# Distributed Computing with MapReduce and Spark

Henry Leclipteur, Samuel Noé

August 2024

## 1 Challenges

The first challenge was to find documentation on HDFS, a lot of different implementations exist in both languages, python and java.
We found a tutorial in java demonstrating how to make a simple Hadoop MapReduce program in java.
https://hadoop.apache.org/docs/stable/hadoop-mapreduce-client/hadoop-mapreduce-client-core/MapReduceTutorial.html

The second challenge was to find a way to iterate over a job and to get the state of the algorithm to stop it when it is over.

And finally, the translation of the hadoop algorithms was not straightforward to scala.

## 2 Key algorithmic steps

### 2.1 Separation Degree with Map reduce hadoop

- 1. Preprocess the data
  It is composed of one Map.
  The map takes each line of the title.principals.tsv file and outputs: "$title \backslash t\ actor \backslash t\ distance$"
  Where:
  - *actor* is the actor or actress code.
  - *title* is the title code.
  - *distance* is either 0 or 2147483647.
  The distance is 0 if it is the separation actor and 2147483647 for all other actors.

- 2. Algo1
  This step is composed of a Map and a Reduce.
  The Map outputs: $Key(title)\ Data(actor \backslash t\ distance)$
  The Reduce put all actors that have an infinite distance in an array.
  We also store the first non-infinite value (if any).

  – If the title is "$\_END\_$"
  simply outputs:"$\_END\_ \backslash t\ actor \backslash t\ distance$"

  – If we detected a non-infinite value
  outputs for all actors: "$title \backslash t\ actor \backslash t\ non\_infinite\_value + 1$"

  – If we didn't detect a non-infinite value
  outputs for all actors: "$title \backslash t\ actor \backslash t$ 2147483647"

  The goal of this job is to spread the distance of an actor to all the actors that played in the same film and "delete" the actor-film pair that already have their distance.
  The title = "$\_END\_$" case is for actor that already found their separation distance.

- 3. Algo2
  This step is composed of a Map and a Reduce.
  The Map outputs: $Key(actor)$ $Data(title \backslash t\ distance)$
  The Reduce put all titles that have an infinite distance in an array.
  We also store the first non-infinite value (if any).

  - If we detected a non-infinite value
    outputs once :"$\_END\_ \backslash t\ actor \backslash t\ distance$"
    outputs for all title in array: "$title \backslash t\ actor \backslash t\ non\_infinite\_value$"
    increment a user-defined counter.
  - If we didn't detect a non-infinite value
    outputs for all actors: "$title \backslash t\ actor \backslash t$ 2147483647"

  The goal of this job is to spread the distance of an actor to all of its films and "delete" the actor-film pair that already have their distance.
  If an actor got its distance, we output the special title = "$\_END\_$" line that wont be deleted.
  Algo1 et Algo2 will loop until the user-defined counter is equal to 0, meaning this time no actor spreaded its value and all the separation degree of all actors were calculated.

- 4. Final This step contains a Map and a Reduce.
  The Map outputs: $Key(actor)$ $Data(title \backslash t\ distance)$
  The Reduce outputs once for each actor: "$actor \backslash t\ distance$".
  Note: the real outputs is "$sep\_actor \backslash t\ actor \backslash t\ distance$"
  That is because our algorithm can calculate multiple separation degree number at the same time by adding the separation actor in the key in each step.

## 2.2 Separation Degree with Spark Scala

- We load the *title.principals.tsv* as an RDD.
- We do a first map that format to get a new RDD with only actors, and with each entry as (key : title, value : actor dist), with any entry corresponding to Kevin Bacon given an initial distance of 0, and all others of infinity - which is just the maximum integer on 32 bits.
- We then call a recursive function that is going to be the main part of the job. It takes a argument *inf_count*, the number of actors with an infinite distance to KB in the previous iteration (intially infinite), used to stop the recursion, *current_dist*, which is the current distance from KB we are currently expecting, *stop* to indicate wheter to stop the recursion, and *start_rdd*, which is a RDD with (key : title, value : actor dist).
- From our start_rdd, we do a reduceByKey(), to obtain (key : title, value : actor1, ..., actorN dist), where dist is the minimum of all encountered distances. We take particular attention to make sure that the actor with the minimum distance is first in the list.
- We perform a flatMap() to obtain (key : actor, value : title dist). The 1st actor in the list retains its distance, the other get that same distance plus one, and the actors in movies with no link to KB yet - that is the minimum distance is still infinity - keep infinity as their distance.
- We perform another reduceByKey() to obtain(key : actor, value : title1, ..., titleN dist), dist being the minimum distance encountered.
- We peform a final flatMap() to obtain (key : title, title : actor dist) once again, that we can pass to the next call

- We check if we should stop.
  At some point the only actors left who still have an infinite distance are those not linked to KB whatsoever. Therefore we simply count the amount of actors who still have an infinite distance, which should decrease after each step. If at some step it remains equal, that means we are done.

- Once all of that is done, we simply remove the titles, reduceByKey() to have only one instance of each actor with the minimum distance that is its Kevin Bacon degree.

### 2.3 Average Rating with Map reduce hadoop

- 1. Preprocess the data
  It is composed of a Map and a Reduce.
  The map takes each line of the title.principals.tsv file and outputs: $Key(title)$ $Data(actor \backslash t\ a)$
  Where:
  - *actor* is the actor or actress code.
  - *title* is the title code.
  - $a$ is to distinguish average rating lines from title principals lines.

  The map also takes each line of the title.ratings.tsv file and outputs: $Key(title)$ $Data(rating \backslash t\ r)$
  Where:
  - *rating* is the rating of the film.
  - *title* is the title code.
  - $r$ is to distinguish average rating lines from title principals lines.

  The Reduce puts all actors in an array and save the rating of the film.
  Then if there is a rating for that film,
  outputs: "$actor \backslash t\ rating$"

- 2. Final
  This step is composed of a Map and a Reduce.
  The Map outputs: $Key(actor)$ $Data(rating)$
  The Reduce adds all ratings together and count them.
  It then outputs: "$actor \backslash t\ average\_rating$".
  Which is the average rating of each actor for all the films they participated in.

### 2.4 Average Rating with Spark Scala

The steps are basically the same as in the Average Rating with Map reduce hadoop.
The steps 1 to 3 correspond to the preprocessing in the hadoop version.
The steps 4 to 6 correspond to the final in the hadoop version.

## 3 Performance analysis

| **Program** | **Execution Time** |
|---|---|
| Separation_Degree.java | 36m 20sec |
| Separation_Degree.scala | - |
| AvgRating.java | 4 min |
| AverageRating.scala | 58 sec |

See the github page for the laptop specification.

We can see that the scala spark outperforms the map reduce implementation on these tasks.
This can be explained by the fact that hadoop does a lot of I/O in between jobs compared to Spark that does everything in the memory.

Framework comparaison:

- Performance
  Hadoop is slower as it relies on disk I/O compared to Spark that does in-memory computation.
- Memory
  Hadoop is less memory intensive than Spark.
- Fault tolerence
  The fault tolerance is stronger in the Hadoop framework due to the fact that it is disk based.

# 4 Involvement

- Henry Leclipteur did:
  - the Hadoop implementation of the separation degree and the average rating
  - the Scala Spark implementation of the average rating
  - the github and the report
  - all the performance tests
- Samuel Noé did Spark Scala implementations of the separation degree.
- Gregory Voskertchian did not participate in the project.

# 5 Links

Github link: https://github.com/Longferret/docker-hadoop-spark
It contains all the implementations, how to set up docker, how to run the code, the outputs we generated and the videos of our algorithms in action.