

TRANSFORMATIONS AND ACTIONS

<http://training.databricks.com/visualapi.pdf>



A Visual Guide of the API



[LinkedIn](#)

Blog: [data-frack](#)

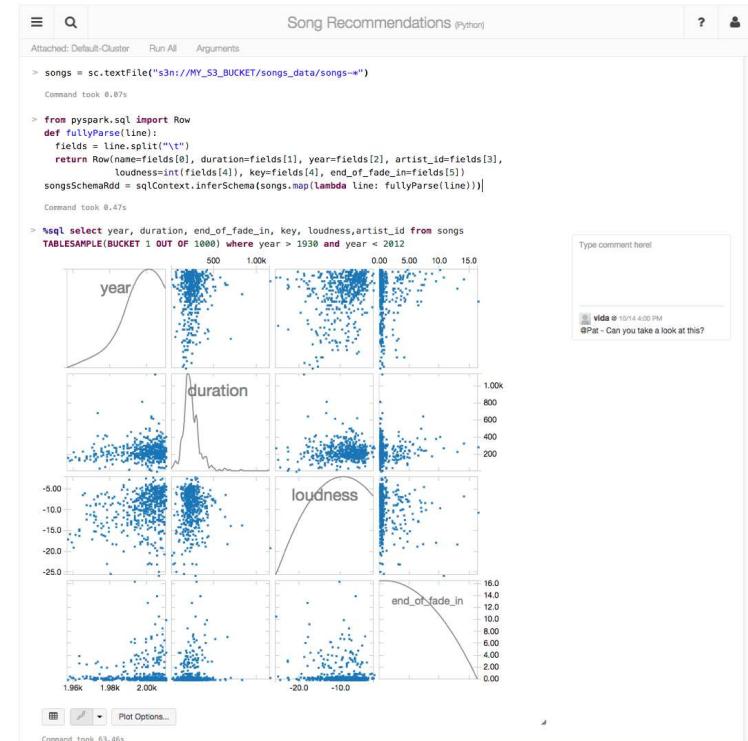
Databricks would like to give a special thanks to Jeff Thompson for contributing 67 visual diagrams depicting the Spark API under the MIT license to the Spark community.

Jeff's original, creative work can be found [here](#) and you can read more about Jeff's project in his [blog post](#).

After talking to Jeff, Databricks commissioned [Adam Breindel](#) to further evolve Jeff's work into the diagrams you see in this deck.



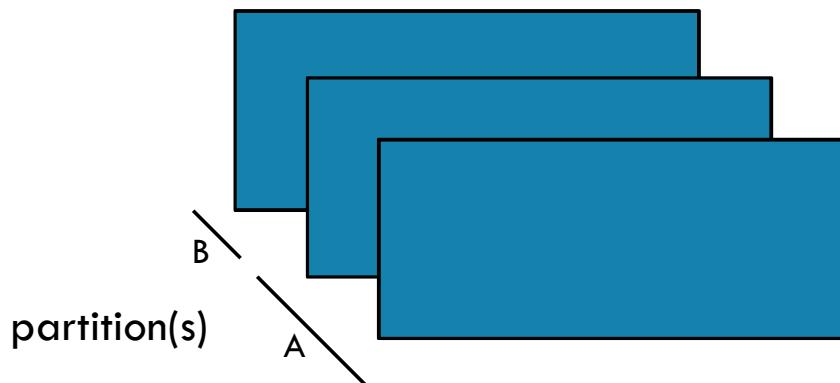
- Founded in late 2013
- by the creators of Apache Spark
- Original team from UC Berkeley AMPLab
- Raised \$47 Million in 2 rounds
- ~55 employees
- We're hiring! (<http://databricks.workable.com>)
- Level 2/3 support partnerships with
 - Hortonworks
 - MapR
 - DataStax



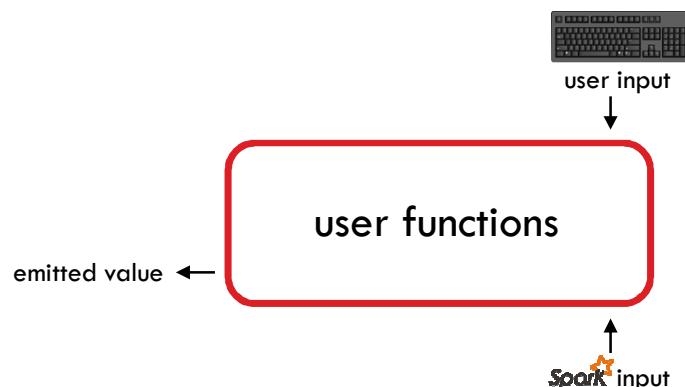
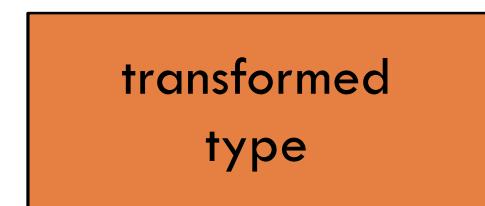
Databricks Cloud:

“A unified platform for building Big Data pipelines – from ETL to Exploration and Dashboards, to Advanced Analytics and Data Products.”

RDD



RDD Elements

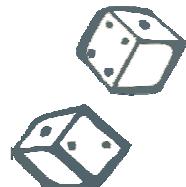


object on driver

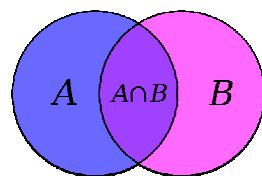




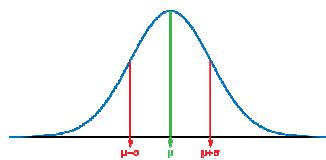
Legend



Randomized operation



Set Theory / Relational operation



Numeric calculation

 **Spark** Operations =

+



TRANSFORMATIONS



ACTIONS



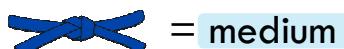
TRANSFORMATIONS



ACTIONS



= easy



= medium

Essential Core & Intermediate Spark Operations

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

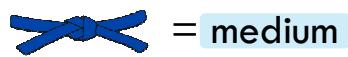
- count
- takeSample
- max
- min
- sum
- histogram
- mean
- variance
- stdev
- sampleVariance
- countApprox
- countApproxDistinct

- takeOrdered

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



= easy



= medium

Essential Core & Intermediate PairRDD Operations

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

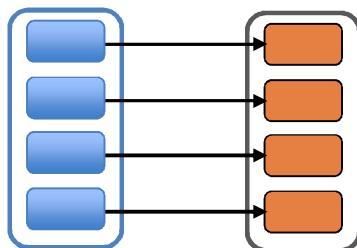
- countByKey
- countByValue
- countByValueApprox
- countApproxDistinctByKey
- countApproxDistinctByKey
- countByKeyApprox
- sampleByKeyExact



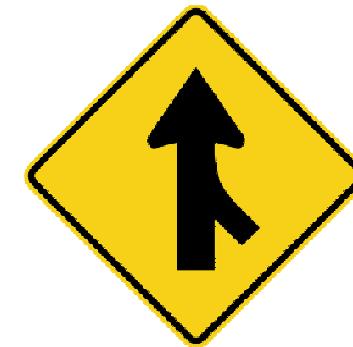


narrow

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

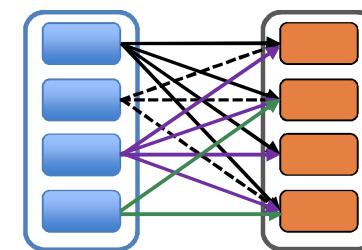


vs



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.”

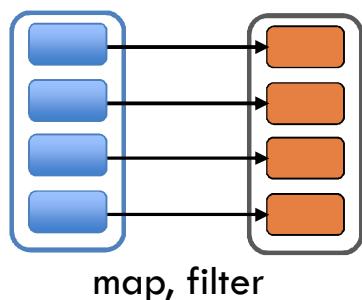
“We found it both sufficient and useful to classify dependencies into two types:

- **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.”

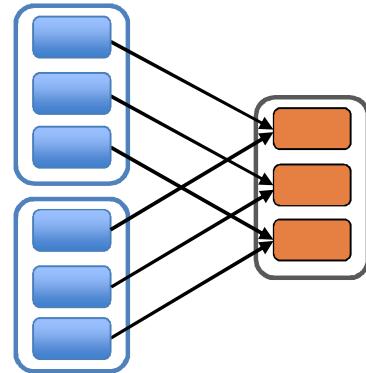


narrow

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

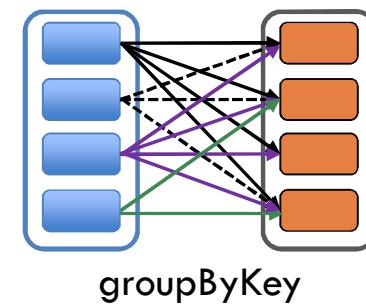


join w/ inputs co-partitioned

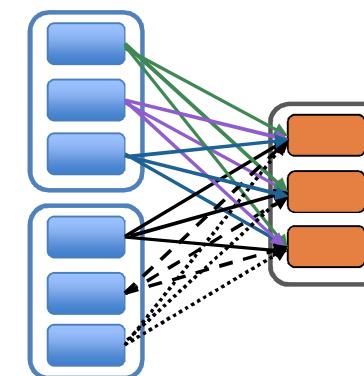


wide

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

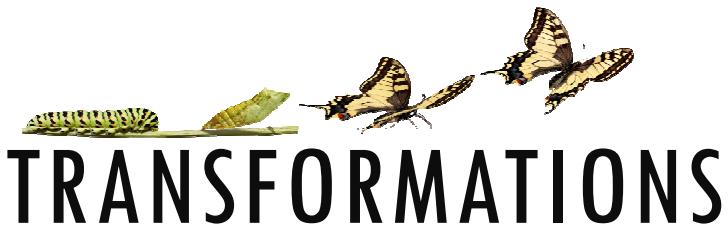


groupByKey

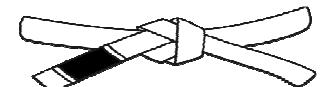


join w/ inputs not co-partitioned





TRANSFORMATIONS

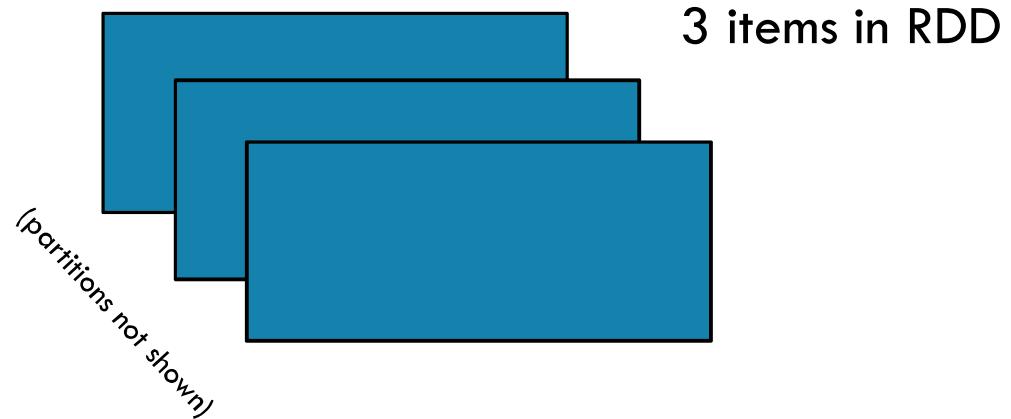


Core Operations



MAP

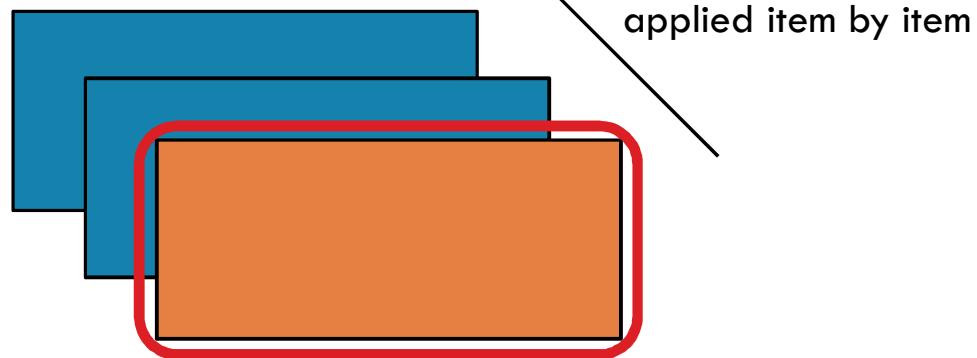
RDD: **x**



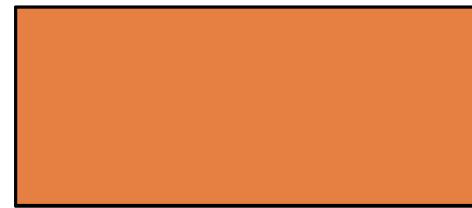


MAP

RDD: **x**



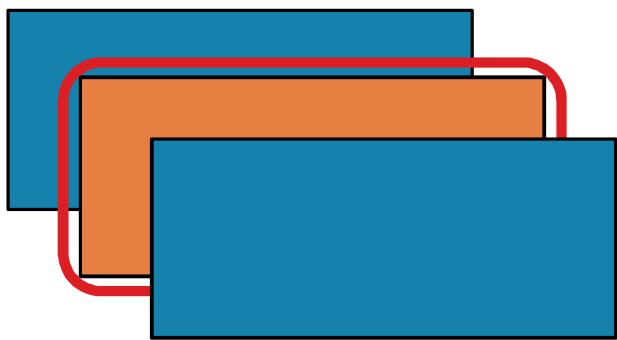
RDD: **y**





MAP

RDD: **x**



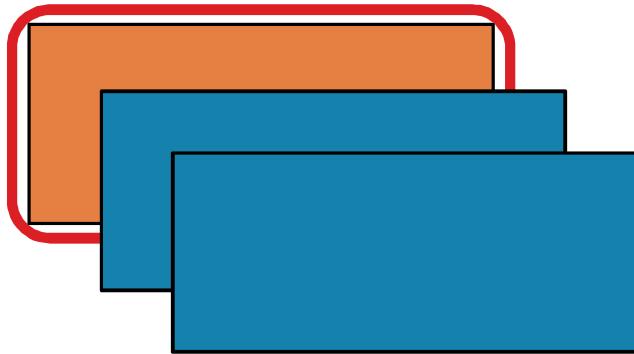
RDD: **y**





MAP

RDD: x



RDD: y

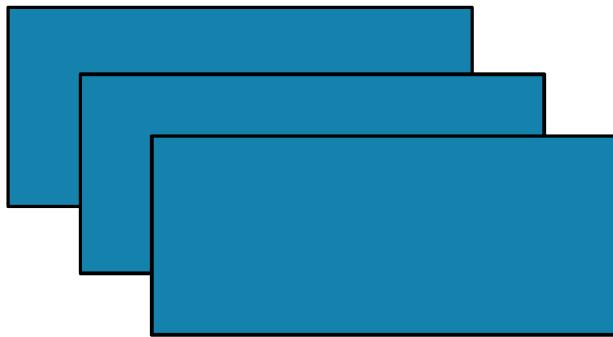




MAP

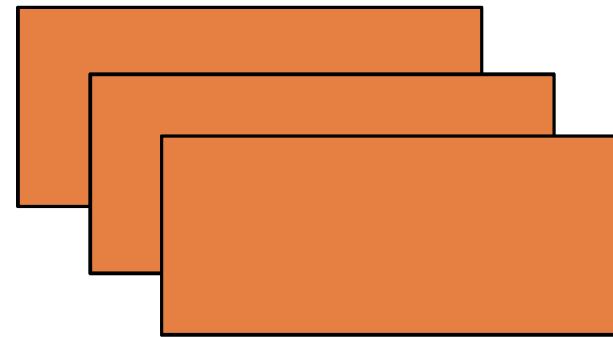
After `map()` has been applied...

RDD: `x`



before

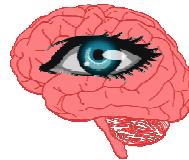
RDD: `y`



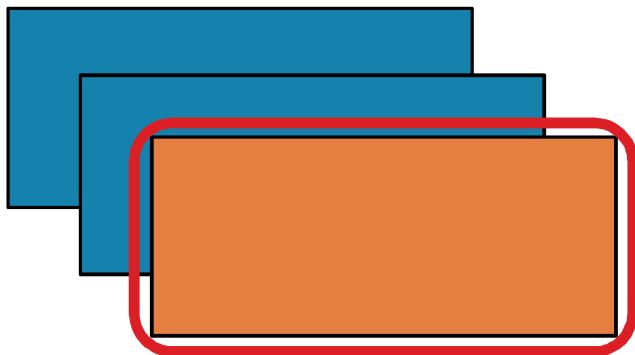
after



MAP



RDD: **x**



RDD: **y**



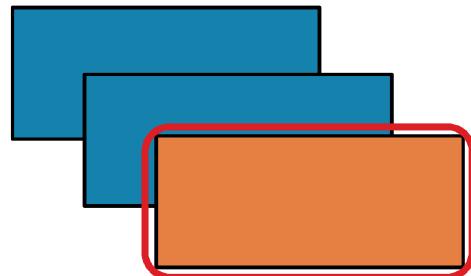
Return a new RDD by applying a function to each element of this RDD.



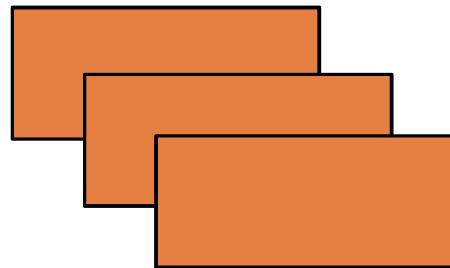


MAP

RDD: **x**



RDD: **y**



`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())
```



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(", "))
```

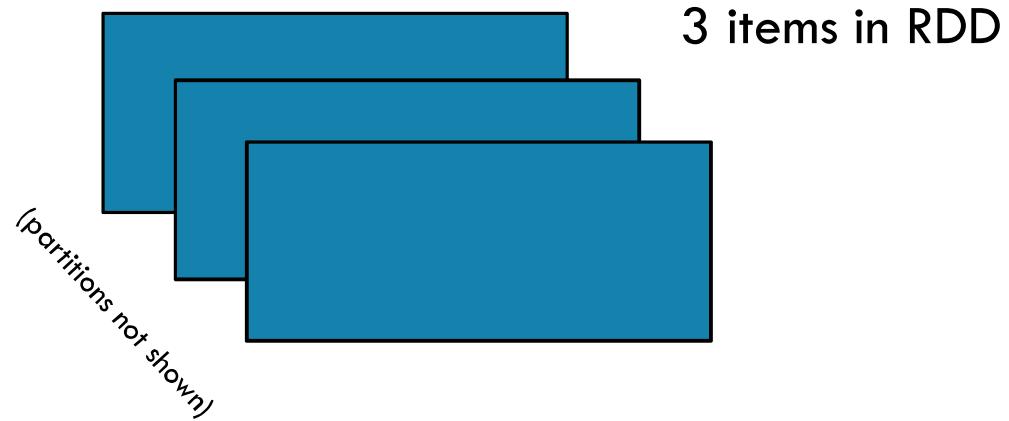
b, a, c
(b,1), (a,1), (c,1)





FILTER

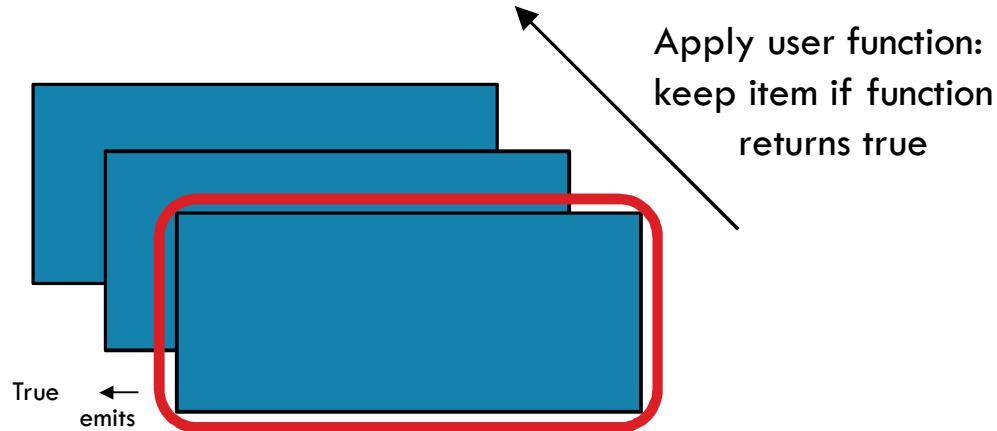
RDD: `x`





FILTER

RDD: x



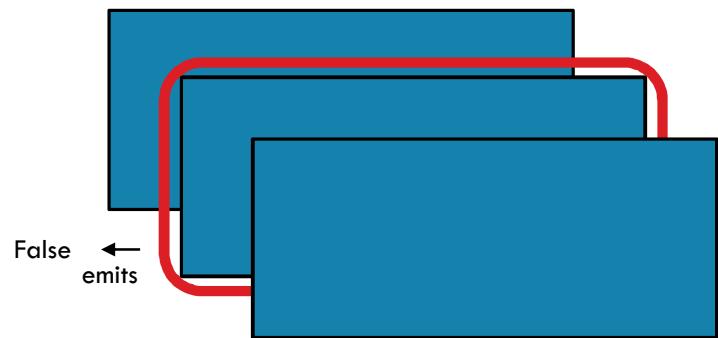
RDD: y



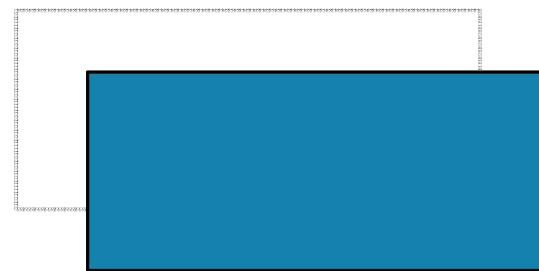


FILTER

RDD: x



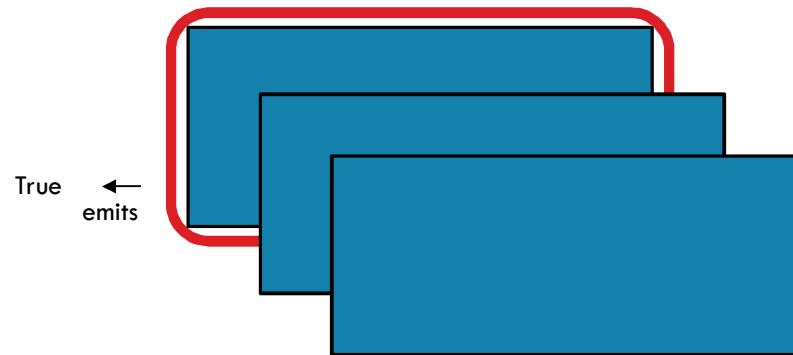
RDD: y



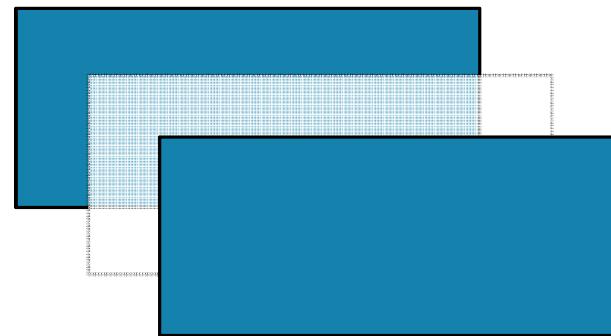


FILTER

RDD: x



RDD: y

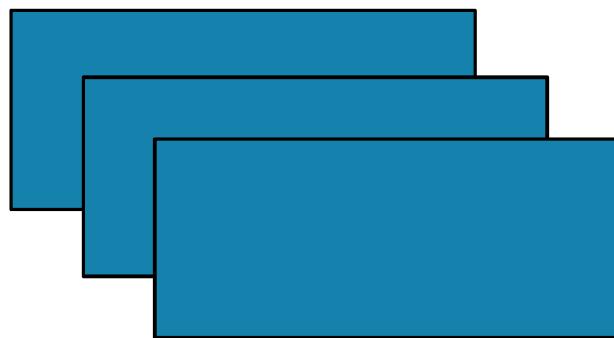




FILTER

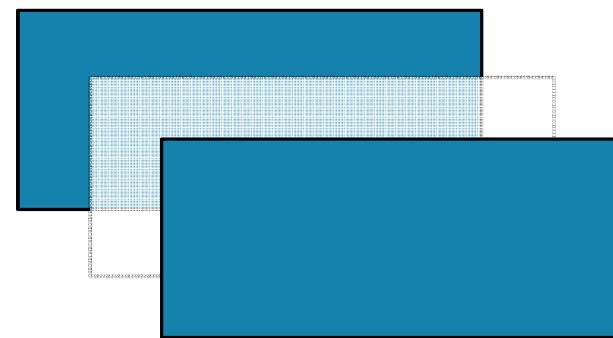
After filter() has been applied...

RDD: **x**



before

RDD: **y**

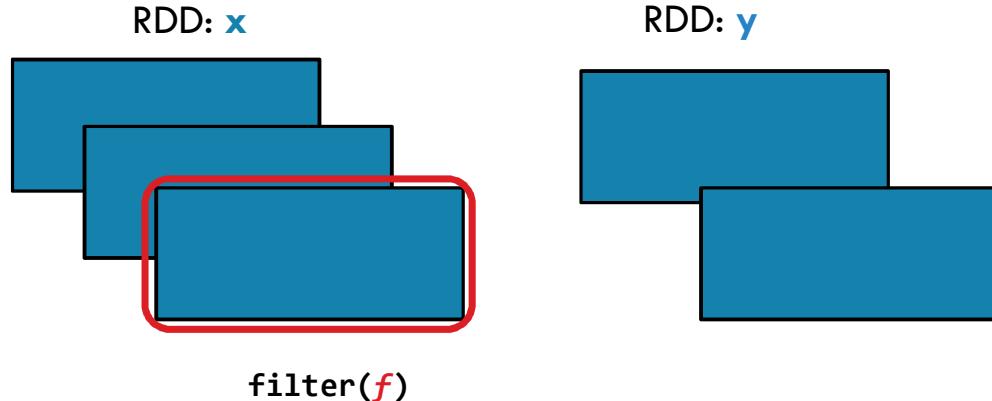


after





FILTER



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



```
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(", "))
```

1, 2, 3

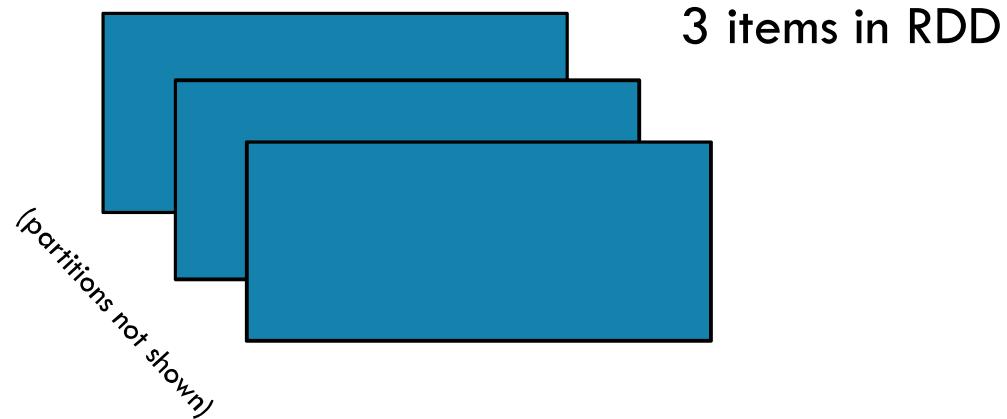
1, 3





FLATMAP

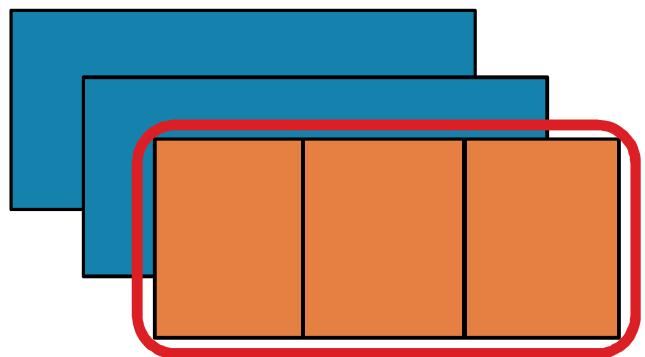
RDD: `x`



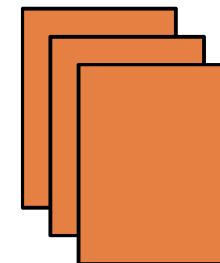


FLATMAP

RDD: `x`



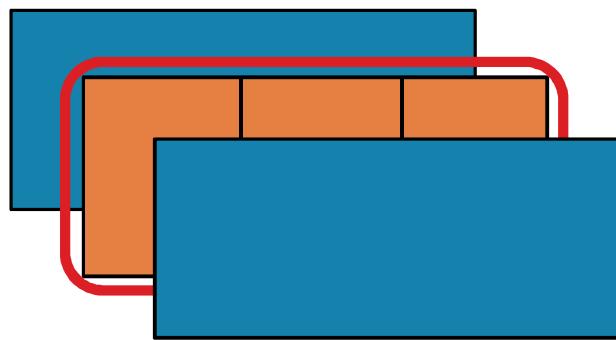
RDD: `y`



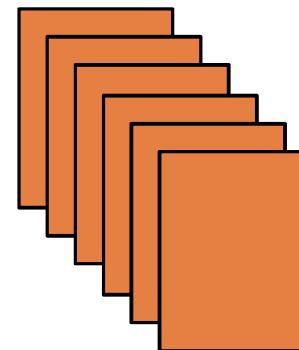


FLATMAP

RDD: `x`



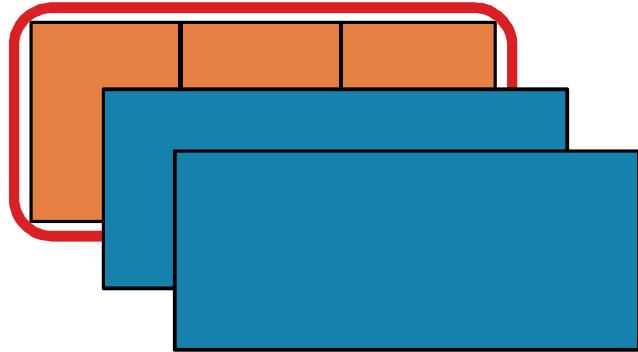
RDD: `y`



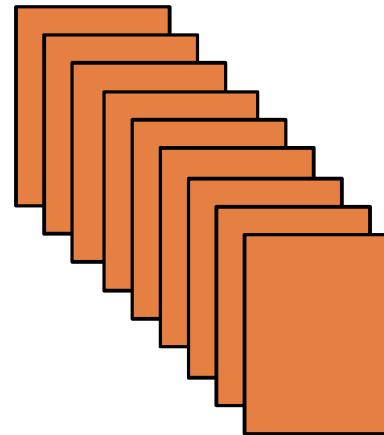


FLATMAP

RDD: `x`



RDD: `y`





FLATMAP

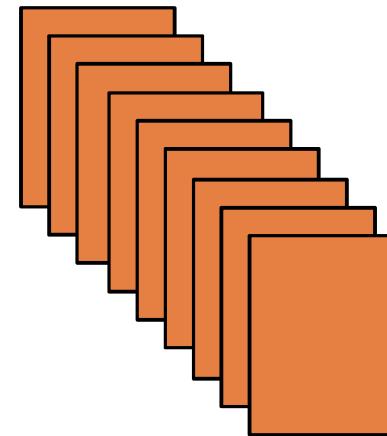
After flatmap() has been applied...

RDD: **x**



before

RDD: **y**

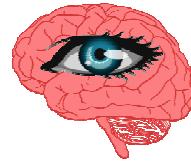


after





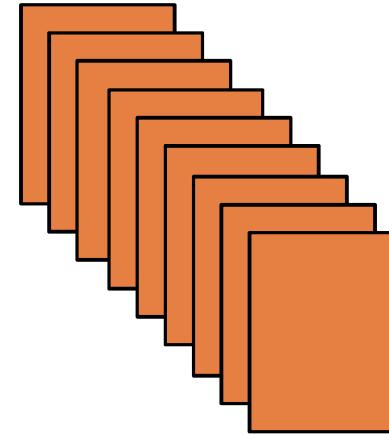
FLATMAP



RDD: **x**



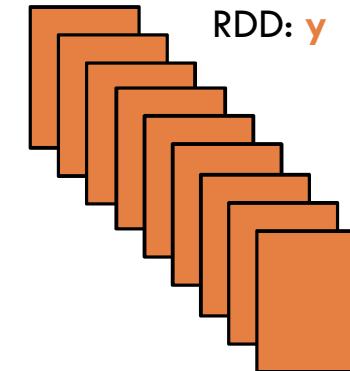
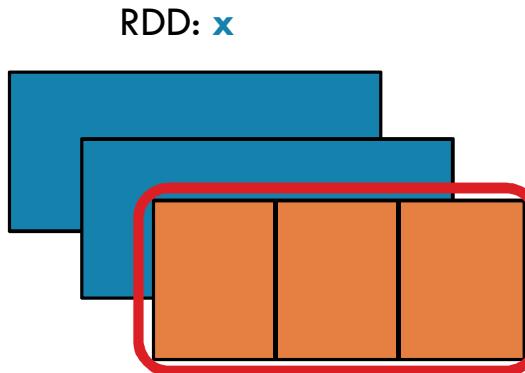
RDD: **y**



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())
```



`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(", "))
```

`1, 2, 3`

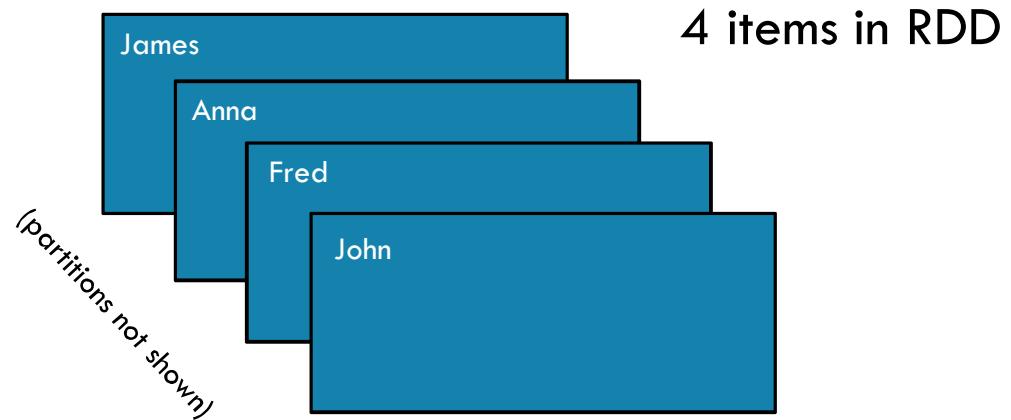
`1, 100, 42, 2, 200, 42, 3, 300, 42`





GROUPBY

RDD: `x`

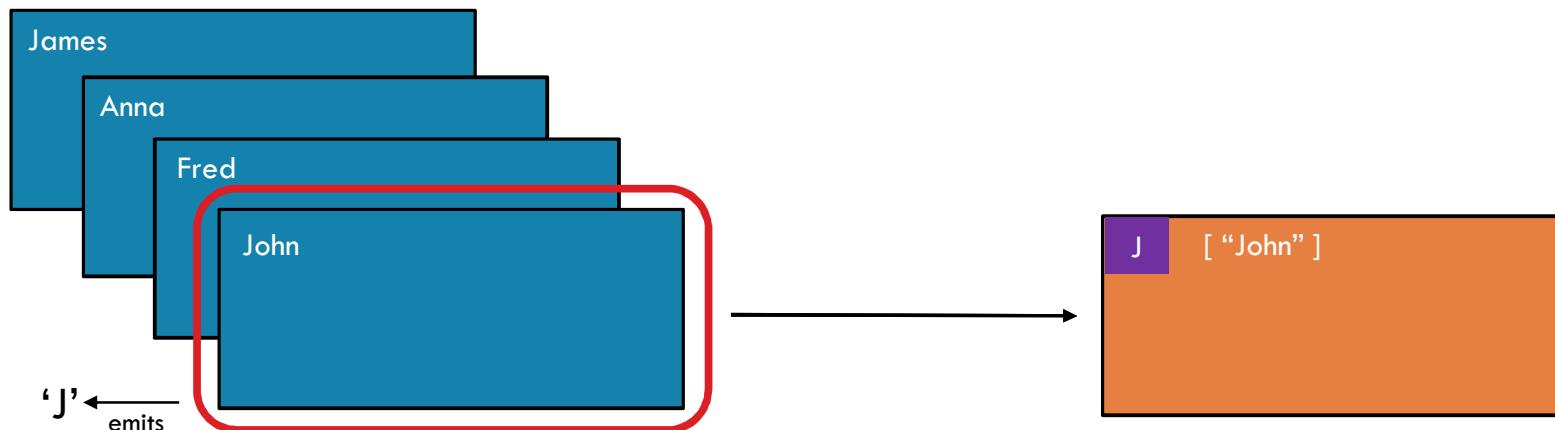




GROUPBY

RDD: **x**

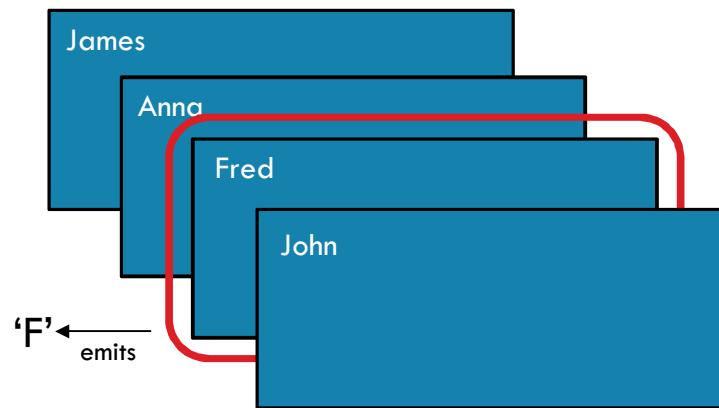
RDD: **y**



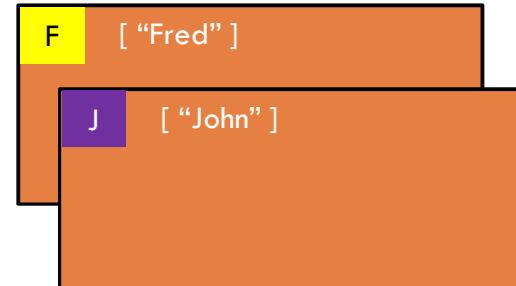


GROUPBY

RDD: **x**



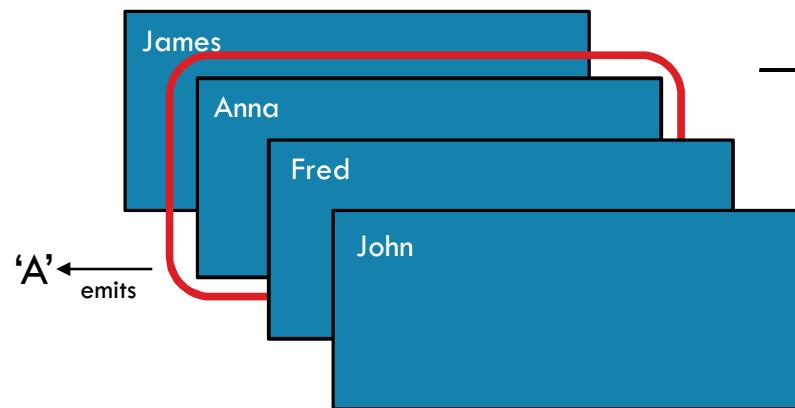
RDD: **y**





GROUPBY

RDD: **x**



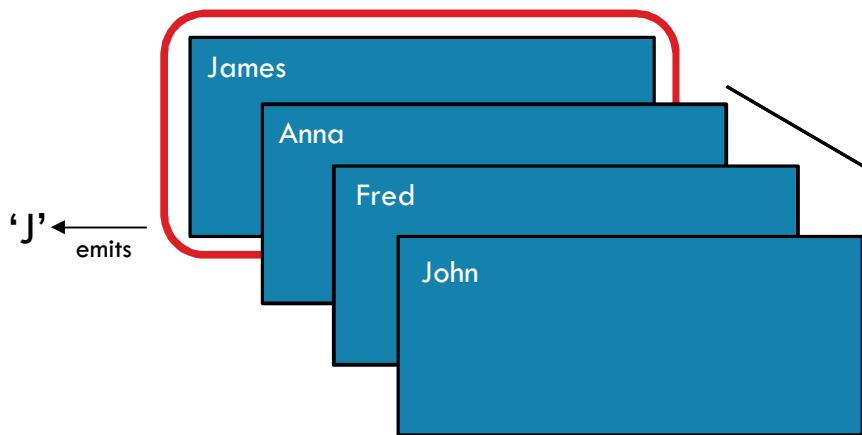
RDD: **y**





GROUPBY

RDD: **x**



RDD: **y**

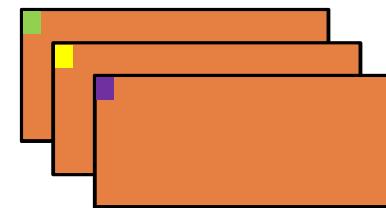


GROUPBY

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()]
```



['John**X**', 'Fred', 'Anna', 'James']

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

[('A'**Y**,'Anna')], ('J', ['John', 'James']), ('F', ['Fred'])]

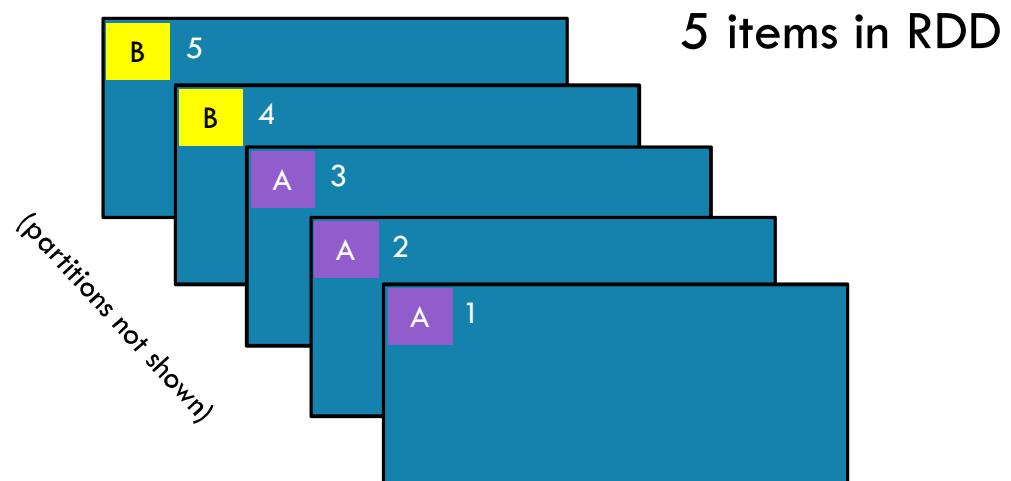
x: org.apache.spark.rdd.RDD[String] = ParallelCollectionRDD[4] at parallelize at <console>:29
y: org.apache.spark.rdd.RDD[(Char, Iterable[String])] = ShuffledRDD[6] at groupBy at <console>:31
(J,CompactBuffer(John, James)), (F,CompactBuffer(Fred)), (A,CompactBuffer(Anna))





GROUPBYKEY

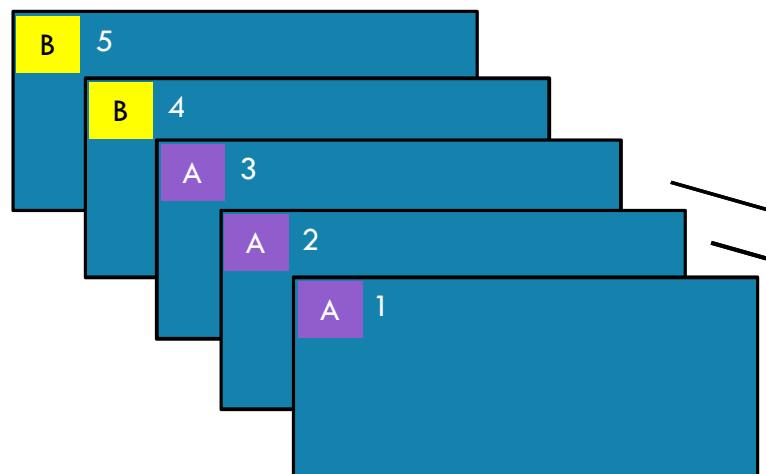
Pair RDD: `x`





GROUPBYKEY

Pair RDD: `x`



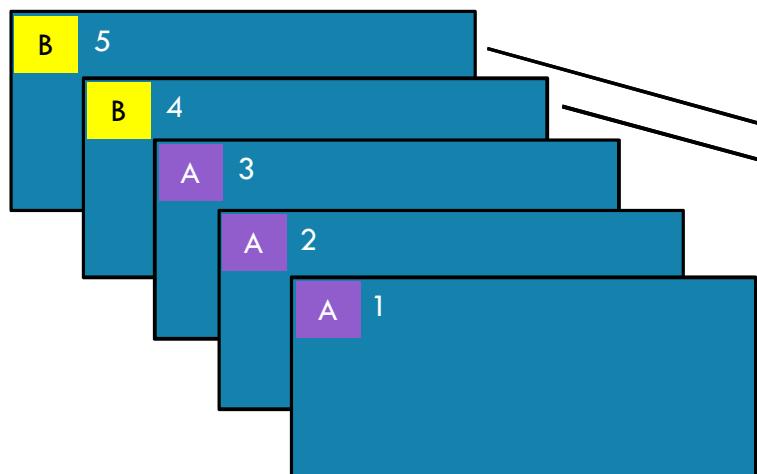
RDD: `y`



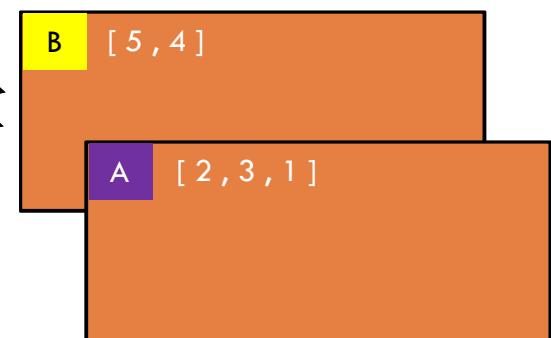


GROUPBYKEY

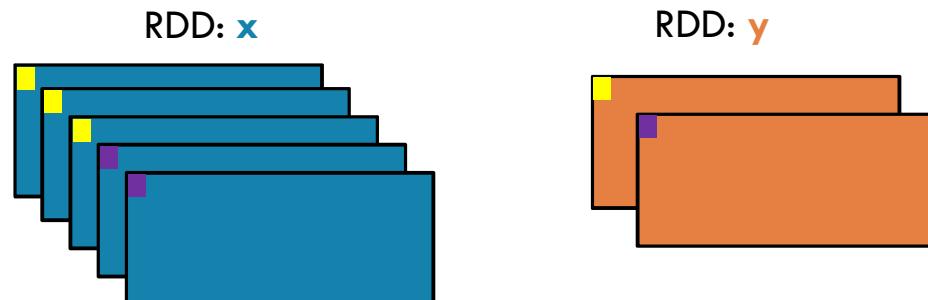
Pair RDD: `x`



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.

 `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(", "))`



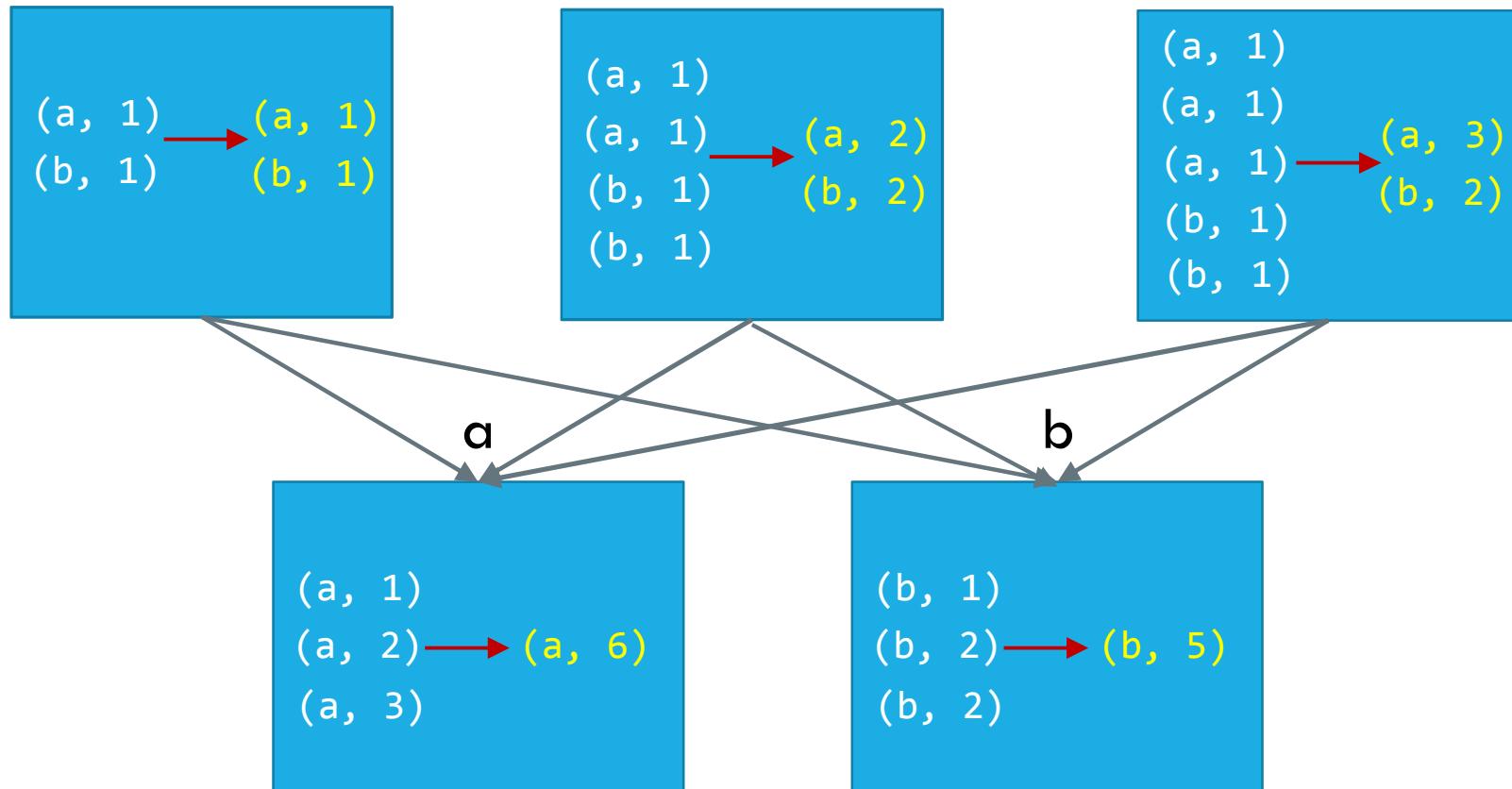
REDUCEBYKEY vs GROUPBYKEY

```
val words = Array("one", "two", "two", "three", "three", "three")
val wordPairsRDD = sc.parallelize(words).map(word => (word, 1))

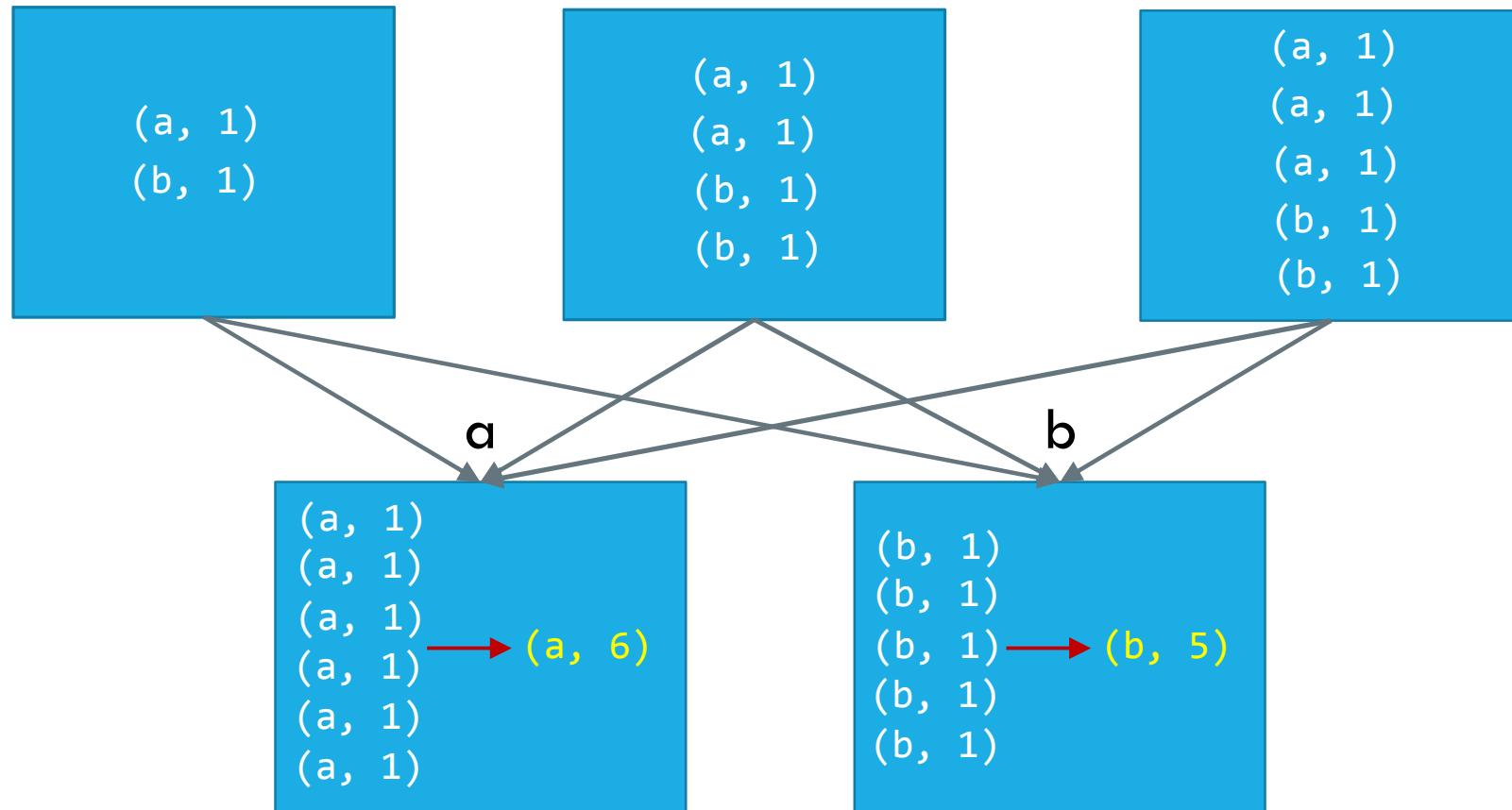
val wordCountsWithReduce = wordPairsRDD
    .reduceByKey(_ + _)
    .collect()

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

REDUCEBYKEY



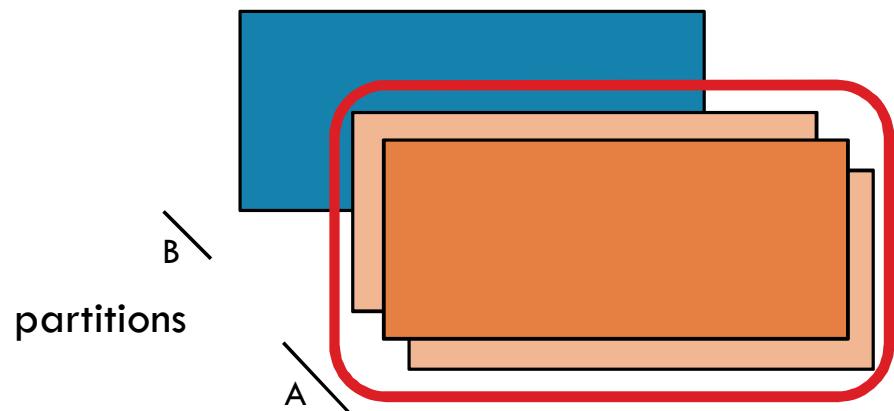
GROUPBYKEY



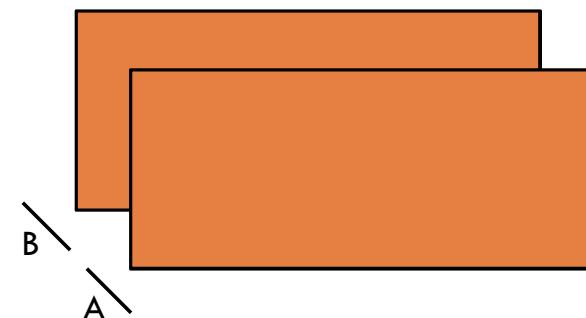


MAPPARTITIONS

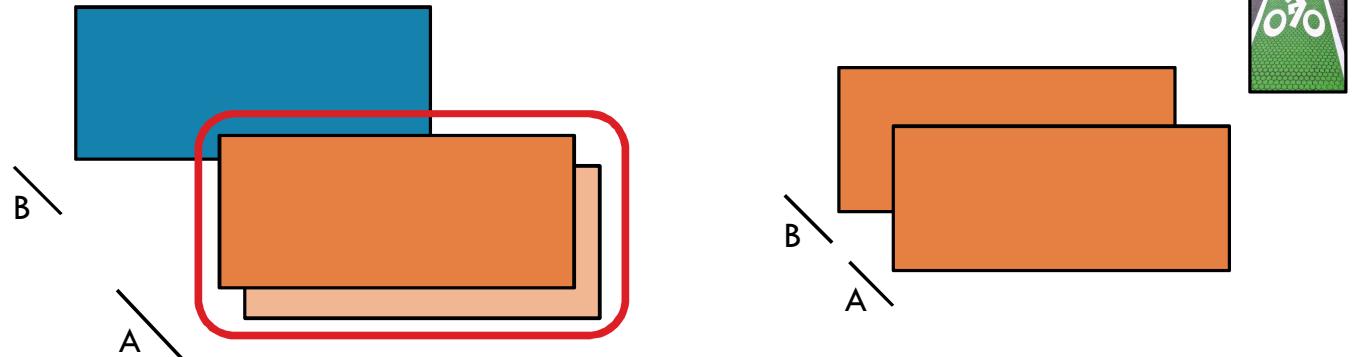
RDD: **x**



RDD: **y**



MAPPARTITIONS



`mapPartitions(f, preservesPartitioning=False)`

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

```
x = sc.parallelize([1,2,3,4,5,6,7,8,9,10], 2)

def f(iterator): yield sum(iterator);

y = x.mapPartitions(f)

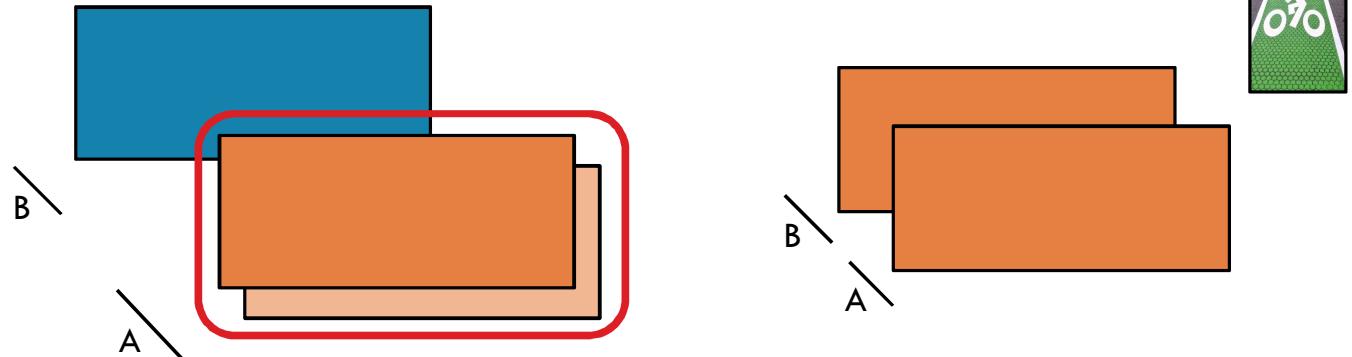
# glom() flattens elements on the same partition
print(x.glom().collect())
print(y.glom().collect())
```



```
x: [[1, 2, 3, 4, 5], [6, 7, 8, 9, 10]]
y: [[15], [40]]
```



MAPPARTITIONS



`mapPartitions(f, preservesPartitioning=False)`

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

```
val x = sc.parallelize(Array(1,2,3,4,5,6,7,8,9,10), 2)
```

```
def f(i:Iterator[Int])={ (i.sum,1).productIterator }
//The 1 is just a number so that we can
//use productIterator
```



```
val y = x.mapPartitions(f)
```

```
// glom() flattens elements on the same partition
```

```
val xOut = x.glom().collect()
val yOut = y.glom().collect()
```

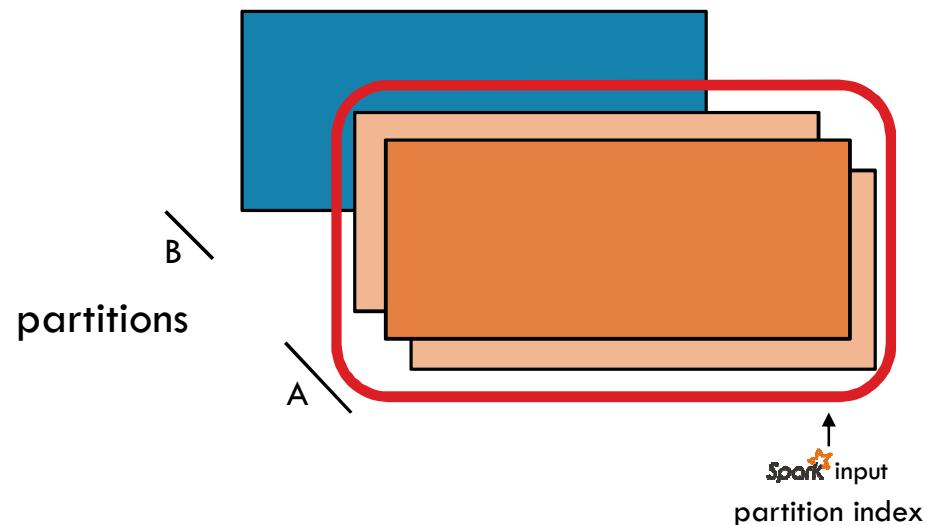
```
xOut: Array[Array[Int]] = Array(Array(1, 2, 3, 4, 5),
Array(6, 7, 8, 9, 10))
yOut: Array[Array[Any]] = Array(Array(15, 1),
Array(40, 1))
```



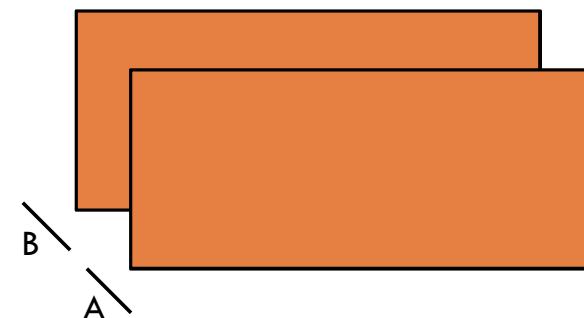


MAPPARTITIONSWITHINDEX

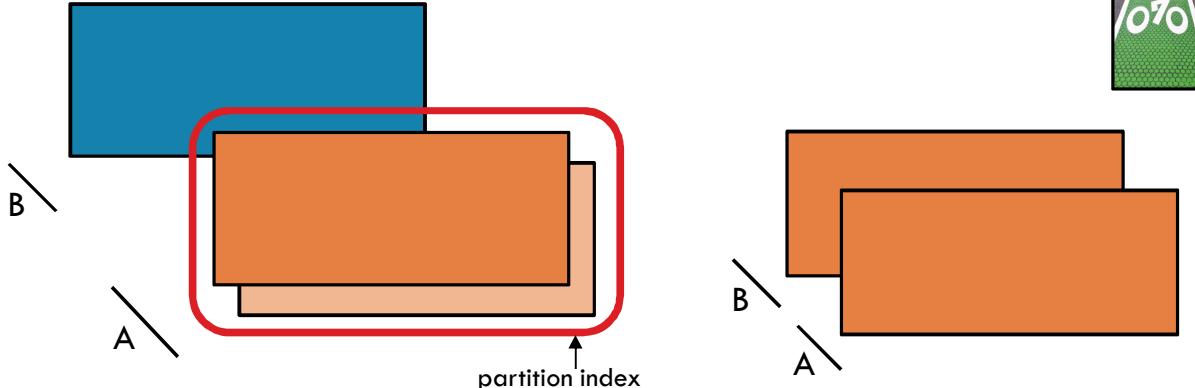
RDD: **x**



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))

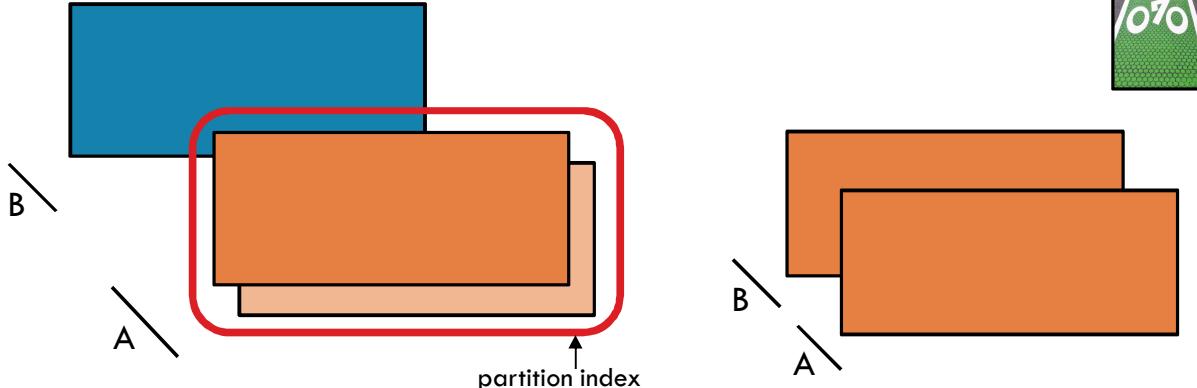
y = x.mapPartitionsWithIndex(f)

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

 B → x: [[1], [2, 3]]
A → 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()
```



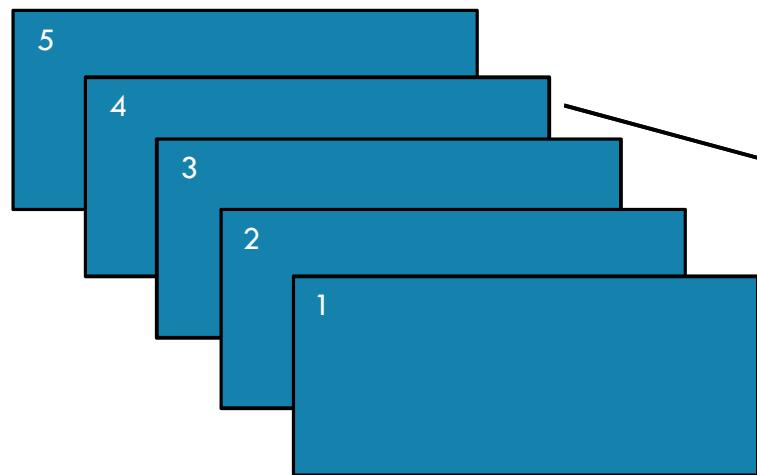
x: Array(Array(1), Array(2, 3))
y: Array(Array(0, 1), Array(1, 5))



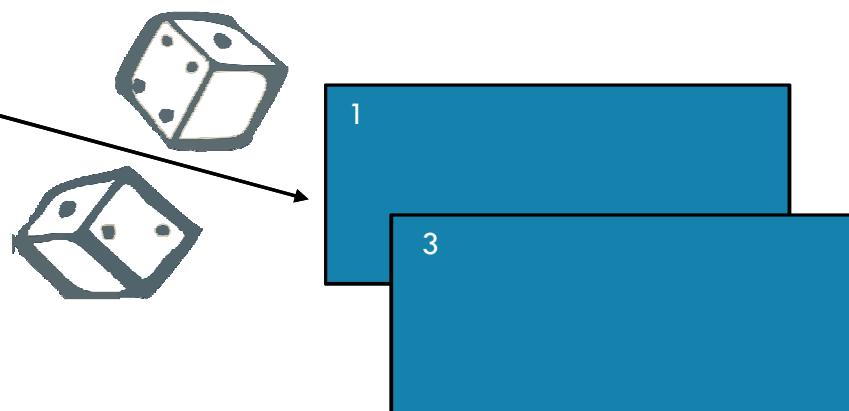


SAMPLE

RDD: **x**

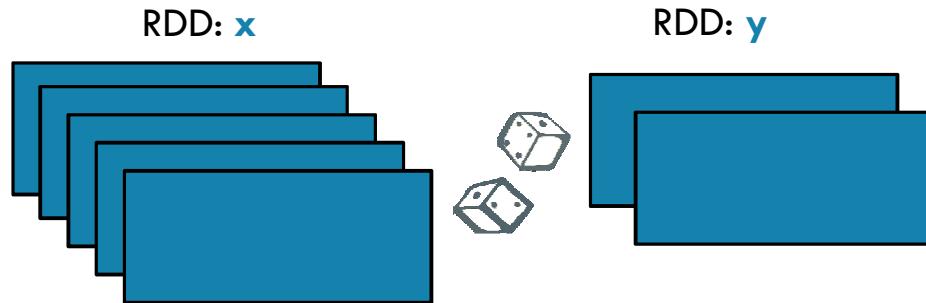


RDD: **y**





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())
```



```
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(", "))
```

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

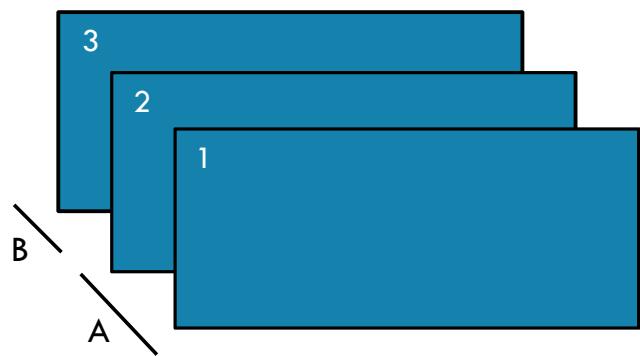
`y: [1, 3]`





UNION

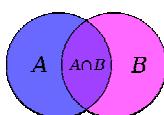
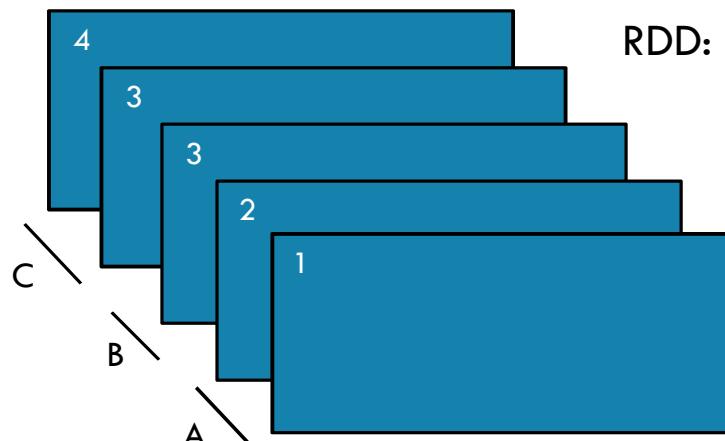
RDD: x



RDD: y

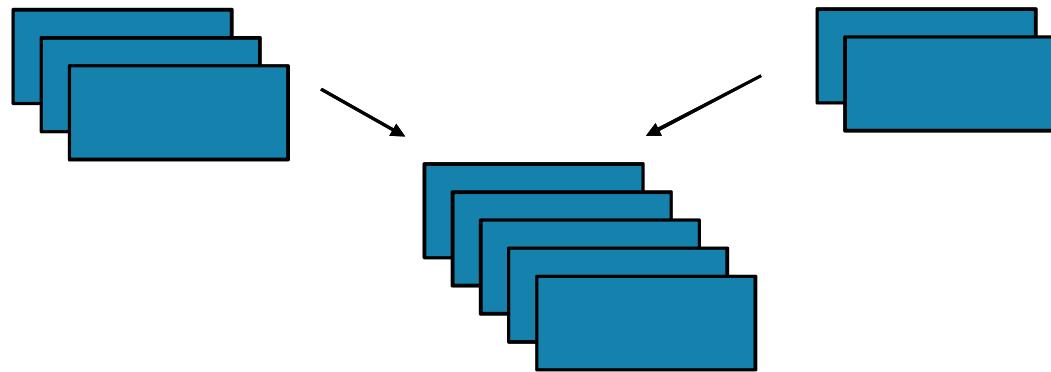


RDD: z





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())
Note: try without glom and check results
```



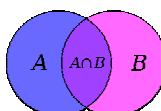
`x: [1, 2, 3]`

`y: [3, 4]`

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



```
val x = sc.parallelize(Array(1,2,3), 2)
val y = sc.parallelize(Array(3,4,5,1), 2)
val z = x.union(y)
val zOut = z.glom().collect()
```



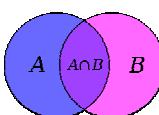
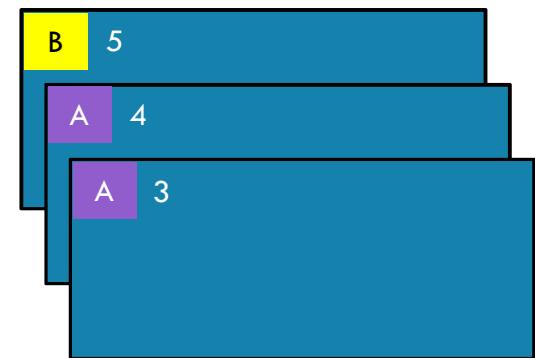


JOIN

RDD: **x**



RDD: **y**



JOIN

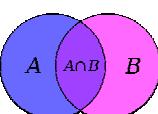
RDD: x



RDD: y



RDD: z

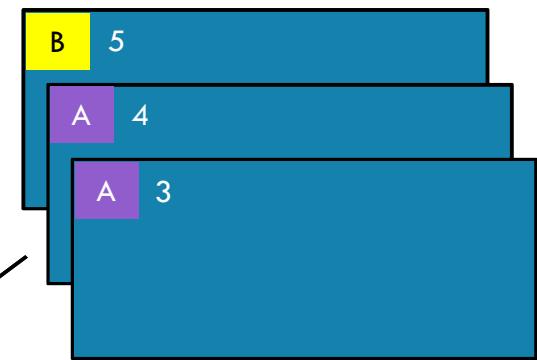


JOIN

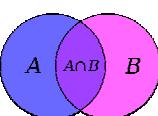
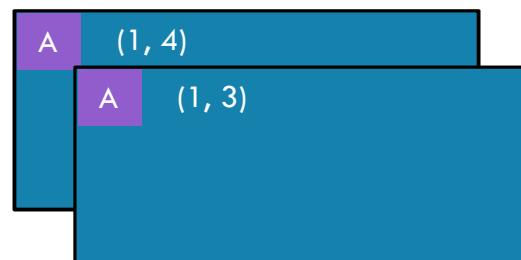
RDD: x



RDD: y

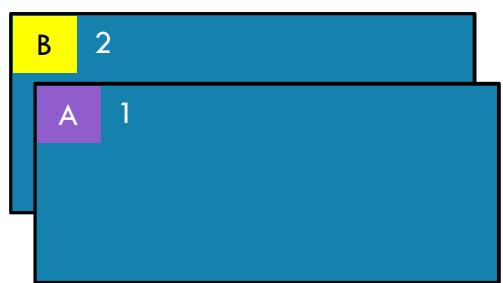


RDD: z

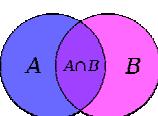
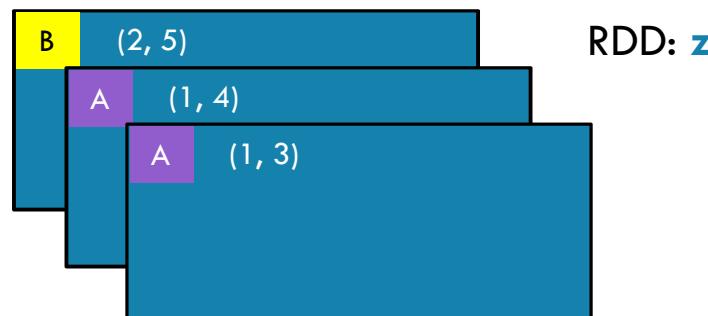
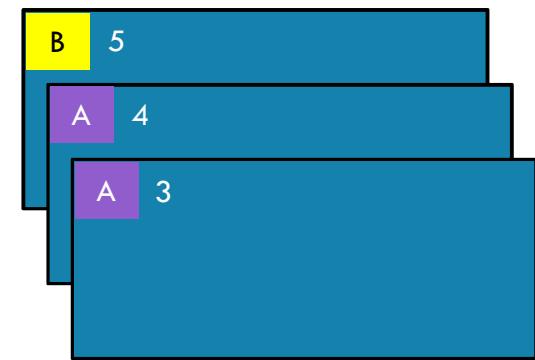


JOIN

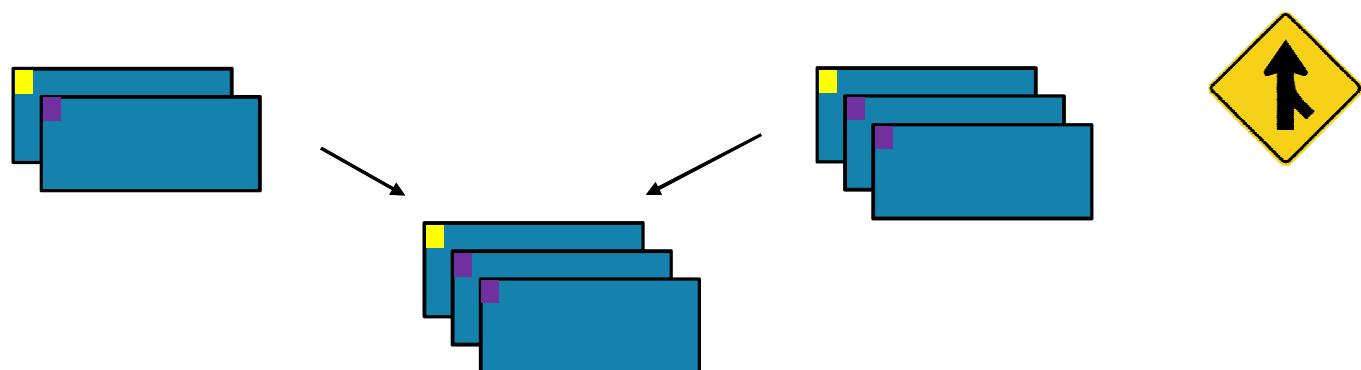
RDD: x



RDD: y



JOIN



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

`join(otherRDD, numPartitions=None)`



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



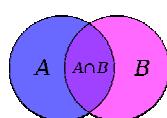
`x: [("a", 1), ("b", 2)]`

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

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



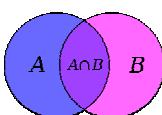
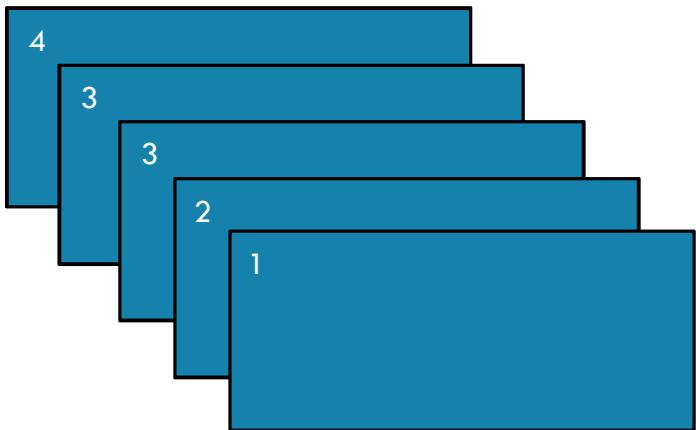
```
val x = sc.parallelize(Array(("a", 1), ("b", 2)))
val y = sc.parallelize(Array(("a", 3), ("a", 4), ("b", 5)))
val z = x.join(y)
println(z.collect().mkString(", "))
```





DISTINCT

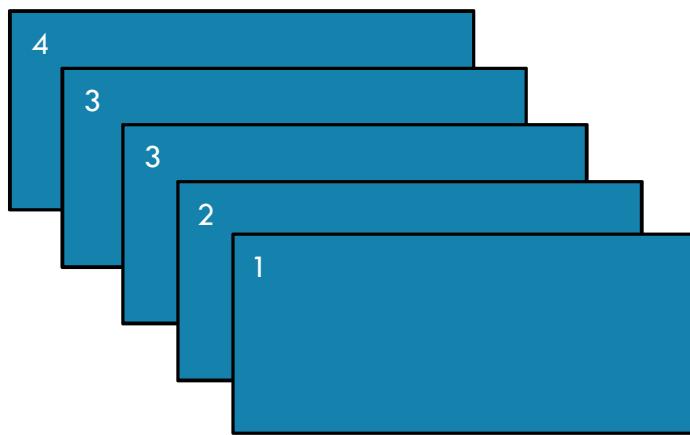
RDD: `x`



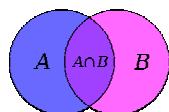
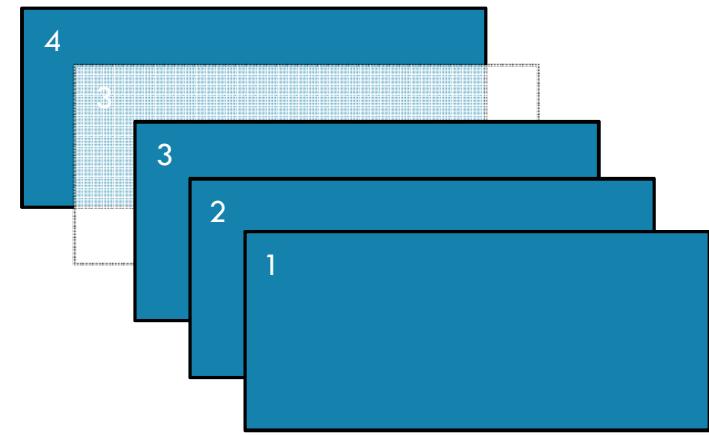


DISTINCT

RDD: x



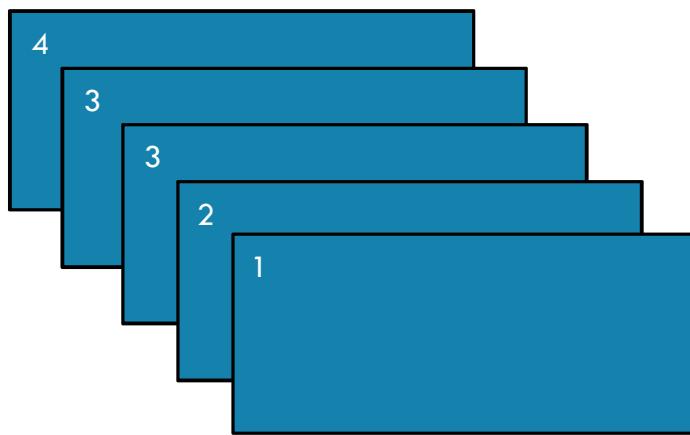
RDD: y



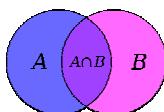
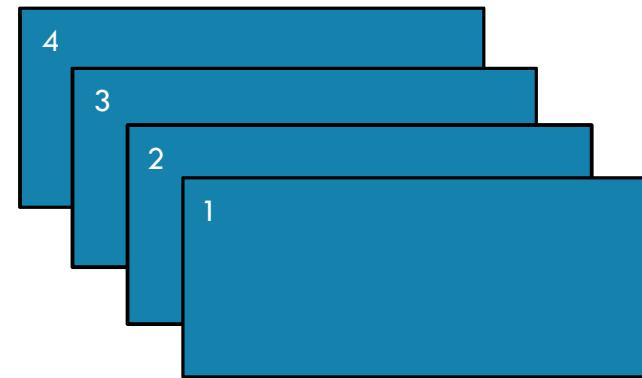


DISTINCT

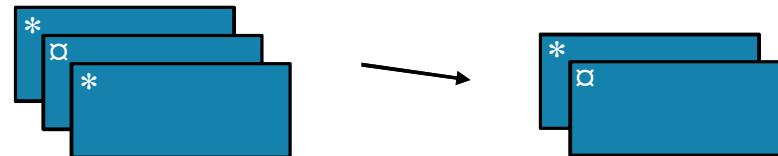
RDD: x



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())
```



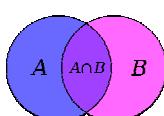
`x: [1, 2, 3, 3, 4]`

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



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

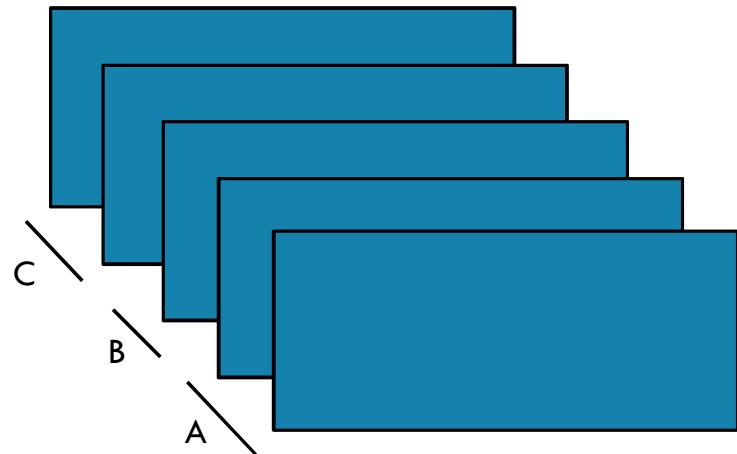
println(y.collect().mkString(", "))
```





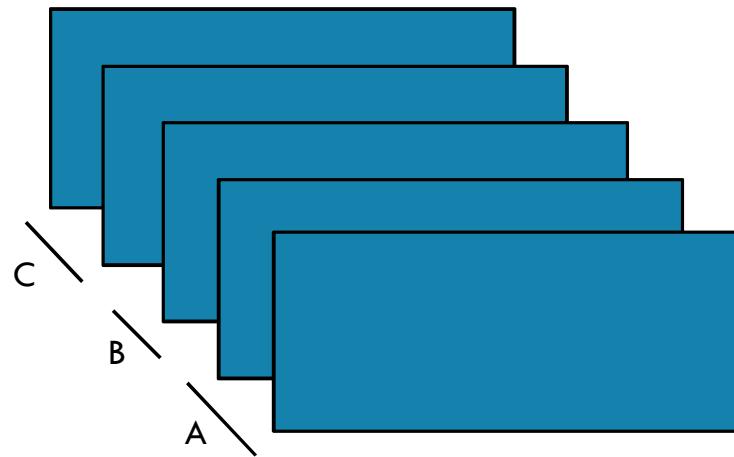
COALESCE

RDD: `x`

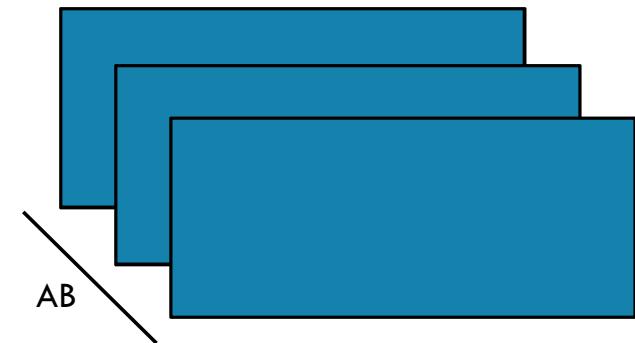


COALESCE

RDD: `x`

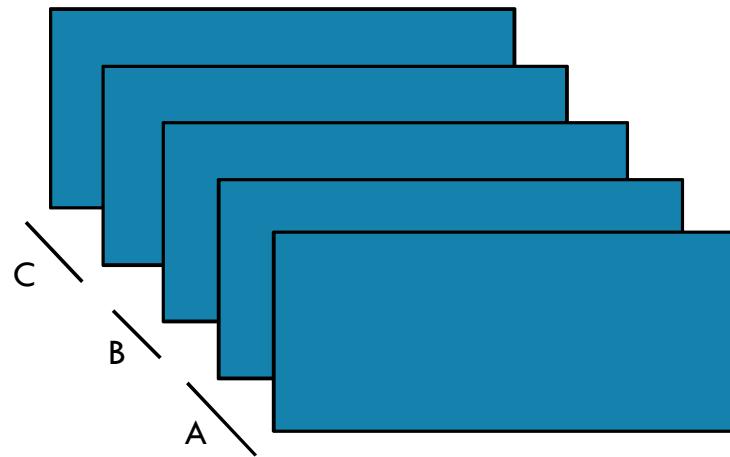


RDD: `y`

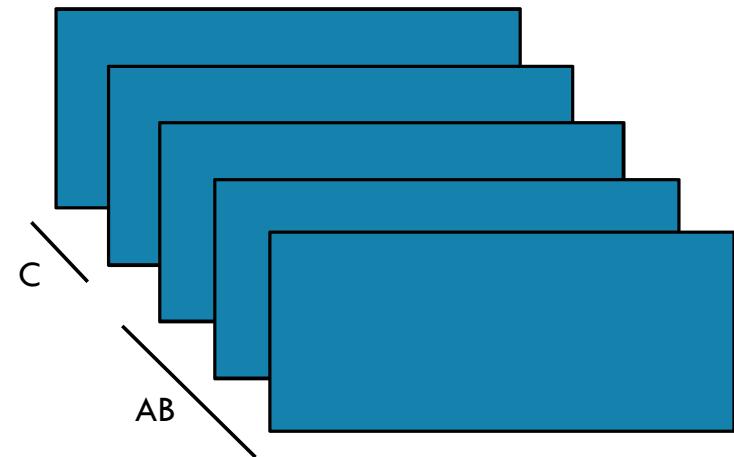


COALESCE

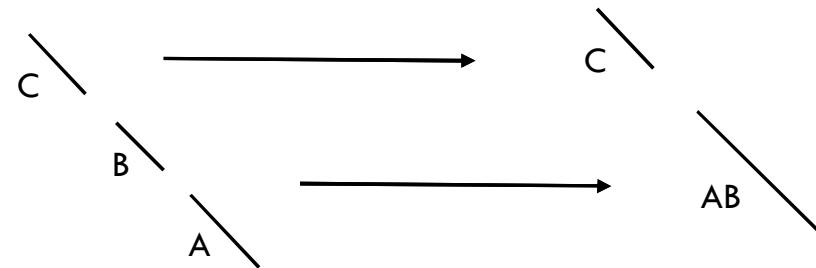
RDD: **x**



RDD: **y**



COALESCE



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

`coalesce(numPartitions, shuffle=False)`



```
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]]`



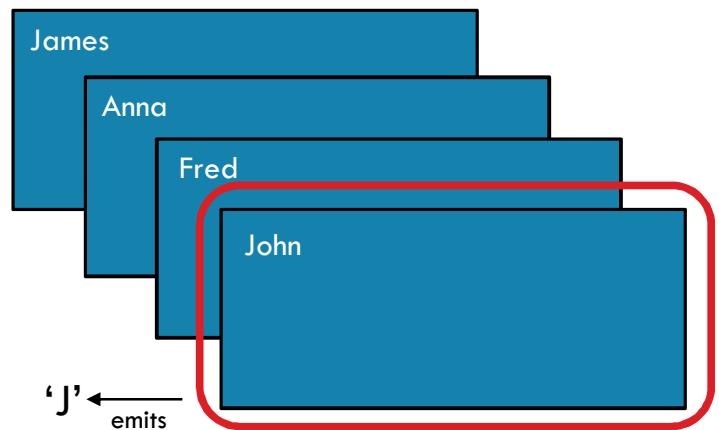
```
val x = sc.parallelize(Array(1, 2, 3, 4, 5), 3)
val y = x.coalesce(2)
val xOut = x.glom().collect()
val yOut = y.glom().collect()
```





KEYBY

RDD: **x**



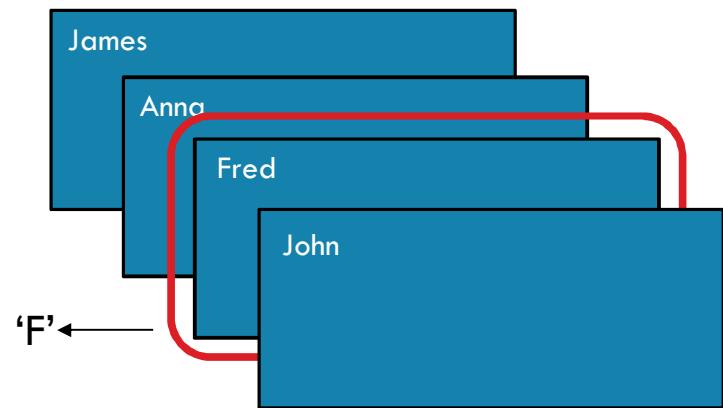
RDD: **y**





KEYBY

RDD: **x**



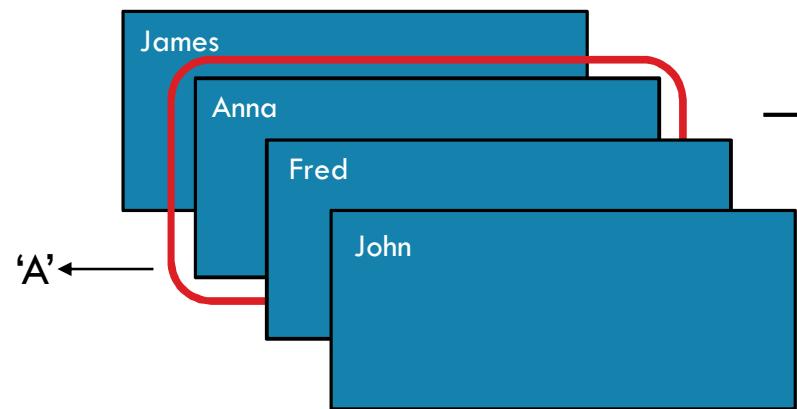
RDD: **y**



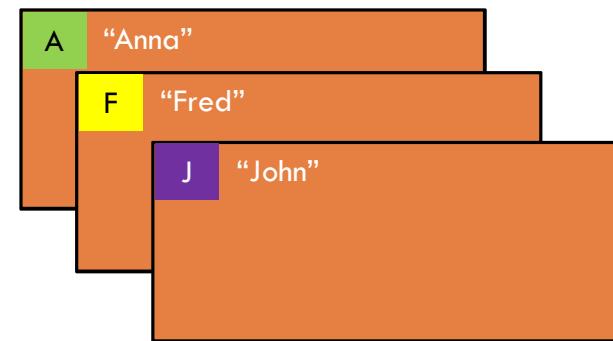


KEYBY

RDD: **x**



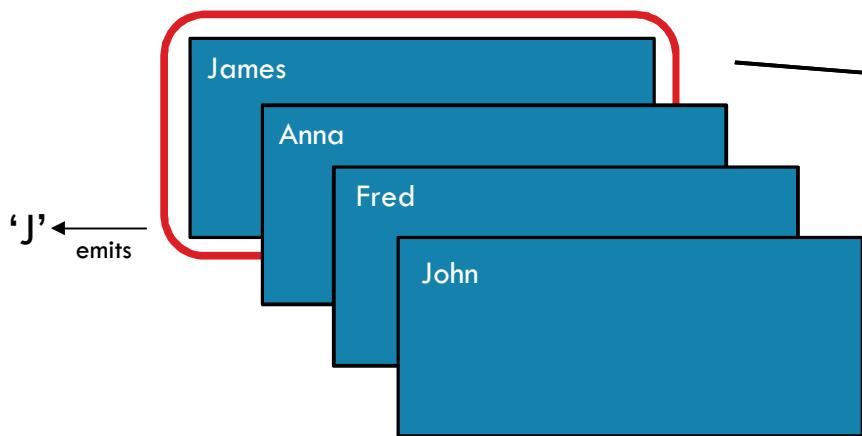
RDD: **y**



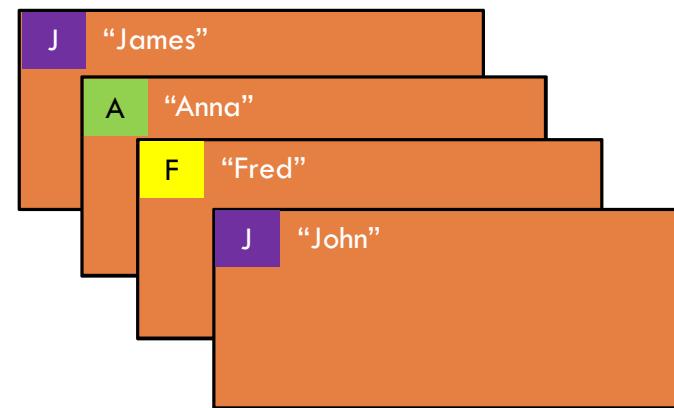


KEYBY

RDD: **x**



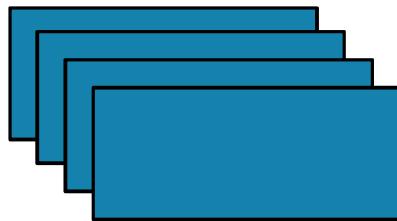
RDD: **y**





KEYBY

RDD: `x`



RDD: `y`



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



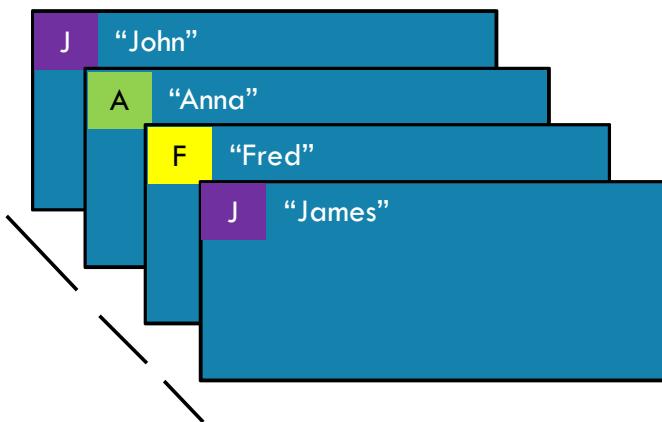
```
x: ['John', 'Fred', 'Anna', 'James']
y: [('J','John'), ('F','Fred'), ('A','Anna'), ('J','James')]
```





PARTITIONBY

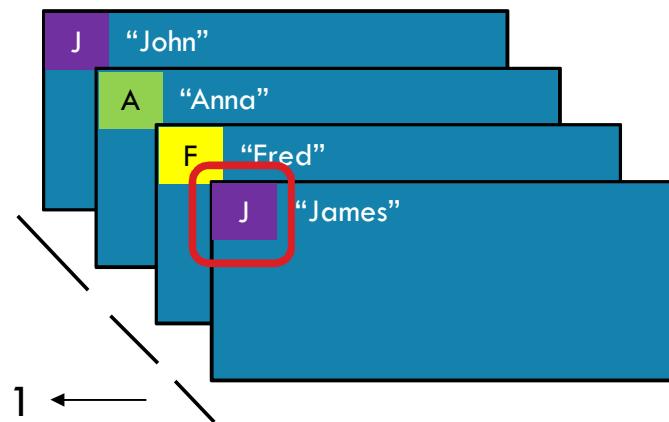
RDD: `x`





PARTITIONBY

RDD: `x`



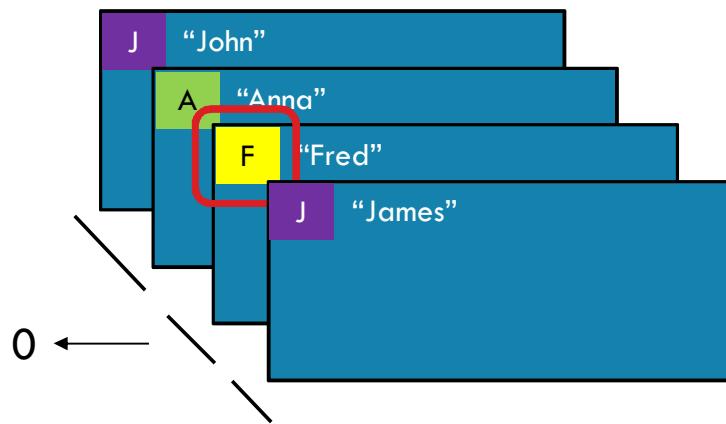
RDD: `y`





PARTITIONBY

RDD: `x`



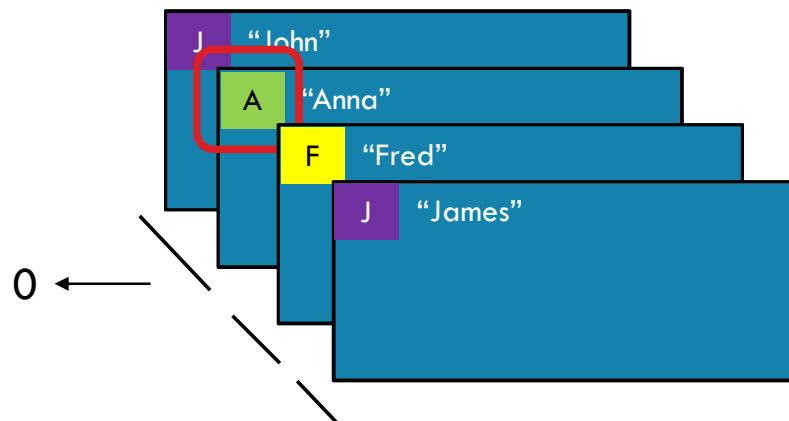
RDD: `y`





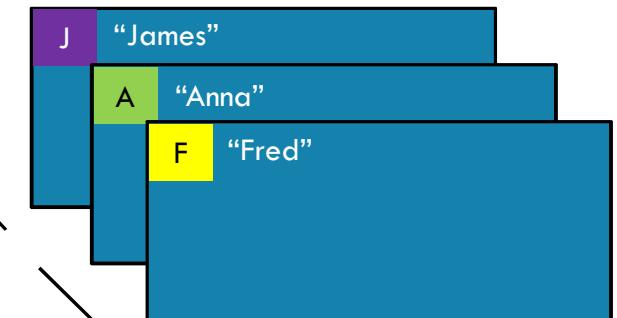
PARTITIONBY

RDD: `x`



0 ←

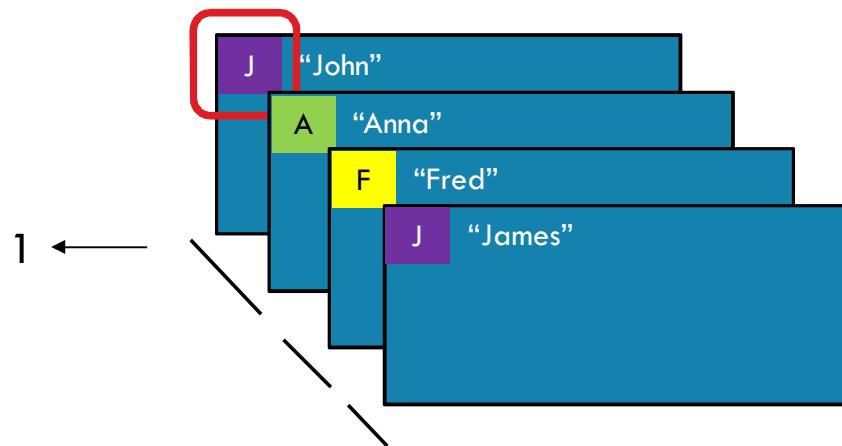
RDD: `y`



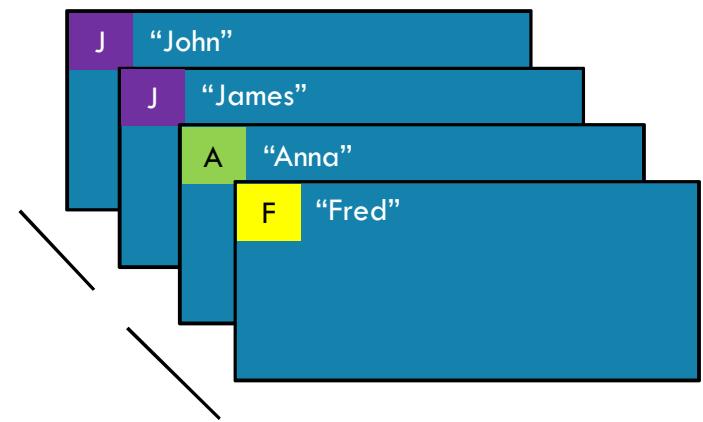


PARTITIONBY

RDD: **x**

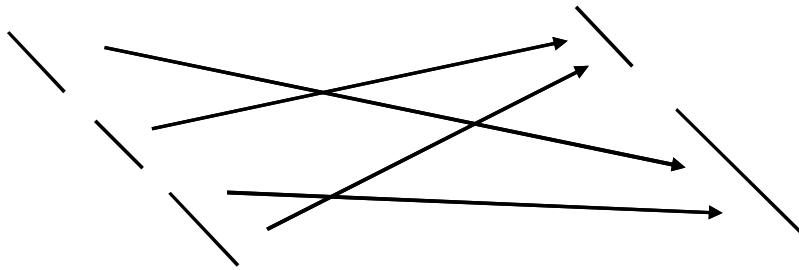


RDD: **y**





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())
```

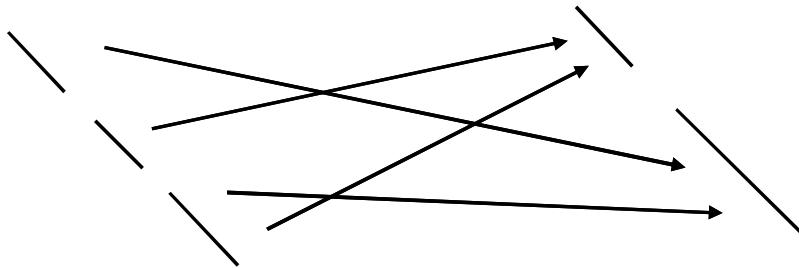


```
x: [[('J', 'James')], [('F', 'Fred')],  
     [('A', 'Anna'), ('J', 'John')]]  
  
y: [[('A', 'Anna'), ('F', 'Fred')],  
     [('J', 'James'), ('J', 'John')]]
```





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() {
    val numPartitions = 2
    def getPartition(k:Any) = {
        if (k.asInstanceOf[Char] < 'H') 0 else 1
    }
})

val yOut = y.glm().collect()
```



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

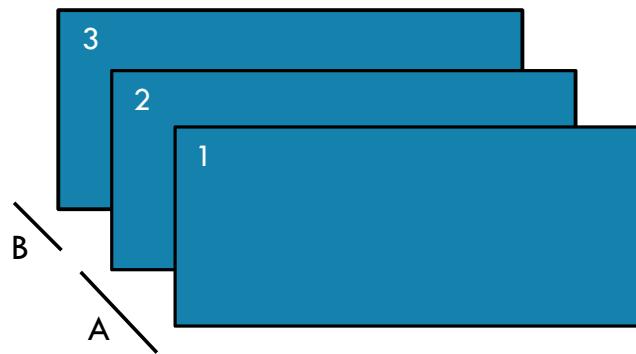
`y:` `Array(Array((F,Fred), (A,Anna)),
 Array((J,John), (J,James)))`



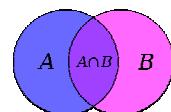
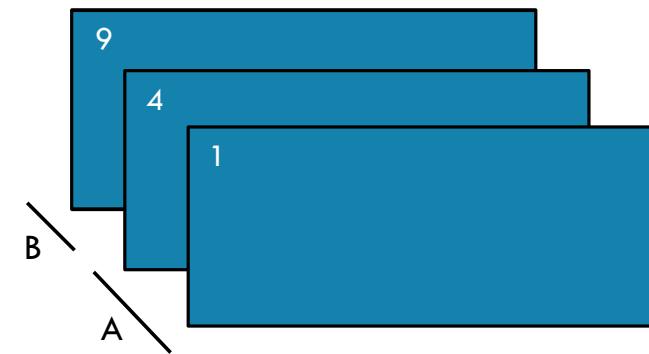


ZIP

RDD: **x**



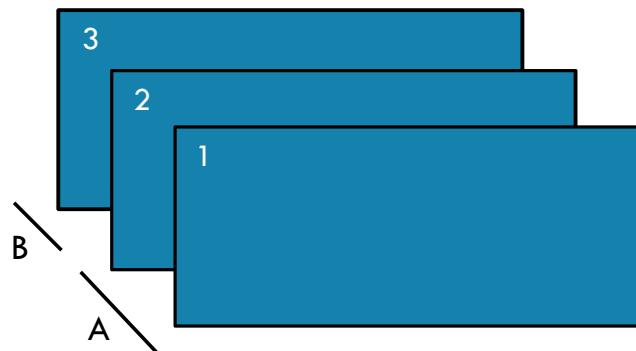
RDD: **y**



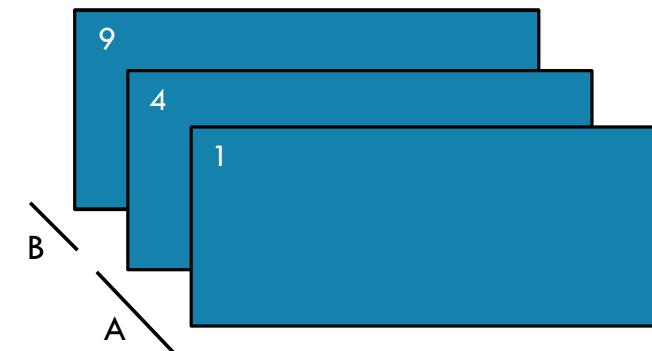


ZIP

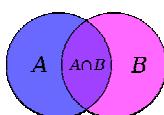
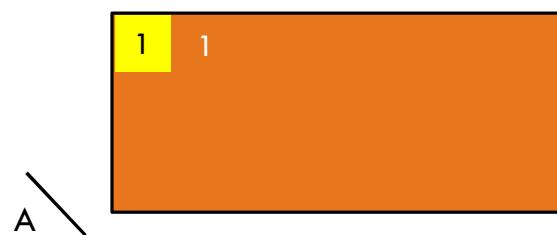
RDD: x



RDD: y



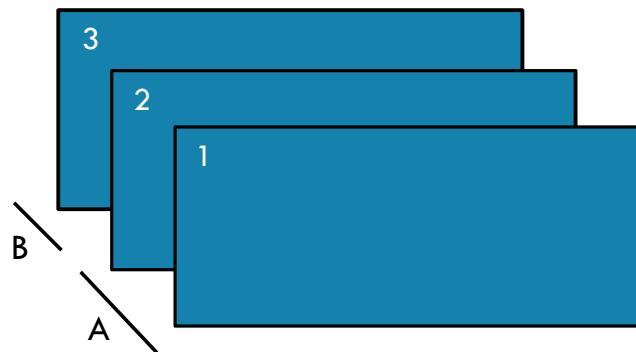
RDD: z



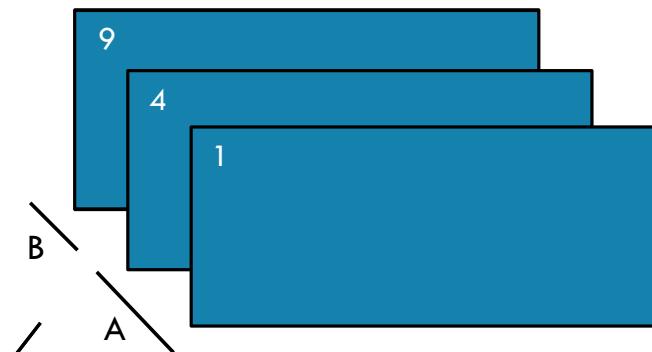


ZIP

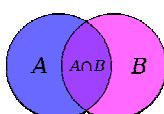
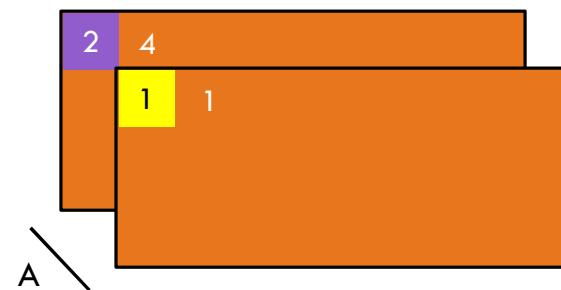
RDD: x



RDD: y



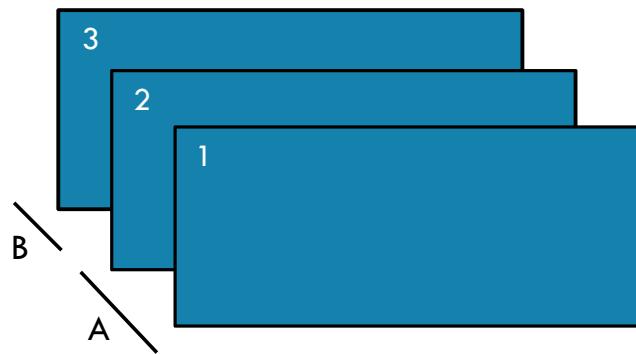
RDD: z





ZIP

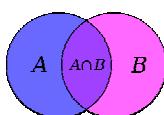
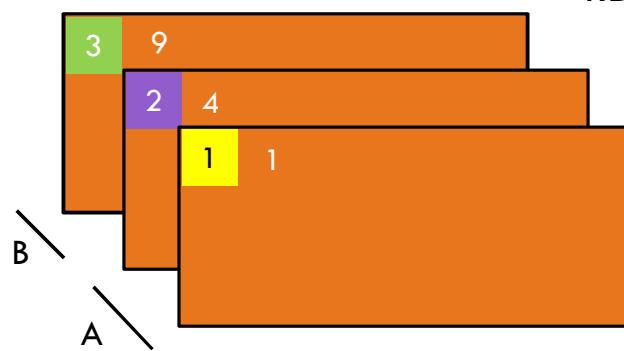
RDD: x



RDD: y

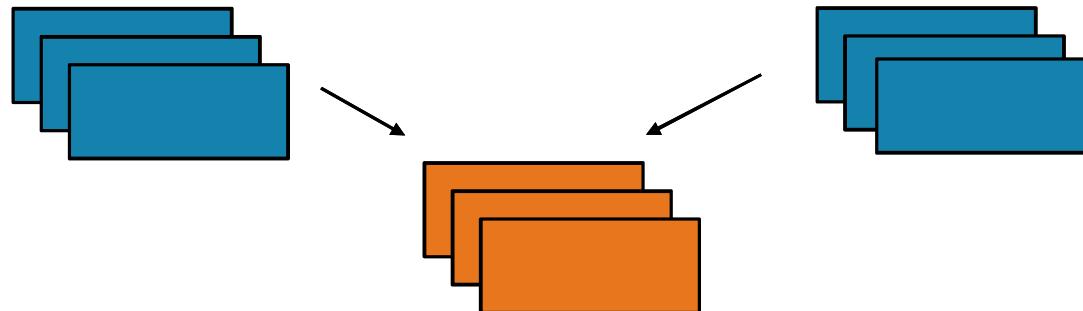


RDD: z





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)

print(z.collect())
```



`x:` [1, 2, 3]

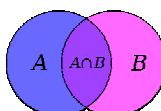
`y:` [1, 4, 9]

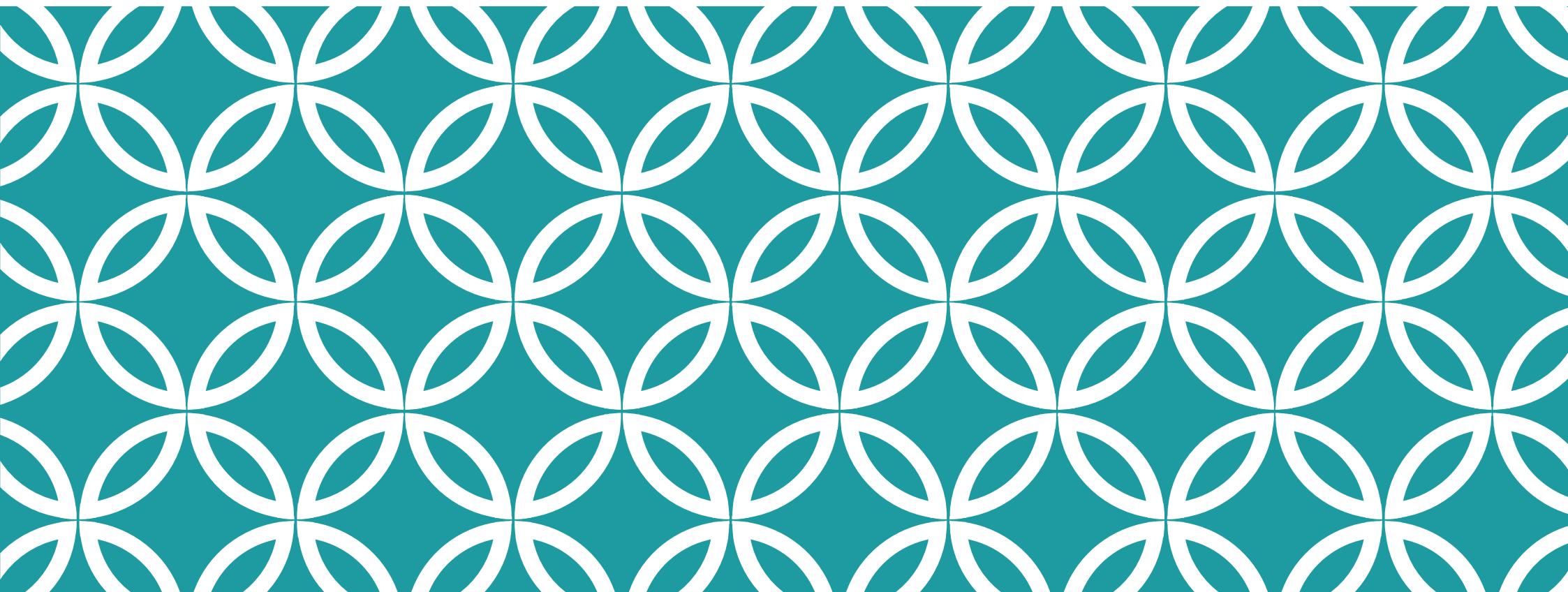
`z:` [(1, 1), (2, 4), (3, 9)]



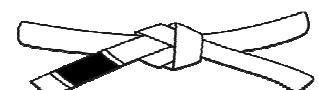
```
val x = sc.parallelize(Array(1,2,3))
val y = x.map(n=>n*n)
val z = x.zip(y)

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





ACTIONS



Core Operations



VS

distributed

occurs across the cluster

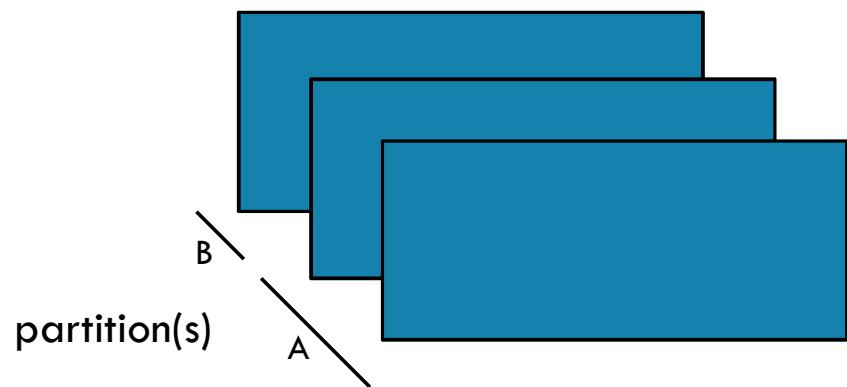
```
[root@userip-10-0-12-60 ~]# ./startspark  
Welcome to  
  
version 1.1.0  
  
Using Scala version 2.10.4 (Java HotSpot(TM) 64-Bit Server VM, Java 1.7_71)  
Type in expressions to have them evaluated.  
Type help for more information.  
Spark context available as sc.  
Created spark context.  
Spark context available as sc.  
Type help to see how they're evaluated.  
Type help for more information.  
  
scala> val cassandraRDD = sc.parallelize("tinykeyspace", 1)(keyvaluelist)  
keyvaluelist: com.datastax.spark.connector.rdd.CassandraRDD[com.datastax.spark.connector.CassandraRow] = RDD at CassandraRDD.scala:149  
  
scala> keyvaluelist.count()  
res0: long = 4  
  
scala> keyvaluelist
```

driver

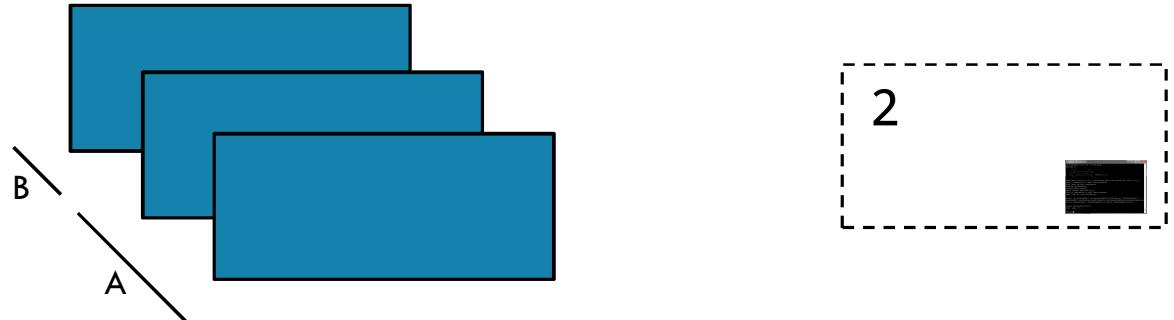
result must fit in driver JVM



GETNUMPARTITIONS



GETNUMPARTITIONS



`getNumPartitions()`

Return the number of partitions in RDD



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

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



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

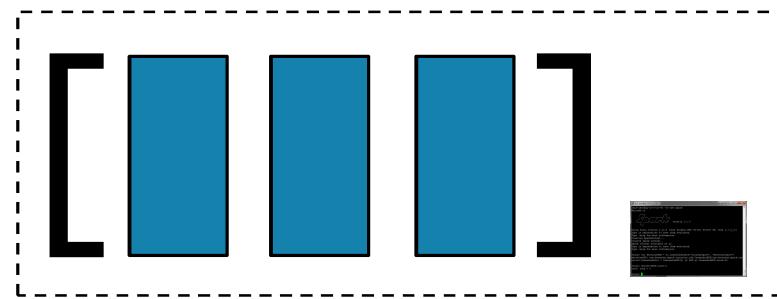
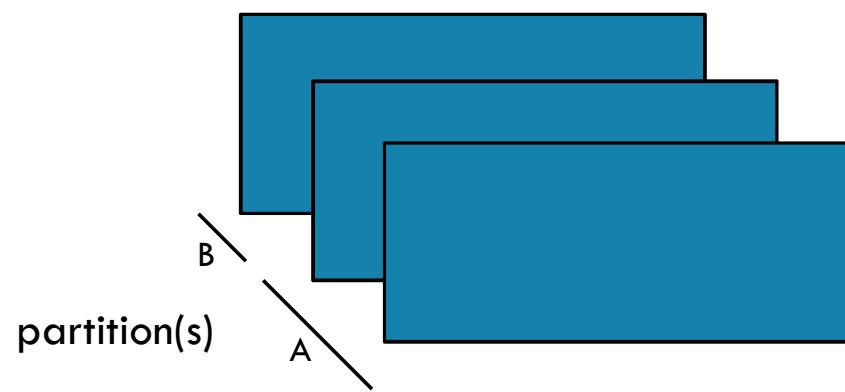
`y: 2`



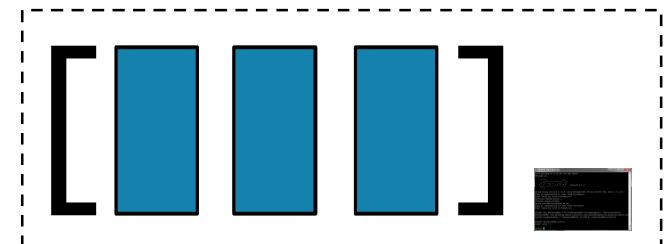
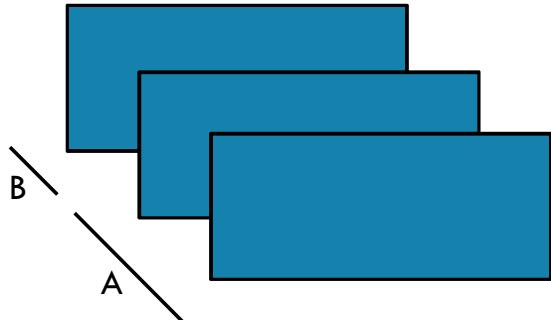
```
val x = sc.parallelize(Array(1,2,3), 2)
val y = x.partitions.size
val xOut = x.glom().collect()
println(y)
```



COLLECT



COLLECT



`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]]

`y:` [1, 2, 3]

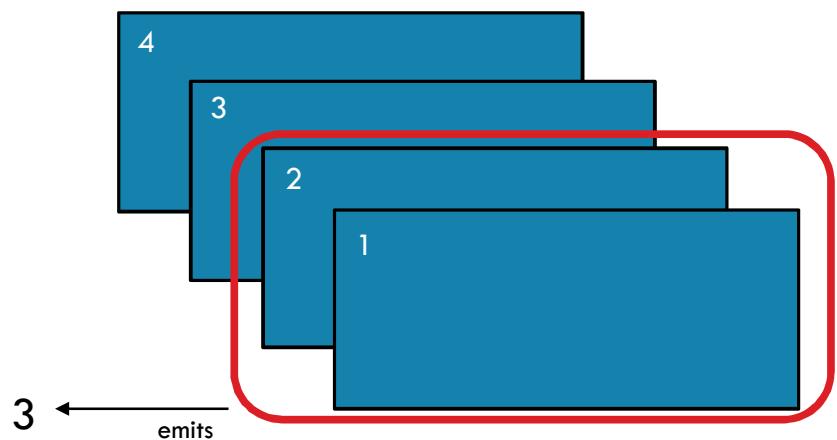


```
val x = sc.parallelize(Array(1,2,3), 2)
val y = x.collect()

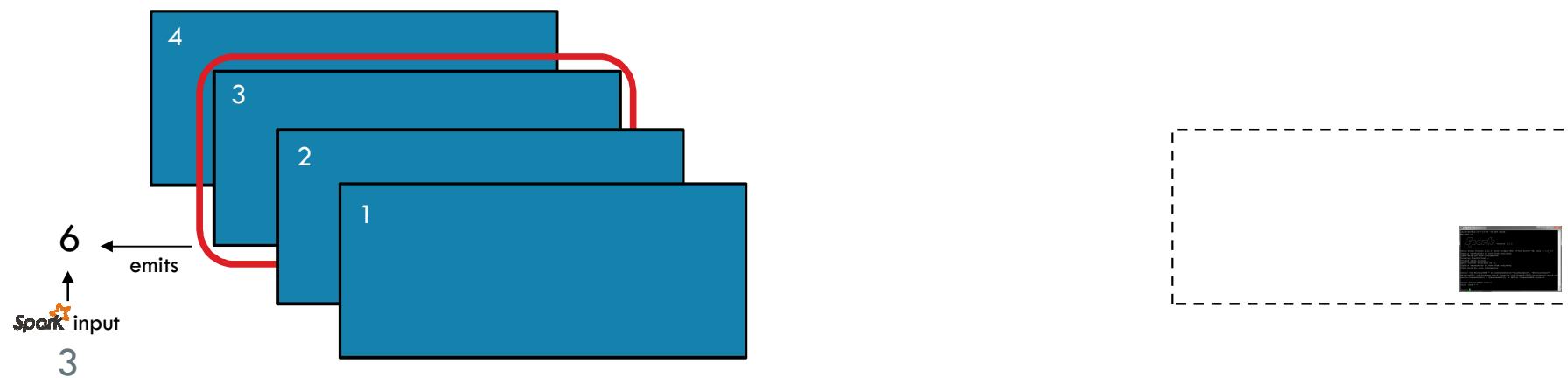
val xOut = x.glom().collect()
println(y)
```



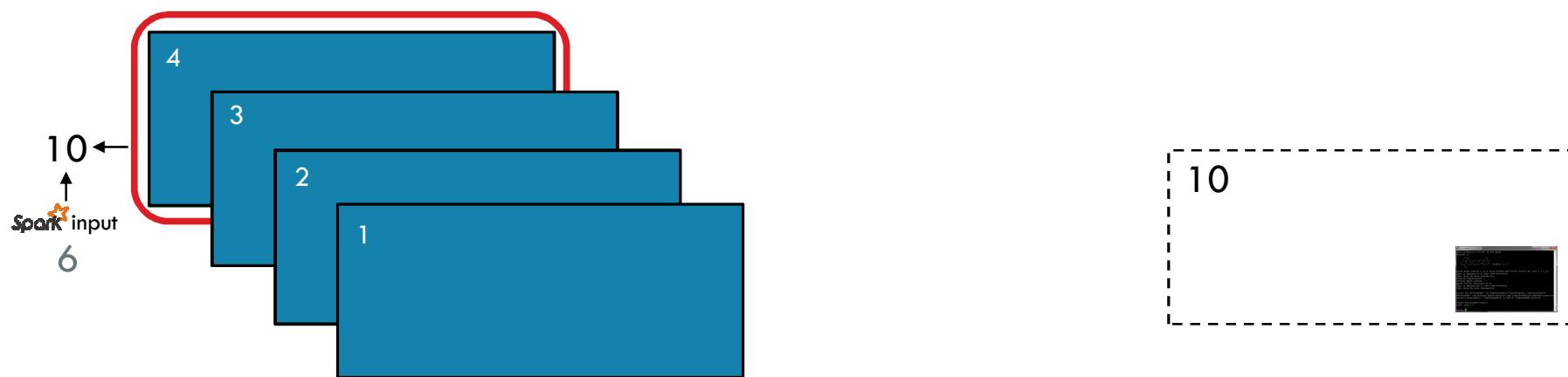
REDUCE



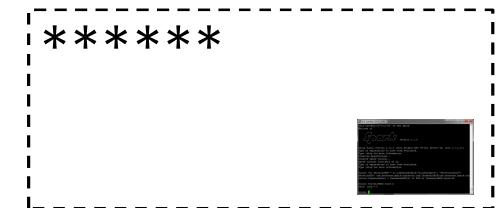
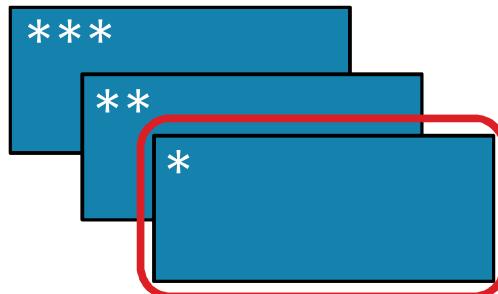
REDUCE



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

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



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



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

`y: 10`

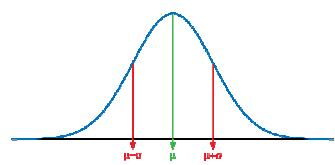
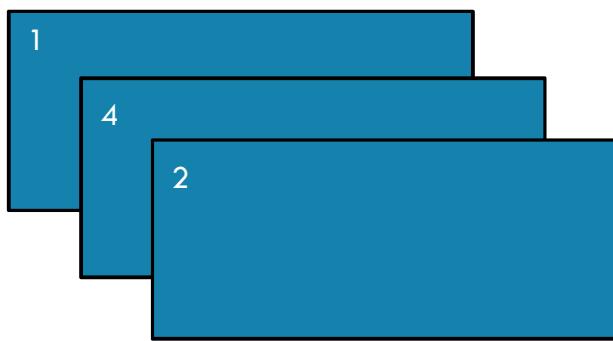
```
val x = sc.parallelize(Array(1,2,3,4))
val y = x.reduce((a,b) => a+b)
```



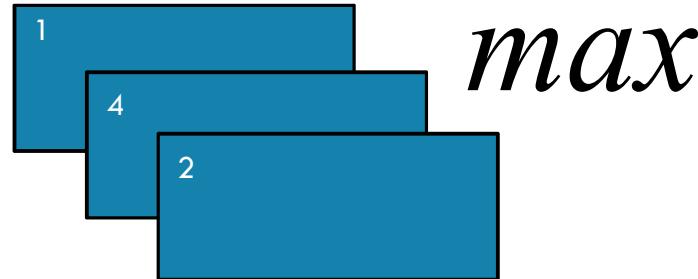
```
println(x.collect.mkString(", "))
println(y)
```



MAX



MAX



`max()`

Return the maximum item in the RDD

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

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



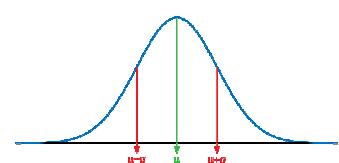
```
val x = sc.parallelize(Array(2,4,1))
val y = x.max

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

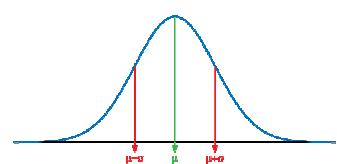
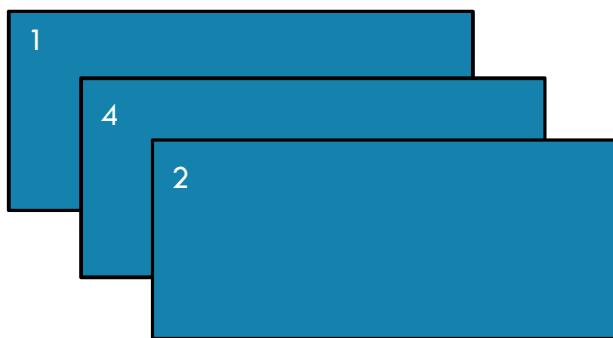


`x: [2, 4, 1]`

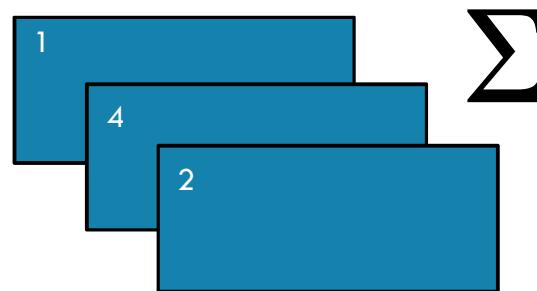
`y: 4`



SUM



SUM



`sum()`

Return the sum of the items in the RDD

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

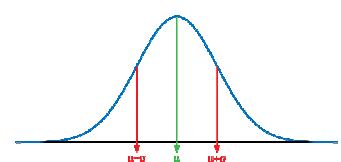
print(x.collect())
print(y)`



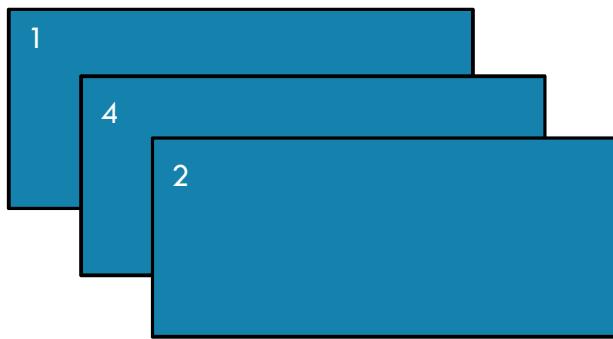
`x: [2, 4, 1]
y: 7`

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

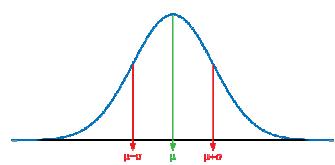
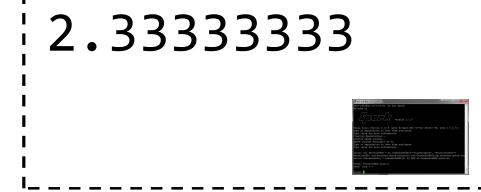
println(x.collect().mkString(", "))
println(y)`



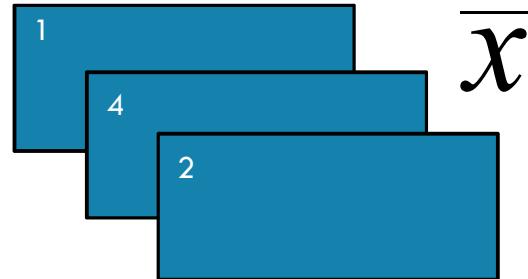
MEAN



2.33333333



MEAN



2.333333



`mean()`

Return the mean of the items in the RDD


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

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

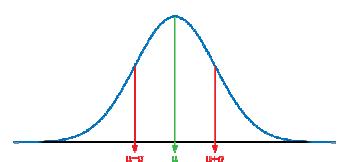


`x: [2, 4, 1]`

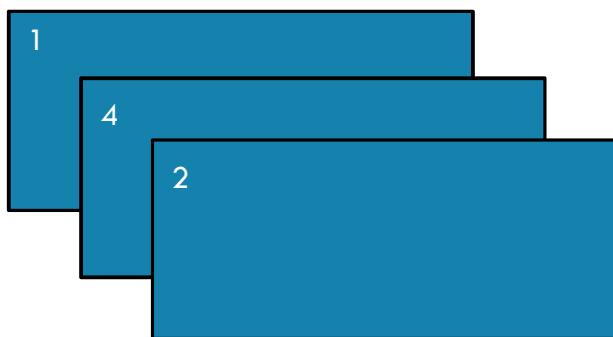
`y: 2.333333`


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

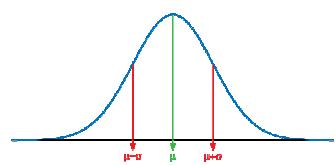
println(x.collect().mkString(", "))
println(y)`



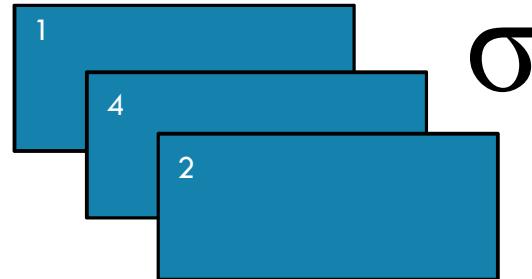
STDEV



1.2472191



STDEV



1.2472191



`stdev()`

Return the standard deviation of the items in the RDD

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

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



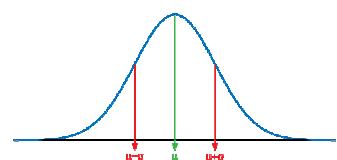
```
val x = sc.parallelize(Array(2,4,1))
val y = x.stdev

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

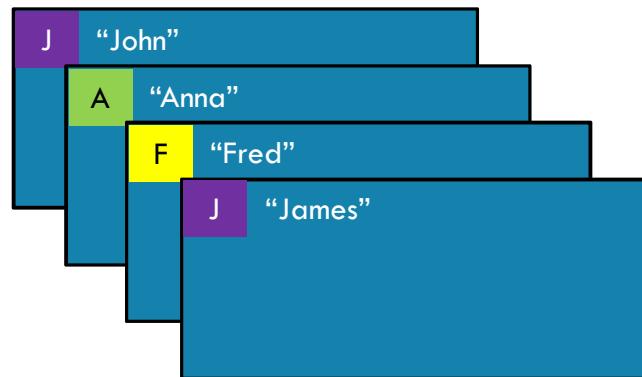


`x: [2, 4, 1]`

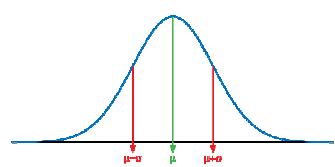
`y: 1.2472191`



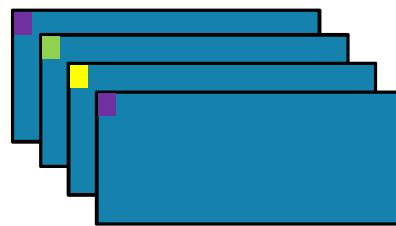
COUNTBYKEY



{'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)
```

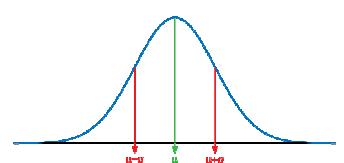


```
val x = sc.parallelize(Array(('J', "James"), ('F', "Fred"),  
                           ('A', "Anna"), ('J', "John")))
```

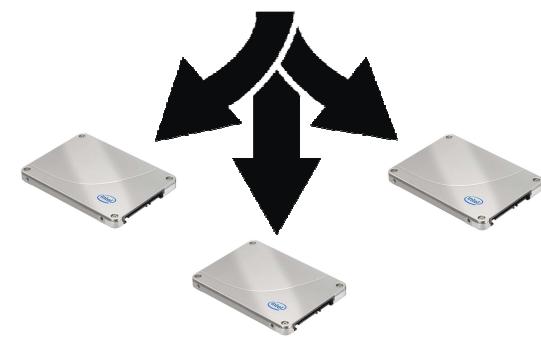
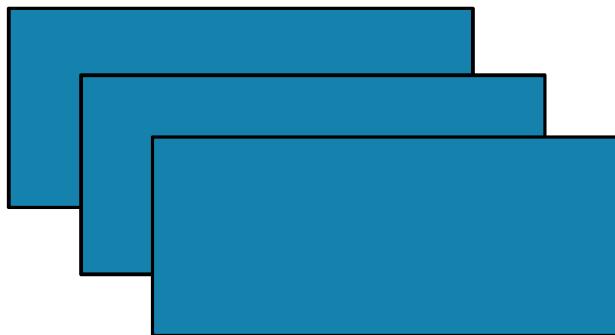
```
val y = x.countByKey()  
println(y)
```



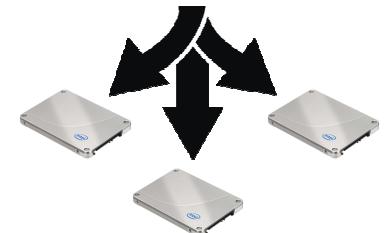
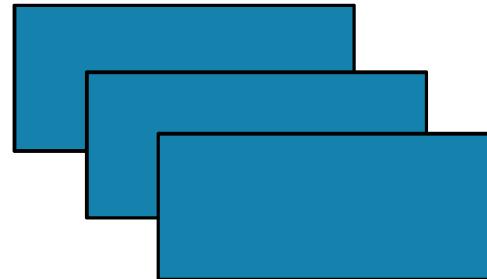
```
x: [('J', 'James'), ('F', 'Fred'),  
      ('A', 'Anna'), ('J', 'John')]  
y: {'A': 1, 'J': 2, 'F': 1}
```



SAVEASTEXTFILE



SAVEASTEXTFILE



`saveAsTextFile(path, compressionCodecClass=None)`

Save the RDD to the filesystem indicated in the path



```
x = sc.parallelize([2,4,1])
x.saveAsTextFile("PATH")
```

```
y = sc.textFile("PATH")
print(y.collect())
```

```
val x = sc.parallelize(Array(2,4,1))
x.saveAsTextFile("PATH")
```

```
val y = sc.textFile("PATH")
println(y.collect().mkString(", "))
```



`x: [2, 4, 1]`

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



LAB



Q&A

