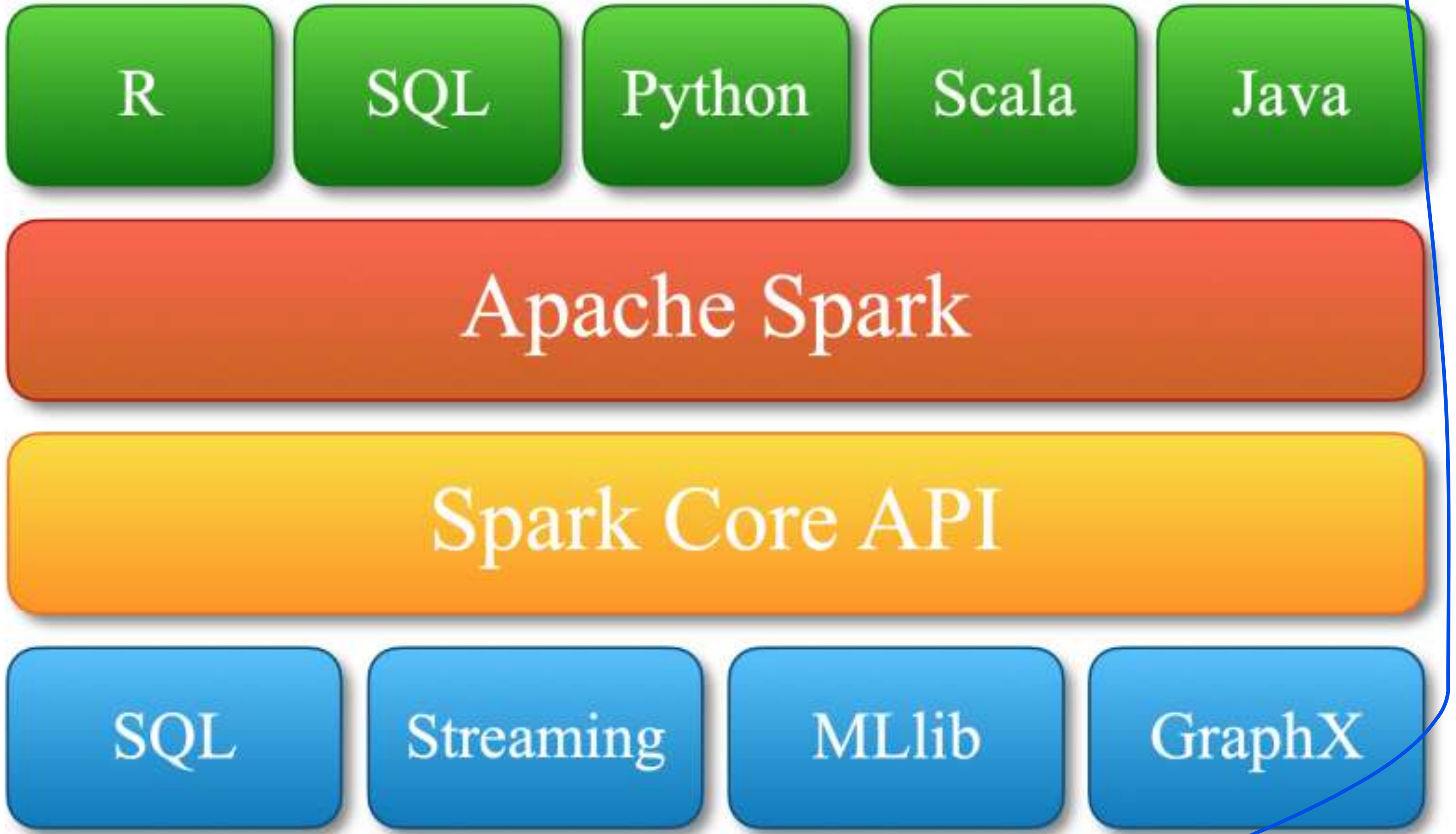
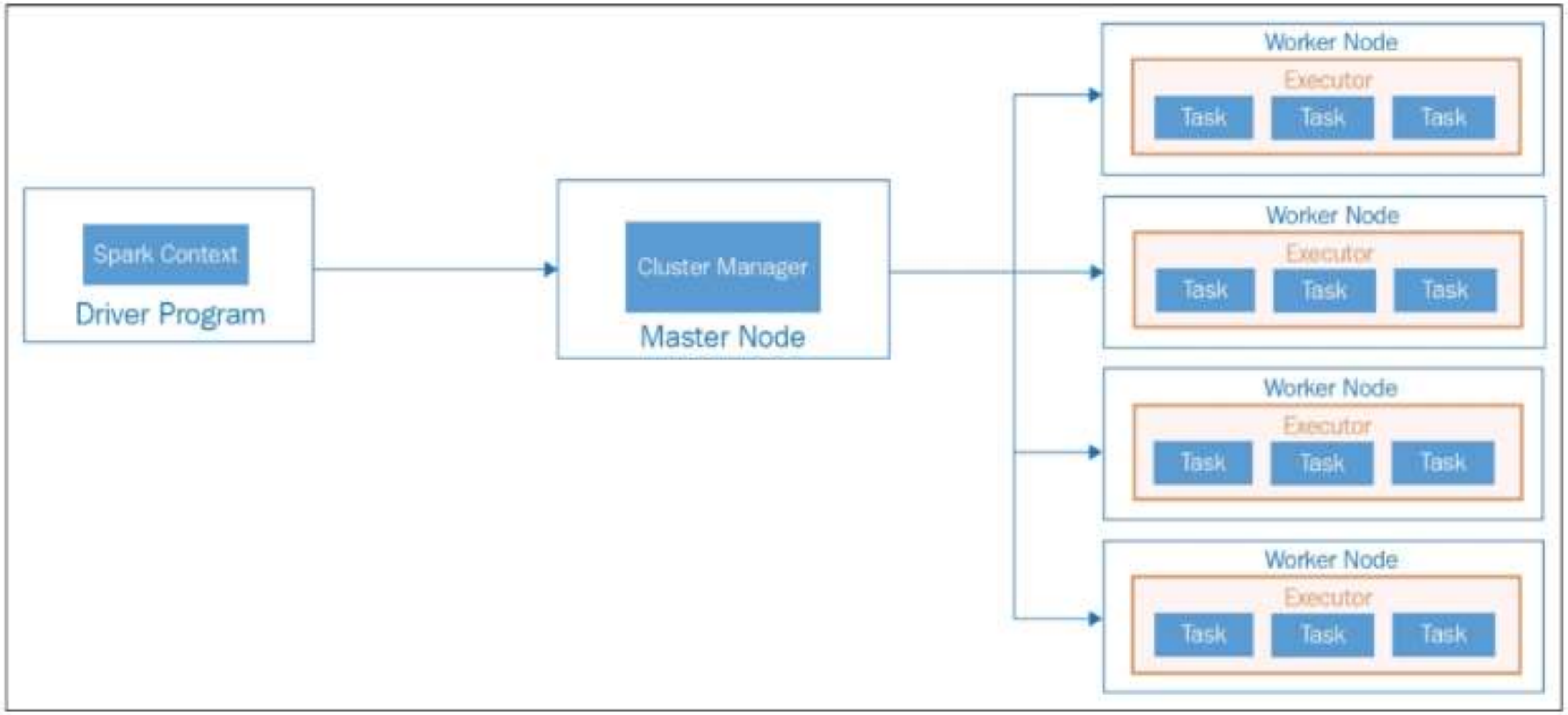
Spark ecosystem



Advantages of spark

- Spark is a general-purpose, **in-memory**, fault-tolerant, **distributed processing** engine that allows you to process data efficiently in a distributed fashion.
- Applications running on Spark are **100x** faster than traditional systems.
- Using Spark we can process data from Hadoop **HDFS**, **AWS S3**, **Databricks DBFS**, **Azure Blob Storage,** and many file systems.
- Spark also is used to process real-time data using Streaming and Kafka.
- Using Spark Streaming you can also stream files from the file system and also stream from the socket.
- Spark natively has machine learning and **graph libraries**.
- Provides connectors to store the data in NoSQL databases like MongoDB.

Spark architecture



- Spark applications run as independent sets of processes on a cluster, coordinated by the SparkContext object in your main program (called the driver program).

- Specifically, to run on a cluster, the SparkContext can connect to several types of cluster managers (either Spark's own standalone cluster manager, Mesos, YARN or Kubernetes), which allocate resources across applications.

- Once connected, Spark acquires executors on nodes in the cluster, which are processes that run computations and store data for your application. Next, it sends your application code (defined by JAR or Python files passed to SparkContext) to the executors. Finally, SparkContext sends tasks to the executors to run.

There are several useful things to note about this architecture:

- Each application gets its own executor processes, which stay up for the duration of the whole application and run tasks in multiple threads. This has the benefit of isolating applications from each other, on both the scheduling side (each driver schedules its own tasks) and executor side (tasks from different applications run in different JVMs). However, it also means that data cannot be shared across different Spark applications (instances of SparkContext) without writing it to an external storage system.
- Spark is agnostic to the underlying cluster manager. As long as it can acquire executor processes, and these communicate with each other, it is relatively easy to run it even on a cluster manager that also supports other applications (e.g.Mesos/YARN/Kubernetes).
- The driver program must listen for and accept incoming connections from its executors throughout its lifetime (e.g., see spark.driver.port in the network config section). As such, the driver program must be network addressable from the worker nodes.
- Because the driver schedules tasks on the cluster, it should be run close to the worker nodes, preferably on the same local area network. If you'd like to send requests to the cluster remotely, it's better to open an RPC to the driver and have it submit operations from nearby than to run a driver far away from the worker nodes.

Spark RDD

- An RDD is an immutable distributed collection of objects.
- Each RDD is split into multiple partitions, which may be computed on different nodes of the cluster.
- RDDs can contain any type of Python, Java, or Scala objects, including user-defined classes.
- Users create RDDs in two ways:
  - by loading an external dataset, or
  - by distributing a collection of objects (e.g., a list or set) in their driver program.
- Example 3-1. Creating an RDD of strings with textFile() in Python
- `>>> lines = sc.textFile("README.md")`

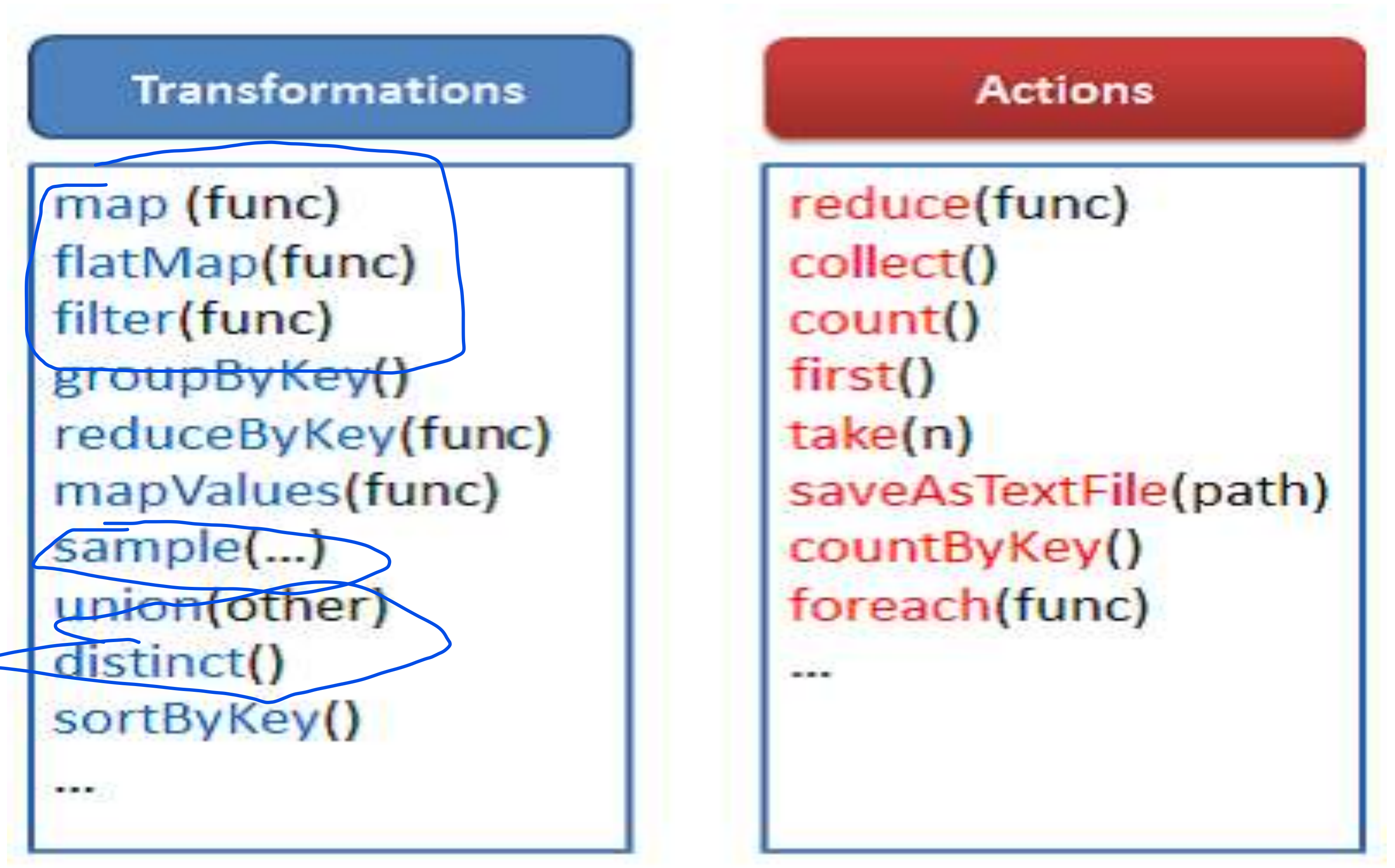
- Once created, RDDs offer two types of operations: transformations and actions. Transformations construct a new RDD from a previous one.
- For example, one common transformation is filtering data that matches a predicate.
- In our text file example, we can use this to create a new RDD holding just the strings that contain the word Python, as shown in Example 3-2.

Example 3-2. Calling the filter() transformation

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

- Actions, on the other hand, 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).
- One example of an action we called earlier is first(), which returns the first element in an RDD and is demonstrated in Example 3-3.
- Example 3-3. Calling the first() action

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



Transformations

map (func)
flatMap(func)
filter(func)
groupByKey()
reduceByKey(func)
mapValues(func)
sample(...)
union(other)
distinct()
sortByKey()
...

Actions

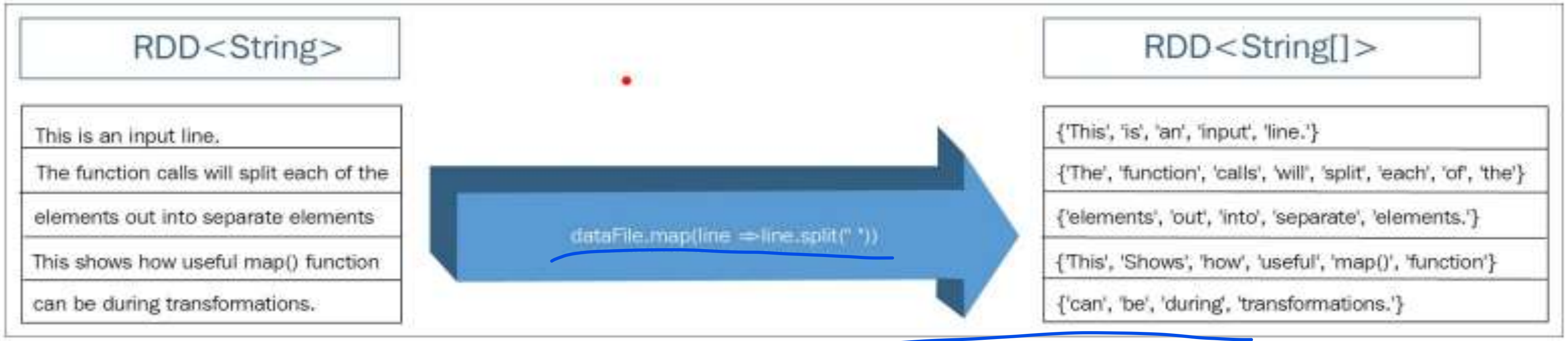
reduce(func)
collect()
count()
first()
take(n)
saveAsTextFile(path)
countByKey()
foreach(func)
...

Transformations:

Map(func)

- The map transformation is the most commonly used and the simplest of transformations on an RDD.
- The map transformation applies the function passed in the arguments to each of the elements of the source RDD.

Example: val dataFile = sc.textFile("README.md")



RDD<String>
This is an input line.
The function calls will split each of the
elements out into separate elements
This shows how useful map() function
can be during transformations.

```
dataFile.map(line =>line.split(" "))
```

RDD<String[]>
{'This', 'is', 'an', 'input', 'line.'}
{'The', 'function', 'calls', 'will', 'split', 'each', 'of', 'the'}
{'elements', 'out', 'into', 'separate', 'elements.'}
{'This', 'Shows', 'how', 'useful', 'map()', 'function'}
{'can', 'be', 'during', 'transformations.'}

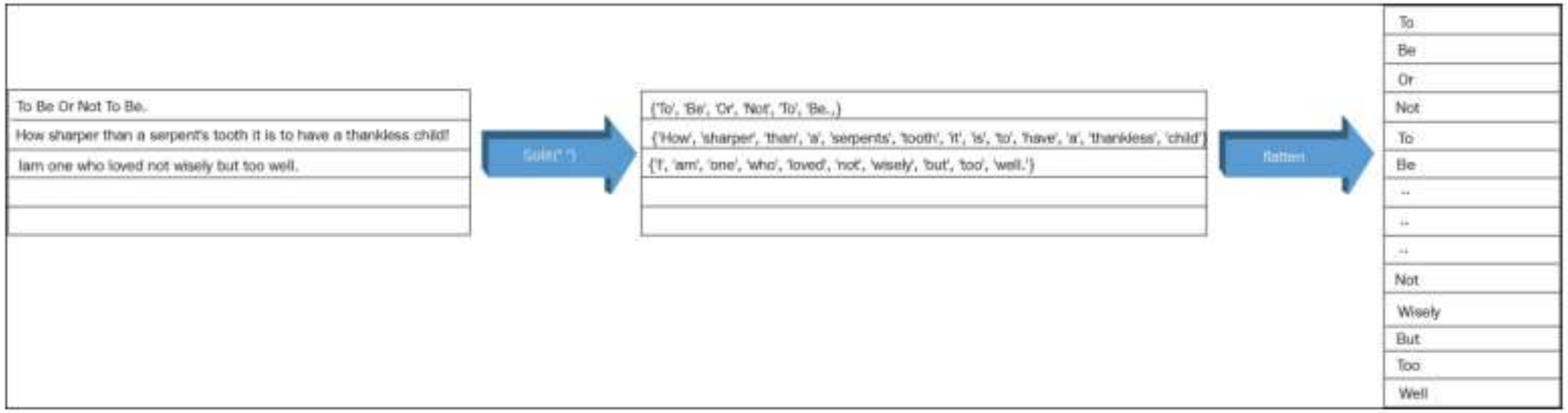
Filter(func)

- Filter, as the name implies, filters the input RDD, and creates a new dataset that satisfies the predicate passed as arguments.
- Python filtering example:

```
 dataFile = sc.textFile("README.md")
linesWithApache = datafile. Filter(lambda line: "Apache" in line)
```

flatMap(func)

- The flatMap() transformation is similar to map, but it offers a bit more flexibility.
- From the perspective of similarity to a map function, it operates on all the elements of the RDD, but the flexibility stems from its ability to handle functions that return a sequence rather than a single item.


To Be Or Not To Be.
How sharper than a serpent's tooth it is to have a thankless child!
Iam one who loved not wisely but too well.
Split(" ")
{'To', 'Be', 'Or', 'Not', 'To', 'Be.,}
{'How', 'sharper', 'than', 'a', 'serpents', 'tooth', 'it', 'is', 'to', 'have', 'a', 'thankless', 'child'}
{'I', 'am', 'one', 'who', 'loved', 'not', 'wisely', 'but', 'too', 'well.'}
flatten
To
Be
Or
Not
To
Be
..
..
..
Not
Wisely
But
Too
Well

```
movies = sc.parallelize(["Pulp Fiction","Requiem for a dream","A clockwork Orange"])
movies.flatMap(lambda movieTitle: movieTitle.split(" ")).collect()
```

Operation	**Output**	**Explanation**
flatMap	['Pulp', 'Fiction', 'Requiem', 'for', 'a', 'dream', 'A', 'clockwork', 'Orange']	Flattens the result into a single list of words.
map	[['Pulp', 'Fiction'], ['Requiem', 'for', 'a', 'dream'], ['A', 'clockwork', 'Orange']]	Creates a list of lists, one for each movie title.

Sample (withReplacement, fraction, seed)

- Spark provides an easy way to sample RDD's for your calculations, if you would prefer to quickly test your hypothesis on a subset of data before running it on a full dataset.
- overview of the parameters that are passed onto the method:
  - withReplacement: Is a Boolean (True/False), and it indicates if elements can be sampled multiple times. Sampling with replacement means that the two sample values are independent. In practical terms this means that, if we draw two samples with replacement, what we get on the first one doesn't affect what we get on the second draw, and hence the covariance between the two samples is zero

    - If we are sampling without replacement, the two samples aren't independent. Practically, this means what we got on the first draw affects what we get on the second one and hence the covariance between the two isn't zero.

- fraction: Fraction indicates the expected size of the sample as a fraction of the RDD's size. The fraction must be between 0 and 1. For example, if you want to draw a 5% sample, you can choose 0.05 as a fraction.
- seed: The seed used for the random number generator.
- The sample() example in Python:

```
data =
sc.parallelize([1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19,20])
data.sample(1,0.1,12345).collect()
```

Set operations in Spark

- Distinct: Distinct operation provides you a non-duplicated set of data from the dataset
- Intersection: The intersection operations returns only those elements that are available in both datasets
- Union: A union operation returns the elements from both datasets
- Subtract: A subtract operation returns the elements from one dataset by taking away all the matching elements from the second dataset
- Cartesian: A Cartesian product of both datasets

Distinct in Python Example:

movieList = sc.parallelize(["A Nous Liberte","Airplane","The Apartment","The Apartment"])

movieList.distinct().collect()

Intersection()

Intersection looks at elements from both RDDs and returns the elements that are available across both data sets.

For example, you might have candidates based on skillset:

java_skills = "Tom Mahoney","Alicia Whitekar","Paul Jones","Rodney Marsh"

db_skills = "James Kent", "Paul Jones", Tom Mahoney", "Adam Waugh"

java_and_db_skills = java_skills.intersection(db_skills)

- Union()

Union is basically an aggregation of both the datasets. If few data elements are available across both datasets, they will be duplicated.

Example Union in Scala:

```
val java_skills=sc.parallelize(List("Tom Mahoney","Alicia
Whitekar","Paul Jones","Rodney Marsh"))
val db_skills= sc.parallelize(List("James Kent","Paul
Jones","Tom Mahoney","Adam Waugh"))
java_skills.union(db_skills).collect()
//The Result shown would be like:
Tom Mahoney, Alicia Whitekar, Paul Jones, Rodney Marsh, James
Kent, Paul Jones, Tom Mahoney, Adam Waugh
```

- Subtract()

Subtraction as the name indicates, removes the elements of one dataset from the other.

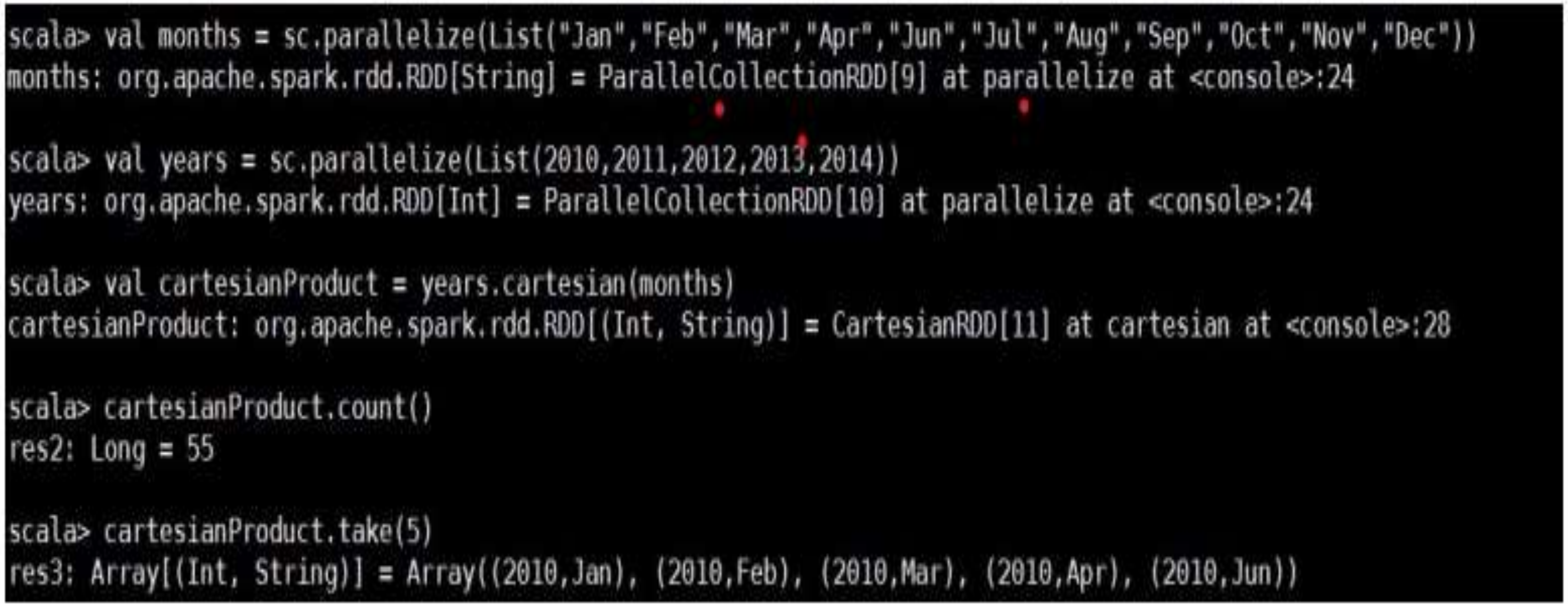
Example : The subtract() in Scala:

```
val java_skills=sc.parallelize(List("Tom Mahoney","Alicia
Whitekar","Paul Jones","Rodney Marsh"))
val db_skills= sc.parallelize(List("James Kent","Paul
Jones","Tom Mahoney","Adam Waugh"))
java_skills.subtract(db_skills).collect()
```

output: Alicia Whitekar, Rodney Marsh

Cartesian()

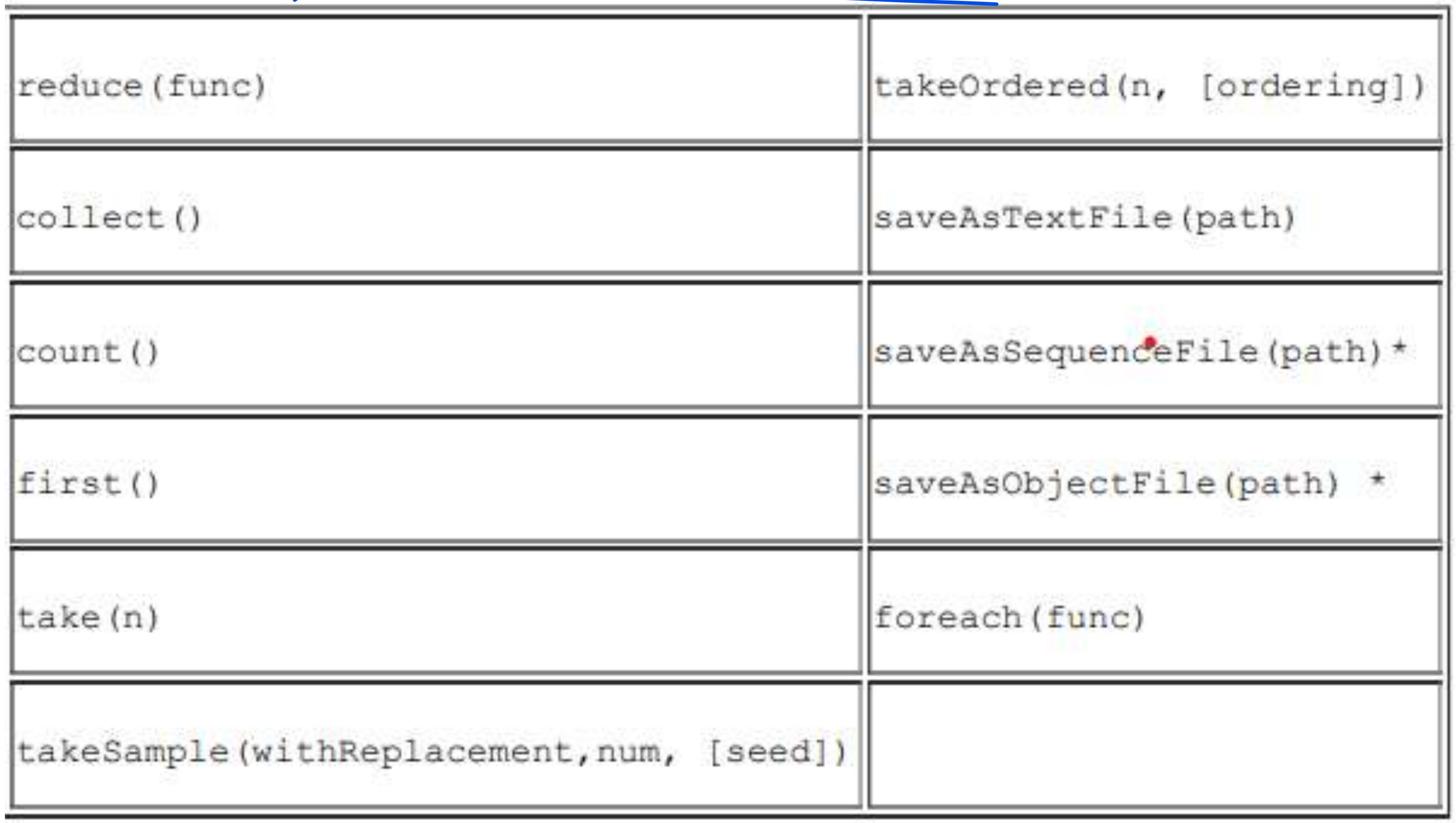
- Cartesian simulates the cross-join from an SQL system, and basically gives you all the possible combinations between the elements of the two datasets



```
scala> val months = sc.parallelize(List("Jan","Feb","Mar","Apr","Jun","Jul","Aug","Sep","Oct","Nov","Dec"))
months: org.apache.spark.rdd.RDD[String] = ParallelCollectionRDD[9] at parallelize at <console>:24

scala> val years = sc.parallelize(List(2010,2011,2012,2013,2014))
years: org.apache.spark.rdd.RDD[Int] = ParallelCollectionRDD[10] at parallelize at <console>:24

scala> val cartesianProduct = years.cartesian(months)
cartesianProduct: org.apache.spark.rdd.RDD[(Int, String)] = CartesianRDD[11] at cartesian at <console>:28

scala> cartesianProduct.count()
res2: Long = 55

scala> cartesianProduct.take(5)
res3: Array[(Int, String)] = Array((2010,Jan), (2010,Feb), (2010,Mar), (2010,Apr), (2010,Jun))
```

Actions

Actions are what makes Spark perform the actual computation from the graph that the framework has been building in the background while you were busy performing transformations on it.



reduce(func)	takeOrdered(n, [ordering])
collect()	saveAsTextFile(path)
count()	saveAsSequenceFile(path) *
first()	saveAsObjectFile(path) *
take(n)	foreach(func)
takeSample(withReplacement,num, [seed])	

Reduce is an aggregation of elements using a function.

The syntax of RDD reduce() method is:

```
RDD.reduce(<function>)
```

Following are the two important properties that an aggregation function should have:

- **Commutative** A+B = B+A – ensuring that the result would be independent of the order of elements in the RDD being aggregated.
- **Associative** (A+B)+C = A+(B+C) – ensuring that any two elements associated in the aggregation at a time does not effect the final result.
- Examples of such function are Addition, Multiplication, OR, AND, XOR, XAND

Example:

```
numbers = sc.parallelize([1,7,8,9,5,77,48])
  # aggregate RDD elements using addition function
  sum = numbers.reduce(lambda a,b:a+b)
  print "sum is : " + str(sum)
```

Output:

```
Sum is :155
```

Collect()

Collect will return the contents of the RDD upon which it is called, back to the driver program. Typically this is a subset of the input data that you have transformed and filtered by applying Spark's list of transformations, for example, Map() and Filter().

Count()

count() will return the total number of elements in the RDD.

For example, in case of loading a file from a filesystem, the count() function returned the total number of lines in the file.

- Take(n) Take is a very useful function if you want to have a peek at the resultant dataset. This function will fetch the first n number of elements from the RDD.

To display the content of RDD

- data = [1, 2, 3, 4, 5] rdd = sc.parallelize(data) # Display RDD content
- print("Collect all elements:", rdd.collect()) # Output: [1, 2, 3, 4, 5]
- print("First 3 elements:", rdd.take(3)) # Output: [1, 2, 3]
- print("Lineage of the RDD:", rdd.toDebugString())

Lazy evaluation in spark

- Lazy Evaluation in Sparks means Spark will not start the execution of the process until an ACTION is called.
- Until we are doing only transformations on the dataframe /dataset /RDD, Spark is the least concerned. Once Spark sees an ACTION being called, it starts looking at all the transformations and creates a DAG.
- DAG Simply sequence of operations that need to be performed in a process to get the resultant output.
- If Spark could wait until an Action is called, it may merge some transformation or skip some unnecessary transformation and prepare a perfect execution plan.

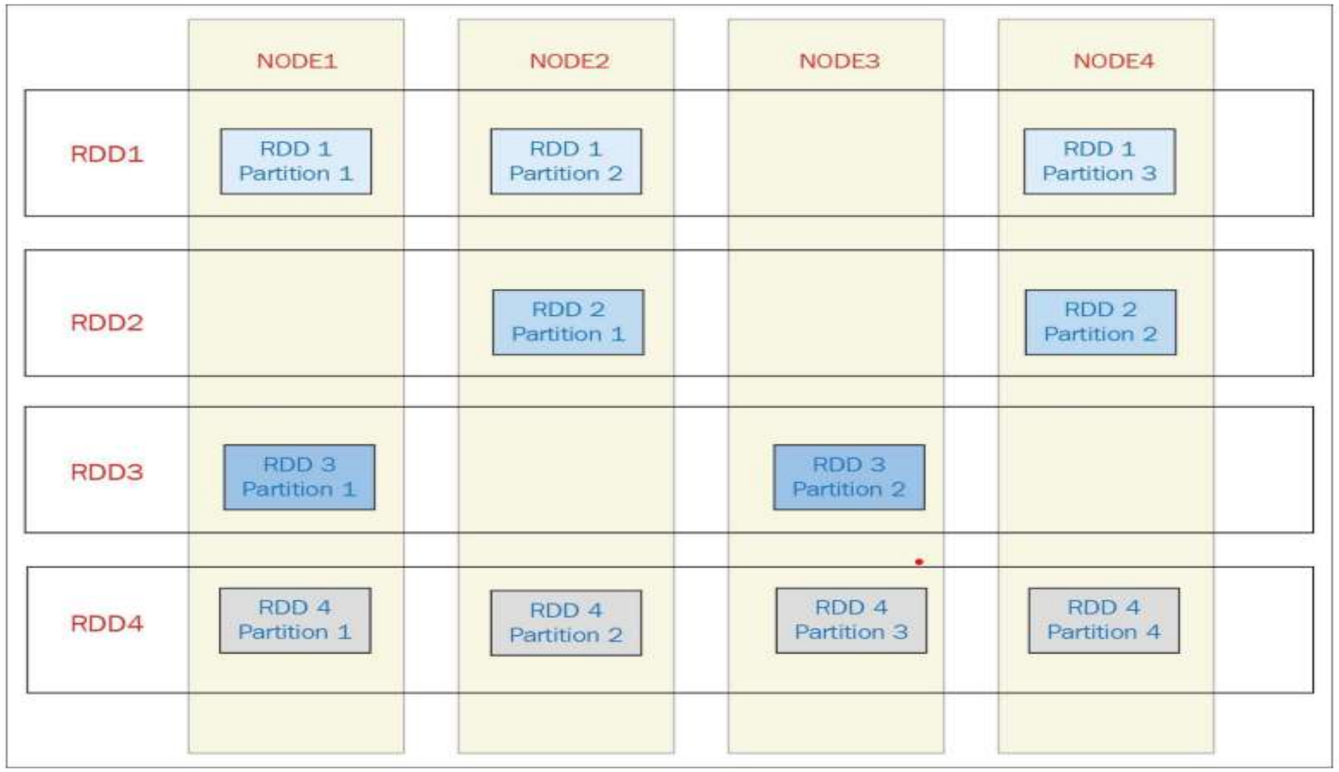


Figure 2.1: RDD split across a cluster

Persistence (Caching)

- Spark RDDs are lazily evaluated, and sometimes we may wish to use the same RDD multiple times.
- Spark will recompute the RDD and all of its dependencies each time we call an action on the RDD.
- This can be especially expensive for iterative algorithms, which look at the data many times.

Example: Double execution in Scala

```
val result = input.map(x => x*x)
println(result.count())
println(result.collect().mkString(","))
```

- To avoid computing an RDD multiple times, we can ask Spark to persist the data.
- When we ask Spark to persist an RDD, the nodes that compute the RDD store their partitions.
- If a node that has data persisted on it fails, Spark will recompute 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 slowdown.
- Spark has many levels of persistence to choose from based on what our goals are, as you can see in Table below:

Storage Level	Description
MEMORY_ONLY	It stores the RDD as deserialized Java objects in the JVM. This is the default level. If the RDD doesn't fit in memory, some partitions will not be cached and recomputed each time they're needed.
MEMORY_AND_DISK	It stores the RDD as deserialized Java objects in the JVM. If the RDD doesn't fit in memory, store the partitions that don't fit on disk, and read them from there when they're needed.
MEMORY_ONLY_SER (Java and Scala)	It stores RDD as serialized Java objects (i.e. one-byte array per partition). This is generally more space-efficient than deserialized objects.
MEMORY_AND_DISK_SER (Java and Scala)	It is similar to MEMORY_ONLY_SER, but spill partitions that don't fit in memory to disk instead of recomputing them.
DISK_ONLY	It stores the RDD partitions only on disk.
MEMORY_ONLY_2, MEMORY_AND_DISK_2, etc.	It is the same as the levels above, but replicate each partition on two cluster nodes.
OFF_HEAP (experimental)	It is similar to MEMORY_ONLY_SER, but store the data in off-heap memory. The off-heap memory must be enabled.

Example :persist() in Scala

```
val result = input.map(x => x * x)
result.persist(StorageLevel.DISK_ONLY)
println(result.count())
println(result.collect().mkString(","))
```

Fault Tolerance in Spark

- A feature of self-recovery is one of the most powerful keys on spark platform. Which means at any stage of failure, RDD itself can recover the losses.
- If any of the nodes of processing data gets crashed, that results in a fault in a cluster. In other words, RDD is logically partitioned and each node is operating on a partition at any point in time.
- Operations which are being performed is a series of scala functions. Those operations are being executed on that partition of RDD. This series of operations are merged together and create a DAG, it refers to **Directed Acyclic Graph**. That means DAG keeps track of operations performed.
- If any node crashes in the middle of an operation, the cluster manager finds out that node. Then, it tries to assign another node to continue the processing at the same place.

Apache Spark Paired RDD

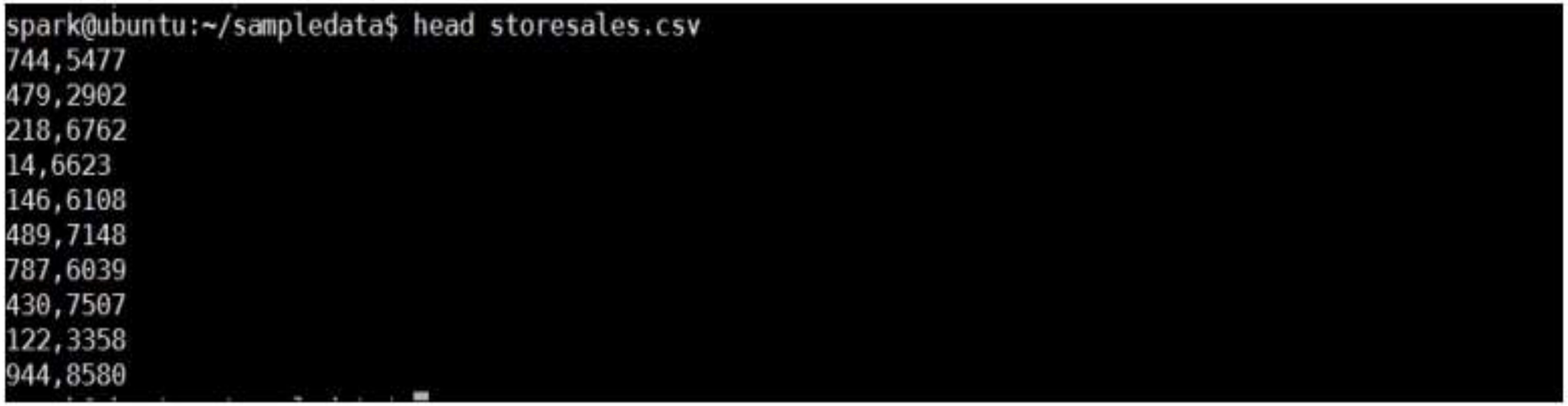
K - V

- In Apache Spark, *Key-value* pairs are known as **paired RDD.**
- There is two linked data item in a **key-value pair** (KVP).The key is the *identifier*, while the value is the *data corresponding* to the key value.
- The major advantage of key value pair is, we can operate on data belonging to a particular key in parallel, which includes operations such as aggregation or joining.

- Converting a data set into key/value pairs requires careful consideration. The most important of those considerations include answering the following key questions:
  - What should be the key?
  - What should be the value?
- The answer to this lies in the business problem that you are trying to solve.
- Example:

If you had a text file with StoreId (1000 different stores) and sales per month (000's of dollars), the key would be the StoreId, and the value would be the sales.

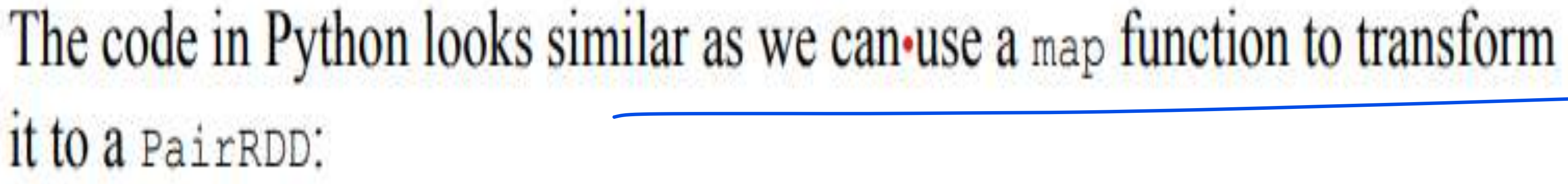
Example: consider a file storesales.csv



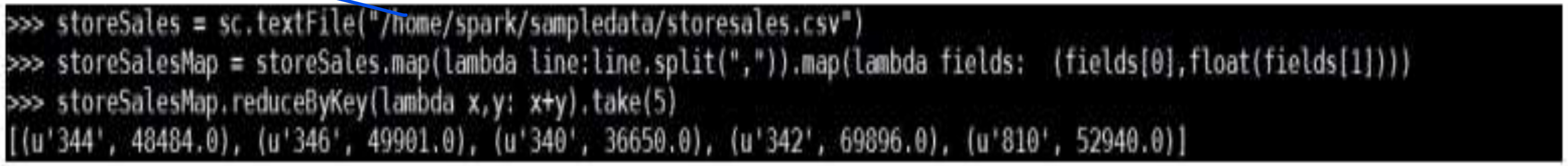
```
spark@ubuntu:~/sampledata$ head storesales.csv
744,5477
479,2902
218,6762
14,6623
146,6108
489,7148
787,6039
430,7507
122,3358
944,8580
```

Figure 2.24: File structure of storesales.csv

- The structure of the file indicates that we have a three digit StoreID, followed by a comma and the value of store sales.
- Our objective is to calculate the total store sales per store, to understand how each store is doing.
- The file can be converted to pairRDD using map function.



The code in Python looks similar as we can use a `map` function to transform it to a `PairRDD`:



```
>>> storeSales = sc.textFile("/home/spark/sampledata/storesales.csv")
>>> storeSalesMap = storeSales.map(lambda line:line.split(",")).map(lambda fields:  (fields[0],float(fields[1])))
>>> storeSalesMap.reduceByKey(lambda x,y: x+y).take(5)
[(u'344', 48484.0), (u'346', 49901.0), (u'340', 36650.0), (u'342', 69896.0), (u'810', 52940.0)]
```

Spark Pair RDD Transformation Functions

PAIR RDD FUNCTIONS	FUNCTION DESCRIPTION
aggregateByKey	Aggregate the values of each key in a data set. This function can return a different result type then the values in input RDD.
combineByKey	Combines the elements for each key.
combineByKeyWithClassTag	Combines the elements for each key.
flatMapValues	It's flatten the values of each key with out changing key values and keeps the original RDD partition.
foldByKey	Merges the values of each key.
groupByKey	Returns the grouped RDD by grouping the values of each key.
mapValues	It applied a map function for each value in a pair RDD with out changing keys.
reduceByKey	Returns a merged RDD by merging the values of each key.
reduceByKeyLocally	Returns a merged RDD by merging the values of each key and final result will be sent to the master.
sampleByKey	Returns the subset of the RDD.
subtractByKey	Return an RDD with the pairs from this whose keys are not in other.
keys	Returns all keys of this RDD as a RDD[T].
values	Returns an RDD with just values.
partitionBy	Returns a new RDD after applying specified partitioner.
fullOuterJoin	Return RDD after applying fullOuterJoin on current and parameter RDD
join	Return RDD after applying join on current and parameter RDD
leftOuterJoin	Return RDD after applying leftOuterJoin on current and parameter RDD
rightOuterJoin	Return RDD after applying rightOuterJoin on current and parameter RDD

Spark Pair RDD Actions

PAIR RDD ACTION FUNCTIONS	FUNCTION DESCRIPTION
collectAsMap	Returns the pair RDD as a Map to the Spark Master.
countByKey	Returns the count of each key elements. This returns the final result to local Map which is your driver.
countByKeyApprox	Same as countByKey but returns the partial result. This takes a timeout as parameter to specify how long this function to run before returning.
lookup	Returns a list of values from RDD for a given input key.
reduceByKeyLocally	Returns a merged RDD by merging the values of each key and final result will be sent to the master.
saveAsHadoopDataset	Saves RDD to any hadoop supported file system (HDFS, S3, ElasticSearch, e.t.c), It uses Hadoop JobConf object to save.
saveAsHadoopFile	Saves RDD to any hadoop supported file system (HDFS, S3, ElasticSearch, e.t.c), It uses Hadoop OutputFormat class to save.
saveAsNewAPIHadoopDataset	Saves RDD to any hadoop supported file system (HDFS, S3, ElasticSearch, e.t.c) with new Hadoop API, It uses Hadoop Configuration object to save.
saveAsNewAPIHadoopFile	Saves RDD to any hadoop supported file system (HDFS, S3, ElasticSearch, e.t.c), It uses new Hadoop API OutputFormat class to save.

Example:

```
val rdd = spark.sparkContext.parallelize(List("Germany India USA","USA India
Russia","India Brazil Canada China")  )
 val wordsRdd = rdd.flatMap(_.split(" "))
 val pairRDD = wordsRdd.map(f=>(f,1))
 pairDD.foreach(println)
```

This snippet creates a pair RDD by splitting by space on every element in an RDD, flatten it to form a single word string on each element in RDD and finally assigns an integer "1" to every word

```
// Output:
(Germany,1)
(India,1)
(USA,1)
(USA,1)
(India,1)
(Russia,1)
(India,1)
(Brazil,1)
(Canada,1)
(China,1)
```

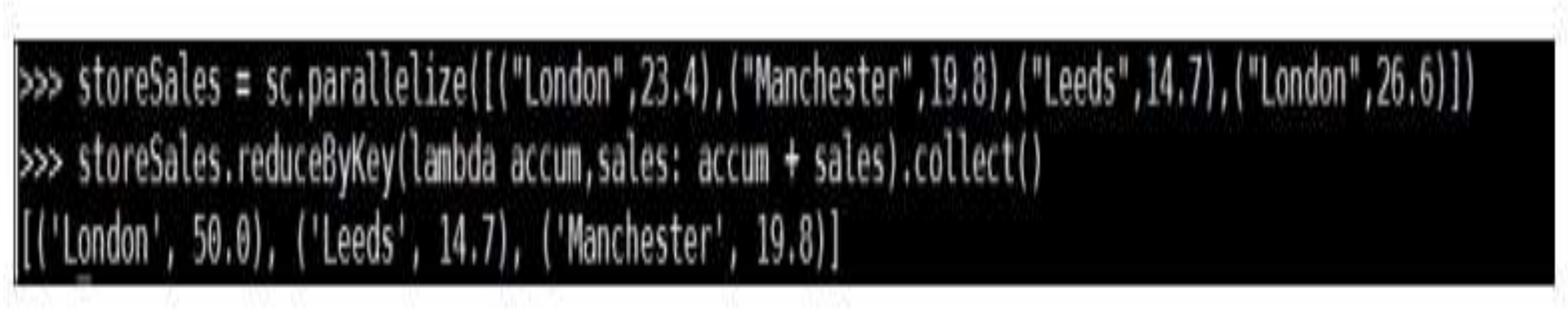
```
pairRDD.distinct().foreach(println)
```

Output:

```
// Output:
(Germany,1)
(India,1)
(Brazil,1)
(China,1)
(USA,1)
(Canada,1)
(Russia,1)
```

```
// SortByKey() on pairRDD
   println("Sort by Key ==>")
   val sortRDD = pairRDD.sortByKey()
   sortRDD.foreach(println)
Output:
// Output:
Sort by Key ==>
(Brazil,1)
(Canada,1)
(China,1)
(Germany,1)
(India,1)
(India,1)
(India,1)
(Russia,1)
(USA,1)
(USA,1)
```

Examples of reduceByKey(func) where it combines values for the same key, in our case, store sales based on the storeid(key).The function that has to be passed as an argument has to be an associative function, which applies to the source RDD and creates a new RDD with the resulting values.



```
>>> storeSales = sc.parallelize([("London",23.4),("Manchester",19.8),("Leeds",14.7),("London",26.6)])
>>> storeSales.reduceByKey(lambda accum,sales: accum + sales).collect()
[('London', 50.0), ('Leeds', 14.7), ('Manchester', 19.8)]
```

reduceByKey vs. groupByKey - Performance Implications

- reduceByKey() and groupByKey() can be used for similar results, but their execution plan is different and can have a huge impact on your performance.
- groupByKey() operation may be very expensive especially if you are grouping in order to perform an aggregation (such as a sum or average) over each key. In such a scenario, using aggregateByKey or reduceByKey() will provide much better performance.
- Key differences are:
- reduceByKey(): Offers a Map side combine, which means the amount of data shuffled will be lesser than the groupByKey().
- groupByKey(): Must be able to hold all key-value pairs for any key in memory. If a key has too many values (quite common with a popular e.g. stock trades), it can potentially result in an OutOfMemoryError.
- groupByKey() calls the combineByKey() with the mapSideCombine parameter as false, which results in exaggerated shuffling, thus impact performance.

Example 2.19: `groupByKey()` and `reduceByKey()` in Scala:

```
#Input Data
val storeSales = sc.parallelize(Array(("London", 23.4),
("Manchester",19.8),("Leeds",14.7),("London",26.6)))
```

```
#GroupByKey
storeSales.groupByKey().map(location=>
(location._1,location._2.sum)).collect()

#SampleResult
#res2: Array[(String, Double)] = Array((Manchester,19.8),
(London,50.0), (Leeds,14.7))

#ReduceByKey
storeSales.reduceByKey(_+_).collect()

#Sample Result
#res1: Array[(String, Double)] = Array((Manchester,19.8),
(London,50.0), (Leeds,14.7))
```

Shared variables

Spark, however, provides two types of shared variables:

- Broadcast variables - Read-only variables cached on each machine
- Accumulators - Variables that can be added through associative and commutative property

Broadcast variables

ROV

- Largescale data movement is often a major factor in negatively affecting performance in MPP environments.
- One of the ways to reduce data movement is to cache frequently accessed data objects on the machines, which is essentially what Spark's broadcast variables are about.
- Broadcast variables are set by the calling program/driver program and will be retrieved by the workers across the cluster.
- Since the objective is to share the data across the cluster, they are read-only after they have been set .
- The value of a broadcast variable is retrieved and stored only on the first read

Example: processing weblogs

where the weblogs contain only the pageId, whereas the page titles are stored in a lookup table.

During the analysis of the weblogs you might want to join the page Id from the weblog to the one in the lookup table to identify what particular page was being browsed, which page gets the most hits, which page loses the most customers, and so on.

This can be done using the web page lookup table being broadcasted across the cluster

Accumulator

- Accumulator is a shared variable that is used with RDD and DataFrame to perform sum and counter operations similar to Map-reduce counters.
- These variables are shared by all executors to update and add information through aggregation or computative operations.
- Accumulators are write-only and initialize once variables where only tasks that are running on workers are allowed to update and updates from the workers get propagated automatically to the driver program. But, only the driver program is allowed to access the Accumulator variable using the **value** property.

- **sparkContext.accumulator()** is used to define accumulator variables.
- **add()** function is used to add/update a value in accumulator
- **value** property on the accumulator variable is used to retrieve the value from the accumulator.
- We can create Accumulators in PySpark for primitive types int and float. Users can also create Accumulators for custom types using AccumulatorParam class of PySpark.

Example:

```
accum=spark.sparkContext.accumulator(0)
rdd=spark.sparkContext.parallelize([1,2,3,4,5])
rdd.foreach(lambda x:accum.add(x))
print(accum.value)
```

Example:2

```
accumCount=spark.sparkContext.accumulator(0)
rdd2=spark.sparkContext.parallelize([1,2,3,4,5])
rdd2.foreach(lambda x:accumCount.add(1))
Print(accumCount.value)
```

In summary, PySpark Accumulators are shared variables that can be updated by executors and propagate back to driver program. These variables are used to add sum or counts and final results can be accessed only by driver program

Example:Spark program to create an array of 10 elements and find the maximum element in an array and add the maximum element to all the elements and display

```
import org.apache.spark.{SparkConf, SparkContext}
object MaxElementAddition {
  def main(args: Array[String]): Unit = {
    // Create a SparkConf and SparkContext
    val conf = new SparkConf().setAppName("MaxElementAddition").setMaster("local")
    val sc = new SparkContext(conf)
    // Create an array of 10 elements
    val inputArray = Array(5, 8, 2, 15, 7, 10, 3, 12, 9, 6)
    // Parallelize the array into an RDD
    val inputRDD = sc.parallelize(inputArray)
 // Find the maximum element in the array
    val maxElement = inputRDD.max()
    // Add the maximum element to all elements in the array
    val resultRDD = inputRDD.map(element => element + maxElement)
    // Display the result
    println("Input Array: " + inputArray.mkString(", "))
    println("Max Element: " + maxElement)
    println("Result Array: " + resultRDD.collect().mkString(", "))
    // Stop the SparkContext
    sc.stop()  }
}
```