Université de Liège

INFO8002: Topics in Distributed Systems

Distributed Computing with MapReduce and Spark

Meunier Loïc, Abdelaleem Aly, Hoorelbeke Jordi

Academic year 2023-2024

Contents

1 Initial Setup **2**
1.1 Demonstration video . . . 2

2 Challenge A: Determine the Degrees of Separation Between Actors and a Given Actor **3**
2.1 MapReduce Implementation . . . 3
2.1.1 Code Implementation . . . 3
2.1.2 Launching the Program . . . 4
2.1.3 Results . . . 4
2.2 Spark Implementation . . . 5
2.2.1 Code Implementation . . . 5
2.2.2 Launching the Program . . . 6
2.2.3 Results . . . 6
2.3 Analysis . . . 7

3 Challenge B: Determine the Average Rating per Actor and Producer **8**
3.1 MapReduce Implementation . . . 8
3.1.1 Code Implementation . . . 8
3.1.2 Launching the Program . . . 8
3.1.3 Results . . . 9
3.2 Spark Implementation . . . 9
3.2.1 Code Implementation . . . 9
3.2.2 Launching the Program . . . 9
3.2.3 Results . . . 10

4 Challenges Faced **10**

5 Description of everyone's involvement in the assignment **11**

1 Initial Setup

To establish the initial environment required to execute the programs discussed in this report, please follow the steps outlined below[1].

1. Clone the repository

   Listing 1: Repository Cloning.

   ```
   git clone git@github.com:jordi-h/bacon-number-mapreduce-spark.git
   ```

   For easier copying:

   `git clone git@github.com:jordi-h/bacon-number-mapreduce-spark.git`

2. Navigate to the `workdir/` folder and launch the containers

   Listing 2: Container Launching.

   ```
   docker compose up -d
   ```

3. Download the appropriate datasets from the IMDb website: `title.principals.tsv` and `title.ratings.tsv`, and put them inside the `workdir/datasets/` folder.

4. Upload the datasets to the HDFS

   Listing 3: HDFS Populating.

   ```
   ./upload-datasets.sh
   ```

1.1 Demonstration video

We filmed a small demonstration video, as requested. The video is available on YouTube, at the following link: https://youtu.be/zyVwqFZ-LD0.

[1] All commands in this report were executed within a Windows WSL 2 environment running Ubuntu and on MacOS.

2 Challenge A: Determine the Degrees of Separation Between Actors and a Given Actor

2.1 MapReduce Implementation

2.1.1 Code Implementation

The algorithm is implemented using Python, and leverages the *mrjob*[2] package to write the MapReduce jobs.

The implementation accepts 2 command-line arguments that describe the behaviour of the program:

1. `--source` The source actor from which the degrees of separation should be computed. Default is `nm0000102`, i.e. Kevin Bacon.

2. `--depth` The upper bound on the degrees of separation to investigate. The program will only return relevant results for actors within `depth` degrees of separation from the `source`. Default is `6`, as a reference to the Six Degrees of Separation[3].

The implementation can be divided into 3 main parts:

1. `Preprocessing:` Starting from the IMDb datasets, more precisely the `title.principals.tsv` one, the preprocessing is composed of 2 MapReduce jobs. The output of the first MapReduce job is chained into the second job input.

   The first MapReduce job outputs key-value pairs in the form of
   `(tconst1, [nconst1, nconst2, ...])`
   where `tconst` and `nconst` are respectively the film title and actor IDs. Logically speaking, each title (key) is associated with its participating actors/crew members (values).

   The second MapReduce job then uses this output and transforms it into
   `(nconst [distance, [adjacency list]])`
   where `nconst` is the actor's ID, `distance` is its distance to the source actor, and `adjacency list` is the list of actors with which `nconst` collaborated directly. Logically speaking, this second MapReduce job builds a graph of actors, using their adjacency list representation. It also initializes the distance of every `nconst` to the source actor to a very large and unattainable value, except for the source itself that has a `distance` of 0.[2]

2. `Parallel BFS:` Starting from the graph representation depicted above, we iteratively perform `depth` rounds of MapReduce jobs. The MapReduce job can be described as:

   - **Data:**
     - Key: `nconst`

[2] https://mrjob.readthedocs.io/en/latest/
[3] https://en.wikipedia.org/wiki/Six_degrees_of_separation

- Value: `distance` (distance from source), `[adjacency list]` (list of neighbors directly reachable from `nconst`)

- **Mapper:**
  - `emit(nconst, [adjacency list])`
  - `emit(nconst, distance)`
  - `emit(n, distance + 1)` $\forall n \in$ `[adjacency list]`
- **Reducer:** Group and select the minimum distance for each reachable node, and rebuild the graph structure thanks to the emitted adjacency lists.

This process is indeed iterative, in the sense that the parallel BFS works with a *frontier* of known nodes that expands by one hop in every direction for each iteration. The `depth` parameter described above is thus the number of iterations to be performed.[4]

If we define D as the "source diameter" of the actors' graph, i.e. the length of the shortest path between the source and its most distanced node, then doing less than D iterations will result in incomplete information, with some nodes being marked as 'unattainable' because they haven't been discovered yet. On the contrary, doing exactly D iterations or more than D iterations will always provide complete information, but the more 'useless' iterations there are, the more computational power is being wasted.

3. `Postprocessing:` This last step is also a MapReduce job, but it is much simpler than the previous steps. The mapper simply emits `(nconst, distance)` pairs while removing adjacency lists for clarity purposes. The reducer simply

With very little overhead, one could also keep note of the path that links the discovered nodes to the source once they have been discovered by the algorithm. We only computed the distance for the sake of simplicity.

2.1.2 Launching the Program

Listing 4: Launch Challenge A with Hadoop MapReduce.

```
./mapreduce-bacon-script.sh
```

The script will run the `parallel_bfs_mapreduce.py` file with the first `2 000 000` rows from the `title.principals.tsv` dataset. The dataset's size has been reduced because of the excessive execution time on the original dataset.

2.1.3 Results

The Bacon numbers obtained are stored in the file named `workdir/outputs/mapreduce-bacon_-number.txt` and are organized as shown below.

Listing 5: Contents of `/outputs/mapreduce-bacon_number.txt` file.

```
"nm0000001"     [2]
"nm0000002"     [2]
```

```
"nm0000003"     [3]
"nm0000004"     [1]
...
"nm0000100"     [2]
"nm0000101"     [2]
"nm0000102"     [0]
"nm0000103"     [2]
...
"nm1342395"     [10000]
"nm1342397"     [4]
"nm1342455"     [3]
"nm1342474"     [4]
...
```

The execution time took around `28 minutes`[4]. The results are based on the first `2 000 000` rows of the `title.principals.tsv` file, hence, the results are not representative of the real Bacon Numbers. Furthermore, a Bacon number of `10 000` means that there exist no link between actors.

2.2 Spark Implementation

2.2.1 Code Implementation

The algorithm is implemented using Scala and is composed of three distinct parts:

1. The first part is the main code, responsible for putting the dataset into a DataFrame, calling the two subsequent functions, and writing the result into an output text file.

2. The `findActorPairs()` function filters the dataset by actor and actress and transforms it into a resilient distributed dataset (RDD) so that elements are partitioned across the nodes of the cluster. Finally, all unique 2-actor combinations from the list of actors, who have played together, are generated.

3. The `findActorShortestPaths()` function computes the shortest path distances from a specified target actor (i.e. Kevin Bacon) to all other actors using the Breadth-First Search (BFS) algorithm. It first sets the target actor's distance to zero and iteratively explores all connections. Using a broadcast variable so it can be accessed in parallel by the nodes, it checks connections involving active actors each round to discover and track new actors' distances. Distances are updated by unioning newly found distances with the ongoing dataset, and the algorithm progresses by updating the set of active actors until no new actors are left to explore.

[4]The execution time was computed using a stopwatch. For a complete rundown of the program execution, refer to the `workdir/outputs/mapreduce-bacon-terminal-output.txt` file.

2.2.2 Launching the Program

Listing 6: Launch Challenge A with Spark.

```
./scala-script.sh BaconNumber
```

This script is responsible for launching the scala shell inside the `spark-master` container, compiling the given `.scala` code file, and launching the process. It is to be noted that Spark specifications have been set up according to the machine resources. Since the container's maximum memory is `28.55 GB`, we allocate `7 GB` (with an additional `1 GB` for overhead) to each of the three executors. The master node, which oversees task execution and resource management but doesn't engage in computations, is allocated `4 GB`, which is more than sufficient. Hence, **please adapt the following part in the script according to your machine specs**.

Listing 7: Spark Command.

```
/spark/bin/spark-shell \
  --master spark://spark-master:7077 \
  --conf spark.executor.memory=7g \
  --conf spark.driver.memory=4g \
  --conf spark.serializer=org.apache.spark.serializer.
     KryoSerializer \
  --conf spark.executor.memoryOverhead=1g
```

or replace it with the following one-liner, which should be suitable for *most* machines:

Listing 8: Spark One-liner.

```
 /spark/bin/spark-shell --master spark://spark-master:7077
```

For easier copying:
`/spark/bin/spark-shell -master spark://spark-master:7077`

2.2.3 Results

The Bacon numbers obtained are stored in the file named `workdir/outputs/spark-bacon_number.txt` and are organized as shown below.

Listing 9: Contents of `workdir/outputs/spark-bacon_number.txt` file.

```
(nm0000102,0)
(nm1955791,1)
(nm1395858,1)
(nm0094048,1)
...
(nm10722411,2)
(nm0346730,2)
(nm1775177,2)
(nm1831376,2)
...
```

```
(nm5503384,3)
(nm14876127,3)
(nm1489584,3)
(nm8393437,3)
...
```

Upon completion, the code provides the following statistics:

Application ID	Name	Cores	Memory per Executor	Resources Per Executor	Submitted Time	User	State	Duration
app-20240503105418-0025	Spark shell	48	7.0 GiB		2024/05/03 10:54:18	root	FINISHED	6.5 min

Figure 1: Statistics accessible from http://localhost:8080/ after completion.

2.3 Analysis

First, it is important to note that to validate our results, we compared them with those provided by The Oracle of Bacon website. Our results are similar, except for some actors. This difference is likely due to the fact that the website is using a different dataset to compute its Bacon numbers.

Moreover, our implementation in Spark significantly outperforms our Hadoop MapReduce implementation, both in terms of ease of implementation and efficiency. MapReduce writes intermediate results to the file system, which incurs considerable overhead. In contrast, Spark caches data in-memory, which greatly enhances system performance, which is a key area where Spark excels over MapReduce. Additionally, Spark provides tools that facilitate implementation and optimize execution time. For instance, DataFrames allow for the pre-definition of schema, enhancing query optimization and execution, and RDDs offer robust capabilities to efficiently manage parallel tasks across multiple nodes[1].

3 Challenge B: Determine the Average Rating per Actor and Producer

For challenge B, we use the distributed system to compute the average rating per actor and per producer.

3.1 MapReduce Implementation

3.1.1 Code Implementation

The algorithm is implemented using Python, and leverages the *mrjob*[5] package to write the MapReduce jobs.

The implementation accepts 1 command-line argument that describes the behaviour of the program:

1. `--ratings-file` The path to the *ratings file* to be used for film ratings. We use IMDb's `title.ratings.tsv` by default.

In this implementation, we will use a known pattern for MapReduce designs. The pattern is named `Replicated Join`. It is described as follows: "A replicated join is a special type of join operation between one large and many small data sets that can be performed on the map side."[3]

Our implementation makes use of the replicated join pattern, where the small dataset that can fit in memory is the `title.ratings.tsv` file from IMDb, and the large dataset is the `title.principals.tsv` file.

The algorithm can be divided into 3 main parts:

1. `mapper_init:` We read the content of the ratings file and store it in a dictionary that resides in memory.

2. `mapper:` We read the `title.principals.tsv` file line by line, and we emit (key, value) pairs where the key is the `nconst`, i.e. the actor's ID, and the value is the rating associated with the `tconst`, i.e. the film title ID, present on the current input line.

3. `reducer:` Now, the reducer only has to group all rating values according to the `nconst`, and compute the average value of the list of ratings. We see that this replicated join pattern removes the need to shuffle data during the reduce phase, hence making it faster than other join patterns.

3.1.2 Launching the Program

Listing 10: Challenge B Average Rating Per Actor with Hadoop MapReduce.

```
./mapreduce-average-script.sh
```

[5] https://mrjob.readthedocs.io/en/latest/

3.1.3 Results

The ratings obtained are stored in the file named `workdir/outputs/mapreduce-average_-ratings_per_actor.txt`.

Listing 11: Contents of `workdir/outputs/mapreduce-average_ratings_per_-actor.txt` file.

```
"nm0000001"     7.121962616822428
"nm0000002"     7.196808510638296
"nm0000003"     6.307407407407406
"nm0000004"     7.105882352941177
...
"nm8400924"     6.6
"nm8400940"     6.1
"nm8400941"     6.1
"nm8400942"     6.1
...
"nm9993693"     6.5600000000000005
"nm9993694"     8.833333333333334
"nm9993703"     7.894117647058823
"nm9993713"     7.584210526315788
...
```

The execution time took around `4 minutes`.

3.2 Spark Implementation

3.2.1 Code Implementation

For calculating the average rating per actor and per producer, the implementation approach is equivalent. Below are the main steps involved in the process:

1. Data from the files `title.principals.tsv` and `title.ratings.tsv` are loaded into DataFrames. These DataFrames are then filtered to include only entries classified as *actor*, *actress*, or *producer*. Subsequently, they are joined on the common identifier (`tconst`) to correlate principal roles with movie ratings.

2. The `.avg` function is applied to the `averageRating` column to calculate the mean rating, and the results are sorted in descending order to highlight those with the highest ratings.

3.2.2 Launching the Program

Both programs can be launched with the following commands:

Listing 12: Challenge B Average Rating Per Actor with Spark.

```
./scala-script.sh AverageRatingPerActor
```

Listing 13: Challenge B Average Rating Per Producer with Spark.

```
./scala-script.sh AverageRatingPerProducer
```

3.2.3 Results

The ratings obtained are stored in the file named `workdir/outputs/spark-average_-ratings_per_actor.txt` or `workdir/outputs/spark-average_ratings_per_producer.txt` and are organized as shown below.

Listing 14: Contents of `workdir/outputs/spark-average_ratings_per_actor.txt` file.

```
[nm7496515,10.0]
[nm8873997,10.0]
[nm3552884,10.0]
[nm3661793,10.0]
...
[nm4594804,7.6]
[nm4084092,7.6]
[nm4371202,7.6]
[nm2805468,7.6]
...
[nm3329221,4.1]
[nm3334573,4.1]
[nm2813075,4.1]
[nm9513871,4.1]
...
```

Upon completion, both codes for computing the average rating per actor and per producer provide similar statistics.

Application ID	Name	Cores	Memory per Executor	Resources Per Executor	Submitted Time	User	State	Duration
app-20240505183018-0001	Spark shell	48	7.0 GiB		2024/05/05 18:30:18	root	FINISHED	46 s

Figure 2: Statistics accessible from http://localhost:8080/ after completion.

4 Challenges Faced

- Making the Hadoop MapReduce implementation work is hard; making it efficient is very hard.
- Some considerable time was spent on understanding how to use Spark efficiently.
- It took us some time to figure out we had to install Python on all the machines as the errors we were getting were not descriptive.
- Learning to debug using Hadoop

- Discovering whether the problem is code-related or configuration-related was the hardest part due to the generic Java error thrown by the Hadoop cluster.
- Solving the bacon number problem with MapReduce as a constraint was challenging as it requires an unintuitive approach to find a solution.

5 Description of everyone's involvement in the assignment

A considerable amount was spent among us to figure out the setup, and upon the completion of this part, the work was divided among us as follows:

1. Jordi: Spark implementation for both challenges. The implementation was rather straightforward. Although understanding the Spark framework took time, Jordi worked very efficiently and did not need any help for both challenges.
2. Aly and Loïc: MapReduce implementation. The concept was rather tricky to completely grasp, and the majority of our working time was spent trying to design a solution that would effectively use MapReduce jobs. Once we figured out the design, the implementation went rather smoothly. Jordi was already familiar with the Hadoop cluster intricacies, and running both MapReduce solutions on Hadoop only required a little debugging thanks to him.

Everybody participated in writing the report and in the creation of the video. The report themes were split as explained above, and the whole video was prepared and filmed as a team.

References

[1] Christophe Debruyne. Info8002 topics in distributed systems: Resilient distributed datasets and apache spark.

[2] Jimmy Lin. Map-reduce applications: Counting, graph shortest paths. https://gktcs.com/media/Reference2/Surendra%20Panpaliya/C-Edge_Hadoop/L11-MapReduce-Dijkstra-BFS.pdf.

[3] Donald Miner and Adam Shook. *MapReduce design patterns: building effective algorithms and analytics for Hadoop and other systems*. " O'Reilly Media, Inc.", 2012.

[4] Gregory Provan. Graph algorithms in mapreduce. https://www.cs.ucc.ie/~gprovan/CS4407/2020/Lectures/L1-Introduction.pdf.