How to write MapReduce programs in Hadoop

Hadoop implementation of MapReduce

2

MapReduce and Hadoop

- Designers/Developers focus on the definition of the Map and Reduce functions (i.e., m and r)
  - No need to manage the distributed execution of the map, shuffle and sort, and reduce phases
- The Hadoop framework coordinates the execution of the MapReduce program
  - Parallel execution of the map and reduce phases
  - Execution of the shuffle and sort phase
  - Scheduling of the subtasks
  - Synchronization

3

MapReduce programs

- The programming language is Java
- A Hadoop MapReduce program consists of three main parts
  - Driver
  - Mapper
  - Reducer
- Each part is "implemented" by means of a specific class

4

Terminology

- Driver class
  - The class containing the method/code that coordinates the configuration of the job and the "workflow" of the application
- Mapper class
  - A class "implementing" the map function
- Reducer class
  - A class "implementing" the reduce function
- Driver
  - Instance of the Driver class (i.e., an object)
- Mapper
  - Instance of the Mapper class (i.e., an object)
- Reducer
  - Instance of the Reducer class (i.e., an object)

5

Terminology

- (Hadoop) Job
  - Execution/run of a MapReduce code over a data set
- Task
  - Execution/run of a Mapper (Map task) or a Reducer (Reduce task) on a slice of data
  - Many tasks for each job
- Input split
  - Fixed-size piece of the input data
    - Usually each split has approximately the same size of a HDFS block/chunk

6

Driver

- The Driver
  - Is characterized by the main() method, which accepts arguments from the command line
    - i.e., it is the entry point of the application
  - Configures the job
  - Submits the job to the Hadoop Cluster
  - "Coordinates" the work flow of the application
  - Runs on the client machine
    - i.e., it does not run on the cluster

7

Mapper

- The Mapper
  - Is an instance of the Mapper class
  - "Implements" the map phase
  - Is characterized by the map(…) method
    - Processes the (key, value) pairs of the input file and emits (key, value) pairs
    - Is invoked one time for each input (key, value) pair
  - Runs on the cluster

8

Reducer

- The Reducer
  - Is an instance of the Reduce class
  - "Implements" the reduce phase
  - Is characterized by the reduce(…) method
    - Processes (key, [list of values]) pairs and emits (key, value) pairs
    - Is invoked one time for each distinct key
  - Runs on the cluster

9

Hadoop implementation of the MapReduce phases

- Input key-value pairs are read from the HDFS file system
- The map method of the Mapper
  - Is invoked over each input key-value pair
  - Emits a set of intermediate key-value pairs that are stored in the local file system of the computing server (they are not stored in HDFS)
- Intermediate results
  - Are aggregated by means of a shuffle and sort procedure
  - A set of (key, [list of values]) pairs is generated
    - One (key, [list of values]) for each distinct key

10

Hadoop implementation of the MapReduce phases

- The reduce method of the Reduder
  - Is applied over each (key, [list of values]) pair
  - Emits a set of key-value pairs that are stored in HDFS (the final result of the MapReduce application)
- Intermediate key-value pairs are transient:
  - They are not stored on the distributed files system
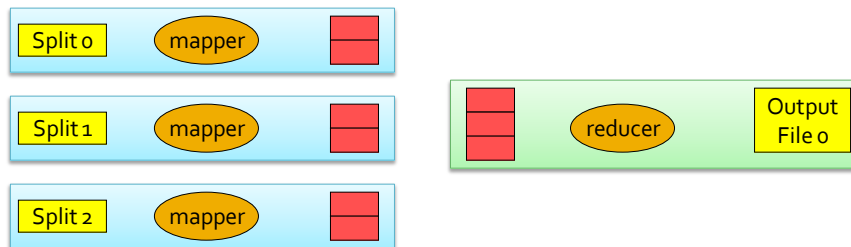 - They are stored locally to the node producing or processing them

11

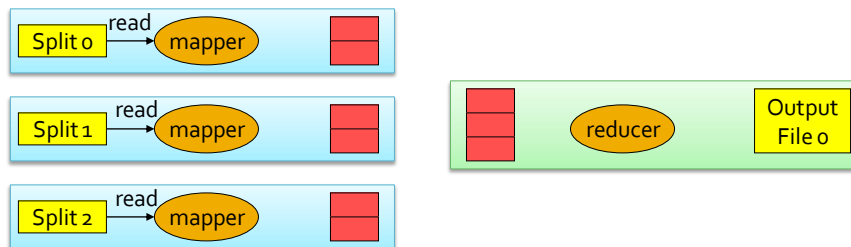## Hadoop implementation of the MapReduce phases

- In order to parallelize the work/the job, Hadoop executes a set of tasks in parallel
  - It instantiates one Mapper (Task) for each input split
  - And a user-specified number of Reducers
    - Each reducer is associated with a set of keys
      - It receives and processes all the key-value pairs associated with its set of keys
  - Mappers and Reducers are executed on the nodes/servers of the clusters
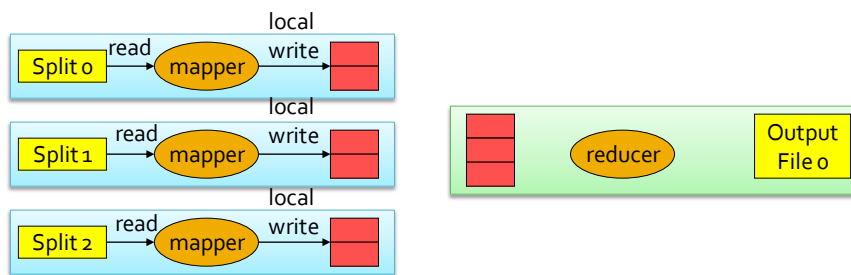
12

MapReduce data flow with a single reducer

MapReduce data flow with a single reducer

MapReduce data flow with a single reducer

Split 0 —read→ mapper —local write→ [ ]

Split 1 —read→ mapper —local write→ [ ]

Split 2 —read→ mapper —local write→ [ ]

[ ] reducer Output File 0

MapReduce data flow with a single reducer

Split 0 —read→ mapper —local write→ [ ]

Split 1 —read→ mapper —local write→ [ ]

Split 2 —read→ mapper —local write→ [ ]

[ ] reducer Output File 0
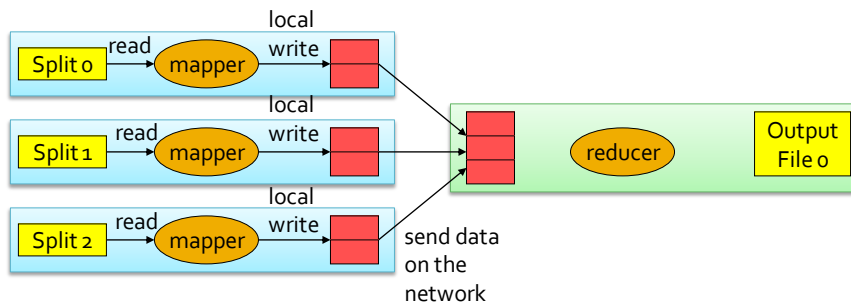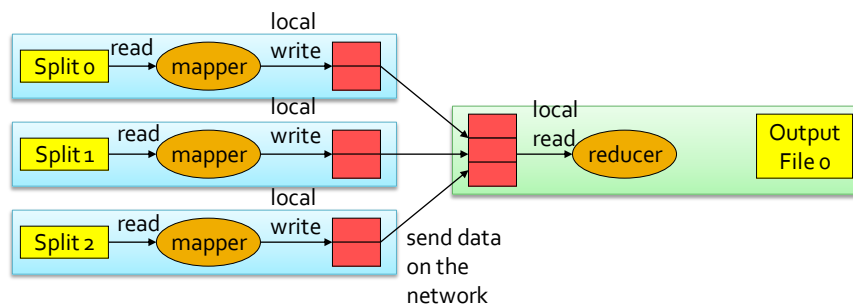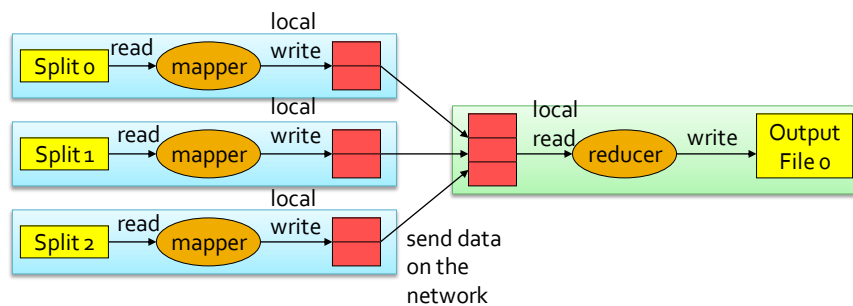
send data on the network

MapReduce data flow with a single reducer

MapReduce data flow with a single reducer

MapReduce data flow with a single reducer

Temporary files
(local file system)

Input Data
(**HDFS** file system)

Output Data
(**HDFS** file system)

| Split 0 | read → mapper → local write | | local read → reducer → write | Output File 0 |

send data
on the network

MapReduce data flow with multiple reducers

Split 0	read → mapper → local write
Split 1	read → mapper → local write
Split 2	read → mapper → local write

local read → reducer → write → Output File 0

local read → reducer → write → Output File 1

send data
on the network
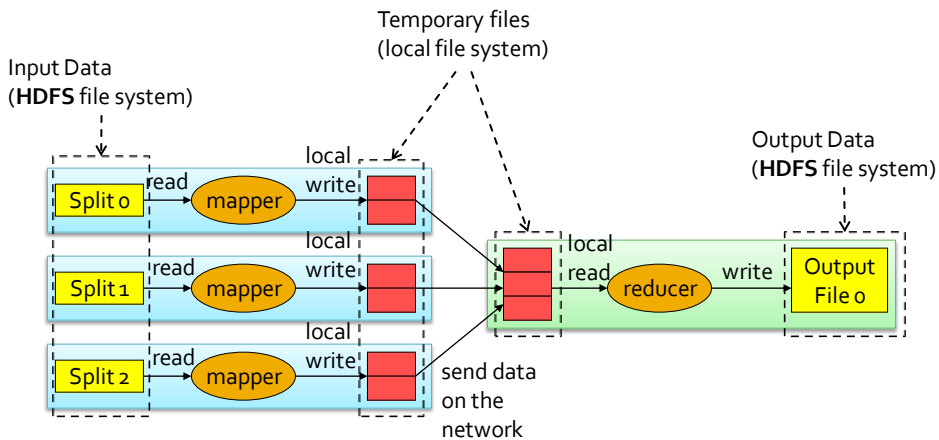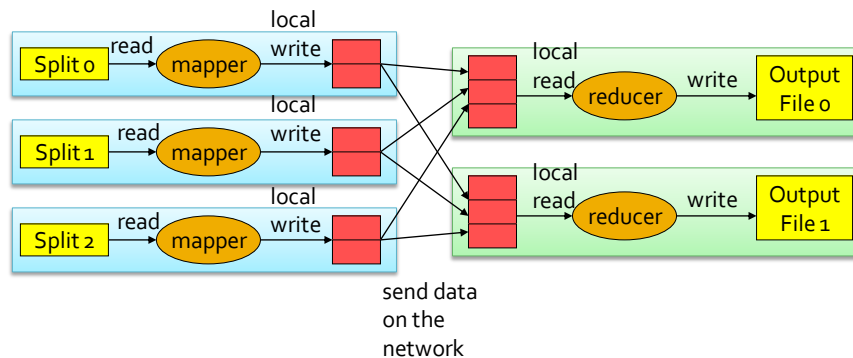
MapReduce data flow with multiple reducers
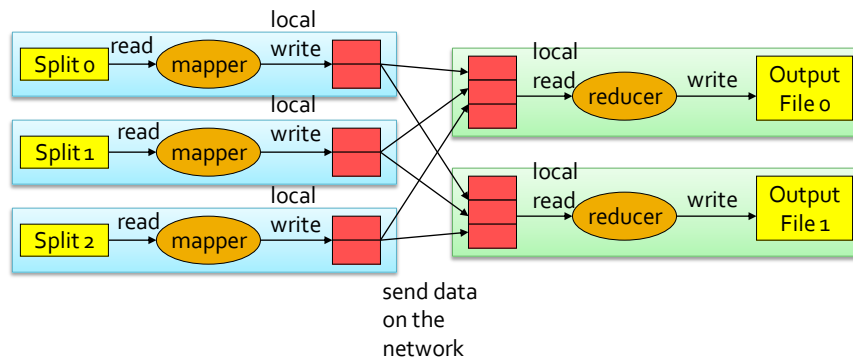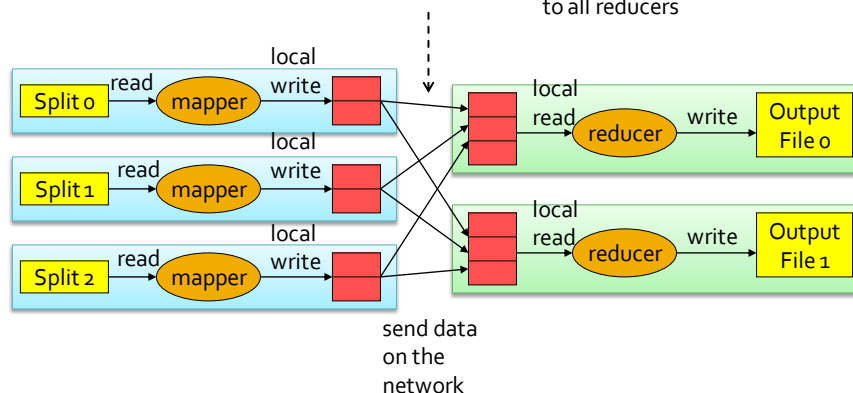
Each key is assigned to/managed by one reducer

MapReduce data flow with multiple reducers

Each key is assigned to/managed by one reducer

Potentially, all mappers send data (a set of (key,value) pairs) to all reducers

MapReduce programs - Driver

- The Driver class extends the org.apache.hadoop.conf.Configured class and implements the org.apache.hadoop.util.Tool interface
  - You can write a Driver class that does not extend Configured and does not implement Tool
    - However, you need to manage some low level details related to some command line parameters in that case
- The designer/developer implements the **main**(…) and **run**(…) methods

23

MapReduce programs - Driver

- The **run**(…) method
  - Configures the job
    - Name of the Job
    - Job Input format
    - Job Output format
    - Mapper class
      - Name of the class
      - Type of its input (key, value) pairs
      - Type of its output (key, value) pairs

24

MapReduce programs - Driver

- Reducer class
  - Name of the class
  - Type of its input (key, value) pairs
  - Type of its output (key, value) pairs
- Number of reducers

25

MapReduce programs - Mapper

- The Mapper class extends the org.apache.hadoop.mapreduce.Mapper class
  - The org.apache.hadoop.mapreduce.Mapper class
    - Is a generic type/generic class
    - With four type parameters: input key type, input value type, output key type, output value type
- The designer/developer implements the map(…) method
  - That is automatically called by the framework for each (key, value) pair of the input file

26

MapReduce programs - Mapper

- The **map**(…) method
  - Processes its input (key, value) pairs by using standard Java code
  - Emits (key, value) pairs by using the context.write(key, value) method

27

MapReduce programs - Reducer

- The Reducer class extends the org.apache.hadoop.mapreduce.Reducer class
  - The org.apache.hadoop.mapreduce.Reducer class
    - Is a generic type/generic class
    - With four type parameters: input key type, input value type, output key type, output value type
- The designer/developer implements the **reduce**(…) method
  - That is automatically called by the framework for each (key, [list of values]) pair obtained by aggregating the output of the mapper(s)

28

MapReduce programs - Reducer

- The **reduce**(…) method
  - Processes its input (key, [list of values]) pairs by using standard Java code
  - Emits (key, value) pairs by using the context.write(key, value) method

29

MapReduce Data Types

- Hadoop has its own basic data types
  - Optimized for network serialization
  - org.apache.hadoop.io.Text: like Java String
  - org.apache.hadoop.io.IntWritable: like Java Integer
  - org.apache.hadoop.io.LongWritable: like Java Long
  - org.apache.hadoop.io.FloatWritable : like Java Float
  - Etc

30

MapReduce Data Types

- The basic Hadoop data types implement the org.apache.hadoop.io.Writable and org.apache.hadoop.io.WritableComparable interfaces
- All classes (data types) used to represent keys are instances of WritableComparable
  - Keys must be "comparable" for supporting the sort and shuffle phase
- All classes (data types) used to represent values are instances of Writable
  - Usually, they are also instances of WritableComparable even if it is not indispensable

31

MapReduce Data Types

- Developers can define new data types by implementing the org.apache.hadoop.io.Writable and/or org.apache.hadoop.io.WritableComparable interfaces
  - It is useful for managing complex data types

32

16

InputFormat

- The input of the MapReduce program is an HDFS file (or an HDFS folder)
- While the input of the Mapper is a set of (key, value) pairs
- The classes extending the org.apache.hadoop.mapreduce.InputFormat abstract class are used to read the input data and "logically transform" the input HDFS file in a set of (key, value) pairs

33

InputFormat

- InputFormat "describes" the input-format specification for a MapReduce application and processes the input file(s)
- The InputFormat class is used to
  - Read input data and validate the compliance of the input file with the expected input-format
  - Split the input file(s) into logical Input Splits
    - Each input split is then assigned to an individual Mapper
  - Provide the RecordReader implementation to be used to divide the logical input split in a set of (key,value) pairs (also called records) for the mapper

34

Getting Data to the Mapper

Input (HDFS) file

Input Format

Input Split	Input Split	Input Split	Input Split
Iterator	Iterator	Iterator	Iterator
Record Reader	Record Reader	Record Reader	Record Reader
Mapper	Mapper	Mapper	Mapper

35

Getting Data to the Mapper

Input (HDFS) file

Input Format

Input Split	Input Split	Input Split	Input Split
Iterator	Iterator	Iterator	Iterator
Record Reader	Record Reader	Record Reader	Record Reader
Mapper	Mapper	Mapper	Mapper

Sets of (key, value) pairs
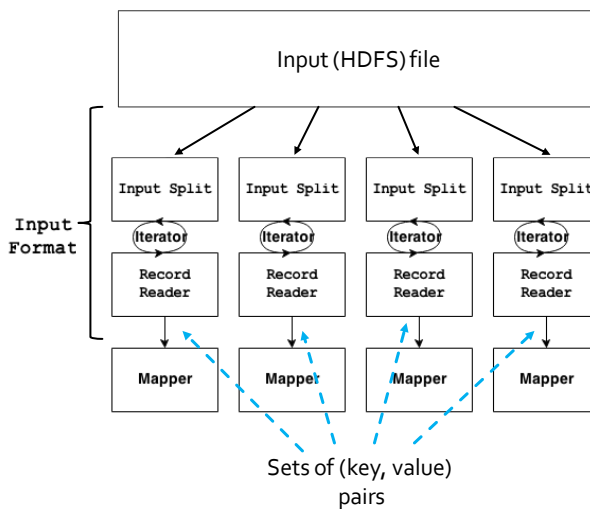
36

Reading Data

- InputFormat identifies partitions of the data that form an input split
  - Each input split is a (reference to a) part of the input file processed by a single mapper
  - Each split is divided into records, and the mapper processes one record (i.e., a (key,value) pair) at a time

37

InputFormat

- A set of predefined classes extending the InputFormat abstract class are available for standard input file formats
  - TextInputFormat
    - An InputFormat for plain text files
  - KeyValueTextInputFormat
    - Another InputFormat for plain text files
  - SequenceFileInputFormat
    - An InputFormat for sequential/binary files
  - ....

38

TextInputFormat

- TextInputFormat
  - An InputFormat for plain text files
  - Files are broken into lines
    - Either linefeed or carriage-return are used to signal end of line
  - One pair (key, value) is emitted for each line of the file
    - Key is the position (offset) of the line in the file
    - Value is the content of the line

39

TextInputFormat example

Input HDFS file

Toy example file for Hadoop.\n
Hadoop running example.\n
TextinputFormat is used to split data.\n

(key, value) pairs generated by using TextInputFormat

(0, "Toy example file for Hadoop.")
(31, "Hadoop running example.")
(56, "TextinputFormat is used to split data.")

40

KeyValueTextInputFormat

- KeyValueTextInputFormat
  - An InputFormat for plain text files
    - Each line of the file must have the format
      key<separator>value
    - The default separator is tab (\t)
  - Files are broken into lines
    - Either linefeed or carriage-return are used to signal end of line
    - Each line is split into key and value parts by considering the separator symbol/character
  - One pair (key, value) is emitted for each line of the file
    - Key is the text preceding the separator
    - Value is the text following the separator

41

KeyValueTextInputFormat

Input HDFS file

10125\tMister John\n
10236\tMiss Jenny\n
1\tMister Donald Duck\n

(key, value) pairs generated by using KeyValueTextInputFormat

(10125, "Mister John")
(10236, "Miss Jenny")
(1, "Mister Donald Duck")

42

OutputFormat

- The classes extending the org.apache.hadoop.mapreduce.OutputForm at abstract class are used to write the output of the MapReduce program in HDFS

43

OutputFormat

- A set of predefined classes extending the OutputFormat abstract class are available for standard output file formats
  - TextOutputFormat
    - An OutputFormat for plain text files
  - SequenceFileOutputFormat
    - An OutputFormat for sequential/binary files
  - ....

44

TextOutputFormat

- TextOutputFormat
  - An OutputFormat for plain text files
  - For each output (key, value) pair TextOutputFormat writes one line in the output file
  - The format of each output line is
    key\tvalue\n

45

Structure of a MapReduce program in Hadoop

46

Basic structure of a MapReduce program - Driver (1)

```
/* Set package */
package it.polito.bigdata.hadoop.mypackage;

/* Import libraries */
import java.io.IOException;

import org.apache.hadoop.mapreduce.Job;
import org.apache.hadoop.util.Tool;
import org.apache.hadoop.util.ToolRunner;
import org.apache.hadoop.conf.Configuration;
import org.apache.hadoop.conf.Configured;
Import org.apache.hadoop.io.*;
import org.apache.hadoop.mapreduce.lib.input.TextInputFormat;
import org.apache.hadoop.mapreduce.lib.output.TextOutputFormat;
.....
```

47

Basic structure of a MapReduce program - Driver (2)

```
/* Driver class */
public class MapReduceAppDriver extends Configured implements
    Tool {
 @Override
 public int run(String[] args) throws Exception {
   /* variables */
   int exitCode;
   .....

   // Parse parameters
   numberOfReducers = Integer.parseInt(args[0]);
   inputPath = new Path(args[1]);
   outputDir = new Path(args[2]);
   ....
```

48

Basic structure of a MapReduce program - Driver (3)

```
// Define and configure a new job
Configuration conf = this.getConf();
Job job = Job.getInstance(conf);

// Assign a name to the job
job.setJobName("My First MapReduce program");
```

49

Basic structure of a MapReduce program - Driver (4)

```
// Set path of the input file/folder (if it is a folder, the job reads all
the files in the specified folder) for this job
FileInputFormat.addInputPath(job, inputPath);

// Set path of the output folder for this job
FileOutputFormat.setOutputPath(job, outputDir);

// Set input format
// TextInputFormat = textual files
job.setInputFormatClass(TextInputFormat.class);

// Set job output format
job.setOutputFormatClass(TextOutputFormat.class);
```

50

Basic structure of a MapReduce program - Driver (5)

```
// Specify the class of the Driver for this job
job.setJarByClass(MapReduceAppDriver.class);

// Set mapper class
job.setMapperClass(MyMapperClass.class);

// Set map output key and value classes
job.setMapOutputKeyClass(output key type.class);
job.setMapOutputValueClass(output value type.class);
```

51

Basic structure of a MapReduce program - Driver (6)

```
// Set reduce class
job.setReducerClass(MyReducerClass.class);

// Set reduce output key and value classes
job.setOutputKeyClass(output key type.class);
job.setOutputValueClass(output value type.class);

// Set number of reducers
job.setNumReduceTasks(numberOfReducers);
```

52

Basic structure of a MapReduce program - Driver (7)

```
    // Execute the job and wait for completion
    if (job.waitForCompletion(true)==true)
        exitCode=0;
    else
        exitCode=1;
    return exitCode;
}   // End of the run method
```

53

Basic structure of a MapReduce program - Driver (8)

```
/* main method of the driver class   */
   public static void main(String args[]) throws Exception {
    /* Exploit the ToolRunner class to "configure" and run the
       Hadoop application */

    int res = ToolRunner.run(new Configuration(),
                       new MapReduceAppDriver(), args);

    System.exit(res);
   } // End of the main method

} // End of public class MapReduceAppDriver
```

54

Basic structure of a MapReduce program - Mapper (1)

```
/* Set package */
package it.polito.bigdata.hadoop.mypackage;

/* Import libraries */
import java.io.IOException;

import org.apache.hadoop.mapreduce.Mapper;
import org.apache.hadoop.io.*;
.....
```

55

Basic structure of a MapReduce program - Mapper (2)

```
/* Mapper Class */
class myMapperClass extends Mapper<
            MapperInputKeyType,      // Input key type (must be
    consistent with the InputFormat class specified in the Driver)
            MapperInputValueType,    // Input value type (must be
    consistent with the InputFormat class specified in the Driver)
            MapperOutputKeyType,    // Output key type
            MapperOutputValueType> // Output value type
{
```

56

Basic structure of a MapReduce program - Mapper (3)

```
/* Implementation of the map method */
protected void map(
    MapperInputKeyType key,        // Input key
    MapperInputValueType value,    // Input value
    Context context) throws IOException, InterruptedException {

    /* Process the input (key, value) pair and
       emit a set of (key, value) pairs.
       context.write(..) is used to emit (key, value) pairs
       context.write(new outputkey, new outputvalue); */
} // End of the map method

} // End of class myMapperClass
```

57

Basic structure of a MapReduce program - Reducer (1)

```
/* Set package */
package it.polito.bigdata.hadoop.mypackage;

/* Import libraries */
import java.io.IOException;

import org.apache.hadoop.mapreduce.Reducer;
Import org.apache.hadoop.io.*;
.....
```

58

Basic structure of a MapReduce program - Reducer (2)

```
/* Reducer Class */
class myReducerClass extends Reducer<
          ReducerInputKeyType,     // Input key type (must be
    consistent with the OutputKeyType of the Mapper)
          ReducerInputValueType,    // Input value type (must be
    consistent with the OutputValueType of the Mapper)
          ReducerOutputKeyType,    // Output key type (must be
    consistent with the OutputFormat class specified in the Driver)
          ReducerOutputValueType> // Output value type (must
    be consistent with the OutputFormat class specified in the Driver)
{
```

59

Basic structure of a MapReduce program - Reducer (3)

```
/* Implementation of the reduce method */
protected void reduce(
    ReducerInputKeyType key,              // Input key
    Iterable<ReducerInputValueType> values, // Input values (list of
values)
    Context context) throws IOException, InterruptedException {

    /* Process the input (key, [list of values]) pair and
       emit a set of (key,value) pairs.
       context.write(..) is used to emit (key, value) pairs
       context.write(new outputkey, new outputvalue); */
} // End of the reduce method

} // End of class myReducerClass
```

60

MapReduce programs in Hadoop: Word Count Example

61

Word Count Example

- Word count problem
  - Input: (unstructured) textual file
    - Each line of the input file can contains a set of words
  - Output: number of occurrences of each word appearing in the input file
- Parameters/arguments of the application:
  - args[0]: number of instances of the reducer
  - args[1]: path of the input file
  - args[2]: path of the output folder

62

Word Count Example

- Input file

  Toy example file for Hadoop. Hadoop running example.

- Output pairs
  (toy, 1)
  (example, 2)
  (file, 1)
  (for, 1)
  (hadoop, 2)
  (running, 1)

63

Word Count Example - Driver (1)

```
/* Set package */
package it.polito.bigdata.hadoop.wordcount;

/* Import libraries */
import org.apache.hadoop.conf.Configuration;
import org.apache.hadoop.conf.Configured;
import org.apache.hadoop.fs.Path;
import org.apache.hadoop.io.IntWritable;
import org.apache.hadoop.io.Text;
import org.apache.hadoop.mapreduce.Job;
import org.apache.hadoop.mapreduce.lib.input.FileInputFormat;
import org.apache.hadoop.mapreduce.lib.input.TextInputFormat;
import org.apache.hadoop.mapreduce.lib.output.FileOutputFormat;
import org.apache.hadoop.mapreduce.lib.output.TextOutputFormat;
import org.apache.hadoop.util.Tool;
import org.apache.hadoop.util.ToolRunner;
```

64

Word Count Example - Driver (2)

```
/* Driver class */
public class WordCount extends Configured implements Tool {
 @Override
 public int run(String[] args) throws Exception {

  Path inputPath;
  Path outputDir;
  int numberOfReducers;
  int exitCode;

  // Parse input parameters
  numberOfReducers = Integer.parseInt(args[0]);
  inputPath = new Path(args[1]);
  outputDir = new Path(args[2]);
```

65

Word Count Example - Driver (3)

```
  // Define and configure a new job
  Configuration conf = this.getConf();
  Job job = Job.getInstance(conf);

  // Assign a name to the job
  job.setJobName("WordCounter");
```

66

Word Count Example - Driver (4)

```
// Set path of the input file/folder (if it is a folder, the job reads all the files
in the specified folder) for this job
FileInputFormat.addInputPath(job, inputPath);

// Set path of the output folder for this job
FileOutputFormat.setOutputPath(job, outputDir);

// Set input format
// TextInputFormat = textual files
job.setInputFormatClass(TextInputFormat.class);

// Set job output format
job.setOutputFormatClass(TextOutputFormat.class);
```

67

Word Count Example - Driver (5)

```
// Specify the class of the Driver for this job
job.setJarByClass(WordCount.class);

// Set mapper class
job.setMapperClass(WordCountMapper.class);

// Set map output key and value classes
job.setMapOutputKeyClass(Text.class);
job.setMapOutputValueClass(IntWritable.class);
```

68

Word Count Example - Driver (6)

```
//Set reduce class
job.setReducerClass(WordCountReducer.class);

//Set reduce output key and value classes
job.setOutputKeyClass(Text.class);
job.setOutputValueClass(IntWritable.class);

//Set number of reducers
job.setNumReduceTasks(numberOfReducers);
```

69

Word Count Example - Driver (7)

```
//Execute the job and wait for completion
if (job.waitForCompletion(true)==true)
     exitCode=0;
else
     exitCode=1;
return exitCode;
}  //End of the run method
```

70

Word Count Example - Driver (8)

```
/* main method of the driver class   */
   public static void main(String args[]) throws Exception {
     /* Exploit the ToolRunner class to "configure" and run the
       Hadoop application */

     int res = ToolRunner.run(new Configuration(),
                              new WordCount(), args);

     System.exit(res);
  } // End of the main method

} // End of public class WordCount
```

71

Word Count Example - Mapper (1)

```
/* Set package */
package it.polito.bigdata.hadoop.wordcount;

/* Import libraries */
import java.io.IOException;

import org.apache.hadoop.io.IntWritable;
import org.apache.hadoop.io.LongWritable;
import org.apache.hadoop.io.Text;
import org.apache.hadoop.mapreduce.Mapper;
```

72

36

Word Count Example - Mapper (2)

```
/* Mapper Class */
class WordCountMapper extends Mapper<
          LongWritable,  // Input key type
          Text,          // Input value type
          Text,          // Output key type
          IntWritable>   // Output value type
{
    /* Implementation of the map method */
    protected void map(
        LongWritable key,   // Input key type
        Text value,         // Input value type
        Context context) throws IOException, InterruptedException {
```

73

Word Count Example - Mapper (3)

```
        // Split each sentence in words. Use whitespace(s) as delimiter
        // The split method returns an array of strings
        String[] words = value.toString().split("\\s+");

        // Iterate over the set of words
        for(String word : words) {
          // Transform word case
          String cleanedWord = word.toLowerCase();

          // emit one pair (word, 1) for each input word
          context.write(new Text(cleanedWord), new IntWritable(1));
        }
    } // End map method

} // End of class WordCountMapper
```

74

37

Word Count Example - Reducer (1)

```
/* Set package */
package it.polito.bigdata.hadoop.wordcount;

/* Import libraries */
import java.io.IOException;

import org.apache.hadoop.io.IntWritable;
import org.apache.hadoop.io.Text;
import org.apache.hadoop.mapreduce.Reducer;
```

75

Word Count Example - Reducer (2)

```
/* Reducer Class */
class WordCountReducer extends Reducer<
        Text,              // Input key type
        IntWritable,       // Input value type
        Text,              // Output key type
        IntWritable>       // Output value type
{
    /* Implementation of the reduce method */
    protected void reduce(
        Text key, // Input key type
        Iterable<IntWritable> values, // Input value type
        Context context) throws IOException, InterruptedException {
```

76

Word Count Example - Reducer (3)

```
/* Implementation of the reduce method */
protected void reduce(
     Text key, // Input key type
     Iterable<IntWritable> values, // Input value type
     Context context) throws IOException, InterruptedException {
           int occurrances = 0;

           // Iterate over the set of values and sum them
           for (IntWritable value : values) {
                 occurrances = occurrances + value.get();
           }
           // Emit the total number of occurrences of the current word
           context.write(key, new IntWritable(occurrances));
     } // End reduce method
} // End of class WordCountReducer
```

77

Combiner

78

Combiner

- "Standard" MapReduce applications
  - The (key,value) pairs emitted by the Mappers are sent to the Reducers through the network
- Some "pre-aggregations" could be performed to limit the amount of network data by using Combiners (also called "mini-reducers")
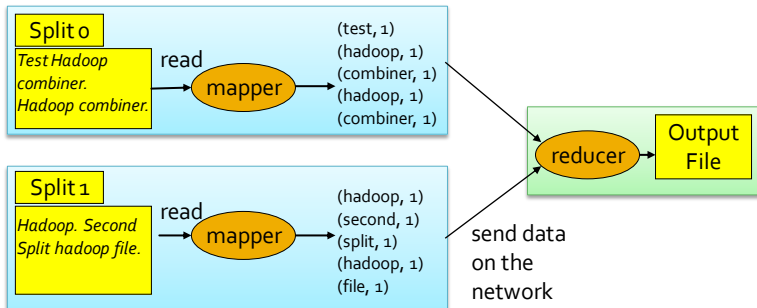
79

Combiner – Word count example

- Consider the standard word count problem
- Suppose the input file is split in two input splits
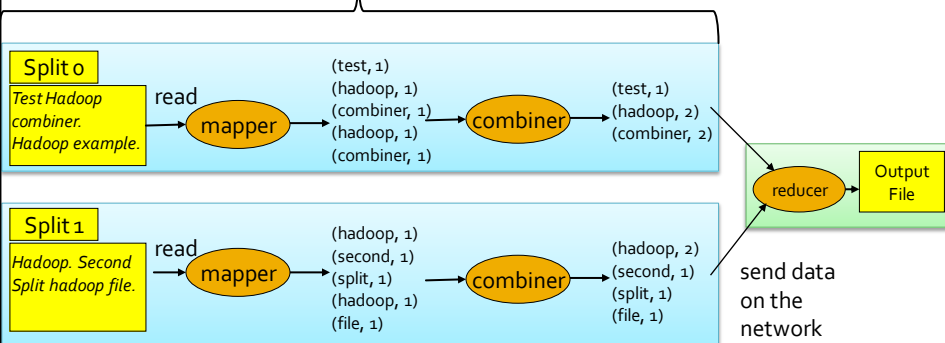 - Hence, two Mappers are instantiated (one for each split)

80

Word count example without combiner

Split 0

Test Hadoop combiner. Hadoop combiner.

read → mapper →
(test, 1)
(hadoop, 1)
(combiner, 1)
(hadoop, 1)
(combiner, 1)

Split 1

Hadoop. Second Split hadoop file.

read → mapper →
(hadoop, 1)
(second, 1)
(split, 1)
(hadoop, 1)
(file, 1)

send data on the network

reducer → Output File

81

Word count example with combiner

The combiner is **locally** called on the output (key, value) pairs of the mapper (it works on data stored in the main-memory or on the local hard disks)

Split 0

Test Hadoop combiner. Hadoop example.

read → mapper →
(test, 1)
(hadoop, 1)
(combiner, 1)
(hadoop, 1)
(combiner, 1)
→ combiner →
(test, 1)
(hadoop, 2)
(combiner, 2)

Split 1

Hadoop. Second Split hadoop file.

read → mapper →
(hadoop, 1)
(second, 1)
(split, 1)
(hadoop, 1)
(file, 1)
→ combiner →
(hadoop, 2)
(second, 1)
(split, 1)
(file, 1)

send data on the network

reducer → Output File

Combiner

- MapReduce applications with **Combiners**
  - The (key,value) pairs emitted by the Mappers are analyzed in main-memory (or on the local disk) and aggregated by the Combiners
    - Each Combiner pre-aggregates the values associated with the pairs emitted by the Mappers of a cluster node
    - Combiners perform "pre-aggregations" to limit the amount of network data generated by each cluster node

83

Combiner

- **Combiners** work only if the **reduce function** is **commutative** and **associative**
- The **execution** of combiners **is not guaranteed**
  - Hadoop may or **may not execute** a **combiner**
    - The decision is taken at runtime by Hadoop and you cannot check it in your code
  - Your **MapReduce job** **should not depend on** the **Combiner execution**

84

Combiner

- The Combiner
  - Is an instance of the org.apache.hadoop.mapreduce.Reducer class
    - There is not a specific combiner-template class
  - "Implements" a pre-reduce phase that aggregates the pairs emitted in each node by Mappers
  - Is characterized by the reduce(…) method
    - Processes (key, [list of values]) pairs and emits (key, value) pairs
  - Runs on the cluster

85

MapReduce programs - Combiner

- The Combiner class extends the org.apache.hadoop.mapreduce.Reducer class
  - The org.apache.hadoop.mapreduce.Reducer class
    - Is a generic type/generic class
    - With four type parameters: input key type, input value type, output key type, output value type
  - i.e., Combiners and Reducers extend the same class
- The designer/developer implements the reduce(…) method
  - It is automatically called by Hadoop for each (key, [list of values]) pair obtained by aggregating the local output of a Mapper

86

43

MapReduce programs - Combiner

- The Combiner class is specified by using the job.setCombinerClass() method in the run method of the Driver
  - i.e., in the job configuration part of the code

87

Word Count Example with Combiner

- Word count problem
  - Input: (unstructured) textual file
    - Each line of the input file can contains a set of words
  - Output: number of occurrences of each word appearing in the input file
- Parameters/arguments of the application:
  - args[0]: number of instances of the reducer
  - args[1]: path of the input file
  - args[2]: path of the output folder

88

44

Word Count Example with Combiner

- Input file

  > Toy example file for Hadoop. Hadoop running example.

- Output pairs (toy, 1)
  (example, 2)
  (file, 1)
  (for, 1)
  (hadoop, 2)
  (running, 1)

89

Word Count Example with Combiner

- Differences with the solution without combiner
  - Specify the combiner class in the Driver
  - Define the Combiner class
    - The reduce method of the combiner aggregates local pairs emitted by the mappers of a single cluster node
      - It emits partial results (local number of occurrences for each word) from each cluster node that is used to run our application

90

Word Count Example with Combiner - Driver (1)

```
/* Set package */
package it.polito.bigdata.hadoop.wordcount;

/* Import libraries */
import org.apache.hadoop.conf.Configuration;
import org.apache.hadoop.conf.Configured;
import org.apache.hadoop.fs.Path;
import org.apache.hadoop.io.IntWritable;
import org.apache.hadoop.io.Text;
import org.apache.hadoop.mapreduce.Job;
import org.apache.hadoop.mapreduce.lib.input.FileInputFormat;
import org.apache.hadoop.mapreduce.lib.input.TextInputFormat;
import org.apache.hadoop.mapreduce.lib.output.FileOutputFormat;
import org.apache.hadoop.mapreduce.lib.output.TextOutputFormat;
import org.apache.hadoop.util.Tool;
import org.apache.hadoop.util.ToolRunner;
```

91

Word Count Example with Combiner - Driver (2)

```
/* Driver class */
public class WordCount extends Configured implements Tool {
 @Override
 public int run(String[] args) throws Exception {

  Path inputPath;
  Path outputDir;
  int numberOfReducers;
  int exitCode;

  // Parse input parameters
  numberOfReducers = Integer.parseInt(args[0]);
  inputPath = new Path(args[1]);
  outputDir = new Path(args[2]);
```

92

Word Count Example with Combiner - Driver (3)

```
// Define and configure a new job
Configuration conf = this.getConf();
Job job = Job.getInstance(conf);

// Assign a name to the job
job.setJobName("WordCounter");
```

93

Word Count Example with Combiner - Driver (4)

```
// Set path of the input file/folder (if it is a folder, the job reads all the files
in the specified folder) for this job
FileInputFormat.addInputPath(job, inputPath);

// Set path of the output folder for this job
FileOutputFormat.setOutputPath(job, outputDir);

// Set input format
// TextInputFormat = textual files
job.setInputFormatClass(TextInputFormat.class);

// Set job output format
job.setOutputFormatClass(TextOutputFormat.class);
```

94

Word Count Example with Combiner - Driver (5)

```
// Specify the class of the Driver for this job
job.setJarByClass(WordCount.class);

// Set mapper class
job.setMapperClass(WordCountMapper.class);

// Set map output key and value classes
job.setMapOutputKeyClass(Text.class);
job.setMapOutputValueClass(IntWritable.class);
```

95

Word Count Example with Combiner - Driver (6)

```
// Set reduce class
job.setReducerClass(WordCountReducer.class);

// Set reduce output key and value classes
job.setOutputKeyClass(Text.class);
job.setOutputValueClass(IntWritable.class);

// Set number of reducers
job.setNumReduceTasks(numberOfReducers);

// Set combiner class
job.setCombinerClass(WordCountCombiner.class);
```

96

Word Count Example with Combiner - Driver (7)

```
    // Execute the job and wait for completion
    if (job.waitForCompletion(true)==true)
        exitCode=0;
    else
        exitCode=1;
    return exitCode;
}   // End of the run method
```

97

Word Count Example with Combiner - Driver (8)

```
/* main method of the driver class   */
    public static void main(String args[]) throws Exception {
     /* Exploit the ToolRunner class to "configure" and run the
        Hadoop application */

     int res = ToolRunner.run(new Configuration(),
                              new WordCount(), args);

     System.exit(res);
   } // End of the main method

} // End of public class WordCount
```

98

49

Word Count Example with Combiner - Mapper (1)

```
/* Set package */
package it.polito.bigdata.hadoop.wordcount;

/* Import libraries */
import java.io.IOException;

import org.apache.hadoop.io.IntWritable;
import org.apache.hadoop.io.LongWritable;
import org.apache.hadoop.io.Text;
import org.apache.hadoop.mapreduce.Mapper;
```

99

Word Count Example with Combiner - Mapper (2)

```
/* Mapper Class */
class WordCountMapper extends Mapper<
        LongWritable,  // Input key type
        Text,          // Input value type
        Text,          // Output key type
        IntWritable>   // Output value type
{
    /* Implementation of the map method */
    protected void map(
        LongWritable key,  // Input key type
        Text value,        // Input value type
        Context context) throws IOException, InterruptedException {
```

100

Word Count Example with Combiner - Mapper (3)

```
        // Split each sentence in words. Use whitespace(s) as delimiter
        // The split method returns an array of strings
        String[] words = value.toString().split("\\s+");

        // Iterate over the set of words
        for(String word : words) {
            // Transform word case
            String cleanedWord = word.toLowerCase();

            // emit one pair (word, 1) for each input word
            context.write(new Text(cleanedWord), new IntWritable(1));
        }
    } // End map method

} // End of class WordCountMapper
```

101

Word Count Example with Combiner - Combiner (1)

```
/* Set package */
package it.polito.bigdata.hadoop.wordcount;

/* Import libraries */
import java.io.IOException;

import org.apache.hadoop.io.IntWritable;
import org.apache.hadoop.io.Text;
import org.apache.hadoop.mapreduce.Reducer;
```

102

51

Word Count Example with Combiner - Combiner (2)

```
/* Combiner Class */
class WordCountCombiner extends Reducer<
        Text,             // Input key type
        IntWritable,      // Input value type
        Text,             // Output key type
        IntWritable>      // Output value type
{
    /* Implementation of the reduce method */
    protected void reduce(
        Text key, // Input key type
        Iterable<IntWritable> values, // Input value type
        Context context) throws IOException, InterruptedException {
```

103

Word Count Example with Combiner - Combiner (3)

```
    /* Implementation of the reduce method */
    protected void reduce(
        Text key, // Input key type
        Iterable<IntWritable> values, // Input value type
        Context context) throws IOException, InterruptedException {
                int occurrances = 0;

                // Iterate over the set of values and sum them
                for (IntWritable value : values) {
                        occurrances = occurrances + value.get();
                }
                // Emit the total number of occurrences of the current word
                context.write(key, new IntWritable(occurrances));
        } // End reduce method
} // End of class WordCountCombiner
```

104

Word Count Example with Combiner - Reducer (1)

```
/* Set package */
package it.polito.bigdata.hadoop.wordcount;

/* Import libraries */
import java.io.IOException;

import org.apache.hadoop.io.IntWritable;
import org.apache.hadoop.io.Text;
import org.apache.hadoop.mapreduce.Reducer;
```

105

Word Count Example with Combiner - Reducer (2)

```
/* Reducer Class */
class WordCountReducer extends Reducer<
        Text,                   // Input key type
        IntWritable,        // Input value type
        Text,                   // Output key type
        IntWritable>        // Output value type
{
    /* Implementation of the reduce method */
    protected void reduce(
        Text key, // Input key type
        Iterable<IntWritable> values, // Input value type
        Context context) throws IOException, InterruptedException {
```

106

Word Count Example with Combiner - Reducer (3)

```
/* Implementation of the reduce method */
protected void reduce(
    Text key, // Input key type
    Iterable<IntWritable> values, // Input value type
    Context context) throws IOException, InterruptedException {
        int occurrances = 0;

        // Iterate over the set of values and sum them
        for (IntWritable value : values) {
            occurrances = occurrances + value.get();
        }
        // Emit the total number of occurrences of the current word
        context.write(key, new IntWritable(occurrances));
    } // End reduce method
} // End of class WordCountReducer
```

107

Word Count Example with Combiner

- The reducer and the combiner classes perform the same computation (the reduce method of the two classes is the same)
- We do not really need two different classes
  - We can simply specify that WordCountReducer is also the combiner class
    - In the driver
      job.setCombinerClass(WordCountReducer.class);
  - In 99% of the Hadoop applications the same class can be used to implement both combiner and reducer

108

54

Personalized Data Types

109

Personalized Data Types and Values

- Personalized Data Types are useful when the value of a key-value pair is a complex data type
- Personalized Data Types are defined by implementing the org.apache.hadoop.io.Writable interface
  - The following methods must be implemented
    - public void readFields(DataInput in)
    - public void write(DataOutput out)
  - To properly format the output of the job usually also the following method is "redefined"
    - public String toString()

110

Personalized Data Types - Example

- Suppose to be interested in "complex" values composed of two parts:
  - a counter (int)
  - a sum (float)
- An ad-hoc Data Type can be used to implement this complex data type in Hadoop

111

Personalized Data Types - Example (1)

```
package it.polito.bigdata.hadoop.combinerexample;
import java.io.DataInput;
import java.io.DataOutput;
import java.io.IOException;

public class SumAndCountWritable implements
    org.apache.hadoop.io.Writable {
    /* Private variables */
    private float sum = 0;
    private int count = 0;
```

112

Personalized Data Types - Example (2)

```
/* Methods to get and set private variables of the class */
public float getSum() {
    return sum;
}

public void setSum(float sumValue) {
    sum=sumValue;
}

public int getCount() {
    return count;
}

public void setCount(int countValue) {
    count=countValue;
}
```

113

Personalized Data Types - Example (3)

```
/* Methods to serialize and deserialize the contents of the
instances of this class */
@Override /* Serialize the fields of this object to out */
public void write(DataOutput out) throws IOException {
    out.writeFloat(sum);
    out.writeInt(count);
}

@Override /* Deserialize the fields of this object from in */
public void readFields(DataInput in) throws IOException {
    sum=in.readFloat();
    count=in.readInt();
}
```

114

Personalized Data Types - Example (4)

```
/* Specify how to convert the contents of the instances of this
class to a String
* Useful to specify how to store/write the content of this class
* in a textual file */
public String toString()
{
    String formattedString=
                    new String("sum="+sum+",count="+count);

    return formattedString;
}

}
```

115

Personalized Data Types and Keys

- Personalized Data Types can be used also to manage complex keys
- In that case the Personalized Data Type must implement the org.apache.hadoop.io. WritableComparable interface
  - Because keys must be compared/sorted
    - Implement the compareTo() method
  - And split in groups
    - Implement the hashCode() method

116

Sharing parameters among Driver, Mappers, and Reducers

Sharing parameters among Driver, Mappers, and Reducers

- The configuration object is used to share the (basic) configuration of the Hadoop environment across the driver, the mappers and the reducers of the application/job
- It stores a list of (property-name, property-value) pairs
- Personalized (property-name, property-value) pairs can be specified in the driver
  - They can be used to share some parameters of the application with mappers and reducers

118

Sharing parameters among Driver, Mappers, and Reducers

- Personalized (property-name, property-value) pairs are useful to shared small (constant) properties that are available only during the execution of the program
  - The driver set them
  - Mappers and Reducers can access them
    - Their values cannot be modified by mappers and reducers

119

Sharing parameters among Driver, Mappers, and Reducers

- In the driver
  - Configuration conf = this.getConf();
    - Retrieve the configuration object
  - conf.set("*property-name*", "*value*");
    - Set personalized properties
- In the Mapper and/or Reducer
  - context.getConfiguration().get("*property-name*")
    - This method returns a String containing the value of the specified property

120

Counters

Counters

- Hadoop provides a set of basic, built-in, counters to store some statistics about jobs, mappers, reducers
  - E.g., number of input and output records (i.e., pairs)
  - E.g., number of transmitted bytes
- Ad-hoc, user-defined, counters can be defined to compute global "statistics" associated with the goal of the application

122

User-defined Counters

- User-defined counters
  - Are defined by means of Java enum
    - Each application can define an arbitrary number of enums
  - Are incremented in the Mappers and Reducers
  - The global/final value of each counter is available a the end of the job
    - It is stored/printed by the Driver (at the end of the execution of the job)

123

User-defined Counters

- The name of the enum is the group name
  - Each enum as a number of "fields"
- The enum's fields are the counter names
- In mappers and/or reduces counters are incremented by using the increment() method
  - context.getCounter(countername).increment(value);

124

User-defined Counters

- The getCounters() and findCounter() methods are used by the Driver to retrieve the final values of the counters

125

User-defined Dynamic Counters

- User-defined counters can be also defined on the fly
  - By using the method incrCounter("group name", "counter name", value)
- Dynamic counters are useful when the set of counters is unknown at design time

126

Example of user-defined counters

- In the driver

```
public static enum COUNTERS {
    ERROR_COUNT,
    MISSING_FIELDS_RECORD_COUNT
}
```

  - This enum defines two counters
    - COUNTERS.ERROR_COUNT
    - COUNTERS.MISSING_FIELDS_RECORD_COUNT

127

Example of user-defined counters

- This example increments the COUNTERS.ERROR_COUNT counter

  - In the mapper or the reducer

```
context.getCounter(COUNTERS.ERROR_COUNT).increment(1);
```

128

Example of user-defined counters

- This example retrieves the final value of the COUNTERS.ERROR_COUNT counter

- In the driver

Counter errorCounter = job.getCounters().findCounter(COUNTERS.ERROR_COUNT);

129

Map-only job

Map-only job

- In some applications all the work can be performed by the mapper(s)
  - E.g., record filtering applications
- Hadoop allows executing Map-only jobs
  - The reduce phase is avoided
  - Also the shuffle and sort phase is not executed
  - The output of the map job is directly stored in HDFS
    - i.e., the set of pairs emitted by the map phase is already the final output

131

Map-only job

- Implementation of a Map-only job
  - Implement the map method
  - Set the number of reducers to 0 during the configuration of the job (in the driver)
    - job.setNumReduceTasks(0);

132

In-Mapper combiner

Setup and cleanup method

- Mapper classes are also characterized by a setup and a cleanup method
  - They are empty if they are not overridden
- The **setup** method **is called once for each mapper** prior to the many calls of the map method
  - It can be used to set the values of in-mapper variables
  - In-mapper variables are used to maintain in-mapper statistics and preserve the state (locally for each mapper) within and across calls to the map method

134

Setup and cleanup method

- The map method, invoked many times, updates the value of the in-mapper variables
  - **Each mapper** (each instance of the mapper class) **has its how copy of the in-mapper variables**
- The **cleanup** method **is called once for each mapper** after the many calls to the map method
  - It can be used to emit (key,value) pairs based on the values of the in-mapper variables/statistics

135

Setup and cleanup method

- Also the reducer classes are characterized by a setup and a cleanup method
- The **setup** method **is called once for each reducer** prior to the many calls of the reduce method
- The **cleanup** method **is called once for each reducer** after the many calls of the reduce method

136

In-Mapper Combiners

- In-Mapper Combiners, a possible improvement over "standard" Combiners
  - Initialize a set of in-mapper variables during the instance of the Mapper
    - Initialize them in the setup method of the mapper
  - Update the in-mapper variables/statistics in the map method
    - Usually, no (key,value) pairs are emitted in the map method of an in-mapper combiner

137

In-Mapper Combiners

- After all the input records (input (key, value) pairs) of a mapper have been analyzed by the map method, emit the output (key, value) pairs of the mapper
  - (key, value) pairs are emitted in the cleanup method of the mapper based on the values of the in-mapper variables

138

In-Mapper Combiners

- The in-mapper variables are used to perform the work of the combiner in the mapper
  - It can allow improving the overall performance of the application
  - But **pay attention** to the amount of **used main memory**
    - Each mapper can use a limited amount of main-memory
    - Hence, **in-mapper variables** should be **"small"** (at least smaller than the maximum amount of memory assigned to each mapper)

139

In-Mapper Combiner – Word count Pseudo code

```
class MAPPER
    method setup
        A ← new AssociativeArray

    method map(offset key, line l)
        for all word w ∈ line l do
            A{w} ← A{w} + 1

    method cleanup
    for all word w ∈ A do
        EMIT(term w , count A{w})
```

140

In-Mapper Combiner – Word count Pseudo code

class MAPPER
    method setup
        A ← new AssociativeArray   Invoked **one time** for each mapper

    method map(offset key, line l)
        for all word w ∈ line l do
            A{w} ← A{w} + 1

    method cleanup
    for all word w ∈ A do
        EMIT(term w , count A{w})   Invoked **one time** for each mapper

141

In-Mapper Combiner – Word count Pseudo code

class MAPPER
    method setup
        A ← new AssociativeArray   Invoked **one time** for each mapper

    method map(offset key, line l)
        for all word w ∈ line l do
            A{w} ← A{w} + 1   Invoked **multiple times** for each mapper

    method cleanup
    for all word w ∈ A do
        EMIT(term w , count A{w})   Invoked **one time** for each mapper

142