Big data: architectures and data analytics

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

4

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);
```

5

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
- 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., a file, a table of a database, …)
  - By parallelizing a local collection of object created in the Driver (e.g., a Java collection)

9

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

10

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 s 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

13

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);
```

14

Create RDDs from a local collection

- An RDD can be build from a "local" Java collection
  - 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

15

Create RDDs from a local 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);
```

16

Create RDDs from a local 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

17

Create RDDs from a local 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

18

Create RDDs from a local 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 );
```

19

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

20

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);
```

21

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

22

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

23

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

- The **collect()** method of the **JavaRDD** class
  - The method is invoked on the RDD that we want to "retrieve"
  - The method 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**

24

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
// Each element of the RDD becomes one element
// of the local Java collection
List<String> contentOfLines=lines.collect();
```

25

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
// Each element of the RDD becomes one element
// of the local Java collection
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

26

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 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" the result in a new RDD

29

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

30

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 recompute the content of an RDD when a failure occurs

31

RDD operations

- The lineage graph is also useful for optimization purposes
  - When the content of an RDD is needed, Spark considers the chain of transformations that are applied to compute the content of the RDD and decides how to execute the chain of transformations
    - Spark can change the order of some transformations or merge some of them based on its optimization engine

32

RDD operations

- Actions
  - Are operations that
    - Return a result to the Driver program
      - i.e., a local (Java) variable
      - **Pay attention to the size of the returned result** because it 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

33

Example of lineage graph (DAG)

- Consider the following code

```
// 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");
    }
});
```

34

Example of lineage graph (DAG)

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

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

35

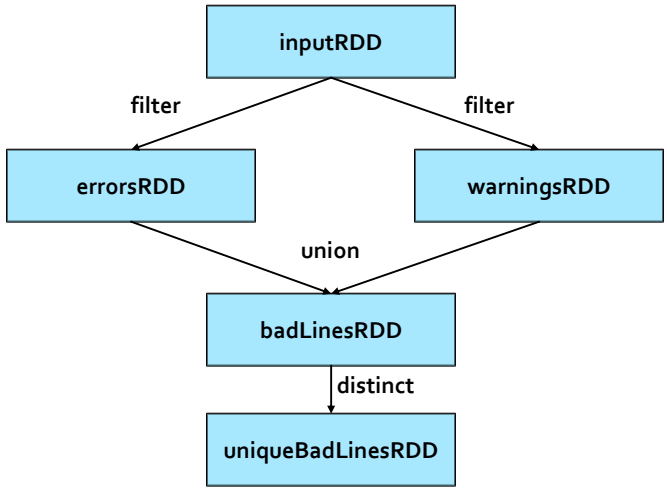
Example of lineage graph (DAG)

```
// 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
int numBadLines= uniqueBadLinesRDD.count();
```

36

Example of lineage graph (DAG)



37

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
  - Hence, they are computed lazily
- Also textFile() is computed lazily
  - However, it is not a transformation because it is not applied on an RDD

38

Example of lineage graph (DAG)

- Spark, similarly to an SQL optimizer, can optimize the "execution" of the transformations
  - For instance, in this case the two filters + union + distinct can be automatically transformed in a single filter applying the constraint
    - The element contains the string "error" or "warning"
  - This optimization improves the efficiency of the application

39

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

41

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
- There are many Spark's function interfaces
  - The difference is given by the data types of the analyzed and returned data

42

Some Spark's function interfaces

Spark's function interface name	Method to implement	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 zero or more objects of type R

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

43

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

44

Solution based on a named class

```
// 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());
```

45

Solution based on a named class

```
// 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());
```

46

Solution based on a named class

```
// 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());
```

47

Solution based on a named class

```
// 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());
```

48

Solution based on an anonymous class

```
// 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");
        }
    }
);
```

49

Solution based on an anonymous class

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");
        }
    }
);
```

50

Solution based on an anonymous class

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");
        }
    }
);
```

51

Named classes vs Anonymous classes

- The two solutions are equivalent in terms of efficiency
  - Use the approach that you prefer
- The named class version is usually preferred when you want to pass something to the constructor of the new class
  - The code is more readable

52

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()

54

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

55

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

57

Filter transformation

- An object of a class implementing 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 of the "input" RDD
      - If the condition is satisfied then call method returns true
      - Otherwise, it returns false

58

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

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

Filter transformation: Example 1

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

We are working with Strings.
Hence, the data type T of the Function<T, Boolean> interface we are implementing is String

61

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

62

Filter transformation: Example 2

```
// Define a class implementing the Function interface
class GreaterThan2 implements Function<Integer, Boolean> {
	// Implement the call method
	public Boolean call(Integer element) {
		if (element>2)
			return true;
		else
			return false;
	}
}
.........

// 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(new GreaterThan2());
```

63

Filter transformation: Example 2

```
// Define a class implementing the Function interface
class GreaterThan2 implements Function<Integer, Boolean> {
	// Implement the call method
	public Boolean call(Integer element) {
		if (element>2)
			return true;
		else
			return false;
	}
}
.........

// 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(new GreaterThan2());
```

We are working with Integers.
Hence, the data type T of the Function<T, Boolean> interface we are implementing is Integer

64

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 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

66

Map transformation

- Method
  - The map transformation is based on the JavaRDD<R> map(Function<T, R>) method of the JavaRDD<T> class
  - An object of a class implementing 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 on each element of the "input" RDD to create the elements of the returned RDD
        - For each element of the "input" RDD one single new element is returned by the call method

67

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

68

Map transformation: Example 1

```
// Define a class implementing the Function interface
class LengthClass implements Function<String, Integer> {
	// Implement the call method
	public Integer call(String element) {
		return new Integer(element.length());
	}
}
.........

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

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

69

Map transformation: Example 1

```
// Define a class implementing the Function interface
class LengthClass implements Function<String, Integer> {
	// Implement the call method
	public Integer call(String element) {
		return new Integer(element.length());
	}
}
.........

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

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

The input RDD is an RDD of Strings.

70

Map transformation: Example 1

```
// Define a class implementing the Function interface
class LengthClass implements Function<String, Integer> {
    // Implement the call method
    public Integer call(String element) {
        return new Integer(element.length());
    }
}
.........

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

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

The new RDD is an RDD of Integers.

71

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

72

Map transformation: Example 2

```
// Define a class implementing the Function interface
class SquareClass implements Function<Integer, Integer> {
	// Implement the call method
	public Integer call(Integer element) {
		return new Integer(element*element);
	}
}
.........

// 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(new SquareClass());
```

73

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

75

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 merge of the lists obtained by applying f over all the elements of the input RDD
- The data type of y can be different from the data type of x
  - i.e., R and T can be different

76

FlatMap transformation

- Method
  - The flatMap transformation is based on the **JavaRDD<R> flatMap(FlatMapFunction<T, R>)** method of the **JavaRDD<T>** class
  - An object of a class implementing 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 included in the 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

77

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

78

FlatMap transformation: Example 1

```
// Define a class implementing the Function interface
class ExtractWords FlatMapFunction<String, String> {
	// Implement the call method
	public Iterable<String> call(String x) {
		return Arrays.asList(x.split(" "));
	}
}
.........

// 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(new
ExtractWords());
```

79

FlatMap transformation: Example 1

```
// Define a class implementing the Function interface
class ExtractWords FlatMapFunction<String, String> {
	// Implement the call method
	public Iterable<String> call(String x) {
		return Arrays.asList(x.split(" "));
	}
}
.....
```

In this case the call method returns a "list" of values

```
// 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(new
ExtractWords());
```

80

FlatMap transformation: Example 1

```
// Define a class implementing the Function interface
class ExtractWords FlatMapFunction<String, String> {
    // Implement the call method
    public Iterable<String> call(String x) {
        return Arrays.asList(x.split(" "));
    }
}
.........

// 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(new
ExtractWords());
```

The new RDD contains the merge of the lists obtained by applying the call method over all the elements of inputRDD

81

FlatMap transformation: Example 1

```
// Define a class implementing the Function interface
class ExtractWords FlatMapFunction<String, String> {
    // Implement the call method
    public Iterable<String> call(String x) {
        return Arrays.asList(x.split(" "));
    }
}
.........

// 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(new
ExtractWords());
```

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

82

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

84

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

85

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();
```

86

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

87

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();
```

88

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])

90

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%)

91

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);
```

92

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

93

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);
```

94

Set transformations

Set transformations

- Spark provides also a set of transformations that operate on two input RDDs
- Some of them implement some of the standard set transformations
  - Union
  - Intersection
  - Subtract
  - Cartesian

Set transformations

- All these transformations
  - Operate on two input RDDs
    - One is the RDD on which the method is invoked
    - The other RDD is passed as parameter to the method
- and an 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

97

Union transformation

- The union transformation is based on the JavaRDD<T> union(JavaRDD<T> secondInputRDD) method of the JavaRDD<T> class
  - **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 → It is a very expensive operation

98

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

Intersection transformation

- The intersection transformation is based on the JavaRDD<T> intersection(JavaRDD<T> secondInputRDD) method of the JavaRDD<T> class

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 and not in the RDD passed as parameter

101

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 have a different data type
  - The returned RDD is an RDD of pairs (JavaPairRDD) containing all the combinations composed of one element of the first input RDD and the second input RDD

102

Set transformations: Example 1

- Create two RDDs integers
  - inputRDD1 contains the values {1, 2, 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

103

Set transformations: Example 1

```
// Create two RDD of integers.
List<Integer> inputList1 = Arrays.asList(1, 2, 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.interectio(inputRDD2);

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

104

Cartesian transformation: Example 1

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

105

Cartesian transformation: Example 1

```
// Create two RDD of integers.
List<Integer> inputList1 = Arrays.asList(1, 2, 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
JavaRDD<Integer> outputCartesianRDD =
                    inputRDD1.cartesian(inputRDD2);
```

106

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

107

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);
```

108

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}

111

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}

112

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

113

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, 3}
  - inputRDD2 {3, 4, 5}

114

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, 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}

115

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}
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)}

116