SCALA

Map Reduce

analysis and implementation of

Pierre Weymeels

19-01-2016

Table des matières

Chapitre 1

Map Reduce Analysis

1.1 Context

This programming model serves for processing high volumes of data.
Furthermore, it is particularly useful for parallel computing because it is based on a division of the main task in several smaller task can be provided in parallel.

1.1.1 WordCount flow

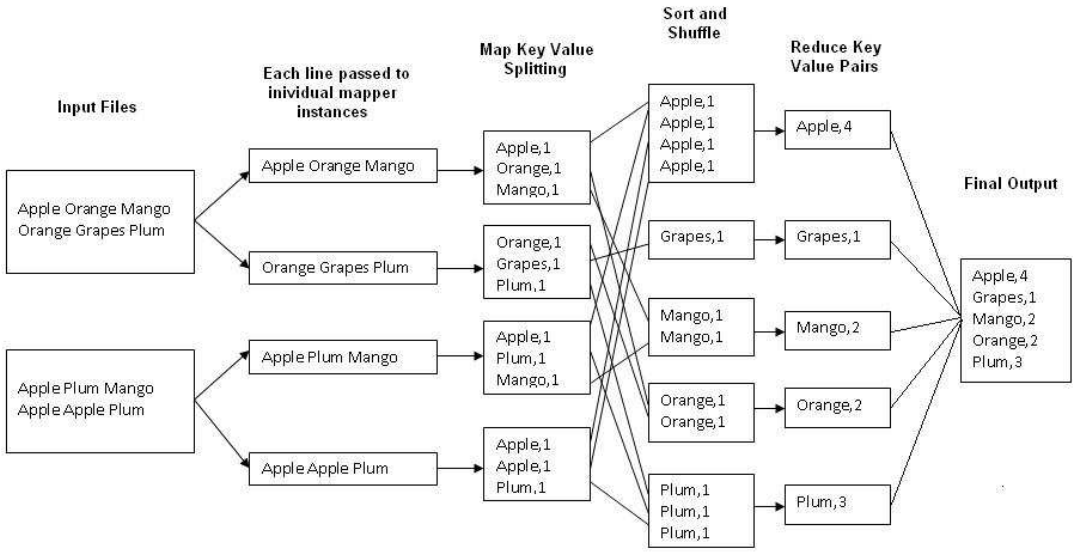

FIGURE 1.1 – WordCount flow diagram
from `http://kickstarthadoop.blogspot.be/2011/04/word-count-hadoop-map-reduce-example`

The diagram above describes the different phases related to the implementation of the MapReduce model to counting words in a list.

This kind of program can be decomposed into a series of step :

Text partition : the list is divided and each division is assigned to one process.

Map : each process transforms the data in the form of a list of pairs key-value.

Sort and shuffle :
each process set the main list with his list of pairs by bringing the values of the same key in an associated list. The final result is one list of <Key, List[Value]> pairs with each key appear once and only once.

Reduce partition : the main process distributes one key-list pair to each new process.

Reduce :
each process receives one pair, sums values and set one second main list with his result.
This main list has the form of a key-sum pair.

In my program, I respect every step but only with one process.
Furthermore, I replace the list with a text (type String), take only into account words of more than three letters and capitalize the words in the results.

1.2 Architecture

The application architecture was built for multi-tasking , so that will be easy to implement the multi-threading in the future.

1.2.1 Design pattern

In order to facilitate future evolutions, I adopted the Model View Controller approach and used a variant of the state pattern in regards to tasks of master thread (text distribution and reduce distribution), but also as regards to tasks of workers process (map, sort, shuffle, reduce).

1.2.2 Software classes diagram

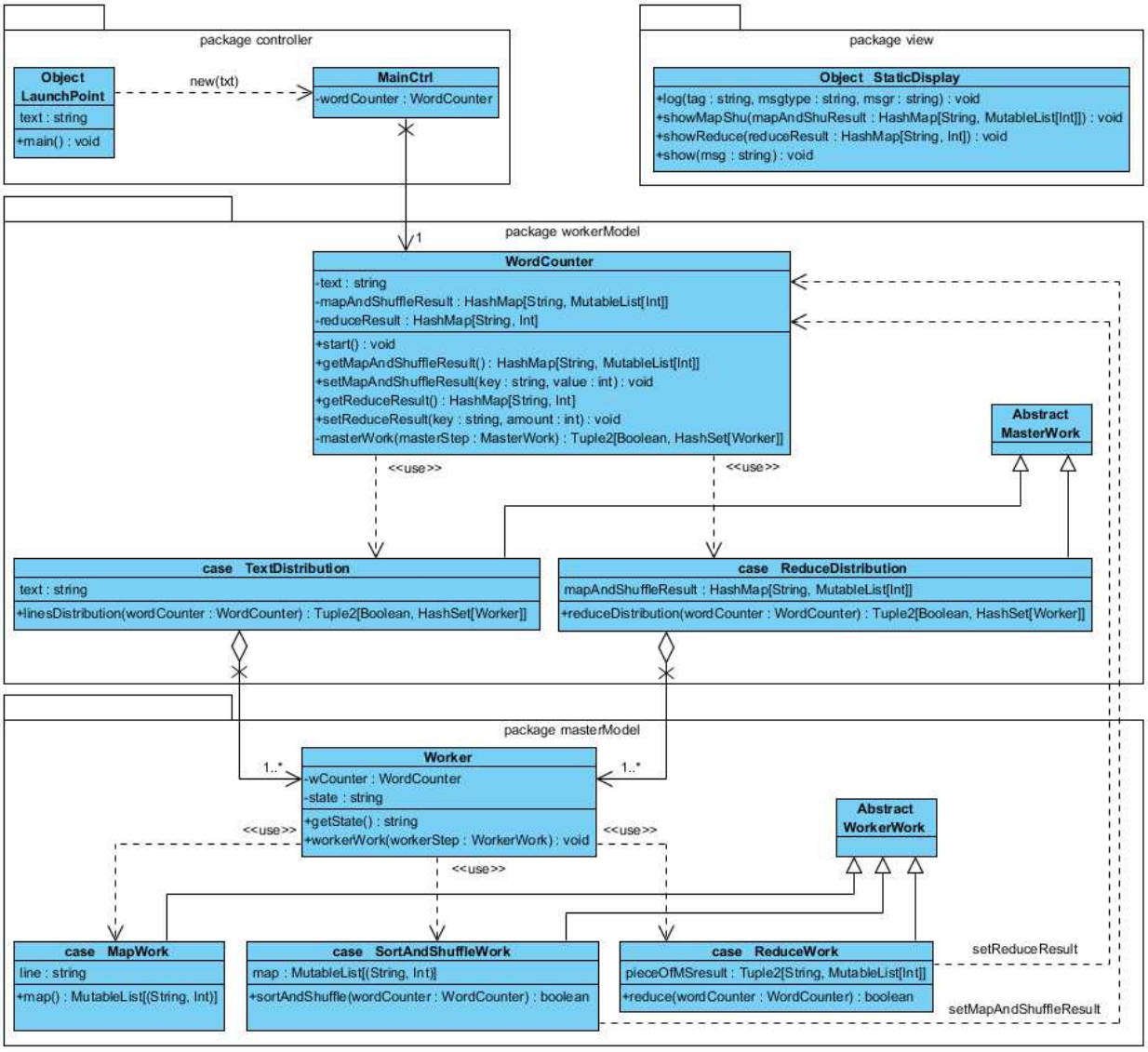

FIGURE 1.2 – Classes diagram.

1.3 Algorithm

1.3.1 Scala's pattern matching methods

My algorithm is based primarily on two Scala's pattern matching methods.
these ones organize the various tasks expressed via MasterWork or WorkerWork case classes (cf. classes diagram).

1.3.1.1 masterWork method

```
private def masterWork(masterStep: MasterWork):
  Tuple2[Boolean, HashSet[Worker]] = masterStep match {
    case TextDistribution(text) =>
      masterStep.asInstanceOf[TextDistribution].linesDistribution(this)

    case ReduceDistribution(mapAndShuffleResult) =>
      masterStep.asInstanceOf[ReduceDistribution].reduceDistribution(this)
  }
```

Description
This method uses the MasterWork case classes to distribute data.
These classes take parameter respectively a text (String) and a map-shuffle result (Map<String,MutableList<Int>>).
Then, their method respectively linesDistribution and reduceDistribution, create, distribute and return a set of classes Worker responsible for the work of map and sort-shuffle (after text distribution step) and for the reduce work (after reduce distribution step).

Note
The program operates with a single process, so a single instance of the Worker class is created
(cf. text distribution sequence diagram note).

1.3.1.2 workerWork method

```
def workerWork(workerStep: WorkerWork): Any = workerStep match {
    case MapWork(line) =>
      state = "MapWork"
      val result = workerStep.asInstanceOf[MapWork].map()
      if (result != null)
        workerWork(SortAndShuffleWork(result))
      else
        state = "Error"

    case SortAndShuffleWork(map) =>
      state = "SortAndShuffleWork"
      if(workerStep.asInstanceOf[SortAndShuffleWork].sortAndShuffle(wCounter))
        state  = "workCompleted"
      else
        state = "Error"

    case ReduceWork(pieceOfMSresult) =>
      state = "ReduceWork"
      if(workerStep.asInstanceOf[ReduceWork].reduce(wCounter))
        state  = "workCompleted"
      else
        state = "Error"
  }
```

Description
This method uses the WorkerWork case classes to manage the working phases of each worker (Worker class). "wCounter" is an instance of WordCounter. This one helps sortAndShuffle and reduce methods to set respectively the main map-and-shuffle result and the main reduce result.
There are three type of work :

Case MapWork :
takes a line (String) as parameter, his "map" method return a partial result in the form of MutableList<(String,Int)> containing one word-1 for each occurence of each word in the line.

Case SortAndShuffleWork :
takes a MutableList<(String,Int)> as parameter, his "sortAndShuffle" method sort and shuffle the main map and shuffle result with this parameter.
This main result contains a list of all occurrences for each different words of the initial text (word-(list of his occurences)) in the form of HashMap(String,MutableList<Int>).

Case ReduceWork :
takes a word-(list of his occurences) pair as parameter, his "reduce" method sums occurences between them and capitalizes the word.
Them, set the main reduce result so that the latter is in the form of HashMap(String,Int).
This main result contains a WORD-sum list of pairs.

Note
The passage between the map working phase and sort-shuffle working phase, happens thanks to a recursive call of the "workerWork" method because both work on the same line of work.

The state variable is a Worker instance parameter.
This one will be use in the case of a program with more than one process to allow the main process to know the status of all of its workers (threads).

1.3.2 Text distribution sequence diagram

Note
The program operates with a single process, so a single instance of the Worker class is created. Naturally, in a program to several processes, there will be as many instances of the class worker process as and it will implement Runnable or Callable trait (this remark also applies to the reduce sequence diagram).

Precondition : class WorkCount instantiated and start method called by class MainCtrl.

Postcondition : launch ReduceDistribution case class(cf. reduce distribution sequence diagram).

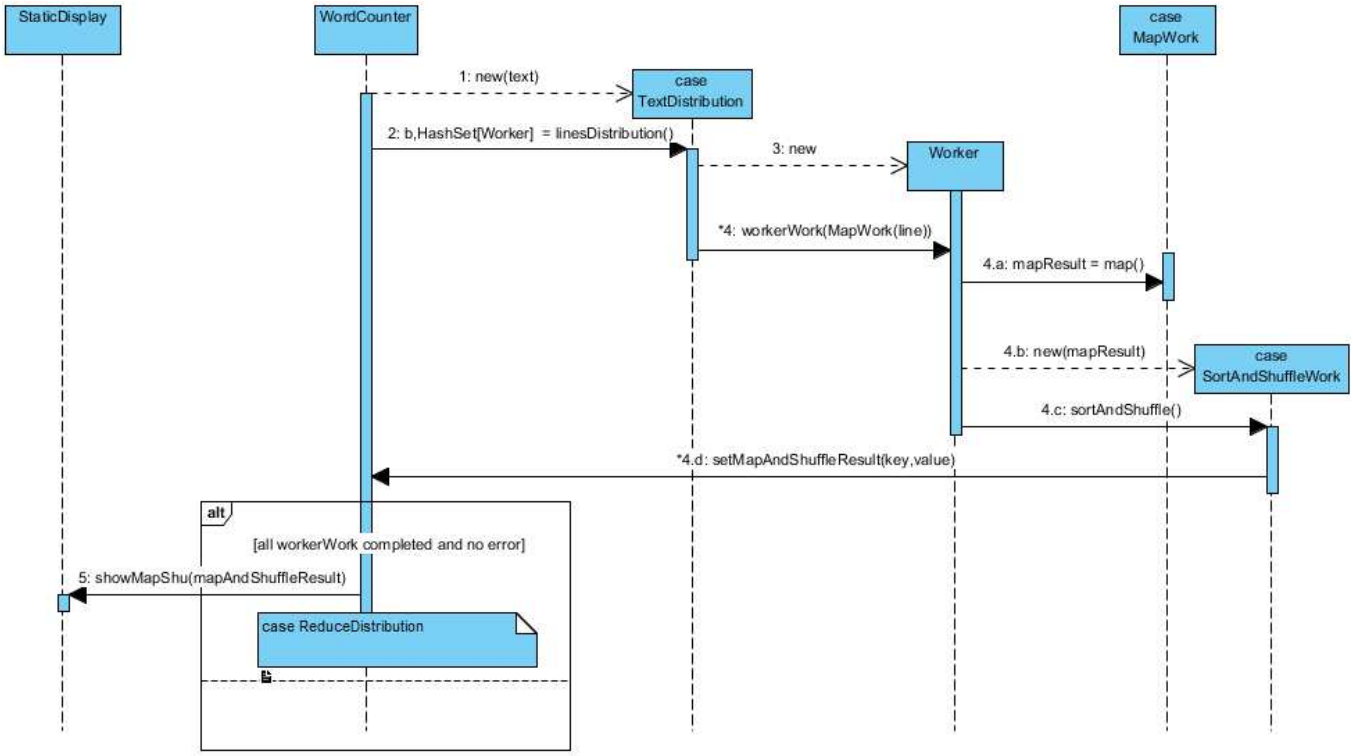

FIGURE 1.3 – Text distribution sequence diagram

Description

1 : new instance of TestDistribution case class with the text as parameter.

2 : call his "lineDistribution" method. This method return a boolean and a set of Worker instance.

3 : this method create one Worker instance (one new Worker(line) per line if multi-thread).

***4 :**
lineDistribution method call the workerWork method of this worker with an instance of MapWork on one line of the initial text as many times as there are lines (in the case of multi-thread it will be worker.run() once per worker).

For each call :

4.a : workerWork method call map method returning the map result.

4.b : workerWork create an instance of SortAndShuffleWork case class with this result as parameter.

4.c : workerWork call his "sortAndShuffle" method.

4.d : this one set the main map and shuffle result

5 :
if all workerWork completed (or all state of the set of worker egals "workCompleted" for a multi-thread program), the main process call showMapShu method to show the main map and shuffle result.

1.3.3 Reduce distribution sequence diagram

Note

The program operates with a single process, so a single instance of the Worker class is created (cf. Text distribution sequence diagram note).

Precondition : all map and shuffle process completed and no error(cf. text distribution sequence diagram).

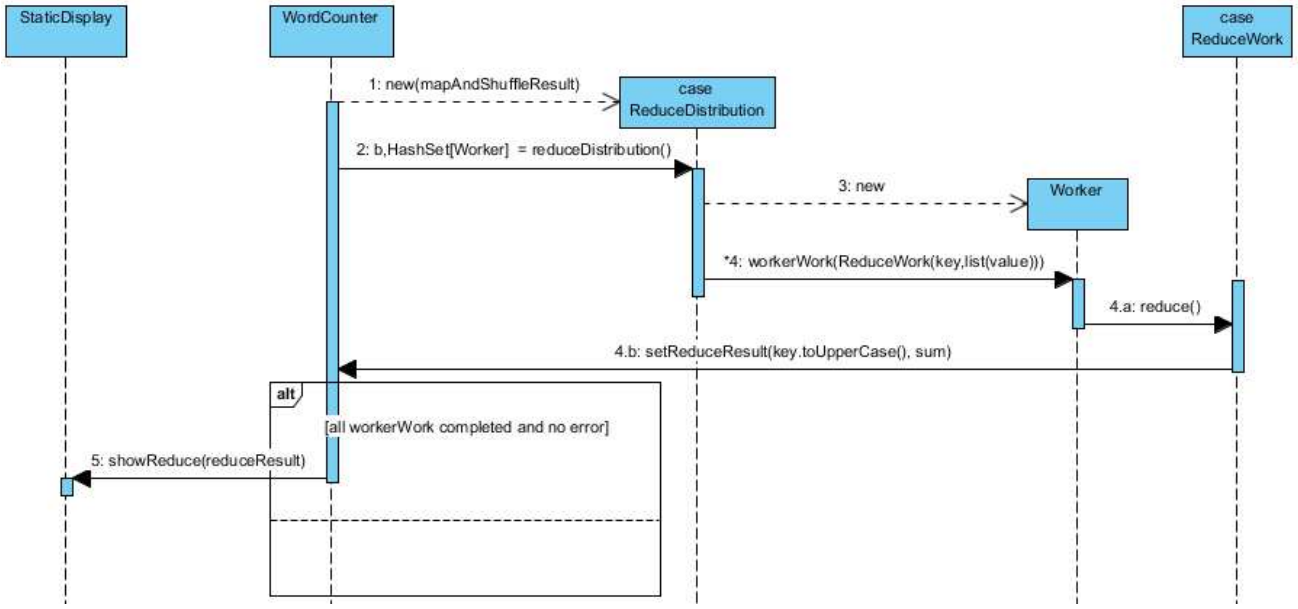

FIGURE 1.4 – Reduce distribution sequence diagram

Description

1 : new instance of ReduceDistribution case class with the main map and shuffle result as parameter.

2 : call his "reduceDistribution" method. This method return a boolean and a set of Worker instance.

3 : this method create one Worker instance (one new Worker(line) per key-list pair if multi-thread).

***4 :**
reduceDistribution method call the workerWork method of this worker with an instance of ReduceWork on one key-list pair of the main map and shuffle result as many times as there are pair (in the case of multi-thread it will be worker.run() once per worker).

For each call :

4.a : workerWork method call reduce method of this ReduceWork instance.

4.b : reduce method set the main reduce result with the word capitalized and the sum of his occurences.

5 :
if all workerWork completed (or all state of the set of worker egals "workCompleted" for a multi-thread program), the main process call showReduce method to show the main reduce result.

1.3.4 Parrallel execution and distribution

The next step is the division of tasks on several process, beforehand it will be important to make sure of the following conditions :

- Ensure that no more than one process accesses the shared methods at the same time (Synchronized Methods) to preserve data integrity.
- Optimize partitions such that all processors finish task at the same time.
- Reflect on the interaction between sequential and parallel computation.
- Coupling underutilized resources to improve their efficiency.