RDDs: creation, basic transformations and actions

RDD-based programming

Spark context

SparkContext

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

SparkContext

- Example

```
// Create a configuration object
// and set the name of the application
SparkConf conf=new SparkConf().setAppName("Application
name");

// Create a Spark Context object
JavaSparkContext sc = new JavaSparkContext(conf);
```

RDD basics

RDD basics

- 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
- RDDs can contain any type of Scala, Java, and Python objects
  - Including user-defined classes

RDD: create and save

RDD creation

- RDDs can be created
  - By loading an external dataset (e.g., the content of a folder, a single file, a database table, etc.)
  - By parallelizing a local collection of objects created in the Driver (e.g., a Java collection)

Create RDDs from files

- An RDD can be build from an input textual file
  - It is based on the **textFile(String inputPath)** method of the **JavaSparkContext** class
    - The created RDD is an RDD of Strings
      - JavaRDD<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
      - To support data locality

Create RDDs from files

- Example

```
// Build an RDD of Strings from the input textual file
// Each element of the RDD is a line of the input file
JavaRDD<String> lines=sc.textFile(inputFile);
```

Create RDDs from files

- Example

```
// Build an RDD of Strings from the input textual file
// Each element of the RDD is a line of the input file
JavaRDD<String> lines=sc.textFile(inputFile);
```

No computation occurs when sc.textFile() is invoked

- Spark only records how to create the RDD
- The data is lazily read from the input file only when the data is needed (i.e., when an action is applied on the lines RDD, or on one of its "descendant" RDDs)

Create RDDs from files

- The developer can manually set the number of partitions
  - It is useful when reading file from the local file system
  - In this case the **textFile(String inputPath, int numPartitions)** method of the **JavaSparkContext** class is used

Create RDDs from files

- Example

```
// Build an RDD of Strings from a local input textual file.
// The number of partitions is manually set to 5
// Each element of the RDD is a line of the input file
JavaRDD<String> lines=sc.textFile(inputFile, 5);
```

Create RDDs from a local Java collection

- An RDD can be build from a “local” Java collection/list of local Java objects
  - It is based on the **parallelize(List<T> list)** method of the **JavaSparkContext** class
    - The created RDD is an RDD of objects of type T
      - JavaRDD<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

Create RDDs from a local Java collection

- Example

```
// Create a local Java list
List<String> inputList = Arrays.asList("First element",
"Second element", "Third element");

// Build an RDD of Strings from the local list.
// The number of partitions is set automatically by Spark
// There is one element of the RDD for each element
// of the local list
JavaRDD<String> distList = sc.parallelize(inputList);
```

Create RDDs from a local Java collection

- Example

```
// Create a local Java list
List<String> inputList = Arrays.asList("First element",
"Second element", "Third element");

// B
// T
// T
the
JavaRDD<String> distList = sc.parallelize(inputList);
```

No computation occurs when sc.parallelize() is invoked
- Spark only records how to create the RDD
- The data is lazily read from the input file only when the data is needed (i.e., when an action is applied on the distList RDD, or on one of its “descendant” RDDs)

Create RDDs from a local Java collection

- When the parallelize(List<T> list) is invoked
  - Spark tries to set the number of partitions automatically based on your cluster’s characteristics
- The developer can set the number of partition by using the method parallelize(List<T> list, int numPartitions) of the JavaSparkContext class

Create RDDs from a local Java collection

- Example

```
// Create a local Java list
List<String> inputList = Arrays.asList("First element",
"Second element", "Third element");

// Build an RDD of Strings from the local list.
// The number of partitions is set to 3
// There is one element of the RDD for each element
// of the local list
JavaRDD<String> distList = sc.parallelize(inputList,3 );
```

Save RDDs

- An RDD can be easily stored in a textual (HDFS) file
  - It is based on the **saveAsTextFile(String outputPath)** method of the **JavaRDD** 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

Save RDDs

- Example

```
// Store the lines RDD in the output textual file
// Each element of the RDD is stored in a line
// of the output file
lines.saveAsTextFile(outputPath);
```

Save RDDs

- Example

```
// Store the lines RDD in the output textual file
// Each element of the RDD is stored in a line
// of the output file
lines.saveAsTextFile(outputPath);
```

The content of lines is computed when saveAsTextFile() is invoked

- Spark computes the content associated with an RDD only when the content is needed

Retrieve the content of RDDs and "store" it in local Java variables

- 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 **List<T> collect()** method of the **JavaRDD<T>** class

Retrieve the content of RDDs and "store" it in local Java variables

- The collect() method of the JavaRDD class
  - Is invoked on the RDD that we want to "retrieve"
  - Returns a local Java list of objects containing the same objects of the considered RDD
  - **Pay attention to the size of the RDD**
  - **Large RDD cannot be stored in a local variable of the Driver**

Retrieve the content of RDDs and “store” it in local Java variables

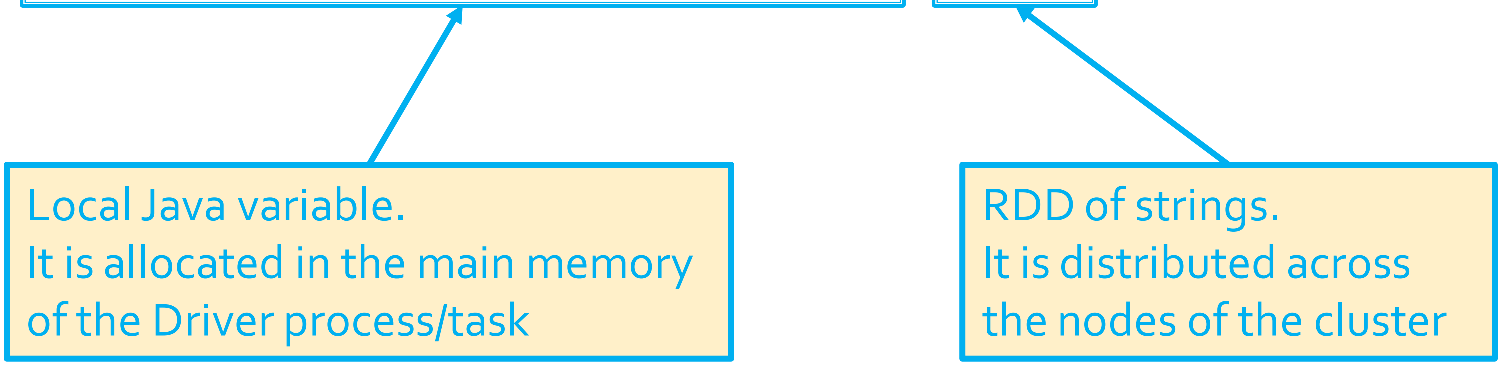
- Example

```
// Retrieve the content of the lines RDD and store it
// in a local Java collection
// The local Java collection contains a copy of each
// element of the RDD
List<String> contentOfLines=lines.collect();
```

Retrieve the content of RDDs and "store" it in local Java variables

- Example

```
// Retrieve the content of the lines RDD and store it
// in a local Java collection
// The local Java collection contains a copy of each
// element of the RDD
List<String> contentOfLines=lines.collect();
```



Local Java variable.
It is allocated in the main memory of the Driver process/task

RDD of strings.
It is distributed across the nodes of the cluster

Transformations and Actions

RDD operations

- RDD support two types of operations
  - Transformations
  - Actions

RDD operations

- Transformations
  - 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
    - Remember that RDDs are immutable
      - Hence, you cannot change the content of an already existing RDD
      - You can only apply a transformation on the content of an RDD and “store/assign” the result in/to a new RDD

RDD operations

- Transformations
  - Are **computed lazily**
    - i.e., 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 RDD** and the **new RDD** returned by the transformation
      - The content of the new RDD is not computed

RDD operations

- 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

RDD operations

- 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

RDD operations

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

Example of lineage graph (DAG)

- Consider the following code

```
....
public static void main(String[] args) {
// Initialization of the application
....

// Read the content of a log file
JavaRDD<String> inputRDD = sc.textFile("log.txt");
```

Example of lineage graph (DAG)

```
// Select the rows containing the word “error”
JavaRDD<String> errorsRDD =
                inputRDD.filter(line -> line.contains("error"));

// Select the rows containing the word “warning”
JavaRDD<String> warningRDD =
                inputRDD.filter(line -> line.contains("warning"));

// Union errorsRDD and warningRDD
// The result is associated with a new RDD: badLinesRDD
JavaRDD<String> badLinesRDD =
                errorsRDD.union(warningRDD);
```

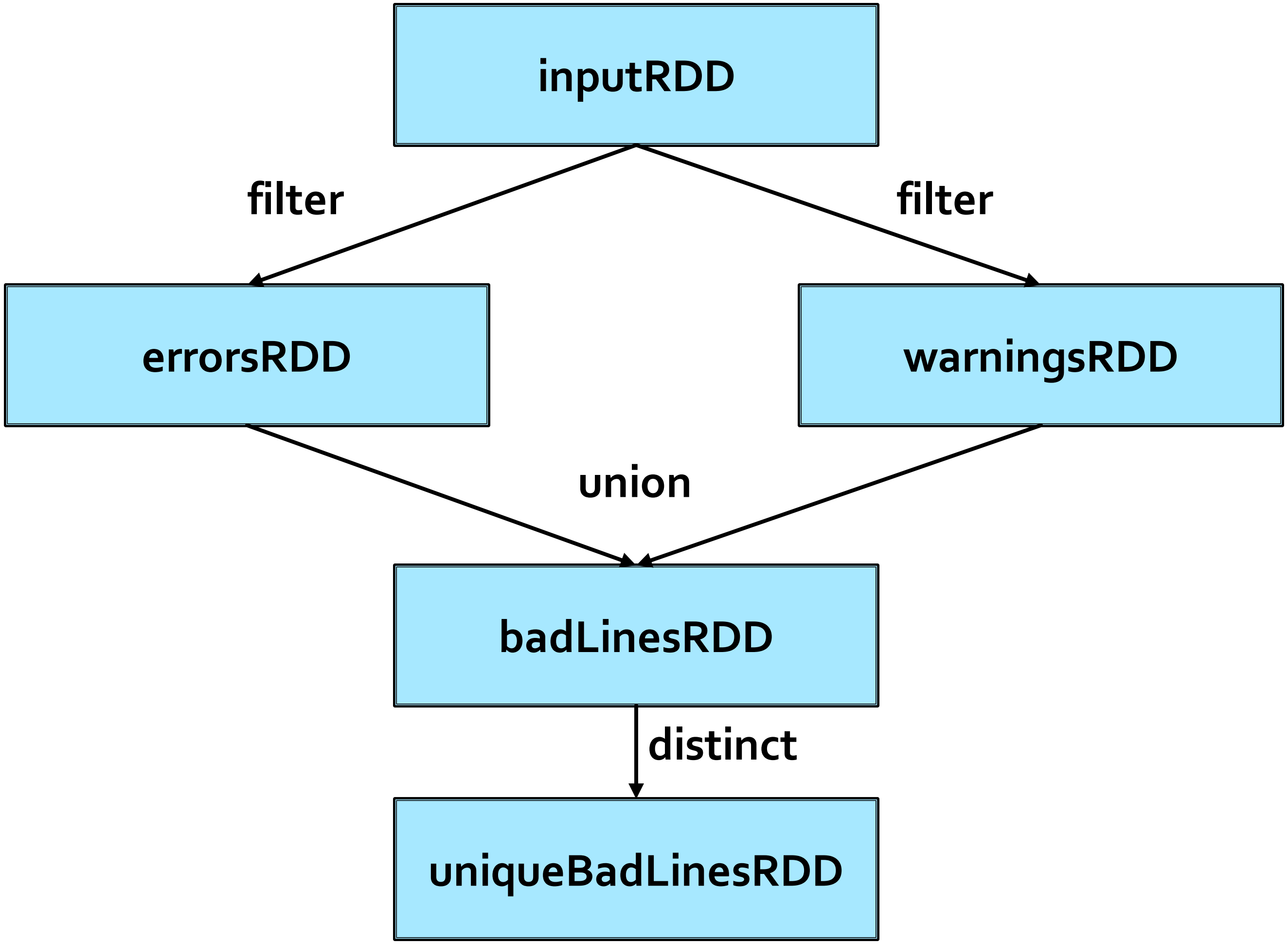
Example of lineage graph (DAG)

```
 // Remove duplicates lines (i.e., those lines containing
 // both “error” and “warning”)
JavaRDD<String> uniqueBadLinesRDD =
                                   badLinesRDD.distinct();

// Count the number of bad lines by applying
// the count() action
long numBadLines = uniqueBadLinesRDD.count();

// Print the result on the standard output of the driver
System.out.println("Lines with problems:"+numBadLines);
….
```

Example of lineage graph (DAG)



Example of lineage graph (DAG)

- 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

Example of lineage graph (DAG)

- Spark, similarly to an SQL optimizer, 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

Passing function to Transformations and Actions

Passing “functions” to Transformations and Actions

- Many transformations and some actions are based on user provided functions that specify with “transformation” function must be applied on each element of the “input” RDD
- For example the filter() transformation selects the elements of an RDD satisfying a user specified constraint
  - The user specified constraint is a function applied on each element of the “input” RDD

Passing “functions” to Transformations and Actions

- Each language has its own solution to pass functions to Spark’s transformations and actions
- In Java, we pass objects of the classes that implement one of the Spark’s function interfaces
  - Each class implementing a Spark’s function interface is characterized by the **call** method
    - The call method contains the “function” that we want to pass to a transformation or to an action

Passing “functions” to Transformations and Actions

- There are many Spark’s function interfaces
  - The difference is given by the number of parameters of the call method and the data types of the input parameters and returned values

Some Spark’s function interfaces

Spark’s function interface name	Method to be implemented	Usage
Function<T, R>	R call(T)	This “function” takes in input an object of type T and returns an object of type R
Function2<T1, T2, R>	R call(T1, T2)	This “function” takes in input an object of type T1 and an object of type T2 and returns an object of type R
FlatMapFunction<T, R>	Iterable<R> call(T)	This “function” takes in input an object of type T and returns an Iterable “containing” zero or more objects of type R

- There are others Spark’s function interfaces
  - We will see them in the following slides

Example based on the filter() transformation

- Create an RDD from a log file
- Create a new RDD containing only the lines of the log file containing the word “error”
  - The filter() transformation applies the filter constraint on each element of the input RDD
  - The filter constraint is specified by creating a class implementing the Function<T, R> interface an passing it to the filter method
    - T is a String
    - R is a Boolean

Solution based on named classes

```
// Define a class implementing the Function interface
class ContainsError implements Function<String, Boolean> {
    // Implement the call method
    public Boolean call(String x) {
            return x.contains("error");
    }
}
.....
// Read the content of a log file
JavaRDD<String> inputRDD = sc.textFile("log.txt");

// Select the rows containing the word “error”
JavaRDD<String> errorsRDD = inputRDD.filter(new ContainsError());
```

Solution based on named classes

```
// Define a class implementing the Function interface
class ContainsError implements Function<String, Boolean> {
        // Implement the call method
        public Boolean call(String x) {
                return x.contains("error");
        }
}
```

When it is invoked, this method analyzes the value of the parameter x and returns true if the string x contains the substring “error”. Otherwise, it returns false

```
// Select the rows containing the word “error”
JavaRDD<String> errorsRDD = inputRDD.filter(new ContainsError());
```

Solution based on named classes

```
// Define a class implementing the Function interface
class ContainsError implements Function<String, Boolean> {
        // Implement the call method
        public Boolean call(String x) {
                return x.contains("error");
        }
}
```

Apply the filter() transformation on inputRDD.
The filter transformation selects the elements of inputRDD satisfying the constraint specified in the call method of the ContainsError class

```
// Select the rows containing the word “error”
JavaRDD<String> errorsRDD = inputRDD.filter(new ContainsError());
```

Solution based on named classes

```
// Define a class implementing the Function interface
class ContainsError implements Function<String, Boolean> {
	// Implement the call method
	public Boolean call(String x) {
			return x.contains("error");
	}
}
```

An object of the ContainsError is instantiated and its call method is applied on each element x of inputRDD. If the function returns true then x is "stored" in the new errorsRDD RDD. Otherwise x is discarded

```
// Select the rows containing the word "error"
JavaRDD<String> errorsRDD = inputRDD.filter(new ContainsError());
```

Solution based on anonymous classes

```
// Read the content of a log file
JavaRDD<String> inputRDD = sc.textFile("log.txt");

// Select the rows containing the word “error”
JavaRDD<String> errorsRDD = inputRDD.filter(
        new Function<String, Boolean>() {
                public Boolean call(String x) {
                        return x.contains("error");
                }
        }
);
```

Solution based on anonymous classes

This part of the code defines on the fly a temporary anonymous class implementing the Function<String, Boolean> interface. An object of this class is instantiated and its call method is applied on each element x of inputRDD. If the call method returns true then x is “stored” in the new errorsRDD RDD. Otherwise x is discarded

```
// Select the rows containing the word “error”
JavaRDD<String> errorsRDD = inputRDD.filter(
        new Function<String, Boolean>() {
                public Boolean call(String x) {
                        return x.contains("error");
                }
        }
);
```

Solution based on anonymous classes

This part of the code is equal to the content of the ContainErrors class defined in the previous solution based on named classes

```
// Select the rows containing the word “error”
JavaRDD<String> errorsRDD = inputRDD.filter(
	new Function<String, Boolean>() {
		public Boolean call(String x) {
			return x.contains("error");
		}
	}
);
```

Solution based on lambda functions

```
// Read the content of a log file
JavaRDD<String> inputRDD = sc.textFile("log.txt");

// Select the rows containing the word “error”
JavaRDD<String> errorsRDD =
        inputRDD.filter(x -> x.contains("error"));
```

Solution based on lambda functions

This part of the code, which is based on a lambda function, defines on the fly the implementation of the call method of the Function<String, Boolean> interface. This part of the code is applied on each element x of inputRDD. If it returns true then x is "stored" in the new errorsRDD RDD. Otherwise x is discarded

```
// Rea
JavaF

// Select the rows containing the word "error"
JavaRDD<String> errorsRDD =
      inputRDD.filter(x -> x.contains("error"));
```

Solution based on lambda functions

This part of the code is equal to the content of the ContainErrors class defined in the previous solution based on named classes

```
// Rea
JavaRDD<String> inputRDD = sc.textFile("log.txt");

// Select the rows containing the word "error"
JavaRDD<String> errorsRDD =
        inputRDD.filter(x -> x.contains("error"));
```

Named classes vs Anonymous classes vs Lambda functions

- The three solutions are more or less equivalent in terms of efficiency
- Lambda function-based code
  - More concise code
  - Slightly faster than the other approaches
  - Lambda functions are **not available in Java 1.7** or lower
- Named classes are usually preferred
  - When the same function is used in several applications
    - Code reuse
  - When something is passed to the constructor of the new class
    - The code is more readable w.r.t. unnamed classes

Basic Transformations

Basic RDD transformations

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

Syntax

- In the following, the following syntax is used
  - T = Type of the objects of the RDD on which the transformation is applied
  - R= Type of the objects of the new RDD returned by the applied transformation when the returned RDD can have a data type different from T
    - i.e., when the returned RDD can have a data type different from the “input” data type
  - The RDD on which the transformation is applied in referred as “input” RDD

Filter transformation

Filter transformation

- 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 **JavaRDD<T> filter(Function<T, Boolean>)** method of the **JavaRDD<T>** class

Filter transformation

- A lambda function implementing the call method of the Function<T, Boolean> interface is passed to the filter method
  - The **public Boolean call(T element)** method of the Function<T, Boolean> interface must be implemented
    - It contains the code associated with the condition that we want to apply on each element **e** of the "input" RDD
      - If the condition is satisfied then the call method returns true and the input element **e** is selected
      - Otherwise, it returns false and the **e** element is discarded

Filter transformation: Example 1

- Create an RDD from a log file
- Create a new RDD containing only the lines of the log file containing the word “error”

Filter transformation: Example 1

```
// Read the content of a log file
JavaRDD<String> inputRDD = sc.textFile("log.txt");

// Select the rows containing the word “error”
JavaRDD<String> errorsRDD =
                inputRDD.filter(element -> element.contains("error"));
```

Filter transformation: Example 1

```
// Read the content of a log file
JavaRDD<String> inputRDD = sc.textFile("log.txt");

// Select the rows containing the word "error"
JavaRDD<String> errorsRDD =
          inputRDD.filter(element -> element.contains("error"));
```

We are working with RDDs of Strings.
Hence, the data type T of the Function<T, Boolean> interface we are implementing is String and also each element on which the lambda function is applied is a String

Filter transformation: Example 2

- Create an RDD of integers containing the values {1, 2, 3, 3}
- Create a new RDD containing only the values greater than 2

Filter transformation: Example 2

```
………

// Create an RDD of integers. Load the values 1, 2, 3, 3 in this RDD
List<Integer> inputList = Arrays.asList(1, 2, 3, 3);
JavaRDD<Integer> inputRDD = sc.parallelize(inputList);

// Select the values greater than 2
JavaRDD<Integer> greaterRDD =
    inputRDD.filter(element -> { if (element>2)
                                     return true;
                                 else
                                     return false;
                               });
```

Filter transformation: Example 2

We are working with Integers.
Hence, the data type T of the Function<T, Boolean> interface we are implementing is Integer and also the elements we are analyzing by means of the lambda functions are integer values

```
..

//                                                             RDD
List<Integer> inputList = Arrays.asList(1, 2, 3, 3);
JavaRDD<Integer> inputRDD = sc.parallelize(inputList);

// Select the values greater than 2
JavaRDD<Integer> greaterRDD =
      inputRDD.filter(element -> { if (element>2)
                                             return true;
                                     else
                                             return false;
                                   });
```

Filter transformation: Example 2

```
.........

// Create an RD
List<Integer> i
JavaRDD<Integ

// Select the values greater than 2
JavaRDD<Integer> greaterRDD =
        inputRDD.filter(element -> { if (element>2)
                                            return true;
                                       else
                                            return false;
                                   });
```

In this case the lambda function contains more than one line of code. We use curly brackets to contain the code implementing the call method

Map transformation

Map transformation

- 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**)
  - The data type of **y** can be different from the data type of **x**
    - i.e., R and T can be different

Map transformation

- Method
  - The map transformation is based on the JavaRDD<R> map(Function<T, R>) method of the JavaRDD<T> class
  - A lambda function implementing the call method of the Function<T, R> interface is passed to the map method
    - The public R call(T element) method of the Function<T, R> interface must be implemented
      - It contains the code that is applied over each element of the "input" RDD to create the elements of the returned RDD
        - For each input element of the "input" RDD exactly one single new element is returned by the call method

Map transformation: Example 1

- Create an RDD from a textual file containing the surnames of a list of users
  - Each line of the file contains one surname
- Create a new RDD containing the length of each surname

Map transformation: Example 1

```
………

// Read the content of the input textual file
JavaRDD<String> inputRDD = sc.textFile("usernames.txt");

// Compute the lengths of the input surnames
JavaRDD<Integer> lenghtsRDD =
                    inputRDD.map(element -> new Integer(element.length());
```

Map transformation: Example 1

The input RDD is an RDD of Strings.
Hence also the input element of the lambda function is a String

```
// Read the content of the input textual file
JavaRDD<String> inputRDD = sc.textFile("usernames.txt");

// Compute the lengths of the input surnames
JavaRDD<Integer> lenghtsRDD =
                        inputRDD.map(element -> new Integer(element.length()));
```

Map transformation: Example 1

………

The new RDD is an RDD of Integers.
Hence the lambda function returns a new Integer for each input element

```
// Compute the lengths of the input surnames
JavaRDD<Integer> lenghtsRDD =
                    inputRDD.map(element -> new Integer(element.length()));
```

Map transformation: Example 2

- Create an RDD of integers containing the values {1, 2, 3, 3}
- Create a new RDD containing the square of each input element

Map transformation: Example 2

```
………

// Create an RDD of integers. Load the values 1, 2, 3, 3 in this RDD
List<Integer> inputList = Arrays.asList(1, 2, 3, 3);
JavaRDD<Integer> inputRDD = sc.parallelize(inputList);

// Compute the square of each input element
JavaRDD<Integer> squaresRDD =
	inputRDD.map(element -> new Integer(element*element));
```

FlatMap transformation

FlatMap transformation

- 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

FlatMap transformation

- 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 transformation

- Method
  - The flatMap transformation is based on the **JavaRDD<R> flatMap(FlatMapFunction<T, R>)** method of the **JavaRDD<T>** class

FlatMap transformation

- A lambda function implementing the call method of the FlatMapFunction<T, R> interface is passed to the flatMap method
  - The **public Iterable<R> call(T element)** method of the FlatMapFunction<T, R> interface must be implemented
    - It contains the code that is applied on each element of the "input" RDD and returns a list of elements which will be included in the new returned RDD
    - For each element of the "input" RDD a list of new elements is returned by the call method
      - The list can be empty

FlatMap transformation: Example 1

- Create an RDD from a textual file containing a text
- Create a new RDD containing the list of words, with repetitions, occurring in the input textual document
  - Each element of the returned RDD is one of the words occurring in the input textual file
  - The words occurring multiple times in the input file appear multiple times, as distinct elements, also in the returned RDD

FlatMap transformation: Example 1

```
………

// Read the content of the input textual file
JavaRDD<String> inputRDD = sc.textFile("document.txt");

// Compute the list of words occurring in document.txt
JavaRDD<String> listOfWordsRDD =
                    inputRDD.flatMap(x -> Arrays.asList(x.split(" ")).iterator());
```

FlatMap transformation: Example 1

```
………

// Read the content of the input textual file
JavaRDD<String> inputRDD = sc.textFile("document.txt");

// Compute the list of words occurring in document.txt
JavaRDD<String> listOfWordsRDD =
                    inputRDD.flatMap(x -> Arrays.asList(x.split(" ")).iterator());
```

In this case the lambda function returns a “list” of values (an Iterator) for each input element

FlatMap transformation: Example 1

```
………

// Read the content of the input textual file
JavaRDD<String> inputRDD = sc.textFile("document.txt");

// Compute the list of words occurring in document.txt
JavaRDD<String> listOfWordsRDD =
                    inputRDD.flatMap(x -> Arrays.asList(x.split(" ")).iterator());
```

The new RDD contains the "concatenation" of the lists obtained by applying the lambda function over all the elements of inputRDD

FlatMap transformation: Example 1

```
………

// Read the content of the input textual file
JavaRDD<String> inputRDD = sc.textFile("document.txt");

// Compute the list of words occurring in document.txt
JavaRDD<String> listOfWordsRDD =
            inputRDD.flatMap(x -> Arrays.asList(x.split(" ")).iterator());
```

The new RDD is an RDD of Strings and not an RDD of Lists of strings

Distinct transformation

Distinct transformation

- 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
  - The distinct transformation is based on the **JavaRDD<T> distinct()** method of the **JavaRDD<T>** class
  - No classes implementing Spark's function interfaces are needed in this case

Distinct transformation

- 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
    - All the repetitions of the same input element are associated with the same output partition (in which one single copy of the element is stored)
    - A hash function assigns each input element to one of the new partitions

Distinct transformation: Example 1

- Create an RDD from a textual file containing the names of a list of users
  - Each line of the file contains one name
- Create a new RDD containing the list of distinct names occurring in the input textual file
  - The type of the new RDD is the same of the "input" RDD

Distinct transformation: Example 1

```
// Read the content of a textual input file
JavaRDD<String> inputRDD = sc.textFile("names.txt");

// Select the distinct names occurring in inputRDD
JavaRDD<String> distinctNamesRDD = inputRDD.distinct();
```

Distinct transformation: Example 2

- Create an RDD of integers containing the values {1, 2, 3, 3}
- Create a new RDD containing only the distinct values appearing in the “input” RDD

Distinct transformation: Example 2

```
// Create an RDD of integers. Load the values 1, 2, 3, 3 in this RDD
List<Integer> inputList = Arrays.asList(1, 2, 3, 3);
JavaRDD<Integer> inputRDD = sc.parallelize(inputList);

// Compute the set of distinct words occurring in inputRDD
JavaRDD<Integer> distinctIntRDD = inputRDD.distinct();
```

Sample transformation

Sample transformation

- Goal
  - The sample transformation is applied on one single RDD and returns a new RDD containing a random sample of the elements (values) of the "input" RDD
- Method
  - The sample transformation is based on the **JavaRDD<T> sample(boolean withReplacement, double fraction)** method of the **JavaRDD<T>** class
    - withReplacement specifies if the random sample is with replacement (true) or not (false)
    - fraction specifies the expected size of the sample as a fraction of the "input" RDD's size (values in the range [0, 1])

Sample transformation: Example 1

- Create an RDD from a textual file containing a set of sentences
  - Each line of the file contains one sentence
- Create a new RDD containing a random sample of sentences
  - Use the "without replacement" strategy
  - Set fraction to 0.2 (i.e., 20%)

Sample transformation: Example 1

```
// Read the content of a textual input file
JavaRDD<String> inputRDD = sc.textFile("sentences.txt");

// Create a random sample of sentences
JavaRDD<String> randomSentencesRDD = inputRDD.sample(false,0.2);
```

Sample transformation: Example 2

- Create an RDD of integers containing the values {1, 2, 3, 3}
- Create a new RDD containing a random sample of the input values
    - Use the "replacement" strategy
    - Set fraction to 0.2

Sample transformation: Example 2

```
// Create an RDD of integers. Load the values 1, 2, 3, 3 in this RDD
List<Integer> inputList = Arrays.asList(1, 2, 3, 3);
JavaRDD<Integer> inputRDD = sc.parallelize(inputList);

// Create a sample of the inputRDD
JavaRDD<Integer> randomSentencesRDD = inputRDD.sample(true,0.2);
```

Set transformations

Set transformations

- Spark provides also a set of transformations that operate on two input RDDs and return a new RDD
- Some of them implement standard set transformations
  - Union
  - Intersection
  - Subtract
  - Cartesian

Set transformations

- All these transformations have
  - Two input RDDs
    - One is the RDD on which the method is invoked
    - The other RDD is passed as parameter to the method
  - One output RDD
- All the involved RDDs have the same data type when union, intersection, or subtract are used
- "Mixed" data types can be used with the cartesian transformation

Union transformation

- The union transformation is based on the JavaRDD<T> union(JavaRDD<T> secondInputRDD) method of the JavaRDD<T> class
  - The result is the union of the two input RDDs
  - **Duplicates elements are not removed**
    - This choice is related to optimization reasons
      - Removing duplicates means having a global view of the whole content of the two input RDDs
      - Since each RDD is split in partitions that are stored in different nodes of the cluster, the contents of all partitions should be “shared” to remove duplicates → A very expensive operation
    - The shuffle operation is not needed in this case

Union transformation

- If you really need to union two RDDs and remove duplicates you can apply the distinct() transformation on the output of the union() transformation
  - But pay attention that **distinct() is a heavy operation**
    - It is associated with a shuffle operation
  - Use distinct() if and only if duplicate removal is indispensable for your application

Intersection transformation

- The intersection transformation is based on the **JavaRDD<T> intersection(JavaRDD<T> secondInputRDD)** method of the **JavaRDD<T>** class
  - The result is the intersection of the two input RDDs
  - The **output will not contain** any **duplicate elements**, even if the input RDDs did
  - A **shuffle** operation is executed for computing the result of **intersection**
    - Elements from different input partitions must be compared to find common elements

Subtract transformation

- The subtract transformation is based on the **JavaRDD<T> subtract(JavaRDD<T> secondInputRDD)** method of the **JavaRDD<T>** class
  - The result contains the elements appearing only in the RDD on which the subtract method is invoked
    - In this transformation the two input RDDs play different roles
  - Duplicates are not removed
  - A **shuffle** operation is executed for computing the result of **subtract**
    - Elements from different input partitions must be compared

Cartesian transformation

- The cartesian transformation is based on the JavaPairRDD<T, R> cartesian(JavaRDD<R> secondInputRDD) method of the JavaRDD<T> class
  - The two "input" RDDs can contain objects of two different data types
  - The returned RDD is an RDD of pairs (JavaPairRDD) containing all the combinations composed of one element of the first input RDD and one element of the second input RDD
    - We will see later what a JavaPairRDD is

Cartesian transformation

- A large amount of data is sent on the network
  - Elements from different input partitions must be combined to compute the returned pairs
    - The elements of the two input RDDs are stored in different partitions, which could be in different servers

Set transformations: Example 1

- Create two RDDs of integers
  - inputRDD1 contains the values {1, 2, 2, 3, 3}
  - inputRDD2 contains the values {3, 4, 5}
- Create three new RDDs
  - outputUnionRDD contains the union of inputRDD1 and inputRDD2
  - outputIntersectionRDD contains the intersection of inputRDD1 and inputRDD2
  - outputSubtractRDD contains the result of inputRDD1 \ inputRDD2

Set transformations: Example 1

```
// Create two RDD of integers.
List<Integer> inputList1 = Arrays.asList(1, 2, 2, 3, 3);
JavaRDD<Integer> inputRDD1 = sc.parallelize(inputList1);

List<Integer> inputList2 = Arrays.asList(3, 4, 5);
JavaRDD<Integer> inputRDD2 = sc.parallelize(inputList2);

// Create three new RDDs by using union, intersection, and subtract
JavaRDD<Integer> outputUnionRDD = inputRDD1.union(inputRDD2);

JavaRDD<Integer> outputIntersectionRDD =
                        inputRDD1.intersection(inputRDD2);

JavaRDD<Integer> outputSubtractRDD =
                        inputRDD1.subtract(inputRDD2);
```

Cartesian transformation: Example 1

- Create two RDDs of integers
  - inputRDD1 contains the values {1, 2, 2, 3, 3}
  - inputRDD2 contains the values {3, 4, 5}
- Create a new RDD containing the cartesian product of inputRDD1 and inputRDD2

Cartesian transformation: Example 1

```
// Create two RDD of integers.
List<Integer> inputList1 = Arrays.asList(1, 2, 2, 3, 3);
JavaRDD<Integer> inputRDD1 = sc.parallelize(inputList1);

List<Integer> inputList2 = Arrays.asList(3, 4, 5);
JavaRDD<Integer> inputRDD2 = sc.parallelize(inputList2);

// Compute the cartesian product
JavaPairRDD<Integer, Integer> outputCartesianRDD =
                                        inputRDD1.cartesian(inputRDD2);
```

Cartesian transformation: Example 1

```
// Create two RDD of integers.
List<Integer> inputList1 = Arrays.asList(1, 2, 2, 3, 3);
JavaRDD<Integer> inputRDD1 = sc.parallelize(inputList1);

List<Integer> inputList2 = Arrays.asList(3, 4, 5);
JavaRDD<Integer> inputRDD2 = sc.parallelize(inputList2);

// Compute the cartesian product
JavaPairRDD<Integer, Integer> outputCartesianRDD =
                         inputRDD1.cartesian(inputRDD2);
```

Each element of the returned RDD is a pair of integer elements

Cartesian transformation: Example 2

- Create two RDDs
  - inputRDD1 contains the Integer values {1, 2, 3}
  - inputRDD2 contains the String values {"A", "B"}
- Create a new RDD containing the cartesian product of inputRDD1 and inputRDD2

Cartesian transformation: Example 2

```
// Create an RDD of Integers and an RDD of Strings
List<Integer> inputList1 = Arrays.asList(1, 2, 3);
JavaRDD<Integer> inputRDD1 = sc.parallelize(inputList1);

List<String> inputList2 = Arrays.asList("A", "B");
JavaRDD<String> inputRDD2 = sc.parallelize(inputList2);

// Compute the cartesian product
JavaPairRDD<Integer,String> outputCartesianRDD =
                                    inputRDD1.cartesian(inputRDD2);
```

Cartesian transformation: Example 2

```
// Create an RDD of Integers and an RDD of Strings
List<Integer> inputList1 = Arrays.asList(1, 2, 3);
JavaRDD<Integer> inputRDD1 = sc.parallelize(inputList1);

List<String> inputList2 = Arrays.asList("A", "B");
JavaRDD<String> inputRDD2 = sc.parallelize(inputList2);

// Compute the cartesian product
JavaPairRDD<Integer,String> outputCartesianRDD =
                              inputRDD1.cartesian(inputRDD2);
```

Each element of the returned RDD is a pair of type <Integer, String>

Basic transformations: Summary

Basic transformations based on one single RDD: Summary

- All the examples reported in the following tables are applied on an RDD of integers containing the following elements (i.e., values)
  - {1, 2, 3, 3}

Basic transformations based on one single RDD: Summary

Transformation	Purpose	Example of applied function	Result
JavaRDD<T> filter(Function<T, Boolean>)	Return an RDD consisting only of the elements of the "input"" RDD that pass the condition passed to filter(). The "input" RDD and the new RDD have the same data type.	x != 1	{2,3,3}
JavaRDD<R> map(Function<T, R>)	Apply a function to each element in the RDD and return an RDD of the result. The applied function return one element for each element of the "input" RDD. The "input" RDD and the new RDD can have a different data type.	x -> x+1 (i.e., for each input element x, the element with value x+1 is included in the new RDD)	{2,3,4,4}

Basic transformations based on one single RDD: Summary

Transformation	Purpose	Example of applied function	Result
JavaRDD<R> flatMap(FlatMapFunction<T, R>)	Apply a function to each element in the RDD and return an RDD of the result. The applied function return a set of elements (from 0 to many) for each element of the "input" RDD. The "input" RDD and the new RDD can have a different data type.	x ->x.to(3) (i.e., for each input element x, the set of elements with values from x to 3 are returned)	{1,2,3,2, 3,3,3}

Basic transformations based on one single RDD: Summary

Transformation	Purpose	Example of applied function	Result
JavaRDD<T> distinct()	Remove duplicates	-	{1, 2, 3}
JavaRDD<T> sample(boolean withReplacement, double fraction)	Sample the content of the "input" RDD, with or without replacement and return the selected sample. The "input" RDD and the new RDD have the same data type.	-	Nondeterministic

Basic transformations based on two RDDs: Summary

- All the examples reported in the following tables are applied on the following two RDDs of integers
  - inputRDD1 {1, 2, 2, 3, 3}
  - inputRDD2 {3, 4, 5}

Basic transformations based on two RDDs: Summary

Transformation	Purpose	Example	Result
JavaRDD<T> union(JavaRDD<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.union(inputRDD2)	{1, 2, 2, 3, 3, 3, 4, 5}
JavaRDD<T> intersection(JavaRDD<T>)	Return a new RDD containing the intersection of the elements of the "input"" RDD and the elements of the one passed as parameter to intersection(). All the RDDs have the same data type.	inputRDD1.intersection(inputRDD2)	{3}

Basic transformations based on two RDDs: Summary

Transformation	Purpose	Example	Result
JavaRDD<T> subtract(JavaRDD<T>)	Return a new RDD the elements appearing only in the “input”” RDD and not in the one passed as parameter to subtract(). All the RDDs have the same data type.	inputRDD1.subtract(inputRDD2)	{1, 2, 2}
JavaRDD<T> cartesian(JavaRDD<T>)	Return a new RDD containing the cartesian product of the elements of the “input”” RDD and the elements of the one passed as parameter to cartesian(). All the RDDs have the same data type.	inputRDD1.cartesian(inputRDD2)	{(1, 3), (1, 4), … , (3,5)}

Basic Actions

Basic RDD actions

- 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 Java variable of the Driver program
    - **Pay attention to the size of the returned value**
    - **Pay attentions that date are sent on the network from the nodes containing the content of RDDs and the executor running the Driver**
  - Or store the content of an RDD in an output folder or database

Basic RDD actions

- The spark actions that return a result that is stored in local (Java) variables of the Driver
  1. Are executed locally on each node containing partitions of the RDD on which the action is invoked
     - Local results are generated in each node
  2. 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 returning (Java) objects to the Driver are
  - collect(), count(), countByValue(), take(), top(), takeSample(), reduce(), fold(), aggregate(), foreach()

Syntax

- In the following, the following syntax is used
  - T = Type of the objects of the RDD on which the transformation is applied
  - The RDD on which the action is applied in referred as “input” RDD

Collect action

Collect action

- Goal
  - The collect action returns a local Java list of objects containing the same objects of the considered RDD
  - **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 **List<T> collect()** method of the **JavaRDD<T>** class

Collect action: Example 1

- Create an RDD of integers containing the values {1, 2, 3, 3}
- Retrieve the values of the created RDD and store them in a local Java list that is instantiated in the Driver

Collect action: Example 1

```
// Create an RDD of integers. Load the values 1, 2, 3, 3 in this RDD
List<Integer> inputList = Arrays.asList(1, 2, 3, 3);
JavaRDD<Integer> inputRDD = sc.parallelize(inputList);

// Retrieve the elements of the inputRDD and store them in
// a local Java list
List<Integer> retrievedValues = inputRDD.collect();
```

Collect action: Example 1

```
// Create an RDD of integers. Load the values 1, 2, 3, 3 in this RDD
List<Integer> inputList = Arrays.asList(1, 2, 3, 3);
JavaRDD<Integer> inputRDD = sc.parallelize(inputList);

// Retrieve the elements of the inputRDD and store them in
// a local Java list
List<Integer> retrievedValues = inputRDD.collect();
```

inputRDD is distributed across the nodes of the cluster.
It can be large and it is stored in the local disks of the nodes if it is needed

Collect action: Example 1

```
// Create an RDD of integers. Load the values 1, 2, 3, 3 in this RDD
List<Integer> inputList = Arrays.asList(1, 2, 3, 3);
JavaRDD<Integer> inputRDD = sc.parallelize(inputList);

// Retrieve the elements of the inputRDD and store them in
// a local Java list
List<Integer> retrievedValues = inputRDD.collect();
```

retrievedValues is a local Java variable.
It can only be stored in the main memory of the process/task associated with the Driver.
Pay attention to the size of the list.
Use the collect() action if and only if you are sure that the list is small.
Otherwise, store the content of the RDD in a file by using the saveAsTextFile method

Count action

Count action

- Goal
  - Count the number of elements of an RDD
- Method
  - The count action is based on the **long count()** method of the **JavaRDD<T>** class

Count action: Example 1

- Consider the textual files “document1.txt” and “document2.txt”
- Print the name of the file with more lines

Count action: Example 1

```
// Read the content of the two input textual files
JavaRDD<String> inputRDD1 = sc.textFile("document1.txt");
JavaRDD<String> inputRDD2 = sc.textFile("document2.txt");

// Count the number of lines of the two files = number of elements
// of the two RDDs
long numLinesDoc1 = inputRDD1.count();
long numLinesDoc2 = inputRDD2.count();

if (numLinesDoc1> numLinesDoc2) {
        System.out.println ("document1.txt");
}
else {
        if (numLinesDoc2> numLinesDoc1)
                System.out.println ("document2.txt");
        else
                System.out.println("Same number of lines");
}
```

CountByValue action

CountByValue action

- Goal
  - The countByValue action returns a local Java Map object containing the information about the number of times each element occurs in the RDD
- Method
  - The countByValue action is based on the **java.util.Map<T, java.lang.Long> countByValue()** method of the **JavaRDD<T>** class

CountByValue action: Example 1

- Create an RDD from a textual file containing the first names of a list of users
  - Each line contain one name
- Compute the number of occurrences of each name and “store” this information in a local variable of the Driver

CountByValue action: Example 1

```
// Read the content of the input textual file
JavaRDD<String> namesRDD = sc.textFile("names.txt");

// Compute the number of occurrencies of each name
 java.util.Map<String, java.lang.Long> namesOccurrences =
namesRDD.countByValue();
```

CountByValue action: Example 1

```
// Read the content of the input textual file
JavaRDD<String> namesRDD = sc.textFile("names.txt");

// Compute the number of occurrencies of each name
java.util.Map<String, java.lang.Long> namesOccurrences =
namesRDD.countByValue();
```

Also in this case, pay attention to the size of the returned map (i.e., the number of names in this case). Use the countByValue() action if and only if you are sure that the returned java.util.Map is small. Otherwise, use an appropriate chain of Spark's transformations and write the final result in a file by using the saveAsTextFile method.

Take action

Take action

- Goal
  - The take(n) action returns a local Java list of objects containing the first **n** elements of the considered RDD
    - The order of the elements in an RDD is consistent with the order of the elements in the file or collection that has been used to create the RDD
- Method
  - The take action is based on the **List<T> take(int n)** method of the **JavaRDD<T>** class

Take action: Example 1

- Create an RDD of integers containing the values {1, 5, 3, 3, 2}
- Retrieve the first two values of the created RDD and store them in a local Java list that is instantiated in the Driver

Take action: Example 1

```
// Create an RDD of integers. Load the values 1, 5, 3, 3,2 in this RDD
List<Integer> inputList = Arrays.asList(1, 5, 3, 3, 2);
JavaRDD<Integer> inputRDD = sc.parallelize(inputList);

// Retrieve the first two elements of the inputRDD and store them in
// a local Java list
List<Integer> retrievedValues = inputRDD.take(2);
```

First action

First action

- Goal
  - The first() action returns a local Java object containing the first element of the considered RDD
    - The order of the elements in an RDD is consistent with the order of the elements in the file or collection that has been used to create the RDD
- Method
  - The first action is based on the **T first()** method of the **JavaRDD<T>** class

First vs Take(1)

- The only difference between first() and take(1) is given by the fact that
  - first() returns a **single element** of type T
    - The returned element is the first element of the RDD
  - take(1) returns a **list** of elements **containing one single element** of type T
    - The only element of the returned list is the first element of the RDD

Top action

Top action

- Goal
  - The top(n) action returns a local Java list of objects containing the top n (largest) elements of the considered RDD
    - The ordering is the default one of class T (the class of the objects of the RDD)
    - The descending order is used
- Method
  - The top action is based on the **List<T> top(int n)** method of the **JavaRDD<T>** class

Top action: Example 1

- Create an RDD of integers containing the values {1, 5, 3, 3, 2}
- Retrieve the top-2 greatest values of the created RDD and store them in a local Java list that is instantiated in the Driver

Top action: Example 1

```
// Create an RDD of integers. Load the values 1, 5, 3, 3,2 in this RDD
List<Integer> inputList = Arrays.asList(1, 5, 3, 3, 2);
JavaRDD<Integer> inputRDD = sc.parallelize(inputList);

// Retrieve the top-2 elements of the inputRDD and store them in
// a local Java list
List<Integer> retrievedValues = inputRDD.top(2);
```

TakeOrdered action

TakeOrdered action

- Goal
  - The takeOrdered(n, comparator<T>) action returns a local Java list of objects containing the top **n** (smallest) elements of the considered RDD
    - The ordering is specified by the developer by means of a class implementing the java.util.Comparator<T> interface
- Method
  - The takeOrderedaction is based on the **List<T> takeOrdered (int n, java.util.Comparator<T> comp)** method of the **JavaRDD<T>** class

TakeSample action

TakeSample action

- Goal
  - The takeSample(withReplacement, n, [seed]) action returns a local Java list of objects containing n random elements of the considered RDD
- Method
  - The takeSampleaction is based on the **List<T> takeSample(boolean withReplacement, int n)** method of the **JavaRDD<T>** class
    - withReplacement specifies if the random sample is with replacement (true) or not (false)

TakeSample action

- Method
  - The **List<T> takeSample(boolean withReplacement, int n, long seed)** method of the **JavaRDD<T>** class is used when we want to set the seed

TakeSample action: Example 1

- Create an RDD of integers containing the values {1, 5, 3, 3, 2}
- Retrieve randomly, without replacement, 2 values from the created RDD and store them in a local Java list that is instantiated in the Driver

TakeSample action: Example 1

```
// Create an RDD of integers. Load the values 1, 5, 3, 3,2 in this RDD
List<Integer> inputList = Arrays.asList(1, 5, 3, 3, 2);
JavaRDD<Integer> inputRDD = sc.parallelize(inputList);

// Retrieve randomly two elements of the inputRDD and store them in
// a local Java list
List<Integer> randomValues= inputRDD.takeSample(false, 2);
```

Reduce action

Reduce action

- Goal
  - Return a single Java object obtained by combining the objects of the RDD by using a user provide "function"
    - The provided "function" must be **associative** and **commutative**
      - otherwise the result depends on the content of the partitions and the order used to analyze the elements of the RDD's partitions
    - The returned object and the ones of the "input" RDD are all instances of the same class (T)

Reduce action

- Method
  - The reduce action is based on the **T reduce(Function2<T, T, T> f)** method of the **JavaRDD<T>** class
  - A lambda function implementing the call method of the Function2<T, T, T> interface is passed to the reduce method
    - The **public T call(T element1, T element2)** method of the Function2<T, T, T> interface must be implemented
      - It contains the code that is applied to combine the values of the elements of the RDD

Reduce action: 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
  1. Apply the user specified "function" on a pair of elements e_1 and e_2 occurring in *L* and obtain a new element e_{new}
  2. Remove the "original" elements e_1 and e_2 from *L* and then insert the element e_{new} in *L*
  3. If *L* contains only one value then return it as final result of the reduce action. Otherwise, return to step 1

Reduce action: how it works

- The **"function" must be associative and commutative**
  - The computation of the reduce action can be performed in parallel without problems

Reduce action: how it works

- 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
  - i.e., 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 action: Example 1

- Create an RDD of integers containing the values {1, 2, 3, 3}
- Compute the sum of the values occurring in the RDD and “store” the result in a local Java integer variable in the Driver

Reduce action: Example 1

```
.....
// Create an RDD of integers. Load the values 1, 2, 3, 3 in this RDD
List<Integer> inputListReduce = Arrays.asList(1, 2, 3, 3);
JavaRDD<Integer> inputRDDReduce = sc.parallelize(inputListReduce);

// Compute the sum of the values;
Integer sum= inputRDDReduce.reduce(
                (element1, element2) -> element1+element2);
```

Reduce action: Example 1

```
.....
// Create an RDD of integers. Load the values 1, 2, 3, 3 in this RDD
List<Integer> inputListReduce = Arrays.asList(1, 2, 3, 3);
JavaRDD<Integer> inputRDDReduce = sc.parallelize(inputListReduce);

// Compute the sum of the values;
Integer sum= inputRDDReduce.reduce(
                    (element1, element2) -> element1+element2);
```

This lambda function combines two input integer elements at a time and returns theirs sum

Reduce action: Example 2

- Create an RDD of integers containing the values {1, 2, 3, 3}
- Compute the maximum value occurring in the RDD and "store" the result in a local Java integer variable in the Driver

Reduce action: Example 2

```
.....
// Create an RDD of integers. Load the values 1, 2, 3, 3 in this RDD
List<Integer> inputListReduce = Arrays.asList(1, 2, 3, 3);
JavaRDD<Integer> inputRDDReduce = sc.parallelize(inputListReduce);

// Compute the maximum value
Integer max = inputRDDReduce.reduce(
                (element1, element2) ->
                {
                        if (element1>element2)
                                return element1;
                        else
                                return element2;
                });
```

Fold action

Fold action

- Goal
  - Return a single Java object obtained by combining the objects of the RDD by using a user provide "function"
  - The provided "function"
    - Must be **associative**
      - Otherwise the result depends on how the RDD is partitioned
    - It is **not required to be commutative**
  - An initial "zero" value is also specified

Fold action

- Method
  - The fold action is based on the **T fold(T zeroValue, Function2<T, T, T> f)** method of the **JavaRDD<T>** class
  - The "zero" value of type T is passed
  - A lambda function implementing the call method of the Function2<T, T, T> interface is passed to the fold method
    - The **public T call(T element1, T element2)** method of the Function2<T, T, T> interface must be implemented
      - It contains the code that is applied to combine the values of the elements of the RDD

Fold vs Reduce

- Fold is characterized by the “zero” value
- Fold can be used to parallelize functions that are associative but non-commutative
  - E.g., concatenation of a list of strings

Aggregate action

Aggregate action

- Goal
  - Return a single Java 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 () and fold( )

Aggregate action

- Method
  - The aggregate action is based on the **U aggregate(U zeroValue, Function2<U,T,U> seqOp, Function2<U,U,U> combOp)** method of the **JavaRDD<T>** class
  - The “input” RDD contains objects of type T while the returned object is of type U
    - We need one “function” for merging an element of type T with an element of type U to return a new element of type U
      - It is used to merge the elements of the input RDD and the zero value
    - We need one “function” for merging two elements of type U to return a new element of type U
      - It is used to merge two elements of type U obtained as partial results generated by two different partitions

Aggregate action

- The first “function” is based on a lambda function implementing the call method of the Function2<U, T, U> interface
  - The **public U call(U element1, T element2)** method of the Function2<U, T, U> interface must be implemented
    - It contains the code that is applied to combine the zero value, and the intermediate values, with the elements of the RDD
- The second “function ” is based on a lambda function implementing the call method of the Function2<U, U, U> interface
  - The **public U call(U element1, U element2)** method of the Function2<U, U, U> interface must be implemented
    - It contains the code that is applied to combine two elements of type U returned as partial results by two different partitions

Aggregate action: how it woks

- 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 $\{L_1, .., L_n\}$
- The aggregate action computes a partial result in each partition and then combines/merges the results.
- It operates as follows
  1. Aggregate the partial results in each partition, obtaining a set of partial results (of type U) $P= \{p_1, .., p_n\}$
  2. Apply the second user specified “function” on a pair of elements p_1 and p_2 in P and obtain a new element p_{new}
  3. Remove the “original” elements p_1 and p_2 from P and then insert the element p_{new} in P
  4. If P contains only one value then return it as final result of the aggregate action. Otherwise, return to step 2

Aggregate action: how it woks

- Suppose that
  - *L_i* 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 *L_i* the aggregate action operates as follows
  1. Set **accumulator** to **zeroValue** (accumulator=zeroValue)
  2. Apply the first user specified "function" on **accumulator** and an elements *e_j* in *L_i* and update **accumulator** with the value returned by the function
  3. Remove the "original" elements *e_j* from *L_i*
  4. If *L_i* is empty return **accumulator** as (final) partial result of the current partition. Otherwise, return to step 2

Aggregate action: Example 1

- Create an RDD of integers containing the values {1, 2, 3, 3}
- Compute both the sum of the values occurring in the input RDD and the number of elements of the input RDD and finally “store” in a local Java variable of the Driver the average computed over the values of the input RDD

Aggregate action: Example 1

```
// Define a class to store two integers: sum and numElements
class SumCount implements Serializable {
        public int sum;
        public int numElements;

        public SumCount(int sum, int numElements)              {
                this.sum = sum;
                this.numElements = numElements;
        }

        public double avg() {
                return sum/ (double) numElements;
        }
}
```

Aggregate action: Example 1

```
.....
// Create an RDD of integers. Load the values 1, 2, 3, 3 in this RDD
List<Integer> inputListAggr = Arrays.asList(1, 2, 3, 3);
JavaRDD<Integer> inputRDDAggr = sc.parallelize(inputListAggr);
```

Aggregate action: Example 1

```
// Instantiate the zero value
SumCount zeroValue = new SumCount(0, 0);

// Compute sum and number over the elements of inputRDDAggr
SumCount result = inputRDDAggr.aggregate(zeroValue,
    (a, e) -> {
            a.sum = a.sum + e;
            a.numElements = a.numElements + 1;
            return a;
    },
    (a1, a2) -> {
            a1.sum = a1. sum + a2.sum;
            a1.numElements = a1.numElements + a2.numElements;
            return a1;
    } );
```

Aggregate action: Example 1

Instantiate the zero value

```
// Instantiate the zero value
SumCount zeroValue = new SumCount(0, 0);

// Compute sum and number over the elements of inputRDDAggr
SumCount result = inputRDDAggr.aggregate(zeroValue,
        (a, e) -> {
                a.sum = a.sum + e;
                a.numElements = a.numElements + 1;
                return a;
        },
        (a1, a2) -> {
                a1.sum = a1. sum + a2.sum;
                a1.numElements = a1.numElements + a2.numElements;
                return a1;
        } );
```

Aggregate action: Example 1

```
// Instantiate the zero value
SumCount zeroValue = new SumCount(0, 0);

// Compute sum and number over the elements of inputRDDAggr
SumCount result = inputRDDAggr.aggregate(zeroValue,
        (a, e) -> {
                a.sum = a.sum + e;
                a.numElements = a.numElements + 1;
                return a;
        },
        (a1, a2) -> {
```

Define the code that is used to implement the call method of the Function2<SumCount, Integer, SumCount> interface.
It is used to combine the elements of the input RDD (Integer objects) with the zero value (SumCount objects) and the intermediate values (SumCount objects)

Aggregate action: Example 1

```
// Instantiate the zero value
SumCount zeroValue = new SumCount(0, 0);

// Compute sum and number over the elements of inputRDDAggr
SumCount result = inputRDDAggr.aggregate(zeroValue,
        (a, e) -> {
                a.sum = a.sum + e;
                a.numElements = a.numElements + 1;
                return a;
        },
        (a1, a2) -> {
```

Data types:
- a is a SumCount object
- e is an Integer

The lambda function returns an updated version of a (SumCount object)

Aggregate action: Example 1

```
// Instantiate the zero value
SumCount zeroValue = new SumCount(0, 0);

// Co
Sum
```

Define the code that is used to implement the call method of the Function2<SumCount, SumCount, SumCount> interface.

It is used to combine the partial results (SumCount objects) emitted by partitions

```
},
(a1, a2) -> {
            a1.sum = a1. sum + a2.sum;
            a1.numElements = a1.numElements + a2.numElements;
            return a1;
} );
```

Aggregate action: Example 1

```
// Instantiate the zero value
SumCount zeroValue = new SumCount(0, 0);

// Compute sum and number over the elements of inputRDDAggr
Su

    },
    (a1, a2) -> {
              a1.sum = a1. sum + a2.sum;
              a1.numElements = a1.numElements + a2.numElements;
              return a1;
    } );
```

Data types:

- a1 is a SumCount object
- a2 is a SumCount object

The lambda function returns an updated version of a1 (SumCount object)

Aggregate action: Example 1

```
// Compute the average value
double avg = result.avg();

// Print the average on the standard output of the driver
System.out.println(avg);
```

Aggregate action: Simulation

- inputRDDAggr = {1, 2, 3, 3}
- Suppose inputRDDAggr is split in the following two partitions
  - {1, 2} and {3, 3}

Aggregate action: Simulation

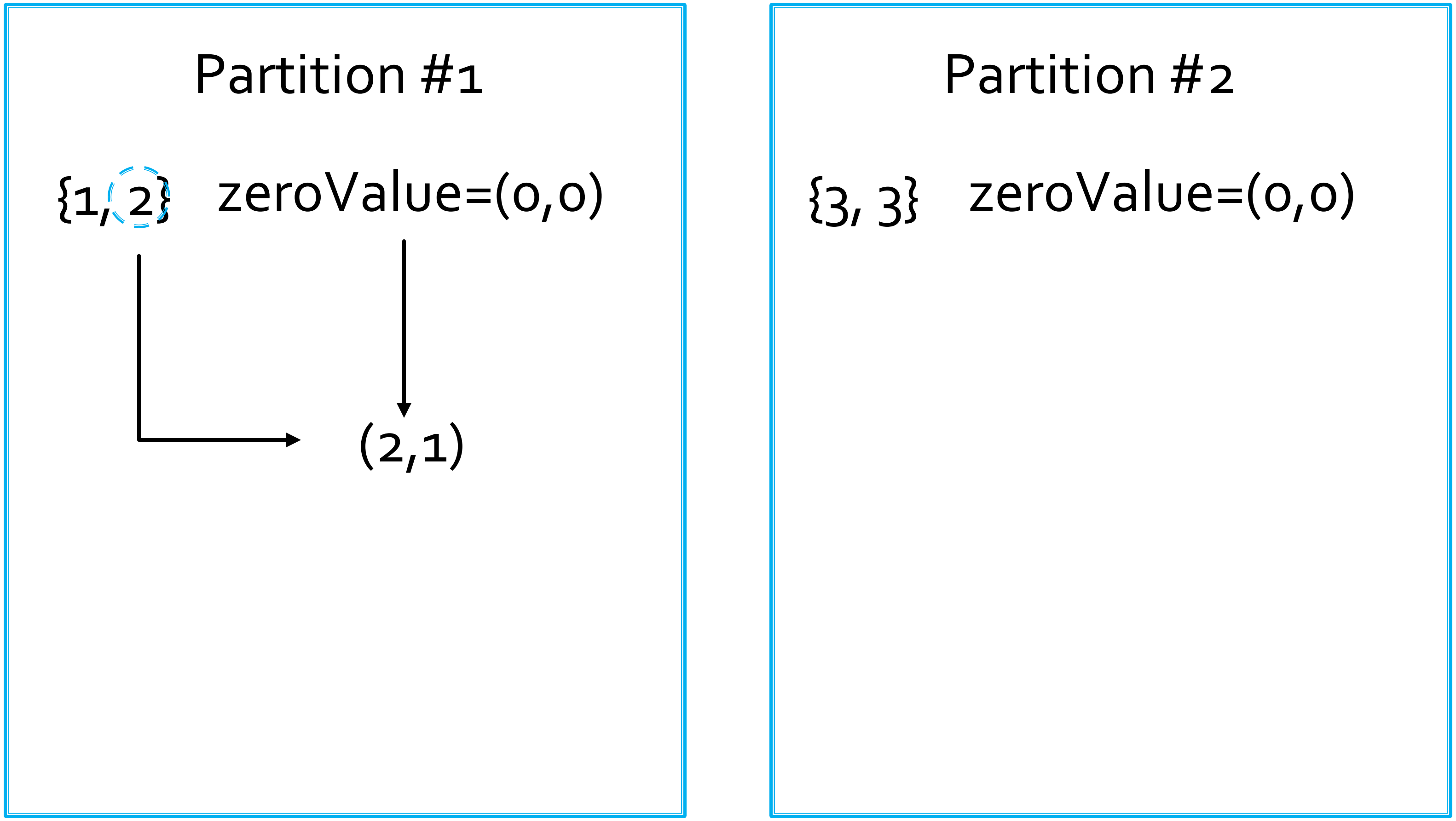
Partition #1

{1, 2} zeroValue=(0,0)

Partition #2

{3, 3} zeroValue=(0,0)

Aggregate action: Simulation



Partition #1

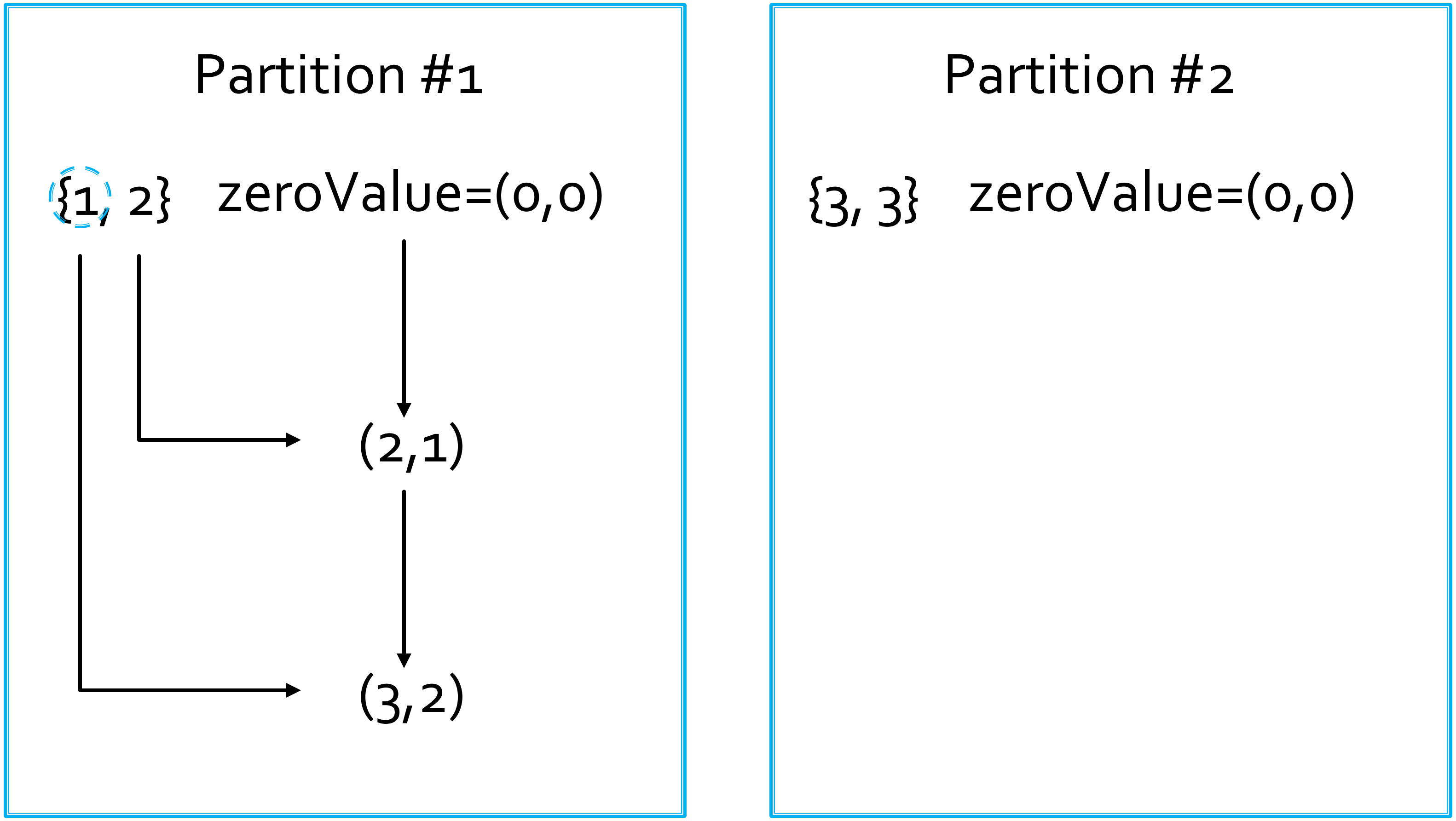
{1, 2} zeroValue=(0,0)

(2,1)

Partition #2

{3, 3} zeroValue=(0,0)

Aggregate action: Simulation



Partition #1

{1, 2} zeroValue=(0,0)

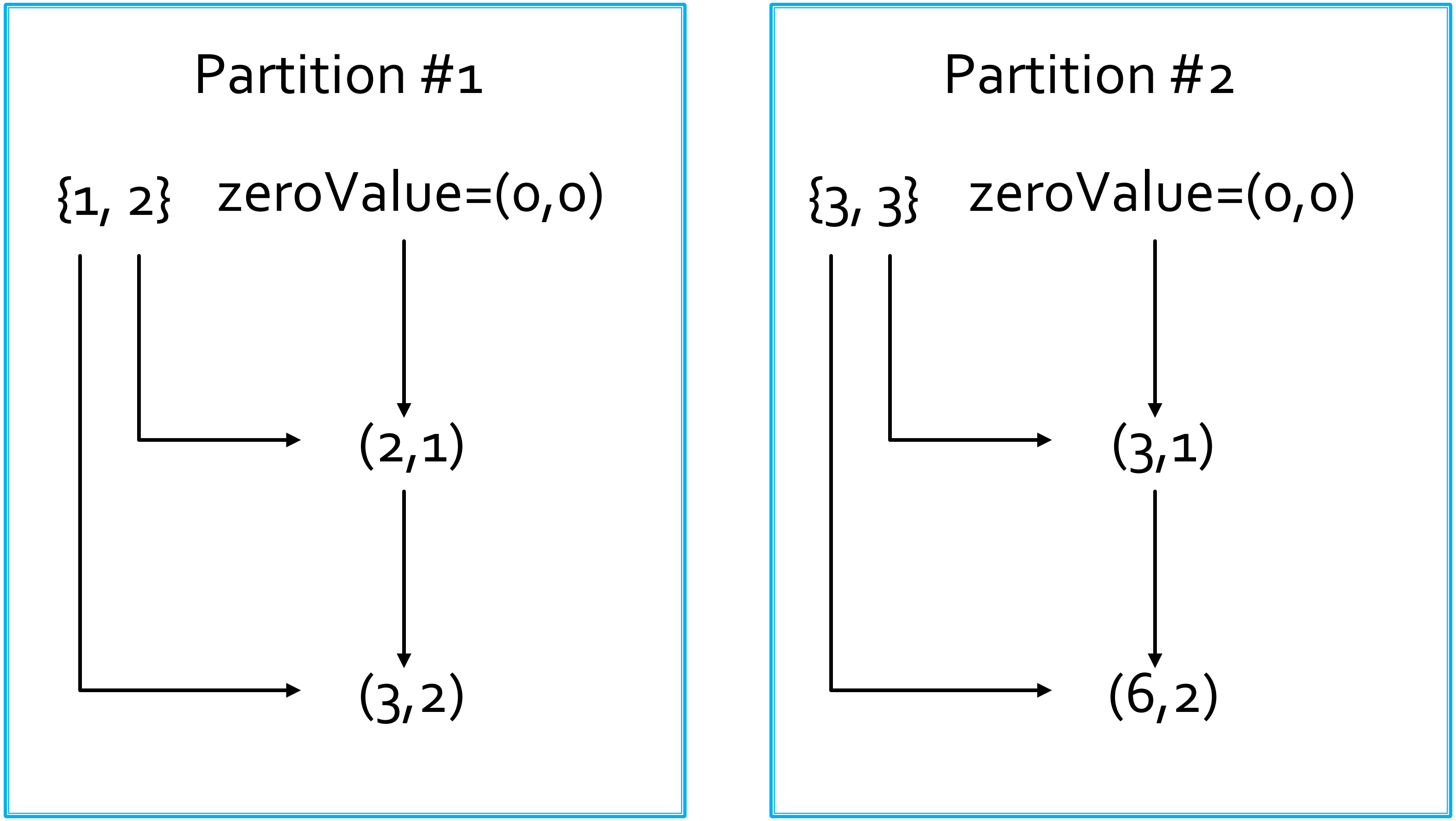
(2,1)

(3,2)

Partition #2

{3, 3} zeroValue=(0,0)

Aggregate action: Simulation



Partition #1

{1, 2} zeroValue=(0,0)

(2,1)

(3,2)

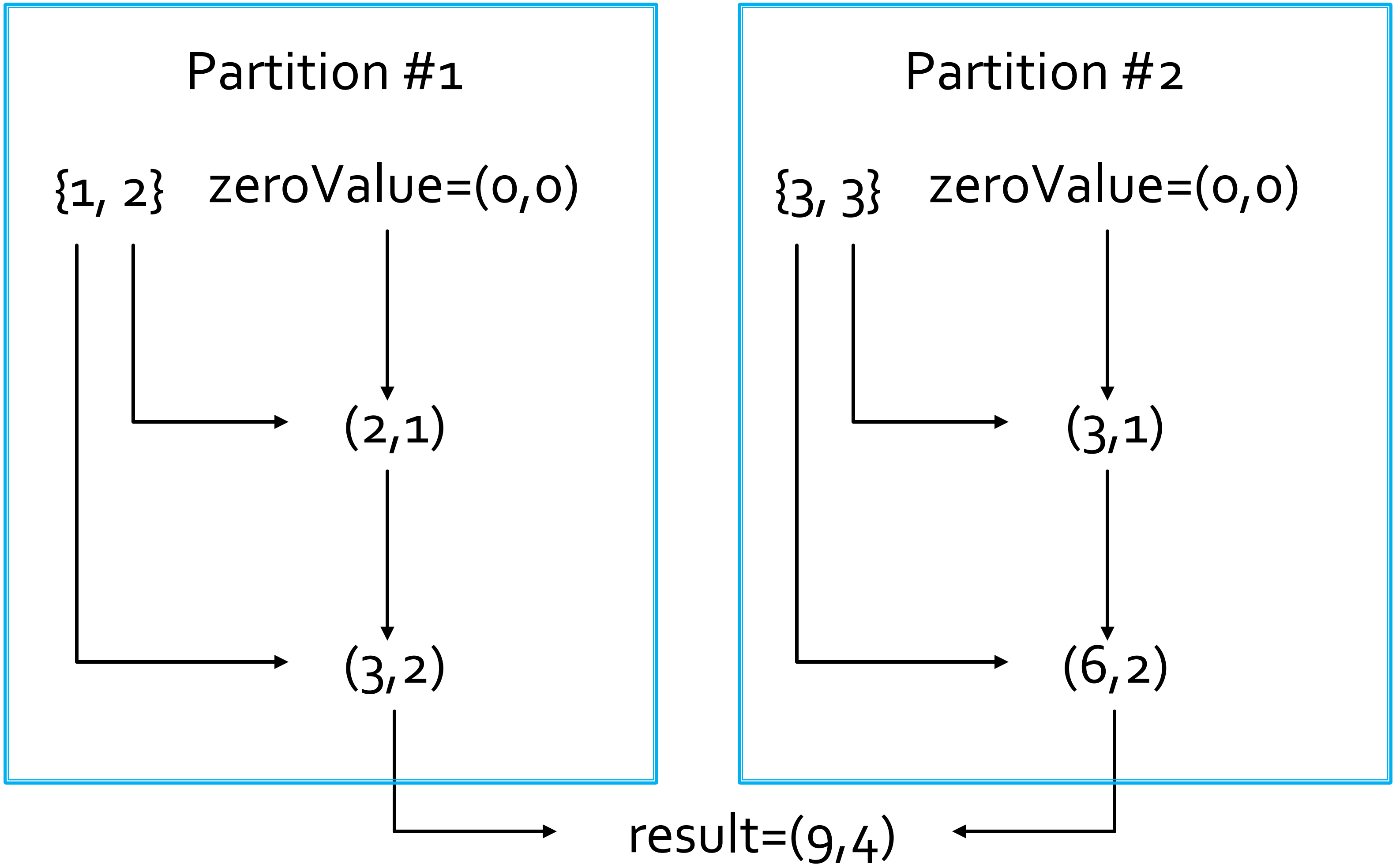
Partition #2

{3, 3} zeroValue=(0,0)

(3,1)

(6,2)

Aggregate action: Simulation



Basic actions: Summary

Basic actions: Summary

- All the examples reported in the following tables are applied on inputRDD that is an RDD of integers containing the following elements (i.e., values)
  - {1, 2, 3, 3}

Basic actions: Summary

Action	Purpose	Example	Result
java.util.List<T> collect()	Return a (Java) List containing all the elements of the RDD on which it is applied. The objects of the RDD and objects of the returned list are objects of the same class.	inputRDD.collect()	{1,2,3,3}
long count()	Return the number of elements of the RDD	inputRDD.count()	4
java.util.Map<T,java.lang.Long> countByValue()	Return a Map object containing the information about the number of times each element occurs in the RDD.	inputRDD. countByValue()	{(1, 1), (2, 1), (3, 2)}

Basic actions: Summary

Action	Purpose	Example	Result
java.util.List<T> take(int n)	Return a (Java) 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}
T first()	Return the first element of the RDD	first()	{1}
java.util.List<T> top(int n)	Return a (Java) 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}

Basic actions: Summary

Action	Purpose	Example	Result
java.util.List<T> takeSample(boolean withReplacement, int n, [long seed])	Return a (Java) 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)	Nondeterministic
T reduce(Function2<T, T, T> f)	Return a single Java 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

Basic actions: Summary

Action	Purpose	Example	Result
T fold(T zeroValue, Function2<T,T,T> f)	Same as reduce but with the provided zero value.	The passed "function" is the sum function and the passed zeroValue is 0	9
<U> U aggregate(U zeroValue, Function2<U,T,U> seqOp, Function2<U,U,U> combOp)	Similar to reduce() but 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)