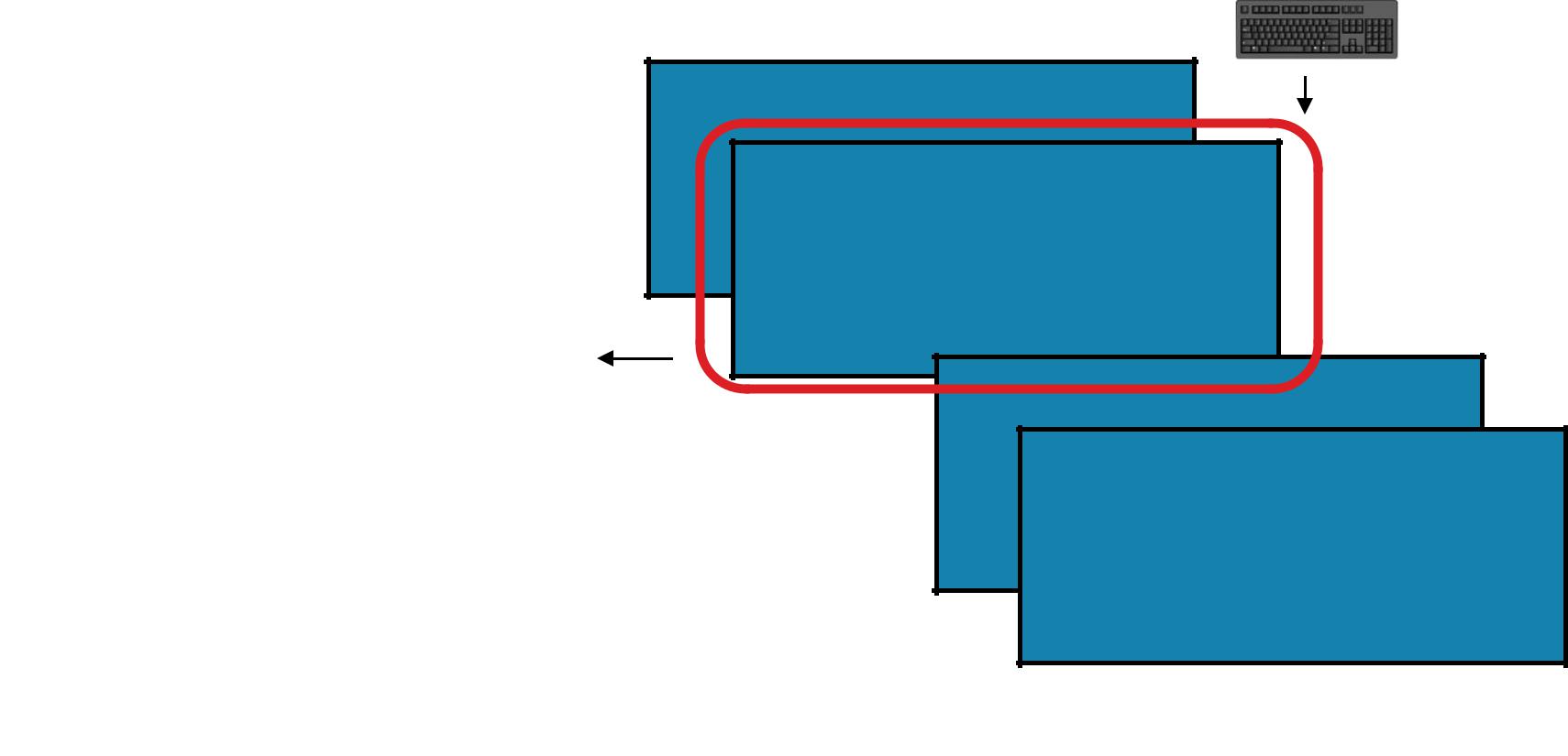
([2], 2)

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



AGGREGATE

([], 0)



4

3



([3], 3)

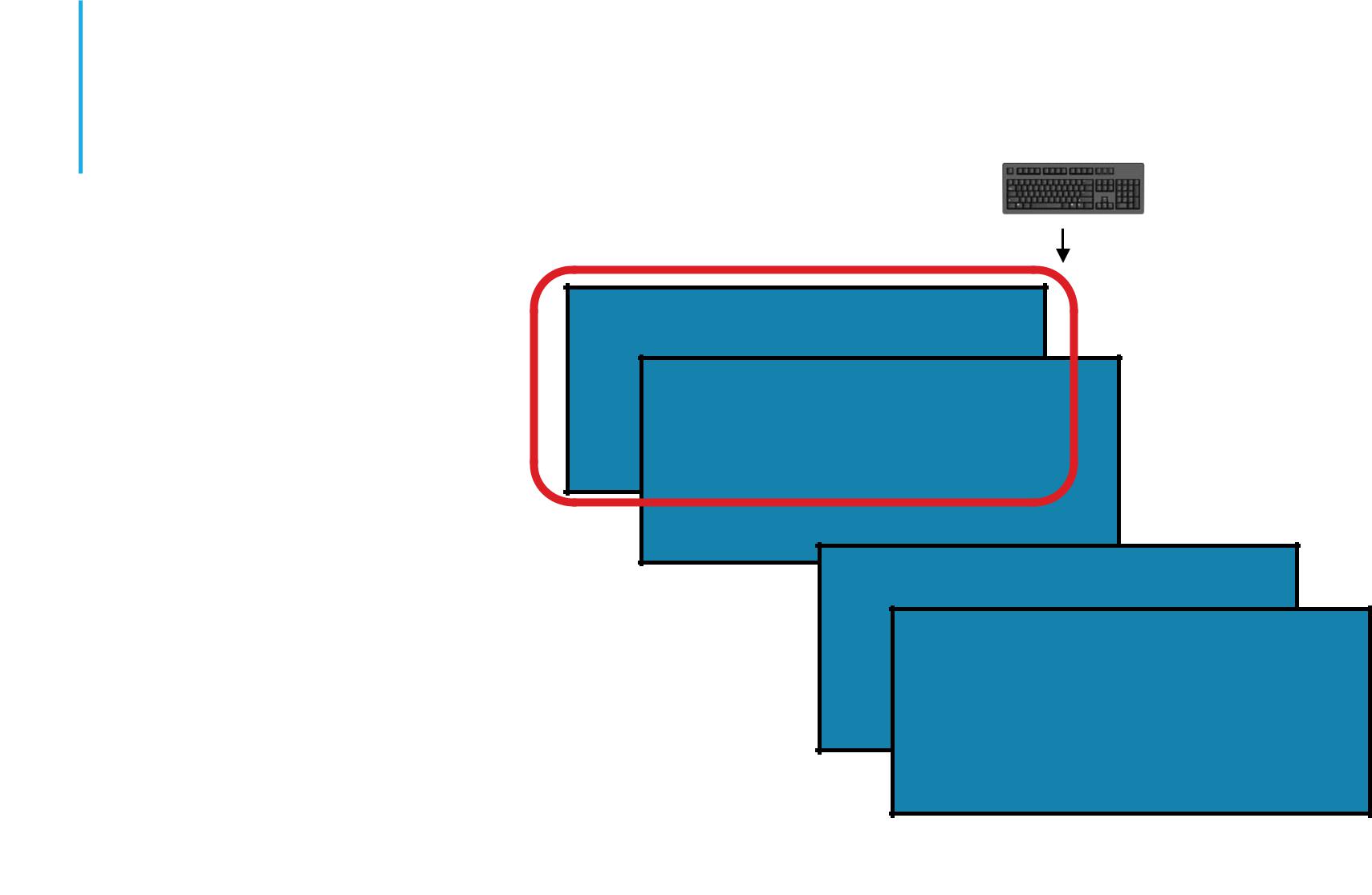
2

1

([2], 2)

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





|  |  |
| --- | --- |
| AGGREGATE | ([], 0) |

4

3

([4], 4)



([3], 3)

2

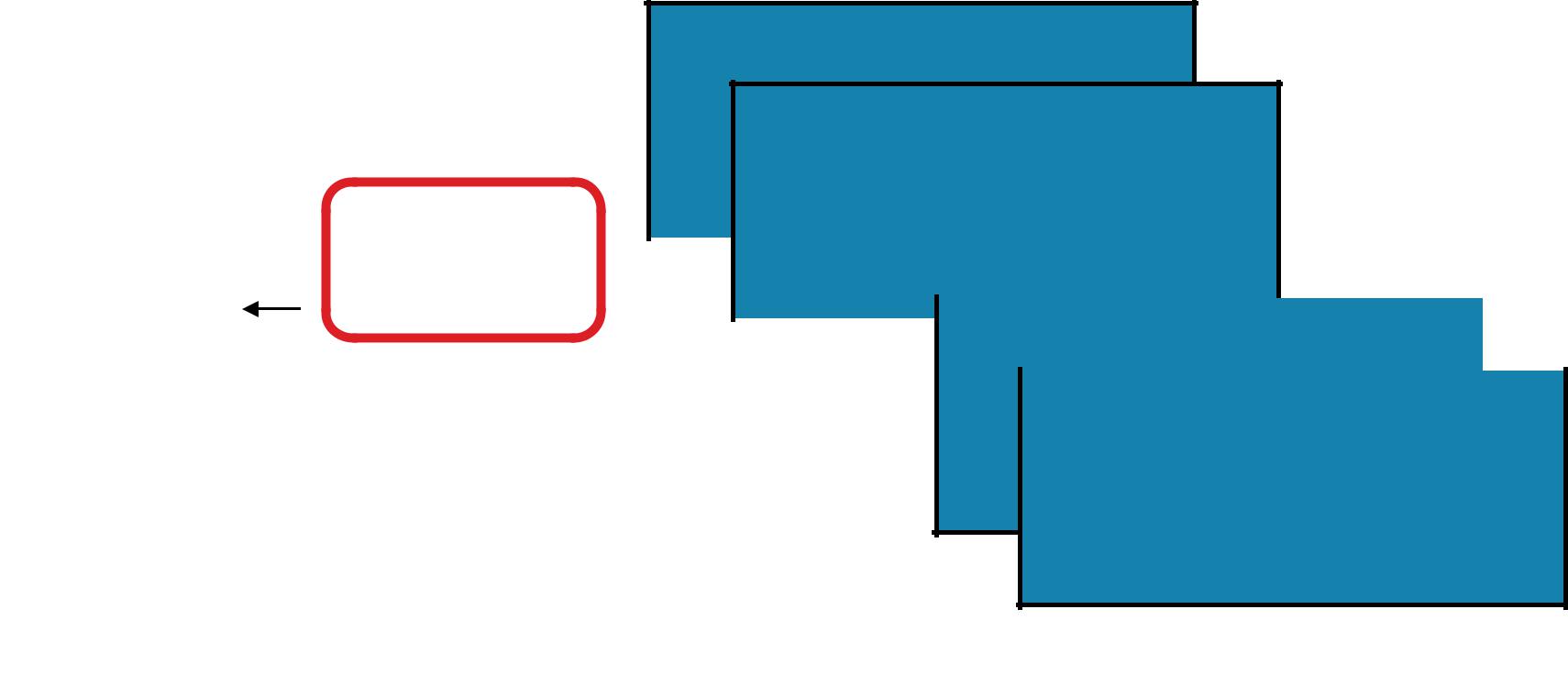
1

([2], 2)

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



AGGREGATE



4

3

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | ([4], 4) |  |  |  |  |  |  |  |  |
| ([3,4], 7) | ([3], 3) |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |
|  | 2 | | | | | |  |  |  |
|  |  |  |  |  |  |  |  |  |  |



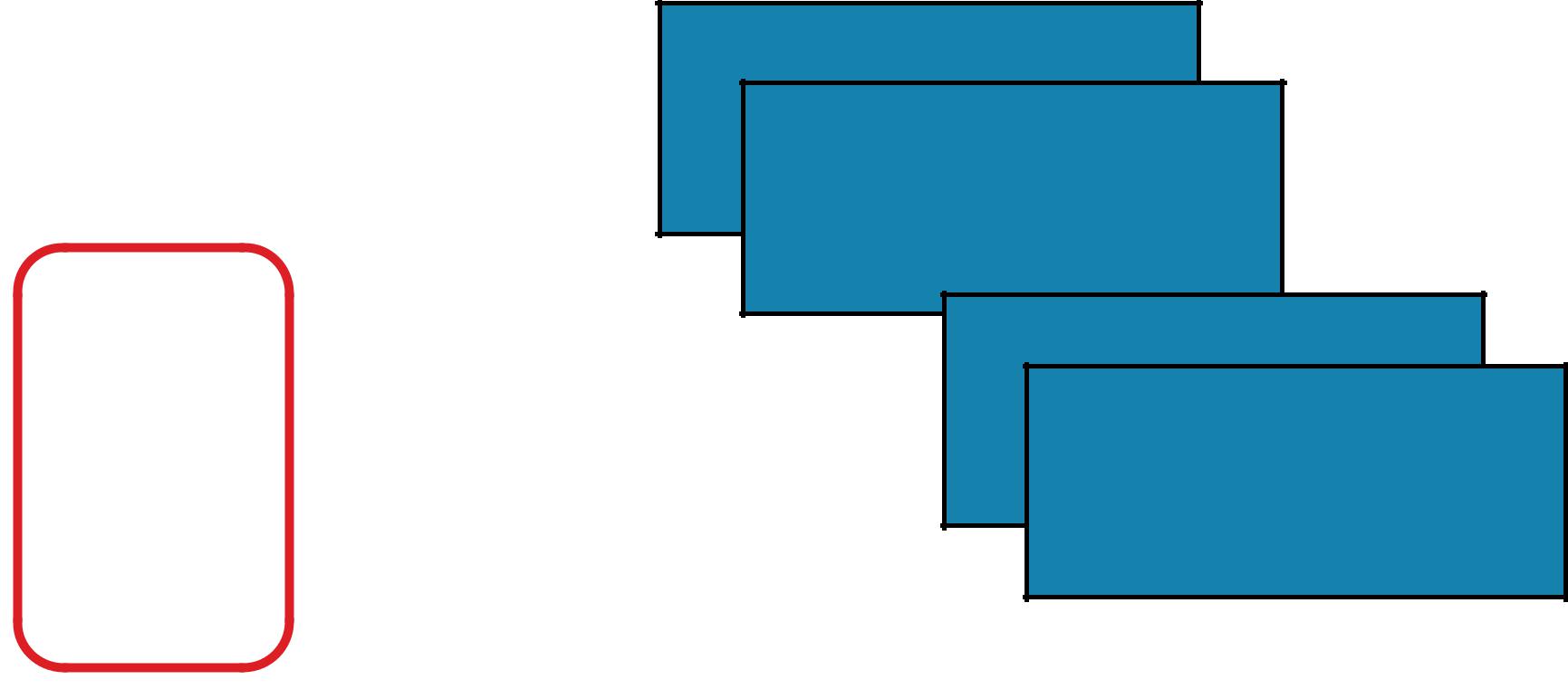
1

([2], 2)

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



AGGREGATE



4

3

|  |  |
| --- | --- |
|  | ([4], 4) |
| ([3,4], 7) | ([3], 3) |
|  | 2 |



1

([2], 2)

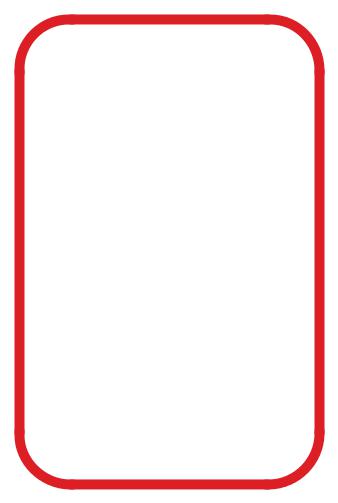
|  |  |
| --- | --- |
| ([1,2], 3) | ([1], 1) |



AGGREGATE

4

3



([3,4], 7)

2

1

([1,2], 3)

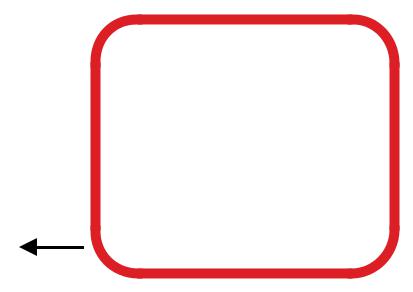


AGGREGATE

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



([3,4], 7)



([1,2], 3)

4

3



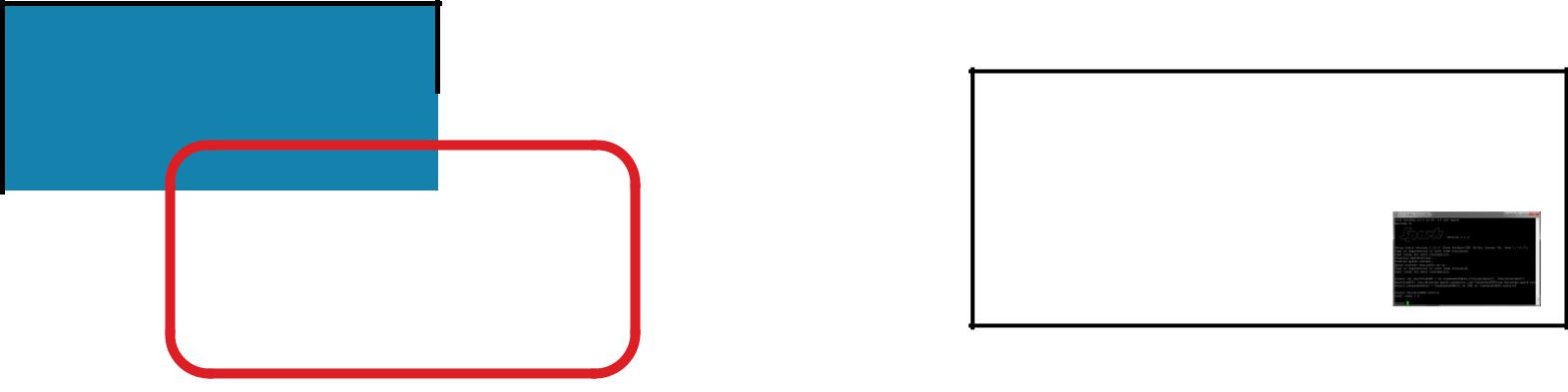
([1,2,3,4], 10)

2

1

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***)

**x:**



**y:**

[1, 2, 3, 4]

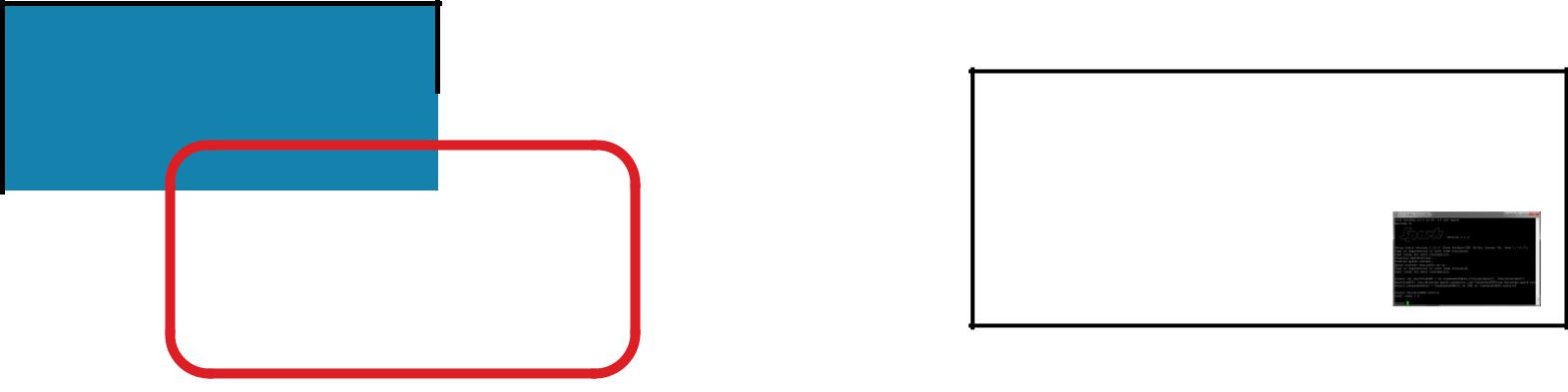
([1, 2, 3, 4], 10)

print(**y**)



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***)



**x:**



**y:**

[1, 2, 3, 4]

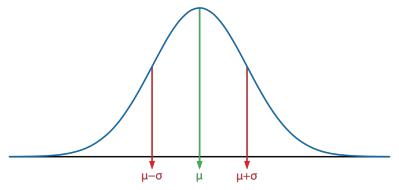
(Array(3, 1, 2, 4),10)

println(**y**)

MAX

1

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | 4 |  |  | 4 |
|  |  | 2 |  |  |
|  |  |  |  |  |
|  |  |  |  |  |
|  |  |  |  |  |



|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  |  |  | *max* |  |
| 4 | |  |
| 1 |  |  |  |  |
|  |  |  |  |  |



4

|  |  |  |
| --- | --- | --- |
| MAX | 2 |  |
|  |  |

**max()**

**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(", "))



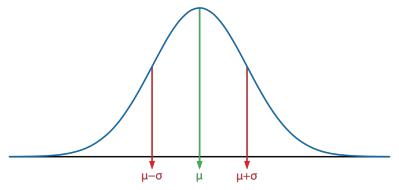
println(**y**)

Return the maximum item in the RDD

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



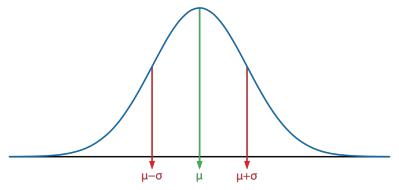
**y:** 4



SUM

1

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | 4 |  |  | 7 |
|  |  | 2 |  |  |
|  |  |  |  |  |
|  |  |  |  |  |
|  |  |  |  |  |



|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  |  |  | Σ |  |
| 4 | |  |
| 1 |  |  |  |  |
|  |  |  |  |  |



7

|  |  |  |
| --- | --- | --- |
| SUM | 2 |  |
|  |  |

**sum()**

**x** = sc.parallelize([2,4,1])



**y** = **x**.sum()

print(**x**.collect())

print(**y**)

val **x** = sc.parallelize(Array(2,4,1))



val **y** = **x**.sum

println(**x**.collect().mkString(", "))



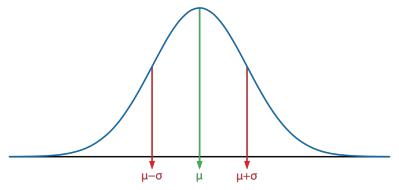
println(**y**)

Return the sum of the items in the RDD

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



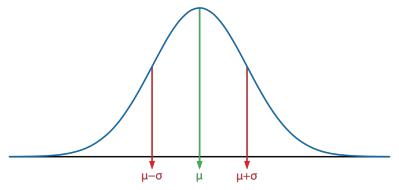
**y:** 7



MEAN

1

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | 4 |  |  | 2.33333333 |
|  |  | 2 |  |  |
|  |  |  |  |  |
|  |  |  |  |  |
|  |  |  |  |  |



|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | 1 |  |  |  |  |  |  |  |  |
|  |  |  |  |  | *~~x~~* | | 2.3333333 |  |
|  |  |  |  |  |  |  |
|  |  | 4 |  |  |  |
| MEAN |  |  | 2 |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |



**mean()**

**x** = sc.parallelize([2,4,1])



**y** = **x**.mean()

print(**x**.collect())

print(**y**)

val **x** = sc.parallelize(Array(2,4,1))



val **y** = **x**.mean

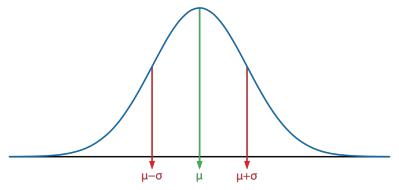
println(**x**.collect().mkString(", "))



println(**y**)

Return the mean of the items in the RDD

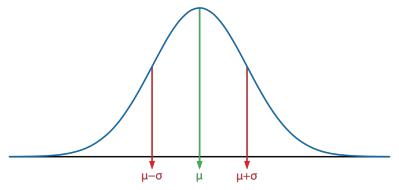
1. [2, 4, 1]
   1. 2.3333333



STDEV

1

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
|  | 4 |  | 1.2472191 | |  |
|  |  |  |  |  |
|  |  | 2 |  |  |  |
|  |  |  |  |  |  |
|  |  |  |  |  |  |
|  |  |  |  |  |  |



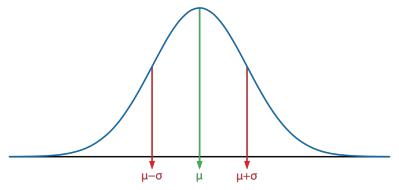
|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  |  |  | σ |  |
| 4 | |  |
| 1 |  |  |  |  |
|  |  |  |  |  |



1.2472191

|  |  |  |
| --- | --- | --- |
| STDEV | 2 |  |
|  |  |

|  |  |  |  |
| --- | --- | --- | --- |
|  |  | **stdev()** |  |
|  | Return the standard deviation of the items in the RDD | |  |
| **x** = sc.parallelize([2,4,1]) |  |  |  |
| **y** = **x**.stdev() |  |  |  |
| print(**x**.collect()) |  |  |  |
| print(**y**) |  |  |  |
|  |  | **x:** [2, 4, 1] |  |
|  |  |  |
| val **x** = sc.parallelize(Array(2,4,1)) |  | **y:** 1.2472191 |  |
|  |  |  |
| val **y** = **x**.stdev |  |  |  |
| println(**x**.collect().mkString(", ")) |  |  |  |
| println(**y**) |  |  |  |

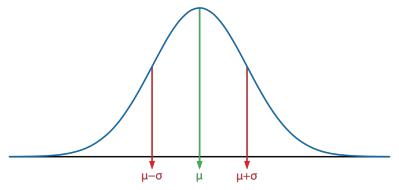


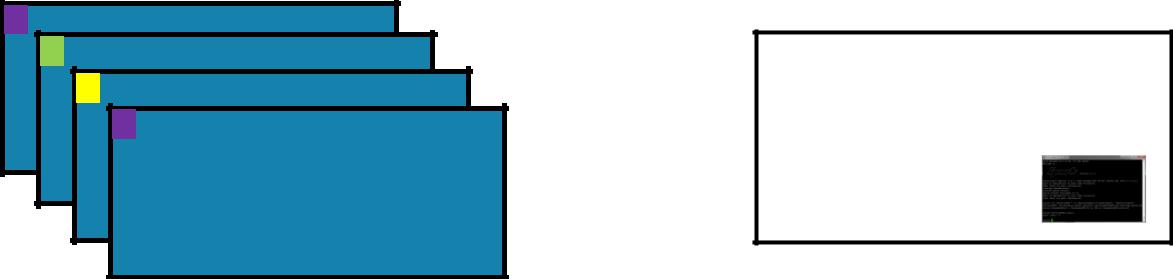
COUNTBYKEY

|  |  |
| --- | --- |
| J “John” |  |
| A “Anna” |  |
| F “Fred” | {'A': 1, 'J': 2, 'F': 1} |



1. “James”



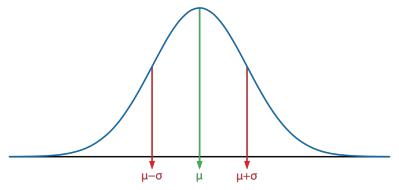
{A: 1, 'J': 2, 'F': 1}

COUNTBYKEY

**countByKey()**

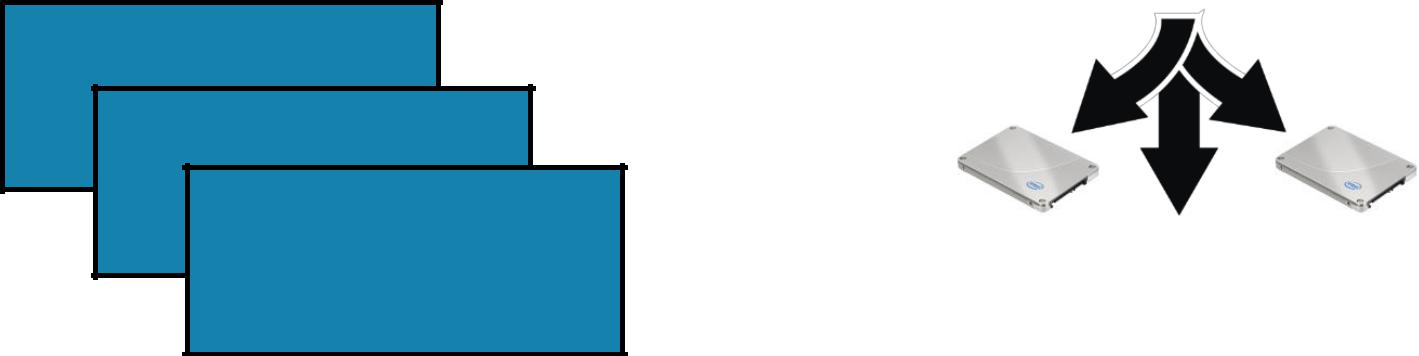
Return a map of keys and counts of their occurrences in the RDD

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **x** = | sc.parallelize([('J', 'James'), ('F','Fred'), | |  |  |  |
|  | ('A','Anna'), ('J','John')]) | |  |  |  |
| **y** = | **x**.countByKey() | |  |  |  |
| print(**y**) | | |  |  |  |
|  |  |  | **x:** [('J', 'James'), ('F','Fred'), | |  |
|  |  |  |  |
|  |  |  |  | ('A','Anna'), ('J','John')] |  |
| val | **x** = sc.parallelize(Array(('J',"James"),('F',"Fred"), | | **y:** | {'A': 1, 'J': 2, 'F': 1} |  |
|  | ('A',"Anna"),('J',"John"))) | |  |
|  |  |  |  |
| val | **y** = **x**.countByKey() | |  |  |  |
| println(**y**) | | |  |  |  |



SAVEASTEXTFILE





SAVEASTEXTFILE



**saveAsTextFile(*path*, *compressionCodecClass=None*)**

Save the RDD to the filesystem indicated in the path

|  |  |  |
| --- | --- | --- |
| dbutils.fs.rm("/temp/demo", True) |  |  |
| **x** = sc.parallelize([2,4,1]) |  |  |
| **x**.saveAsTextFile("/temp/demo") |  |  |
| **y** = sc.textFile("/temp/demo") |  |  |
| print(**y**.collect()) | **x:** [2, 4, 1] |  |
|  |  |
|  | **y:** [u'2', u'4', u'1'] |  |
| 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(", ")) |  |  |



Reference from: http://training.databricks.com/visualapi.pdf