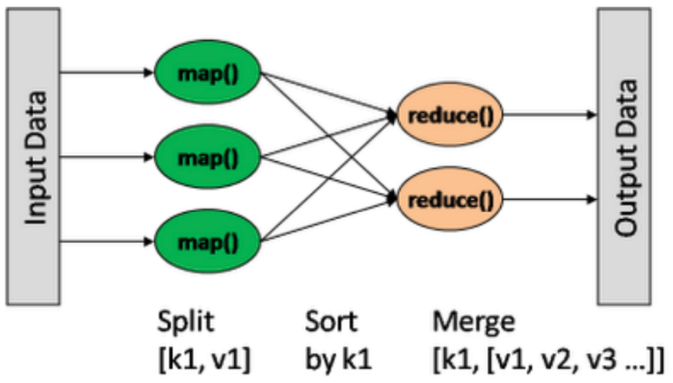
Lab 2 – Big Data Processing with MapReduce

MapReduce

MapReduce is a parallel programming paradigm proposed by Google and successfully used by several Internet service providers to perform computations on massive amounts of data. A computation takes as input a set of data and produces as output a set of key-value pairs extracted from the inputs. The user of the MapReduce library expresses the computation as two functions:

- **map** which processes a portion of the input data to generate a set of intermediate key-value pairs;
- **reduce** which merges all intermediate values associated with the same intermediate key.

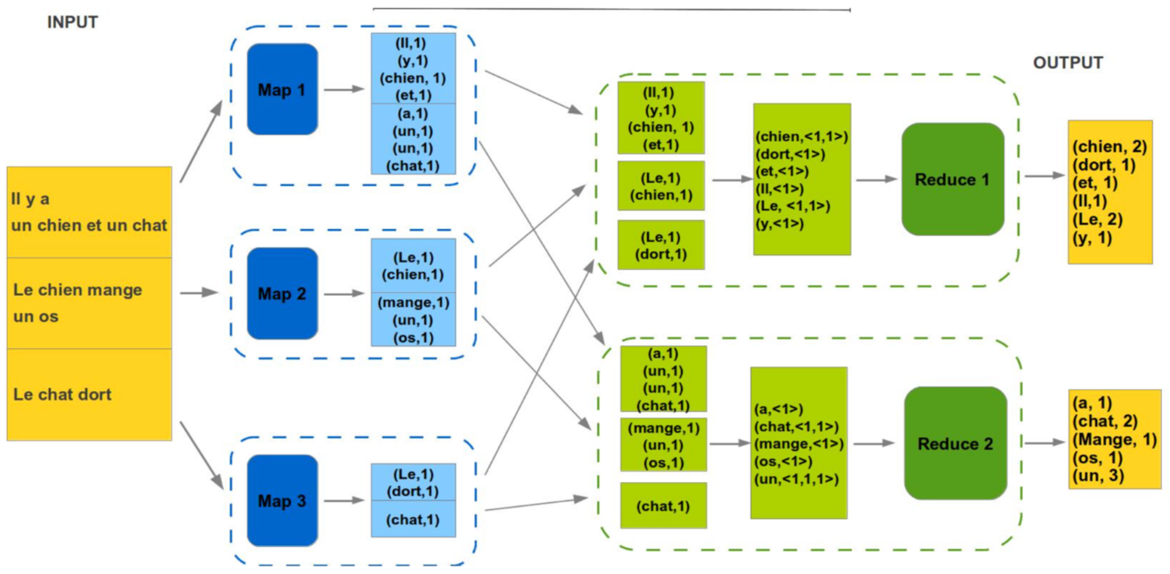


The two processing steps can be parallelized on a set of nodes. The framework takes care of splitting the input data, scheduling the component tasks, monitoring and re-executing the failed ones.

Word Count

Word Count is a classic MapReduce use case. It consists of counting the number of occurrences of words in a text stored in one or more very large files. Word Count is used by Google to build the index of the search engine and to calculate the PageRank (the input files are .html pages).

Below are the detailed data flows of Word Count using MapReduce with 3 Mappers and 2 Reducers:



As a single lab is not enough to install and learn how to use Hadoop (the open-source implementation of MapReduce), we propose to simulate this model for Word Count calculation, in Java. For the parallel execution we will rely on Java **Threads**.

A program is often designed as a (sequential) sequence of instructions. Nevertheless, it is possible to design programs where several tasks run simultaneously, in parallel; these different tasks are called **Threads** and we say that the application is multithreaded. This parallelism can only be effective if the machine used is multi-processor; if this is not the case, the different tasks share the processor successively; we will assume in this chapter that the machine used is mono-processor. To obtain multithreaded applications, one starts, generally with a main method, a first thread which can start others without stopping; the new threads can in their turn start other threads. A thread is sometimes also called a lightweight process. A multi-threaded program is also called concurrent.

The **Thread** class of the `java.lang` package is the one that must be derived in order for a class to be considered as a thread and therefore executable in parallel. The code that you want to be executed when activated must be placed in the `run()` method of your class, which extends the **Thread** class. In our case, the Mappers and the Reducers will be Threads.

Exercises

The lab archive is available online on the e-learning platform. It contains 3 `.java` source files and several `.txt` files to test your program.

As in MapReduce, the whole part of splitting the data, launching Threads and synchronization is already implemented:

- `WordCount.java` - it is the Master that splits the input file into different blocks, and launches the Mapper and Reducer threads by assigning the data to them.

It remains to fill the run() methods of the 2 Mapper and Reducer classes with the Word Count logic:

- `Mapper.java` - class for the threads that count the occurrences of each word in the local piece of text and write the results (key-value pairs) in a dictionary;
- `Reducer.java` - class for threads that merge all intermediate values associated with the same intermediate key.

Exercise 1

Read the code of the 3 source files, compile and run the provided skeleton. What do you observe?

Exercise 2

Fill the `run()` function of the Mapper. The results of each Mapper are stored in a dictionary in the form <word, nbOccurences>.

Example:

    Alice, 5
    in, 10
    the, 20
    rabbit, 7
    ...

Recall: One of the fairly common programming tasks is to parse a text into words or "lexical units" (tokens). These units are separated by a set of delimiters. The **`StringTokenizer`** class is used to perform this analysis. To extract a word from a text, you can use the `StringTokenizer` class of the `java.util` package. See the java documentation of the `StringTokenizer` class and more particularly its constructor and its `nextToken()` method.

To store the word occurrence counters, we will use the **`HashMap`** class of the `java.util` package. The keys will be the words of the text, instances of the `String` class (this class redefines the `hashCode()` and `equals()` methods for String). The object ordered according to a key (a word) will be an instance of the `Integer` class giving the number of occurrences of the associated word. Consult the java documentation of the `HashMap` class and more particularly its constructor without argument and its methods `put()`, `get()`, `containsKey()` and `keys()`.

Exercise 3

The results of the Mappers are read by the Reducers for the aggregation. For this exercise we assume that we have only one Reducer which will merge all the intermediate results into a single `HashMap`. Fill the `run()` function of the Reducer.