Going Deeper into Spark Core

Xavier Morera
HELPING DEVELOPERS UNDERSTAND SEARCH & BIG DATA
@xmorera www.xaviermorera.com

Going Deeper into Spark Core

Anonymous Functions / Lambdas

Named Functions in Spark

```
def split_the_line(x):
    return x.split(',')
```

```
badges.map(add_one)
```

Why Are Lambdas so Useful?

Anonymous Functions in Spark

~~`badges.map(add_one)`~~

```
badges.map(lambda x: x.split(','))
```

You will find yourself using lambdas all the time with Spark

Believe me...

```
spark2-submit --packages com.databricks:spark-xml_2.11:0.4.1,
                         com.databricks:spark-csv_2.11:1.5.0
                         prepare_data_posts_simple_titles.py
```

Extract Titles from Posts.xml

Data preparation step

```
lines = sc.textFile('/user/cloudera/stackexchange/simple_titles_txt')

words = lines.flatMap(lambda line: line.split(' '))

word_for_count = words.map(lambda x: (x,1))

word_for_count.reduceByKey(lambda x,y: x + y).collect()
```

A Closer Look at Map, FlatMap, Filter, Sort, ...

map() is one of the most commonly used transformations

Followed by flatMap(), filter() and sort()

And later on aggregations

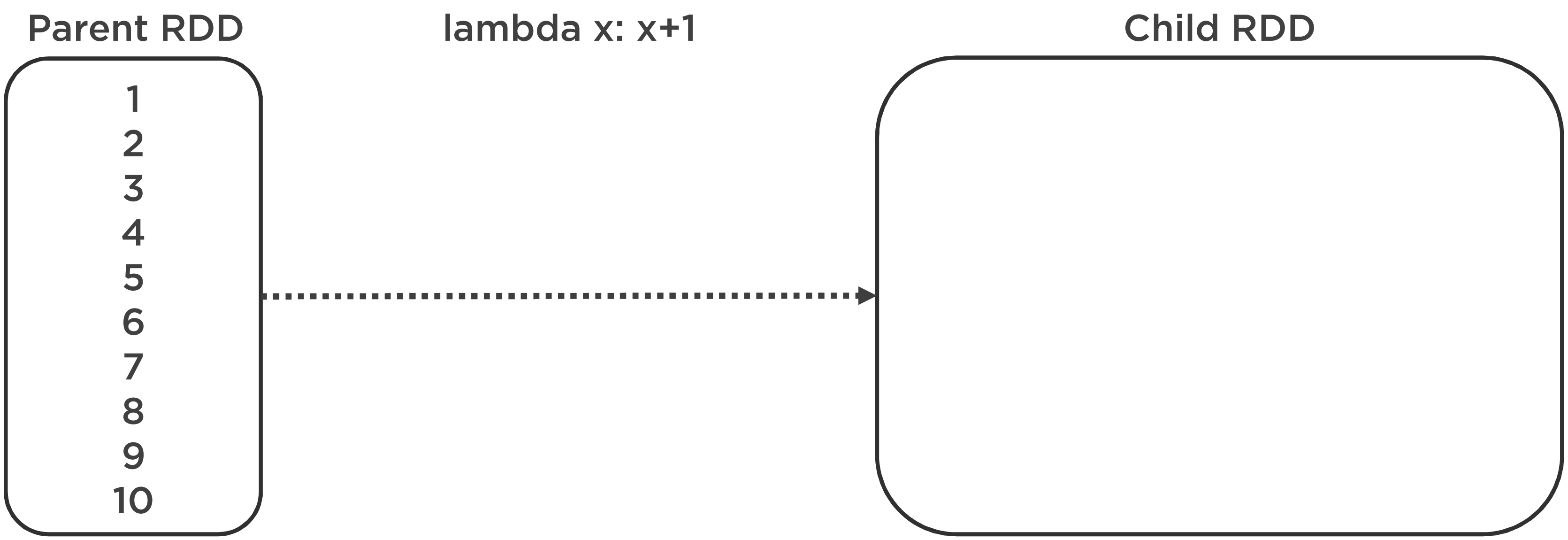
```
word_for_count = words.map(lambda x: (x,1))

word_for_count.take(1)

words.map(lambda x: x.lower())

words.map(lambda x: x.upper())
```

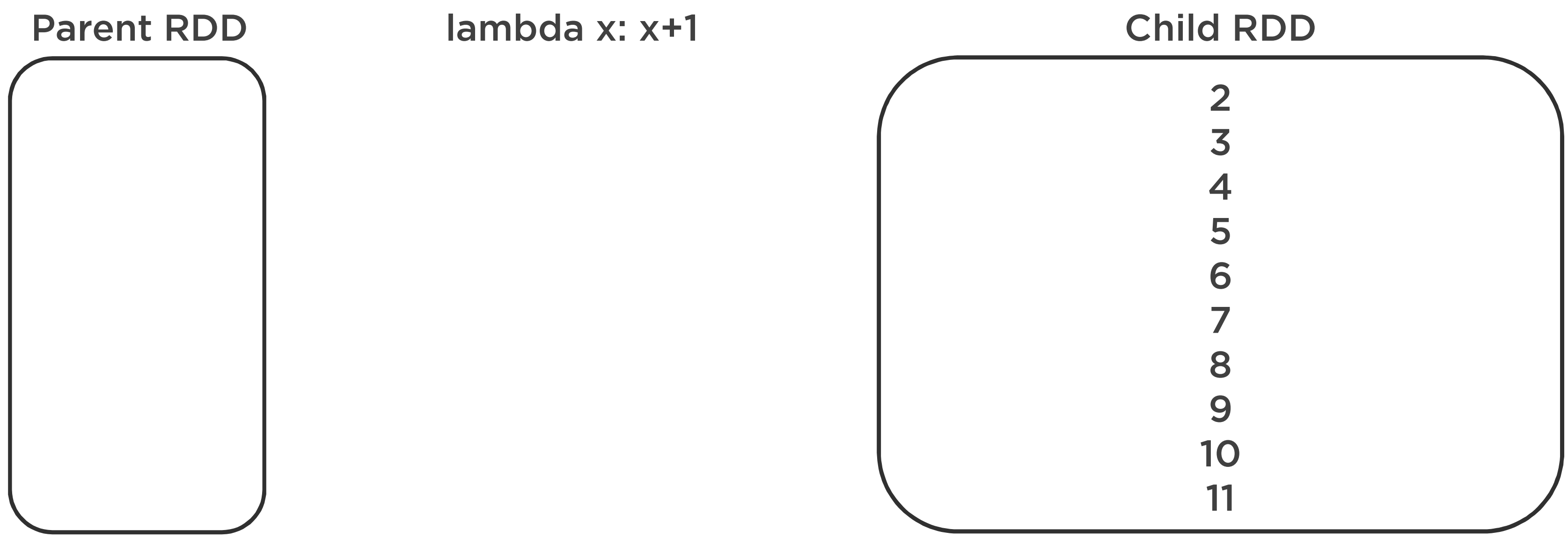
Map

Apply function to each element

map(), mapPartitions(), mapValues(), mapPartitionsWithIndex() ...

RDD of length N transformed to RDD of length N


Parent RDD
lambda x: x+1
Child RDD
1
2
3
4
5
6
7
8
9
10


Parent RDD
lambda x: x+1
Child RDD
2
3
4
5
6
7
8
9
10
11

```
word_for_count = words.map(lambda x: (x,1))

word_for_count.take(1)

words.map(lambda x: x.lower())

words.map(lambda x: x.upper())
```

Map

Apply function to each element

map(), mapPartitions(), mapValues(), mapPartitionsWithIndex() ...

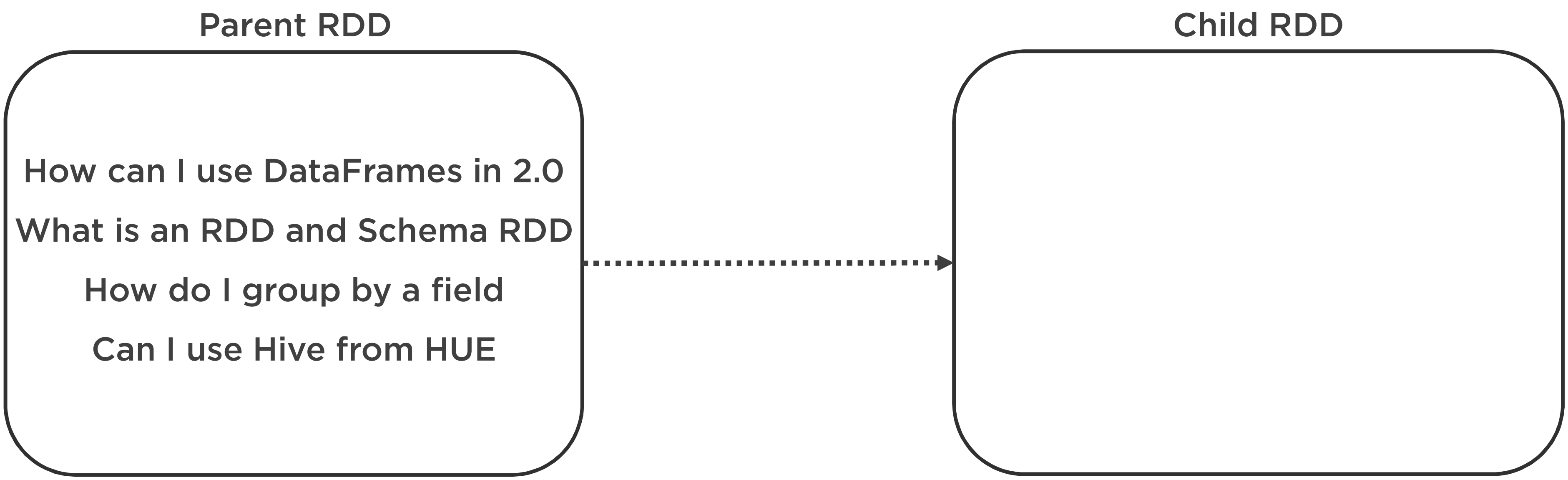
Each element in parent RDD mapped to one element in the child RDD

```
words = lines.flatMap(lambda line: line.split(' '))
```

FlatMap

Apply function to each element and returns list of elements

Returns 0, 1 or more elements, "flattens" the results with map


Parent RDD
How can I use DataFrames in 2.0
What is an RDD and Schema RDD
How do I group by a field
Can I use Hive from HUE
Child RDD

Parent RDD

Child RDD

How, can, I, use, DataFrames, in, 2.0, What, is, an, RDD, and, Schema, RDD, How, do, I, group, by, a, field, Can, I, use, Hive, from, HUE

```
def starts_h(word):
return word[0].lower().startswith('h')

word_for_count.filter(starts_h).collect()
```

Filter

Apply a function to each element of the RDD

If the function returns false, element is not included in new RDD

Filter

Filter

```
word_count = word_for_count.reduceByKey(lambda x,y: x + y)

word_count.take(10)

word_count.map(lambda
(x,y):(y,x)).sortByKey(False).map(lambda
(x,y):(y,x)).take(10)

word_count.sortBy(lambda (x,y): -y).take(10)
```

SortBy and SortByKey

Sort elements of an RDD

- By key on PairRDD with sortByKey()
- By a function using sortBy()

```
word_for_count.distinct().filter(starts_h).collect()
```

Many More Transformations

Plenty of transformations to go around

Some of them very powerful and/or very useful

Plenty of transformations to go around…

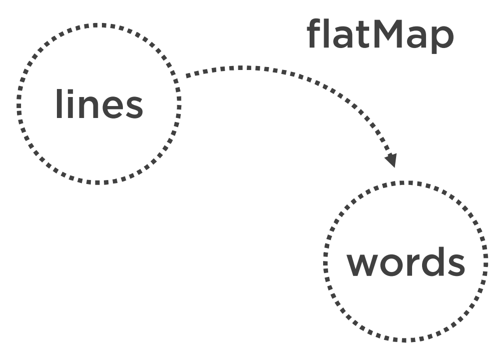
```
lines = sc.textFile('/user/cloudera/stackexchange/simple_titles_txt')
words = lines.flatMap(lambda line: line.split(' ‘))
word_for_count = words.map(lambda x: (x,1))
grouped_words = word_for_count.reduceByKey(lambda x,y: x + y)
grouped_words.collect()
```

Transformations

Start with method from SparkContext to load data

Transformations perform a computation

And create new RDDs



map(*f*, *preservesPartitioning=False*)

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

```
>>> rdd = sc.parallelize(["b", "a", "c"])
>>> sorted(rdd.map(lambda x: (x, 1)).collect())
[('a', 1), ('b', 1), ('c', 1)]
```

mapPartitions(*f*, *preservesPartitioning=False*)

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

```
>>> rdd = sc.parallelize([1, 2, 3, 4], 2)
>>> def f(iterator): yield sum(iterator)
>>> rdd.mapPartitions(f).collect()
[3, 7]
```

mapPartitionsWithIndex(*f*, *preservesPartitioning=False*)

Return a new RDD by applying a function to each partition of this RDD, while tracking the index of the original partition.

```
>>> rdd = sc.parallelize([1, 2, 3, 4], 4)
>>> def f(splitIndex, iterator): yield splitIndex
```

Transformations

groupBy cartesian
flatMap sample repartition intersection
map filter union subtract
keyBy sortBy
coalesce zipWithIndex
zip mapPartitions
distinct

Transformations

PairRDDs

combineByKey sampleByKey

reduceByKey

join

leftOuterJoin

fullOuterJoin

sortByKey

cogroup

subtractByKey

groupByKey

rightOuterJoin

aggregateByKey

flatMapValues

foldByKey

reduceByKeyLocally

partitionBy

```
lines = sc.textFile('/user/cloudera/stackexchange/simple_titles_txt')
words = lines.flatMap(lambda line: line.split(' ‘))
word_for_count = words.map(lambda x: (x,1))
grouped_words = word_for_count.reduceByKey(lambda x,y: x + y)
grouped_words.collect()
```

Previously on Transformations

Transformations are what "changes" your data

Remember: Spark is lazy

No computation done when you specify transformation

```
lines = sc.textFile('/user/cloudera/stackexchange/simple_titles_txt')
words = lines.flatMap(lambda line: line.split(' ‘))
word_for_count = words.map(lambda x: (x,1))
grouped_words = word_for_count.reduceByKey(lambda x,y: x + y)
grouped_words.collect()
```

Actions

Action triggers computation

i.e. can return data to the driver or save an RDD to storage

Operations that produce non RDD values

pyspark package — PySp ×

Secure | https://spark.apache.org/docs/2.2.0/api/python/pyspark.html

collect()

Return a list that contains all of the elements in this RDD.

> **Note:** This method should only be used if the resulting array is expected to be small, as all the data is loaded into the driver's memory.

collectAsMap()

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

> **Note:** this method should only be used if the resulting data is expected to be small, as all the data is loaded into the driver's memory.

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

Actions

histogram
forEach
take
saveAsHadoopDataset
saveAsSequenceFile
collectAsMap
saveAsObjectFile
saveAsNewAPIHadoopDataset
treeReduce
collect
saveAsNewAPIHadoopFile
forEachPartition
treeAggregate
count
takeSample
saveAsHadoopFile
countApproxDistinct
max
variance
top
reduce
mean
sampleVariance
saveAsTextFile
min
stdev
countApprox
sum
takeOrdered
aggregate
fold
first

Actions

PairRDD

countApproxDistinctByKey

keys

values

countByKey

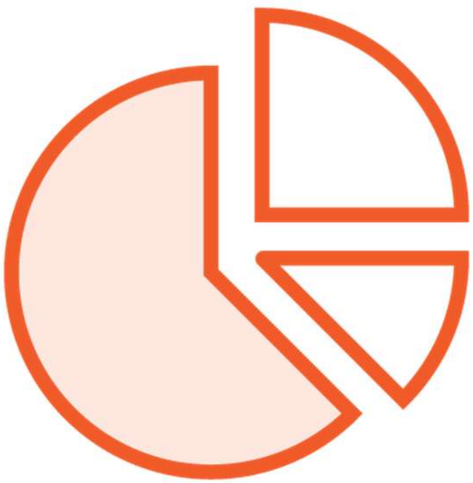
countByValueApprox

countByKeyApprox

sampleByKeyExact

countByValue

A Thing or Two on Partitions



- Partition is just a 'bunch' of data
- One of the foundations of parallelism
- Faster to operate within partition
  - Than shuffling data
- Group data to minimize network traffic

How Does Spark Partition Data?

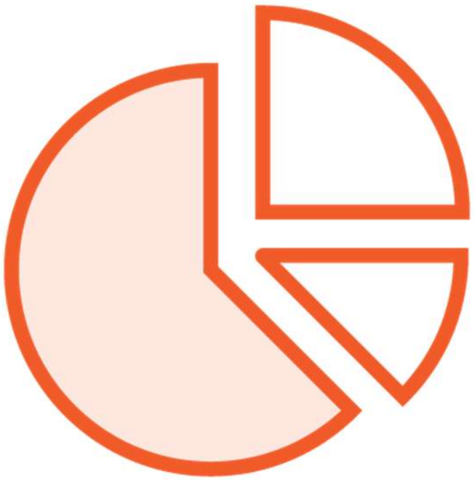
Data locality

- Partition per HDFS block

Resources

Configuration or parameters

How Does Spark Partition Data?

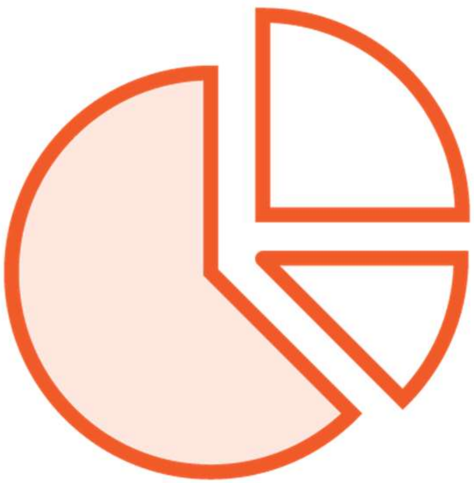


Partitioner

- Hash partitioner
- Range partitioner

Repartition

More or Less Partitions?



More partitions

- Less data per partition
- Smaller jobs
- More parallelism

Less partitions

- More data per partition
- Larger jobs

```
badges_for_part = badges_columns_rdd.map(lambda x:
(x[2],x)).repartition(50)

print badges_for_part.partitioner
def badge_partitioner(badge):
    return hash(badge)

badges_by_badge = badges_for_part.partitionBy(50,
badge_partitioner)
```

PartitionBy

Returns an RDD partitioned using a specific partitioner

Useful to get keyed data into same partition

Not yet a group operation

```
print badges_by_badge.partitioner

badges_for_part.saveAsTextFile('/user/cloudera/
stackexchange/badges_nei_partitioner')

badges_by_badge.saveAsTextFile('/user/cloudera/
stackexchange/badges_yei_partitioner')
```

PartitionBy

Create a function to be used for partitioning

Pass function as parameter to partitionBy()

Save with and without partitioner, and review results

```
for p in badges_by_badge.map(lambda (x,y):x)
.glom().collect():

print p
```

Glom

There is an action to coalesce all rows in a partition into an array

Useful for operations on all items within a partition

Let's print our keys per partition

Count Badges

```
def count_badges(iterator):
    total = 0
    for ite in iterator:
        total += 1
        yield total
```

```
counted_badges =
badges_by_badge.mapPartitions(count_badges)

counted_badges.collect()
```

MapPartitions

Apply a function to each partition

Done at a single pass

Returns after entire partition is processed

```
def next_value(value_list):
    for I in value_list:
    yield I
test_yield = next_value([1, 2, 3])
test_yield.next()
```

Yield

Returns a generator

Iterate with next()

Until StopIteration

```
posts_all.count()
```

Sampling Data

Selecting a representative part of the population

Faster, but you may lose accuracy

Also useful if you are resource constrained or very large dataset

```
posts_all.count()

sample_posts=posts_all.sample(False,0.1,50)

sample_posts.count()

posts_all.countApprox(100, 0.95)
```

Sampling Data

Transformation to obtain a sample from your data with sample()

- withReplacement
- fraction
- seed

```
posts_all.count()

sample_posts=posts_all.sample(False,0.1,50)

sample_posts.count()

posts_all.countApprox(100, 0.95)
```

Approximate Counts

Obtain an approximate count with countApprox()

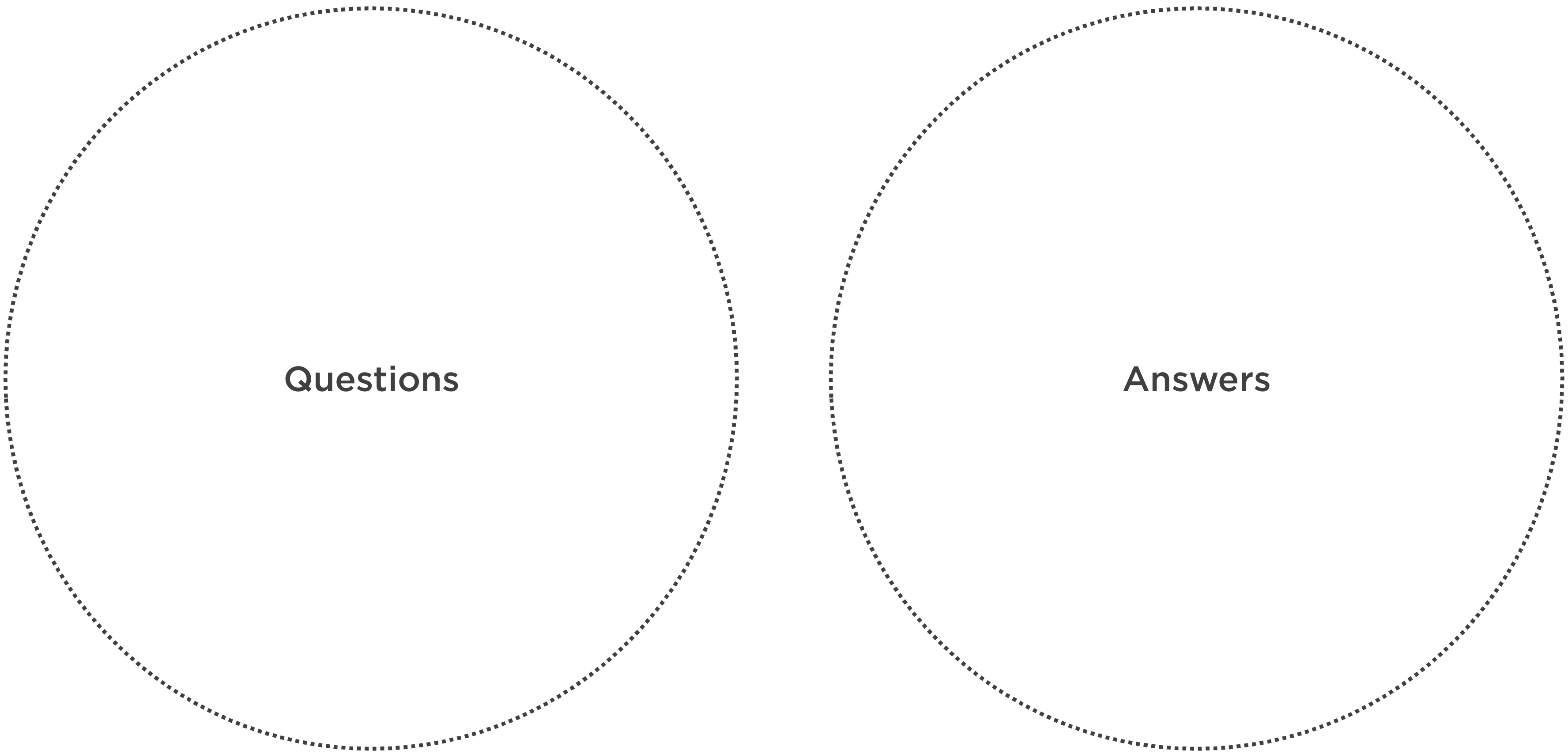
Note:	Experimental

```
posts_all.takeSample(False,10,50)

len(posts_all.takeSample(False,10,50))
```

Take a Sample of Exact Size

Action available for exact count is called takeSample()

Set Operations



Set Operations

Posts Questions

('xavier', 1)

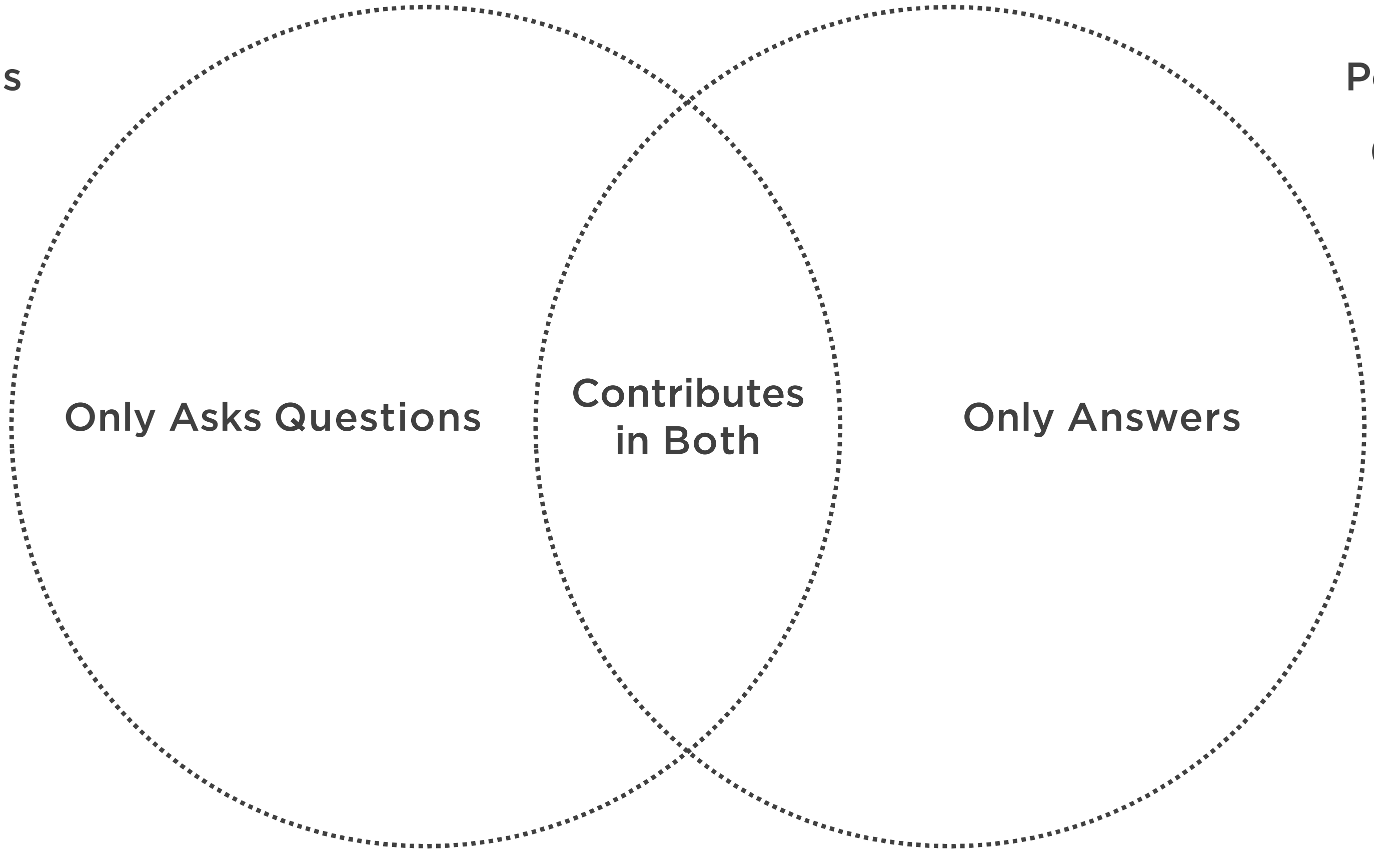
('troy', 2)

('xavier', 5)

Posts Answers

('xavier', 3)

('beth', 4)

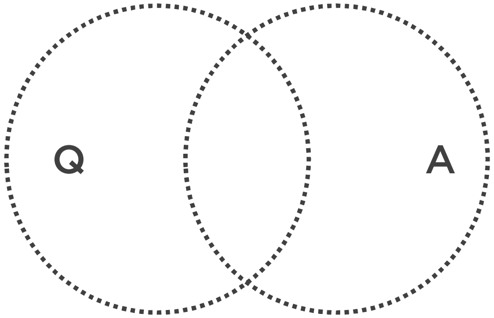


```
questions=sc.parallelize([("xavier",1),("troy",2),
("xavier",5)])

answers=sc.parallelize([("xavier",3),("beth",4)])
```

Our Data

Create with parallelize

If you feel confident, go for the full dataset



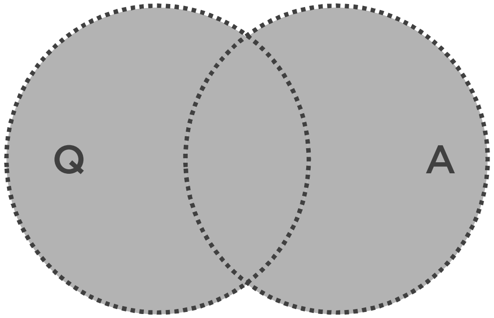
```
questions.union(answers).collect()

questions.union(sc.parallelize(['irene', 'juli',
'luci'])).collect()

questions.union(sc.parallelize(range(10))).collect()
```

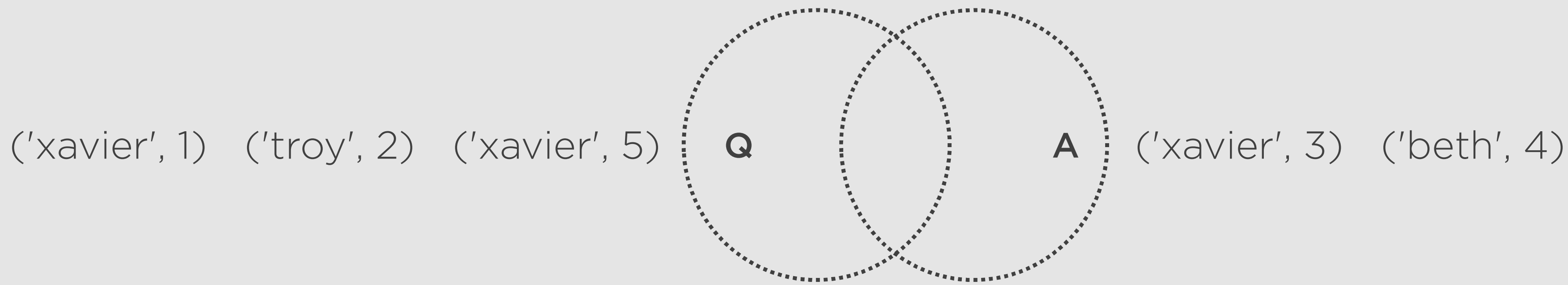
Union

RDD with all elements in both RDDs

Questions + answers

Can be different types





Union

All questions and answers

Elements remain the same

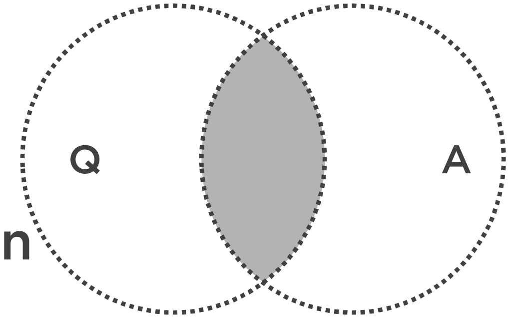
```
questions.join(answers).collect()

questions.join(sc.parallelize(range(10))).collect()
```

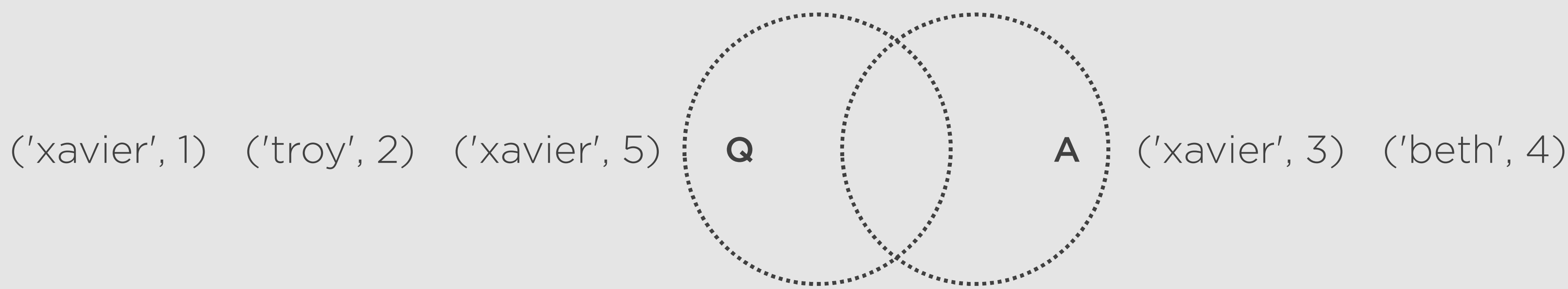
Join

Elements with same keys in both, joined values

Hash join over the cluster, thus expensive

Unless known partitioner for narrow transformation





Join

People who have asked questions AND answered questions

Key is the person, value shows posts

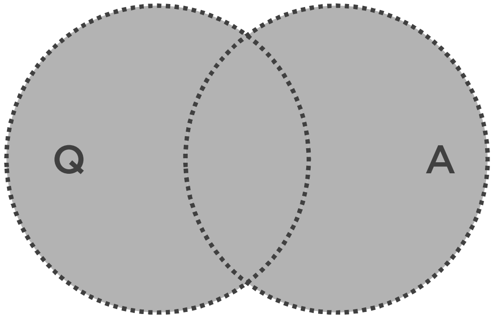
Excludes those that do not contribute

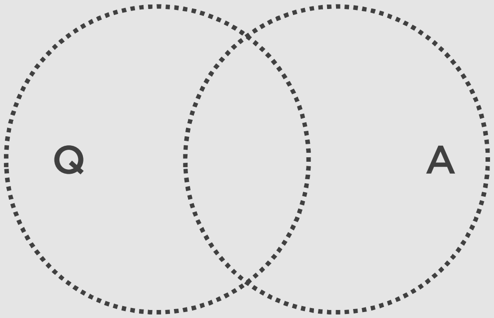
```
questions.fullOuterJoin(answers).collect()
```

fullOuterJoin

Like join(), but....

None where key does not appear in one RDD





('xavier', (1, 3)) ('xavier', (5, 3)) ('troy', (2, None)) ('beth', (None, 4))

fullOuterJoin

All questions and answers, joined by key

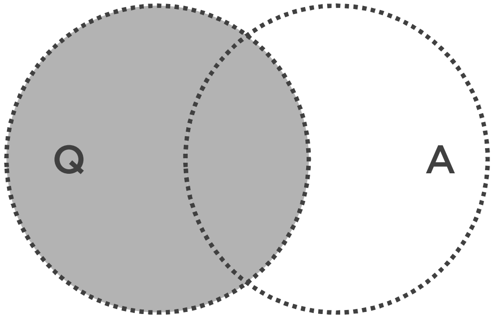
- None when user does not appear in one set

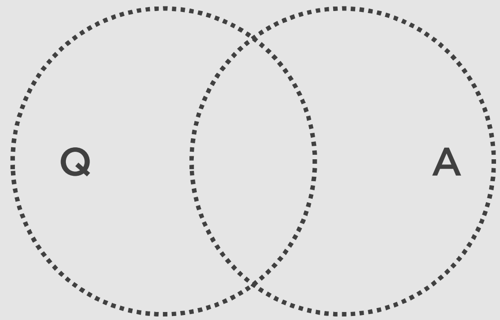
```
questions.leftOuterJoin(answers).collect()
```

leftOuterJoin

Join using keys from left set

None when key not found on right set





('beth', 4)

('xavier', (1, 3)) ('xavier', (5, 3)) ('troy', (2, None))

leftOuterJoin

Join using keys from left set

None when key not found on right set

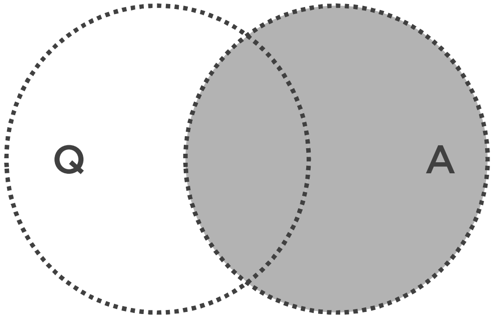
```
questions.rightOuterJoin(answers).collect()
```

rightOuterJoin

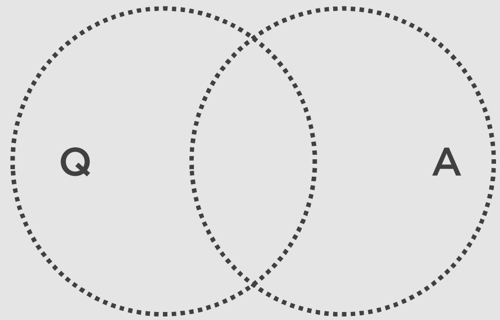
Opposite of a leftOuterJoin

Join using keys from the right set

None where keys not available in left set



('troy', 2)



('xavier', (1, 3)) ('xavier', (5, 3)) ('beth', (None, 4))

rightOuterJoin

Opposite of a leftOuterJoin

Join using keys from the right set

None where keys not available in left set

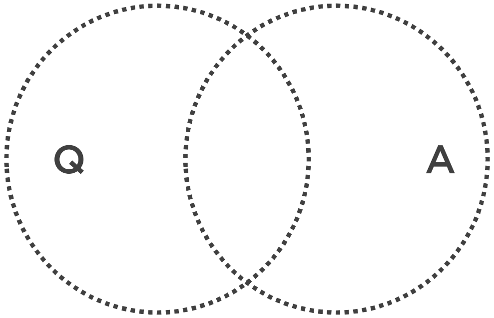
```
questions.leftOuterJoin(answers)

answers.rightOuterJoin(questions)
```

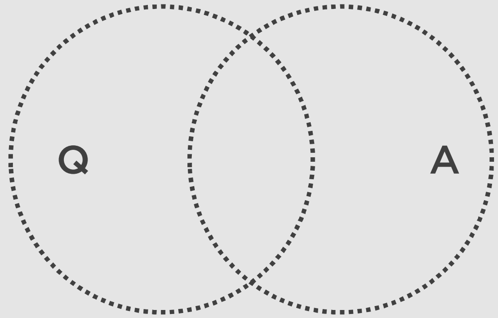
leftOuterJoin and rightOuterJoin

questions.leftOuterJoin(answers)

Equivalent to

answers.rightOuterJoin(questions)





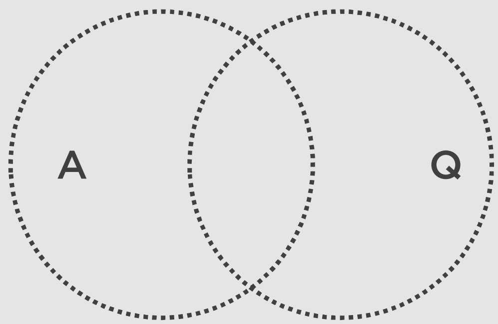
('beth', 4)

leftOuterJoin ('xavier', (1, 3)) ('xavier', (5, 3)) ('troy', (2, None))

---

rightOuterJoin

('beth', 4)



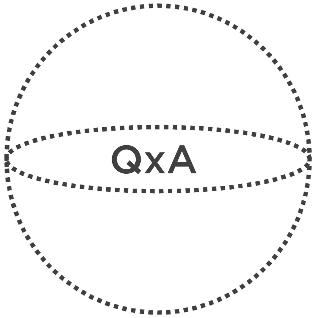
('xavier', (1, 3)) ('xavier', (5, 3)) ('troy', (2, None))

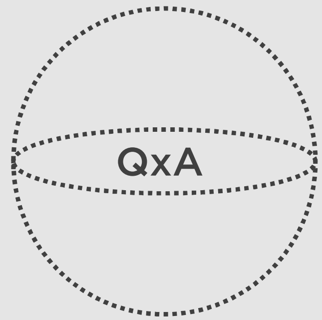
```
questions.cartesian(answers).collect()
```

Cartesian

Join of all elements in left set

With all elements in the right set





(('xavier', 1) ('xavier', 3)) (('xavier', 1) ('beth', 4)) (('troy', 2) ('xavier', 3))
(('xavier', 5) ('xavier', 3)) (('troy', 2) ('beth', 4)) (('xavier', 5) ('beth', 4))

Cartesian

Join of all elements in left set

With all elements in the right set

Aggregation

Grouping elements together

Foundations of Big Data analytics

```
each_post_owner=posts_all.map(lambdax: x.split(",")[6])
posts_owner_pair_rdd=each_post_owner.map(lambdax: (x,1))
top_user_posts.map(lambda(x,y): (x,len(y))).take(1)
```

Prepare Some Data

Extract user from each post

PairRDD

- Key is user
- Value is 1

```
top_posters_gbk=posts_owner_pair_rdd.groupByKey()
```

GroupByKey

Values grouped by each key

Data sent over the network and collected on reduce workers

Can cause problems on larger datasets

```
top_user_posts = top_posters_gbk.filter(lambda (x,y):
x == "51")
```

GroupByKey

Tuple of user id and list of 1's

Posts per user? → User id and number of posts

Use sortBy for top poster

```
from operator import add)
```

ReduceByKey

Perform an operation on all elements with same key

Specify a function

Reduce operation done within partition

```
top_posters_rbk=posts_owner_pair_rdd.reduceByKey(add)

top_posters_rbk.lookup('51')

top_posters_rbk.map(lambda(x,y):
(y,x)).sortByKey(False).map(lambda(x,y): (y,x)).take(10
```

ReduceByKey

Use add

Pass to reduceByKey()

Use lookup() to find top poster and confirm

```
top_posters_gbk.count()

top_posters_rbk.count()
```

groupByKey vs. reduceByKey

Do we get the same results?

Indeed we do

aggregateByKey

```
questions_asked=posts_all_entries.filter(lambda x:x[1]=="1")
user_question_score=questions_asked.map(lambda x: (x[6],int(x[4])))
for_keeping_count=(0,0)
```

```
aggregated_user_question=user_question_score.aggregateByKey(

for_keeping_count,lambdatuple_sum_count,next_score:
(tuple_sum_count[0]+next_score,

tuple_sum_count[1]+1),lambdatuple_sum_count,tuple_next_partition_sum_c
ount:(tuple_sum_count[0]+tuple_next_partition_sum_count[0],

tuple_sum_count[1]+tuple_next_partition_sum_count[1]))
```

aggregateByKey

Like reduceByKey()

But takes an initial value

Specify functions for merging and combining

aggregateByKey

Combining

- Within partition

Merging

- Across partitions

```
aggregated_user_question.lookup('51')
```

aggregateByKey

Only questions, include score and user id

Define initial value, merging function, and combining function

Check with top poster

```python
user_post = questions_asked.map(lambda x: (x[6],int(x[0])))

def to_list(postid):
    return[postid]

def merge_posts(posta,postb):
    posta.append(postb)
    return posta

def combine_posts(posta, postb):
    posta.extend(postb)
    return posta
```

```
combined=user_post.combineByKey(to_list, merge_posts,
combine_posts)

combined.filter(lambda(x,y): x=='51').collect()

combined.lookup('51')
```

CombineByKey

Specify an initial value can be a function that returns a new value

Provide merge and combine functions

Like aggregateByKey(), but more flexible

```
user_post.lookup('51')

user_post.countByKey()['51']
```

CountByKey

Dictionary with keys and counts of occurrences

Like a reduceByKey() where we count based on key

reduceByKey & groupByKey

```
add_them = lambda x,y: x + y
add_in_list = lambda x: sum(list(x))
reduced = word_for_count.reduceByKey(add_them)
grouped =
word_for_count.groupByKey().mapValues(add_in_list)
```

```
reduced.take(1)

grouped.take(1)

grouped.count()

reduced.count()
```

reduceByKey & groupByKey

Both can be used for the same purpose

Aggregate by keys

Work very differently underneath

Comparing groupByKey vs. reduceByKey

groupByKey

(Spark,1)	(Spark,1)	(Cloudera,1)
(Spark,1)	(Spark,1)	(Cloudera,1)
(Spark,1)	(Spark,1)	
(Spark,1)	(Spark,1)	
(HUE,1)	(Cloudera,1)	

reduceByKey

(Spark,1)	(Spark,1)	(Cloudera,1)
(Spark,1)	(Spark,1)	(Cloudera,1)
(Spark,1)	(Spark,1)	
(Spark,1)	(Spark,1)	
(HUE,1)	(Cloudera,1)	

groupByKey

(Spark,1)
(Spark,1)
(Spark,1)
(Spark,1)
(HUE,1)

(Spark,1)
(Spark,1)
(Spark,1)
(Spark,1)
(Cloudera,1)

(Cloudera,1)
(Cloudera,1)

(Spark,1)
(Spark,1)
(Spark,1)
(Spark,1)
(Spark,1)
(Spark,1)
(Spark,1)
(Spark,1)

(HUE,1)

(Cloudera,1)
(Cloudera,1)
(Cloudera,1)

(Spark,8) **(HUE,1)** **(Cloudera,3)**

reduceByKey

(Spark,4)
(HUE,1)

(Spark,4)
(Cloudera,1)

(Cloudera,2)

(Spark,8) **(HUE,1)** **(Cloudera,3)**

12 elements vs. 5 elements

Histogram

A diagram consisting of rectangles whose area is proportional to the frequency of a variable and whose width is equal to the class interval.

```
badges_reduced.map(lambda(x,y): y).histogram(7)
```

Grouping Data into Buckets with Histogram

Histograms are very powerful graphic tools

An image is worth a thousand words

Getting the data is usually the hardest part

```
badges_reduced.map(lambda(x,y): y)
.histogram([0,1000,2000,3000,4000,5000,6000,7000])

badges_reduced.sortBy(lambdax:-x[1]).take(10)

badges_reduced.filter(lambdax: x[1]<1000).count()
```

Grouping Data into Buckets with Histogram

Specify number of intervals

- Returns array with intervals and array of counts within intervals

Explicitly state which intervals to use

Cache

Store data for future use, to improve response times

Persist to disk, memory or both

```
reduced.setName('Reduced RDD')

reduced.cache()
```

Cache & Persist

Spark may perform caching of intermediate results

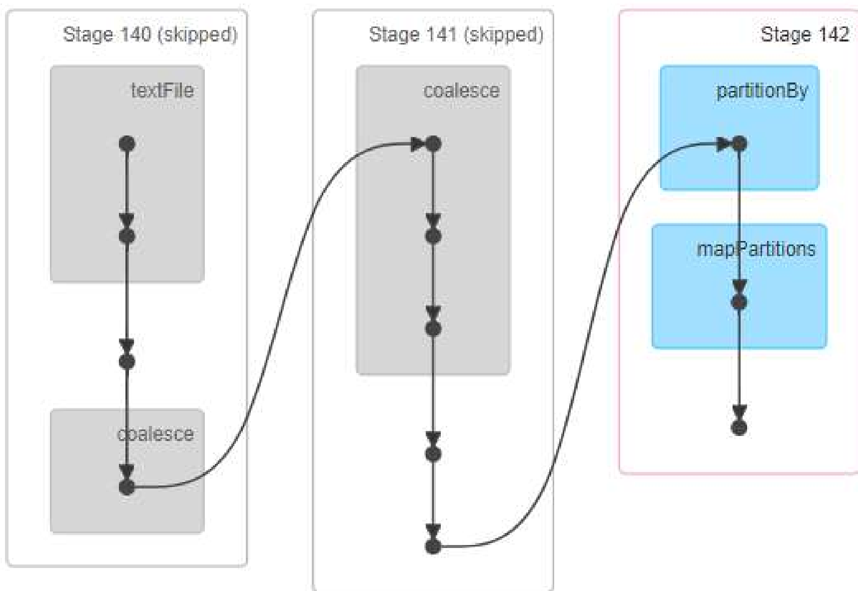
- On expensive operations, to avoid recomputing when nodes fail

Spark 2.2.0.cloudera1 | Jobs | Stages | Storage | Environment | Executors | SQL | PySparkShell application UI

Details for Job 95

Status: SUCCEEDED
Completed Stages: 1
Skipped Stages: 2

▸ Event Timeline
▾ DAG Visualization



Completed Stages (1)

Stage Id ▾	Description	Submitted	Duration	Tasks: Succeeded/Total	Input	Output	Shuffle Read	Shuffle Write
142	runJob at PythonRDD.scala:446 +details	2018/01/12 13:00:58	75 ms	1/1			177.8 KB	

```
badges_sorted.persist()
```

Cache & Persist

Spark may perform caching of intermediate results

- On expensive operations, to avoid recomputing when nodes fail

If the same job called twice, entire operation may be recomputed

```
grouped.setName('Grouped RDD')
grouped.persist(pyspark.storagelevel.StorageLevel.DISK_ONLY)
```

Cache & Persist

Call explicitly cache() and persist() when beneficial

- cache() is equivalent to persist(MEMORY_ONLY)

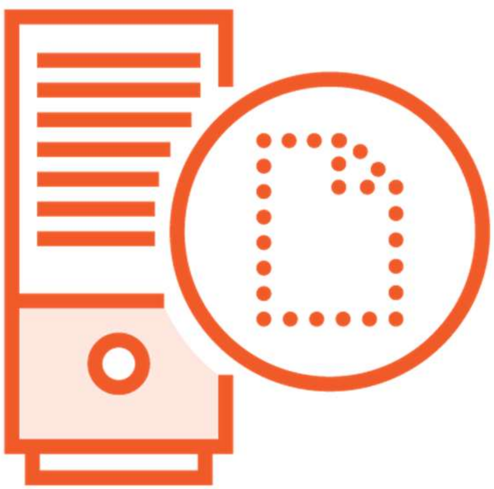
When RDD not needed anymore, call unpersist()

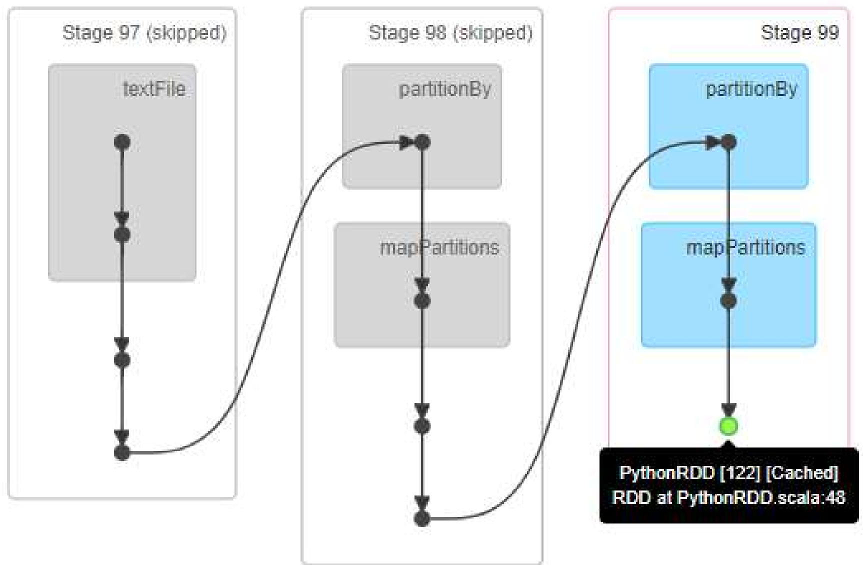
Storage Levels

MEMORY_ONLY
MEMORY_AND_DISK
DISK_ONLY
MEMORY_ONLY_2,
MEMORY_AND_DISK_2, etc.
OFF_HEAP (experimental)

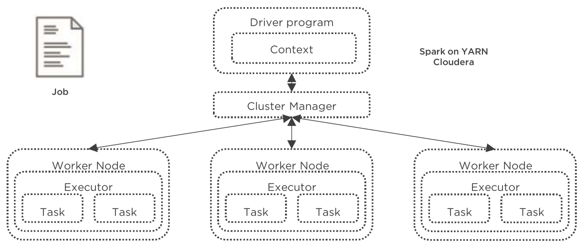
MEMORY_ONLY_SER
MEMORY_AND_DISK_SER

Cache & Persist





Spark Processing



Distributed and parallel processing

Each executor has separate copies

- Variables and functions

No propagation data back to driver

- Except on certain necessary cases
- Accumulators & Broadcast Variables

Shared Variables

Accumulators

"Added"

Associate and commutative

Numeric accumulator

Other types possible

Counter is one common scenario

Accumulator may not be reliable

Case of failed task

Potential duplicate counts

Broadcast Variables

Read only variable

Immutable

Fits in memory

Distributed efficiently to the cluster

Do not modify after shipped

Good case is a lookup table

Accumulator

```
accumulator_badge=sc.accumulator(0)
accumulator_badge

def add_badge(item):
    accumulator_badge.add(1)
badges_by_badge.foreach(add_badge)
```

```
accumulator_badge.value
```

Accumulator

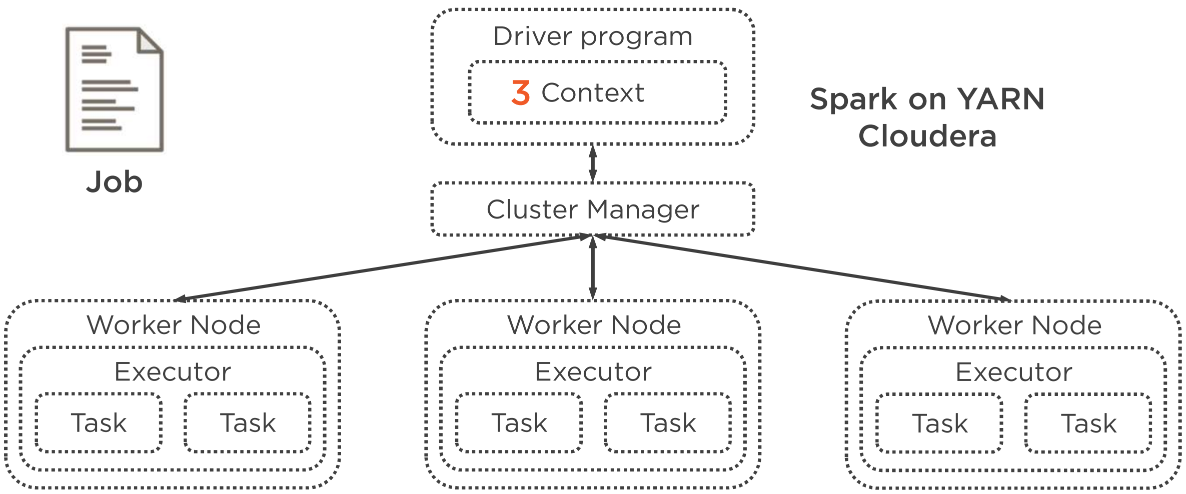
Create accumulator and check current value

Increment accumulator function and run

Get value

Accumulators

Executors write to accumulator in Driver program



```
def get_name(user_column):
    user_id = user_column[0]
    user_name = user_column[3]
    user_post_count = '0'
    if user_id in broadcast_tp.value:
        user_post_count = broadcast_tp.value[user_id]
        return (user_id, user_name, user_post_count)
```

Broadcast Variable

Create a broadcast variable using the context

Access when necessary, i.e. lookup

Use value

```
tp = top_posters_rbk.collectAsMap()
broadcast_tp = sc.broadcast(tp)
user_info = users_columns.map(get_name)
user_info.take(1)
```

Broadcast Variable

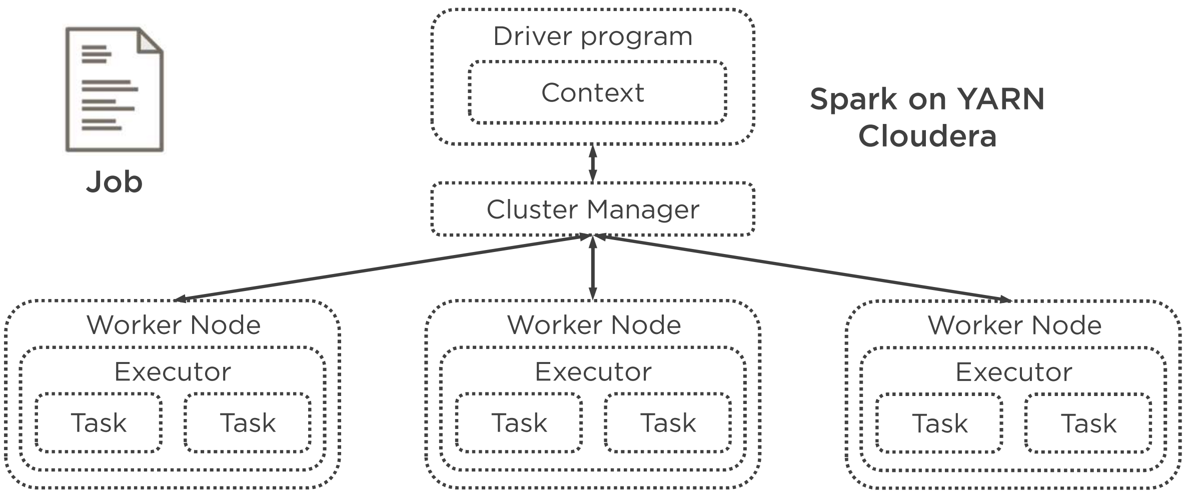
Create using sc.broadcast()

- Assign to a variable

Access using variable.value

Broadcast Variables

Executors read from Broadcast variable



Developing Self Contained PySpark Apps

Requires

- Create the SparkContext
- Dependencies
- Execute using spark2-submit

```
from pyspark import SparkContext

sc = SparkContext("yarn", "Standalone App")
```

Creating the SparkContext

Corresponding import

Create sc

```
spark2-submit --py-files dependency.egg --jars ...
```

Dependencies

Use py-files for distributing files to cluster, i.e. zip file

Use also jars parameter

- Supports file, hdfs, http, ftp or local, but no directory expansion

```
spark2-submit <params-dependencies-conf> prepare_posts.py
```

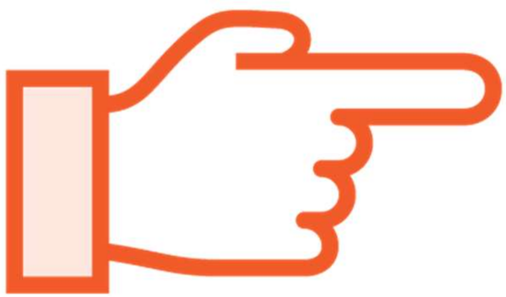
Executing Application

Using spark2-submit

Pass any necessary configuration, dependencies and parameters

Code to be executed, submitted as a job

Disadvantages of RDDs



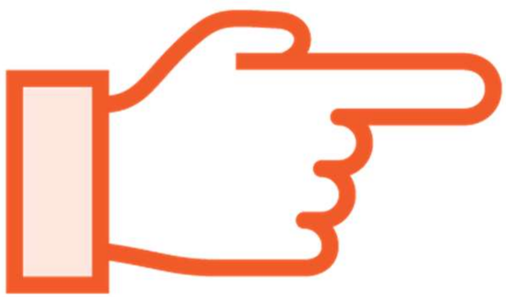
Don't take this the wrong way

RDDs are still used, even internally

Extremely powerful

Limitations o n potential optimizations

Disadvantages of RDDs



Performance

Schema less

Steeper learning curve

"Everybody knows SQL"

APACHE Spark™ SQL

Takeaway

Anonymous Functions

- Lambdas

Transformations vs. Actions

- Transformations return RDDs
- Actions trigger computation

Takeaway

Map, FlatMap, Filter, Sort, ...

Partitions

Sampling

Set operations

Aggregations

Takeaway

Histogram

Caching & Persisting

Shared variables

Self contained applications

Takeaway

Disadvantages of RDDs