Advanced Spark Operations

Rosa Filgueira

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

TRANSFORMATIONS

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

ACTIONS

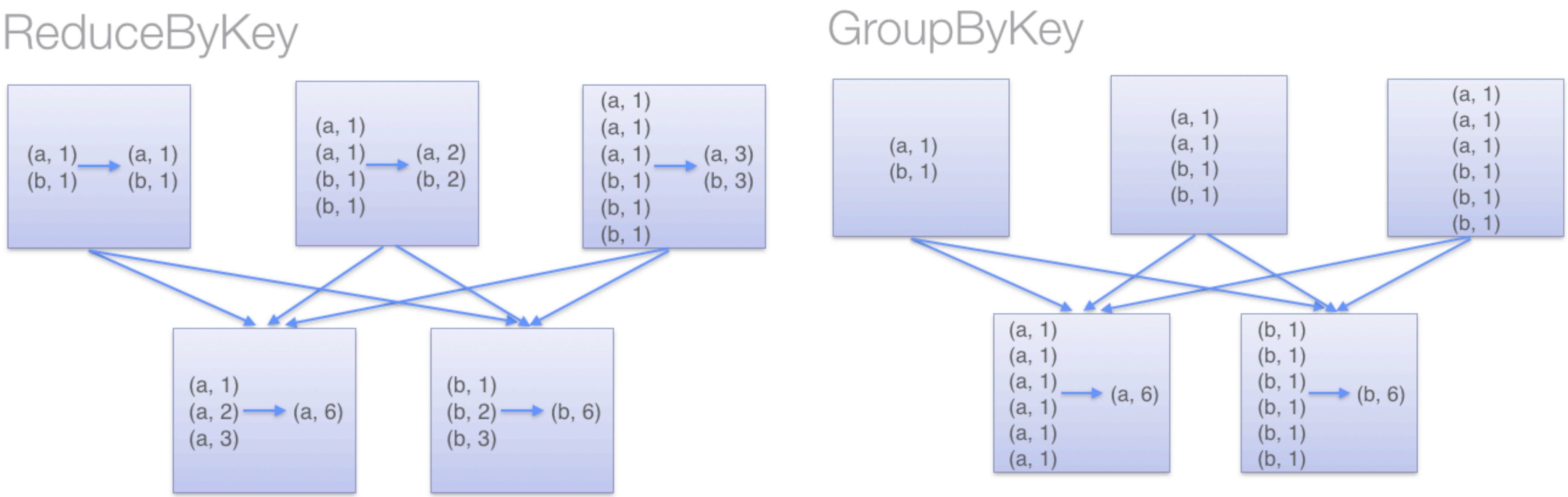
- keys
- values

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

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, 3), (dog, 1)}

pets.sortByKey() # by default ascending parameter

# => {(cat, 1), (cat, 2), (dog, 1)}

pets.map(lambda (key,value): (key, value *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 (key,value): (value > 1))

# => {(cat, 2)}
```

GroupByKey vs ReduceByKey



Both will give you the same answer:

→ `reduceByKey` also automatically implements combiners on the map side

→ `groupByKey` can cause a lot of data to shuffled between workers

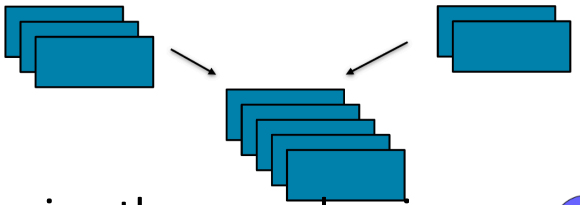
→ `reduceByKey` is more efficient but has fixed memory space

→ Prefer `reduceByKey` , `combineByKey`, `foldByKey` over `groupByKey`

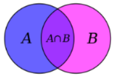
UNION, JOIN (inner)

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

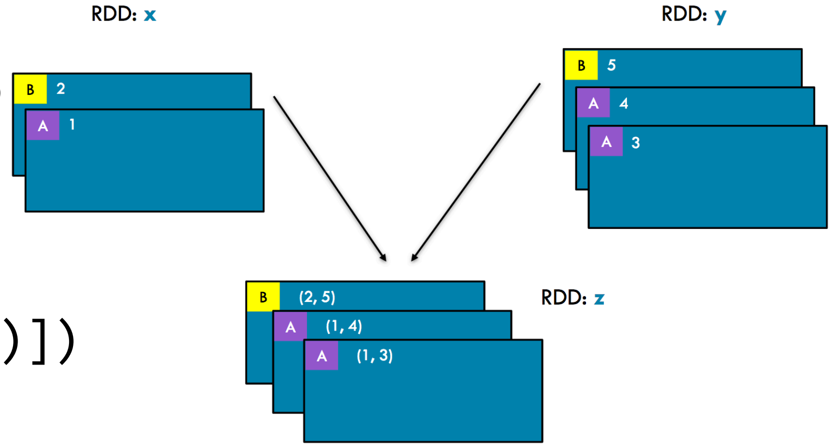
```
x= sc.parallelize([1,2,3])
y= sc.parallelize([5,6])
z= x.union(y) ➔ [1,2,3,5,6]
```



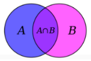
Join: Return a new RDD containing all pairs of elements having the same key in the original RDD – inner join between 2 RDDs




```
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) → ?
```

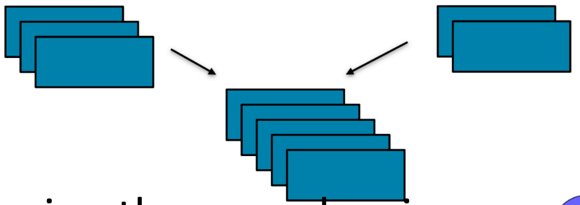


More details at: http://training.databricks.com/visualapi.pdf

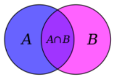
UNION, JOIN (inner)

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

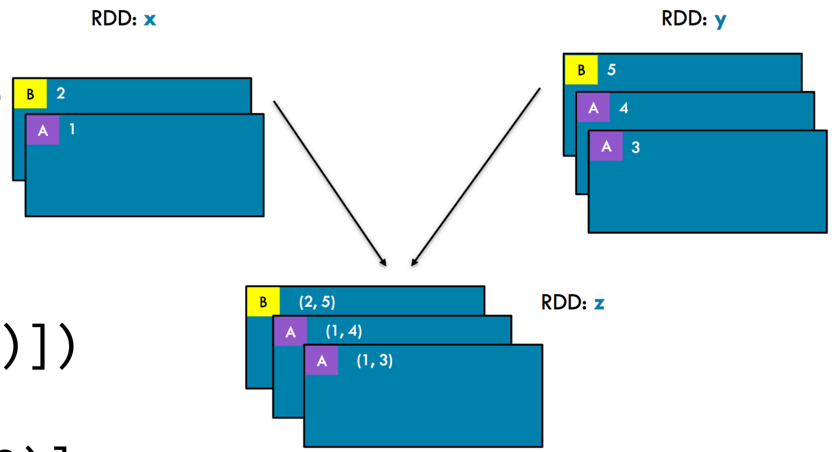
```
x= sc.parallelize([1,2,3])
y= sc.parallelize([5,6])
z= x.union(y) ➔ [1,2,3,5,6]
```



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

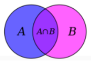



```
x= sc.parallelize([(“a”,1), (“b,2”)])
y= sc.parallelize([(“a”,3), (“a,4”), (“b,5”)])
z= x.join(y) ➔ [(“a”, (1,3), (“a”, (1,4)),
(“b”,(2,5))]
```



-- 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)]
```



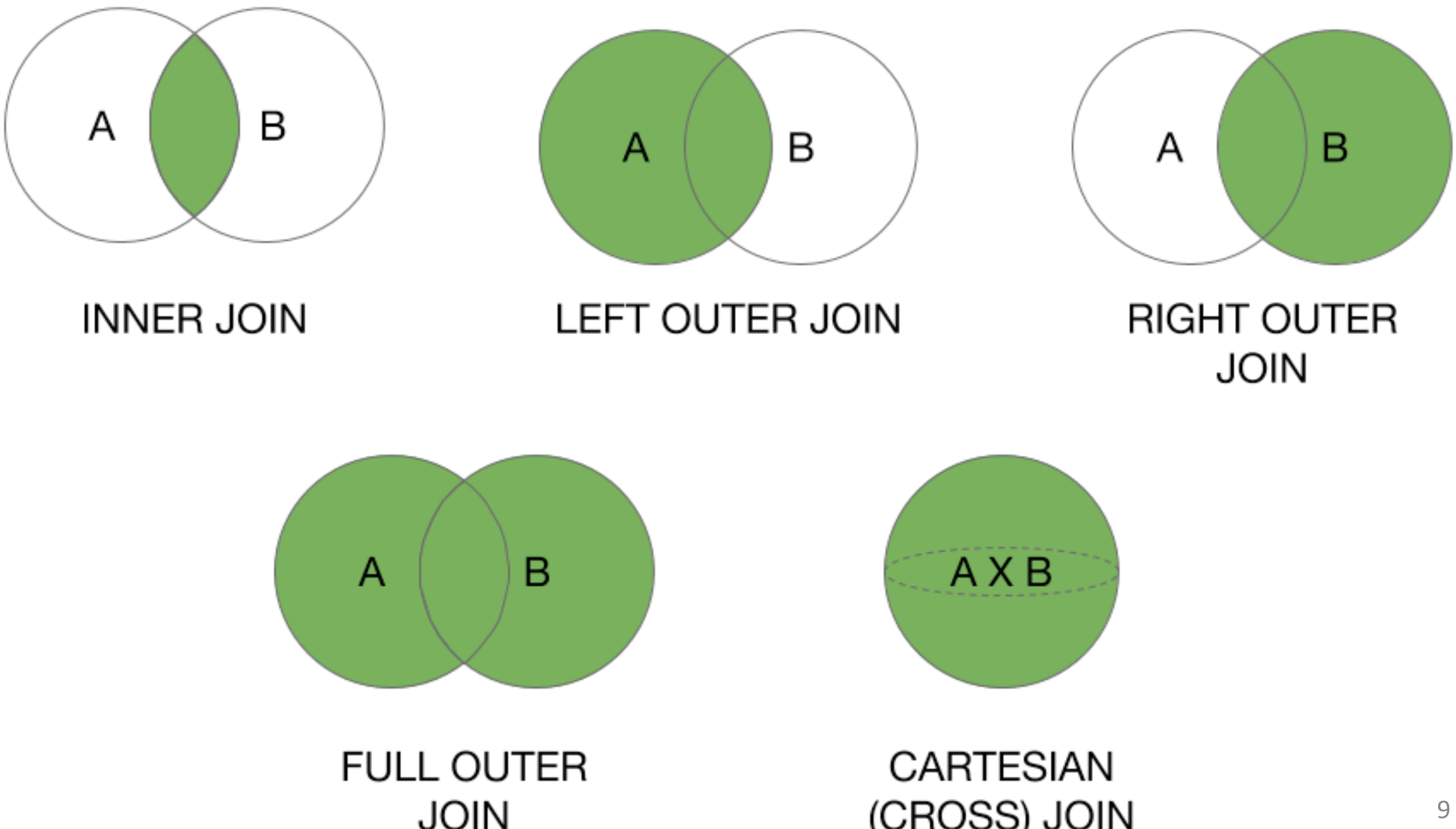
More details at: http://training.databricks.com/visualapi.pdf

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:
  - lefOuterJoin
  - rightOutherJoin

```
x= sc.parallelize([(‘A’,1),(‘B’,2), (‘C’,1))])
y= sc.parallelize([(‘A’,3), (‘C’,2), (‘D’,4)])
Z = x.lefOuter(x) → [(‘A’, (1,3)), (‘C’,(1,2)), (‘B’, (2,None))]
Z = x.rightOuter(x) → [(‘A’, (1,3), (‘C’, (1,2), (‘D’, (4, None))]
Z = x.join(x) → [(‘A’, (1,3), (‘C’, (1,2))]
```

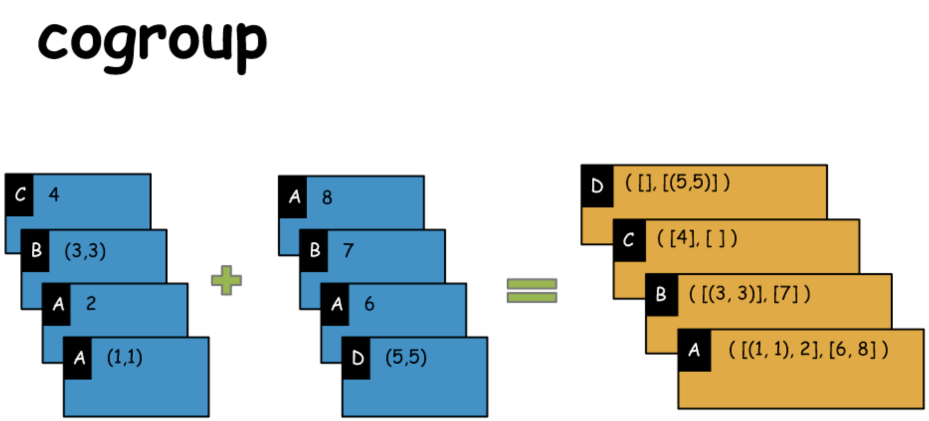
Beyond the traditional JOIN



CoGroup

CoGroup: Given two RDDs sharing the same key type K, with the respesctive 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)])
```



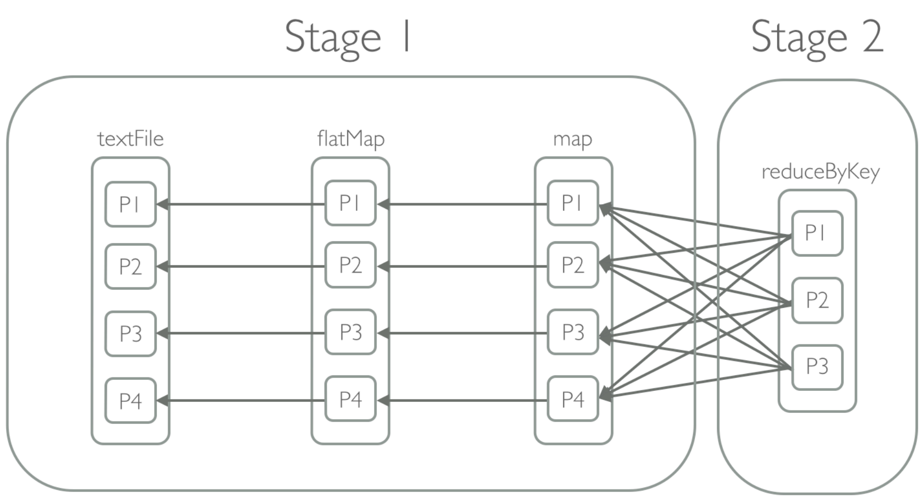
Pair RDD actions

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

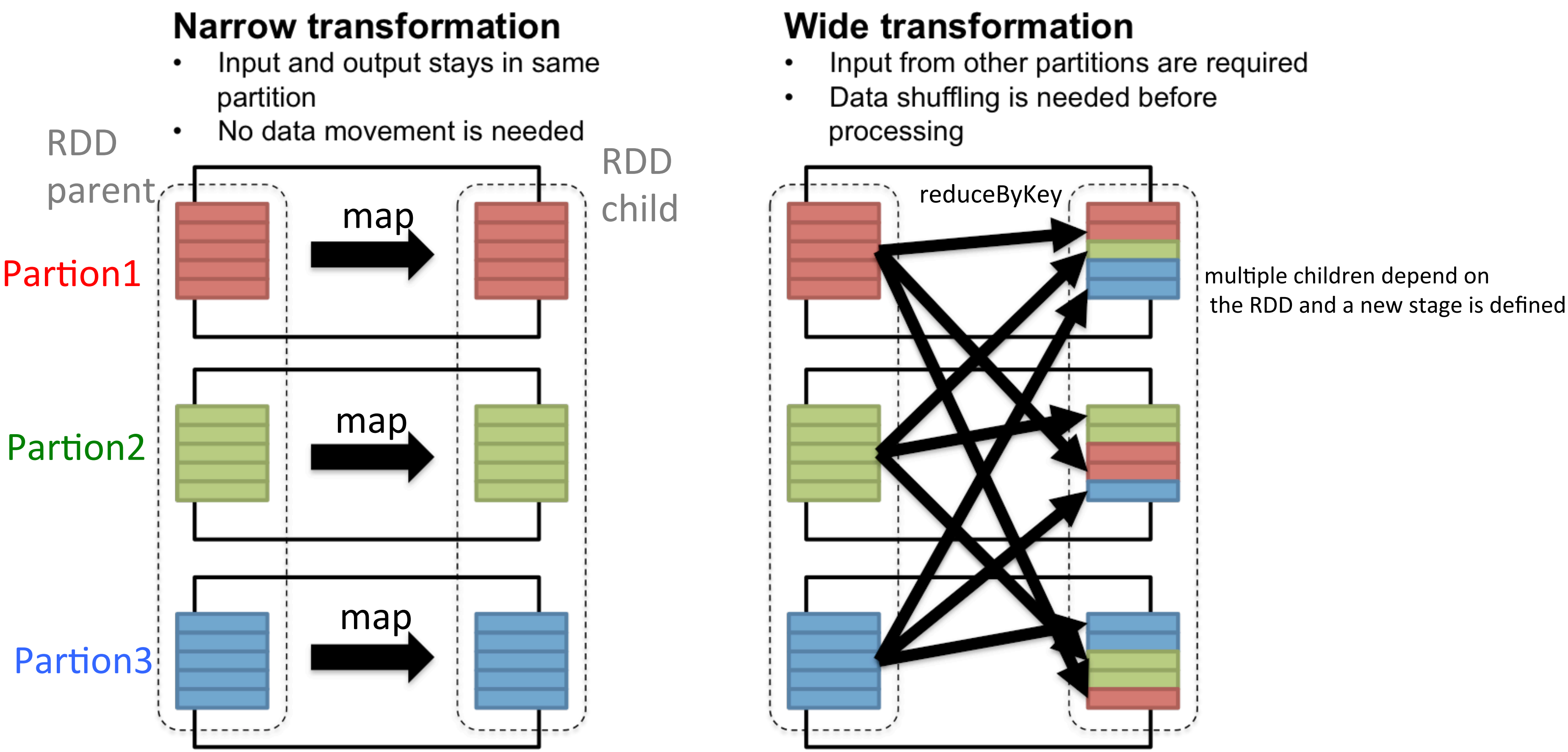
```
pets = sc.parallelize([(“cat”, 1), (“dog”, 1), (“cat”, 2)])
pets.countByKey() # count the number of elements for each key
# => {(cat, 2), (dog, 1)}

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



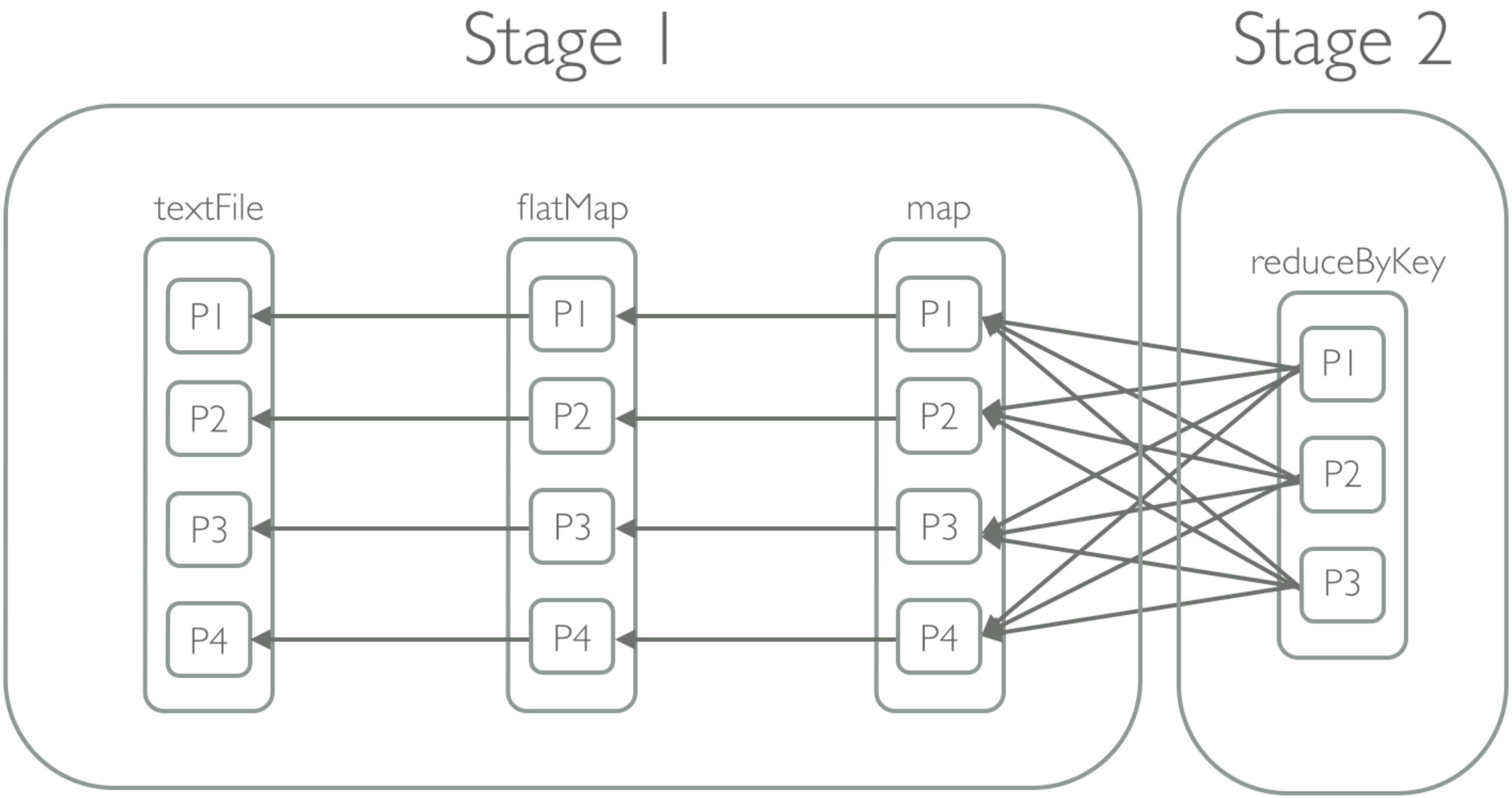
Narrow vs Wide transformations



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

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

Stages


Stage 1
Stage 2
textFile
flatMap
map
reduceByKey
P1
P2
P3
P4
P1
P2
P3
P4
P1
P2
P3
P4
P1
P2
P3

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_medas)

print gold.value
```


https://spark.apache.org/docs/latest/programming-guide.html#accumulators

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.
  - Use it if you application needs to send a large, read-only lookup table o a large feature vector in a ML to al the nodes

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

Example: PageRank

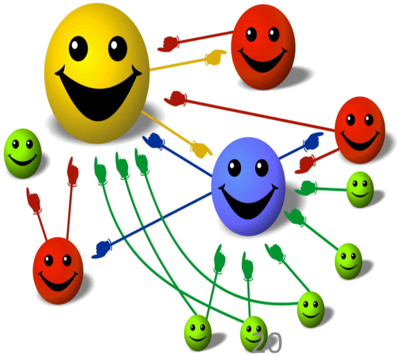
Why PageRank?

- 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

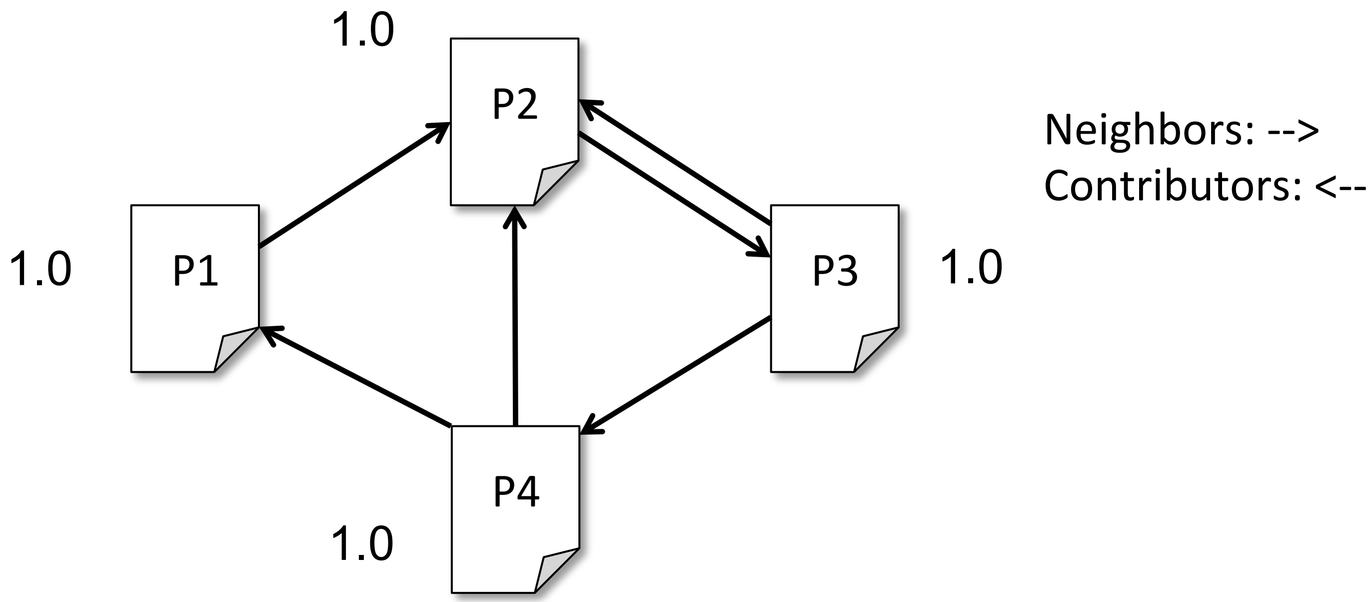
- **PageRank** (PR) is an algorithm used by Google Search to rank websites in their search engine results.
- Give pages ranks (scores) based on links to them
  - Links from many pages ➔ high rank
  - Link from a high-rank page ➔ high rank

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



Algorithm

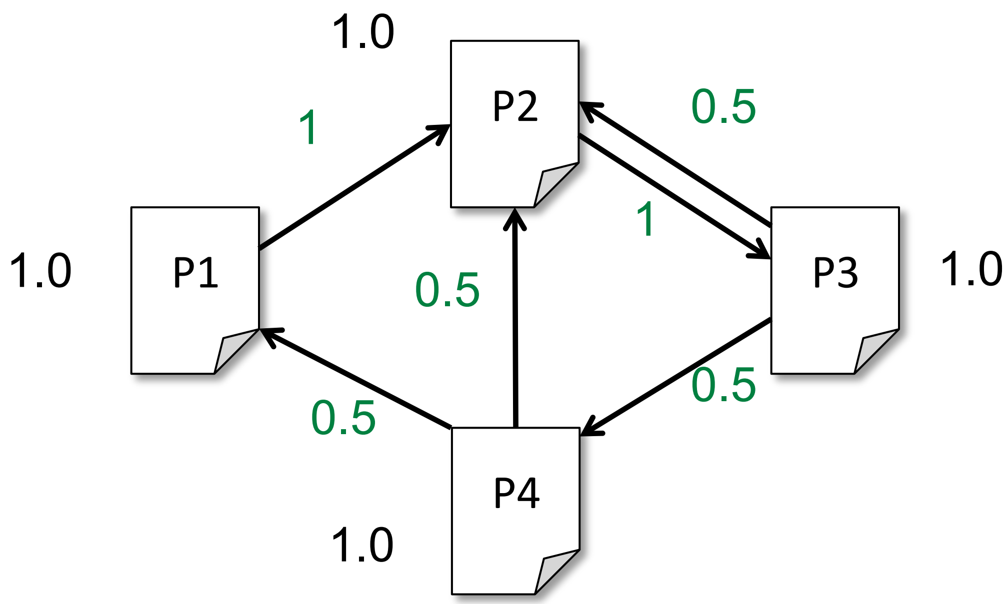
1. Start each page at a rank of 1
2. On each iteration, have page p send a contribution of rank(p)/numNeighbors(p) to its neighbors (the pages it has links to).
3. Set each page's rank to $0.15 + 0.85 \times \text{contribs}$



http://knowledge-shareing-xinyu.blogspot.co.uk/2016/03/an-anatomy-of-implementation-of.html

Algorithm

1. Start each page at a rank of 1
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 × contribs

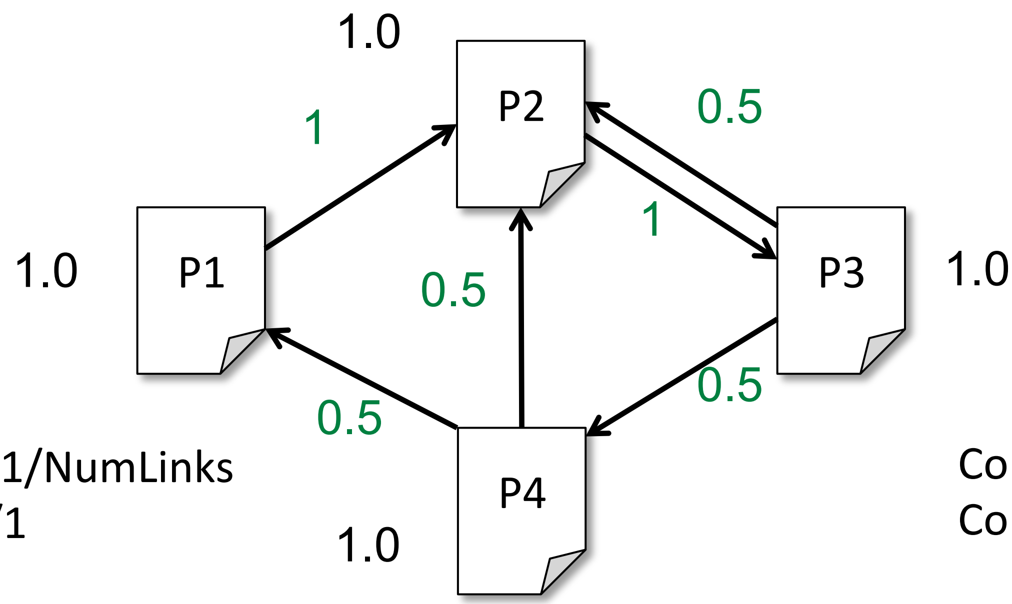


Neighbors: -->
Contributors: <--

http://knowledge-shareing-xinyu.blogspot.co.uk/2016/03/an-anatomy-of-implementation-of.html

Algorithm

1. Start each page at a rank of 1
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 × contribs



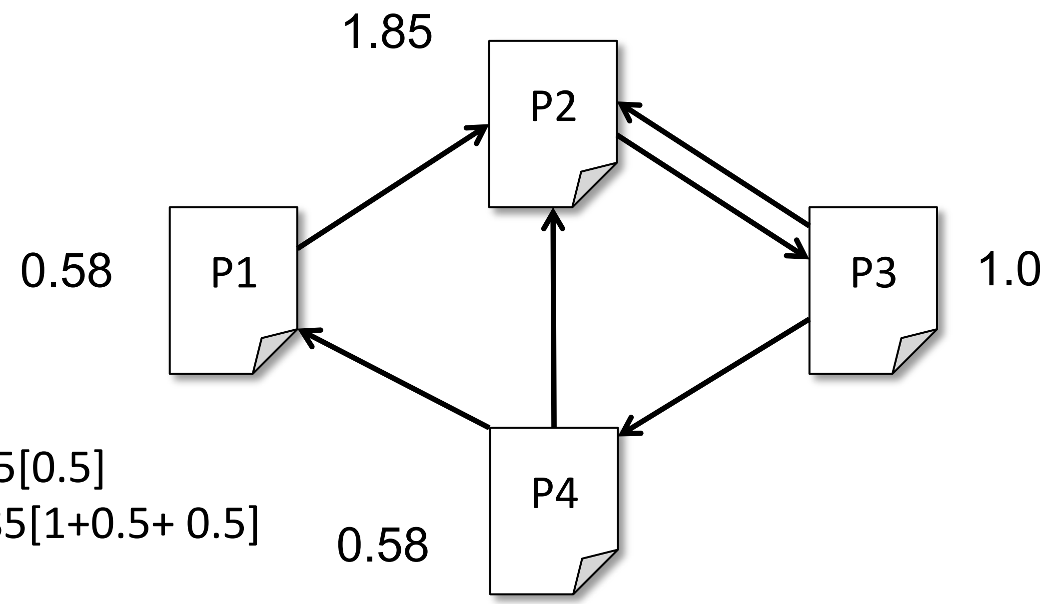
Contrib of P1: PR1/NumLinks
Contrib of P1= 1/1

Contrib of P4: PR4/NumLink
Contrib of P4= ½ = 0.5

http://knowledge-shareing-xinyu.blogspot.co.uk/2016/03/an-anatomy-of-implementation-of.html

Algorithm

1. Start each page at a rank of 1
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 × contribs

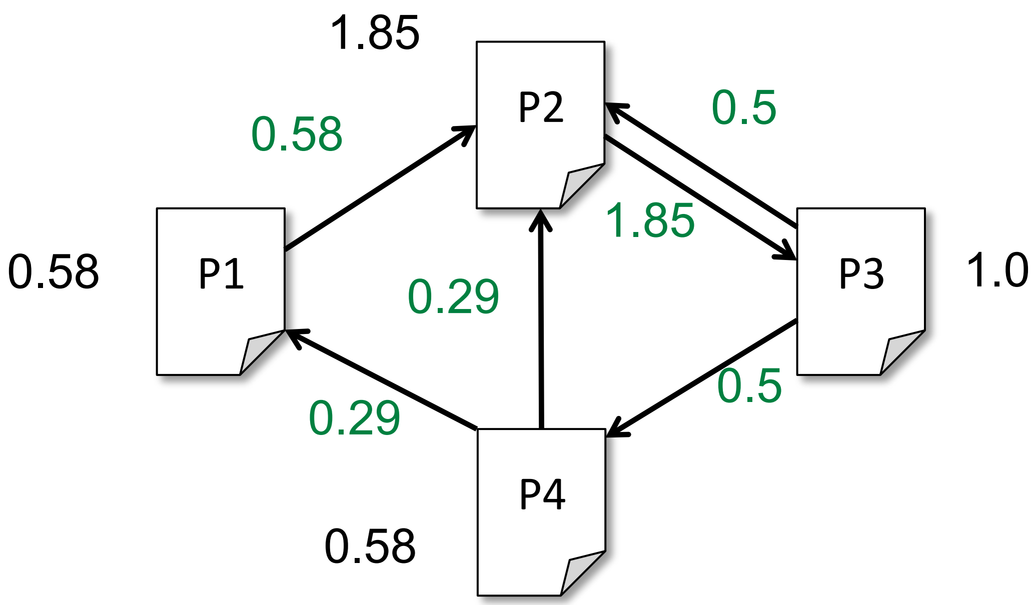


PR1 = 0.15 +0.85[0.5]
PR2 = 0.15 + 0.85[1+0.5+ 0.5]

http://knowledge-shareing-xinyu.blogspot.co.uk/2016/03/an-anatomy-of-implementation-of.html

Algorithm

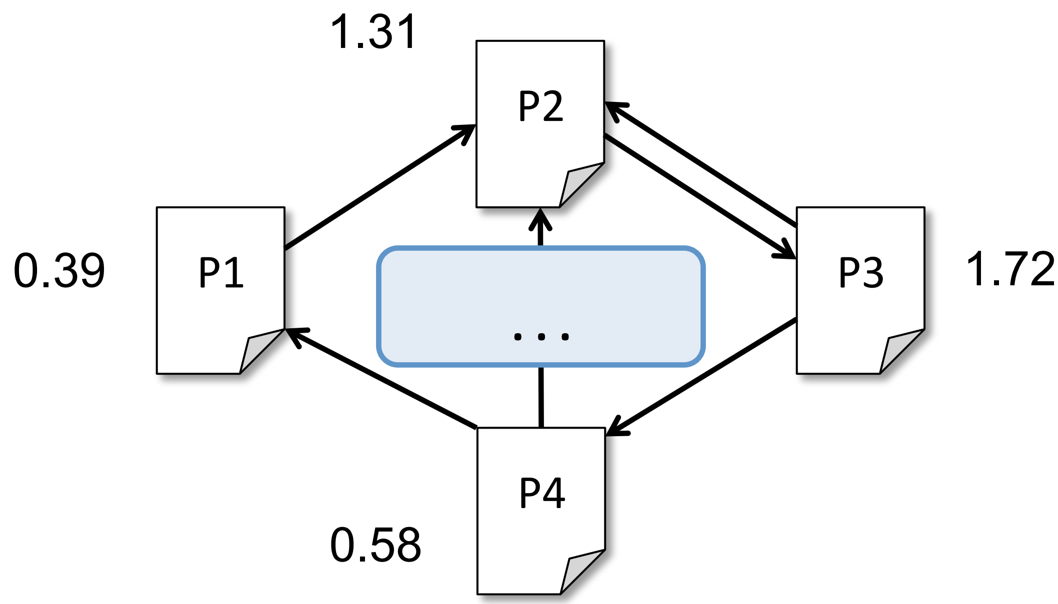
1. Start each page at a rank of 1
2. On each iteration, have page p send a contribution of rank(p)/numNeighbors(p) to its neighbors (the pages it has links to).
3. Set each page's rank to 0.15 + 0.85 × contribs



http://knowledge-shareing-xinyu.blogspot.co.uk/2016/03/an-anatomy-of-implementation-of.html

Algorithm

1. Start each page at a rank of 1
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 $0.15 + 0.85 \times \text{contribs}$

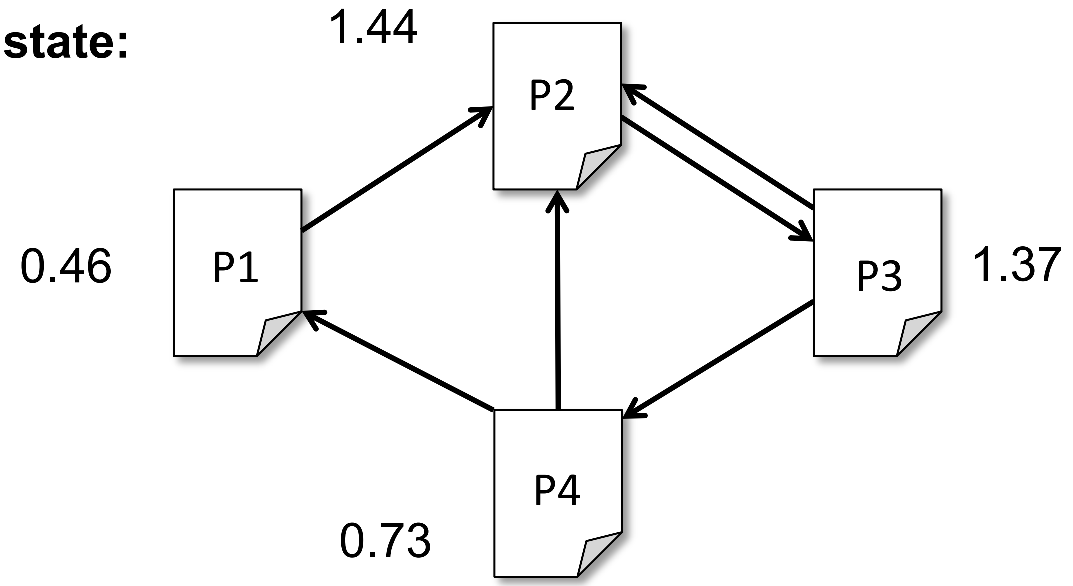


http://knowledge-shareing-xinyu.blogspot.co.uk/2016/03/an-anatomy-of-implementation-of.html

Algorithm

1. Start each page at a rank of 1
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 × contribs

Final state:



http://knowledge-shareing-xinyu.blogspot.co.uk/2016/03/an-anatomy-of-implementation-of.html

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.paralelize[(“p1_url”,”p2”), (“p2_url”,”p3”), (“p3_url”, (“p2,”p4”)),
(“p4_url”,(“p1”,”p2”))
ranks = sc.paralelize[(“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(...)
```

join

(“p1_url”, (“p2”,1)),
(“p2_url”, (“p3”, 1),
(“p3_url”, ((“p2”,”p4”), 1),
(“p4_url”, ((“p1”,”p2”), 1))

[url, [links, rank]]

[“p1_url”, [“p2”] ,1],
[“p2_url”, [“p3”], 1],
[“p3_url”, [“p2”,”p4”], 1],
[“p4_url”, [“p1”,”p2”], 1]

(dest, rank/len(links))

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

flatMap

[(p2, 1], (p3,1), (p2, 0.5),
(p4, 0,5), (p1 , 0.5), (p2, 0.5)]

reduceByKey

[(p2, (1 +0.5 +0.5), (p3,1), (p4, 0,5), (p1 , 0.5)

reduceByKey

[(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

Complete article at:
https://robertovitillo.com/2015/06/30/spark-best-practices/