

Resilient

Distributed

Datasets

RDD - the definition

RDD stands for resilient distributed dataset

R esilient - if data is lost, data can be recreated

D istributed - stored in nodes among the cluster

D ataset - initial data comes from a file

or can be created programmatically

What about Resilience?

RDD stands for resilient distributed dataset

Resilient - if data is lost, data can be recreated

Distributed - stored in nodes among the cluster

Dataset - initial data comes from a file

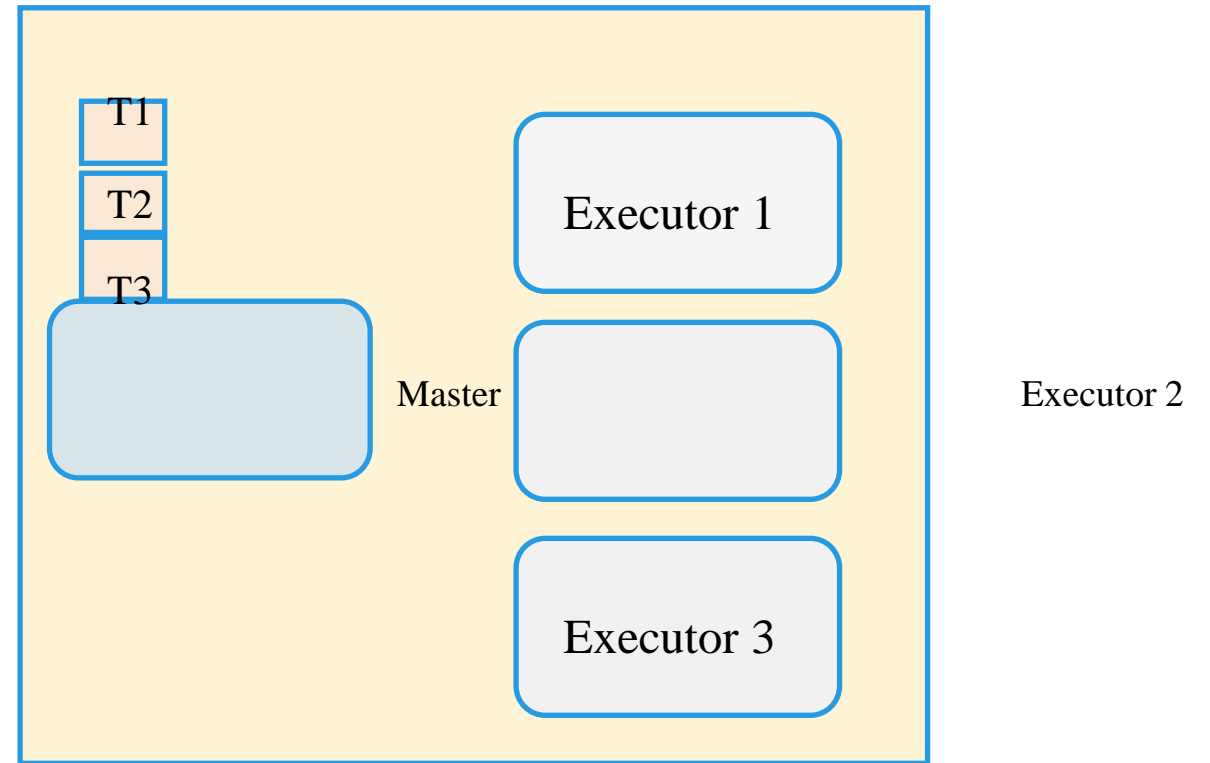
or can be created programmatically

Resilience

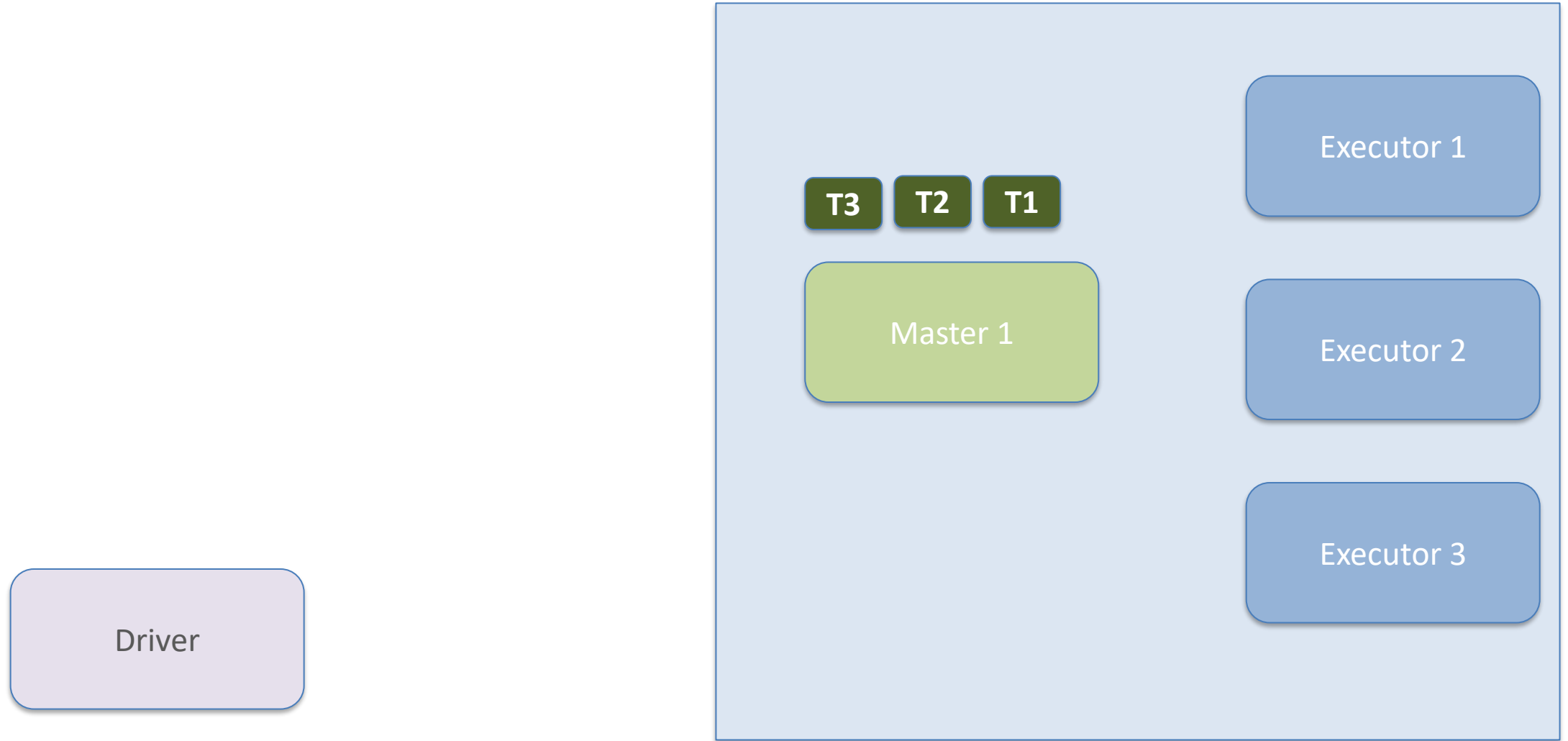
```
val master = "spark://host:pt"
val conf = new SparkConf()
    .setMaster(master)
val sc = new SparkContext
    (conf)
    txt")
val logs = sc.textFile("logs.
```

```
println(logs.count())
```

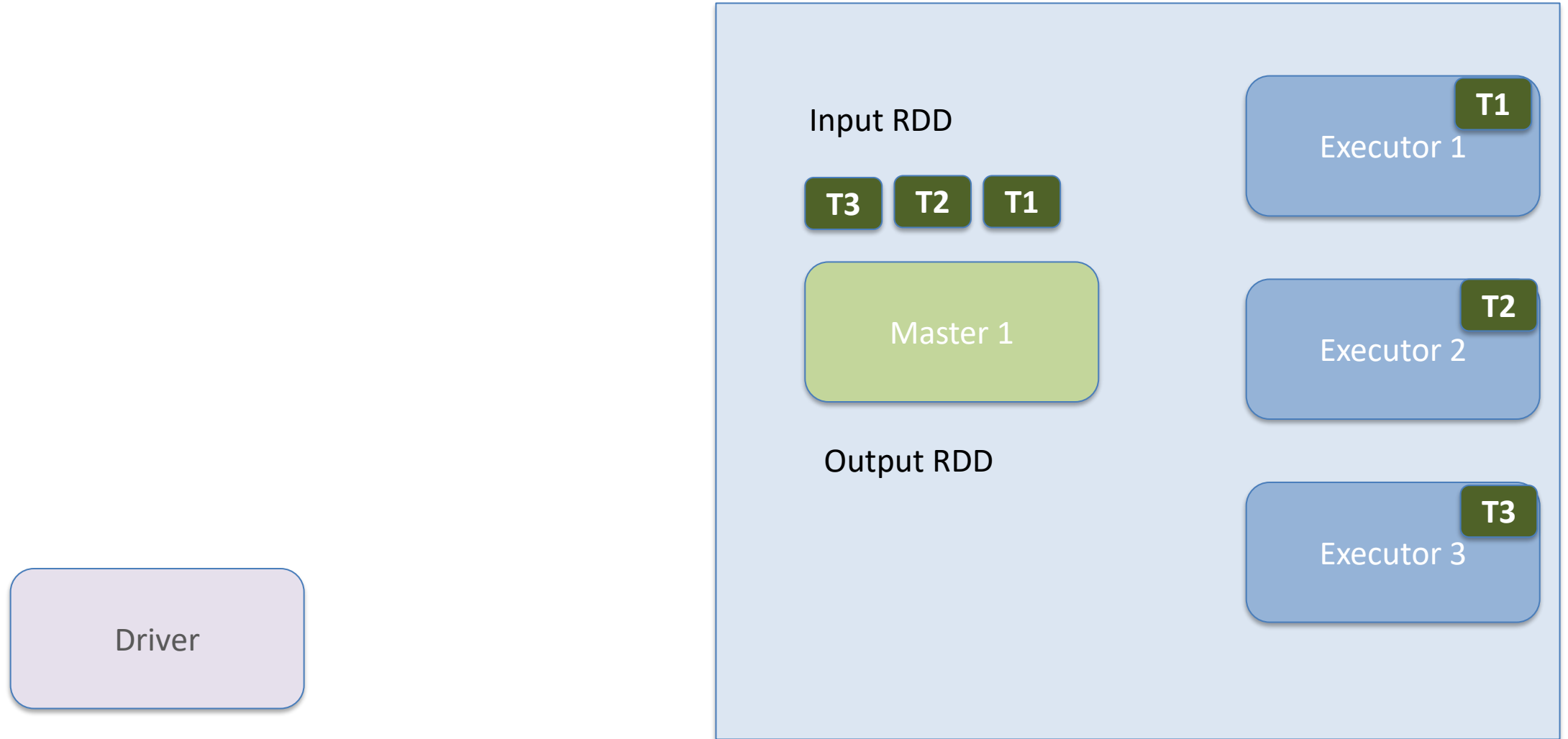
Driver Program



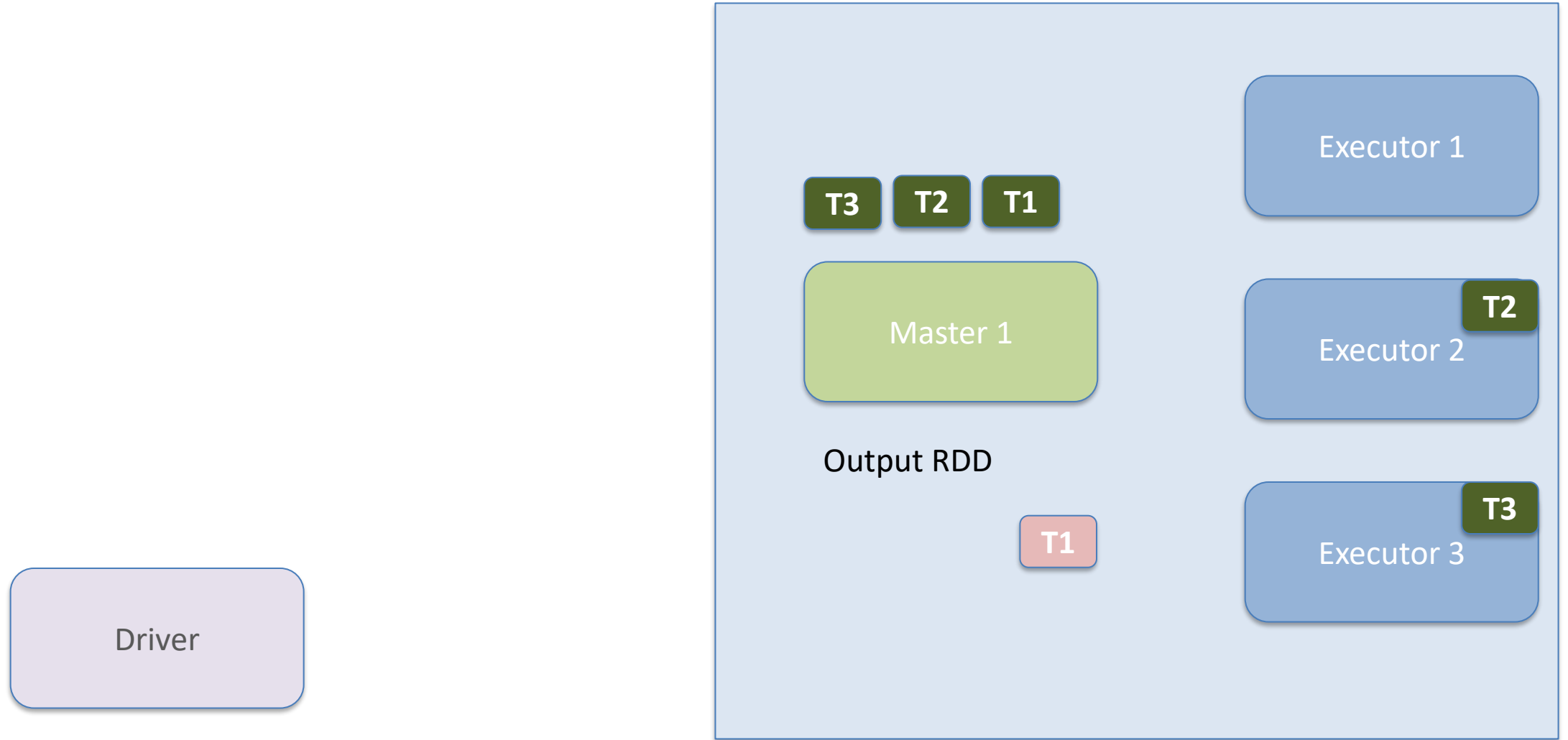
RESILLIENCE-Managing Failure



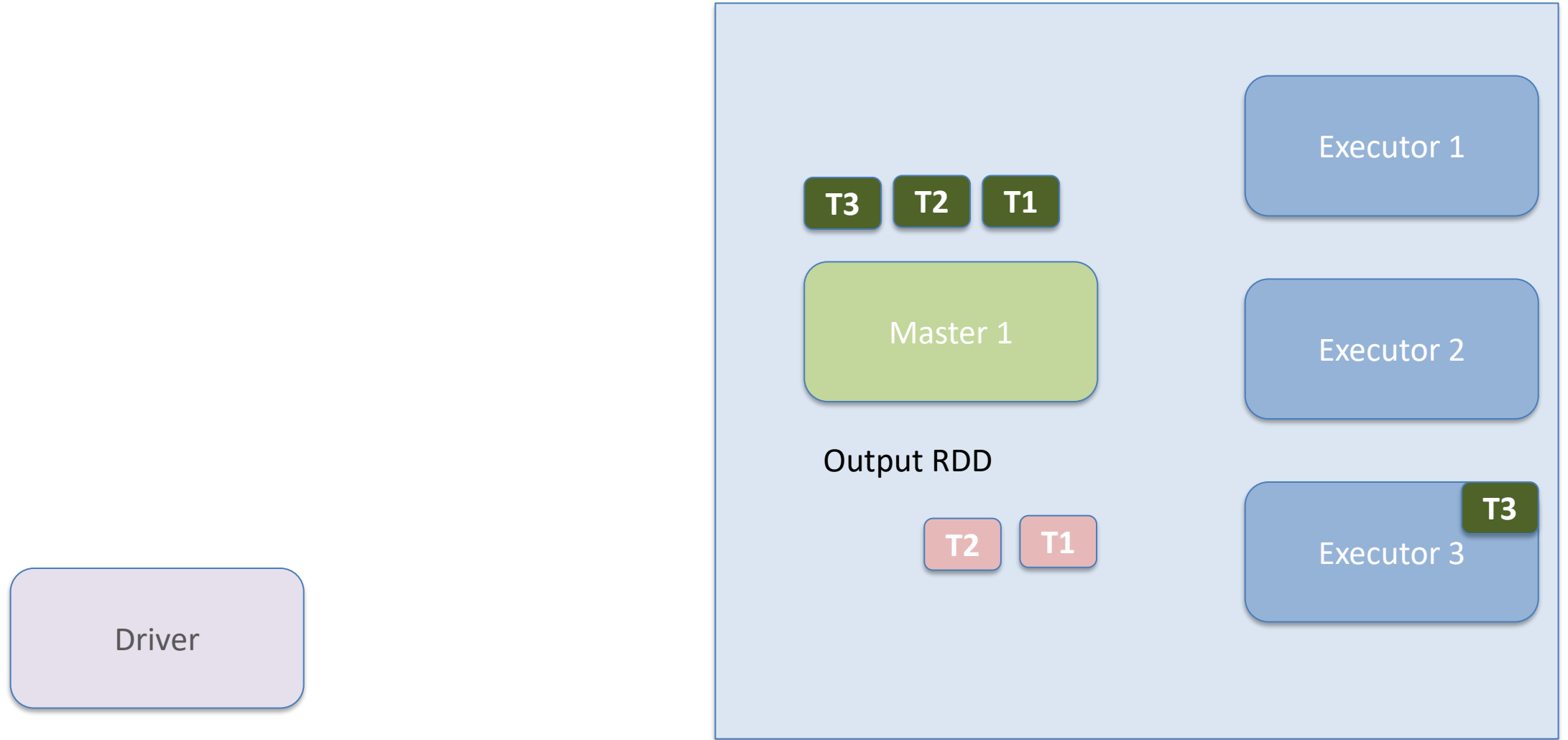
RESILLIENCE



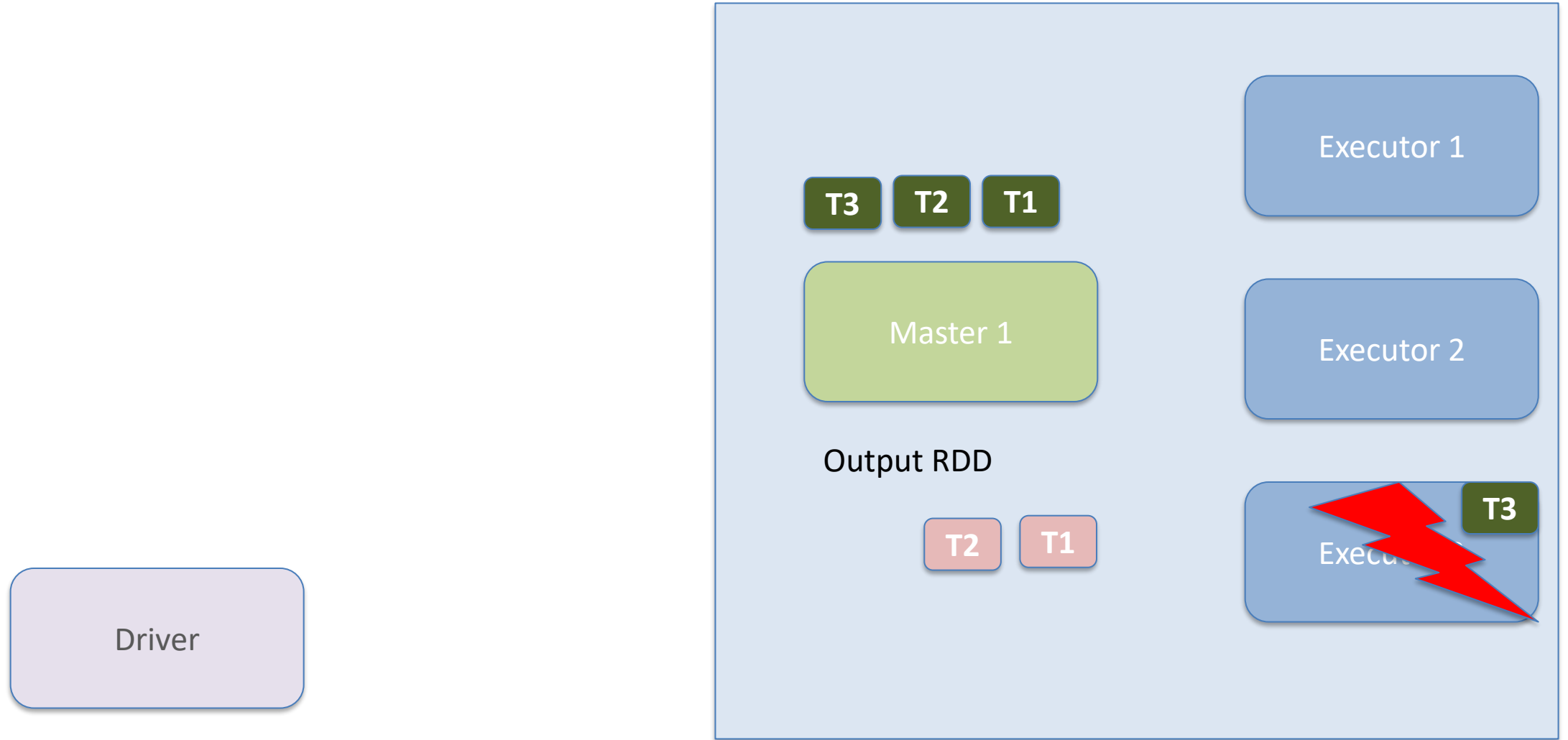
RESILLIENCE



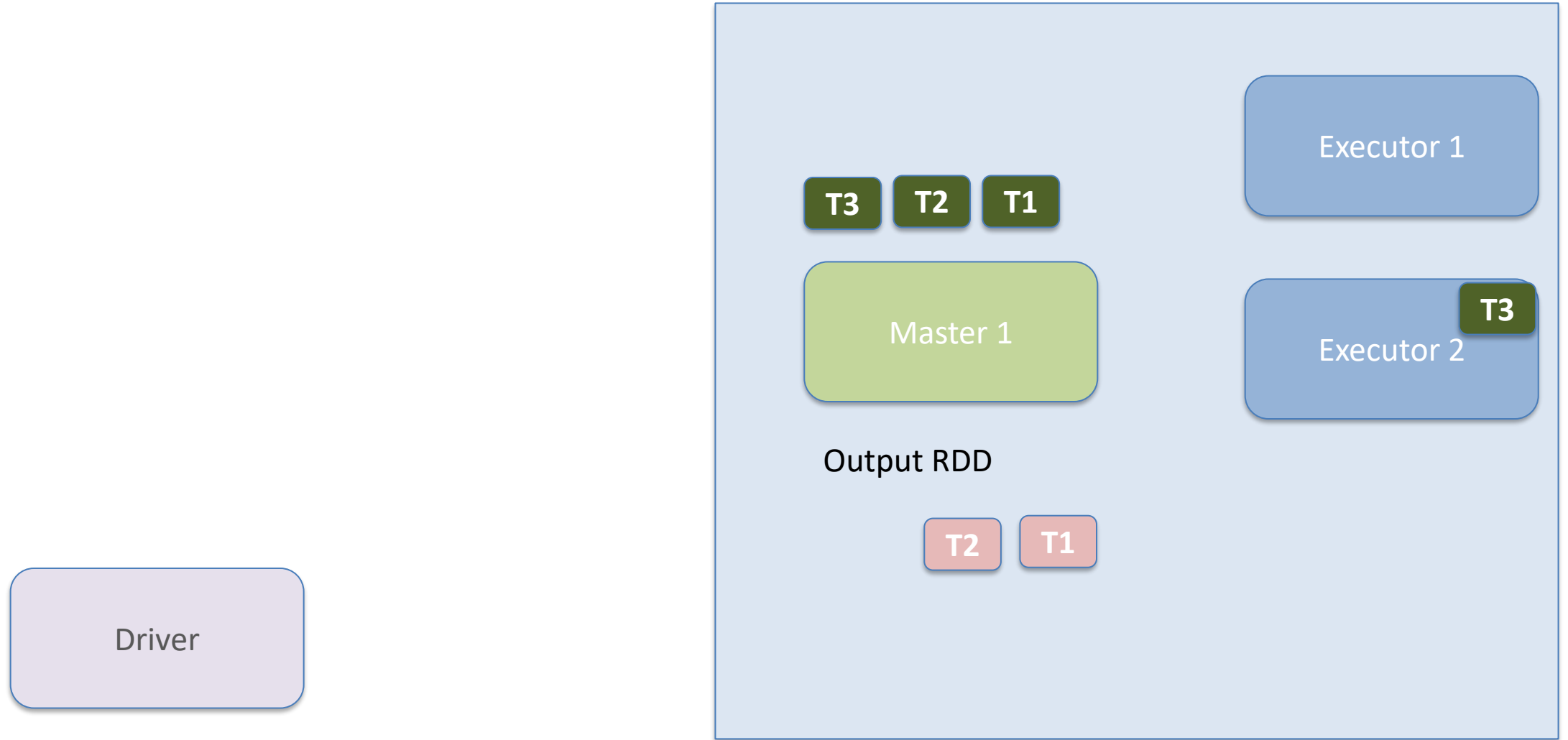
RESILLIENCE



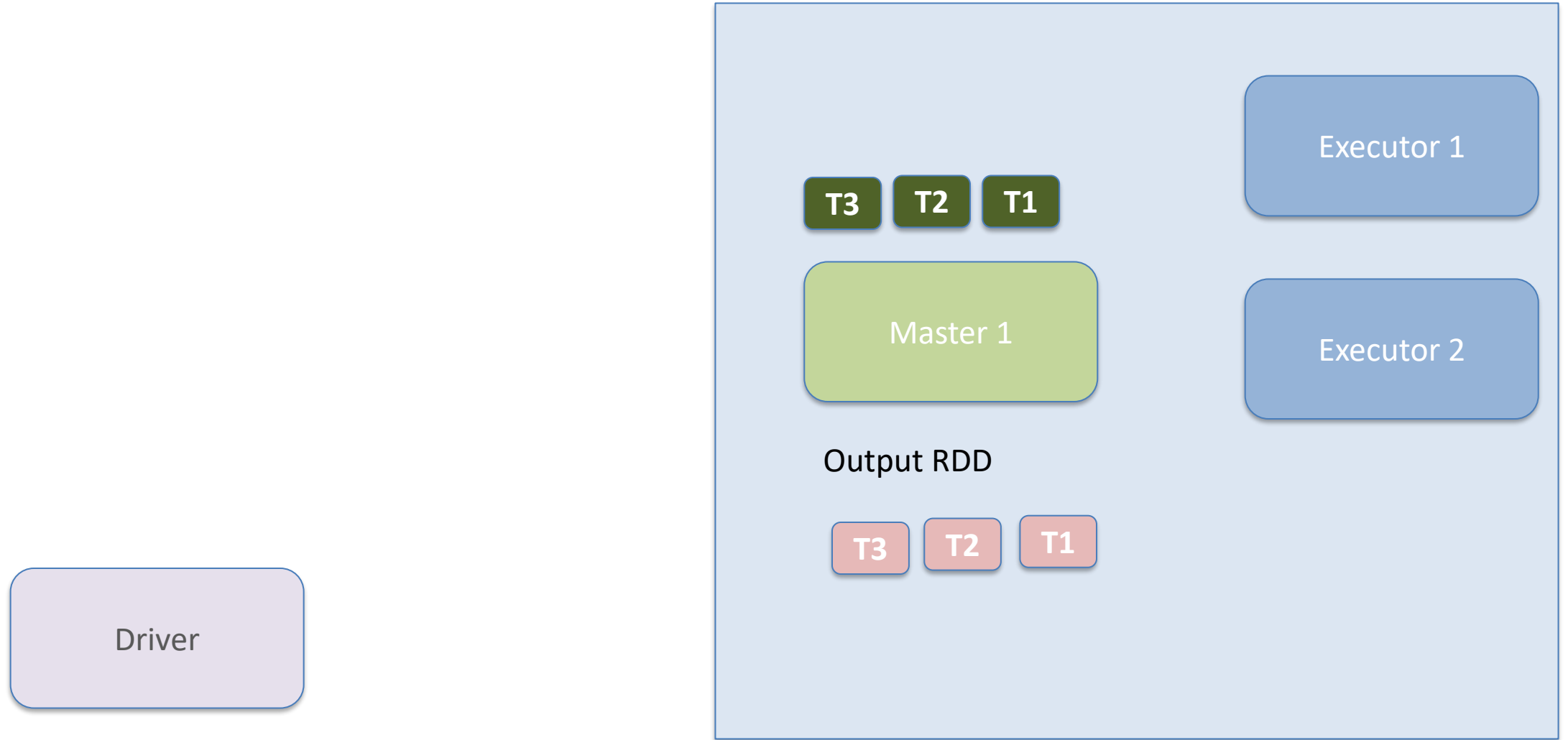
RESILLIENCE



RESILLIENCE



RESILLIENCE



RDDs

- Primary abstraction object used by Apache Spark
- **R**esilient **D**istributed **D**ataset
 - Fault-tolerant
 - Collection of elements that can be operated on in parallel
 - Distributed collection of data from any source
- Contained in an RDD:
 - Set of partitions
 - Atomic pieces of a dataset
 - Set of dependencies on parent RDDs
 - Lineage (Directed Acyclic Graph - DAG)
 - A function for computing the RDD based on its parents
 - Metadata about its partitioning scheme and data placement

RDDs (Cont.)

- RDDs are immutable
 - Allows for more effective fault tolerance
- Intended to support abstract datasets while also maintain MapReduce properties like automatic fault tolerance, locality-aware scheduling and scalability.

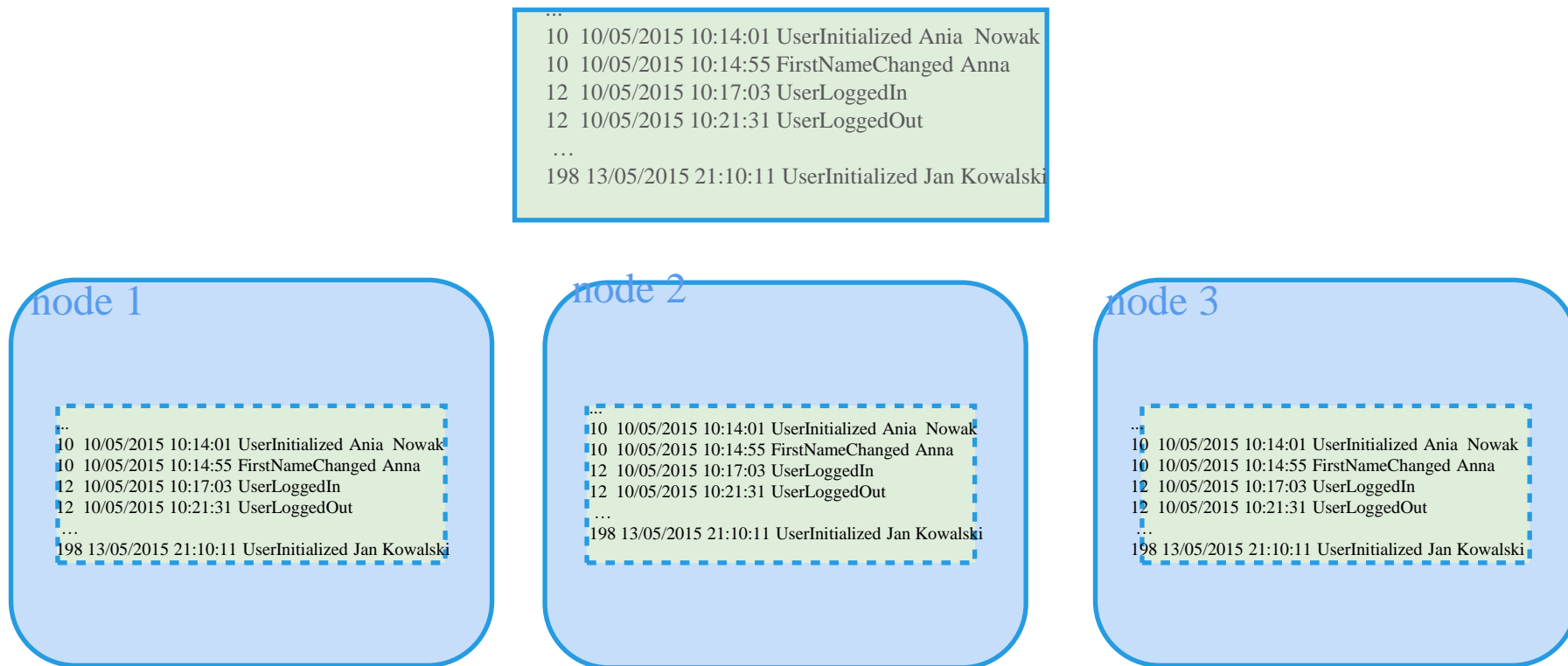
RDD

- A collection of elements partitioned across cluster
- Compile time type safe
- An **immutable distributed** collection of objects.
- Split in **partitions** which may be on multiple **nodes**
- Store any type of data [Number, String, Boolean, Object]
- Similar to Scala Collection API

RDD

- RDD Can be persisted in memory
- Or Storage
- RDD Auto recover from node failures
- Can have any data type but has a special dataset type for key-value

What is a Distributed RDD?



What is a RDD?

RDD needs to hold 3 chunks of information in order to do its work:

What is a RDD?

RDD needs to hold 3 chunks of information in order to do its work:

1. pointer to his parent

What is a RDD?

RDD needs to hold 3 chunks of information in order to do its work:

1. pointer to his parent
2. how its internal data is partitioned

What is a RDD?

RDD needs to hold 3 chunks of information in order to do its work:

1. pointer to his parent
2. how its internal data is partitioned
3. how to evaluate its internal data

What is a RDD?

RDD needs to hold 3 chunks of information in order to do its work:

1. pointer to his parent
2. how its internal data is partitioned
3. how to evaluate its internal data

What is a partition?

A partition represents subset of data within your distributed collection.

What is a partition?

A partition represents subset of data within your distributed collection.

override def getPartitions: Array [Partition] = ???

What is a partition?

A partition represents subset of data within your distributed collection.

```
override def getPartitions: Array [ Partition ] = ???
```

How this subset is defined depends on type of the RDD

example: HadoopRDD

```
val journal = sc.textFile("hdfs://journal/*")
```

example: HadoopRDD

```
val journal = sc.textFile("hdfs://journal/*")
```

How HadoopRDD is partitioned?

example: HadoopRDD

```
val journal = sc.textFile("hdfs://journal/*")
```

How HadoopRDD is partitioned?

In HadoopRDD partition is exactly the same as file chunks in HDFS

example: HadoopRDD

10	10/05/2015 10:14:01	UserInit	16	10/05/2015 10:14:01	UserInit	10	10/05/2015 10:14:01	UserInit	5	10/05/2015 10:14:01	UserInit
3	10/05/2015 10:14:55	FirstNa	20	10/05/2015 10:14:55	FirstNa	10	10/05/2015 10:14:55	FirstNa	4	10/05/2015 10:14:55	FirstNa
12	10/05/2015 10:17:03	UserLo	42	10/05/2015 10:17:03	UserLo	12	10/05/2015 10:17:03	UserLo	12	10/05/2015 10:17:03	UserLo
4	10/05/2015 10:21:31	UserLo	67	10/05/2015 10:21:31	UserLo	12	10/05/2015 10:21:31	UserLo	142	10/05/2015 10:21:31	UserLo
5	13/05/2015 21:10:11	UserIni	12	13/05/2015 21:10:11	UserIni	198	13/05/2015 21:10:11	UserIni	158	13/05/2015 21:10:11	UserIni

example: HadoopRDD

10 10/05/2015 10:14:01 UserInit	16 10/05/2015 10:14:01 UserInit	10 10/05/2015 10:14:01 UserInit	5 10/05/2015 10:14:01 UserInit
3 10/05/2015 10:14:55 FirstNa	20 10/05/2015 10:14:55 FirstNa	10 10/05/2015 10:14:55 FirstNa	4 10/05/2015 10:14:55 FirstNa
12 10/05/2015 10:17:03 UserLo	42 10/05/2015 10:17:03 UserLo	12 10/05/2015 10:17:03 UserLo	12 10/05/2015 10:17:03 UserLo
4 10/05/2015 10:21:31 UserLo	67 10/05/2015 10:21:31 UserLo	12 10/05/2015 10:21:31 UserLo	142 10/05/2015 10:21:31 UserLo
5 13/05/2015 21:10:11 UserIni	12 13/05/2015 21:10:11 UserIni	198 13/05/2015 21:10:11 UserIni	158 13/05/2015 21:10:11 UserIni

node 1

node 2

node 3

example: HadoopRDD

10	10/05/2015 10:14:01	UserInit	16	10/05/2015 10:14:01	UserInit	10	10/05/2015 10:14:01	UserInit	5	10/05/2015 10:14:01	UserInit
3	10/05/2015 10:14:55	FirstNa	20	10/05/2015 10:14:55	FirstNa	10	10/05/2015 10:14:55	FirstNa	4	10/05/2015 10:14:55	FirstNa
12	10/05/2015 10:17:03	UserLo	42	10/05/2015 10:17:03	UserLo	12	10/05/2015 10:17:03	UserLo	12	10/05/2015 10:17:03	UserLo
4	10/05/2015 10:21:31	UserLo	67	10/05/2015 10:21:31	UserLo	12	10/05/2015 10:21:31	UserLo	142	10/05/2015 10:21:31	UserLo
5	13/05/2015 21:10:11	UserIni	12	13/05/2015 21:10:11	UserIni	198	13/05/2015 21:10:11	UserIni	158	13/05/2015 21:10:11	UserIni

node 1



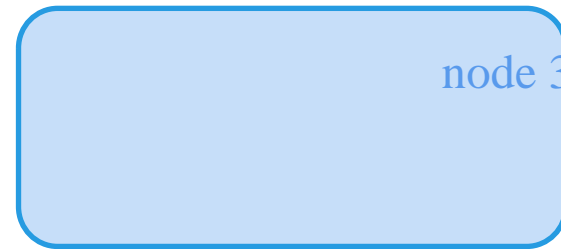
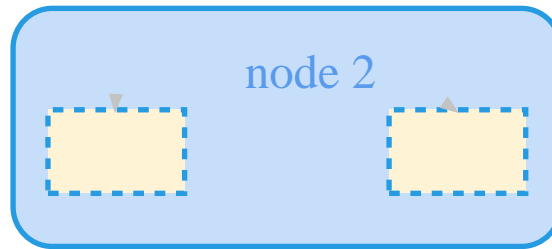
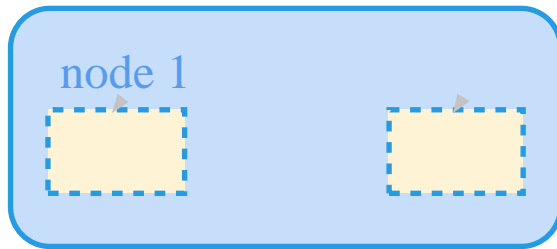
node 2



node 3

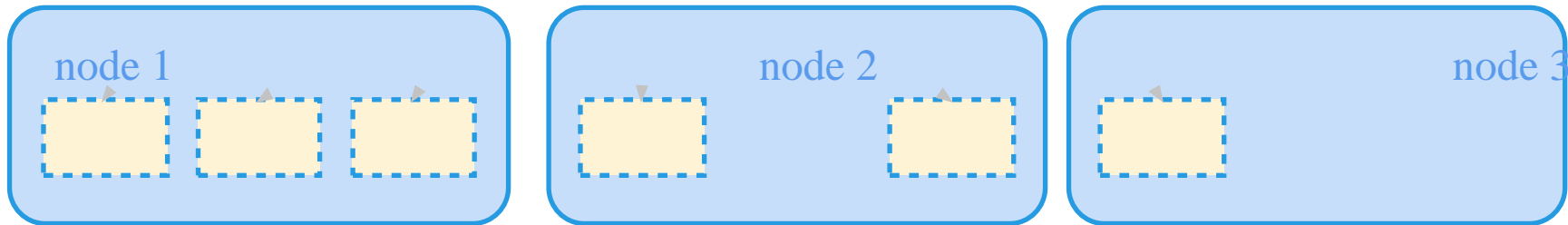
example: HadoopRDD

10	10/05/2015 10:14:01	UserInit	16	10/05/2015 10:14:01	UserInit	10	10/05/2015 10:14:01	UserInit	5	10/05/2015 10:14:01	UserInit
3	10/05/2015 10:14:55	FirstNa	20	10/05/2015 10:14:55	FirstNa	10	10/05/2015 10:14:55	FirstNa	4	10/05/2015 10:14:55	FirstNa
12	10/05/2015 10:17:03	UserLo	42	10/05/2015 10:17:03	UserLo	12	10/05/2015 10:17:03	UserLo	12	10/05/2015 10:17:03	UserLo
4	10/05/2015 10:21:31	UserLo	67	10/05/2015 10:21:31	UserLo	12	10/05/2015 10:21:31	UserLo	142	10/05/2015 10:21:31	UserLo
5	13/05/2015 21:10:11	UserIni	12	13/05/2015 21:10:11	UserIni	198	13/05/2015 21:10:11	UserIni	158	13/05/2015 21:10:11	UserIni



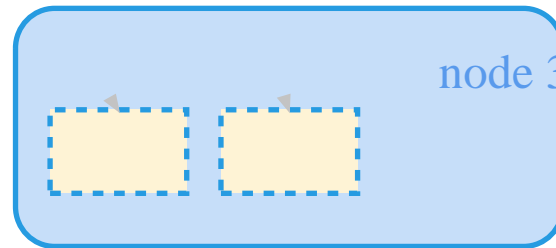
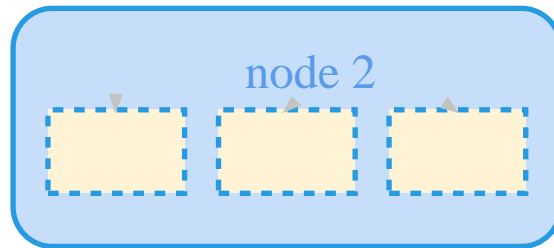
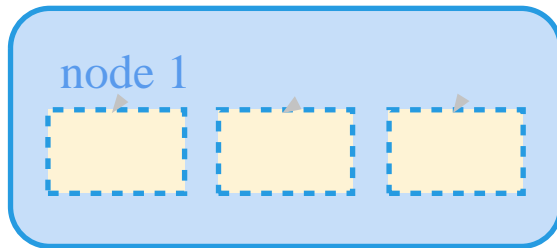
example: HadoopRDD

10	10/05/2015 10:14:01	UserInit	16	10/05/2015 10:14:01	UserInit	10	10/05/2015 10:14:01	UserInit	5	10/05/2015 10:14:01	UserInit
3	10/05/2015 10:14:55	FirstNa	20	10/05/2015 10:14:55	FirstNa	10	10/05/2015 10:14:55	FirstNa	4	10/05/2015 10:14:55	FirstNa
12	10/05/2015 10:17:03	UserLo	42	10/05/2015 10:17:03	UserLo	12	10/05/2015 10:17:03	UserLo	12	10/05/2015 10:17:03	UserLo
4	10/05/2015 10:21:31	UserLo	67	10/05/2015 10:21:31	UserLo	12	10/05/2015 10:21:31	UserLo	142	10/05/2015 10:21:31	UserLo
5	13/05/2015 21:10:11	UserIni	12	13/05/2015 21:10:11	UserIni	198	13/05/2015 21:10:11	UserIni	158	13/05/2015 21:10:11	UserIni



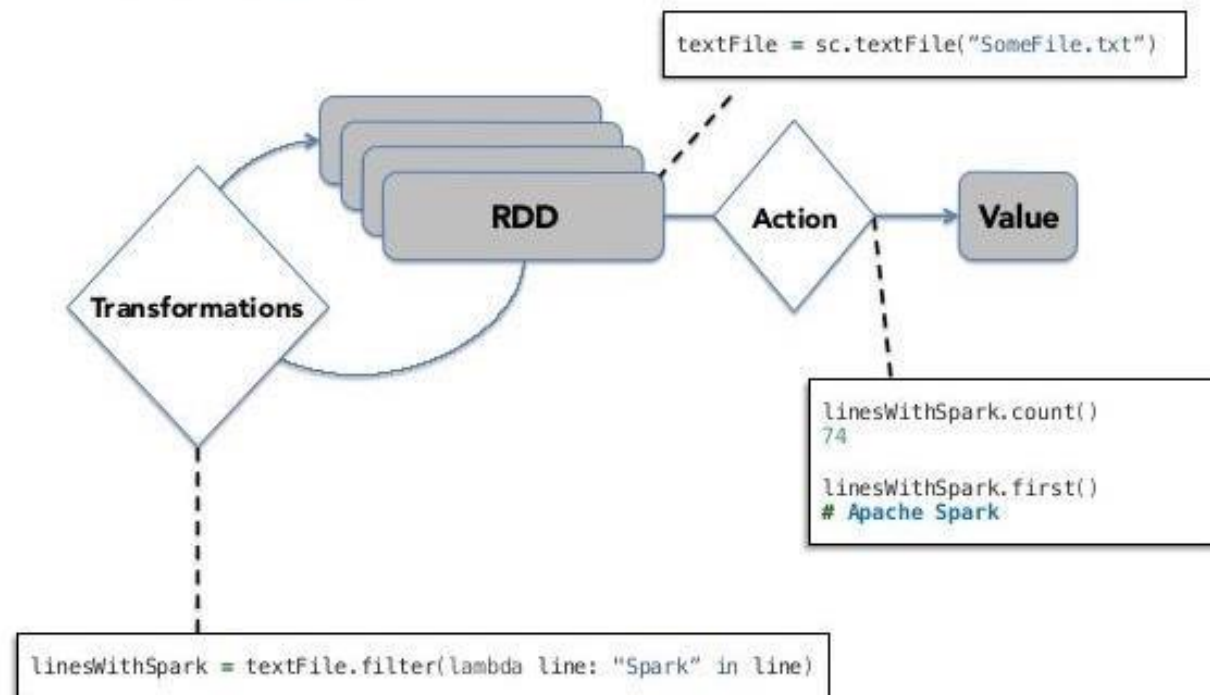
example: HadoopRDD

10	10/05/2015 10:14:01	UserInit	16	10/05/2015 10:14:01	UserInit	10	10/05/2015 10:14:01	UserInit	5	10/05/2015 10:14:01	UserInit
3	10/05/2015 10:14:55	FirstNa	20	10/05/2015 10:14:55	FirstNa	10	10/05/2015 10:14:55	FirstNa	4	10/05/2015 10:14:55	FirstNa
12	10/05/2015 10:17:03	UserLo	42	10/05/2015 10:17:03	UserLo	12	10/05/2015 10:17:03	UserLo	12	10/05/2015 10:17:03	UserLo
4	10/05/2015 10:21:31	UserLo	67	10/05/2015 10:21:31	UserLo	12	10/05/2015 10:21:31	UserLo	142	10/05/2015 10:21:31	UserLo
5	13/05/2015 21:10:11	UserIni	12	13/05/2015 21:10:11	UserIni	198	13/05/2015 21:10:11	UserIni	158	13/05/2015 21:10:11	UserIni



RDDs (Cont.)

- Lazy Evaluation
 - Waits for action to be called before distributing actions to worker nodes



Create RDD

- Can only be created using the SparkContext or by adding a Transformation to an existing RDD
- Using the SparkContext:

- Parallelized Collections – take an existing collection and run functions on it in parallel

```
rdd = sc.parallelize([ "some", "list", "to", "parallelize"], [numTasks])
```

- File Datasets – run functions on each record of a file in Hadoop distributed file system or any other storage system supported by Hadoop

```
rdd = sc.textFile("/path/to/file", [numTasks])
```

```
rdd = sc.objectFile("/path/to/file", [numTasks])
```

Spark in Action (Shell)

- Open shell
- Execute commands

Scala

```
val data = 1 to 5
val dataRDD = sc.parallelize(data)
val filteredDataRDD =
dataRDD.filter(_ < 3)
filteredDataRDD.collect()
```

Python

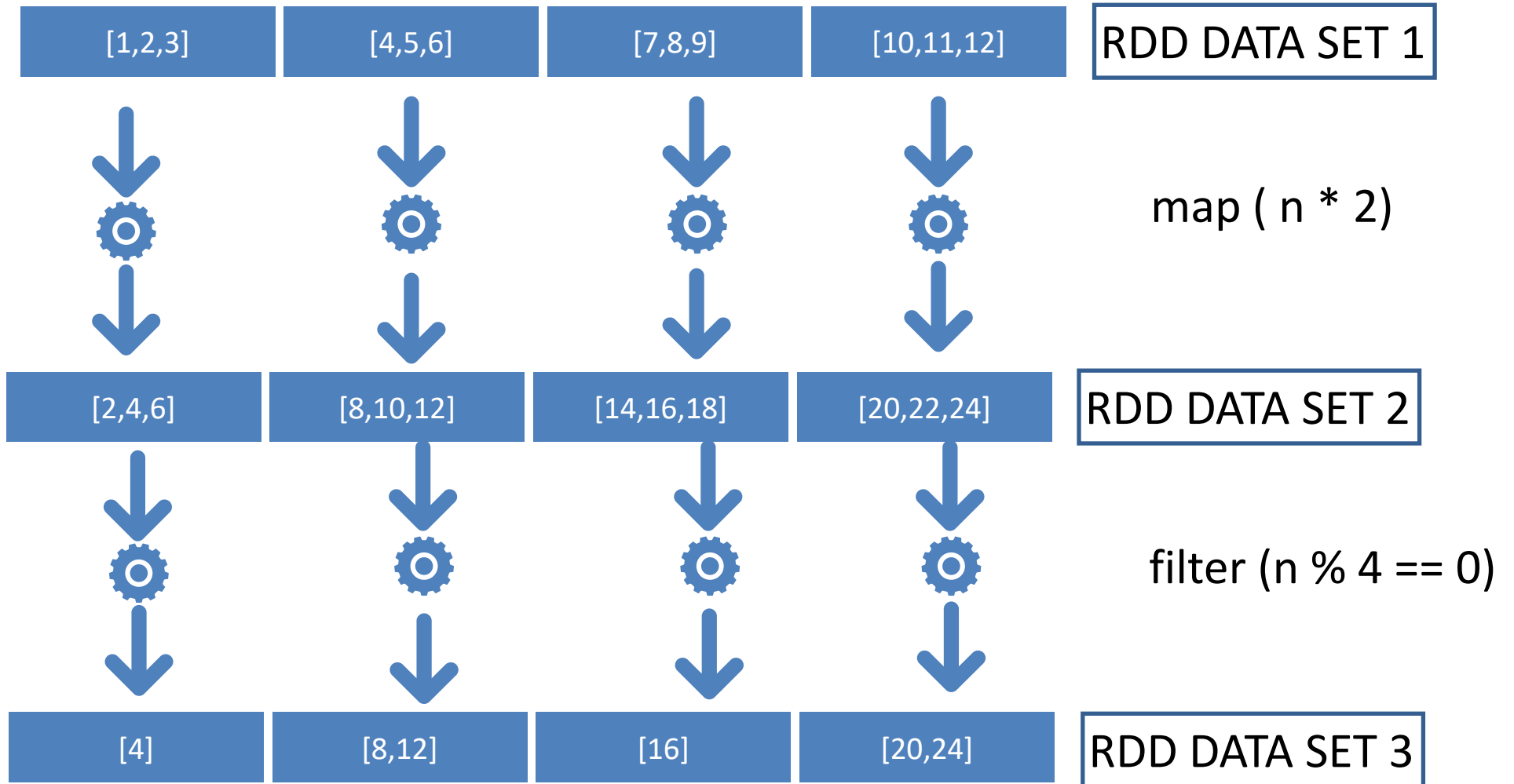
```
data = range(1, 6)
dataRDD = sc.parallelize(data)
filteredDataRDD =
dataRDD.filter(lambda x : x < 3)
filteredDataRDD.collect()
```

Operations

Two type of operations

- Transformation
- Action

RDD Transformation



RDD-Transformation

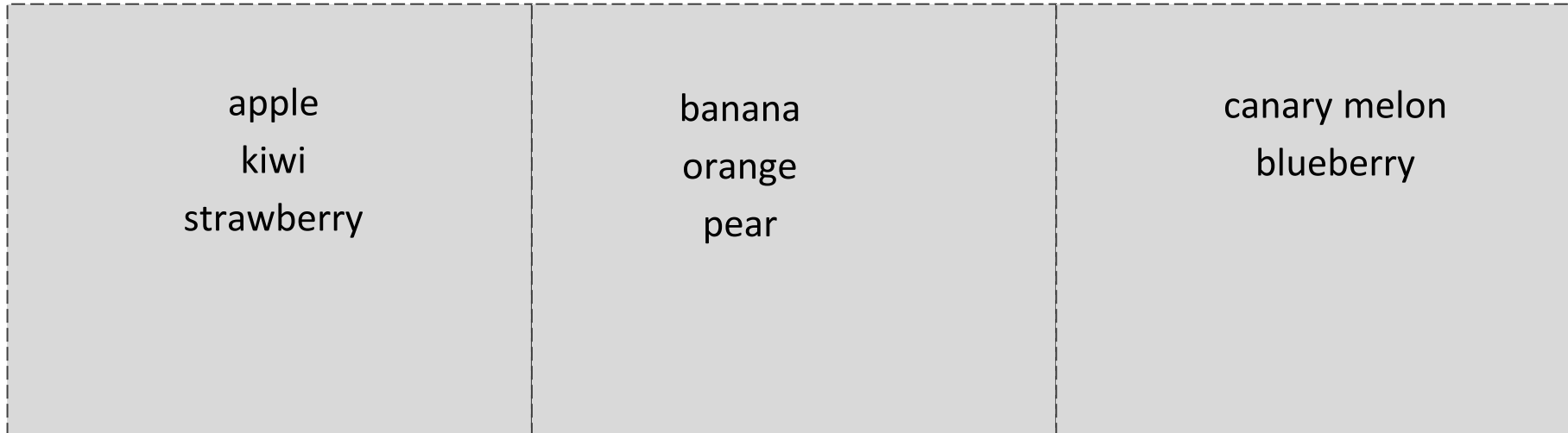
- Transformation - Operations Applied on RDD
- Every Transformation returns new RDD
- Examples map, filter methods

Spark Transmission

- Spark Transformation is a function that produces new RDD from the existing RDDs.
- It takes RDD as input and produces one or more RDD as output.
- Each time it creates new RDD when we apply any transformation.
- Input RDDs, cannot be changed since RDD are immutable in nature.

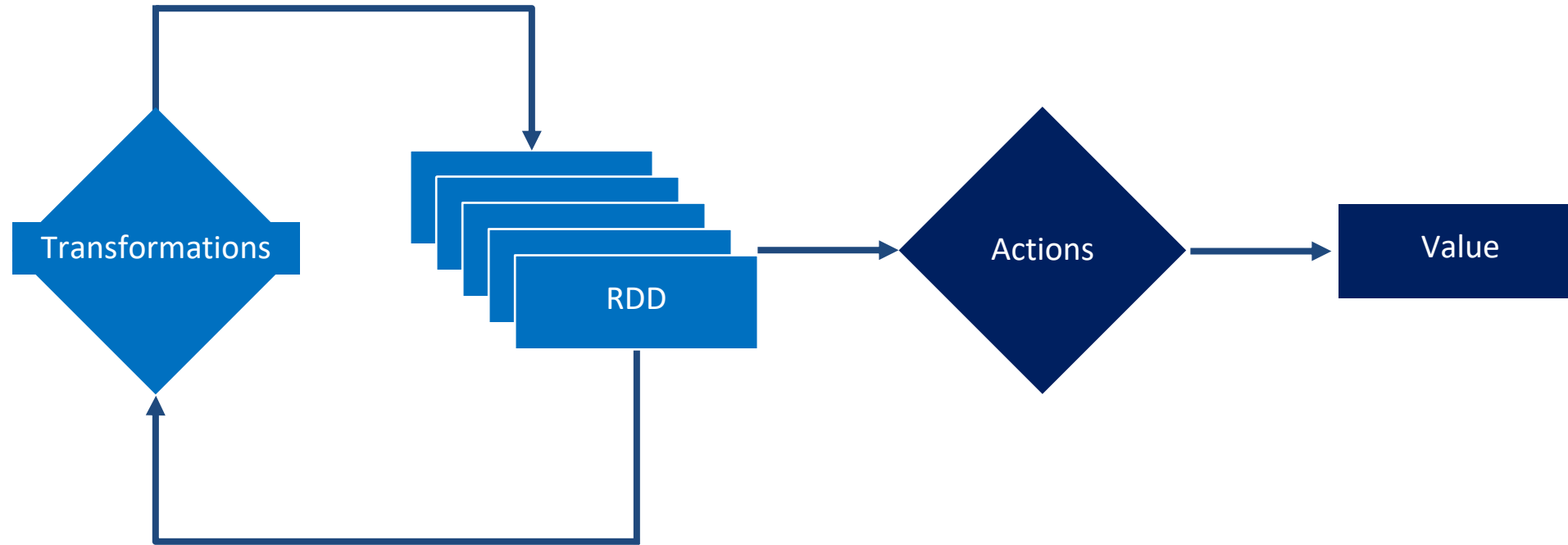
Creating RDDs

```
val fruits =  
spark.sparkContext.textFile("file:///example/data/fruits.txt")
```



- Methods to create a RDD:
 - Parallelize a collection (list)
 - Read data from an external source (blob, data lake store)

RDDs: Transformations and actions



Operations on RDDs

- There are two types of operations you can do on RDDs:
 - *Transformations*
 - *Actions*
- *Transformations* are lazily evaluated
- *Actions* are eagerly computed
- Any operation that takes an RDD in, and provides an RDD out, is a *transformation*
- Any operation that takes an RDD in, and outputs anything else, is an *action*
- At any time in the data processing operation, you can persist (cache) distributed data to memory or disk

Transformations (API) (map vs flatMap)

Contents:

Deer Bear River

Car Car River

Deer Car Bear

```
data.flatMap(line => line.split(" ")).collect()
```

```
//Returns:
```

```
//Array(Deer, Bear, River, Car, Car, River, Deer, Car, Bear)
```

```
data.map(line => line.split(" ")).collect()
```

```
//Returns:
```

```
//Array(Array(Deer, Bear, River), Array(Car, Car, River),
```

```
Array(Deer, Car, Bear))
```

Actions (API)

- `rdd.count() : Long`
 - Returns the number of elements in the RDD.
- `rdd.collect() : Array[T]`
 - Returns an array that contains all of the elements in this RDD.
- `rdd.reduce(Function<T,T> => R) : R`
 - Reduces the elements of this RDD using the specified commutative and associative binary operator.
- `rdd.saveAsTextFile("<Path>") : Unit`
 - Save this RDD as a text file, using string representations of elements.

MapReduce using Spark

Basic

```
data.flatMap(myMap)  
  .groupByKey()  
  .map((k, v) => myReduce(k, v))
```

With Combiner

```
data.flatMap(myMap)  
  .reduceByKey(myCombiner)  
  .map((k, v) => myReduce(k, v))
```

Java RDDs

`JavaRDDLike` - Parent Java RDD object for all Java RDD

`JavaRDD<T>` - Regular RDD

`JavaPairRDD<K, V>` - RDD with <key, value>

`JavaDoubleRDD` - RDD of only Double entries

`JavaHadoopRDD<K, V>` - An RDD that provides core functionality for reading data stored in Hadoop (e.g., files in HDFS, sources in HBase, or S3), using the older MapReduce API (`org.apache.hadoop.mapred`). hadoop 1.0

`JavaNewHadoopRDD<K, V>` - An RDD that provides core functionality for reading data stored in Hadoop (e.g., files in HDFS, sources in HBase, or S3), using the new MapReduce API (`org.apache.hadoop.mapreduce`) - hadoop 2.0

Java RDD Functions

- T - Input Type, R - Return Type, K - Key Type, V - Value Type
- `DoubleFlatMapFunction<T>`
 - Returns Iterable of Doubles
- `DoubleFunction<T>`
 - Returns Double
- `FlatMapFunction<T,R>`, `FlatMapFunction2<T1,T2,R>`
 - Returns Iterable of type R
- `Function0<R>`, `Function<T, R>`, `Function2<T1,T2,R>`,
`Function3<T1,T2,T3,R>`
 - Returns value of type R
- `PairFlatMapFunction<T,K,V>`
 - Returns Iterable of Tuple with type <K,V>
- `PairFunction<T,K,V>`
 - Returns single Tuple with type <K,V>
- `VoidFunction<T>`
 - Returns void

Creating PairRDDs

Scala

```
rdd.map(key => (key, "value"))
```

Python

```
rdd.map(lambda key: (key, "value"))
```

Java

```
rdd.mapToPair(new PairFunction<String, String, String>() {  
    public Tuple2<String, String> call(String key) {  
        return new Tuple2<String, String>(key, "value");  
    }  
});
```

Accessing Tuple Attributes

Scala

```
pairRDD.foreach(tuple => {  
    val key    = tuple._1  
    val value = tuple._2  
})
```

Python

```
pairRDD.foreach(lambda tuple:  
    key    = tuple[0]  
    value = tuple[1]  
)
```

Java

```
pairRDD.foreach(new VoidFunction<Tuple2<String, Integer>>() {  
    public void call(Tuple2<String, Integer> tuple) {  
        String key    = tuple._1();  
        Integer value = tuple._2();  
    }  
});
```



Exercise 2 – Access Logs

See “Setup and Exercise” Document

Spark Operations

Transformations (define a new RDD)	map filter sample groupByKey reduceByKey sortByKey	flatMap union join cogroup cross mapValues
Actions (return a result to driver program)	collect reduce count save lookupKey	

RDD Lineage Graph

```
val textFile = spark.textFile("/path/to/file.txt")
val counts = textFile.flatMap(line => line.split(" "))
                    .map(word => (word, 1))
                    .reduceByKey(_ + _)
counts.toDebugString
```

res1: String =

(1) ShuffledRDD[7] at reduceByKey at <console>:23 []

+-(1) MapPartitionsRDD[6] at map at <console>:23 []

| MapPartitionsRDD[5] at flatMap at <console>:23 []

| /path/to/file.txt MapPartitionsRDD[3] at textFile at <console>:21 []

| /path/to/file.txt HadoopRDD[2] at textFile at <console>:21 []



RDD Dependencies

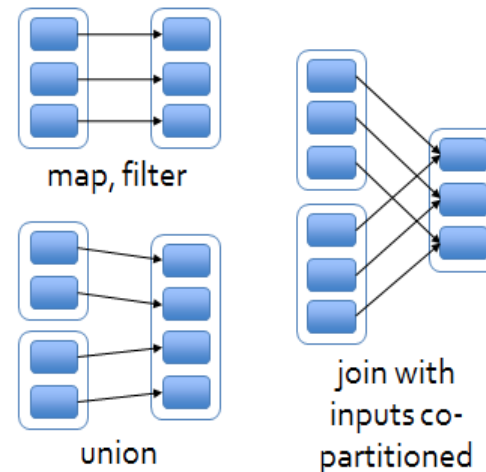
Narrow Dependencies

- Allow pipelined execution on one cluster node
- Recovery after a node failure is more efficient. Only the lost parent partitions need to be recomputed which can be done in parallel on different nodes.

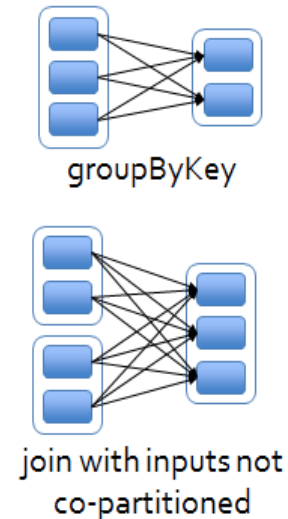
Wide Dependencies

- Requires data from all parent partitions to be available and to be shuffled across the nodes using a MapReduce like shuffling algorithm.
- A single failed node might cause the loss of a partition from all the ancestors of an RDD, requiring a complete re-execution.

“Narrow” deps:



“Wide” (shuffle) deps:



Berkely.edu, *RDD Dependencies*

<http://www.eecs.berkeley.edu/Pubs/TechRpts/2011/EECS-2011-82.pdf>

RDD Dependencies (Cont.)

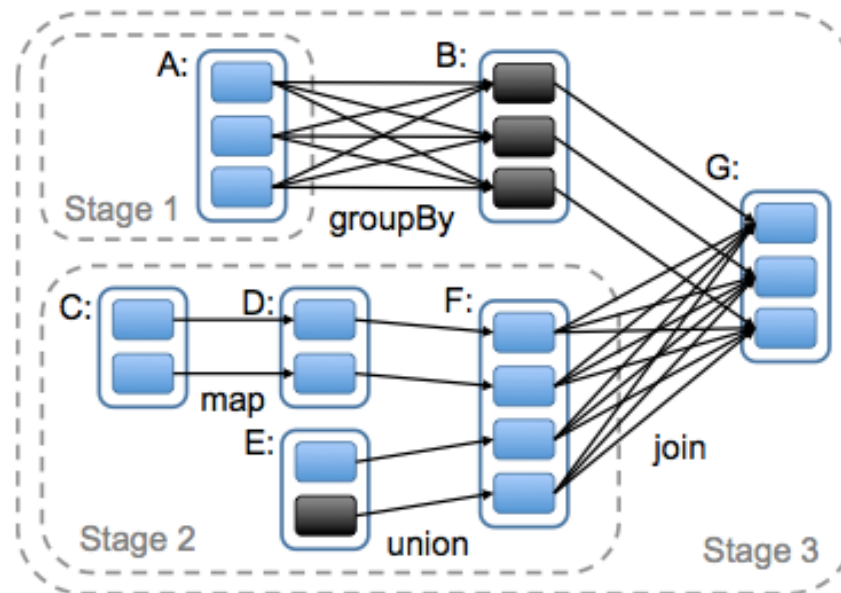


Figure 5: Example of how Spark computes job stages. Boxes with solid outlines are RDDs. Partitions are shaded rectangles, in black if they are already in memory. To run an action on RDD G, we build stages at wide dependencies and pipeline narrow transformations inside each stage. In this case, stage 1's output RDD is already in RAM, so we run stage 2 and then 3.

RDD Persistence

- Each node stores any partitions of it that it computes in memory and reuses them in other actions on that dataset.
- After making an RDD to be persisted, the first time the dataset is computed in an action, it will be kept in memory on the nodes.
- Allows future actions to be much faster (often by more than 10x) since you're not recomputing some data every time you perform an action.
- If data is too big to be cached, then it will spill to disk and memory will gradually degrade
- Least Recently Used (LRU) replacement policy

RDD Persistence (Storage Levels)

Storage Level	MEANING
MEMORY_ONLY	Store RDD as deserialized Java objects in the JVM. If the RDD does not fit in memory, some partitions will not be cached and will be recomputed on the fly each time they're needed. This is the default level.
MEMORY_AND_DISK	Store RDD as deserialized Java objects in the JVM. If the RDD does not fit in memory, store the partitions that don't fit on disk, and read them from there when they're needed.
MEMORY_ONLY_SER	Store RDD as serialized Java objects (one byte array per partition). This is generally more space-efficient than deserialized objects, especially when using a fast serializer, but more CPU-intensive to read.
MEMORY_AND_DISK_SER	Similar to MEMORY_ONLY_SER, but spill partitions that don't fit in memory to disk instead of recomputing them on the fly each time they're needed.
DISK_ONLY	Store the RDD partitions only on disk.
MEMORY_ONLY_2, MEMORY_AND_DISK_2, etc.	Same as the levels above, but replicate each partition on two cluster nodes.

Persist (API)

`rdd.persist()`

`rdd.persist(StorageLevel)`

- Persist this RDD with the default storage level (MEMORY_ONLY).
- You can override the StorageLevel for fine grain control over persistence

Caching (API)

```
rdd.cache()
```

- Persists the RDD with the default storage level (MEMORY_ONLY)

Checkpoint (API)

```
rdd.checkpoint()
```

- RDD will be saved to a file inside the checkpoint directory set with `SparkContext#setCheckpointDir("/path/to/dir")`
- Used for RDDs with long lineage chains with wide dependencies since it would be expensive to recompute
- For now it is left to the user when to checkpoint
 - In the future spark will automatically checkpoint for you
- Can be expensive to replicate a large amount of data
 - Writing data to disk



Unpersist (API)

```
rdd.unpersist()
```

- Marks it as non-persistent and/or removes all blocks of it from memory and disk

Persistence Example

```
val textFile = sc.textFile("/path/to/file.txt")
```

```
textFile.cache()//<-Data not cached on workers yet
```

```
val counts = textFile  
    .flatMap(line => line.split(" "))  
    .map(word => (word, 1))  
    .reduceByKey(_ + _)
```

```
counts.collect()//<-Action executes graph which caches data
```

Fault Tolerance

- RDDs contain lineage graphs (coarse grained updates/transformations) to help it rebuild partitions that were lost
- Only the lost partitions of an RDD need to be recomputed upon failure.
- They can be recomputed in parallel on different nodes without having to roll back the entire app
- Also lets a system tolerate slow nodes (stragglers) by running a backup copy of the troubled task.
- Original process on straggling node will be killed when new process is complete
- Cached/Check pointed partitions are also used to recompute lost partitions if available in shared memory

Scheduler

- Uses RDD lineage to find efficient execution plan for action
 - Optimizes operations by collapsing down inline narrow dependencies into one task instead of doing individual tasks for each operation
- Schedules tasks based on data locality
- Takes into account which RDDs are in cache
 - If a task needs to process a cached partition, then the task is started on the node where the data is cached
- Tasks are launched to compute missing partitions



Accumulators

- Accumulators are variables that can only be “added” to through an associative operation
- Used to implement counters and sums, efficiently in parallel
- Spark natively supports accumulators of numeric value types and standard mutable collections, and programmers can extend for new types
- Only the driver program can read an accumulator’s value, not the tasks

Accumulators (Code)

Scala

```
val accum = sc.accumulator(0)

rdd.foreach(entry =>
    accum += 1
)

println(accum.value)
```

Python

```
accum = sc.accumulator(0)

def foreachFunction(x):
    global accum
    accum += x

rdd.foreach(foreachFunction)

print(accum.value)
```

Broadcast Variables

- Broadcast variables let programmer keep a read-only variable cached on each machine rather than shipping a copy of it with tasks
- For example, to give every node a copy of a large input dataset efficiently
- Spark also attempts to distribute broadcast variables using efficient broadcast algorithms to reduce communication cost

Broadcast Variables (Code)

Scala

```
var list = List(1,2,3)

val listBroadcasted=
sc.broadcast(list)

val rdd = sc.textFile("hdfs://...")

rdd.foreach( entry =>
    print(listBroadcasted.value)
    //Prints: List(1,2,3)
)
```

Python

```
list = [1,2,3]

listBroadcasted =
sc.broadcast(list)

rdd = sc.textFile("hdfs://...")

rdd.foreach( lambda entry:
    print(listBroadcasted.value)
    # Prints: [1,2,3]
)
```

Spark Transmission

- Applying transformation built an RDD lineage, with the entire parent RDDs of the final RDD(s).
- RDD lineage, also known as RDD operator graph or RDD dependency graph.
- It is a logical execution plan
- Represented as Directed Acyclic Graph (DAG) of the entire parent RDDs of RDD.

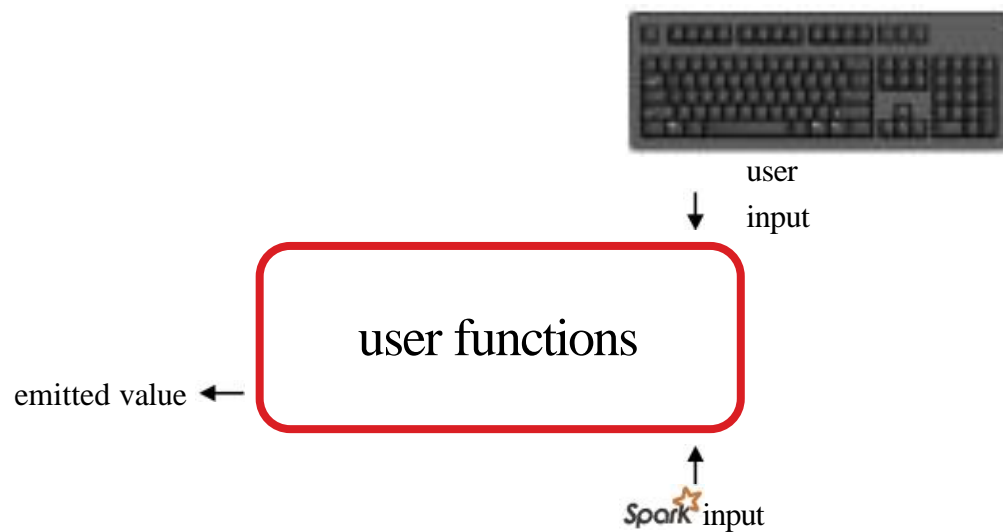
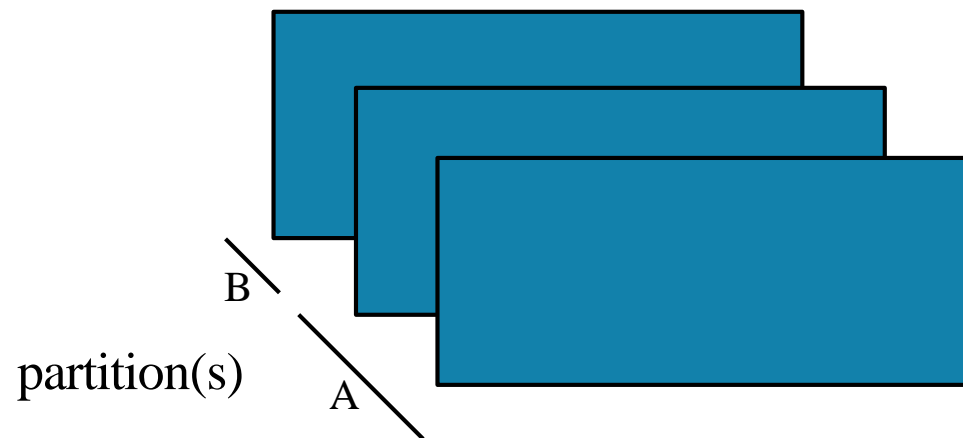
RDD-Actions

- Causes the full execution of transformations
- Involves both spark driver as well as the nodes
- Example - Take(): Brings back the data to driver

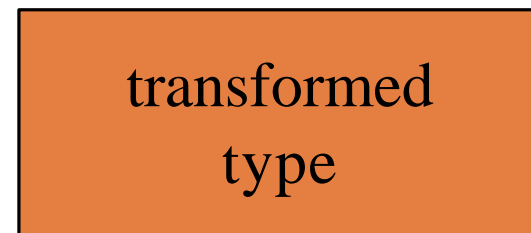
RDD



Legend



RDD Elements

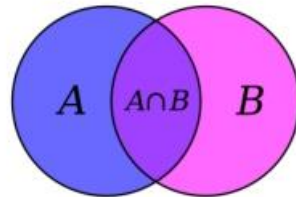




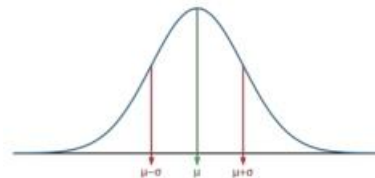
Legend



Randomized operation



Set Theory / Relational operation



Numeric calculation

Spark  Operations =

TRANSFORMATION

+

ACTIONS

 = easy

 = medium

Essential Core & Intermediate Spark Operations

Transform

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

Action


- 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

Transform

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

Action

- keys
- values

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



VS

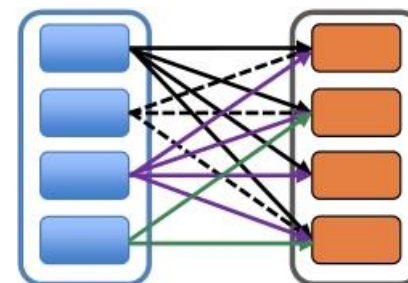
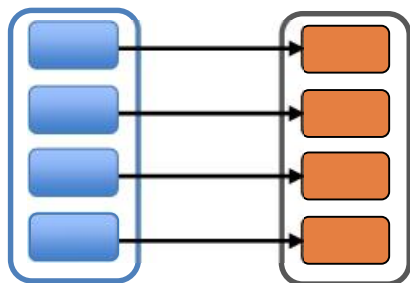


narrow

wide

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

*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.”

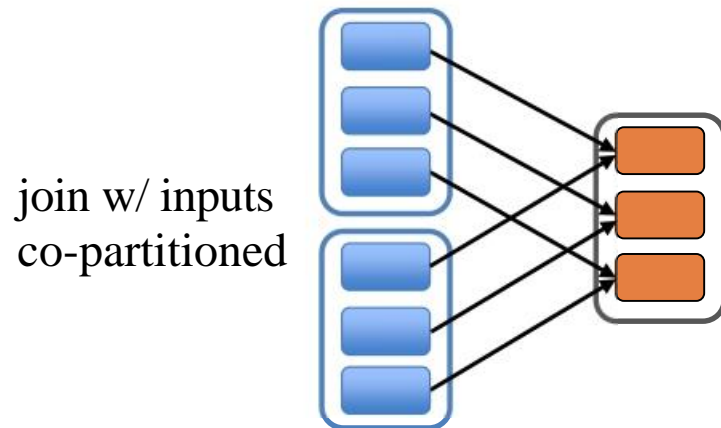
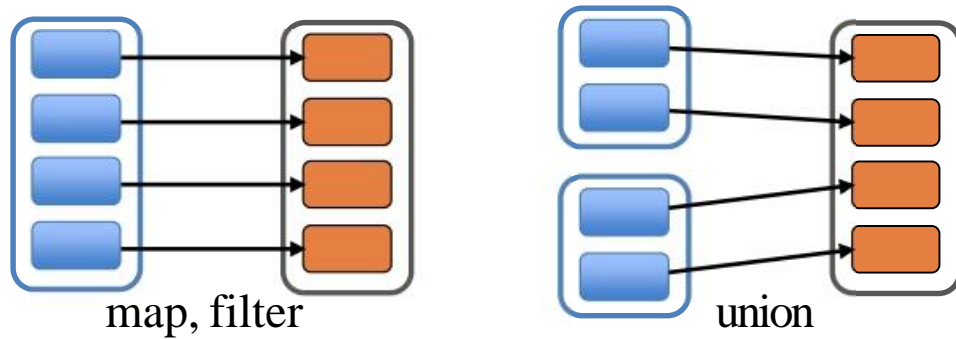
“The most interesting question in designing this interface is how to represent dependencies between RDDs.”

“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.”

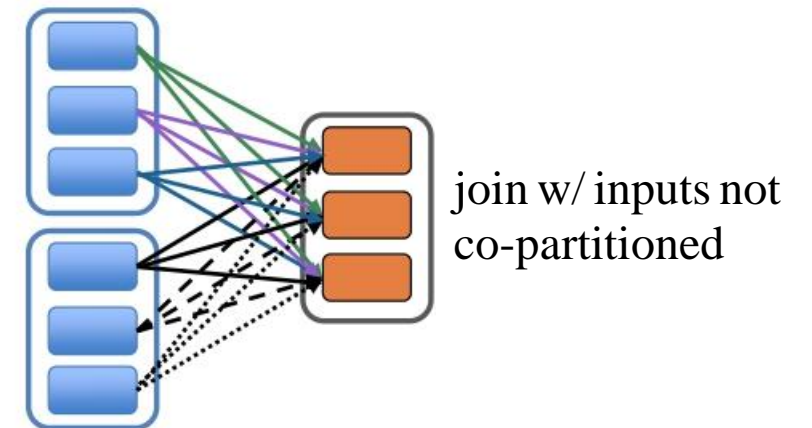
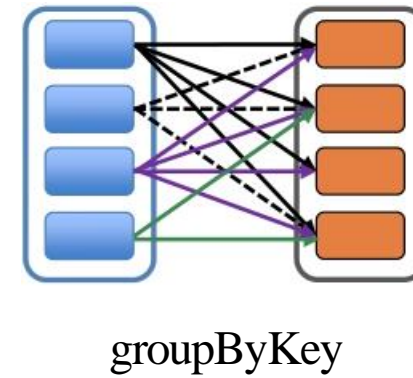
narrow

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

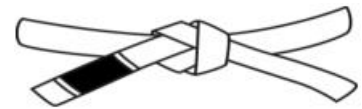


wide

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



TRANSFORMATION



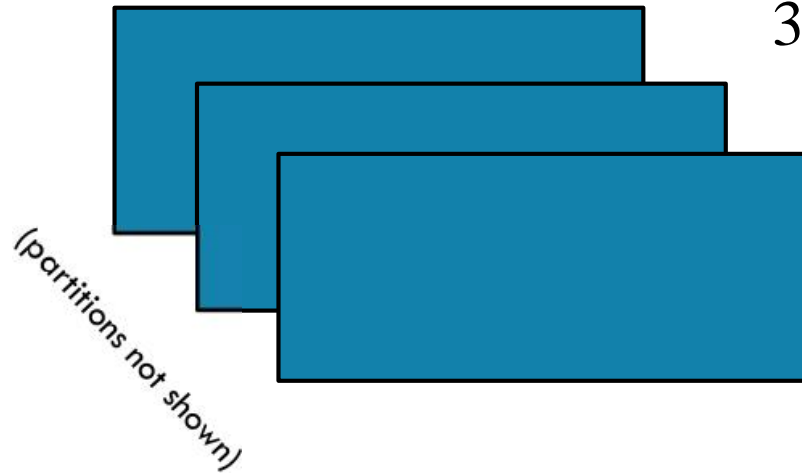
Core Operations



MAP

RDD: **x**

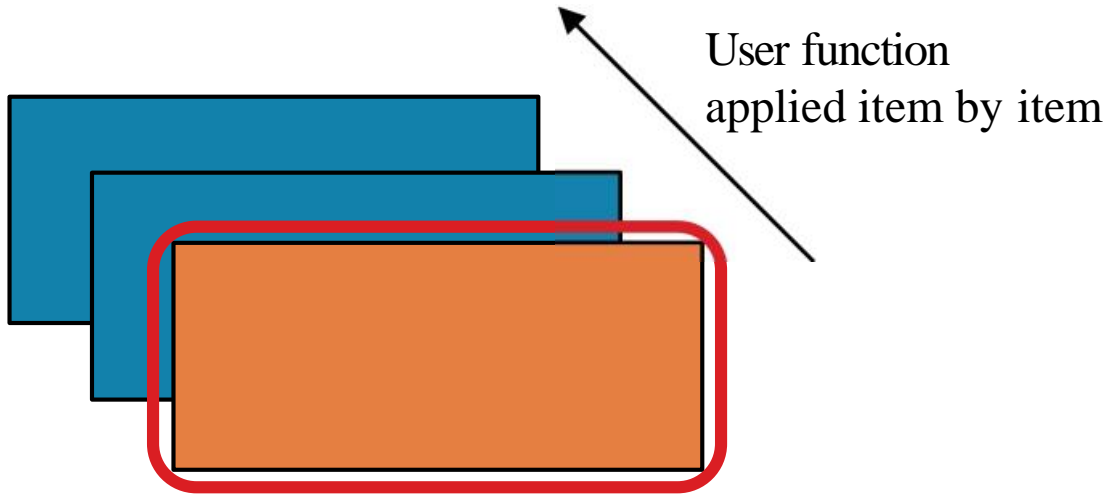
3 items in RDD





MAP

RDD: **x**



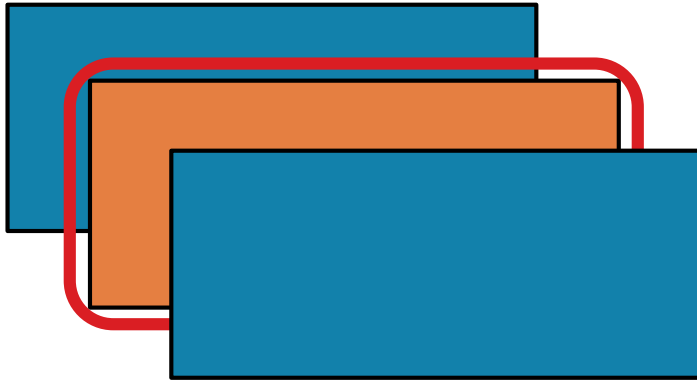
RDD: **y**





MAP

RDD: **x**



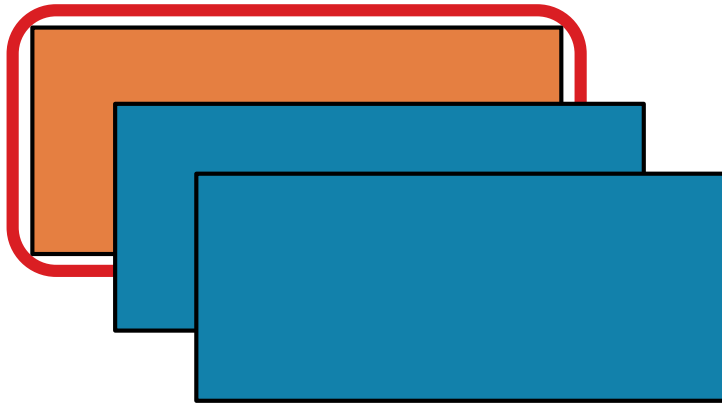
RDD: **y**



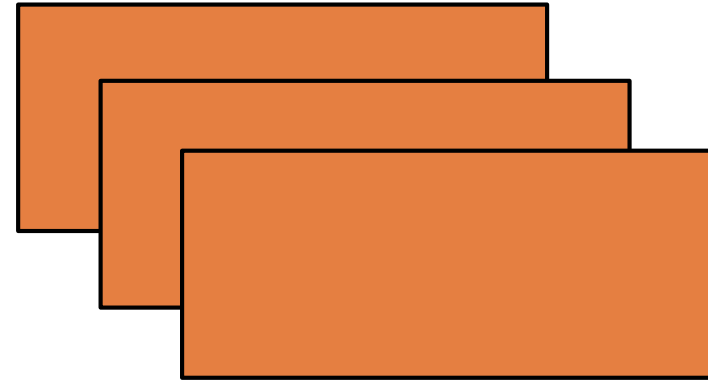


MAP

RDD: **x**



RDD: **y**





MAP

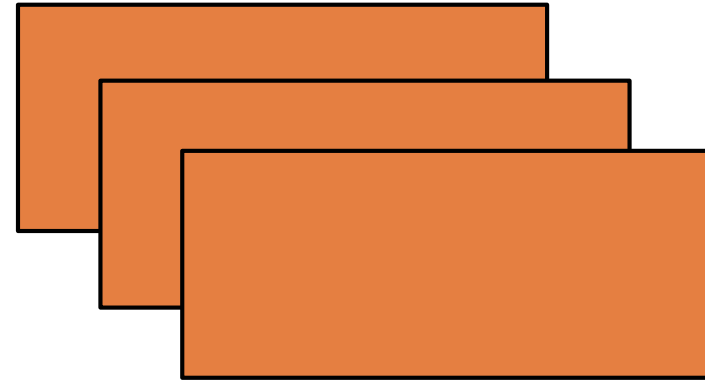
After `map()` has been applied...

RDD: **x**



before

RDD: **y**

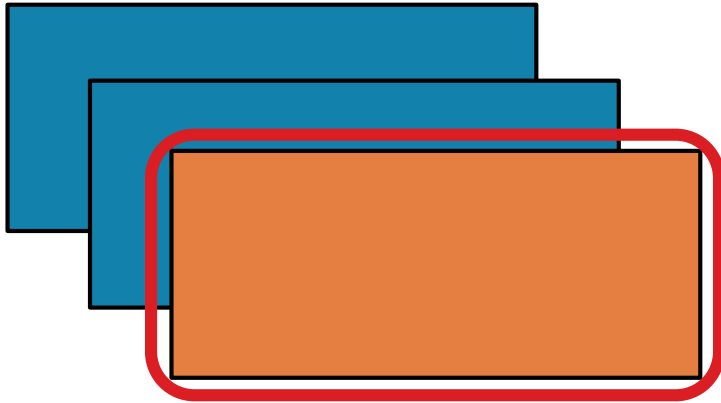


after

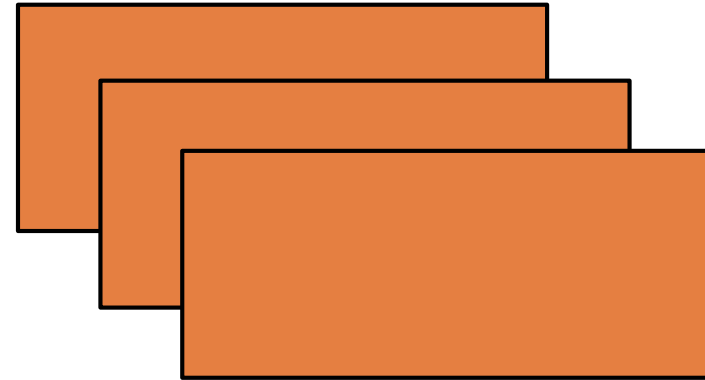


MAP

RDD: **x**

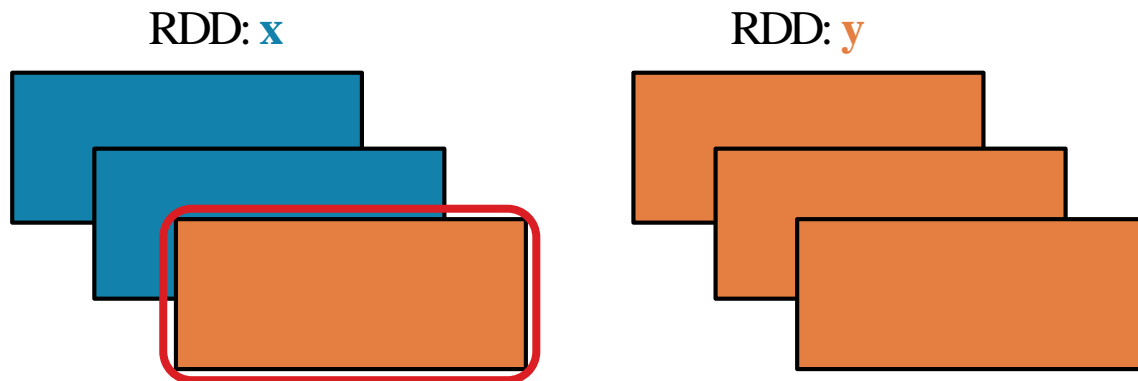


RDD: **y**



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

MAP



`map(f, preservesPartitioning=False)`

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



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

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

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



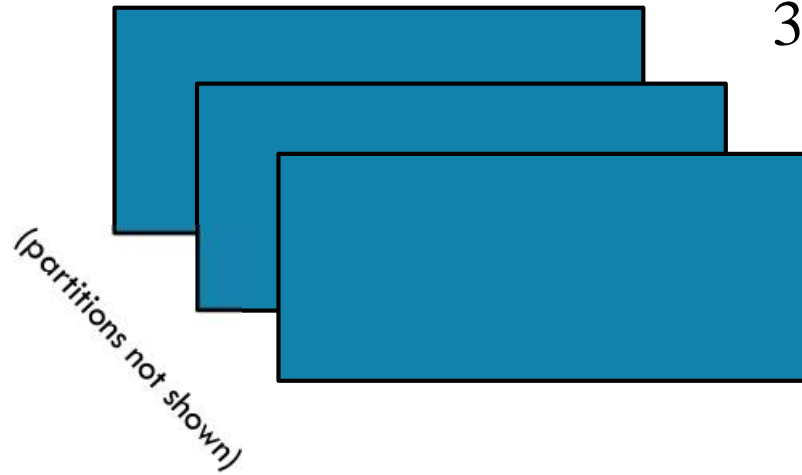
```
val x = sc.parallelize(Array("b", "a", "c"))
val y = x.map(z => (z,1))
println(x.collect().mkString(", "))
println(y.collect().mkString(", "))
```



FILTER

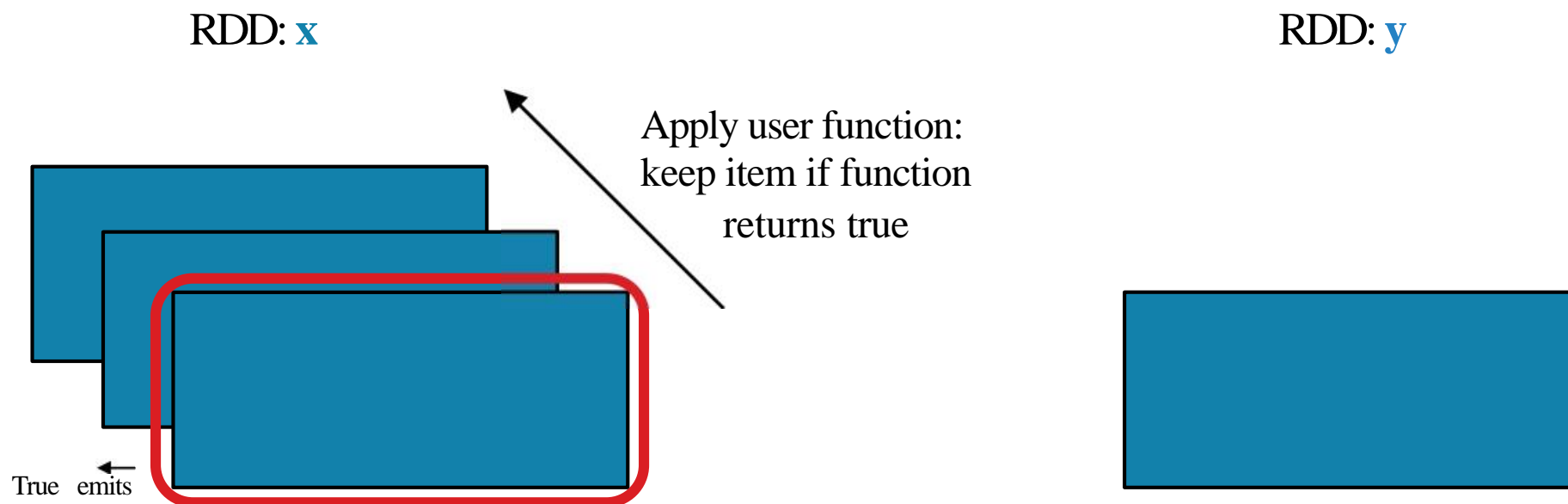
RDD: **x**

3 items in RDD





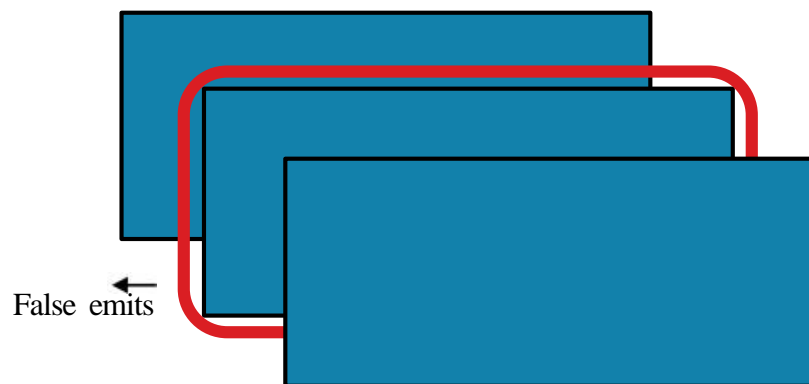
FILTER



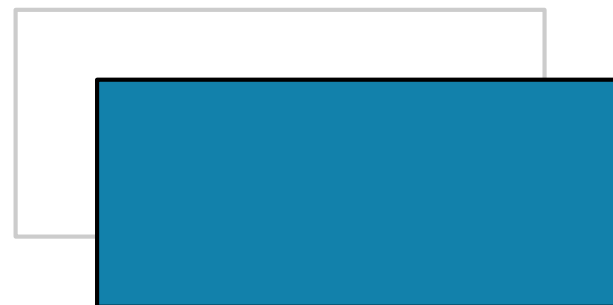


FILTER

RDD: **x**



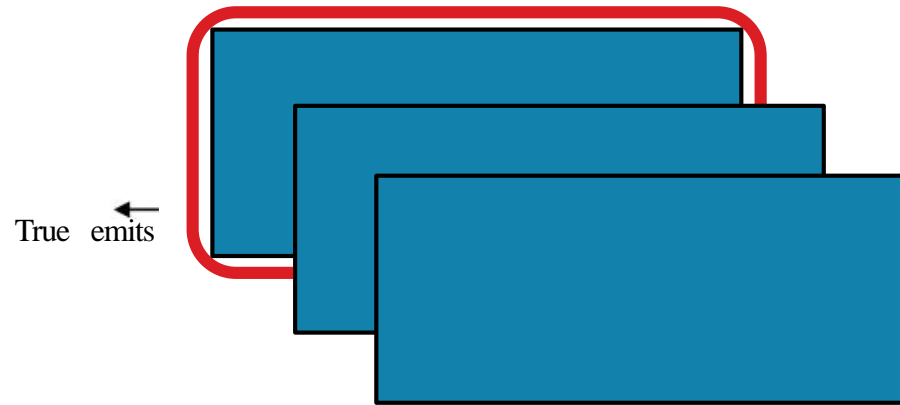
RDD: **y**





FILTER

RDD: **x**



RDD: **y**

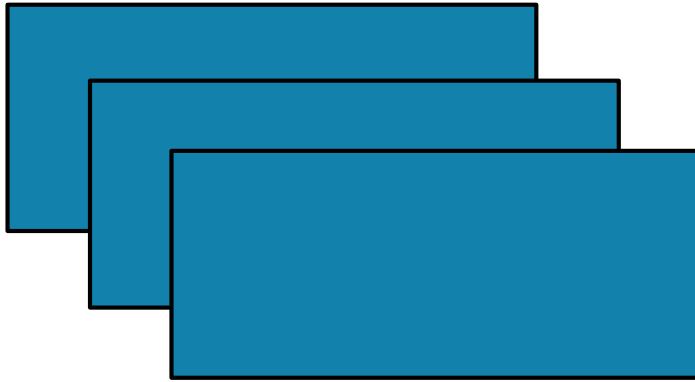


FILTER



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

RDD: **x**



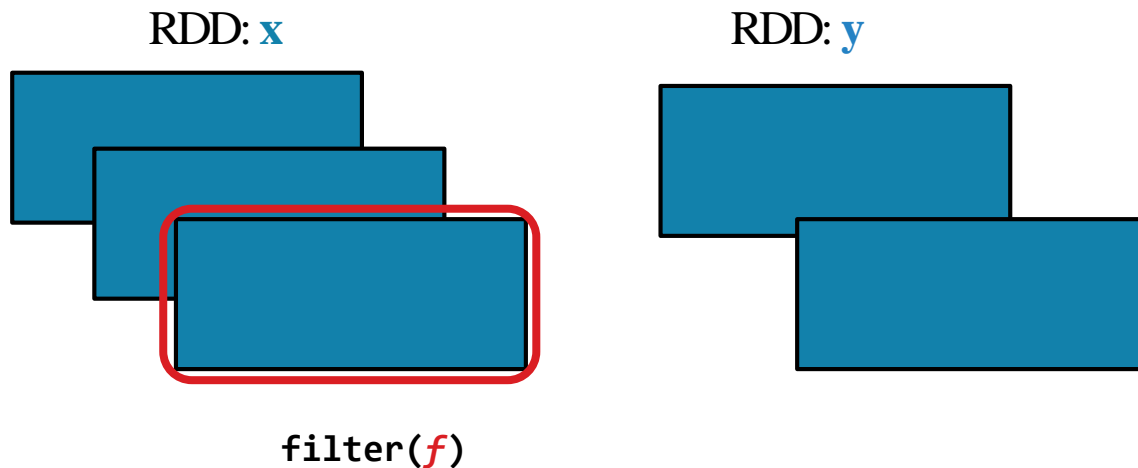
before

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]

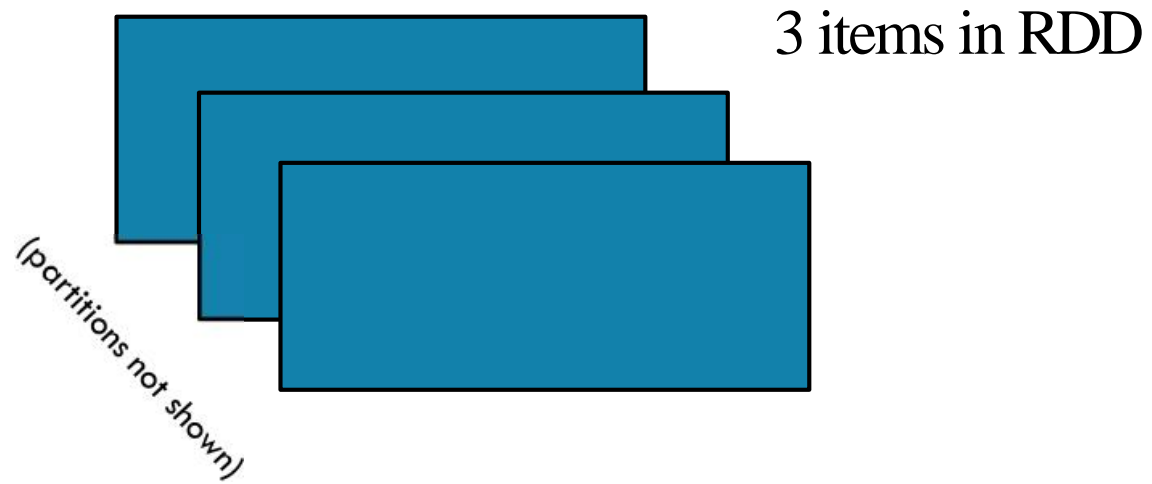
y : [1, 3]



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

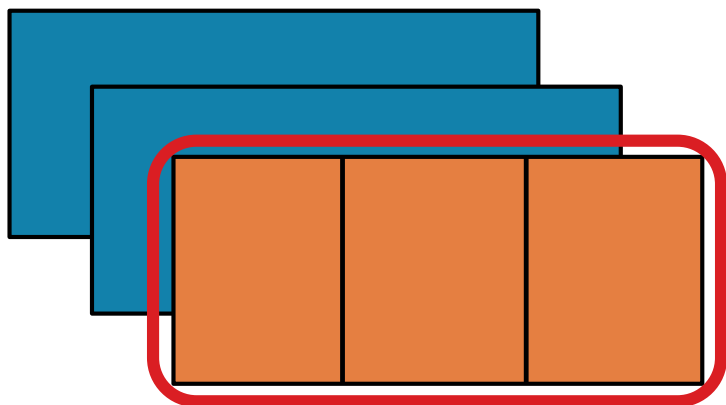


FLATMAP

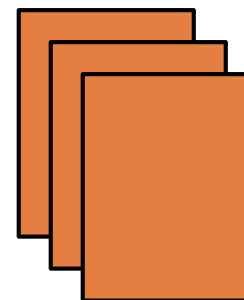


FLATMAP

RDD: **x**

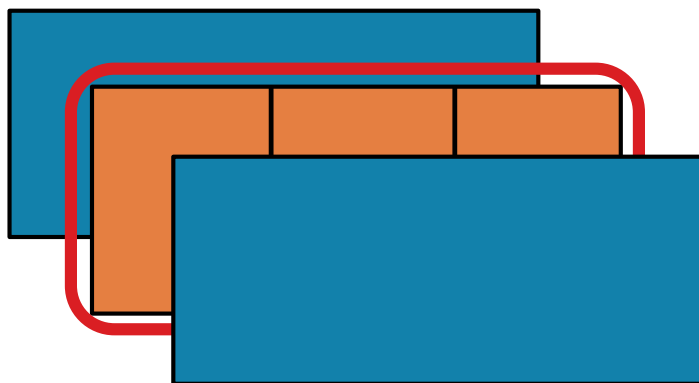


RDD: **y**

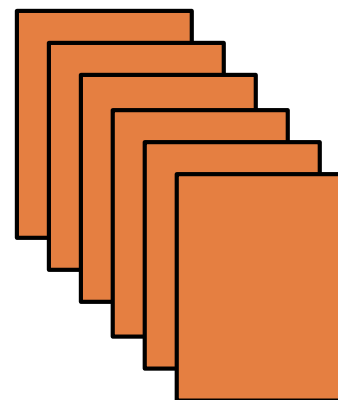


FLATMAP

RDD: **x**

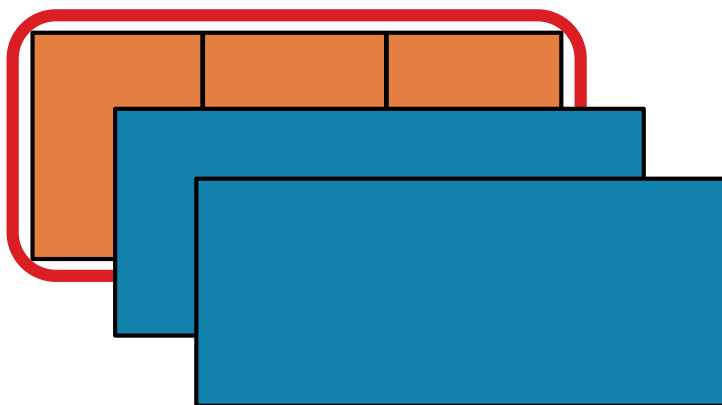


RDD: **y**

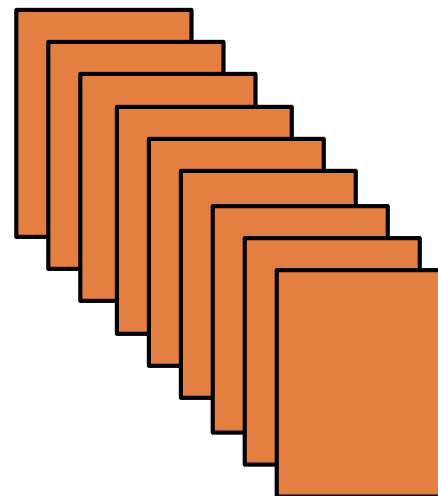


FLATMAP

RDD: **x**



RDD: **y**



FLATMAP

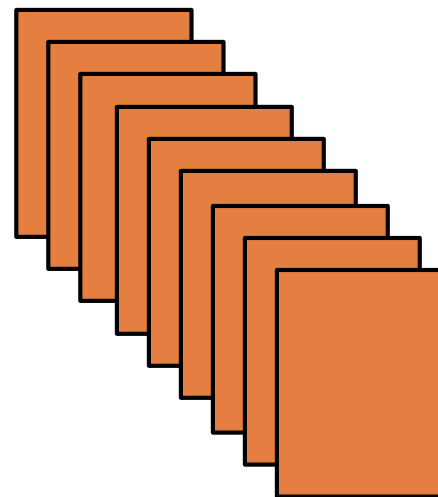
After `flatMap()` has been applied...

RDD: **x**



before

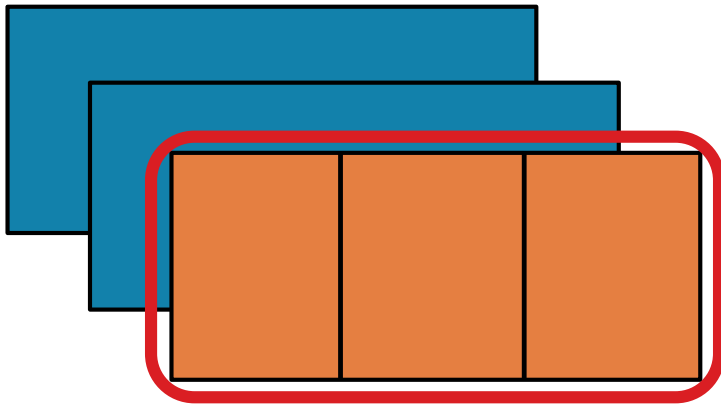
RDD: **y**



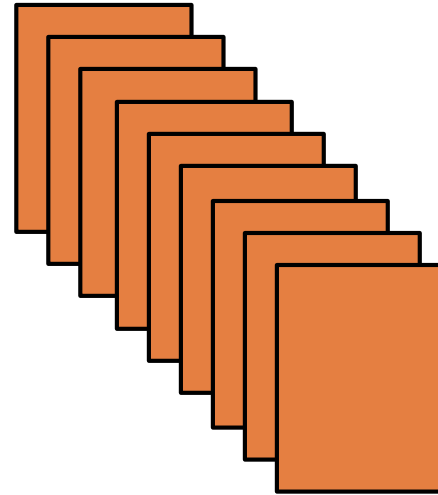
after

FLATMAP

RDD: **x**

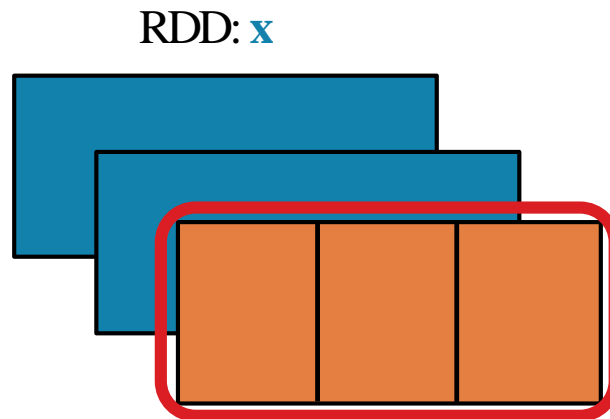


RDD: **y**



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]

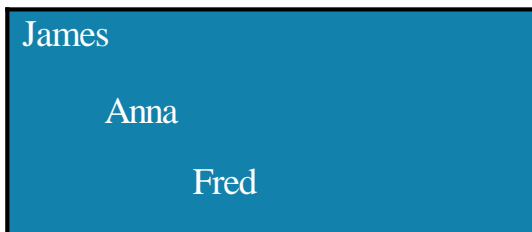


```
val x = sc.parallelize(Array(1,2,3))
val y = x.flatMap(n => Array(n, n*100, 42))
println(x.collect().mkString(", "))
println(y.collect().mkString(", "))
```

GROUPBY

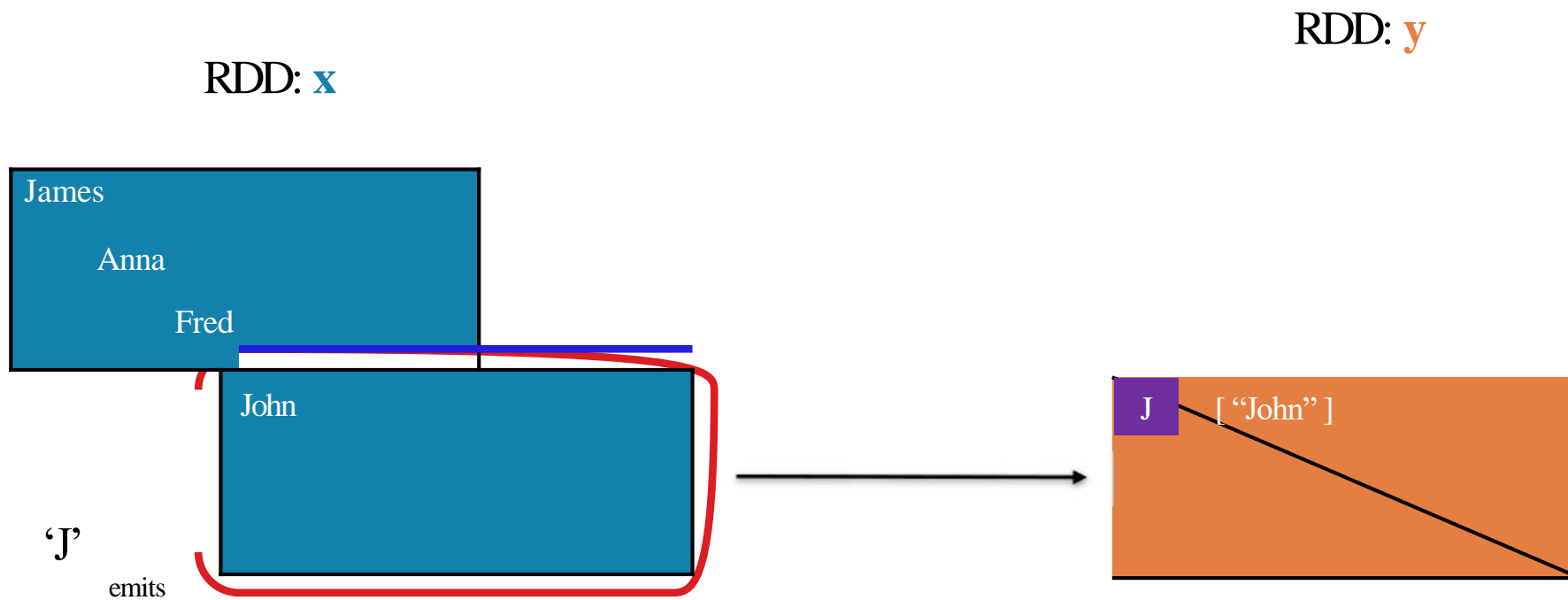
RDD: **x**

4 items in RDD



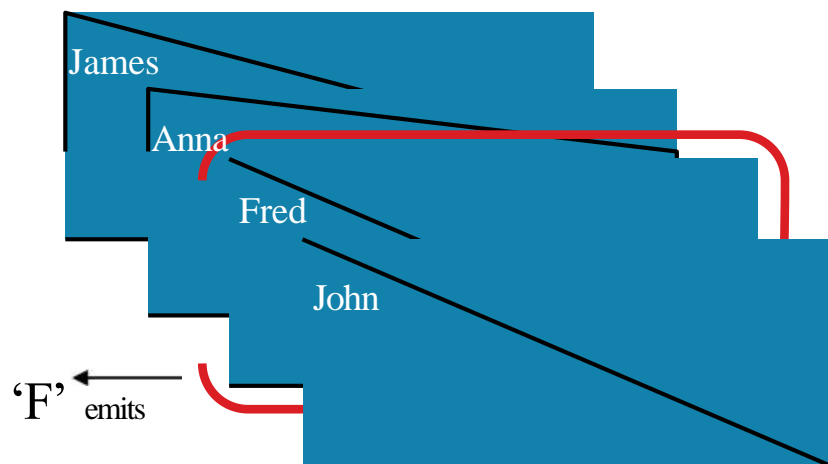
(partitions not shown)

GROUPBY

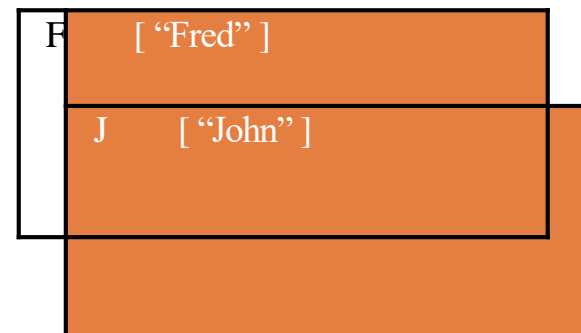


GROUPBY

RDD: **x**



RDD: **y**

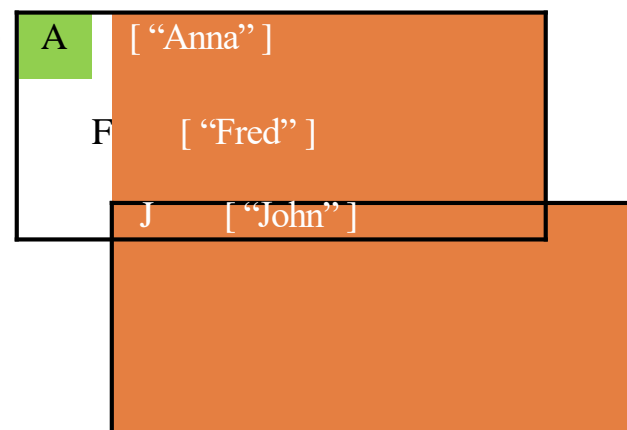


GROUPBY

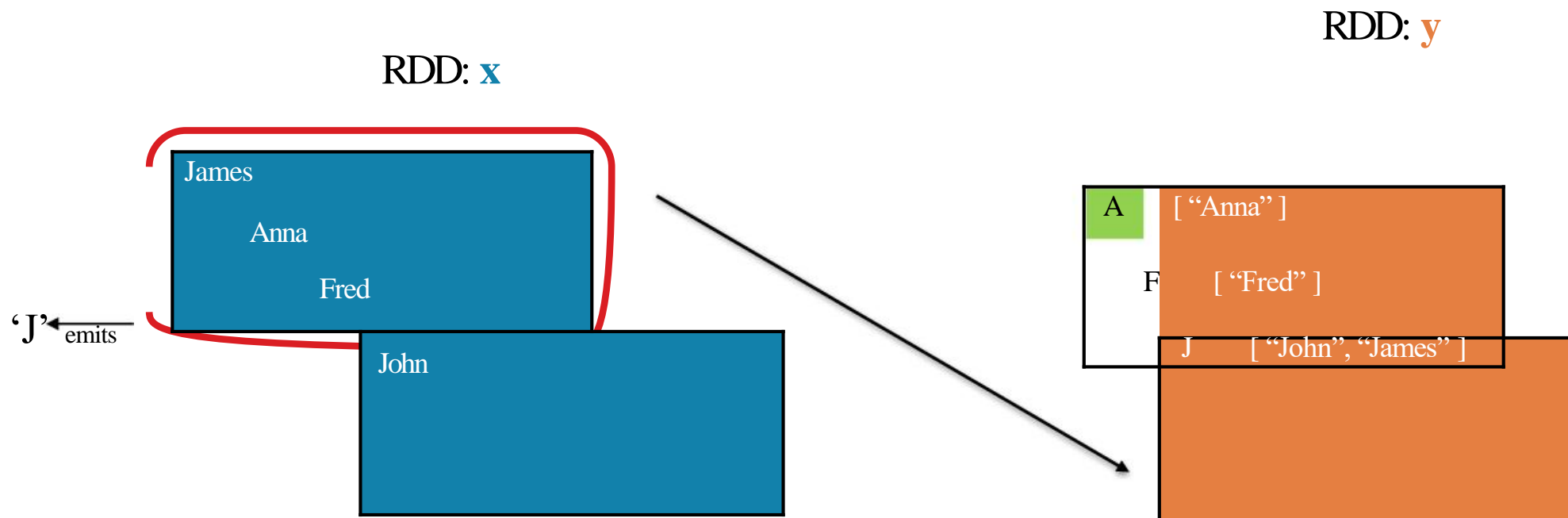
RDD: **x**



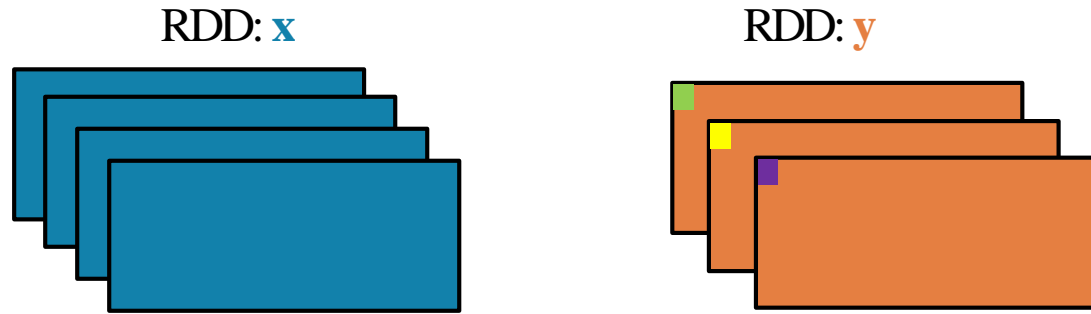
RDD: **y**



GROUPBY



GROUPBY



`groupBy(f, numPartitions=None)`

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



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

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

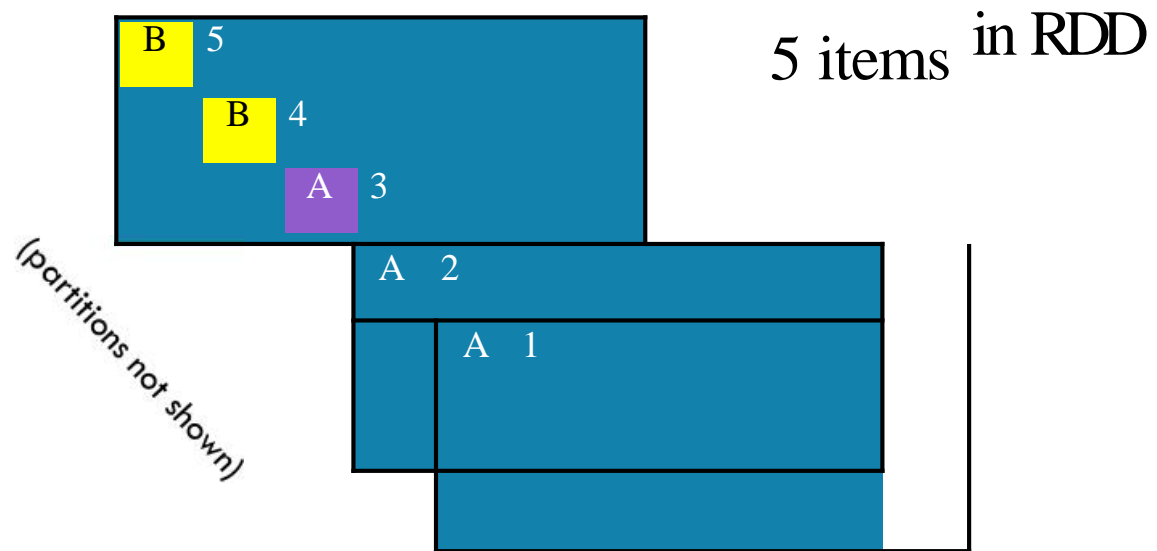
y: [('A', ['Anna']), ('J', ['John', 'James']), ('F', ['Fred'])]
]



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

GROUPBY

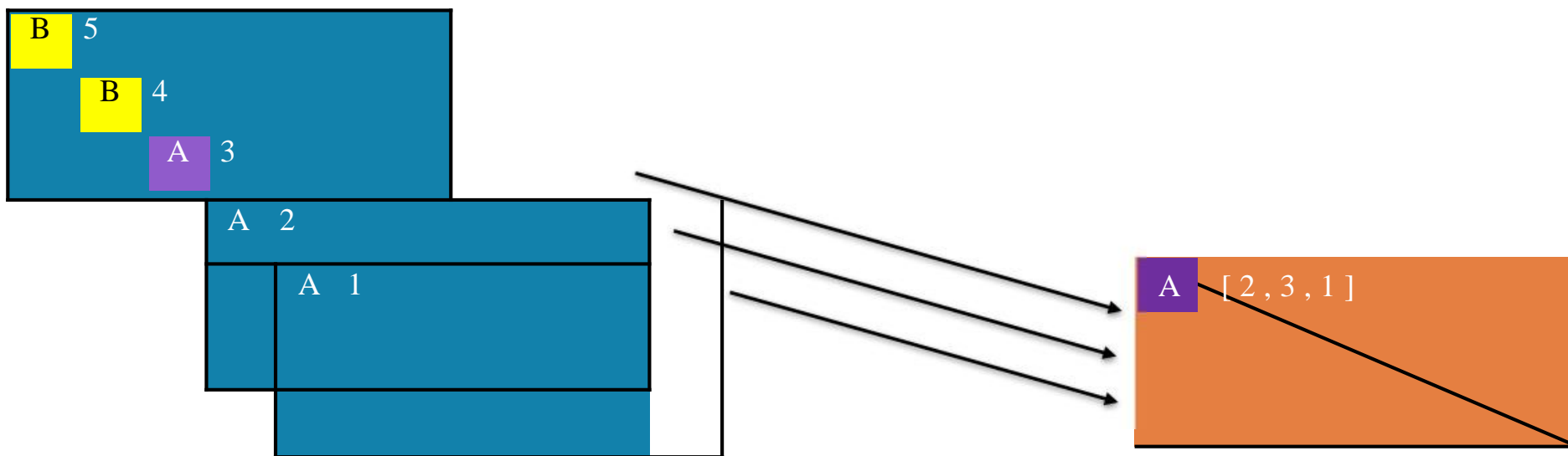
Pair RDD: **x**



GROUPBY

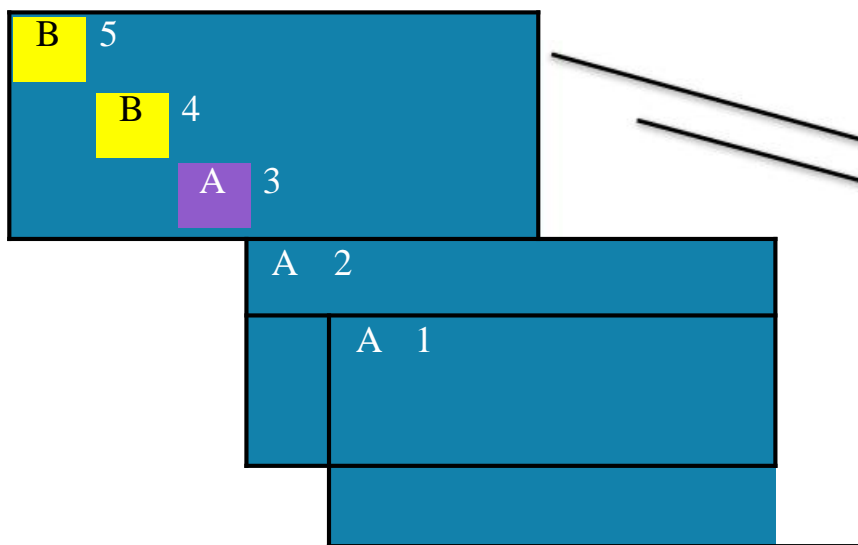
Pair RDD: **x**

RDD: **y**

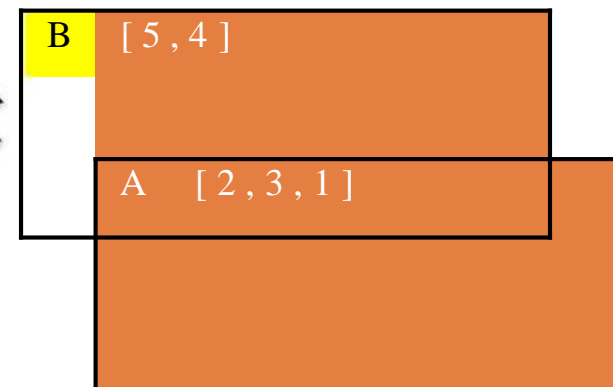


GROUPBY

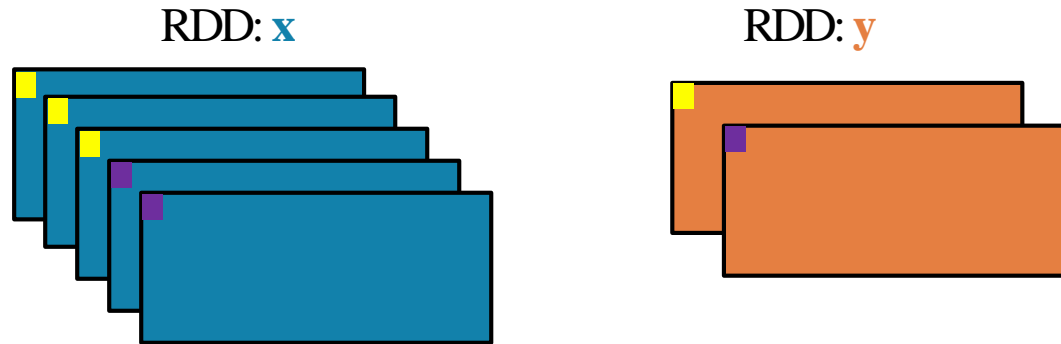
Pair RDD: **x**



RDD: **y**




GROUPBY



`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)]`

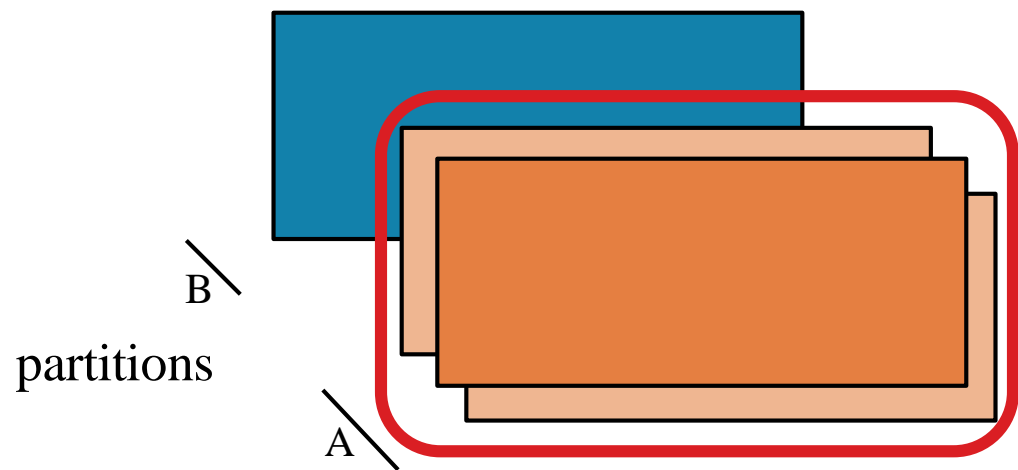
y: `[('A', [2, 3, 1]), ('B', [5, 4])]`



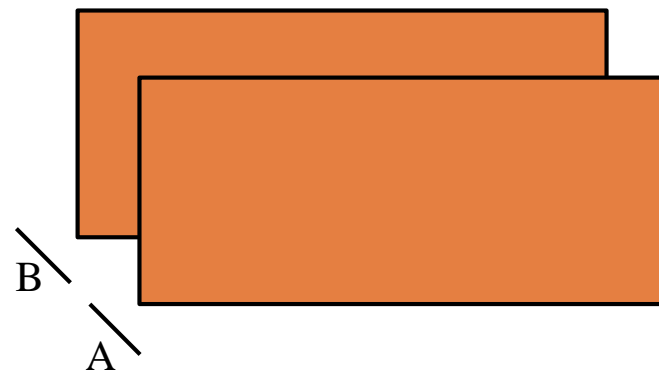
```
val x = sc.parallelize(
  Array(('B',5),('B',4),('A',3),('A',2),('A',1)))
val y = x.groupByKey()
println(x.collect().mkString(", "))
println(y.collect().mkString(", "))
```

REDUCEBYKEY

RDD: **x**



RDD: **y**



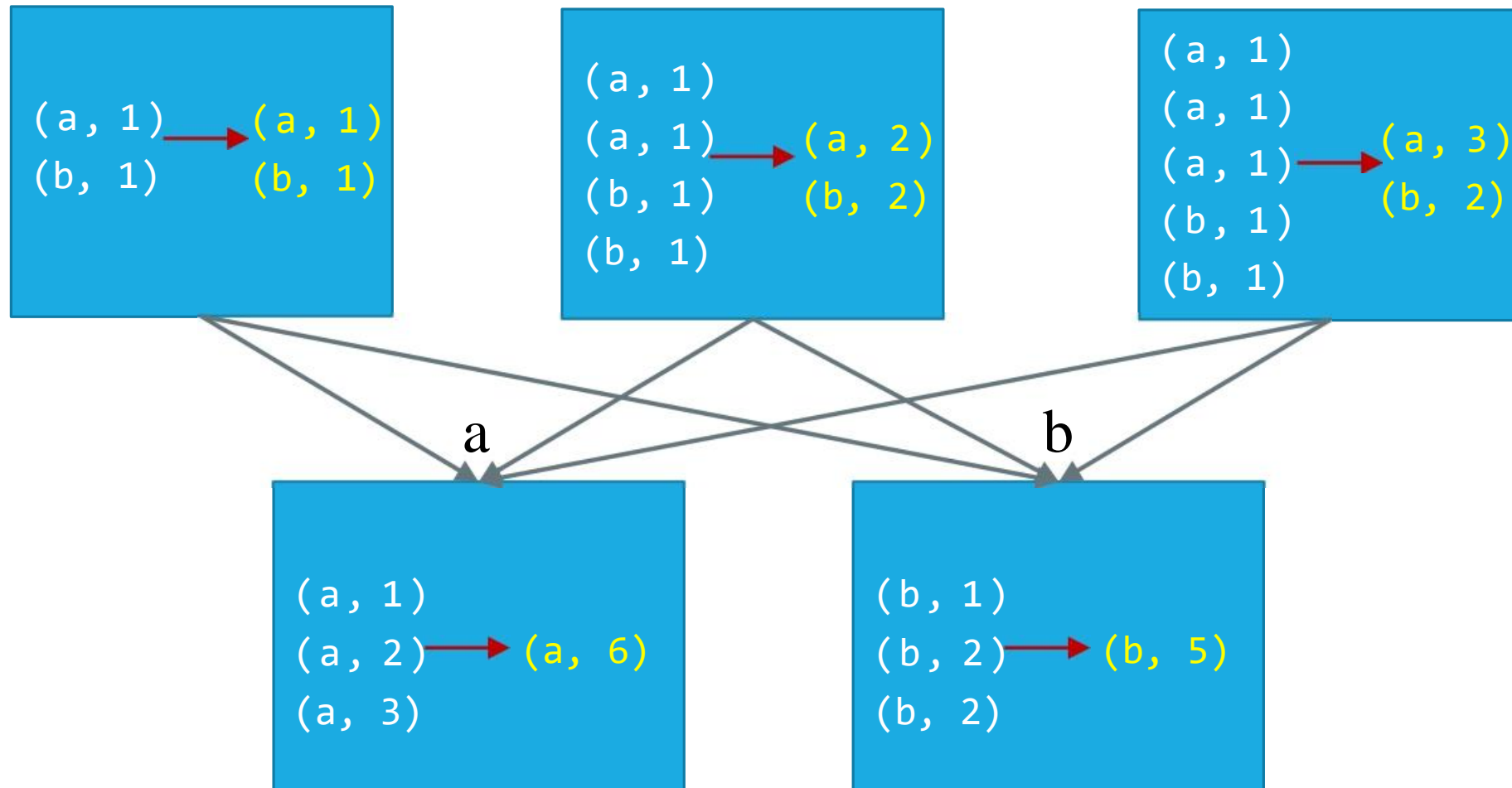
REDUCEBYKEY vs GROUPBYKEY

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

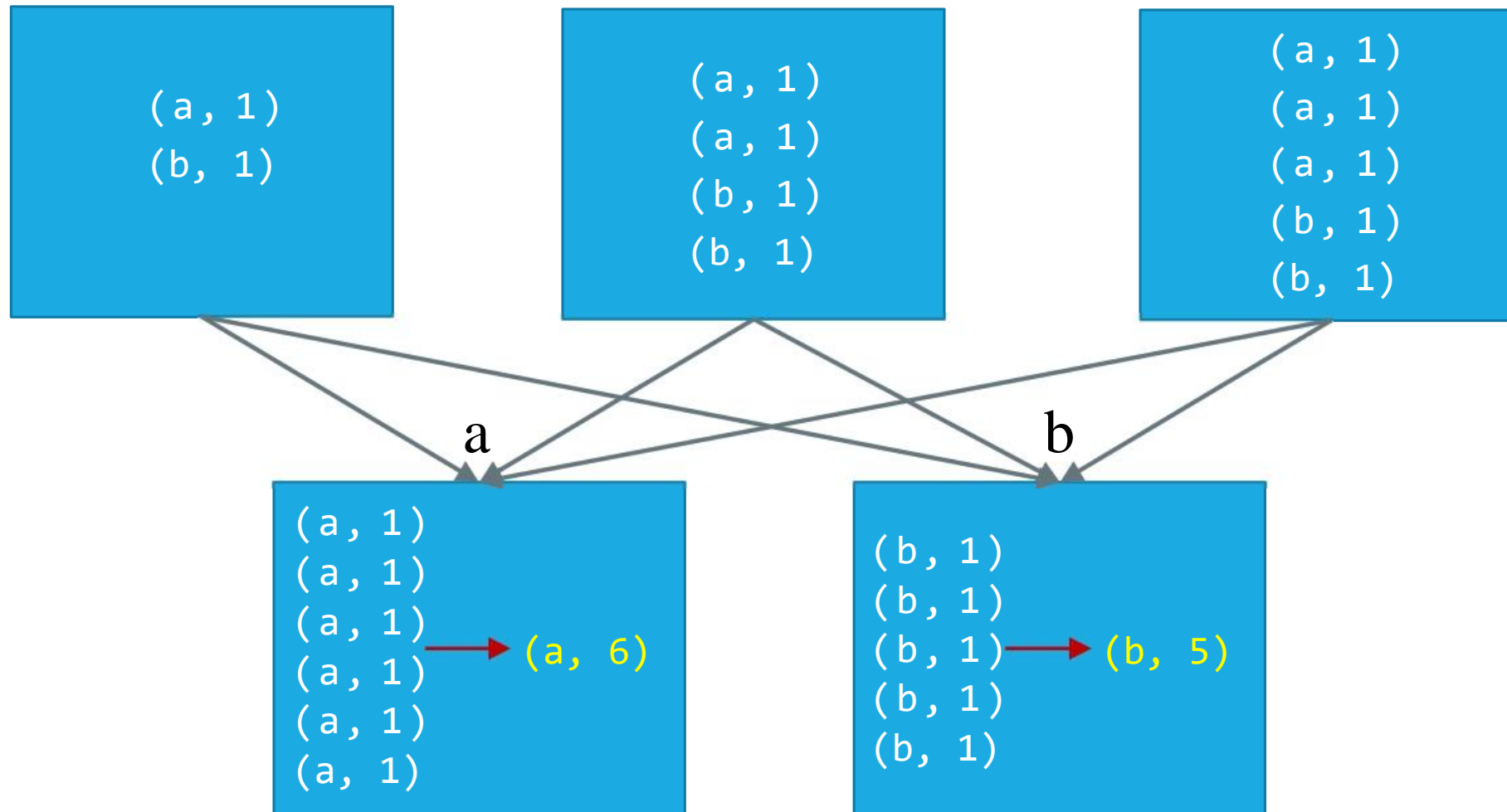
```
val wordCountsWithReduce = wordPairsRDD  
  .reduceByKey(_ + _)  
  .collect()
```

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

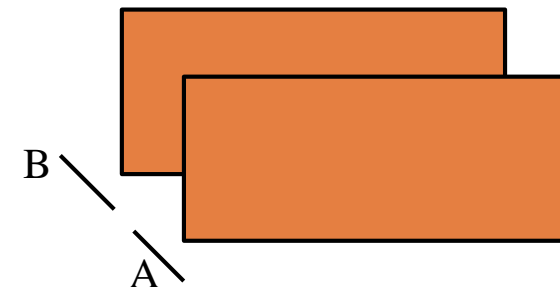
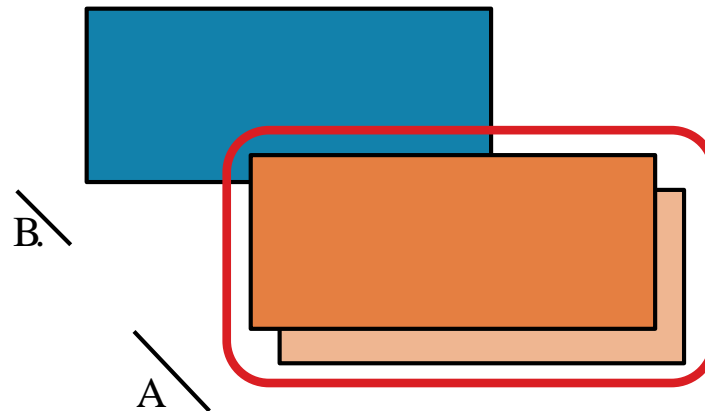
REDUCEBYKEY



GROUPBYKEY



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)
```

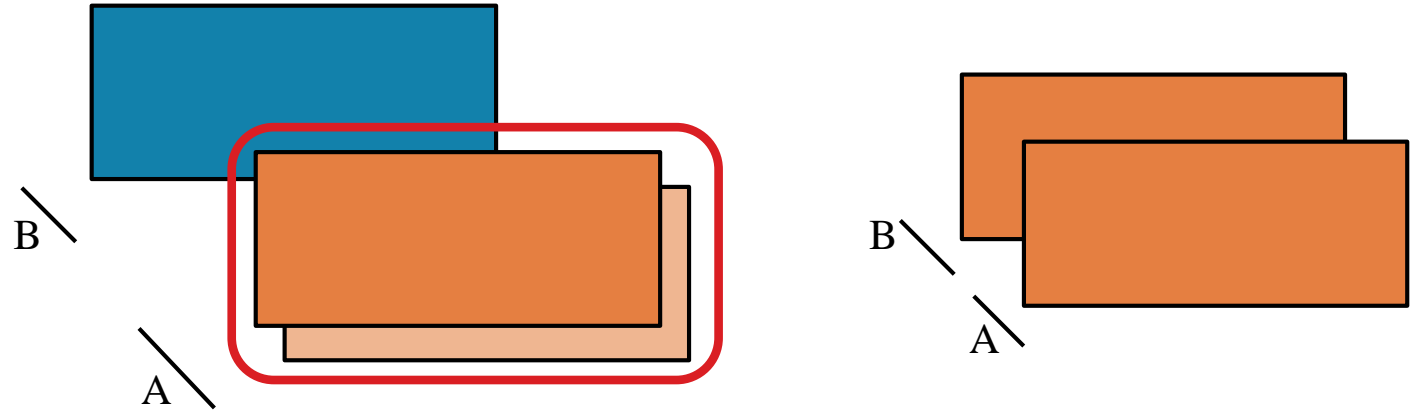
```
# glom() flattens elements on the same partition  
print(x.glom().collect())  
print(y.glom().collect())
```

```
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()
```

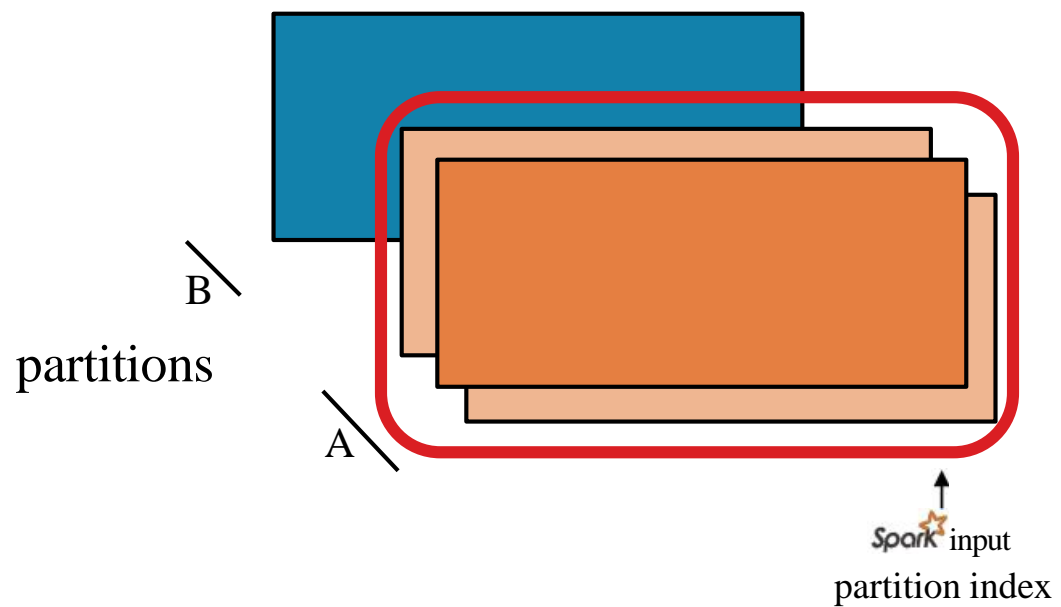
```
val yOut = y.glom().collect()
```

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

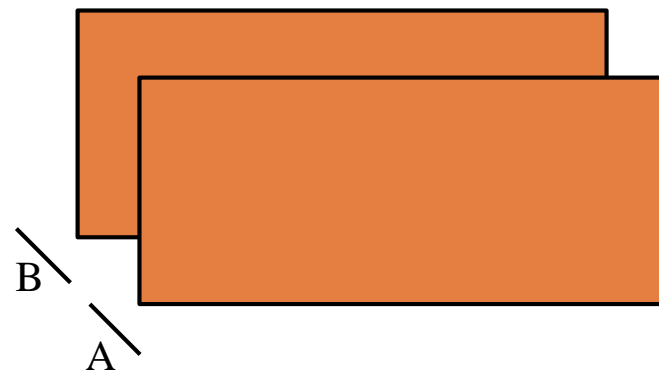
```
y: Array(Array(1, 42), Array(5, 42))
```

MAP PARTITIONS WITH INDEX

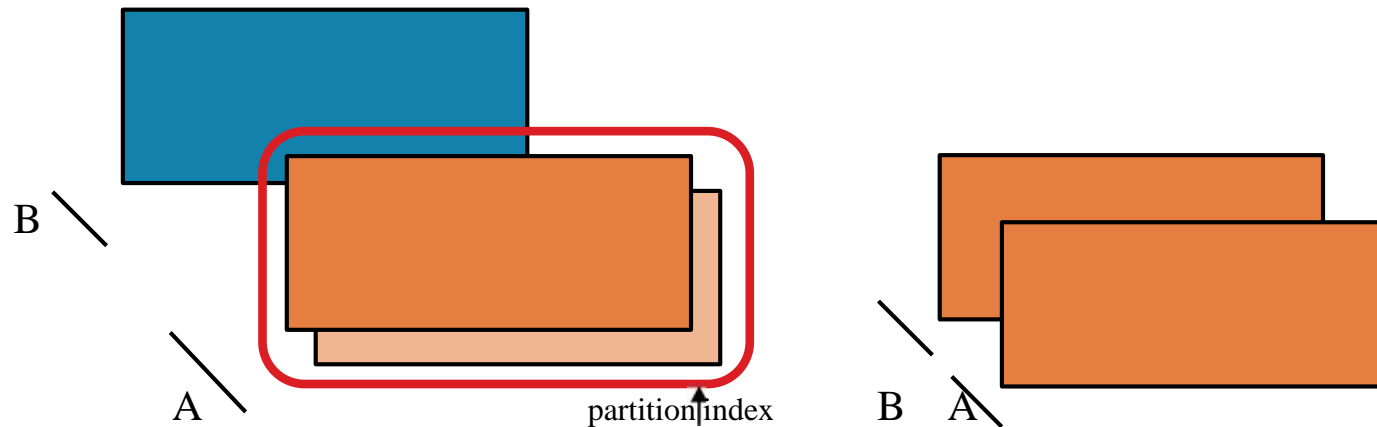
RDD: **x**



RDD: **y**



MAPPARTITIONS WITH INDEX



`mapPartitionsWithIndex(f, preservesPartitioning=False)`

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


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

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

```
y = x.mapPartitionsWithIndex(f)
```

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

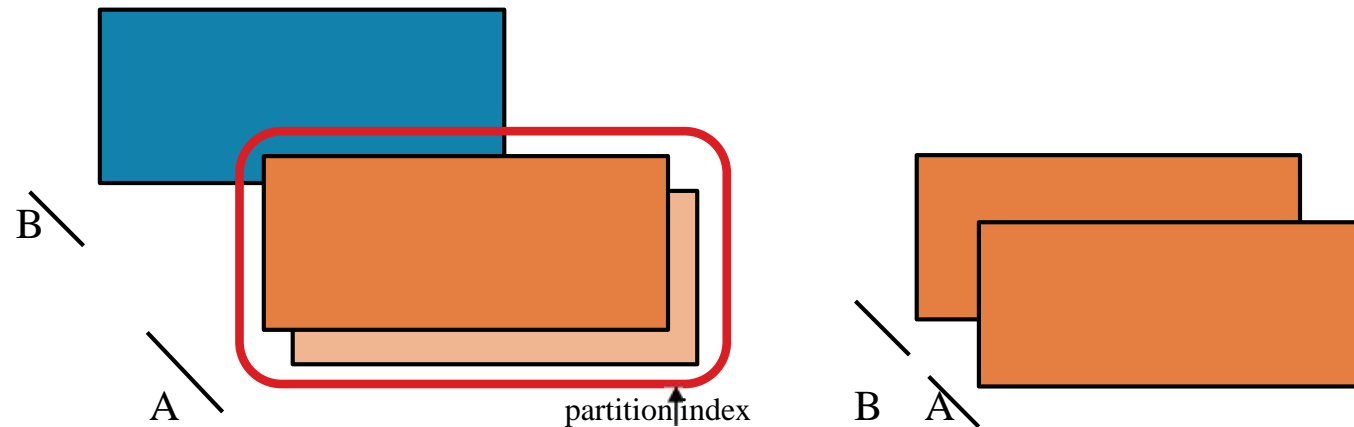


B A 

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

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

MAPPARTITIONSWITHINDEX



`mapPartitionsWithIndex(f, preservesPartitioning=False)`

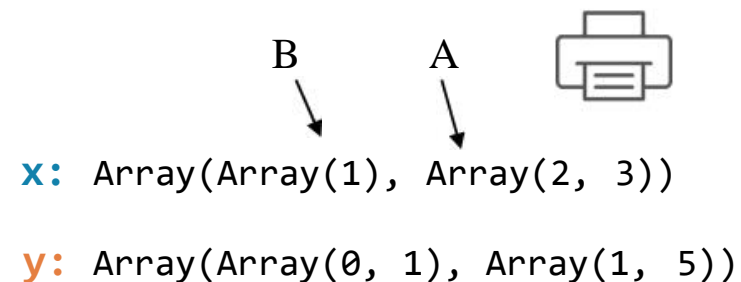
Return a new RDD by applying a function to each partition of this RDD, while tracking the index of the original partition.

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

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

val y = x.mapPartitionsWithIndex(f)

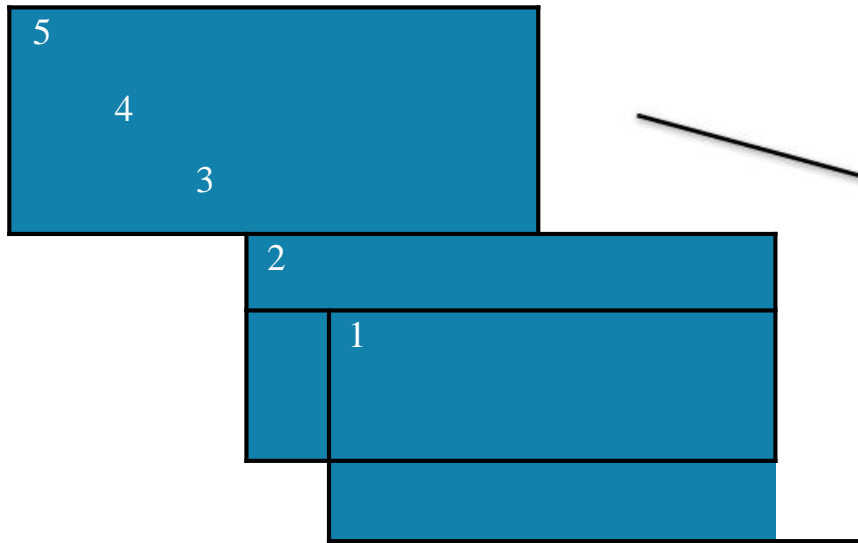
// glom() flattens elements on the same partition
val xOut = x.glom().collect()
val yOut = y.glom().collect()
```



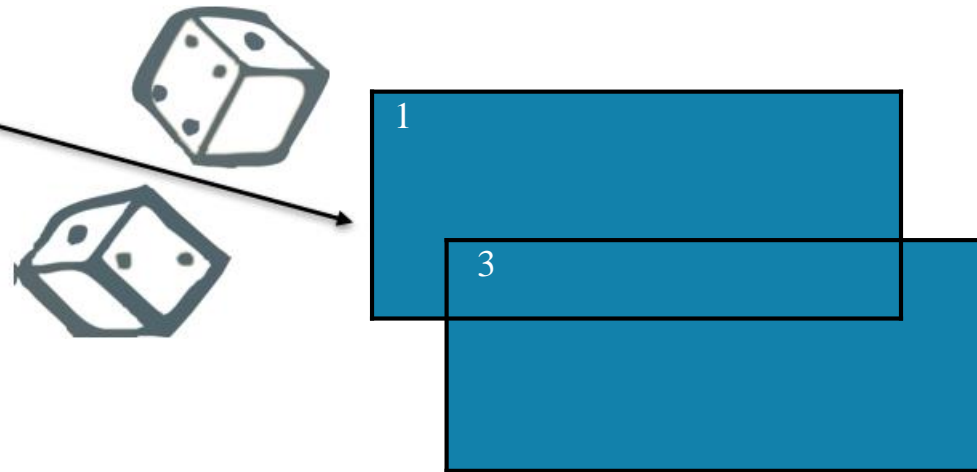
`x: Array(Array(1), Array(2, 3))`
`y: Array(Array(0, 1), Array(1, 5))`

SAMPLE

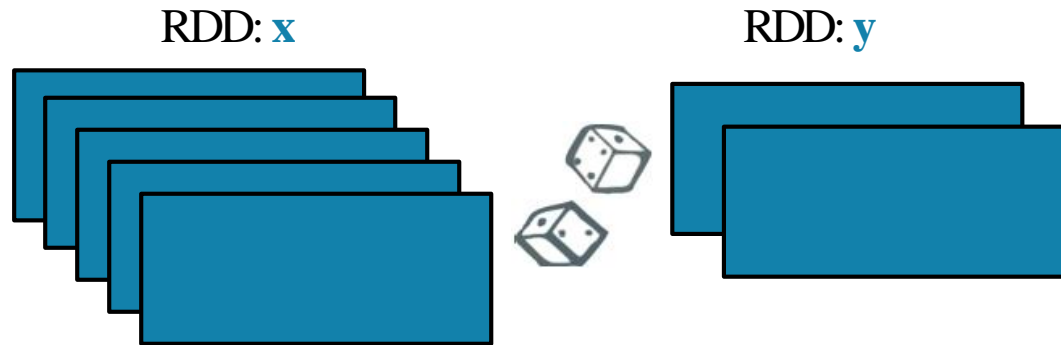
RDD: **x**



RDD: **y**



SAMPLE



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

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



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



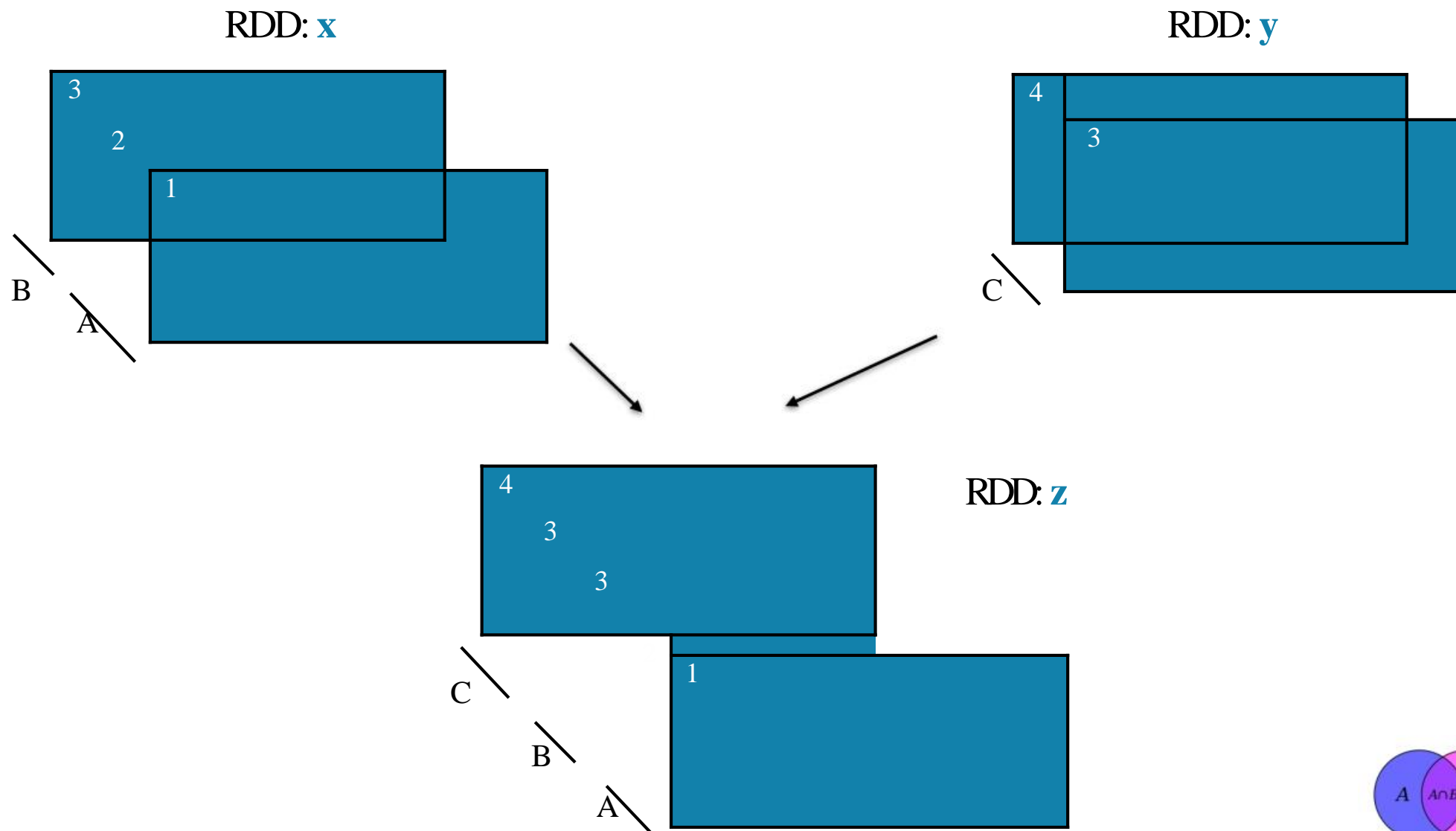
```
val x = sc.parallelize(Array(1, 2, 3, 4, 5))
val y = x.sample(false, 0.4)
```

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

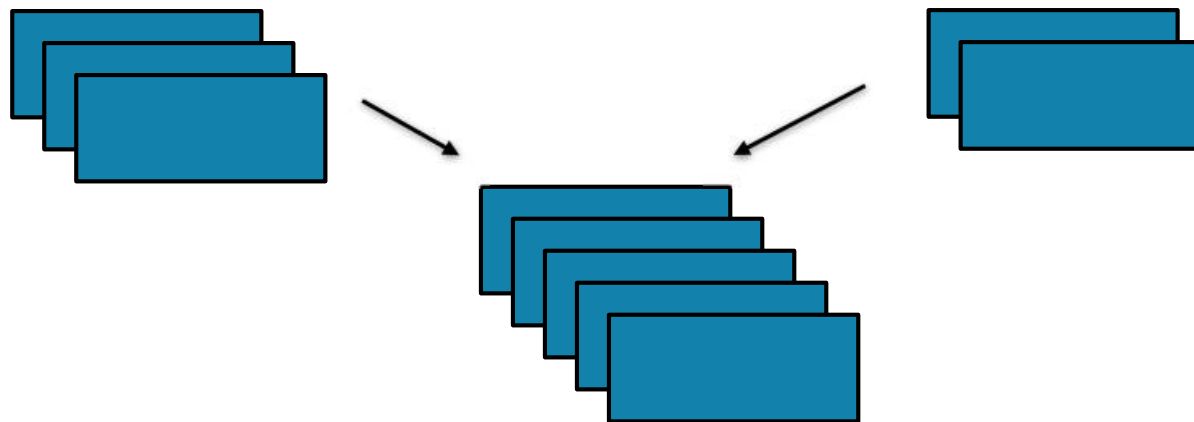
`x:` [1, 2, 3, 4, 5]

`y:` [1, 3]

UNION



UNION



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

`union(otherRDD)`



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



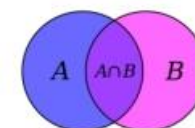
`x:` [1, 2, 3]

`y:` [3, 4]

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

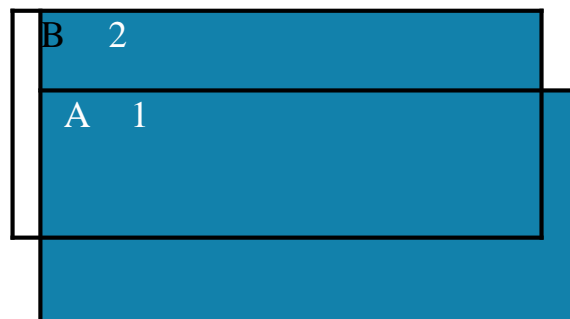


```
val x = sc.parallelize(Array(1,2,3), 2)
val y = sc.parallelize(Array(3,4), 1)
val z = x.union(y)
val zOut = z.glom().collect()
```

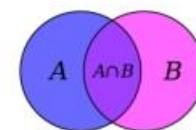
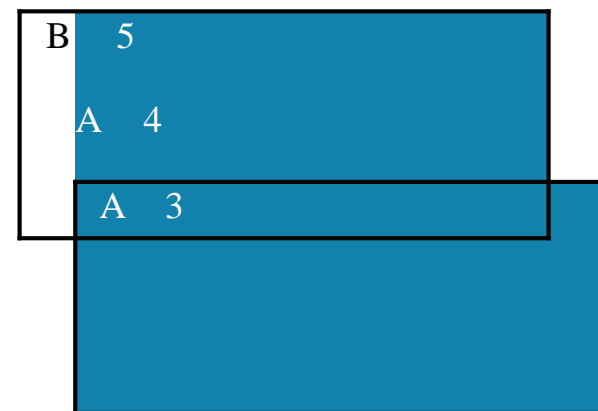


JOIN

RDD: **x**

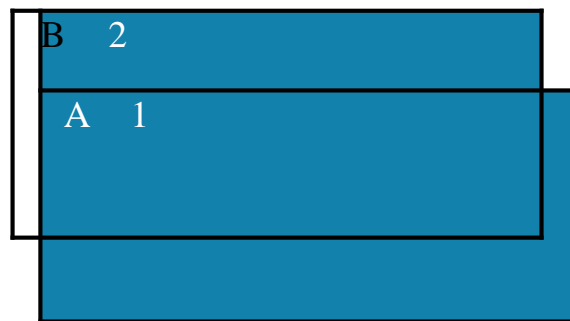


RDD: **y**

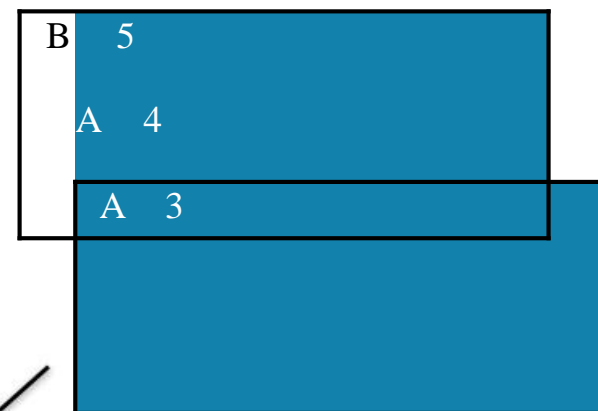


UNION

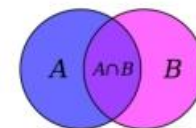
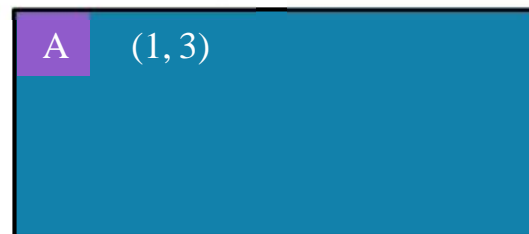
RDD: **x**



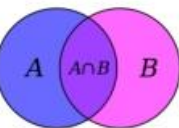
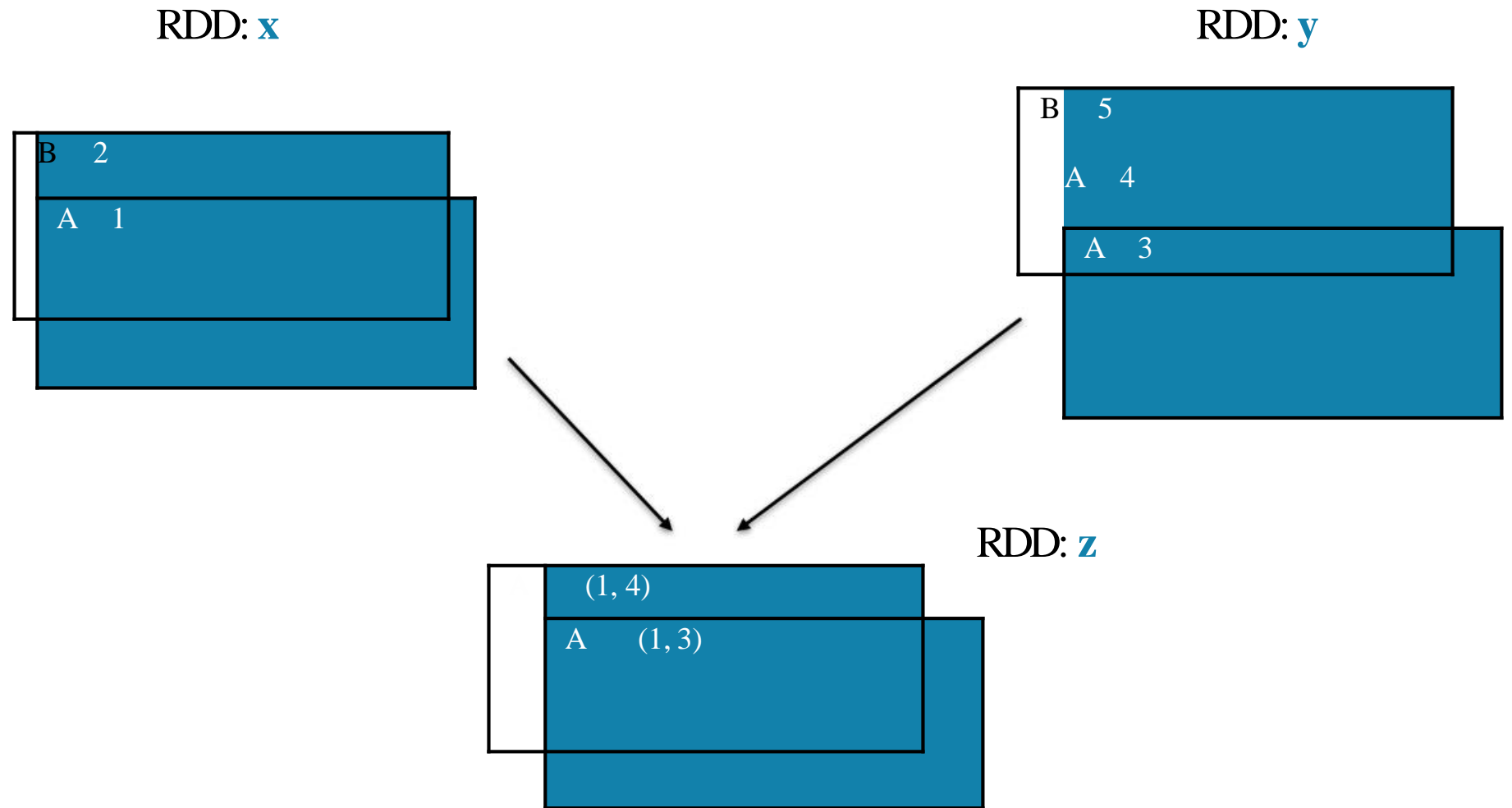
RDD: **y**



RDD: **z**



JOIN



Joins

```
rdd.join(otherRDD, [numTasks]) // inner join
```

- Requires two PairedRDDs
- When called on datasets of type (K, V) and (K, W) , returns a dataset of $(K, (V, W))$ pairs with all pairs of elements for each key.
- You can create compound keys to join over more variables

Joins (Cont.) (Scala)

```
case class Register (date: String, uuid: String, cust_id: String, lat: Float,
lng: Float)
case class Click (date: String, uuid: String, landing_page: Int)

val reg = sc.textFile("/user/cloudera/spark-workshop-data/join-
example/reg.tsv")
    .map(_._split("\t"))
    .map(r => (r(1), Register(r(0), r(1), r(2), r(3).toFloat,
r(4).toFloat)))

val clk = sc.textFile("/user/cloudera/spark-workshop-data/join-
example/clk.tsv")
    .map(_._split("\t"))
    .map(c => (c(1), Click(c(0), c(1), c(2).trim.toInt)))

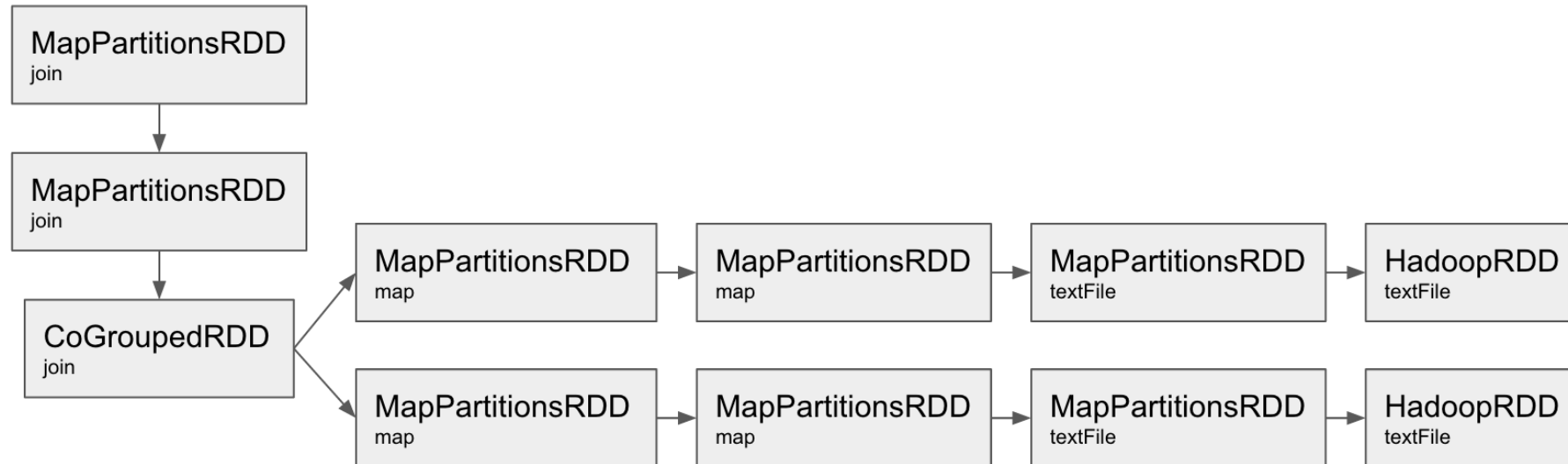
val joined = reg.join(clk)
joined.take(2)
//Returns: Array(
//
//      (81da510acc4111e387f3600308919594, (Register(...), Click(...))),
//
//      (15dfb8e6cc4111e3a5bb600308919594, (Register(...), Click(...)))
//
//)
```

Joins (Lineage Graph)

`joined.toDebugString`

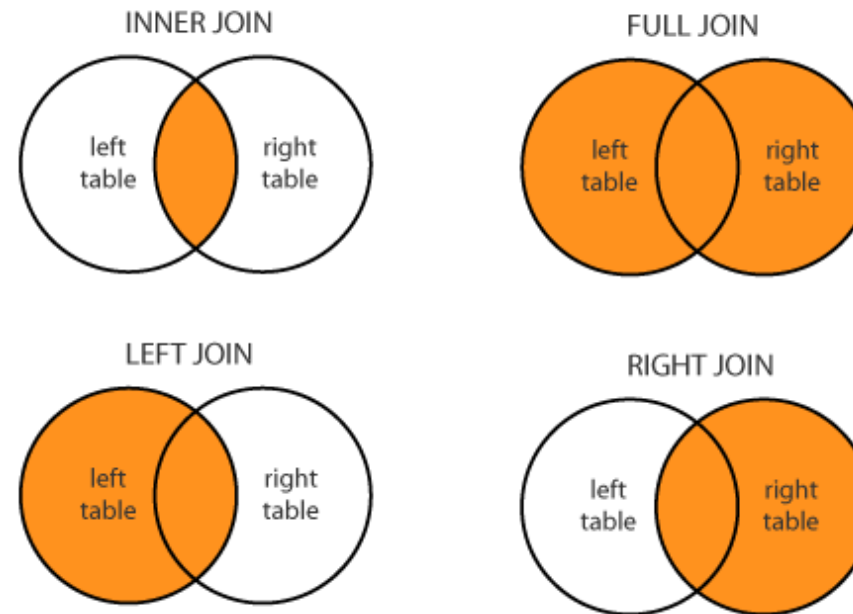
```
(2) MapPartitionsRDD[10] at join at <console>:32 []  
  | MapPartitionsRDD[9] at join at <console>:32 []  
  | CoGroupedRDD[8] at join at <console>:32 []  
+- (2) MapPartitionsRDD[3] at map at <console>:25 []  
  | | MapPartitionsRDD[2] at map at <console>:25 []  
  | | /user/cloudera/spark-workshop-data/join/reg.tsv MapPartitionsRDD[1] at textFile at <console>:25 []  
  | | /user/cloudera/spark-workshop-data/join/reg.tsv HadoopRDD[0] at textFile at <console>:25 []  
+- (2) MapPartitionsRDD[7] at map at <console>:25 []  
  | MapPartitionsRDD[6] at map at <console>:25 []  
  | /user/cloudera/spark-workshop-data/join/clk.tsv MapPartitionsRDD[5] at textFile at <console>:25 []  
  | /user/cloudera/spark-workshop-data/join/clk.tsv HadoopRDD[4] at textFile at <console>:25 []
```


Joins (Lineage Graph) (Cont.)



Join (Other Functions)

```
rdd.join(otherRDD, [numTasks]) // inner join  
rdd.leftOuterJoin(otherRDD, [numTasks])  
rdd.rightOuterJoin(otherRDD, [numTasks])  
rdd.fullOuterJoin(otherRDD, [numTasks])
```



Closures

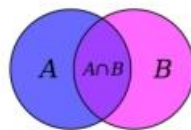
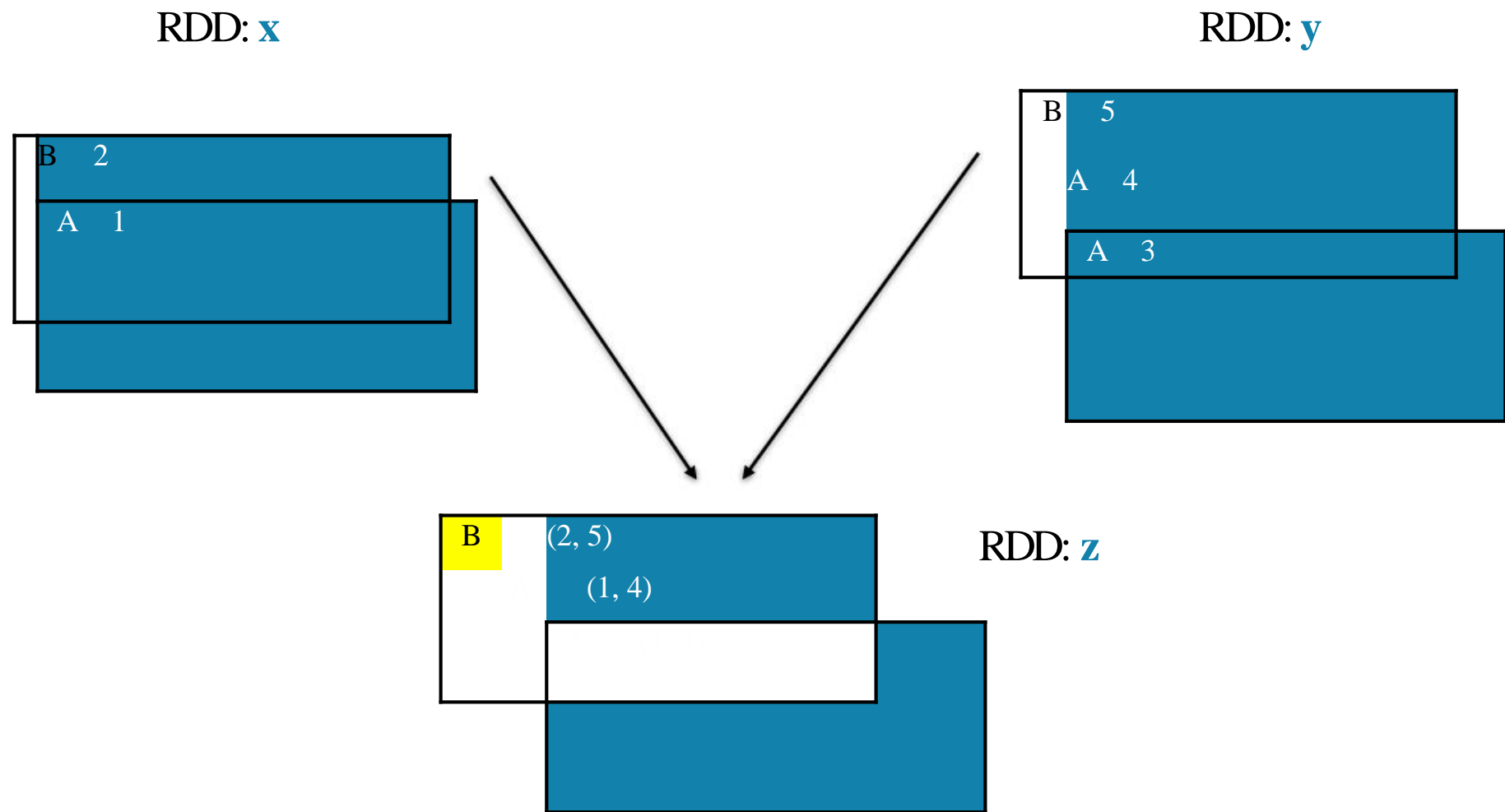
- One closure per worker
- Global variables can be used by workers:

```
var x = 5  
rdd.map( _ + x ) # Successfully add 5 to each element of an RDD
```

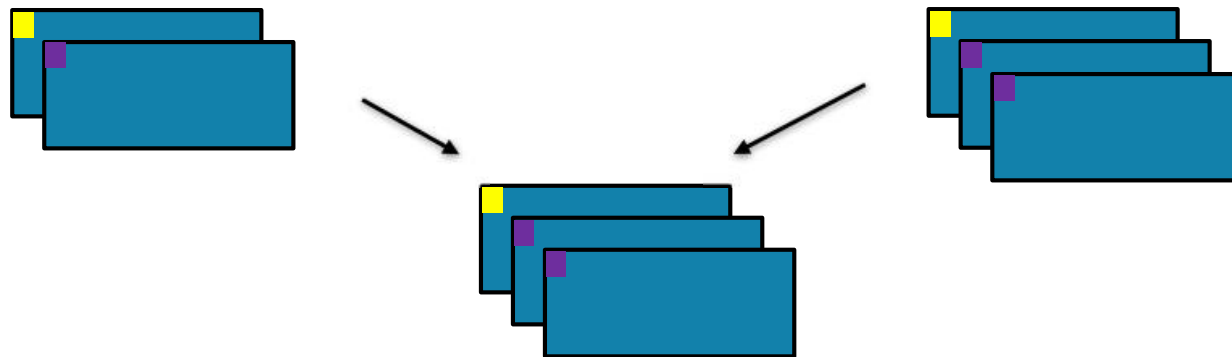
- If worker tries to update global variable, the changes won't be reflected:

```
var counter = 0  
var rdd = sc.parallelize(data)  
rdd.foreach( x => counter += x ) # It won't work!
```

JOIN



JOIN



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

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



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



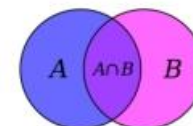
x: [("a", 1), ("b", 2)]

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

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

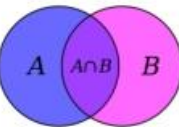
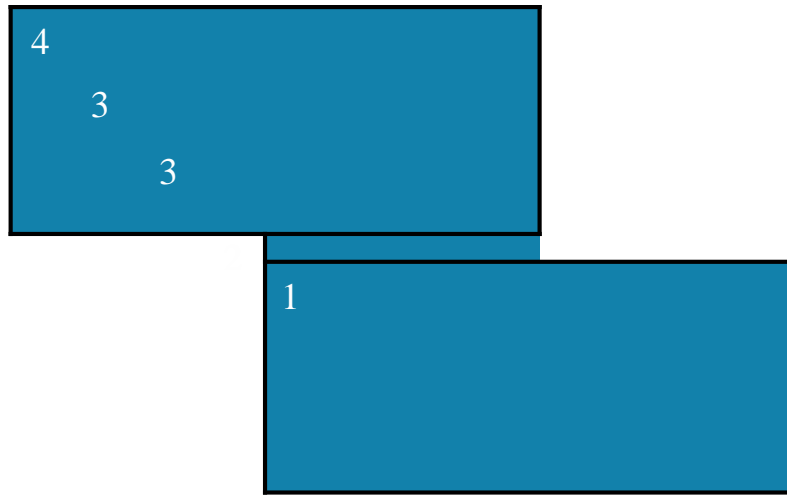


```
val x = sc.parallelize(Array(("a", 1), ("b", 2)))  
val y = sc.parallelize(Array(("a", 3), ("a", 4), ("b", 5)))  
val z = x.join(y) println(z.collect().mkString(", "))
```



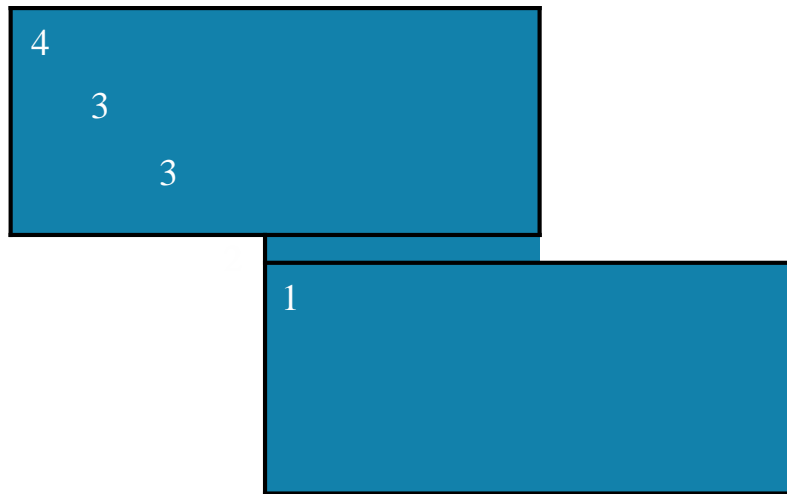
DISTINCT

RDD: **x**

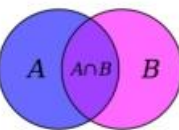
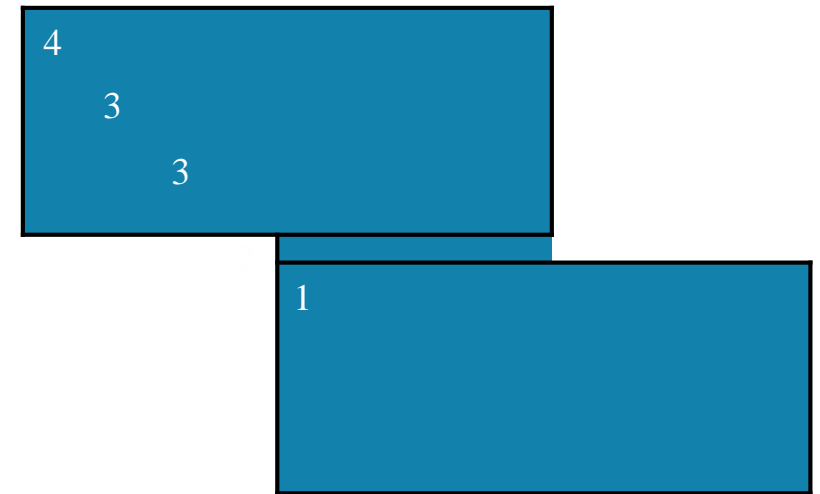


DISTINCT

RDD: **x**



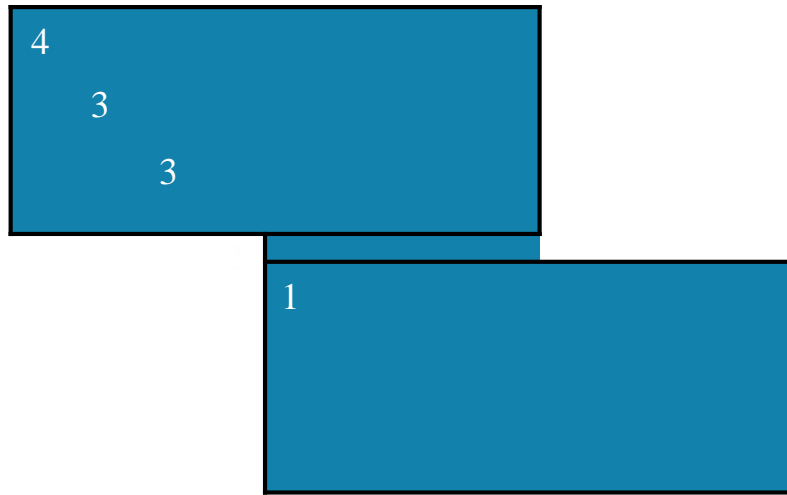
RDD: **y**



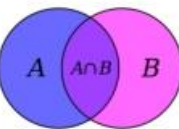
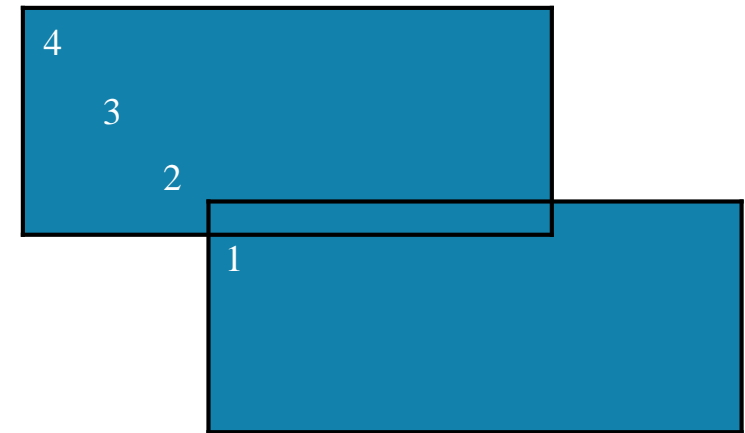


DISTINCT

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())
```

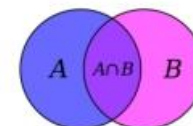


```
val x = sc.parallelize(Array(1,2,3,3,4))  
val y = x.distinct()  
  
println(y.collect().mkString(", "))
```



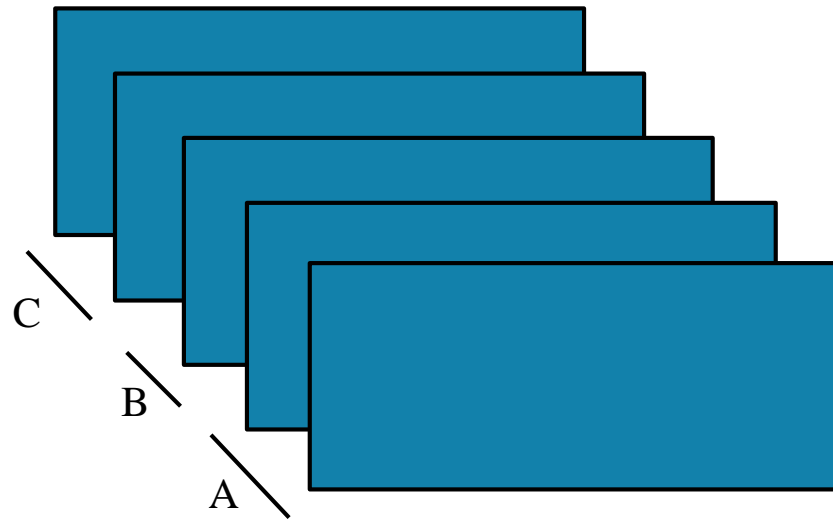
x: [1, 2, 3, 3, 4]

y: [1, 2, 3, 4]



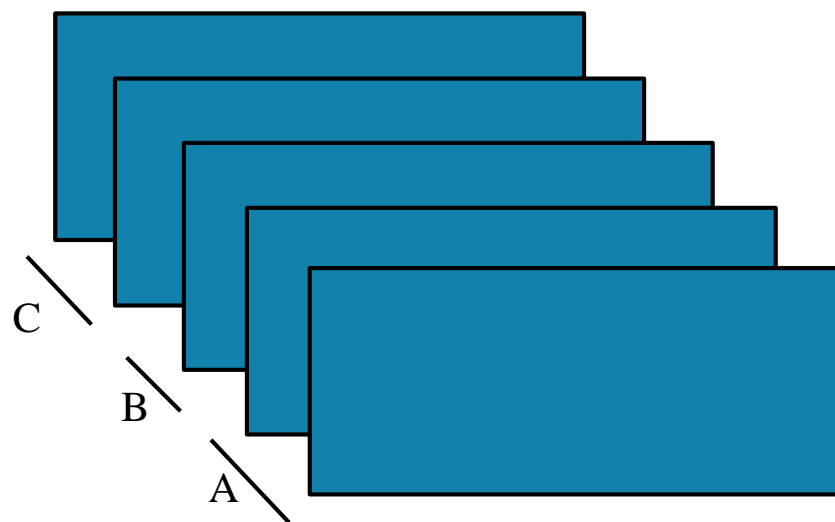
COALESCE

RDD: **x**

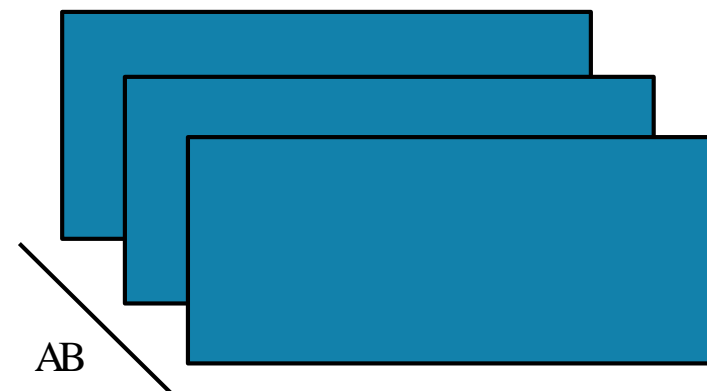


COALESCE

RDD: **x**

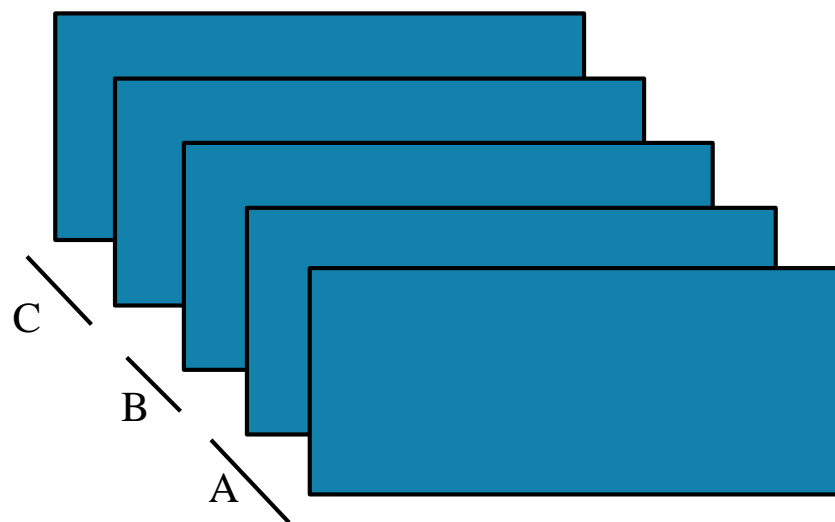


RDD: **y**

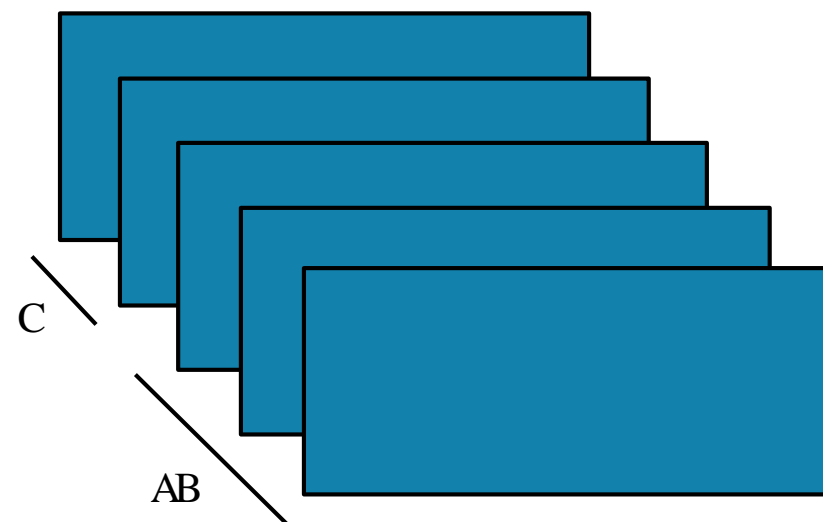


COALESCE

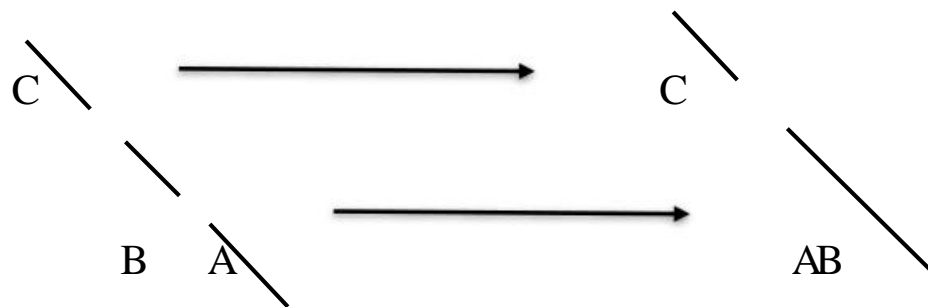
RDD: **x**



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]]

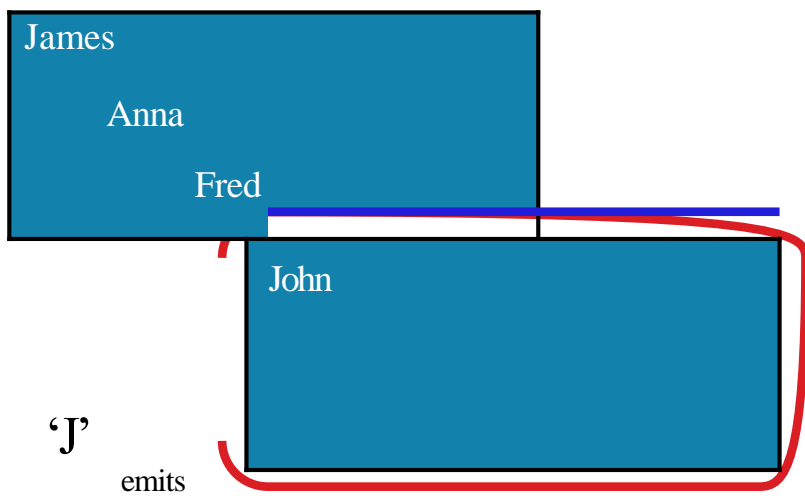
y: [[1], [2, 3, 4, 5]]



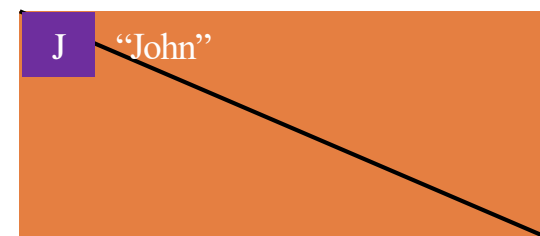
```
val x = sc.parallelize(Array(1, 2, 3, 4, 5), 3)
val y = x.coalesce(2)
val xOut = x.glom().collect()
val yOut = y.glom().collect()
```

KEYBY

RDD: **x**

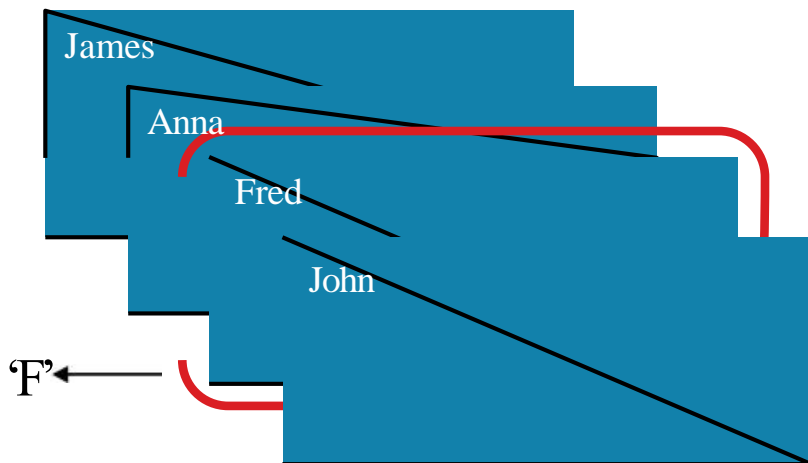


RDD: **y**

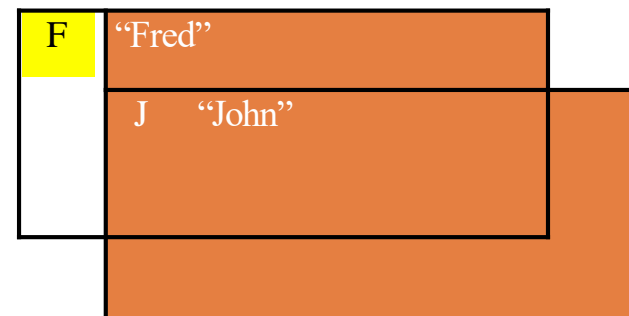


KEYBY

RDD: **x**



RDD: **y**



KEYBY

RDD: **x**

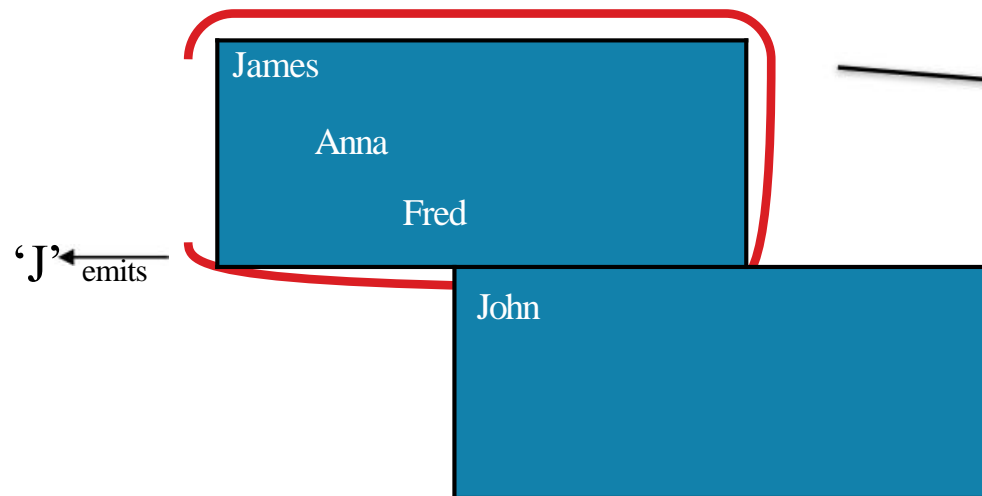


RDD: **y**

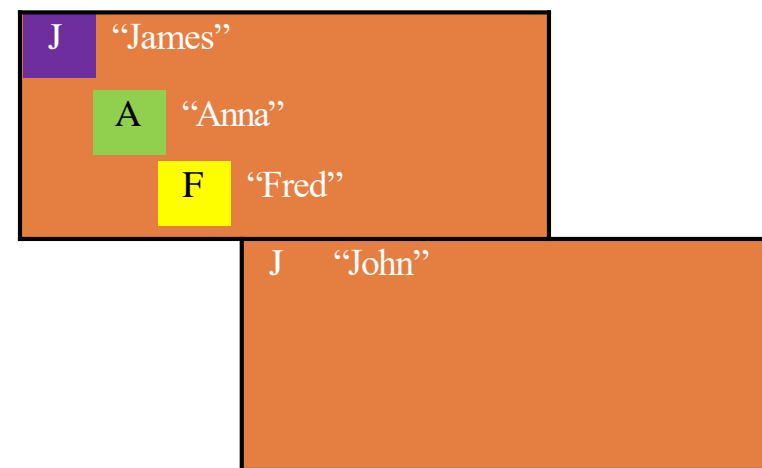


KEYBY

RDD: **x**

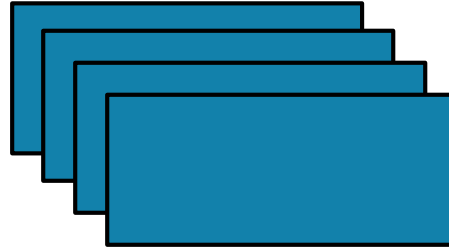


RDD: **y**

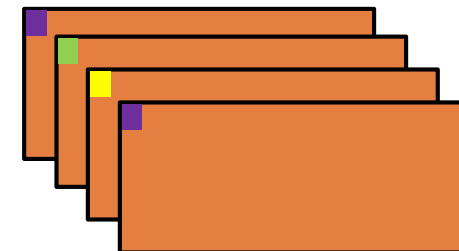


KEYBY

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()
```



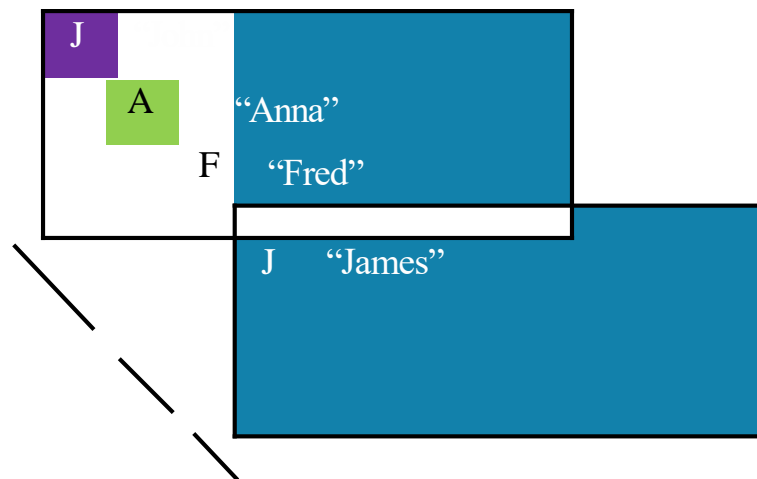
```
val x = sc.parallelize(
  Array("John", "Fred", "Anna", "James"))
val y = x.keyBy(w => w.charAt(0))
println(y.collect().mkString(", "))
```

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

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

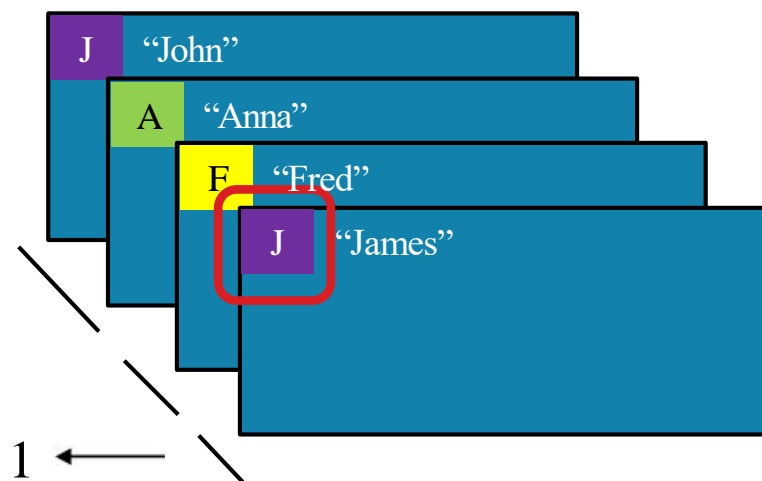
PARTITION BY

RDD: **x**

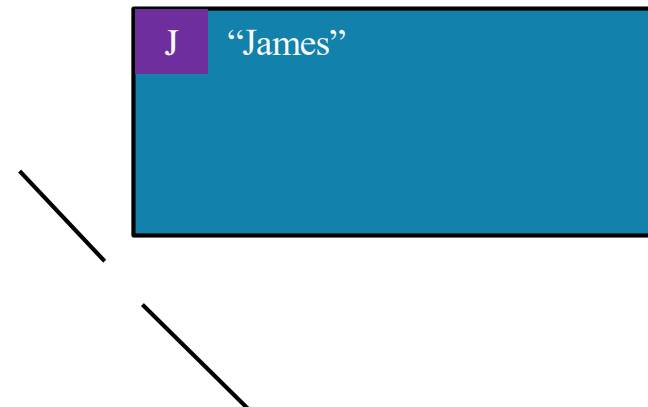


PARTITION BY

RDD: **x**

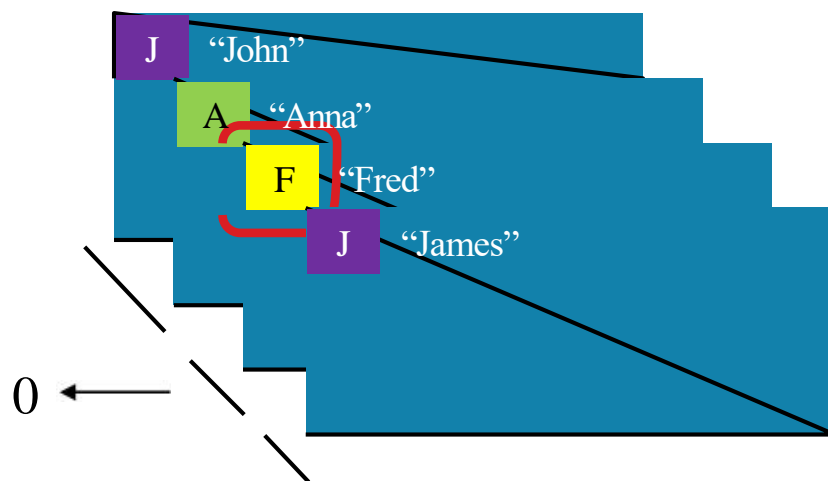


RDD: **y**

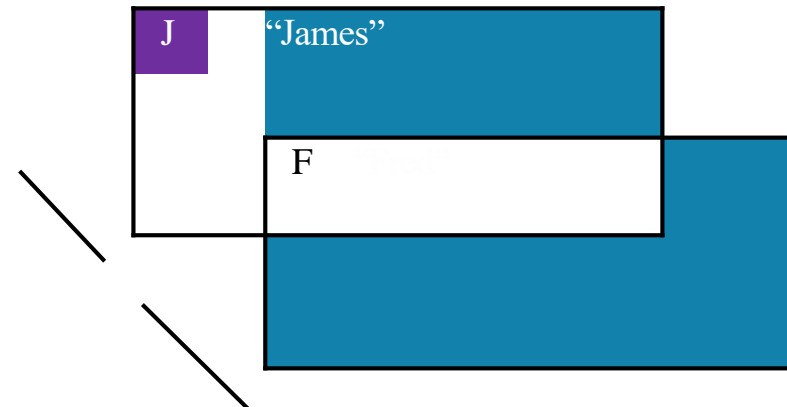


PARTITION BY

RDD: **x**

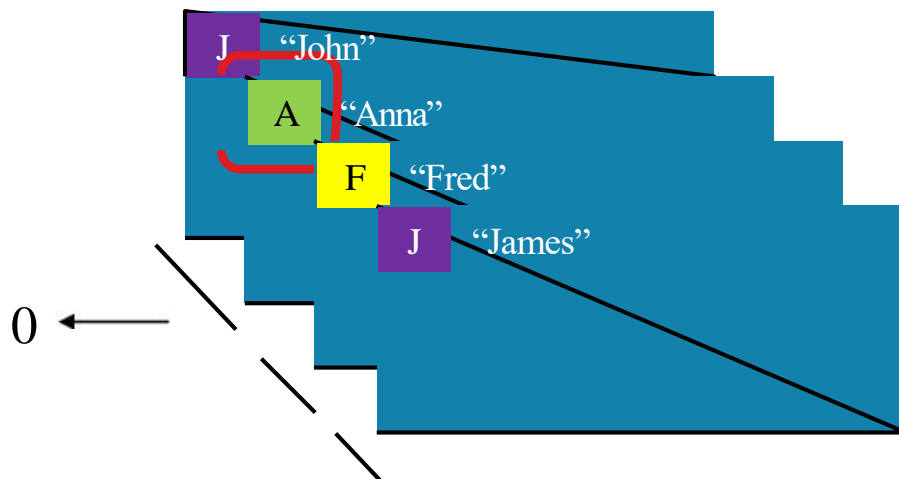


RDD: **y**

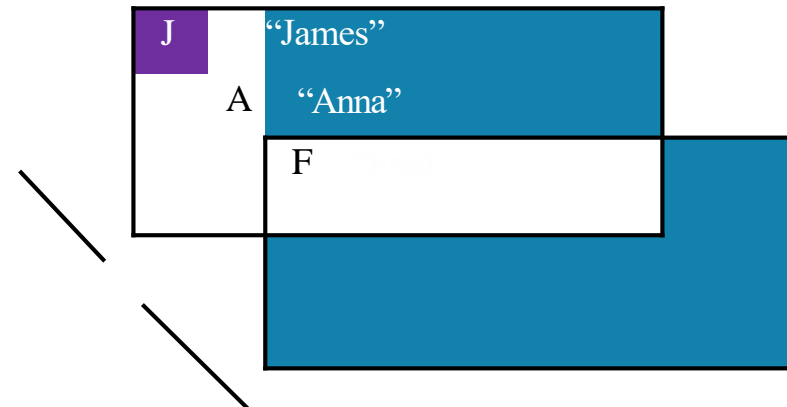


PARTITION BY

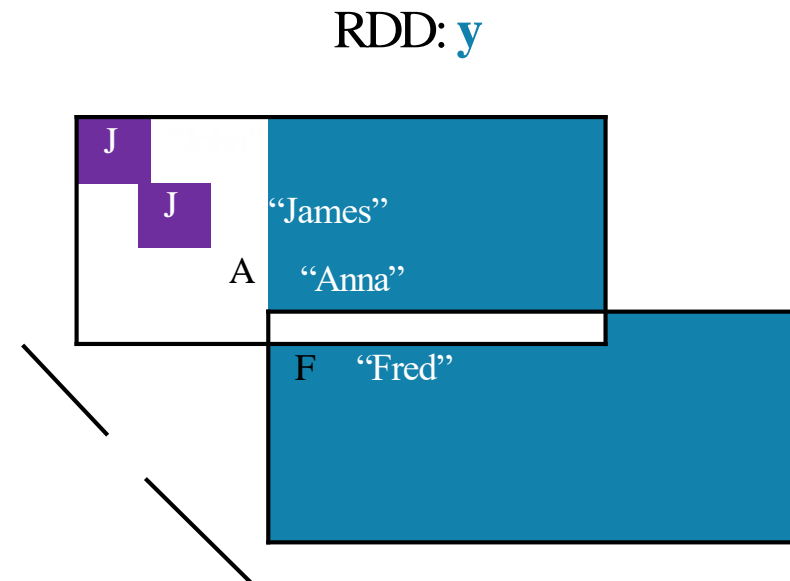
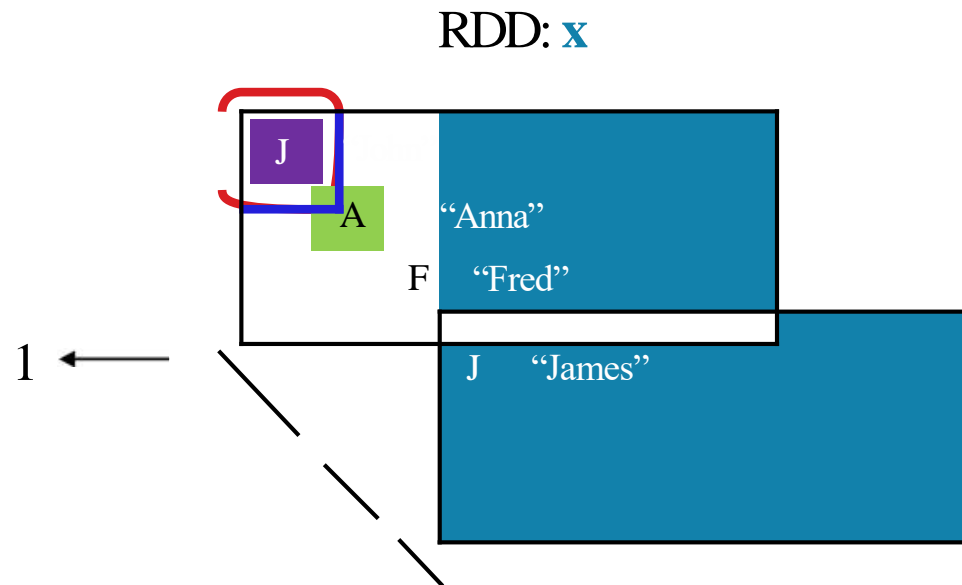
RDD: **x**



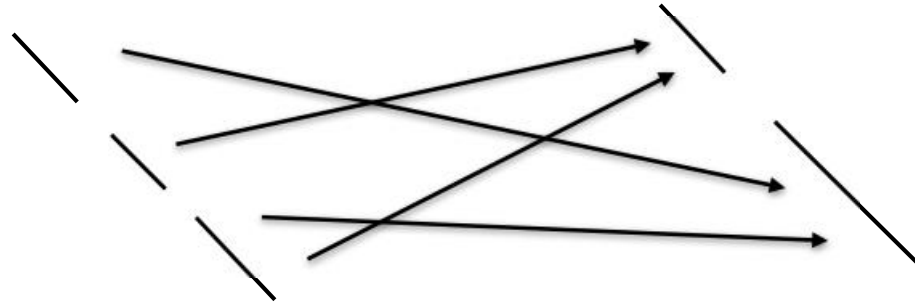
RDD: **y**



PARTITION BY



PARTITION BY



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)`



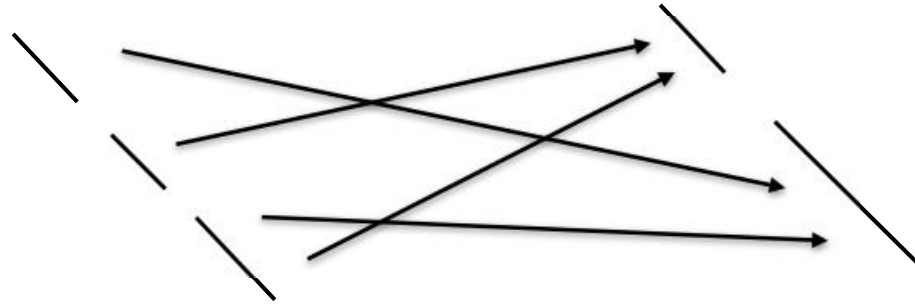
```
x = sc.parallelize([('J','James'),('F','Fred'),  
                  ('A','Anna'),('J','John')], 3)  
  
y = x.partitionBy(2, lambda w: 0 if w[0] < 'H' else 1)  
print x.glom().collect()  
print y.glom().collect()
```



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



PARTITION BY



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 y0ut = y.glom().collect()
```

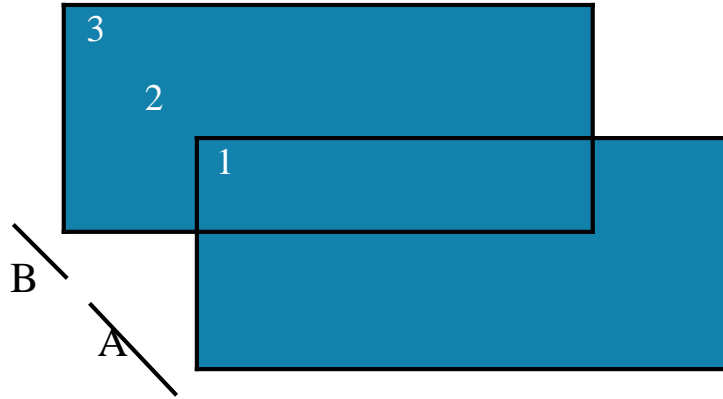


x: Array(Array((A,Anna), (F,Fred)),
 Array((J,John), (J,James)))

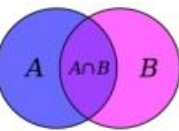
y: Array(Array((F,Fred), (A,Anna)),
 Array((J,John), (J,James)))

ZIP

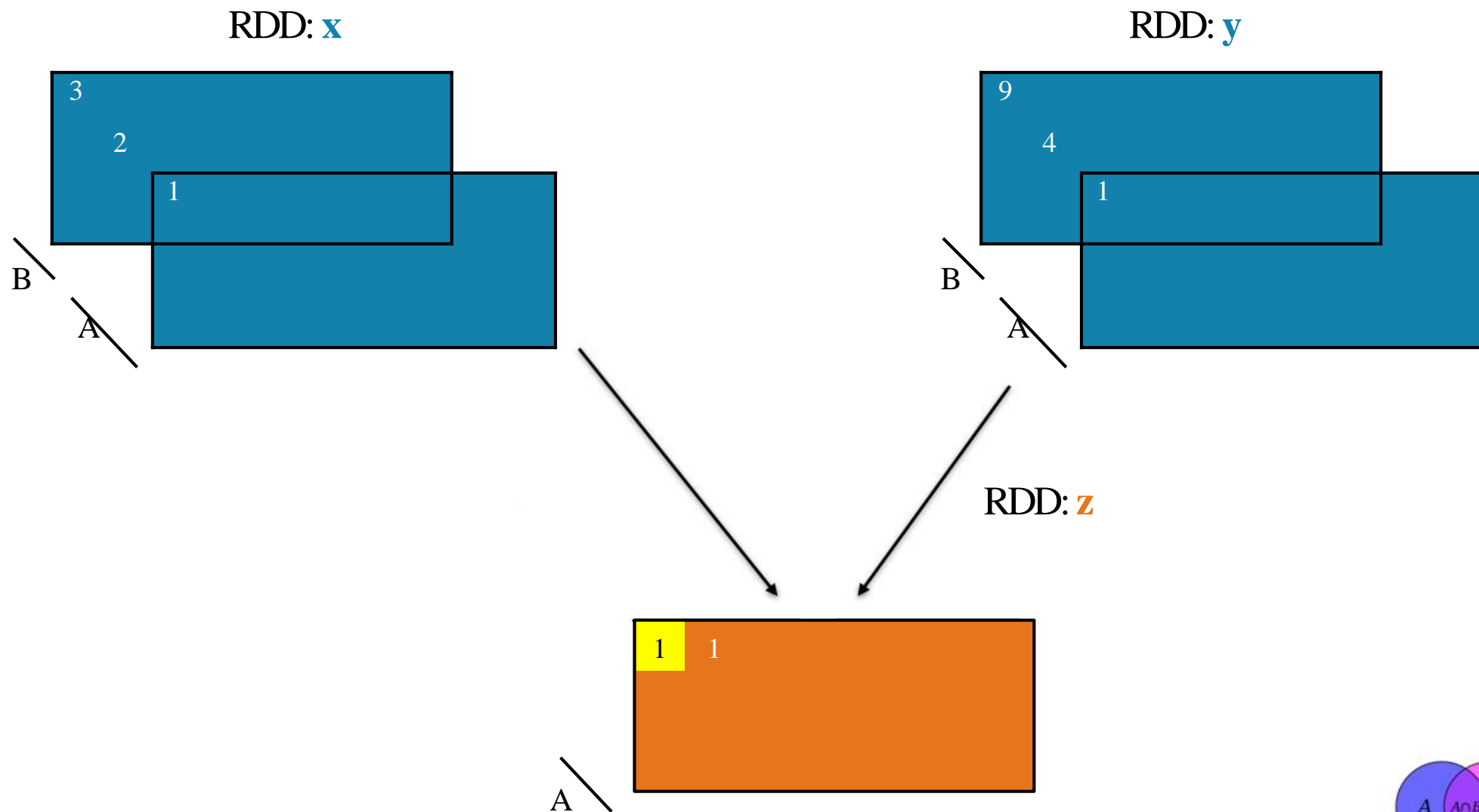
RDD: **x**



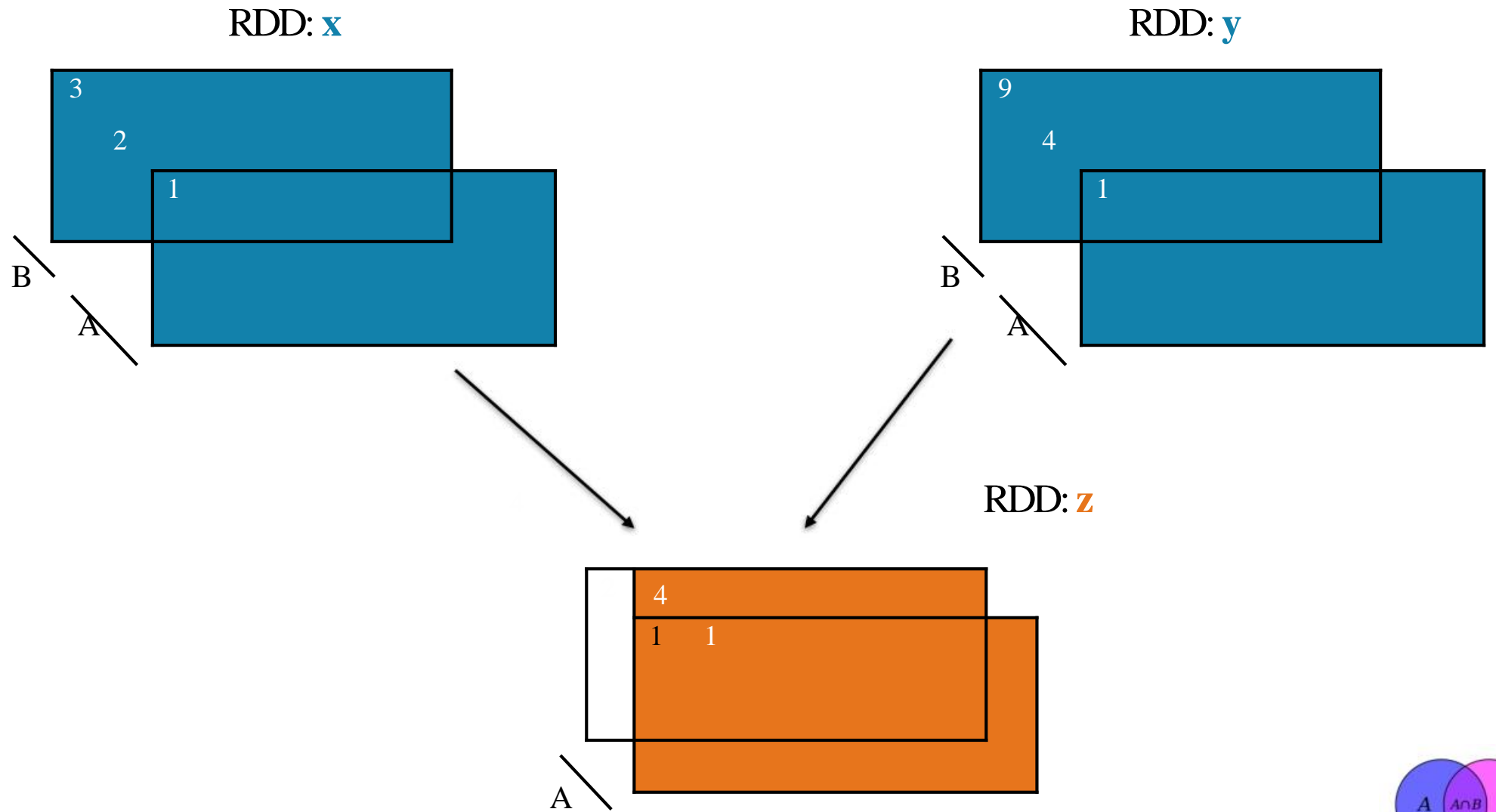
RDD: **y**



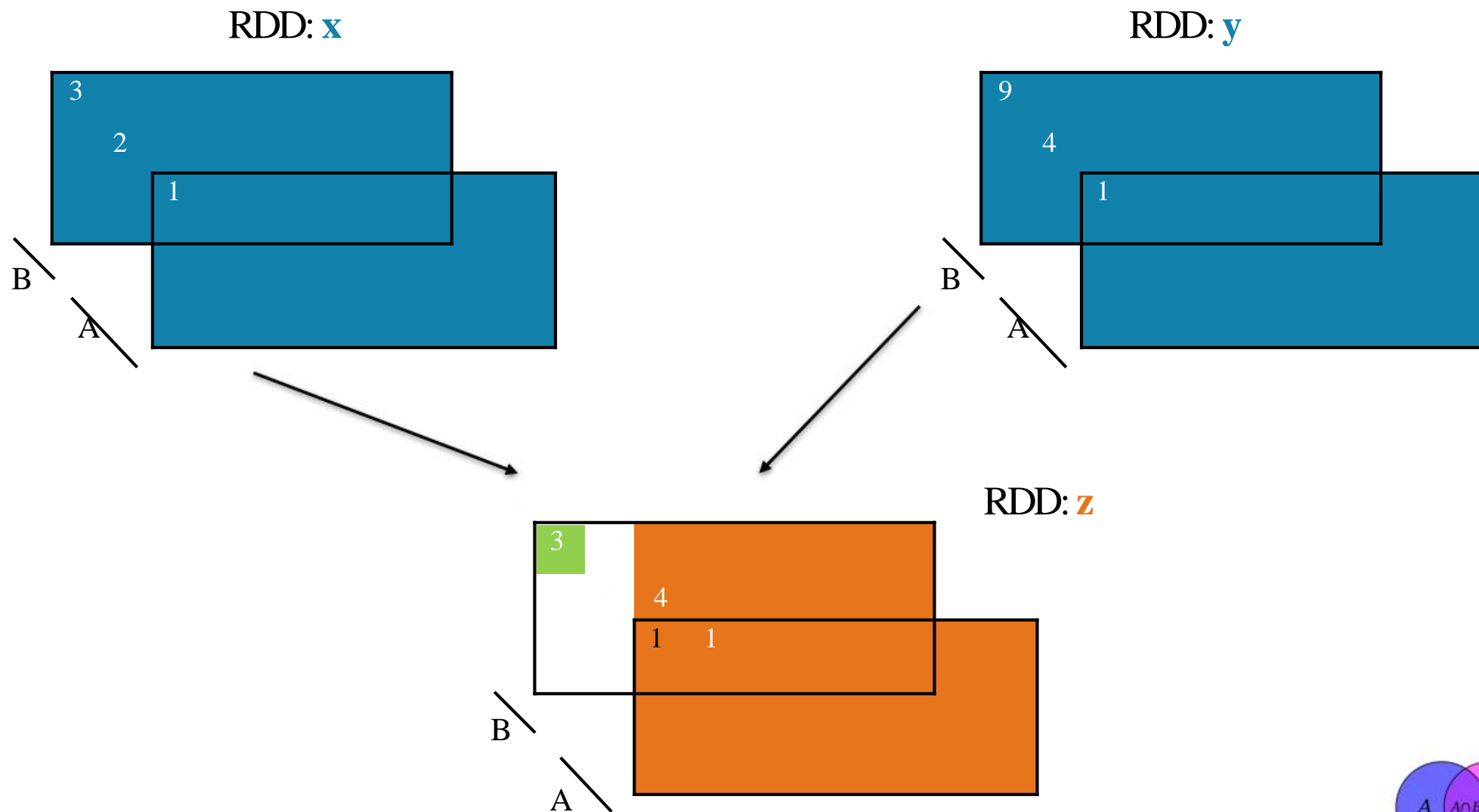
ZIP



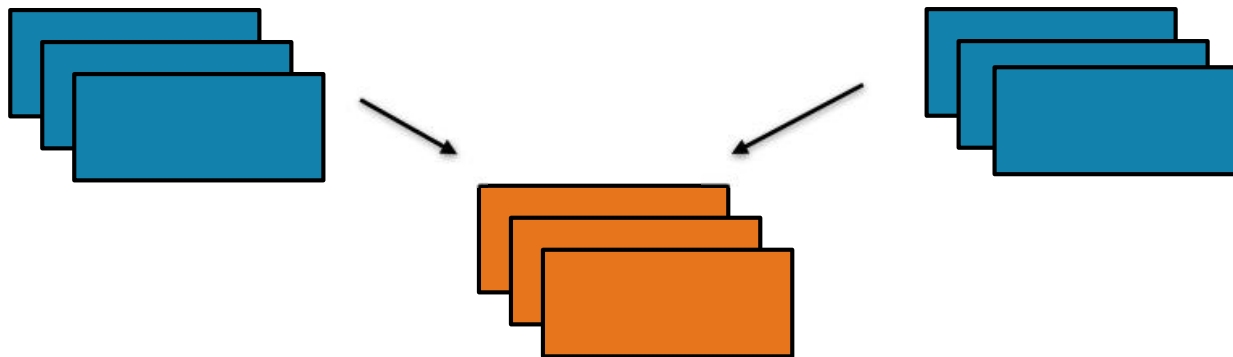
ZIP



ZIP



ZIP



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

`zip(otherRDD)`



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

```
print(z.collect())
```



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

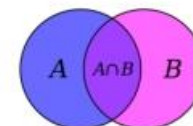
```
println(z.collect().mkString(", "))
```



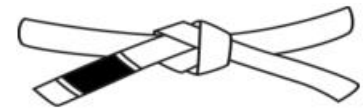
x: [1, 2, 3]

y: [1, 4, 9]

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



ACTIONS



Core Operations



distributed

occurs across the cluster

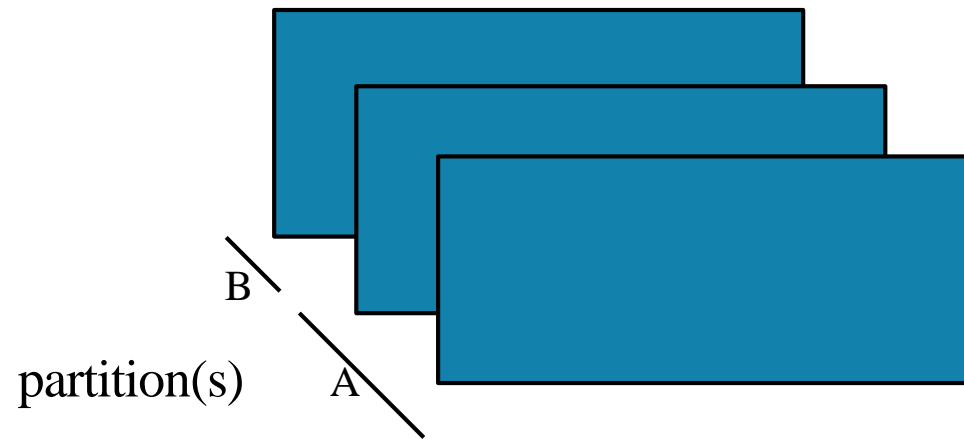
VS



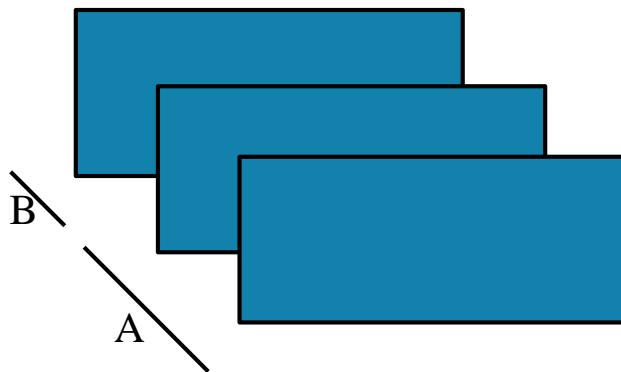
driver

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)
```



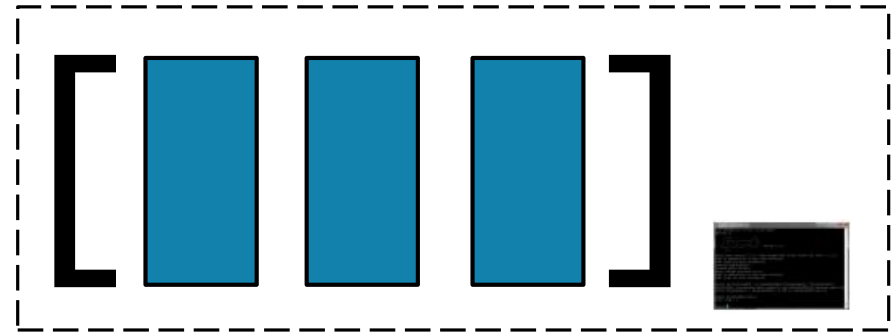
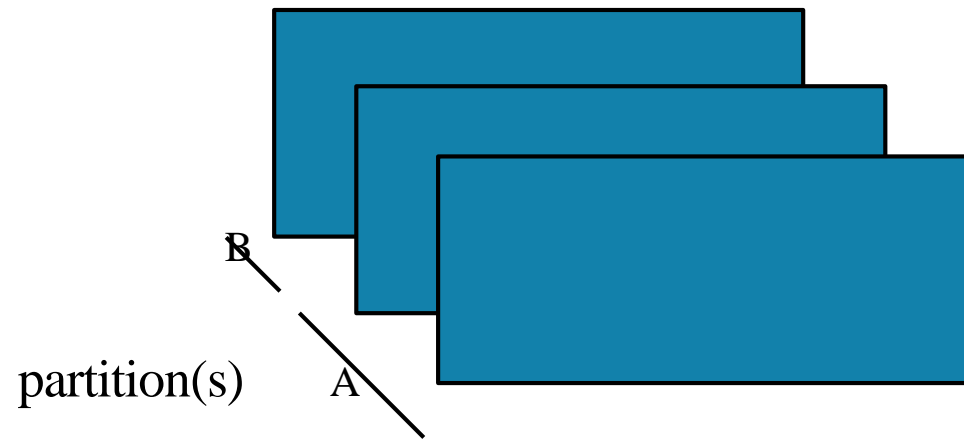
```
val x = sc.parallelize(Array(1,2,3), 2)
val y = x.partitions.size
val xOut = x.glom().collect()
println(y)
```



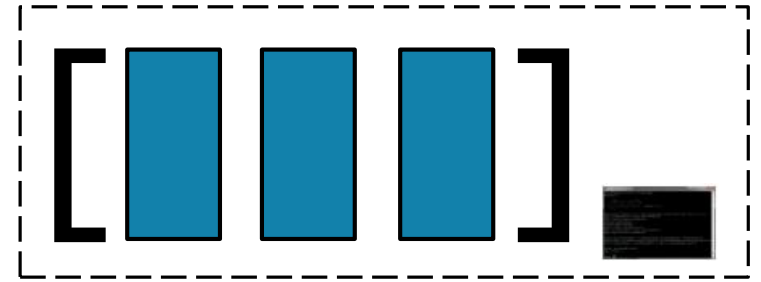
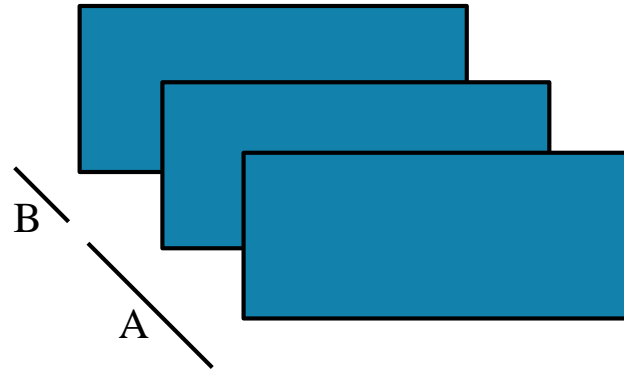
`x:` `[[1], [2, 3]]`

`y:` `2`

COLLECT



COLLECT



collect()

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



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

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

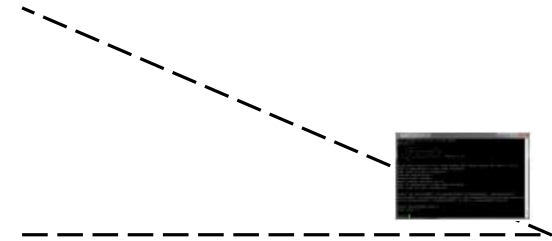
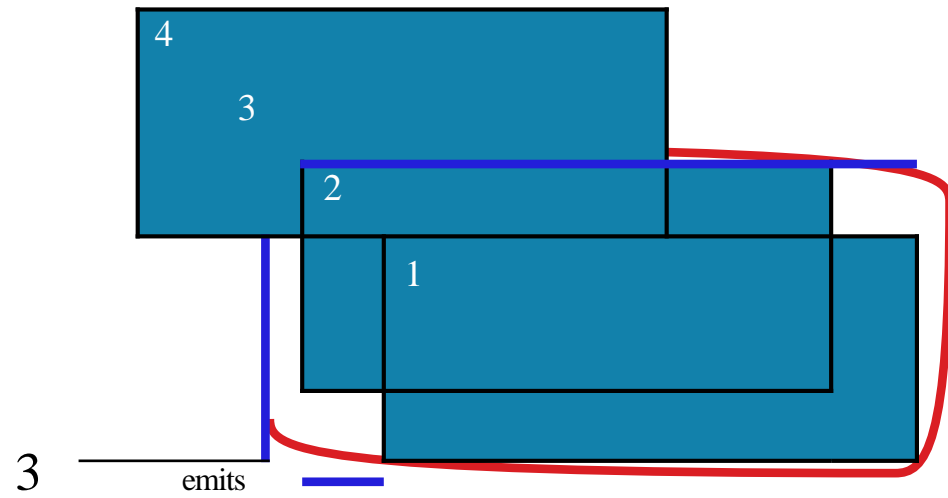


```
val x = sc.parallelize(Array(1,2,3), 2)
val y = x.collect()

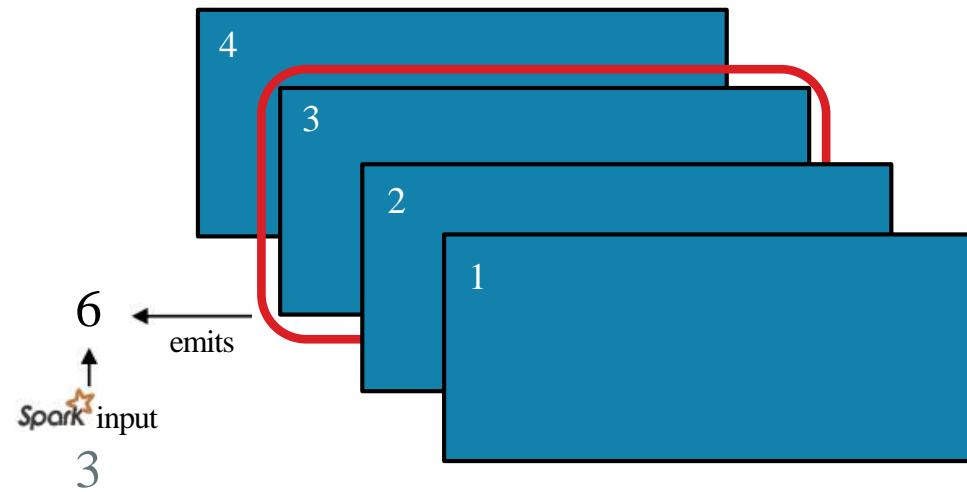
val xOut = x.glom().collect()
println(y)
```

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

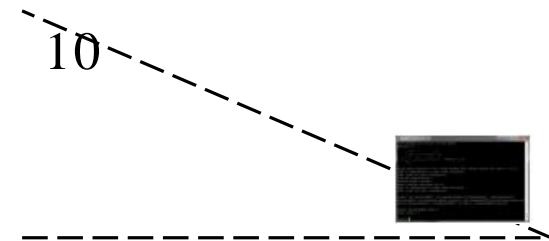
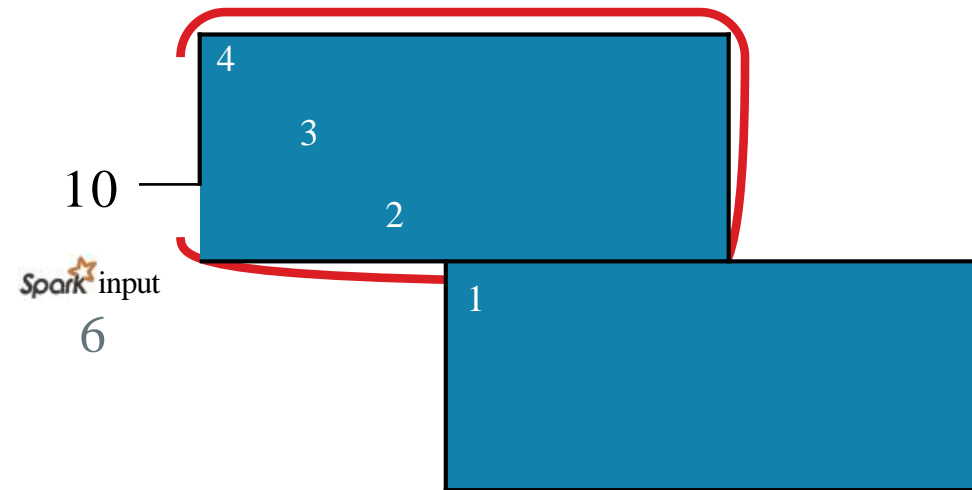
y: `[1, 2, 3]`



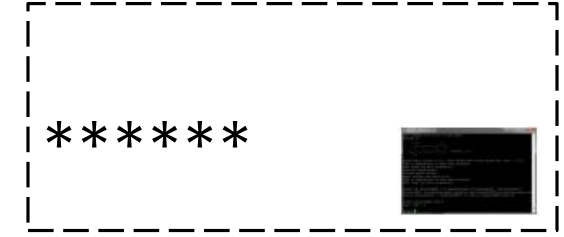
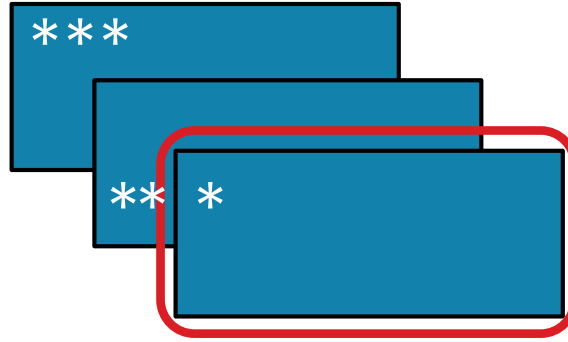
REDUCE



REDUCE



REDUCE



`reduce(f)`

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



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

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



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

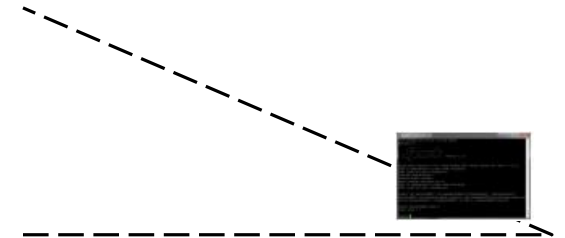
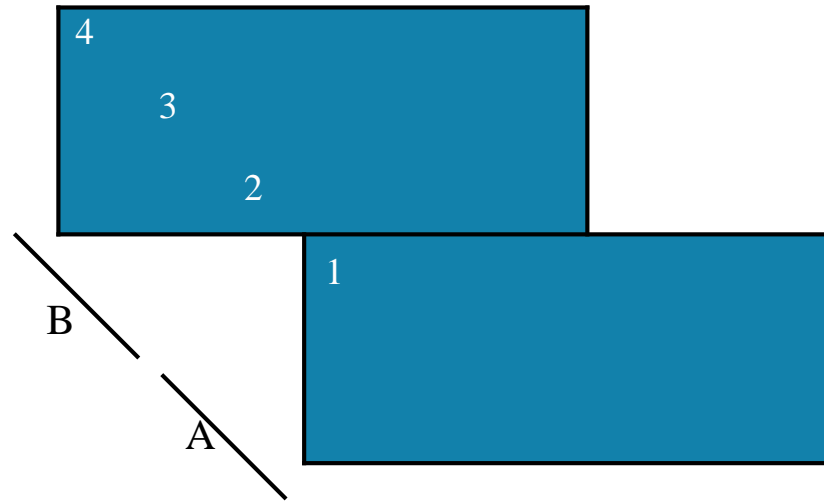
println(x.collect.mkString(", "))
println(y)
```



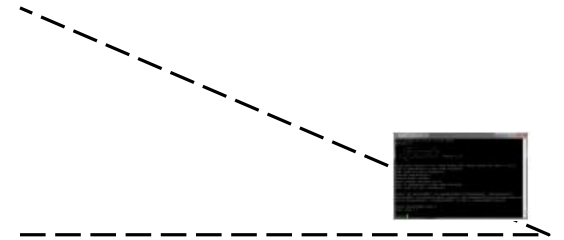
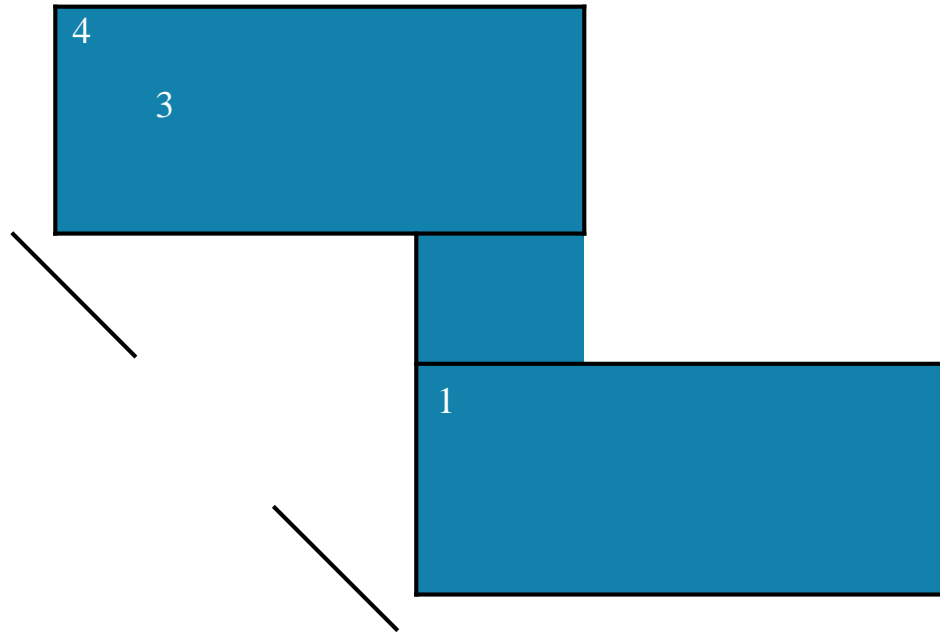
x: [1, 2, 3, 4]

y: 10

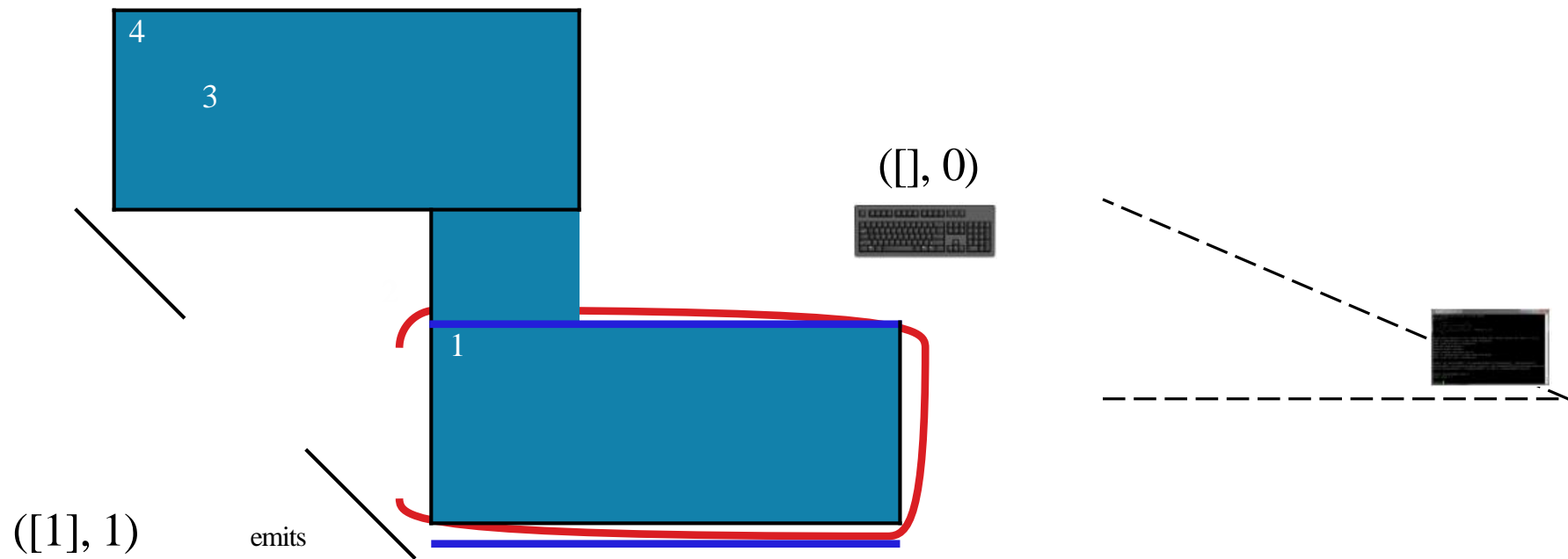
AGGREGATE



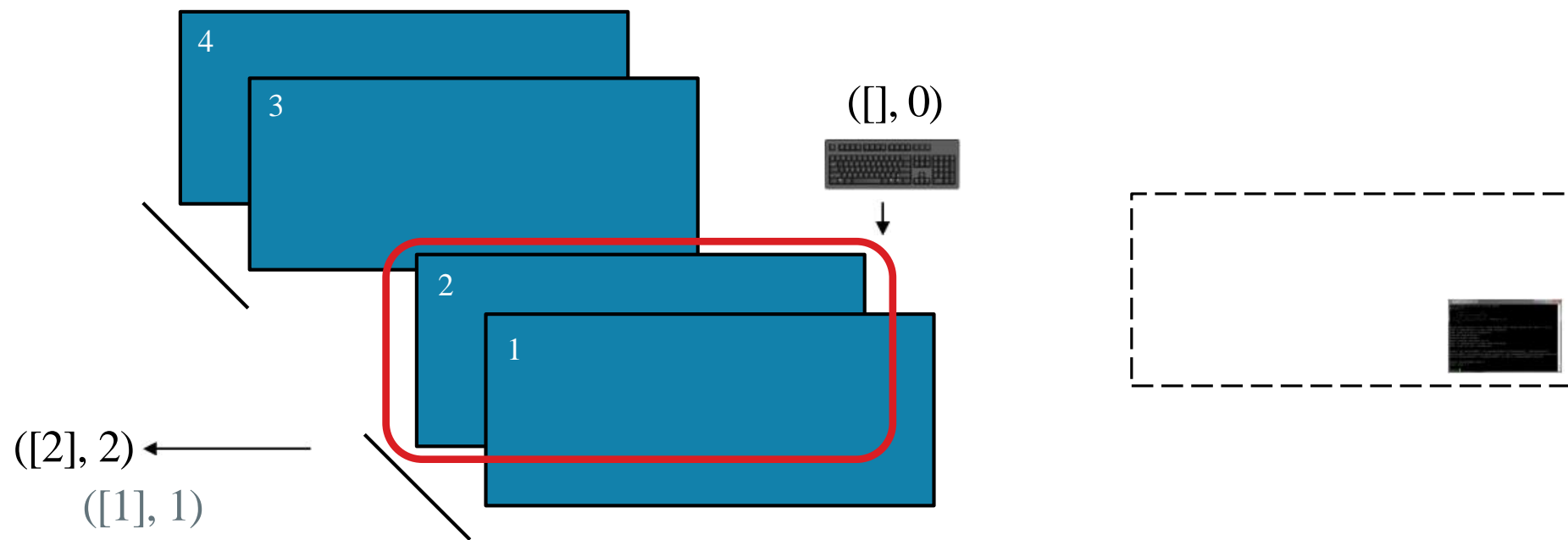
AGGREGATE



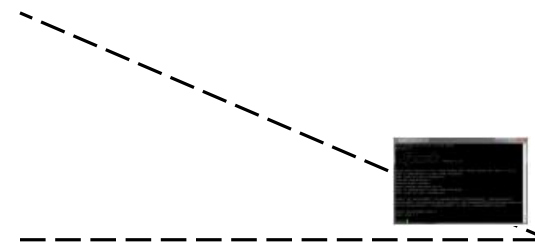
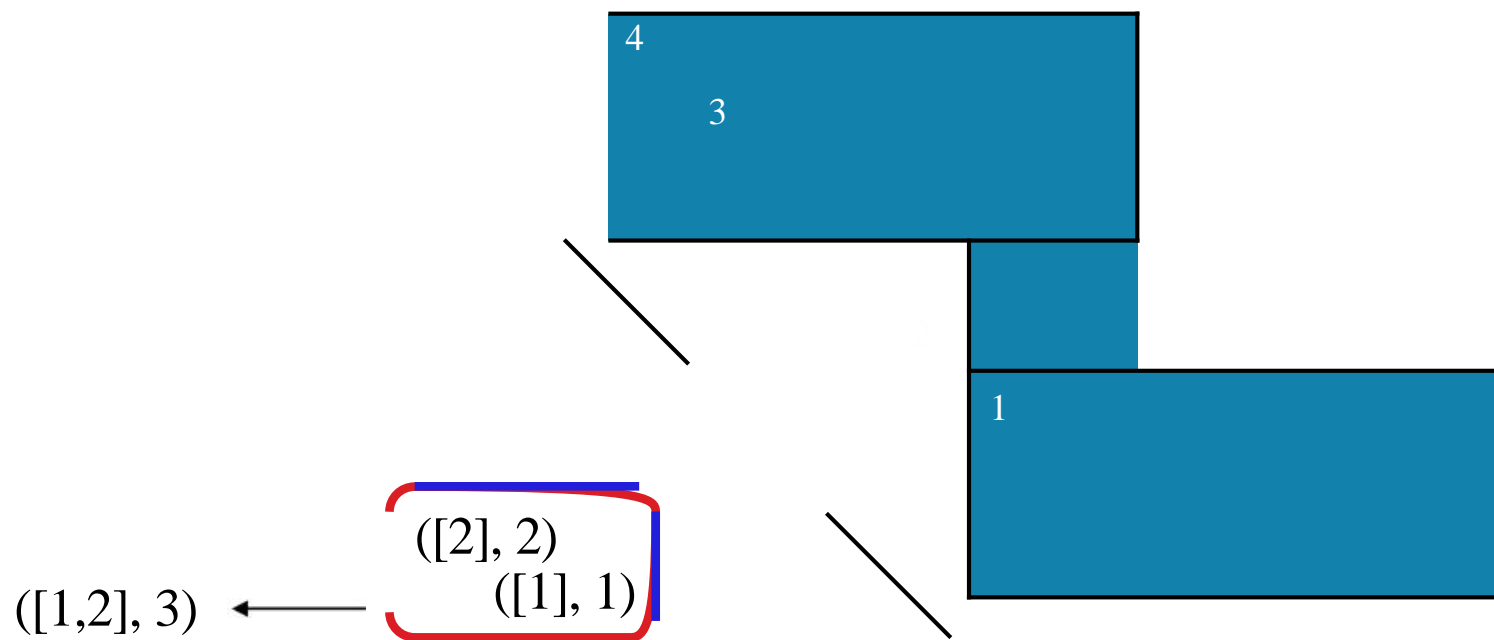
AGGREGATE



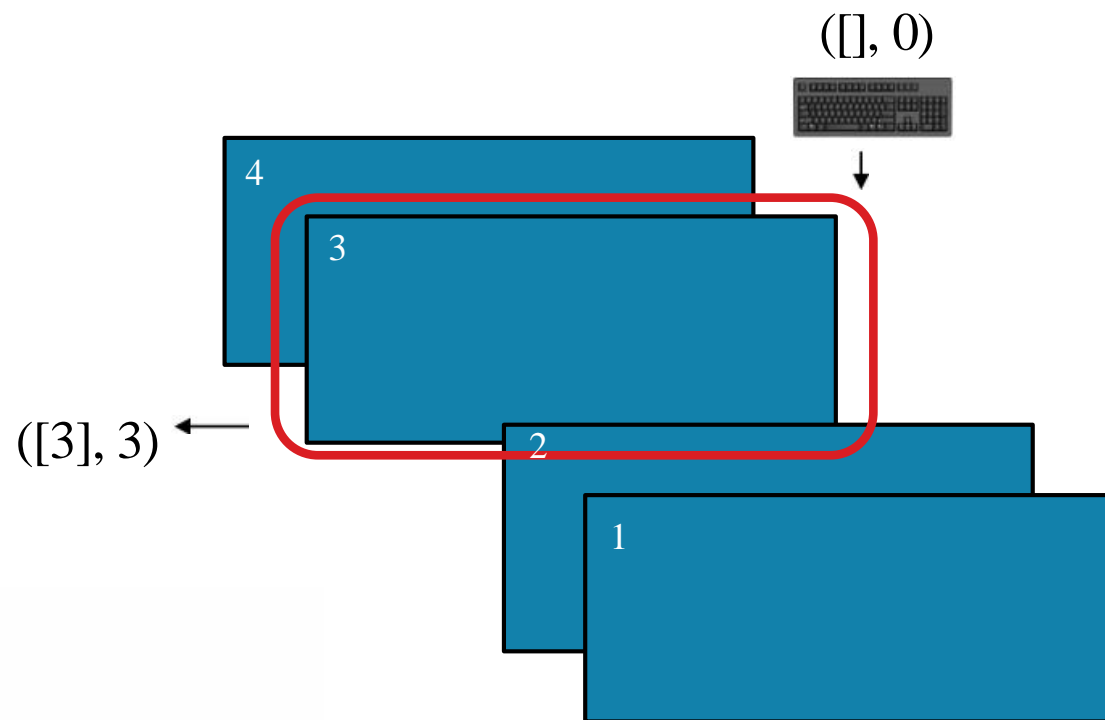
AGGREGATE



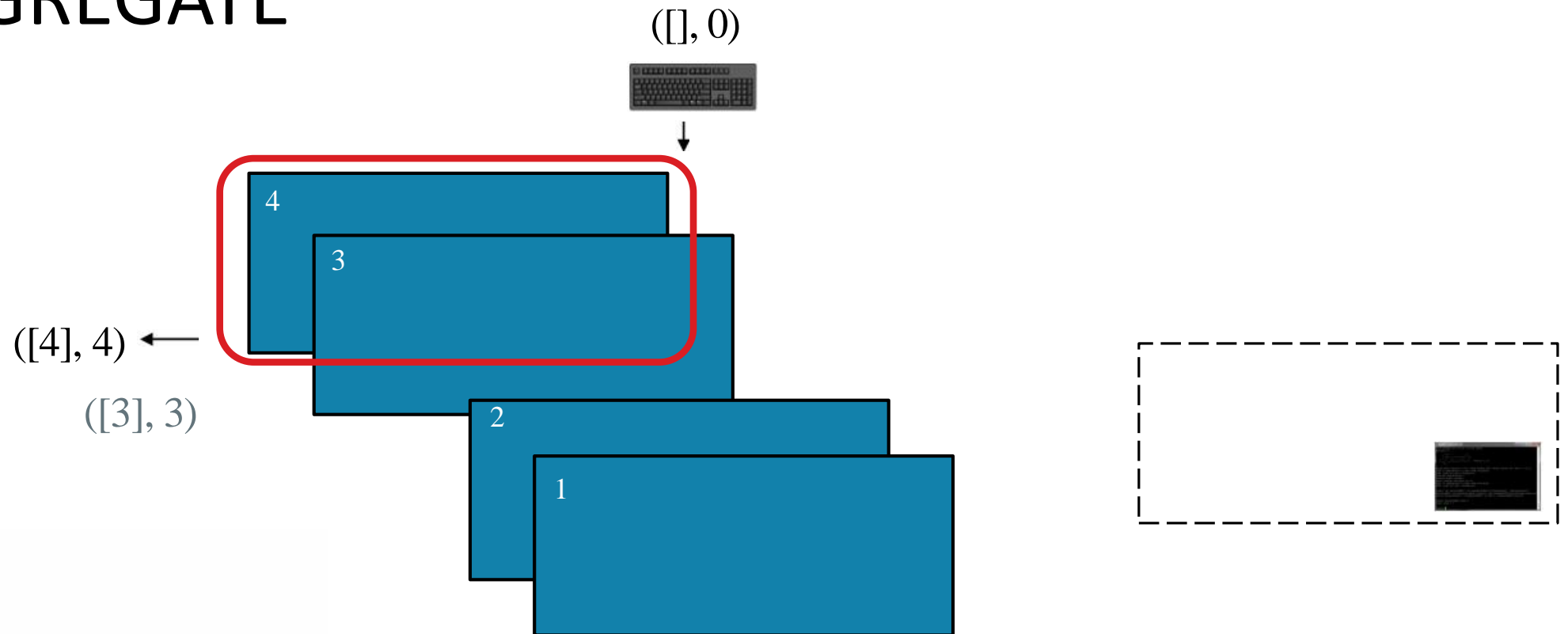
AGGREGATE



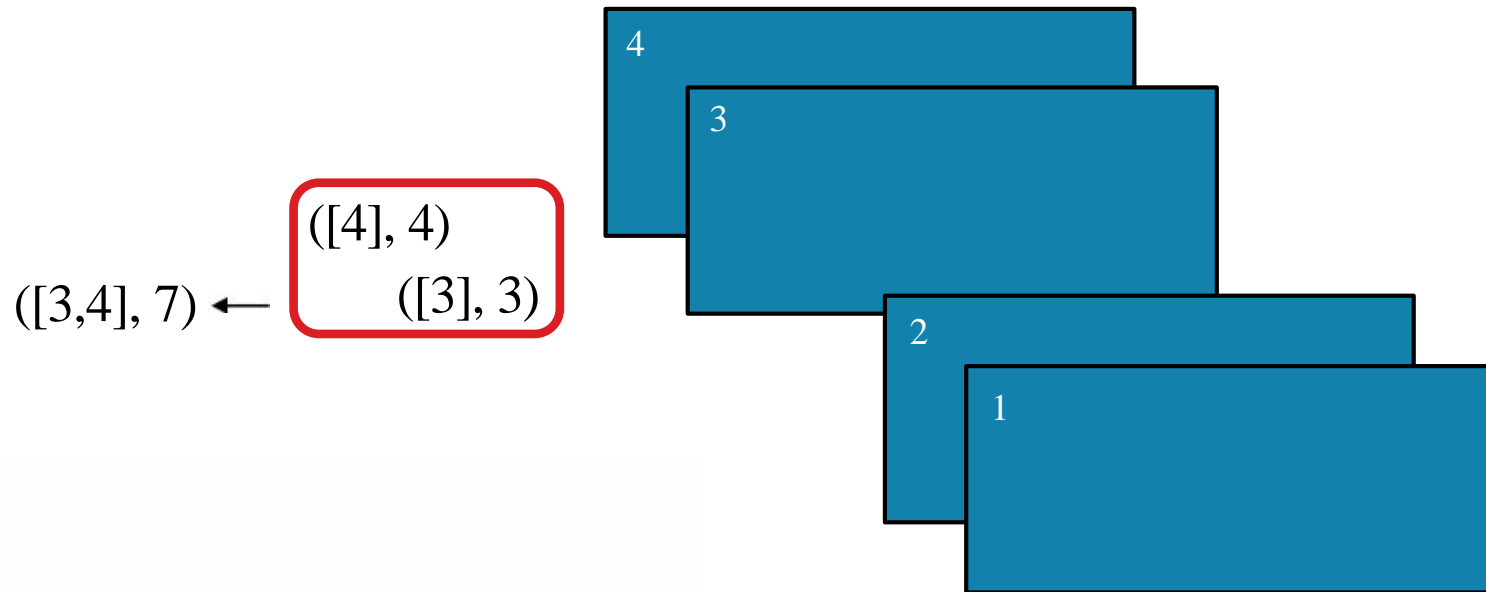
AGGREGATE



AGGREGATE



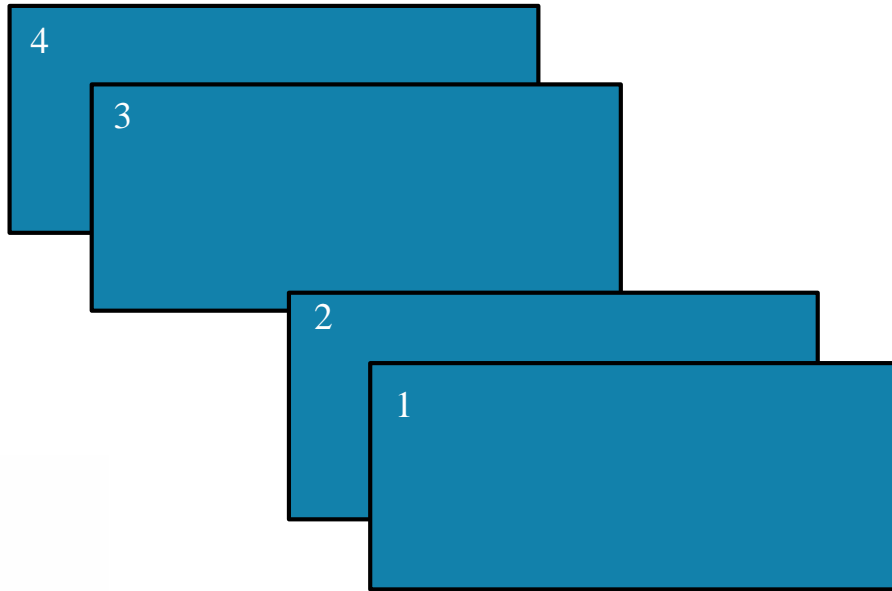
AGGREGATE



AGGREGATE

$([3,4], 7)$

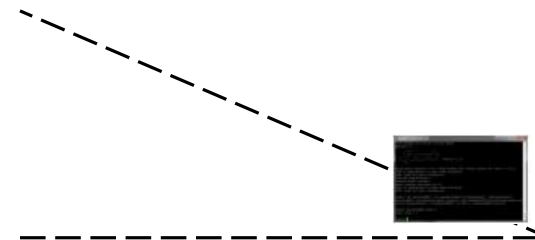
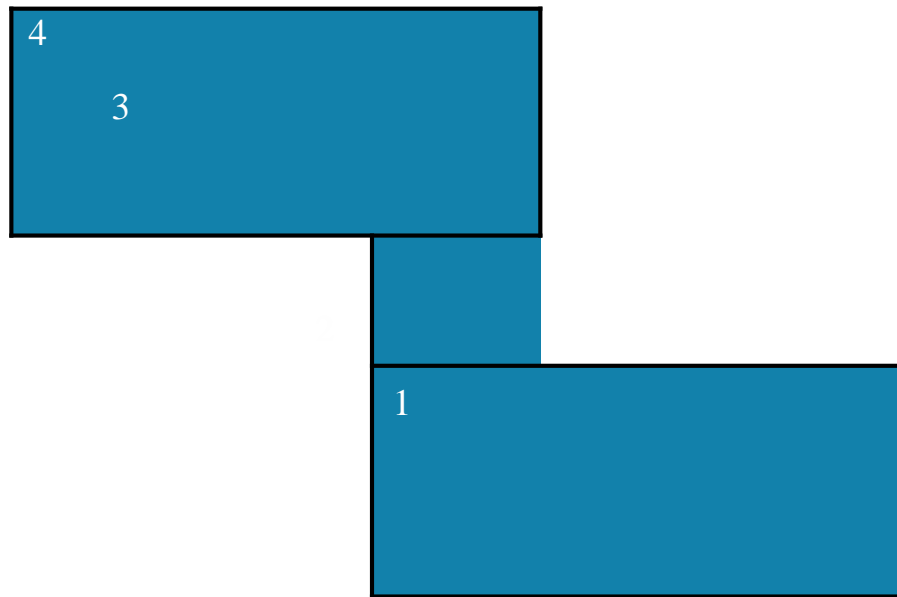
$([1,2], 3)$



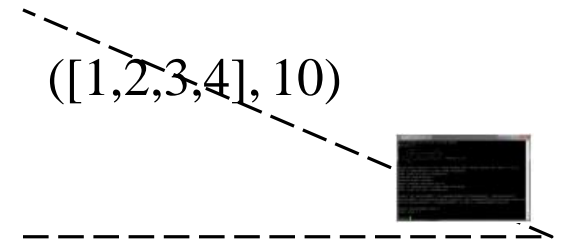
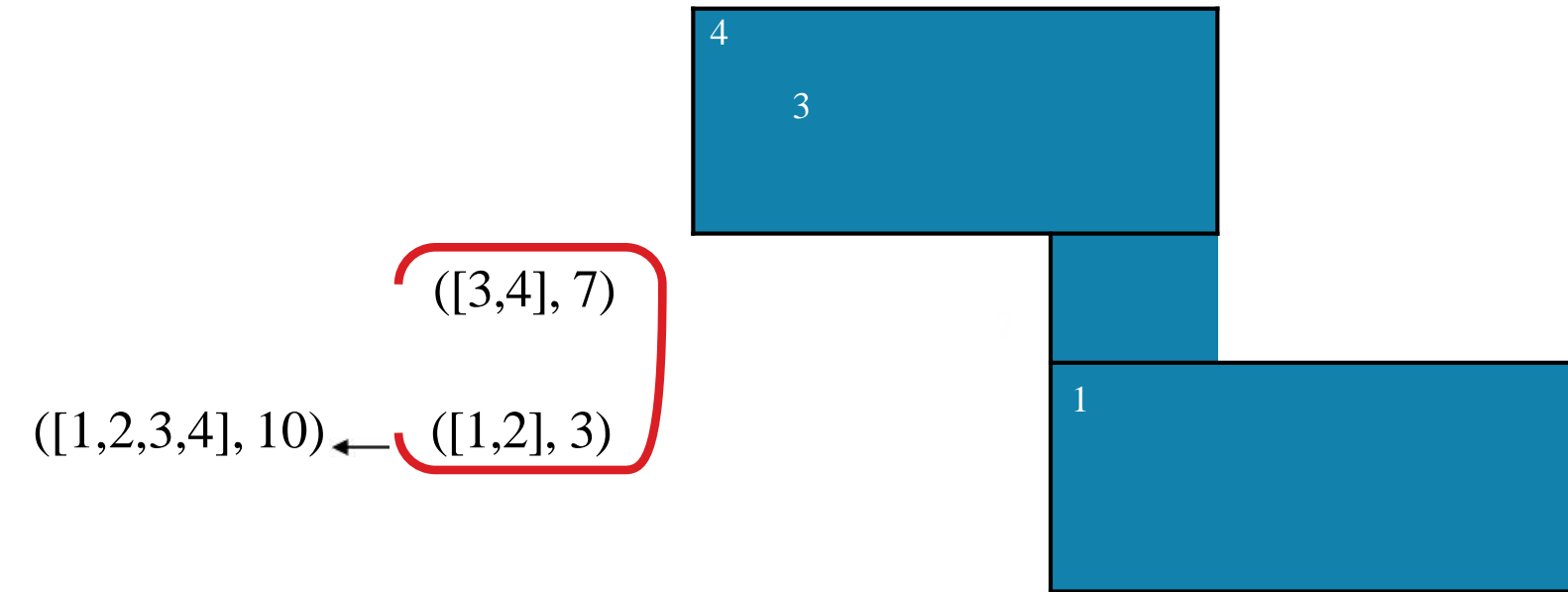
AGGREGATE

$([3,4], 7)$

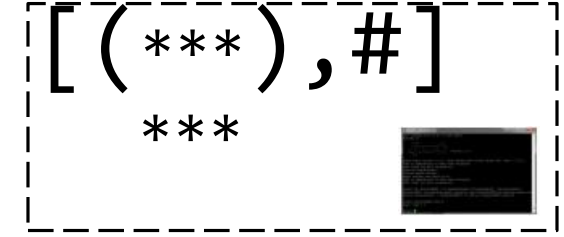
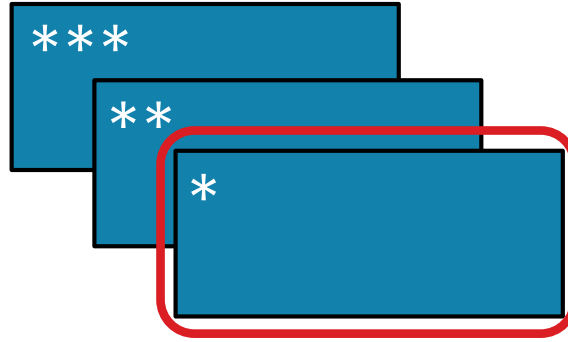
$([1,2], 3)$



AGGREGATE



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)
```

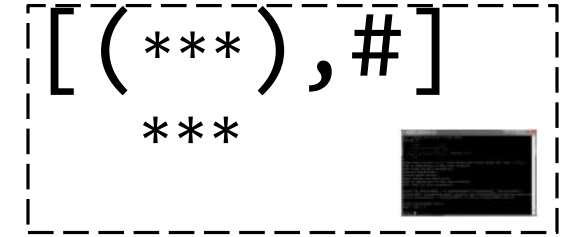
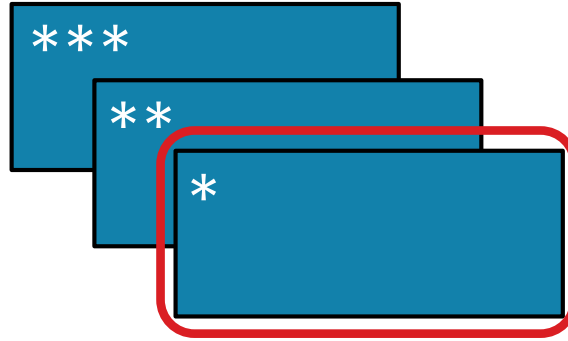
```
print(y)
```



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)

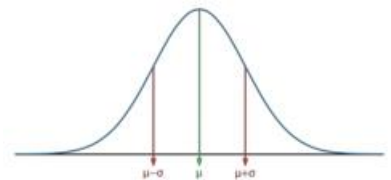
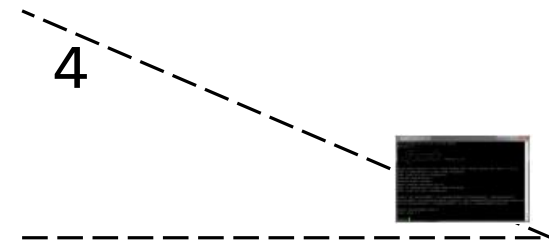
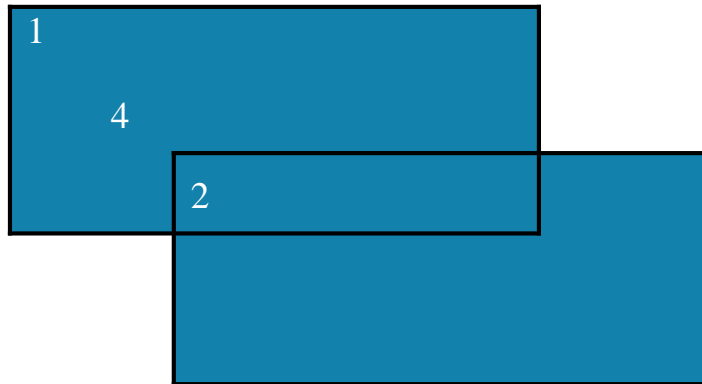
println(y)
```



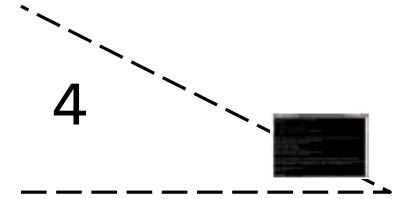
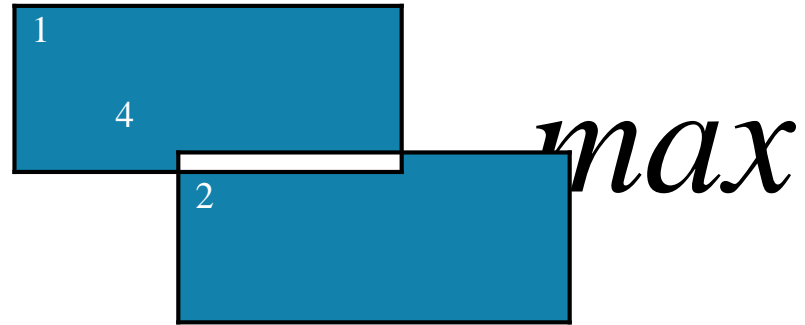
x: [1, 2, 3, 4]

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

MAX



MAX



`max()`

Return the maximum item in the RDD



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

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



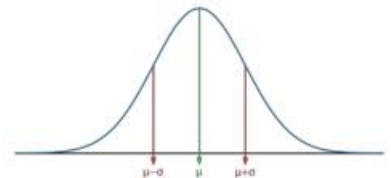
```
val x = sc.parallelize(Array(2,4,1))
val y = x.max

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

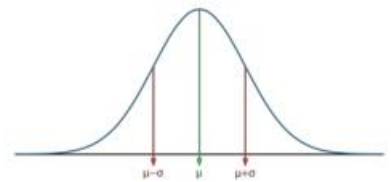
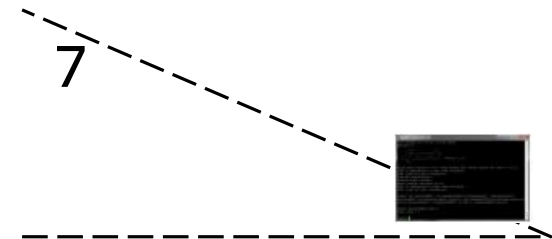
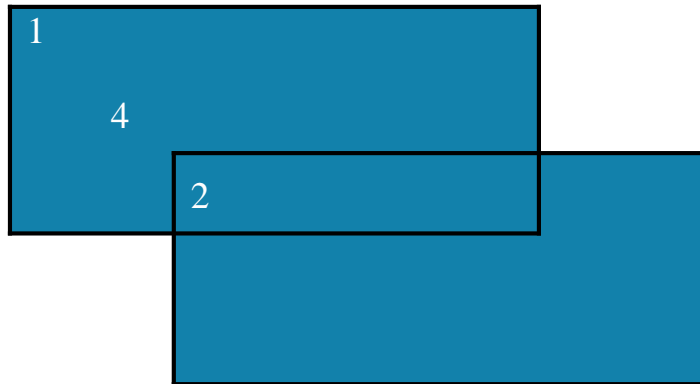


`x:` [2, 4, 1]

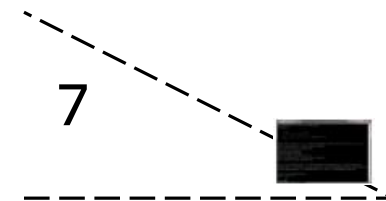
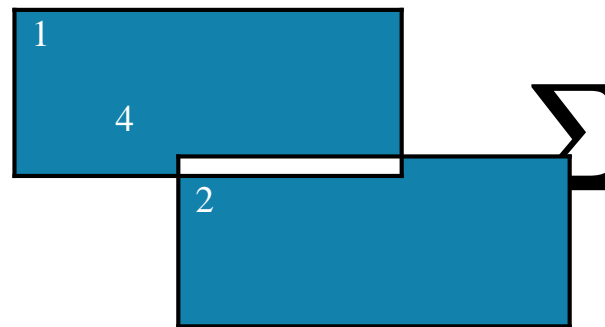
`y:` 4



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)
```

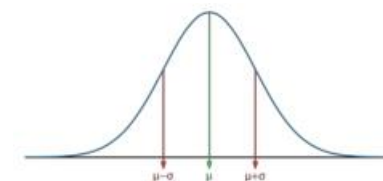


```
val x = sc.parallelize(Array(2,4,1))  
val y = x.sum  
  
println(x.collect().mkString(", "))  
println(y)
```

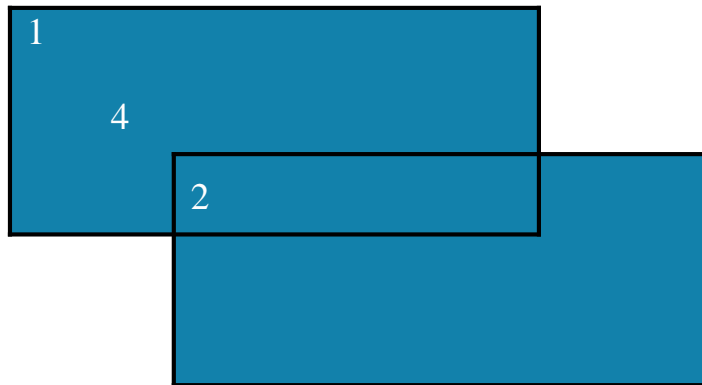


`x:` [2, 4, 1]

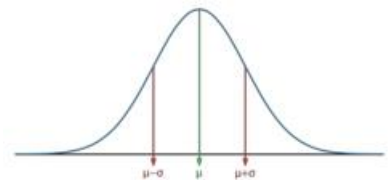

`y:` 7



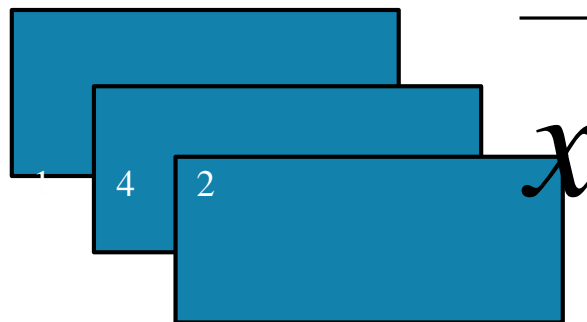
MEAN



2.33333333



MEAN



2.33333333

`mean()`

Return the mean of the items in the RDD



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

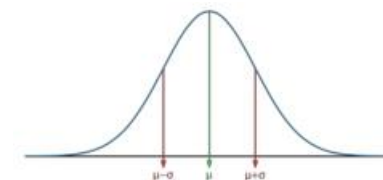


```
val x = sc.parallelize(Array(2,4,1))  
val y = x.mean  
  
println(x.collect().mkString(", "))  
println(y)
```

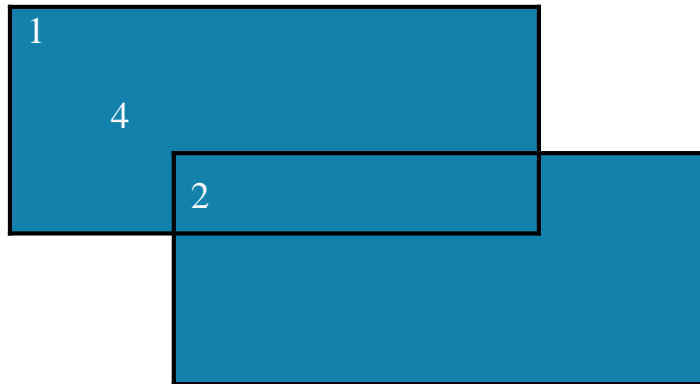


`x:` [2, 4, 1]

`y:` 2.3333333

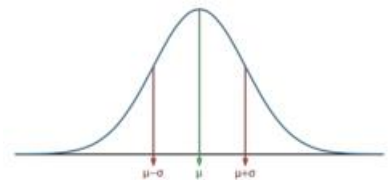


STDDEV

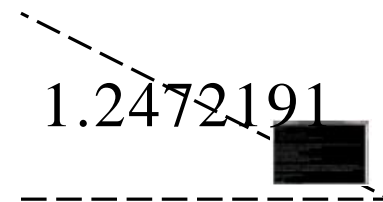
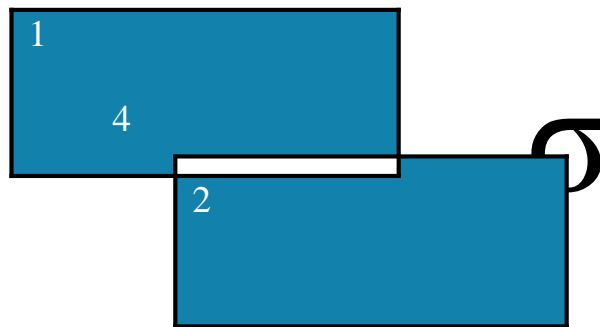


1.2472191

A small black terminal window with white text, connected by dashed lines to the number 1.2472191 and a horizontal dashed line below it.



STDDEV



`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)
```

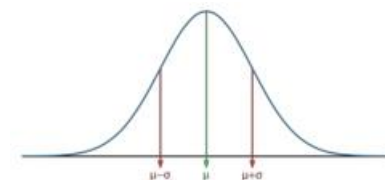


```
val x = sc.parallelize(Array(2,4,1))  
val y = x.stdev  
  
println(x.collect().mkString(", "))  
println(y)
```

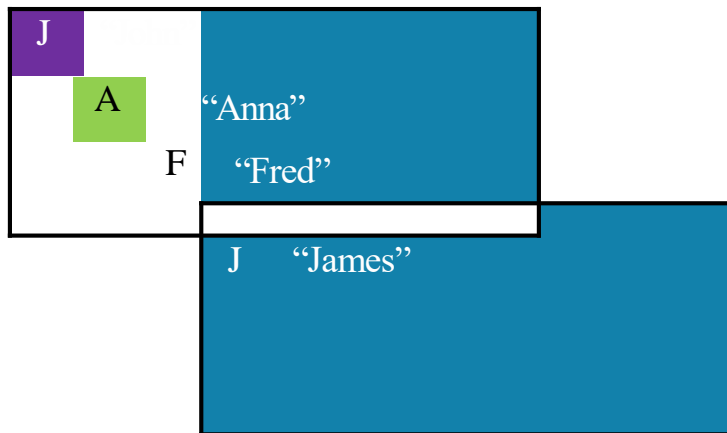


x: [2, 4, 1]

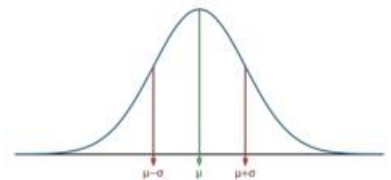
y: 1.2472191



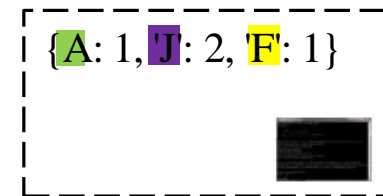
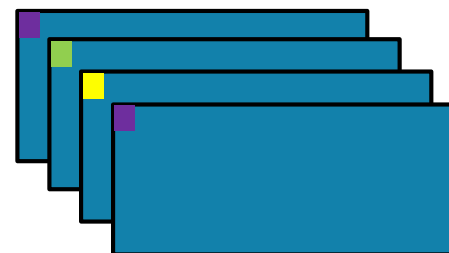
COUNTBYKEY



~~{'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)
```



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

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

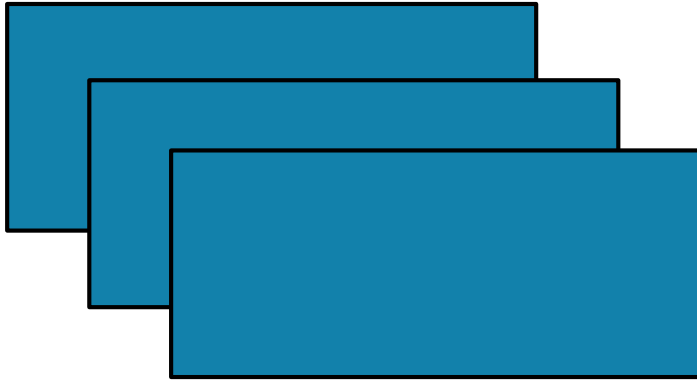


```
val x = sc.parallelize(Array(('J', "James"), ('F', "Fred"),  
                             ('A', "Anna"), ('J', "John")))
```

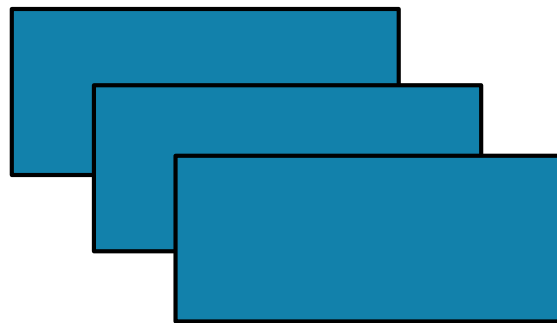
```
val y = x.countByKey()  
println(y)
```



SAVEASTEXTFIKE



SAVEASTEXTFIKE



`saveAsTextFile(path, compressionCodecClass=None)`

Save the RDD to the filesystem indicated in the path



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

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



```
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']