Spark

Spark: Cluster computing with working sets
M. Zaharia, M. Chowdhury, M. J. Franklin, S. Shenker, I. Stoica

Main goals

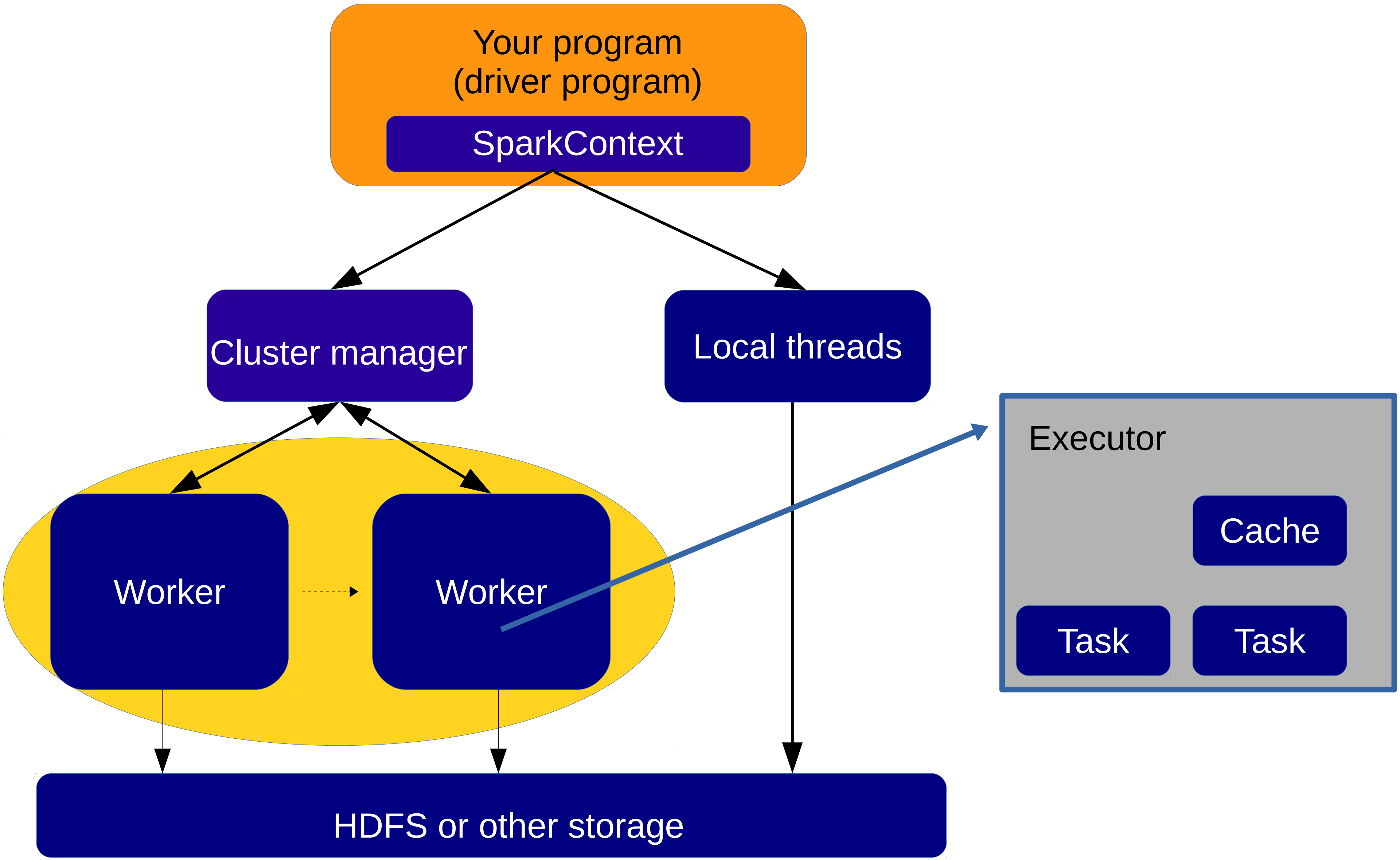
- Locality aware scheduling
- Fault tolerance
- Load balancing
- Non-acyclic data flows
  - Iterative jobs
  - Interactive analytics

MapReduce

Programming model

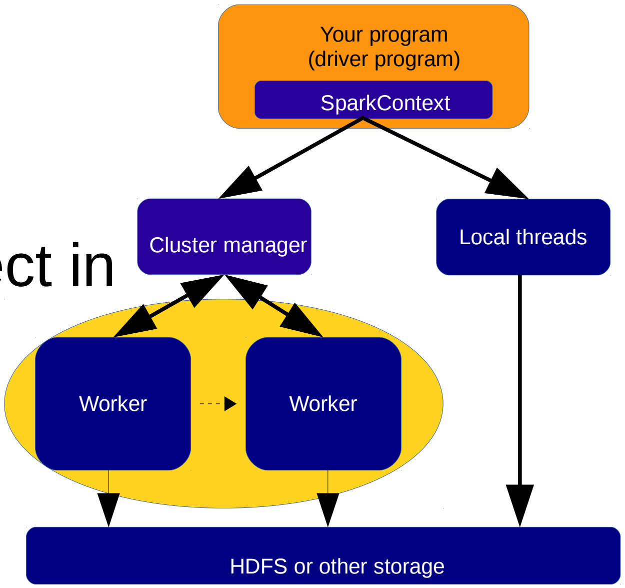
- You write a *driver program*
- Driver program implements high-level control flow
- Can launch various operations in parallel
- Key abstractions
  - Resilient Distributed Datasets
  - Parallel operations on RDD's

Programming model



Programming model

- Spark application
  - Independent set of processes on a cluster
  - Coordinated by a *SparkContext* object in the *Driver*
  - *Executors* at worker nodes
    - Execute computational *tasks* and store applications data
  - *SparkContext* can connect to different types of *cluster managers*
    - Spark's standalone, Yarn (Hadoop) or Mesos



Programming model

- Key abstraction: RDD
  - Resilient, Distributed (across workers), Dataset
- Parallel operations possible on RDD's. Basic operation types:
  - *reduce*
  - *collect*
  - *foreach*

Using Spark with Python - pySpark

Python Spark

- Spark adopts programming interfaces in several languages
  - Scala, Java, Python, R
- We consider the Python programming interface
  - PySpark
- Now an Apache project
  - http://spark.apache.org

What is a Spark program?

- A sequence of operations performed via an interactive shell (e.g., pySpark)
- An application (e.g., a Python module) submitted to the cluster
- *master* parameter defines cluster's type and size

Master Parameter	Description
`local`	run Spark locally with one worker thread (no parallelism)
`local[K]`	run Spark locally with K worker threads (ideally set to number of cores)
`spark://HOST:PORT`	connect to a Spark standalone cluster; PORT depends on config (7077 by default)
`mesos://HOST:PORT`	connect to a Mesos cluster; PORT depends on config (5050 by default)

Installing Spark

- Installing a stand-alone binary
  - 3-steps installation guide
  - Useful for debugging
  - Core-level parallelism
- Cluster mode
  - Check here for an overview

Using Spark

Using Spark (Python)

- Either the interactive (Python) shell …

```
becchett@becchett-Inspiron:~/DOCS/DIS/Didattica/BigData/Spark/spark-1.4.1-bin-ha
doop2.6$ ./bin/pyspark --master local[*]
```

- Or submitting an application

```
becchett@becchett-Inspiron:~/DOCS/DIS/Didattica/BigData/Spark/spark-1.4.1-bin-ha
doop2.6$ ./bin/spark-submit --master local[*] ../MyExamples/wordcount_3.py
```

Creating RDD's - parallelization

```
      ____              __
     / __/__  ___ _____/ /__
    _\ \/ _ \/ _ `/ __/  '_/
   /__ / .__/\_,_/_/ /_/\_\   version 1.4.1
      /_/

Using Python version 2.7.6 (default, Jun 22 2015 18:00:18)
SparkContext available as sc, HiveContext available as sqlContext.
>>> rdd = sc.parallelize([1, 2, 3, 4])
>>> 
```

We had a Python's list at the *driver* node
Now we have a a distributed dataset corresponding to it

```
>>> rdd
ParallelCollectionRDD[0] at parallelize at PythonRDD.scala:396
>>> 
```

Creating RDD's from files

Number of partitions

```
>>> rdd = sc.textFile("./data/MyData/wordcount_data/*.txt", 4, use_unicode=False)
>>> lines = rdd.collect()
>>> print lines
['Pluto Paperone Pluto Tom Pippo Clarabella Topolino Pippo Pippo Titty', 'Titty Tom P
ippo Paperone Titty Pluto Pippo Clarabella', 'Paperone Tom Jerry Pluto Paperino', 'Cl
```

- Example: build an RDD from a collection of text files
  - A *single* RDD corresponding to the original file(s)
  - Distributed across the cluster (4 partitions in this case)
- Collecting the data will bring the text lines corresponding to *all* original files back to the driver as a single Python collection
  - A list of string lines in this case
  - Careful with *collect()* !

Other ways to create RDD's

- *Transform* an existing RDD
  - Further in this lecture
- Create a persisting copy of an existing RDD
  - *cache* or *save* actions
- Create RDD's from other file formats

Transformations

- Transform an RDD into another
- Examples
  - **map**(func)
  - filter(func)
  - **flatMap**(func) → similar to map in MapReduce
  - groupByKey([numTasks])
  - reduceByKey(func, [numTasks])
  - Many more …
- Implemented lazily
  - Will only be executed upon invocation of an *action*

Actions

- Really trigger the computation
- Launch an action implies
  - Return a value to the driver program or …
  - Write data to external storage

Actions - examples

Action	Description
`reduce(func)`	Aggregate the elements of the dataset using a function func (which takes two arguments and returns one). The function should be commutative and associative so that it can be computed correctly in parallel.
`collect()`	Return all the elements of the dataset as an array at the driver program. This is usually useful after a filter or other operation that returns a sufficiently small subset of the data.
`count()`	Return the number of elements in the dataset.

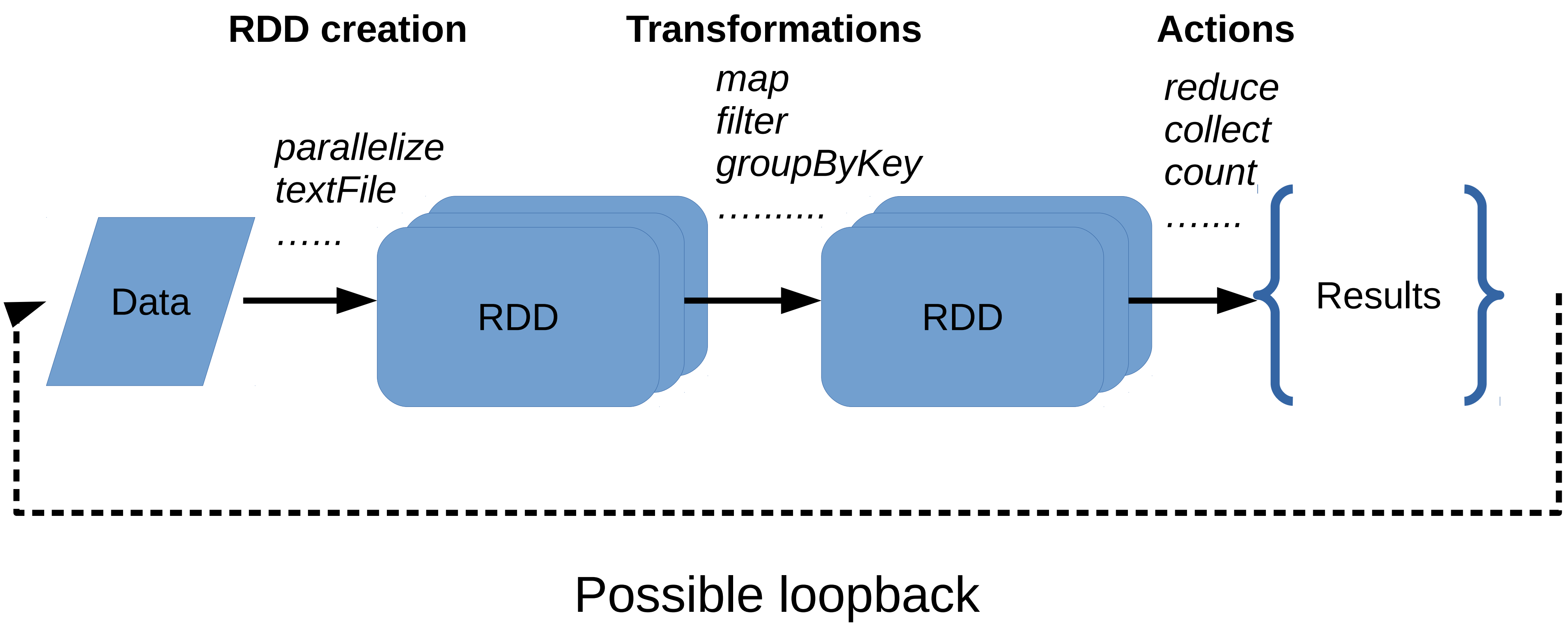
Snapshot from Benjamin Bengfort's slideshar presentation
“Fast Data Analytics with Spark and Python”

Note that *reduce* is an action!

Difference between map and flatMap

```
>>> lines = sc.textFile("./data/MyData/wordcount_data/file5.txt")
>>> res1 = lines.map(lambda x: x.split()).collect()
>>> res1
[[u'Pluto', u'Paperone', u'Pluto', u'Tom', u'Pippo', u'Clarabella', u'Topolino',
 u'Pippo', u'Pippo', u'Titty'], [u'Titty', u'Tom', u'Pippo', u'Paperone', u'Titt
y', u'Pluto', u'Pippo', u'Clarabella'], [u'Paperone', u'Tom', u'Jerry', u'Pluto'
, u'Paperino'], [u'Clarabella', u'Paperino', u'Paperone', u'Pippo', u'Minnie', u
'Jerry', u'Paperone'], [u'Paperino', u'Jerry']]
>>> res2 = lines.flatMap(lambda x: x.split()).collect()
>>> res2
[u'Pluto', u'Paperone', u'Pluto', u'Tom', u'Pippo', u'Clarabella', u'Topolino',
u'Pippo', u'Pippo', u'Titty', u'Titty', u'Tom', u'Pippo', u'Paperone', u'Titty',
 u'Pluto', u'Pippo', u'Clarabella', u'Paperone', u'Tom', u'Jerry', u'Pluto', u'P
aperino', u'Clarabella', u'Paperino', u'Paperone', u'Pippo', u'Minnie', u'Jerry'
, u'Paperone', u'Paperino', u'Jerry']
>>>
```

Typical life-cycle of a Spark application



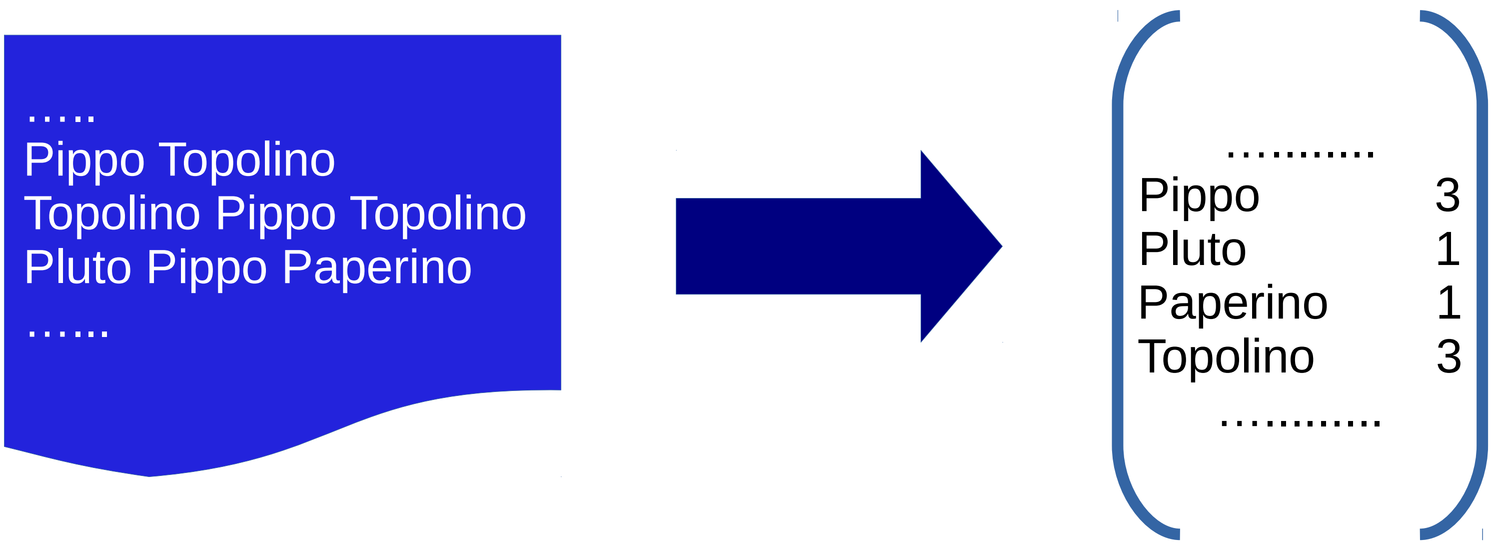
A first example

```
def main():
        conf = SparkConf().setAppName("Lines sum").setMaster("local")
        sc = SparkContext(conf=conf)
        lines = sc.textFile("./data/MyData/wordcount_data/*.txt")

        print "\nUSING LAMBDA"
        start = time.time()

        lineLengths = lines.map(lambda s: len(s.strip()))
        totalLength = lineLengths.reduce(lambda a, b: a + b)
        print "\nSUM LINES LEN: " + str(totalLength)

        stop = time.time()
        print "\nTIME: " + str(stop-start)

        ###########################################################
```

Apply this function to every line of the RDD and return the corresponding result
A list of strings for every line in this case

Another example walk-through

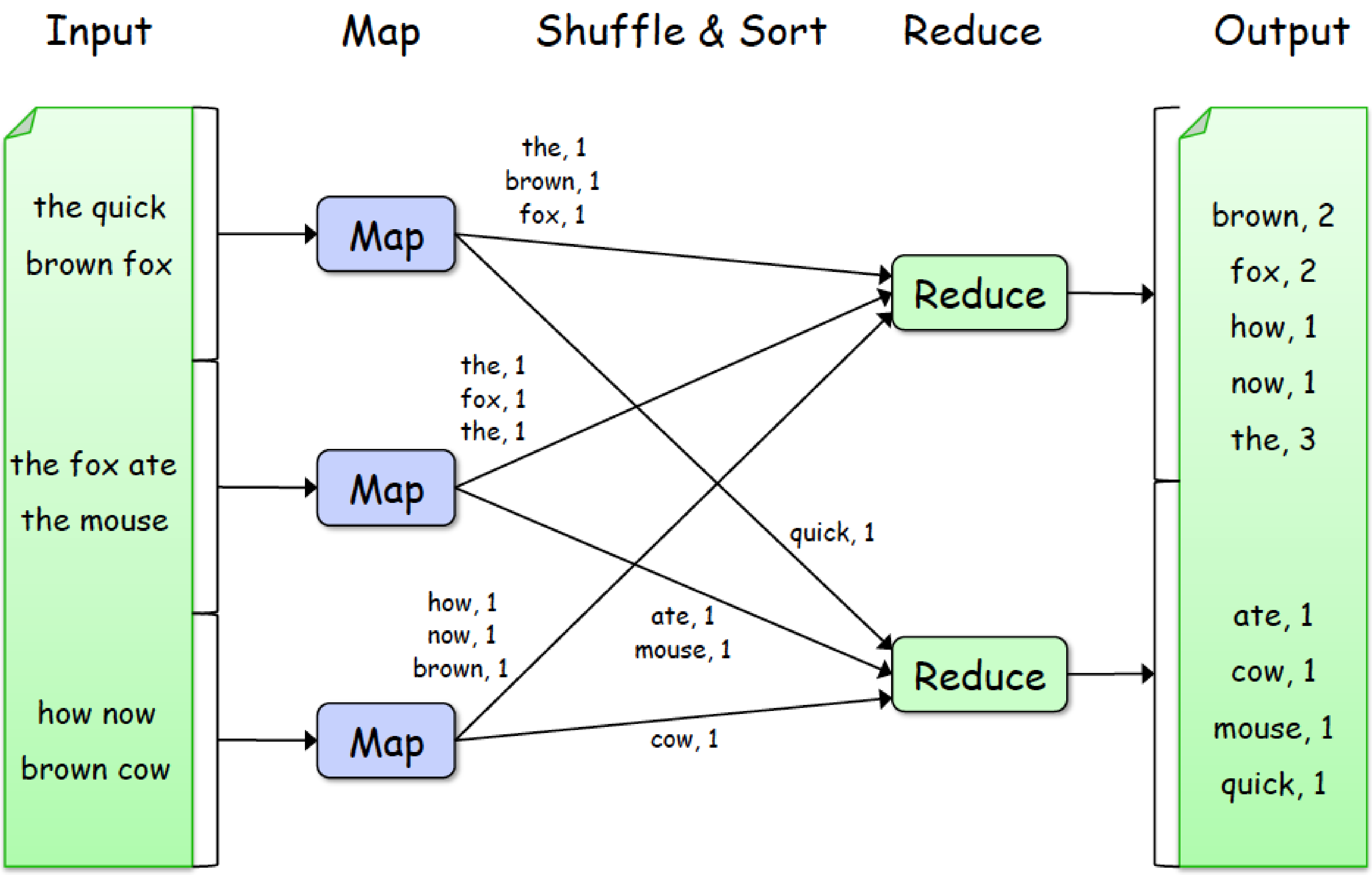
- You have a textual corpus
- Build an array/list giving, for each word, its count across all documents in the corpus



A MapReduce view

```
map(key, value):
// key: document name; value: text of document
    for each word w in value:
        emit(w, 1)

reduce(key, values):
// key: a word; value: an iterator over counts
    result = 0
    for each count v in values:
        result += v
    emit(result)
```

In a picture



Courtesy: Camil Demetrescu & Irene Finocchi

Using pySpark shell

1. Create an RDD from a text file corpus

```
>>> from operator import add
>>> lines = sc.textFile("./data/MyData/wordcount_data/*.txt")
>>> def lines_to_words(line):
...     return line.split()
...
>>> words = lines.flatMap(lines_to_words)
>>> count_vec = words.map(lambda x: (x, 1)).reduceByKey(add)
>>> print(sorted(count_vec.collect()))
[(u'Clarabella', 52), (u'Jerry', 62), (u'Minnie', 44), (u'Paperino', 45), (u'Paperone
', 59), (u'Pippo', 49), (u'Pluto', 63), (u'Silvestro', 43), (u'Titty', 51), (u'Tom',
51), (u'Topolino', 56)]
>>>
```

Using pySpark shell

Use this function to produce map

```
>>> from operator import add
>>> lines = sc.textFile("./data/MyData/wordcount_data/*.txt")
>>> def lines_to_words(line):
...     return line.split()
...
>>> words = lines.flatMap(lines_to_words)
>>> count_vec = words.map(lambda x: (x, 1)).reduceByKey(add)
>>> print(sorted(count_vec.collect()))
[(u'Clarabella', 52), (u'Jerry', 62), (u'Minnie', 44), (u'Paperino', 45), (u'Paperone
', 59), (u'Pippo', 49), (u'Pluto', 63), (u'Silvestro', 43), (u'Titty', 51), (u'Tom',
51), (u'Topolino', 56)]
>>>
```

2. Transform original RDD into a flat word sequence

Using pySpark shell

3. Transform word RDD into a <key, value> pairs RDD distributed over the cluster

```
>>> from operator import add
>>> lines = sc.textFile("./data/MyData/wordcount_data/*.txt")
>>> def lines_to_words(line):
...     return line.split()
...
>>> words = lines.flatMap(lines_to_words)
>>> count_vec = words.map(lambda x: (x, 1)).reduceByKey(add)
>>> print(sorted(count_vec.collect()))
[(u'Clarabella', 52), (u'Jerry', 62), (u'Minnie', 44), (u'Paperino', 45), (u'Paperone', 59), (u'Pippo', 49), (u'Pluto', 63), (u'Silvestro', 43), (u'Titty', 51), (u'Tom', 51), (u'Topolino', 56)]
>>>
```

Using pySpark shell

4. Aggregate data by summing values of all pairs with same key like MapReduce

```
>>> from operator import add
>>> lines = sc.textFile("./data/MyData/wordcount_data/*.txt")
>>> def lines_to_words(line):
...     return line.split()
...
>>> words = lines.flatMap(lines_to_words)
>>> count_vec = words.map(lambda x: (x, 1)).reduceByKey(add)
>>> print(sorted(count_vec.collect()))
[(u'Clarabella', 52), (u'Jerry', 62), (u'Minnie', 44), (u'Paperino', 45), (u'Paperone
', 59), (u'Pippo', 49), (u'Pluto', 63), (u'Silvestro', 43), (u'Titty', 51), (u'Tom',
51), (u'Topolino', 56)]
>>> 
```

Aggregation is performed by addition

Using pySpark shell

5. Collect partial results from workers, aggregate and deliver to *driver* → In this case a Python list of (word, count pairs)

```
>>> from operator import add
>>> lines = sc.textFile("./data/MyData/wordcount_data/*.txt")
>>> def lines_to_words(line):
...     return line.split()
...
>>> words = lines.flatMap(lines_to_words)
>>> count_vec = words.map(lambda x: (x, 1)).reduceByKey(add)
>>> print(sorted(count_vec.collect()))
[(u'Clarabella', 52), (u'Jerry', 62), (u'Minnie', 44), (u'Paperino', 45), (u'Paperone
', 59), (u'Pippo', 49), (u'Pluto', 63), (u'Silvestro', 43), (u'Titty', 51), (u'Tom',
51), (u'Topolino', 56)]
>>> 
```

Standalone application

```python
from pyspark import SparkContext, SparkConf
from operator import add

def lines_to_words(line):
    return line.split()



conf = SparkConf().setAppName("Word count 2").setMaster("local")
sc = SparkContext(conf=conf)
lines = sc.textFile("./data/MyData/wordcount_data/*.txt")
words = lines.flatMap(lines_to_words)
count_vec = words.map(lambda x: (x, 1)).reduceByKey(add)
print(sorted(count_vec.collect()))
```

```
becchett@becchett-Inspiron:~/DOCS/DIS/Didattica/BigData/Spark/spark-1.4.1-bin-had
oop2.6$ ./bin/spark-submit --master local[*] ../MyExamples/wordcount_2.py
```

Allocate to as many worker *threads* as the number of logical cores on your machine