Basics of RDD - More Operations

More Transformations

sample(withReplacement, fraction, [seed])

Sample an RDD, with or without replacement.

More Transformations

sample(withReplacement, fraction, [seed])

Sample an RDD, with or without replacement.

```
$   val seq = sc.parallelize(1 to 100, 5)
$   seq.sample(false, 0.1).collect();
        [8, 19, 34, 37, 43, 51, 70, 83]
```

More Transformations

sample(withReplacement, fraction, [seed])

Sample an RDD, with or without replacement.

```
$   val seq = sc.parallelize(1 to 100, 5)
$   seq.sample(false, 0.1).collect();
        [8, 19, 34, 37, 43, 51, 70, 83]

$   seq.sample(true, 0.1).collect();
        [14, 26, 40, 47, 55, 67, 69, 69]
```

Please note that the result will be different on every run.

Common Transformations (continued..)

mapPartitions(f, preservesPartitioning=False)

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

Common Transformations (continued..)

mapPartitions(f, preservesPartitioning=False)

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

```
$   val rdd = sc.parallelize(1 to 50, 3)
```

Common Transformations (continued..)

mapPartitions(f, preservesPartitioning=False)

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

```
$   val rdd = sc.parallelize(1 to 50, 3)
$   def f(l:Iterator[Int]):Iterator[Int] = {
        var sum = 0
        while(l.hasNext){
            sum = sum + l.next
        }
        return List(sum).iterator
    }
```

Common Transformations (continued..)

mapPartitions(f, preservesPartitioning=False)

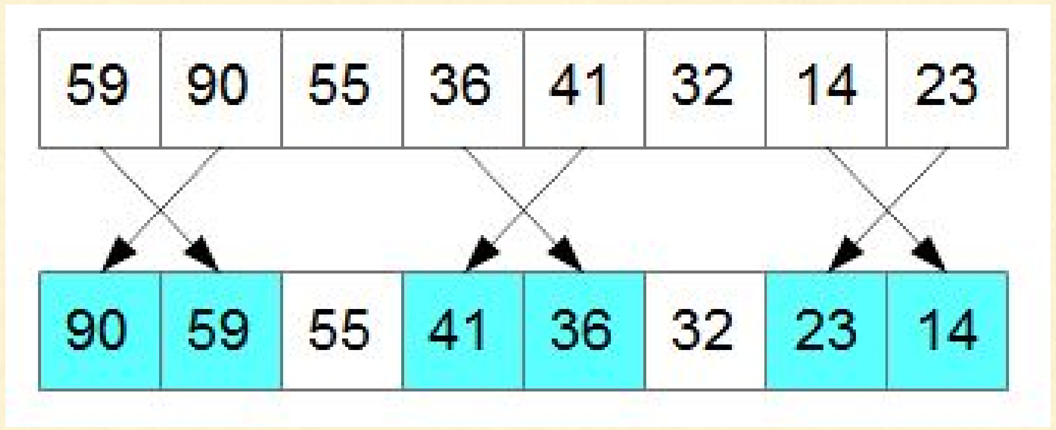
Return a new RDD by applying a function to each partition of this RDD.

```
$   val rdd = sc.parallelize(1 to 50, 3)
$   def f(l:Iterator[Int]):Iterator[Int] = {
        var sum = 0
        while(l.hasNext){
            sum = sum + l.next
        }
        return List(sum).iterator
    }

$   rdd.mapPartitions(f).collect()
        Array(136, 425, 714)
```

Common Transformations (continued..)

sortBy(func, ascending=True, numPartitions=None)

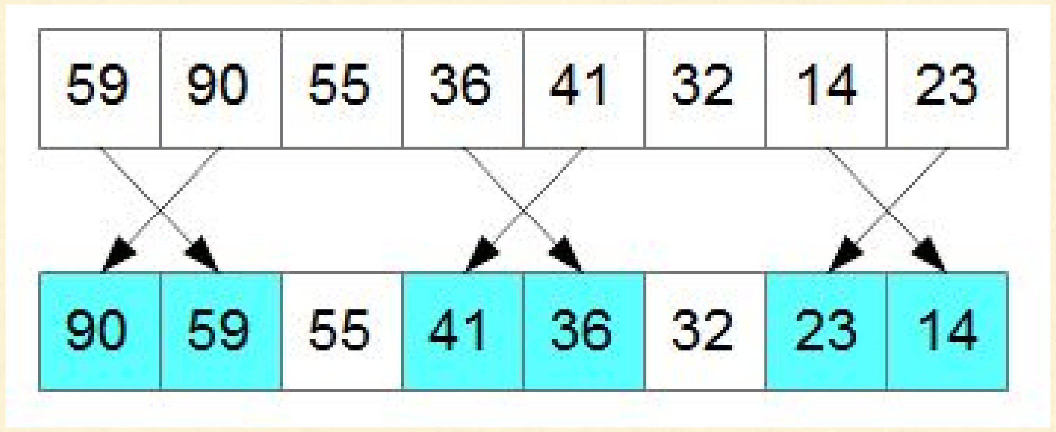
Sorts this RDD by the given func



Common Transformations (continued..)

sortBy(func, ascending=True, numPartitions=None)

Sorts this RDD by the given func

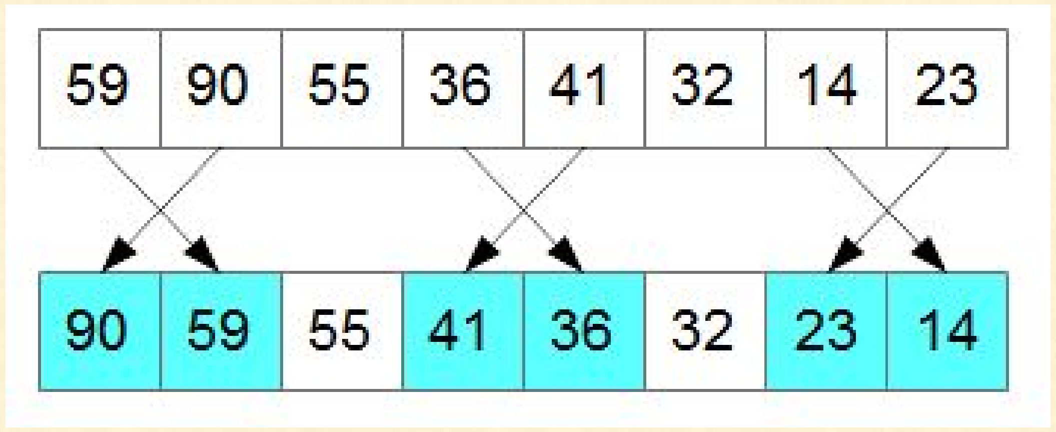


func: *A function used to compute the sort key for each element.*

Common Transformations (continued..)

sortBy(func, ascending=True, numPartitions=None)

Sorts this RDD by the given func



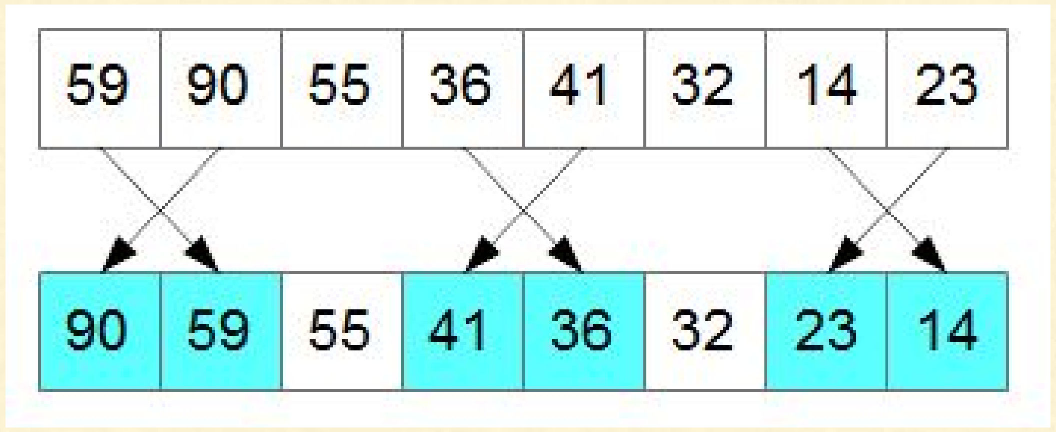
func: *A function used to compute the sort key for each element.*

ascending: *A flag to indicate whether the sorting is ascending or descending.*

Common Transformations (continued..)

sortBy(func, ascending=True, numPartitions=None)

Sorts this RDD by the given func



func: *A function used to compute the sort key for each element.*

ascending: *A flag to indicate whether the sorting is ascending or descending.*

numPartitions: *Number of partitions to create.*

Common Transformations (continued..)

sortBy(keyfunc, ascending=True, numPartitions=None)

Sorts this RDD by the given keyfunc

```
» var tmp = List(('a', 1), ('b', 2), ('1', 3), ('d', 4), ('2', 5))
» var rdd = sc.parallelize(tmp)
```

Common Transformations (continued..)

sortBy(keyfunc, ascending=True, numPartitions=None)

Sorts this RDD by the given keyfunc

```
» var tmp = List(('a', 1), ('b', 2), ('1', 3), ('d', 4), ('2', 5))
» var rdd = sc.parallelize(tmp)

» rdd.sortBy(x => x._1).collect()
  [('1', 3), ('2', 5), ('a', 1), ('b', 2), ('d', 4)]
```

Common Transformations (continued..)

sortBy(keyfunc, ascending=True, numPartitions=None)

Sorts this RDD by the given keyfunc

```
» var tmp = List(('a', 1), ('b', 2), ('1', 3), ('d', 4), ('2', 5))
» var rdd = sc.parallelize(tmp)

» rdd.sortBy(x => x._2).collect()
  [('a', 1), ('b', 2), ('1', 3), ('d', 4), ('2', 5)]
```

Common Transformations (continued..)

sortBy(keyfunc, ascending=True, numPartitions=None)

Sorts this RDD by the given keyfunc

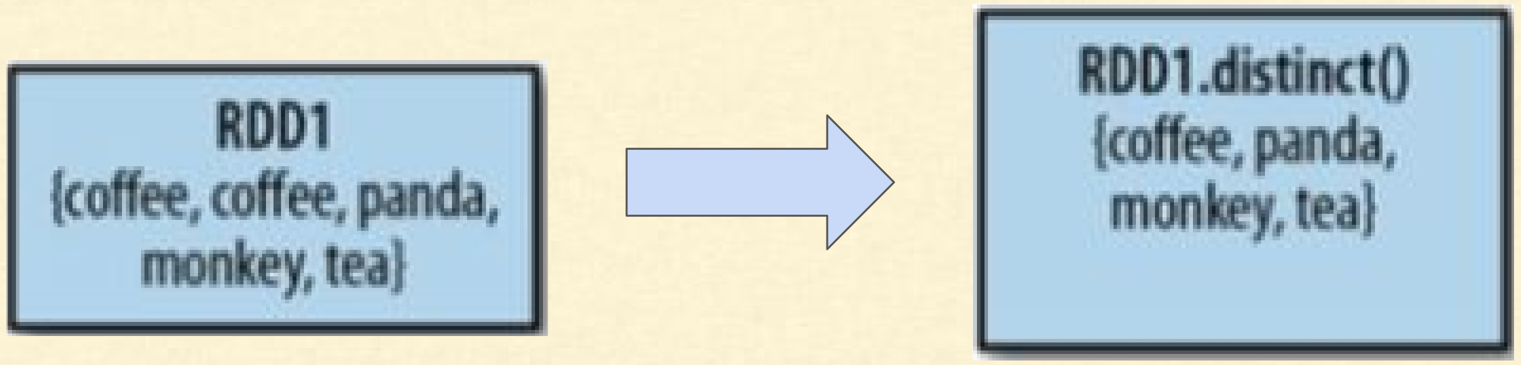
```
var rdd = sc.parallelize(Array(10, 2, 3,21, 4, 5))
var sortedrdd = rdd.sortBy(x => x)
sortedrdd.collect()
```

Common Transformations (continued..)

Pseudo set operations

Though RDD is not really a set but still the set operations try to provide you utility set functions

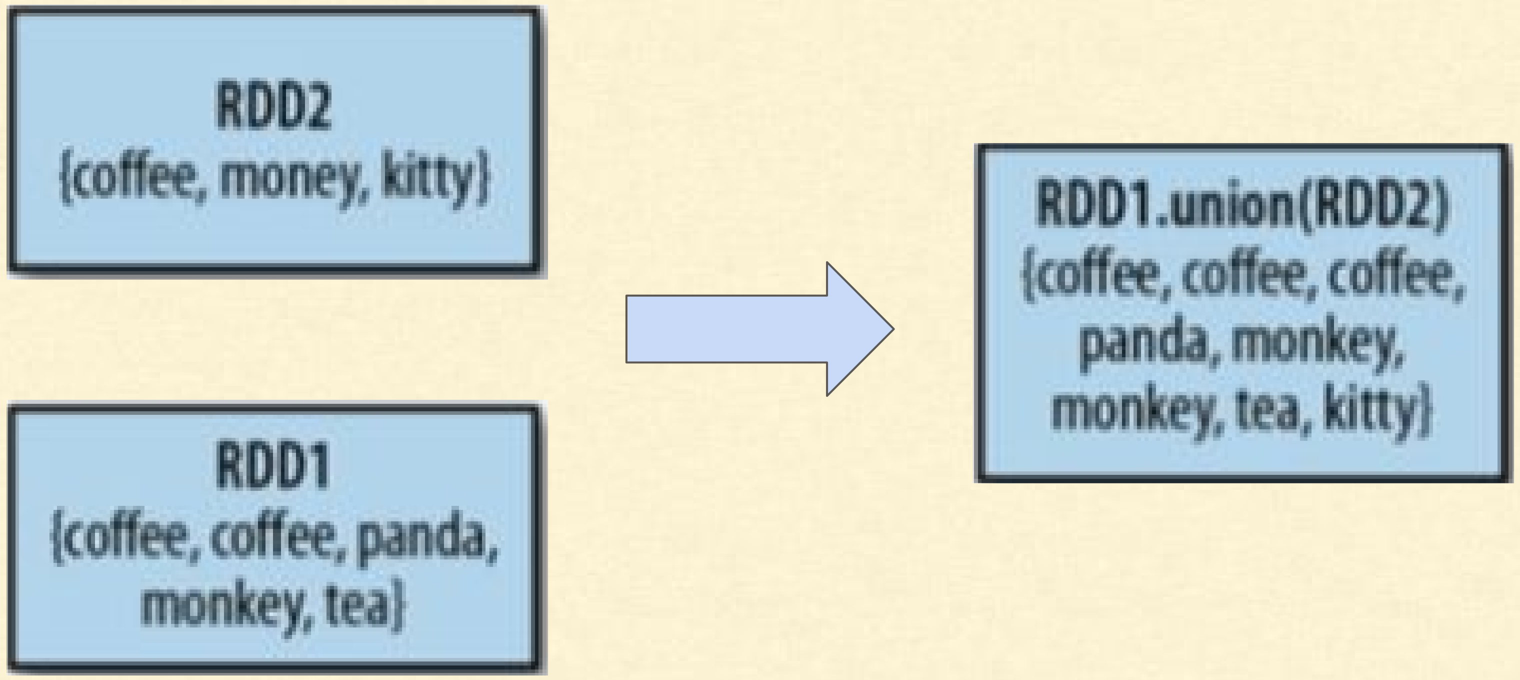
Set operations (Pseudo)


RDD1
{coffee, coffee, panda, monkey, tea}
RDD1.distinct()
{coffee, panda, monkey, tea}

distinct()

- Give the set property to your rdd
- Expensive as shuffling is required

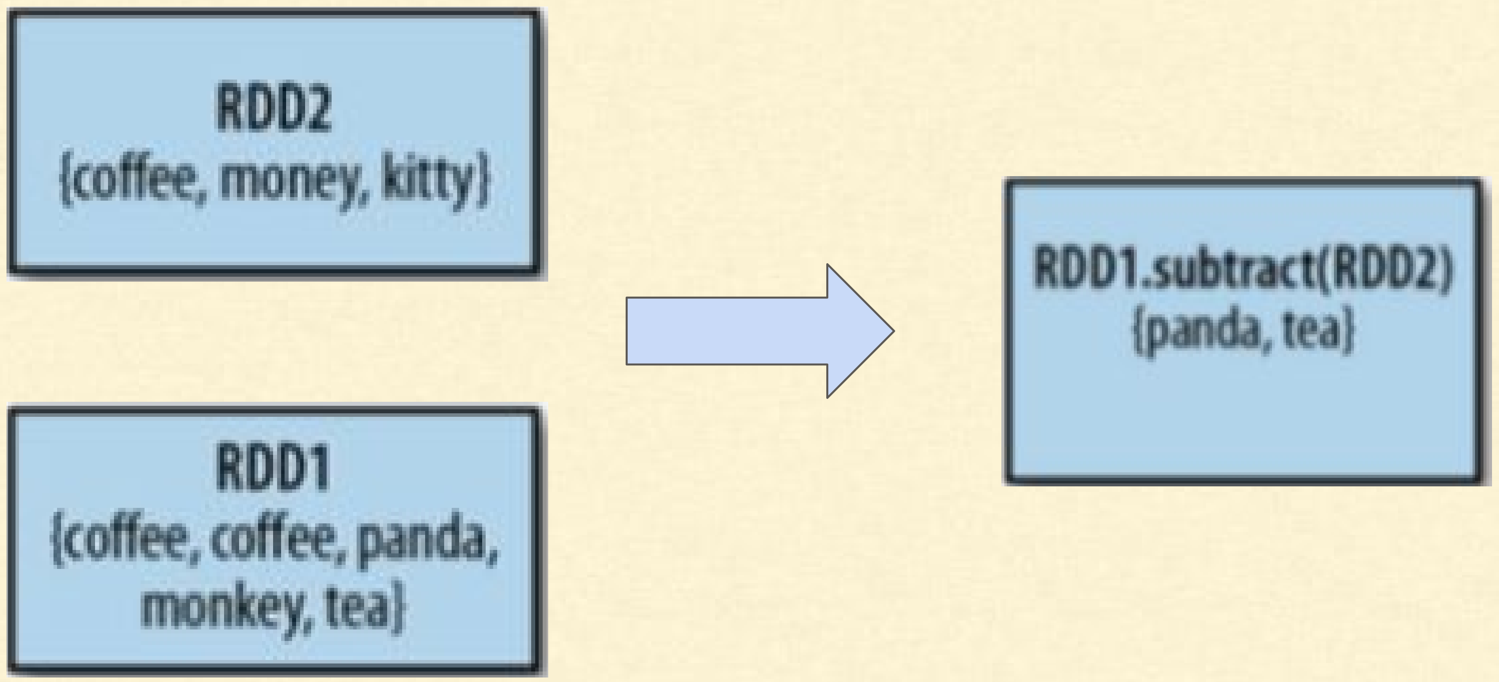
Set operations (Pseudo)


RDD2
{coffee, money, kitty}
RDD1
{coffee, coffee, panda, monkey, tea}
RDD1.union(RDD2)
{coffee, coffee, coffee, panda, monkey, monkey, tea, kitty}

union()

+ Simply appends one rdd to another
+ Is not same as mathematical function
+ It may have duplicates

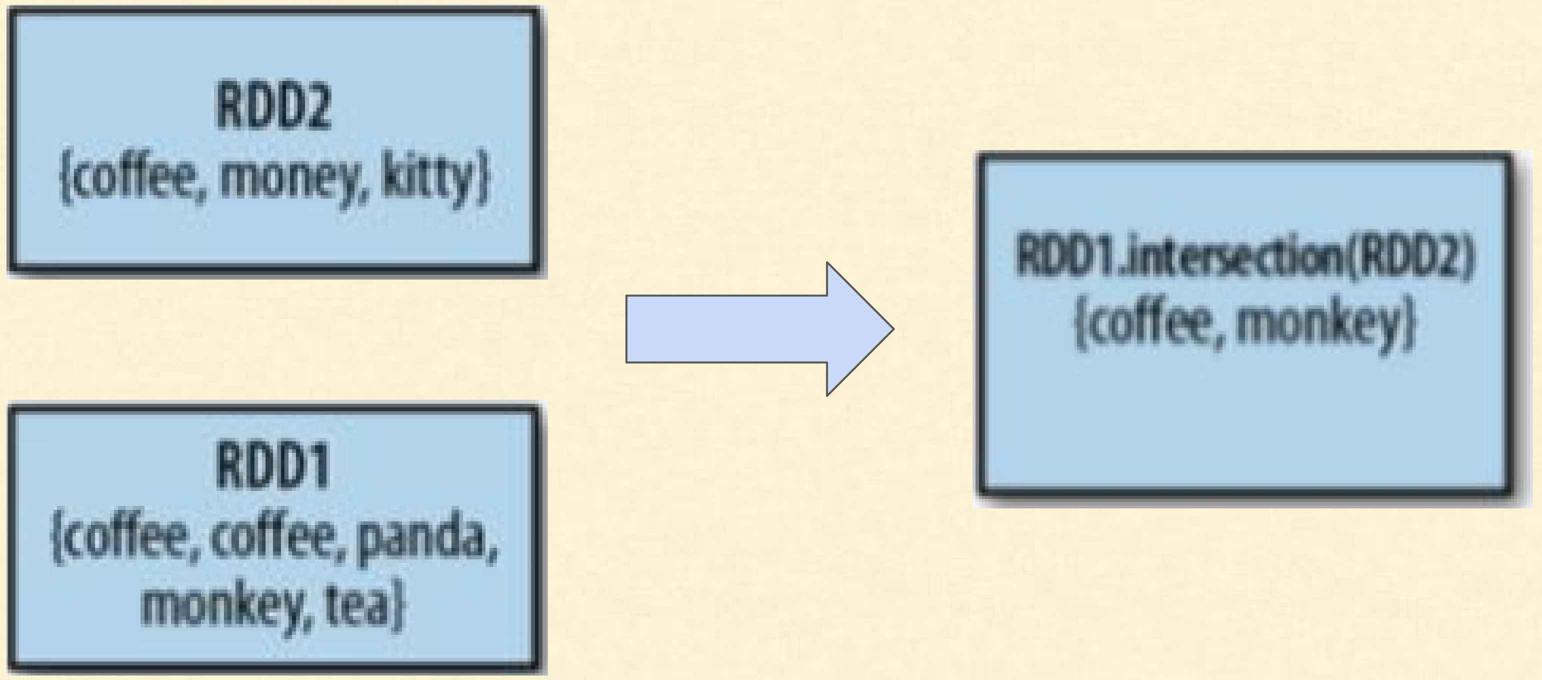
Set operations (Pseudo)



subtract()

+ Returns values in first RDD and not second
+ Requires Shuffling like intersection()

Set operations (Pseudo)



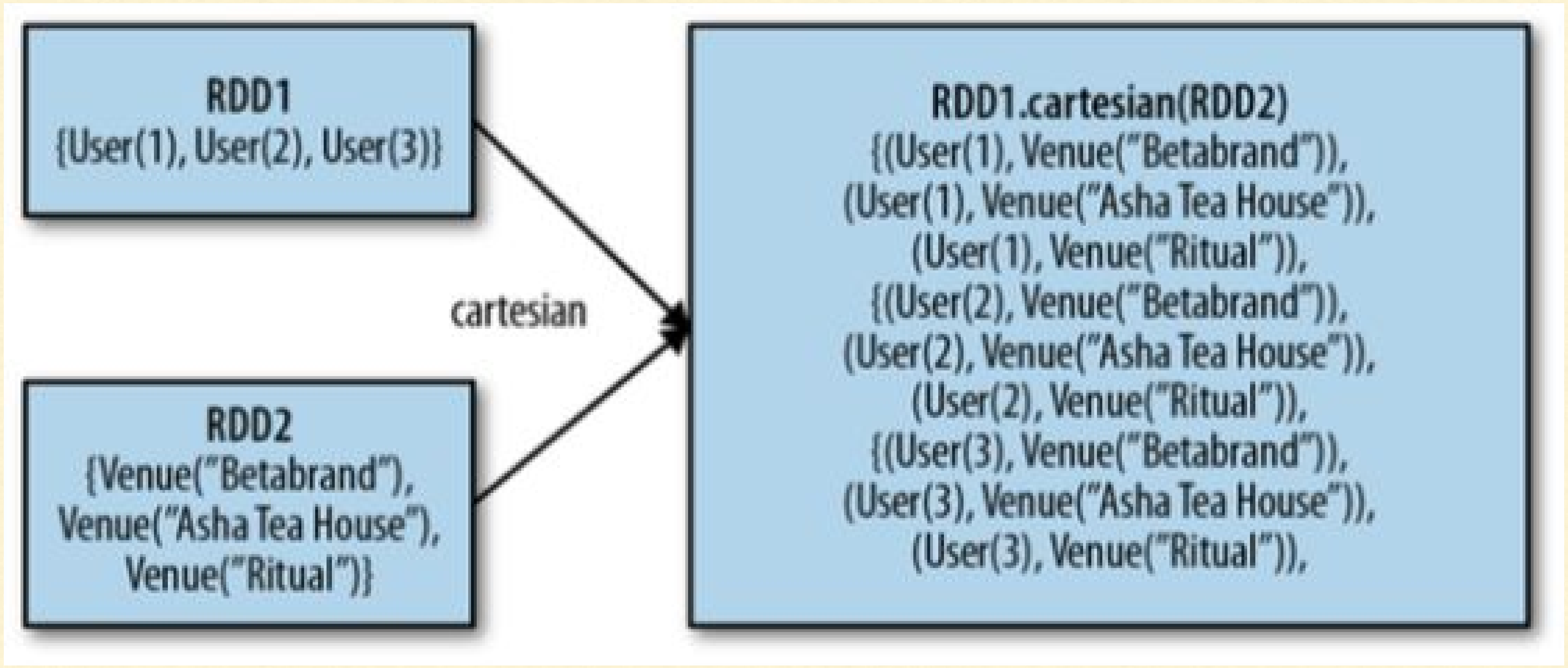
intersection()

- Finds common values in RDDs
- Also removes duplicates
- Requires shuffling

Set operations (Pseudo)

cartesian()

+ Returns all possible pairs of (a,b)
+ a is in source RDD and b is in other RDD


RDD1
{User(1), User(2), User(3)}
RDD2
{Venue("Betabrand"), Venue("Asha Tea House"), Venue("Ritual")}
cartesian
RDD1.cartesian(RDD2)
{(User(1), Venue("Betabrand")),
(User(1), Venue("Asha Tea House")),
(User(1), Venue("Ritual")),
{(User(2), Venue("Betabrand")),
(User(2), Venue("Asha Tea House")),
(User(2), Venue("Ritual")),
{(User(3), Venue("Betabrand")),
(User(3), Venue("Asha Tea House")),
(User(3), Venue("Ritual")),

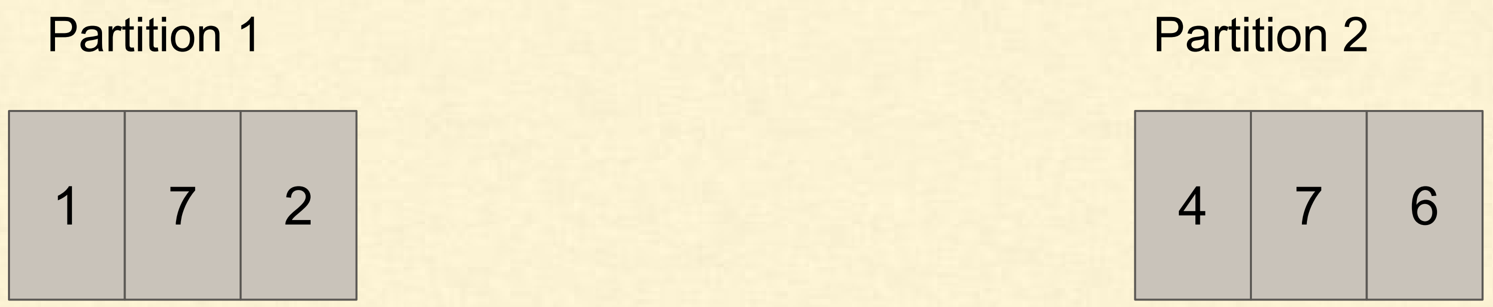
More Actions - fold()

fold(initial value, func)

+ Very similar to reduce
+ Provides a little extra control over the initialisation
+ Lets us specify an initial value

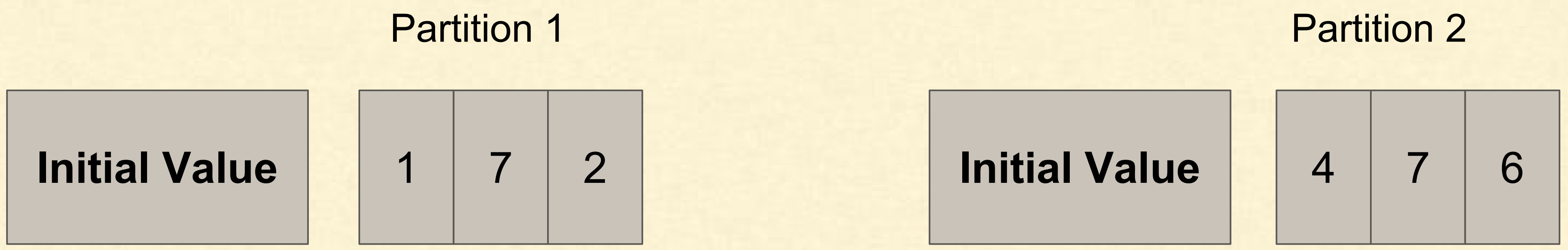
More Actions - fold()

fold(initial value, func)	Aggregates the elements of each partition and then the results for all the partitions using a given associative and commutative function and a neutral "zero value".



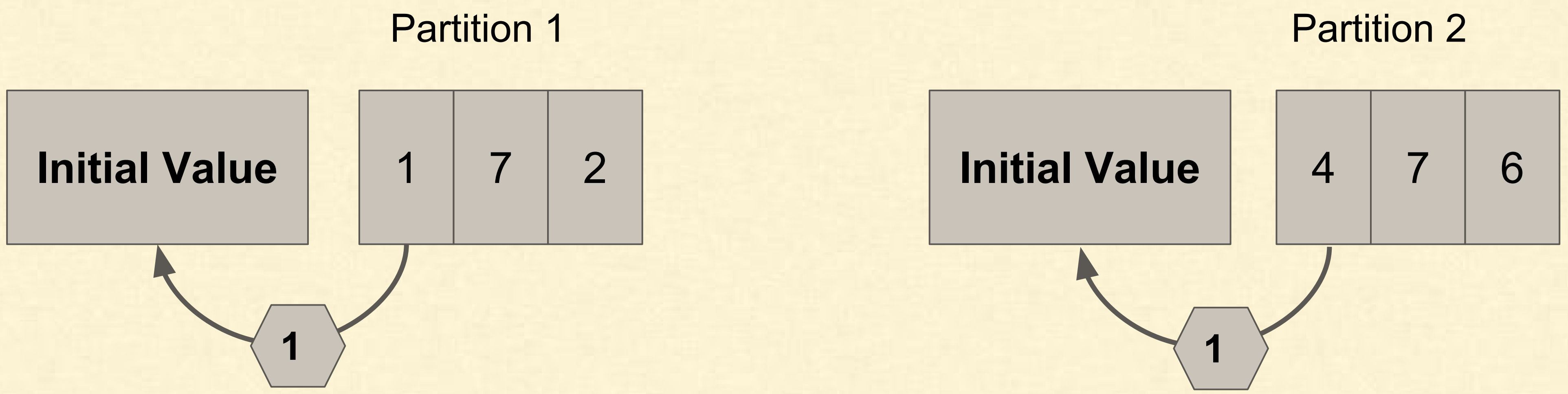
More Actions - fold()

fold(initial value, func)	Aggregates the elements of each partition and then the results for all the partitions using a given associative and commutative function and a neutral "zero value".



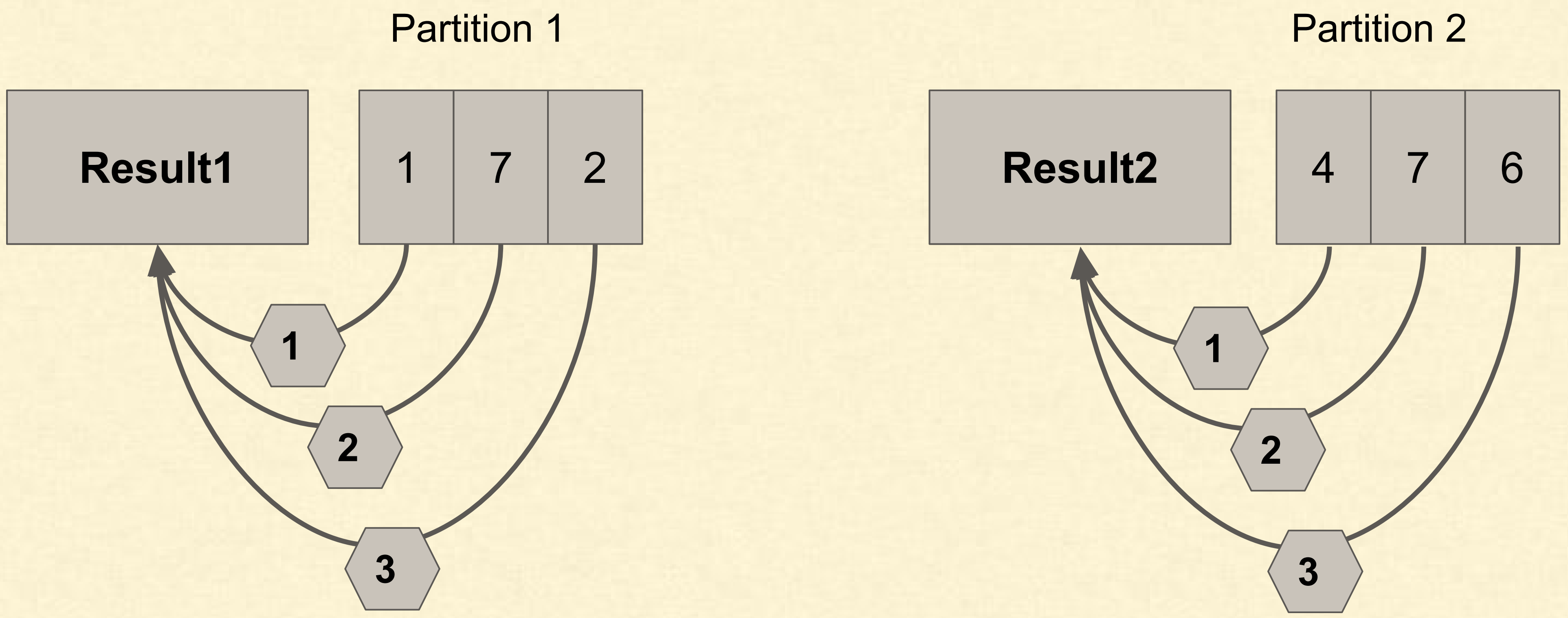
More Actions - fold()

fold(initial value)(func)	Aggregates the elements of each partition and then the results for all the partitions using a given associative and commutative function and a neutral "zero value".



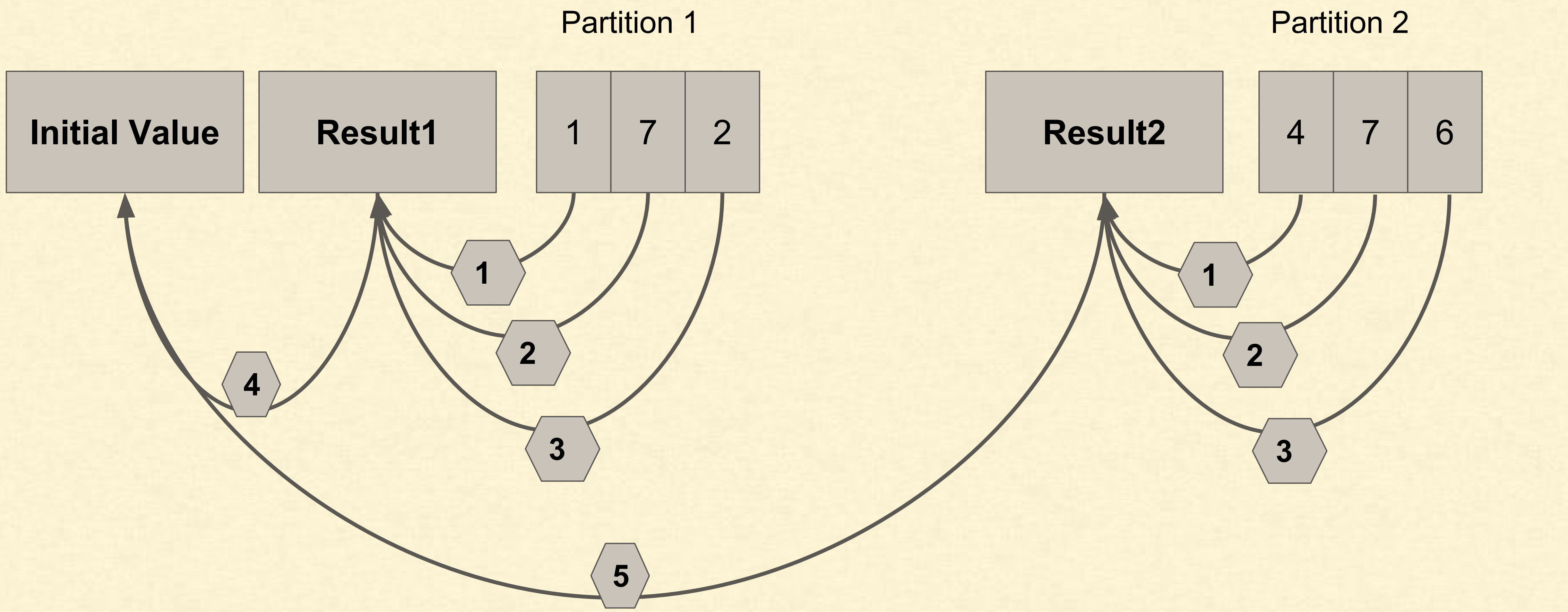
More Actions - fold()

fold(initial value)(func)	Aggregates the elements of each partition and then the results for all the partitions using a given associative and commutative function and a neutral "zero value".



More Actions - fold()

fold(initial value)(func)	Aggregates the elements of each partition and then the results for all the partitions using a given associative and commutative function and a neutral "zero value".



More Actions - fold()

fold(initial value, func)	Example: Concatnating to _

```
var myrdd = sc.parallelize(1 to 10, 2)
```

More Actions - fold()

fold(initial value, func)	Example: Concatnating to _

```
var myrdd = sc.parallelize(1 to 10, 2)
var myrdd1 = myrdd.map(_.toString)
```

More Actions - fold()

fold(initial value, func)	Example: Concatnating to _

```
var myrdd = sc.parallelize(1 to 10, 2)
var myrdd1 = myrdd.map(_.toString)

def concat(s:String, n:String):String = s + n
```

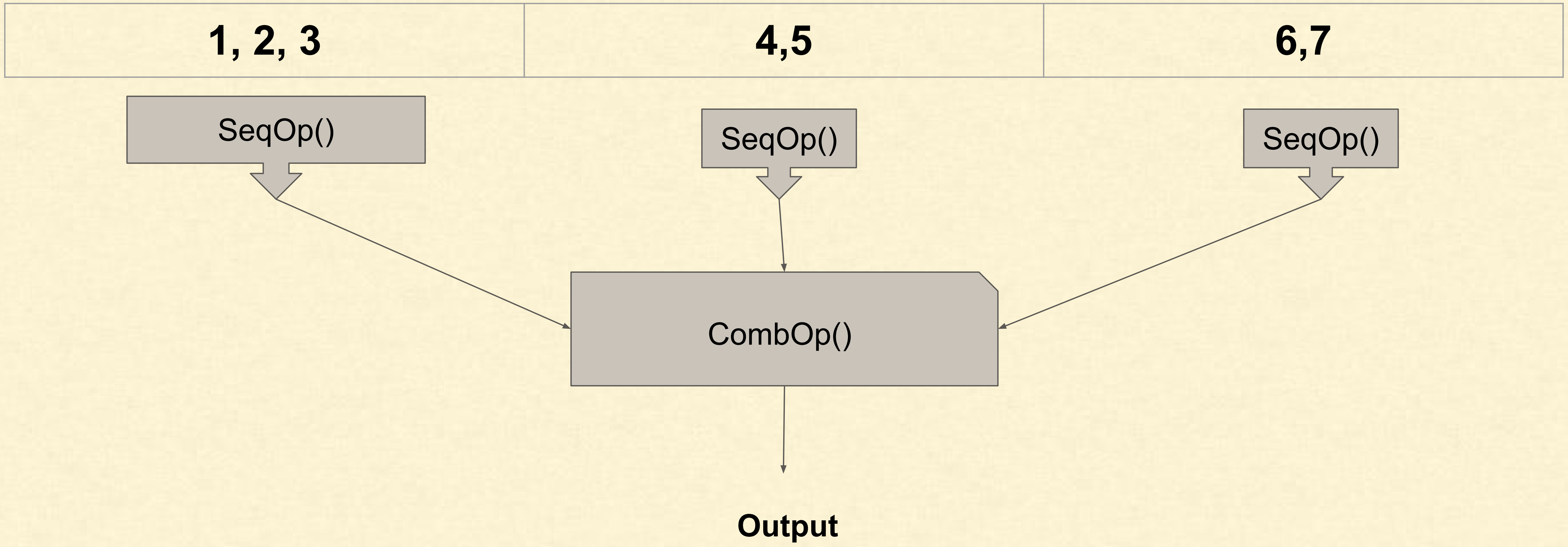
More Actions - fold()

fold(initial value, func)	Example: Concatnating to _

```
var myrdd = sc.parallelize(1 to 10, 2)
var myrdd1 = myrdd.map(_.toString)

def concat(s:String, n:String):String = s + n

var s = "_"
myrdd1.fold(s)(concat)

res1: String = _ _12345 _678910
```

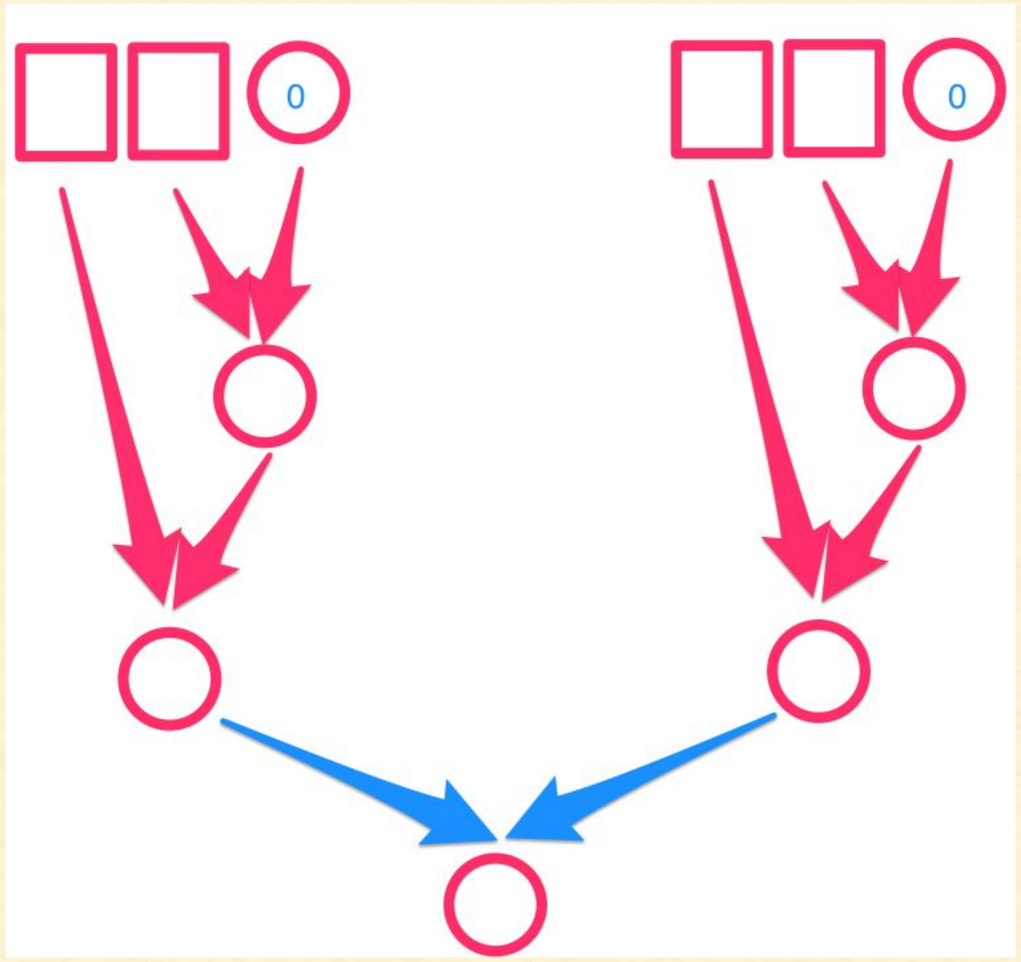
More Actions - aggregate()

aggregate(initial value) (seqOp, combOp)	1. First, all values of each partitions are merged to Initial value using SeqOp() 2. Second, all partitions result is combined together using combOp 3. Used specially when the output is different data type



More Actions - aggregate()

aggregate(initial value) (seqOp, combOp)	1. First, all values of each partitions are merged to Initial value using SeqOp() 2. Second, all partitions result is combined together using combOp 3. Used specially when the output is different data type



More Actions - aggregate()

aggregate(initial value) (seqOp, combOp)	1. First, all values of each partitions are merged to Initial value using SeqOp() 2. Second, all partitions result is combined together using combOp 3. Used specially when the output is different data type

```
var rdd = sc.parallelize(1 to 100)
```

More Actions - aggregate()

aggregate(initial value) (seqOp, combOp)	1. First, all values of each partitions are merged to Initial value using SeqOp() 2. Second, all partitions result is combined together using combOp 3. Used specially when the output is different data type

```
var rdd = sc.parallelize(1 to 100)

var init = (0, 0) // sum, count
```

More Actions - aggregate()

aggregate(initial value) (seqOp, combOp)	1. First, all values of each partitions are merged to Initial value using SeqOp() 2. Second, all partitions result is combined together using combOp 3. Used specially when the output is different data type

```
var rdd = sc.parallelize(1 to 100)

var init = (0, 0) // sum, count
def seq(t:(Int, Int), i:Int): (Int, Int) = (t._1 + i, t._2 + 1)
```

More Actions - aggregate()

aggregate(initial value) (seqOp, combOp)	1. First, all values of each partitions are merged to Initial value using SeqOp() 2. Second, all partitions result is combined together using combOp 3. Used specially when the output is different data type

```
var rdd = sc.parallelize(1 to 100)

var init = (0, 0) // sum, count
def seq(t:(Int, Int), i:Int): (Int, Int) = (t._1 + i, t._2 + 1)
def comb(t1:(Int, Int), t2:(Int, Int)): (Int, Int) = (t1._1 + t2._1, t1._2 + t2._2)

var d = rdd.aggregate(init)(seq, comb)
```

res6: (Int, Int) = (5050,100)

More Actions - aggregate()

aggregate(initial value)(seqOp, combOp)	1. First, all values of each partitions are merged to Initial value using SeqOp() 2. Second, all partitions result is combined together using combOp 3. Used specially when the output is different data type

```
var rdd = sc.parallelize(1 to 100)

var init = (0, 0) // sum, count
def seq(t:(Int, Int), i:Int): (Int, Int) = (t._1 + i, t._2 + 1)
def comb(t1:(Int, Int), t2:(Int, Int)): (Int, Int) = (t1._1 + t2._1, t1._2 + t2._2)

var d = rdd.aggregate(init)(seq, comb)
```

```
res6: (Int, Int) = (5050,100)
```

More Actions: *countByValue()*

Number of times each element occurs in the RDD.

1	2	3	3	5	5	5

```
var rdd = sc.parallelize(List(1, 2, 3, 3, 5, 5, 5))
var  dict = rdd.countByValue()
dict
```

```
Map(1 -> 1, 5 -> 3, 2 -> 1, 3 -> 2)
```

More Actions: *top(n)*

Sorts and gets the maximum n values.

4	4	8	1	2	3	10	9

```
var a=sc.parallelize(List(4,4,8,1,2, 3, 10, 9))
a.top(6)
```

Array(10, 9, 8, 4, 4, 3)

More Actions: *takordered()*

Get the N elements from an RDD ordered in ascending order or as specified by the optional key function.

```
sc.parallelize(List(10, 1, 2, 9, 3, 4, 5, 6, 7)).takeOrdered(6)


var l = List((10, "SG"), (1, "AS"), (2, "AB"), (9, "AA"), (3, "SS"), (4, "RG"), (5, "AU"), (6, "DD"), (7, "ZZ"))
var r = sc.parallelize(l)
r.takeOrdered(6)(Ordering[Int].reverse.on(x => x._1))
(10,SG), (9,AA), (7,ZZ), (6,DD), (5,AU), (4,RG)

r.takeOrdered(6)(Ordering[String].reverse.on(x => x._2))
(7,ZZ), (3,SS), (10,SG), (4,RG), (6,DD), (5,AU)

r.takeOrdered(6)(Ordering[String].on(x => x._2))
(9,AA), (2,AB), (1,AS), (5,AU), (6,DD), (4,RG)
```

More Actions: *foreach()*

Applies a function to all elements of this RDD.

```
>>> def f(x:Int)= println(s"Save $x to DB")
>>> sc.parallelize(1 to 5).foreach(f)

Save 2 to DB
Save 1 to DB
Save 4 to DB
Save 5 to DB
```

More Actions: *foreach()*

Differences from map()

1. Use foreach if you don't expect any result. For example saving to database.
2. Foreach is an action. Map is transformation

More Actions: *foreachPartition(f)*

Applies a function to each partition of this RDD.

More Actions: *foreachPartition(f)*

Applies a function to each partition of this RDD.

```
def partitionSum(itr: Iterator[Int]) =
    println("The sum of the parition is " + itr.sum.toString)
```

More Actions: *foreachPartition(f)*

Applies a function to each partition of this RDD.

```
def partitionSum(itr: Iterator[Int]) =
    println("The sum of the parition is " + itr.sum.toString)
sc.parallelize(1 to 40, 4).foreachPartition(partitionSum)
```

```
The sum of the parition is 155
The sum of the parition is 55
The sum of the parition is 355
The sum of the parition is 255
```

Basics of RDD

Thank you!