Hadoop Lab

LSDM 2025-2026

1 Start on a local install with `WordCount`

Before running a MapReduce job ona cluster, always test locally. In that case, the local file system is used instead of HDFS.

Your starting point will be either an empty skeleton of a MapReduce job `Q00.java` or your current development version.

Question 1.1. **Execute and observe the output of the `WordCount` job.**
We suppose that you have you your Hadoop install and that you have succeeded running `WordCount`.
Launch the program and observe the execution traces.
Verify the results in the ouput directory.
When the program works, at the end of the execution trace are given Hadoop's program counters.
Observe what changes if you change the content of one of your input text file and answer the following questions.

Question 1.2. What does `Map input records` indicate? And `Map output records` ? Explain the observed values.
Answer: 'records' always refers to a key-value pair. These two counters therefore represent the number of input and output pairs of the 'map' function. The input corresponds to the number of lines in the file read. The output corresponds to the number of words in the document.

Question 1.3. What is the relationship between `Map output records` and `Reduce input records` ?
Answer: The reflect the same thing as the data produced by the mapper are direcly sent to the reducer. These are the words in the input files.

Question 1.4. What `Reduce input groups` corresponds to?
Answer: the intermediary couples between the mapper and the reducer are grouped by key. The number of groups is the number of keys (the words here). the reduce function will be applied to each of these groups.

2 Running the `WordCount` on a Google Dataproc Cluster

You have received an e-mail giving you educational credits on the Google cloud. Using these credits you can :

1. Create a Dataproc cluster
2. Upload the input data and the jar of your job
3. Execute the job
4. Visualize its state and get statistics

Let us now count the words in the *Misérables* of Victor Hugo. You can download the file from a link given at the LSDM teaching page
`https://vmarangozova.github.io/teaching.html#LSDM`

NOT TESTED YET :) Another way to get the data is to use the data I put on a publicly accessible Google bucket `gs://hadoop-2025-vania`

Question 2.1. Look for the number of *splits* in the execution trace. Which counter gives this information? What defines the number of *splits* ?
Answer: "Launched map tasks". The framework creates one instance of *Mapper* per *split*. The number of *splits* depends on the HDFS bloc size and the size of the input data.

Keep the trace and the output results to be able to compare with what follows.

Hadoop allows for *combiners* which reduce locally, straight after the map function and before sending data to reducers.

Question 2.2. Define a *combiner* using `job.setCombinerClass(...)` in the main function of your job. Verify that the job runs locally before executing it on the cluster and with the big data.

Question 2.3. Which counters show that the combiner has been used?
Answer: `Combine input records` et `Combine output records` are not zero

Question 2.4. Which counters allow for performance comparison between the version without and with a combiner ? Compare the values and explain.
Answer: Several possibilities: we can compare 'Reduce input records' or 'Map output materialized bytes' with the counter obtained without 'combiner'. If we only have the counters from the version with 'combiner', the ratio between 'Combine input records' and 'Combine output records' also gives the reduction factor. The 'Map output materialized bytes' counter is higher than the 'Map output bytes' counter because it indicates the amount of data written to disk, including management/encoding structures, etc.

Question 2.5. Which word is the most used in the *Les Misérables* ?
Answer: usually "de".

2.1 Number of *reducers*

The default number for Hadoop reducers is one. Change your job to use three reducers with `job.setNumReduceTasks(3)` in the main function.

Question 2.6. Which counter reflects this modification?
Answer: `Launched reduce tasks`

Question 2.7. Do you observe any difference in the output results? Explain.
Answer: The output directory in the previous question contained only one file, `part-r-00000`, whereas there are now three files (because we requested, and received, three *reducers*). Each *reducer* writes to its own file to avoid concurrent writing issues.

Question 2.8. Compare the execution time with 3 *reducers* to the execution time with a single *reducer*. Which metrics do you use?
Answer: Looking at the web interface we have time information. Have not executed on Dataproc yet to see.

2.2 In-Mapper Combiner

A common method to speed up the mapper-combiner process is to manually combine the results in the mapper. This ensures that the combine command is applied. Additionally, it avoids saving data to disk between the mapper and the combiner.

To implement this, we override two methods of `Mapper`:

- `setup`, which is called before the first `map` applied to an *input split*, and
- `cleanup`, which is called after the last `map` of the *input split*.

The in-mapper-combiner involves initializing a collection that will serve as a buffer in `setup` (in our case, we'll use `java.util.HashMap<String, Integer>`), and then writing its contents to `cleanup`.

Therefore, in the `map` function, instead of calling `context.write()`, we update the buffer.

Question 2.9. Implement the proposed solution.

Question 2.10. Do you see any impact of using the *in-mapper* on the execution time?
Answer: should be even faster.

Question 2.11. Do you see any impact of using the *in-mapper* on the used memory?
Answer: more memory usedn may be seen in Virtual memory (bytes) snapshot et Total committed heap usage (bytes)

2.3 Specific Counters

Hadoop provides an API for defining and working with application-specific counters (in addition to the ones you have already observed).

Question 2.12. Let us define a counter of our own.

- Create a `enum` with a single member, this gives the name of the counter.
- In your mapper, define an attribute using the (`Counter`) class.
- In `setup`, get the `Counter` object using `context.getCounter`.
- In `map` count the empty lines using this counter.

Test locally.

3 Working with the Flickr Dataset

We will now analyze photo metadata using part of the *Yahoo! Flickr Creative Commons 100M* dataset. The goal is to find the most used tags per country. A country is identified by the coordinates of the photos with the help of the provided `Country` class.

In the downloaded archives, in the data folder, you have two files:

- `flickrSpecs.txt` describes the file format. To decode use `java.net.URLDecoder.decode(String`

- `flickrSample.txt` is a subset for the local tests. Beware of malformed data which your program should ignore.

The full dataset can be downloaded from a link indicated at the teaching page
https://vmarangozova.github.io/teaching.html#LSDM

You will certainly find useful the Hadoop API
https://hadoop.apache.org/docs/stable/api/index.html

In this API some classes have two versions, be sure to use the `org.apache.hadoop.mapreduce` package and not `org.apache.hadoop.mapred`.

3.1 MapReduce

We will serach for the K most used tags per country. We will thus consider tuples containing country identifiers and tag information. To identify a country, we will use the two-letter code given by `country.toString()`. The tags are word lists associated with each photo taken in a given country.

Question 3.1. Here are the indictaions to implement the mapper and the reducer.

For the *mapper*, reuse the `WordCount` application. You will need the field containing the country code, as well as the field containing the tags.

For the *reducer*, you will need to count the occurrences of tags. To identify the K most frequent ones, use the `MinMaxPriorityQueue` class[1].

This class need a comparison operator. To define it, define a `StringAndInt` class that implements the
`Comparable<StringAndInt>` interface. This class will contain two attributed for respectively a tag and its occurrences. The `compareTo` method will work on the tag occurrences.

To define K, pass it as an additional argument to the main program. To get ist value in the main method, use the `Configuration` class. To get this value at the reducer level, use the `Context` class.

Implement and test your program locally. What are the two most frequent tags for Japan ?

3.2 Combiner

The goal here is to add a combiner to the computation.

Question 3.2. How the *combiner* will work? What type of data should it manipulate?Can you reuse the `StringAndInt` class for this question?
Answer: The combiner will do the same thing as the *reducer*, so it needs to calculate the number of tag occurrences per country. The key will therefore be the country, and the value will be tag+occ, i.e, we will reuse `StringAndInt`

[1] cf. http://docs.guava-libraries.googlecode.com/git/javadoc/com/google/common/collect/MinMaxPriorityQueue.html

Question 3.3. Modify the `StringAndInt` class to make it implement the `Writable` (in addition to `Comparable`) interface[2]. Why is this necessary?

Question 3.4. Using a *combiner* impacts *reducers*. Change the *reducer* class to reflect these changes i.e extend `Reducer<K,V,K,V>`.

Question 3.5. Run this version with $K = 5$, with the full dataset[3], on the cluster. WHat are themost frequent tags for France
Answer: `france, paris, barcelona, spain et europe`

Question 3.6. Discuss the size of the data structures used in a *reducer*. Is there a data structure whose size depends on the input data? Is this a problem?
Answer: It depends on how it's implemented. The HashTable could be created and destroyed per country code; its size depends on the number of tags. Let's say we use 20 bytes per tag: with 1GB of memory, we could therefore count approximately 53 million tags. The final structure, the priority queue, will contain a maximum of 5 tags per country, i.e., a limited size. Finally, in the sample file, there's no problem.

4 Limit the memory in the Flickr Job

Hadoop allows for defining application-specific management functions for the intermediary keys. We can thus control:

- the grouping of keys before the *reduce* i.e decide which values to associate with which keys, and
- the order of the $(cle, valeur)$ pairs that will be analyzed by the *reducer*.

There are more details in the class documentation Reducer.

This part aims at charging Hadoop with the sorting of the tags in order to reduce the RAM of the job. You will do this by implementing a pipeline of two Hadoop jobs.

Question 4.1. The first MapReduce job is to count the number of tag occurrences per country. You can still reuse `WordCount` but with different types of intermediary keys. You can consider the concatenation of the country with a tag (e.g., `country:tag`).

Question 4.2. The second job will take as input the output of the frst job. The *mapper* should define the intermediary keys on which Hadoop will apply the application-specific grouping and sorting operations. The grouping function (`job.setGroupingComparatorClass()`) should consider the country code. The sorting function (`job.setSortComparatorClass()`) will apply a decreasing order on the tags of a given country.

Before coding and testing, confirm your solution with the teachers.
Answer: The first *job* counts the occurrences of each tag in each country.
Since the country code is always two letters, we can concatenate the country code and the tag into a Text object.
The intermediate key, like the output key, is therefore a Text object containing the country code followed by the tag.

[2] The documentation gives you the necessary details.
[3]

The intermediate value, and the output value, is an integer: the counter.
The first *job* is actually analogous to word counters; we can even reuse the *combiner* from the first part. In the second *job*, the input is therefore $(Text, IntWritable)$ pairs which represent a country code and a tag, and the occurrence counter. The `map` function will swap the tag and the counter, so that the type of the intermediate values is `(StringAndInt, Text)`. We must then tell the framework that the groups should only be based on the string (so that '$reduce$' is only called once per country), and that the sorting should be done according to the full key (but in descending order of the counter). This way, each `reduce` function will receive the tags in descending order of frequency.

Question 4.3. Run the jobs. Compare the performances of this version with the initial one.