

ADVANCED SPARK OPERATIONS

Working with Key-Value Pairs

- Spark's "distributed reduce" transformations act on RDDs of *key-value pairs*
- *Key-value* RDDs let you:
 - *Count up reviews for each product*
 - *Group together data with the same key*
 - *Group together two different RDDs*
- Python:

```
pair = (a, b)
        pair[0] # => a → key
        pair[1] # => b → value
```



= easy



= medium

Essential Core & Intermediate PairRDD Operations

General	Math / Statistical	Set Theory / Relational	Data Structure
<ul style="list-style-type: none">flatMapValuesgroupByKeyreduceByKeyreduceByKeyLocallyfoldByKeyaggregateByKeysortByKeycombineByKey	<ul style="list-style-type: none">sampleByKey	<ul style="list-style-type: none">cogroup (=groupWith)joinsubtractByKeyfullOuterJoinleftOuterJoinrightOuterJoin	<ul style="list-style-type: none">partitionBy
<ul style="list-style-type: none">keysvalues	<ul style="list-style-type: none">countByKeycountByValuecountByValueApproxcountApproxDistinctByKeycountApproxDistinctByKeycountByKeyApproxsampleByKeyExact		

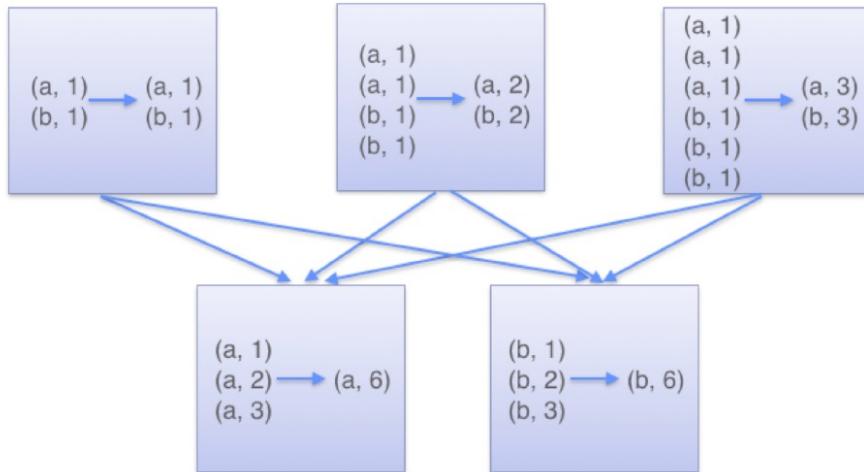


Some Key-Value Transformations

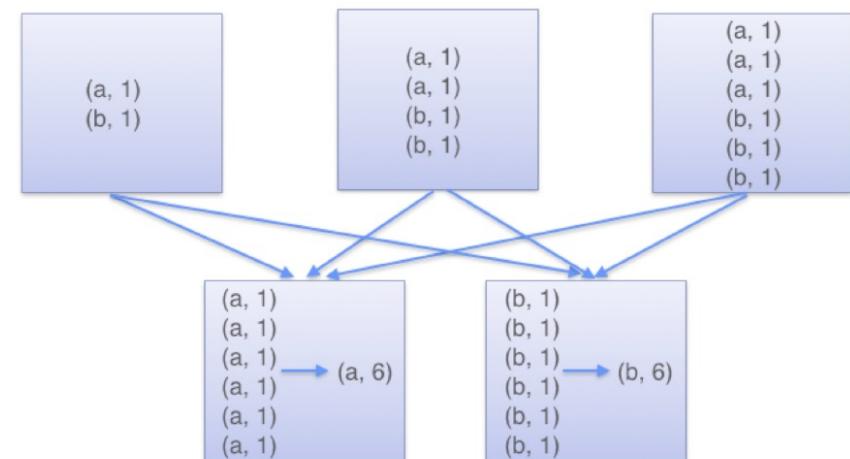
```
pets = sc.parallelize([('cat', 1), ('dog', 1), ('cat', 2)])  
  
pets.reduceByKey(lambda x, y: x + y)  
# => [(cat, 3), (dog, 1)]  
  
pets.groupByKey()  
# => [(cat, [2,1]), (dog, [1])]  
  
pets.sortByKey() # by default ascending parameter  
  
# => [(cat, 1), (cat, 2), (dog, 1)]  
  
pets.map(lambda x: x[0], x[1] *2))  
  
# => [(cat, 2), (dog, 2), (cat, 4)]  
  
pets.mapValues(lambda value: value *2) # convenient to work only on values  
  
# => [(cat, 2), (dog, 2), (cat, 4)]  
  
pets.filter(lambda x: (x[1] > 1))  
  
# => [(cat, 2)]
```

GroupByKey vs ReduceByKey

ReduceByKey



GroupByKey



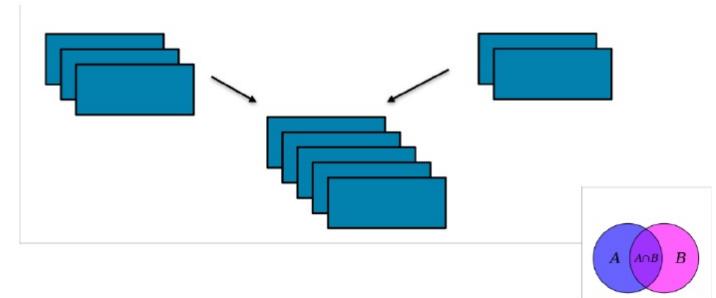
Both will give you the same answer:

- reduceByKey also automatically implements combiners on the map side
- groupByKey can cause a lot of data to be shuffled between workers
- reduceByKey is more efficient but has fixed memory space
- Prefer reduceByKey , combineByKey , foldByKey over groupByKey

UNION, JOIN (inner)

Union: Return a new RDD containing all items for two original RDDs. Duplicates are not culled

```
x= sc.parallelize([1,2,3])  
y= sc.parallelize([5,6])  
z= x.union(y) ➔ [1,2,3,5,6]
```

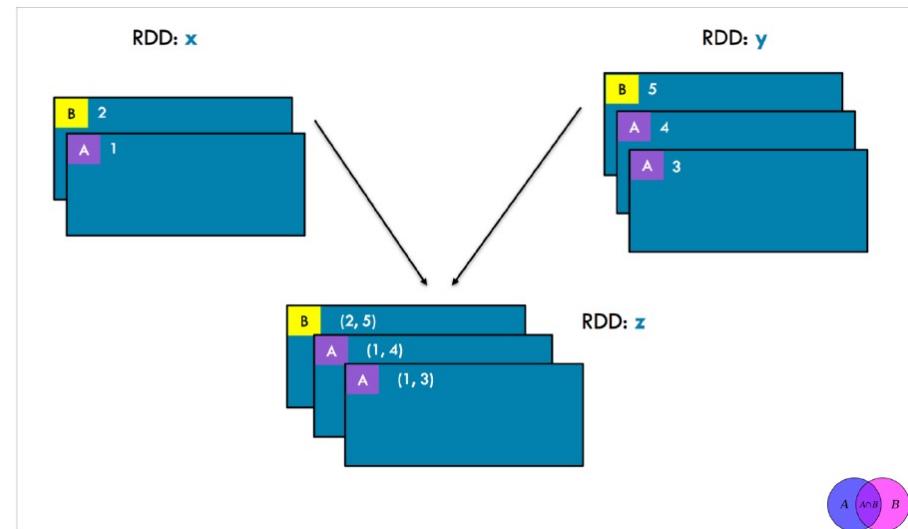


Join: Return a new RDD containing all pairs of elements having the same key in the original RDD – inner join between 2 RDDs

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

-- Another example

```
x= sc.parallelize([(1,2) (3,4), (3,6)])  
y= sc.parallelize([(3,9)])  
z = x.join(y) ➔ ??
```



UNION, JOIN (inner)

Union: Return a new RDD containing all items for two original RDDs. Duplicated are not culled

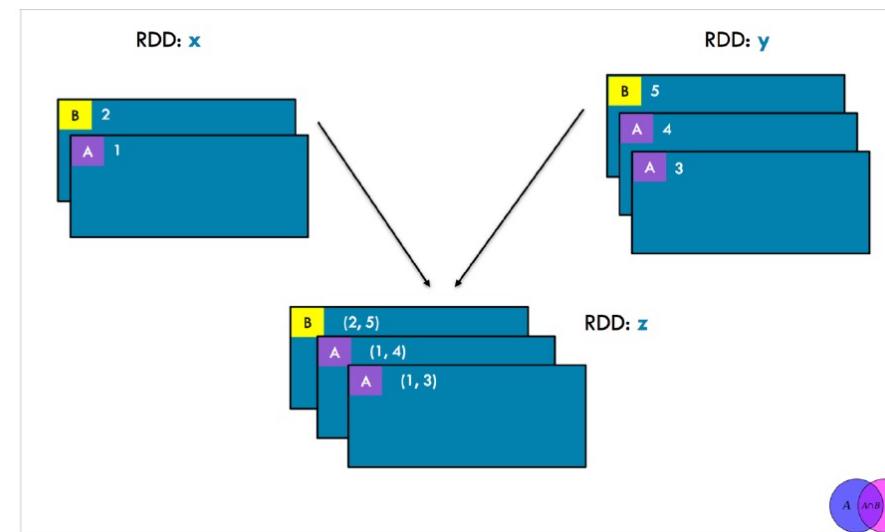
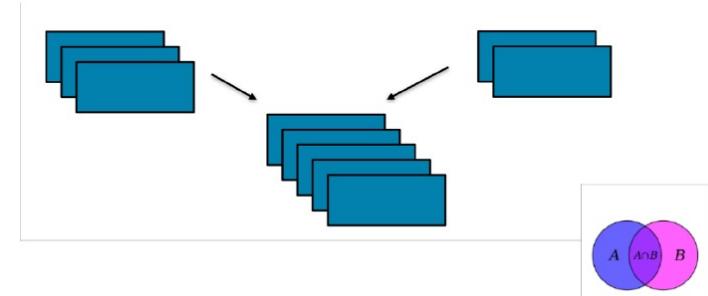
```
x= sc.parallelize([1,2,3])  
y= sc.parallelize([5,6])  
z= x.union(y) ➔ [1,2,3,5,6]
```

Join: Return a new RDD containing all pairs of elements having the same key in the original RDD – inner join between 2 RDDs

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

-- Another example

```
x= sc.parallelize([(1,2) (3,4), (3,6)])  
y= sc.parallelize([(3,9)])  
z = x.join(y) ➔ [(3,(4,9)), (3, (6,9))]
```

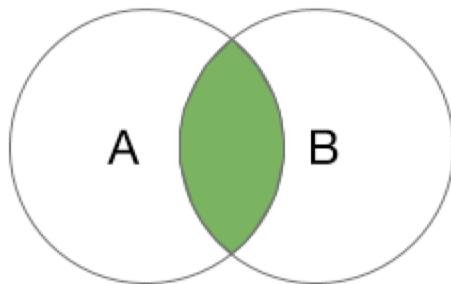


JOIN (outer)

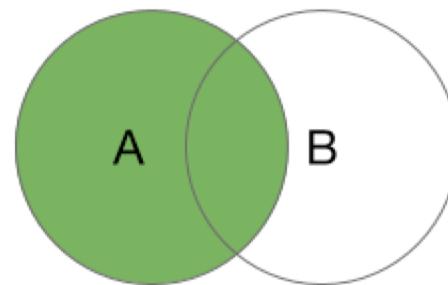
- Inner joins require a key to be present in both RDDs
- Outer joins do not require a key to be present in both RDDs:
 - leftOuterJoin
 - rightOuterJoin

```
x= sc.parallelize([('A',1),('B',2), ('C',1)])  
y= sc.parallelize([('A',3), ('C',2), ('D',4)])  
Z = x.leftOuterJoin(y) → [ ('A', (1,3)), ('C',(1,2)), ('B', (2,None))]  
Z = x.rightOuterJoin(y) → [ ('A', (1,3)), ('C', (1,2)), ('D', (4, None))]  
Z = x.join(y) → [ ('A', (1,3)), ('C', (1,2))]
```

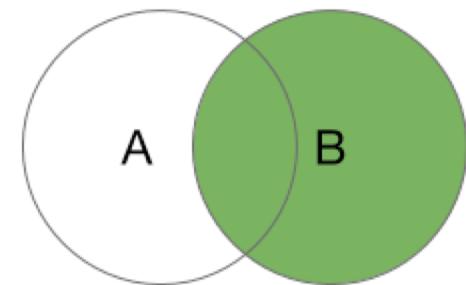
Beyond the traditional JOIN



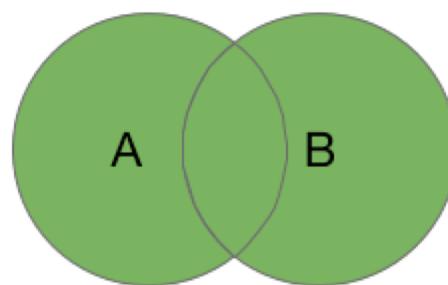
INNER JOIN



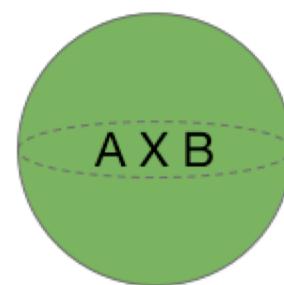
LEFT OUTER JOIN



RIGHT OUTER
JOIN



FULL OUTER
JOIN



CARTESIAN
(CROSS) JOIN

CoGroup

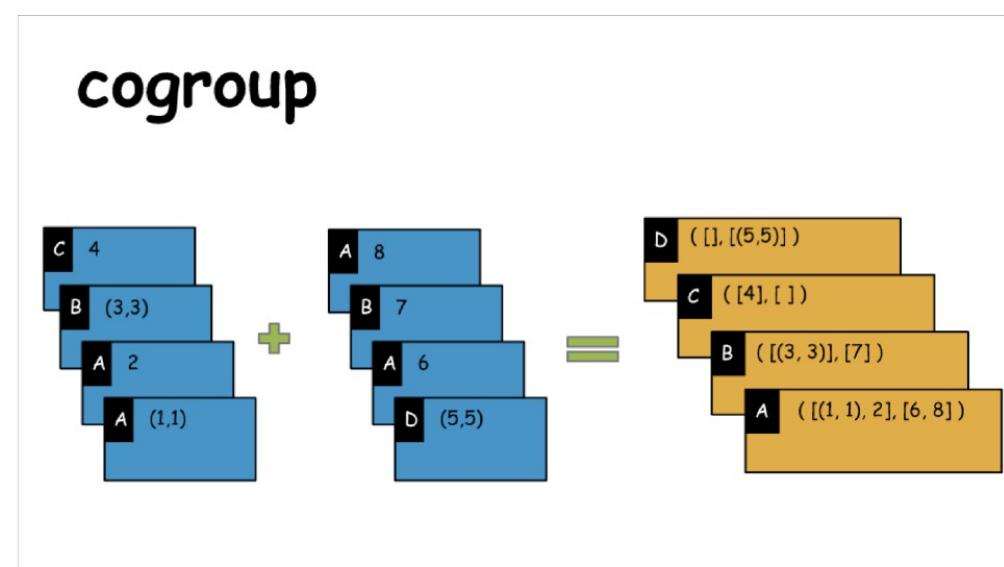
CoGroup: Given two RDDs sharing the same key type K, with the respective values types as V and W, the resulting RDD is of type [K, (iterable[V], Iterable[W])]

```
x= sc.parallelize([('A', (1,1)), ('A', 2), ('B', (3,3)), ('C',4) ])
```

```
y= sc.parallelize([('D', (5,5)), ('A', 6), ('B', 7), ('A',8) ])
```

```
z = x.cogroup(y)
```

```
# ("A", [(1,1), 2], [6,8]))  
# ("B", [(3,3)], [7])  
# ("C", [4], [])  
# ("D", [],[(5,5)])
```



Pair RDD actions

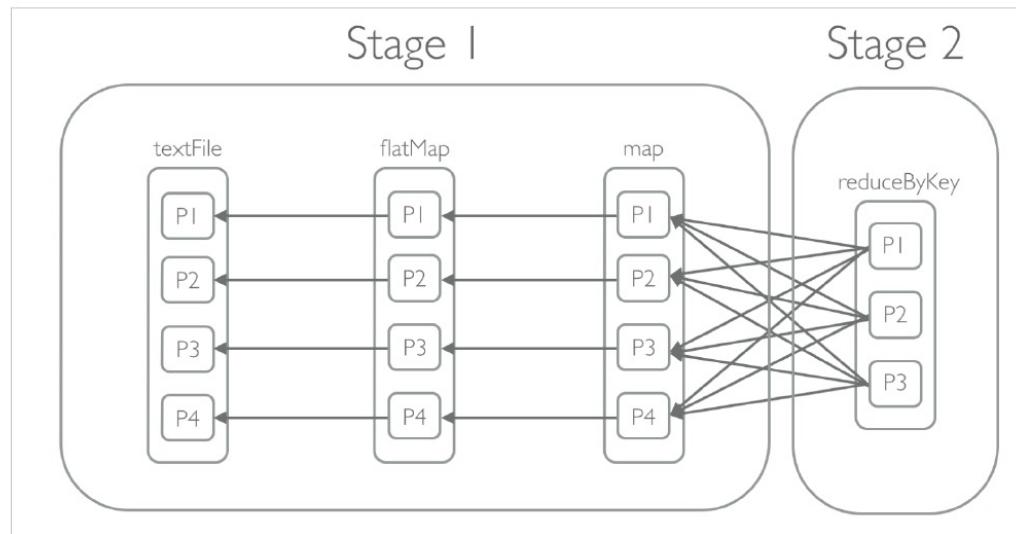
- Actions that make use of the key/value nature of pair RDDs

```
pets = sc.parallelize([('cat', 1), ('dog', 1), ('cat', 2)])
pets.countByKey() # count the number of elements for each key
# => {(cat, 2), (dog, 1)}
```

```
d = pets.collectAsMap() # Lookup through Python dictionary
d['dog']
# => 1
```

Partitions

- Each RDD is split up into a number of partitions
- Parallelism is determined by the number of partitions
 - `rdd.getNumberOfPartitions()`
- Why is important ?
 - Operations can be performed in parallel in each partition:
 - Textfile + flatMap + map operations can be performed in parallel in P1, P2, P3, P4
 - Operations that can run on the same partition are executed in stages



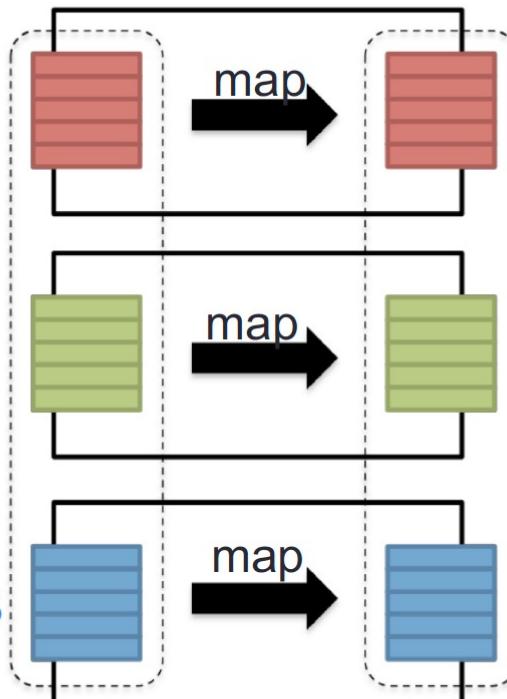
Narrow vs Wide transformations

Narrow transformation

- Input and output stays in same partition
- No data movement is needed

RDD
parent

Partition1



Partition2

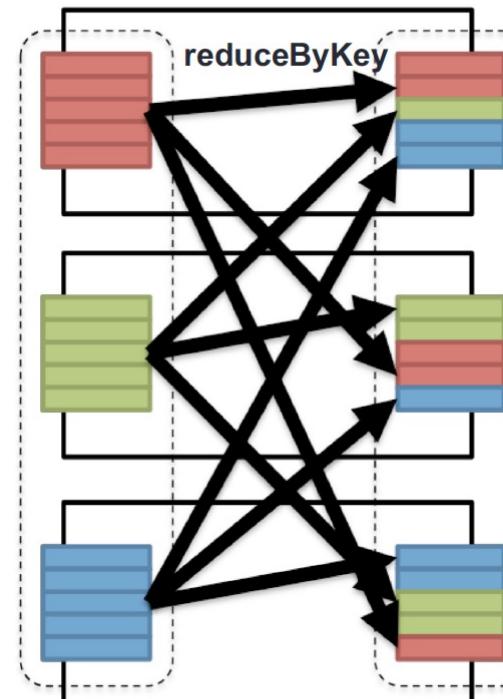
Partition3

e.g. map, flatmap, filter, union
each partition of the parent RDD is used
by at most 1 partition of the child RDD

Wide transformation

- Input from other partitions are required
- Data shuffling is needed before processing

RDD
child

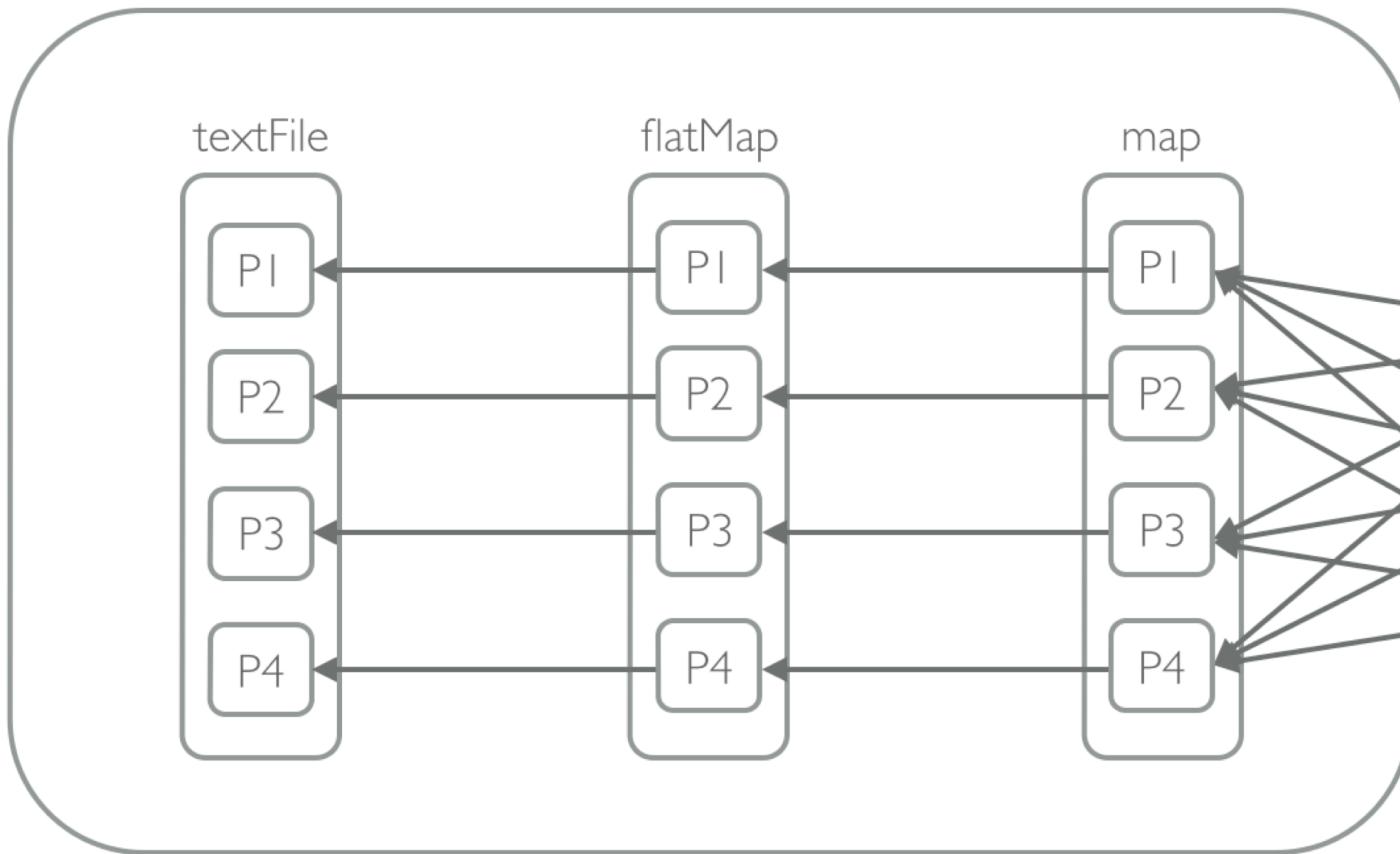


multiple children
depend on the RDD and
a new stage is defined

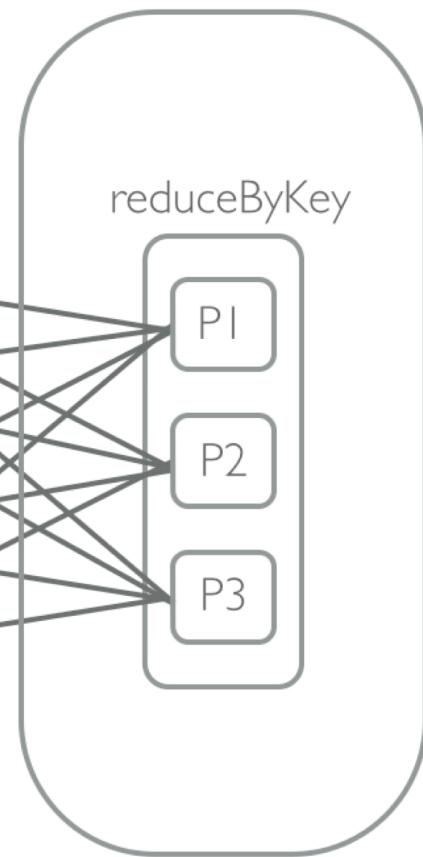
e.g. groupByKey, distinct or join
multiple child RDD partitions
may depend on a single parent
RDD partition

Stages

Stage 1



Stage 2



Controlling the Level of Parallelism

- All the pair RDD operations take an optional second parameter for number of tasks

```
words.reduceByKey(lambda x, y: x + y, 5)
```

```
words.groupByKey(5)
```

```
visits.join(pageViews, 5)
```

Accumulators

- Accumulators are shared variables:
 - Used to aggregate values from workers nodes back to the driver node
 - Only the driver program can read the acummulator's value
 - Can be used as a counter or summations

```
gold = sc.accumulator(0)
def count_medals(events):
    global gold, silver, bronze
    for event in events:
        if event.medal == 'GOLD':
            gold.add(1)
results = sc.textFile('olympics.csv').filter(lambda x: x.country == 'USA')
result.foreach(count_medals)
print gold.value
```

Broadcast Variables

- Spark's second type of shared variable
- Allows the program to efficiently send a large and read-only value to all the workers nodes.
 - For example, use it if your application needs to send a large, read-only lookup table of a large feature vector in a ML to all the nodes

```
import nltk  
  
stopwords= set(ntlk.corpus.stopwords.words('english'))  
  
stopwords =sc.broadcast(stopwords)  
  
if word in stopwords.value:  
    pass
```

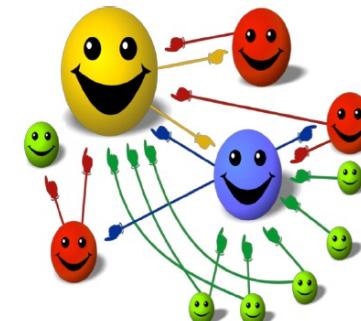
Example: PageRank

- Good example of a more complex algorithm
 - Multiple stages of map & reduce
- Benefits from Spark's in-memory caching
 - Multiple iterations over the same data

Basic Idea

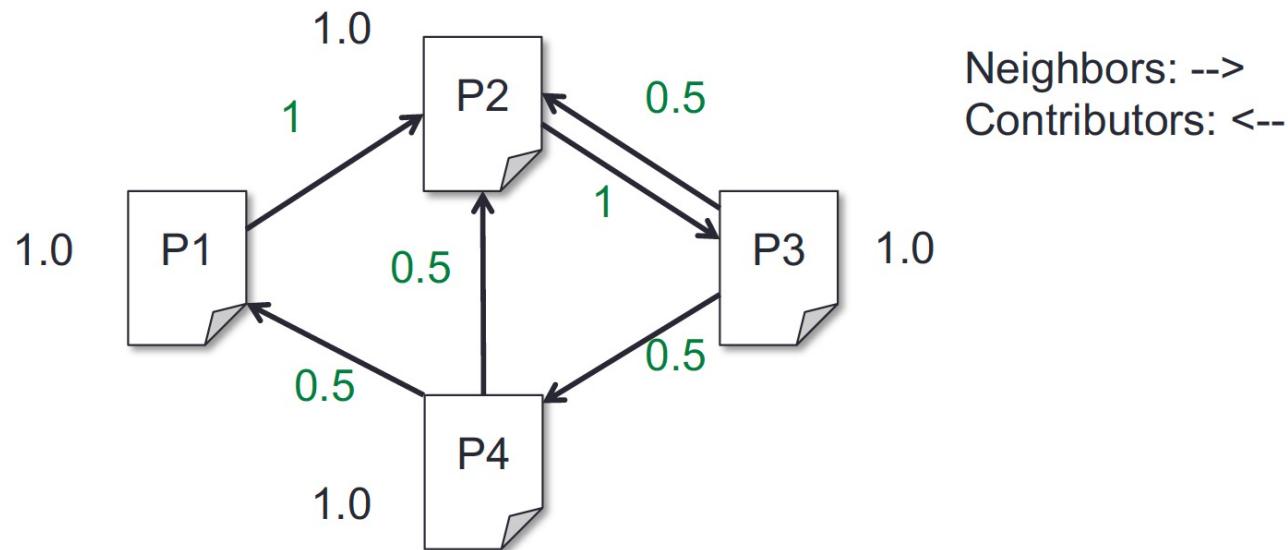
- **PageRank (PR)** is an algorithm used by Google Search to rank websites in their search engine results.
- Give pages ranks (scores) based on links to them
 - Links from many pages → high rank
 - Link from a high-rank page → high rank

Overview of PageRank:
<https://en.wikipedia.org/wiki/PageRank>



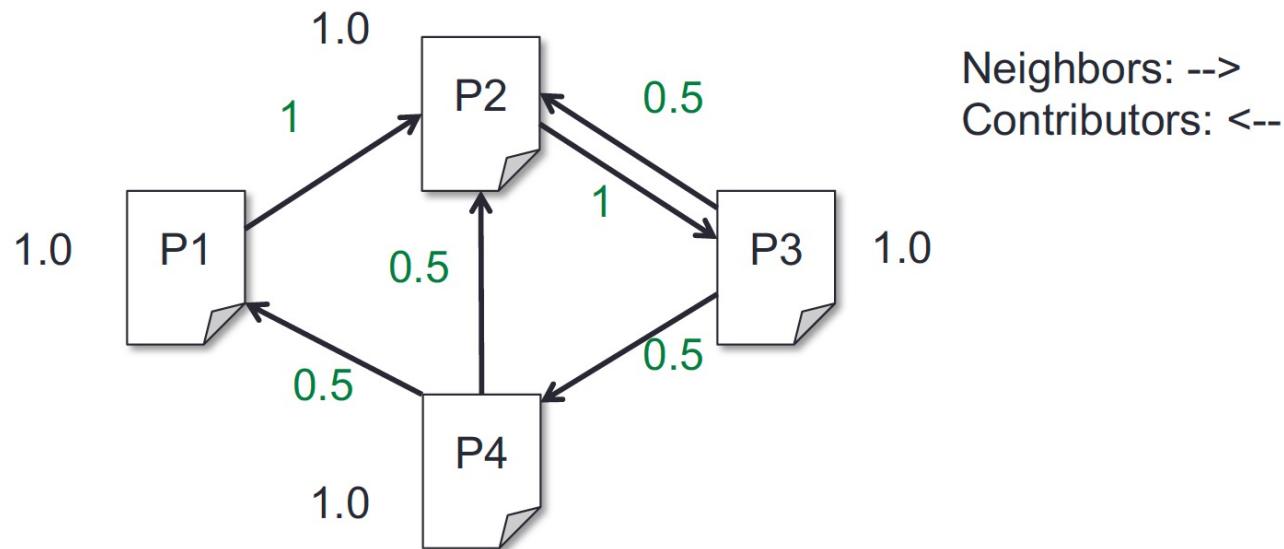
Algorithm

1. Start each page at a rank of 1
2. On each iteration, have page p send a contribution of $\text{rank}(p)/\text{numNeighbors}(p)$ to its neighbors (the pages it has links to).
3. Set each page's rank to $0.15 + 0.85 \times \text{contribs}$



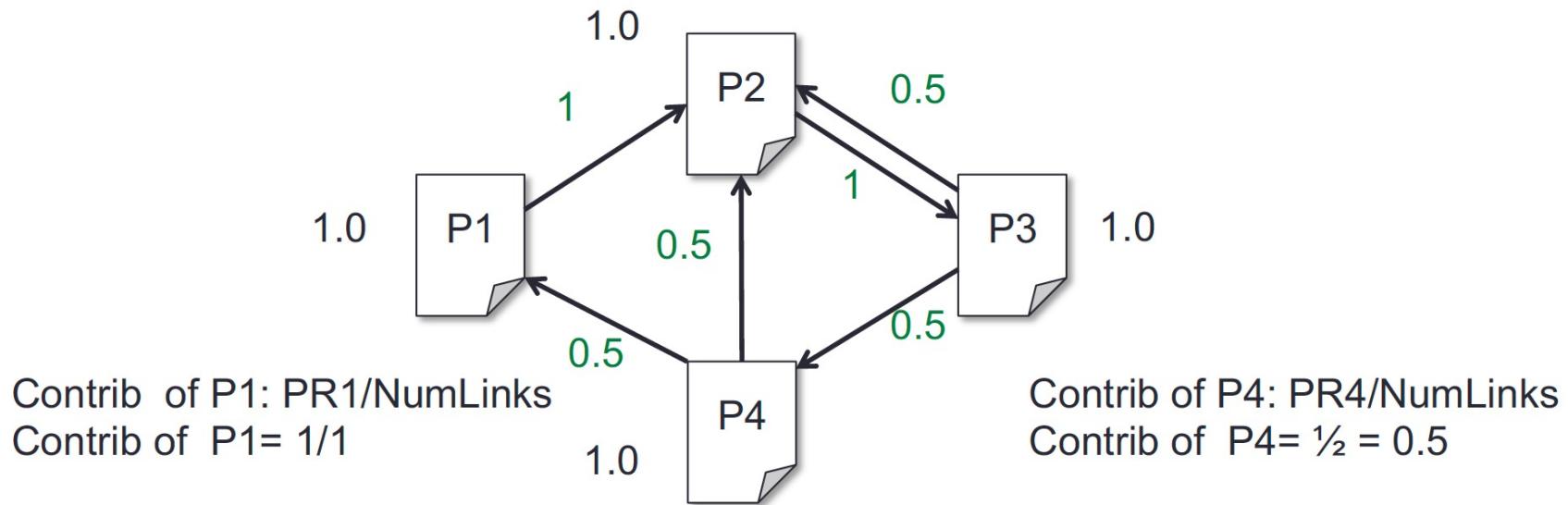
Algorithm

1. Start each page at a rank of 1
2. On each iteration, have page p send a contribution of $\text{rank}(p)/\text{numNeighbors}(p)$ to its neighbors (the pages it has links to).
3. Set each page's rank to $0.15 + 0.85 \times \text{contribs}$



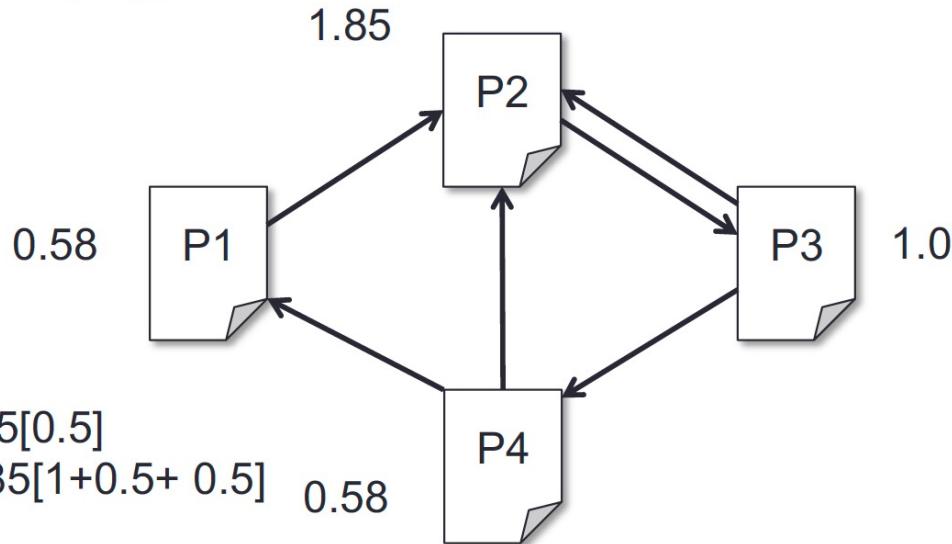
Algorithm

1. Start each page at a rank of 1
2. On each iteration, have page p send a contribution of $\text{rank}(p)/\text{numNeighbors}(p)$ to its neighbors (the pages it has links to “ \rightarrow ”).
3. Set each page’s rank to $0.15 + 0.85 \times \text{contribs}$



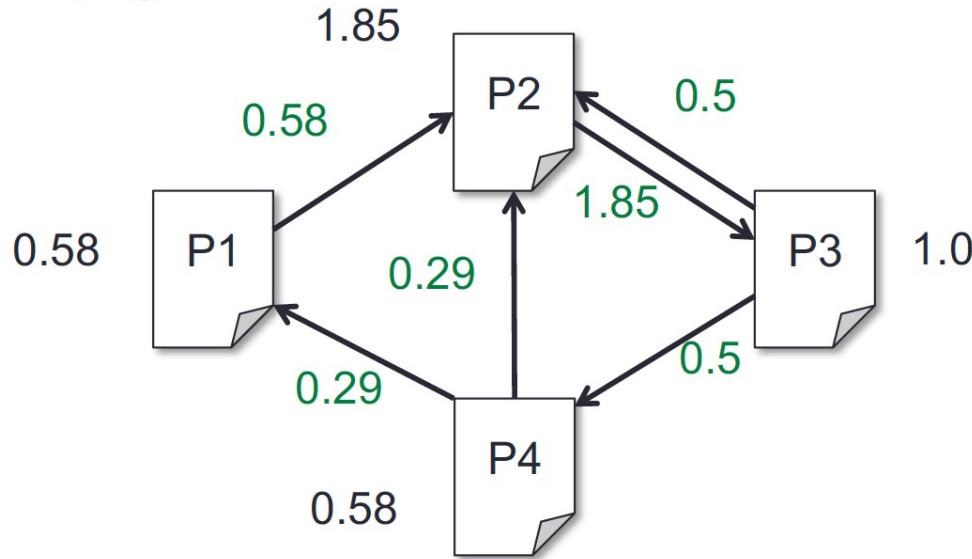
Algorithm

1. Start each page at a rank of 1
2. On each iteration, have page p send a contribution of $\text{rank}(p)/\text{numNeighbors}(p)$ to its neighbors (the pages it has links to).
3. Set each page's rank to $0.15 + 0.85 \times \text{contribs}$



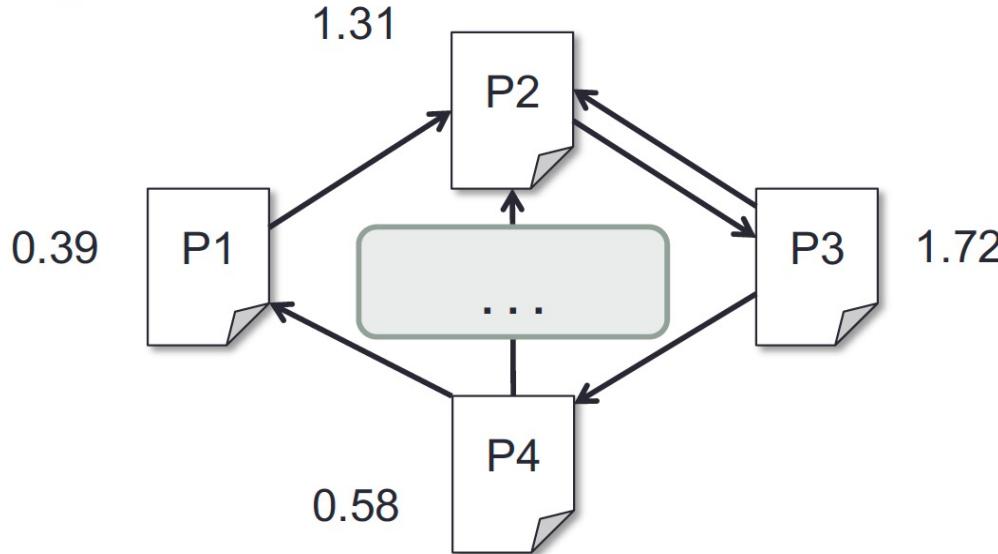
Algorithm

1. Start each page at a rank of 1
2. On each iteration, have page p send a contribution of $\text{rank}(p)/\text{numNeighbors}(p)$ to its neighbors (the pages it has links to).
3. Set each page's rank to $0.15 + 0.85 \times \text{contribs}$



Algorithm

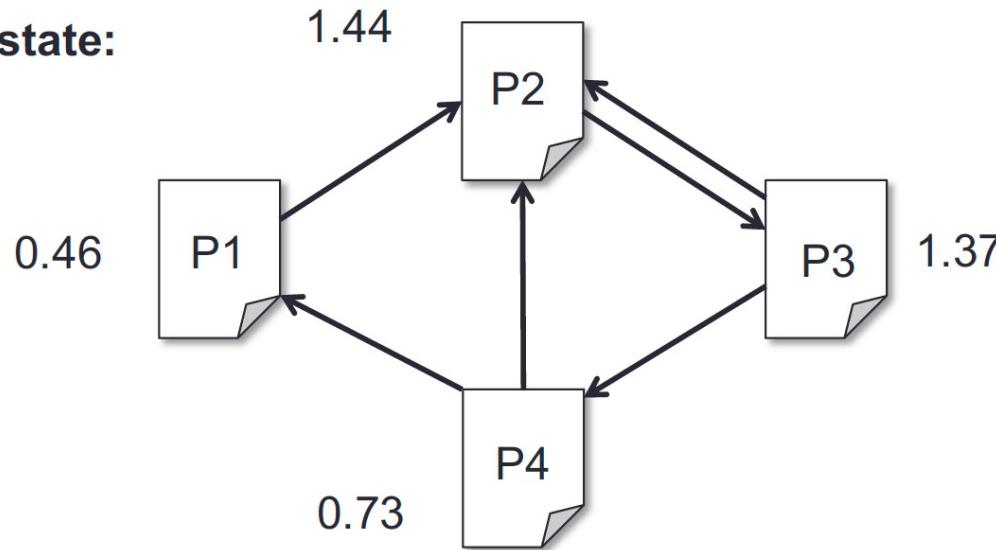
1. Start each page at a rank of 1
2. On each iteration, have page p send a contribution of $\text{rank}(p)/\text{numNeighbors}(p)$ to its neighbors (the pages it has links to).
3. Set each page's rank to $0.15 + 0.85 \times \text{contribs}$



Algorithm

1. Start each page at a rank of 1
2. On each iteration, have page p send a contribution of $\text{rank}(p)/\text{numNeighbors}(p)$ to its neighbors (the pages it has links to).
3. Set each page's rank to $0.15 + 0.85 \times \text{contribs}$

Final state:



PageRank Implementation

```
links = # RDD of (url, neighbors) pairs
ranks = # RDD of (url, rank) pairs

def compute_contribs(pair):
    [url, [links, rank]] = pair # split key-value pair
    return [(dest, rank/len(links)) for dest in links]

for i in range(NUM_ITERATIONS):
    contribs = links.join(ranks).flatMap(compute_contribs)
    ranks = contribs.reduceByKey(lambda x, y: x + y) \
        .mapValues(lambda x: 0.15 + 0.85 * x)

ranks.saveAsTextFile(...)
```

PageRank Implementation

```
links = sc.parallelize[  
    ("p1_url","p2"), ("p2_url","p3"),  
    ("p3_url", ("p2","p4"), ("p4_url",("p1","p2"))]  
ranks = sc.parallelize[("p1_url",1), ("p2_url",1), ("p3_url",1), ("p4_url",1)]  
  
def compute_contribs(pair):  
    [url, [links, rank]] = pair # split key-value pair  
    return [(dest, rank/len(links)) for dest in links]  
  
for i in range(NUM_ITERATIONS):  
    contribs = links.join(ranks).flatMap(compute_contribs)  
    ranks = contribs.reduceByKey(lambda x, y: x + y)  
        .mapValues(lambda x: 0.15 + 0.85 * x)  
  
ranks.saveAsTextFile(...)
```

PageRank Implementation

join	[url, [links, rank]]	(dest, rank/len(links))	flatMap
("p1_url", ("p2",1)), ("p2_url", ("p3", 1), ("p3_url", (("p2","p4"), 1), ("p4_url", (("p1","p2"), 1))	["p1_url", ["p2"] ,1], ["p2_url", ["p3"], 1], ["p3_url", ["p2","p4"], 1], ["p4_url", ["p1","p2"], 1]	P1 contribute to P2 → 1 P2 contribute to P3 → 1 P3 contribute to P2 and P4 → 0.5 P4 contribute to P1 and P2 → 0.5	[(p2, 1], (p3,1), (p2, 0.5), (p4, 0.5), (p1 , 0.5), (p2, 0.5)]
reduceByKey	reduceByKey		
[(p2, (1 +0.5 +0.5), (p3,1), (p4, 0.5), (p1 , 0.5)	[(p2,2), (p3,1), (p4, 0.5), (p1 , 0.5)		
mapValues			
[(p2, (0.15 + 0.85 *2)), (p3, (0.15 + 0.85 *1)), (p4, (0.15 + 0.85 *0.5)), (p1 , (0.15 + 0.85 *0.5))]			
mapValues	[(p2, 1.85), (p3, 1)), (p4, 0.58)), (p1 , 0.58)		

Logistic Regression

```
# Iterative machine learning algorithm  
# Find best hyperplane that separates two sets of points in a  
# multi-dimensional feature space. Applies MapReduce operation  
# repeatedly to the same dataset, so it benefits greatly  
# from caching the input in RAM  
  
points = spark.textFile(...).map(parsePoint).cache()  
w = numpy.random.ranf(size = D) # current separating plane  
for i in range(ITERATIONS):  
    gradient = points.map(lambda p: (1 / (1 + exp(-p.y*(w.dot(p.x))))- 1) *p.y  
                          *p.x).reduce(lambda a, b: a + b)  
    w -= gradient  
print ("Final separating plane", w)
```

Best Practices

- Level of parallelism recommended: 3 tasks per CPU core.
- Reduce working set size
- Avoid groupByKey for associative operations
- Avoid reduceByKey when the input and output value types are different
- Avoid the flatMap-join-groupBy pattern
- Python memory overhead
- Use broadcast variables
- Cache judiciously
- Don't collect large RDDs
- Minimize amount of data shuffled
- Know the standard library
- Use dataframes