PSL
UNIVERSITÉ PARIS

Mémoire de Recherche CPES3
Le traitement massif d'informations : efficience, optimalité et extensibilité

Auteur :
Arthur Cordier

Encadrant de Recherche :
Dario Colazzo

Enseignants référents :
Virginie Gabrel
Guillaume Vigeral

25 juin 2020

Table des matières

Remerciements

Je tenais à remercier mes deux enseignants référents M. Guillaume VIGERAL et Mme Virginie GABREL pour m'avoir permis de choisir tardivement un sujet de mémoire de recherche personnel et adapté à ma situation lors de ce second semestre de CPES 3.

Je tenais aussi à remercier mon encadrant de recherche, M. Dario COLAZZO, qui m'a proposé un sujet sur-mesure sur lequel j'ai pu travailler pendant ces 4,5 derniers mois. Je le remercie aussi pour son aide, ses conseils et sa disponibilité malgré la crise sanitaire de la COVID-19 qui nous a tous touché et forcé à rester confiné pendant plusieurs semaines.

0 Introduction et présentation générale

0.1 Motivations

Le XXIème siècle est marqué par un développement accéléré des nouvelles technologies d'information et de communication. Avec cet essor, les quantités de données à stocker et à analyser ont explosé depuis plusieurs années, en suivant un rythme effréné et continu. L'informatique en particulier nous entoure tous ou pratiquement aujourd'hui : en allant des jeunes consommant des contenus multimédias (sur Internet) et parcourant les réseaux sociaux, jusqu'aux grandes entreprises disposant d'immenses parcs de serveurs aux capacités de calcul et de stockage phénoménales.

Cette confrontation globale du monde à l'informatique fait qu'il est nécessaire de disposer de nouveaux outils, de plus en plus performants pour satisfaire une demande croissante. L'amélioration matérielle permet d'une part d'avoir des capacités de calcul et de stockage plus importantes, mais la programmation et le développement des systèmes, algorithmes, frameworks et logiciels sont une partie essentielle pour permettre le bon fonctionnement des machines et leurs utilisations.

L'informatique est évidemment objet de recherche. En anglais, on nomme cette science "Computer Science", et elle touche à tout ce qui concerne la théorie informatique, les algorithmes et leurs problèmes liés, ainsi que la conception physique des ordinateurs et la création des logiciels et des applications.

Étant donné cet usage généralisé de l'informatique, il est primordial d'en comprendre son fonctionnement, afin de développer des systèmes capables d'analyser massivement les milliards d'octets de données créées chaque jour à travers le monde ; telles que des données d'utilisateurs, des données scientifiques, de finance, de marketing, de forums, d'encyclopédies en ligne, d'ouvrages numérisés, etc.

Nous nous intéressons ici à l'analyse de données de type "texte", et souhaitons en particulier les traiter rapidement, efficacement et à moindre coût. Quel design algorithmique permettrait-il de s'adonner à cette tâche et comment l'utiliser pour l'analyse de graphes ?

0.2 Un bon design d'algorithme ?

Les "computer sciences" se sont propagées ces derniers temps : de plus en plus de monde recherche et crée des algorithmes capables de manipuler des jeux de données de plusieurs milliers, millions, voire milliards de lignes de texte.

Malgré un corpus scientifique fleurissant sur le sujet, seulement certaines méthodes de programmation sortent du lot. En effet, au vue de la taille des données créées aujourd'hui, il est primordial pour les sciences informatiques de s'adapter au changement et de proposer des algorithmes performants et extensibles. Il est présentement impensable de travailler et développer des méthodes incapables de s'adapter aux données du monde réel. Un algorithme ne pouvant performer que sur des échantillons

"test" ou des petits jeux de données risque d'être rangé dans une catégorie de systèmes inutilisables, voire même être qualifiés de "jouet" et être ensuite abandonné et oublié.

Un autre aspect essentiel de la recherche en programmation informatique concerne le temps et le coût des opérations et des manipulations que l'on effectue sur les données. Si, avec un certain design d'algorithme, il est impossible d'obtenir des résultats dans un temps raisonnable, ou si le calcul qui procède coûte excessivement cher, il est également possible que la conception en question soit laissée de côté pour une autre.

C'est l'analyse de la complexité des algorithmes qui permet d'évaluer le temps ou l'espace (et ainsi le coût) nécessaire à l'exécution des algorithmes que nous souhaitons développer. Pour connaître l'ordre de grandeur de leurs temps d'exécution, on évalue leurs complexités en

$$O(1) \qquad O(log(n)) \qquad ... \qquad O(n) \qquad O(n!) \qquad ...$$

Par exemple, un algorithme de complexité $O(n)$ est un algorithme qui aura un nombre d'opérations proportionnel à la taille n des données. Un algorithme dépassant la complexité quadratique, en $O(n^3)$ par exemple, ne sera pas extensible à des jeux de données massif. En effet, pour $n = 1000000$, il faudrait en théorie plusieurs centaines d'années pour obtenir des résultats sur une machine moderne.
Le fameux problème du voyageur de commerce quant à lui, qui est un problème de plus court chemin dans les graphes, atteint une complexité en $O(n!)$, le rendant non résoluble, même appliqué à un petit graphe !

0.3 Déroulement des recherches

Notre problème de recherche concerne le traitement de grandes quantités de textes, quels qu'ils soient. Imaginons que nous souhaitions compter des occurrences d'un certain mot ou d'une certaine lettre dans l'entièreté des pages "Wikipédia" en français, ou encore que nous voulions calculer les fréquences d'utilisation de deux mots successifs dans ce même corpus de textes. Alors, nous aurions affaire ici à plus de deux millions de pages de l'encyclopédie en ligne, contenant chacune une quantité plus ou moins importante de texte qu'il faudra analyser.

La première façon de procéder serait de créer un algorithme de comptage sur une machine unique, assez puissante pour effectuer les calculs rapidement. Mais sur ce genre d'opération sur les textes, nous risquons très vite d'avoir notre ordinateur surchargé, à la fois au niveau de la mémoire, mais aussi au niveau des processeurs.

Comment alors traiter ces données ?
Il est d'abord fondamental de comprendre ce que nous utilisons, le type d'objet que nous manipulons, leurs structures, afin de pouvoir les analyser efficacement. Reprenons par exemple le cas du comptage : est-il logique et surtout nécessaire de faire traverser la totalité de notre corpus "Wikipédia" à travers une seule machine ? Autrement-dit, est-il obligatoire et efficace de stocker l'ensemble du comptage sur un seul disque et de mémoriser toute la liste des mots, lettres ou paires de mots au

même endroit ?
La réponse à cette question est négative. En effet, une telle opération de comptage peut s'effectuer en parallèle sur de multiples machines, ce qui est plus rapide, plus efficace, et est même recommandé. Cela est possible car il n'y a en réalité aucun lien reliant les différentes pages "Wikipédia", ni même les différents paragraphes d'une page à conserver. Pour l'opération que nous souhaitons effectuer, seul le texte nous importe. Il est donc possible de découper le corpus et de traiter les différentes parties séparément sur plusieurs machines simultanément et récupérer les résultats en sortie.

C'est pourquoi notre travail gravitera autour de la découverte d'une méthode de programmation et d'analyse de données en parallèle nommée MapReduce. Nous présenterons en premier lieu le fonctionnement théorique de ce patron d'architecture de développement informatique avant de présenter nos recherches sur une manière d'utiliser MapReduce afin de trouver les composantes connexes dans un graphe quelconque.

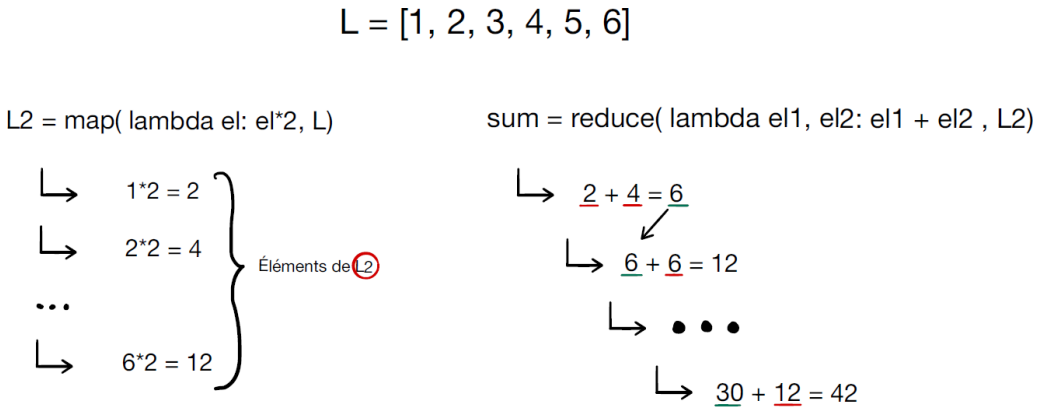
1 MapReduce : design, environnement et applications

MapReduce est un modèle de programmation qui a été popularisé par Google et qui est largement utilisé pour traiter de grandes quantités d'informations. L'idée est simple, et est par expérience la meilleure pour traiter d'immenses jeux de données : diviser pour régner ("Divide and Conquer" en Anglais). Pour cela, MapReduce fonctionne à l'intérieur d'un système de fichiers distribué, disposé sur un réseau de plusieurs machines que l'on appelle plus communément des "nœuds". L'architecture d'un travail MapReduce permet de traiter rapidement des données, qui sont découpées et séparées sur les différents nœuds d'un parc informatique que l'on nomme un "cluster". Nous allons étudier le fonctionnement du système de fichiers distribué ainsi que celui de MapReduce avant de présenter une de ses applications les plus connues sur des graphes.

1.1 Fonctionnement général de MapReduce et vocabulaire utile

Comme son nom l'indique, MapReduce est composé de 2 phases, respectivement la phase de "Map" et celle de "Reduce". Cela n'est pas sans rappeler les fonctions *map* et *reduce* en programmation fonctionnelle, qui possèdent d'ailleurs quelques similarités avec l'architecture que nous allons étudier.

En Python par exemple, $map(fct1(el), [liste])$ applique à chaque élément d'un objet itérable (ici une liste) une fonction, et $reduce(fct2(el1, el2), [liste])$ applique successivement une autre fonction à deux éléments de la liste, et récupère le résultat de la précédente opération pour procéder à la suivante. Un exemple simple codé en Python est présenté ci-dessous :



[Python] Application de 'map' puis 'reduce' pour calculer la somme des 6 premiers entiers multipliés par 2.

En MapReduce, la phase de Map est similaire à celle que l'on vient de décrire ci-dessus à un détail près. En effet, on applique bien à tous les éléments de notre jeu de données une fonction. En revanche, au lieu de renvoyer une simple valeur, la phase Map d'un travail MapReduce renvoie des paires d'éléments composées d'une clé et d'une valeur : $(< key >, < value >)$ qui seront essentielles pour le bon fonctionnement de la phase Reduce.

Effectivement, lors de la phase Reduce, on récupère l'ensemble des éléments possédant la même clé, et on applique une fonction sur la totalité des valeurs associées à cette clé. On a donc en entrée du reducer un objet de ce type : ($< key >, < iterable\ values >$).

```
1: class Mapper
2:     method Map(docid a, doc d)
3:         for all term t ∈ doc d do
4:             Emit(term t, count 1)

1: class Reducer
2:     method Reduce(term t, counts [c1, c2, ...])
3:         sum ← 0
4:         for all count c ∈ counts [c1, c2, ...] do
5:             sum ← sum + c
6:         Emit(term t, count sum)
```

Pseudo-code de 'Map' puis 'Reduce' pour effectuer un comptage de mots dans un texte (source : [6]).

C'est cette dualité clé-valeur qui est au cœur de l'architecture MapReduce et qui permet de faire des opérations en parallèle sur plusieurs nœuds. Un traitement en simultané fonctionne de la sorte : le jeu de données principal est distribué sur l'ensemble des nœuds du cluster, un certain nombre de mappers (ce nombre peut être conseillé par l'utilisateur) est alors créé pour traiter des parties du fichier distribué. Chaque mapper émet alors un certain nombre de paires ($< key >, < value >$). Les objets sortant de la phase Map vont ensuite être mélangés, rangés et regroupés par clés avant de rentrer dans la phase Reduce. Cette opération de mélange et de rangement, indispensable, est cependant invisible à l'œil de l'utilisateur et est complètement opérée par le framework. Elle permet de réunir toutes les paires ayant la même clé dans un même reducer pour que la deuxième phase d'un travail MapReduce fonctionne correctement.

L'architecture d'un travail MapReduce peut être légèrement plus complexe : on peut ajouter, afin de limiter l'impact de la phase de mélange ("shuffle"), des outils que l'on appelle les "combiners". Ils opèrent tels des petits reducers à l'intérieur de la phase Map. Ils permettent dans la plupart des cas d'agréger les valeurs ayant la même clé, avant le mélange des paires intermédiaires (celles vivant entre la phase de Map et celle de Reduce).

La complexité et la durée de la phase de mélange dépend directement du nombre de mappers, du nombre de reducers et de la quantité de paires émises par la phase Map. Réduire le nombre de paires en agrégeant nos objets avant le "shuffle" permet donc d'éviter au framework de mélanger et ordonner des quantités trop importantes de paires. Cela permet de gagner du temps en fluidifiant et réduisant le trafic entre les nœuds du cluster.

D'autres outils, les "partitioners" permettent de diriger les éléments vers les "bons" reducers. Ils ne se concentrent que sur les clés des paires intermédiaires pour déterminer

le meilleur reducer à utiliser pour terminer le travail MapReduce. Tous les éléments en sortie d'un partitioner se retrouvent dans un seul et même reducer.

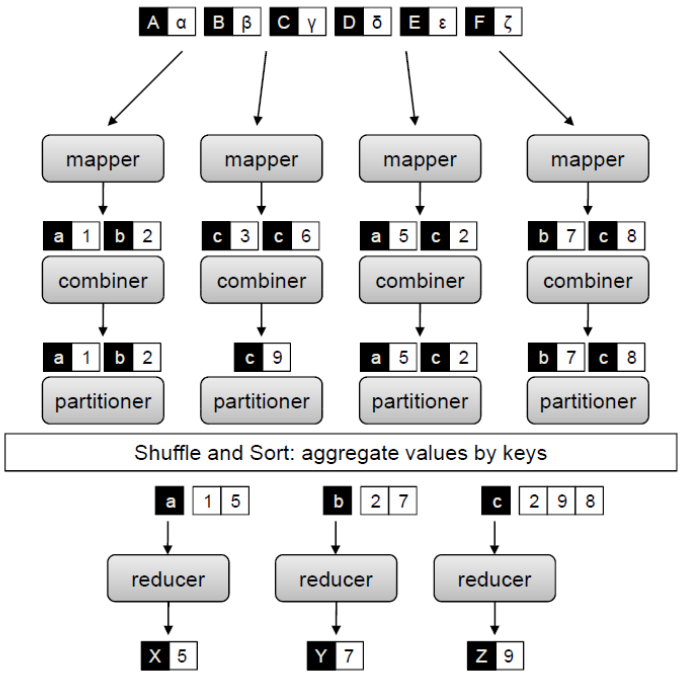


Schéma descriptif de l'architecture MapReduce, illustrant les "combiners" et les "partitioners" en plus des mappers et des reducers. Les "combiners" peuvent être vu comme des "mini-reducers", directement implantés à l'intérieur de la phase de Map. Les "partitioners" déterminent quel "reducer" est responsable de chaque clé (source : [6]).

Une fois le travail MapReduce terminé, l'utilisateur peut récupérer les résultats agrégés en sortie des reducers. L'opération s'arrête alors.

1.2 Le système de fichiers distribué

Nous avons précédemment observé le fonctionnement général de tout travail MapReduce. Concentrons nous maintenant sur l'environnement dans lequel les opérations des mappers, combiners, partioners et reducers sont effectuées : un système de fichiers distribué.

MapReduce est une architecture d'algorithme qui a été pensée pour réaliser des calculs en parallèle, sur de multiples machines simultanément. C'est le traitement de données extrêmement volumineuses, de l'ordre du petabyte (= 1000 terabytes), qui nous force à préférer calculer en parallèle dans un cluster plutôt que sur une seule grosse machine, en particulier aujourd'hui.

Tout d'abord, le rapport efficacité-prix de l'informatique est plutôt mauvais. En améliorant une simple machine, on améliore évidemment ses performances, mais plus on injecte d'argent dans une même machine, plus le gain de performance par euro (ou dollar) supplémentaire dépensé est faible. La courbe performance par rapport au prix est donc concave. Ainsi il reviendrait beaucoup trop cher de souhaiter faire de tels calculs sur une machine unique.

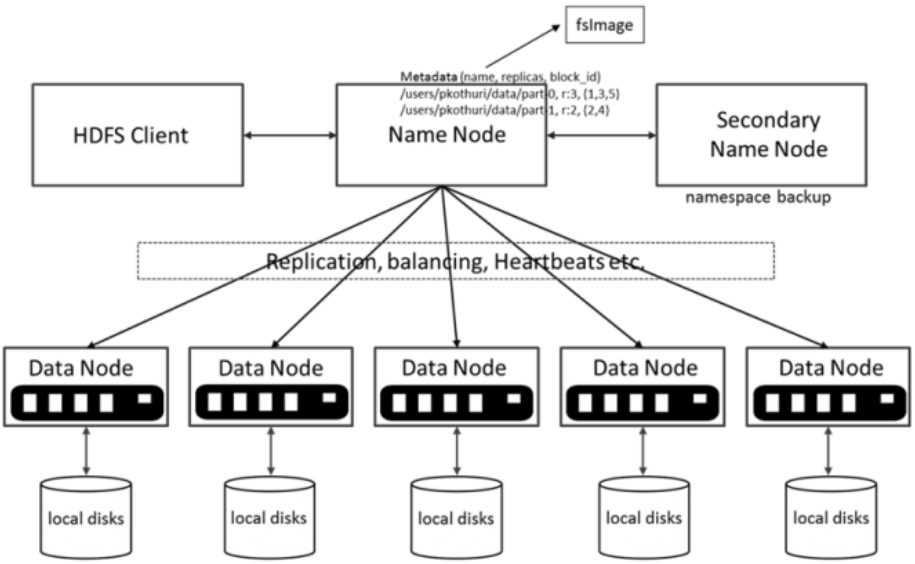
En réalité, il est préférable d'avoir plusieurs machines "faibles", qu'une seule grosse "forte". Aujourd'hui, des firmes sont spécialisées dans la création de clusters informatiques sur demande des utilisateurs. Ainsi, il n'est plus nécessaire de posséder un parc informatique privé pour faire ses calculs. Une entreprise, un institut de recherche, etc. peut donc simplement louer un certain nombre de machines virtuelles pour effectuer une ou plusieurs opérations de MapReduce.

C'est ce que l'on nomme aujourd'hui "computing in the Clouds".
Cette flexibilité permet à l'utilisateur lambda d'effectuer ses calculs "dans les nuages" et de ne payer que ce qu'il utilise. Si le traitement est gros, il suffit de demander plus de nœuds dans son cluster, et inversement si le traitement est plus petit. C'est un système extensible et efficient, tout comme MapReduce possède une architecture extensible et efficiente.

Le framework open source utilisé pour effectuer des tâches MapReduce sur un cluster est généralement Hadoop. Son système de fichiers distribué est appelé HDFS : "Hadoop Distributed File System". Hadoop peut fonctionner sur des clusters de plusieurs dizaines de nœuds ou plus encore. Ce système de fichiers distribué est remarquable et extrêmement fiable.

HDFS possède une architecture dite de "maître-esclave". Dans le cas d'Hadoop, le maître est nommé le "namenode" et les esclaves les "datanodes". Dans un cluster Hadoop, c'est le "namenode" qui ordonnera le stockage de données et les opérations d'écriture, suppression, réplication, etc. Cette première machine se charge de conserver l'emplacement des différents blocs de données répartis sur l'ensemble des autres machines, les "datanodes". Elle conserve donc le registre d'adresse de tous les blocs dénommé le "namespace".

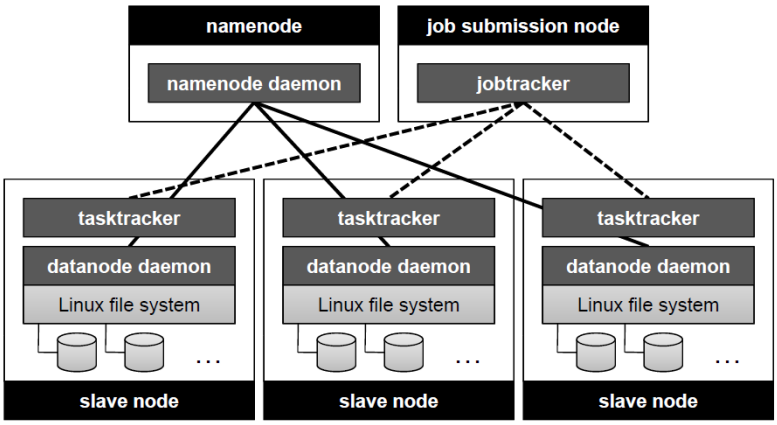
Le fonctionnement est le suivant : lorsqu'un utilisateur souhaite stocker des données sur un cluster Hadoop, il dépose d'abord ses données dans un client HDFS (une application directement reliée au "namenode"). Les données sont ensuite découpées en blocs, par défaut de taille 128 MB, et chacun de ces blocs est ensuite enregistré trois fois, sur trois "datanodes" différents du cluster (par défaut). Cela permet de sécuriser l'enregistrement des données.
En effet, si une machine du cluster venait à s'interrompre, les données originales seraient encore stockées à deux endroits différents. Mieux encore, le "namenode" s'efforcera toujours de conserver trois copies de chaque blocs. Donc si une telle défaillance venait à arriver, automatiquement, le framework d'Hadoop rétablirait l'équilibre en recréant une copie à partir des deux restantes pour en retrouver trois exactement sur le cluster. C'est ce qui garantit un stockage rapide et fiable.



Le fonctionnement d'un cluster Hadoop (source :genericclass.com)

Si l'utilisateur souhaite maintenant réaliser un travail MapReduce, nous observons que le terrain est déjà préparé. Effectivement, notre jeu de données initial est déjà séparé sur plusieurs nœuds du cluster. Il nous suffit d'ordonner aux nœuds d'opérer les opérations Map puis Reduce. Pour cela, un nouveau nœud entre en jeu dans le cluster : le "job submission node" ou seulement "jobtracker". Comme son nom l'indique, c'est lui qui va instancier les objets de classe MAP et REDUCE dans les nœuds et ordonner leurs exécutions. Le jobtracker se charge également de veiller au bon fonctionnement de l'algorithme utilisateur. Il communique avec les "datanodes", pour connaître leurs disponibilités (s'ils sont occupés ou non) et récupère des informations d'avancement du traitement si nécessaire.

Le nombre de reducers instanciés par le jobtracker sur le cluster dépend directement du programmeur, tandis que le nombre de mappers dépend à la fois du programmeur, mais aussi de la disposition des blocs sur le cluster et le découpage en entrée des données (l'"InputSplit" avant la première phase de Map).



Cluster Hadoop muni du "job submission node", permettant entre autres d'instancier un travail MapReduce (source : [6]).

1.3 Design et Applications de MapReduce sur les graphes

De nombreux designs d'algorithme en MapReduce ont été développés et sont largement utilisés aujourd'hui. Les tâches d'agrégation locale (comme dans l'exemple de la somme en partie 1.1), de comptage, de calcul de moyennes, de calcul de matrice de co-occurrence, etc. sont devenues des tâches basiques à réaliser en parallèle avec MapReduce.

L'utilisation de MapReduce est très répandue : en 2004, Google (muni de son système propriétaire de MapReduce) analysait environ 100 TB de données par jour. 4 ans plus tard, en 2008, ce n'est pas moins de 20 TB de données qui sont analysées chaque jour par Google. Le big data s'est emparé de notre société, et les firmes transnationales sont forcées de s'adapter et d'utiliser ces méthodes de traitement en parallèle pour rester dans la course. Les applications de MapReduce sont aussi très utile dans les sciences, particulièrement dans la recherche. Par exemple, l'algorithme de calcul de matrice de co-occurrence peut être utilisé en linguistique pour repérer les fréquences d'utilisation de 2 mots côte à côte ; il pourrait en théorie même être utilisé pour un travail sur une intelligence artificielle, etc.

```
1: class Mapper
2:     method Map(docid a, doc d)
3:         for all term w ∈ doc d do
4:             for all term u ∈ Neighbors(w) do
5:                 Emit(pair (w, u), count 1)        ▷ Emit count for each co-occurrence

1: class Reducer
2:     method Reduce(pair p, counts [c1, c2, ...])
3:         s ← 0
4:         for all count c ∈ counts [c1, c2, ...] do
5:             s ← s + c                              ▷ Sum co-occurrence counts
6:         Emit(pair p, count s)
```

Un algorithme MapReduce de calcul de co-occurrence de mots dans un large corpus de textes. Le mapper émet les couples de mots successifs sous forme de tuple comme "clé" et 1 pour "valeur". Le reducer somme toutes les co-occurrences pour chaque clé (source : [6]).

Nous rejoignons désormais notre problème initial : l'analyse de graphes.

Nous rappelons d'abord qu'un graphe est une structure composée de sommets, reliés par des arêtes (liaison non orientée, un simple trait) ou des arcs (liaison orientée, une flèche). Ils sont très utiles pour représenter des réseaux (par exemple des réseaux routiers) et suscitent des recherches, comme par exemple avec le problème du voyageur de commerce qui n'est toujours pas résolu, et que nous ne développerons pas ici.

Travailler avec MapReduce sur les graphes diffère d'un travail simple de comptage sur un corpus de textes. En effet, l'architecture même de MapReduce implique une séparation de notre jeu de données initial, et une répartition de ces données sur différents nœuds d'un cluster. Il devient donc impossible d'obtenir une vision globale d'un graphe lors d'un travail MapReduce car les informations nécessaire à la reconstruction complète du graphe ne sont pas situées au même endroit et les nœuds d'un cluster, pour rappel, ne communiquent pas entre eux lors des phases de Map

ou de Reduce.

La programmation d'un traitement sur les graphes se doit en réalité d'être un algorithme itératif ou récursif, composé de plusieurs tâches MapReduce. C'est l'utilisateur qui doit déterminer au bout de combien d'itération stopper ou qui doit définir une condition qui fera s'arrêter l'algorithme. Pour cela, les phases de reduce envoient généralement des données qu'elles font remonter au "jobtracker" pour que la décision de continuer (refaire une boucle) ou non soit prise.

Habituellement, une boucle équivaut à une tâche MapReduce dans un traitement dans les graphes, mais il n'est pas rare d'en trouver deux dans certains cas.

Étudions par exemple l'algorithme de Dijkstra, qui est un algorithme de plus court chemin dans les graphes. Cet algorithme détermine, à partir d'un sommet bien précis, la valeur du plus court chemin vers tous les autres sommets du graphe. Les valeurs des arcs ou arêtes doivent être toutes positives pour pouvoir appliquer cet algorithme.

Comparons les pseudos-code de l'algorithme "classique" et sa version programmée en MapReduce (les deux pseudos-codes sont tirés du document [6])

```
1:  DIJKSTRA(G, w, s)
2:      d[s] ← 0
3:      for all vertex v ∈ V do
4:          d[v] ← ∞
5:      Q ← {V}
6:      while Q ≠ ∅ do
7:          u ← EXTRACTMIN(Q)
8:          for all vertex v ∈ u.ADJACENCYLIST do
9:              if d[v] > d[u] + w(u, v) then
10:                 d[v] ← d[u] + w(u, v)
```

Pseudo-code de l'algorithme de Dijkstra (version "classique"). (source : [6]).

Dans sa version "classique", à chaque itération de la boucle WHILE, on choisi (et on retire) un sommet de valeur minimale u encore présent dans "Q", et on met à jour les valeurs de ses suivants à la seule condition que le chemin passant par u soit effectivement plus court.
On recommence jusqu'à ce que la liste "Q" soit vide et on obtient le résultat correct après un nombre d'itération exactement égal au nombre de sommets présents dans le graphe.

```
1: class Mapper
2:     method Map(nid n, node N)
3:         d ← N.Distance
4:         Emit(nid n, N)                          ▷ Pass along graph structure
5:         for all nodeid m ∈ N.AdjacencyList do
6:             Emit(nid m, d + 1)                  ▷ Emit distances to reachable nodes

1: class Reducer
2:     method Reduce(nid m, [d1, d2, ...])
3:         d_min ← ∞
4:         M ← ∅
5:         for all d ∈ counts [d1, d2, ...] do
6:             if IsNode(d) then
7:                 M ← d                           ▷ Recover graph structure
8:             else if d < d_min then              ▷ Look for shorter distance
9:                 d_min ← d
10:        M.Distance ← d_min                      ▷ Update shortest distance
11:        Emit(nid m, node M)
```

Pseudo-code de l'algorithme de Dijkstra (version "MapReduce"). (source : [6]).

Pour l'algorithme MapReduce, il est impossible de sélectionner le sommet de valeur minimale puisqu'il n'y a pas de vision globale du graphe. C'est pourquoi on applique une méthode nommée en anglais la "parallel breath-first search".
Pour cette tâche, les clés correspondent aux identifiants des sommets, et les valeurs contiennent à la fois la liste d'adjacence (des suivants) du sommet, ainsi que la valeur du plus court chemin (initialisé par défaut à $+\infty$ sauf pour le sommet de départ qui est initialisé à 0). Dans l'algorithme ci-dessus, on suppose que toutes les valeurs disposées sur les arcs sont égales à 1.

La phase de map passe sur chacune des paires $(< id >, < adjc + valeur >)$ et renvoie 2 choses. D'abord elle renvoie exactement les données d'entrée, non modifiées, pour conserver la structure du graphe (l.4 mapper) ; et renvoie une autre paire $(< id >, d+1)$ où d correspond à la valeur associée au sommet en entrée de la phase Map.

Dans la phase de Reduce, on récupère toutes les paires possédant la même clé $< id >$ et on va itérer sur les valeurs associées. Avec la phase de Map, on sait qu'il existe 2 types de $valeurs$: des simples entiers, correspondant à l'émission de la seconde paire de la phase Map (l.6 mapper), et des données type $< adjc + valeur >$ qui ont été émises lors de la première émission de paire (l.4 mapper) pour conserver la structure du graphe (on passe les sommets du mapper au reducer).
Le reducer va alors trier parmi les valeurs pour retrouver les données $< adjc+valeur >$, qui permettront de faire passer la structure de graphe à l'itération suivante (l.6-7 reducer). De plus, le reducer va chercher la valeur du chemin le plus court permettant d'arriver jusqu'au sommet sur lequel il est en train de travailler (l.8-9 reducer).
Par ce procédé, on découvre le graphe un saut par un saut et on modifie les valeurs des sommets au niveau de la frontière de découverte. On s'arrête généralement lorsque les valeurs sont toutes strictement inférieures à $+\infty$.
Pour que l'algorithme fonctionne comme celui présenté avant, il faudrait remplacer le +1 dans le mapper, à la ligne 6, par la valeur de l'arc (n, m).

Nous avons donc étudier un algorithme de plus court chemin sur les graphes et avons

découvert la structure générale d'un algorithme MapReduce sur des graphes.
Nous allons désormais pouvoir rechercher un algorithme de détection des composantes connexes, l'implémenter et le tester dans la partie suivante.

2 Une approche pratique de MapReduce

Nos recherches théoriques sur MapReduce nous emportent désormais vers un problème classique d'étude de graphe : la détection de composantes connexes.

En théorie des graphes, on dit que deux sommets x et y appartiennent à une même composante connexe s'il existe une chaîne (suite finie d'arêtes consécutives) reliant x et y. Dans une recherche de composantes connexes, on ne se préoccupe pas de l'orientation des arcs. Seul l'existence de la chaîne importe.

Notre objectif ici est de découvrir et comprendre comment implémenter un tel algorithme de détection, fonctionnel, en utilisant MapReduce.

2.1 Un algorithme de détection des composantes connexes dans un graphe

Nous désirons trouver un moyen de détecter les composantes connexes d'un graphe, à partir d'un fichier texte contenant l'ensemble des informations sur ce graphe.

Notre première approche a été de comprendre à quoi ressembleront les données que nous utiliserons pour appliquer notre algorithme de MapReduce, et si besoin trouver un moyen de les transformer pour les rendre compatibles avec les traitements. On suppose que les informations de notre graphe sont stockées dans un fichier txt, ou csv par exemple. Ce qui nous intéresse ici, c'est la liste des arcs et arêtes du graphe qui permettent en effet de connaître les liaisons entre des sommets voisins.

Si par exemple nous avons une arête (a, b) et une autre (b, c), alors on peut directement conclure que les sommets a, b, et c sont contenus dans une même composante connexe.

C'est sur cet aspect de transitivité de la relation "appartenir à une même composante connexe" que nous allons travailler. Chaque arête définie en réalité une paire d'éléments appartenant à une même composante. Il faut ensuite réussir à connecter, par transitivité de la relation, tous les autres éléments appartenant à cette même composante. Afin de réaliser cette opération, on décide de définir chaque composante trouvée à l'aide d'un identifiant unique. Le plus intuitif est de nommer la composante d'après le membre le plus petit (ou le plus faible) :

Dans l'exemple ci-dessus, on l'appellerait la composante "a" et on dirait ainsi que b et c appartiennent à la composante "a".

Nous avons défini notre objectif : relier tous les éléments d'une composante connexe à son sommet d'identifiant le plus faible. En reliant chaque sommet à une composante, on découvrira ainsi l'ensemble des groupes et on pourra conclure quant à la répartition de nos sommets à travers le graphe.

Rappelons que nous souhaitons effectuer ces regroupements en utilisant MapRe-

duce. Il faut donc définir la structure des paires $< cle, valeur >$. Le plus simple reste le plus efficace : on choisit donc d'avoir comme clé le sommet de départ d'un arc, et comme valeur le sommet d'arrivée d'un arc. On suppose d'ailleurs à partir de maintenant que notre fichier d'informations sur le graphe contient uniquement la liste des arcs, exactement sous cette forme de $< sommetdepart, sommetarrivee >$. Nous sommes prêts à écrire notre algorithme itératif (cf. partie 1.3) de détection des composantes connexes.

Afin de poursuivre sur le sujet, nous avons étudié un design d'algorithme développé par Hakan Kardes, Siddharth Agrawal, Xin Wang, et Ang Sun [4].

La méthodologie appliquée pour gérer la détection des composantes est exactement celle expliquée précédemment : on essaye de retrouver le sommet ayant l'identifiant $< id >$ le plus petit d'une composante, et on connecte tous les autres sommets de cette composante à ce sommet.

Dans leur algorithme itératif, on retrouve non pas une, mais deux tâches de MapReduce : $CCF - Iterate$ et $CCF - Debup$.

$CCF - Iterate$ va, à partir de l'ensemble des arcs du graphe, construire la liste d'adjacence de tous les sommets. Pour chacun d'eux, il va ensuite déterminer à quel élément de sa liste d'adjacence il serait possible de le relier pour le connecter à un groupe d'identifiant minimal (il le connectera forcément à un sommet d'identifiant plus faible que lui si possible).

$CCF - Debup$ quant à lui se charge uniquement de supprimer les doublons obtenus à la fin de la première tâche de MapReduce.

Concentrons nous sur le fonctionnement de $CCF - Iterate$ (pseudo-code disponible plus bas) en utilisant un exemple simple. Prenons un graphe de 4 sommets et possédant 3 arcs : (2,1) ; (3,2) ; (3,4).
Le graphe ne possède qu'une composante, contenant les 4 sommets.

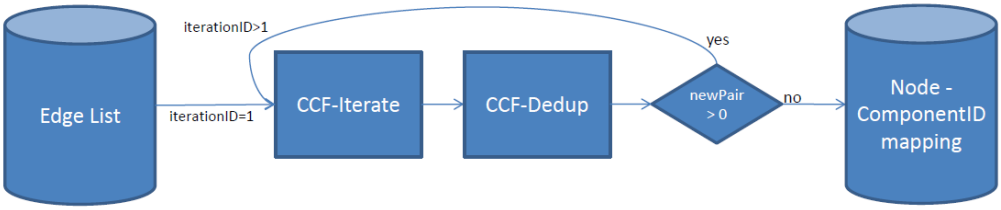
La phase de Map de $CCF - Iterate$ va simplement dédoubler les arcs. Cela est nécessaire car un arc est un objet orienté, or pour la recherche de composantes connexes, l'orientation ne nous importe pas. On transforme donc nos arcs en "doubles arcs" :

Sortie de map : (2,1) ; (1,2) ; (3,2) ; (2,3) ; (3,4) ; (4,3)
Regroupement par clés : (1,[2]) ; (2,[1,3]) ; (3,[2,4]) ; (4,[3])

Le reducer va déterminer pour chaque clé (1, 2, 3 et 4) à quelle valeur relier les sommets. Si la clé est inférieure à tous les éléments de la liste d'adjacence, alors le reducer ne renvoie rien, car cela signifie que tous les éléments de la liste d'adjacence sont déjà réunis avec une clé "minimale".
Par exemple, en récupérant (1,[2]), le reducer remarque que $1 < 2$ donc ne renvoie rien : 2 appartient déjà à la composante 1.

Pour (2,[1,3]) en revanche, le reducer va repérer qu'il existe un élément de la liste d'adjacence plus petit que la clé. Ici, cet élément est 1 car 1 < 2. L'algorithme va alors décider de relier tous les sommets à la composante identifiée par 1. C'est-à-dire qu'il va émettre les paires (2,1) et (3,1). En plus de ces paires, le reducer va incrémenter une variable globale : "newPair" d'une unité pour chaque nouvelle paire émise entre 2 éléments de la liste d'adjacence. Cela permet d'informer le jobtracker qu'il est nécessaire de procéder à une nouvelle itération de l'algorithme complet, car on a réussi à amener un nouveau sommet *new* dans une composante et qu'il existe donc éventuellement d'autres sommets, reliés à *new* et susceptibles d'appartenir à la même composante.

En effet, lorsque la totalité des composantes connexes a été découverte, plus aucun lien n'est effectué entre deux éléments d'une même liste d'adjacence lors de la phase reduce, et on récupère uniquement les sommets reliés à l'identifiant le plus faible de leurs composantes connexes.



L'ensemble du fonctionnement du CCF (source : [4]).

Finissons l'exemple :
Reduce((3,[2,4])) donne les paires (3,2) et (4,2) et incrémente newPair de 1
Reduce((4,[3])) donne la paire (4,3)

Au bout de la première itération, on a donc les paires : (2,1) ; (3,1) ; (3,2) ; (4,2) ; (4,3).
Et newPair = 2

Deuxième itération :
Map et regroupement : (1,[2,3]), (2,[1,3,4]), (3,[1,2,4]), (4,[2,3])
Reduce((1,[2,3])) : ø
Reduce((2,[1,3,4])) : (2,1) ; (3,1) ; (4,1) ; newPair += 2
Reduce((3,[1,2,4])) : (3,1) ; (2,1) ; (4,1) ; newPair += 2
Reduce((4,[2,3])) : (4,2) ; (3,2) ; newPair += 1

On élimine les doublons avec $CCF - Debup$, il nous reste :
(2,1) ; (3,1) ; (4,1) ; (4,2) ; (3,2) ; newPair = 5

Troisième itération :
Map et regroupement : (1,[2,3,4]), (2,[1,4,3]), (3,[1,2]), (4,[1,2])
Reduce((1,[2,3,4])) : ø
Reduce((2,[1,4,3])) : (2,1) ; (4,1) ; (3,1) ; newPair += 2
Reduce((3,[1,2])) : (3,1) ; (2,1) ; newPair += 1

Reduce((4,[1,2])) : (4,1) ; (2,1) ; newPair += 1

On élimine les doublons avec $CCF-Debup$, il nous reste :
(2,1) ; (3,1) ; (4,1) ; newPair = 4

Quatrième et dernière itération :
Map et regroupement : (1,[2,3,4]), (2,[1]), (3,[1]), (4,[1])
Reduce((1,[2,3,4])) : ø
Reduce((2,[1])) : (2,1)
Reduce((3,[1])) : (3,1)
Reduce((4,[1])) : (4,1)

Il nous reste :
(2,1) ; (3,1) ; (4,1) ; newPair = 0

On sort de la boucle, l'algorithme est terminé, tous les sommets sont reliés à leurs composantes connexes. Ici elle est unique et porte l'identifiant 1.

```
CCF-Iterate
map(key, value)
   emit(key, value)
   emit(value, key)

reduce(key, < iterable > values)
   min ← key
   for each (value ∈ values)
      if(value < min)
         min ← value
      valueList.add(value)
   if(min < key)
      emit(key, min)
      for each (value ∈ valueList)
         if(min ≠ value)
            Counter.NewPair.increment(1)
            emit(value, min)
```

Pseudo-code de l'algorithme MapReduce CCF-Iterate (source : [4]).

2.2 Programmation en PySpark et résultats sur un graphe de plusieurs millions d'arêtes

Cette dernière partie est consacrée à l'implémentation de ce que nous avons découvert jusque là.

Nous utiliserons ici un graphe venant des bases de données publiques de l'université de Stanford, trouvé sur le site "Stanford Large Network Dataset Collection" ([5] et [7])

Voici les premières lignes du fichier récupéré :

```
# Directed graph (each unordered pair of nodes is saved once)
# Stanford web graph from 2002
# Nodes: 281903 Edges: 2312497
```

```
# FromNodeId      ToNodeId
1         6548
1         15409
6548      57031
15409     13102
2         17794
```

L'entête du fichier .txt nous apporte les informations nécessaires à la compréhension de la structure des données, et des indications supplémentaires, telles que le nombre de sommets et le nombres d'arcs.

Ce graphe représente l'ensemble de la structure des liens hypertextes des pages du site de l'université de Stanford existant en 2002.

Nous l'avons choisi car il possède de nombreuses composantes connexes, il est donc intéressant à manipuler avec notre algorithme MapReduce. En effet, le taux de sommets présents à l'intérieur de la plus grosse composante connexe est de 0,906. Ce nombre paraît plutôt élevé mais une majorité des graphes du site de Stanford ont un tel taux égal à 1, donc n'ont qu'une seule composante connexe.

Afin d'effectuer notre traitement de données avec MapReduce sur plusieurs machines en parallèle, nous avons travaillé avec le framework Apache Spark et une application en ligne nommée Databricks permettant de créer des clusters Spark tout en travaillant sur des notebooks Python (ici PySpark car c'est une version de Python adaptée à l'utilisation de Spark).

Spark est un framework récent et très utilisé dans les travaux de type MapReduce. Il se veut plus performant que le framework d'Hadoop, car écrit les données intermédiaires (paires émises par les mappers) en mémoire (RAM) et non sur disque. Par rapport à Hadoop, Spark permettrait en théorie d'accélérer les travaux MapReduce d'un facteur 10 voire 100.

Spark donne entre autres aux utilisateurs des opérations supplémentaires à effectuer de manière distribuée et optimisée. Reprenons l'exemple du comptage de mots que nous avions mentionné en introduction. Le but est pour le mapper d'émettre pour chaque mot la paire ($< mot >$,1), et pour le reducer de sommer les occurrences d'un même mot.

En Spark, cet exemple peut se faire en une seule commande :

```
from pyspark import SparkContext
sc = SparkContext()

lines = sc.textFile("text.txt")

w_cts = lines.flatMap(lambda line: line.split(' ')) \
             .map(lambda word: (word, 1)) \
             .reduceByKey(lambda count1, count2: count1 + count2) \
```

```
            .collect()

for (word, count) in w_cts:
    print(word, count)
```

On commence par créer notre SparkContext qui contient les informations nécessaires à la connexion de notre environnement Spark à un cluster et à la création du système de fichiers distribué (RDD :resilient distributed dataset).
On importe ensuite notre fichier texte par sc.textFile(), qui s'occupe automatiquement de séparer les données sur le RDD. Les méthodes distribuées (ici flatMap, map et reduceByKey) sont appliquées en une seule ligne de code, et nous récupérons les résultats avec collect().

Pour l'algorithme CCF étudié dans la partie précédente, on ne peut pas procéder aussi simplement. En effet, le reducer de la tâche $CCF - Iterate$ est beaucoup plus complexe qu'une simple fonction lambda. De plus, nous devons renvoyer un compteur newPair à l'ordinateur qui contrôle le choix de refaire ou non des itérations. Il nous faut donc une variable globale à travers l'ensemble du cluster qui informe sur la nécessité de refaire une boucle ou non.

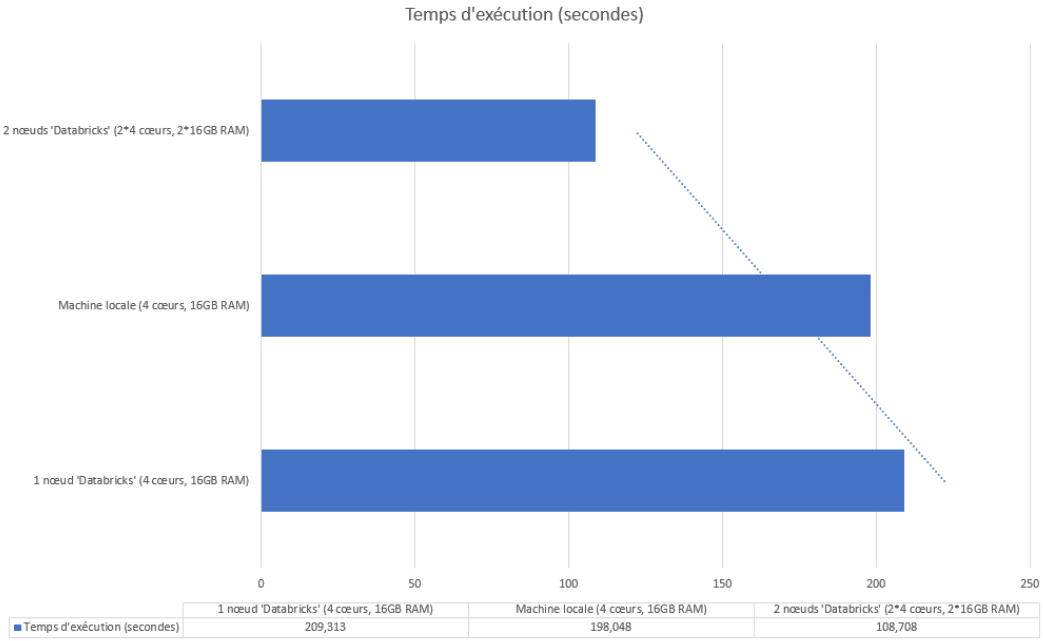
En Spark, ce genre de "compteur" se nomme un accumulateur. On en définira donc un.

Notre implémentation en PySpark est disponible dans les annexes : 4. Elle est parfaitement fonctionnelle, et a été testée sur des petits graphes dessinés à la main pour vérifier l'exactitude des résultats produits en sortie.

En l'appliquant sur le graphe de Stanford, nous retrouvons 365 composantes connexes différentes. La plus grosse, portant l'identifiant 1, contient 255265 sommets, c'est-à-dire environ 90,6% des sommets du graphe. Ces résultats sont en adéquation avec les informations trouvées sur le site de l'université de Stanford, ce qui nous permet une nouvelle fois de confirmer le bon fonctionnement de l'implémentation en PySpark.

Pour terminer cette étude, nous avons voulu effectuer une comparaison expérimentale du fonctionnement de l'algorithme sur des clusters de tailles différentes. Malheureusement, la version d'essai (gratuite) de Databricks, ne permet pas de créer de clusters ayant plus de 2 nœuds. Databricks est normalement un logiciel payant, où l'utilisateur loue le nombre de machines qu'il souhaite utiliser.

Nous avons donc fait tourner l'algorithme sur une machine locale, composé de 4 processeurs virtuels et 16GB de mémoire, puis sur une machine unique de Databricks elle aussi composée de 4 processeurs virtuels et 16GB de mémoire avant de terminer sur un cluster Databricks de 2 machines composées chacune de 4 processeurs virtuels et 16GB de mémoire.
Les résultats sont présentés dans le graphique ci-dessous :



Résultats expérimentaux de nos différentes implémentation de MapReduce sur le graphe de Stanford.

On y remarque qu'effectivement, les temps d'exécution de MapReduce par Spark sur une machine unique sont d'environ 200 secondes, tandis que le temps pour le cluster de deux machines est d'environ 110 secondes.
L'exécution sur le cluster est donc 45% plus rapide que sur les machines uniques.
Le fichier HTML du notebook avec les résultats sur le cluster de 2 nœuds est disponible dans les annexes : 4

Malgré les résultats prometteurs, il serait précipité de généraliser ces résultats. Il nous faudrait observer la progression de ces temps d'exécution pour à la fois des clusters de plus grande taille (payant), mais aussi des jeux de données bien plus importants.
comparé aux opérations qui peuvent être effectuées sur ce genre d'architecture en MapReduce, 2 312 497 arcs à analyser reste une tâche minuscule.

3 Bilan et conclusion

Le traitement massif d'informations est un problème que nous devons affronter, mais les outils développés sont de plus en plus performants et permettent de s'affranchir des limites du passé.

Nos recherches sur MapReduce nous on beaucoup appris sur le fonctionnement algorithmique de tels traitements et les résultats que nous avons réussi à obtenir dans la seconde partie sont très encourageants.

En revanche et afin de découvrir l'intégralité du potentiel du calcul en parallèle grâce à Hadoop ou Spark, il faudrait entrer dans les détails de la conception même du framework, c'est-à-dire étudier plus précisément comment sont programmés les mécanismes invisibles à l'œil de l'utilisateur.

De plus, comme nous le mentionnions en fin de partie 2, pour obtenir des résultats expérimentaux plus précis et plus à même de correspondre à une situation moderne de traitement avec MapReduce, il aurait fallu pouvoir effectuer les opérations sur de plus grands clusters et sur de plus grands graphes. C'est par manque de moyens que nous avons du se limiter à une implémentation sur des clusters composés de peu de nœuds.

A l'échelle de notre test, il ne serait pas effarant d'avoir des résultats très satisfaisant même en implémentant notre algorithme MapReduce sans Hadoop ni Spark, simplement sur une machine locale en Python par exemple. Nos données ne dépassent en effet pas la limite de mémoire (16GB) de notre machine, le traitement pourrait donc en théorie s'effectuer très rapidement si l'on prend soin de garder la liste des arcs en mémoire pendant les différentes itérations de l'algorithme de détection des composantes connexes.

Finalement, nous n'avons ainsi pas déterminé nous même les gains réels de performance du traitement en parallèle sur notre problème des composantes connexes, mais nous savons qu'ils sont déjà prouvés par l'expérience, et c'est pourquoi MapReduce et le traitement sur cluster sont utilisés aujourd'hui dès lors que les jeux de données dépassent l'ordre du TB ou du PB.

4 Annexes

Partie 2.2 : le code PySpark utilisé

```
iterationID = 0
accum = sc.accumulator(0)
newPair = True


def itRedCCF(pair):
  key, values = pair
  min = key
  valueL = []
  for value in values:
    if value < min:
       min = value
    valueL.append(value)
  if min < key:
    yield((key, min))
    for value in valueL:
      if min != value:
        accum.add(1)
        yield((value, min))


#timer on
begin = time()


while newPair == True:
  iterationID += 1
  newPair = False
  accum.value = 0

  #CCF-iterate (MAP)
  firstJobMap = edges.flatMap(lambda e: (e, e[::-1]))

  #CCF-iterate (REDUCE)
  firstJobReduce = firstJobMap.groupByKey().flatMap(lambda pair: itRedCCF(pair)).sortByKey()

  #CFF-debup (MAP & REDUCE)
  debupEdges = firstJobReduce.distinct()

  edges = debupEdges
  newPair = bool(accum.value)
  print("Iteration: ", iterationID, ", newPair? ", newPair)
```

Code PySpark utilisé sur le graphe de Stanford.
On remarque la présence de l'algorithme itératif comprenant les 2 tâches de MapReduce : CCF – Iterate et CCF – Debup.
Cette partie du code PySpark contient pas la définition de l'environnement SparkContext (se situant avant), ni l'affichage des résultats (se situant après).

Le notebook Databricks (à télécharger sous forme de fichier HTML) :

NOTEBOOK (lien Google Drive)

Bibliographie & Sitographie

[1] Apache. *Documentation Apache Spark*. 2020. URL : https://spark.apache.org/docs/latest/.

[2] Databricks. *Documentation Databricks*. 2020. URL : https://docs.databricks.com/.

[3] Ashish Goel et Kamesh Munagala. *Complexity Measures for Map-Reduce, and Comparison to Parallel Computing*. 2012. arXiv : 1211.6526 [cs.DC].

[4] Hakan Kardes et al. "CCF: Fast and scalable connected component computation in MapReduce". In : fév. 2014, p. 994-998. ISBN : 978-1-4799-2358-8. DOI : 10.1109/ICCNC.2014.6785473.

[5] Jure Leskovec. *Stanford Large Network Dataset Collection*. 2020. URL : https://snap.stanford.edu/data/.

[6] Jimmy Lin et Chris Dyer. "Data-intensive text processing with MapReduce". In : *Synthesis Lectures on Human Language Technologies* 3.1 (2010), p. 1-177.

[7] Stanford University. *Stanford web graph*. 2002. URL : https://snap.stanford.edu/data/web-Stanford.html.