Data Processing and Analytics (DISS-DPA)

Database Group -LIRIS CNRS

Introduction to Big Data Management (based on Univ. Tartu's material)

Lyon 1 University
Fall 2022

Speaker: Riccardo Tommasini

Objectives

By the end of this and the following lecture you should be able to Understand :

- Batch Processing with Apache Spark
- In-memory processing wth Spark RDDs
- Spark DataFrames
- Query processing and Optimizaiton of Spark SQL

Apache Spark[1]

- An open source, general-purpose, fast, large-scale data processing engine
- Started in AMPLab @ UC Berkley, now Data Bricks
- Written in Scala
- Emerged as the most powerful, highly in-demand, and the primary big data framework across the major industries of the world.
- It has become the preferred framework by some mainstream players like Alibaba, Amazon, eBay, Yahoo, etc.
- Considered as a third generation distributed data processing system
  - Hadoop is considered as a second generation

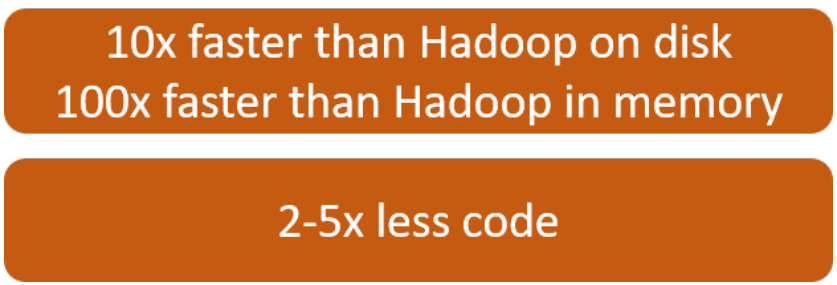
[1] Zaharia, Matei, et al. "Spark: Cluster computing with working sets."HotCloud 10.10-10 (2010): 95.

Apache Spark (cont.)

Why would we need a new generation?

- Limitations of MapReduce (Hadoop)
  - Can not Cache the intermediate data in memory
  - Just two abstractions Map and Reduce
  - Applications can only be written in Java
  - Poor support for low-latency data processing
- Exploit advancements in hardware
  - Memory is much cheaper
  - Multi-core is now a commodity

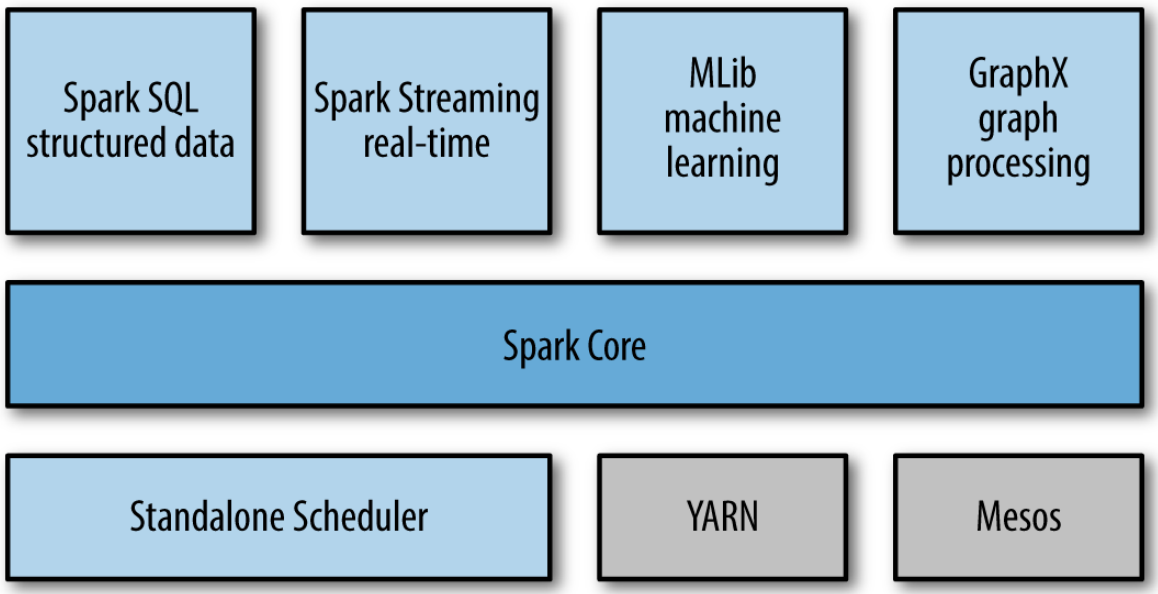
Spark is,



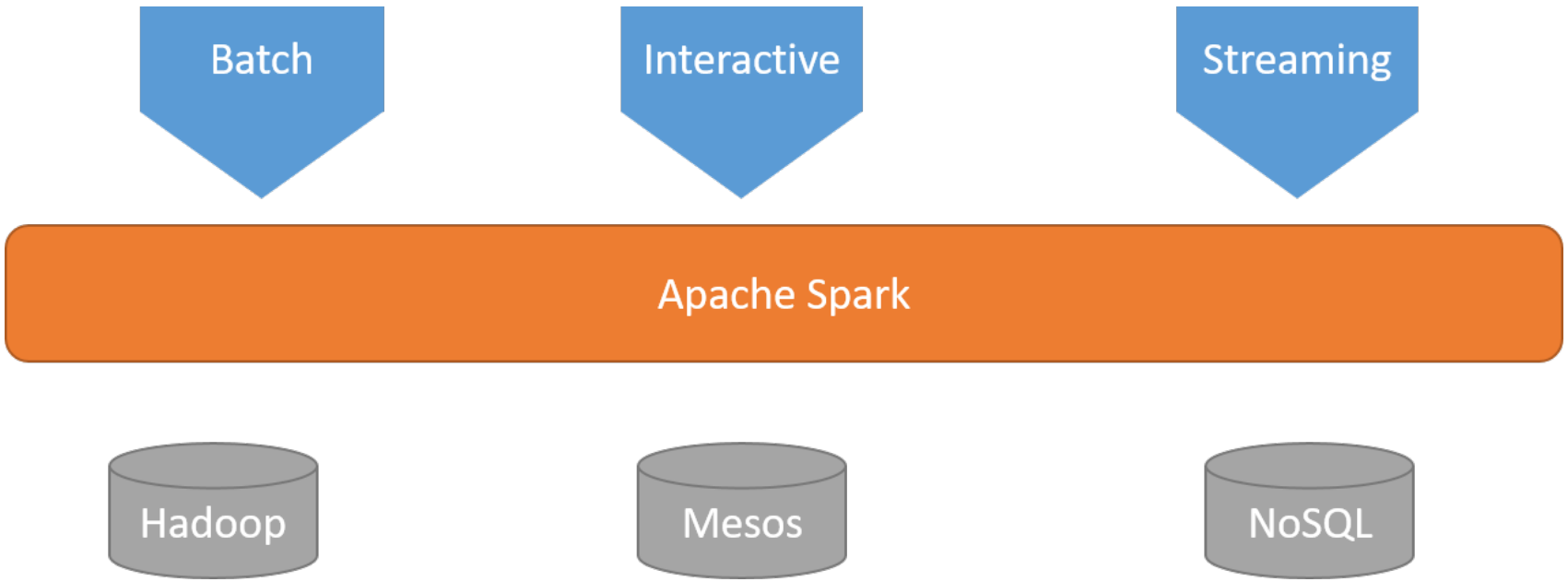
Third Generation Distributed Systems

- Handle both batch and real-time processing
- Exploit RAM as much as disk
- Multiple-core aware
- Do not reinvent the wheel
  - Use HDFS for storage
  - Apache Mesos/YARN for execution
- Plays well with Hadoop
- Iterative processing

Unified Platform for Big Data Processing



Unified Platform for Big Data Processing (cont.)



Why Unification?

- Good for:
  - Developers: One platform to learn
  - Users: Take apps everywhere
  - Distributions: More applications
- Is based on a common abstraction
- Limitations?
  - One size does not fit all
  - It might not be the best under certain constraints

Word count In Spark

- Lookup the code for word count in MapReduce[1]
- How does it look in Spark?

```
sc = SparkContext(appName="PythonWordCount")
lines = sc.textFile(sys.argv[1], 1)
counts = lines.flatMap(lambda x: x.split(' '))
.map(lambda x: (x, 1)).reduceByKey(lambda a, v: a+v)
output = counts.collect()
for (word, count) in output:
    print "%s: %i" % (word, count)
sc.stop()
```

[1] `https://dzone.com/articles/word-count-hello-word-program-in-mapreduce`, at least 20 lines of code

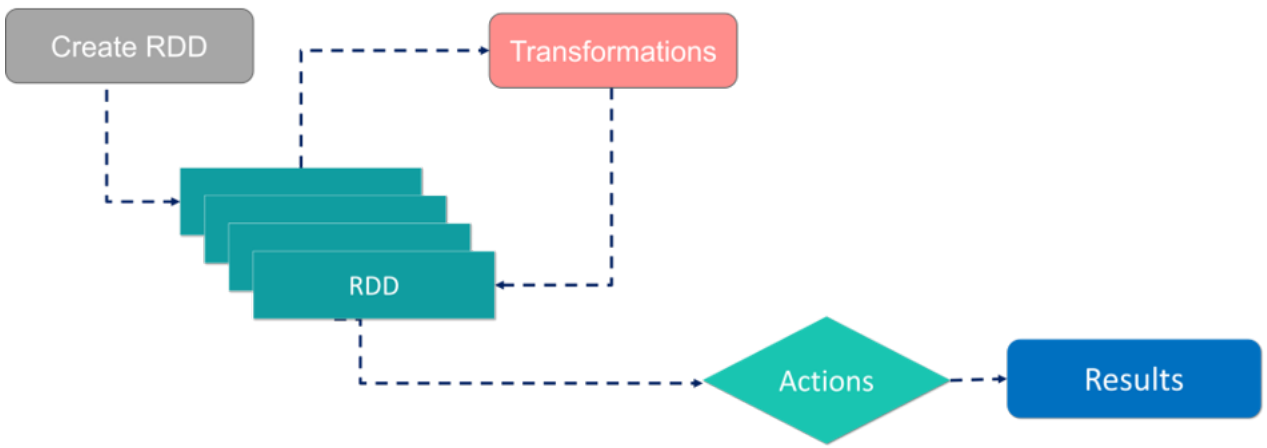
Spark Abstractions

Spark has a layered architecture with all the components and layers integrated with other extensions and libraries.

- Spark core abstraction is Resilient Distributed Dataset (RDD)
  - Resilient: fault tolerant and can be recomputed when recovering from a failure
  - Distributed: processing takes place over several nodes in parallel, like MapReduce
  - Dataset: initial data can come from files, memory, or created programmatically
  - Immutable: once created cannot be changed
  - Lineage: each RDD knows about its parents
- Spark applications are series of operations that transform input RDDs into output RDDs or final values

Resilient Distributed Dataset (RDD)

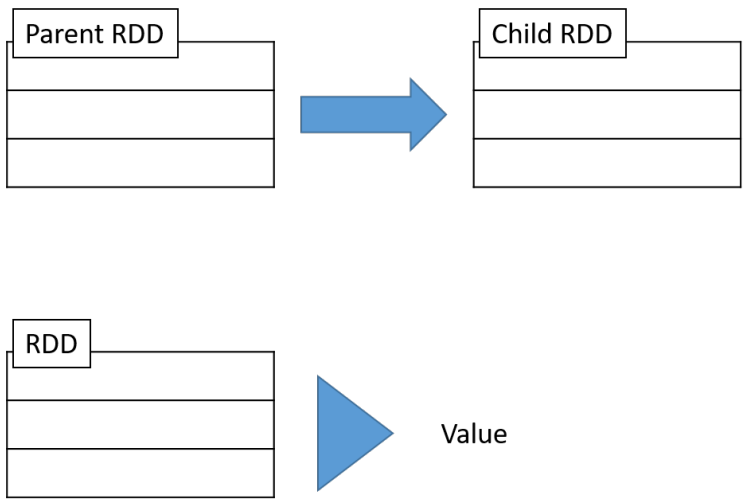
- Provides a distributed computing environment
- RDDs are highly resilient i.e., they are able to recover quickly from any issues as the same data chunks are replicated across multiple executor nodes.
- Once an RDD is created, it becomes immutable i.e., its state cannot be modified after it is created, but they can be transformed.
- Each dataset in RDD is divided into logical partitions, which may be computed on different nodes of the cluster.
- Transformations or Actions can be performed on the complete data parallelly.



RDD Operations

Two main types of RDD operations

- **Transformations:** result in a new RDD (lazy)
  - Can be chained
  - Forked
  - Joined
- **Actions:** return values, no more RDDs (eager)
  - One action at the end of each transformation chain



Note: Laziness/ Eagerness is how we can limit network communication using the programming model.

Transformation and Action

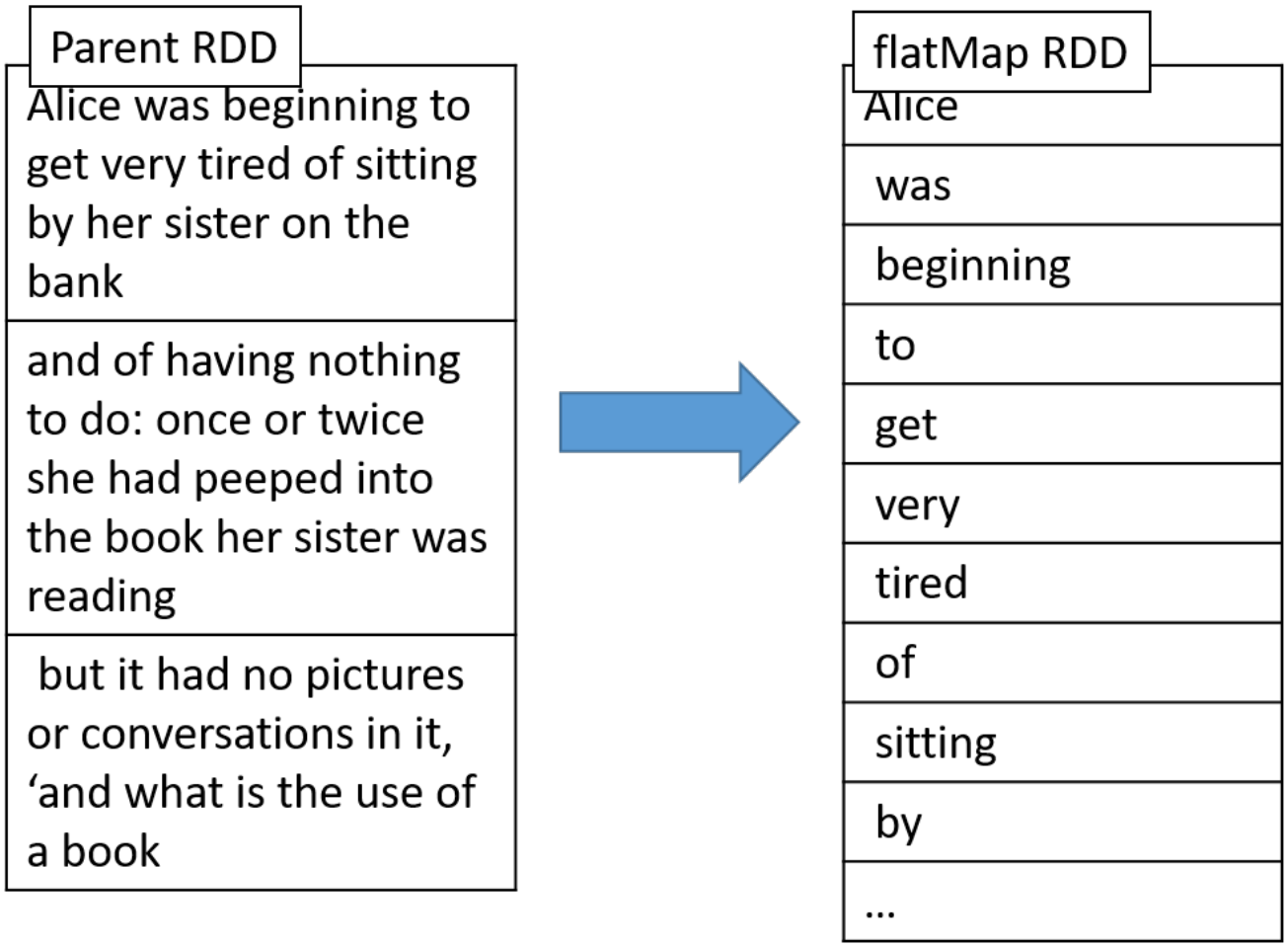
- Transformations:
  var a = sc.textFile("hdfs:// ..... /xyz.text") — RDD0
  var b = a.filter(....) — RDD1
  var c = b.distinct (....) — RDD2
- Actions:
  c.collect()

RDD Transformations

- Transformations create a new RDD from an existing one
- RDDs are immutable
  - Apply a series of transformations to modify the data as needed
- Transformations can be divided into two types:
  - Narrow Transformations:
    - Applied to a single partition of the parent RDD to generate a new RDD
    - The examples for narrow transformations are:
      - Map(function) - Returns a new distributed dataset by passing each element of the source.
      1-to-1 mapping
      - Filter(function) - Returns a new dataset formed by selecting selected elements of the source.
      1-to-1 mapping with selectivity
      - FlatMap(function) - Similar to map, but each input item can be mapped to 0 or more output items.
      1-to-Many mapping
      - mapPartitions() - Similar to map, but runs separately on each partition (block) of the RDD
  - Wide Transformations:
    - Applied on multiple partitions to generate a new RDD.
    - The examples for wide transformations are:
      - ReduceByKey() - Returns a dataset of (K, V) pairs where the values for each key are aggregated
      - Union() - Returns a new dataset that contains the union of the elements in the source dataset and the argument

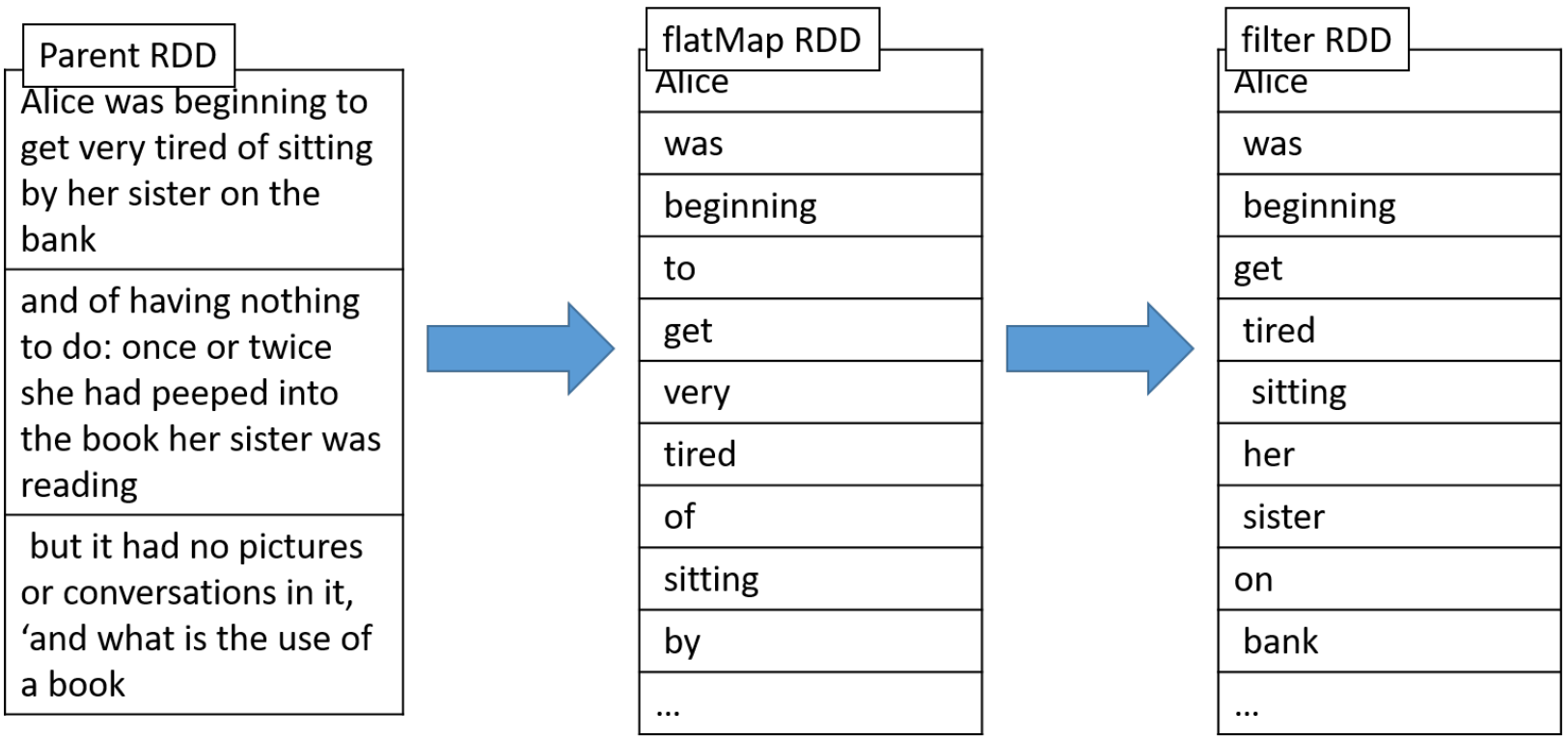
Transformation: flatMap

```
lines = lines.flatMap(lambda x: x.split(' '))
```



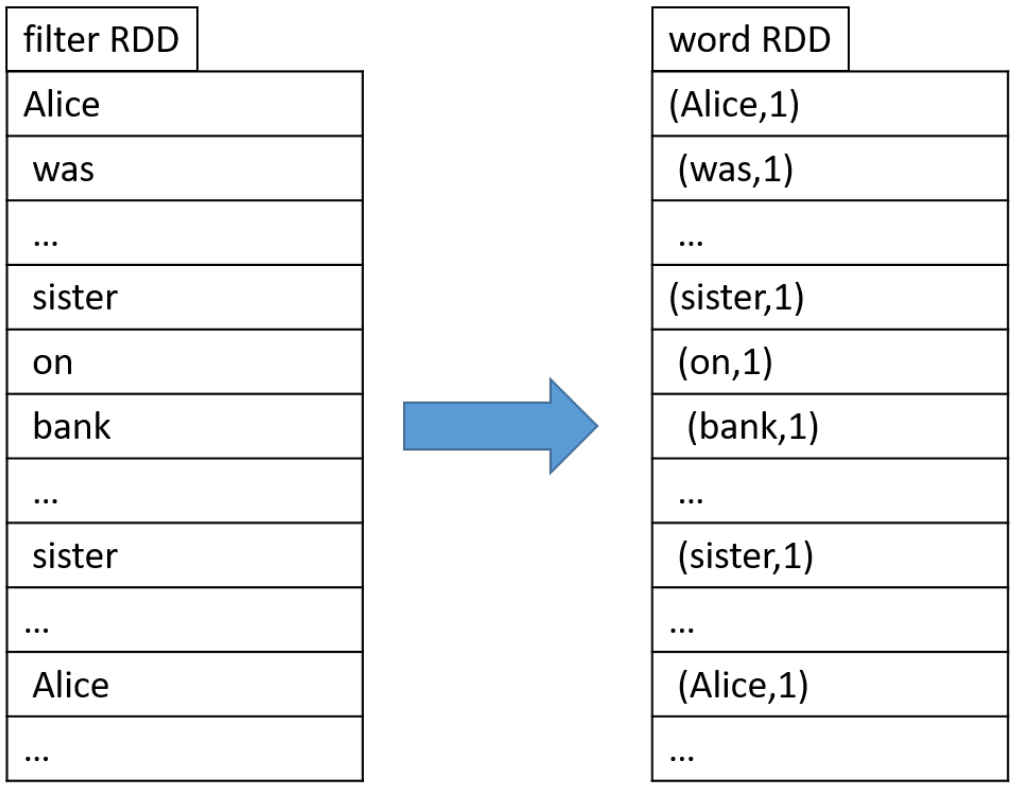
Transformation: filter

```
filtered = lines.filter(lambda x: x not
in ['by', 'very','to', 'the' ])
```



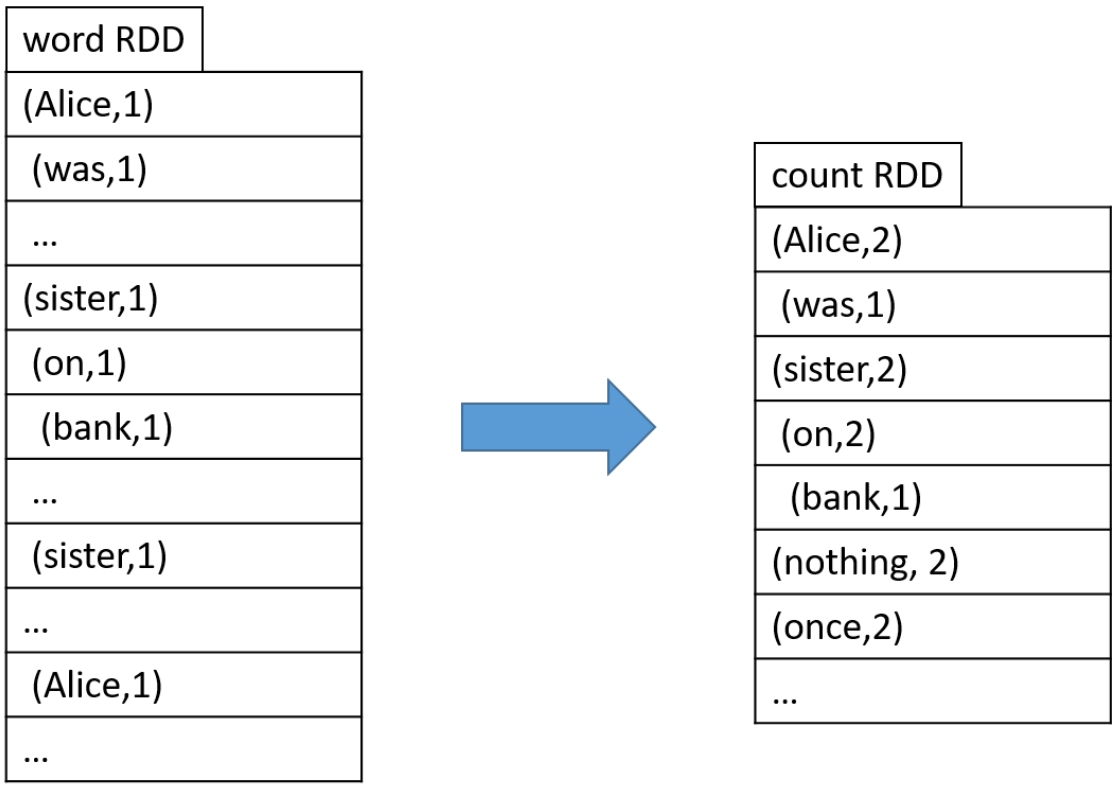
Transformation: map

```
word = filtered.map(lambda x: (x,1))
```



Transformation: reduceByKey

```
counts = word.reduceByKey(lambda x, y: x +y)
```

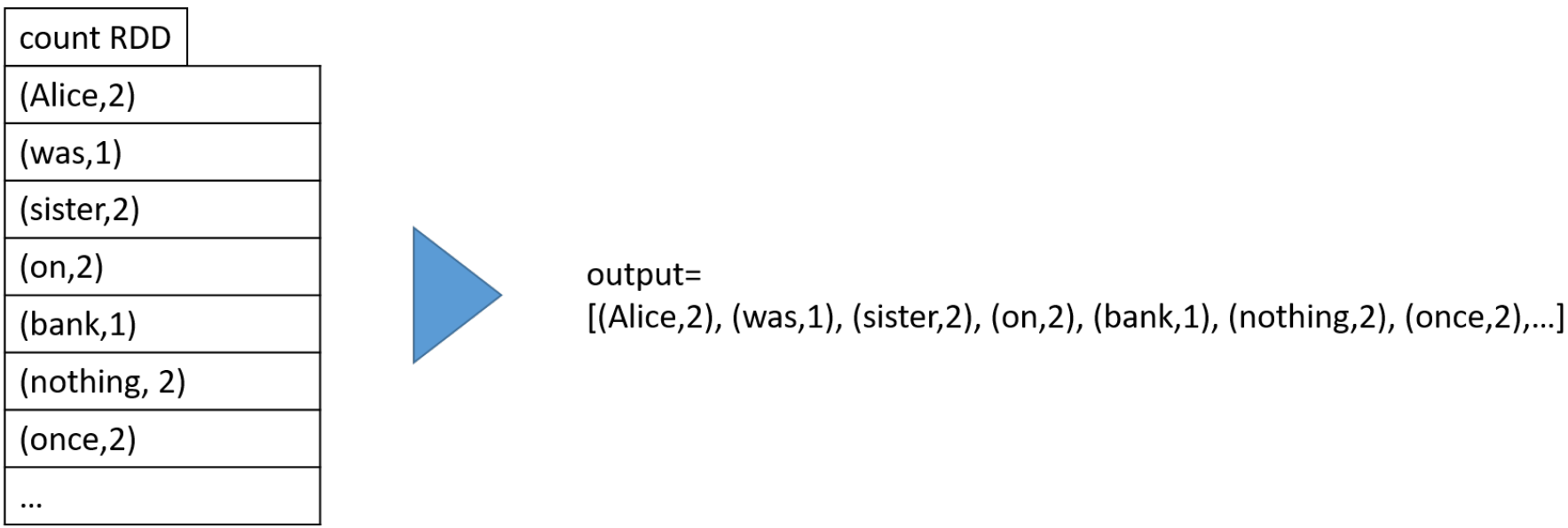


RDD Actions

- Actions trigger execution of transformation chains
- No further RDD transformations
- Common actions
  - Collect(): returns an array of all the elements
  - Take(n): returns an array of the first n elements
  - Count(): returns the number of elements in RDD
  - saveAsTextFile(): saves the data to file system, either HDFS for local

Action: collect

```
output = counts.collect()
```



Lazy Execution

- Spark follows a lazy execution scheme
  - No RDDs are computed until an action is specified
- Why?
  - Help optimize execution plan
- Only lineage is created as moving from one transformation to the other
  - To learn about lineage of a chain of transformations, call toDebugString() after the transformation you are interested in

RDD Lineage

RDD Lineage is a graph of all the parent RDD of a RDD. Since Spark does not replicate data, it is possible to lose some data. In case some Dataset is lost, it is possible to use RDD Lineage to recreate the lost Dataset.

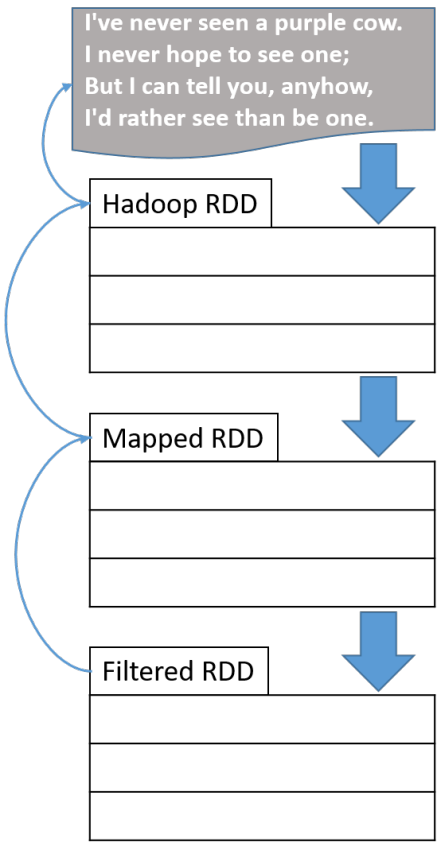
Application

```
mydata\_filt = sc.textFile(''file.txt'')
.map(lambda line: line.toUpperCase())
.filter(lambda line:line.startsWith(''I''))
```

Lineage

```
mydata_filt.toDebugString()
```

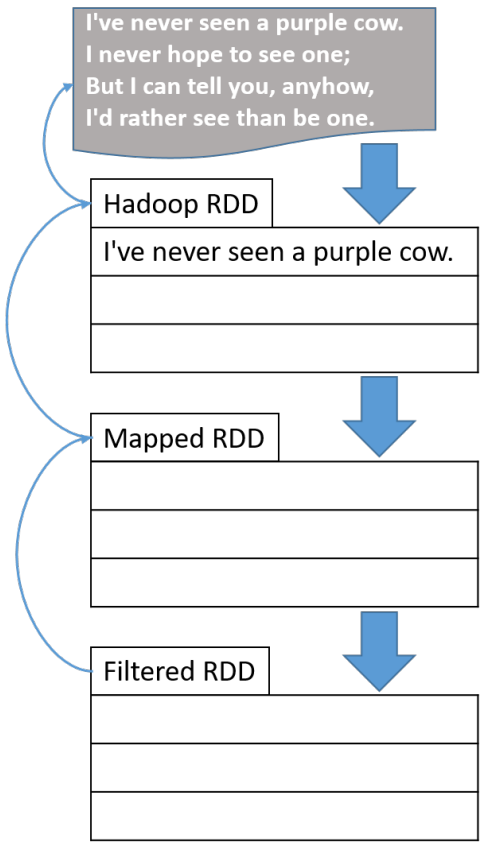
```
(2) FilteredRDD[7] at filter ...
| MappedRDD[6] at map ...
| file.txt MappedRDD[5] ...
| file.txt HadoopRDD[4] ...
```



Pipelining

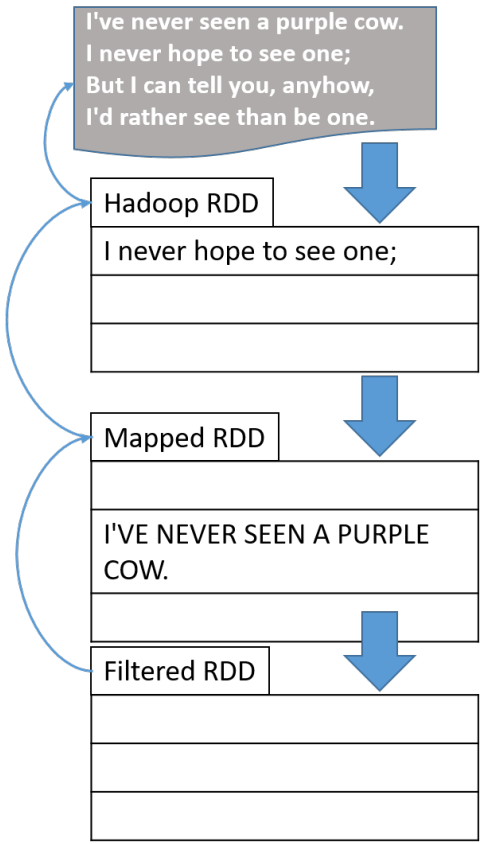
Pipelining is a sequence of algorithms that are executed for processing and learning from data.

- When possible, Spark will pass individual outputs of each transformation to the next.
  - In Hadoop, all intermediate results are completely calculated before beginning the next step
- Pipelining helps reduce latency



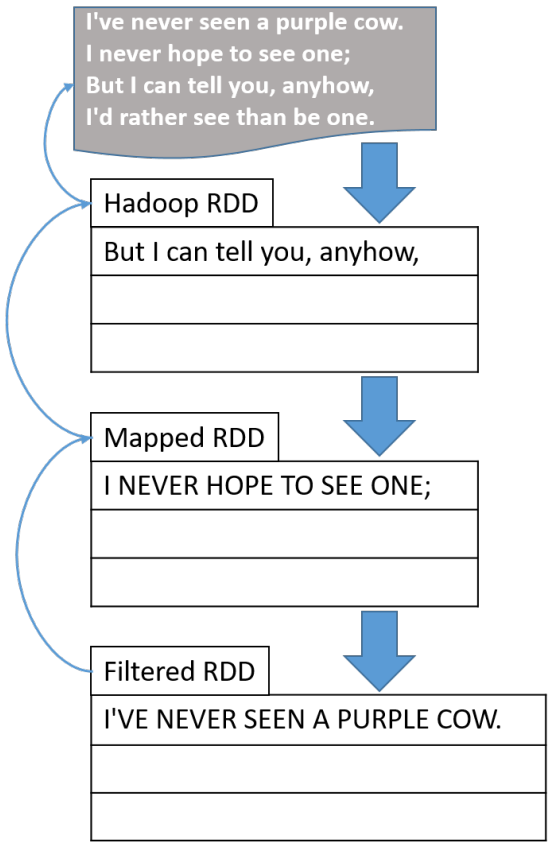
Pipelining

- When possible, Spark will pass individual outputs of each transformation to the next.
  - In Hadoop, all intermediate results are completely calculated before beginning the next step
- Pipelining helps reduce latency



Pipelining

- When possible, Spark will pass individual outputs of each transformation to the next.
  - In Hadoop, all intermediate results are completely calculated before beginning the next step
- Pipelining helps reduce latency



Creating RDDs

- We learned that RDDs can be created as a result of transformations on parent RDDs
- What about root RDDs?
- Can be created from
  - Data in memory, collections/ By distributing collection of objects
  - From files, example from text files as we saw earlier/ By loading an external dataset

Creating RDDs from Collections

Using sc.parallelize() the elements of the collection are copied to, form a distributed dataset that can be operated on in parallel.
This allows Spark to distribute the data across multiple nodes, instead of depending on a single node to process the data:

- SparkContext.parallelize(collection)

```
mydata = ['Alice', 'Jack', 'Andrew', 'Frank']
myRDD = sc.parallelize(mydata)
myRDD.take(2)
output: ['Alice', 'Jack']
```

- Useful for:
  - Testing
  - Integrating

Creating RDDs from Files

- So far, we saw sc.textFile("file")
  - Accepts a single file, a wildcard list of files, or comma-separated list of file names
  - Examples:
    - sc.textFile("myfile.txt")
    - sc.textFile("mydata/*.log")
    - sc.textFile("myfile1.txt,myfile2.txt")
    - textFile only works with line-delimited text files
  - Each line in the file is a separate record in the RDD
  - Files are referenced by relative or absolute URI
    - Absolute URI: file:/home/training/myfile.txt or hdfs://localhost/loudacre/myfile.txt
    - Relative URI (uses default file system):myfile.txt
  - What about other file formats?

Creating RDDs from Other File Formats

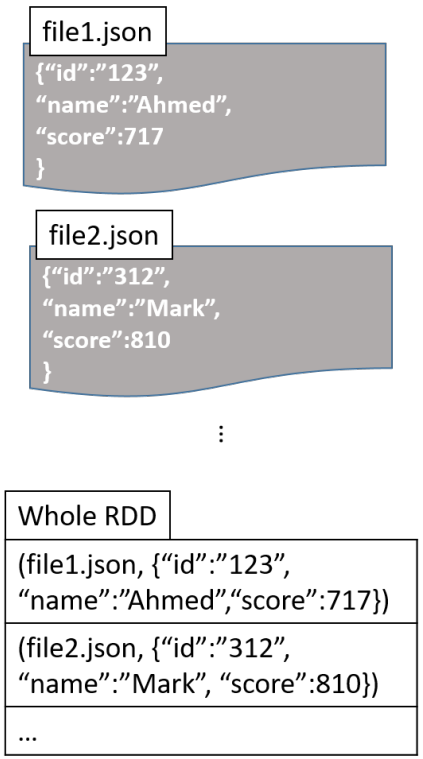
- Spark uses Hadoop's InputFormat and OutputFormat Java classes
  - TextInputFormat/TextOutputFormat
  - SequenceInputFormat/SequenceOutputFormat
  - FixedLengthInputFormat
- Support for other formats
  - AvroInputFormat/AvroOutputFormat

Using Input/output Formats

- Define input format using sc.hadoopFile
  - Or newAPIhadoopFile for New API classes
- Define output format using rdd.saveAsHadoopFile
  - Or saveAsNewAPIhadoopFile for New API classes

Whole-file-based RDDs

- sc.textFile puts each line as a separate element
  - What if you are processing XML or JSON files?
- sc.wholeTextFile(directory) creates a single element in RDD for the whole content of a file in the input directory
  - Creates a special type of RDDs, (paired RDDs), we discuss later
  - Works for files with small sizes (elements must fit in memory)



RDD Content

- RDD can hold elements of any type:
  - Primitive data types
  - Sequence types
  - Scala/Java objects (if serializable)
  - Mixed types
- Special RDDs
  - Pair RDDs: consist of key-value pairs,
    - sc.wholeTextFile
  - Double RDDs
    - RDDs consisting of numeric data

Other General RDD Transformations

- Single RDD transformations
  - distinct: removes duplicate values
  - sortBy: sorts by the input function
- Multi-RDD transformations
  - intersection: outputs common elements of the input RDDs
  - union: add all elements from input RDDs to the output RDD
  - zip: performs a cross product between the input RDDs

Pair RDDs

- A special form of RDDs
  - Elements must be a tuple of two elements (key, value)
  - Keys and values can be of any type
- Why use Pair RDDs
  - To have the benefits of MapReduce
  - Ability to scale by forwarding tuples of the same key value to the same processing node, shuffling.
  - Other additional transformations are built-in, e.g., sorting, joining, grouping, counting, etc.

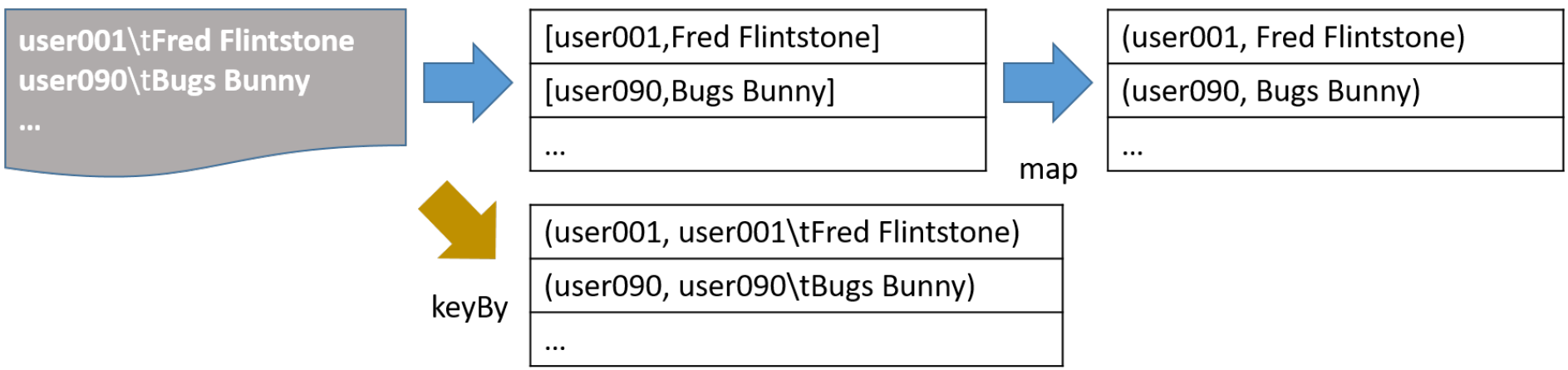
Creating Pair RDDs

- You can have your root RDD as a pair RDD, e.g., sc.wholeTextFile
- You can use a transformation to put the data in pair RDDs
  - map
  - flatMap/flatMapValues
  - keyBy

Example

- Create a pair RDD from a tab-delimited file

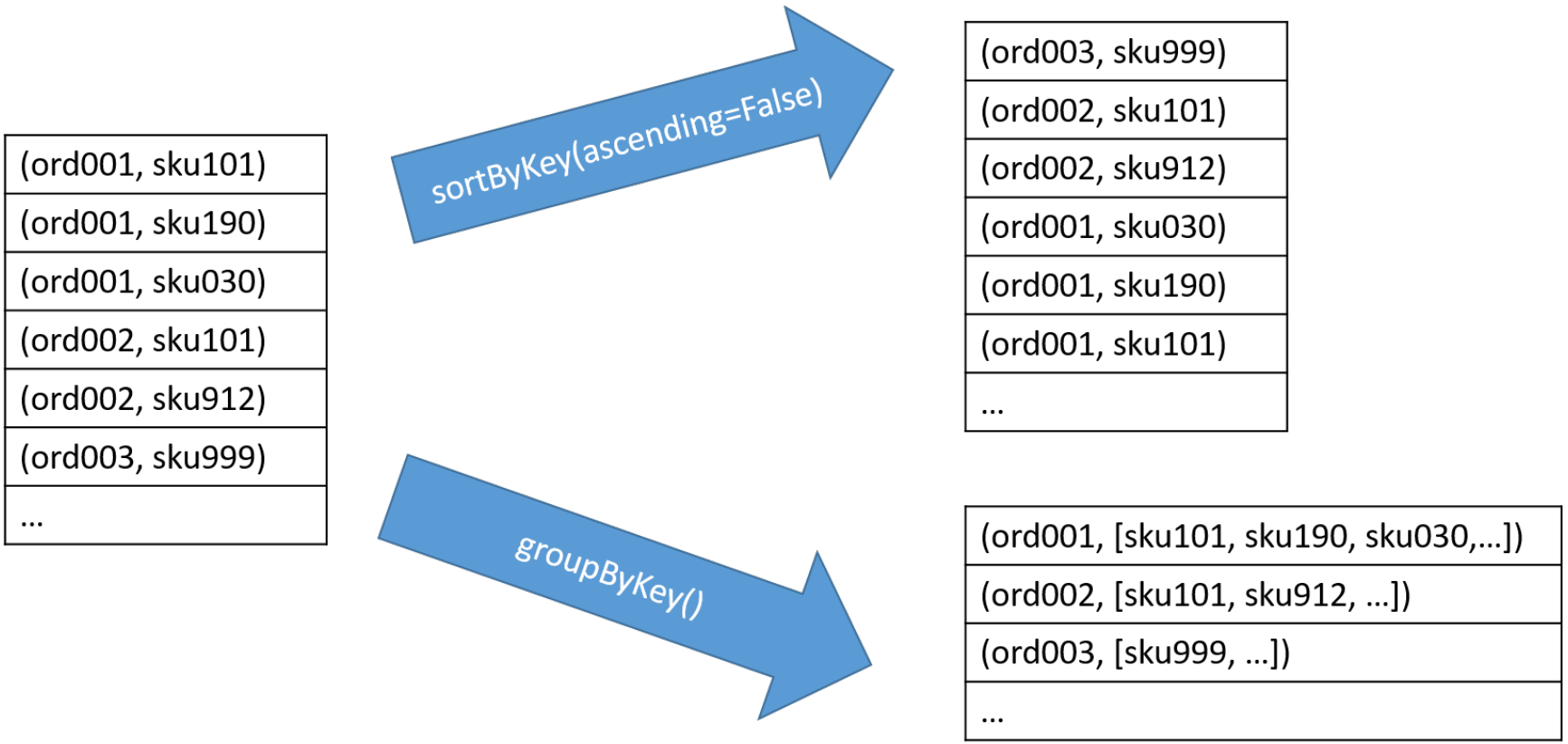
```
users=sc.textFile(file)
.map(lambda line: line.split(''))
.map(lambda elems: (elems[0],elems[1]))
.keyBy(lambda line: line.split('\t')[0])
```



Other Pair RDD Transformations

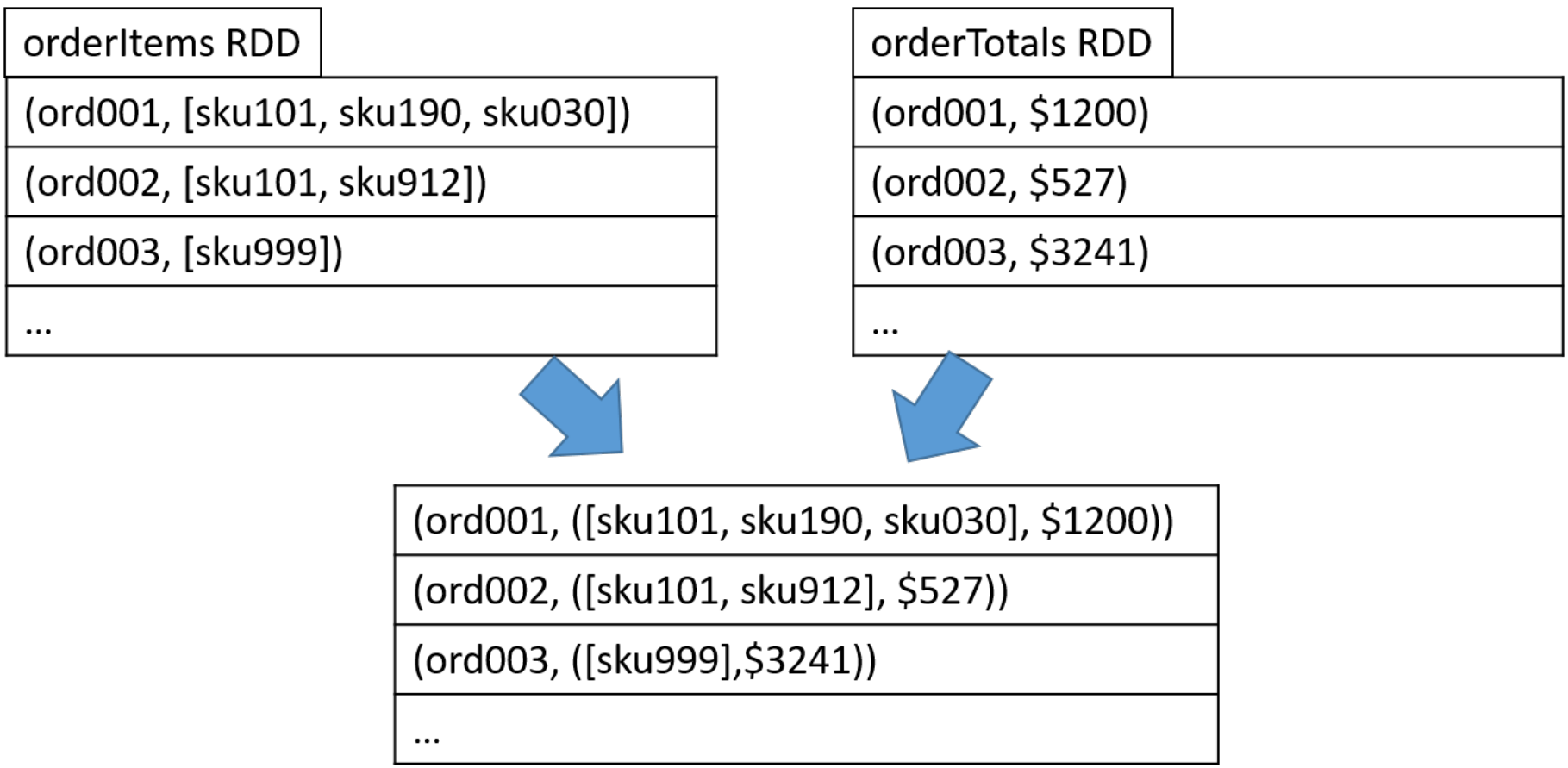
- countByKey
  - Returns a pair RDD with the same key as parent and value is the count of key occurrences
- groupByKey
  - Similar to the input of Hadoop Reducer, (key, [list of values])
- sortByKey(ascending=true/false)
  - Returns a pair RDD sorted by the key
- join
  - Takes two input pair RDDs with the same key (key, value1), (key, value2)
  - Returns (Key, (value1,value2))

Examples


(ord001, sku101)
(ord001, sku190)
(ord001, sku030)
(ord002, sku101)
(ord002, sku912)
(ord003, sku999)
...
sortByKey(ascending=False)
(ord003, sku999)
(ord002, sku101)
(ord002, sku912)
(ord001, sku030)
(ord001, sku190)
(ord001, sku101)
...
groupByKey()
(ord001, [sku101, sku190, sku030,...])
(ord002, [sku101, sku912, ...])
(ord003, [sku999, ...])
...

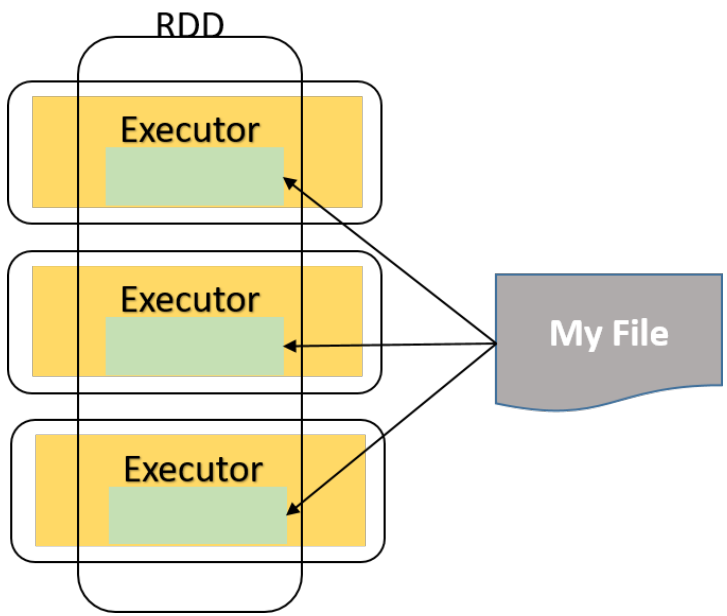
Example: join by key

```
Orders = orderItems.join(orderTotals)
```



RDD Partitions

- Data is partitioned over the worker nodes
  - E.g., to follow blocks of an HDFS file
- Partitioning is done automatically by Spark
  - Optionally, you can control the number of partitions
  - You can specify the minimum number of partitions, default is 2
  - sc.textFile("My File", 3)



Parallel Operations on Partitions

- Spark tries to maximize the localization of data processing
  - Group all transformations that can be processed on the same data partition
- Some transformations are partition-preserving
  - E.g., map, flatMap, filter
- Some transformations repartition
  - E.g., reduceByKey, groupByKey, sortByKey

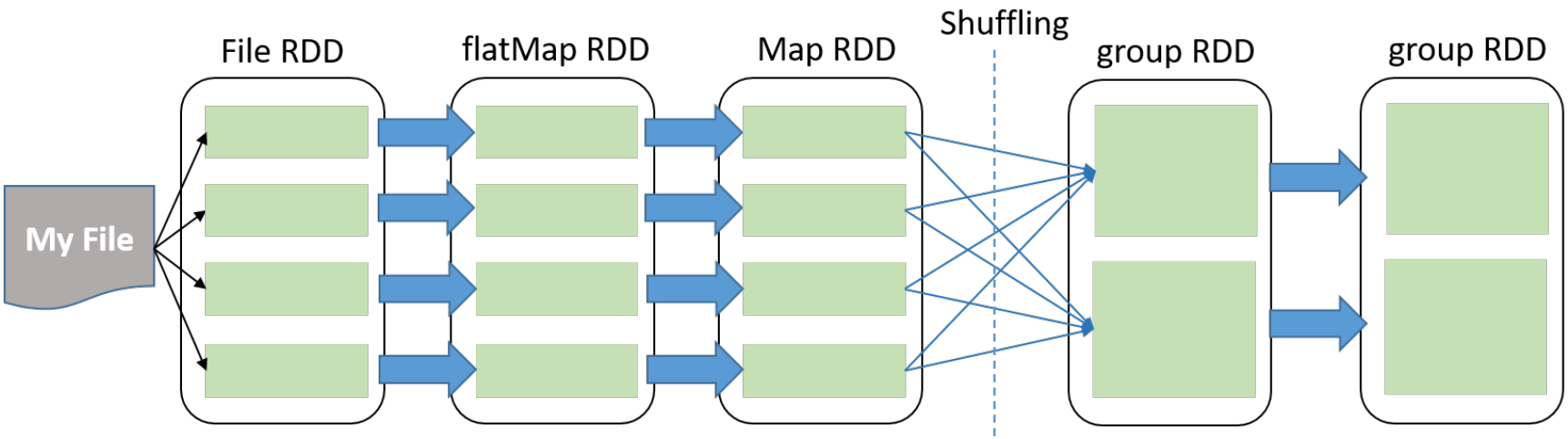
Stages

- All operations that can work on the same data partition are grouped into a stage.
  - Tasks within a stage are pipelined together
- Spark divides the DAG of the job into stages
- How Spark Calculates Stages? Based on RDD dependencies
  - Narrow dependencies
    - Only one child depends on the RDD
    - No shuffle required
  - Wide (shuffle) dependencies
    - Multiple children depend on the RDD
    - Defines a new stage

Example: Average Word Length By First Letter

We have the following chain of operations

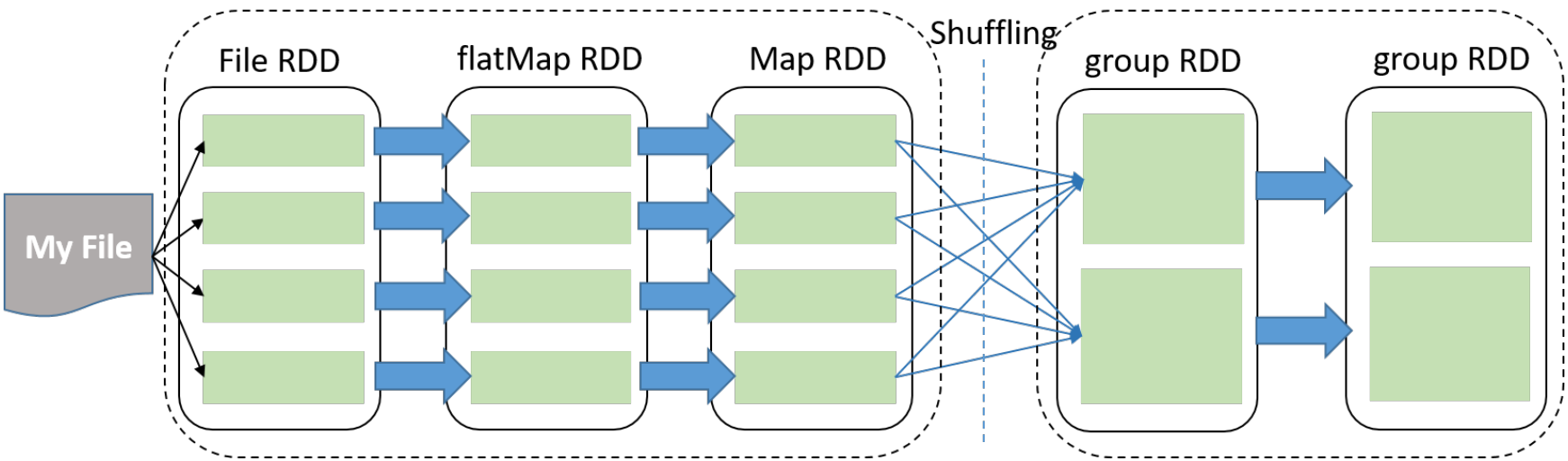
```
avglength=sc.textFile(file).flatMap(line: line.split())
.map(word:(word[0],len(word))).groupByKey()
.map((k,values):(k,sum(values)/len(values)))
```



Execution Stages

We have the following chain of operations

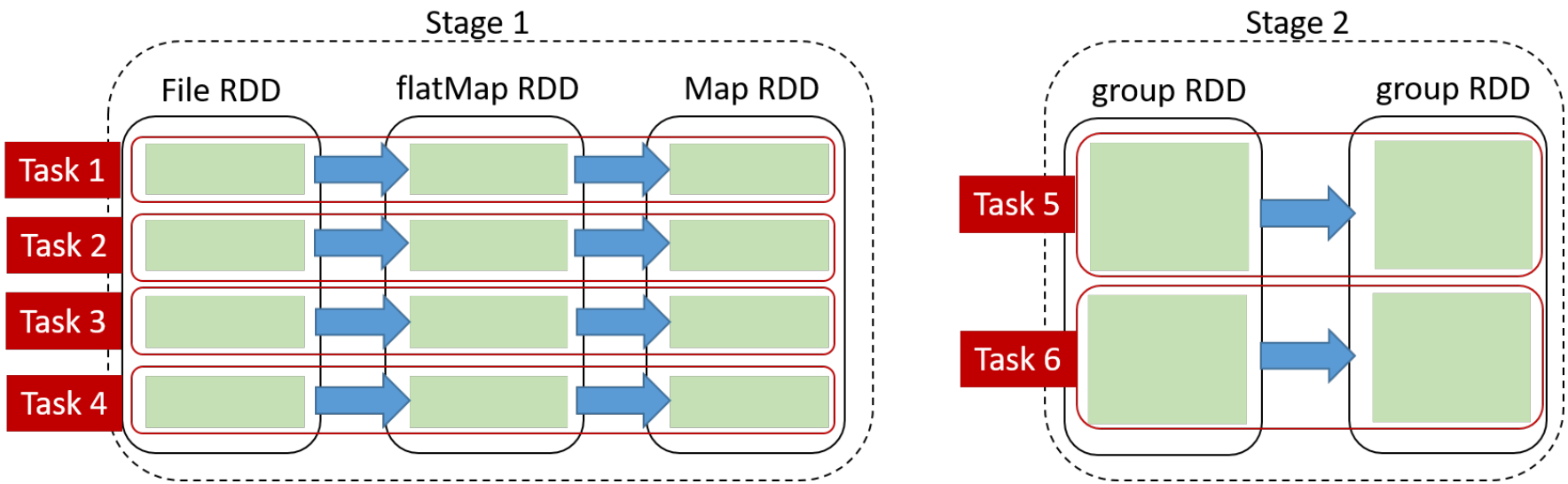
```
avglength=sc.textFile(file).flatMap(line: line.split())
.map(word:(word[0],len(word)).groupByKey()
.map((k,values):(k,sum(values)/len(values)))
```



Tasks Pipelining

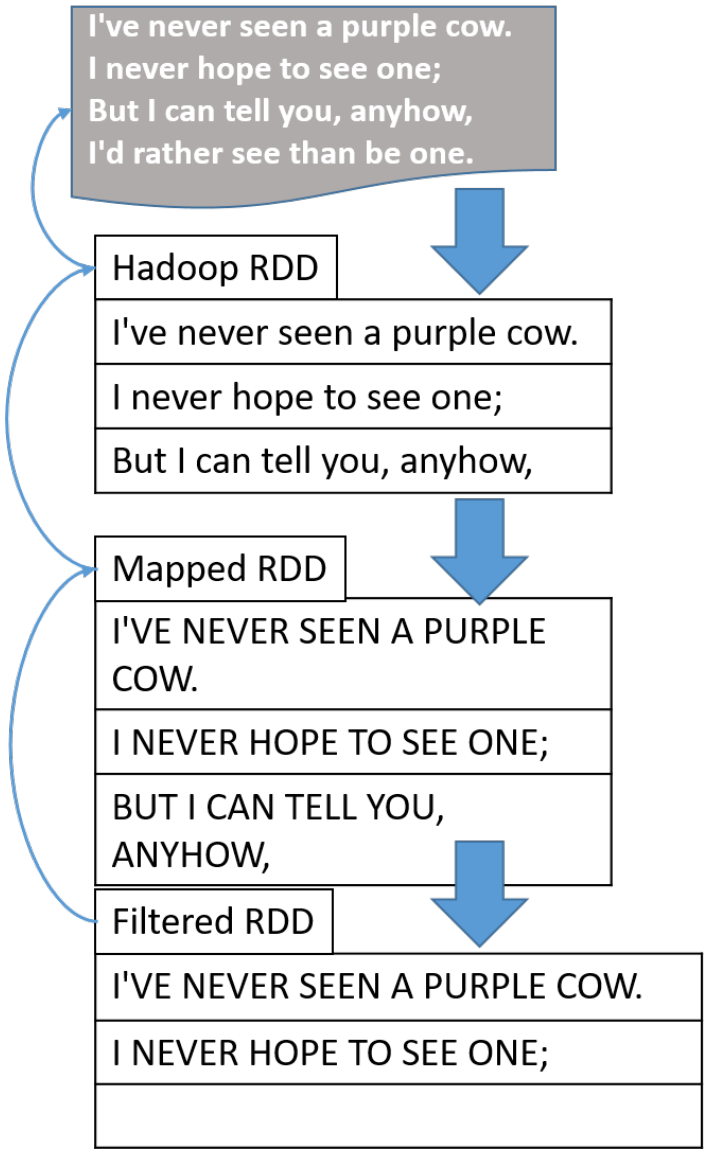
We have the following chain of operations

```
avglength=sc.textFile(file).flatMap(line: line.split())
.map(word:(word[0],len(word))).groupByKey()
.map((k,values):(k,sum(values)/len(values)))
```



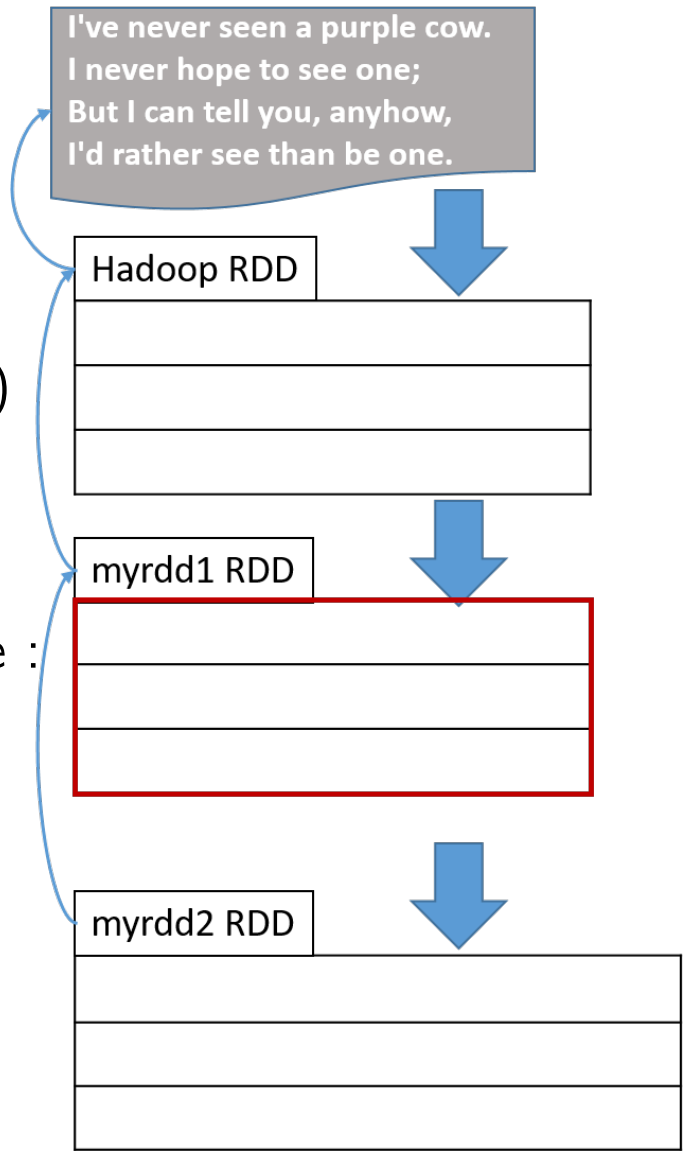
RDD Persistence

- Spark maintains lineage of RDDs by storing a reference to the parent RDD in the child one
- Each time an action is called on an RDD, Spark recursively traverses the lineage and performs the transformation
- This might be costly, especially in case of disk access
- Persistence makes Spark maintain the content of RDDs, default in memory
- Useful for iterative, e.g. machine learning, and interactive processing



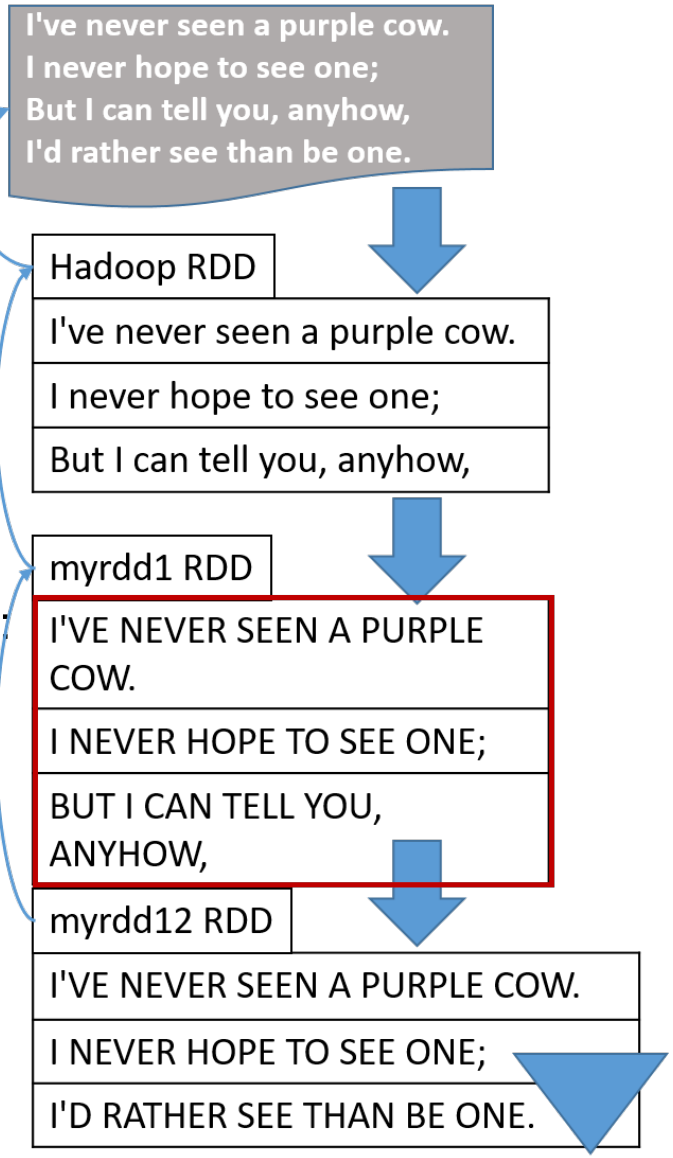
RDD Persistence

```
mydata = sc.textFile(``file.txt'')
myrdd1 = mydata.map(lambda line:
line.toUpperCase())
myrdd1.persist()
myrdd2 = myrdd1.filter(lambda line:
line.startsWith(``I''))
```



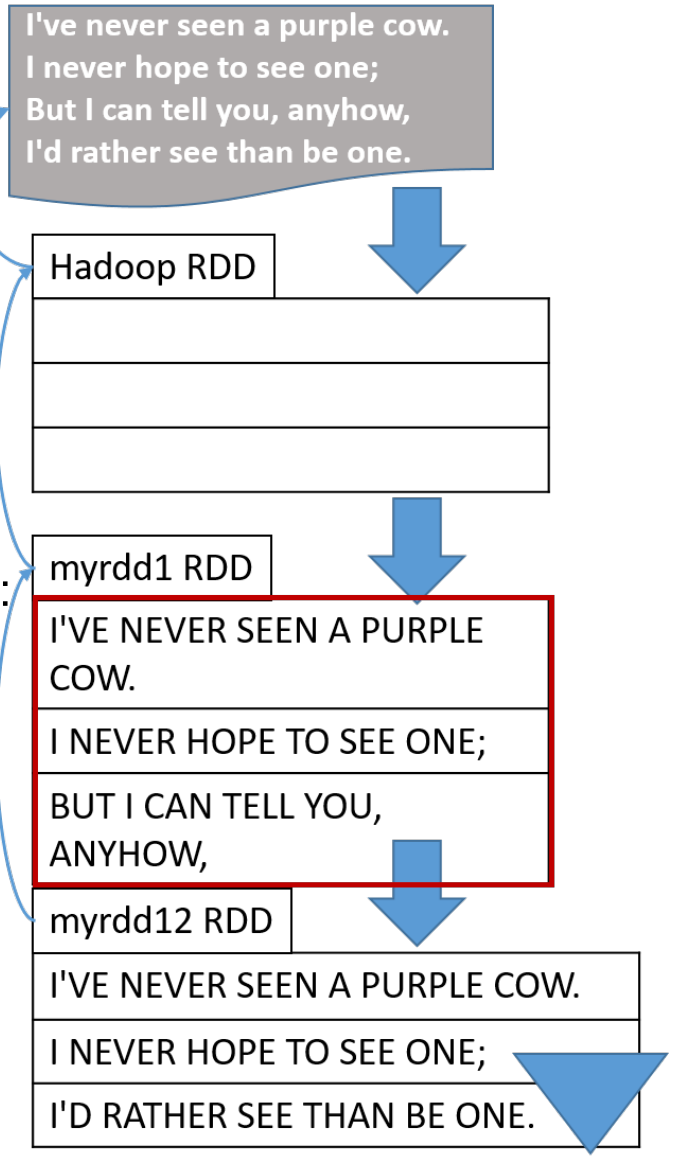
RDD Persistence

```
mydata = sc.textFile(``file.txt'')
myrdd1 = mydata.map(lambda line:
line.toUpperCase())
myrdd1.persist()}
myrdd2 = myrdd1.filter(lambda line:
line.startsWith(``I''))
myrdd2.collect()
```



RDD Persistence

```
mydata = sc.textFile(``file.txt'')
myrdd1 = mydata.map(lambda line:
line.toUpperCase())
myrdd1.persist()
myrdd2 = myrdd1.filter(lambda line:
line.startsWith(``I''))
myrdd2.collect()
myrdd2.count()
```



Persistence Options

- By default, data is stored in-memory only
- In-memory persistence is a suggestion for Spark
  - At the time when memory is full, Spark can choose to remove persisted RDDs
  - Least recently used RDDs are removed first
- If needed again, RDDs are recomputed
- Other persistence options let us control
  - Storage location
  - Format in memory

Persistence Options

- Storage location
  - Memory only: default location
    - RDD.cache() = RDD.persist(MEMORY_ONLY)
    - Data is stored unserialzied (object form)
  - Memory and disk: spill to disk if RDD does not fit in memory
  - Memory only serialized: store data as one byte array in memory
  - Memory and disk serialized
  - Disk only: store all partitions on disk

The End


Thank You
Mahalo
Kiitos
Tack
Toda
Grazie
Thanks
Obrigado
Takk
Merci
Gracias