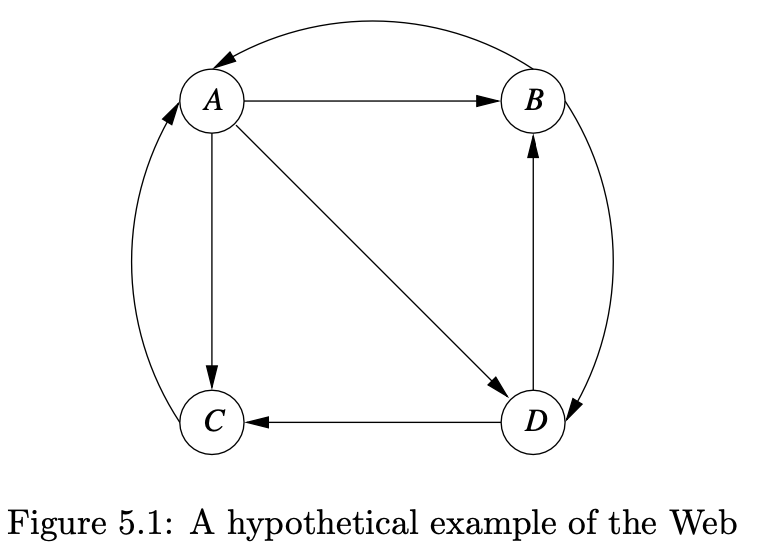
**Projet d’Olivier Brunet et Slim Kachkachi**

**Project description.** The project consists of implementing the standard PageRank algorithm in Spark and Scala, along the lines of the examples seen in the course. More in details, you are expected to :

1. **P**rovide a first implementation without any kind of optimisation, and show how it works on a simple A-B-C-D graph seen in the course, by putting a *cul-de-sac* and a selfedge (“dead-end”) on one node.

**1iere partie implémentation du graphe sans cul-de-sac ni ‘dead end’**

Dans cette partie du projet, l’algorithme en Scala du PageRank a été testé sur le graphe illustré ci-dessous (identique à celui du cours et du livre ‘Mining Massive Data Chapitre 5 [1]), ce qui a permis quelques points de contrôles sur les valeurs calculées du ‘pagerank’.



L’implémentation s’appuie sur deux RDDs avec un pour le ‘pagerank’ de chaque nœud (appelé ‘ranks’) et un autre pour la matrice de connexion des ‘vols’ entre les aéroports (appelé ‘links’).

A noter que le choix d’une structure de données de type RDDs permettra de bénéficier du clustering et de paramétrer dans un second temps le partitionnement.

La matrice de connexion ‘links’ a été modélisée selon le principe suivant (et comme présentée dans [1]) où les lettres représentent ici des aéroports :

|  |  |
| --- | --- |
| Aéroport de départs | Aéroports directement desservis |
| A | (B,C,D) |
| B | (A,D) |
| C | (A) |
| D | (B,C) |

Ce qui se traduit par les RDDs ‘ranks’ et ‘links’ suivants dans le cas d’un graphe avec des lettres (\*) :

ranks: org.apache.spark.rdd.RDD[(String, Double)]

links: org.apache.spark.rdd.RDD[(String, Array[String])]

Une autre solution pour ‘links’ existe sous forme de List() et non d’Array[ ] :

links: org.apache.spark.rdd.RDD[(String, List[String])]

(\*) pour le graphe issu des vols entre aéroports, les collections seront des RDDs d’entiers ( RDD[(Int, Array[Int])] ), les aéroports étant identifiés par des entiers.

A noter que dans [1], la matrice de connexion est légèrement différente et présente en plus pour chaque aéroport, le nombre d’aéroports directement raccordés comme le montre le tableau suivant.

|  |  |  |
| --- | --- | --- |
| Aéroport de départ | Aéroports directement desservis | Nombres de liens |
| A | (B,C,D) | 3 |
| B | (A,D) | 2 |
| C | (A) | 1 |
| D | (B,C) | 2 |

Un essai comparatif de performances a été réalisé et le RDD ‘links’ a été implémenté de la façon suivante : org.apache.spark.rdd.RDD[(String, (Array[String], Int))] (code en annexe).

Concernant l’initiation du ‘pagerank’ (\*), les deux cas suivant été testés :

Cas 1) *avec* et Cas 2) a*vec* où n est le nombre de nœuds du graphe

(\*) pagerank d’initialisation = \*M + (1- où M est la matrice de connexion du graphe, le ‘pagerank’ à l’étape i et .

Et enfin afin de vérifier l’impact de cul-de-sac et /ou ‘dead-end’ sur le calcul des ‘pageranks’, le code a été exécuté avec et sans la taxation.

Ce qui donne le code suivant avec ses variations (cf. notebook « Pagerank\_basic\_nonOpt\_v1 ») :



A noter que le même algorithme a été testé avec des List() pour le RDD ‘links’ plutôt des Array()…sans différence de performance identifiée.

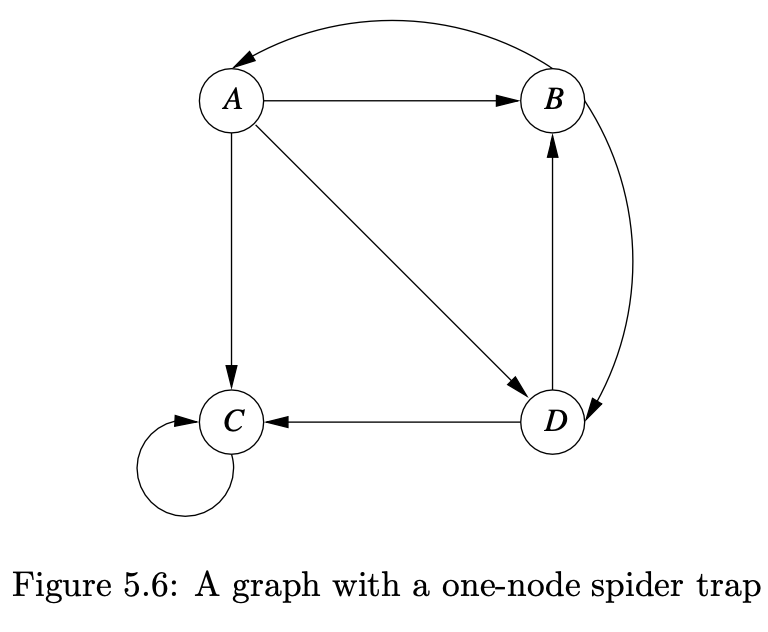
Après avoir exécuté le code selon les différents cas, les résultats suivants ont été obtenus :

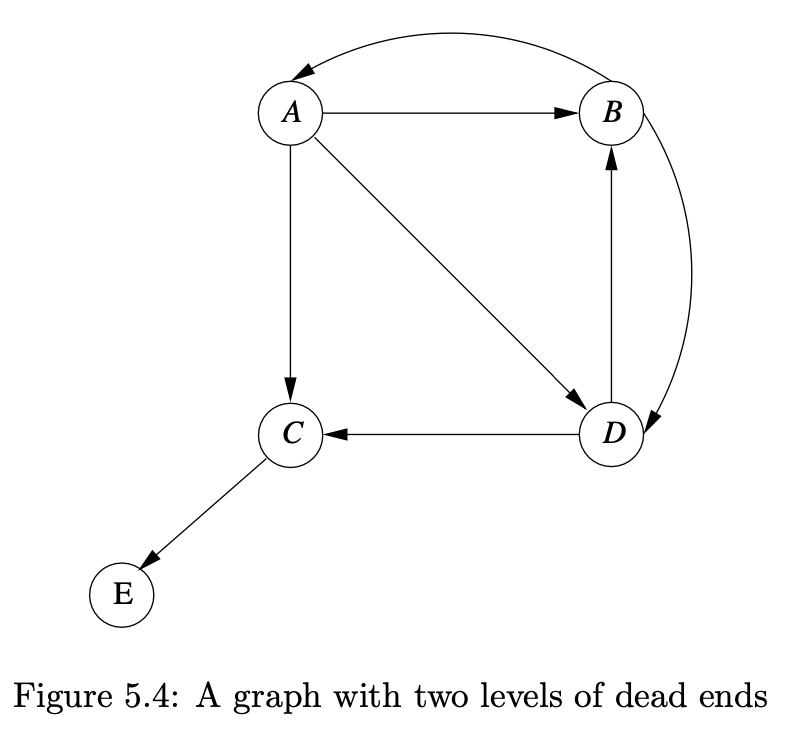
|  |  |
| --- | --- |
| **Version** | **Pageranks obtenu (au bout de 10 itérations)** |
| **Avec** taxation & v0 =[1,…1] | (A,1.298), (B,0.901), (C,0.901), (D,0.901) |
| **Sans** taxation & v0 =[1,…1] | (A,1.333), (B,0.889), (C,0.889), (D,0.889) |
|  |  |
| **Avec** taxation v0 =[1/n…1/n] | (A,0.325), (B,0.225), (C,0.225), (D,0.225) |
| **Sans** taxation v0 =[1/n,…1/n] | (A,0.333), (B,0.222), (C,0.222), (D,0.222) |

* Manifestement, la présence de la taxation dans ce cas idéal ne change pas la limite de convergence des ‘pageranks’.
* Et notons que les proportions entre les ‘pageranks’ obtenus avec des valeurs d’initialisation à 1 plutôt que 1/n sont préservées…ce qui est le point important.

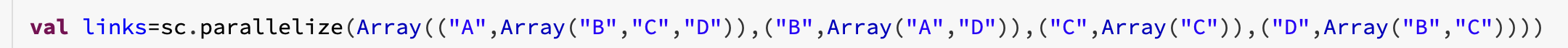
**2ieme partie : implémentation du graphe avec cul-de-sac et ‘dead end’**

Le graphe est modifié pour correspondre aux cas suivants :

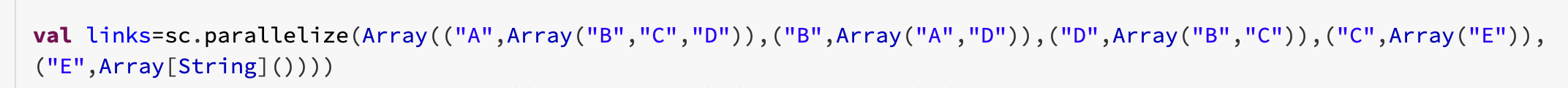




La déclaration de ‘links’ devient dans le cas du spider-trap :



La déclaration de ‘links’ devient dans le cas de deux niveaux de ‘dead-end’ :



Le reste du code utilisé avec ou sans taxation reste le même. Nous avons juste augmenté le nombre d’itérations, pour rendre la convergence plus nette. A noter que pour la suite, nous initialiserons le ‘pagerank’ d’initialisation avec les valeurs en 1/n.

Les ‘pagesrank’ obtenus sont :

|  |  |
| --- | --- |
| Version | Pageranks après 50 itérations |
| Cul de sac **sans** taxation | (A,0.000), (B,0.000), (C,1.000), (D,0.000) |
| Cul de sac **avec** taxation | (A,0.082), (B,0.106), (C,0.706), (D,0.106) |
|  |  |
| Dead End **sans** taxation | (A,0,000), (B,0.000), (C, 0.000), (D, 0.000),(E,0.000) |
| Dead End **avec** taxation | (A,0.066), (B,0.085), (C,0.085), (D,0.085),(E,0.102) |
| Graphe sans Dead End & taxation | (A,0.222),(B,0.4444),(D,0.3333) |

* La taxation dans le cas du cul-de-sac évite que ce nœud ne devienne l’unique probabilité de « destination ».
* La taxation dans le cas du ‘dead-end’ évite une converge vers zéro de toutes les probabilités de destinations.
* A noter que l’on constate une différence dans les valeurs des probabilités de destination, dans le cas d’un graphe nettoyé de tous les ‘dead-end’ par rapport au graphe les ayant conservé. Cela semble normal si considère dans le cas du graphe ‘nettoyé’ que nœuds restants sont moins nombreux à se partir les probabilités de destination.

**-B)** adapt the implementation so that a larger graph of your choice can be processed; please choose a sufficiently large graph. In this case provide both the baseline implementation and the optimised one with partitioning tuning. Perform a simple experimental analysis in order to compare performances of the two versions.

**Préambule : observation de la distribution par défaut des tuples sur les différentes partitions**

Avant de commencer, nous nous sommes intéressés à la distribution des RDDs ‘links’ et ‘ranks’ sur les différentes partitions au cours du cycle d’itération de l’algorithme selon deux cas : avec et sans partitioner (en prenant le **HashPartitioner** qui distribuera selon le modulo des clés).

**Pour le cas sans partitioner**, l’environnement spark de travail (dans ce cas Databrick) dispose de 8 partitions par défaut. Les tableaux ci-dessous permettent de suivre l’évolution de la répartition des clés des deux RDDs (\*) sur ces différentes partitions lors des premières itérations du pagerank.Ici ‘Contribs’ est le résultat de la jointure entre ‘ranks’ et ‘links’.

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Partition 4 | Ranks v0 | Links | Contribs | Ranks v1 | Ranks v2 |
|  |  | B | D |  |
|  |  | C |  |  |
|  |  |  |  |  |
|  |  |  |  |  |
| Partition 5 | Ranks v0 | Links | Contribs | Ranks v1 | Ranks v2 |
| C | C |  |  |  |
|  |  |  |  |  |
|  |  |  |  |  |
|  |  |  |  |  |
| Partition 6 | Ranks v0 | Links | Contribs | Ranks v1 | Ranks v2 |
|  |  |  |  |  |
|  |  |  |  |  |
|  |  |  |  |  |
|  |  |  |  |  |
| Partition 7 | Ranks v0 | Links | Contribs | Ranks v1 | Ranks v2 |
| D | D |  |  |  |
|  |  |  |  |  |
|  |  |  |  |  |
|  |  |  |  |  |

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Partition 0 | Ranks v0 | Links | Contribs | Ranks v1 | Ranks v2 |
|  |  |  |  |  |
|  |  |  |  |  |
|  |  |  |  |  |
|  |  |  |  |  |
| Partition 1 | Ranks v0 | Links | Contribs | Ranks v1 | Ranks v2 |
| A | A | B | A | A |
|  |  | C |  |  |
|  |  | D |  |  |
|  |  |  |  |  |
| Partition 2 | Ranks v0 | Links | Contribs | Ranks v1 | Ranks v2 |
|  |  | A | B | B |
|  |  | D |  |  |
|  |  |  |  |  |
|  |  |  |  |  |
| Partition 3 | Ranks v0 | Links | Contribs | Ranks v1 | Ranks v2 |
| B | B | C | C | C |
|  |  |  |  |  |
|  |  |  |  |  |
|  |  |  |  |  |

(\*) la commande suivante a permis de retrouver l’index de la partition : links.mapPartitionsWithIndex{

(index, itr) => itr.toList.map(x => x.\_1 + "#" + index).iterator }.collect()

* On constate que les mêmes clés des deux RDDs sont colocalisés (et donc co-partitionnés) à l’initialisation et ne le sont plus à partir de la 2ieme itération à l’exception de la clé ’A’. On peut s’attendre dans ce cas à des performances dégradées lors des jointures, puisqu’un shuffle sera nécessaire pour la jointure à chaque nouvelle itération.

**Pour le cas avec partitioner sur le RDD ‘links**’, on constate qu’à partir de la 2ieme itération les clés de même valeur entre les RDD ‘links’ et ‘ranks’ sont colocalisées. On doit s’attendre à de meilleures performances pour la jointure.

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Partition 4 | Ranks v0 | Links | Contribs | Ranks v1 | Ranks v2 |
|  |  |  |  |  |
|  |  |  |  |  |
|  |  |  |  |  |
|  |  |  |  |  |
| Partition 5 | Ranks v0 | Links | Contribs | Ranks v1 | Ranks v2 |
| C |  |  |  |  |
|  |  |  |  |  |
|  |  |  |  |  |
|  |  |  |  |  |
| Partition 6 | Ranks v0 | Links | Contribs | Ranks v1 | Ranks v2 |
|  |  |  |  |  |
|  |  |  |  |  |
|  |  |  |  |  |
|  |  |  |  |  |
| Partition 7 | Ranks v0 | Links | Contribs | Ranks v1 | Ranks v2 |
| D |  |  |  |  |
|  |  |  |  |  |
|  |  |  |  |  |
|  |  |  |  |  |

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Partition 0 | Ranks v0 | Links | Contribs | Ranks v1 | Ranks v2 |
|  | B | A | B | B |
|  | D | D | D | D |
|  |  | B |  |  |
|  |  | C |  |  |
| Partition 1 | Ranks v0 | Links | Contribs | Ranks v1 | Ranks v2 |
| A | A | B | A | A |
|  | C | C | C | C |
|  |  | D |  |  |
|  |  | C |  |  |
| Partition 2 | Ranks v0 | Links | Contribs | Ranks v1 | Ranks v2 |
|  |  |  |  |  |
|  |  |  |  |  |
|  |  |  |  |  |
|  |  |  |  |  |
| Partition 3 | Ranks v0 | Links | Contribs | Ranks v1 | Ranks v2 |
| B |  |  |  |  |
|  |  |  |  |  |
|  |  |  |  |  |
|  |  |  |  |  |

* **A noter qu’en imposant le Hashpartioner(*x*) sur les deux RDD**s (‘links’ et ‘ranks’), on obtient une colocalisation dès le départ avant même la 1iere itération…c’est-à-dire une situation identique une fois la 1iere itération passée.

**Partie 1 : création du graphe et du pagerank à partir d’un fichier CSV et gestion de la boucle de convergence du pagerank**

Pour réaliser cette implémentation, deux aspects ont été traités :

1. Le traitement du fichier CSV afin de produire les deux RDDs (links et ranks).
2. La réalisation d’une boucle afin de d’arrêter les itérations à partir d’un certain seuil au-delà duquel on considère la converge des ‘pageranks’ réalisée.

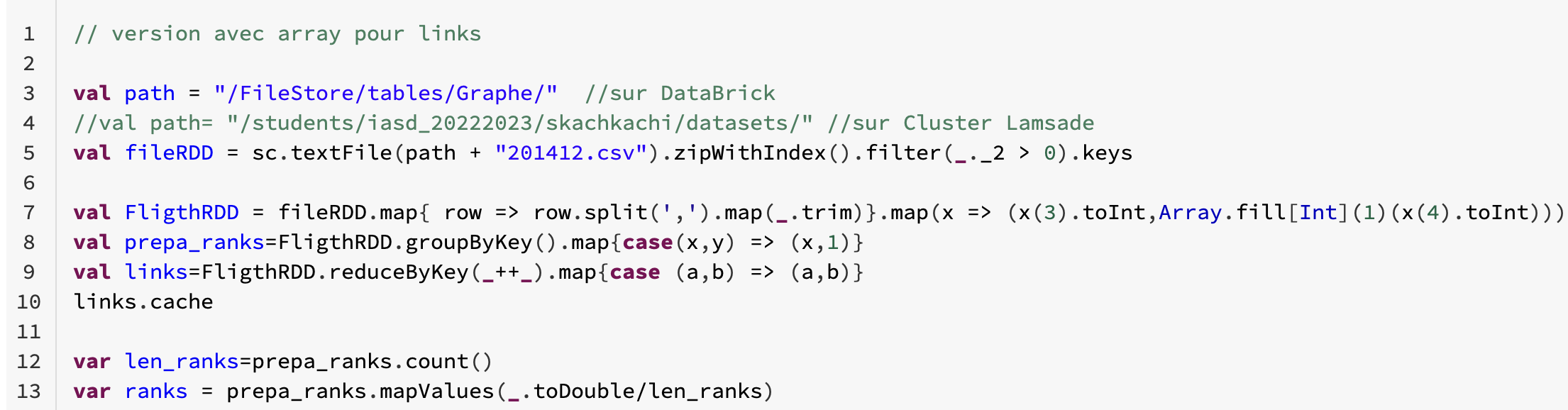
Concernant le point 1, nous sommes partis du fichier « 201412.csv » dont la taille est de 33,2 Mo. Afin de produire les RDDs, il a fallu :

1. Supprimer de la 1iere ligne du fichier CSV (contenant les labels des colonnes).
2. Sélectionner les seules colonnes aéroports de départ et aéroports de destination.

Point important : plusieurs vols desservent les mêmes aéroports. On va donc se retrouver, pour un aéroport de départ donné, avec plusieurs fois le même aéroport de destination dans la liste des « nœuds raccordés ».

Il a eu un choix à faire entre modéliser seulement l’existence d’une connexion entre deux aéroports ou tenir compte du nombre de vols. Nous avons choisi la 2ieme option car les ‘pageranks’ résultant nous semblaient mieux traduire la fréquentation des aéroports.

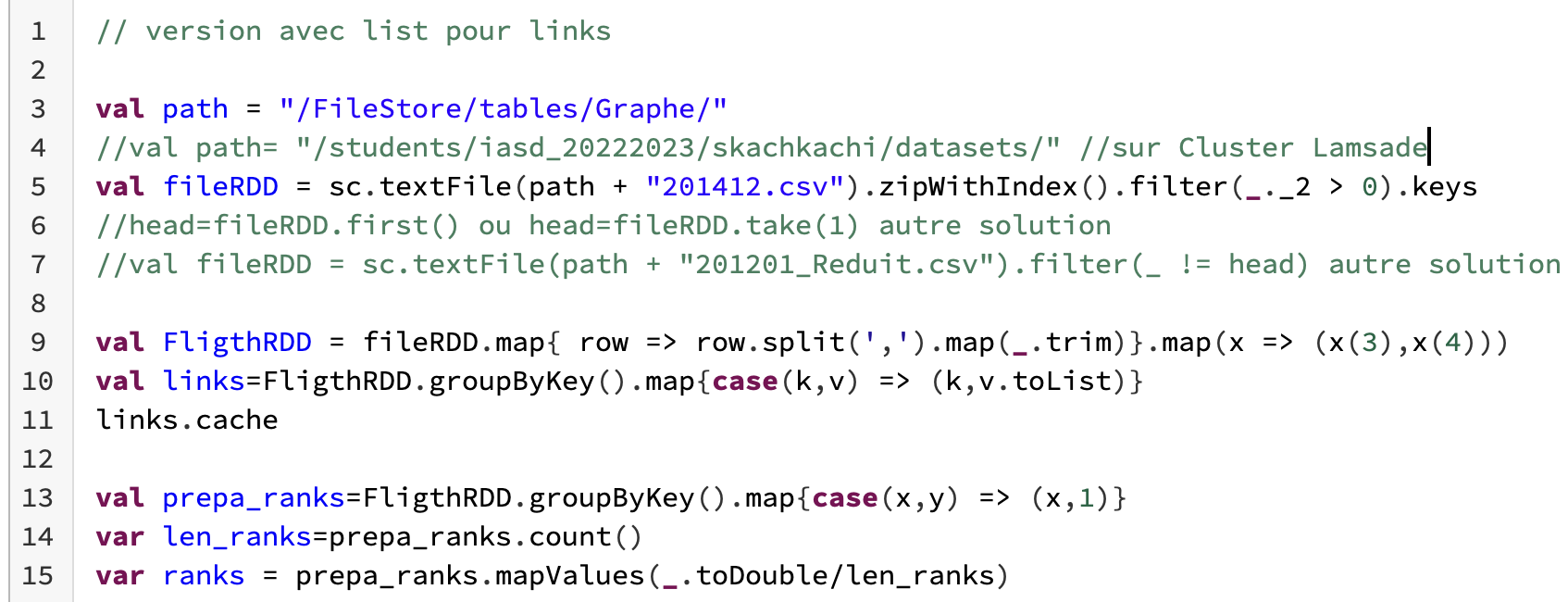
Le code est le suivant pour la création des RDDs pour la version sans ‘partitioner’ (cf. notebook Pagerank\_CSV\_nonOpt\_v1) :



Sinon, limiter la modélisation du graphe aux seules connexions ne change quasiment rien le code ; il suffit de supprimer tous les aéroports de destination en doublon (via un « .distinct ») comme cela est illustré ci-après.

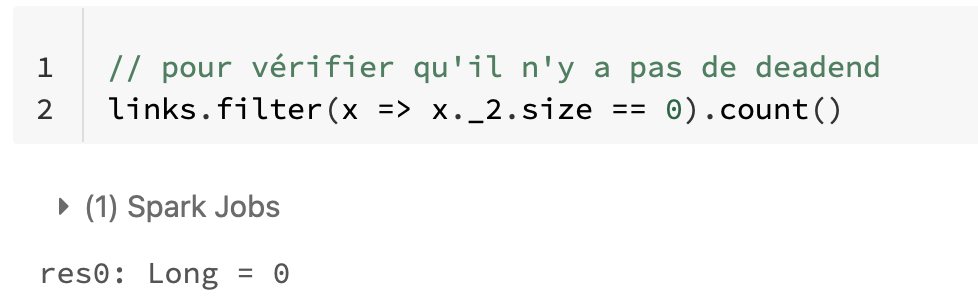


Dernier point, l’utilisation d’une liste (plutôt que d’un array) pour modéliser le RDD ‘links’ donne le même résultat. Le code est présenté ci-dessous (ici sans le partitioner).



A noter que le graphe ‘links’ issu du fichier ‘201412.csv » comprend 312 nœuds.

Avant de lancer l’exécution pour le calcul des ‘pageranks’, on peut accessoirement s’assurer de l’absence de ‘dead-end’ (ce qui n’a rien de surprenant en soit, l’absence de destination pour un avion n’étant pas un cas normal).



Concernant le point 2 (gestion de la convergence via une boucle), le code est :



Le principe est de déterminer entre deux itérations, le nombre de coordonnées du ‘pagerank’ dont la différence avec les valeurs la précédente itération reste supérieur à seuil déterminé (le code est repris du cours sur le k-means). Lorsque ce nombre atteint zéro, l’itération peut s’arrêter. Par précaution on ajoute une seconde condition à satisfaire, comme un nombre maximal d’itérations à ne pas dépasser (ici 50).

**Partie 2 : comparaison des performances (temps d’exécution) entre des versions optimisées et non optimisées**

Nous avons comparé trois implémentations :

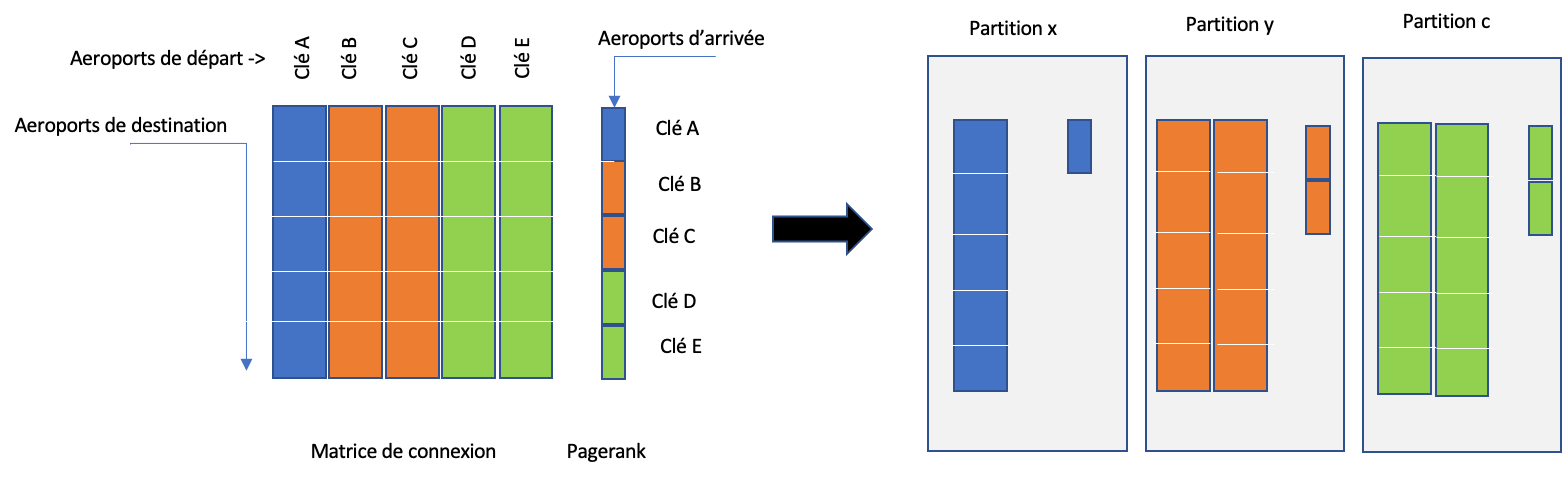
1. Non optimisée
2. Optimisée avec le ‘HashPartitioner() avec caching’ du RDD ‘links’
3. Optimisée avec le ‘BroadcastJoinPartitioner() avec caching’ du RDD ‘links’

A priori tester le ‘BroadcastHashJoinPartitioner() semble adapté à ce contexte, puisque l’on peut considérer que le RDD ‘ranks’ est assez petit pour être distribué (dans sa totalité) sur chacune des partitions du RDD ‘links’.

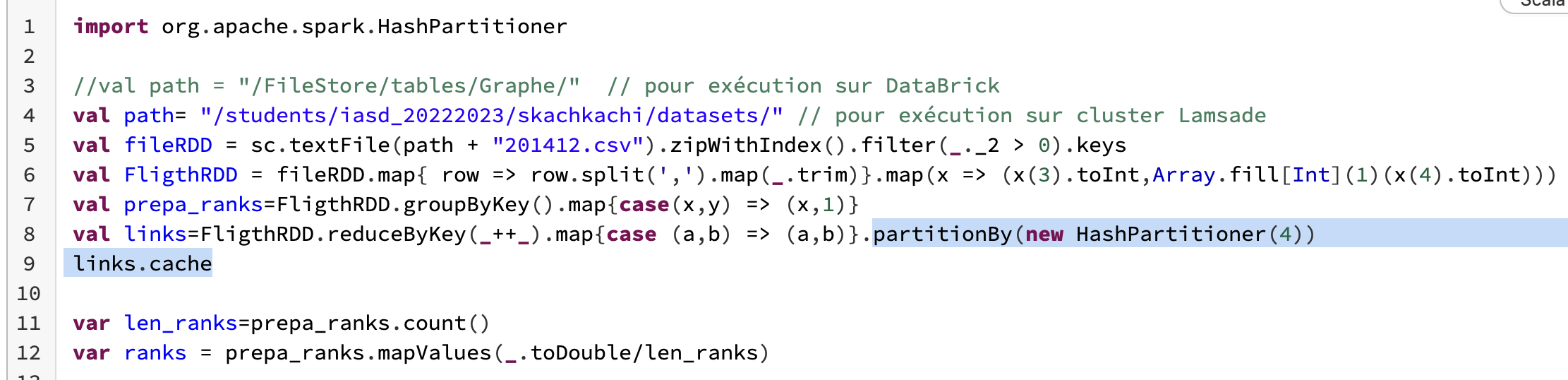
Le ‘cashing’ sur links est utilisé afin de laisser en RAM de chaque ‘executor’ les partitions correspondantes à chacun des modulos des différentes clés, ce RDD restant inchangé à chaque itération ; contrairement au RDD ‘ranks’.

Concernant le HashPartitioner, Il y a un sujet concernant le choix du nombre de partitions. Ce point sera abordé dans le chapitre suivant avec un RDD plus conséquent. Pour l’instant, nous avons choisi arbitrairement un modulo 4 (répartition des tuples sur 4 partitions).

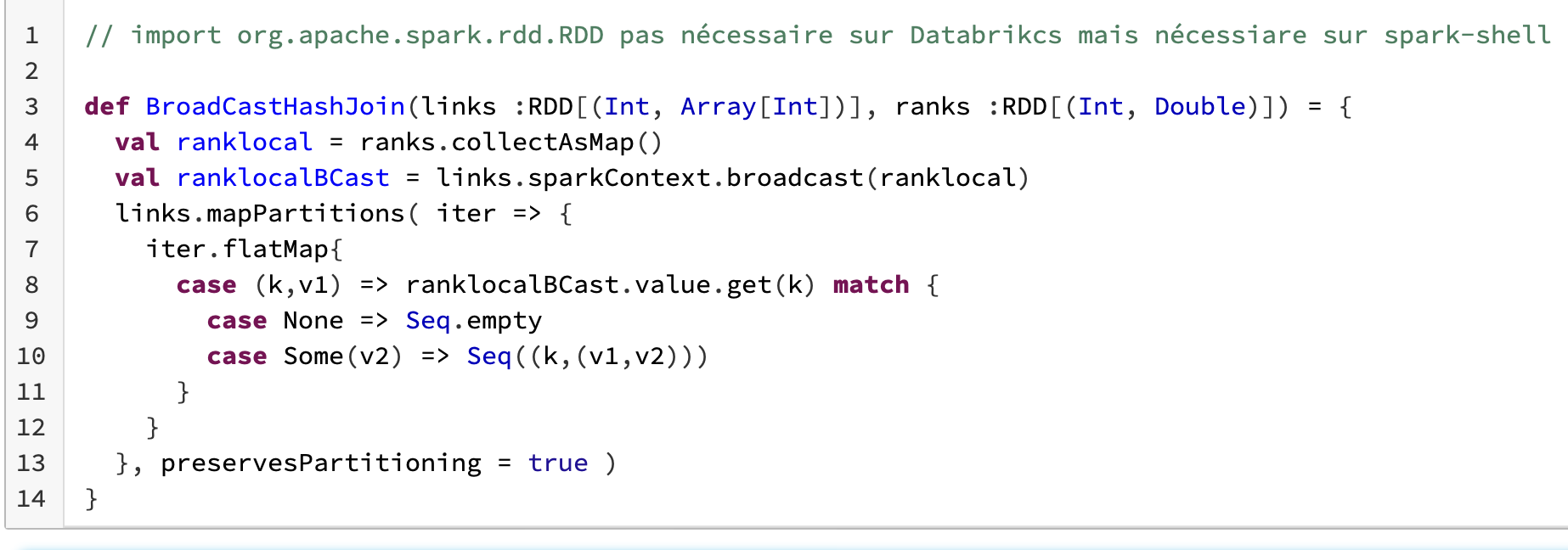
En termes de fonctionnement, on peut remarquer que l’utilisation du HashPartitioner revient à découper la matrice de connexion ‘links’ en ‘tranches’ verticales (par clé de départ) et le ‘pageranks’ en ‘tranches’ horizontales composées des mêmes clés, comme l’illustre le schéma suivant (inspiré de chapitre 2 de [1]), puis à distribuer sur les mêmes partitions les parties de ‘links’ et de ‘ranks’ ayant les mêmes clés :

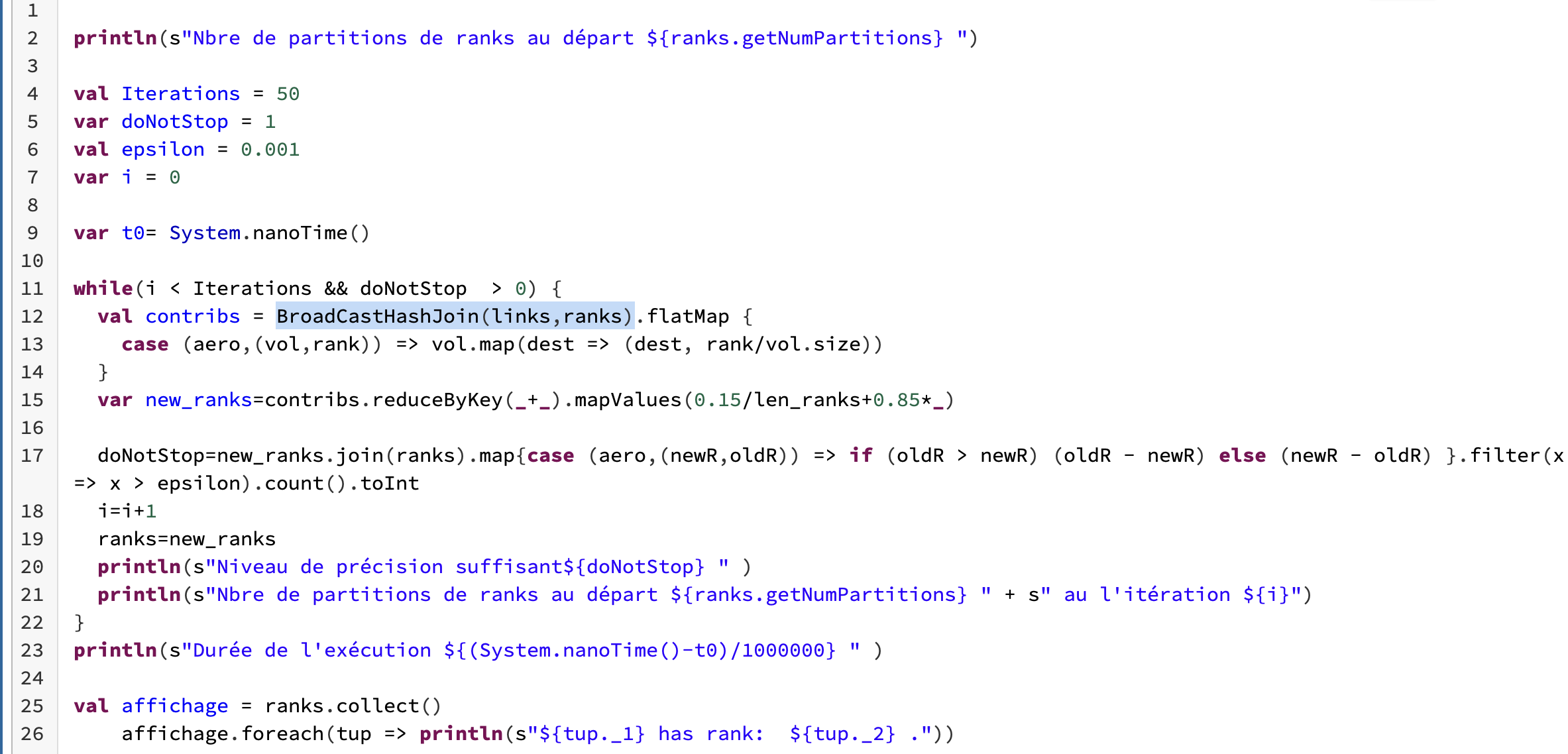


Le code pour le HashPartitioning et caching du RDD ‘links’ est présenté ci-dessous (cf. notebook Pagerank\_HashPartitioner\_v1).



Le code pour le BroadcastJoinPartitioning (adapté du cours) est présenté ci-dessous (cf. notebook Page\_rank\_BroadcastHashJoin\_v1) avec d’une part la définition du ‘BroadcastHashJoinPartitioner’ et d’autre part, le corps de l’itération du ‘pagerank’.





Les tests ont été réalisés sur le cluster de Lamsade (sur la base du fichier 201412.csv de 33,2 Mo, pour 312 noeuds) avec les paramètres par défaut de la session spark (concernant le nombre « d’executors » et de « core » par « executor »).

Les résultats sont présentés dans le tableau ci-dessous. A noter que le RDD ‘links’ une fois créé ne fait plus que 6,2 Koctets (\*).

(\*) Le RDD « links » a été sauvegardé dans HDFS pour récupérer sa taille. Commandes utilisées : 1) « links.saveAsTextFile("/students/iasd\_20222023/skachkachi/201412.csv") » puis 2) « hdfs fsck /students/iasd\_20222023/skachkachi/file.csv -files -blocks -locations »

|  |  |
| --- | --- |
| Différents d’optimisation testés | Temps d’exécution en ms |
| Pagerank non optimisé **avec** caching sur links | 5 051 |
| Pagerank non optimisé **sans** caching sur links | 6 908 |
|  |  |
| Pagerank avec HashPartitioner(4) **avec** caching sur links | 4 339 |
| Pagerank avec HashPartitioner(4) **sans** caching sur links | 4 430 |
|  |  |
| Pagerank avec BroadCastJoinPartioner **avec** caching sur links | 5 503 |
| Pagerank avec BroadCastJoinPartioner **sans** caching sur links | 6 456 |

* L’impact de l’utilisation du HashPartitioner est significatif même sur un jeu de données très réduit.
* Notons que sur ce jeu de données a) l’effet positif du caching de ‘links’ n’a pas d’effet très marqué dans le cas du ‘HashPartitioner’ et b) le ‘BroadcastJoinPartitioner’ semble être moins efficace que le ‘HashPartitioner’ voire contre-productif…résultat inattendu car son fonctionnement à base de map (et non de jointure) aurait dû se montrer plus performant (comme indiqué dans le cours).

Une explication pourrait être que chaque partition se retrouve avec la totalité de ‘ranks’, avec un map à réaliser sur l’ensemble de ce RDD, contrairement au HashPartitioner ?

-**C)** consider the dataset for Flight project and use Spark to build the airport-graph, as discussed in class. Apply your two implementations to the obtained graph and describe their behaviour on this graph, typically in terms of execution time.

Les tests ont été réalisés sur l’ensemble des vols réalisés en 2014 (après la concaténation de tous les fichiers de 2014). La taille finale du fichier final est de l’ordre de 360 Mo mais une fois transformé en RDD, celui-ci ne fait plus que 3,344 Mo (un fichier d’une taille encore très réduite, qui sans doute ne permettra pas de bénéficier des effets d’échelle des partitioners) pour 325 noeuds.



A noter que lors des différents essais, nous avons pu constater de grandes variations de temps d’exécution pour un même jeu de test. Il faut donc considérer que les ordres de grandeur.

**Dans une première série** de tests, nous avons utilisé une configuration ‘spark’ avec 4 ‘cores’ pour se caler sur un partitionnement en 12 parties dans le cas du Hashpartitioner(12), en utilisant la commande suivante lors de la création de la session spark-shell : « spark-shell --num-executors 3 --executor-memory 4g --executor-cores 4 » (\*)(1).

Les résultats obtenus sont présentés ci-dessous :

|  |  |
| --- | --- |
| Différents essais | Durée en ms |
| Pagerank non optimisé **avec** caching sur links | 23 947 |
| Pagerank non optimisé **sans** caching sur links | 24 408 |
| Pagerank avec Hashpartitioner **sans** caching sur links | 22 968 |
| Pagerank avec Hashpartitioner **avec** caching sur links | 22 576 |
| Pagerank avec BroadCastJoinPartitioner **sans** caching sur links | 25 842 |
| Pagerank avec BroadCastJoinPartitioner **avec** caching sur links | 24 342 |

* ‘Hashpartitioner’ confirme sa supériorité.
* Le ‘BroadcastJoinPartitioner’ comme lors des essais précédents se montre moins efficace sur ce jeu de donnée que le Hashpartitioner. Il serait intéressant de faire un essai avec un fichier de taille réellement importante.

(\*) Le fonctionnement idéal est avec des parties pas trop grandes : n cores à disposition -> k\*n partitions avec k entre 1 et 4 en générale (c’est-à-dire 3 à 4 fois plus de partitions que de cores)

**Dans une 2ieme série de tests**, nous nous sommes focalisés sur le ‘HashPartitioner’ avec 24 partitions en testant plusieurs configurations de la session spark. Les résultats obtenus sont présentés dans le tableau suivant :

|  |  |
| --- | --- |
| Différents essais | 1ier run en ms |
| spark-shell --num-executors 8 --executor-memory 4g --executor-cores 3 | 28 655 |
| spark-shell --num-executors 4 --executor-memory 4g --executor-cores 3 | 25 536 |
| spark-shell --num-executors 2 --executor-memory 4g --executor-cores 3 | 30 059 |
| spark-shell --num-executors 1 --executor-memory 4g --executor-cores 3 | 26 107 |

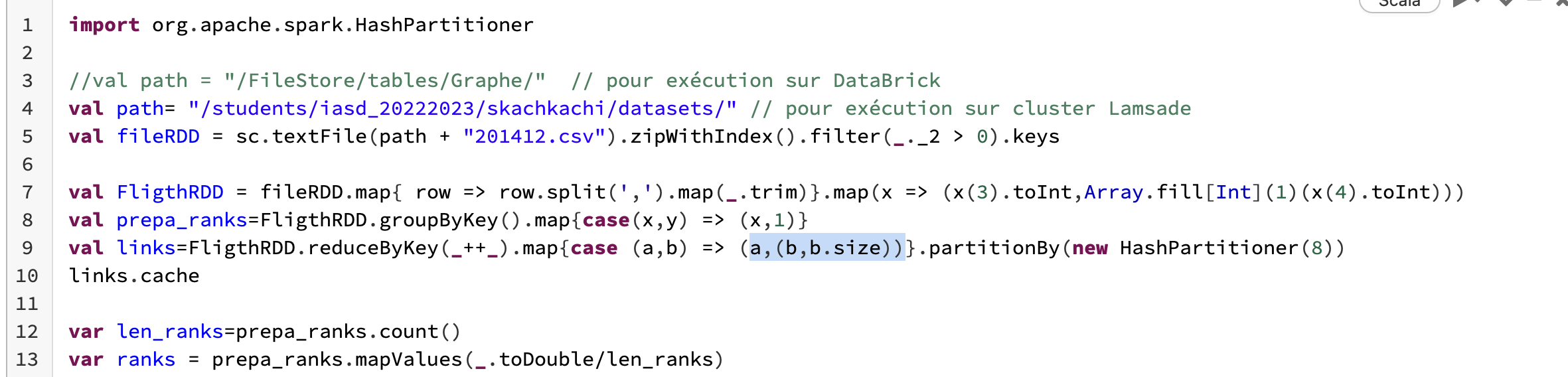
* Difficile de constater une tendance générale sur un échantillon d’essais aussi réduit, mais il apparait qu’un nombre important ‘d’executors ‘n’apporte pas de gain sur le temps d’exécution, et pourrait même être contre-productif (à cause de ‘l’overhead’ de gestion des ‘workers’).

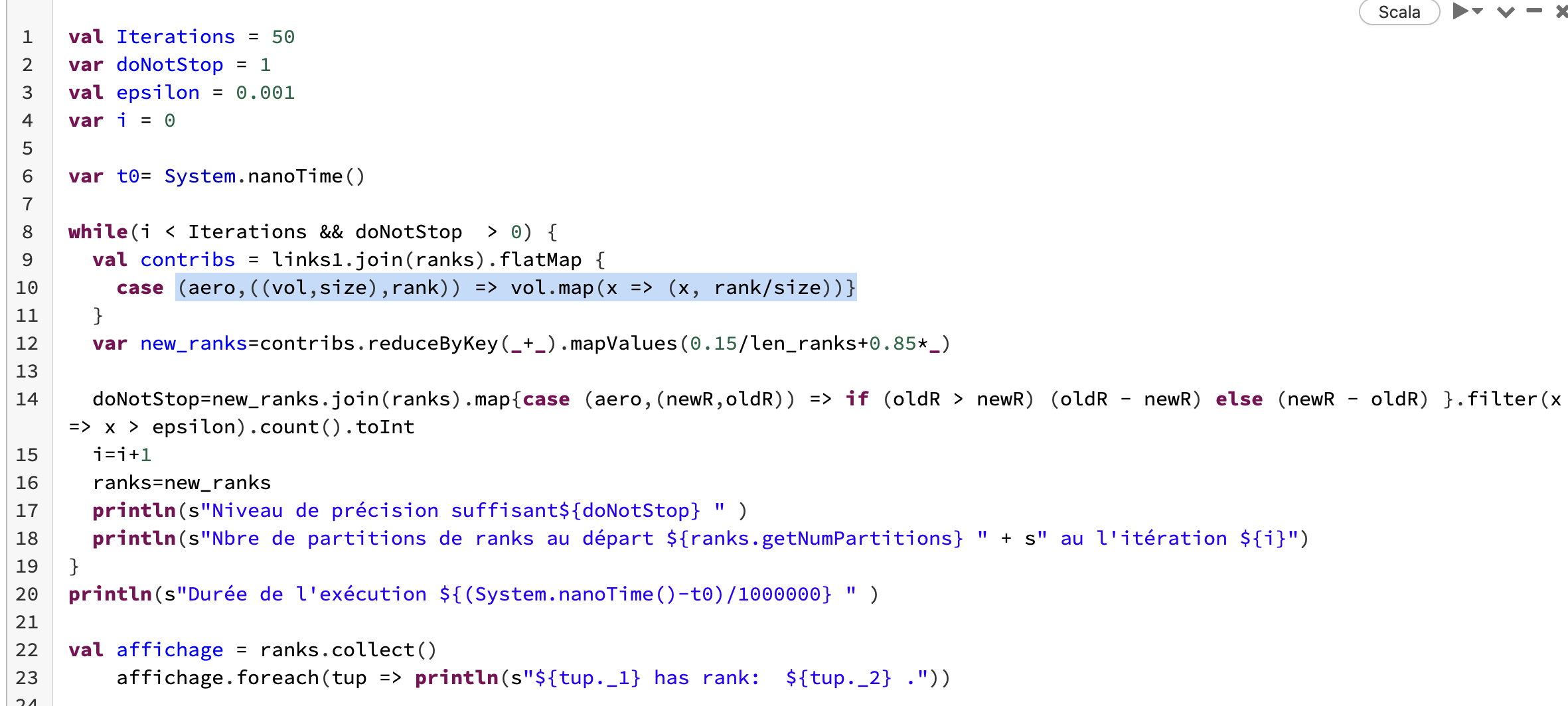
(1) Rappel : la documentation indique par défaut un nombre de « core » par « executor » de 1 et un nombre total « d’executor » de 2.

**Annexes**

**Illustration du pagerank avec la matrice de connexion où la taille du nombre de destinations par aéroport est précalculée**. Les quelques modifications du code sont indiquées en surlignées bleu ci-dessous (notebook .

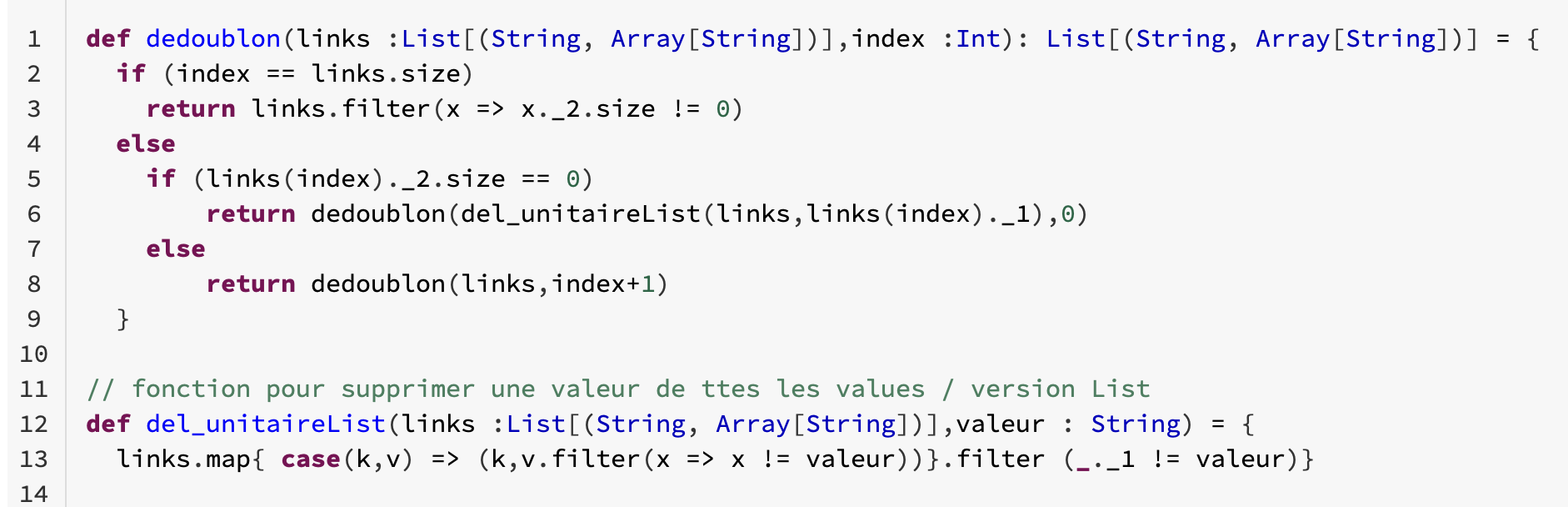
Quelques essais sur le jeu de données « 201412.csv » n’ont montré aucun changement significatif du temps d’exécution d’observé (peut-être le jeu de données est-il trop petit ?).





**Essai de suppression automatique des ‘dead-ends’ par approche récursive :**

Nous avons testé une solution pour supprimer de façon récursive les ‘dead-ends’ dont une 1iere réflexion est présentée ci-dessous (cf. notebook agerank\_SupRecurcif\_deadEnd).

****