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 Thomspon for contributing 67 visual diagrams depicting the Spark API under the MIT license to the Spark community.

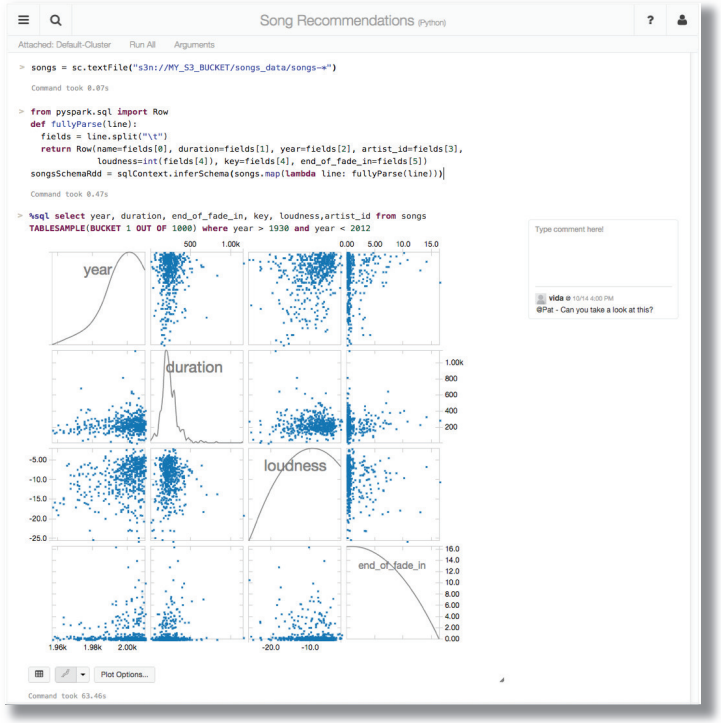
Jeff's original, creative work can be found here and you can read more about Jeff's project in his blog post.

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

databricks™

making big data simple

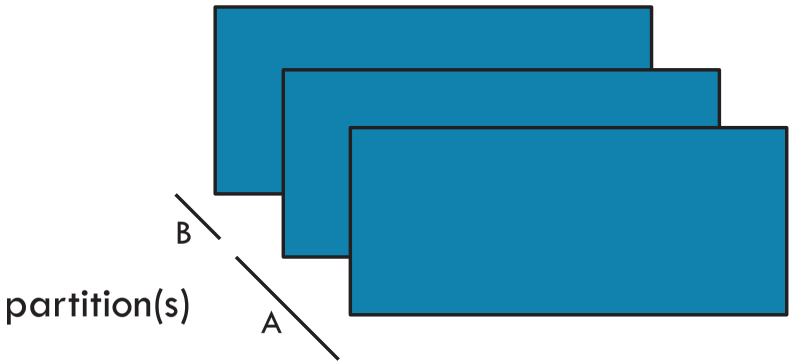
- Founded in late 2013
- by the creators of Apache Spark
- Original team from UC Berkeley AMPLab
- Raised $47 Million in 2 rounds
- ~55 employees
- We're hiring! (http://databricks.workable.com)
- Level 2/3 support partnerships with
  - Hortonworks
  - MapR
  - DataStax

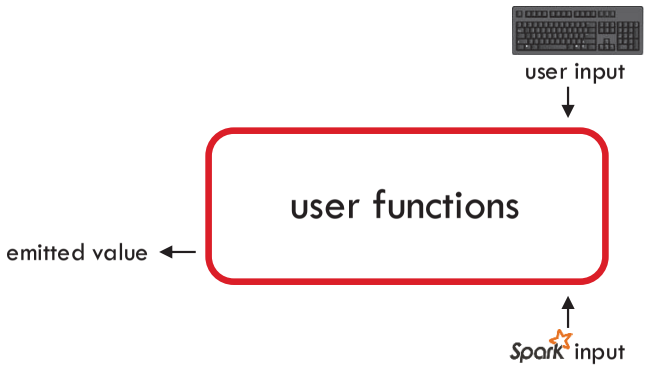


Databricks Cloud:

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

RDD





Legend

RDD Elements

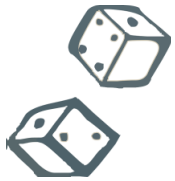
key

original item

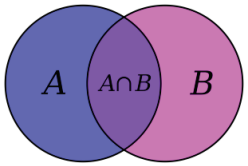
transformed type

object on driver

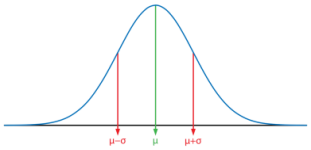
Legend



Randomized operation



Set Theory / Relational operation



Numeric calculation

Spark Operations =  TRANSFORMATIONS

+

 ACTIONS

 = easy  = medium

Essential Core & Intermediate Spark Operations

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

TRANSFORMATIONS

General

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

Math / Statistical

- sampleByKey

Set Theory / Relational

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

Data Structure

- partitionBy

ACTIONS

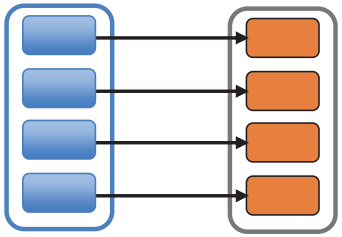
- keys
- values

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

vs

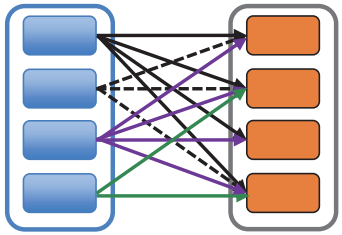
narrow

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



wide

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



LINEAGE

Resilient Distributed Datasets: A Fault-Tolerant Abstraction for In-Memory Cluster Computing

Matei Zaharia, Mosharaf Chowdhury, Tathagata Das, Ankur Dave, Justin Ma, Murphy McCauley, Michael J. Franklin, Scott Shenker, Ion Stoica

University of California, Berkeley

Abstract

We present Resilient Distributed Datasets (RDDs), a distributed memory abstraction that lets programmers perform in-memory computations on large clusters in a fault-tolerant manner. RDDs are motivated by two types of applications that current computing frameworks handle inefficiently: iterative algorithms and interactive data mining tools. In both cases, keeping data in memory can improve performance by an order of magnitude. To achieve fault tolerance efficiently, RDDs provide a restricted form of shared memory, based on coarse-grained transformations rather than fine-grained updates to shared state. However, we show that RDDs are expressive enough to capture a wide class of computations, including recent specialized programming models for iterative jobs, such as Pregel, and new applications that these models do not capture. We have implemented RDDs in a system called Spark, which we evaluate through a variety of user applications and benchmarks.

1 Introduction

Cluster computing frameworks like MapReduce [10] and Dryad [19] have been widely adopted for large-scale data analysis. These systems let users write parallel computations using a set of high-level operators, without having to worry about work distribution and fault tolerance.

Although current frameworks provide numerous abstractions for accessing a cluster's computational resources, they lack abstractions for leveraging distributed memory. This makes them inefficient for an important class of emerging applications: those that reuse intermediate results across multiple computations. Data reuse is common in many iterative machine learning and graph algorithms, including PageRank, K-means clustering, and logistic regression. Another compelling use case is interactive data mining, where a user runs multiple ad-hoc queries on the same subset of the data. Unfortunately, in most current frameworks, the only way to reuse data between computations (e.g., between two MapReduce jobs) is to write it to an external stable storage system (e.g., a distributed file system). This incurs substantial overheads due to data replication, disk I/O, and serialization, which can dominate application execution times.

Recognizing this problem, researchers have developed specialized frameworks for some applications that require data reuse. For example, Pregel [22] provides a graph processing abstraction that keeps intermediate data in memory, while HaLoop [7] offers an iterative MapReduce interface. However, these frameworks only support specific computation patterns (e.g., looping a series of MapReduce steps), and perform data sharing implicitly for these patterns. They do not provide abstractions for more general reuse, e.g., to let a user load several datasets into memory and run ad-hoc queries across them.

In this paper, we propose a new abstraction called resilient distributed datasets (RDDs) that enables efficient data reuse in a broad range of applications. RDDs are fault-tolerant, parallel data structures that let users explicitly persist intermediate results in memory, control their partitioning to optimize data placement, and manipulate them using a rich set of operators.

The main challenge in designing RDDs is defining a programming interface that can provide fault tolerance efficiently. Existing abstractions for in-memory storage on clusters, such as distributed shared memory [24], key-value stores [25], databases, and Piccolo [27], offer an interface based on fine-grained updates to mutable state (e.g., cells in a table). With this interface, the only ways to provide fault tolerance are to replicate the data across machines or to log updates across machines. Both approaches are expensive for data-intensive workloads, as they require copying large amounts of data over the cluster network, whose bandwidth is far lower than that of RAM, and they incur substantial storage overhead.

In contrast to this current framework, RDDs offer an interface based on coarse-grained transformations (e.g., map, filter and join) that apply the same operation to many data items. This allows them to efficiently provide fault tolerance by logging the transformations used to build a dataset (its *lineage*) rather than the actual data.[1] If a partition of an RDD is lost, the RDD has enough information about how it was derived from other RDDs to recompute just that partition. Thus, lost data can be recovered, often quite quickly, without requiring costly replication.

[1] Checkpointing the data in some RDDs may be useful when a lineage chain grows large, however, and we discuss how to do it in Section 5.4.

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

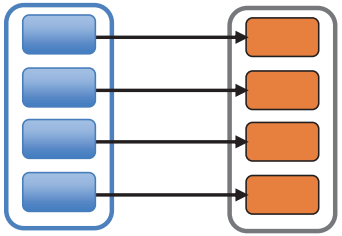
"The most interesting question in designing this interface is how to represent dependencies between RDDs."

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

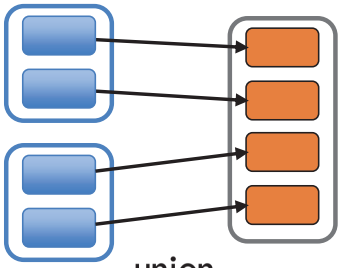
- narrow dependencies, where each partition of the parent RDD is used by at most one partition of the child RDD
- wide dependencies, where multiple child partitions may depend on it."

narrow

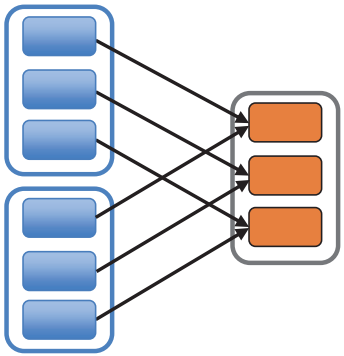
each partition of the parent RDD is used by at most one partition of the child RDD



map, filter



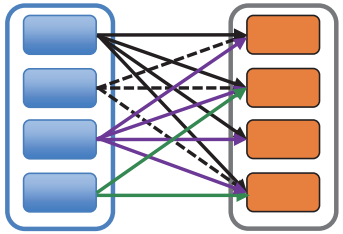
union



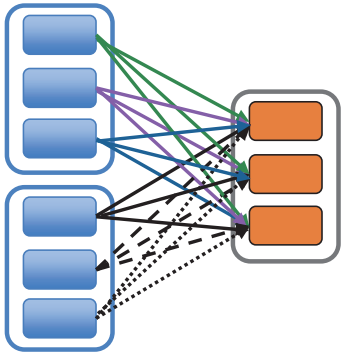
join w/ inputs co-partitioned

wide

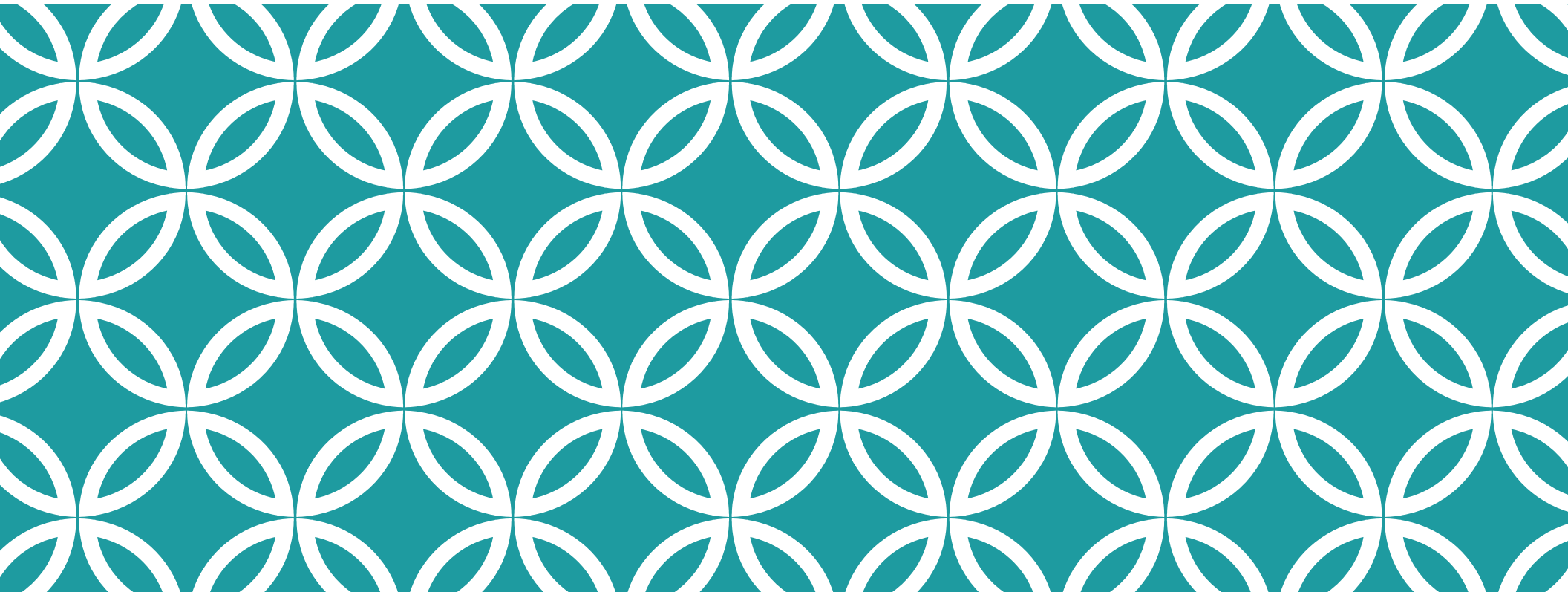
multiple child RDD partitions may depend on a single parent RDD partition



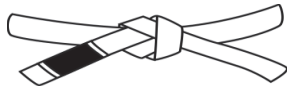
groupByKey



join w/ inputs not co-partitioned



TRANSFORMATIONS

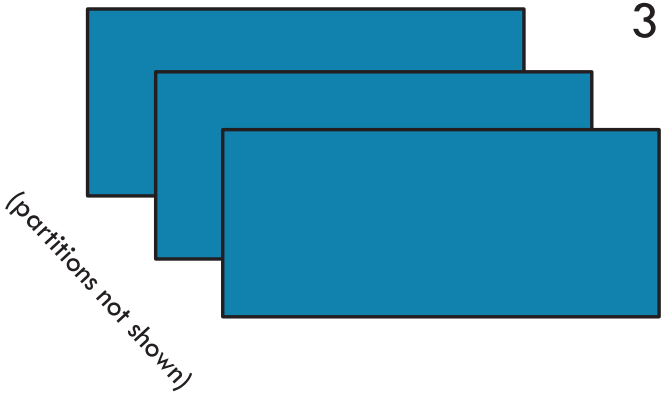


Core Operations

databricks

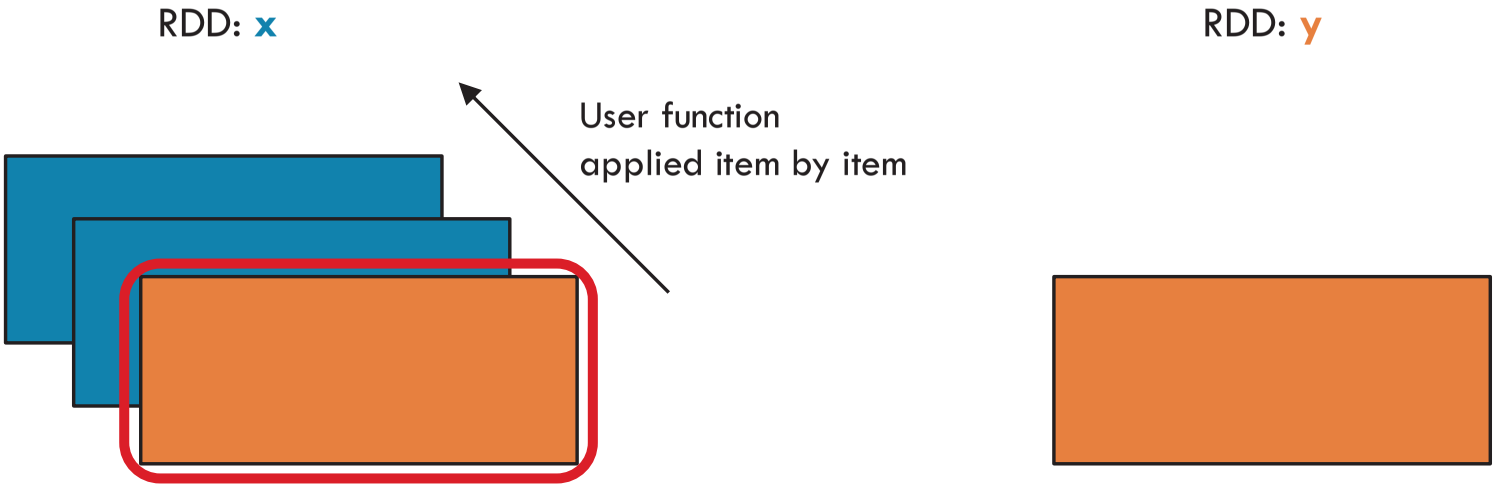
MAP

RDD: **x**



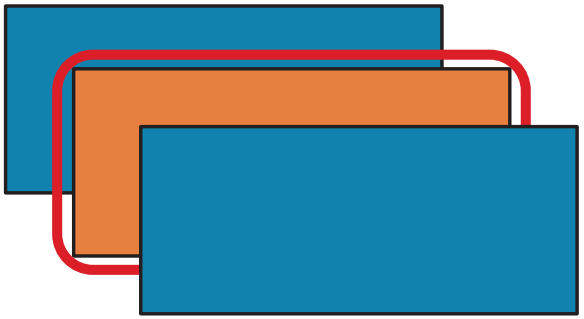
3 items in RDD

MAP



MAP

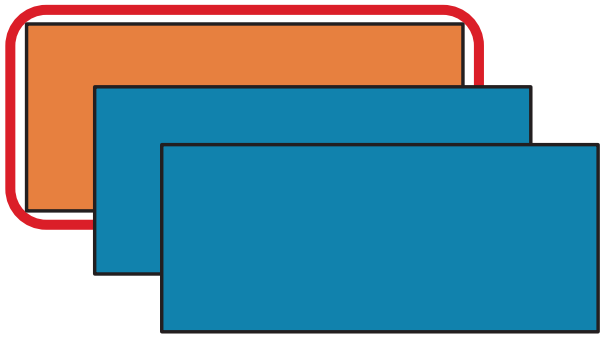
RDD: x



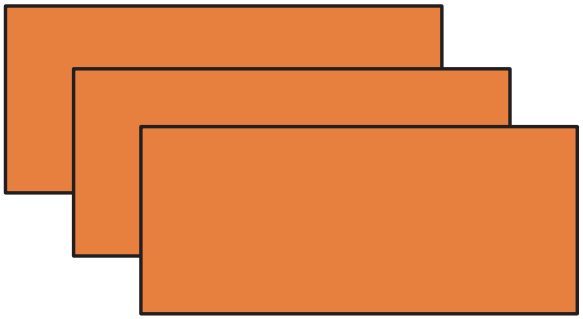
RDD: y

MAP

RDD: **x**



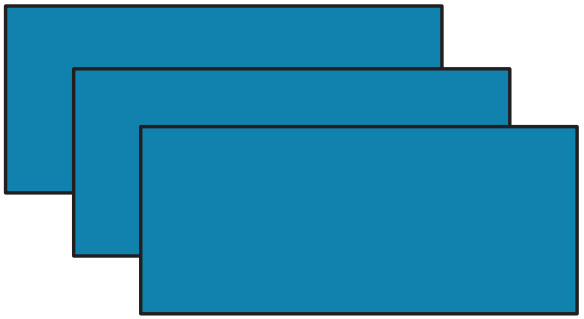
RDD: **y**



MAP

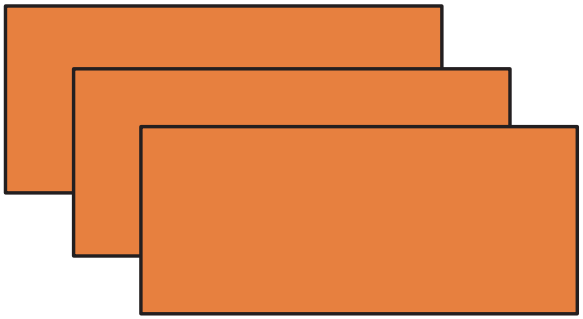
After `map()` has been applied...

RDD: **x**



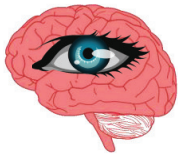
before

RDD: **y**



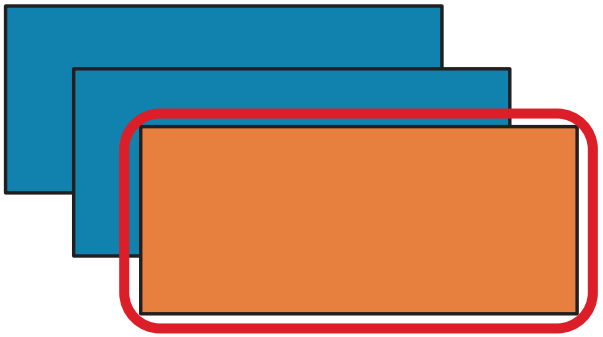
after

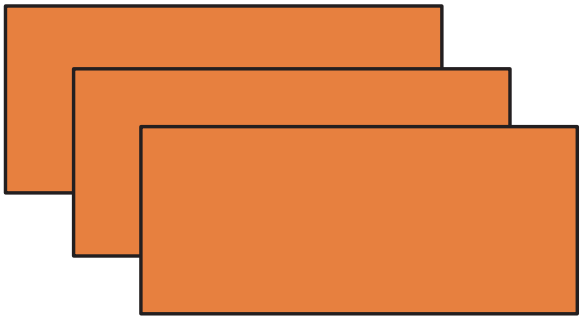
MAP



RDD: **x**

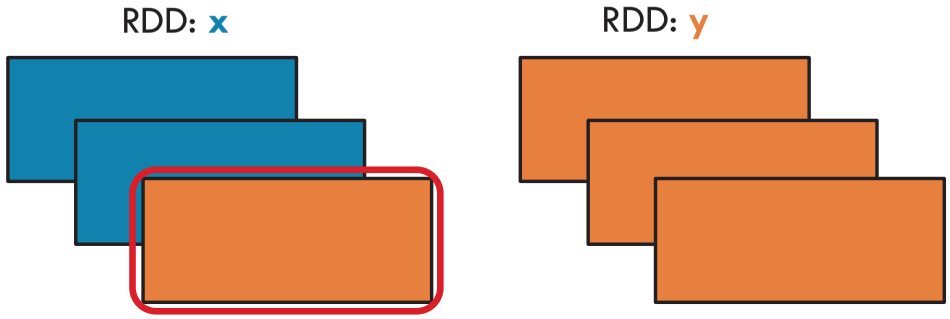
RDD: **y**





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

MAP



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

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

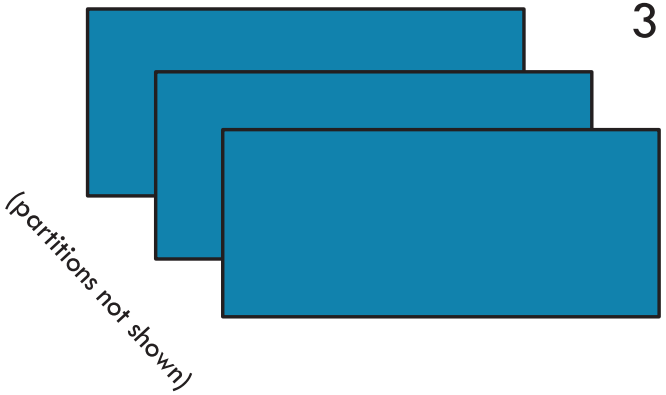
```
x = sc.parallelize(["b", "a", "c"])
y = x.map(lambda z: (z, 1))
print(x.collect())
print(y.collect())
```

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

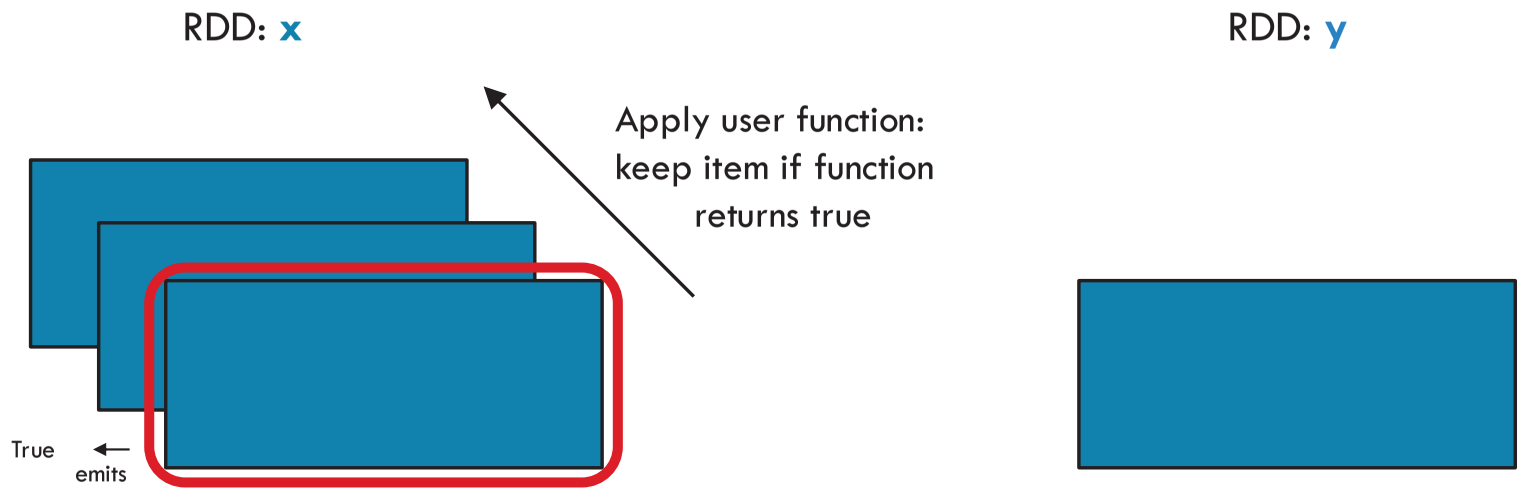
FILTER

RDD: **x**



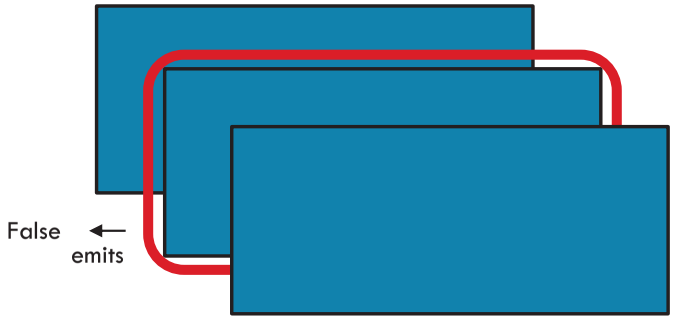
3 items in RDD

FILTER


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

FILTER

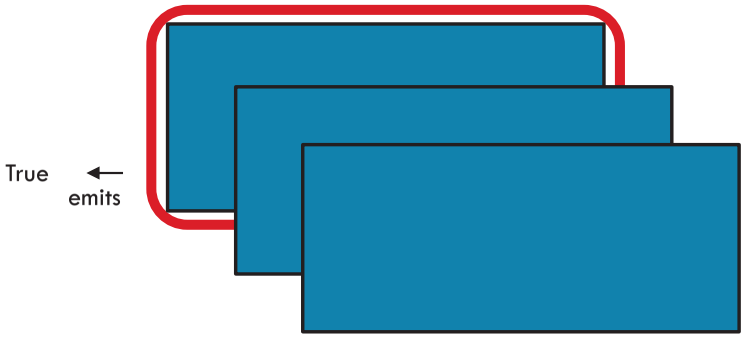
RDD: **x**



RDD: **y**

FILTER

RDD: **x**

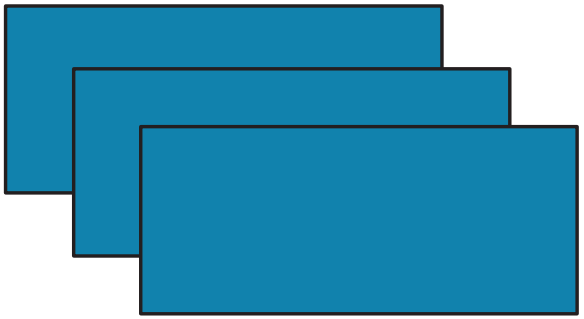


RDD: **y**

FILTER

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

RDD: **x**

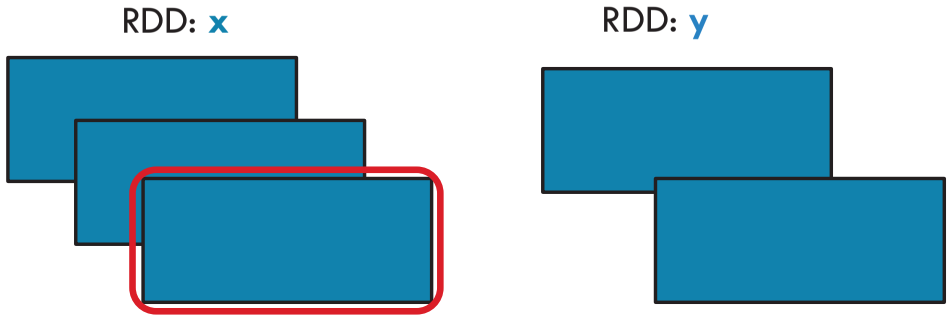


before

RDD: **y**

after

FILTER



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

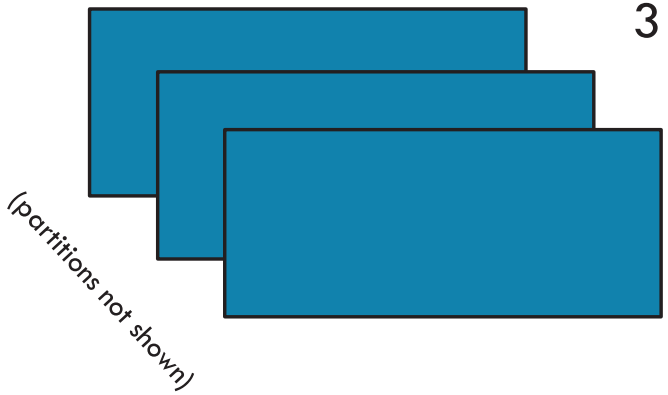
```python
x = sc.parallelize([1,2,3])
y = x.filter(lambda x: x%2 == 1) #keep odd values
print(x.collect())
print(y.collect())
```

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

y: [1, 3]
```

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

FLATMAP

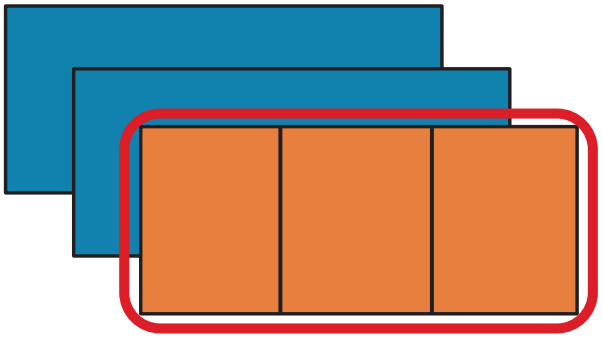
RDD: **x**



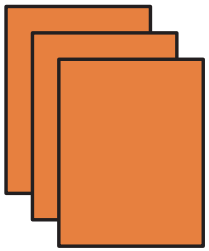
3 items in RDD

FLATMAP

RDD: **x**

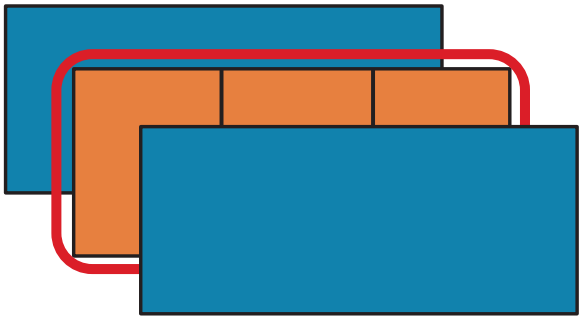


RDD: **y**

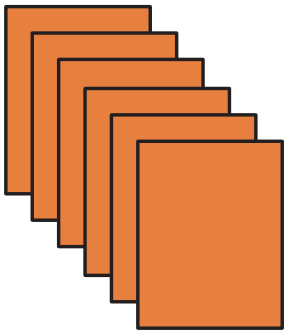


FLATMAP

RDD: **x**

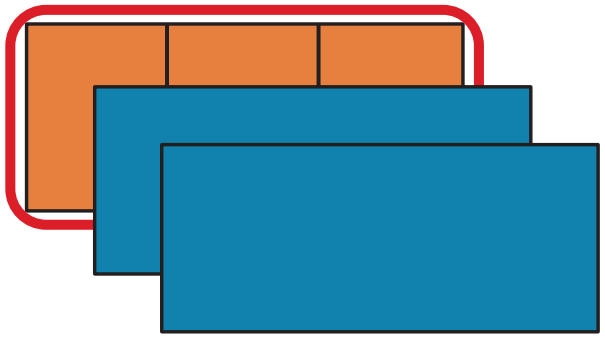


RDD: **y**

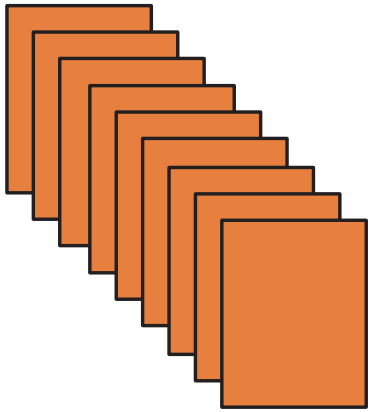


FLATMAP

RDD: x



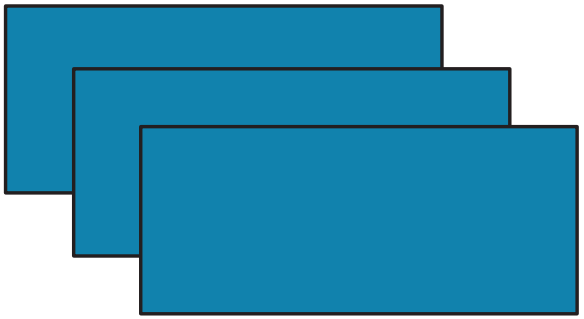
RDD: y



FLATMAP

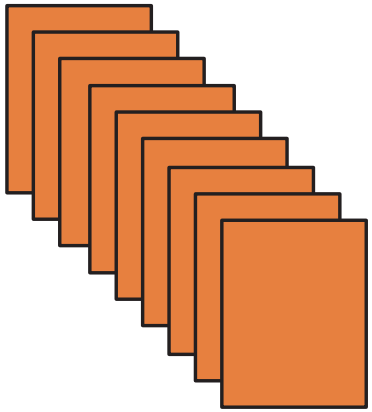
After `flatmap()` has been applied...

RDD: x



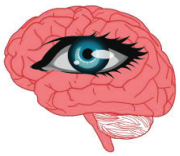
before

RDD: y



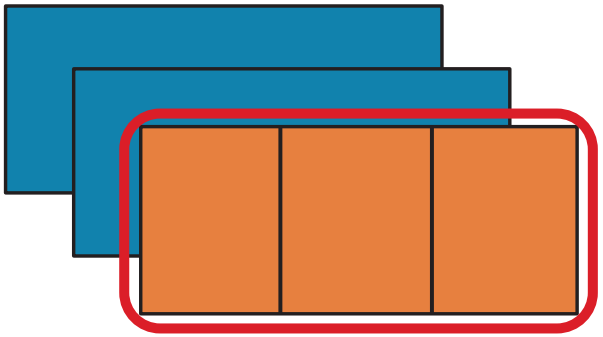
after

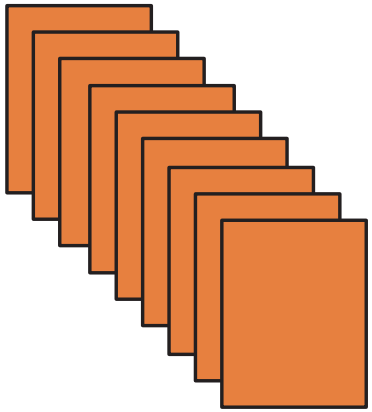
FLATMAP



RDD: **x**

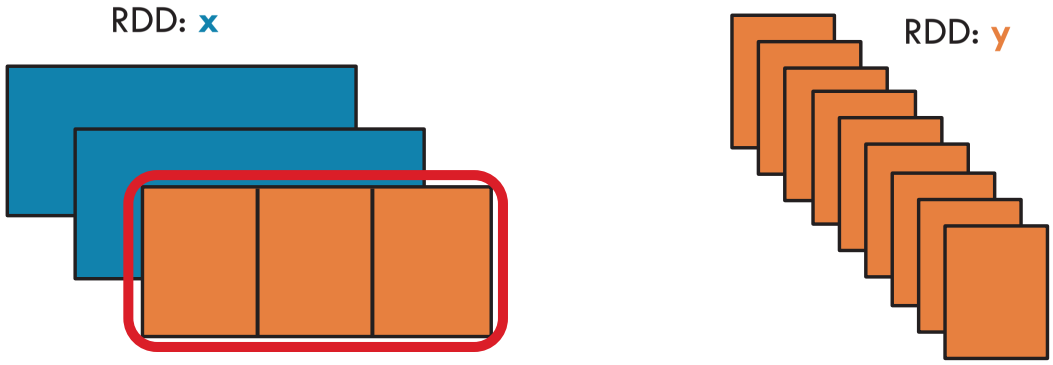
RDD: **y**





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

FLATMAP



`flatMap(f, preservesPartitioning=False)`

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

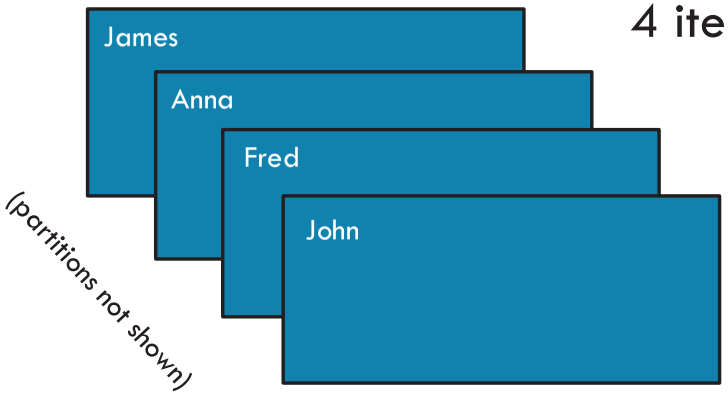
```python
x = sc.parallelize([1,2,3])
y = x.flatMap(lambda x: (x, x*100, 42))
print(x.collect())
print(y.collect())
```

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

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

```scala
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**

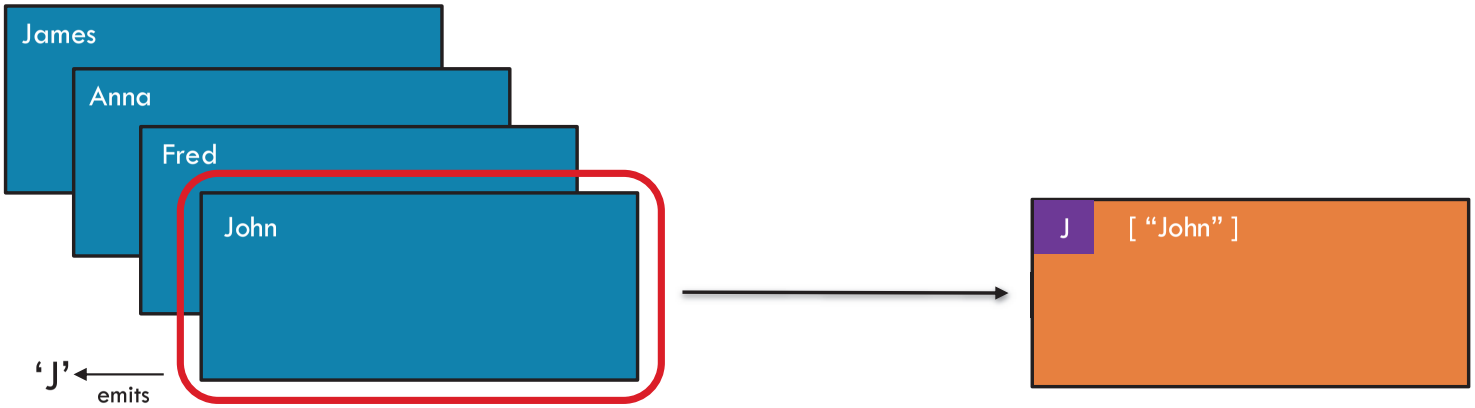


4 items in RDD

GROUPBY

RDD: x

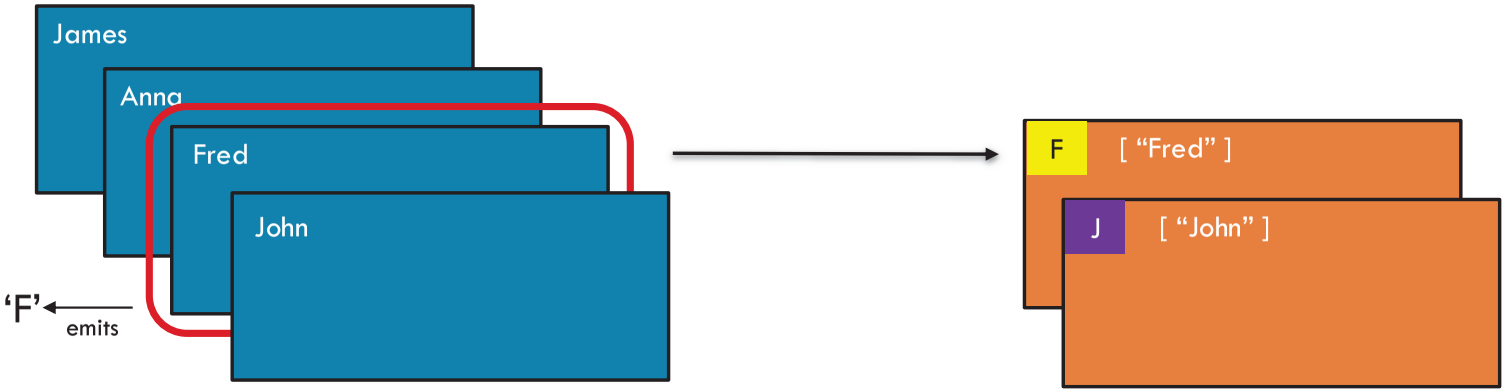
RDD: y



GROUPBY

RDD: x

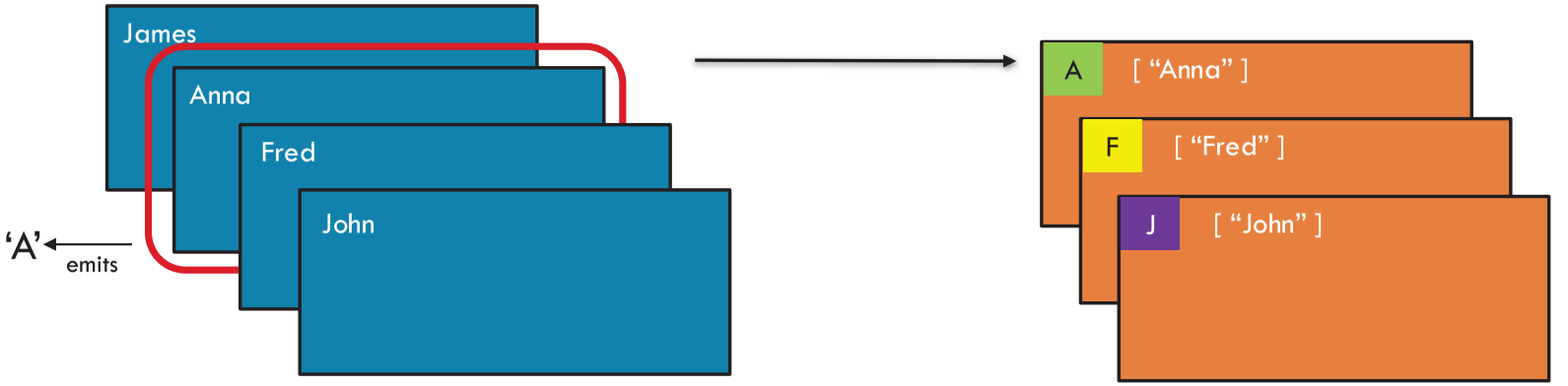
RDD: y



GROUPBY

RDD: x

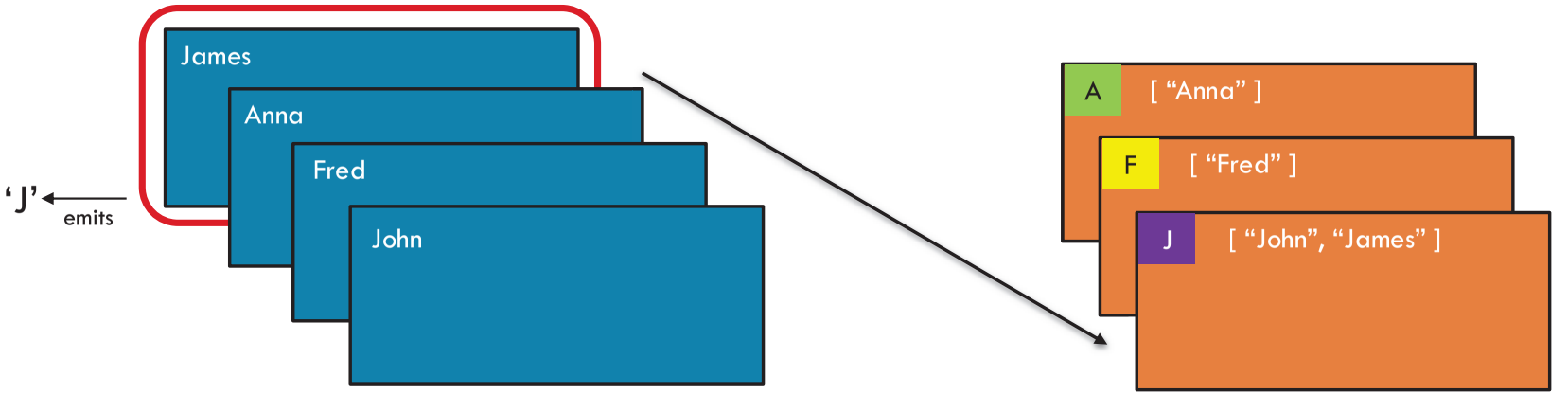
RDD: y



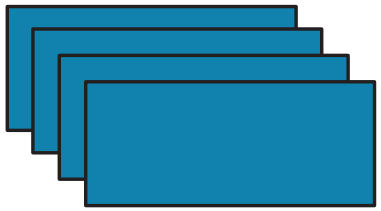
GROUPBY

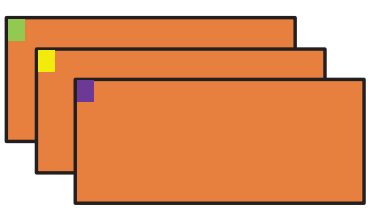
RDD: x

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.

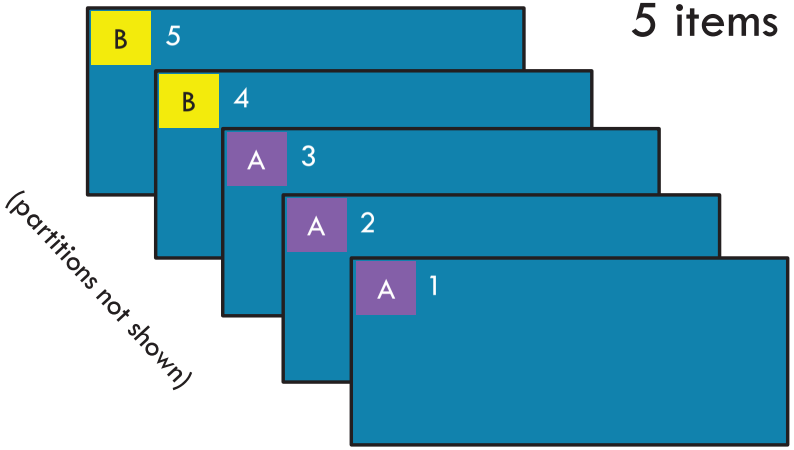
```python
x = sc.parallelize(['John', 'Fred', 'Anna', 'James'])
y = x.groupBy(lambda w: w[0])
print [(k, list(v)) for (k, v) in y.collect()]
```

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

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

```scala
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**

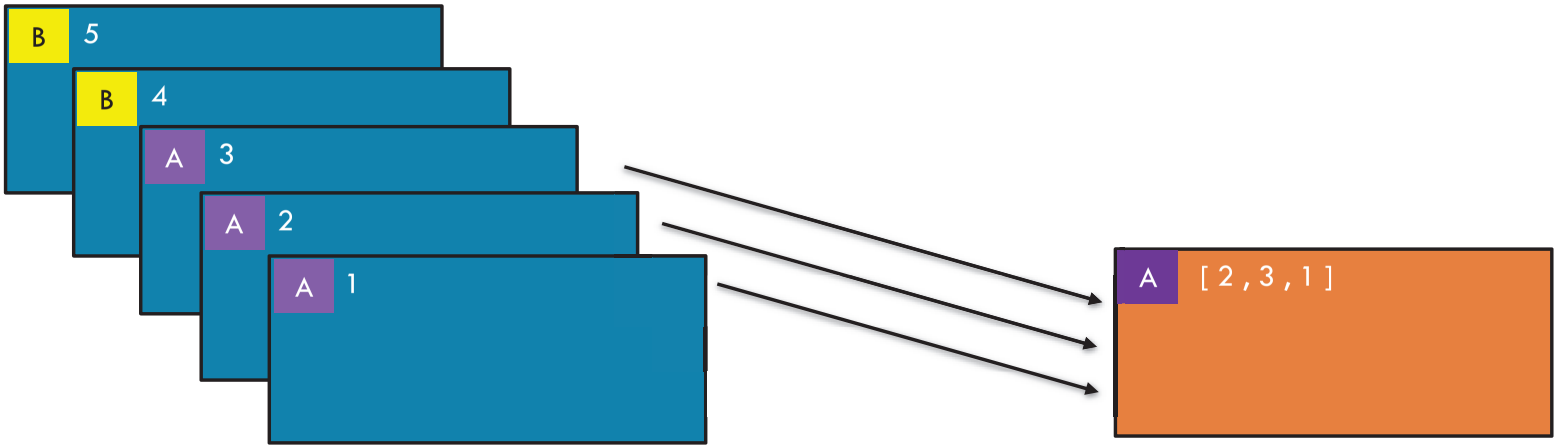


5 items in RDD

GROUPBYKEY

Pair RDD: **x**

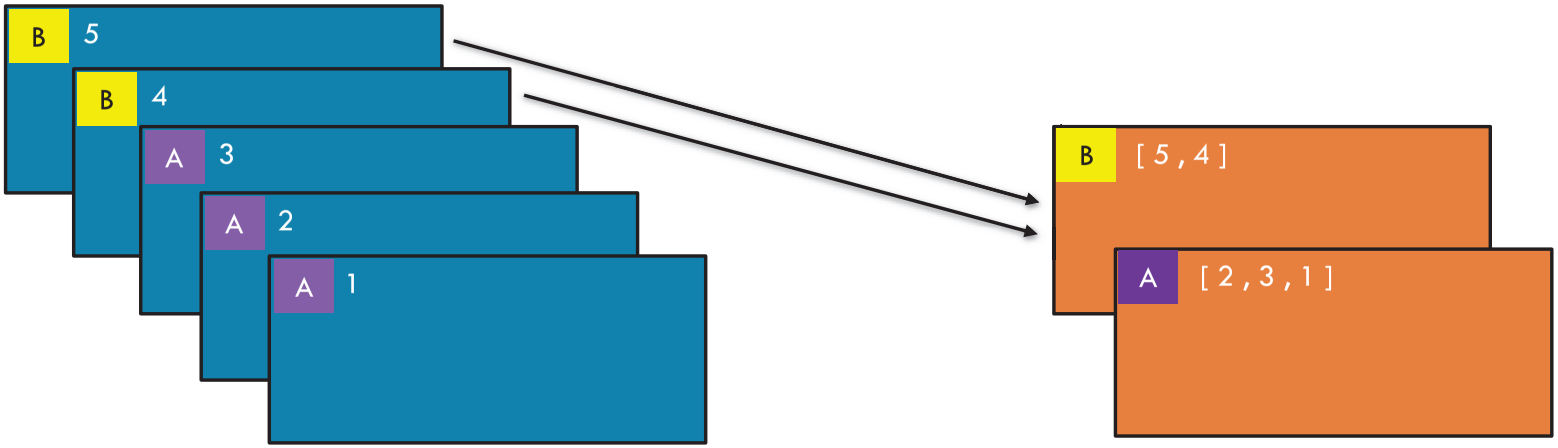
RDD: **y**



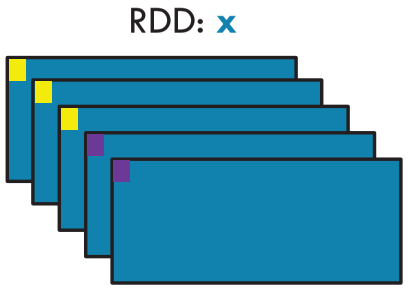
GROUPBYKEY

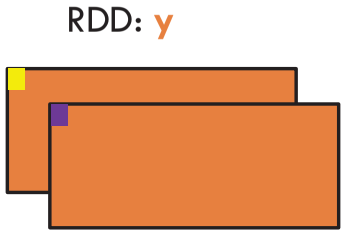
Pair RDD: **x**

RDD: **y**



GROUPBYKEY





groupByKey(*numPartitions=None*)

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

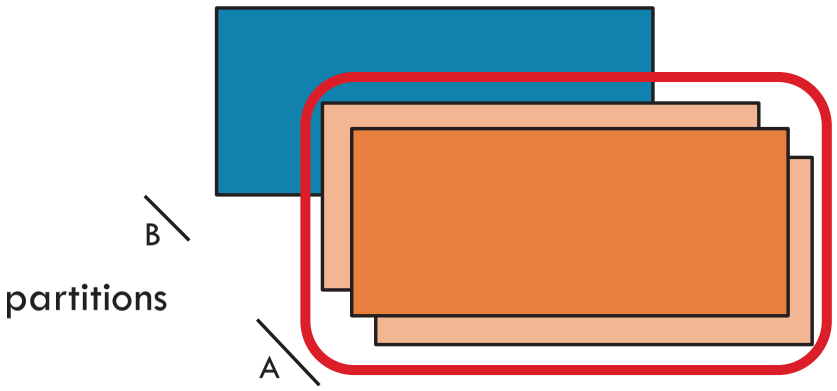
```
x = sc.parallelize([('B',5),('B',4),('A',3),('A',2),('A',1)])
y = x.groupByKey()
print(x.collect())
print(list((j[0], list(j[1])) for j in y.collect()))
```

```
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

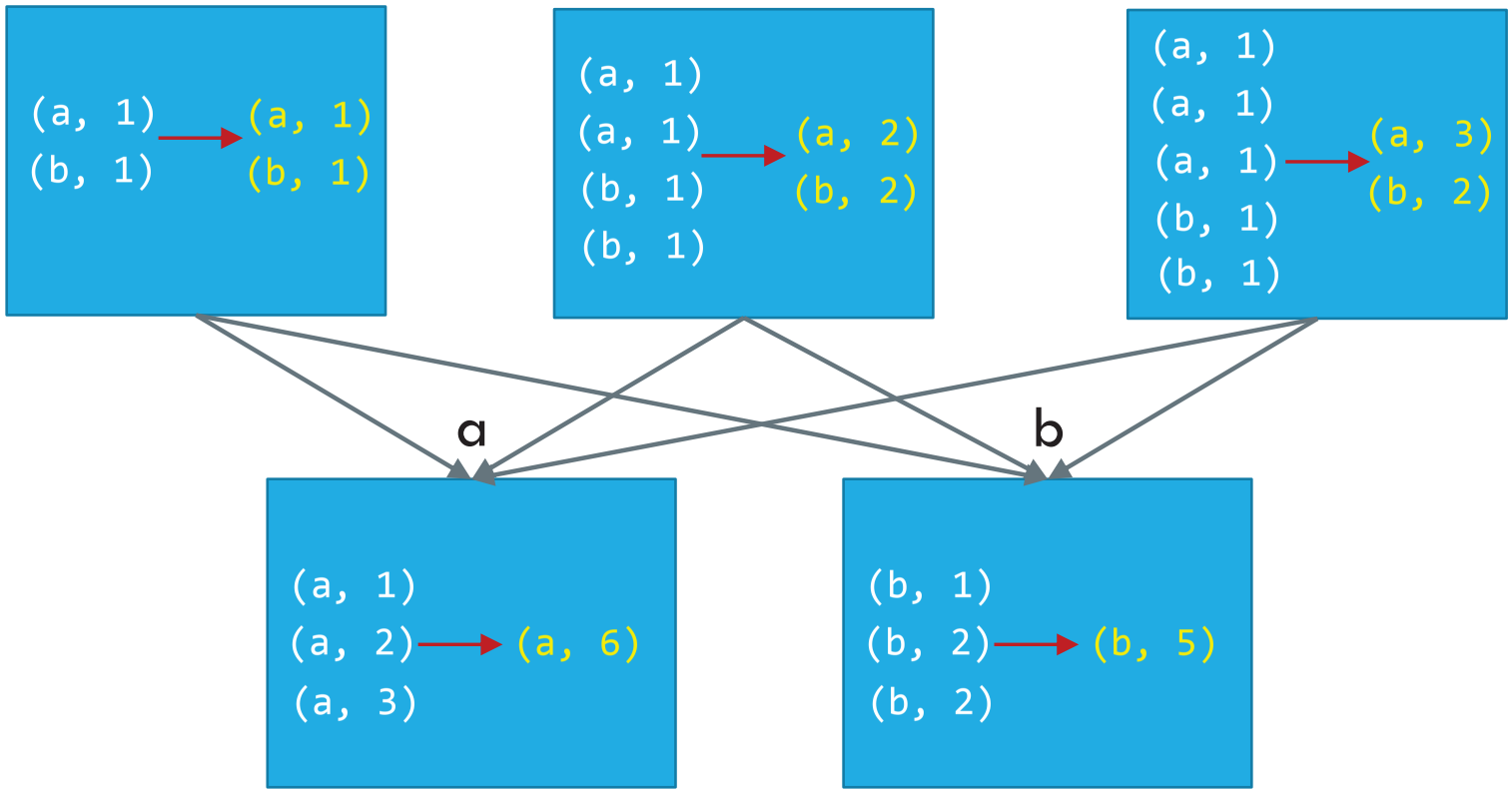


RDD: y

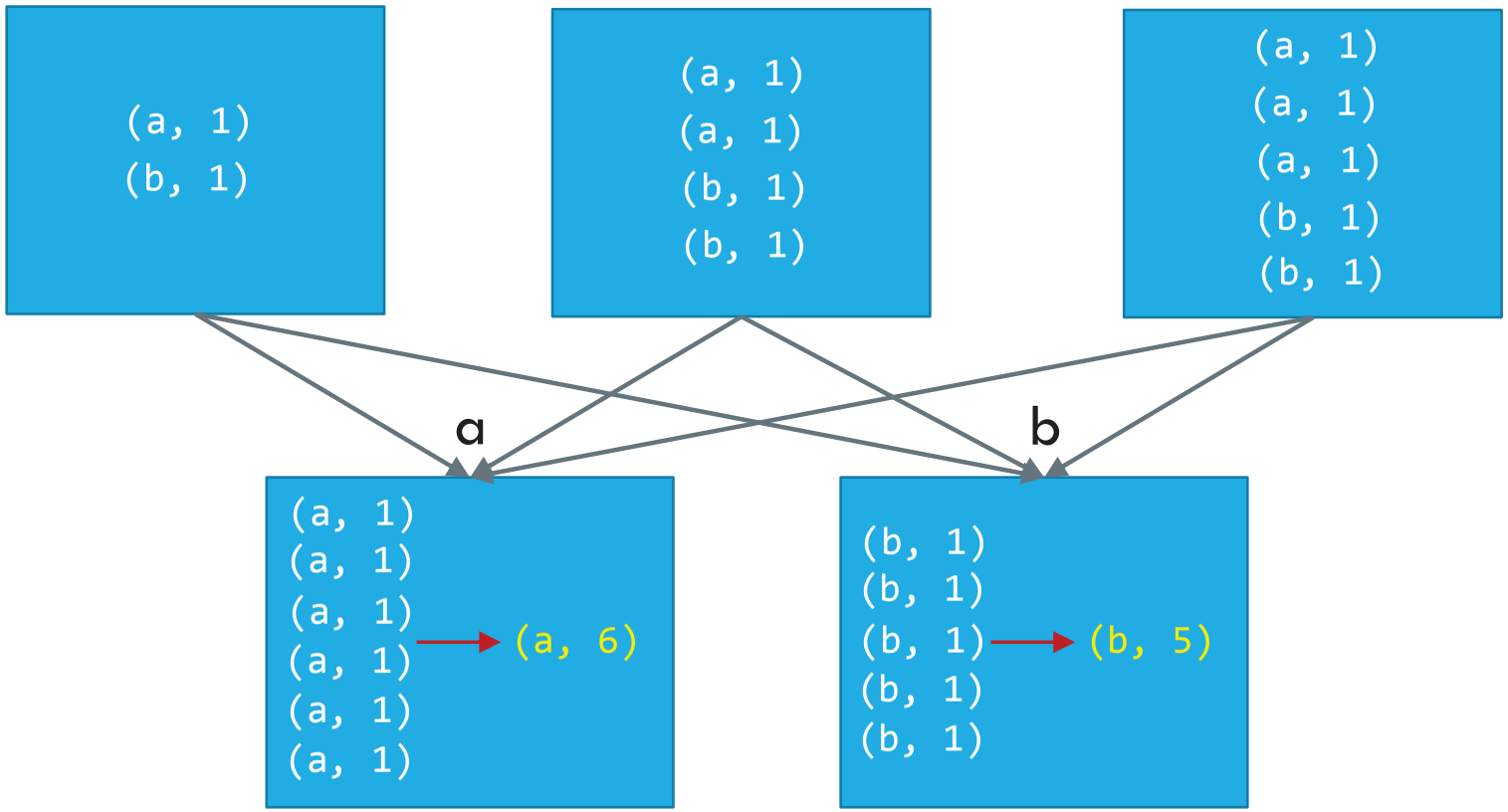
REDUCEBYKEY VS GROUPBYKEY

```
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()
```

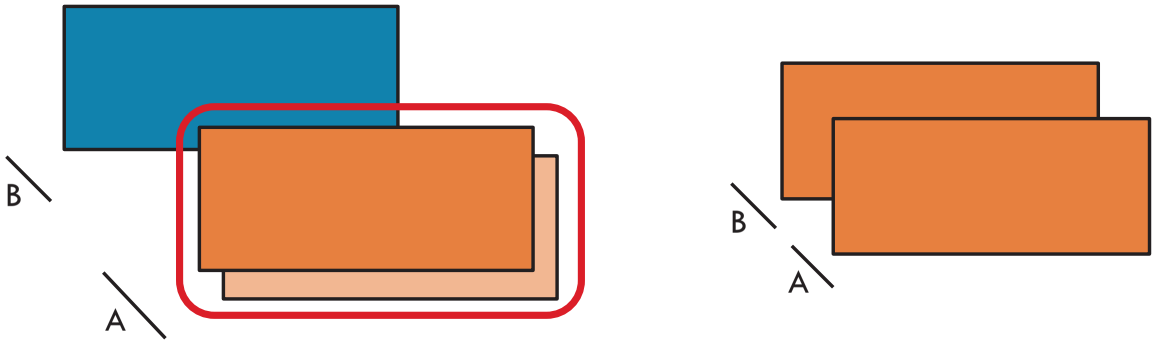
REDUCEBYKEY



GROUPBYKEY


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

MAPPARTITIONS



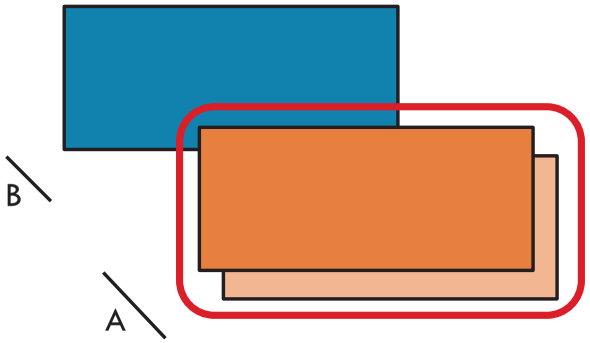
`mapPartitions(f, preservesPartitioning=False)`

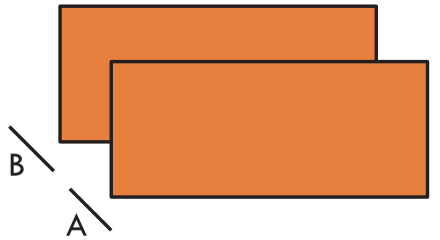
Return a new RDD by applying a function to each partition of this RDD

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

def f(iterator): yield sum(iterator); yield 42

y = x.mapPartitions(f)

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

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

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

MAPPARTITIONS





`mapPartitions(f, preservesPartitioning=False)`

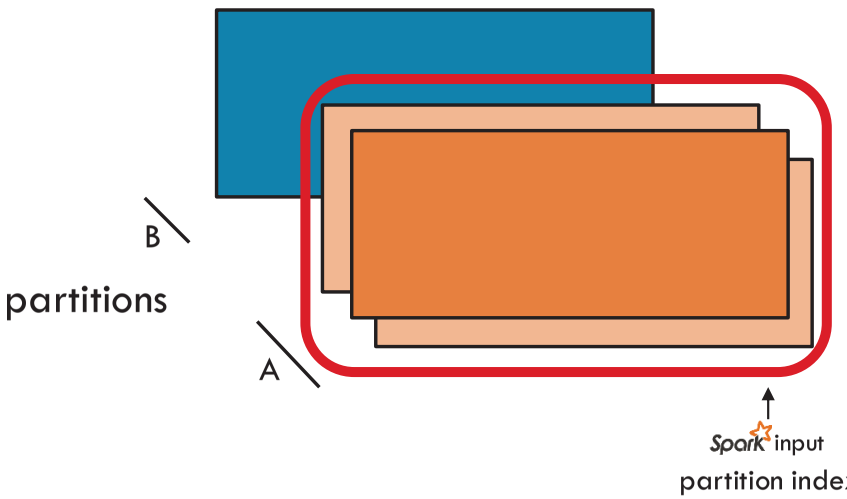
Return a new RDD by applying a function to each partition of this RDD

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

def f(i:Iterator[Int])={ (i.sum,42).productIterator }

val y = x.mapPartitions(f)

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

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

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

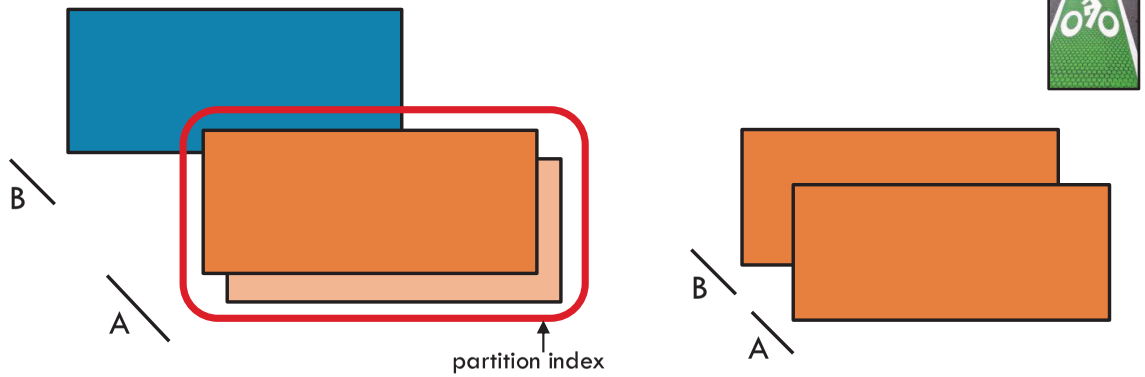
MAPPARTITIONSWITHINDEX

RDD: x



RDD: y

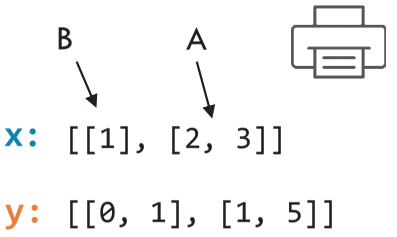
MAPPARTITIONSWITHINDEX



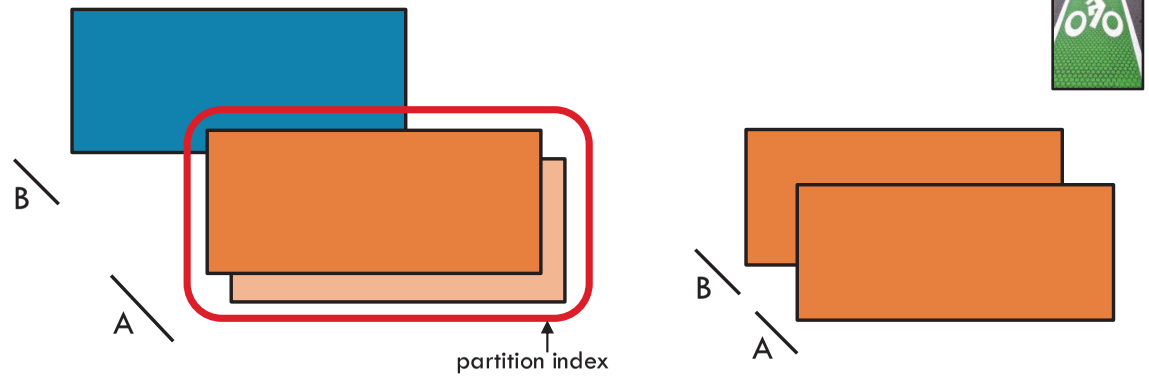

```
mapPartitionsWithIndex(f, preservesPartitioning=False)
```

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

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

def f(partitionIndex, iterator): yield (partitionIndex, sum(iterator))

y = x.mapPartitionsWithIndex(f)

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



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.

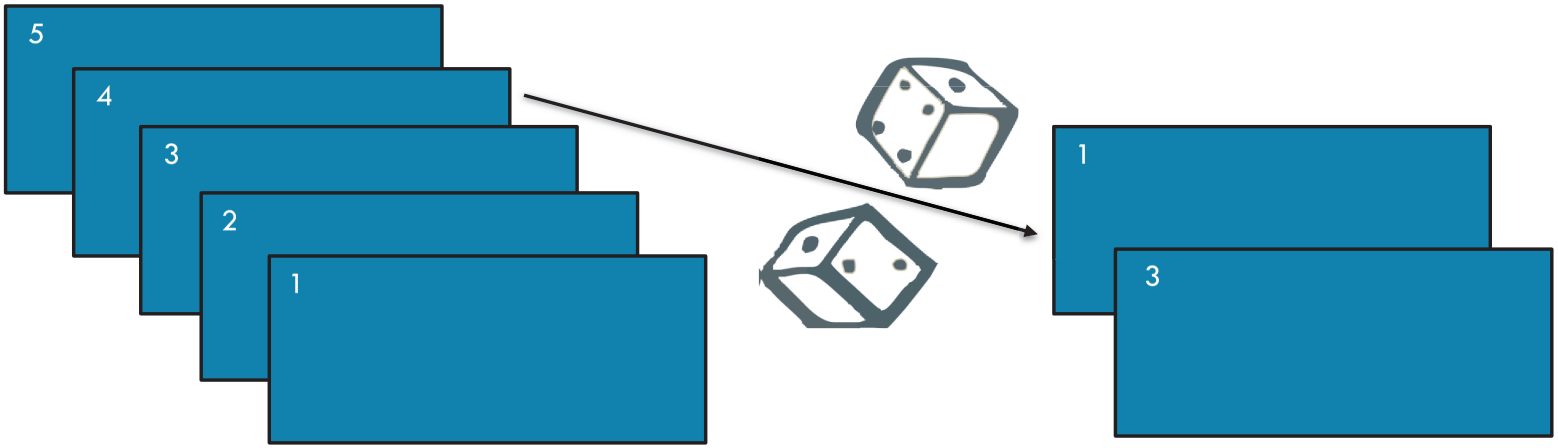
```
val x = sc.parallelize(Array(1,2,3), 2)

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))
```

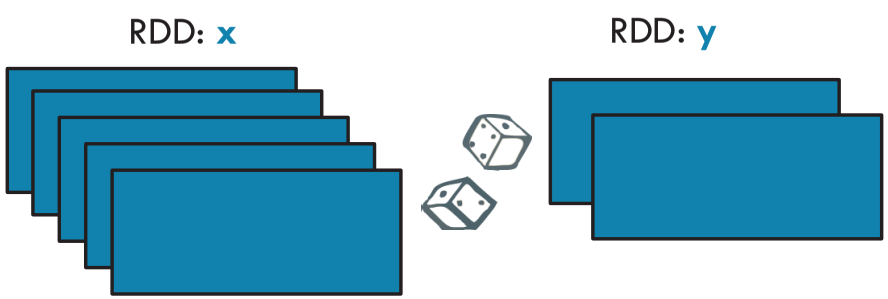
SAMPLE

RDD: **x**

RDD: **y**



SAMPLE



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

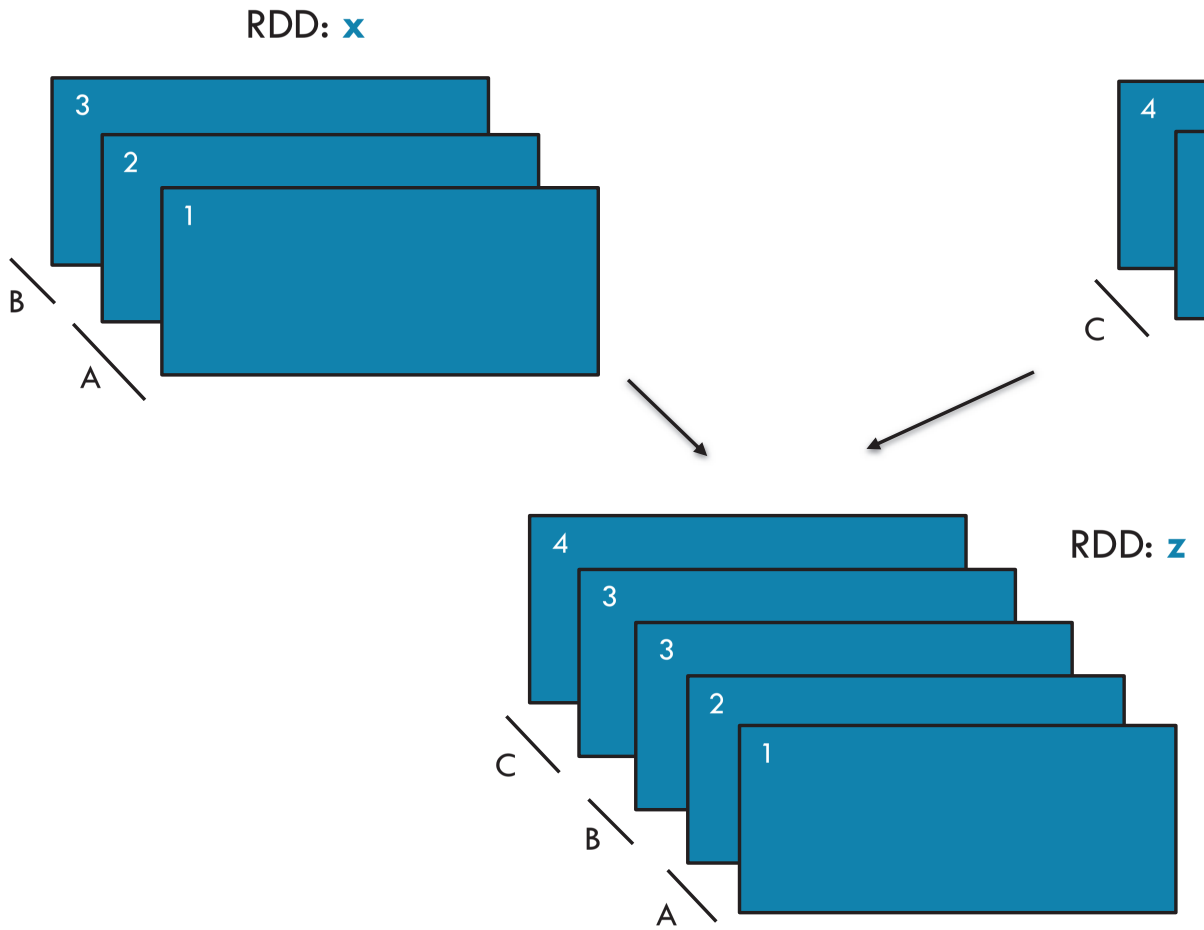
Return a new RDD containing a statistical sample of the original RDD

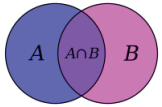
```python
x = sc.parallelize([1, 2, 3, 4, 5])
y = x.sample(False, 0.4, 42)
print(x.collect())
print(y.collect())
```

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

y: [1, 3]
```

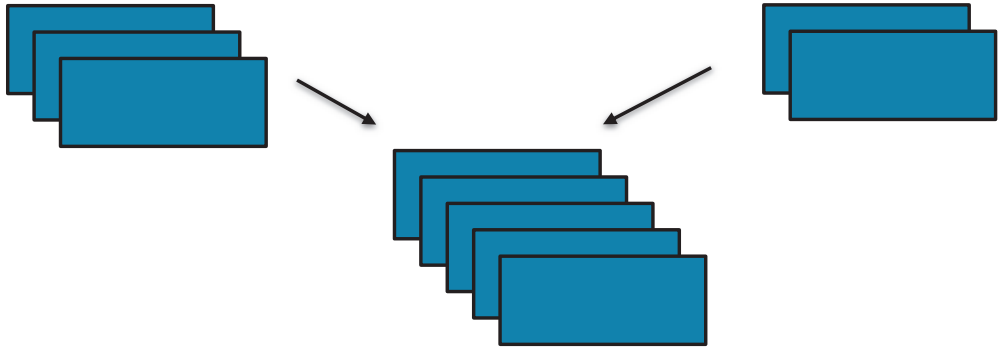
```scala
val x = sc.parallelize(Array(1, 2, 3, 4, 5))
val y = x.sample(false, 0.4)

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

UNION





UNION



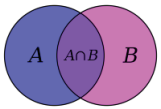
Return a new RDD containing all items from two original RDDs. Duplicates are *not* culled.

`union(otherRDD)`

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

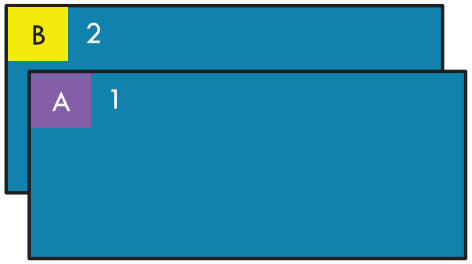
```scala
val x = sc.parallelize(Array(1,2,3), 2)
val y = sc.parallelize(Array(3,4), 1)
val z = x.union(y)
val zOut = z.glom().collect()
```

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

y: [3, 4]

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

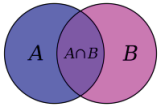


JOIN

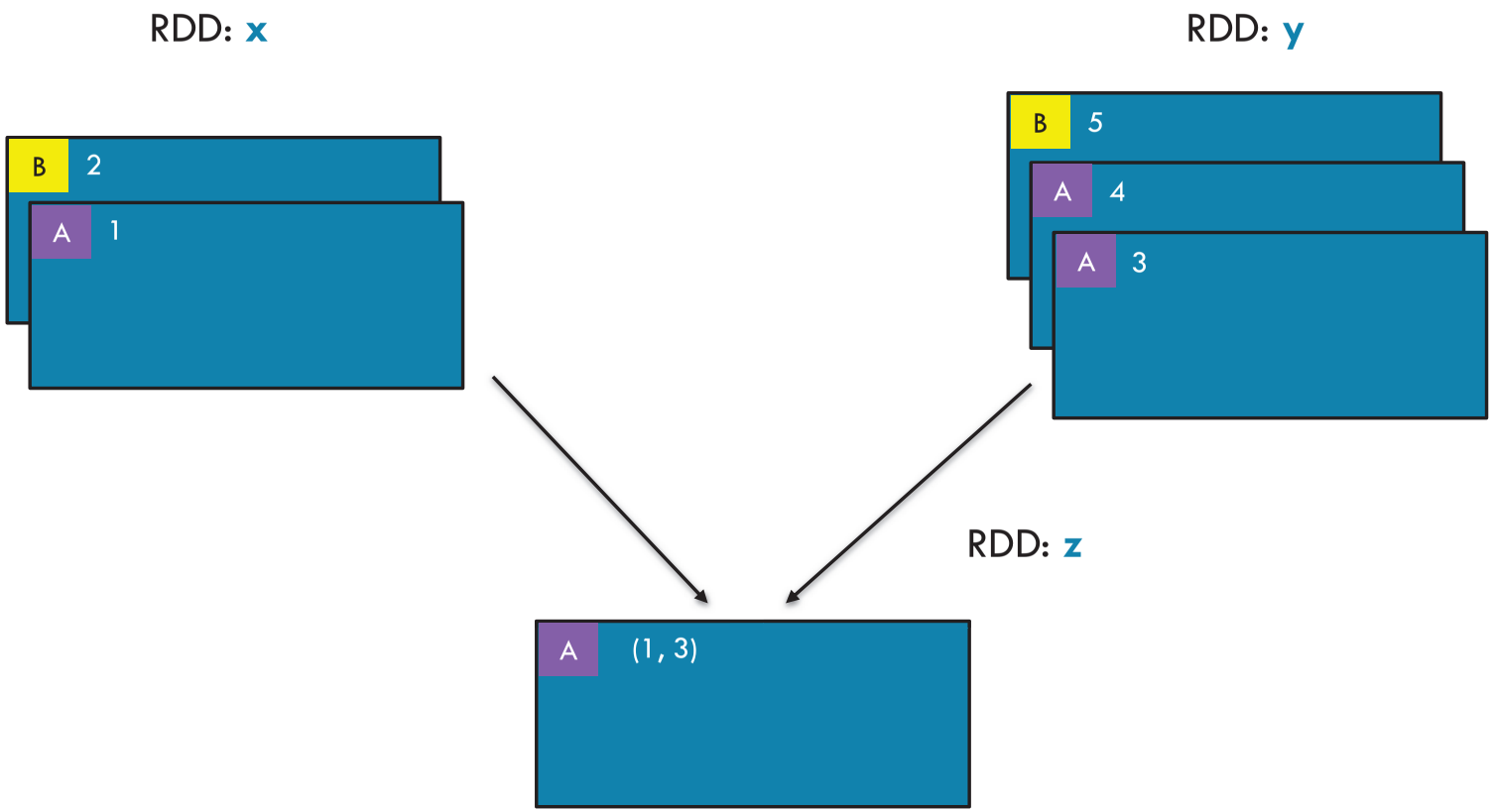
RDD: **x**

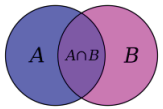


RDD: **y**

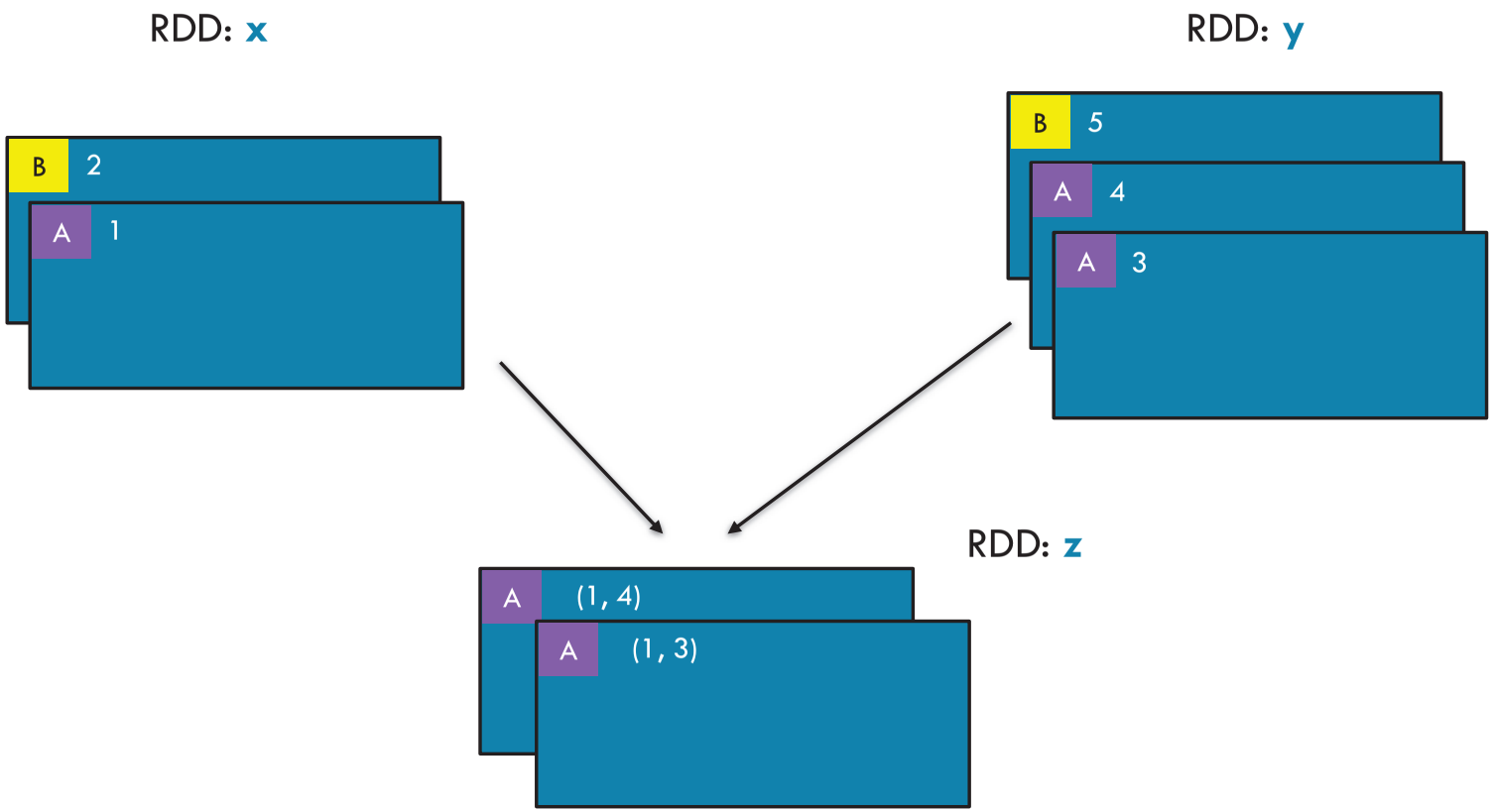


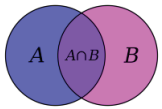
JOIN



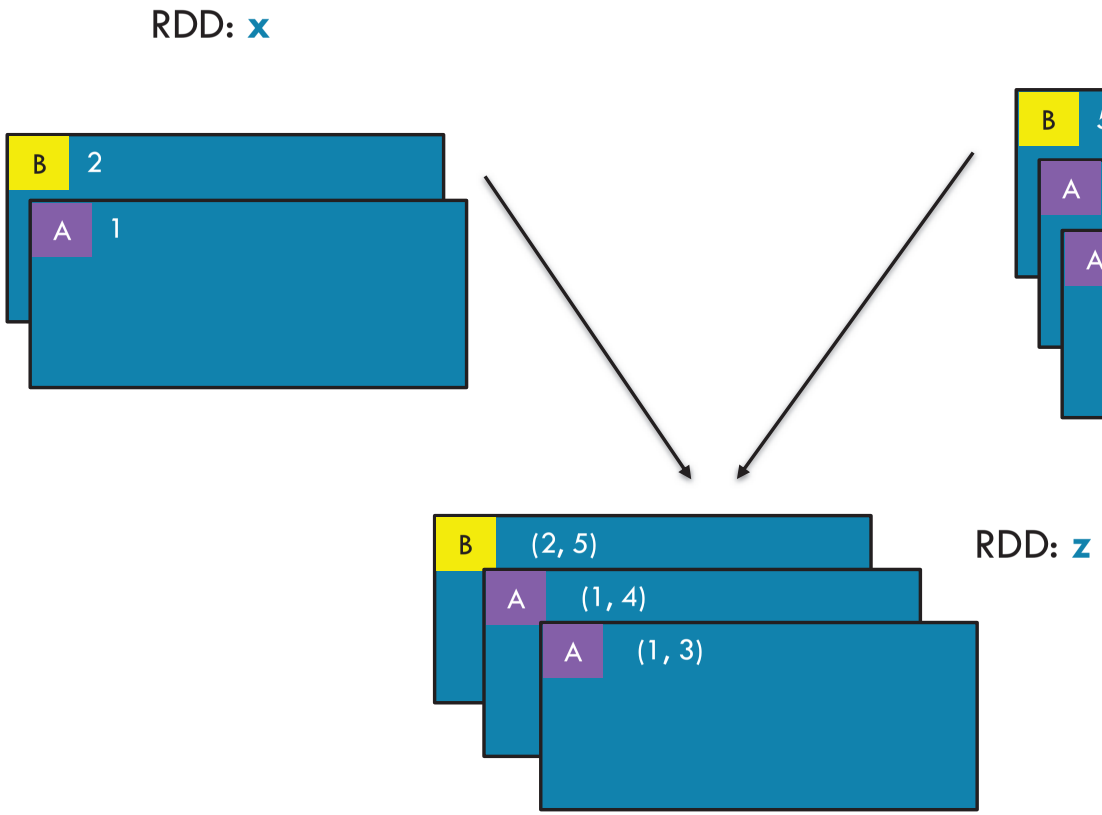


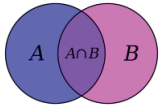
JOIN



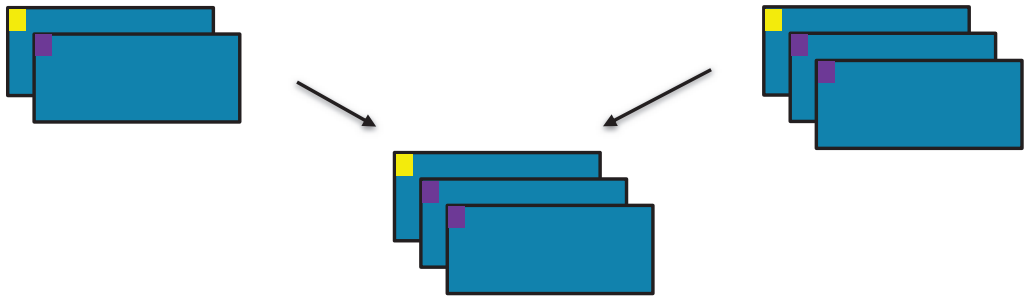


JOIN





JOIN



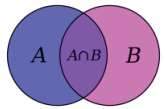
Return a new RDD containing all pairs of elements having the same key in the original RDDs

union(*otherRDD, numPartitions=None*)

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

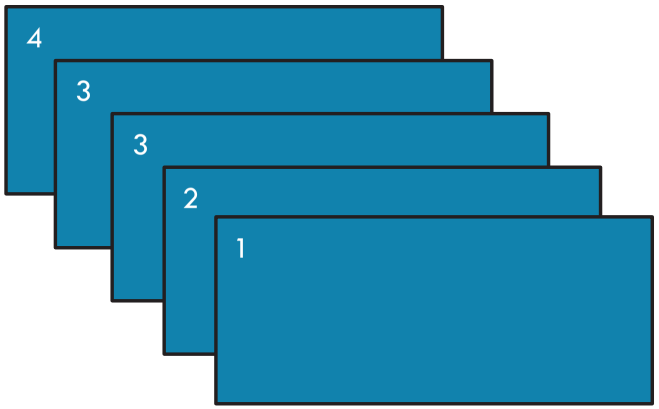
```
val x = sc.parallelize(Array(("a", 1), ("b", 2)))
val y = sc.parallelize(Array(("a", 3), ("a", 4), ("b", 5)))
val z = x.join(y)
println(z.collect().mkString(", "))
```

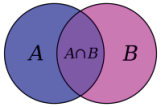
```
x: [("a", 1), ("b", 2)]

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

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



DISTINCT

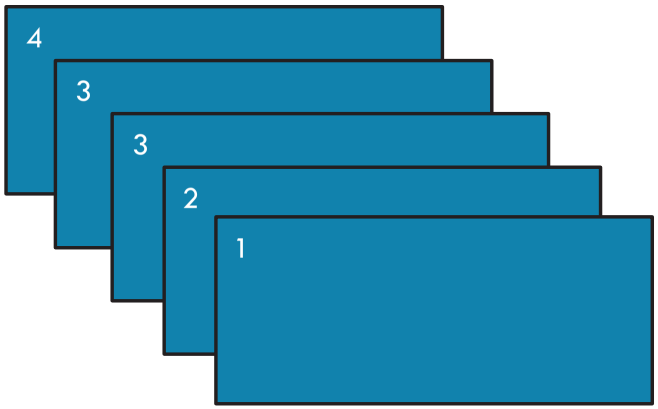
RDD: **x**



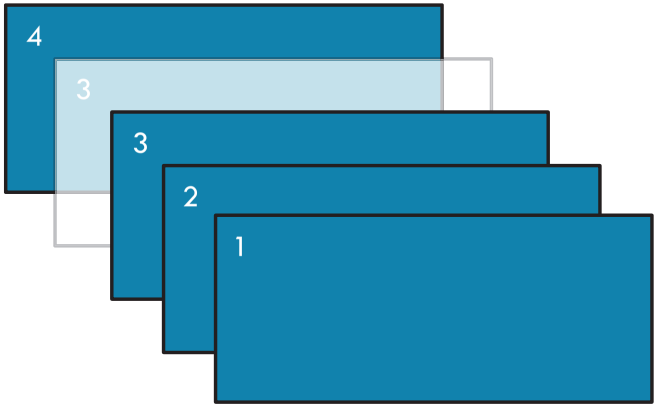


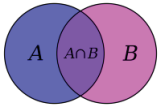
DISTINCT

RDD: **x**



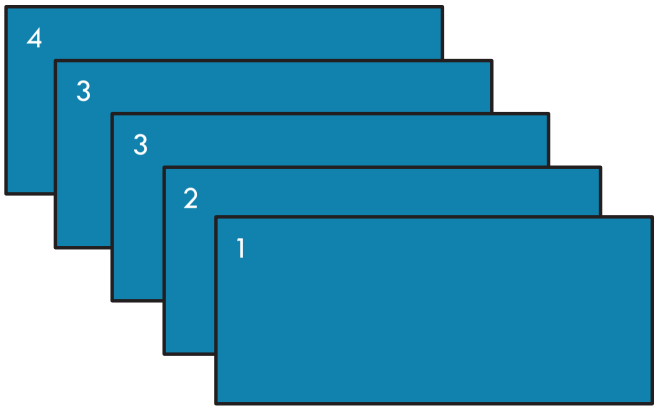
RDD: **y**



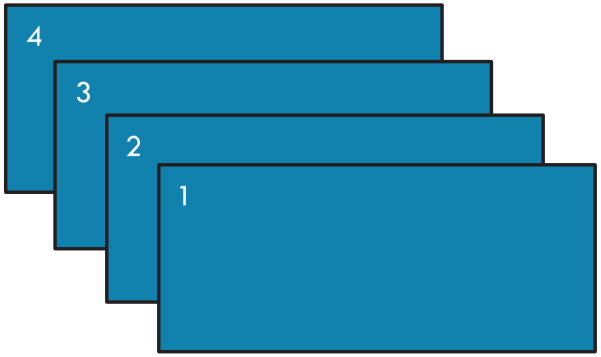


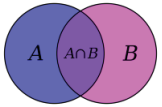
DISTINCT

RDD: **x**

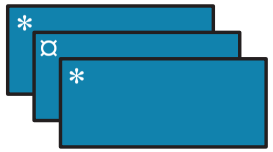


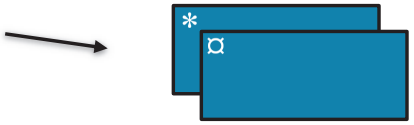
RDD: **y**





DISTINCT





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

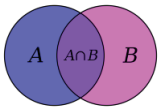
distinct(*numPartitions=None*)

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

print(y.collect())
```

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

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

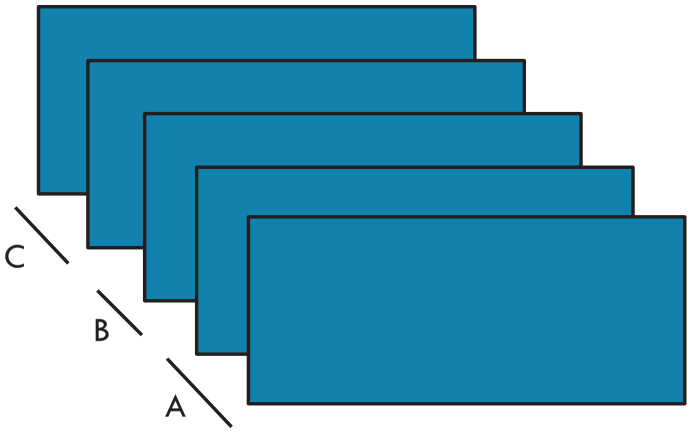
x: [1, 2, 3, 3, 4]

y: [1, 2, 3, 4]



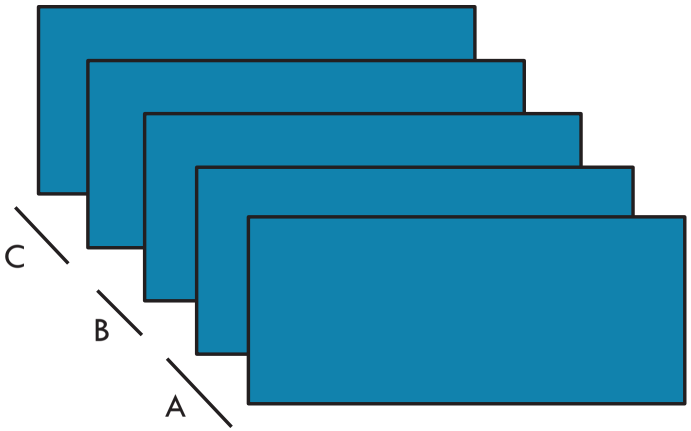
COALESCE

RDD: **x**

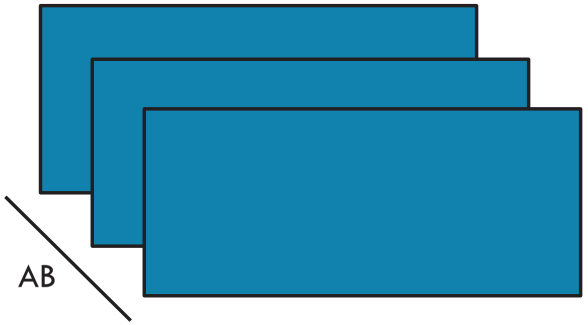


COALESCE

RDD: **x**

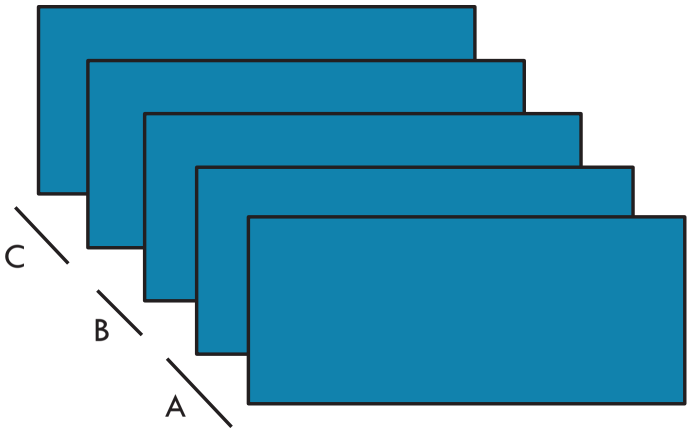


RDD: **y**

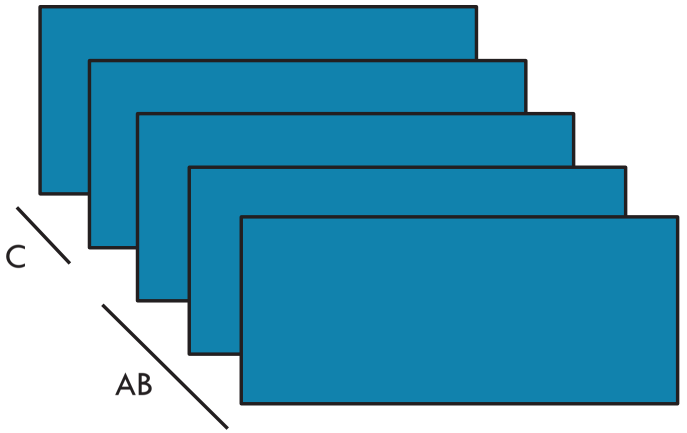


COALESCE

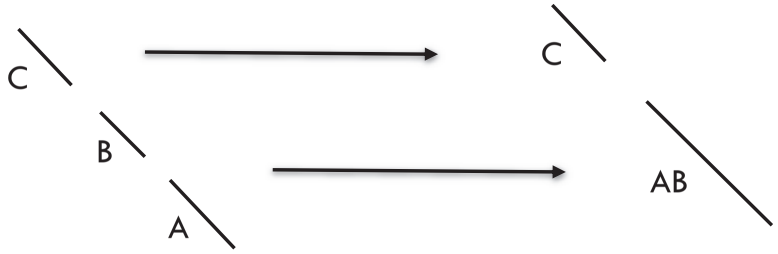
RDD: **x**



RDD: **y**



COALESCE



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

coalesce(*numPartitions, shuffle=False*)

```
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]]

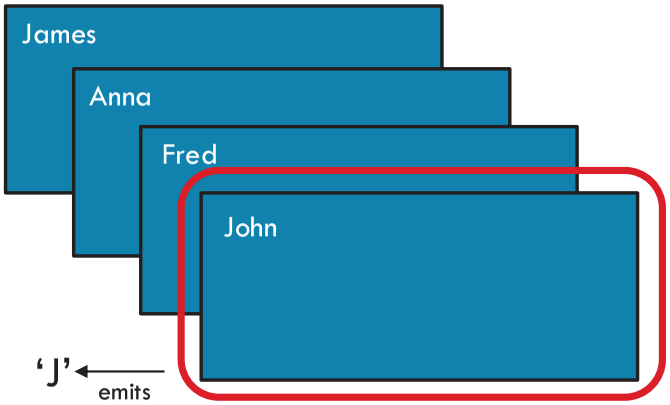
y: [[1], [2, 3, 4, 5]]

```
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

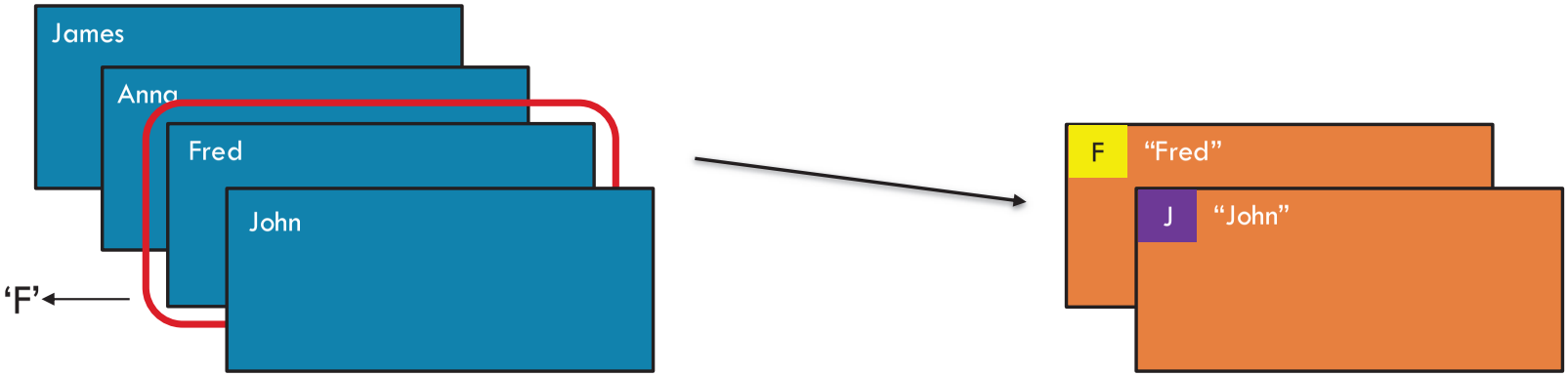
RDD: y



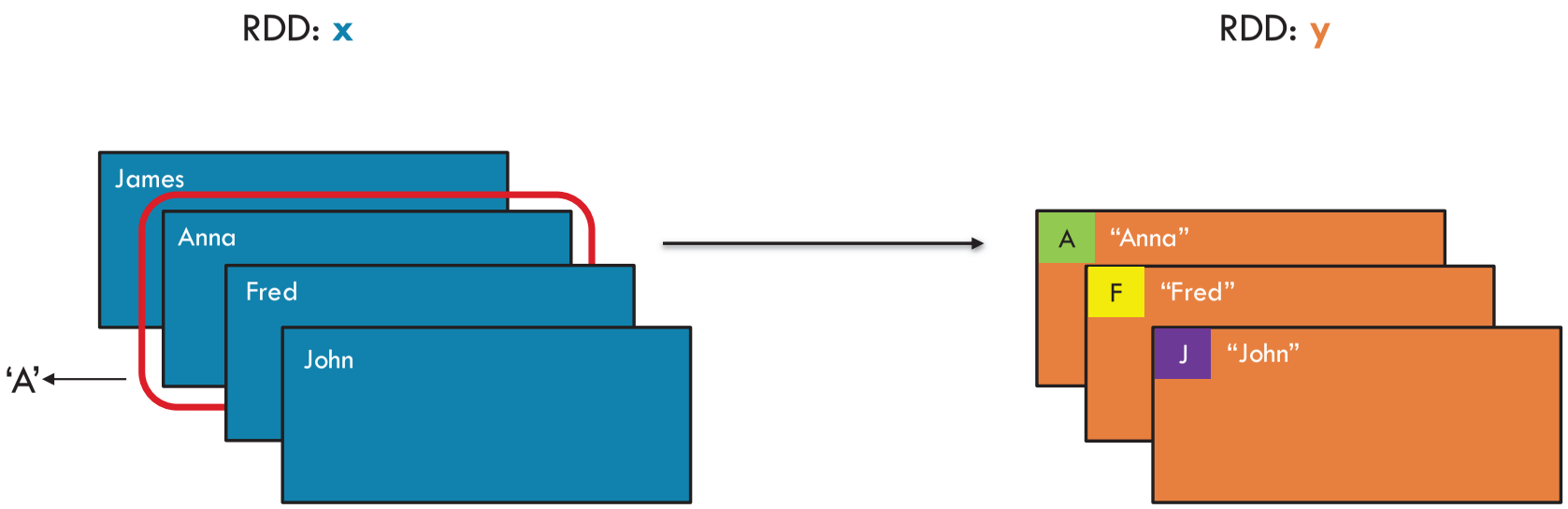
KEYBY

RDD: x

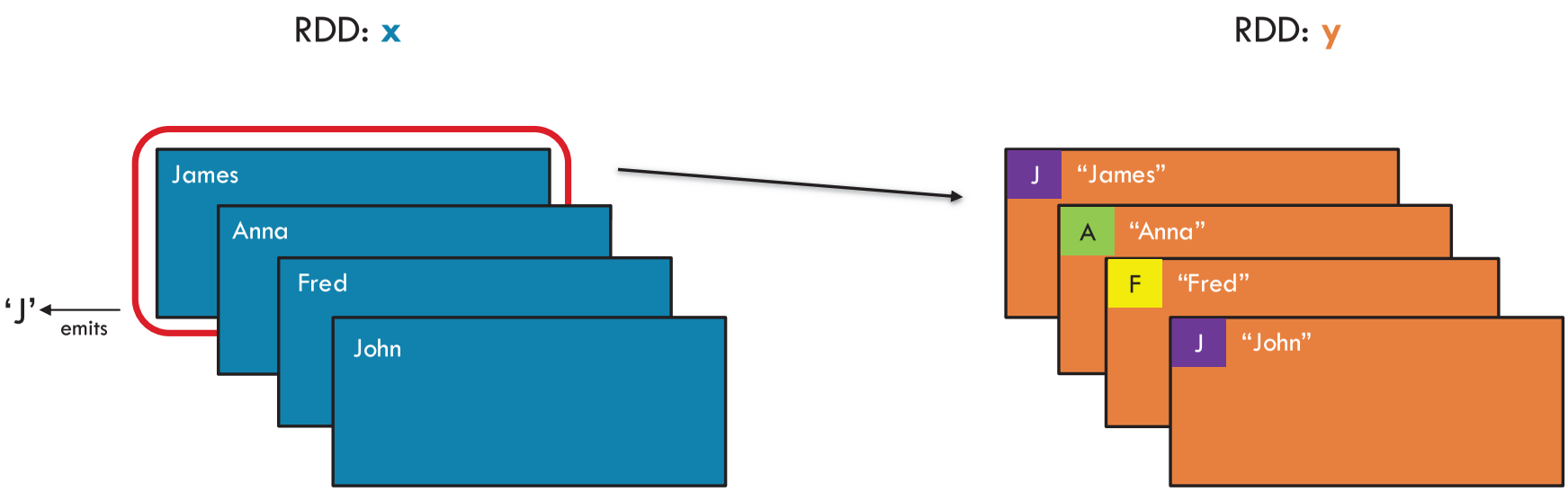
RDD: y



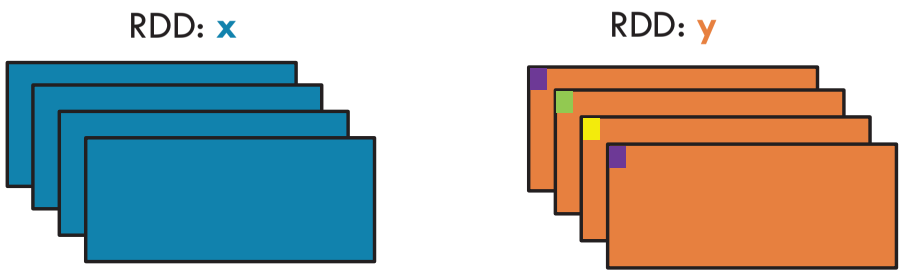
KEYBY



KEYBY


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

KEYBY



keyBy(*f*)

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

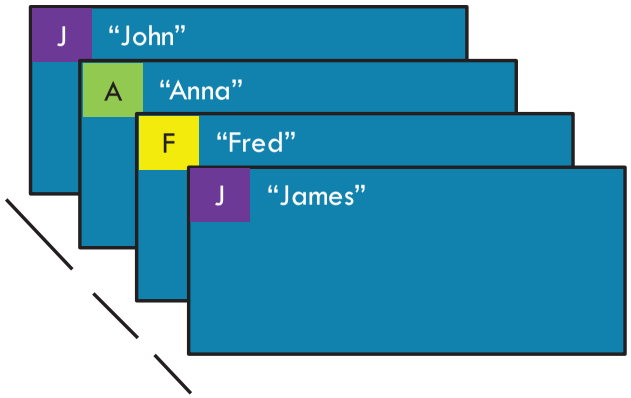
```
x = sc.parallelize(['John', 'Fred', 'Anna', 'James'])
y = x.keyBy(lambda w: w[0])
print y.collect()
```

```
val x = sc.parallelize(
     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')]
```

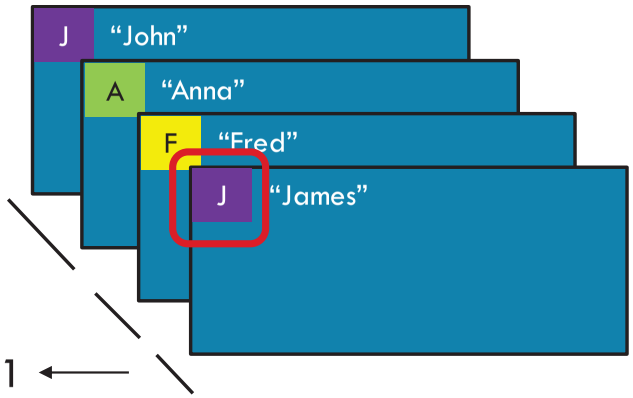
PARTITIONBY

RDD: **x**

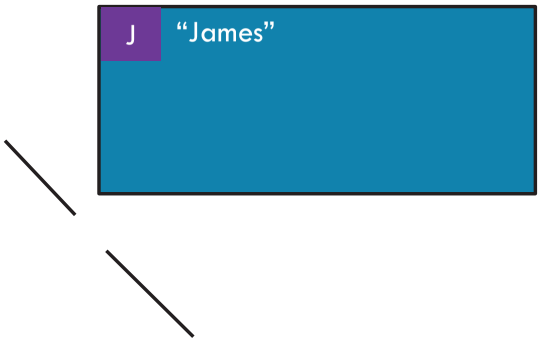


PARTITIONBY

RDD: **x**

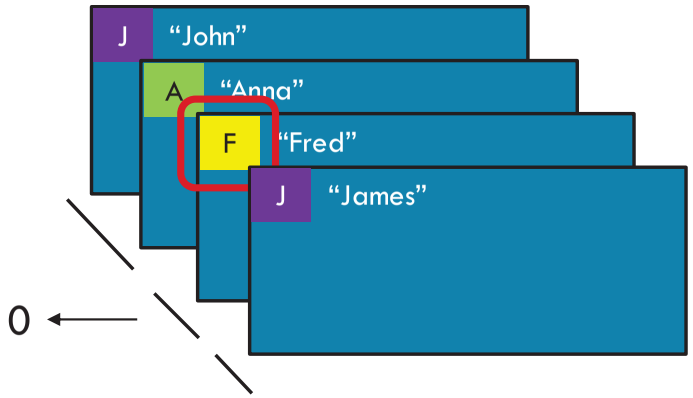


RDD: **y**

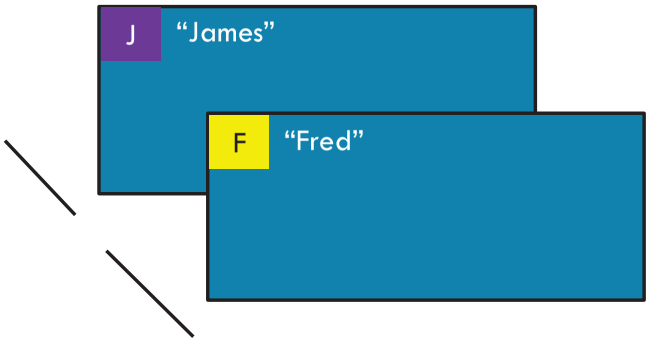


PARTITIONBY

RDD: **x**

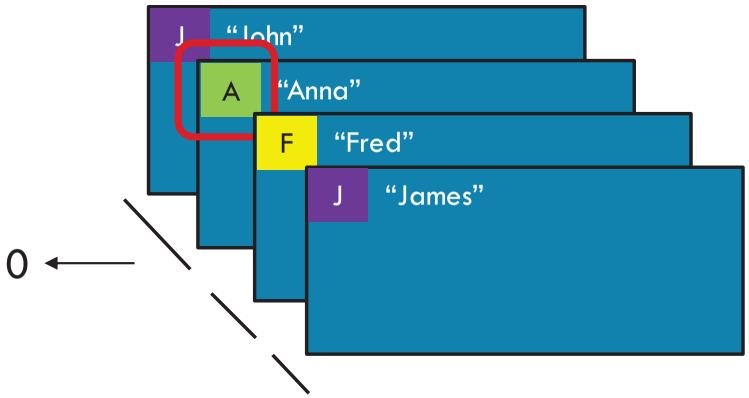


RDD: **y**

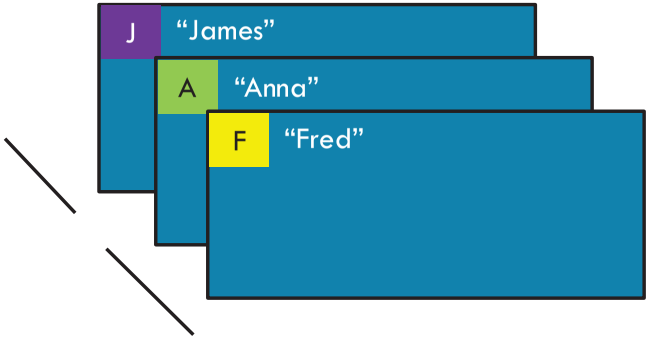


PARTITIONBY

RDD: **x**

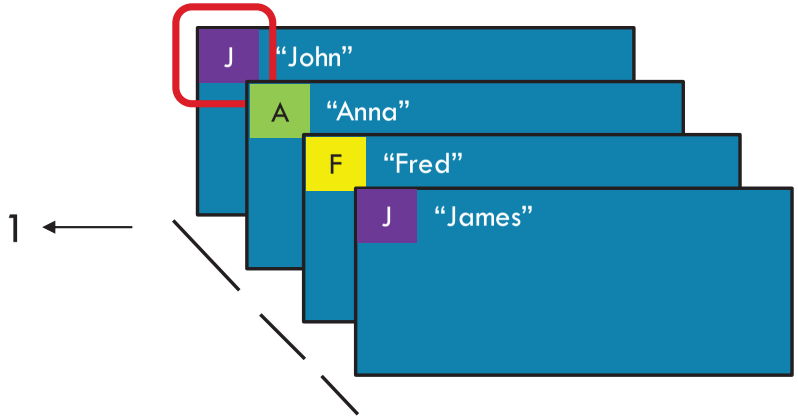


RDD: **y**

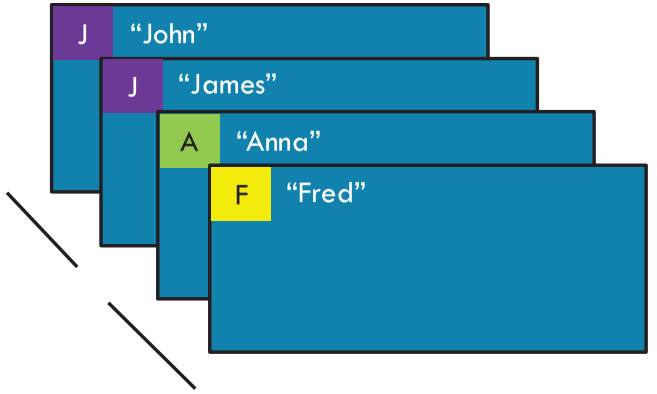


PARTITIONBY

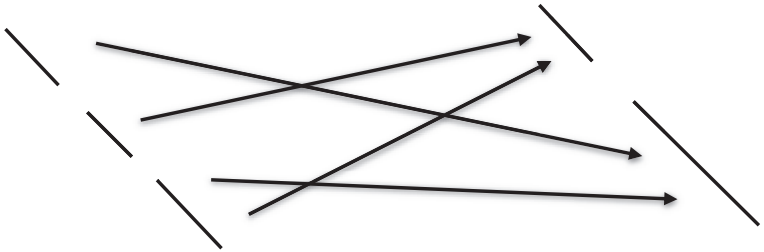
RDD: **x**



RDD: **y**



PARTITIONBY



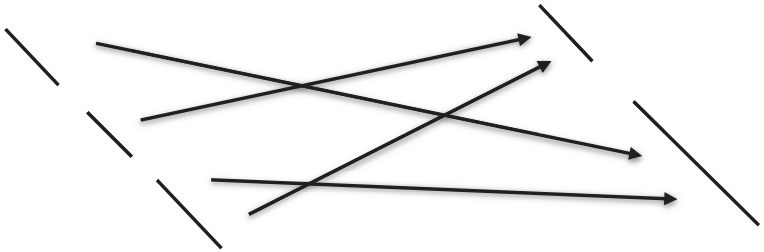
Return a new RDD with the specified number of partitions, placing original items into the partition returned by a user supplied function

```
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



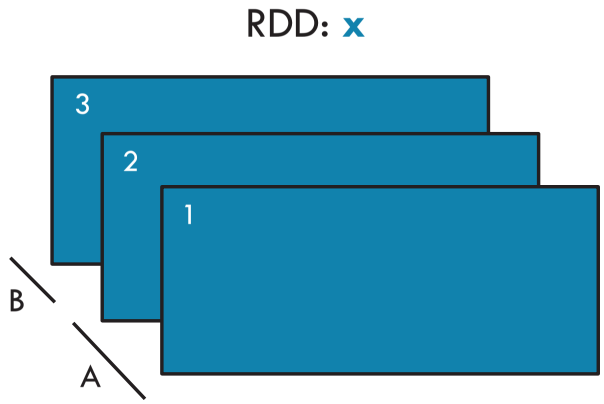
Return a new RDD with the specified number of partitions, placing original items into the partition returned by a user supplied function.

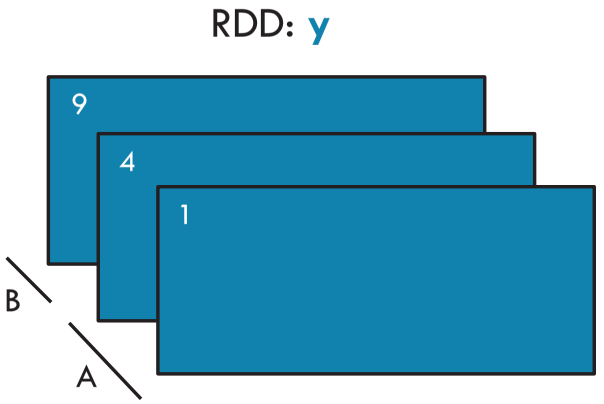
```
partitionBy(numPartitions, partitioner=portable_hash)
```

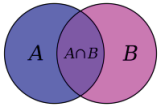
```
import org.apache.spark.Partitioner
val x = sc.parallelize(Array(('J',"James"),('F',"Fred"),
                             ('A',"Anna"),('J',"John")), 3)

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

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

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

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

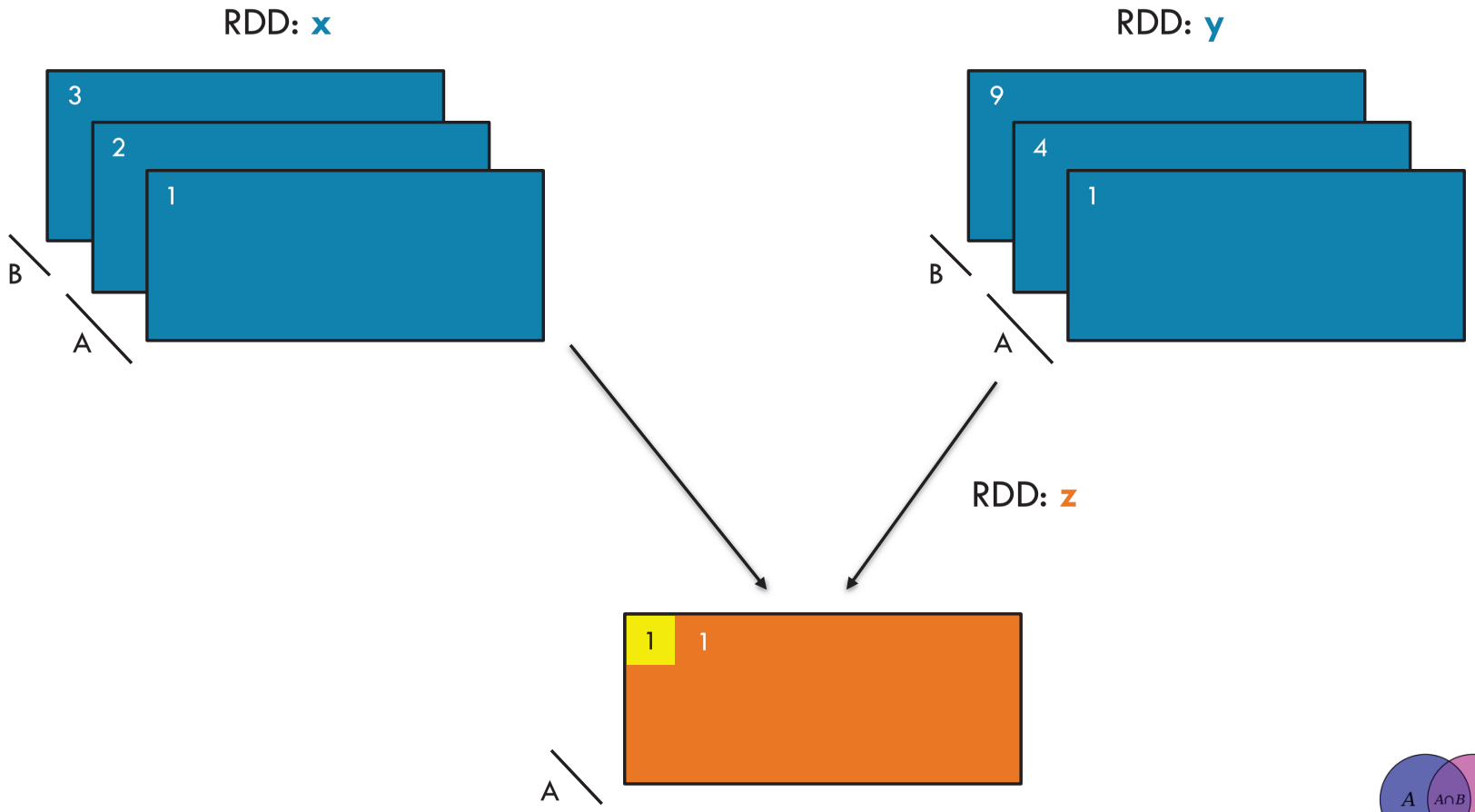
ZIP

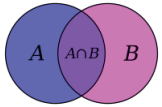




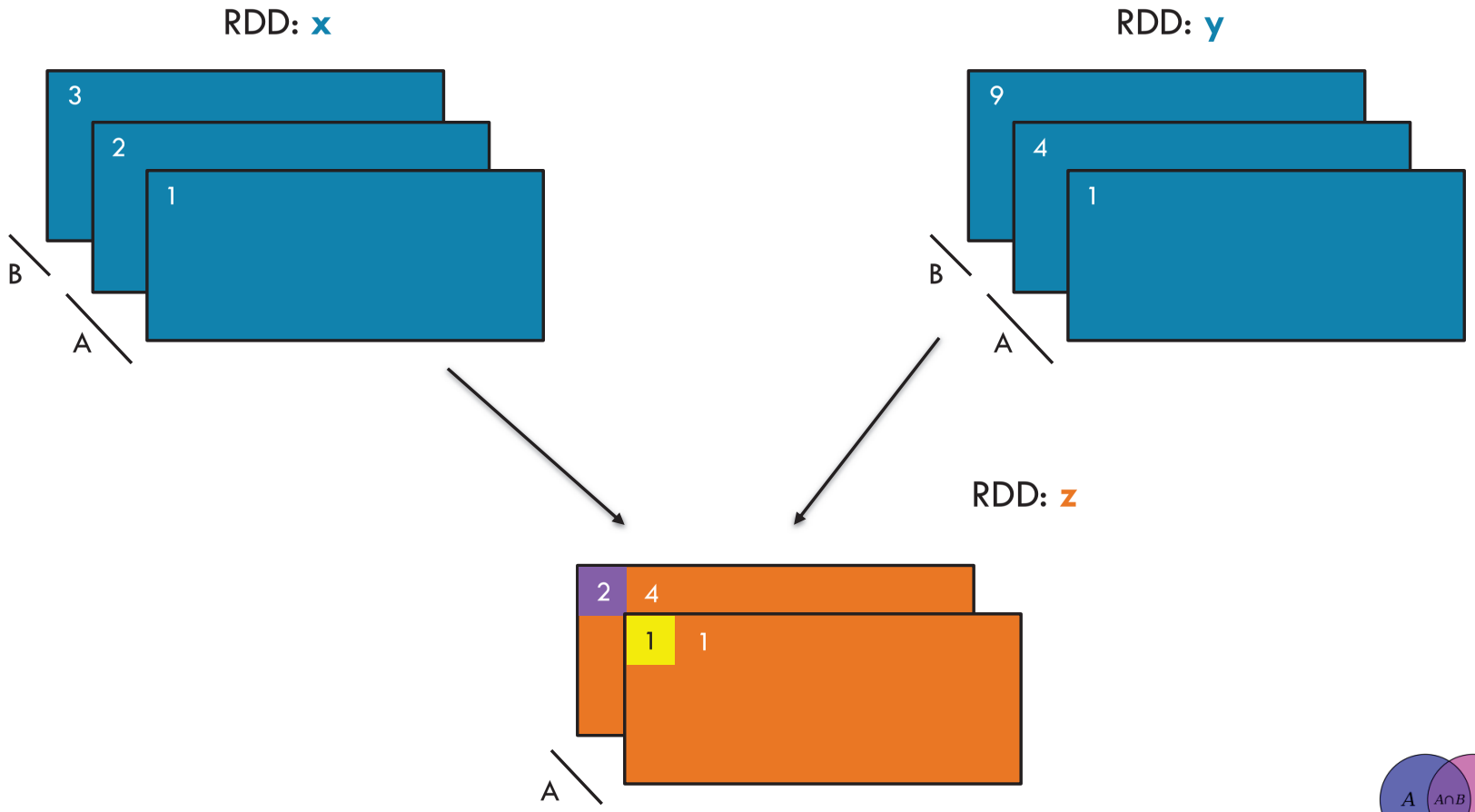


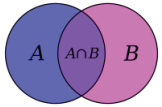
ZIP



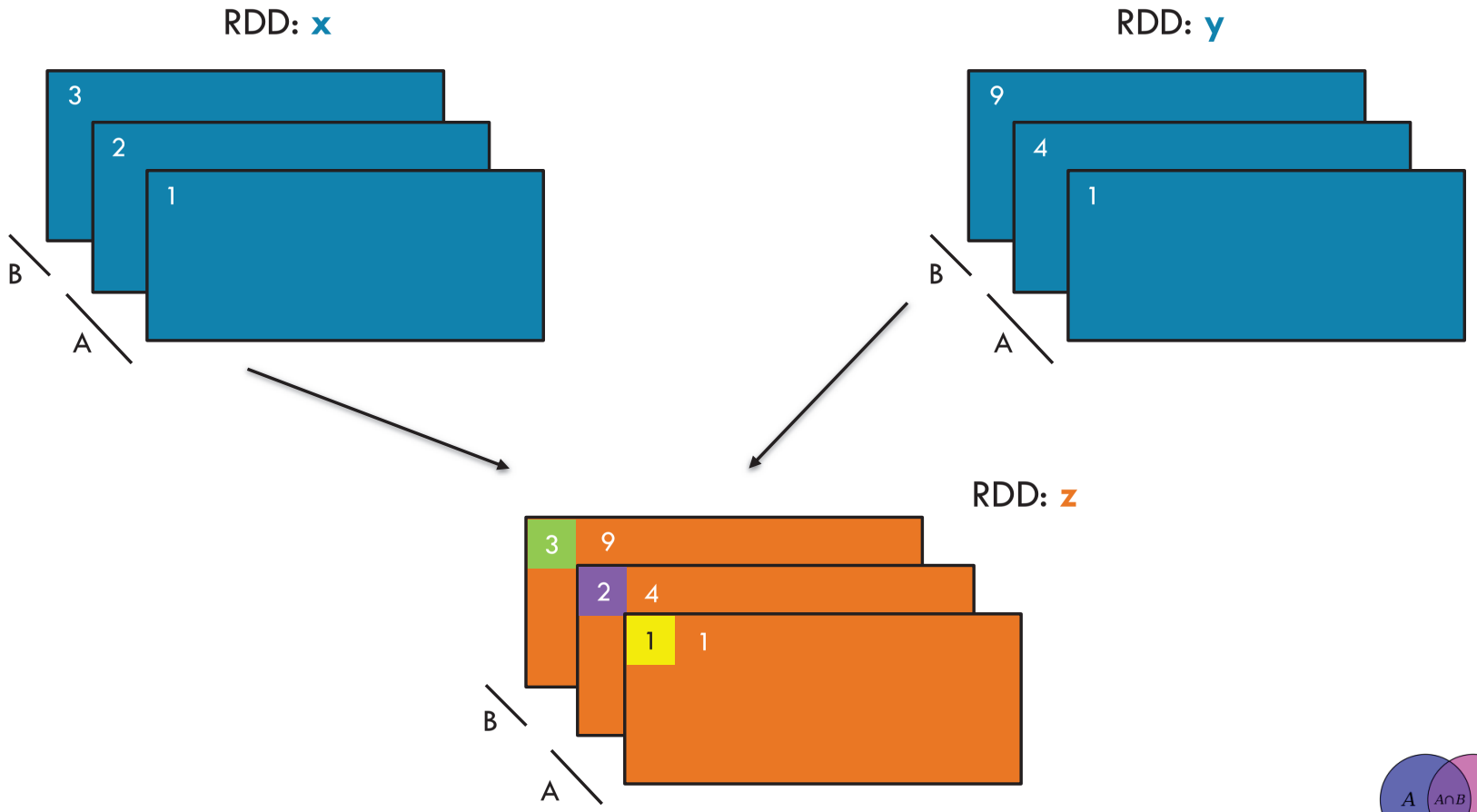


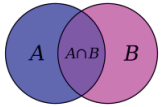
ZIP



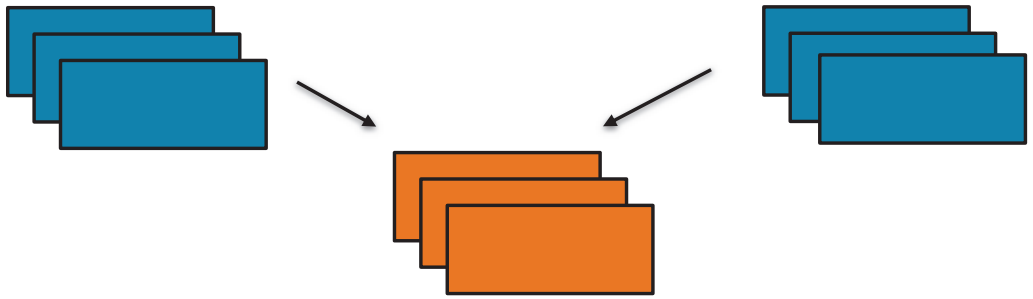


ZIP





ZIP



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

`zip(otherRDD)`

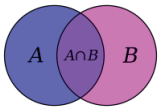
```python
x = sc.parallelize([1, 2, 3])
y = x.map(lambda n:n*n)
z = x.zip(y)

print(z.collect())
```

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

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

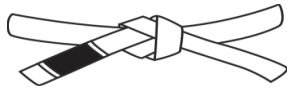
`x:` [1, 2, 3]

`y:` [1, 4, 9]

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



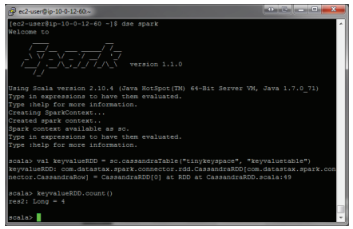
ACTIONS



Core Operations

databricks

vs



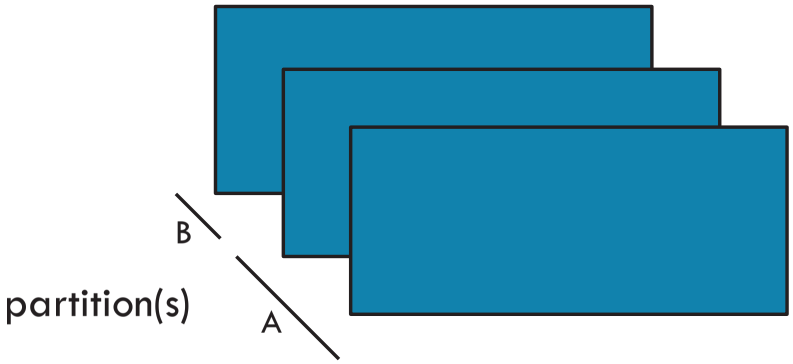
distributed

occurs across the cluster

driver

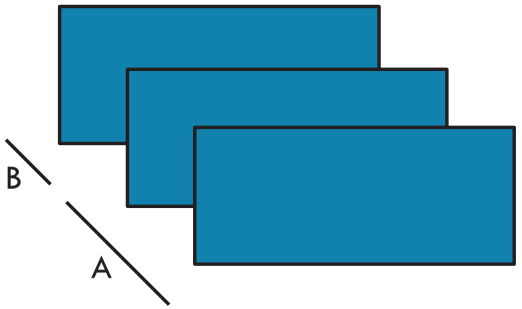
result must fit in driver JVM

GETNUMPARTITIONS



2

GETNUMPARTITIONS



getNumPartitions()

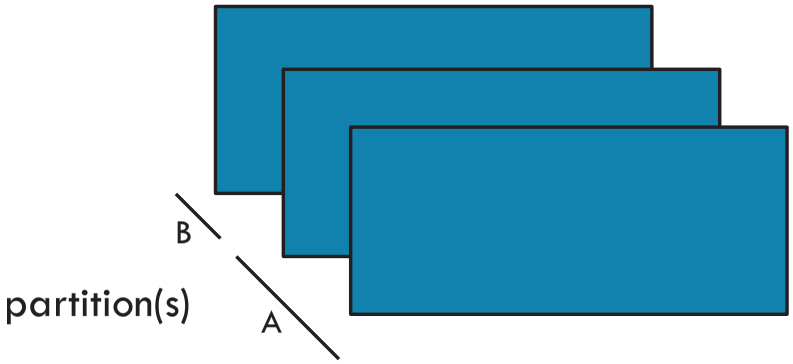
Return the number of partitions in RDD

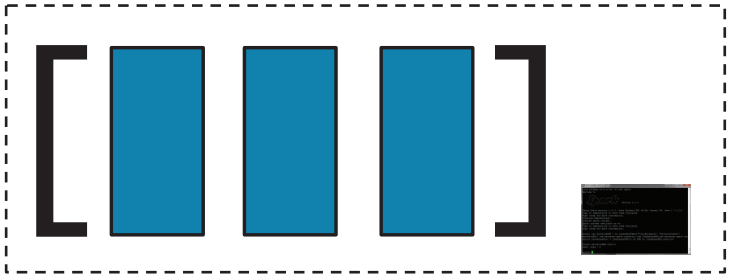
```python
x = sc.parallelize([1,2,3], 2)
y = x.getNumPartitions()

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

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

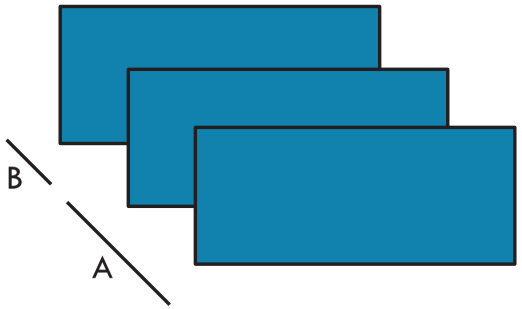
```
x: [[1], [2, 3]]

y: 2
```

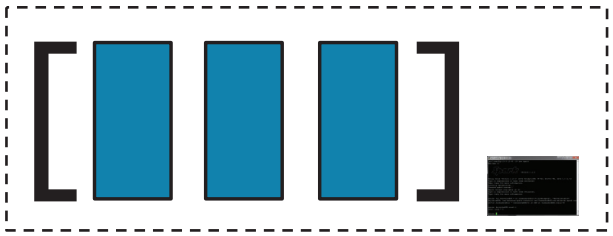
COLLECT





COLLECT





collect()

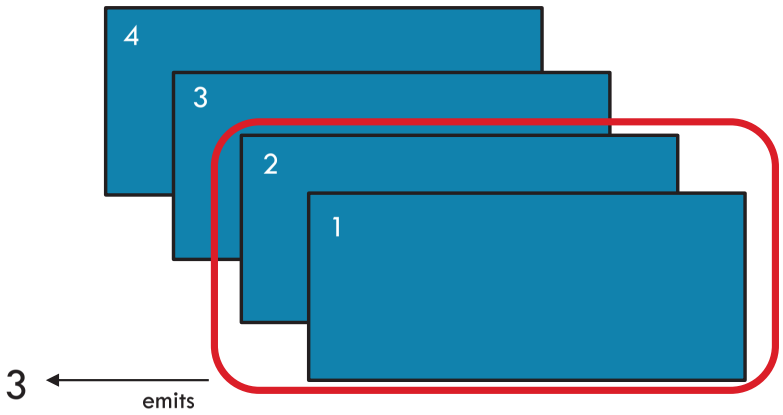
Return all items in the RDD to the driver in a single list

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

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

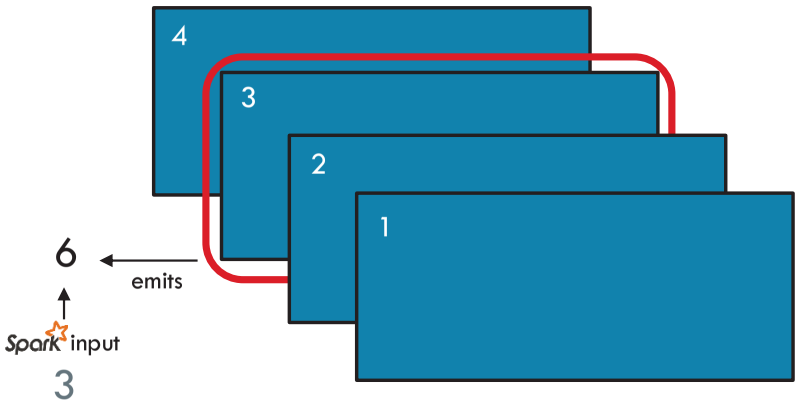
```scala
val x = sc.parallelize(Array(1,2,3), 2)
val y = x.collect()

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

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

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

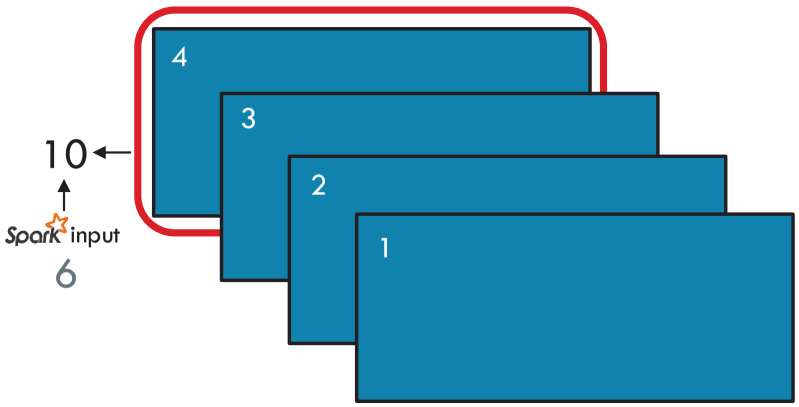
REDUCE



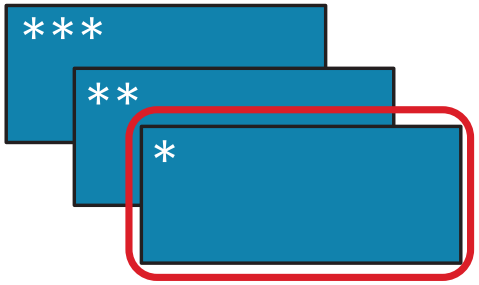
REDUCE

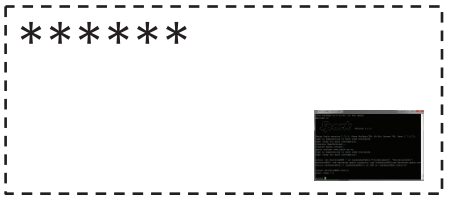


REDUCE



REDUCE





reduce(*f*)

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

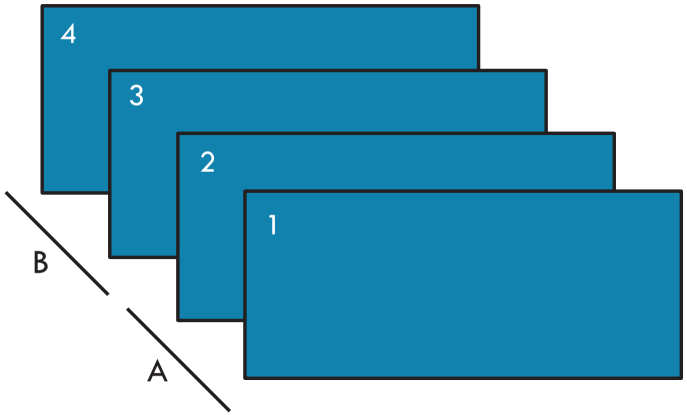
```python
x = sc.parallelize([1,2,3,4])
y = x.reduce(lambda a,b: a+b)

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

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

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

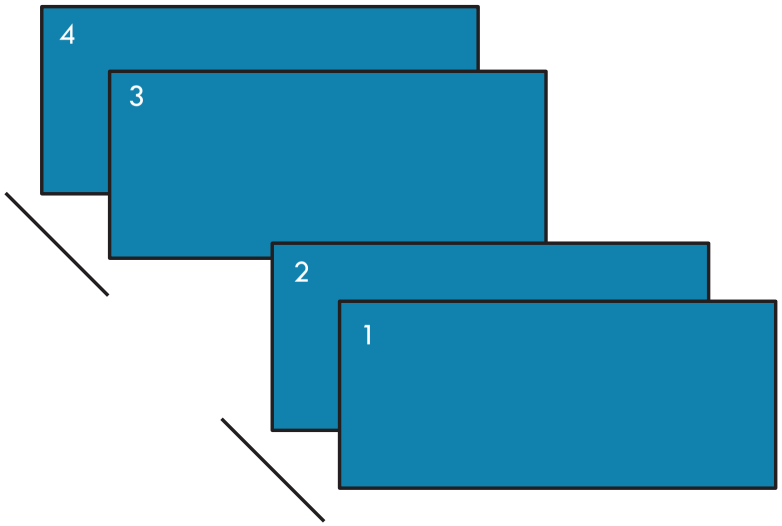
x: [1, 2, 3, 4]

y: 10

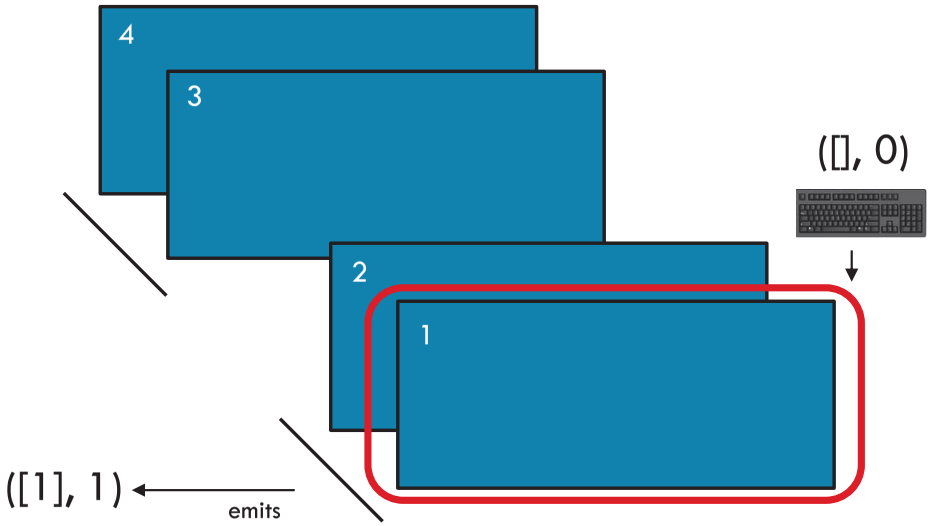
AGGREGATE



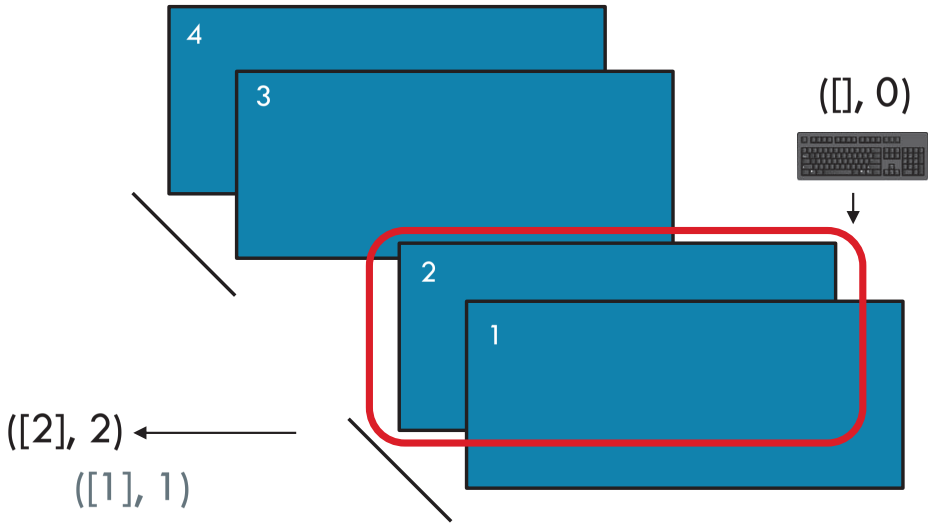
AGGREGATE



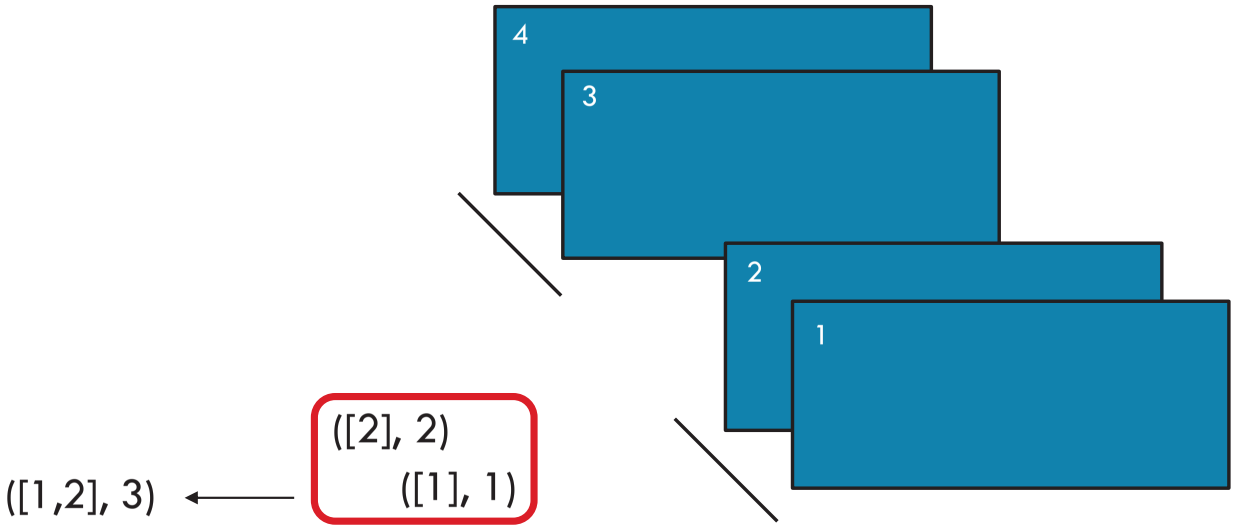
AGGREGATE



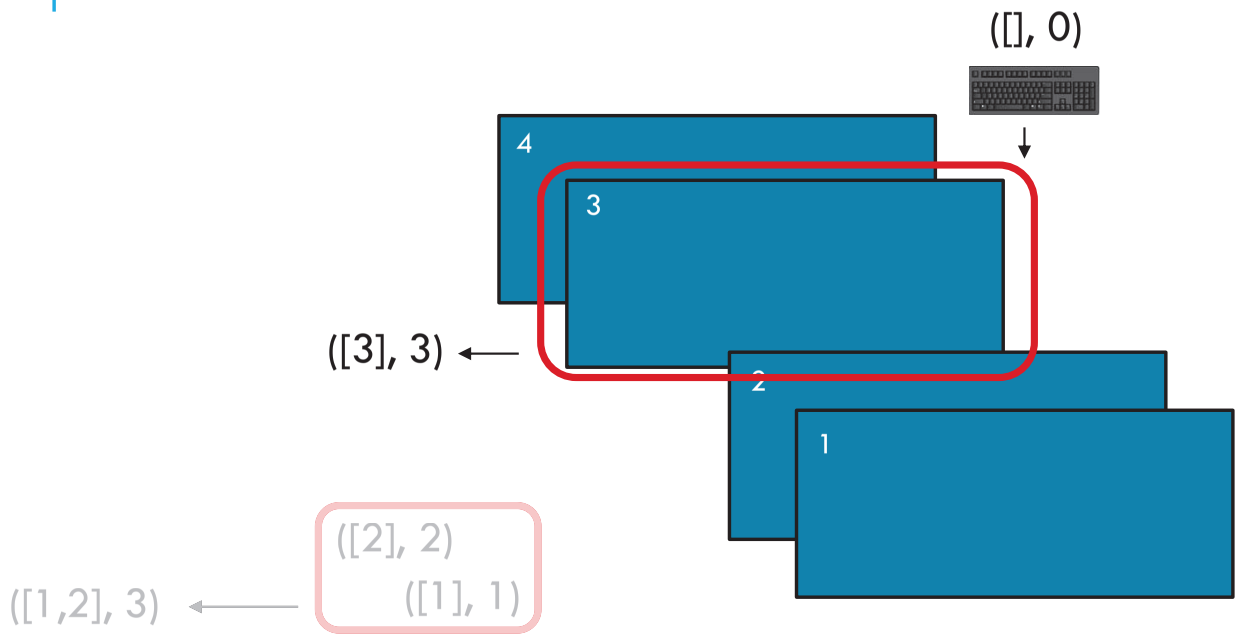
AGGREGATE



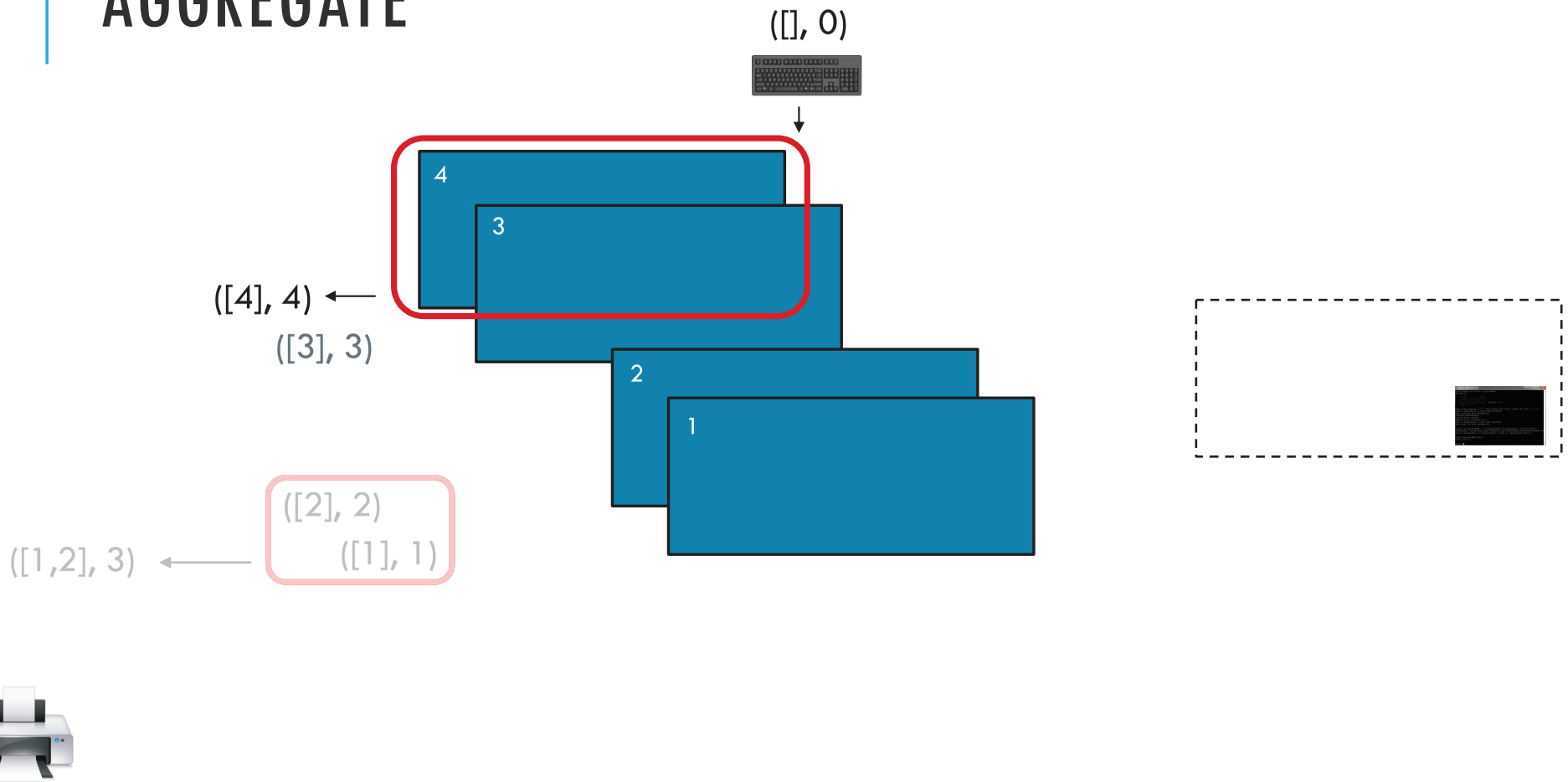
AGGREGATE



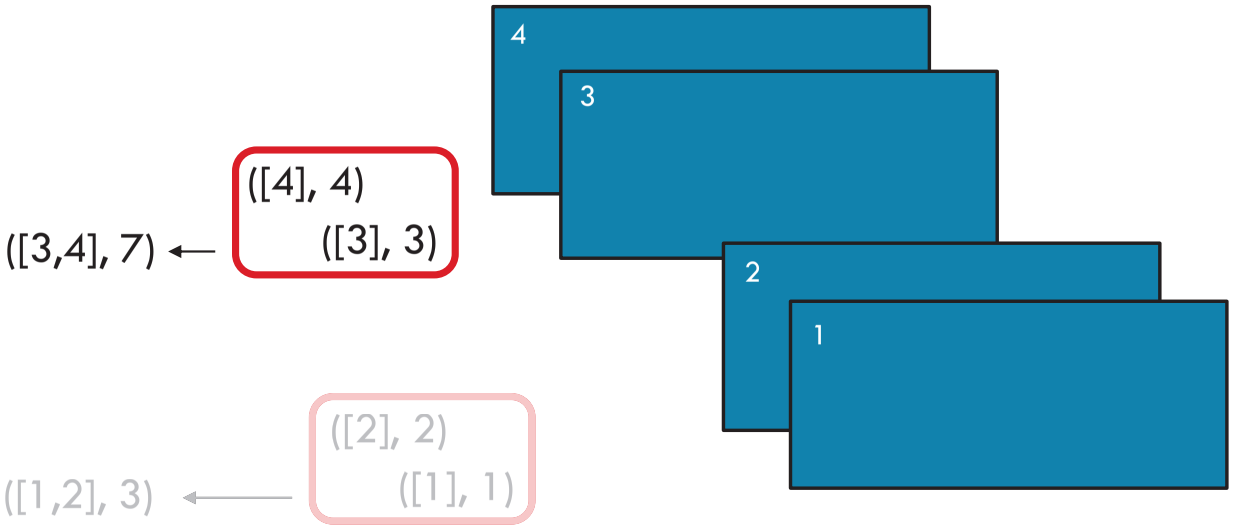
AGGREGATE



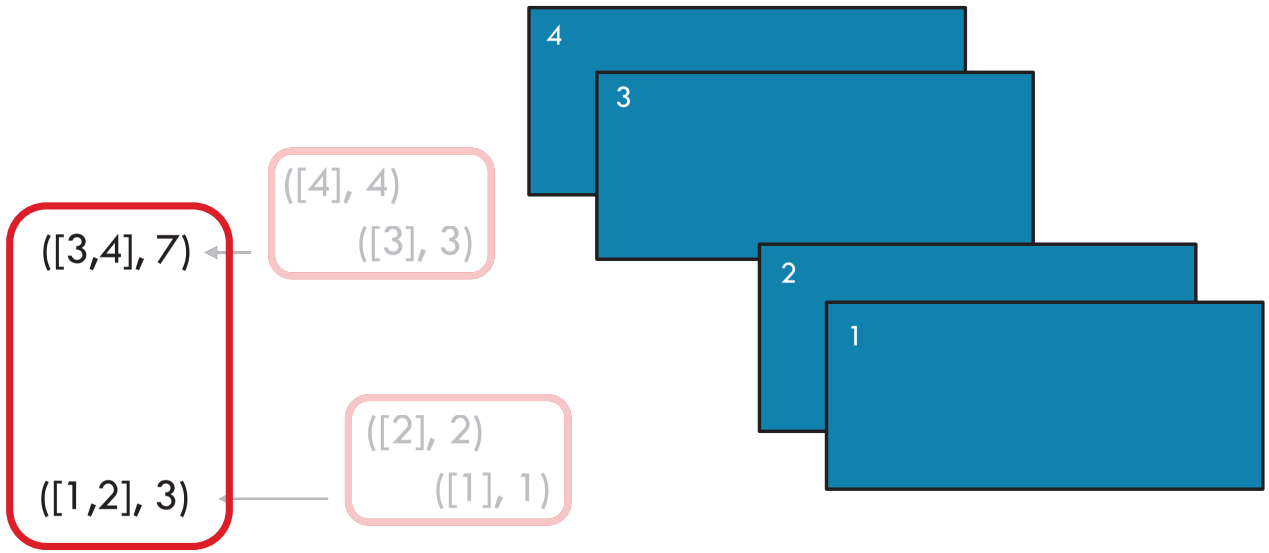
AGGREGATE



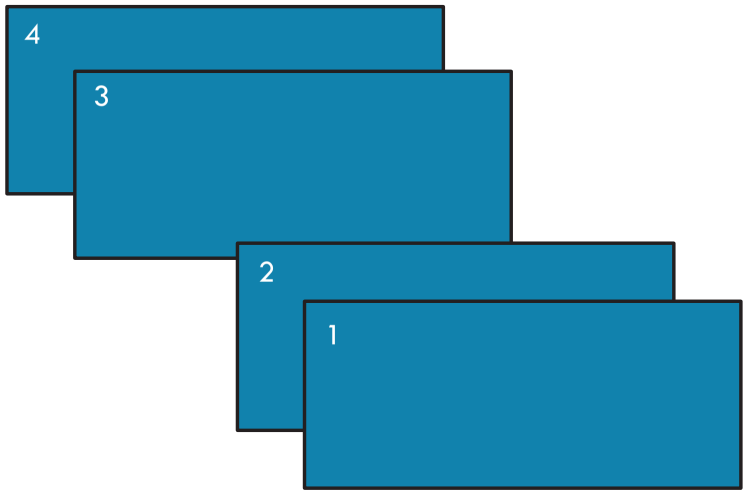
AGGREGATE



AGGREGATE



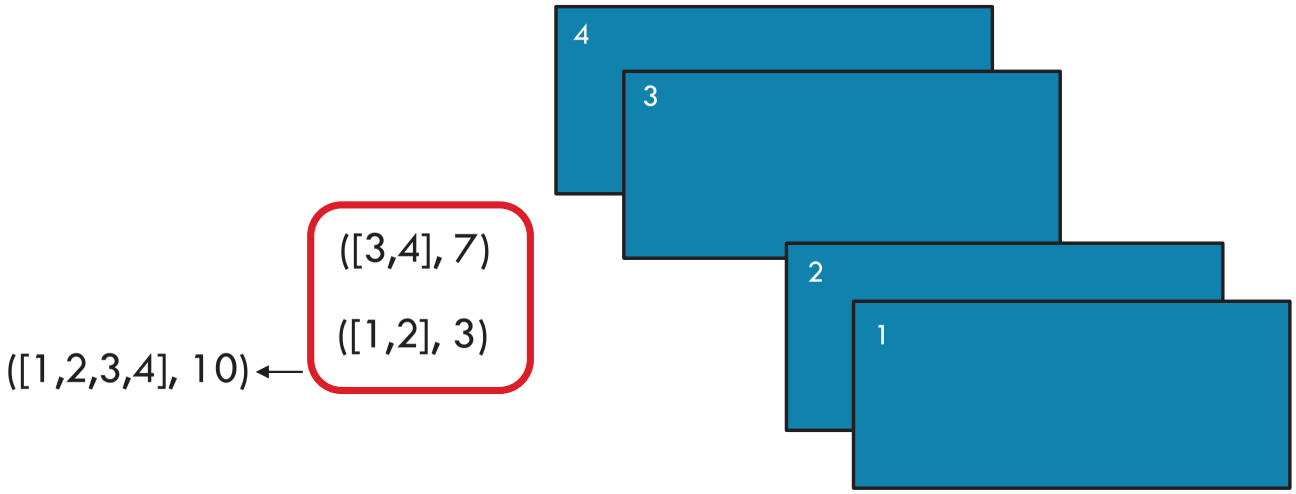
AGGREGATE



([3,4], 7)

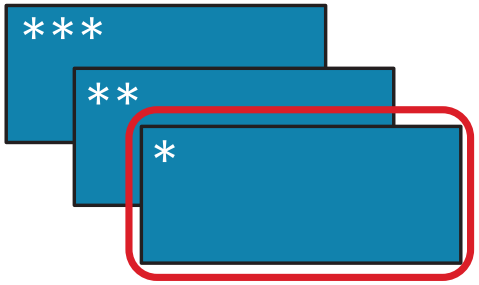
([1,2], 3)

AGGREGATE



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

AGGREGATE



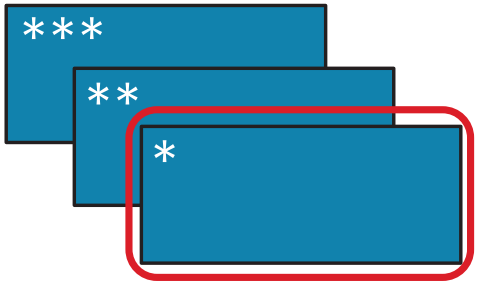
`aggregate(identity, seqOp, combOp)`

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

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

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

print(y)
```

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

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

AGGREGATE



`aggregate(identity, seqOp, combOp)`

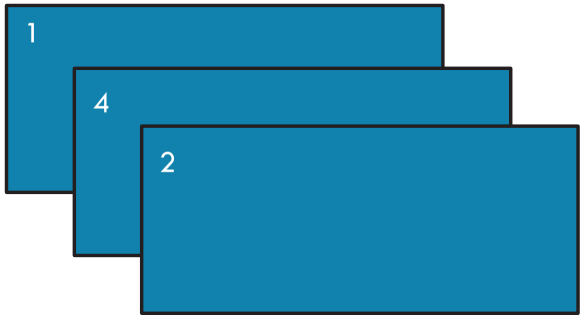
Aggregate all the elements of the RDD by:
- applying a user function to combine elements with user-supplied objects,
- then combining those user-defined results via a second user function,
- and finally returning a result to the driver.

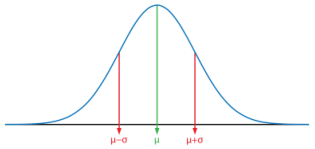
```
def seqOp = (data:(Array[Int], Int), item:Int) =>
     (data._1 :+ item, data._2 + item)

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

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

println(y)
```

x: [1, 2, 3, 4]

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

MAX





MAX

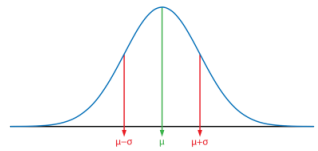
max()

Return the maximum item in the RDD

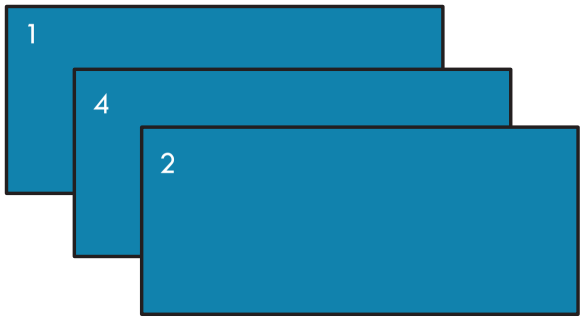
```python
x = sc.parallelize([2,4,1])
y = x.max()

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

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

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

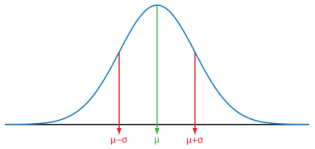
```
x: [2, 4, 1]

y: 4
```



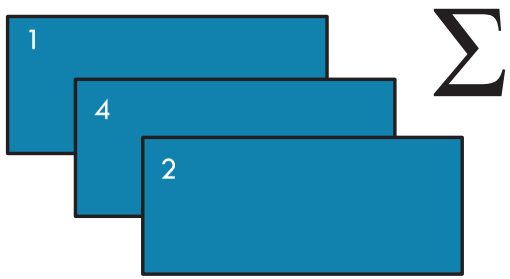
SUM



7



SUM



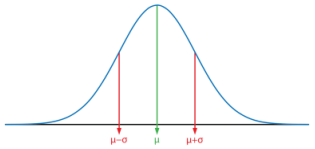
sum()

Return the sum of the items in the RDD

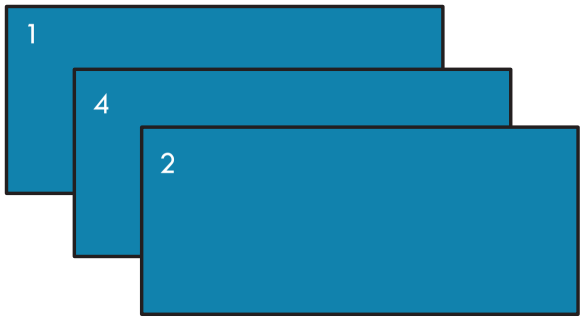
```python
x = sc.parallelize([2,4,1])
y = x.sum()

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

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

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

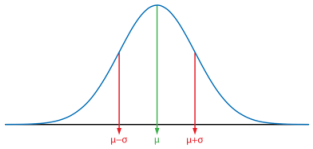
```
x: [2, 4, 1]

y: 7
```



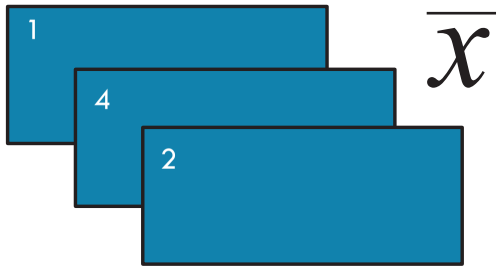
MEAN




```
2.33333333
```



MEAN



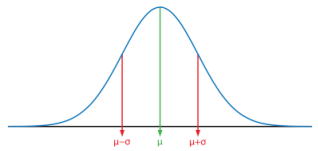
`mean()`

Return the mean of the items in the RDD

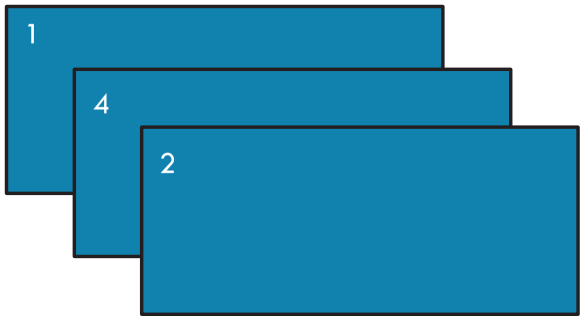
```
x = sc.parallelize([2,4,1])
y = x.mean()

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

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

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

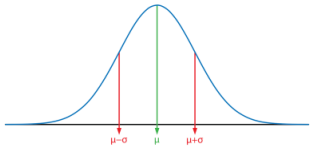
```
x: [2, 4, 1]

y: 2.3333333
```



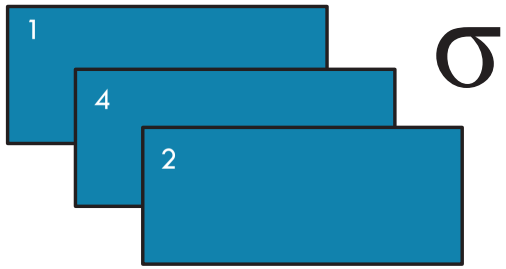
STDEV



1.2472191



STDEV



1.2472191 [redacted]

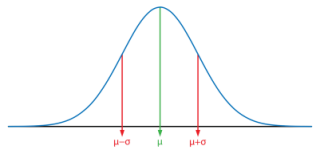
stdev()

Return the standard deviation of the items in the RDD

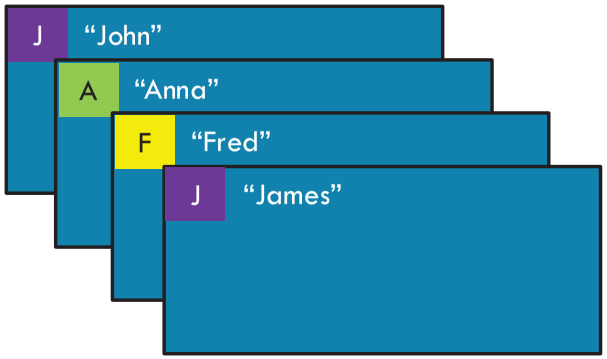
```python
x = sc.parallelize([2,4,1])
y = x.stdev()

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

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

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

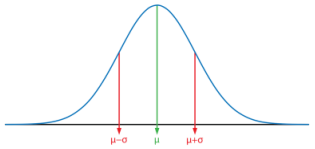
```
x: [2, 4, 1]
y: 1.2472191
```



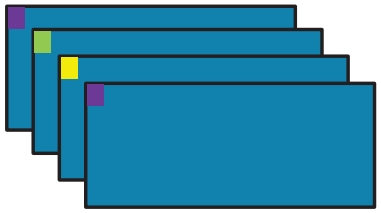
COUNTBYKEY

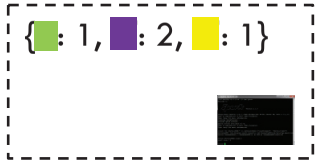


{'A': 1, 'J': 2, 'F': 1}



COUNTBYKEY





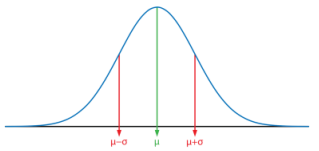
countByKey()

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

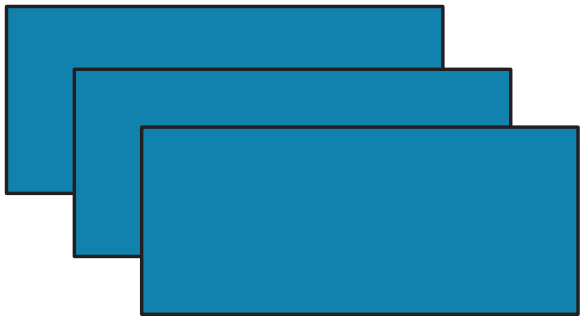
```
x = sc.parallelize([('J', 'James'), ('F','Fred'),
                    ('A','Anna'), ('J','John')])

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

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

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

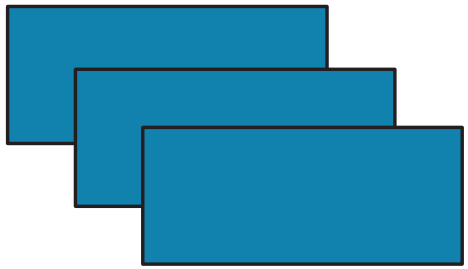
```
x: [('J', 'James'), ('F','Fred'),
    ('A','Anna'), ('J','John')]

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



SAVEASTEXTFILE



SAVEASTEXTFILE



`saveAsTextFile(path, compressionCodecClass=None)`

Save the RDD to the filesystem indicated in the path

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

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

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

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

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

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

LAB

Q&A

databricks™