# APACHE Spark™

Basics of RDD

## RDDs - Resilient Distributed Datasets

# What is RDD?

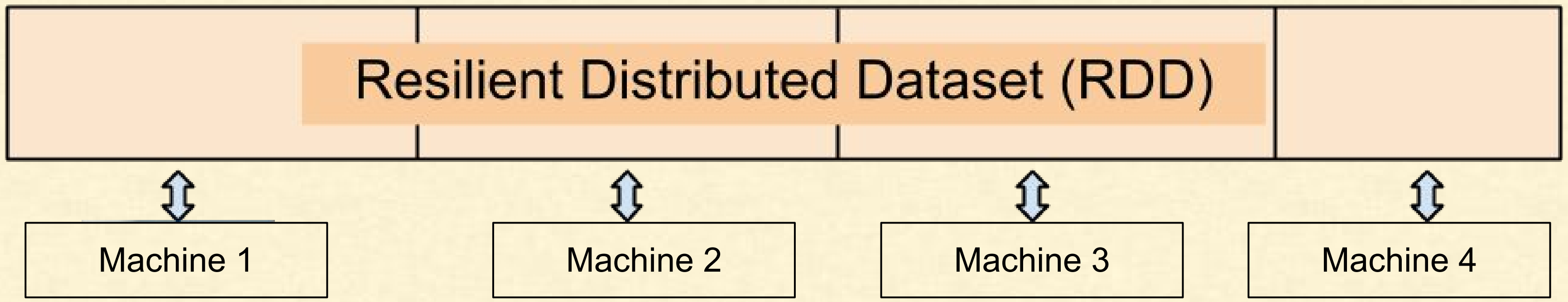
**Dataset:**
*Collection of data elements.*
*e.g. Array, Tables, Data frame (R), collections of mongodb*

**Distributed:**
*Parts Multiple machines*

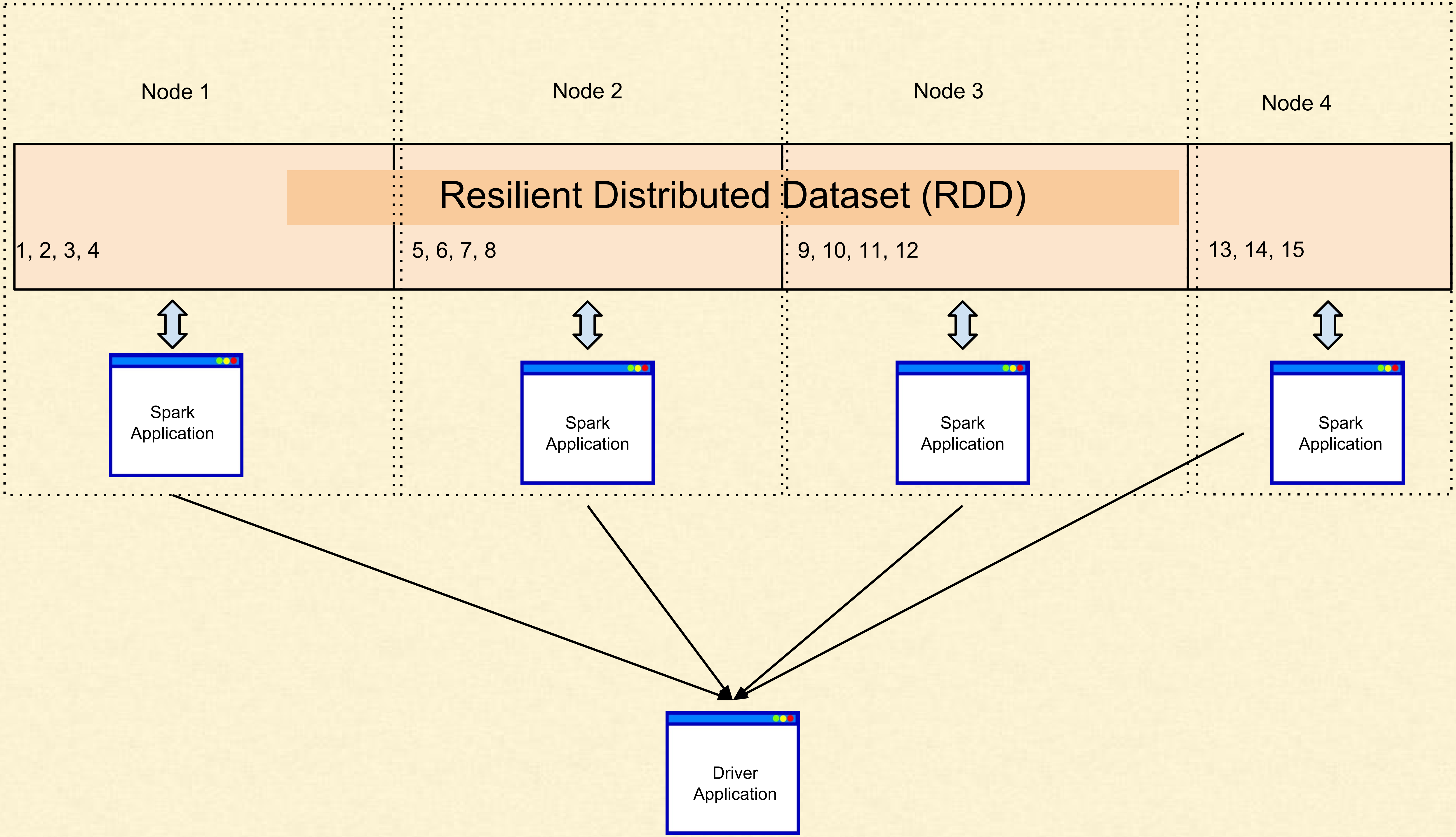
**Resilient:**
*Recovers on Failure*

# SPARK - CONCEPTS - RESILIENT DISTRIBUTED DATASET

## A collection of elements partitioned across cluster



# SPARK - CONCEPTS - RESILIENT DISTRIBUTED DATASET



# SPARK - CONCEPTS - RESILIENT DISTRIBUTED DATASET

## A collection of elements partitioned across cluster

- An immutable distributed collection of objects.
- Split in partitions which may be on multiple nodes
- Can contain any data type:
  - Python,
  - Java,
  - Scala objects
  - including user defined classes

# SPARK - CONCEPTS - RESILIENT DISTRIBUTED DATASET

- RDD Can be persisted in memory
- RDD Auto recover from node failures
- Can have any data type but has a special dataset type for key-value
- Supports two type of operations:
  - ○ Transformation
  - ○ Action

# Creating RDD - Scala

**Method 1: By Directly Loading a file from remote**

```
>>var lines = sc.textFile("/data/mr/wordcount/input/big.txt")
```

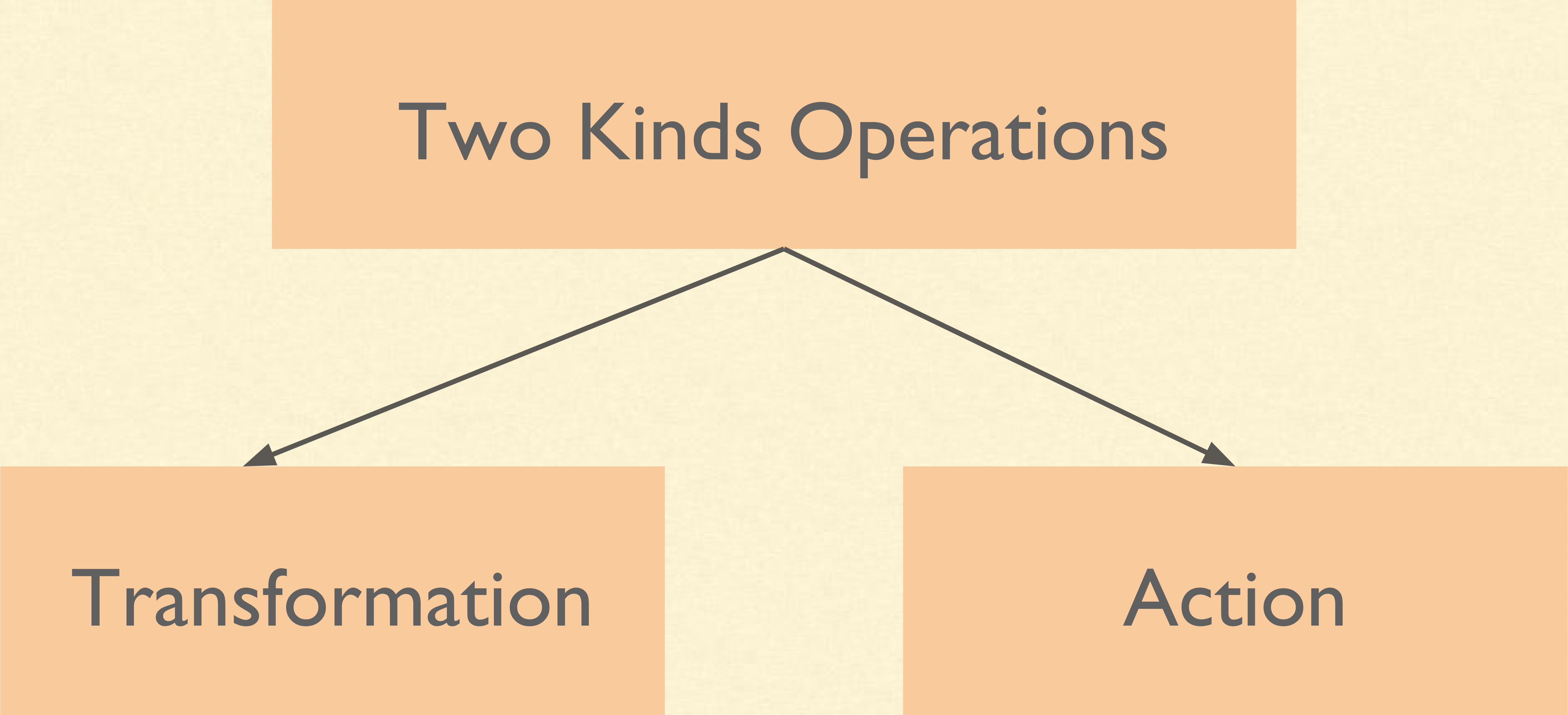
**Method 2: By distributing existing object**

```
>> val arr = 1 to 10000
>> var nums = sc.parallelize(arr)
```

## WordCount - Scala

```
var linesRdd = sc.textFile("/data/mr/wordcount/input/big.txt")
var words = linesRdd.flatMap(x => x.split(" "))
var wordsKv = words.map(x => (x, 1))
var output = wordsKv.reduceByKey(_ + _)
output.take(10)
or
output.saveAsTextFile("my_result")
```

# RDD Operations

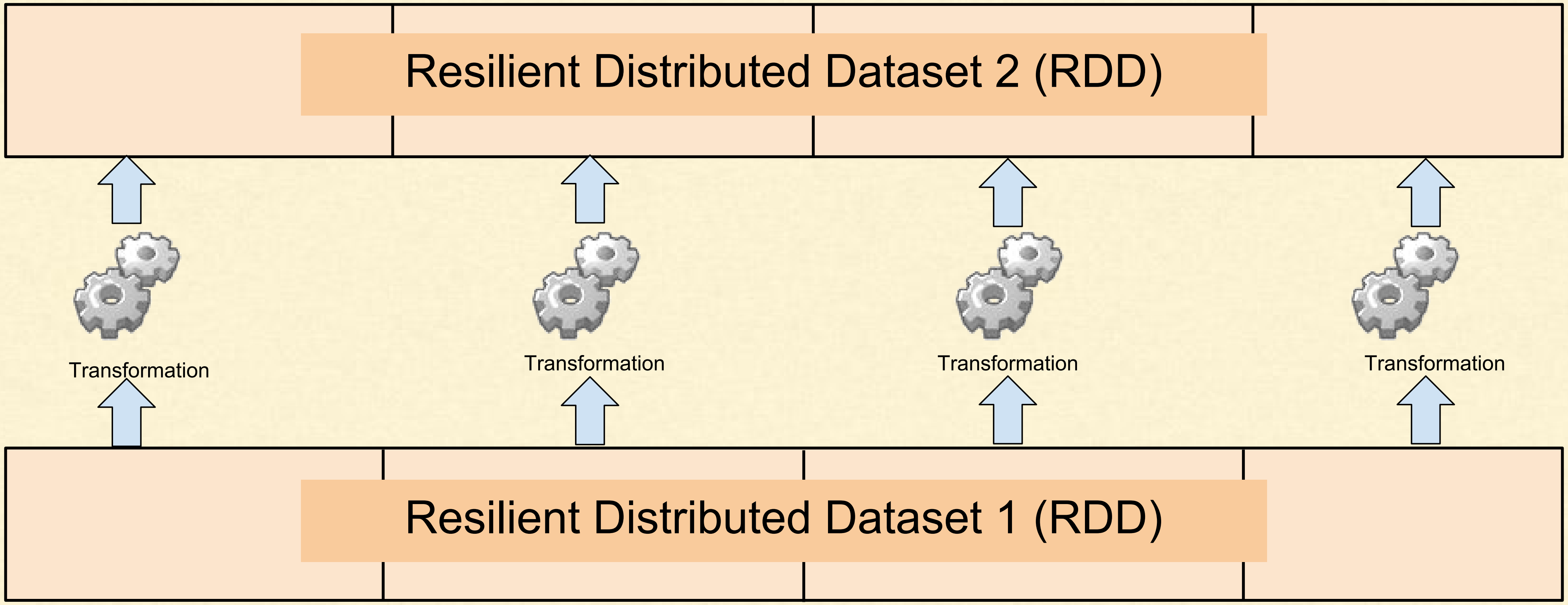


Two Kinds Operations

Transformation

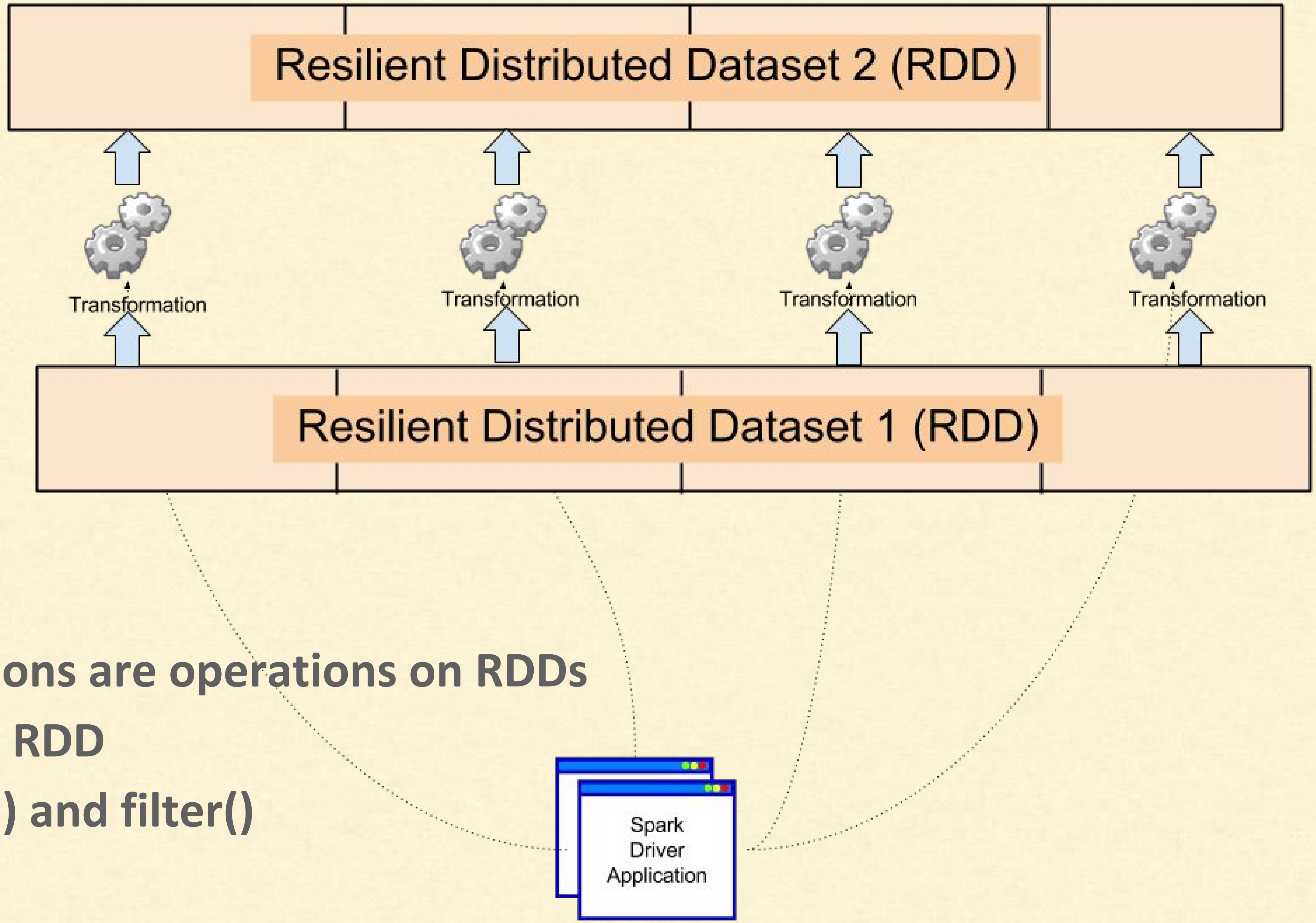
Action

# RDD - Operations : Transformation


Resilient Distributed Dataset 2 (RDD)
Transformation
Transformation
Transformation
Transformation
Resilient Distributed Dataset 1 (RDD)

- **Transformations are operations on RDDs**
- **return a new RDD**
- **such as map() and filter()**

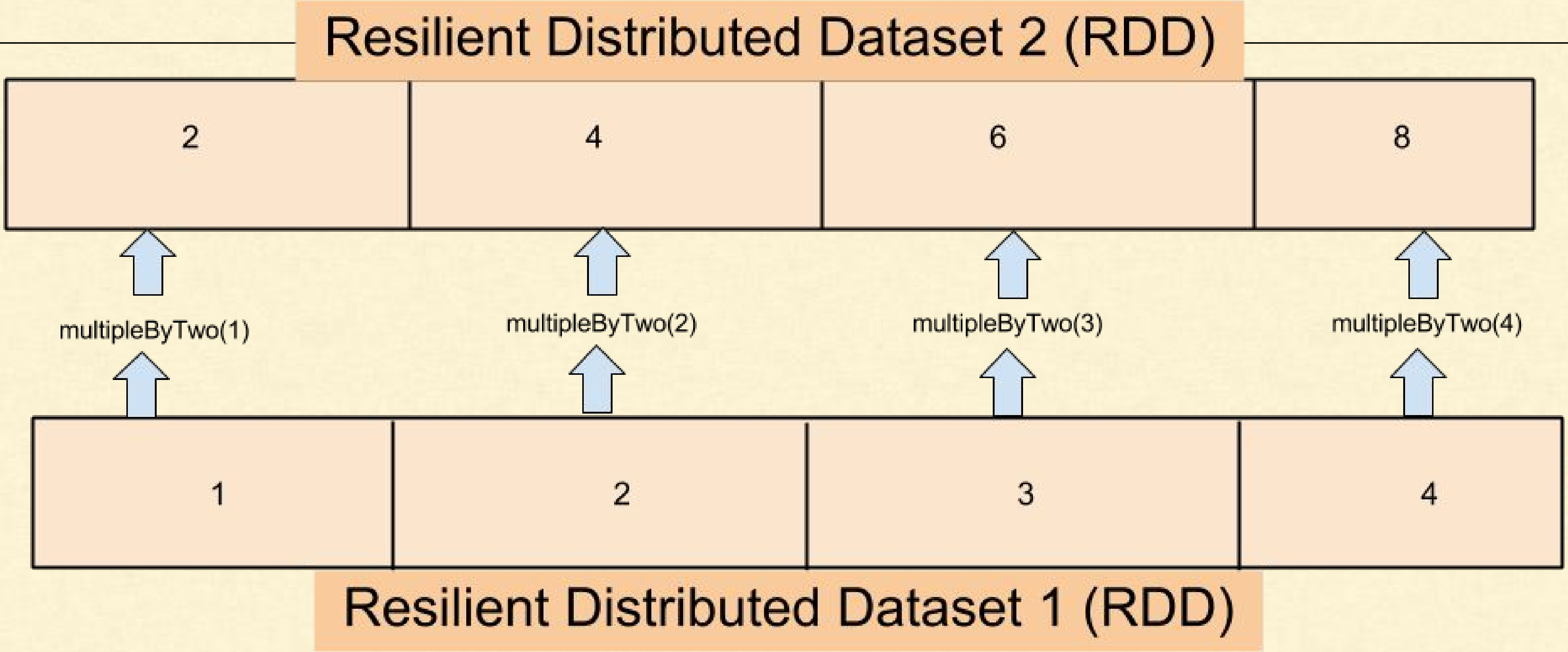
# RDD - Operations : Transformation



- **Transformations are operations on RDDs**
- **return a new RDD**
- **such as map() and filter()**

# Map Transformation

- Map is a transformation
- That runs provided function against each element of RDD
- And creates a new RDD from the results of execution function


Resilient Distributed Dataset 2 (RDD)
2
4
6
8
multipleByTwo(1)
multipleByTwo(2)
multipleByTwo(3)
multipleByTwo(4)
1
2
3
4
Resilient Distributed Dataset 1 (RDD)

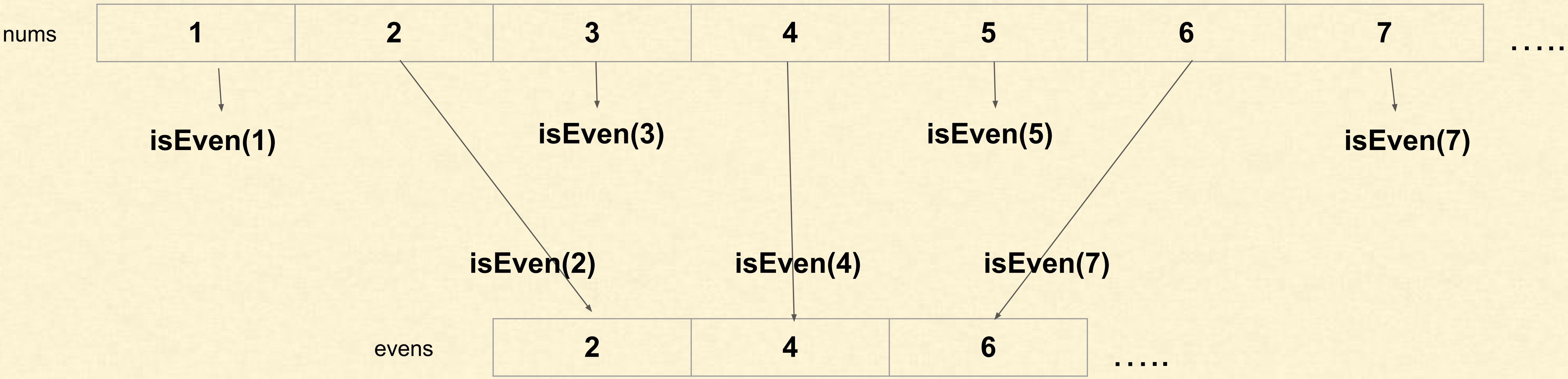
# Map Transformation - Scala

```
➢  val arr = 1 to 10000
➢  val nums = sc.parallelize(arr)
➢  def multiplyByTwo(x:Int):Int = x*2
➢  multiplyByTwo(5)
   10
➢  var dbls = nums.map(multiplyByTwo);
➢  dbls.take(5)
   [2, 4, 6, 8, 10]
```

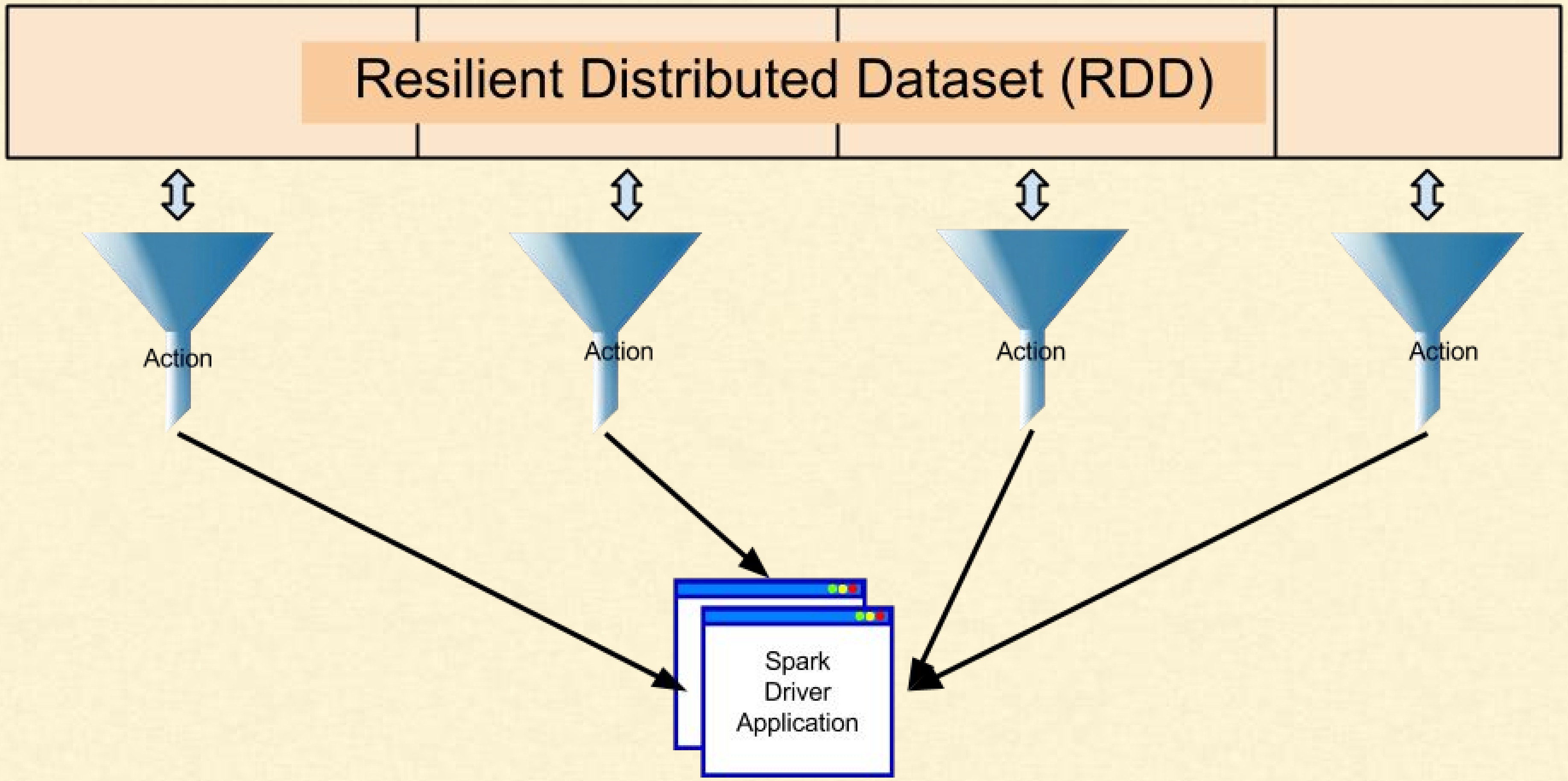
# Transformations - filter() - scala

- var arr = 1 to 1000
- var nums = sc.parallelize(arr)
- def isEven(x:Int):Boolean = x%2 == 0

- *var evens = nums.filter(isEven)*
- *evens*.take(3)
- [2, 4, 6]



# RDD - Operations : Actions

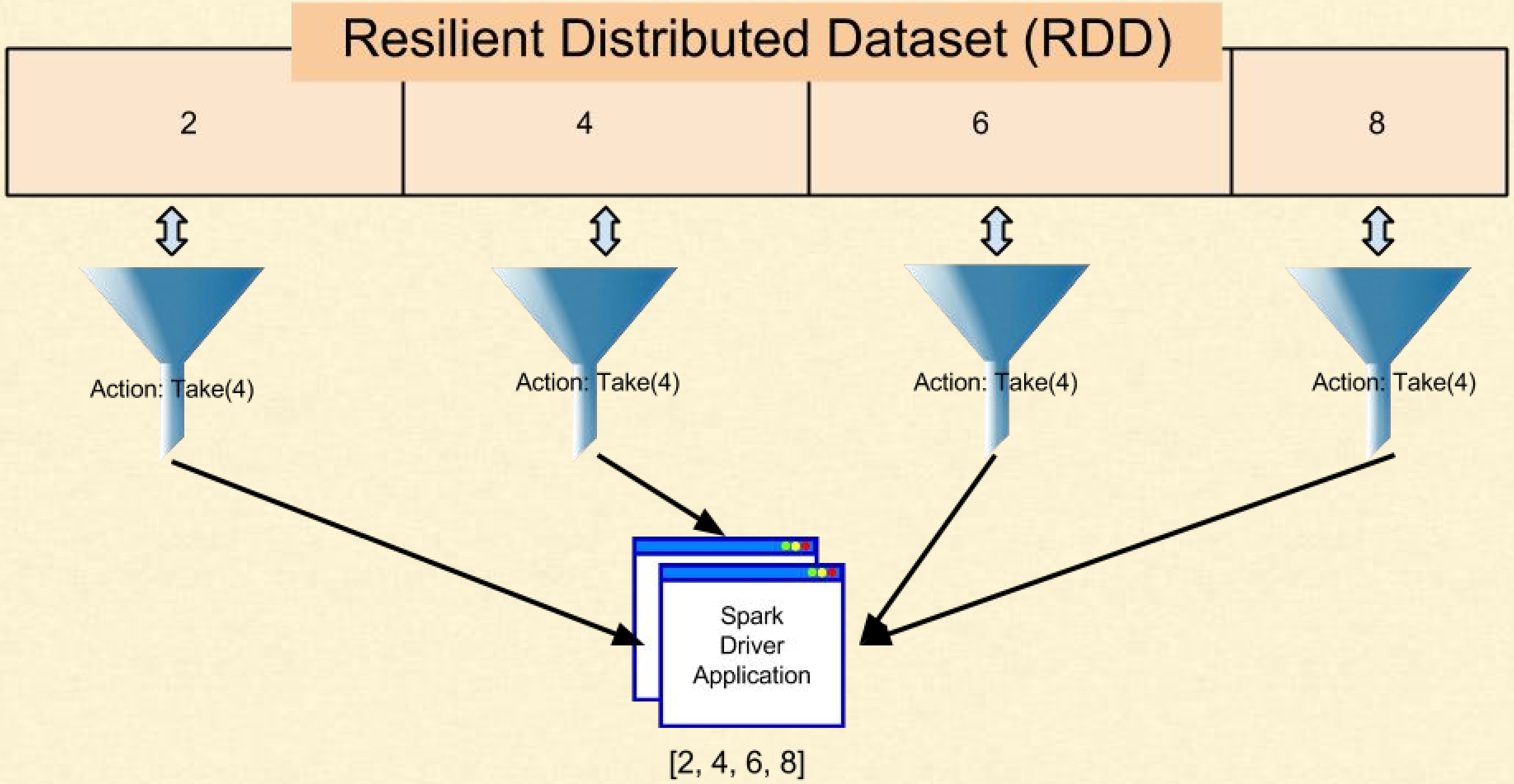


- **Causes the full execution of transformations**
- **Involves both spark driver as well as the nodes**
- **Example - Take(): Brings back the data to driver**

# Action Example - take()

- ➢ val arr = 1 to 1000000
- ➢ val nums = sc.parallelize(arr)
- ➢ def multipleByTwo(x:Int):Int = x*2

- ➢ *var dbls = nums.map(multipleByTwo);*
- ➢ **dbls.take(5)**
- ➢ [2, 4, 6, 8, 10]



# Action Example - saveAsTextFile()

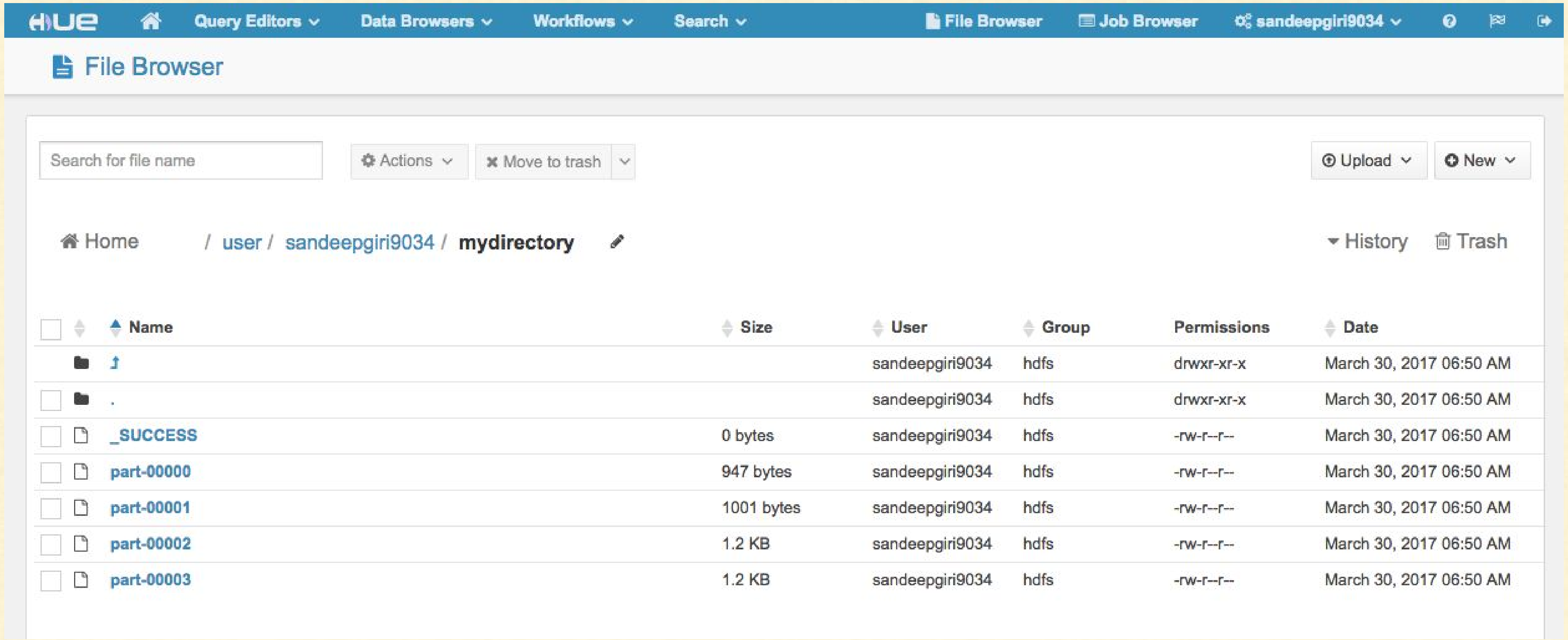
To save the results in HDFS or Any other file system

Call **saveAsTextFile(directoryName)**

It would create directory

And save the results inside it

If directory exists, it would throw error.

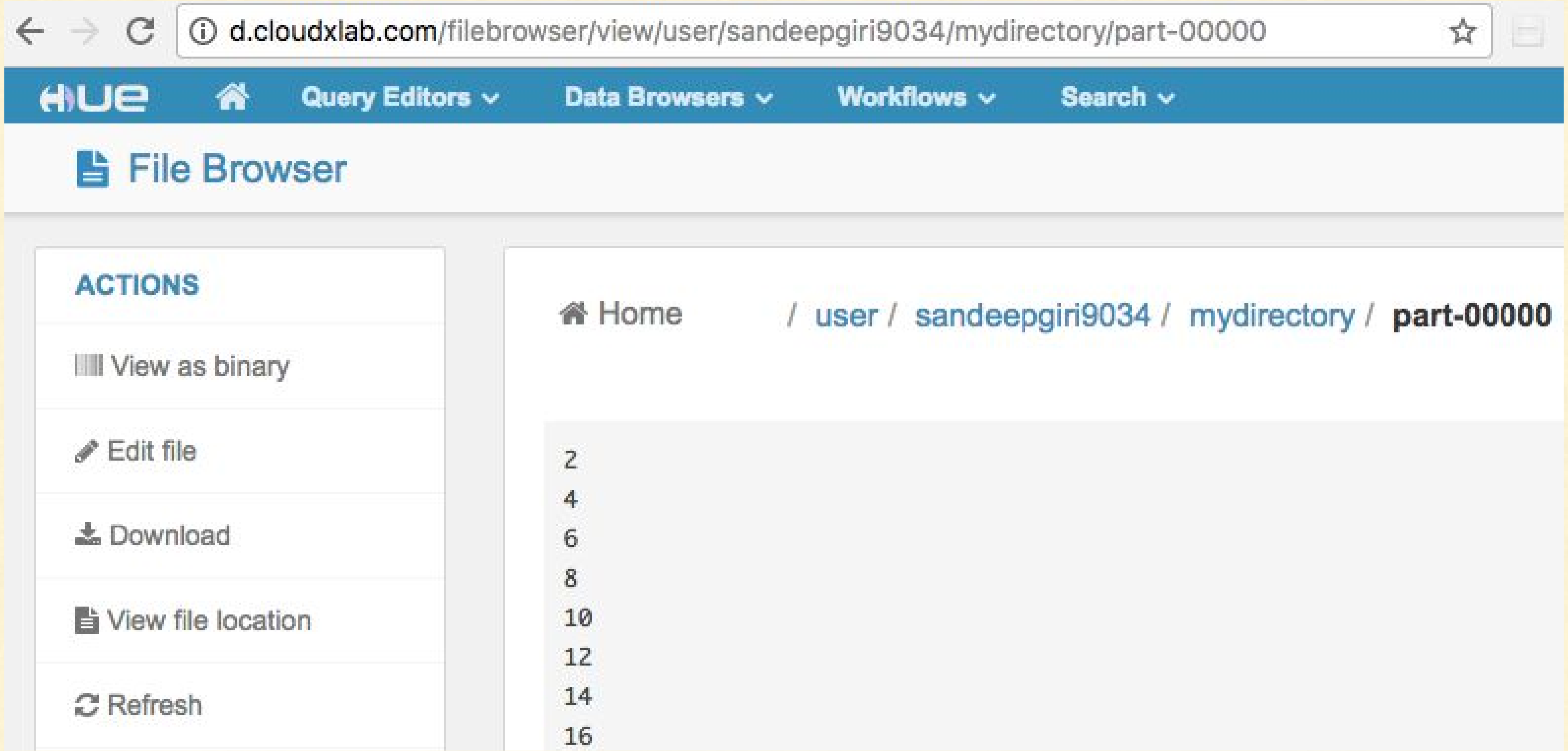


Home / user / sandeepgiri9034 / mydirectory

| Name | Size | User | Group | Permissions | Date |
|---|---|---|---|---|---|
| ↥ | | sandeepgiri9034 | hdfs | drwxr-xr-x | March 30, 2017 06:50 AM |
| . | | sandeepgiri9034 | hdfs | drwxr-xr-x | March 30, 2017 06:50 AM |
| _SUCCESS | 0 bytes | sandeepgiri9034 | hdfs | -rw-r--r-- | March 30, 2017 06:50 AM |
| part-00000 | 947 bytes | sandeepgiri9034 | hdfs | -rw-r--r-- | March 30, 2017 06:50 AM |
| part-00001 | 1001 bytes | sandeepgiri9034 | hdfs | -rw-r--r-- | March 30, 2017 06:50 AM |
| part-00002 | 1.2 KB | sandeepgiri9034 | hdfs | -rw-r--r-- | March 30, 2017 06:50 AM |
| part-00003 | 1.2 KB | sandeepgiri9034 | hdfs | -rw-r--r-- | March 30, 2017 06:50 AM |

# Action Example - saveAsTextFile()

| | |
|---|---|
| val arr = 1 to 1000<br>val nums = sc.parallelize(arr)<br>def multipleByTwo(x:Int):Int = x*2 | *var dbls = nums.map(multipleByTwo);*<br>**dbls.saveAsTextFile("mydirectory")**<br>Check the HDFS home directory |


d.cloudxlab.com/filebrowser/view/user/sandeepgiri9034/mydirectory/part-00000
HUE
Query Editors
Data Browsers
Workflows
Search
File Browser
ACTIONS
View as binary
Edit file
Download
View file location
Refresh
Home
/ user / sandeepgiri9034 / mydirectory / part-00000

```
2
4
6
8
10
12
14
16
```

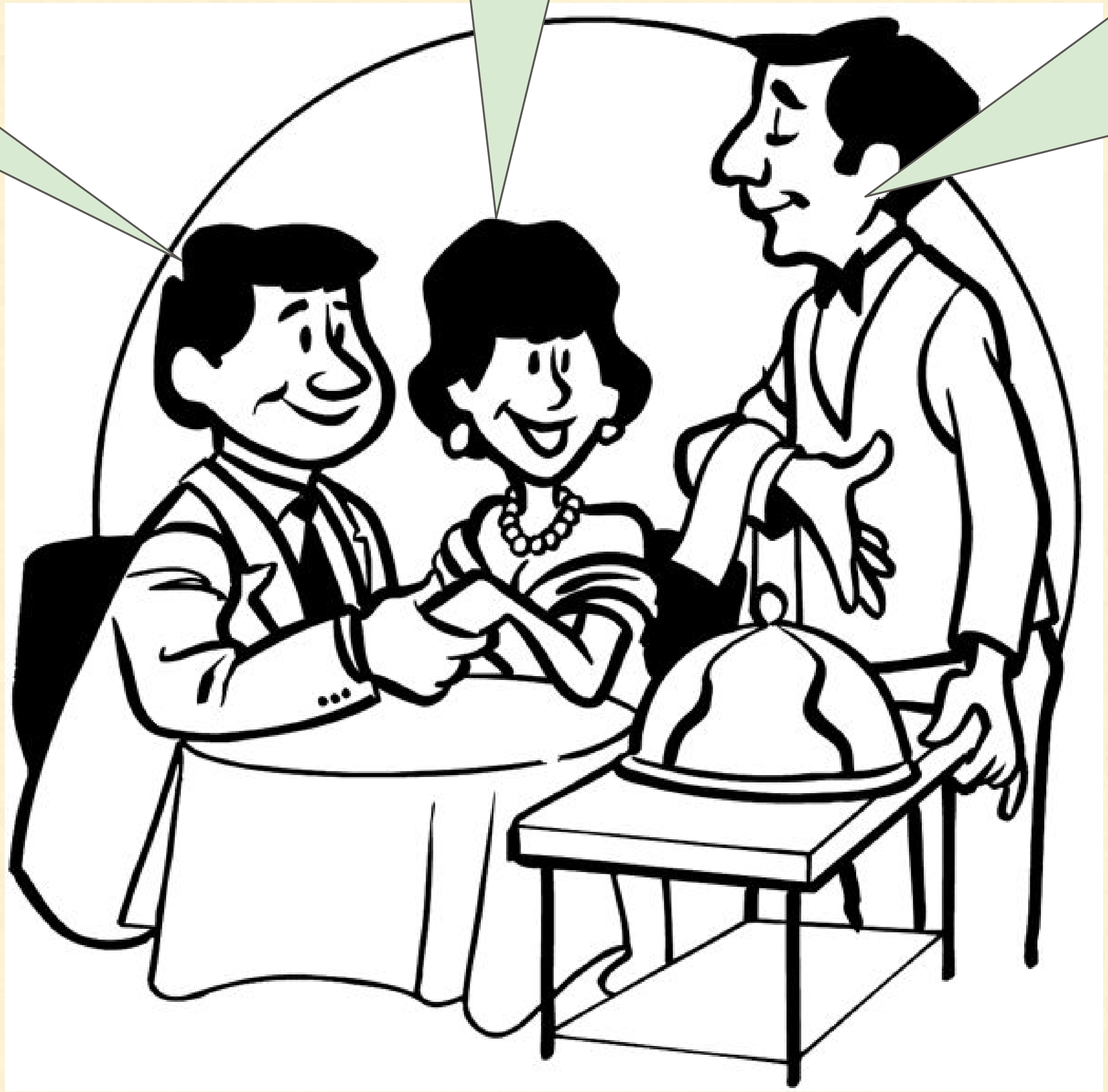
# RDD Operations

| | **Transformation** | **Action** |
|---|---|---|
| Examples | map() | take() |
| Returns | Another RDD | Local value |
| Executes | Lazily | Immediately. Executes transformations |

# Lazy Evaluation Example - The waiter takes orders patiently

Cheese burger,
soup and
a Plate of Noodles
please

Soup and
A Plate of
Noodles for
me



Ok.
One cheese burger
Two soups
Two plates of Noodles
Anything else, sir?

The chef is able to optimize because of clubbing multiple order together

# Instant Evaluation

Cheese Burger...

Let me get a cheese burger for you. I'll be right back!

And Soup?

The soup order will be taken once the waiter is back.

# Instant Evaluation

The usual programing languages have instant evaluation.

As you as you type:

***var x = 2+10.***

It doesn't wait. It immediately evaluates.

# Actions: Lazy Evaluation

1. Every time we call an action, entire RDD must be computed from scratch
2. Everytime d gets executed, a,b,c would be run
    a. lines  = sc.textFile("myfile");
    b. fewlines = lines.filter(...)
    c. uppercaselines = fewlines.map(...)
    d. uppercaselines.count()
3. When we call a transformation, it is not evaluated immediately.
4. It helps Spark optimize the performance
5. Similar to Pig, tensorflow etc.
6. Instead of thinking  RDD as dataset, think of it as the instruction on how to compute data

# Actions: Lazy Evaluation - Optimization - Scala

```
def Map1(x:String):String =
x.trim();

def Map2(x:String):String =
x.toUpperCase();

var lines = sc.textFile(...)
var lines1 = lines.map(Map1);
var lines2 = lines1.map(Map2);

lines2.collect()
```

```
def Map(x:String):String={
    var y = x.trim();
    return y.toUpperCase();
}

lines = sc.textFile(...)
lines2 = lines.map(Map);

lines2.collect()
```

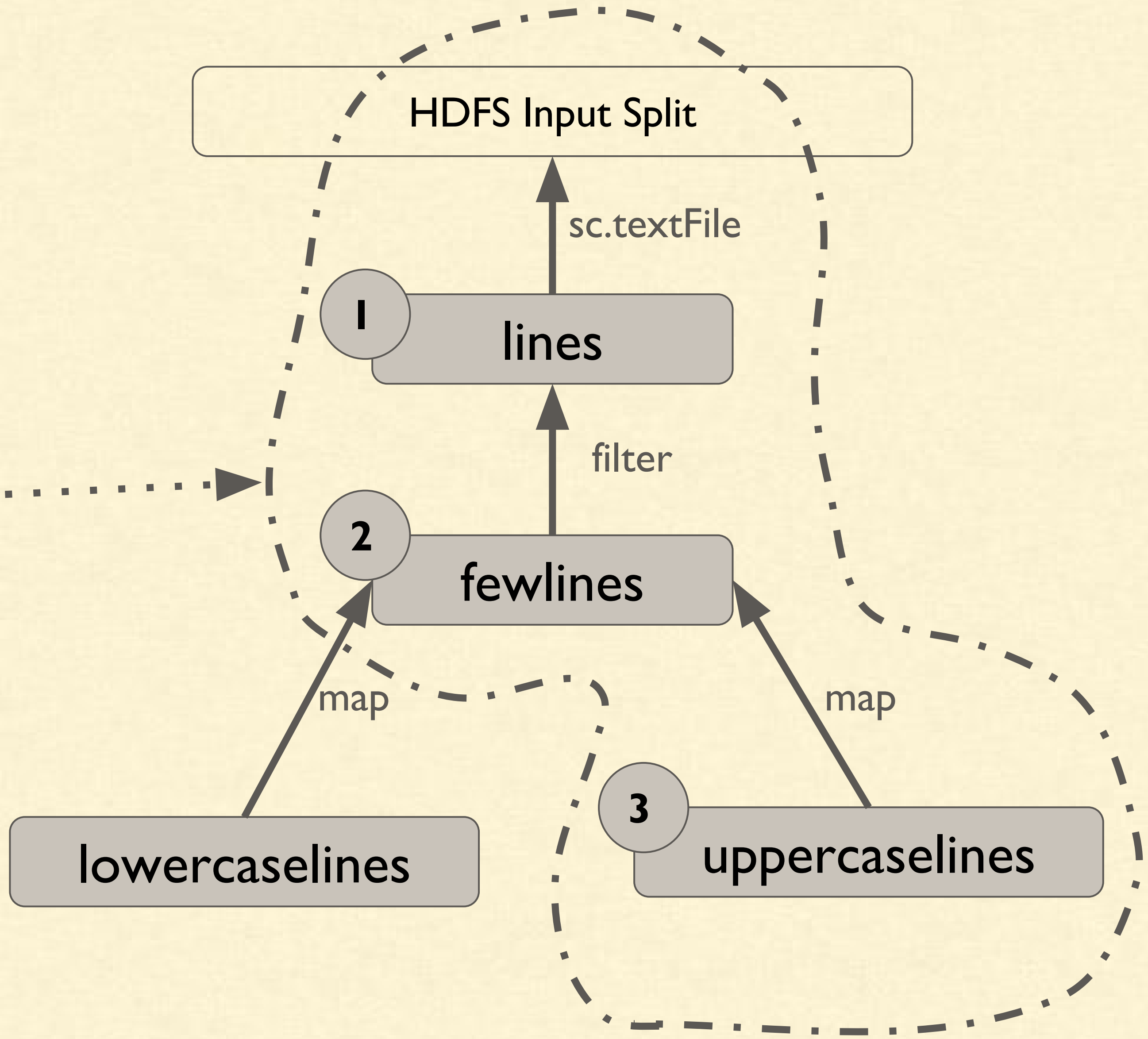
# Lineage Graph

Spark Code

```
lines  = sc.textFile("myfile");
fewlines = lines.filter(...)
uppercaselines = fewlines.map(...)
lowercaselines = fewlines.map(...)
```

**uppercaselines.count()**

Lineage Graph

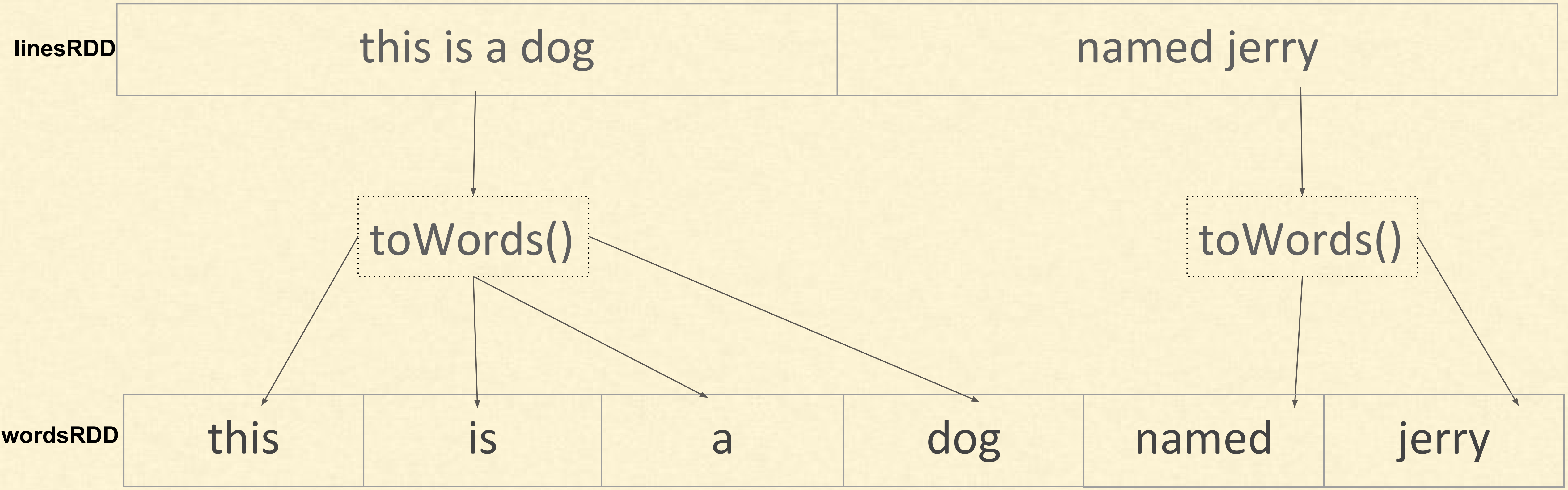


# Transformations:: flatMap() - Scala

To convert one record of an RDD into multiple records.

# Transformations:: flatMap() - Scala

- var linesRDD = sc.parallelize( Array("this is a dog", "named jerry"))
- def toWords(line:String):Array[String]= line.split(" ")
- var wordsRDD = linesRDD.flatMap(toWords)
- var wordsRDD1 = linesRDD.map(toWords)
- wordsRDD.collect()
- *['this', 'is', 'a', 'dog', 'named', 'jerry']*

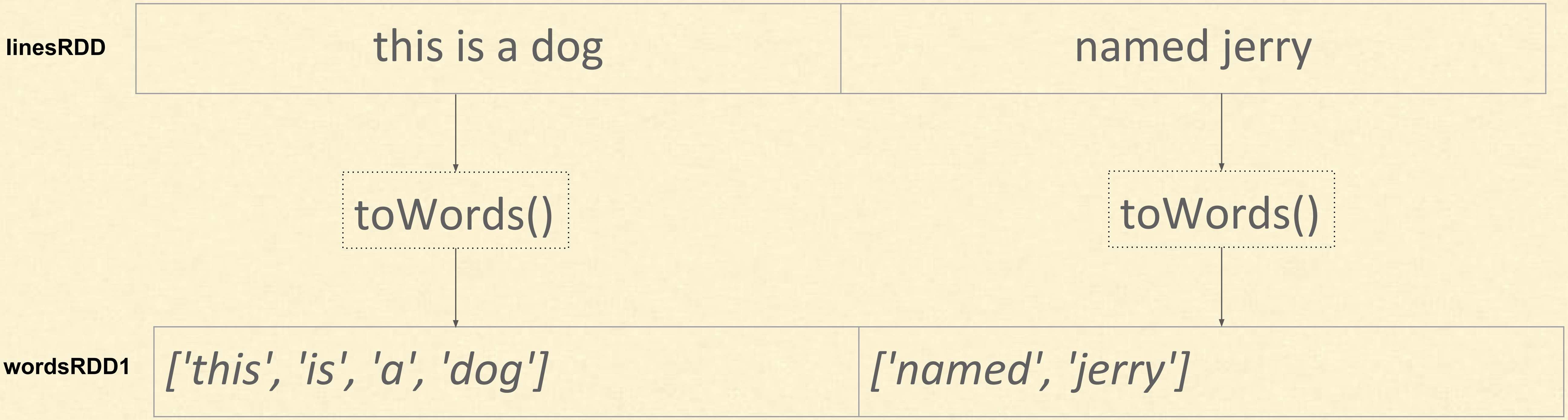


# How is it different from Map()?

- In case of map() the resulting rdd and input rdd having same number of elements.
- map() can only convert one to one while flatMap could convert one to many.

# What would happen if map() is used

```
➢ var linesRDD = sc.parallelize( Array("this is a dog", "named jerry"))
➢ def toWords(line:String):Array[String]= line.split(" ")
➢ var wordsRDD1 = linesRDD.map(toWords)
➢ wordsRDD1.collect()
➢ [['this', 'is', 'a', 'dog'], ['named', 'jerry']]
```



linesRDD

this is a dog | named jerry

toWords() | toWords()

wordsRDD1

*['this', 'is', 'a', 'dog']* | *['named', 'jerry']*

# FlatMap

- Very similar to Hadoop's Map()
- Can give out 0 or more records

# FlatMap

- Can emulate map as well as filter
- Can produce many as well as no value which empty array as output
  - If it give out single value, it behaves like map().
  - If it gives out empty array, it behaves like filter.

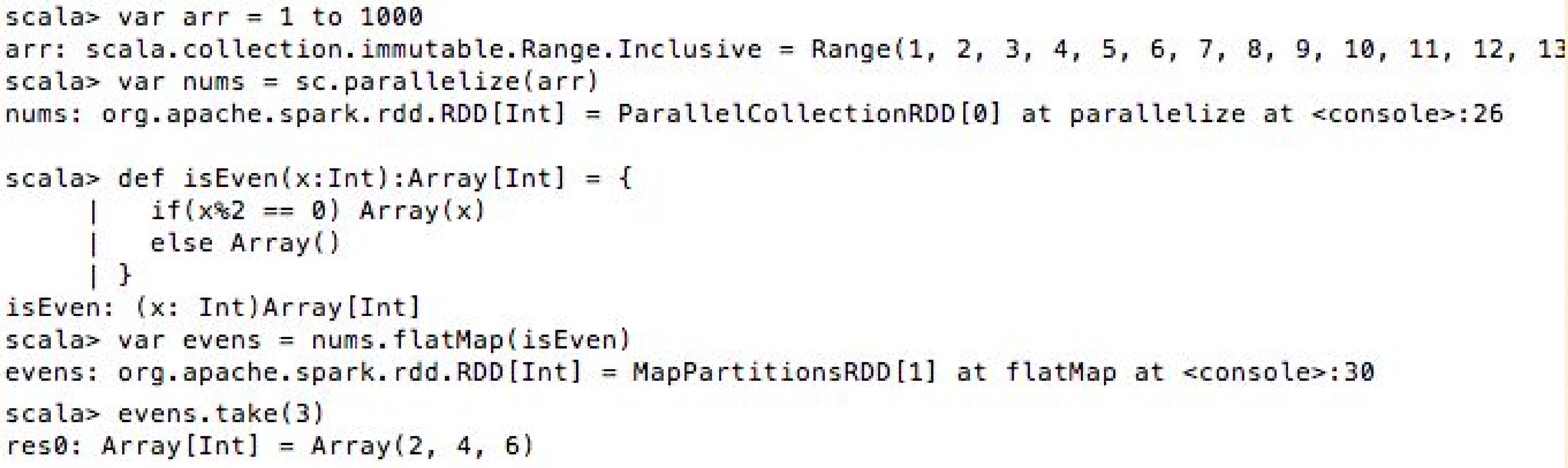
# flatMap as map

```
➢ val arr = 1 to 10000
➢ val nums = sc.parallelize(arr)
➢ def multiplyByTwo(x:Int) = Array(x*2)
➢ multiplyByTwo(5)
  Array(10)
➢ var dbls = nums.map(multiplyByTwo);
➢ dbls.take(5)
  [2, 4, 6, 8, 10]
```

# flatMap as filter

- var arr = 1 to 1000
- var nums = sc.parallelize(arr)
- def isEven(x:Int):Array[Int] = {
-   if(x%2 == 0) Array(x)
-   else Array()
- }

- ***var evens = nums.flatMap(isEven)***
- ***evens***.take(3)
- [2, 4, 6]

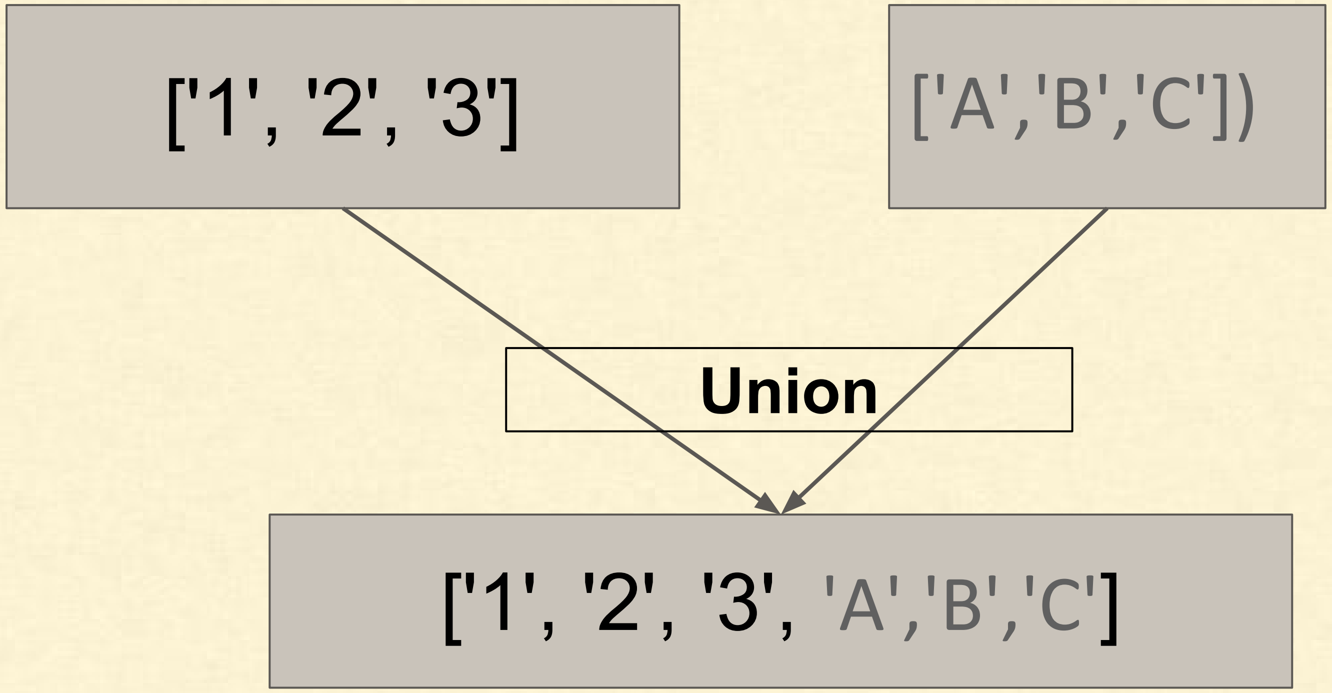


```
scala> var arr = 1 to 1000
arr: scala.collection.immutable.Range.Inclusive = Range(1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13
scala> var nums = sc.parallelize(arr)
nums: org.apache.spark.rdd.RDD[Int] = ParallelCollectionRDD[0] at parallelize at <console>:26

scala> def isEven(x:Int):Array[Int] = {
     |   if(x%2 == 0) Array(x)
     |   else Array()
     | }
isEven: (x: Int)Array[Int]
scala> var evens = nums.flatMap(isEven)
evens: org.apache.spark.rdd.RDD[Int] = MapPartitionsRDD[1] at flatMap at <console>:30

scala> evens.take(3)
res0: Array[Int] = Array(2, 4, 6)
```

# Transformations:: Union

- var a = sc.parallelize(Array('1','2','3'));
- var b = sc.parallelize(Array('A','B','C'));
- ***var c=a.union(b)***
- ***Note: doesn't remove duplicates***
- c.collect();

  *[1, 2, 3, 'A', 'B', 'C']*



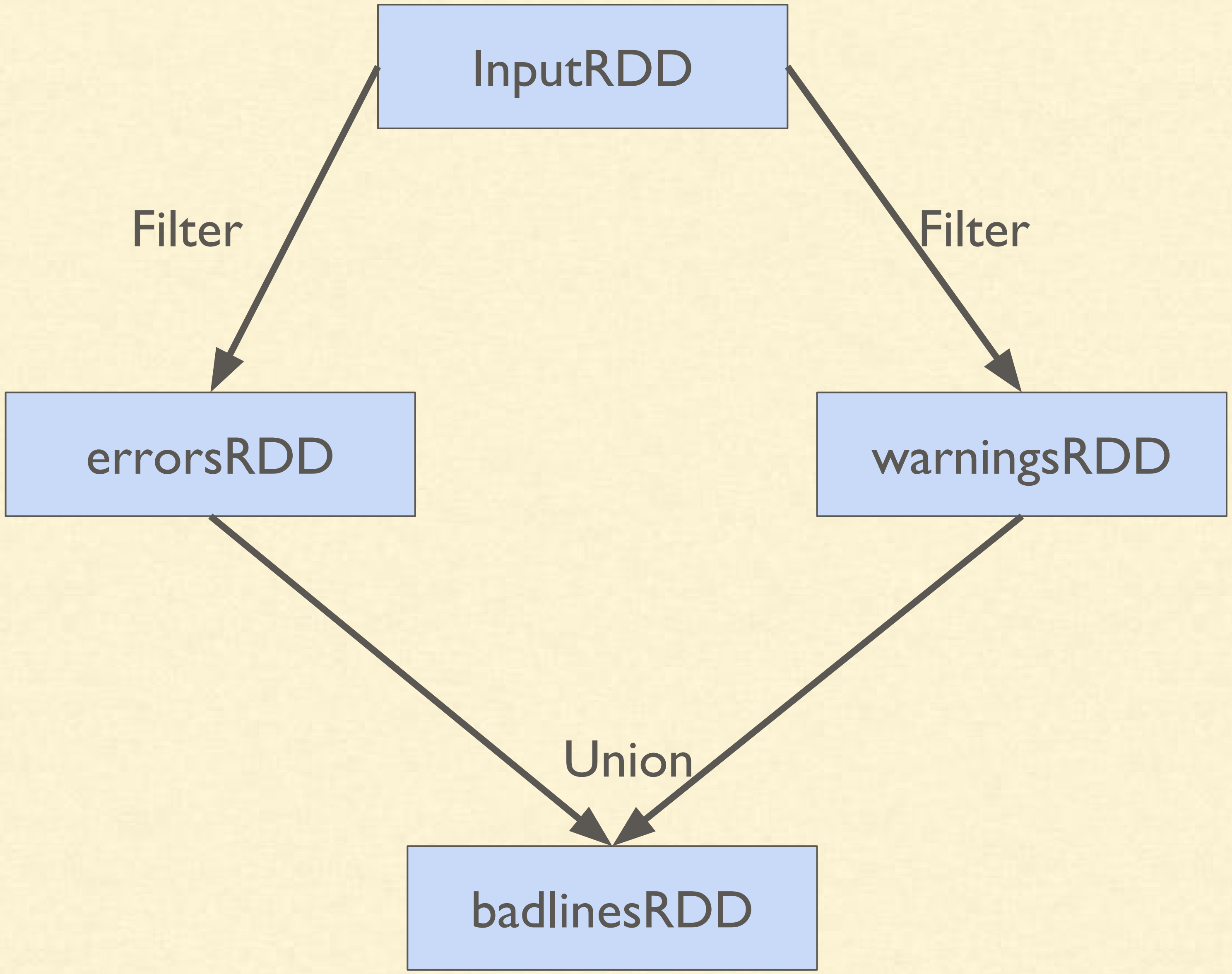
['1', '2', '3']

['A','B','C'])

Union

['1', '2', '3', 'A','B','C']

# Transformations:: union()



RDD lineage graph created during log analysis

# Actions: saveAsTextFile() - Scala

Saves all the elements into HDFS as text files.

- var a = sc.parallelize(Array(1,2,3, 4, 5 , 6, 7));
- ***a.saveAsTextFile("myresult");***
- ***// Check the HDFS.***
- ***//There should myresult folder in your home directory.***

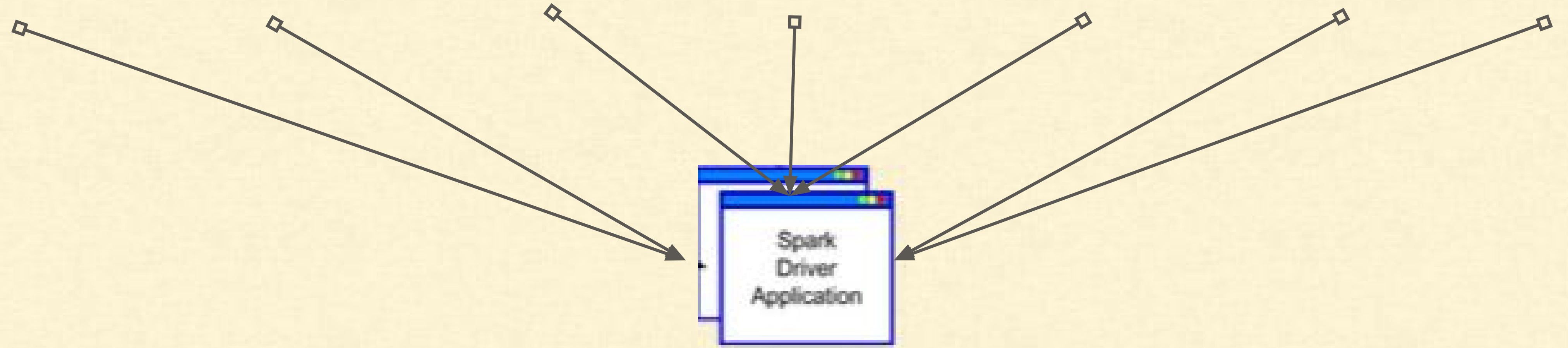
# Actions: collect() - Scala

Brings all the elements back to you. Data must fit into memory.

Mostly it is impractical.

```
➢ var a = sc.parallelize(Array(1,2,3, 4, 5 , 6, 7));
➢ a
  org.apache.spark.rdd.RDD[Int] = ParallelCollectionRDD[16] at parallelize at <console>:21
➢ var localarray =  a.collect();
➢ localarray
  [1, 2, 3, 4, 5, 6, 7]
```

| | | | | | | |
|---|---|---|---|---|---|---|
| 1 | 2 | 3 | 4 | 5 | 6 | 7 |

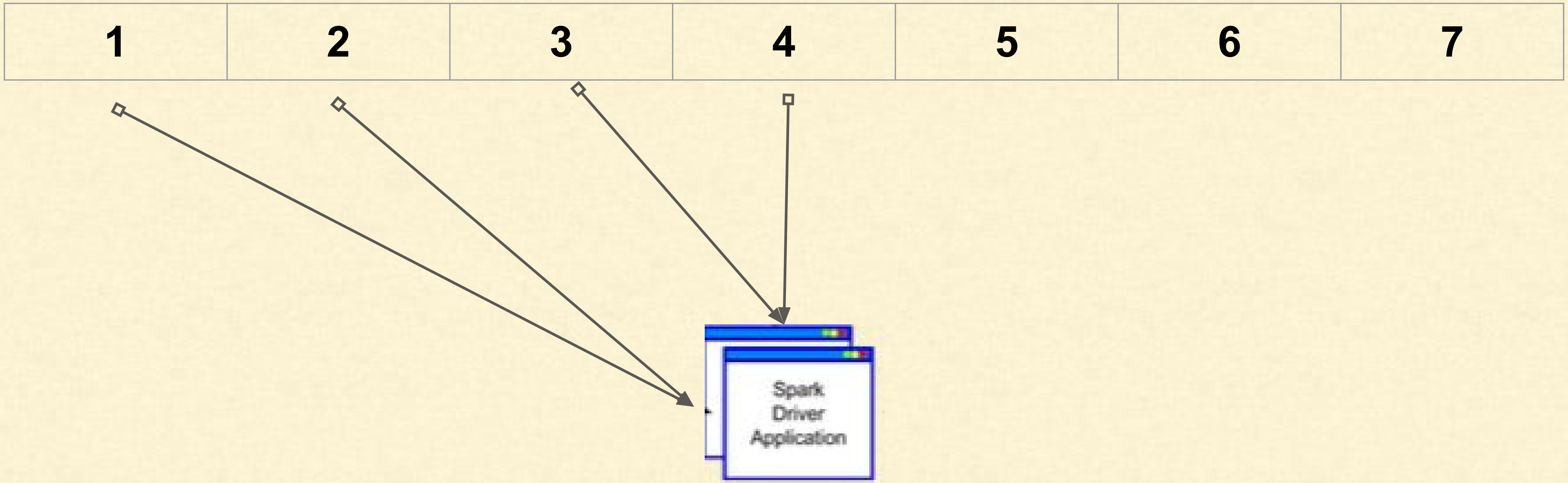


# Actions: take() - Scala

Bring only few elements to the driver.

This is more practical than collect()

```
➢ var a = sc.parallelize(Array(1,2,3, 4, 5 , 6, 7));
➢ var localarray =  a.take(4);
➢ localarray
  [1, 2, 3, 4]
```



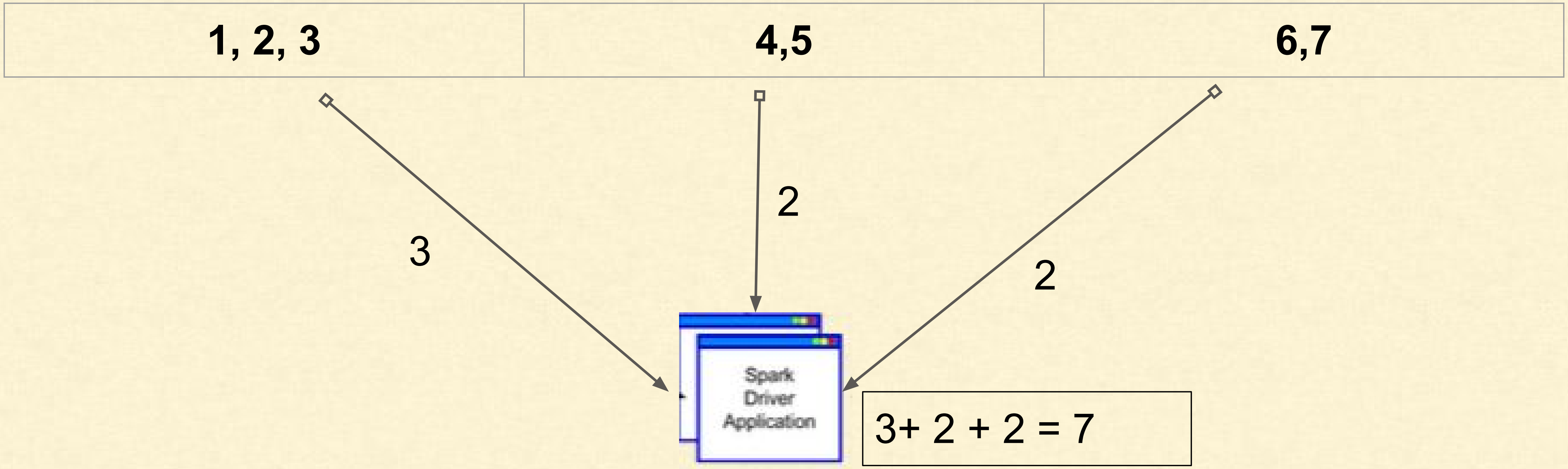
| | | | | | | |
|---|---|---|---|---|---|---|
| 1 | 2 | 3 | 4 | 5 | 6 | 7 |

# Actions: count() - Scala

To find out how many elements are there in an RDD.

Works in distributed fashion.

```
➢ var a = sc.parallelize(Array(1,2,3, 4, 5 , 6, 7), 3);
➢ var mycount =  a.count();
➢ mycount
  7
```



|  |  |  |
|---|---|---|
| **1, 2, 3** | **4,5** | **6,7** |

3+ 2 + 2 = 7

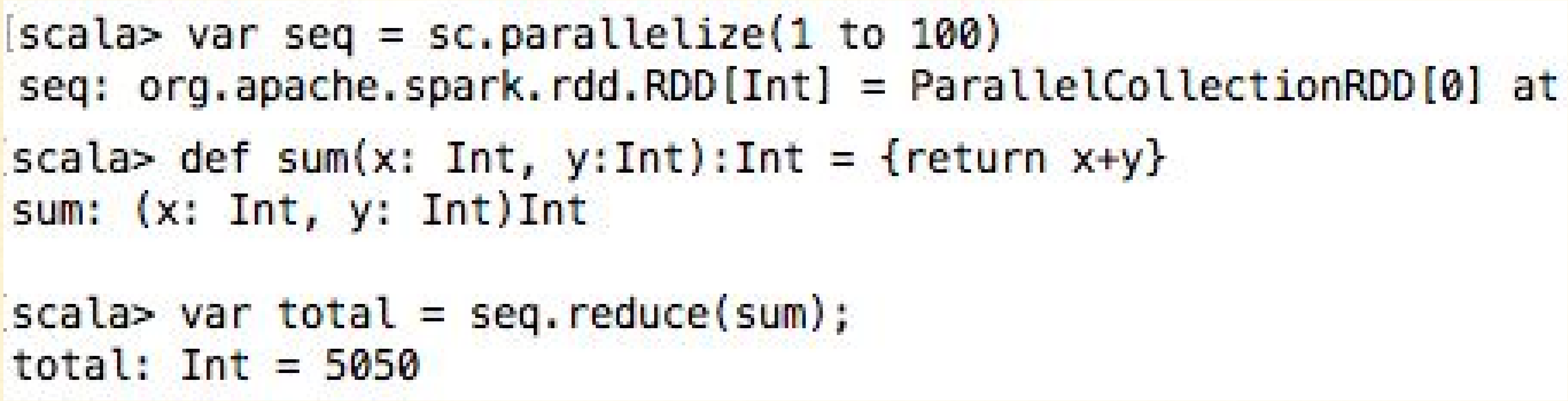
# More Actions - Reduce()

Aggregate elements of dataset using a function:

- Takes 2 arguments and returns only one
- Commutative and associative for parallelism
- Return type of function has to be same as argument

```
➢  var seq = sc.parallelize(1 to 100)
➢  def sum(x: Int, y:Int):Int = {return x+y}
➢  var total = seq.reduce(sum);
 total: Int = 5050
```

# More Actions - Reduce()

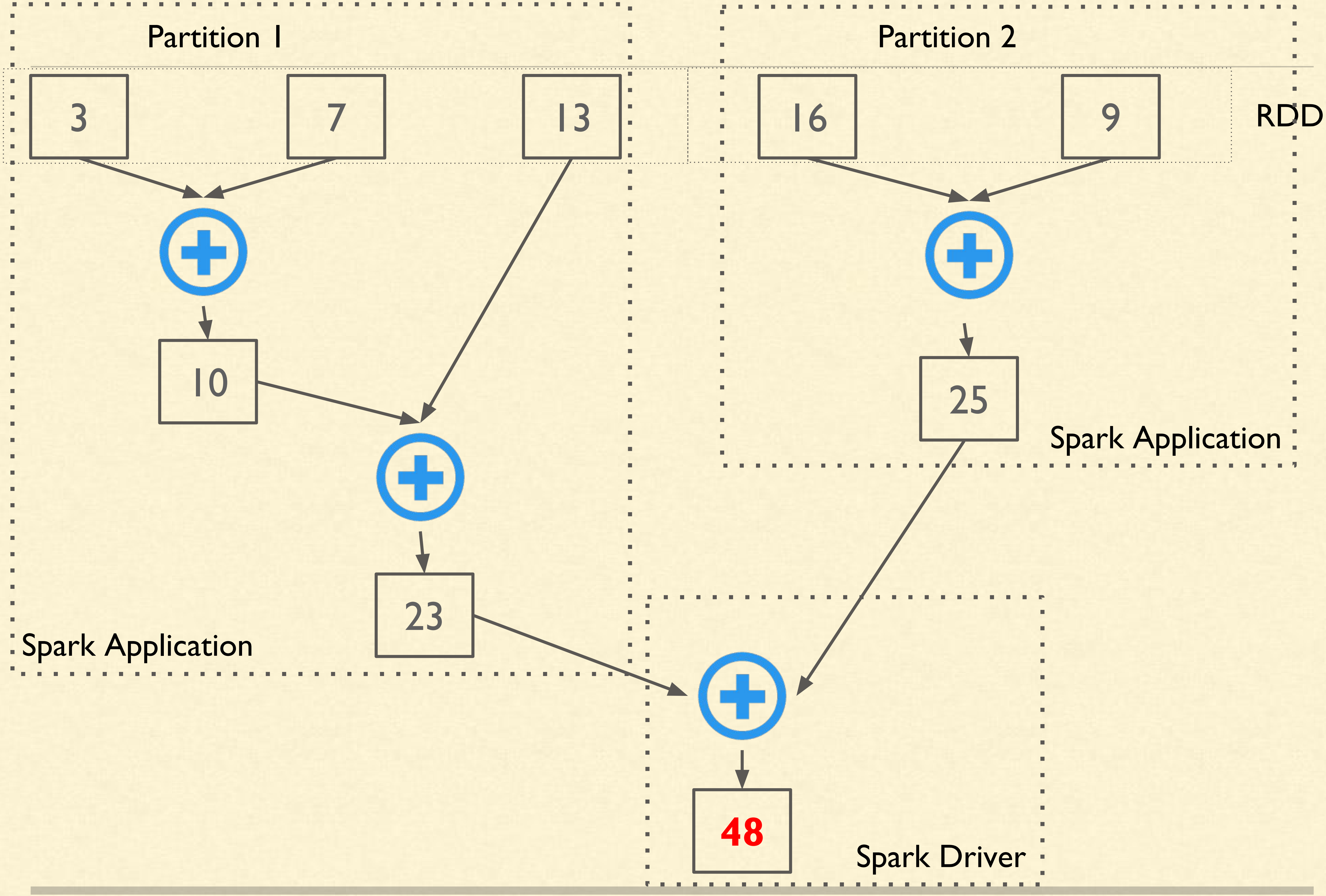


```
scala> var seq = sc.parallelize(1 to 100)
seq: org.apache.spark.rdd.RDD[Int] = ParallelCollectionRDD[0] at
scala> def sum(x: Int, y:Int):Int = {return x+y}
sum: (x: Int, y: Int)Int

scala> var total = seq.reduce(sum);
total: Int = 5050
```

# More Actions - Reduce()

To confirm, you could use the formula for summation of natural numbers
= n*(n+1)/2
= 100*101/2
= 5050

# How does reduce work?



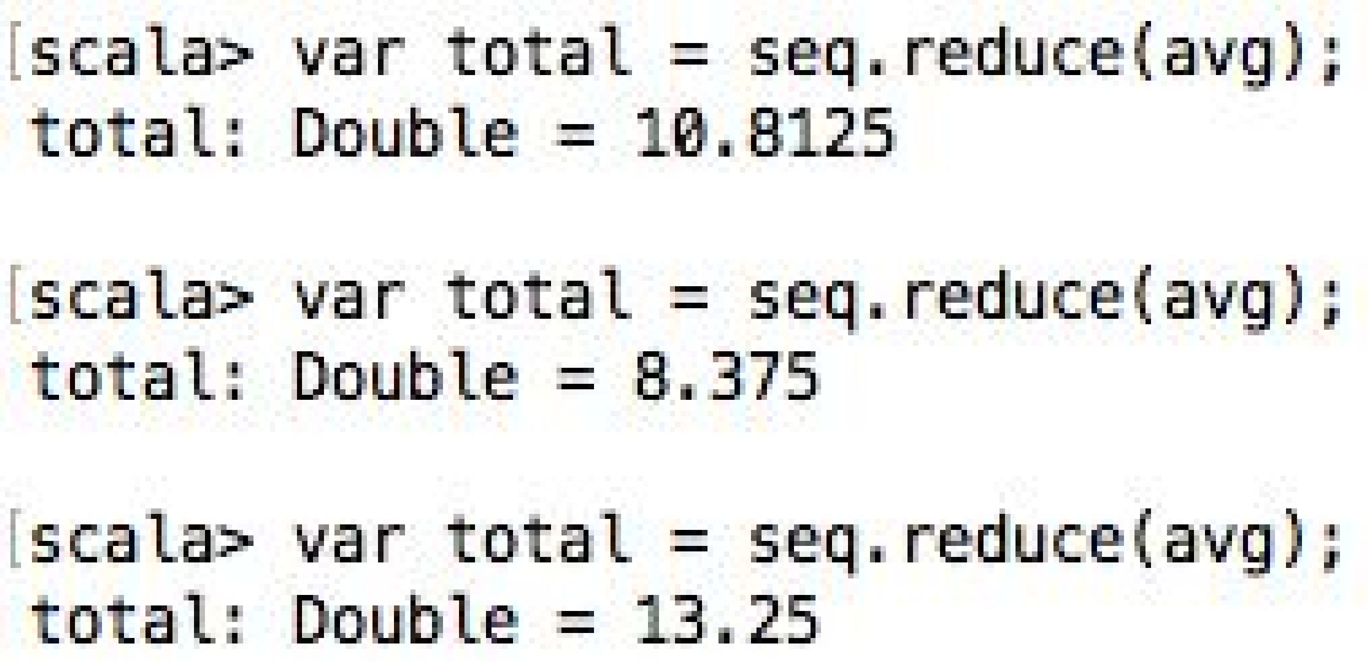
# For avg(), can we use reduce?

The way we had computed summation using reduce,

Can we compute the average in the same way?

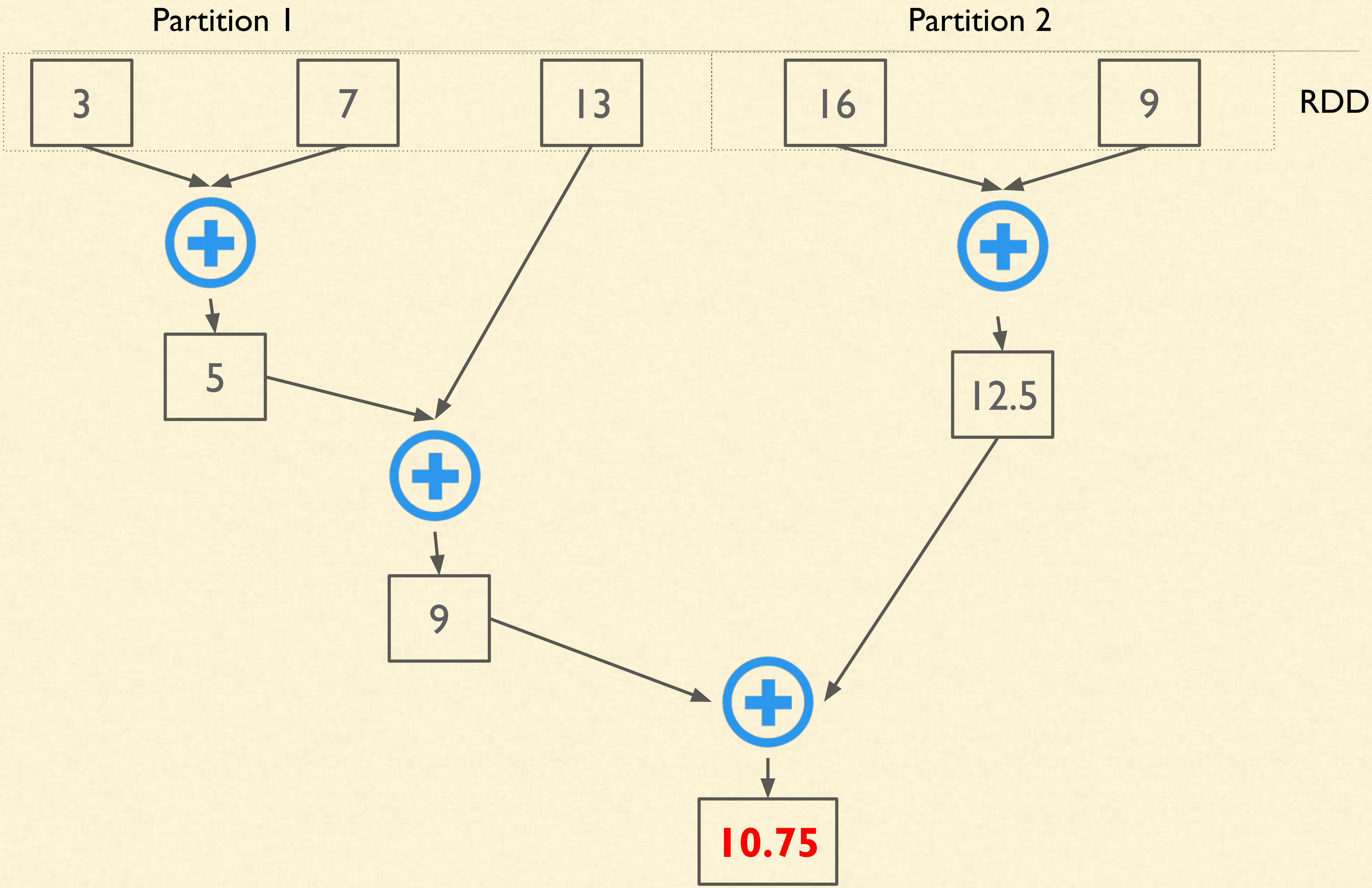
```
var seq = sc.parallelize(Array(3.0, 7, 13, 16, 19))
def avg(x: Double, y:Double):Double = {return (x+y)/2}
var total = seq.reduce(avg);
total: Double = 9.875
```

*Which is wrong. The correct average of 3, 7, 13, 16, 19 is 9.6.*



```
scala> var total = seq.reduce(avg);
total: Double = 10.8125

scala> var total = seq.reduce(avg);
total: Double = 8.375

scala> var total = seq.reduce(avg);
total: Double = 13.25
```

# Why average with reduce is wrong?



## Why average with reduce is wrong?

$$\frac{(3+7+13)}{3} \;!=\; \frac{\frac{3+7}{2}+13}{2}$$

## But sum is ok

$$3 + 4 + 5$$

$$=$$

$$4 \;+\; (3 + 5)$$

$$=$$

$$(4 \;+\; 3) + 5$$

$$=$$

$$(3 + 4) + 5$$

# Reduce

A reduce function must be
**commutative and associative**
otherwise
the results could be unpredictable and wrong.

# Commutative

If changing the order of inputs does not make any difference to output, the function is commutative.

**Examples**

Addition

**2 + 3 = 3 + 2**

Multiplication

2 * 3 = 3*2

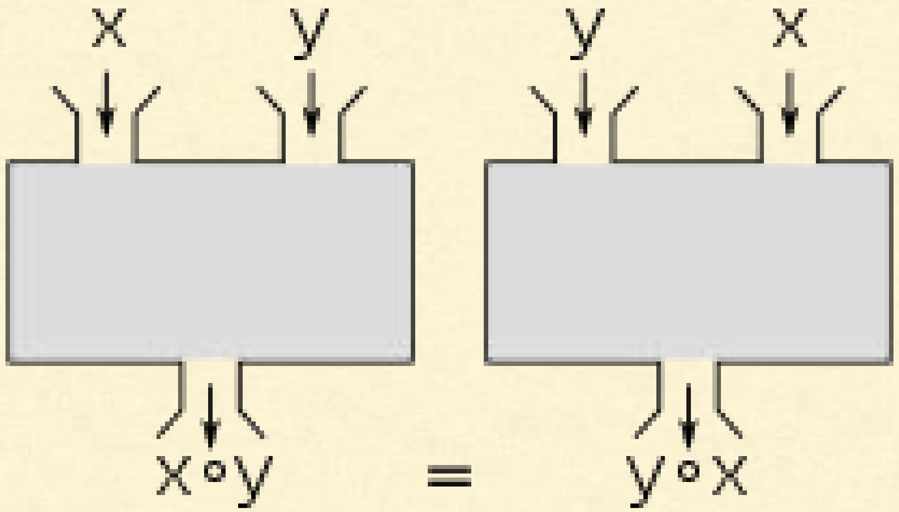
Average:

(3+4+5)/3 = (4+3+5)/3

Euclidean Distance:

$$\sqrt{(x1 - x2)^2 + (y1 - y2)^2}$$

$$= \sqrt{(x2 - x1)^2 + (y2 - y1)^2}$$



**Non Commutative**

Division

**2 / 3 not eq 3 / 2**

Subtraction

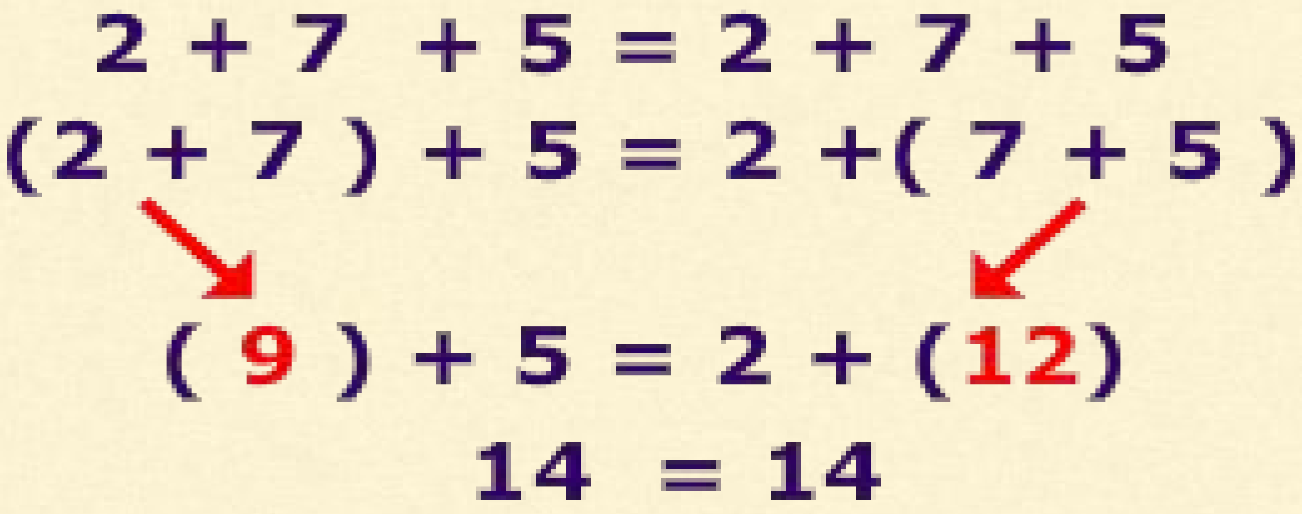
2 - 3 != 3 - 2

Exponent / power

4 ^ 2 != 2^4

# Associative

Associative property:
Can add or multiply regardless of how the numbers are grouped.
By 'grouped' we mean 'how you use parenthesis'.



2 + 7 + 5 = 2 + 7 + 5

(2 + 7 ) + 5 = 2 +( 7 + 5 )

( 9 ) + 5 = 2 + (12)

14 = 14

**Examples**

Multiplication:

(3 * 4 ) * 2 = 3 * ( 4 * 2)

Min:

Min(Min(3,4), 30)
= Min(3, Min(4, 30)) = 3

Max:

Max(Max(3,4), 30)
= Max(3, Min(4, 30)) = 30

**Non Associative**

Division:

***(⅔) / 4** not equal to **2 / (¾)***

Subtraction:

*(2 - 3) - 1 != 2 - (3-1)*

Exponent / power:

*4 ^ 2 != 2^4*

Average:

*avg(avg(2, 3), 4) != avg(avg(2, 3), 4)*

# Solving Some Problems with Spark

# Approach 1 - So, how to compute average?

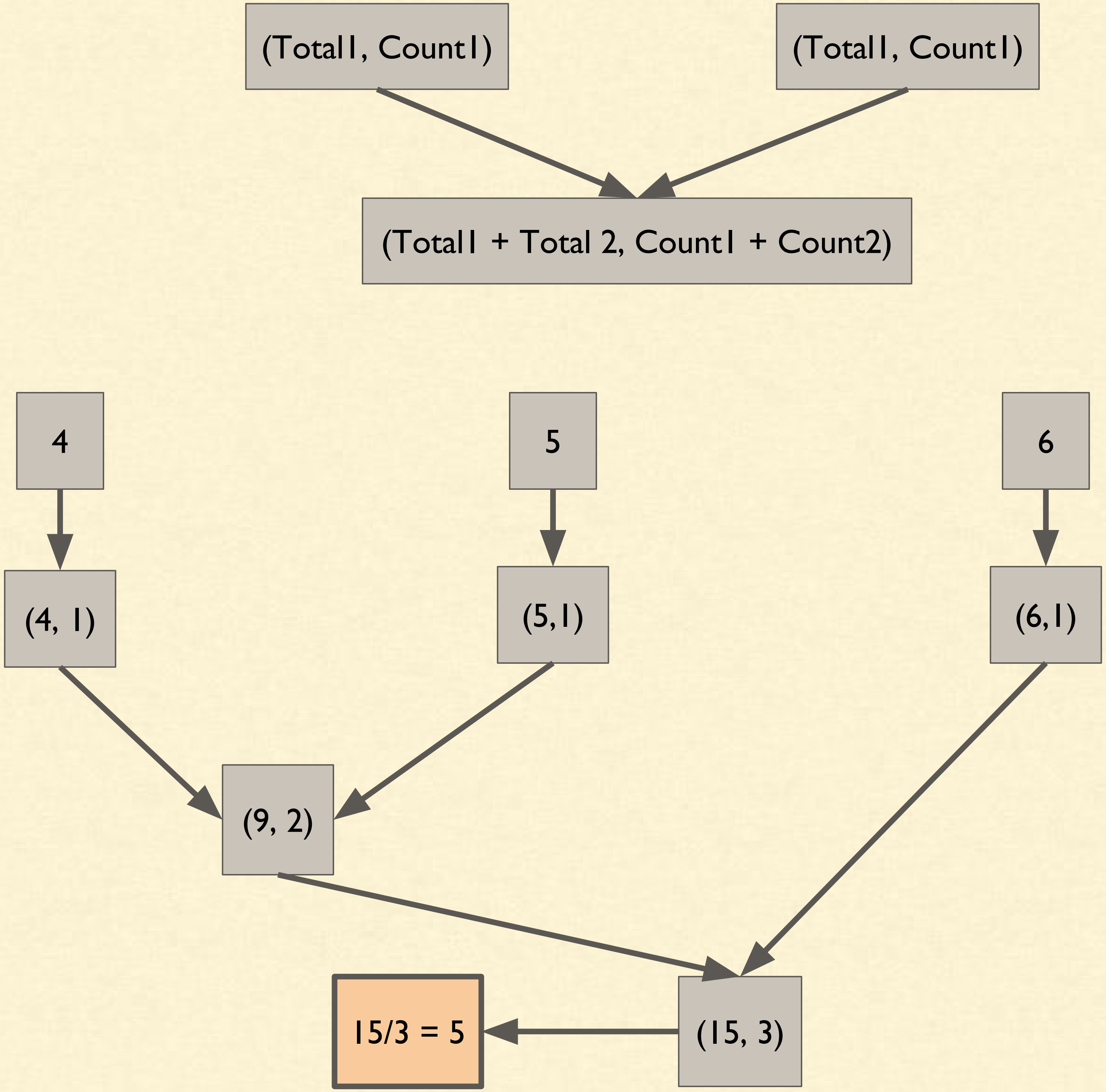
Approach 1

```
➢ var rdd = sc.parallelize(Array(1.0,2,3, 4, 5 , 6, 7), 3);
➢ var avg = rdd.reduce(_ + _) / rdd.count();
```

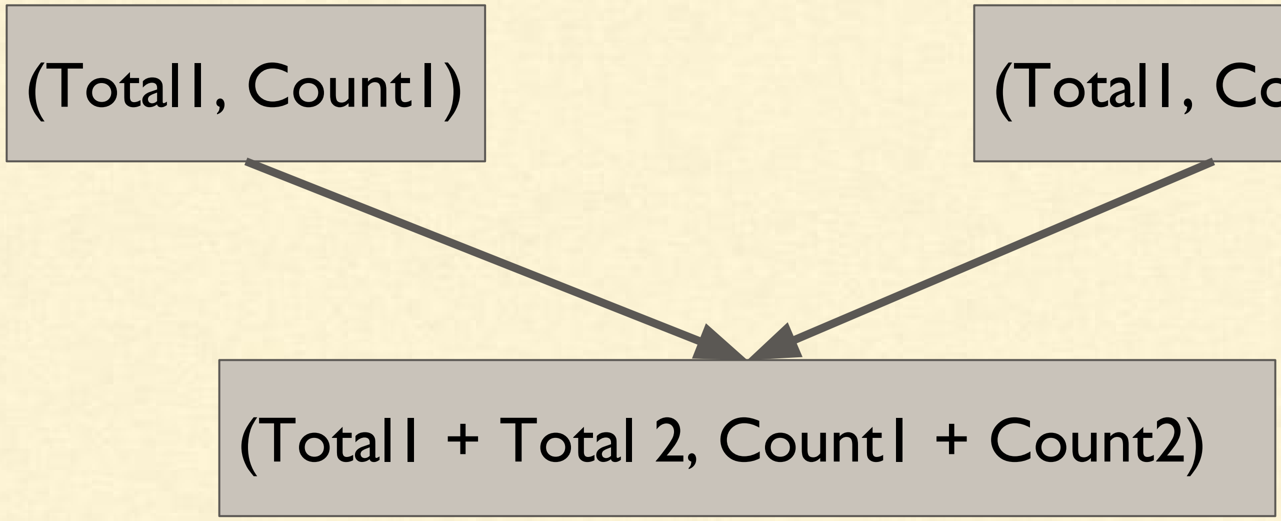
What's wrong with this approach?

We are computing RDD twice - during **reduce** and during **count**.
Can we compute sum and count in a single reduce?

# Approach 2 - So, how to compute average?



# Approach 2 - So, how to compute average?



(Total1, Count1)

(Total1, Count1)

(Total1 + Total 2, Count1 + Count2)

```
   ➢  var rdd = sc.parallelize(Array(1.0,2,3, 4, 5 , 6, 7), 3);
   ➢  var rdd_count = rdd.map((_, 1))
   ➢  var (sum, count) = rdd_count.reduce((x, y) => (x._1 + y._1, x._2 + y._2))
   ➢  var avg = sum / count
avg: Double = 4.0
```

# Comparision of the two approaches?

Approach1:

0.023900 + 0.065180

= **0.089**08 seconds ~ 89 ms

Approach2:

0.058654 seconds ~ 58 ms

Approximately 2X difference.

# So, how to compute Standard deviation?

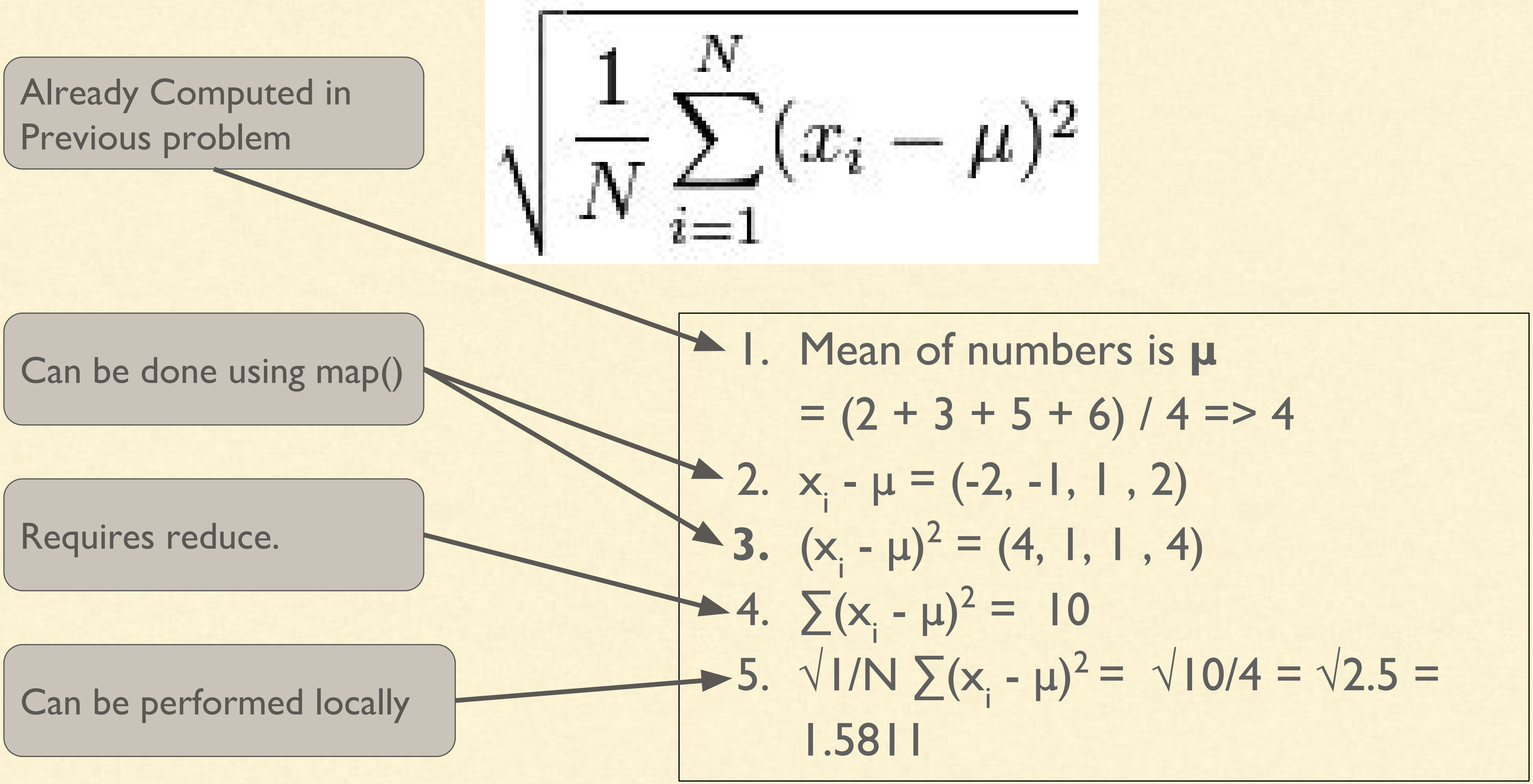
The Standard Deviation is a measure of how spread out numbers are.

$$\sqrt{\frac{1}{N}\sum_{i=1}^{N}(x_i - \mu)^2}$$

1. Work out the Mean (the simple average of the numbers)
2. Then for each number: subtract the Mean and square the result
3. Then work out the mean of those squared differences.
4. Take the square root of that and we are done!

# So, how to compute Standard deviation?

Lets calculate SD of  2 3 5 6



$$\sqrt{\frac{1}{N}\sum_{i=1}^{N}(x_i - \mu)^2}$$

Already Computed in Previous problem

Can be done using map()

Requires reduce.

Can be performed locally

1. Mean of numbers is **μ**
   = (2 + 3 + 5 + 6) / 4 => 4
2. $x_i$ - μ = (-2, -1, 1 , 2)
3. ($x_i$ - μ)$^2$ = (4, 1, 1 , 4)
4. ∑($x_i$ - μ)$^2$ =  10
5. √1/N ∑($x_i$ - μ)$^2$ =  √10/4 = √2.5 = 1.5811

# So, how to compute Standard deviation?

➢ `var rdd = sc.parallelize(Array(2, 3, 5, 6))`

//Mean or average of numbers is **μ**

➢ `var rdd_count = rdd.map((_, 1))`

➢ `var (sum, count) = rdd_count.reduce((x, y) => (x._1 + y._1, x._2 + y._2))`

➢ `var avg = sum / count`

// $(x_i - \mu)^2$

➢ `var sqdiff = rdd.map( _ - avg).map(x => x*x)`

// $\sum(x_i - \mu)^2$

➢ `var sum_sqdiff = sqdiff.reduce(_ + _)`

//$\surd 1/N \sum(x_i - \mu)^2$

➢ `import math._;`

➢ `var sd = sqrt(sum_sqdiff*1.0/count)`

```
sd: Double = 1.5811388300841898
```

# Computing random sample from a dataset

The objective of the excercise is to pick a random sample from a given RDD. Though there is a method provided in RDD but we are create our own.

1. Lets try to understand it for say picking 50% records.
2. The approach is very simple. We pick a record from RDD and do a coin toss. If its head, keep the element otherwise discard it. It can be achived using filter.
3. For picking any fraction, we might use a coin having 100s of faces or in other words a random number generator.
4. Please notice that it would not give the sample of exact size

# Computing random sample from a dataset

```
var rdd = sc.parallelize(1 to 1000);
var fraction = 0.1
def cointoss(x:Int): Boolean = scala.util.Random.nextFloat() <= fraction
var myrdd = rdd.filter(cointoss)
var localsample = myrdd.collect()
localsample.length
```

Basics of RDD

Thank you!