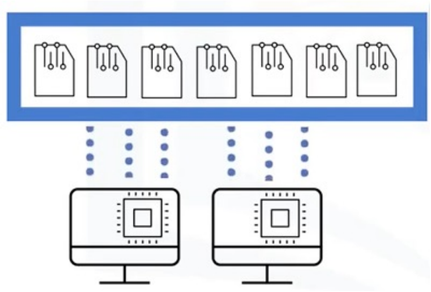
CIT650 Introduction to Big Data

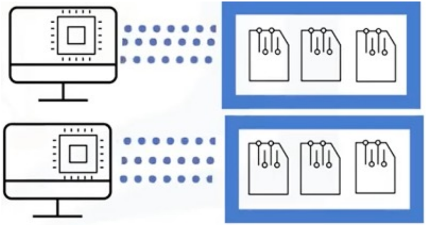
Introduction to SPARK

Distributed Computing vs. Parallel computing

Parallel Computing
processors access shared memory



Distributed Computing
processors usually have their own private memory



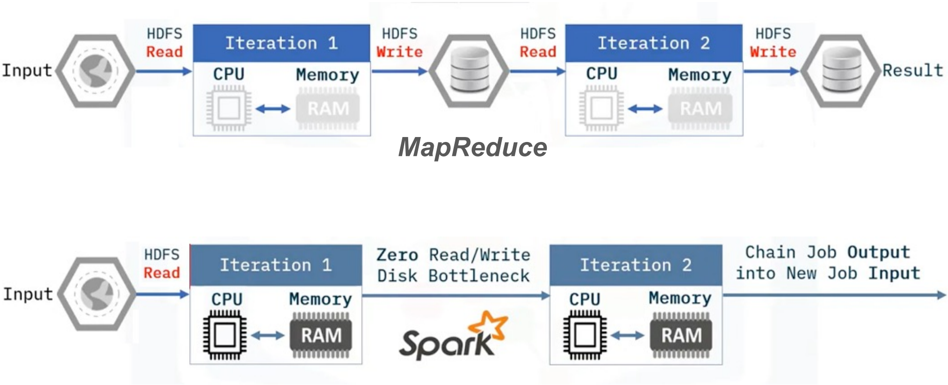
Distributed Computing Benefits

- Distributed systems are inherently scalable as they work across multiple independent machines and scale horizontally.
- Distributed systems offer fault tolerance as independent nodes fail without affecting system integrity.
- Distributed systems provide redundancy that enables business continuity.

Spark for Distributed Computing

- Spark supports a computing framework for large-scale data processing and analysis.
- Spark provides parallel distributed data processing capabilities.
- Spark provides scalability.
- Spark provides fault-tolerance on commodity hardware.
- Spark enables in-memory processing.
- Spark enables programming flexibility with easy-to-use Python, Scala, and Java APIs.

Spark vs. MapReduce



Apache Spark[1]

- A general-purpose, fast, large-scale data processing engine
- Started in AMPLab @UC Berkley, now Data Bricks
- Written in Scala
- Considered as a third-generation distributed data processing system
  - Hadoop is considered as a second generation
- Why would we need a new generation?
  - Limitations of MapReduce (Hadoop)
    - Materialized intermediate results
    - Abstraction of Map and Reduce Applications
    - Can only be written in Java
    - Poor support for real-time data processing
  - Exploit advancements in hardware
    - Memory is much cheaper
    - Multi-core is now a commodity

10x faster than Hadoop on dis
100x faster than Hadoop in mem

2-5x less code

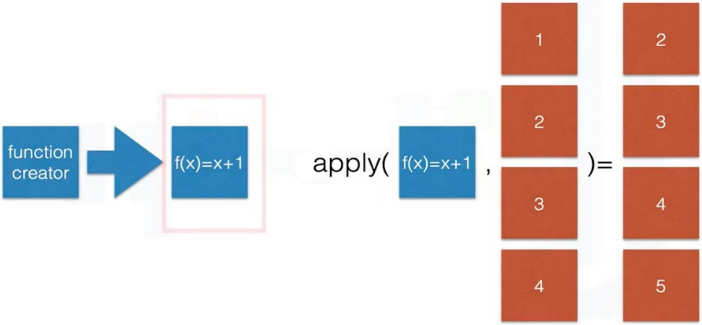
[1]Zaharia, Matei, et al. "Spark: Cluster computing with working sets

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

Functional Programming

- Mathematical **function** programming style
- Follows a declarative programming model
- Empathizes **what** instead of **how to**
- Uses **expressions** instead of **statements**



Functional Programming

Scala code defines a lambda expression with two parameters x and y of type Int. The lambda expression is stored in the variable add.

Python code defines a lambda function with two parameters x and y. The lambda function is assigned to the variable add.

Scala

```
// an example lambda
operator in Scala to add
2 numbers
val add = (x:Int, y:Int)
=> x + y
println(add(1,2))
```

Python

```
// an example lambda
function in Python to add
2 numbers
add = lambda x, y : x + y
print(add(1,2))
```

Functional Programming – Passing Operations

Scala

```scala
// Define the function
def performOperation(x: Int, y: Int, operation: (Int, Int) => Int): Int = { operation(x, y) }

// Call the function with a lambda
val result = performOperation(5, 3, (a, b) => a * b)
println(result) // Output: 15
```

Python

```python
# Define the function
def perform_operation(x, y, operation): return operation(x, y)

# Call the function with a lambda
result = perform_operation(5, 3, lambda a, b: a * b)
print(result)  # Output: 15
```

Functional Programming – using Arrays

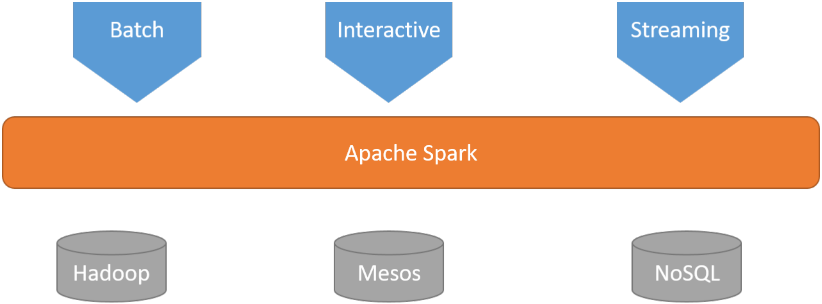
Scala

```scala
val add = (x: Int, y: Int) => x + y
val numbers = List((1,2), (3,4), (5,6))
val sums = numbers.map (pair => add(pair._1, pair._2))
println(sums)    // Prints: List(3, 7, 11)
```

Python

```python
add = lambda x, y: x + y
numbers = [(1,2), (3,4), (5,6)]
sums = list(map(lambda pair: add(pair[0], pair[1]), numbers))
print(sums)    # Prints: [3, 7, 11]
```

Unified Platform for Big Data Processing



Why Unification?

- Good for developers: One platform to learn
- Good for users: Take apps everywhere
- Good for distributions: More applications
- Is based on a common abstraction

Spark Abstractions

- 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

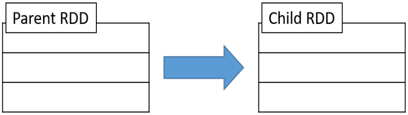
Word count In Spark

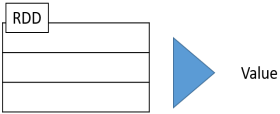
- Recall how we defined the code for word count in MapReduce?
- How does it look in Spark?

```
sc = SparkContext(appName="PythonWordCount")

lines = sc.textFile(sys.argv[1])

counts = (lines.flatMap(lambda x: x.split(' '))  # Split lines into words
            .map(lambda x: (x, 1))          # Map each word to a tuple (word, 1)
            .reduceByKey(lambda a, b: a+b)) # Reduce by key, sum occurrences

output = counts.collect()

for (word, count) in output:
   print("%s: %i" % (word, count))
sc.stop()
```

RDD Operations

- Two main types of RDD operations
  - Transformations: result in a new RDD
    - Can be chained
    - Forked
    - joined
  - Actions: return values, no more RDDs
    - One action at the end of each transformation chain





RDD Transformations

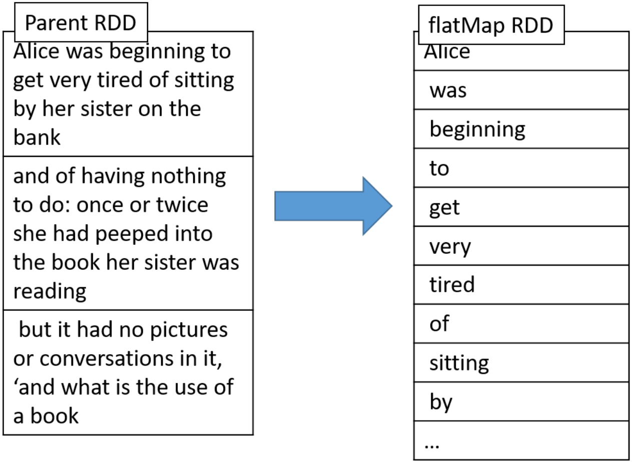
- Transformations create a new RDD from an existing one
- RDDs are immutable
  - Apply a series of transformations to modify the data as needed
- Common transformations
  - Map(function): 1-to-1 mapping
  - FlatMap(function): 1-to-Many mapping
  - Filter(function): 1-to-1 mapping with selectivity
- Special transformations
  - ReduceByKey
  - GroupByKey

Map vs. FlatMap

Aspect	Map	FlatMap
Functionality	Transforms each element of a collection independently.	Transforms each element and then flattens the result.
Input Example	"Hello world", "This is a test"	"Hello world", "This is a test"
Operation	Splits each line into words.	Splits each line into words and merges them into a single collection.
Output	[["Hello", "world"], ["This", "is", "a", "test"]]	["Hello", "world", "This", "is", "a", "test"]
Structure	Collection of collections (array of arrays).	Single flat collection (array).

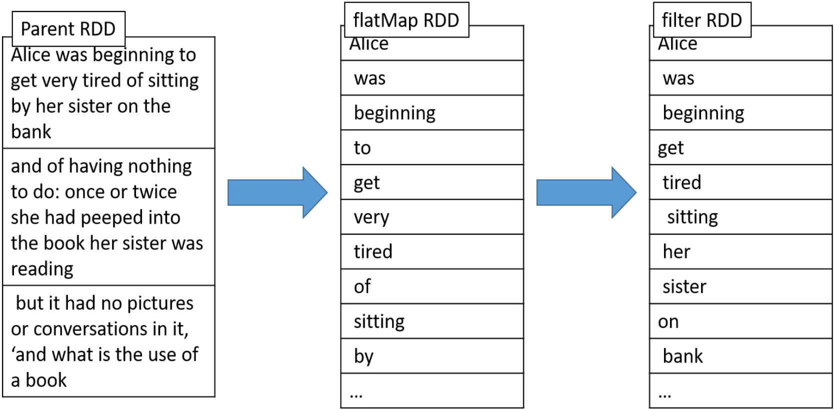
Transformation: flatMap

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



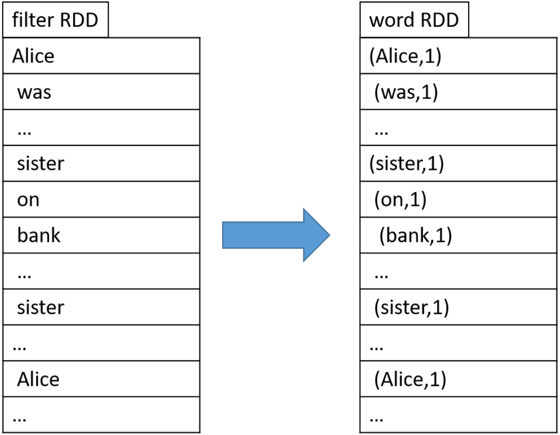
Transformation: filter

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



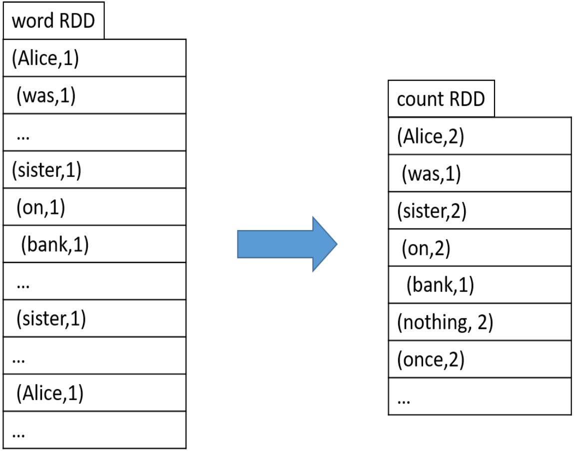
Transformation: map

```
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( )
```

count RDD
(Alice,2)
(was,1)
(sister,2)
(on,2)
(bank,1)
(nothing, 2)
(once,2)
…

output=
[(Alice,2), (was,1), (sister,2), (on,2), (bank,1), (nothing,2), (once,2),…]

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

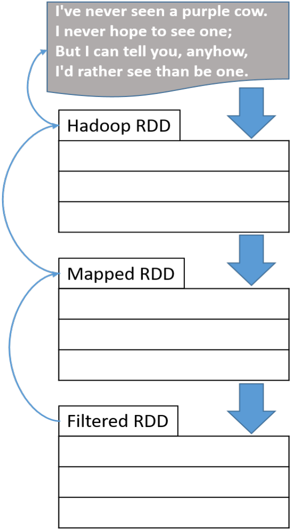
Application

```
 mydata_filt = sc.textFile('file.txt')
.map(lambda line: line.upper())
.filter(lambda line: line.startswith('I'))
```

Lineage

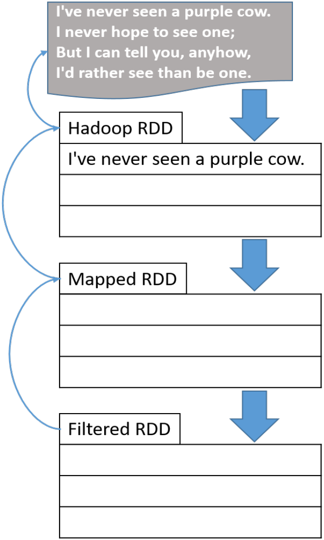
```
mydata_filt.toDebudString()
```

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



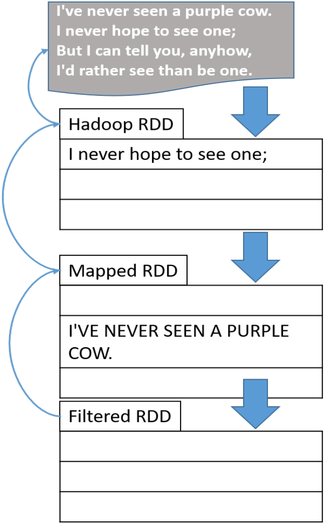
Pipelining

- 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



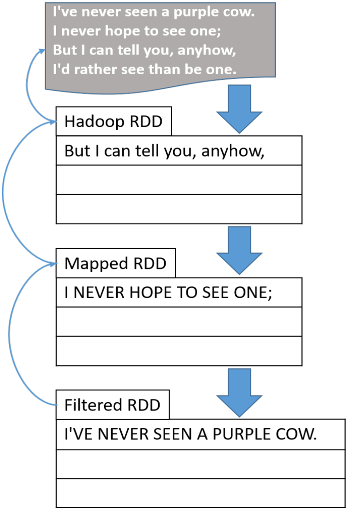
Pipelining

- 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



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
  - From files, example from text files as we saw earlier

Creating RDDs from Collections

- 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

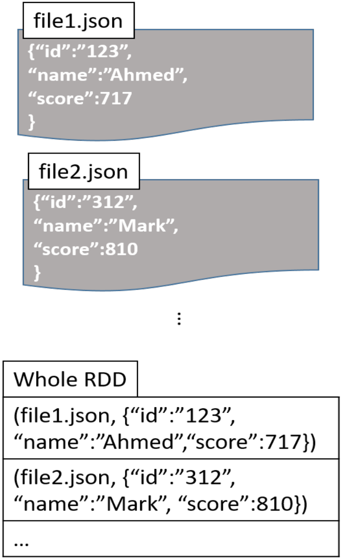
- Define input format using sc.hadoopFile
  - Or newAPIhadoopFile for New API classes
- Define output format using rdd.saveAsHadoopFile
  - Or saveAsNewAPIhadoopFile for New API classes

Example:

```
input_rdd = sc.newAPIHadoopFile(
"path/to/textfile.txt",
"org.apache.hadoop.mapreduce.lib.input.TextInputFormat",
"org.apache.hadoop.io.LongWritable",
"org.apache.hadoop.io.Text" )
```

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,
    - recall map step of word count example
    - 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

[1, 2, 3] zip ['a', 'b', 'c'] → [(1, 'a'), (2, 'b'), (3, 'c')]

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.

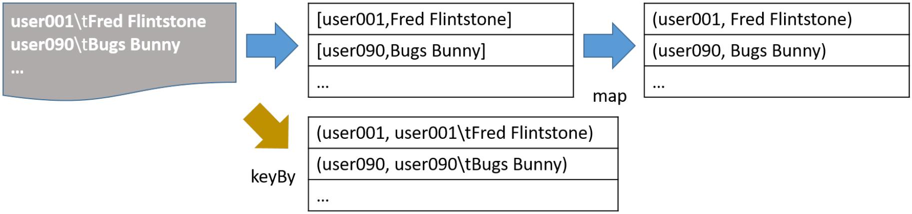
Creating Pair RDDs

- 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

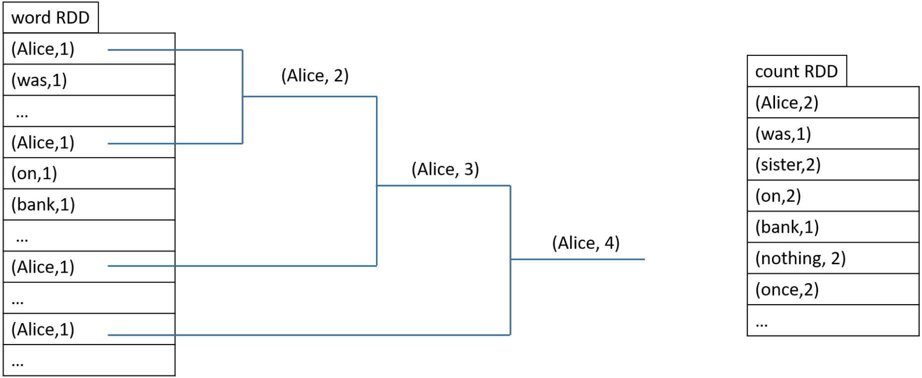
- Create a pair RDD from a tab-delimited file

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



Transformation: reduceByKey

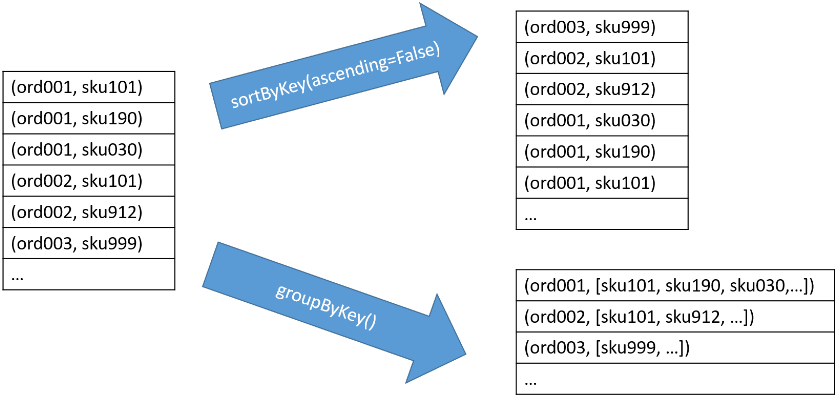
```
counts = word.reduceByKey(lambda x, y: x+y)
```



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

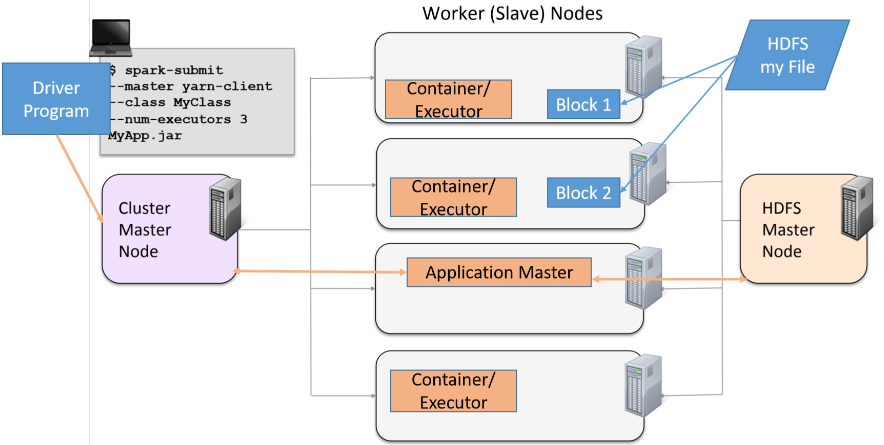
Orders = orderItems.join(orderTotals)

orderItems RDD
(ord001, [sku101, sku190, sku030])
(ord002, [sku101, sku912])
(ord003, [sku999])
…

orderTotals RDD
(ord001, $1200)
(ord002, $527)
(ord003, $3241)
…

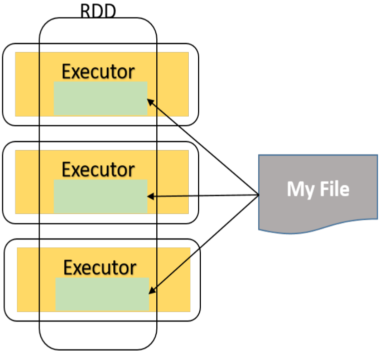
(ord001, ([sku101, sku190, sku030], $1200))
(ord002, ([sku101, sku912], $527))
(ord003, ([sku999],$3241))
…

Running A Spark Job on YARN



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

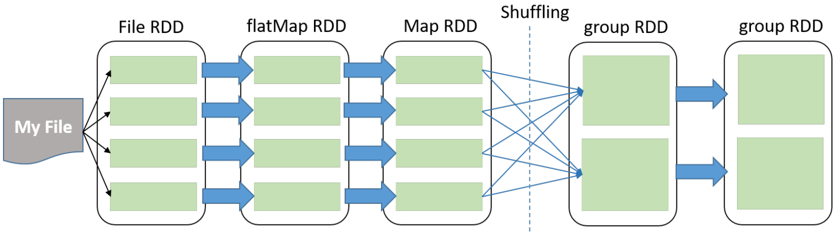
- All operations that can work on the same data partition are grouped into a stage.
  - Tasks within a stage are pipelined together
- Spark divides the DAG of the job into stages
- How Spark Calculates Stages? Based on RDD dependencies
  - Narrow dependencies
    - Only one child depends on the RDD
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

*DAG = Directed Acyclic Graph

Example: Average Word Length By First Letter

We have the following chain of operations

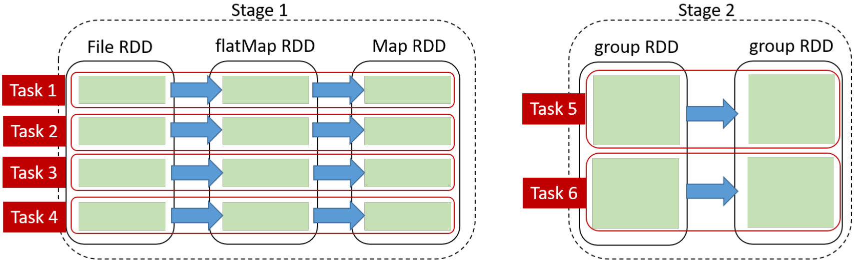
```
Avglength = sc.textFile(file).flatMap(line: line.split())
.map(word:(word[0], len (word)).groupByKey()
.map((k , values) : (k, sum(values)/ len(values)))
```



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