« Big Data » course

CentraleDigitalLab@LaPlateforme - 2022-2023

# Lab -2: Advanced distributed programming with Spark

Gianluca Quercini & Stéphane Vialle

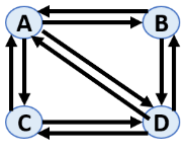
In this lab you'll execute Spark programs on the distributed environment of a Spark-HDFS cluster at CentraleSupélec's Teaching Data Center. For details of the implementation and access to distributed files, see part 1 of lab 1.

## Exercise 1: Finding common friends in a social network.

We consider a social network graph encoded in a text file.

- One line of the file is a list of identifiers separated by a comma: A, B, C, D
  This means that A is friend with B, C et D.
- The set of lines in the file describes the relation "is friend with" in the graph. Ex :

A, B, C, D
B, A, D
C, A, D
D, A, B, C




- We assume that the relations are symmetrical: if we have A, B, then we also have B, A.
  We'd like to obtain the list of common friends of any two individuals, as a key-value RDD:

((A, B), [D])
((A, C), [D])
((A, D), [B, C])
((B, C), [A, D])
((B, D), [A])
((C, D), [A])

Note: each pair should appear at most once in the output (as shown above). If we list the common friends of the pair (A, B), then we won't list the common friends of (B, A).

We use the following files under the HDFS folder **hdfs://sar01:9000/data/social-network/**:

| | |
|---|---|
| *sn_tiny.csv* | :Small social network, used for testing. |
| *sn_10k_100k.csv* | :Social network with roughly $10^4$ individuals and $10^5$ links. |
| *sn_100k_100k.csv* | :Social network with roughly $10^5$ individuals and $10^5$ links. |
| *sn_1k_100k.csv* | :Social network with roughly $10^3$ individuals and $10^5$ links. |
| *sn_1m_1m.csv* | :Social network with roughly $10^6$ individus et $10^6$ liens environ. |

### Questions 3.1 : conception of a solution

- Copy the following template to your home directory:

  **`~cpu_vialle/DCE-Spark/template_common_friends.py`**

- Propose a solution using a *reduceByKey*(…)
- Propose a solution using a *groupByKey*()
- Test and validate your implementations on file *sn_tiny.csv*.

### Question 3.2 : tests and performances

- Execute your implementations on the other test files and note down the execution times.
- Note all your measures in the following table.

| Input file | sn_tiny | sn_10k_100k | sn_100k_100k | sn_1k_100k | sn_1m_1m |
|---|---|---|---|---|---|
| File size (MB) | | | | | |
| groupByKey (s) | | | | | |
| reduceByKey (s) | | | | | |
| Avg nb of friends (Q 3.3) | - | | | | |
| Nb of intermediate pairs (Q 3.4) | - | | | | |

### Question 3.3: computing the minimum, maximum and average degree of the graph nodes.

Write a Spark program that computes the minimum, maximum and average node degree in the social network graph (the degree of a node is the number of its friends).

**Note**: keep using a RDD until the very end of the computation and compute the minimum, maximum and average values in the RDD (don't leave the RDD to compute the values on Python lists, which would make your code not parallelizable).

**Note**: the average degree must be a floating-point number.

- Execute your program on the four test files to compute the minimum, maximum and average number of friends of an individual.

| Input file | sn_tiny | sn_10k_100k | sn_100k_100k | sn_1k_100k | sn_1m_1m |
|---|---|---|---|---|---|
| Min nb of friends | - | | | | |
| Max nb of friends | - | | | | |
| Avg nb of friends | - | | | | |

- Complete the table in question 3.2 with the average number of friends of a node.

### Question 3.4 : performance analysis

- As we have seen in the tutorial, we suppose that the number of friends of each node is equal to the average number of friends.
- Compute the intermediate pairs ((A, B), X) generated before the *wide* operation (*groupByKey* or *reduceByKey*) by your program that computes common friends.

  Justify your calculation.
- Complete the table in question 3.2 with the number of generated intermediate pairs.
- Analyse the obtained performances:
  - Plot the execution time (1) as a function of the file sizes, then (2) as a function of the average number of friends, then (3) as a function of the number of the generated intermediate pairs.

    You're free to choose either linear, logarithmic, or semi-logarithmic scales.
  - Which plot better explains the evolution of the execution time of the solution using *reduceByKey* ?
  - What about the solution using *groupByKey*?
- Is the number of intermediate pairs a good predictor of the program workload and execution time?