
Apache Spark Core

(RDDs, Transformations, Actions, Persistence)

Nagiza F. Samatova, samatova@csc.ncsu.edu

**Professor, Department of Computer Science
North Carolina State University**

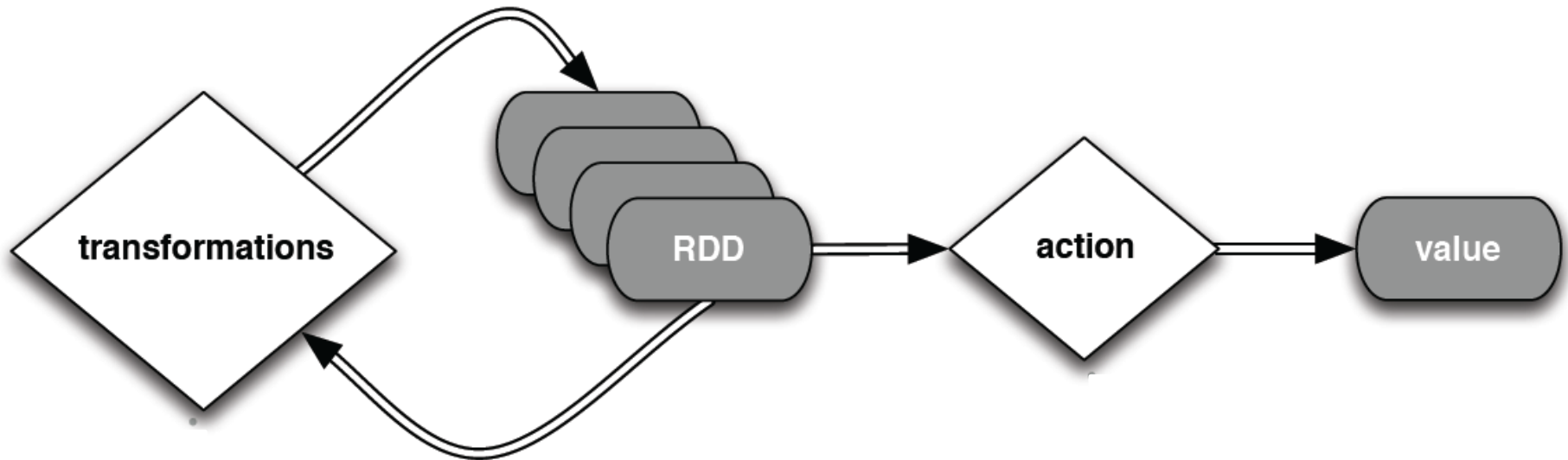
**Senior Scientist, Computer Science & Mathematics Division
Oak Ridge National Laboratory**

Outline

- **Spark Program Elements**
- **SparkContext: sc**
- **RDD: Resilient Distributed**
- **Operations on RDDs: Transformations**
- **Operations on RDDs: Actions**
- **Passing Functions**
- **Lazy Evaluation and Persistence**

Every Spark program will

1. Create SparkContext object (**SC**)
2. Create some input **RDDs** from external data
3. Transform RDDs to define new RDDs using **transformations** like filter()
4. Ask Spark to **persist** any intermediate RDDs for reuse
5. Launch **actions** such as count() to kick off a parallel computation, which is then optimized and executed by Spark



Python API

- **The Python API is structured differently than Java and Scala**
- **In Python all of the functions are implemented on the base RDD class but will fail at runtime if the type of data in the RDD is incorrect**

Outline

- Spark Program Elements
- **SparkContext: sc**
- RDD: Resilient Distributed
- Operations on RDDs: Transformations
- Operations on RDDs: Actions
- Passing Functions
- Lazy Evaluation and Persistence

SparkContext: sc

- Main entry point to Spark functionality
- Available in shell as variable **sc**
- In standalone programs, you'd make your own

Python

```
from pyspark import SparkContext
```

```
sc = SparkContext("masterUrl", "name", "sparkHome", ["library.py"])
```

Cluster URL, or
local / local[N]

App
name

Spark install
path on cluster

List of JARs with
app code (to ship)

Outline

- Spark Program Elements
- SparkContext: sc
- **RDD: Resilient Distributed**
- Operations on RDDs: Transformations
- Operations on RDDs: Actions
- Passing Functions
- Lazy Evaluation and Persistence

RDD: Resilient Distributed Dataset

- **Resilient Distributed Dataset (RDD)** is Spark's core abstraction for working with distributed data
- RDD in Spark is an *immutable* distributed collection of objects/records
- Under the hood, Spark *automatically* distributes the data contained in RDDs across the cluster and parallelizes the operations performed on RDDs
 - Each RDD is split into multiple partitions, which may be computed on different nodes⁸ of the cluster
- RDDs can contain any type of Python, Java, or Scala objects, including user defined classes

Creating RDDs

- **RDDs can only be created:**
 1. **From files:** by loading data from an external storage:
 - e.g., using `SparkContext.textFile()`

```
lines = sc.textFile("/path/to/README.md")
```
 2. **From in-memory objects:** by parallelizing an existing collection in the program
 - ```
lines = sc.parallelize(["pandas", "i like pandas"])
```

**No actual activity takes place, only a definition!!!**

# RDD Creation from Spark Python REPL

---

```
nums = [1, 2, 3, 4, 5, 6, 7, 8]
```

```
Creating RDD from Python List
```

```
nums_rdd = sc.parallelize(nums)
```

```
Creating RDD from text List
```

```
lines_rdd = sc.textFile("data/logs.txt")
```

**No actual activity takes place, only a definition!!!**

# Examples: Create RDDs

---

# Turn a Python collection into an RDD

> `sc.parallelize([1, 2, 3])`

# Load text file from local FS, HDFS, or S3

> `sc.textFile("file.txt")`

> `sc.textFile("directory/*.txt")`

> `sc.textFile("hdfs://namenode:9000/path/file")`

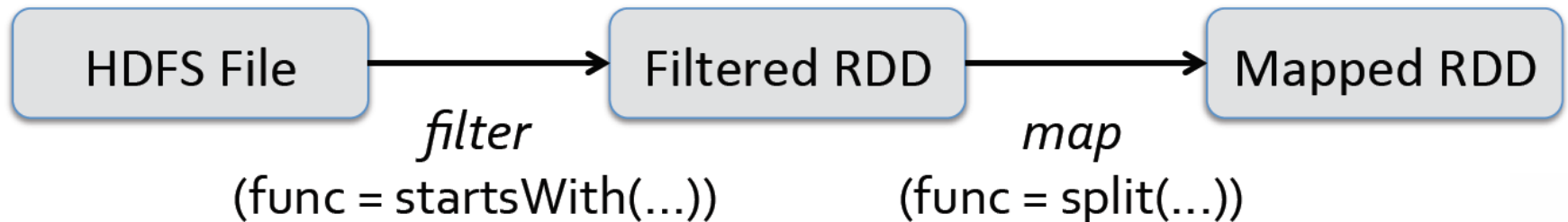
# Use existing Hadoop InputFormat (Java/Scala only)

> `sc.hadoopFile(keyClass, valClass, inputFmt, conf)`

# Fault Recovery

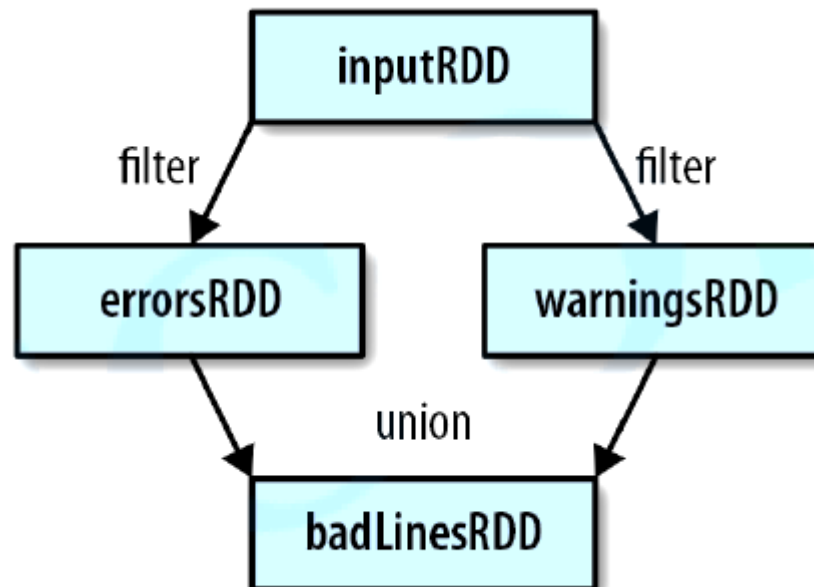
RDDs track *lineage* information that can be used to efficiently re-compute lost data

```
msgs = textFile.filter(lambda s: s.startsWith("ERROR"))
 .map(lambda s: s.split("\t")[2])
```



# Lineage Graph

- Spark keeps track of the set of dependencies between different RDDs, called lineage graph.
- It uses this information to compute each RDD on demand and recover lost data if part of a persistent RDD is lost



# Outline

---

- Spark Program Elements
- SparkContext: sc
- RDD: Resilient Distributed
- **Operations on RDDs: Transformations**
- Operations on RDDs: Actions
- Passing Functions
- Lazy Evaluation and Persistence

# Operations on RDD

- **RDDs offer two types of operations:**
  - transformations
  - actions
- **Transformations** construct a new RDD from a previous one
  - e.g., filtering data that matches a predicate
  - e.g., new RDD holding just the strings that contain word Python

```
>>>pythonLines=lines.filter(lambda line: "Python" in line)
```

- **Actions** compute a result based on an RDD, and either return it to the driver program or save it to an external storage system (e.g. HDFS)
  - e.g. `first()`, which returns the first element in an RDD

```
>>>pythonLines.first()
u'## Interactive Python Shell'
```

# Transformations vs. Actions

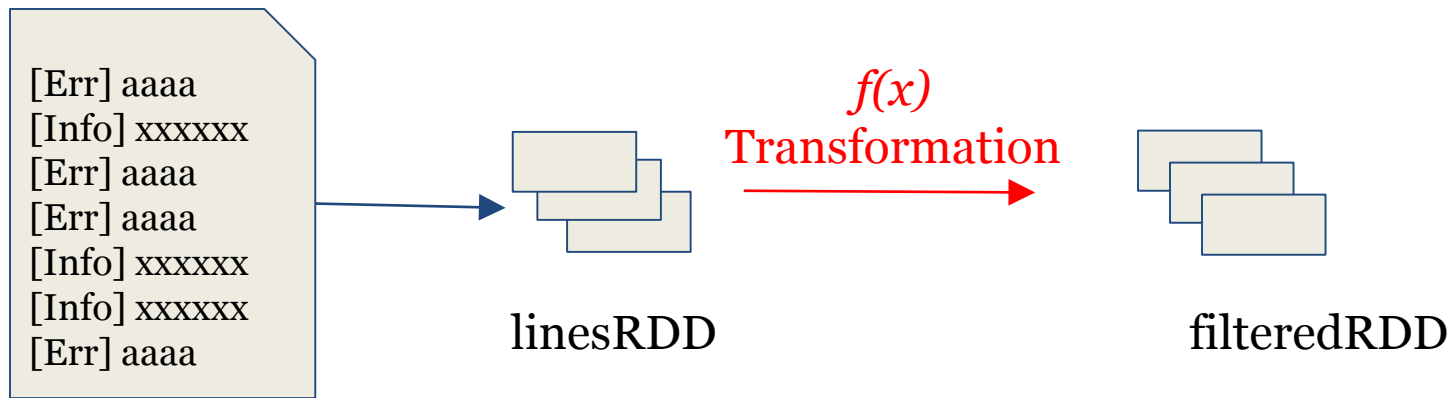
---

- **Spark treats transformations and actions very differently:**
  - Transformations return RDDs
  - Actions return some other data type



# Spark Transformations

**Transformations construct a **new RDD** from a previous one: e.g:  
Filtering all the lines that contain a particular key word.**



Error  
Logs

$f(x)$  can be a filter that filters all the line  
starting with “[Err]”

# Examples: Basic Transformations

```
> nums = sc.parallelize([1, 2, 3])
```

```
Pass each element through a function
```

```
> squares = nums.map(lambda x: x*x) // {1, 4, 9}
```

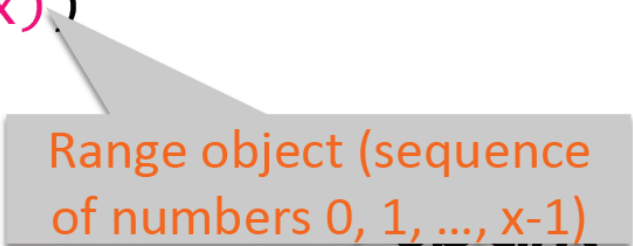
```
Keep elements passing a predicate
```

```
> even = squares.filter(lambda x: x % 2 == 0) // {4}
```

```
Map each element to zero or more others
```

```
> nums.flatMap(lambda x: => range(x))
```

```
> # => {0, 0, 1, 0, 1, 2}
```



Range object (sequence of numbers 0, 1, ..., x-1)

# Transformations: Lazy Evaluation

- Transformed RDDs are computed **lazily**, only when you use them in an action
- Many transformations are **element-wise**
  - they work one element at a time (not all)
- **Example:**
  - Suppose we have a logfile, log.txt, with a number of messages, and we want to select only the error messages. We can use the filter() transformation

```
inputRDD = sc.textFile("log.txt")
errorsRDD = inputRDD.filter(lambda x: "error" in x)
```

**Note:** *filter() does not mutate the existing inputRDD. Instead, it returns a pointer to an entirely new RDD*

**No actual activity takes place, only a definition!!!**

# Transformations: Examples

---

- **map(func)** : Pass each element through **func**
  - `words_rdd = filtered_rdd.map(lambda x: x.split())`
- **distinct()** : Return a new RDD with distinct elements within a source RDD
  - `unique_words_rdd = words_rdd.distinct()`
- **groupByKey()** : When called on an RDD of (K, V) pairs, returns an RDD of (K, Iterable<V>) pairs
- **union(rdd)** : Returns an RDD that is union of two RDDs

# Basic RDD transformations on {1,2,3,3}

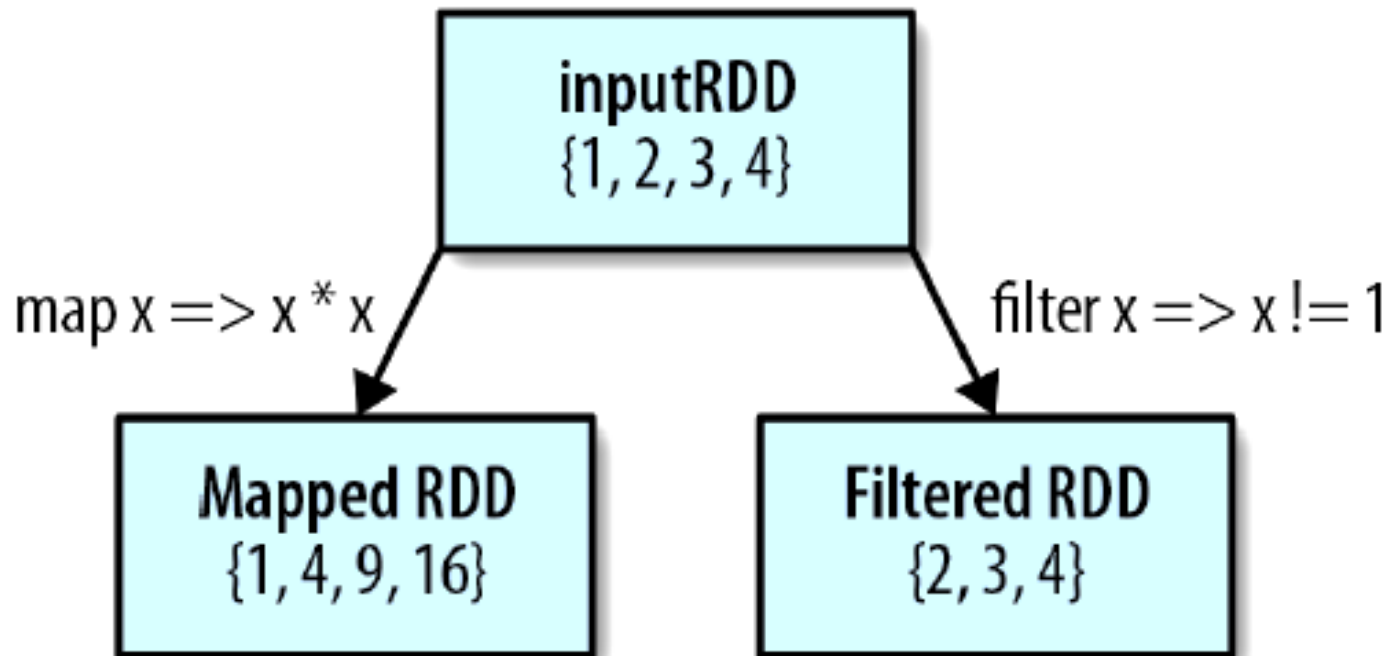
| Function name                                     | Purpose                                                                                                                              | Example                                 | Result           |
|---------------------------------------------------|--------------------------------------------------------------------------------------------------------------------------------------|-----------------------------------------|------------------|
| <b>map()</b>                                      | Apply a function to each element in the RDD and return an RDD of the result                                                          | <code>rdd.map(x =&gt; x+1)</code>       | {2,3,4,4}        |
| <b>flatMap()</b>                                  | Apply a function to each element in the RDD and return an RDD of the contents of the iterators returned. Often used to extract words | <code>rdd.flatMap(x=&gt;x.to(3))</code> | {1,2,3,2,3,3,3}  |
| <b>filter()</b>                                   | Returns an RDD consisting of only elements that pass the condition passed to filter()                                                | <code>rdd.filter(x=&gt;x!=1)</code>     | {2,3,3}          |
| <b>distinct()</b>                                 | Remove duplicates                                                                                                                    | <code>rdd.distinct()</code>             | {1,2,3}          |
| <b>sample</b> (withReplacement, fraction, [seed]) | Sample an RDD, with or without replacement                                                                                           | <code>rdd.sample(false, 0.5)</code>     | Nondeterministic |

# Two-RDD transformations {1,2,3}{3,4,5}

| Function name         | Purpose                                                     | Example                              | Result                  |
|-----------------------|-------------------------------------------------------------|--------------------------------------|-------------------------|
| <b>union()</b>        | Produce an RDD containing elements from either RDD          | <code>rdd.union(other)</code>        | {1,2,3,3,4,5}           |
| <b>intersection()</b> | RDD containing only elements found in both RDDs             | <code>rdd.intersection(other)</code> | {3}                     |
| <b>subtract()</b>     | Remove the contents of one RDD (e.g., remove training data) | <code>rdd.subtract(other)</code>     | {1,2}                   |
| <b>cartesian()</b>    | Cartesian product with the other RDD                        | <code>rdd.cartesian(other)</code>    | {(1,3),(1,4),...,(3,5)} |

# Element-wise transformations

- The two most common transformations: **map()** and **filter()**
  - **map()** : takes in a function and applies it to each element in the RDD with the result of the function being the new value of each element in the resulting RDD.
  - **filter()** : takes in a function and returns an RDD that only has elements that pass the filter() function



# map(): Element-wise mapping

- **can be used to do a number of things**
  - fetch the website associated with each URL
  - square the numbers

```
nums = sc.parallelize([1, 2, 3, 4])
squared = nums.map(lambda x: x * x).collect()
for num in squared:
 print "%i " % (num)
```

- **map()'s return type does not have to be the same as the input type**
  - e.g. if had an RDD String and map() function were to parse the strings and return a Double then
  - input RDD type would be RDD[String] and the resulting RDD type would be RDD[Double]



# Example: map()

---

```
>>> nums = sc.parallelize([1,2,3,4])
>>> squared1 = nums.map(lambda x: [x, x*x]).collect()
>>> squared1
[[1, 1], [2, 4], [3, 9], [4, 16]]

>>> for m in squared1:
... print("%i %i " % (m[0], m[1]))
...
1 1
2 4
3 9
4 16
```

# Handling Errors in map()

---

```
def someparsing(input):
 try:
 #if all is well
 return somevalue
 except ValueError:
 return None

data = data.map(someparsing).filter(lambda x: x!=None).collectAsMap()
```

# flatMap(): Multiple output elements

- **flatMap()** : to produce multiple output elements for each input element
  - called individually for each element in our input RDD
  - Instead of returning a single element, returns an iterator with return values
- “Flatten” it all into a single collection
- Rather than producing an RDD of iterators, we get back an RDD that consists of the elements from all off the iterators.
- Example: flatMap() splitting up an input string into words

```
lines = sc.parallelize(["hello world", "hi"])
words = lines.flatMap(lambda line: line.split(" "))
words.first() # returns "hello"
```

## Example: flatMap()

---

```
>>> lines = sc.parallelize(["hello world", "again
hi", "how are you"])
```

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

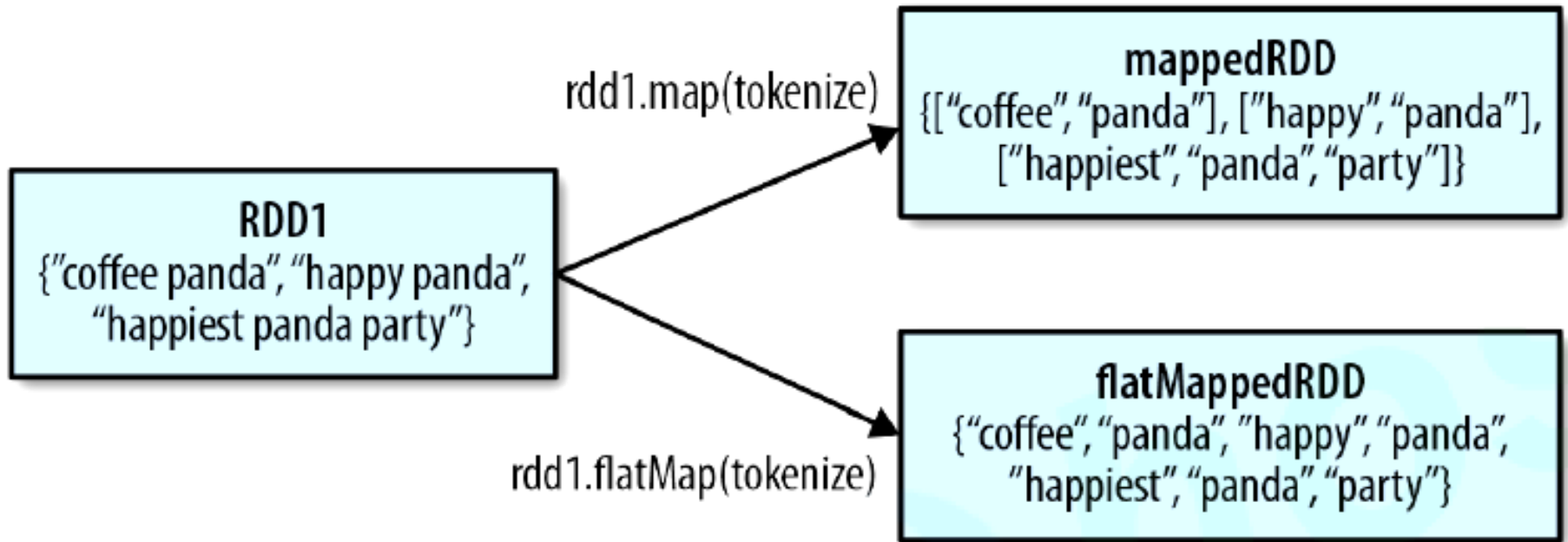
```
>>> words.take(5)
```

```
['hello', 'world', 'again', 'hi', 'how']
```

# flatMap() vs. map()

- You can think of flatMap() as “flattening” the iterators returned to it, that instead of ending up with an RDD of lists we have an RDD of the elements in those lists

`tokenize("coffee panda") = List("coffee", "panda")`



# Pseudo Set Operations

- All of these operations require that the RDDs being operated on are of *the same* type

**RDD1**  
{coffee, coffee, panda,  
monkey, tea}

**RDD2**  
{coffee, money, kitty}

**RDD1.distinct()**  
{coffee, panda,  
monkey, tea}

**RDD1.union(RDD2)**  
{coffee, coffee, coffee,  
panda, monkey,  
monkey, tea, kitty}

**RDD1.intersection(RDD2)**  
{coffee, monkey}

**RDD1.subtract(RDD2)**  
{panda, tea}

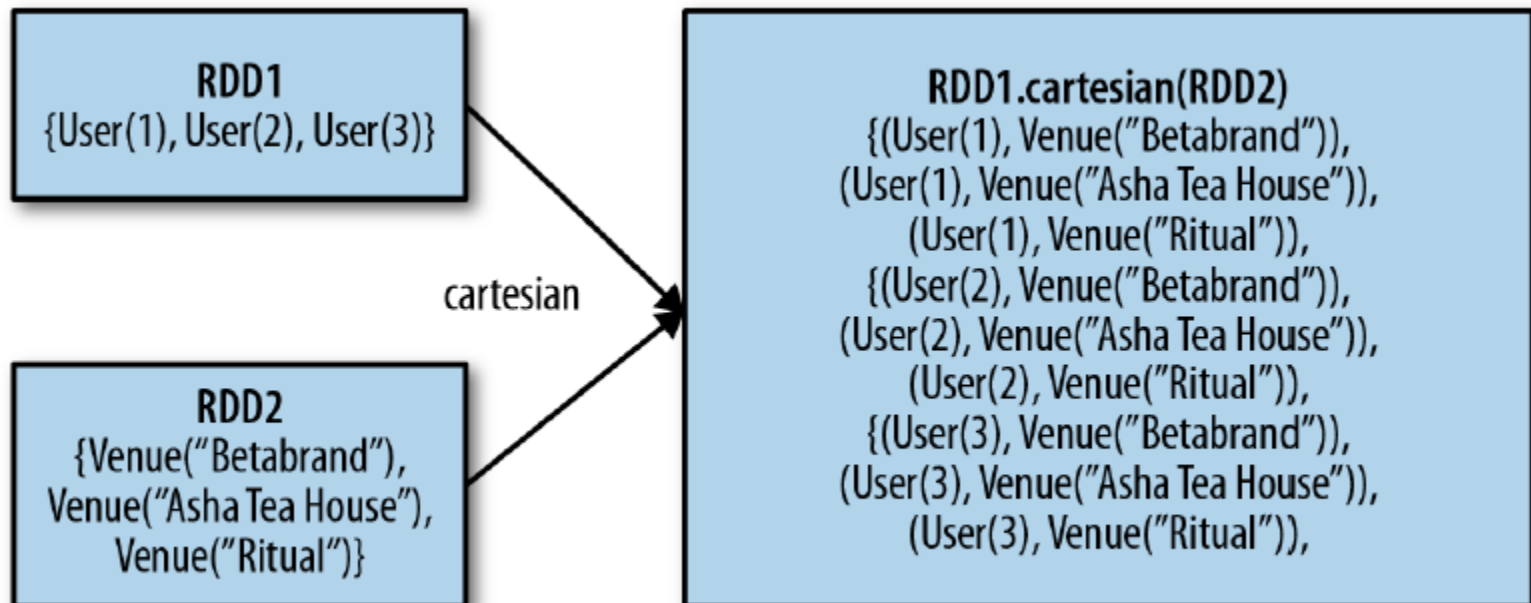
# Pseudo Set Operations

---

- **distinct()** is expensive as it requires shuffling all the data over the network to ensure that we receive only one copy of each element
- **intersection(other)** method
  - returns only elements in both RDDs
  - removes all duplicates (including duplicates from a single RDD) while running
  - the performance of intersection() is much worse since it requires a shuffle over the network to identify common elements
- **subtract(other)** function
  - takes in another RDD and returns an RDD that has only values present in the first RDD and not in the second RDD.
  - subtract() like intersection() performs a shuffle

# Cartesian product

- **cartesian(other)** transformation returns all possible pairs of  $(a,b)$  where  $a$  is in the source RDD and  $b$  is in the other RDD
- The Cartesian product can be useful when we wish to consider the similarity between all possible pairs
  - e.g. computing every user's expected interest in each other
- Cartesian product of an RDD with itself is useful for tasks like *user similarity*
- Cartesian product is **VERY EXPENSIVE** for large RDDs





# Example: union()

---

- **print out the number of lines that contained either error or warning**

```
errorsRDD = inputRDD.filter(lambda x: "error" in x)
warningsRDD = inputRDD.filter(lambda x: "warning" in x)
badLinesRDD = errorsRDD.union(warningsRDD)
```

- **union() operates on two RDDs instead of one.**  
**Transformations can operate on any number of input RDDs**

# Examples: subtract() and intersection()

```
a = sc.parallelize([1,1,2,2,3,4])
b = sc.parallelize([2,2,3,3,3,5])
c = a.subtract(b)
print("a.subtract(b) \n")
c1 = c.collect()
for x in c1:
 print("%i " % (x))
d = a.intersection(b)
d1 = d.collect()
print("a.intersection(b) \n")
for x in d1:
 print("%i " % (x))
```

a.subtract(b)

4

1

1

a.intersection(b)

2

3

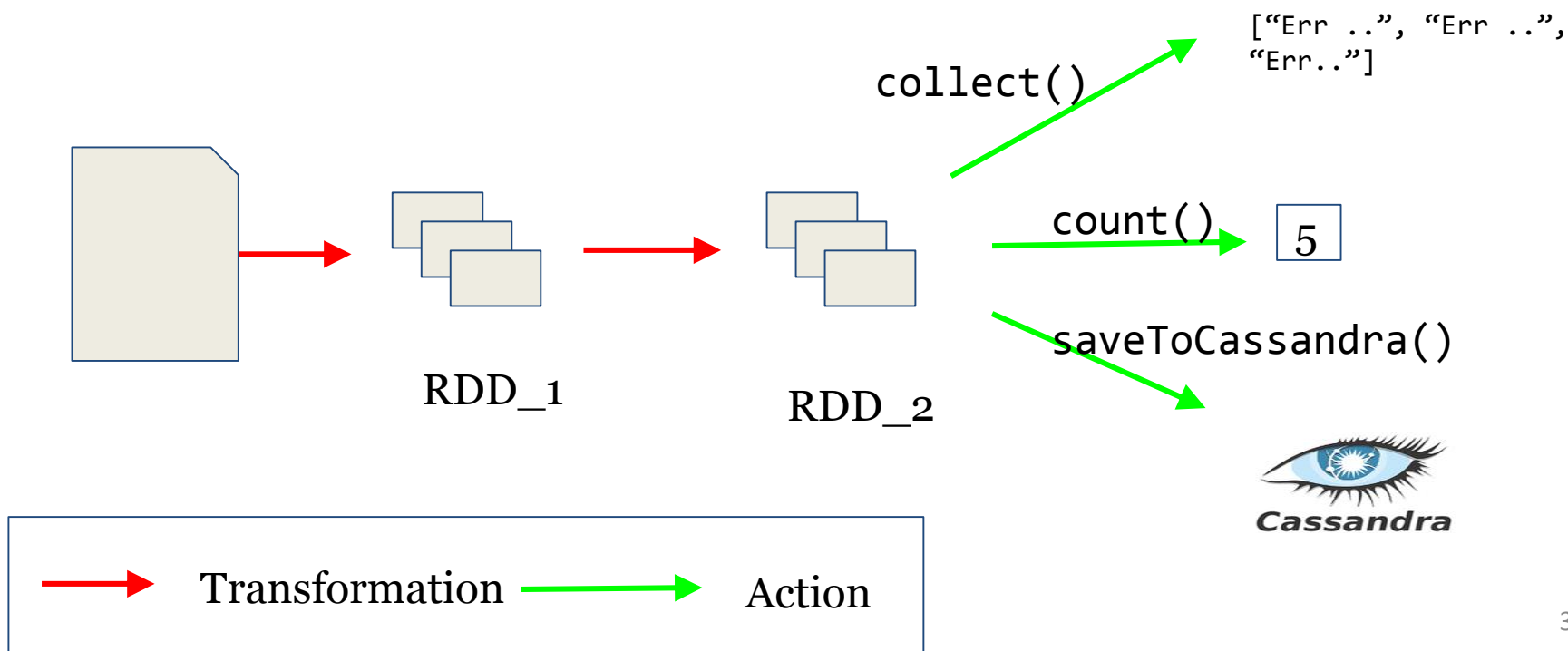
# Outline

---

- Spark Program Elements
- SparkContext: sc
- RDD: Resilient Distributed
- Operations on RDDs: Transformations
- **Operations on RDDs: Actions**
- Passing Functions
- Lazy Evaluation and Persistence

# Spark Actions

- **Actions** are the operations performed on RDDs that return a final value to the driver program or save it to an external storage system (e.g., HDFS)
- Actions *force the evaluation of the transformations* required for the RDD they were called on, since they need to actually *produce output*



# Examples: Basic Actions

---

```
> nums = sc.parallelize([1, 2, 3])

Retrieve RDD contents as a local collection
> nums.collect() # => [1, 2, 3]

Return first K elements
> nums.take(2) # => [1, 2]

Count number of elements
> nums.count() # => 3

Merge elements with an associative function
> nums.reduce(lambda x, y: x + y) # => 6

Write elements to a text file
> nums.saveAsTextFile("hdfs://file.txt")
```

# Basic actions on an RDD containing {1,2,3,3}

| Function name               | Purpose                                         | Example                         | Result                         |
|-----------------------------|-------------------------------------------------|---------------------------------|--------------------------------|
| <code>collect()</code>      | Return all elements from the RDD.               | <code>rdd.collect()</code>      | {1, 2, 3, 3}                   |
| <code>count()</code>        | Number of elements in the RDD.                  | <code>rdd.count()</code>        | 4                              |
| <code>countByValue()</code> | Number of times each element occurs in the RDD. | <code>rdd.countByValue()</code> | {(1, 1),<br>(2, 1),<br>(3, 2)} |

# Basic actions on an RDD containing {1,2,3,3}

| Function name                                         | Purpose                                                           | Example                                     | Result           |
|-------------------------------------------------------|-------------------------------------------------------------------|---------------------------------------------|------------------|
| <code>take(num)</code>                                | Return num elements from the RDD.                                 | <code>rdd.take(2)</code>                    | {1, 2}           |
| <code>top(num)</code>                                 | Return the top num elements the RDD.                              | <code>rdd.top(2)</code>                     | {3, 3}           |
| <code>takeOrdered(num)(ordering)</code>               | Return num elements based on provided ordering.                   | <code>rdd.takeOrdered(2)(myOrdering)</code> | {3, 3}           |
| <code>takeSample(withReplacement, num, [seed])</code> | Return num elements at random.                                    | <code>rdd.takeSample(false, 1)</code>       | Nondeterministic |
| <code>reduce(func)</code>                             | Combine the elements of the RDD together in parallel (e.g., sum). | <code>rdd.reduce((x, y) =&gt; x + y)</code> | 9                |

# Basic actions on an RDD containing {1,2,3,3}

| Function name                                    | Purpose                                                               | Example                                                                                                                         | Result  |
|--------------------------------------------------|-----------------------------------------------------------------------|---------------------------------------------------------------------------------------------------------------------------------|---------|
| <code>fold(zero)(func)</code>                    | Same as <code>reduce()</code> but with the provided zero value.       | <code>rdd.fold(0)((x, y) =&gt; x + y)</code>                                                                                    | 9       |
| <code>aggregate(zeroValue)(seqOp, combOp)</code> | Similar to <code>reduce()</code> but used to return a different type. | <code>rdd.aggregate((0, 0))<br/>((x, y) =&gt;<br/>(x._1 + y, x._2 + 1),<br/>(x, y) =&gt;<br/>(x._1 + y._1, x._2 + y._2))</code> | (9, 4)  |
| <code>foreach(func)</code>                       | Apply the provided function to each element of the RDD.               | <code>rdd.foreach(func)</code>                                                                                                  | Nothing |



# Spark Actions: `count()` and `take()`

- **Example:**

```
print "Input had " + badLinesRDD.count() + " concerning lines"
print "Here are 10 examples:"
for line in badLinesRDD.take(10):
 print line
```

- print out information about the `badLinesRDD`
- **`count()`** : returns the count as a number
- **`take()`** : collects a number of elements from the RDD
- We then iterate over them locally to print out information at the driver

# Spark Actions: collect()

---

```
list_of_errors = sc.textFile("data/logs.txt").filter(lambda
 line: "[Err]" in line).collect()
```

- **collect()** is the simplest and most common operation that returns the entire RDD's contents to the driver program
  - commonly used in unit tests where the entire contents of the RDD are expected to fit in memory, as that makes it easy to compare the value of our RDD with our expected result
  - suffers from the restriction that all of the data must fit on a single machine as it all needs to be copied to the driver
  - does not return the elements in the order you might expect
  - In case of Python, it returns List of elements

# Actions: top(), take(n)

- **top()** will use the default ordering on the data, but we can supply our own comparison function to extract the top elements
- **take(n)** returns *n* elements from the RDD and attempts to minimize the number of partitions it accesses, so it may represent a biased collection
  - does not return the elements in the order you might expect

```
>>> b = sc.parallelize([1,3,2,1,4,1,4,2])
>>> a = b.take(2)
>>> a
[1, 3]
>>> a = b.take(4)
>>> a
[1, 3, 2, 1]
>>>
>>> c = b.top(2)
>>> c
[4, 4]
>>> d = b.top(3)
>>> d
[4, 4, 3]
```

# Actions: foreach()

- **foreach()** action lets us perform computations on each element in the RDD without bringing it back locally
  - e.g. posting JSON to a webserver on inserting records into a database

```
>>> a.collect()
[1, 3, 2, 4, 5]
>>> def fun1(x):
... print(str(x*x))
... return True
...
>>> b = a.foreach(fun1)
9
16
4
1
25
```

# Actions: takeSample()

- **takeSample (withReplacement, num, seed)** function allows us to take a sample of the data either with or without replacement

```
>>> a = b.takeSample(False, 3)
```

```
>>> a
```

```
[2, 1, 2]
```

```
>>> a = b.takeSample(False, 3)
```

```
>>> a
```

```
[2, 4, 1]
```

```
>>> a = b.takeSample(False, 3, 5)
```

5 is seed

```
>>> a
```

```
[3, 1, 1]
```

```
>>> a = b.takeSample(False, 3, 5)
```

Same seed

```
>>> a
```

```
[3, 1, 1]
```

# Actions: reduce()

- **reduce()** : the most common action on basic RDD
  - takes a function that operates on **two** elements of the type in your RDD and returns a **new** element of the same type
- **Example, +, which we can use to sum our RDD**
- **With reduce() we can**
  - sum the elements of our RDD
  - count the number of elements,
  - perform other types of aggregations

```
sum = rdd.reduce(lambda x, y: x + y)
```

```
>>> a = sc.parallelize([1,3,2,4,5])
```

```
>>> sum = a.reduce(lambda x,y: x + y)
```

```
>>> sum
```

```
15
```

```
>>> sum = a.reduce(lambda x,y: x * y)
```

```
>>> sum
```

```
120
```

# Actions: fold()

- **fold()** takes a function with the same signature as needed for `reduce()`, but in addition it takes a “zero value” to be used for the initial call on each partition
  - The zero value should be the identity element for the operation
- **Applying identity element multiple times with the function should not change the value**
  - e.g. 0 for +, 1 for \*, or an empty list for concatenation
- **fold() require the return type of the result to be the same type as that of the elements in the RDD we are operating over**

```
>>> a = sc.parallelize([1,3,2,4,5])
>>> sum = a.fold(1, lambda x,y: x * y)
>>> sum
120
>>> a.fold(1, lambda x,y: x + y)
22
>>> a.fold(0, lambda x,y: x + y)
15
>>> a.fold(-1, lambda x,y: x + y)
8
>>> a = sc.parallelize([1,3,2,4,5],3)
>>> a.fold(1, lambda x,y: x + y)
19
>>> a = sc.parallelize([1,3,2,4,5])
>>> a.fold(1, lambda x,y: x + y)
22
>>> a = sc.parallelize([1,3,2,4,5], 6)
>>> a.fold(1, lambda x,y: x + y)
22
```

# Actions: aggregate()

- **aggregate()** function does not have to return the same type as the RDD we are working on
  - supply an initial zero value of the type we want to return
  - supply a function to combine the elements from our RDD with the accumulator
  - supply a second function to merge two accumulators, given that each node accumulates its own results locally
- **Example: we can use aggregate() to compute the average of an RDD, avoiding a map() before the fold()**

```
sumCount = nums.aggregate((0, 0),
 (lambda acc, value: (acc[0] + value, acc[1] + 1),
 (lambda acc1, acc2: (acc1[0] + acc2[0], acc1[1] + acc2[1]))))
return sumCount[0] / float(sumCount[1])
```



# Example: aggregate()

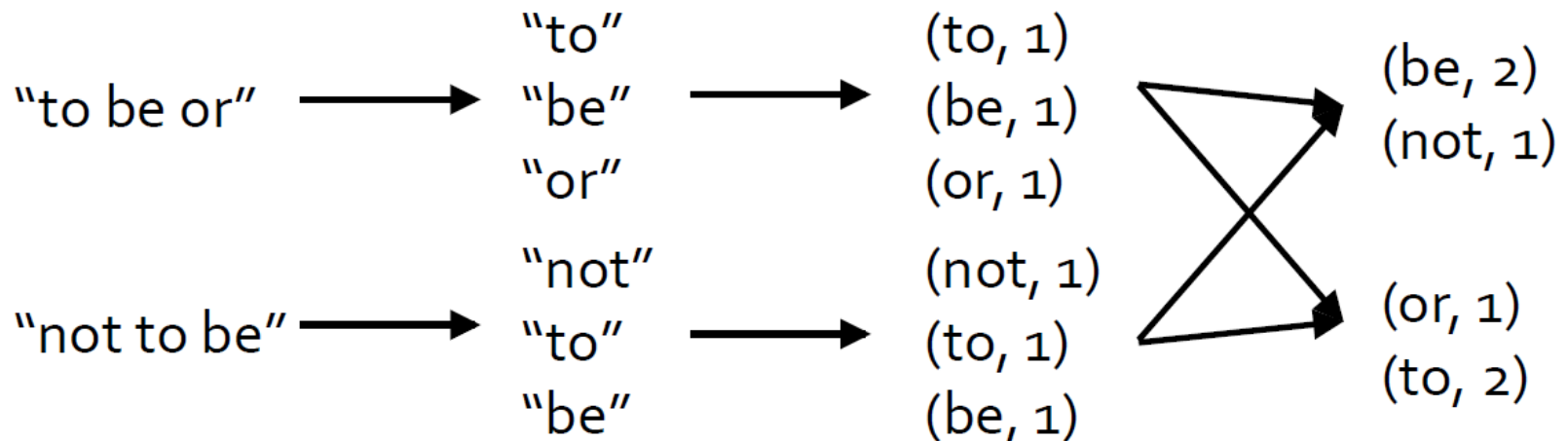
---

```
>>> a = sc.parallelize([1,3,2,4,5])
>>> sum = a.aggregate((0,0), Zero value of 2 entries accumulator
... (lambda acc, value: (acc[0] + value, acc[1] + 1)), Per partition acc
... (lambda acc1, acc2: (acc1[0] + acc2[0], acc1[1] + acc2[1])))
>>> sum
(15, 5)
>>> sum[0]/float(sum[1])
3.0
```

Combine  
accumulators of  
2 partitions

# Classic Word Count Example in Spark

```
> lines = sc.textFile("hamlet.txt")
> counts = lines.flatMap(lambda line: line.split(" "))
 .map(lambda word => (word, 1))
 .reduceByKey(lambda x, y: x + y)
> counts.saveAsTextFile("word_count.txt")
```



# Process Logs

---

```
text_file_rdd = sc.textFile("data/logs.txt")
errors = text_file_rdd.filter(lambda line: "Err" in line)
Count all the errors
errors.count()
Count errors mentioning MySQL
errors.filter(lambda line: "MySQL" in line).count()
Fetch the MySQL errors as an array of strings
errors.filter(lambda line: "MySQL" in line).collect()
```

# Example: countByValue()

---

```
>>> b = sc.parallelize([1,3,2,1,4,1,4,2])
>>> cbv = b.countByValue()
>>> cbv
defaultdict(<type 'int'>, {1: 3, 2: 2, 3: 1, 4: 2})
```

Dictionary

# Outline

---

- Spark Program Elements
- SparkContext: sc
- RDD: Resilient Distributed
- Operations on RDDs: Transformations
- Operations on RDDs: Actions
- **Passing Functions**
- Lazy Evaluation and Persistence

# Passing Functions to Spark

---

- **Three options in Python:**

- **pass in lambda expressions**

```
word = rdd.filter(lambda s: "error" in s)
```

```
def containsError(s):
```

```
 return "error" in s
```

```
word = rdd.filter(containsError)
```

- **pass in top level functions**

- **pass in locally defined functions**

- **Issue: inadvertently serializing the object containing the function**

# Don't do this

- When you pass a function that is the member of an object, or contains references to fields in an object (`self.field`), Spark sends the *entire* object to worker nodes, which can be much larger than the bit of information you need
- Fails if your class contains objects that Python can't figure out how to pickle

```
class SearchFunctions(object):
 def __init__(self, query):
 self.query = query
 def isMatch(self, s):
 return self.query in s
 def getMatchesFunctionReference(self, rdd):
 # Problem: references all of "self" in "self.isMatch"
 return rdd.filter(self.isMatch)
 def getMatchesMemberReference(self, rdd):
 # Problem: references all of "self" in "self.query"
 return rdd.filter(lambda x: self.query in x)
```

# Do this instead

---

Extract the fields you need from you object into a local variable and pass it in

```
class WordFunctions(object):
 ...
 def getMatchesNoReference(self, rdd):
 # Safe: extract only the field we need into a local variable
 query = self.query
 return rdd.filter(lambda x: query in x)
```



# Outline

---

- Spark Program Elements
- SparkContext: sc
- RDD: Resilient Distributed
- Operations on RDDs: Transformations
- Operations on RDDs: Actions
- Passing Functions
- **Lazy Evaluation and Persistence**

# Lazy Evaluation in Spark

- **Lazy evaluation** means that a transformation on an RDD (e.g. `map()`) is not immediately performed.
- You can define new RDDs any time
- Spark computes RDDs only in **lazy** fashion – the first time they are used in action
- Transformations on RDDs are lazily evaluated, meaning that Spark will not begin to execute until it sees an action.
  - Spark sees the whole chain of transformations, it can compute just the data needed for its result
  - e.g. for the `first()` action, Spark scans the file only until it finds the first matching line; it does not read the whole file

# Lazy Evaluation: Internals

---

- Spark internally records metadata to indicate that this operation has been requested.
- Rather than "thinking" of an RDD as containing specific data, it "thinks" of each RDD as consisting of *instructions* on how to compute the data to be built up through transformations
  - To force Spark to execute transformations at any time, run an action, such as `count()`. This is an easy way to test out part of a program
- Spark uses lazy evaluation to reduce the number of passes it has to take over the data by grouping operations together

# Persistence (Caching) with `RDD.persist()`

---

- **Spark RDDs are lazily evaluated**
- **By default, Spark's RDDs and their dependencies are re-computed each time you run an action on them.**
- **Sometimes we may wish to use the same RDD multiple times**
  - Natively, Spark will re-compute the RDD and all of its dependencies each time we call an action on the RDD
  - Used by iterative algorithms, accessing data many times
  - When doing a count and writing out the same RDD
- **To avoid computing an RDD multiple times, we can ask Spark to persist the data: the nodes that compute the RDD store their partitions.**
  - If a node that has data persisted on it fails, Spark will re-compute the lost partitions of the data when needed
  - We can also replicate our data on multiple nodes if we want to be able to handle node failure without shutdown

# Persisting RDDs: Mechanics

---

- After computing it the first time , Spark will store the RDD contents in memory (partitioned across the machines in cluster) and reuse them in future actions
- The ability to re-compute an RDD is actually why RDDs are called “*resilient*”
- When a machine holding RDD data fails, Spark uses this ability to re-compute the missing partitions, transparent to the user
- Persisting RDDs on disk instead of memory is also possible.
- No persisting by default: if you will not use the RDD, there's not reason to waste storage space when Spark could instead stream through the data once and just compute the result

# Persisting RDDs in Memory

- In practice, use `persist()` to load a subset of data into memory and query it repeatedly
  - e.g. if we want to compute multiple results about the README lines that contain Python, we could write the script

```
>>> pythonLines.persist
```

```
>>> pythonLines.count()
```

```
2
```

```
>>> pythonLines.first()
```

```
u'## Interactive Python Shell'
```

# Persistence Levels

- Persistence levels from `org.apache.spark.storage.StorageLevel` and `pyspark.StorageLevel`

| Level               | Space used | CPU time | In memory | On disk | Comments                                                                                               |
|---------------------|------------|----------|-----------|---------|--------------------------------------------------------------------------------------------------------|
| MEMORY_ONLY         | High       | Low      | Y         | N       |                                                                                                        |
| MEMORY_ONLY_SER     | Low        | High     | Y         | N       |                                                                                                        |
| MEMORY_AND_DISK     | High       | Medium   | Some      | Some    | Spills to disk if there is too much data to fit in memory.                                             |
| MEMORY_AND_DISK_SER | Low        | High     | Some      | Some    | Spills to disk if there is too much data to fit in memory. Stores serialized representation in memory. |
| DISK_ONLY           | Low        | High     | N         | Y       |                                                                                                        |

# **persist()**

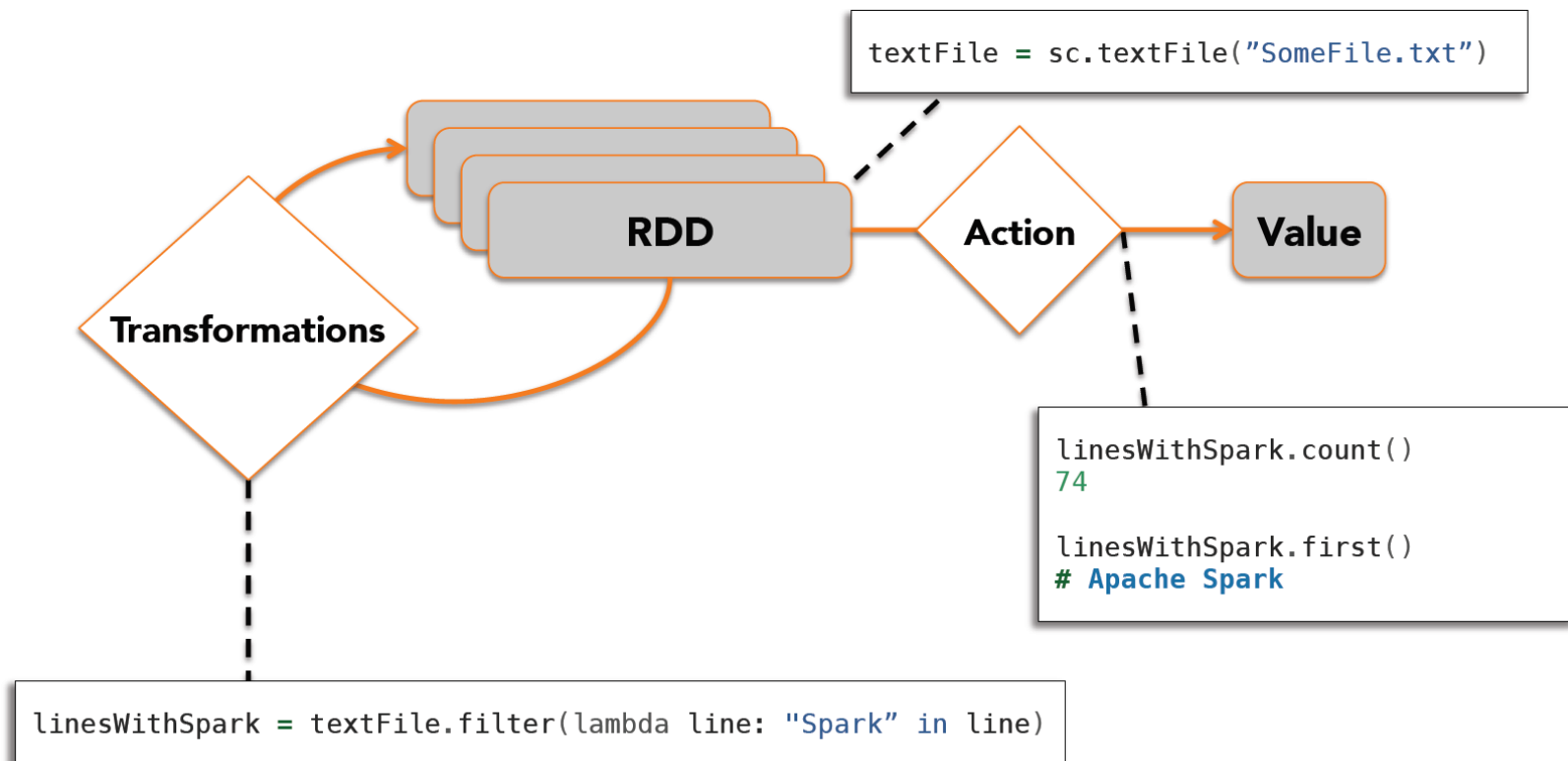
---

- **The persist() call on its own does not force evaluation**
- **If you attempt to cache too much data to fit in memory, Spark will automatically evict old partitions using a Least Recently Used (LRU) cache policy**
- **For the memory-only storage levels, it will re-compute these partitions the next time they are accessed**
- **Caching unnecessary data can lead to eviction of useful data and more re-computation time**
- **unpersist() lets you manually remove RDDs from the cache**



# Summary: Spark Core

1. Create SparkContext object (**sc**)
2. Create some input **RDDs** from external data
3. Transform RDDs to define new RDDs using **transformations**
4. Ask Spark to **persist** any intermediate RDDs for reuse
5. Launch **actions** such as count() to kick off a parallel computation, which is then optimized and executed by Spark



# Summary: Complete Application

---

```
import sys
from pyspark import SparkContext

if __name__ == "__main__":
 sc = SparkContext("local", "wordcount", sys.argv[0], None)
 lines = sc.textFile(sys.argv[1])

 counts = lines.flatMap(lambda s: s.split(" ")) \
 .map(lambda word: (word, 1)) \
 .reduceByKey(lambda x, y: x + y)

 counts.saveAsTextFile(sys.argv[2])
```

# Acknowledgements

---

- Nirmesh Khandelwal, NCSU
- Anatoli Melechko, NCSU
- Learning Spark by Holden Karau, Andy Konwinski, Patrick Wendell, Matei Zaharia  
(<http://shop.oreilly.com/product/0636920028512.do>)
- Spark Training from Spark Summit-2015 (<https://spark-summit.org/2015/training/#intro>)
- Official Spark documentation  
(<http://spark.apache.org/docs/latest/>)
- Pat McDonough – Databricks: "Using Apache Spark"