# Going Deeper into Spark Core

Xavier Morera

HELPING DEVELOPERS UNDERSTAND SEARCH & BIG DATA

@xmorera www.xaviermorera.com

## Going Deeper into Spark Core

APACHE Spark™

## Why Lambdas?

=>

**Convenient and expressive**

- Don't disregard named functions either

**Quick to define, use and reuse**

**Testing and refactoring is easier**

**A.k.a Anonymous Functions**

## Anonymous Functions / Lambdas

Named Functions in Spark

```
def split_the_line(x: String):
Array[String] =
  x.split(",")


badges.map(split_the_line)
```

## Why Are Lambdas so Useful?

Anonymous Functions in Spark

```
~~badges.map(split_the_line)~~
```

```
badges.map(x => x.split(","))
```

## You will find yourself using lambdas all the time with Spark

**Believe me…**

```
cd posts_simple_titles

sbt package

spark2-submit --class "PreparePostsSimpleTitlesApp"
  target/scala-2.11/posts-simple-titles-project_2.11-1.0.jar
```

Extract Titles from Posts.xml

## Data preparation step

```
val words_in_line = lines.map(x => x.split(" "))

words_in_line.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**

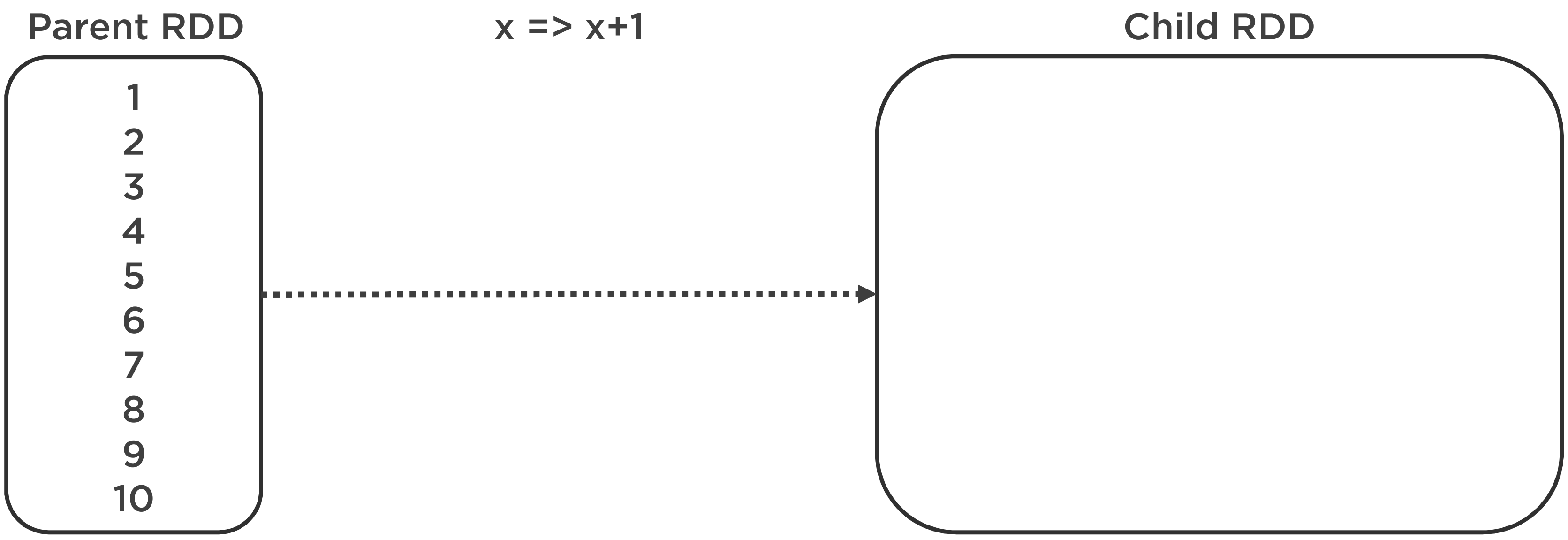
```
word_for_count = words.map(lambda x: (x,1))

word_for_count.take(1)

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

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

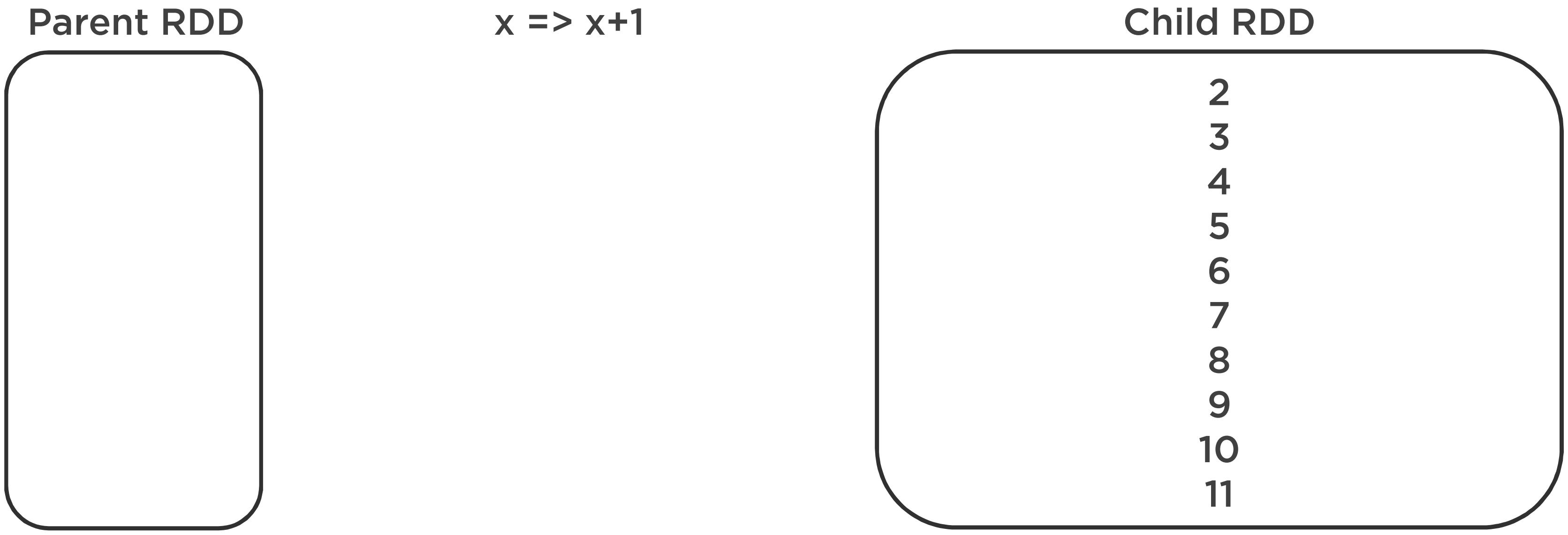
## Map

Apply function to each element

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

RDD of length N transformed to RDD of length N


Parent RDD
x => x+1
Child RDD
1
2
3
4
5
6
7
8
9
10


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

```
val word_for_count = words.map(x => (x,1))

word_for_count.take(1)

word_for_count.take(5)
```

## Map

**Apply function to each element**

**map(), mapPartitions(), mapValues(), mapPartitionsWithIndex() …**

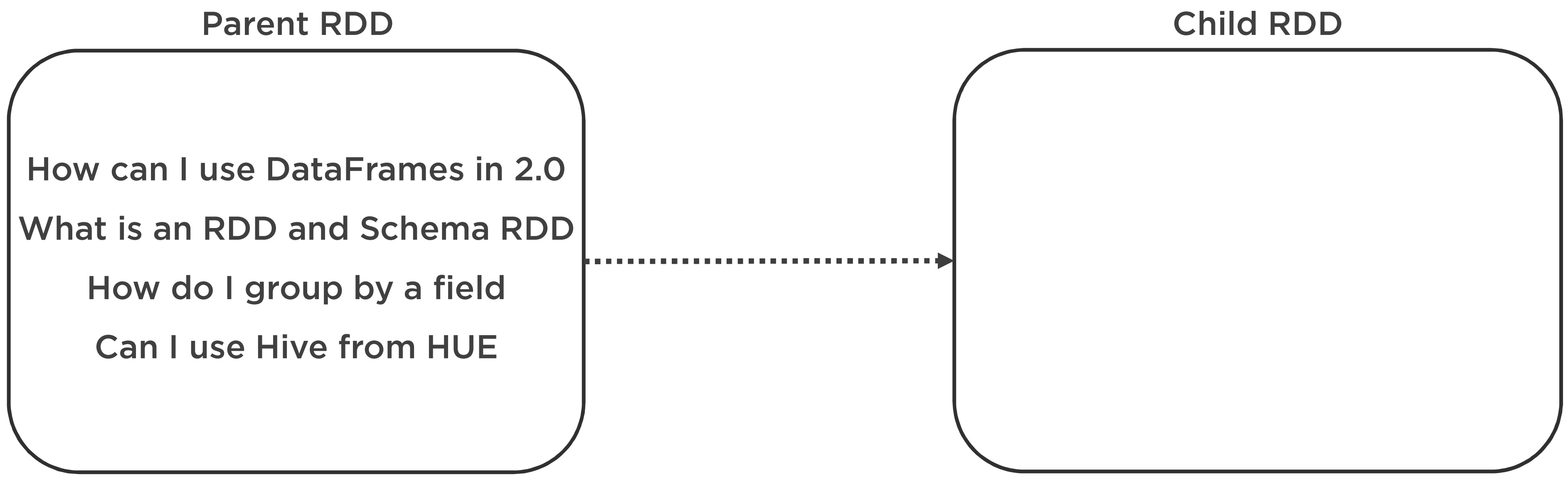
**Each element in parent RDD mapped to one element in the child RDD**

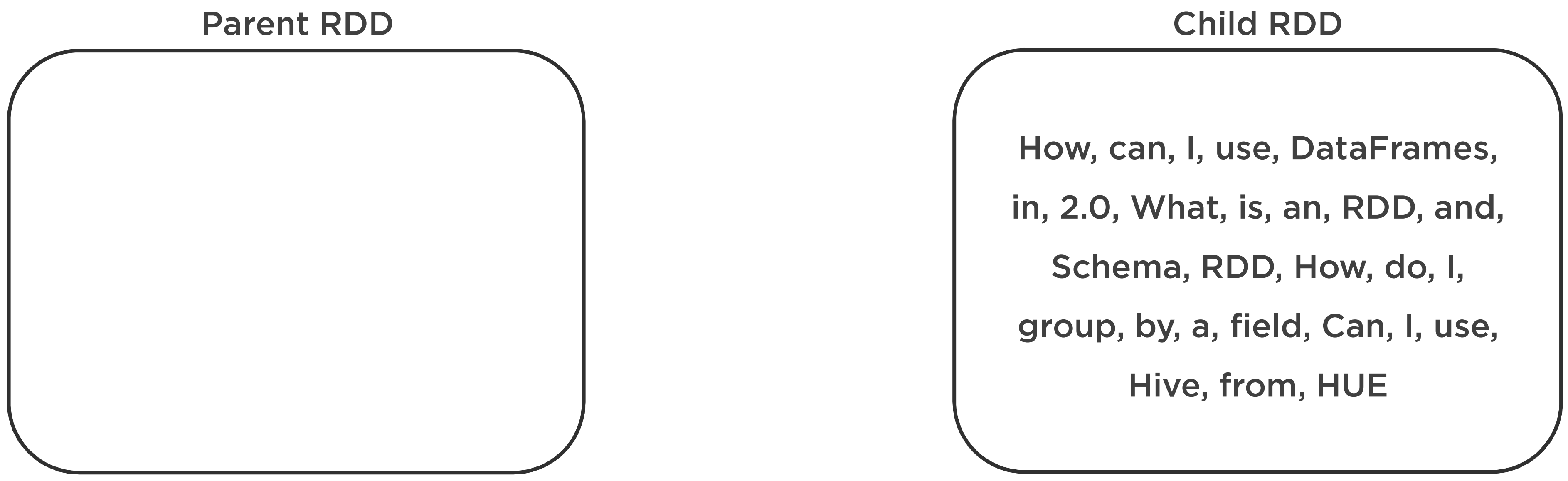
```
val words = lines.flatMap(line => line.split(" "))

words.collect()
```

## 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

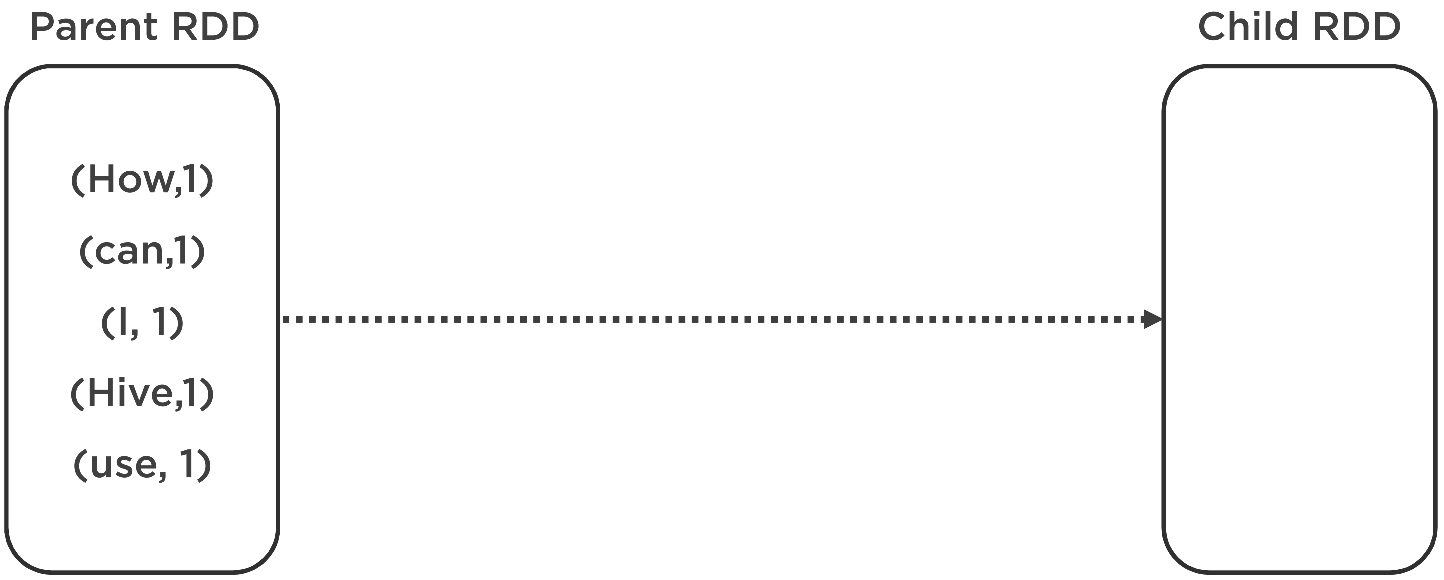
```
def starts_h(word: (String, Int)) =
  word._1.toLowerCase.startsWith("h")


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

## Filter

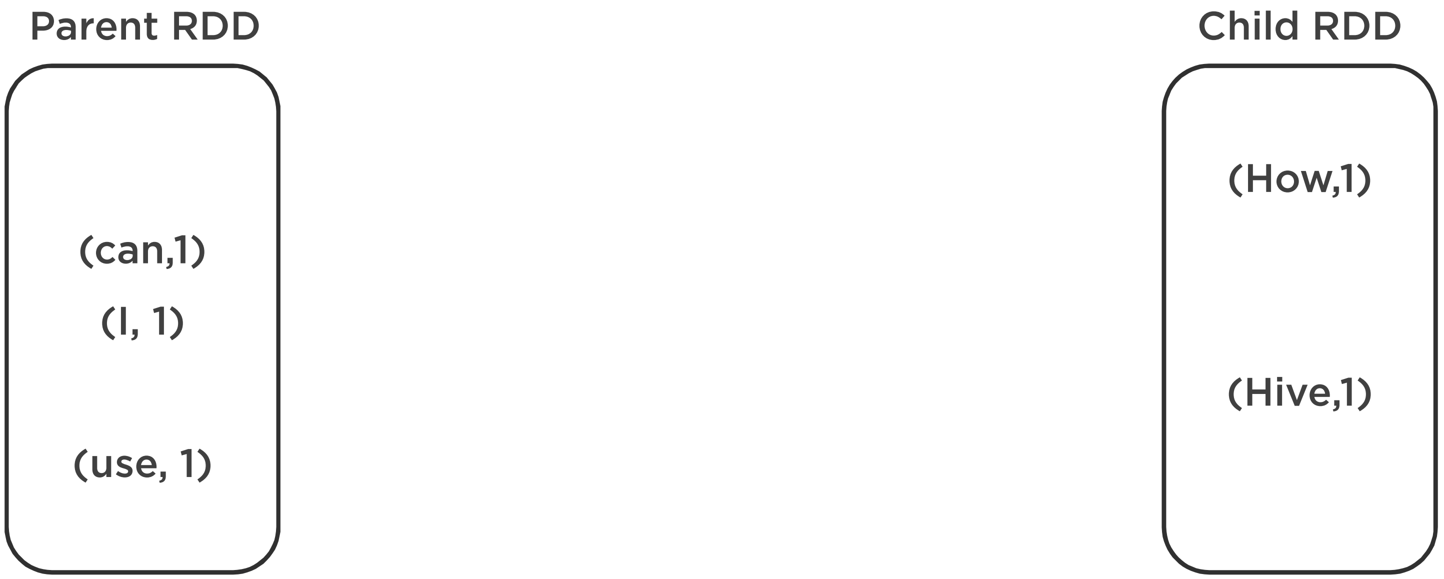
Apply a function to each element of the RDD

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

## Filter



## Filter



```
val word_count = word_for_count.reduceByKey(_ + _)

word_count.sortByKey().collect()

word_count.sortByKey(false).collect()

word_count.map({ case (x,y) => (y,x) }).sortByKey()
  .map(x => x.swap).collect()

word_count.sortBy({ case (x,y) => -y }).collect()
```

## 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…

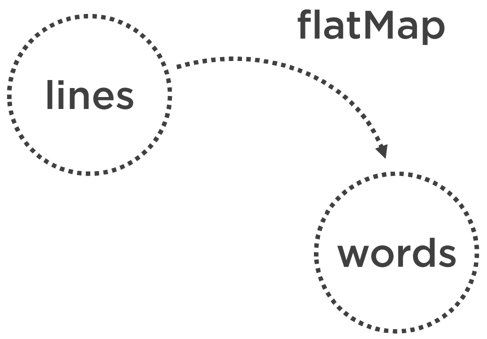
```
val lines = sc.textFile("/user/cloudera/se/simple_titles.txt")

val words = lines.flatMap(line => line.split(" "))

val word_for_count = words.map(x => (x,1))

val grouped_words = word_for_count.reduceByKey(_ + _)

grouped_words.collect()
```

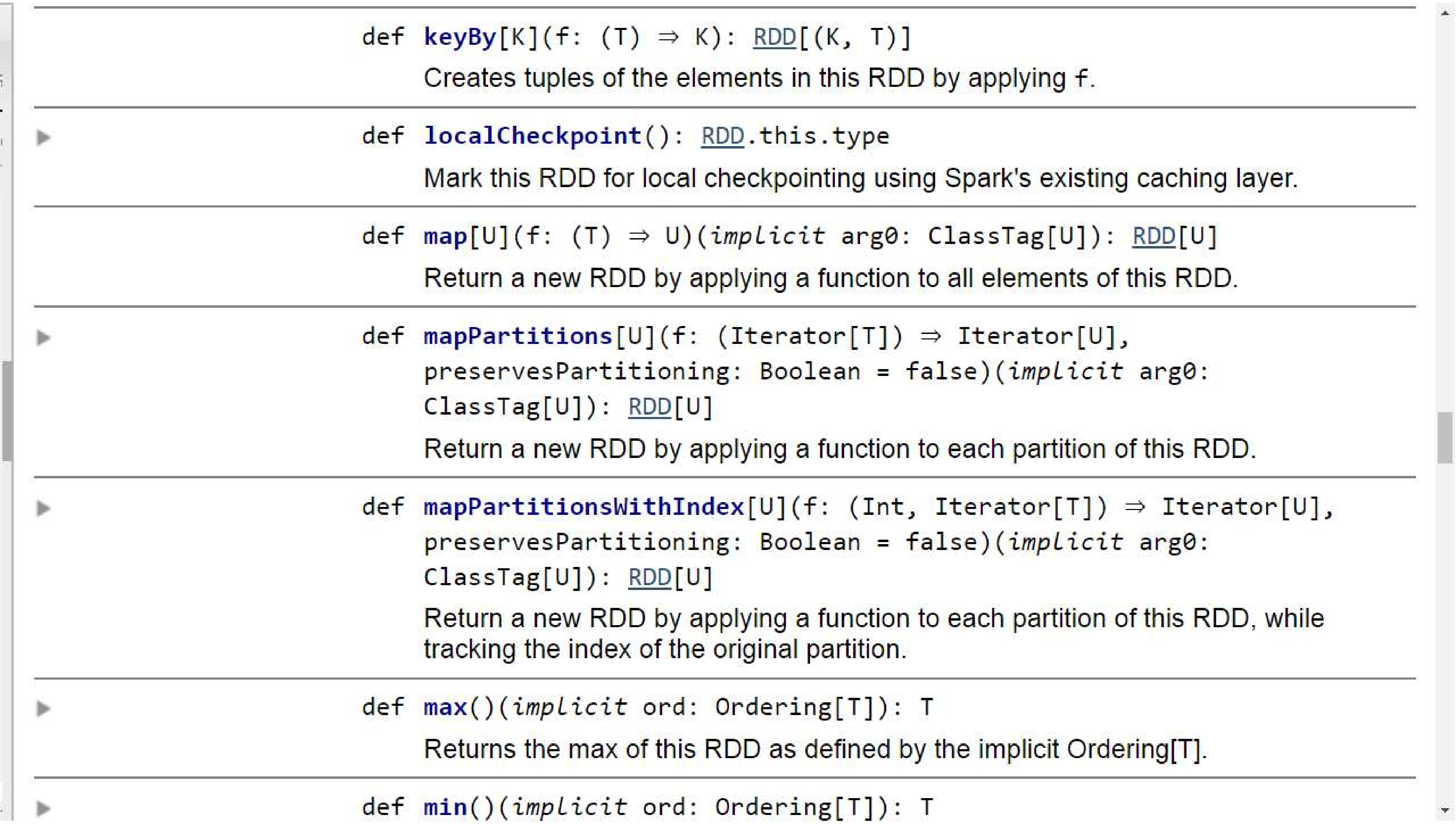
## Transformations

**Start with method from SparkContext to load data**

**Transformations perform a computation**

**And create new RDDs**





```
def keyBy[K](f: (T) ⇒ K): RDD[(K, T)]
```
Creates tuples of the elements in this RDD by applying `f`.

```
def localCheckpoint(): RDD.this.type
```
Mark this RDD for local checkpointing using Spark's existing caching layer.

```
def map[U](f: (T) ⇒ U)(implicit arg0: ClassTag[U]): RDD[U]
```
Return a new RDD by applying a function to all elements of this RDD.

```
def mapPartitions[U](f: (Iterator[T]) ⇒ Iterator[U],
    preservesPartitioning: Boolean = false)(implicit arg0:
    ClassTag[U]): RDD[U]
```
Return a new RDD by applying a function to each partition of this RDD.

```
def mapPartitionsWithIndex[U](f: (Int, Iterator[T]) ⇒ Iterator[U],
    preservesPartitioning: Boolean = false)(implicit arg0:
    ClassTag[U]): RDD[U]
```
Return a new RDD by applying a function to each partition of this RDD, while tracking the index of the original partition.

```
def max()(implicit ord: Ordering[T]): T
```
Returns the max of this RDD as defined by the implicit Ordering[T].

```
def min()(implicit ord: Ordering[T]): T
```

## 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

```
val lines = sc.textFile("/user/cloudera/se/simple_titles.txt")
val words = lines.flatMap(line => line.split(" "))
val word_for_count = words.map(x => (x,1))
val grouped_words = word_for_count.reduceByKey(_ + _)
grouped_words.collect()
```

## Previously on Transformations

**Transformations are what "changes" your data**

**Remember: Spark is lazy**

**No computation done when you specify transformation**

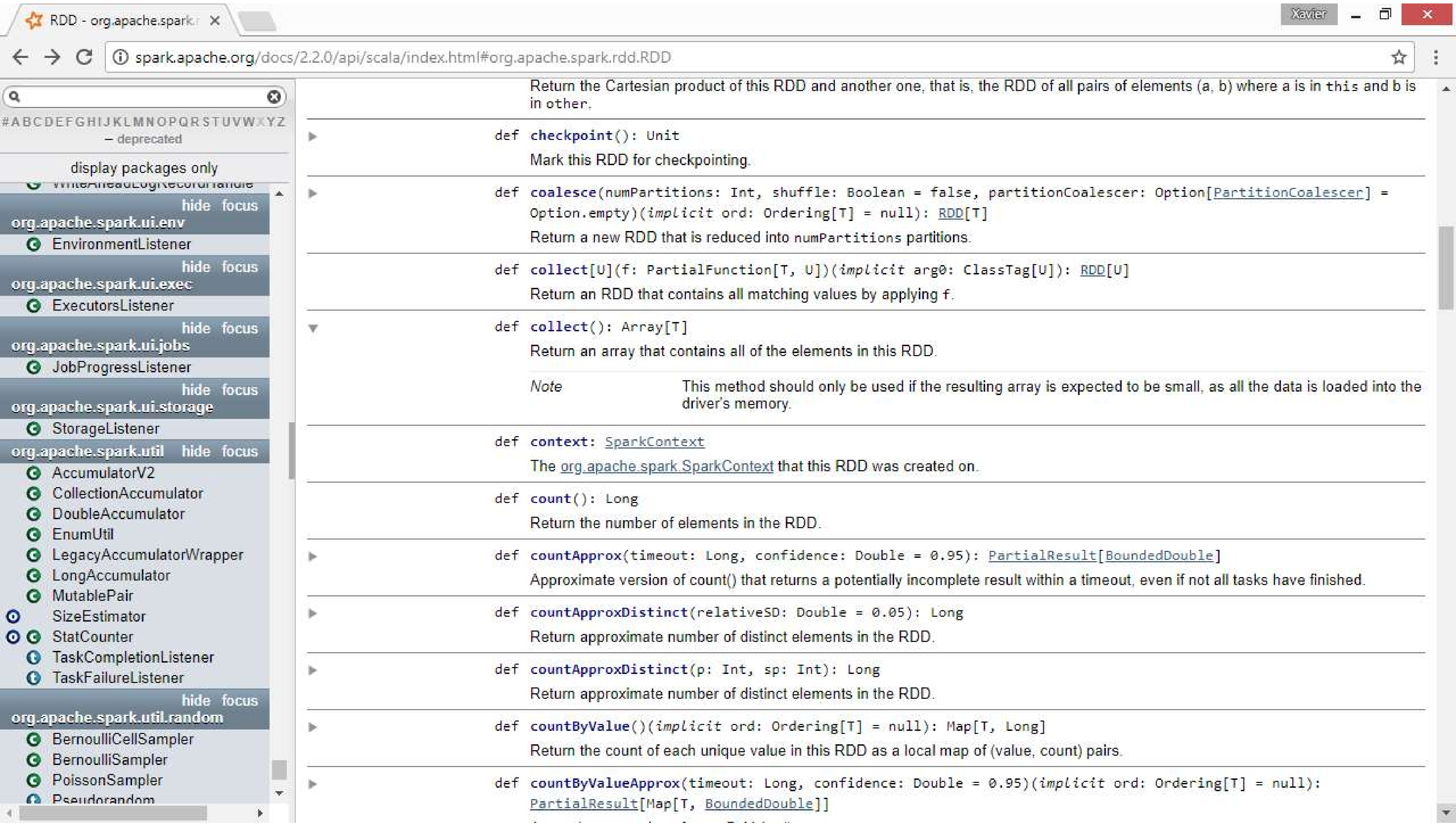
```
val lines = sc.textFile("/user/cloudera/se/simple_titles.txt")
val words = lines.flatMap(line => line.split(" "))
val word_for_count = words.map(x => (x,1))
val grouped_words = word_for_count.reduceByKey(_ + _)
grouped_words.collect()
grouped_words.saveAsTextFile("/user/cloudera/stackexchange/words")
```

## Actions

**Action triggers computation**

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

**Operations that produce non RDD values**



Return the Cartesian product of this RDD and another one, that is, the RDD of all pairs of elements (a, b) where a is in `this` and b is in `other`.

`def checkpoint(): Unit`

Mark this RDD for checkpointing.

`def coalesce(numPartitions: Int, shuffle: Boolean = false, partitionCoalescer: Option[PartitionCoalescer] = Option.empty)(implicit ord: Ordering[T] = null): RDD[T]`

Return a new RDD that is reduced into `numPartitions` partitions.

`def collect[U](f: PartialFunction[T, U])(implicit arg0: ClassTag[U]): RDD[U]`

Return an RDD that contains all matching values by applying `f`.

`def collect(): Array[T]`

Return an array 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.

`def context: SparkContext`

The org.apache.spark.SparkContext that this RDD was created on.

`def count(): Long`

Return the number of elements in the RDD.

`def countApprox(timeout: Long, confidence: Double = 0.95): PartialResult[BoundedDouble]`

Approximate version of count() that returns a potentially incomplete result within a timeout, even if not all tasks have finished.

`def countApproxDistinct(relativeSD: Double = 0.05): Long`

Return approximate number of distinct elements in the RDD.

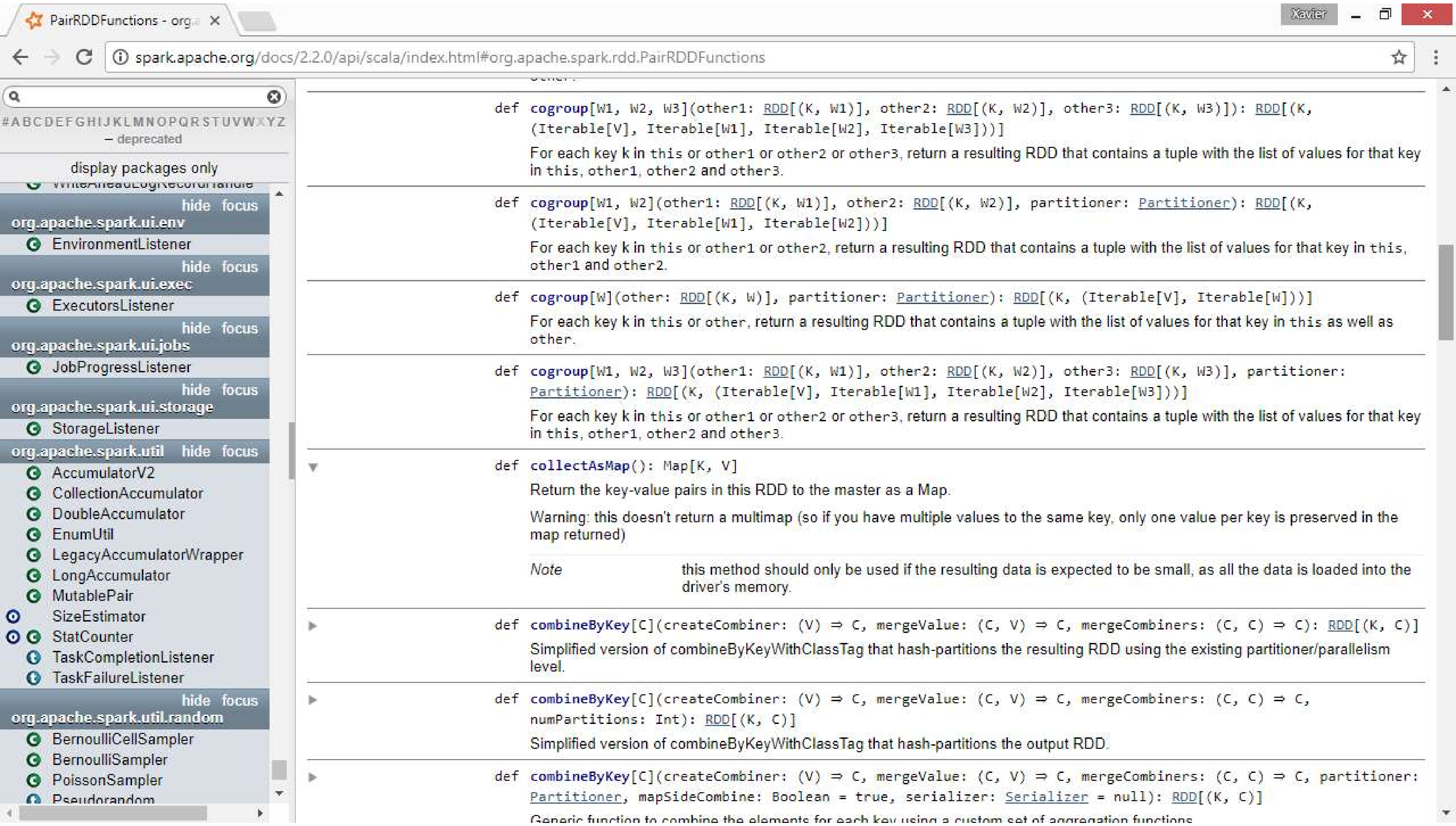
`def countApproxDistinct(p: Int, sp: Int): Long`

Return approximate number of distinct elements in the RDD.

`def countByValue()(implicit ord: Ordering[T] = null): Map[T, Long]`

Return the count of each unique value in this RDD as a local map of (value, count) pairs.

`def countByValueApprox(timeout: Long, confidence: Double = 0.95)(implicit ord: Ordering[T] = null): PartialResult[Map[T, BoundedDouble]]`



```
def cogroup[W1, W2, W3](other1: RDD[(K, W1)], other2: RDD[(K, W2)], other3: RDD[(K, W3)]): RDD[(K,
(Iterable[V], Iterable[W1], Iterable[W2], Iterable[W3]))]
```

For each key k in `this` or `other1` or `other2` or `other3`, return a resulting RDD that contains a tuple with the list of values for that key in `this`, `other1`, `other2` and `other3`.

```
def cogroup[W1, W2](other1: RDD[(K, W1)], other2: RDD[(K, W2)], partitioner: Partitioner): RDD[(K,
(Iterable[V], Iterable[W1], Iterable[W2]))]
```

For each key k in `this` or `other1` or `other2`, return a resulting RDD that contains a tuple with the list of values for that key in `this`, `other1` and `other2`.

```
def cogroup[W](other: RDD[(K, W)], partitioner: Partitioner): RDD[(K, (Iterable[V], Iterable[W]))]
```

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

```
def cogroup[W1, W2, W3](other1: RDD[(K, W1)], other2: RDD[(K, W2)], other3: RDD[(K, W3)], partitioner:
Partitioner): RDD[(K, (Iterable[V], Iterable[W1], Iterable[W2], Iterable[W3]))]
```

For each key k in `this` or `other1` or `other2` or `other3`, return a resulting RDD that contains a tuple with the list of values for that key in `this`, `other1`, `other2` and `other3`.

```
def collectAsMap(): Map[K, V]
```

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

Warning: this doesn't return a multimap (so if you have multiple values to the same key, only one value per key is preserved in the map returned)

*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.

```
def combineByKey[C](createCombiner: (V) ⇒ C, mergeValue: (C, V) ⇒ C, mergeCombiners: (C, C) ⇒ C): RDD[(K, C)]
```

Simplified version of combineByKeyWithClassTag that hash-partitions the resulting RDD using the existing partitioner/parallelism level.

```
def combineByKey[C](createCombiner: (V) ⇒ C, mergeValue: (C, V) ⇒ C, mergeCombiners: (C, C) ⇒ C,
numPartitions: Int): RDD[(K, C)]
```

Simplified version of combineByKeyWithClassTag that hash-partitions the output RDD.

```
def combineByKey[C](createCombiner: (V) ⇒ C, mergeValue: (C, V) ⇒ C, mergeCombiners: (C, C) ⇒ C, partitioner:
Partitioner, mapSideCombine: Boolean = true, serializer: Serializer = null): RDD[(K, C)]
```

# Actions

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

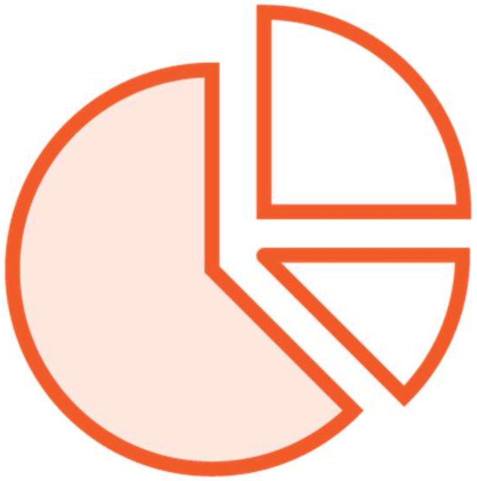
## How Does Spark Partition Data?

**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

```
val badges_for_part = badges_columns_rdd.map(x => (x(2),
x.mkString(","))).repartition(50)

badges_for_part.partitioner

import org.apache.spark.HashPartitioner
```

## PartitionBy

Returns an RDD partitioned using a specific partitioner

Useful to get keyed data into same partition

Not yet a group operation

```
val badges_by_badge = badges_for_part.
  partitionBy(new HashPartitioner(50))

badges_by_badge.partitioner

badges_for_part.saveAsTextFile("/user/cloudera/
  stackexchange/badges_no_partitioner")

badges_by_badge.saveAsTextFile("/user/cloudera/
  stackexchange/badges_yes_partitioner")
```

## PartitionBy

Create a function to be used for partitioning

Pass function as parameter to partitionBy()

Save with and without partitioner, and review results

```
badges_by_badge.map({ case (x,y) => x }).glom().take(1)
```

## 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

```
badges_by_badge.
  mapPartitions(x => Array(x.size).iterator, true)
  .collect()

badges_for_part.
  mapPartitions(x => Array(x.size).iterator, true)
  .collect()
```

## MapPartitions

**Apply a function to each partition**

**Done at a single pass**

**Returns after entire partition is processed**

```
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**

```
val sample_posts = posts_all.sample(false,0.1,50)

sample_posts.count()
```

## Sampling Data

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

- withReplacement
- fraction
- seed

```
posts_all.count()

posts_all.countApprox(100, 0.95)
```

## Approximate Counts

**Obtain an approximate count with countApprox()**

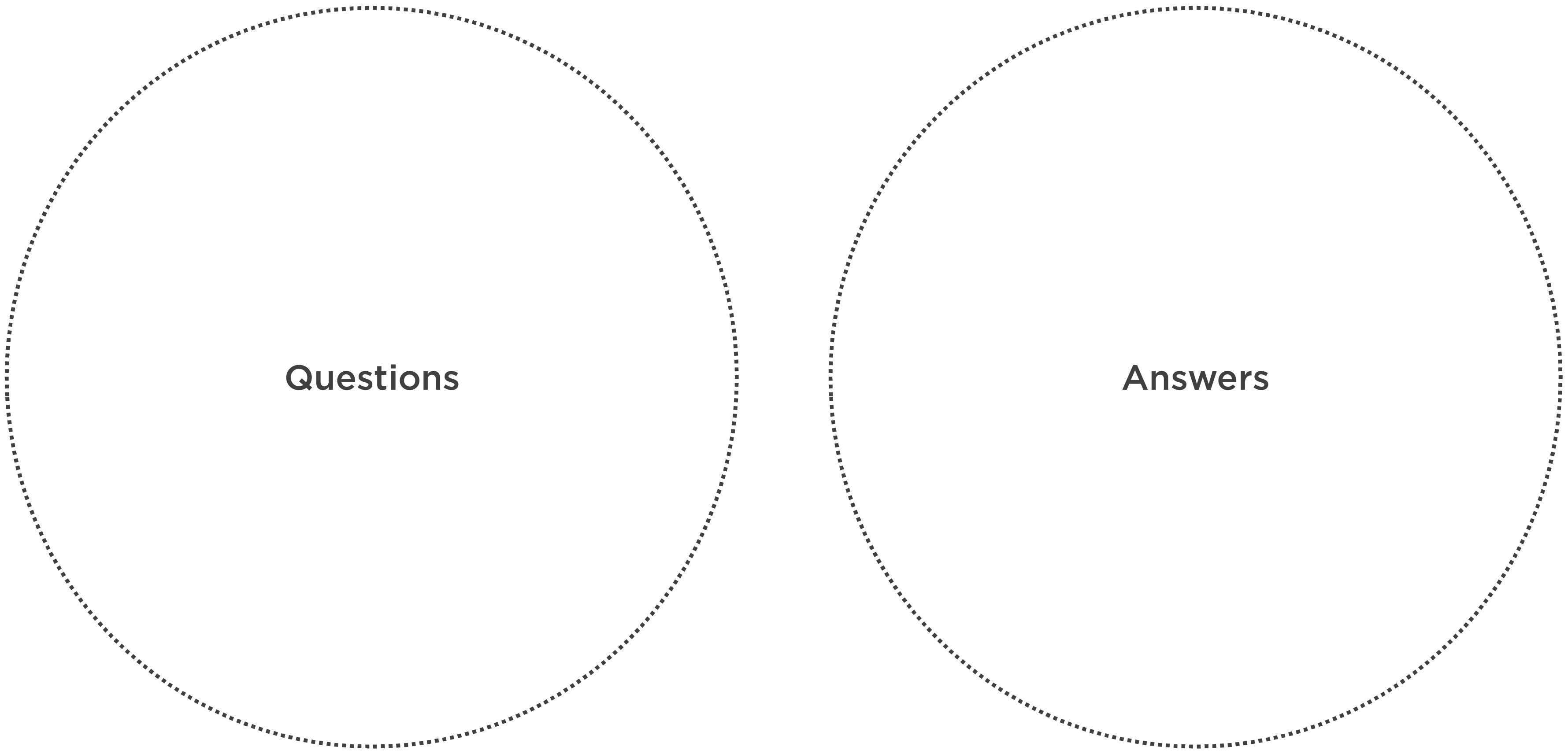
**Note:** Experimental

```
posts_all.takeSample(false,15,50)

posts_all.takeSample(false,15,50).size
```

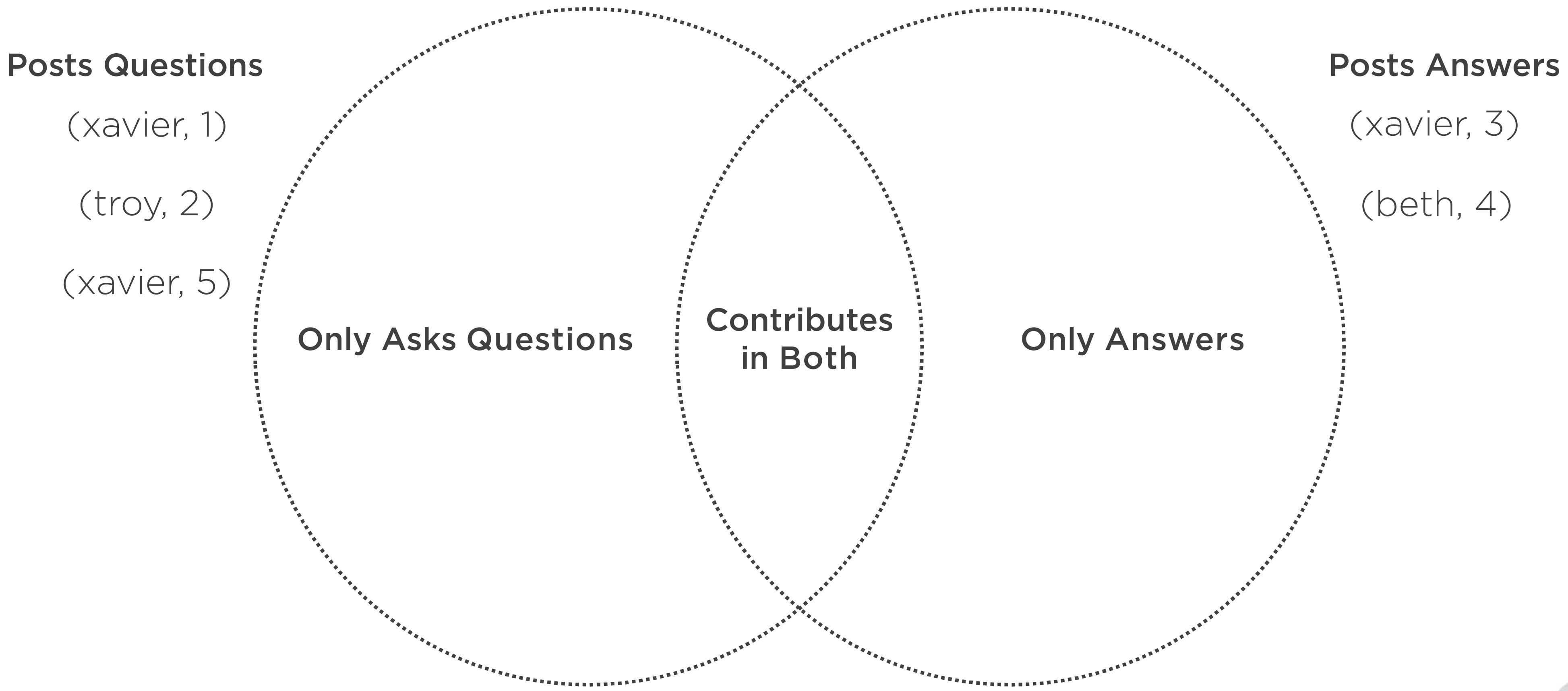
## Take a Sample of Exact Size

**Action available for exact count is called takeSample()**

## Set Operations



## Set Operations

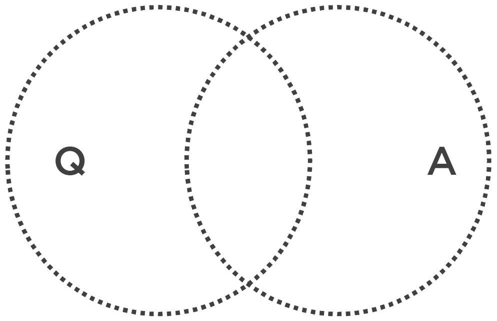


```
val questions =
sc.parallelize(Array(("xavier",1),("troy",2),("xavier",5)))

val answers =
sc.parallelize(Array(("xavier",3),("beth",4)))

questions.collect()

answers.collect()
```

## Our Data

**Create with parallelize**

**If you feel confident, go for the full dataset**



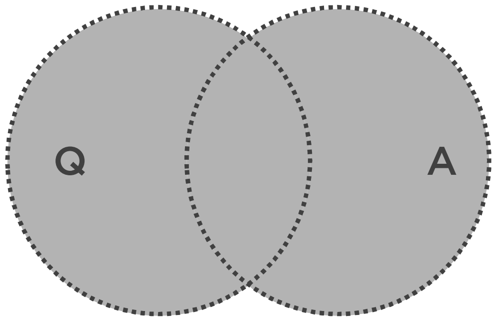
```
questions.union(answers).collect()

questions.union(questions).collect()

questions.union(sc
  .parallelize(Array("irene", "juli", "luci"))).collect()
```

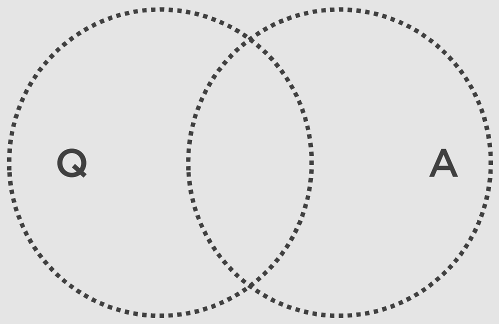
## Union

RDD with all elements in both RDDs

Questions + answers

Be careful with types





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

## Union

**All questions and answers**

**Elements remain the same**

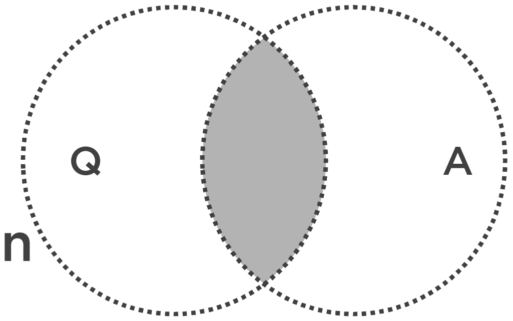
```
questions.join(answers).collect()
```

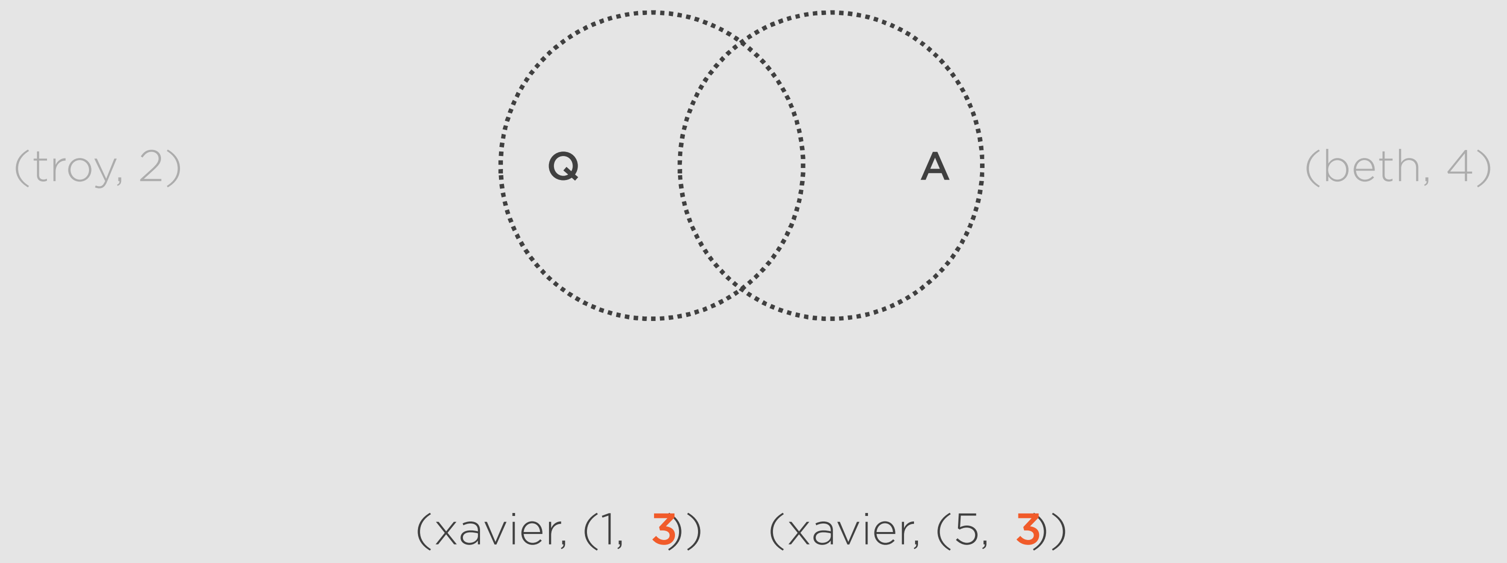
## Join

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

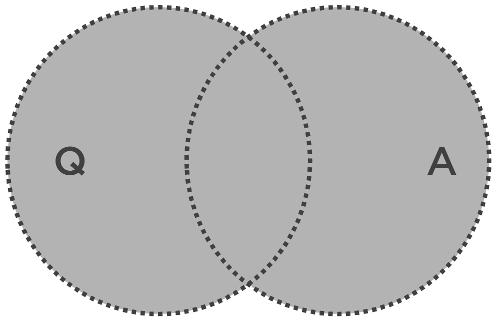
Excludes those that do not contribute

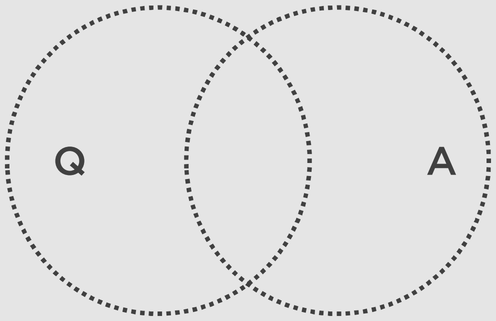
```
questions.fullOuterJoin(answers).collect()
```

## 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**

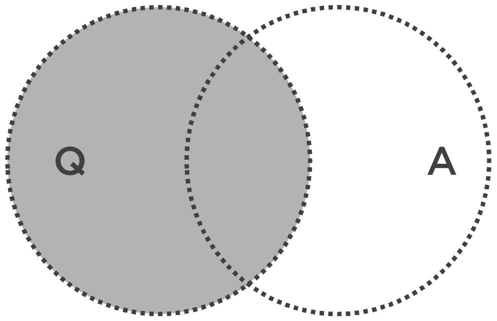
- None when user does not appear in one set

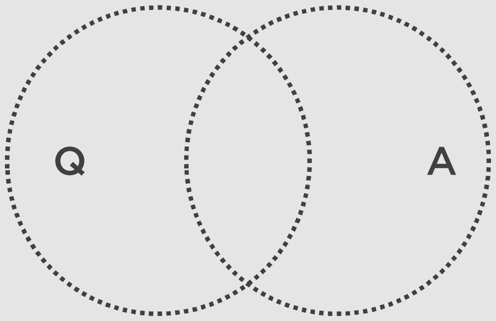
```
questions.leftOuterJoin(answers).collect()
```

## 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 on all objects from the left**

**List of all**

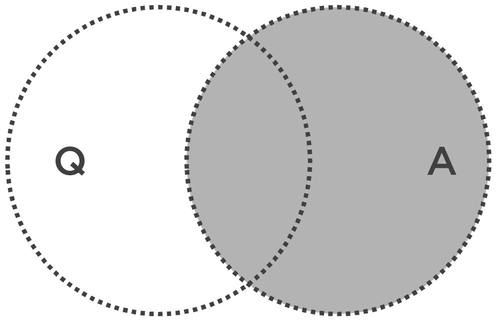
```
questions.rightOuterJoin(answers).collect()
```

## rightOuterJoin

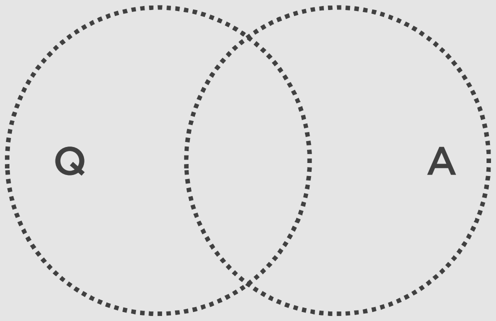
**Opposite of a leftOuterJoin**

**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

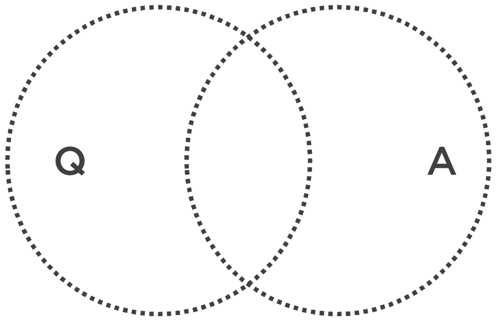
```
questions.leftOuterJoin(answers).collect()

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

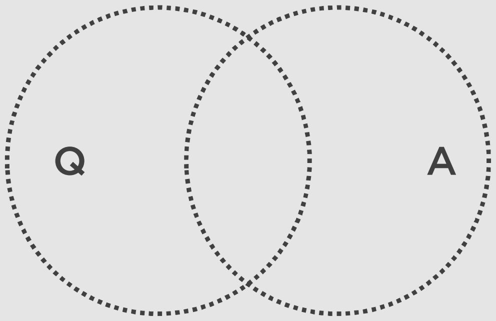
## leftOuterJoin and rightOuterJoin

**questions.leftOuterJoin(answers)**

**Equivalent to**

**answers.rightOuterJoin(questions)**



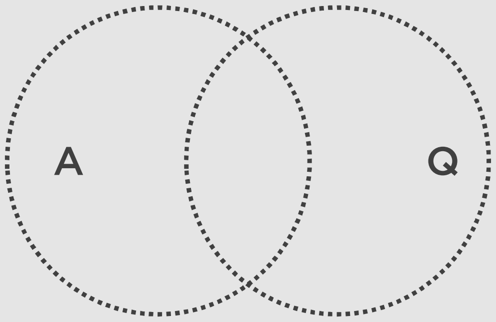


(beth, 4)

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

**rightOuterJoin**

(beth, 4)



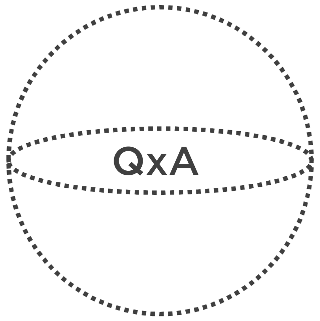
(xavier, (1, 3)) (xavier, (5, 3)) (troy, (2, None))

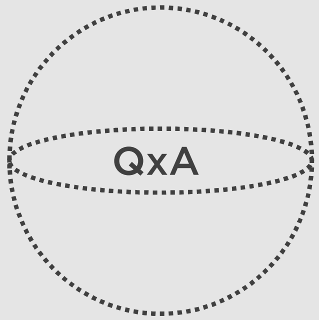
```
questions.cartesian(answers).collect()
```

## 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

## Aggregation

Grouping elements together

Foundations of Big Data analytics

```
posts_all.take(1)

val each_post_owner = posts_all.map(x => x.split(",")(6))

val posts_owner_pair_rdd = each_post_owner.map(x => (x,1))

posts_owner_pair_rdd.take(1)
```

## Prepare Some Data

**Extract user from each post**

**PairRDD**

- Key is user
- Value is 1

```
val top_posters_gbk = posts_owner_pair_rdd.groupByKey()

top_posters_gbk.take(10)
```

## GroupByKey

Values grouped by each key

Data sent over the network and collected on reduce workers

Can cause problems on larger datasets

```
top_posters_gbk.map({ case (x,y) => (x, y.toList) })
  .take(10)

top_posters_gbk.map({ case (x,y) => (x, y.size) }).take(10)

top_posters_gbk.map({ case (x,y) => (x, y.size) })
  .sortBy({ case (x, y) => -y}).take(1)
```

## GroupByKey

Tuple of user id and list of 1's

Posts per user? → User id and number of posts

Use sortBy for top poster

```
val top_posters_rbk = posts_owner_pair_rdd
  .reduceByKey(_ + _)
```

## ReduceByKey

**Perform an operation on all elements with same key**

**Specify a function**

**Reduce operation done within partition**

```
top_posters_rbk.lookup("51")
```

## ReduceByKey

**Perform add using _ + _**

**Pass function as parameter 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**

## Preparation for aggregateByKey

```
val posts_all_entries = posts_all.map(x => x.split(","))

val questions = posts_all_entries.filter(x => x(1) == "1")

val user_question_score = questions.map(x =>
(x(6),x(4).toInt))

user_question_score.take(5).foreach(println)
```

```
val posts_all_entries = posts_all.map(x => x.split(","))

val questions = posts_all_entries.filter(x => x(1) == "1")

val user_question_score = questions.map(x =>
  (x(6),x(4).toInt))

user_question_score.take(5).foreach(println)
```

## aggregateByKey

Like reduceByKey()

But takes an initial value

Specify functions for merging and combining

```
var for_keeping_count = (0,0)

def combining (tuple_sum_count: (Int, Int), next_score: Int) =
  (tuple_sum_count._1 + next_score, tuple_sum_count._2 + 1)

def merging (tuple_sum_count: (Int, Int),
tuple_next_partition_sum_count: (Int, Int)) = (tuple_sum_count._1 +
  tuple_next_partition_sum_count._1, tuple_sum_count._2 +
  tuple_next_partition_sum_count._2)

val aggregated_user_question =
  user_question_score.aggregateByKey(for_keeping_count)(combining,
  merging)

aggregated_user_question.take(1)

aggregated_user_question.lookup("51")
```

```
val aggregated_user_question =
  user_question_score.aggregateByKey(for_keeping_count)
  (combining, merging)
```

## 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

```
val user_post = questions.map(x => (x(6), x(0).toInt))

def to_list(postid: Int): List[Int] = List(postid)

def merge_posts(posta: List[Int], postb: Int) = postb ::
posta

def combine_posts(posta: List[Int], postb: List[Int]):
List[Int] = posta ++ postb
```

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

combined.filter({ case (x,y) => x == "51" }).collect()
```

## 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**

```
word_for_count.groupByKey().count()

word_for_count.reduceByKey(_ + _).count()
```

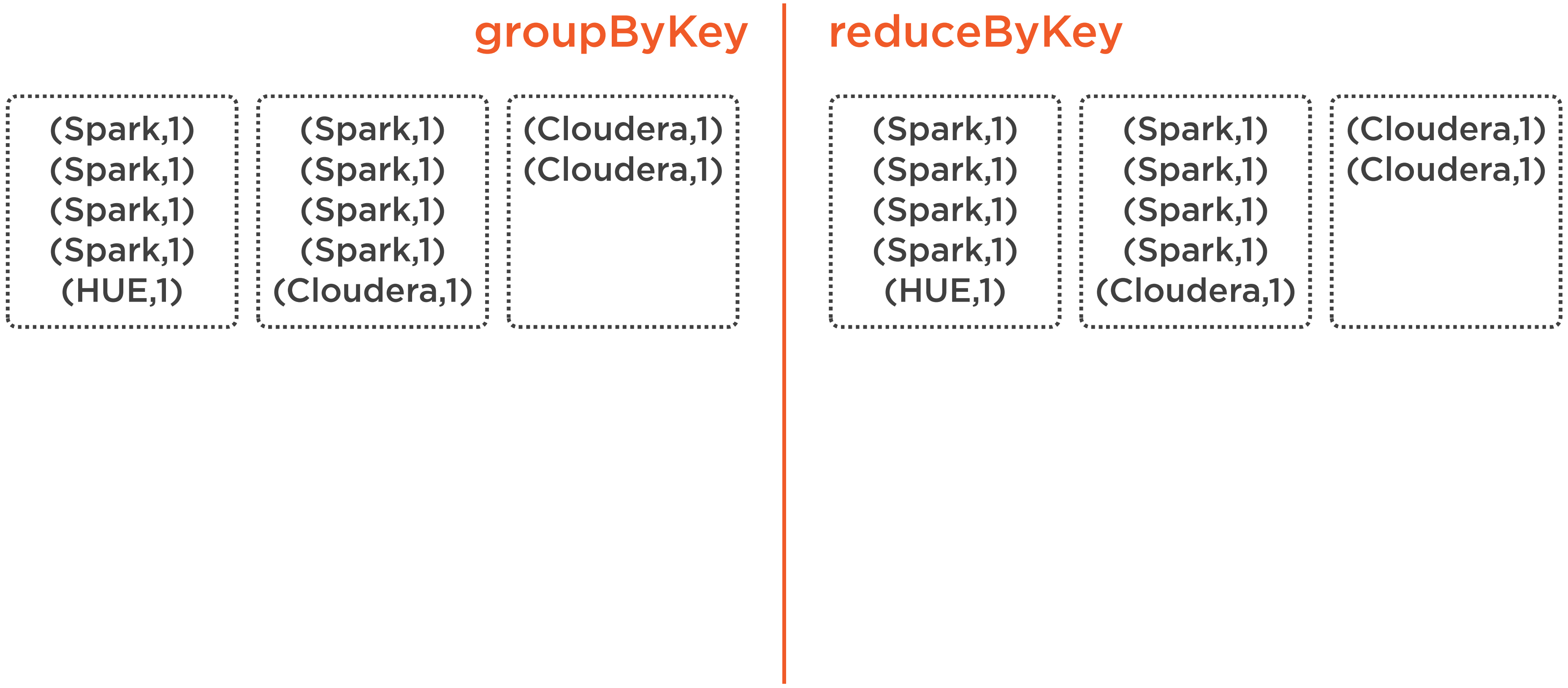
## reduceByKey & groupByKey

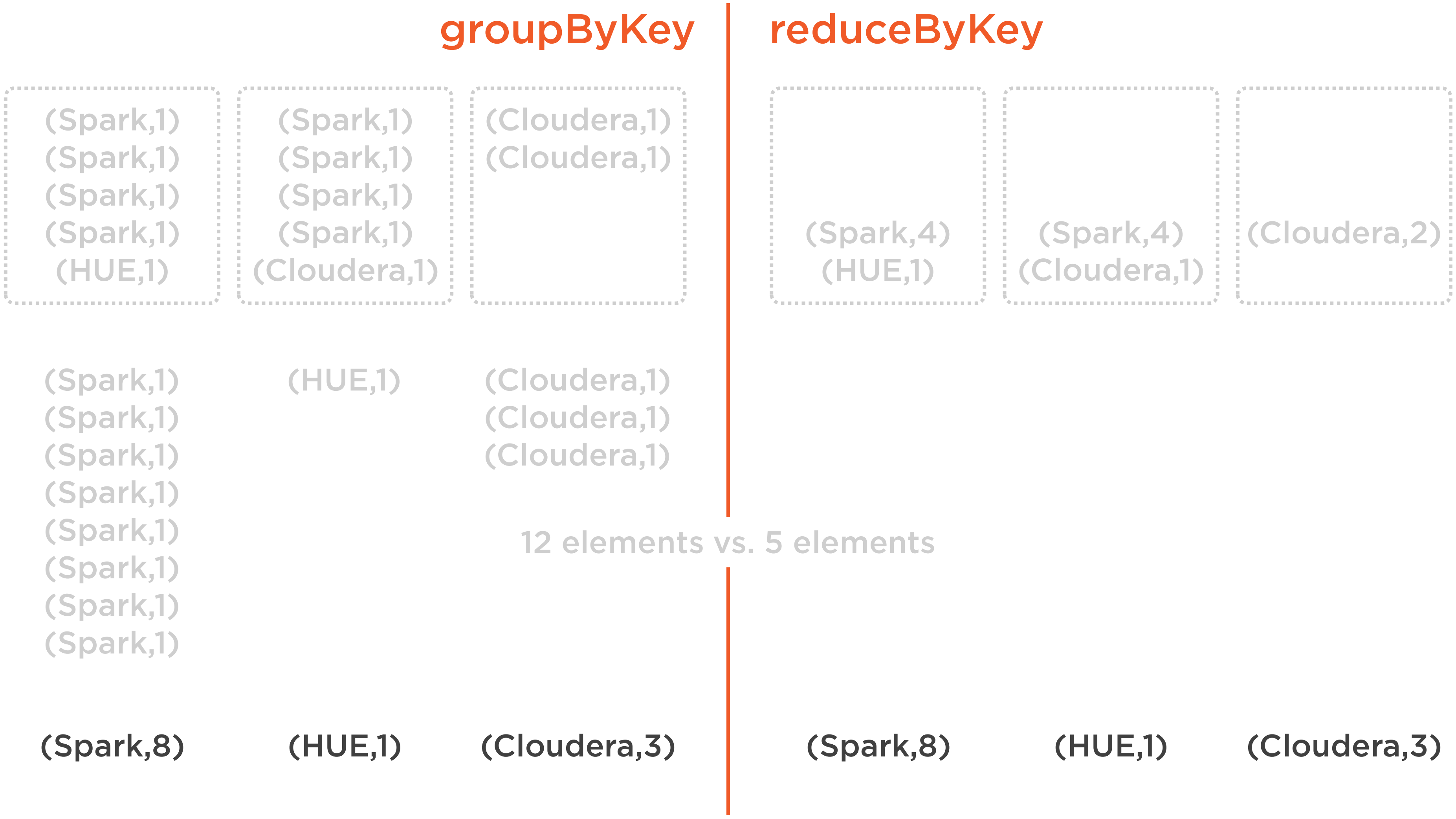
**Both can be used for the same purpose**

**Aggregate by keys**

**Work very differently underneath**

# Comparing groupByKey vs. reduceByKey




groupByKey
reduceByKey
(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,4) (HUE,1)
(Spark,4) (Cloudera,1)
(Cloudera,2)
(Spark,1) (Spark,1) (Spark,1) (Spark,1) (Spark,1) (Spark,1) (Spark,1) (Spark,1)
(HUE,1)
(Cloudera,1) (Cloudera,1) (Cloudera,1)
12 elements vs. 5 elements
(Spark,8)
(HUE,1)
(Cloudera,3)
(Spark,8)
(HUE,1)
(Cloudera,3)

## 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.take(10)

badges_reduced.map({ case (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**

```
val intervals: Array[Double] =
  Array(0,1000,2000,3000,4000,5000,6000,7000)

badges_reduced.map({ case (x,y) => y }).histogram(intervals)

badges_reduced.sortBy(x => -x._2).take(10)

badges_reduced.filter(x => x._2 < 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

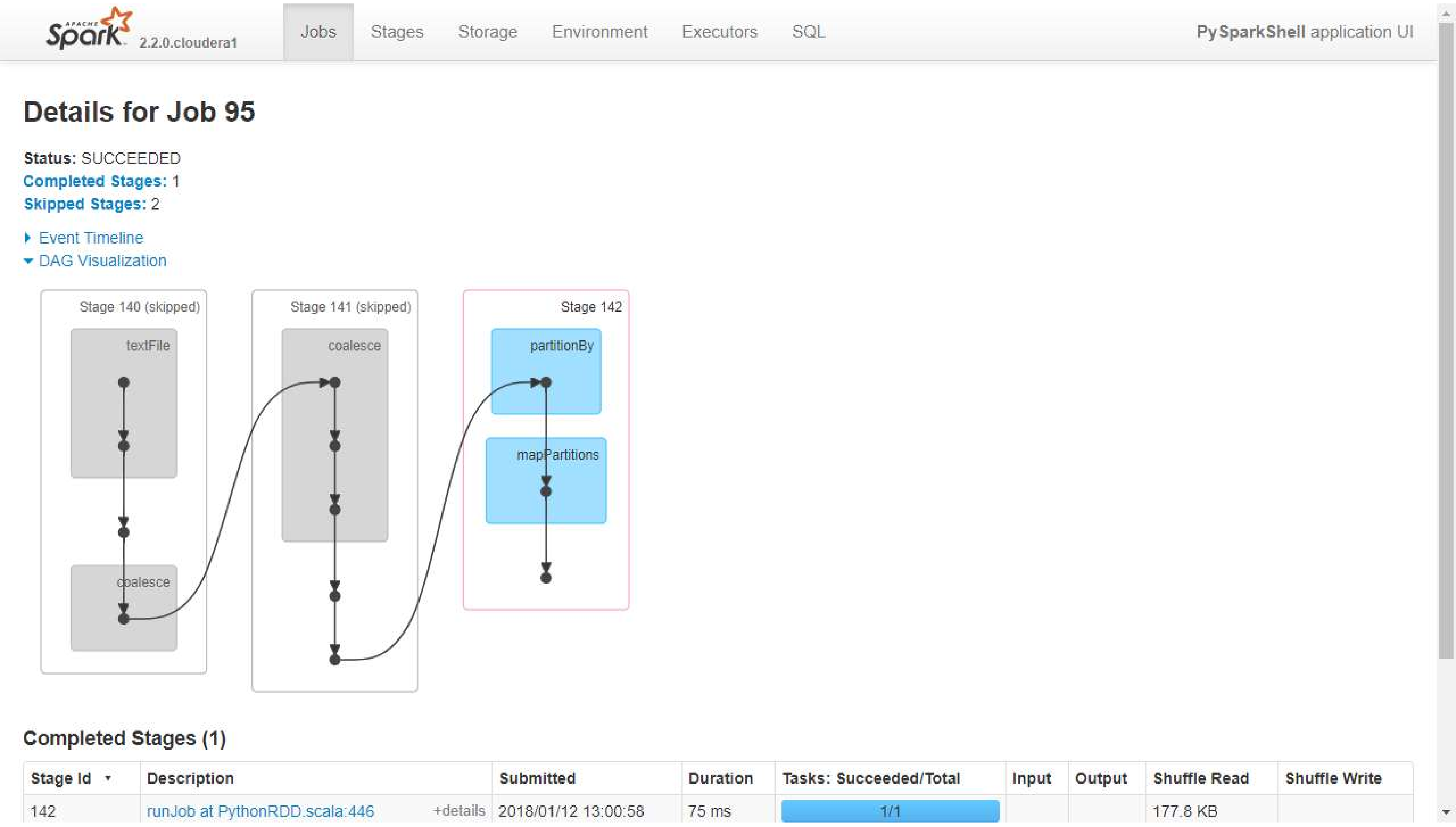
Persist to disk, memory or both

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

reduced.cache()
```

## Cache & Persist

**Spark may perform caching of intermediate results**

- On expensive operations, to avoid recomputing when nodes fail



## Details for Job 95

**Status:** SUCCEEDED
**Completed Stages:** 1
**Skipped Stages:** 2

### 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 | |

```
reduced.cache()
```

## 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**

```
import org.apache.spark.storage.StorageLevel

grouped.persist(StorageLevel.DISK_ONLY)
```

## Cache & Persist

**Call explicitly cache() and persist() when beneficial**

- cache() is equivalent to persist(MEMORY_ONLY)

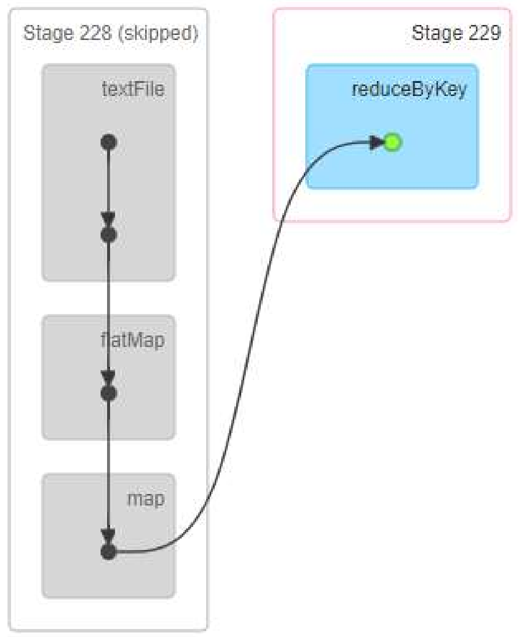
**When RDD not needed anymore, call unpersist()**

## Storage Levels

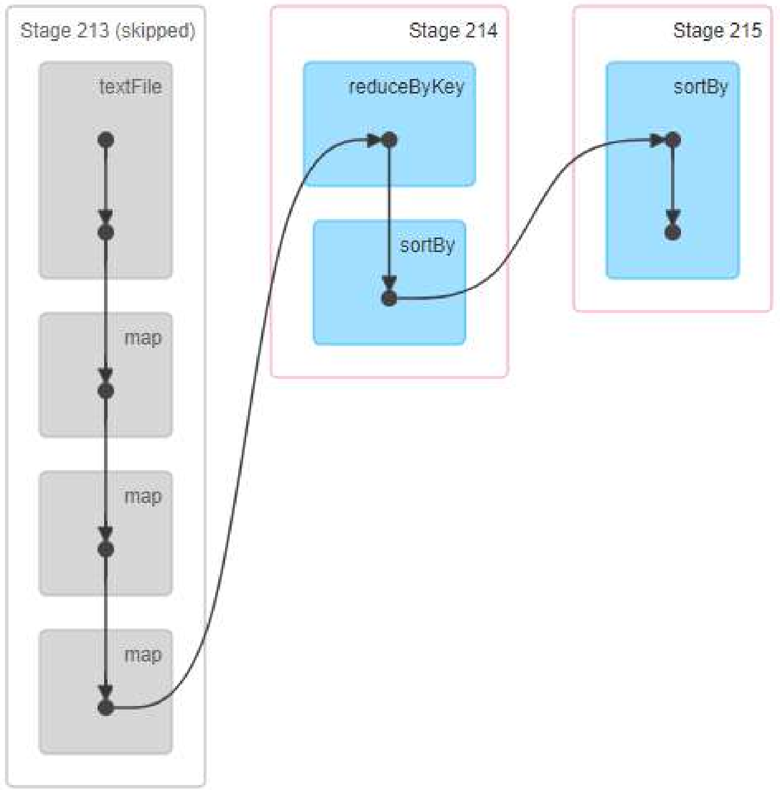
MEMORY_ONLY
MEMORY_AND_DISK
DISK_ONLY
MEMORY_ONLY_2,
MEMORY_AND_DISK_2, etc.
OFF_HEAP (experimental)

MEMORY_ONLY_SER
MEMORY_AND_DISK_SER

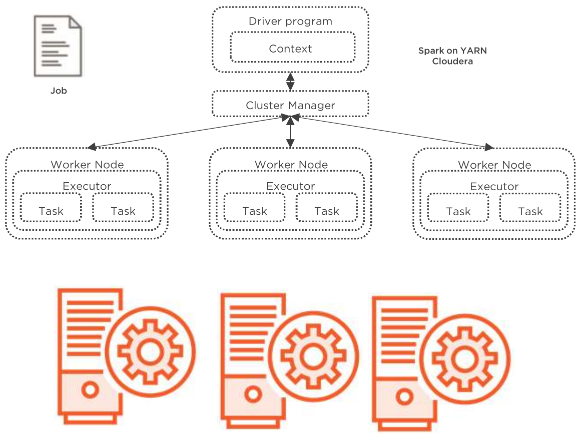
# Cache & Persist



## Cache in Spark UI



# 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 and broadcast variables

## Shared Variables

| Accumulators | Broadcast Variables |
|---|---|
| "Added" | Read only variable |
| Associate and commutative | Immutable |
| Numeric accumulator | Fits in memory |
| Other types possible | Distributed efficiently to the cluster |
| Counter is one common scenario | Do not modify after shipped |
| Accumulator may not be reliable | Good case is a lookup table |
| Case of failed task | |
| Potential duplicate counts | |

```
val accumulator_badge = sc.
  longAccumulator("Badge Accumulator")

def add_badge(item: (String, String)) =
  accumulator_badge.add(1)

badges_by_badge.foreach(add_badge)

accumulator_badge.value
```

## Accumulator

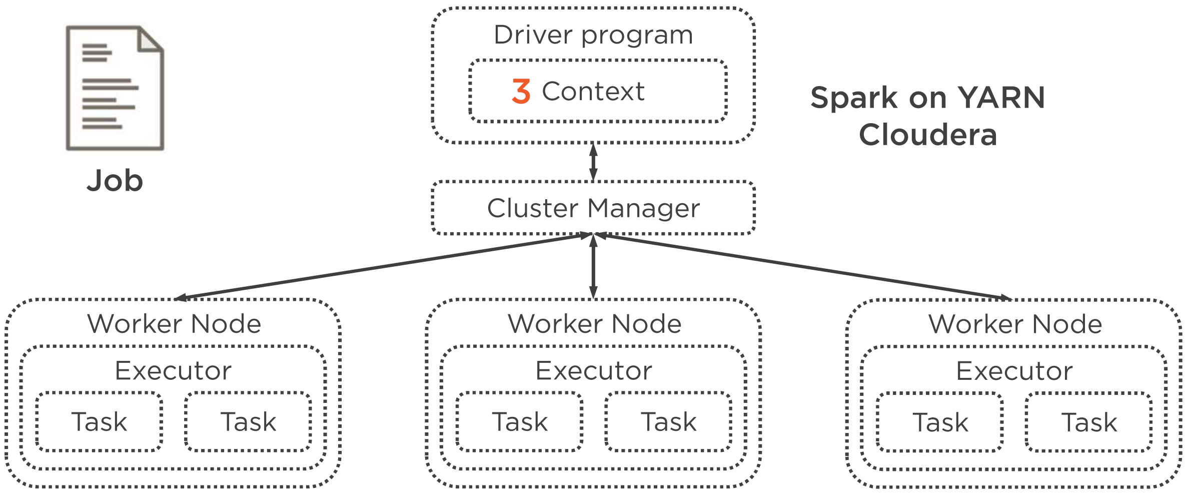
**Create accumulator and check current value**

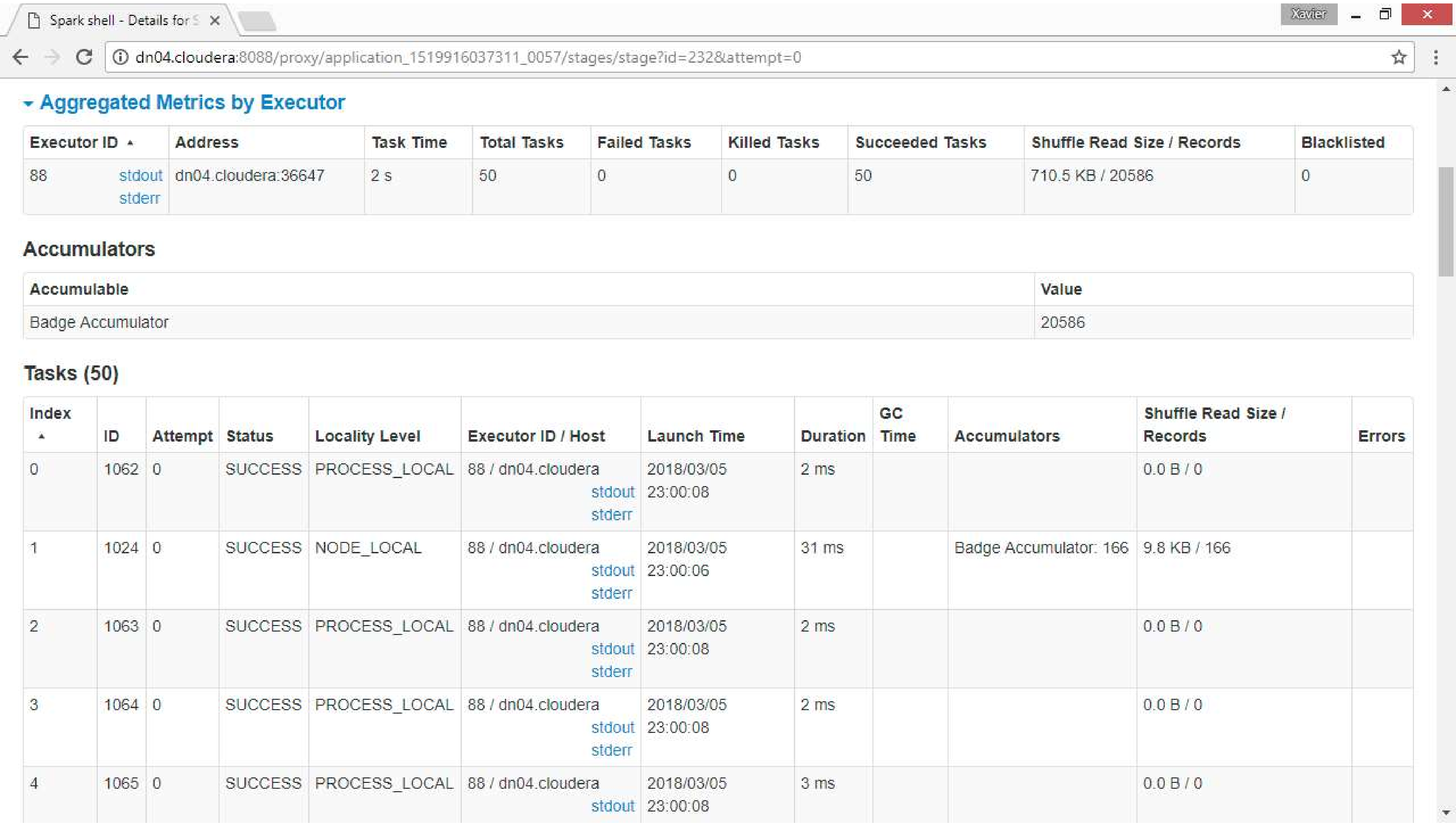
**Increment accumulator function and run**

**Get value**

# Accumulators

Executors write to accumulator in Driver program





Spark shell - Details for S

dn04.cloudera:8088/proxy/application_1519916037311_0057/stages/stage?id=232&attempt=0

Xavier

## Aggregated Metrics by Executor

| Executor ID ▲ | Address | Task Time | Total Tasks | Failed Tasks | Killed Tasks | Succeeded Tasks | Shuffle Read Size / Records | Blacklisted |
|---|---|---|---|---|---|---|---|---|
| 88 stdout stderr | dn04.cloudera:36647 | 2 s | 50 | 0 | 0 | 50 | 710.5 KB / 20586 | 0 |

## Accumulators

| Accumulable | Value |
|---|---|
| Badge Accumulator | 20586 |

## Tasks (50)

| Index ▲ | ID | Attempt | Status | Locality Level | Executor ID / Host | Launch Time | Duration | GC Time | Accumulators | Shuffle Read Size / Records | Errors |
|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 1062 | 0 | SUCCESS | PROCESS_LOCAL | 88 / dn04.cloudera stdout stderr | 2018/03/05 23:00:08 | 2 ms | | | 0.0 B / 0 | |
| 1 | 1024 | 0 | SUCCESS | NODE_LOCAL | 88 / dn04.cloudera stdout stderr | 2018/03/05 23:00:06 | 31 ms | | Badge Accumulator: 166 | 9.8 KB / 166 | |
| 2 | 1063 | 0 | SUCCESS | PROCESS_LOCAL | 88 / dn04.cloudera stdout stderr | 2018/03/05 23:00:08 | 2 ms | | | 0.0 B / 0 | |
| 3 | 1064 | 0 | SUCCESS | PROCESS_LOCAL | 88 / dn04.cloudera stdout stderr | 2018/03/05 23:00:08 | 2 ms | | | 0.0 B / 0 | |
| 4 | 1065 | 0 | SUCCESS | PROCESS_LOCAL | 88 / dn04.cloudera stdout | 2018/03/05 23:00:08 | 3 ms | | | 0.0 B / 0 | |

```
cd users

sbt package

spark2-submit --class "PrepareUsersApp"
               target/scala-2.11/users-project_2.11-1.0.jar
```

Convert Users.xml to CSV

## Data preparation step

```
val users_all =
sc.textFile("/user/cloudera/stackexchange/users_csv")

users_all.take(10)

val users_columns = users_all.map(split_the_line)

users_columns.take(3)


top_posters_rbk.take(10)

top_posters_rbk.lookup("51")
```

```
def get_name(user_column: Array[String]) = {
  val user_id = user_column(0)
  val user_name = user_column(3)
  var user_post_count = "0"
  if (broadcast_tp.value.keySet.exists(_ == user_id))
    user_post_count = broadcast_tp.value(user_id).toString
  (user_id, user_name, user_post_count)
}
```

```
val tp = top_posters_rbk.collectAsMap()

val broadcast_tp = sc.broadcast(tp)
```

## Broadcast Variable

**Create a broadcast variable using the context**

**Access when necessary,  i.e. lookup**

**Use value**

```
val user_info = users_columns.map(get_name)

user_info.take(10)
```

## Broadcast Variable

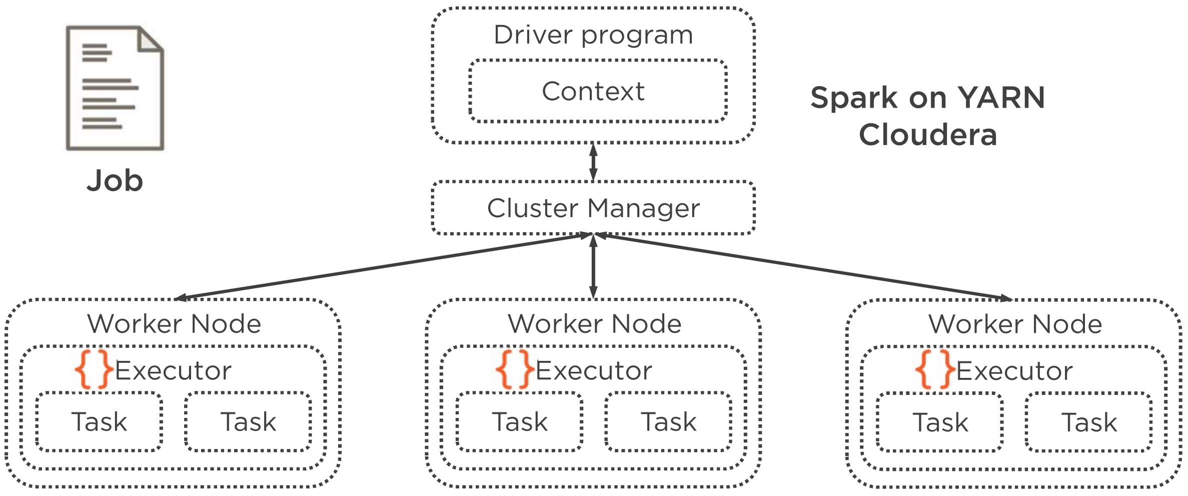
**Create using sc.broadcast()**

- Assign to a variable

**Access using variable.value**

## Broadcast Variables

Executors read from Broadcast variable



## Developing Self-contained Spark Apps

**Requires**

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

```
import org.apache.spark.{SparkConf, SparkContext}

val conf = new SparkConf()

  .setMaster("yarn")

  .setAppName("Self Contained Application")
sc = new SparkContext(conf)
```

## Creating the SparkContext

**Corresponding import**

**Create sc**

```
cd users

sbt package
```

## Bundling Your Application

**Use sbt or Maven to create an "uber jar"**

**Use sbt package, respect folder structure**

**We covered in the Scala refresher module**

```
spark2-submit --class "PrepareUsersApp"
              target/scala-2.11/users-project_2.11-1.0.jar
```

## Launching Application

**Using spark2-submit**

**Code to be executed, submitted as a job**

```
spark2-submit --class "PrepareUsersApp"
              --jars my_dependency.jar
              --packages com.databricks:spark-xml_2.11:0.4.1
               target/scala-2.11/users-project_2.11-1.0.jar
```

## Dependencies

**Use jars parameter**

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

**Maven coordinates with packages**

## Disadvantages of RDDs

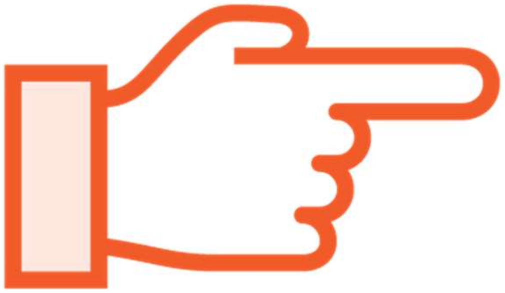
Don't take this the wrong way

RDDs are still used, even internally

Extremely powerful

Limitations on 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