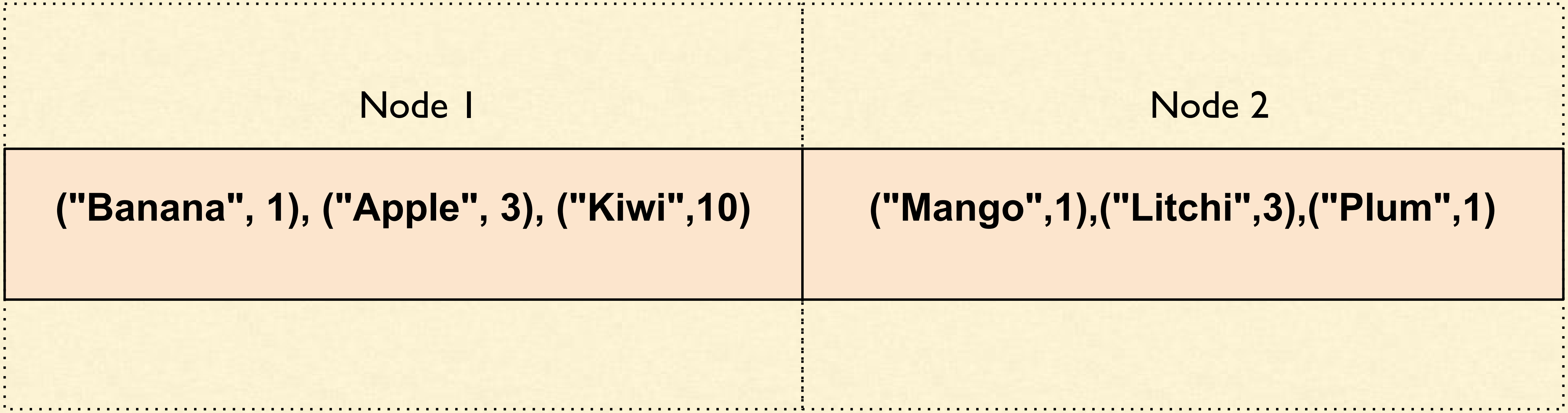
APACHE
Spark™

Key-Value RDD

Key - Value RDD



Node 1	Node 2
("Banana", 1), ("Apple", 3), ("Kiwi",10)	**("Mango",1),("Litchi",3),("Plum",1)**

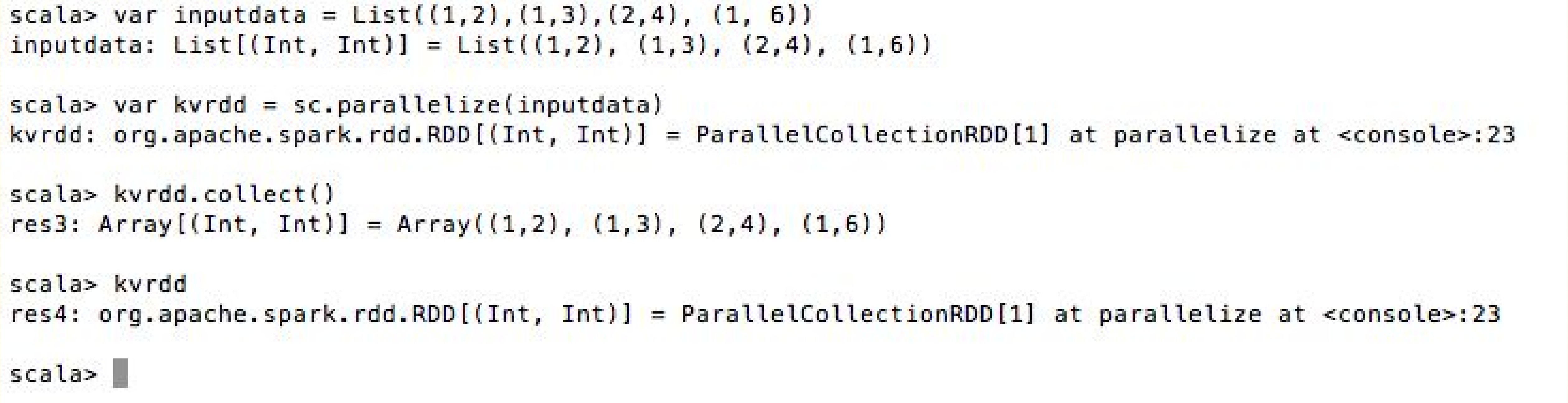
1. You can keep any kind of data in an RDD
2. Having tuple is quite useful
    a. For example, pair RDDs have a reduceByKey() & joins
3. Key-value RDD containing tuples with first value as key

Key - Value Pair RDDs

1. A Pair is defined as (x, y) - also known as tuple
2. A tuple is an immutable sequence of objects.
3. You can convert a list into tuple

Creating Key-Value Pair RDDs

```
var inputdata = List((1,2),(1,3),(2,4), (1, 6))
var kvrdd = sc.parallelize(inputdata)
kvrdd.collect()
```



```
scala> var inputdata = List((1,2),(1,3),(2,4), (1, 6))
inputdata: List[(Int, Int)] = List((1,2), (1,3), (2,4), (1,6))

scala> var kvrdd = sc.parallelize(inputdata)
kvrdd: org.apache.spark.rdd.RDD[(Int, Int)] = ParallelCollectionRDD[1] at parallelize at <console>:23

scala> kvrdd.collect()
res3: Array[(Int, Int)] = Array((1,2), (1,3), (2,4), (1,6))

scala> kvrdd
res4: org.apache.spark.rdd.RDD[(Int, Int)] = ParallelCollectionRDD[1] at parallelize at <console>:23

scala> 
```

Transformations on Pair RDDs

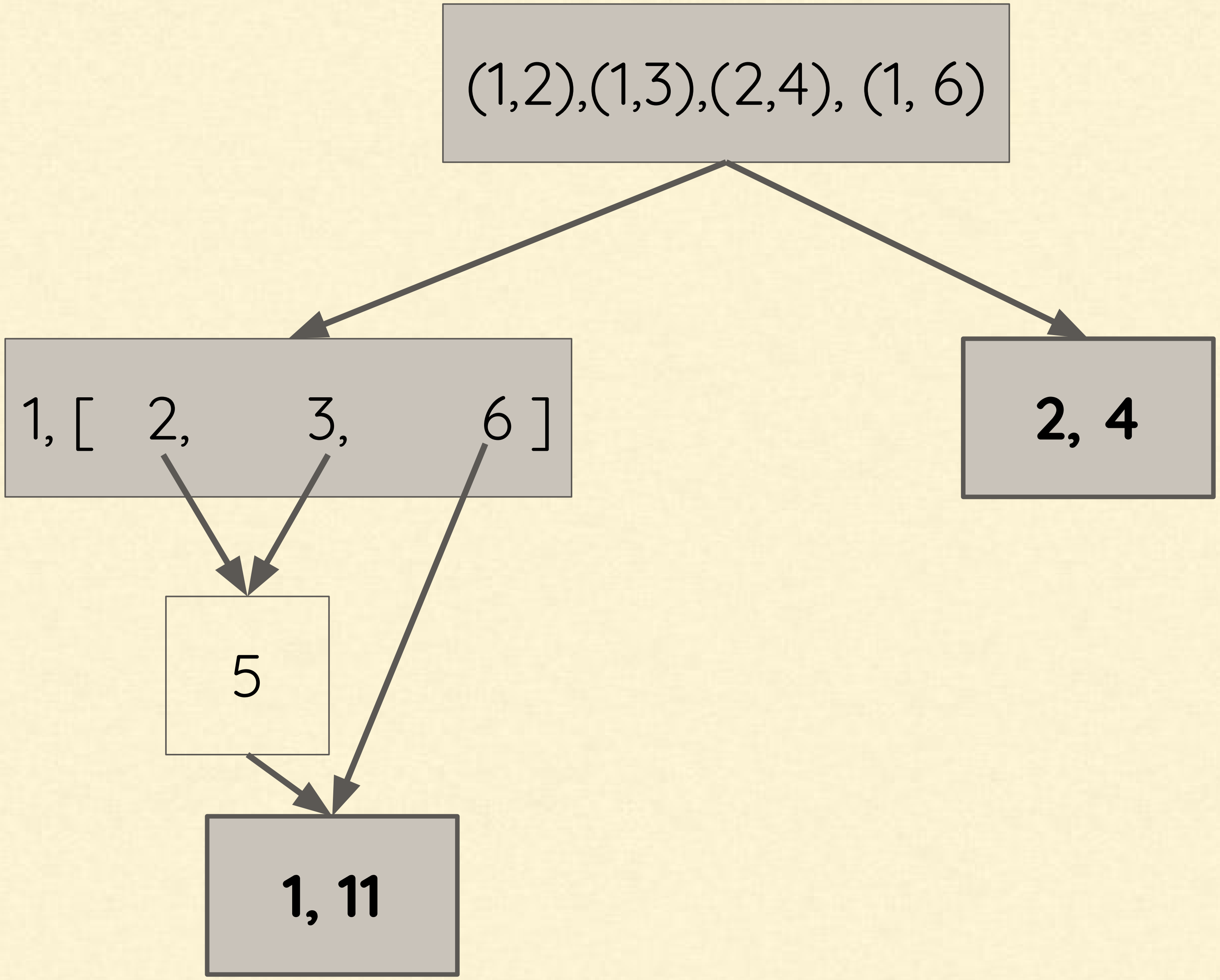
reduceByKey(func)

Groups the elements of an RDD based on the key and then reduces the values for each key using function provided as argument

```
var inputdata = List((1,2),(1,3),(2,4), (1, 6))
var kvrdd = sc.parallelize(inputdata)
val out = kvrdd.reduceByKey((a, b) => a + b)
out.collect()

Array((1,11), (2,4))
```

Transformations on Pair RDDs

reduceByKey(func)



(1,2),(1,3),(2,4), (1, 6)

1, [2, 3, 6]

2, 4

5

1, 11

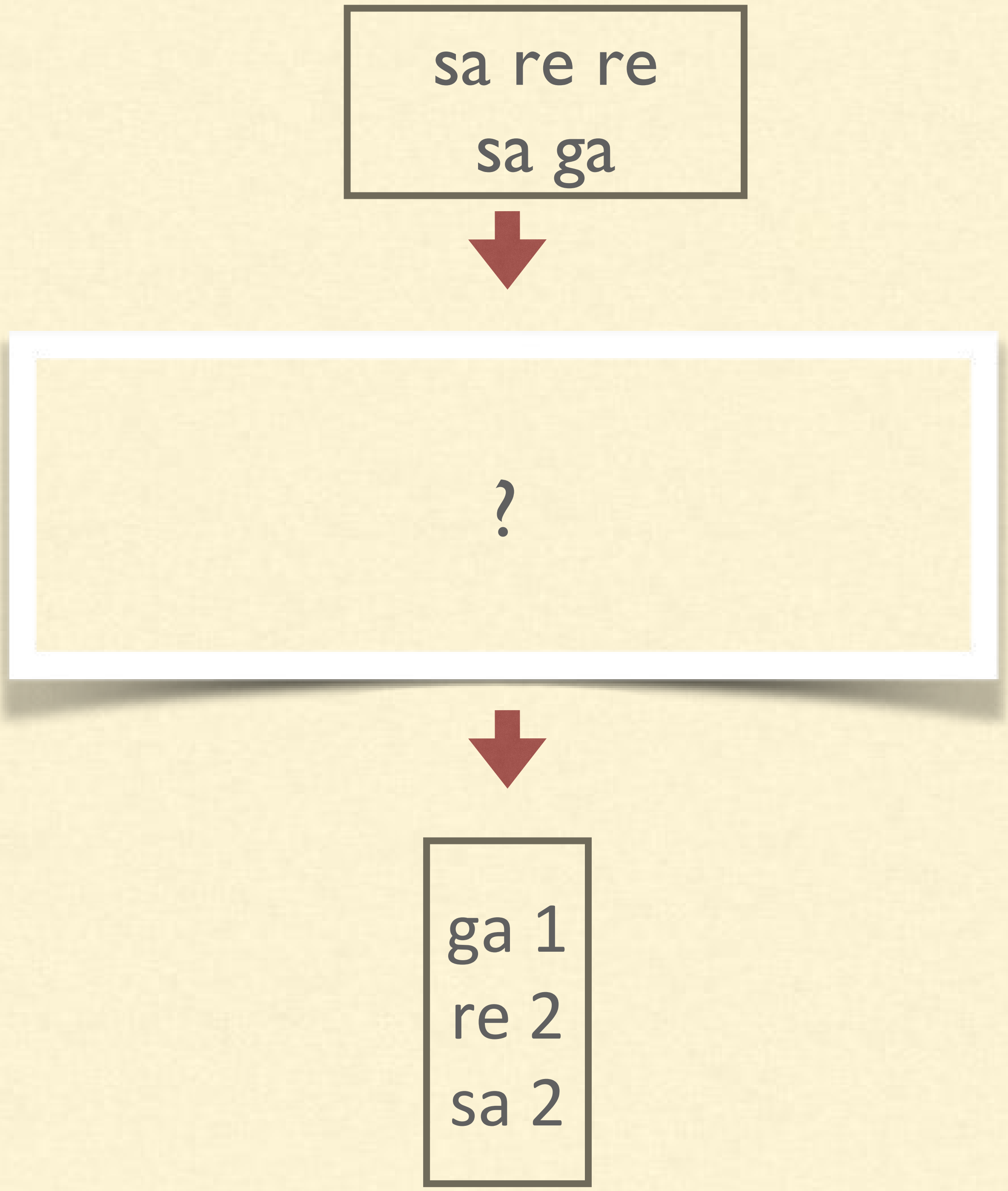
Example: Words Frequencies

- Compute the frequencies of words in a huge text file.

OR

- To find the unique words in the text and find the how many times each word has occurred

Word frequency problem.

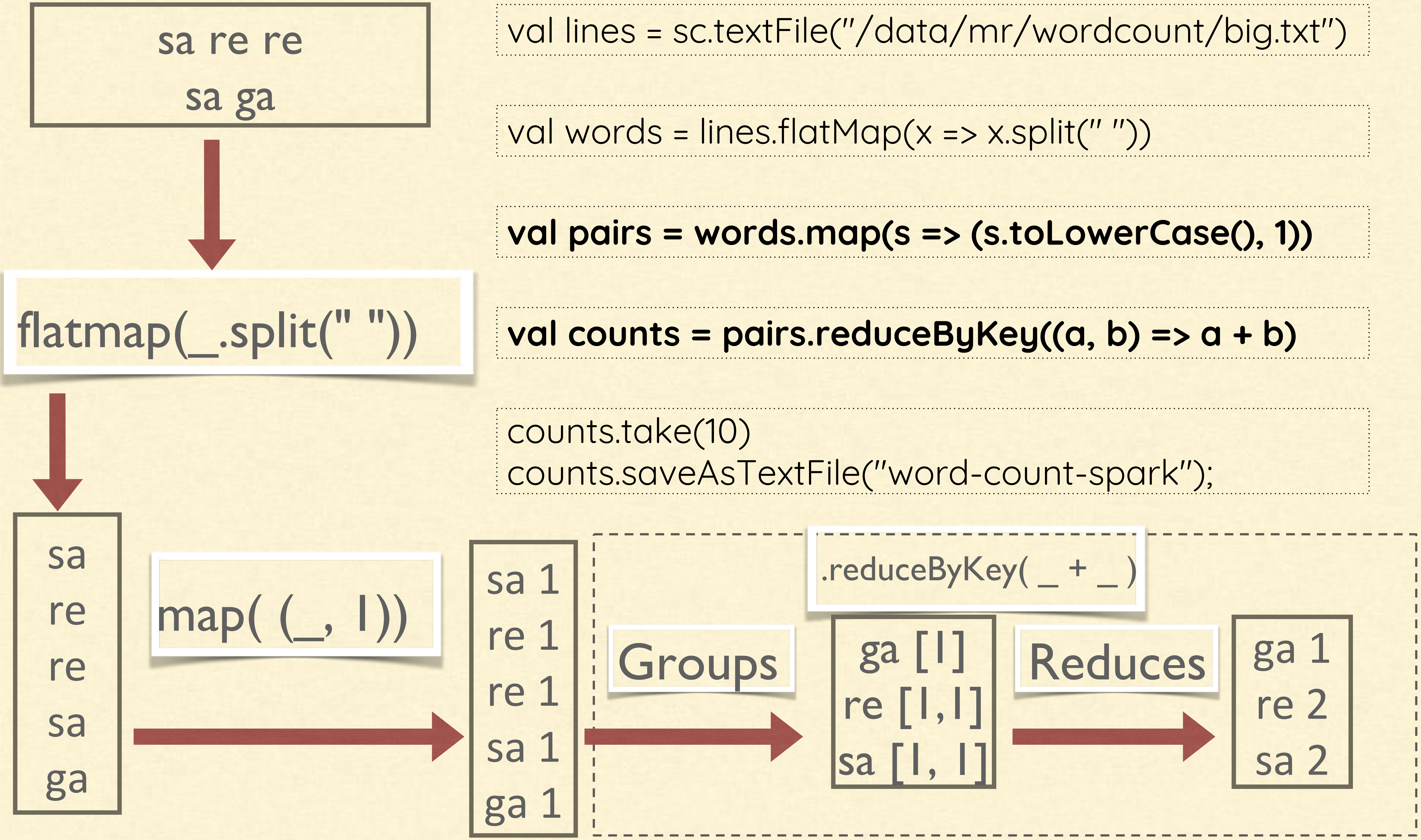


sa re re
sa ga

?

ga 1
re 2
sa 2

Spark - Word frequency problem



```
val lines = sc.textFile("/data/mr/wordcount/big.txt")
```

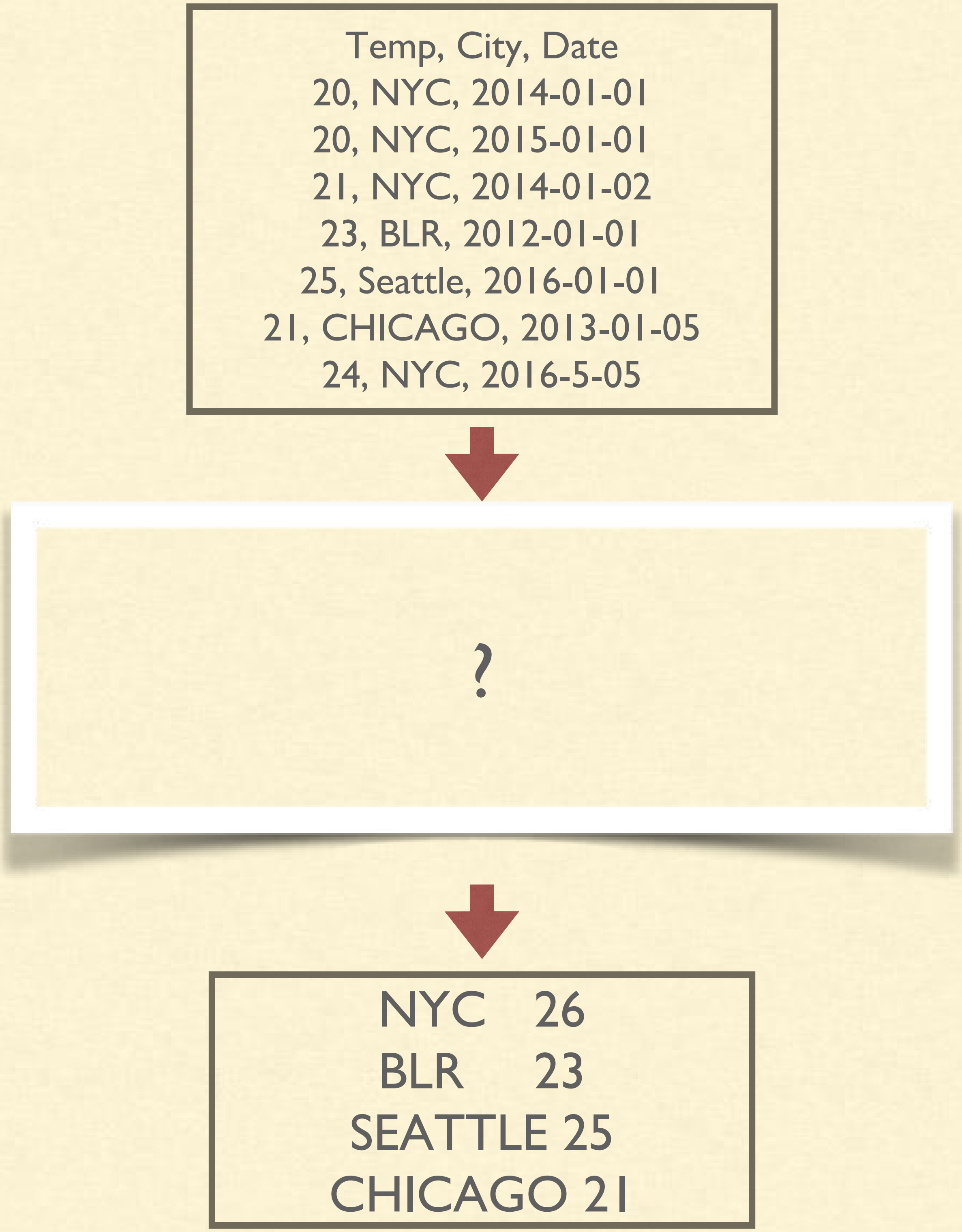
```
val words = lines.flatMap(x => x.split(" "))
```

```
val pairs = words.map(s => (s.toLowerCase(), 1))
```

```
val counts = pairs.reduceByKey((a, b) => a + b)
```

```
counts.take(10)
counts.saveAsTextFile("word-count-spark");
```

Spark - Computing max temp



Temp, City, Date
20, NYC, 2014-01-01
20, NYC, 2015-01-01
21, NYC, 2014-01-02
23, BLR, 2012-01-01
25, Seattle, 2016-01-01
21, CHICAGO, 2013-01-05
24, NYC, 2016-5-05

?

NYC 26
BLR 23
SEATTLE 25
CHICAGO 21

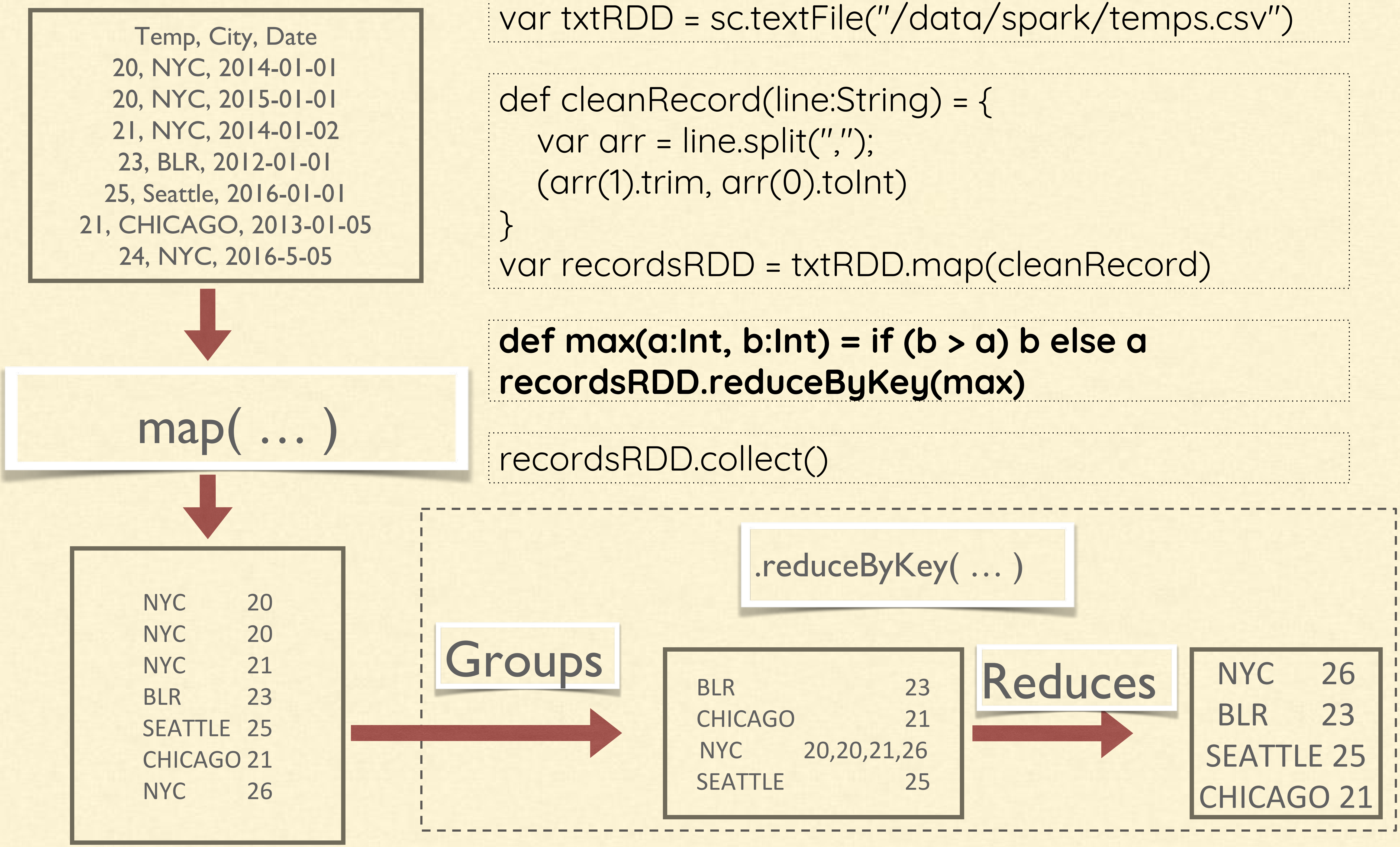
Spark - Computing max temp

```
var txtRDD = sc.textFile("/data/spark/temps.csv")
```

```
def cleanRecord(line:String) = {
    var arr = line.split(",");
    (arr(1).trim, arr(0).toInt)
}
var recordsRDD = txtRDD.map(cleanRecord)
```

```
def max(a:Int, b:Int) = if (b > a) b else a
recordsRDD.reduceByKey(max)
```

```
recordsRDD.collect()
```



Temp, City, Date
20, NYC, 2014-01-01
20, NYC, 2015-01-01
21, NYC, 2014-01-02
23, BLR, 2012-01-01
25, Seattle, 2016-01-01
21, CHICAGO, 2013-01-05
24, NYC, 2016-5-05

NYC	20
NYC	20
NYC	21
BLR	23
SEATTLE	25
CHICAGO	21
NYC	26

BLR	23
CHICAGO	21
NYC	20,20,21,26
SEATTLE	25

NYC	26
BLR	23
SEATTLE	25
CHICAGO	21

Transformations on Pair RDDs

keys()

Returns an RDD with the keys of each tuple.

```
>>> var m = sc.parallelize(List((1, 2), (3, 4))).keys
>>> m.collect()
Array[Int] = Array(1, 3)
```

Transformations on Pair RDDs

values()

Return an RDD with the values of each tuple.

```
>>> var m = sc.parallelize(List((1, 2), (3, 4))).values
>>> m.collect()
Array(2, 4)
```

Transformations on Pair RDDs

groupByKey()

Group values with the same key.

```
var rdd = sc.parallelize(List((1, 2), (3, 4), (3, 6)));
var rdd1 = rdd.groupByKey()
var vals = rdd1.collect()
for( i <- vals){
    for (k <- i.productIterator) {
        println("\t" + k);
    }
}
```

Questions - Set Operations

What will be the result of the following?

```
rdd = sc.parallelize([("a", 1), ("b", 1), ("a", 1)]);
rdd.groupByKey().mapValues(len).collect()
```

Questions - Set Operations

What will be the result of the following?

```
rdd = sc.parallelize([(″a″, 1), (″b″, 1), (″a″, 1)]);
rdd.groupByKey().mapValues(len).collect()
```

[('a', 2), ('b', 1)]

[('a', 1), ('b', 1), ('a', 1)]
=> groupBY => [('a', [1, 1]), ('b', [1])]
=> mapValues(len) => [('a', 2), ('b', 1)]

Transformations on Pair RDDs

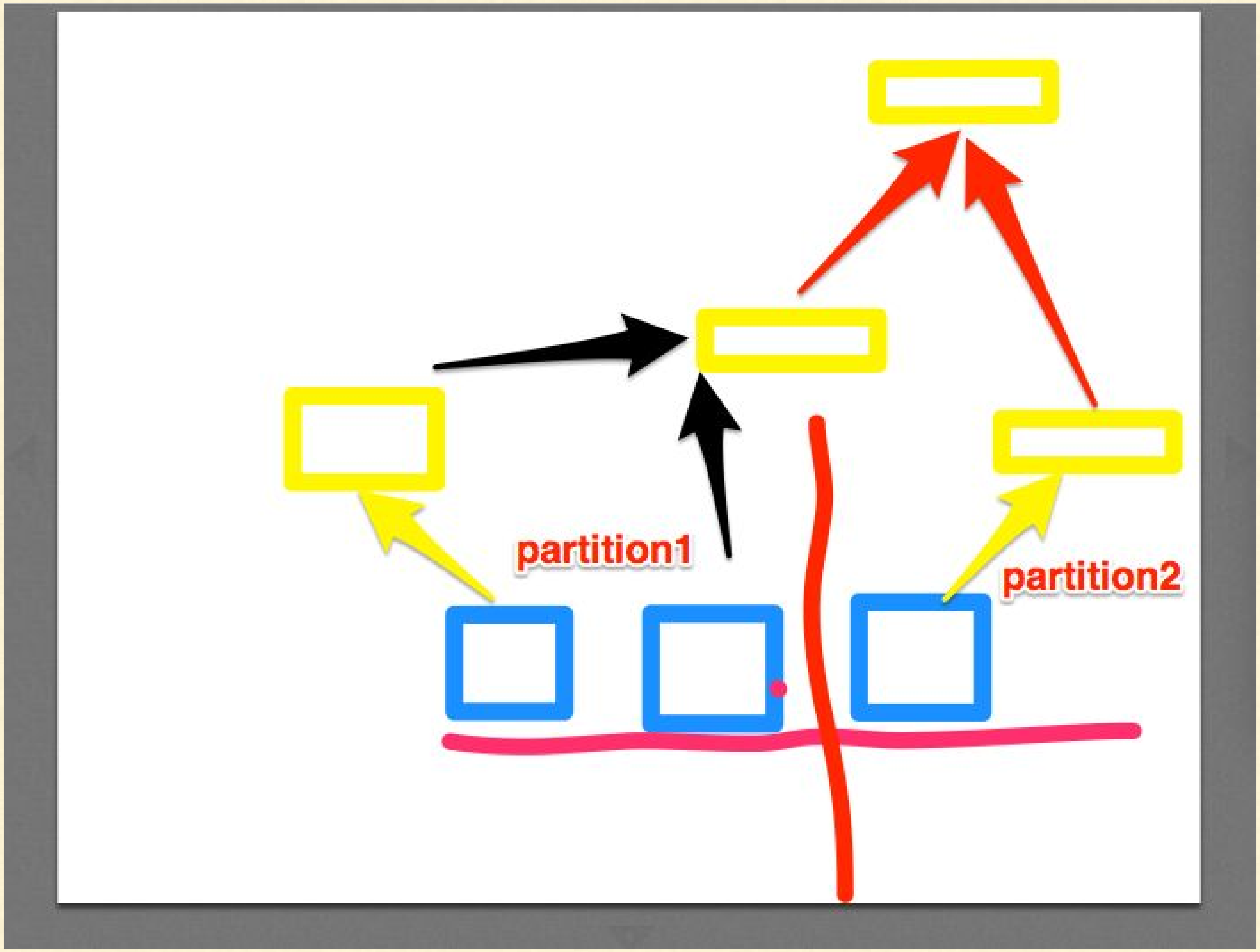
combineByKey(createCombiner, mergeValue, mergeCombiners, numPartitions=None)

Combine values with the same key using a different result type.
Turns RDD[(K, V)] into a result of type RDD[(K, C)]

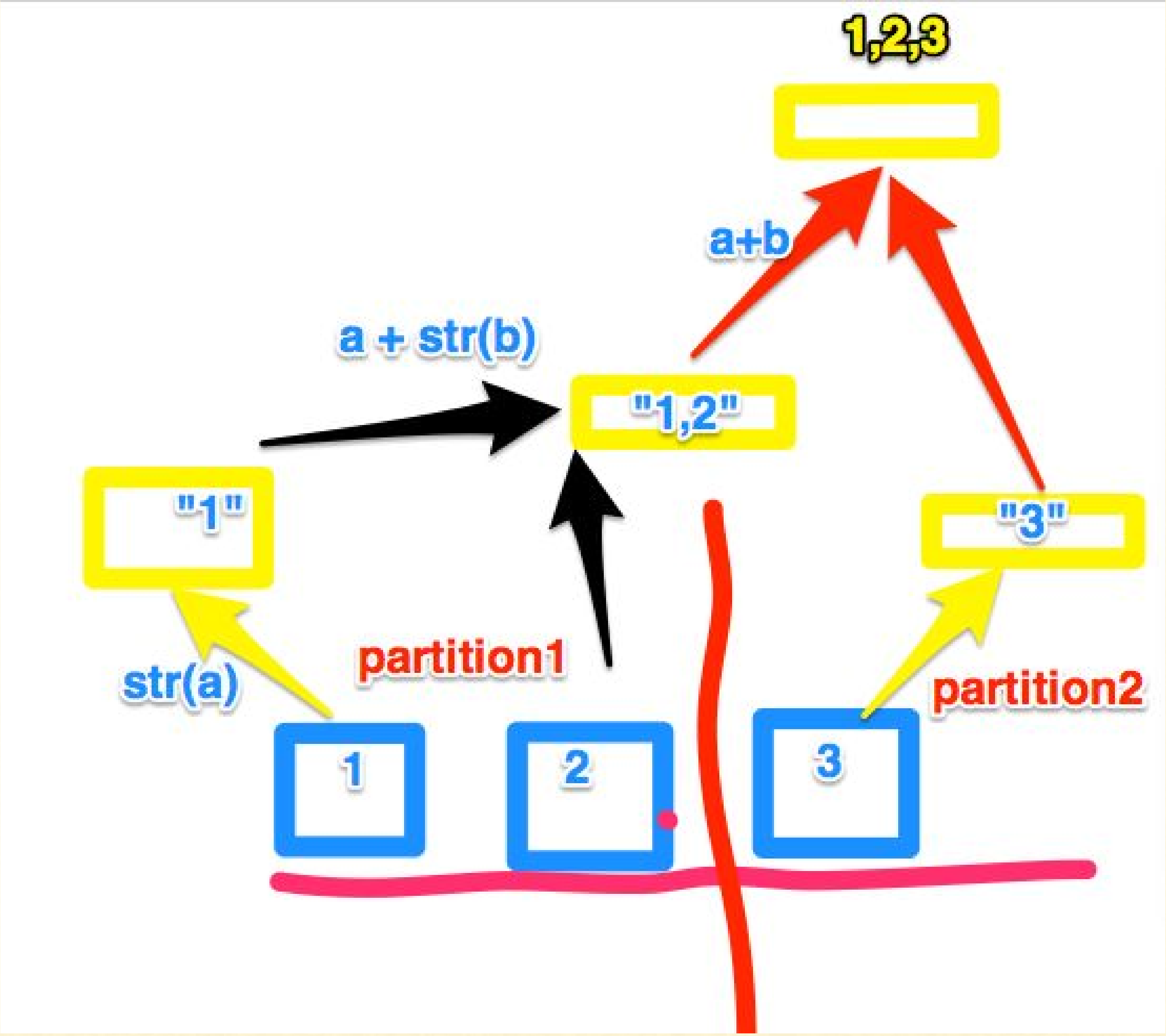
createCombiner, which turns a V into a C (e.g., creates a one-element list)
mergeValue, to merge a V into a C (e.g., adds it to the end of a list)
mergeCombiners, to combine two C's into a single one.

```
rdd1 = sc.parallelize([("a", 1), ("a", 2), ("a", 3), ("b", 3)])
def f(x): return str(x)
def g(x, y): return str(x)+ "," +y;
def h(x, y): return x+"," + y;
mergedrdd = rdd1.combineByKey(f, g, h, numPartitions=None)
[('b', '3'), ('a', '1,2,3')]
```

Transformations on Pair RDDs



Transformations on Pair RDDs



Questions - Set Operations

What will be the result of the following?

```
def cc (v): return ("[" , v , "]");

def mv (c, v): return c[0:-1] + (v, "]")

def mc(c1,c2):    return c1[0:-1] + c2[1:]

mc(mv(cc(1), 2), cc(3))
```

Questions - Set Operations

What will be the result of the following?

```
def cc (v): return ("[" , v , "]");

def mv (c, v): return c[0:-1] + (v, "]")

def mc(c1,c2):    return c1[0:-1] + c2[1:]

mc(mv(cc(1), 2), cc(3))
```

('[', 1, 2, 3, ']')

Questions - Set Operations

What will be the result of the following?

```
def cc (v): return ("[" , v , "]");

def mv (c, v): return c[0:-1] + (v, "]")

def mc(c1, c2): return c1[0:-1] + c2[1:]

rdd = sc.parallelize([("a", 1), ("b", 2), ("a", 3)])
rdd.combineByKey(cc,mv, mc).collect()
```

Questions - Set Operations

What will be the result of the following?

```
def cc (v): return ("[" , v , "]");

def mv (c, v): return c[0:-1] + (v, "]")

def mc(c1, c2):    return c1[0:-1] + c2[1:]

rdd = sc.parallelize([("a", 1), ("b", 2), ("a", 3)])
rdd.combineByKey(cc,mv, mc).collect()
```

```
[('a', ('[', 1, 3, ']')), ('b', ('[', 2, ']'))]
```

Transformations on Pair RDDs

sortByKey(ascending=True, numPartitions=None, keyfunc=<function <lambda>>)

Sorts this RDD, which is assumed to consist of (key, value) pairs.

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

>>> sc.parallelize(tmp).sortByKey(true, 1).collect()
[('1', 3), ('2', 5), ('a', 1), ('b', 2), ('d', 4)]

>>> sc.parallelize(tmp).sortByKey(ascending=true, numPartitions=2).collect()
[('1', 3), ('2', 5), ('a', 1), ('b', 2), ('d', 4)]
```

Transformations on Pair RDDs

sortByKey(ascending=True, numPartitions=None, keyfunc=<function <lambda>>)

Sorts this RDD, which is assumed to consist of (key, value) pairs.

```
>>> tmp = [('a', 1), ('b', 2), ('1', 3), ('d', 4), ('2', 5)]
>>> sc.parallelize(tmp).sortByKey().first()
('1', 3)

>>> sc.parallelize(tmp).sortByKey(True, 1).collect()
[('1', 3), ('2', 5), ('a', 1), ('b', 2), ('d', 4)]

>>> sc.parallelize(tmp).sortByKey(ascending=True,
numPartitions=2).collect()
[('1', 3), ('2', 5), ('a', 1), ('b', 2), ('d', 4)]
```

Transformations on Pair RDDs

sortByKey(ascending=True, numPartitions=None, keyfunc=<function <lambda>>)

Sorts this RDD, which is assumed to consist of (key, value) pairs.

```
>>> tmp = [('Mary', 1), ('had', 2), ('a', 3), ('little', 4), ('lamb', 5), ('whose', 6),
('fleece', 7), ('was', 8), ('white', 9)]
>>> sc.parallelize(tmp).sortByKey(True, 3, keyfunc=lambda k:
k.lower()).collect()
[('a', 3), ('fleece', 7), ('had', 2), ('lamb', 5),...('white', 9), ('whose', 6)]
```

Transformations on Pair RDDs

subtractByKey(other, numPartitions=None)

Return each (key, value) pair in self that has no pair with matching key in other.

```
>>> var x = sc.parallelize(List(("a", 1), ("b", 4), ("b", 5), ("a", 2)))
>>> var y = sc.parallelize(List(("a", 3), ("c", None)))
>>> x.subtractByKey(y).collect()
[('b', 4), ('b', 5)]
```

Transformations on Pair RDDs

subtractByKey(other, numPartitions=None)

Return each (key, value) pair in self that has no pair with matching key in other.

```
>>> x = sc.parallelize([(″a″, 1), (″b″, 4), (″b″, 5), (″a″, 2)])
>>> y = sc.parallelize([(″a″, 3), (″c″, None)])
>>> x.subtractByKey(y).collect()
[(′b′, 4), (′b′, 5)]
```

Transformations on Pair RDDs

join(other, numPartitions=None)

Return an RDD containing all pairs of elements with matching keys in self and other.
Each pair of elements will be returned as a (k, (v1, v2)) tuple, where (k, v1) is in self and (k, v2) is in other.
Performs a hash join across the cluster.

```
>>> var x = sc.parallelize(List(("a", 1), ("b", 4)))
>>> var y = sc.parallelize(List(("a", 2), ("a", 3)))
>>> x.join(y).collect()
Array((a,(1,2)), (a,(1,3)))
```

Transformations on Pair RDDs

join(other, numPartitions=None)

Return an RDD containing all pairs of elements with matching keys in self and other.
Each pair of elements will be returned as a (k, (v1, v2)) tuple, where (k, v1) is in self and (k, v2) is in other.
Performs a hash join across the cluster.

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

Transformations on Pair RDDs

leftOuterJoin(other, numPartitions=None)

Perform a left outer join of self and other.

For each element (k, v) in self, the resulting RDD will either contain all pairs (k, (v, w)) for w in other, or the pair (k, (v, None)) if no elements in other have key k.

Hash-partitions the resulting RDD into the given number of partitions.

```
>>> var x = sc.parallelize(List(("a", 1), ("b", 4)))
>>> var  y = sc.parallelize(List(("a", 2)))
>>> x.leftOuterJoin(y).collect()
Array((a,(1,Some(2))), (b,(4,None)))
```

Transformations on Pair RDDs

leftOuterJoin(other, numPartitions=None)

Perform a left outer join of self and other.

For each element (k, v) in self, the resulting RDD will either contain all pairs (k, (v, w)) for w in other, or the pair (k, (v, None)) if no elements in other have key k.

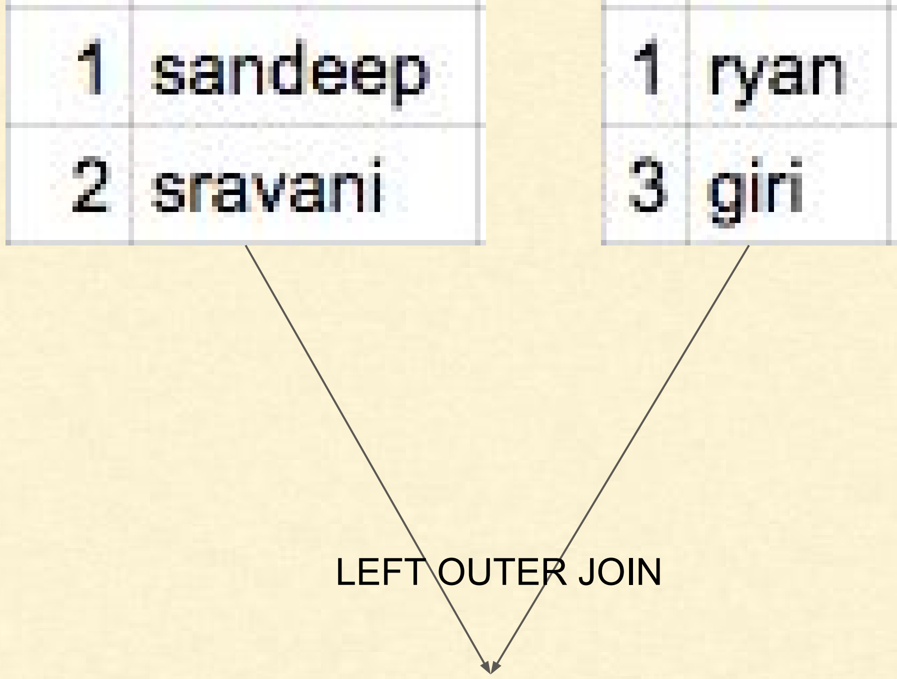
Hash-partitions the resulting RDD into the given number of partitions.

```
>>> x = sc.parallelize([("a", 1), ("b", 4)])
>>> y = sc.parallelize([("a", 2)])
>>>x.leftOuterJoin(y).collect()
[('a', (1, 2)), ('b', (4, None))]
```

Questions - Set Operations

What will be the result of the following?

```
x = sc.parallelize(
    [(1, "sandeep"), ("2", "sravani")])
y = sc.parallelize(
    [(1, "ryan"), (3, "giri")])

x.leftOuterJoin(y).collect()
```



1	sandeep
2	sravani

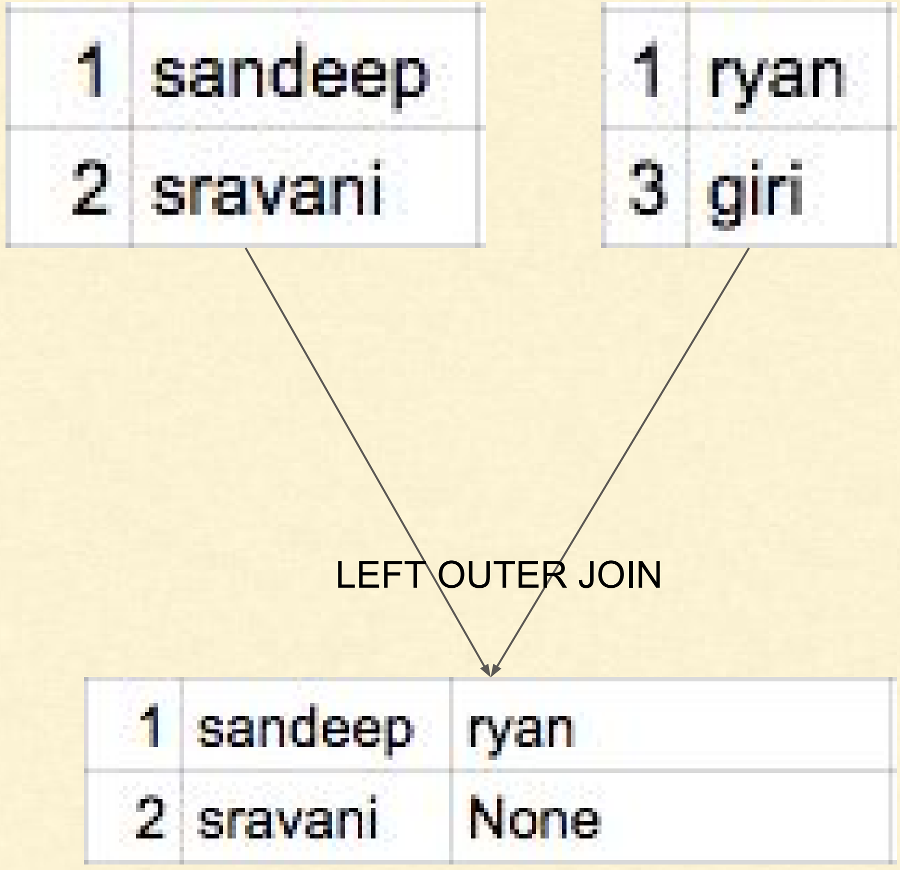
1	ryan
3	giri

LEFT OUTER JOIN

Questions - Set Operations

What will be the result of the following?

```
x = sc.parallelize(
    [(1, "sandeep"), ("2", "sravani")])
y = sc.parallelize(
    [(1, "ryan"), (3, "giri")])

x.leftOuterJoin(y).collect()
```



1	sandeep
2	sravani

1	ryan
3	giri

LEFT OUTER JOIN

1	sandeep	ryan
2	sravani	None

[(1, ('sandeep', 'ryan')), ('2', ('sravani', None))]

Transformations on Pair RDDs

rightOuterJoin(other, numPartitions=None)

Perform a right outer join of ***self*** and ***other***.

For each element (k, w) in ***other***, the resulting RDD will either contain all pairs (k, (v, w)) for v in this, or the pair (k, (None, w)) if no elements in ***self*** have key k.

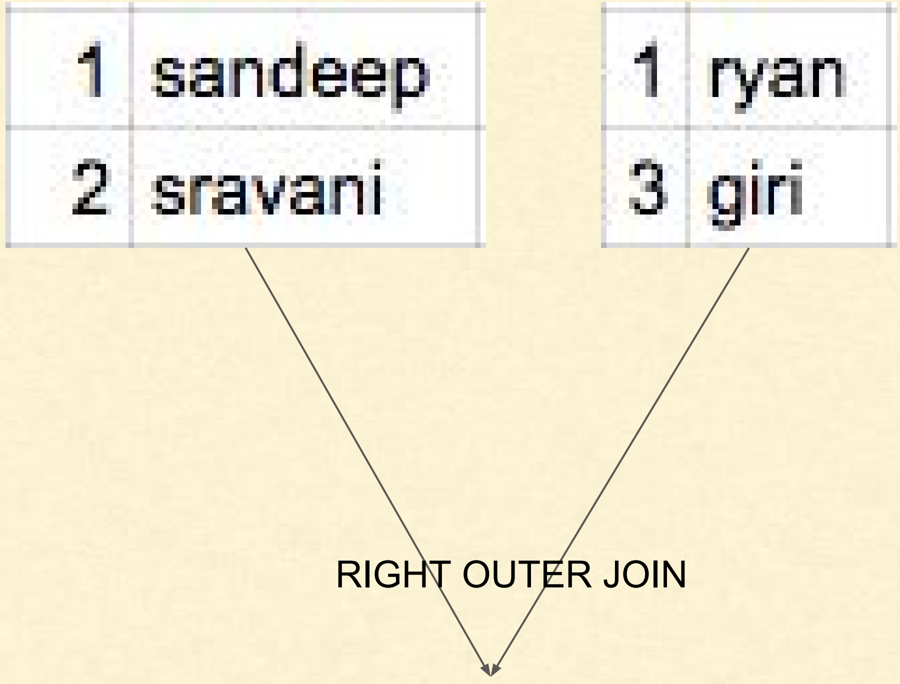
Hash-partitions the resulting RDD into the given number of partitions.

```
>>> x = sc.parallelize([("a", 1), ("b", 4)])
>>> y = sc.parallelize([("a", 2)])
>>> y.rightOuterJoin(x).collect()
[('a', (2, 1)), ('b', (None, 4))]
```

Questions - Set Operations

What will be the result of the following?

```
x = sc.parallelize(
    [(1, "sandeep"), ("2", "sravani")])
y = sc.parallelize(
    [(1, "ryan"), (3, "giri")])

x.rightOuterJoin(y).collect()
```



1	sandeep
2	sravani

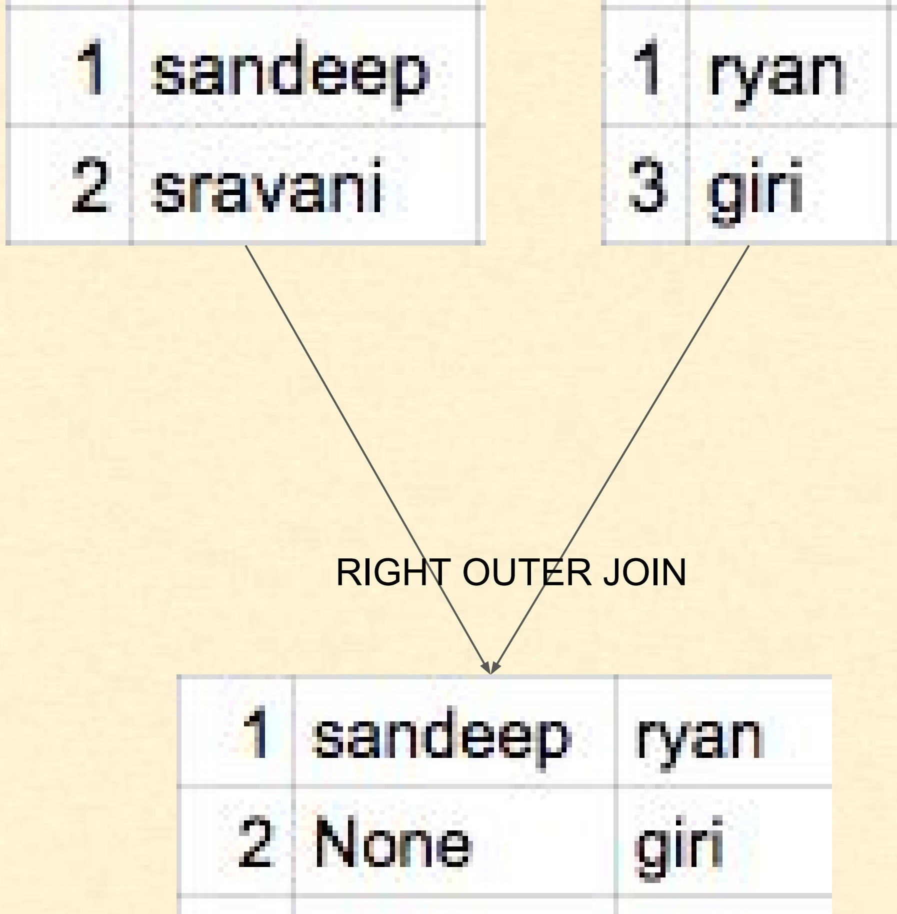
1	ryan
3	giri

Questions - Set Operations

What will be the result of the following?

```
x = sc.parallelize(
    [(1, "sandeep"), ("2", "sravani")])
y = sc.parallelize(
    [(1, "ryan"), (3, "giri")])

x.rightOuterJoin(y).collect()
```

[(1, ('sandeep', 'ryan')), (3, (None, 'giri'))]



1	sandeep
2	sravani

1	ryan
3	giri

RIGHT OUTER JOIN

1	sandeep	ryan
2	None	giri

Transformations on Pair RDDs

cogroup(other, numPartitions=None)

For each key k in self or other, return a resulting RDD that contains a tuple with the list of values for that key in self as well as other.

```
>>> x = sc.parallelize([("a", 1), ("b", 4)])
>>> y = sc.parallelize([("a", 2), ("a", 3)])
>>> cg = x.cogroup(y)
>>> cgl = cg.collect()
>>> [(x, tuple(map(list, y))) for x, y in sorted(list(cgl))]
[('a', ([1], [2, 3])), ('b', ([4], []))]
---
for x, y in list(cg.collect()):
    for z in y:
        for z1 in z:
            print str(x) + ":" + str(z1)
```

Transformations on Pair RDDs

cogroup(other, numPartitions=None)

For each key k in self or other, return a resulting RDD that contains a tuple with the list of values for that key in self as well as other.

```
>>> x = sc.parallelize([("a", 1), ("b", 4)])
>>> y = sc.parallelize([("a", 2), ("a", 3)])
>>> cg = x.cogroup(y)
>>> cgl = cg.collect()
```

Questions - Set Operations

What will be the result of the following?

```
>>> x = sc.parallelize([("a", 1),("a", 3), ("b", 4)])
>>> y = sc.parallelize([("a", 2)])
>>> [(x, tuple(map(list, y))) for x, y in
sorted(list(x.cogroup(y).collect()))]
```

Questions - Set Operations

What will be the result of the following?

```
>>> x = sc.parallelize([("a", 1),("a", 3), ("b", 4)])
>>> y = sc.parallelize([("a", 2)])
>>> [(x, tuple(map(list, y))) for x, y in
sorted(list(x.cogroup(y).collect()))]
```

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

Actions Available on Pair RDDs

countByKey()

Count the number of elements for each key, and return the result to the master as a dictionary.

```
>>> rdd = sc.parallelize([("a", 1), ("b", 1), ("a", 1), ('a', 10)])
>>> rdd.countByKey().items()
[('a', 3), ('b', 1)]
```

Actions Available on Pair RDDs

collectAsMap()

Return the key-value pairs in this RDD to the master as a dictionary.

```
>>> m = sc.parallelize([("1", 2), ("3", 4)]).collectAsMap()
>>> m['1']
2
>>> m['3']
4
```

Actions Available on Pair RDDs

lookup(key)

Return the list of values in the RDD for key. This operation is done efficiently if the RDD has a known partitioner by only searching the partition that the key maps to.

```
var l = 1 to 1000
var lr = sc.parallelize(l)
var lr1 = lr.zip(lr)
lr1.lookup(42) # slow
[42]
var sorted = lr1.sortByKey()
sorted.lookup(42)  # fast
[42]
sorted.lookup(1024)
[]
```

Actions Available on Pair RDDs

lookup(key)

Return the list of values in the RDD for key. This operation is done efficiently if the RDD has a known partitioner by only searching the partition that the key maps to.

```
>>> l = range(1000)
>>> rdd = sc.parallelize(zip(l, l), 10)
>>> rdd.lookup(42)  # slow
[42]
>>> sorted = rdd.sortByKey()
>>> sorted.lookup(42)  # fast
[42]
>>> sorted.lookup(1024)
[]
```

Project: Find top 10 IP address from /data/spark/project/access/

```
//Load the data
var accessLogs = sc.textFile("/data/spark/project/access/")
accessLogs.take(10)

//Keep only the lines which have IP
def containsIP(line:String):Boolean = return line matches "^([0-9\\.]+) .*$"
var ipaccesslogs = accessLogs.filter(containsIP)

//Extract only IP
def extractIP(line:String):(String) = {
    val pattern = "^([0-9\\.]+) .*$".r
    val pattern(ip:String) = line
    return (ip.toString)
}
var ips = ipaccesslogs.map(line => (extractIP(line),1));
```

Project: Find top 10 IP address from /data/spark/project/access/

```
//Count
var ipcounts = ips.reduceByKey((a,b) => (a+b))
var ipcountsOrdered = ipcounts.sortBy(f => f._2, false);
ipcountsOrdered.take(10)
```

Data Partitioning

- Layout data to minimize transfer
- Not helpful if you need to scan the dataset only once
- Helpful when dataset is reused multiple times in key-oriented operations
- Available for k-v rdds
- Causes system to group elements based on key
- Spark does not give explicit control which worker node has which key
- It lets program control/ensure which set of key will appear together
    - - based on some hash value for e.g.
    - - or you could range-sort

Data Partitioning - Example

Application Scenerio

- Keeps a large table **UserData**
  - which is an RDD of **(UserID, UserSubscriptionTopicsInfo)** pairs
- Periodically combines UserData with last five mins **events** (smaller)
  - say, a table of (UserID, LinkInfo) pairs for users who have clicked a link on a website in those five minutes.

Objective:

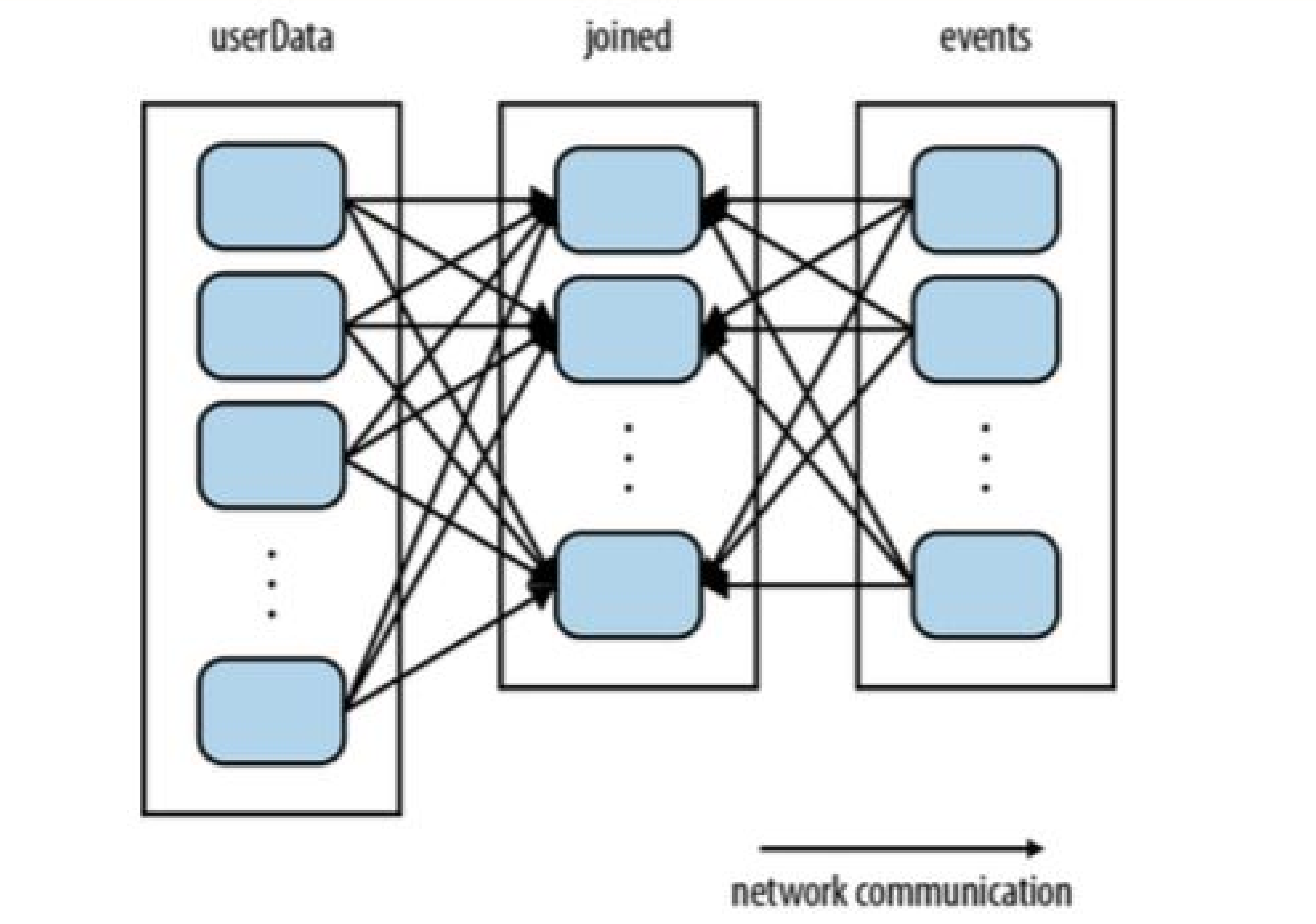
Count how many users visited a link that was not one of their subscribed topics.

Data Partitioning - Example

```
# Initialization code; we load the user info from a Hadoop SequenceFile on HDFS.
# This distributes elements of userData by the HDFS block where they are found,
# and doesn't provide Spark with any way of knowing in which partition a
# particular UserID is located.

sc = SparkContext(...)
userData = sc.sequenceFile("hdfs://...").persist()

# Function called periodically to process a logfile of events in the past 5 minutes
# we assume that this is a SequenceFile containing (UserID, LinkInfo) pairs.

def processNewLogs(logFileName):
    events = sc.sequenceFile(logFileName)
    joined = userData.join(events)
    # RDD of (UserID, (UserInfo, LinkInfo)) pairs

    offTopicVisits = joined.filter(
        lambda (userId, (userInfo, linkInfo)): not userInfo.topics.contains(linkInfo.topic)
    ).count()
 println("Number of visits to non-subscribed topics: " + offTopicVisits)
}
```

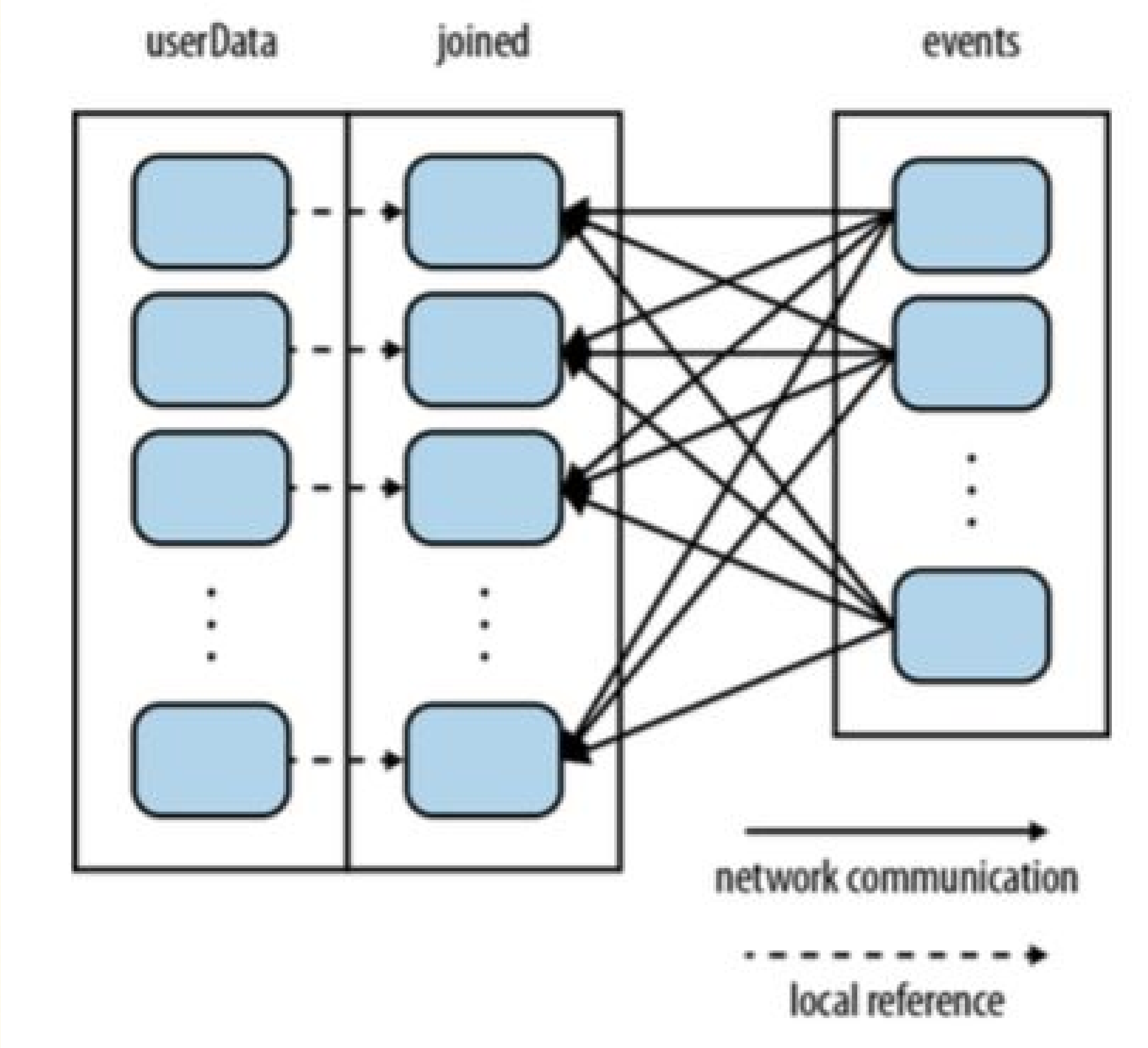
Data Partitioning - Example

Code will run fine as is, but it will be inefficient. Lot of network transfer



Data Partitioning - Example

Fix: userData = sc.sequenceFile("hdfs://...").**partitionBy(10)**.persist()
Note: partitionBy() is a transformation, so it always returns a new RDD



Data Partitioning - Example

```
var l = 1 to 1000
var lr = sc.parallelize(l)
var lr1 = lr.zip(lr)
var result = lr1.repartition(10)
```

Data Partitioning - Example

Determining an RDD’s Partitioner

rdd.partitioner

Operations That Benefit from Partitioning

- cogroup()
- groupWith(), groupByKey(), reduceByKey(),
- join(), leftOuterJoin(), rightOuter Join()
- combineByKey(), and lookup().

Data Partitioning - Customize

```
import urlparse
def hash_domain(url):
    return hash(urlparse.urlparse(url).netloc)

rdd.partitionBy(20, hash_domain) # Create 20 partitions
```

Data Partitioning - Example Problem - PageRank

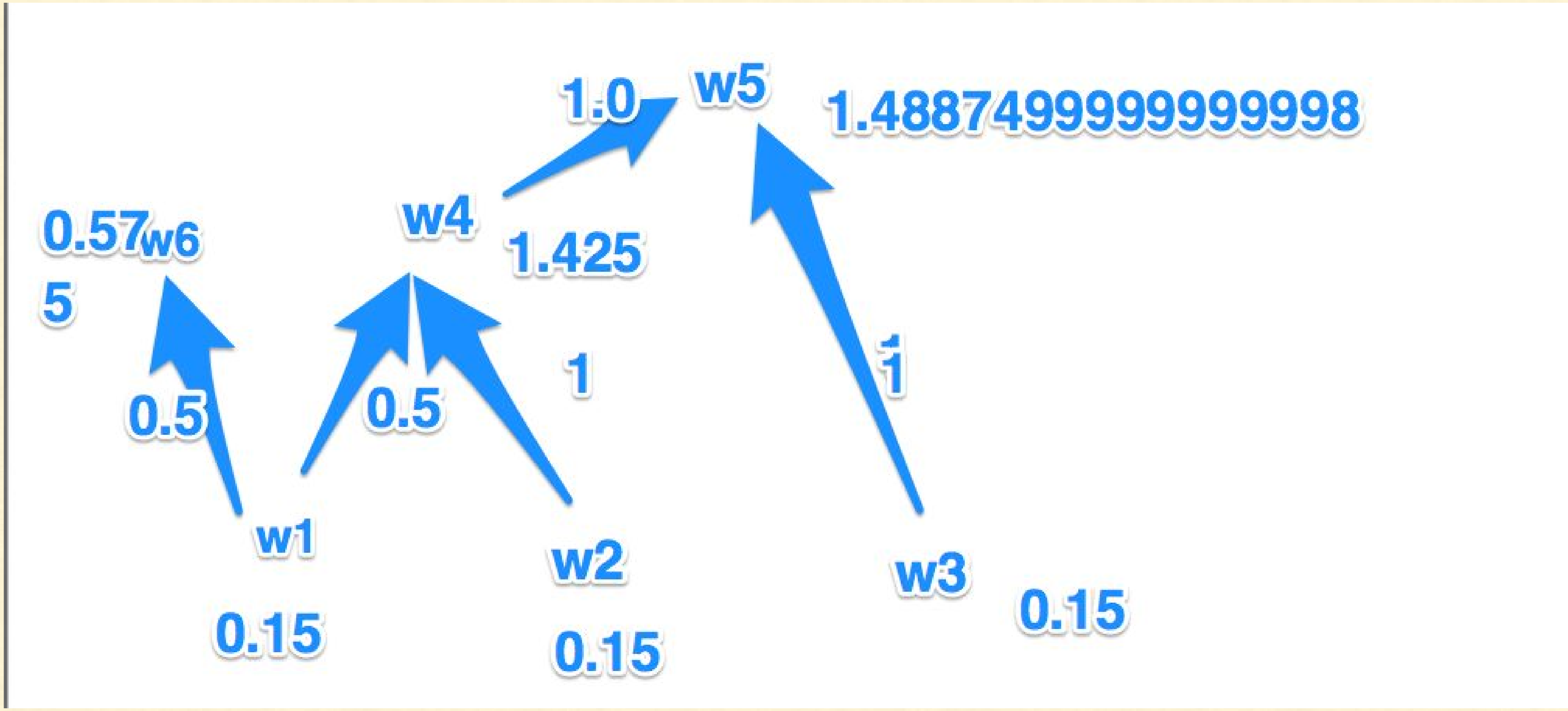
The PageRank algorithm aims to assign a measure of importance (a “rank”) to each document in a set based on how many documents have links to it. PageRank is an iterative algorithm that performs many joins, so it is a good use case for RDD partitioning.
The algorithm maintains two datasets: one of (pageID, link List) elements containing the list of neighbors of each page, and one of (pageID, rank) elements containing the current rank for each page. It proceeds as follows:

1. Initialize each page’s rank to 1.0.
2. On each iteration, have page p send a contribution of rank(p)/numNeighbors(p) to its neighbors (the pages it has links to).
3. Set each page’s rank to 0.15 + 0.85 * contributionsReceived.

The last two steps repeat for several iterations, during which the algorithm will converge to the correct PageRank value for each page. In practice, it’s typical to run about 10 iterations.

[(w1, w6, w4), (w2, w4), (w4, w5), (w3, w5)]

Data Partitioning - Example Problem - PageRank



Data Partitioning - Example Problem - PageRank

```
# Assume that our neighbor list was saved as a Spark objectFile
links = sc.objectFile("links").partitionBy(100).persist()

# Initialize each page's rank to 1.0; since we use mapValues, the resulting RDD
# will have the same partitioner as links
ranks = links.mapValues(lambda v: 1.0)
// Run 10 iterations of PageRank
for (i in range(0:10):
        contributions = links.join(ranks).flatMap(
            lambda (pageId, (links, rank)):
                links.map(dest => (dest, rank / links.size
            )
        ranks = contributions.reduceByKey(
    lambda (x, y): x + y).mapValues(v => 0.15 + 0.85*v)

// Write out the final ranks
ranks.saveAsTextFile("ranks")
```

Basics of RDD

Thank you!