Distributed System Lab 2: An introduction to Spark core & RDD-based programming

Center of Computer Engineering (CCE)
HPC Lab - Faculty of Computer Science and Engineering
HCMC University of Technology

spark-submit

What is spark-submit

- Spark programs are executed (submitted) by using the **spark-submit** command
    - It is a command line program
    - It is characterized by a set of parameters
        - E.g., the name of the jar file containing all the classes of the Spark application we want to execute
        - The name of the Driver class
        - The parameters of the Spark application
        - etc.

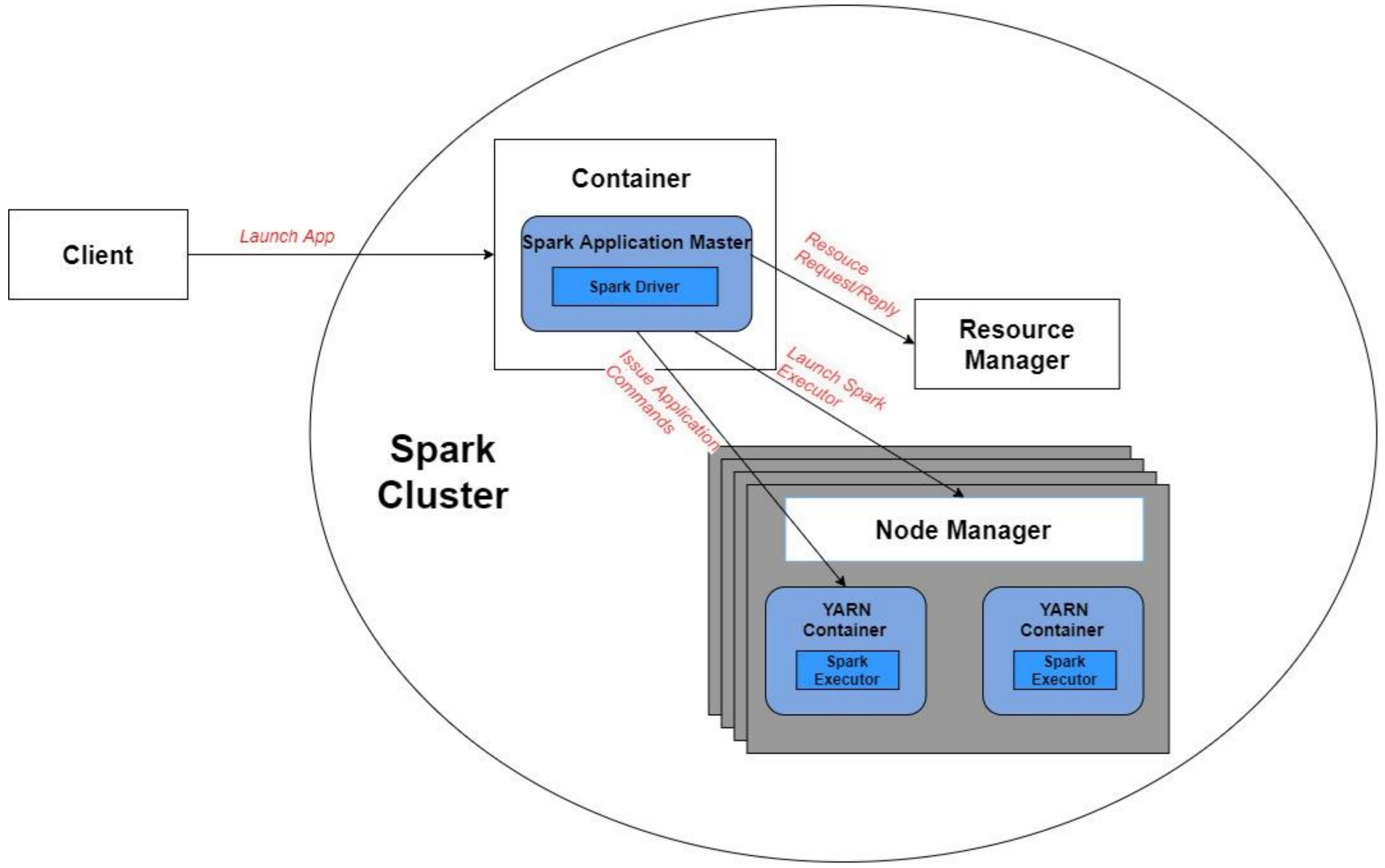
How to use it

- Spark programs are executed (submitted) by using the **spark-submit** command
    - It is a command line program
    - It is characterized by a set of parameters
        - E.g., the name of the jar file containing all the classes of the Spark application we want to execute
        - The name of the Driver class
        - The parameters of the Spark application
        - etc.
- It has two important parameters that are used to specify where the application is executed
    - **--master**: Specify which environment/scheduler is used to execute the application (local, standalone, or another node manager, such as Mesos, YARN, etc.)
    - **--deploy-mode**: Specify where the Driver is launched/executed (client or cluster)

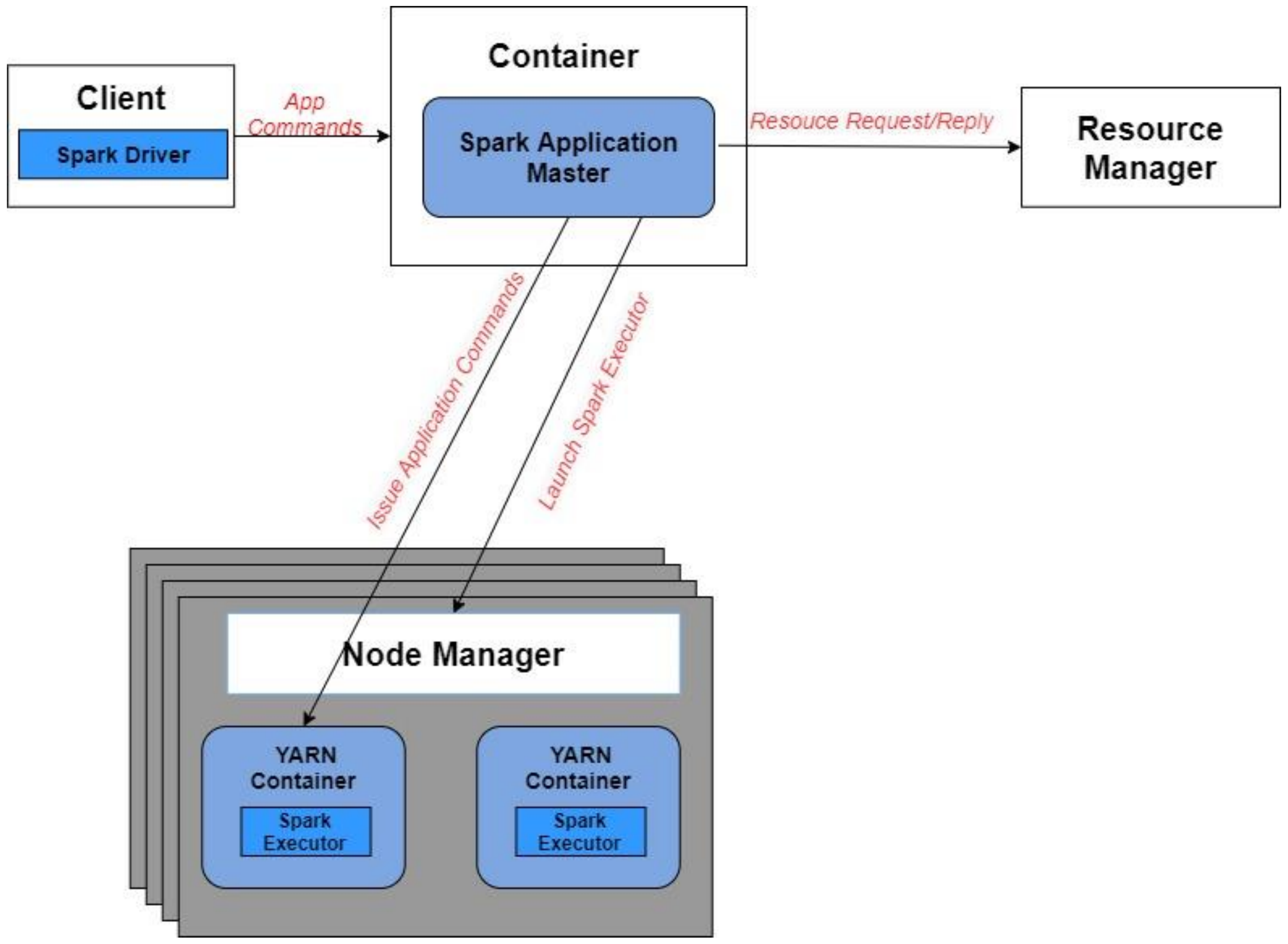
Deployment Mode

- There are 2 types of deployment mode:
  - **Client mode:** Spark Driver will run on the machine from which job is submitted.
    - When the submitting machine is near the Spark executors, there is no high network latency of data movement for final result generation. Client modeis a good option.
    - When the submitting machine is very remote to the executors, also have high network latency. In this case, this spark mode does not work in a good manner.
  - **Cluster mode:** Spark Driver will **not** run on the machine from which job is submitted. Alternately, it will run on a remote node in the cluster.
    - When the submitting machine is remote from executors. Since executors and driver run nearly. Thus, it reduces data movement between the submitting machine and the executors. => this mode works totally fine.
    - While we work with this mode, the chance of network disconnection between driver and executors reduces. => reduces the chance of job failure.

Cluster Deployment Mode in YARN



Client Deployment Mode in YARN



Setting Parameters: Executor

- Spark-submit allows specifying
  - The number of executors
    - **--num-executors** NUM (default value: 2)
  - The number of cores per executor
    - **--executor-cores** NUM (default value: 1)
  - Main memory per executor
    - --executor-memory MEM (default value: 1G)
- The maximum values of these parameters are limited by the configuration of the cluster

Setting Parameters: Driver

- Spark-submit allows specifying
  - The number of cores for the driver
    - **--driver-cores** NUM (default value: 1)
  - Main memory for the driver
    - **--driver-memory** MEM (default value: 1GB)
- Also the maximum values of these parameters are limited by the configuration of the cluster when the deploy-mode is set to cluster

Execution on cluster

- The following command submits a Spark application on a Hadoop cluster

  spark-submit --class it.polito.spark.DriverMyApplication --deploy-mode cluster --master yarn MyApplication.jar arguments

- It executes/submits the application it.polito.spark.DriverMyApplication contained in MyApplication.jar
- The application is executed on a Hadoop cluster based on the YARN scheduler
  - Also the Driver is executed in a node of cluster (deploy mode = cluster)

Execution on local machine

- The following command submits a Spark application on a local machine.

```
spark-submit --class it.polito.spark.DriverMyApplication
--deploy-mode client --master local MyApplication.jar arguments
```

- It executes/submits the application it.polito.spark.DriverMyApplication contained in MyApplication.jar
- The application is executed on a Hadoop cluster based on the local machine.
  - **Both Driver and Executors.**
  - **Hadoop is not needed in this case**.
  - You only need the Spark software.

SparkContext

Definition

- The “connection” of the driver to the cluster is based on the SparkContext object
    - In Scala/Java the name of the class is **SparkContext/JavaSparkContext.**
- The Spark Context is build by means of the constructor of the SparkContext class
    - The only parameter is a configuration object
- Example:

```
val conf = new SparkConf().setAppName(appName).setMaster(“local[4]”) // we
can set a remote master

val sc = new SparkContext(conf)
```

Resilient Distributed Dataset (RDD)

Definition

- A Spark RDD is an immutable distributed collection of objects.
- Each RDD is split in partitions.
  - This choice allows parallelizing the code based on RDDs
    - Code is executed on each partition in isolation. (data cannot be shared between applications).
- RDDs can contain any type of **Scala**, Java, and **Python** objects
  - Including user-defined classes.

Create and Save

- An RDD can be build from an input textual file.
- It is based on the ***textFile()*** method of the ***SparkContext*** class
  - The created RDD is an RDD of Strings
    - org.apache.spark.rdd.RDD[String]
  - Each line of the input file is associated with an object (a string) of the created RDD.
- If the input file is an HDFS file the number of partitions of the created RDD is equal to the number of HDFS blocks used to store the file.
- Example:

```
val lines = sc.textFile(inputFile)
```

- Notice: At this time, the textFile function is not executed, it is lazy evaluation!!

Create and Save

- The developer can manually set the number of partitions
  - It is useful when reading file from the local file system
  - In this case, the ***partition*** parameter of ***textFile*** function is used.
- Example:

```
val lines = sc.textFile(inputFile, 5)
```

Create from a local List

- An RDD can be build from a collection/list of local Scala objects
- It is based on the ***parallelize(list:List<T>)*** method of the SparkContext class The created RDD is an RDD of objects of type T
  - RDD<T>
  - In the created RDD, there is one object (of type T) for each element of the input list
  - Spark tries to set the number of partitions automatically based on your cluster's characteristics
- Example

```
val list = List(1,2,3)
val distList = sc.parallelize(list)

val distList = sc.parallelize(list, 5) // partition = 5
```

- Notice: At this time, the parallelize function is not executed, it is lazy evaluation!!

Save RDD

- An RDD can be easily stored in a textual (HDFS) file
  - It is based on the ***saveAsTextFile(outputPath:String)*** method of the RDD class
    - The method is invoked on the RDD that we want to store
  - Each line of the output file is an object of the RDD on which the saveAsTextFile method is invoked
- Example:

```
lines.saveAsTextFile(outputPath)
```

- Notice: At this time, the content of of lines is computed when saveAsTextFile() is invoked.

Retrieve RDDs

- The content of an RDD can be retrieved from the nodes of the cluster and “stored” in a local Java variable of the Driver
  - It is based on the ***collect()*** method of the RDD class.
- The ***collect()*** method of the RDD class
  - Is invoked on the RDD that we want to “retrieve”.
  - Returns a local list of objects containing the same objects of the considered RDD.
  - Notice: Pay attention to the size of the RDD. Large RDD cannot be stored in a local variable of the Driver.
- Example:

```
val contentOfLine = lines.collect()
// contentOfLine is local variable and allocated on main memory of the Driver.
// lines is the RDD of strings, which is distributed across the nodes of the cluster
```

Transformation and Action

Operations of RDD

- RDD supports 2 types of operation:
  - Transformation:
    - Are operations on RDDs that return a new RDD.
    - Apply a transformation on the elements of the input RDD(s) and the result of the transformation is “stored in/associated to” a **new RDD**. (RDD is immutable).
    - Are computed lazily.
      - Transformations are computed (“executed”) only when an action is applied on the RDDs generated by the transformation operations
    - When a transformation is invoked
      - Spark keeps only track of the dependency between the input RDDand the new RDD returned by the transformation
      - The content of the new RDD is not computed

Operations of RDD

- Actions
    - Are operations that
        - Return results to the Driver program
            - i.e., return local variables

          (Notice: Pay attention to the size of the returned results because they must be stored in the main memory of the Driver program)
        - Or write the result in the storage (output file/folder)
            - The size of the result can be large in this case since it is directly stored in the (distributed) file system.

Computation Graph in Spark

- The graph of dependencies between RDDs represents the information about which RDDs are used to create a new RDD .
    - This is called lineage graph
        - It is represented as a **DAG** (Directed Acyclic Graph)
    - It is needed to compute the content of an RDD the first time an action is invoked on it
    - Or to recompute the content of an RDD when failures occur.
- The lineage graph is also useful for optimization purposes
    - When the content of an RDD is needed, Spark can consider the chain of transformations that are applied to compute the content of the needed RDD and potentially decide how to execute the chain of transformations
        - Spark can potentially change the order of some transformations or merge some of them based on its optimization engine.

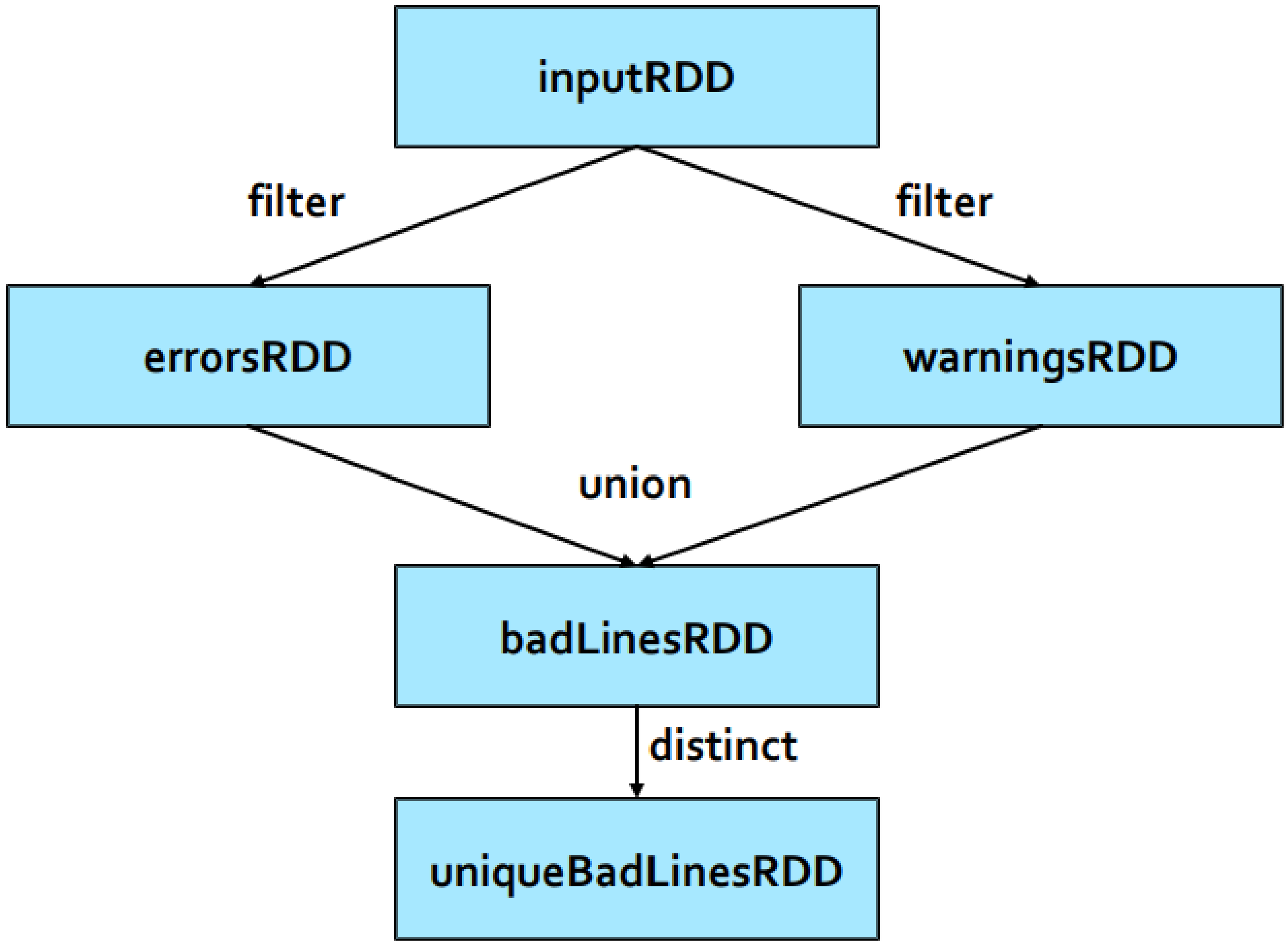
(A research question is "can we customize a optimizer to boost up Spark's performance?)

Computation Graph in Spark

Example:

```
val inputRDD = sc.textFile(“log.txt”)
val errorsRDD = inputRDD.filter(line -> line.contains(“error”))
val warningsRDD = inputRDD.filter(line -> line.contains(“warning”))
val badLinesRDD = errorsRDD.union(warningsRDD)
val uniqueBadLinesRDD = badLinesRDD.distinct()
val numberOfBadLines = uniqueBadLinesRDD.count()
```

Computation Graph in Spark



Computation Graph in Spark

- The application reads the input log file only when the ***count()*** action is invoked
  - It is the first action of the program
- ***filter()***, ***union()***, and ***distinct()*** are transformations
  - They are computed lazily
- Also ***textFile()*** is computed lazily
  - However, it is not a transformation because it is not applied on an RDD

Computation Graph in Spark

- Spark can potentially optimize the “execution” of some transformations
  - For instance, in this case the two filters + union + distinct can be optimized and transformed in one single filter applying the constraint
    - The element contains the string “error” or “warning”
  - This optimization improves the efficiency of the application
    - Spark can performs this kind of optimizations only on particular types of RDDs: Datasets and DataFrames

Basic RDD transformations

Transformations in Spark

- Some basic transformations analyze the content of one **single** RDD and return a new RDD
  - E.g., filter(), map(), flatMap(), etc.
- Some other transformations analyze the content of **two** (input) RDDs and return a new RDD
  - E.g., union(), intersection(), substract(), etc.

Filter

- **Goal**
  - The filter transformation is applied on one single RDD and **returns a new RDD** containing only the elements of the "input" RDD that satisfy a user specified condition.
- **Method**
  - The filter transformation is based on the filter function with syntax:
    - `filter(func: T => Boolean): Dataset[T]`
- Example 1:

```
val inputRDD = sc.textFile("log.txt")
val errorsRDD = inputRDD.filter(line -> line.contains("error"))
```

- Example 2:

```
val inputList = List(1,2,3)
val inputRDD = sc.parallelize(inputList)
val filteredRDD = inputRDD.filter(rows -> {if (row > 2){
                                                                   true
                                 } else{
                                                false}})
```

Map

- **Goal**
  - The map transformation is used to create a new RDD by applying a function on each element of the "input" RDD
  - The new RDD contains exactly one element y for each element x of the "input" RDD
  - The value of y is obtained by applying a user defined function f on x
    - y= f(x) , U is data type of y and T is data type of x
  - The data type of y can be different from the data type of x
    - i.e., U and T can be different
- **Method**
  - Based on the map method with syntax:
    - `map[U](f : Function[T, U]) : RDD[U]`
  - Need to implement the abstract method Function[T,U], which get input with type T and return type U.

Map

- Example:

```
val inputRDD = sc.textFile("log.txt") // Data type: RDD[String]
val countCharRDD = inputRDD.map(line -> line.length)// Data type: RDD[Int]
val countFirstWordRDD = inputRDD.map(line ->
    {
        val firstWord = line.split("\t").map(_.trim).toList(0)
        (firstWord, 1L)
    }) // Data type: RDD[(String, Long)]
```

FlatMap

- **Goal**
  - The flatMap transformation is used to create a new RDD by applying a function f on each element of the "input" RDD
  - The new RDD contains a list of elements obtained by applying f on each element x of the "input" RDD
  - The function f applied on an element x of the "input" RDD returns a list of values [y]
    - [y] = f(x)
    - [y] can be the empty list
  - The final result is the concatenation of the list of values obtained by applying f over all the elements of the "input" RDD
    - i.e., the final RDD contains the "concatenation" of the lists obtained by applying f over all the elements of the input RDD
    - Duplicates are not removed
  - The data type of y can be different from the data type of x
    - i.e., R and T can be different

FlatMap

- **Method**
  - Based on the flatMap method with syntax:
    - `flatMap[U](f : Function1[T, TraversableOnce[U]]): RDD[U]`
    - `TraversableOnce` : is a trait of Scala. It is simply a collection which can be traversed either once only or one or more times.
- Example:

```
    val inputRDD = sc.textFile("log.txt") // Data type: RDD[String]
    val countCharRDD = inputRDD.flatMap(line -> 
line.split("\t").asList().iterator)
    // Note: the lambda function return a list of values (as an iterator)
    // The type of countCharRDD is RDD[String], which contains a concatenation 
of the lists obtained by applying the lambda function over all the elements of 
inputRDD.
```

Distinct

- **Goal**
  - The distinct transformation is applied on one single RDD and returns a new RDD containing the list of distinct elements (values) of the “input” RDD
- **Method**
  - Based on the distinct() method.
  - No classes implementing Spark’s function interfaces are needed in this case
- **Shuffle**
  - A shuffle operation is executed for computing the result of the distinct transformation
    - Data from different input partitions must be compared to remove duplicates
  - The shuffle operation is used to repartition the input data
  - Notice: This is a heavy operation !!!

Set Transformation

Transformation	Purpose	Example	Result

union(input:RDD<T>):RDD<T>	- Return a new RDD containing the union of the elements of the “input”” RDD and the elements of the one passed as parameter to union().
	- Duplicate values are not removed.
	- All the RDDs have the same data type.
		inputRDD1={1,2,2,3}
		inputRDD2={3,3,4,5}
		
		inputRDD1.union(i nputRDD2)
			{1, 2, 2, 3, 3, 3, 4, 5}

intersection(input:RDD<T>):RD D<T>	- It is similar as above, except that result is intersection of two input RDDs.	inputRDD1.intersection (i nputRDD2)	{3}
subtract(input:RDD<T>):RDD< T>	- Return a new RDD the elements appearing only in the “input”” RDD and not in the one passed as parameter to subtract().	inputRDD1.subtract(i nputRDD2)	{1,2,3}

Basic RDD actions

Actions in Spark

- Spark actions can retrieve the content of an RDD or the result of a function applied on an RDD and
  - "Store" it in a local variable of the Driver program
    - Pay attention to the size of the returned value
  - Or store the content of an RDD in an output folder or database
  - The spark actions that return a result that is stored in local variables of the Driver
    - Are executed locally on each node containing partitions of the RDD on which the action is invoked
      - Local results are generated in each node
    - Local results are sent on the network to the Driver that computes the final result and store it in local variables of the Driver
- The basic actions returningobjects to the Driver are
  - **collect**(), count(), countByValue(), take(), top(), takeSample(), **reduce**(), fold(), **aggregate**(), foreach()

Collect

- **Goal**
  - The collect action returns a local list of objects containing the same objects of the considered RDD.
  - Notice: Pay attention to the size of the RDD. Large RDD cannot be memorized in a local variable of the Driver.
- **Method**
  - The collect action is based on the collect():List<T> method.
- Example

```
    val list = List(1,2,3,3)
    val inputRDD = sc.parallelize(list) //Assume that it is very large and
distributed in nodes of the clusters. It can be stored at hard disk of the
local node if it is needed.
    val retrievedRDD = inputRDD.collect() // This is a local variable of
driver node and it will be stored in main memory. If the amount of returned
values is large, it will raise exceptions.
    // Note: if you need to save content of the RDD on driver node, you can
use saveAsTextFile method to store data on hard disk.
```

Reduce

- **Goal**
  - Return a single object obtained by combining the objects of the RDD by using a user provide “function”
    - z = f(x,y), with x,y is belong to value set of RDD.
  - The provided “function” must be associative and commutative.
    - A function f is associative—i.e., f (f (x, y), z) = f (x, f (y, z))
    - A function f is commutative—i.e., f (x, y) = f (y, x)
- **Method**
  - The collect action is based on the reduce() method with syntax:
    - `reduce(f: (T, T) => T): T`

Reduce

- How it works:
    - Suppose *L* contains the list of elements of the "input" RDD. To compute the final element, the reduce action operates as follows
        - Apply the user specified function on a pair of elements e1 and e2 occurring in *L* and obtain a new element *enew*
        - Remove the "original" elements *e1* and *e2* from *L* and then insert the element *enew* in *L*
        - If *L* contains only one value then return it as final result of the reduce action. Otherwise, return to step 1
        - Notice: The "function" must be associative and commutative
            - The computation of the reduce action can be performed in parallel without problems
            - Otherwise the result depends on how the input RDD is partitioned. For the functions that are not associative and commutative the output depends on how the RDD is split in partitions and how the content of each partition is analyzed

Reduce

- Example:

```
val inputRDD = sc.parallelize(List(1,2,3,4,5), numPartitions)
val resultRDD = inputRDD.reduce((e1, e2) => e1+e2)
// function here is associate and commutative
val resultRDD = inputRDD.reduce((e1, e2) => e1/e2)
// function here is non-commutative !!
val resultRDD = inputRDD.reduce((e1, e2) => e1-e2)
// function here is non-associative  and non-commutative!!
```

- HW Question: Propose other examples of non-associative and non-commutative function.

Aggregate

- **Goal**
  - Return a single object obtained by combining the objects of the RDD and an initial “zero” value by using two user provide functions
    - The provided functions must be associative
      - Otherwise the result depends on how the RDD is partitioned
    - The returned objects and the ones of the input RDD can be instances of different classes
      - This is the main difference with respect to reduce method
- **Method**
  - The “input” RDD contains objects of type T while the returned object is of type U
  - The collect action is based on the aggregate() method with syntax:

    -
      ```
      aggregate[U](zeroValue: U)(seqOp: (U, T) => U, combOp: (U, U) => U):
      U
      ```

Aggregate

- How it works:
  - Suppose that *L* contains the list of elements of the "input" RDD and this RDD is split in a set of partitions, i.e., a set of lists *{L1 , .., Ln }*
  - The aggregate action computes a partial result in each partition and then combines/merges the results.
  - It operates as follows
    - Aggregate the partial results in each partition, obtaining a set of partial results (of type U) ***P****= {p1 , .., pn }*
    - Apply the second user specified "function" on a pair of elements *p1* and *p2* in ***P*** and obtain a new element *pnew*
    - Remove the "original" elements *p1* and *p2* from P and then insert the element *pnew* in ***P***
    - If ***P*** contains only one value then return it as final result of the aggregate action. Otherwise, return to step 2

Aggregate

- Suppose that
  - *Li* is the list of elements on the i-th partition of the “input” RDD
  - And zeroValue is the initial zero value
- To compute the partial result over the elements in *Li* the aggregate action operates as follows
  - Set accumulator to zeroValue (accumulator=zeroValue)
  - Apply the first user specified function on accumulator and an elements *ej* in *Li* and update accumulator with the value returned by the function
  - Remove the original elements *ej* from *Li*
  - If *Li* is empty return accumulator as (final) partial result of the current partition. Otherwise, return to step 2

Aggregate

- Example:

```
val inputRDD = sc.parallelize(List(1,2, 3, 3)
def function1 (accu:Int, v: Int) => accu + v
def function2 (accu1:Int, accu2:Int) => accu1 + accu2
val resultRDD = inputRDD.aggregate(function1, function2)
```

Summary

- Assumption: We have a list of integers as follows: {1,2,3,3}

Transformation	Purpose	Example	Result
collect	Return a List containing the first num elements of the RDD. The objects of the RDD and objects of the returned list are objects of the same class.	inputRDD.take(2)	{1,2}
first	Return the first element of the RDD	first()	{1}
top(	Return a List containing the top num elements of the RDD based on the default sort order/comparator of the objects. The objects of the RDD and objects of the returned list are objects of the same class.	inputRDD.top(2)	{3,3}
takeSample	Return a List containing a random sample of size n of the RDD. The objects of the RDD and objects of the returned list are objects of the same class.	inputRDD. takeSample (false, 1)	Non-det erministic

Summary

reduce	Return a single object obtained by combining the values of the objects of the RDD by using a user provide "function". The provided "function" must be associative and commutative The object returned by the method and the objects of the RDD belong to the same class	The passed "function" is the sum function	9
fold	Same as reduce but with the provided zero value.	The passed "function" is the sum function and the passed zeroValue is 0	9
aggregate	Similar to reduce() but usually used to return a different type.	Compute a pair of integers where the first one is the sum of the values of the RDD and the second the number of elements	(9, 4)

PairRDDs

What is PairRDDs

- Spark supports also RDDs of key-value pairs, a.k.a pair RDDs.
- Pair RDDs are characterized also by the operations available for the "standard" RDDs.
- Many applications are based on pair RDDs
  - Pair RDDs allow "grouping" data by key performing computation by key (i.e., group)
  - The basic idea is similar to the one of the MapReduce-based programs
  - But there are more operations already available
- From a local list of pairs, (Scala)
  - It is similar as "standard" RDD, we can use parallelize method to convert a local list of pairs to a PairRDDs.
  - Example:

```
val inputRDD = sc.parallelize(List((“A”, 1), (“B”, 2)), numPartitions)
```

- From "standard" RDDs,
  - We can use methods, such as: map and flatMap.

Transformation on PairRDDs

- The transformations of “standard” RDDs can be applied to PairRDDs.
- Moreover, PairRDDs have some more transformation:
  - reduceByKey(), combineByKey, groupByKey(), mapValues(), join(), etc,.

Transformation on a single PairRDD

ReduceByKey

- Similarly to the reduce() action, the reduceByKey() transformation aggregate values
- However,
    - reduceByKey() is executed on RDDs of key-value pairs and returns a set of key-value pairs
    - reduce() is executed on an RDD and returns one single value (stored in a local python variable)
- And
    - reduceByKey() is a transformation
        - reduceByKey() is executed lazily and its result is stored in another RDD
    - Whereas reduce() is an action
- Shuffle
    - A shuffle operation is executed for computing the result of the reduceByKey() transformation
    - The result/value for each group/key is computed from data stored in different input partitions

ReduceByKey

- Example:

```
val inputRDD = sc.parallelize(List((“A”, 1), (“B”, 2), (“B”, 5), (“C”, 9)))
val resultRDD = inputRDD.reduceByKey((v1, v2) => {

          if (v1 > v2){

                    v1
                                                  } else {
                                                            v2
                                                  }})
          // result: List((“A”, 1), (“B”, 5), (“C”, 9))
```

GroupByKey

- **Goal**
    - Create a new PairRDD where there is one pair for each distinct key k of the input PairRDD. The value associated with key k in the new PairRDD is the list of values associated with k in the input PairRDD
- Notice:
    - If you are grouping values per key to perform then an aggregation such as sum or average over the values of each key then groupByKey is not the right choice reduceByKey, aggregateByKey or combineByKey provide better performances for associative and commutative aggregations
    - groupByKey is useful if you need to apply an aggregation/compute a function that is not associative
    - Question: Can you reason that ?
- Shuffle
    - A shuffle operation is executed for computing the result of the groupByKey() transformation
        - Each group/key is associated with/is composed of values which are stored in different partitions of the input RDD

GroupByKey

- Example

```
val inputRDD = sc.parallelize(List((“A”, 1), (“B”, 2), (“B”, 5), (“C”, 9)))
val resultRDD = inputRDD.groupByKey()
// result: List((“A”, List(1)), (“B”, List(2,5)), (“C”, List(9)))
```

MapValues

- **Goal**
  - Apply a user-defined function over the value of each pair of an input PairRDD and return a new PairRDD.
  - One pair is created in the returned PairRDD for each input pair
    - The key of the created pair is equal to the key of the input pair
    - The value of the created pair is obtained by applying the user-defined function on the value of the input pair
  - The data type of the values of the new PairRDD can be different from the data type of the values of the “input” PairRDD
  - The data type of the key is the same
  - Notice: The difference between map() and mapValues() that with map(), you can change both key, and value of a pair in RDD. Meanwhile, the mapValues() only changes the value.

MapValues

- Example

```
val inputRDD = sc.parallelize(List(("A", 1), ("B", 2), ("B", 5), ("C", 9)))
val resultRDD = inputRDD.mapValues(value => value * 2)
val resultRDD = inputRDD.map((key, value) => (key, value * 2))
```

Summary

- Assumption: We have a list of pairs as follows:
    - {("k1", 2), ("k3", 4), ("k3", 6)}
        - The key of each tuple is a String
        - The value of each tuple is an Integer

Transformation	Purpose	Example	Result
reduceByKey	Return a PairRDD containing one pair for each key of the "input" PairRDD. The value of each pair of the new PairRDD is obtained by combining the values of the input PairRDD with the same key. The "input" PairRDD and the new PairRDD have the same data type.	sum per key	{("k1", 2), ("k3", 5)}
foldByKey	Similar to the reduceByKey() transformation. However, foldByKey() is characterized also by a zero value	sum per key with zero value = 1	{("k1", 3), ("k3", 11)}

Summary

combineByKey	Return a PairRDD containing one pair for each key of the “input” PairRDD. The value of each pair of the new PairRDD is obtained by combining the values of the input PairRDD with the same key. The “input” PairRDD and the new PairRDD can have different data types. (combineByKey is a more general “form” of reduceByKey)	average value per key	{(“k1”, 2), (“k3”, 10)}
groupByKey	Return a PairRDD containing one pair for each key of the “input” PairRDD. The value of each pair of the new PairRDD is a “list” containing the values of the input PairRDD with the same key		{(“k1”, [2]), (“k3”, [4, 6])}
mapValues	Apply a function over each pair of a PairRDD and return a new PairRDD. The applied function returns one pair for each pair of the “input” PairRDD. The function is applied only to the value without changing the key. The “input” PairRDD and the new PairRDD can have a different data type.	v -> v+1 (i.e., for each input pair (k,v), the pair (k,v+1) is included in the new PairRDD)	{(“k1”, 3), (“k3”, 5), (“k3”, 7)}

Summary

flatMapValues	Apply a function over each pair in the input PairRDD and return a new RDD of the result. The applied function returns a set of pairs (from 0 to many) for each pair of the “input” RDD. The function is applied only to the value without changing the key. The “input” RDD and the new RDD can have a different data type.	v -> v.to(5) (i.e., for each input pair (k,v), the set of pairs (k,u) with values of u from v to 5 are returned and included in the new PairRDD)	{(“k1”, 2), (“k1”, 3), (“k1”, 4), (“k1”, 5), (“k3”, 4), (“k3”, 5)}
keys	Return an RDD containing the keys of the input pairRDD		{“k1”, “k3”, “k3”}
values	Return an RDD containing the values of the input pairRDD		{2, 4, 6}
sortByKey	Return a PairRDD sorted by key. The “input” PairRDD and the new PairRDD have the same data type.		{(“k1”, 2), (“k3”, 3), (“k3”, 6)}

Transformation on two PairRDDs

Transformations on pairs of PairRDDs

- Spark supports also some transformations on two PairRDDs
  - SubtractByKey, join, coGroup, etc.

SubtractByKeys

- **Goal**
  - Create a new PairRDD containing only the pairs of the input PairRDD associated with a key that is not appearing as key in the pairs of the other PairRDD
    - The data type of the new PairRDD is the same of the "input" PairRDD
    - The input PairRDD and the other PairRDD must have the same type of keys
      - The data type of the values can be different
- Shuffle
  - A shuffle operation is executed for computing the result of the subtractByKey() transformation
    - Keys from different partitions of the two input RDDs must be compared

SubtractByKeys

- Example

```
val inputRDD = sc.parallelize(List((“A”, 1), (“B”, 2), (“C”, 9)))
val otherRDD = sc.parallelize(List((“A”, “XYZ”)))
val resultRDD = inputRDD.subtractByKeys(otherRDD)
//Result: List((“B”, 2), (“C”, 9))
```

Join

- **Goal**
  - Join the key-value pairs of two PairRDDs based on the value of the key of the pairs
    - Each pair of the input PairRDD is combined with all the pairs of the other PairRDD with the same key
    - The new PairRDD
      - Has the same key data type of the “input” PairRDDs
      - Has a tuple as value (the pair of values of the two joined input pairs)
    - The input PairRDD and the other PairRDD
      - Must have the same type of keys
      - But the data types of the values can be different
- Shuffle
  - A shuffle operation is executed for computing the result of the join() transformation
    - Keys from different partitions of the two input RDDs must be compared and values from different partitions must be retrieved

Join

- Example

```
val inputRDD_1 = sc.parallelize(List((“A”, 1), (“B”, 2), (“C”, 9)))
val inputRDD_2 = sc.parallelize(List((“A”, “TYU”), (“B”, “XYZ”), (“C”,
“YUI”)))
val resultRDD = inputRDD_1.join(inputRDD_2)
//Result Type: RDD[(String, (Int, String))]
```

Actions on PairRDDs

Actions on PairRDDs

- Spark supports also some specific actions on PairRDDs
  - countByKey, collectAsMap, lookup
- Notice: When using actions, you should pay attention on the size of returned value.

Some basic actions

- Assumption: We have a list of pairs as follows:
  - {("k1", 2), ("k3", 4), ("k3", 6)}
    - The key of each tuple is a String
    - The value of each tuple is an Integer

Transformation	Purpose	Example	Result
countByKey	Return a local Map containing the number of elements in the input PairRDD for each key of the input PairRDD.	inputRDD. countByKey()	{("K1",1), ("K3",2)}
collectAsMap	Return a local Map containing the pairs of the input PairRDD	inputRDD. collectAsMap()	{("k1", 2), ("k3", 6)} Or {("k1", 2), ("k3", 4)} Depending on the order of the pairs in the PairRDD
lookup	Return a local List containing all the values associated with the key specified as parameter	inputRDD. lookup("k3")	{4, 6}

Advanced Topics on RDDs

Questions in class