Alignement d’entités sémantiques basé sur Map-Reduce

Mohamed Mahdi Benaïssa

Université Claude Bernard Lyon1- Master 2 Recherche IADE- Juin 2014
Mohamed.benaissa@etu.univ-lyon1.fr

Résumé. La quantité de données disponibles et partageables ne cesse de croître depuis l’émergence du Web et de la réduction des coûts de stockage. Aussi les sources de données se multiplient et les informations qu’elles contiennent peuvent être redondantes, complémentaires ou incohérentes. Afin de résoudre ces problèmes, l’alignement d’entités s’intéresse à la détection d’entités équivalentes issues de différentes sources de données. Cette détection d’équivalence constitue un traitement souvent lourd et couteux lié au très grand nombre de comparaisons effectuées. Afin de réduire les coûts de traitement, des approches ont été proposées et passent généralement par la création de blocs d’entités pour limiter les comparaisons entre les entités affectées à un même bloc. Bien qu’étudié depuis plusieurs décennies, l’alignement d’entités fait récemment l’objet d’une attention particulière face aux défis que représentent les volumes de données générés par certaines applications (ex., bioinformatique, astronomie, e-commerce). D’un autre côté, le traitement de tâches en parallèle, popularisé par le paradigme Map-Reduce, représente une solution intéressante et pertinente pour l’alignement d’entités, notamment de par sa phase de partitionnement et son découpage en blocs. Cependant, ce parallélisme, bénéfique en termes de performances, ne doit pas dégrader la qualité de l’alignement. Dans cet article, nous proposons l’approche MREER (Map Reduce Et Entity Resolution), qui permet l’alignement d’entités réparties sur différentes sources de données en exploitant le principe de Map-Reduce. L’originalité de notre approche repose sur le fait qu’au-delà des aspects liés au temps de traitement et à la répartition de charge des calculs, la qualité de l’alignement en termes d’équivalences trouvées reste au cœur du processus. Des expérimentations ont été menées pour valider notre approche et la comparer à une solution existante sur une grappe de 4 machines et portant sur des collections de publications scientifiques. Les résultats des expérimentations ont montré que notre approche MREER permet d’obtenir une meilleure qualité d’alignement, et permet de réduire le nombre de comparaisons inutiles de plus de 75% en moyenne, par rapport à la méthode existante.

Mots-clés: alignement d’entités, Map-Reduce, détection de doublons, passage à l’échelle

Abstract. Nowadays, the amount of data available and shareable is growing more and more due to the emergence of Web and the reduction of storage costs. In addition to this, data sources multiply. Thus, the information they contain

may be redundant, complementary or inconsistent. To solve these problems, the entity resolution allows the detection of equivalent entities derived from multiple data sources. This equivalence detection is often an expensive and heavy process due to the large number of comparisons made. In order to reduce processing costs, many blocking-based approaches have been proposed. Nowadays, entity resolution is applied in several domains, such as bioinformatics, astronomy, e-commerce, etc. Moreover, the processing of tasks in parallel, popularized by the Map-Reduce paradigm represents an interesting and relevant solution for entity resolution, including its phase of partitioning and bloc creation. However, this parallelism beneficial in terms of performance must not degrade the quality of alignment. In this paper, we propose MREER (Map Reduce Et Entity Resolution), which allows alignment of entities derived from different data sources by exploiting the principle of Map-Reduce approach. The originality of our approach relies on the fact that beyond the aspects related to the processing and load balancing, the quality of the alignment in terms of equivalences found remains at the heart of the process. Experiments were established in order to validate our approach and compare it to an existing solution on a cluster of 4 machines and on collections of scientific publications. The results of our experiments have shown that MREER provides a better alignment quality, and reduces the number of unnecessary comparisons of more than 75% on average, compared with the existing method.

Keywords: entity resolution, Map-Reduce, duplicate detection, scaling.

1 Introduction

Actuellement, il existe plusieurs applications qui traitent des grandes quantités de données (ex., les journaux de requêtes web, les grandes bases de donnée). Cela est dû à l'émergence des techniques de *cloud computing* et notamment *MapReduce* [5, 9], qui facilite la création d'applications s'exécutant sur des clusters de machines.
Parmi ces applications, l'*Entity Resolution (ER)* est un problème omniprésent qui consiste à identifier les entités qui se réfèrent au même objet dans le monde réel. L'*ER* est un problème complexe qui apparaît sous plusieurs terminologies (ex. résolution de coréférence, déduplication, couplage d'enregistrements, entity matching). Il existe plusieurs applications de l'*ER* (ex. traitement d'images, extraction/intégration de l'information) [3].
Un exemple classique utilisé par la communauté travaillant autour de l'*ER* porte sur les articles scientifiques identiques qu'il est possible de trouver à partir de plusieurs sources (ex., *DBLP, ACM, Google Scholar*). Chaque article est caractérisé par un ensemble d'attributs (ex. l'année, les auteurs, le titre, la conférence/le journal). Le but de l'*ER* est de trouver les correspondances qui existent entre les entités, et d'identifier les entités qui sont similaires (alignées).
En plus de la qualité *de l'alignement*, le temps de calcul est un facteur très important qu'il faut prendre en considération en traitant les programmes d'*ER*. L'*ER* est un problème complexe et couteux en termes de temps, et qui peut prendre des heures ou

même des jours pour s'accomplir, surtout dans le cas où on est face à de grandes quantités de données (généralement des *téraoctets* ou plus).

En plus de la qualité *de l'alignement*, le temps de calcul est un facteur très important qu'il faut prendre en considération en traitant les programmes d'*ER*. L'*ER* est un problème complexe et couteux en termes de temps, et qui peut prendre des heures ou même des jours pour s'accomplir, surtout dans le cas où on est face à de grandes quantités de données (généralement des *téraoctets* ou plus).

L'approche basique qu'est le produit cartésien compare chaque entité avec la totalité des entités existantes. La complexité de ce traitement est de l'ordre $O(n^2)$, avec n est le nombre d'entités en entrée. Cette complexité n'est pas acceptable, surtout avec le grand nombre d'entités qui peut être de l'ordre de centaines de milliers ou même des millions.

Une alternative au produit cartésien proposée dans la littérature est de passer par une phase de *blocking*. Cette technique consiste à regrouper les entités en *blocs* en se basant sur une *clé* formée par un attribut ou un ensemble d'attributs de l'entité. Il a été montré que cette technique améliore de manière significative les performances d'alignement par rapport au produit cartésien, lorsqu'elle est déroulée sur des *clusters* de machines (calcul distribué) [3]. En effet, la complexité de traitement dans ce cas est égale à la complexité de traitement du plus grand bloc, et qui est toujours inférieure ou égale à $O(n^2)$.

Beaucoup de travaux de recherche ont été consacré à ce sujet [2, 3, 4], surtout après l'introduction du paradigme de calcul parallèle *MapReduce (MR)* par *J. Dean et S. Ghemawat* en 2004 [5, 9]. En plus des méthodes de calcul parallèle implémentant *MR*, des *Framework*s *MR* ont été mis en place afin de faciliter la création d'applications parallèles à grandes échelles. *Hadoop* [6], développé par la fondation *Apache*, représente le *Framework MapReduce* libre le plus fréquemment utilisé par la communauté.

L'utilisation de *MapReduce* a rendu facile des tâches qui étaient considérées comme très difficiles ou impossibles à résoudre avec les techniques de programmation séquentielles classiques. Néanmoins, l'application de ce paradigme peut être accompagnée de certaines faiblesses qui doivent être prises en considération lors de la conception des méthodes.

Dans le cas de l'ER, le problème de *déséquilibre de charge* (*load imbalances*) constituent une faiblesse naturelle du modèle *MapReduce* [3]. Généralement, les blocks obtenus seront de tailles différentes, et il peut y avoir des blocs dont les tailles sont beaucoup plus grandes que celles des autres. Dans ce cas, l'alignement des entités des grands blocs va prendre beaucoup plus de temps par rapport au reste des blocks. Pour remédier à ce type de problèmes, *L. Klob et al.* ont proposé deux techniques *BlockSplit* et *PairRange* qui ont comme objectif d'équilibrer la distribution des entités entre les blocks [3]. Un autre problème fréquent dans le domaine de l'ER, et surtout dans les méthodes basées sur le regroupement des entités en blocs, est le problème d'omission de comparaisons d'entités. Ce problème est le résultat de l'isolement des blocs. En effet une fois crées, l'alignement des entités de chaque bloc se fait indépendamment de l'alignement des entités des autres blocs. Des solutions ont été proposées dans la littérature afin de faire face à ce problème, et notamment les

méthodes dites collectives qui établissent l'alignement des entités d'une manière collective (i.e., l'alignement d'une paire d'entité peut dépondre de l'alignement des autres paires) [2]. Par contre, l'inconvénient majeur de ses approches est qu'elles souffrent d'un faible passage à l'échelle dans le cas de jeux de données volumineux.

Donc, la recherche d'un compromis entre passage à l'échelle et qualité de l'alignement posent donc les problèmes suivants :

- Le problème d'omission de comparaisons d'entités et le déséquilibre de la charge des blocs pour le cas des approches distribuées
- Le non passage à l'échelle des approches collectives

Dans cet article, nous proposons l'approche MREER, pour "Map Reduce Et Entity Resoution". Cette dernière s'appuie sur le calcul distribué (via Map Reduce) pour améliorer les performances. Comme *L. Kolb et al.* [3], le problème lié à la distribution provient de la taille variable des blocs, et nous avons adopté le même principe que celui de la méthode PairRange qu'ils ont proposée, pour construire des blocs de même taille. MREER apporte plusieurs contributions sur les aspects qualitatifs. Premièrement l'aspect itératif avec changement de la clé de blocking pour chaque itération. Cet aspect permet principalement de remédier au problème d'omission de comparaisons d'entités. De plus, l'un des défis soulevés concerne la sélection automatique de la clé de blocking à chaque itération. La clé doit en effet permettre de constituer des blocs équilibrés et de réaliser de nouvelles comparaisons entre entités. Dans cette optique, nous avons proposé une nouvelle stratégie qui permet de choisir automatiquement la meilleur clé parmi un ensemble de clés représentées sous la forme d'un treillis, à chaque itération du processus d'alignement.
La suite de l'article s'organise comme suit. La section 2 présente des notions générales sur Map-Reduce et l'alignement d'entités. La section 3 donne un aperçu des travaux existants, et notre proposition MREER est détaillée en section 4. Des expérimentations permettent de valider notre approche en section 5. Enfin, nous concluons et soulignons des perspectives en section 6.

2 Prérequis

Dans cette section nous allons présenter le paradigme de programmation parallèle MapReduce ainsi que le principe de l'entity resolution. Pour ce faire, nous utilisons l'exemple des articles scientifiques. En effet, nous disposant disposons de plusieurs articles (un article représente une entité) provenant de différentes bases de données (ex. ACM, DBLP et Google Scholar). Chaque article est caractérisé par un ensemble d'attributs (ex, l'année, les auteurs, le titre, la conférence/le journal). En premier temps, nous présenterons un traitement simple qui consiste à calculer le nombre d'articles par année à l'aide d'un processus MR. Ensuite, dans la prochaine partie de cette section, on entamera le problème de l'ER, où on présentera les principales classes de méthodes d'ER, puis on formulera le problème et on le déroulera sur l'exemple des articles scientifiques précédemment expliqué.

2.1 Le modèle de programmation Map-Reduce

MapReduce (MR) est un paradigme de programmation qui a été introduit par les chercheurs de Google, J. Dean et S. Ghemawat en 2004 [5, 9]. Ce modèle est destiné à la création de programmes parallèles qui traitent des grandes quantités de données, et qui s'exécutent sur des clusters de machines. Un programme MR s'écrit sous la forme de deux fonctions map et reduce, et les données sont représentées sous la forme de paires (clé, valeur).

$$map\ (Clé_{in}, Valeur_{in}) \quad \rightarrow \quad list\ (Clé_{int}, Valeur_{int})$$

$$reduce\ (Clé_{int}, list(Valeur_{in})) \quad \rightarrow \quad (Clé_{out}, Valeur_{out})$$

La figure 1 représente le principe général du paradigme Map-Reduce.

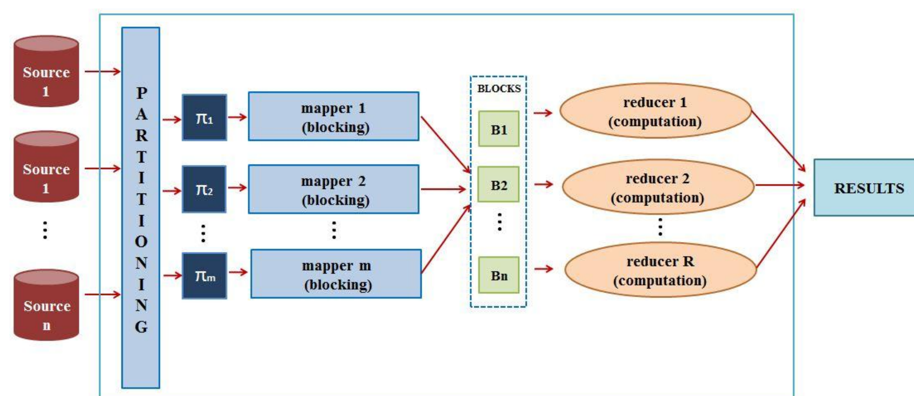

Fig. 1. Principe général du paradigme Map-Reduce

Dans ce qui suit, nous allons expliquer le principe de MR à l'aide de l'exemple de calcul de nombre d'articles par année.

Nous disposons d'un ensemble d'articles (entités) issues de plusieurs sources (ex. *DPLP*, *ACM* et *Google Scholar*). L'ensemble *E* d'entités en entrée est divisé en *M* partitions tel que *M* représente le nombre de tâches *map* que l'utilisateur spécifie. Chaque tâche *map* lit la partition qui lui correspond, avec comme clé d'entrée $(Clé_{in})$ le numéro de la partition donnée et comme valeur d'entrée $(Valeur_{in})$ l'ensemble des articles appartenant à cette partition. La lecture des différentes partitions se fait dans le même temps du fait que l'ensemble des tâches *map* s'exécutent en parallèle. Cela permet d'obtenir un temps de lecture égal à T/M, tel que T représente le temps nécessaire pour la lecture de la totalité des entités en séquentiel. Chaque tâche map aura comme entrée une paire de la forme *(Numéro de la partition, Les articles de la partition),* où le numéro de la partition représente la $Clé_{in}$ et le contenu de la partition (ensemble des articles de la partition) représente la Val_{in}.

Après avoir lu sa partition d'entités, chaque tâche *map* effectue les traitements qui ont été spécifiés par l'utilisateur et elle génère un ensemble de paires intermédiaires *(Clé$_{int}$, Valeur$_{int}$)*. Le choix de la clé d'entrée, la valeur d'entrée, la clé intermédiaire, la valeur intermédiaire et les traitements effectués par chaque tâche est effectué par l'utilisateur selon le besoin de son application. Dans notre exemple, la phase *map* servira principalement pour faire le *blocking* qui est considéré comme une option par défaut du modèle MR. Dans ce cas, la sortie de chaque tâche *map* sera un ensemble de paires *(Année, 1)*. Donc l'idée est qu'à chaque fois que la fonction map trouve un article dans la partition qui lui a été attribuée, elle génère une paire intermédiaire constituée de l'année de l'article et un compteur égal à 1. La fonction *map* ressemblera au pseudo-code 1.

```
map (int cléin,  List valeurin)
// cléin : numéro de la partition
// valeurin : liste des articles de la partition
Pour chaque article de la partition :
Générer (année, 1) ;
```

Pseudo-code 1 : fonction map

A la fin de la phase *map,* les paires *(Clé$_{int}$, Valeur$_{int}$)* seront regroupées automatiquement sans l'intervention de l'utilisateur, à l'aide de la fonction *part* qui établit le partitionnement *(blocking)* des entités et affecte chaque partition à la tâche *reduce* qui lui correspond parmi les R tâches *reduce* existantes. Ensuite, les tâches *reduce* qui s'exécutent en parallèle, établissent les traitements correspondants et écrivent les résultats *(Clé$_{out}$, Valeur$_{out}$)* soit dans un même fichier soit dans des fichiers différents. Dans le cas de notre exemple, chaque fonction *reduce* établit la somme des compteurs intermédiaires (les 1) correspondants à une année donnée, puis génère en sortie une paire *(année, nombre d'articles)*. La fonction *reduce* ressemblera au pseudo-code 2.

```
reduce (int cléint,  List valeurint)
// cléint : l'année
// valeurint : liste des compteurs intermédiaires
Int somme =0 ;
Pour chaque compteur de la liste valeur:
Somme = somme + compteur ;
Générer (année, somme) ;
```

Pseudo-code 2: function reduce

2.2 Alignement d'entités

Cette partie pose les bases du processus appelé alignement d'entités, qui prend en entrée des sources de données.

Definition 1: une **source de données** est un ensemble d'entités noté D. On note par */D/* le nombre d'entités que contient la source D. Dans notre travail, on se place dans un contexte où les schémas (attributs) des entités sont connus et homogènes.

Dans notre exemple d'articles scientifiques, nous avons trois sources de données : l'une avec les publications du site ACM, une autre avec les publications de DBLP et une dernière avec les publications issues de Google Scholar. Elles possèdent toutes le même schéma composé des attributs suivants : identifiant, titre, auteurs, lieu (de publication), année.

Définition 2: une entité est un objet du monde réel, décrit par des valeurs (instanciation des attributs/concepts du modèle). Voici une entité de la source ACM : *<(id,671326), (titre, "Using Schema Matching to Simplify Heterogeneous Data Translation"), (auteurs, "Tova Milo, Sagit Zohar"), (lieu, "Very Large Data Bases"), (année, 1998)>.*

Définition 3: une mesure de similarité est une fonction qui calcule un score de similarité entre deux objets. On distingue les mesures appliquées à des objets simples (e.g., entiers, chaînes de caractères) des mesures appliquées à des objets plus complexes, comme les entités dans notre cas.

Définition 4: une correspondance définit un lien sémantique entre deux entités. Le lien le plus utilisé est l'équivalence, mais l'on dispose également de relations telles que la subsomption ou l'inclusion. Une correspondance se représente par un triplé sous la forme *(e, e', r)* avec *e* et *e'* deux entités et *r* la relation sémantique qui les lie. L'ensemble des correspondances résultant d'un processus d'alignement se nomme **alignement**, que l'on note A.

3 Travaux connexes

On distingue dans cette section d'une part les travaux sur l'alignement d'entités, plutôt focalisés sur la qualité de l'alignement, et d'autre part les approches dont l'objectif principal est le passage à l'échelle, avec notamment l'utilisation de Map Reduce.

Comme le produit cartésien s'avère trop coûteux, la phase de blocking (aussi appelée indexation) permet de réduire le nombre de comparaisons entre entités. Afin de garantir une qualité d'alignement optimale, il est nécessaire de choisir judicieusement la clé de blocking, c'est à dire la combinaison d'attributs qui constituera les blocs. Nous présentons ici quelques-unes de ces approches les plus récentes, mais un tour d'horizon de toutes les approches est décrit par Ioannou et al. [11].

Plusieurs travaux de recherches ont été réalisés afin de mettre en place des méthodes qui font l'alignement des entités avec *MapReduce* [1, 2, 3, 4]. Le but des méthodes proposées est de maximiser la précision de l'alignement et de minimiser le temps de calcul. Par contre, il existe certains problèmes qui accompagnent les solutions de l'alignement de entités (ex, déséquilibre de charge entre les blocs (*load imba-*

lances), omission de comparaison de certaines entités), et qui mènent à l'obligation de trouver des compromis entre le temps de calcul et la qualité de l'alignement.

Dans ce qui suit, nous allons présenter le travail de *L. Klob et al.* constitué des deux méthodes *BlockSplit* et *PairRange* qui ont été proposées afin de remédier au problème de déséquilibre des tailles des blocs [3]. Ensuite nous allons présenter la méthode proposée par *V. Rastogi et al.* qui permet d'établir l'alignement des entités d'une manière collective [2]. Cette méthode permet principalement de remédier au problème de l'omission de comparaisons de certaines entités. A la fin de cette section, nous allons présenter l'idée de notre proposition, la méthode itérative et distribuée *MREER* qui garantit à la fois de remédier au problème de déséquilibre de charge et au problème d'omission de comparaisons d'entités.

3.1 Equilibrage de la charge des blocs ('load balancing')

Le problème de déséquilibre de charge entre les blocks ('load imbalances') constitue un problème très fréquent qui apparait dans la majorité des solutions d'alignement d'entités basées sur le modèle Map-Reduce (MR). Nous allons utiliser l'exemple des articles scientifiques afin d'expliquer ce problème.

Nous considérons que nous avons une source constituée de 100 entités (articles) que nous devons aligner avec une méthode d'alignement basée sur le modèle MR. Nous considérons aussi qu'à la fin de la phase *map*, nous avons obtenu 5 blocs dont les tailles sont présentées dans le tableau 1.

Tableau 1: Tailles des blocs obtenus après la phase *map*

N° du bloc	**Taille (entités)**
1	60
2	20
3	10
4	8
5	2

Chaque bloc parmi les 5 sera affecté à une tâche *reduce* pour établir l'alignement entre les entités qui y sont incluses. L'exécution des différentes tâches *reduce* se fait en parallèle sur 5 machines différentes. Comme les blocs sont de tailles différentes, les temps pris par les tâches *reduce* pour faire l'alignement des entités seront différents, et le temps d'exécution global sera égal au temps pris pour le traitement du plus grand bloc (bloc N°1). L'absence de mécanismes d'équilibrage de charge entre les blocs, constitue un véritable problème qui réduit énormément l'efficacité de l'exécution et la scalabilité des programmes MR dans le cas de jeux de données volumineux.

Maintenant nous allons présenter les deux méthodes de *load balancing*, qui sont *BlockSplit* et *PairRange* de *L. Klob et al.* et qui ont prouvé leur capacité à améliorer considérablement le temps d'exécution par rapport aux méthodes d'ER basiques (sans *load balancing*) [3]. Mais avant d'aborder ces deux méthodes, nous allons présenter l'étape d'analyse qui consiste en un processus MR complet durant lequel se fait le calcul d'une matrice appelée *Block Distribution Matrix (BDM)* qui permettra de définir le nombre d'entités que chaque bloc devra contenir. Cette matrice va être utilisée durant le deuxième processus MR par les deux méthodes *BlockSplit* et *PairRange* pour établir le *matching* des entités. Pour présenter leurs méthodes, les auteurs ont considéré qu'ils disposent d'une seule source *E* d'entités en entrée (i.e., l'ensemble des entités sur lesquelles ils vont appliquer leurs méthodes). Ils ont supposé également que toutes les entités en entrée disposent d'une clé de *blocking* valide. Ils ont proposé une idée simple qui permet la considération des entités qui ne disposent pas d'une clé de *blocking* valide. Cette idée consiste à comparer chacune de ces entités avec tout le reste des entités de *E*. En effet, les entités ne disposant pas d'une clé de *blocking* valide seront regroupées dans un ensemble E_0 tel que $E_0 \in E$, et ensuite faire le produit Cartésien entre E_0 et *E*.

Phase d'analyse.

Cette phase consiste à calculer la matrice de distribution de blocs *BDM* (*Block Distribution Matrix*). *BDM* est une matrice *B x M* tel que *B* est le nombre de blocs par partition et *M* représente le nombre de partitions des données en entrée (*M* représente aussi le nombre de tâches *map* à utiliser).

Le calcul de la *BDM* se fait à travers un processus MR complet (i.e., une phase *map* et une phase *reduce*). Durant la phase *map*, chaque fonction *map* lit une partition d'entités et détermine pour chaque entité sa clé de *blocking*, puis elle génère en sortie une paire $(Clé_{int}, Valeur_{int})$ avec comme $Clé_{int}$ une clé composite constituée du numéro de la partition à laquelle appartient l'entité et la clé de *blocking* de l'entité concernée ($Clé_{int}$ = *Clé de blocking* • *Numéro de la partition*) et une $Valeur_{int}$ correspondante égale à 1. Ensuite, les paires intermédiaires sont regroupées selon la clé composite pour s'assurer que les entités du même bloc seront tous affectées à la même tâche *reduce.* Après, dans la phase *reduce,* chaque tâche *reduce* fait la somme des valeurs intermédiaires appartenant au même bloc et génère en sortie des triplets de la forme *(Clé de blocking, Numéro de la Partition, Nombre d'entités)*.

Phase de matching.

Après la première phase d'analyse durant laquelle se fait le calcul de la matrice *BDM*, arrive la phase de *matching* qui occupera à son tour un processus MR complet. La phase *map* permet d'établir le *blocking* des entités et mettre en place le *load balancing* en se basant sur le contenu de la matrice *BDM*. La phase *reduce* permettra d'établir l'alignement des entités des différents blocs résultants de la phase *map*.

Dans ce qui suit, nous allons présenter les deux méthodes de *load balancing* *BlockSplit* et *PairRange* de *L. Klob et al.*[3].

Méthode 'BlockSplit'

La première méthode proposée par L. Klob et al. BlockSplit, permet d'établir un load balancing en se basant sur les tailles des différents blocs qui ont été calculées durant la phase d'analyse (la matrice BDM). Elle repose sur deux idées principales :

- Les petits blocs sont affectés à une seule tâche *reduce* pour établir l'alignement des entités qui y sont incluses. Par contre, les grands blocs sont divisés en *M* sous-blocs tel que *M* est le nombre de partitions d'entités en entrée. Ensuite, chaque sous bloc est affecté à une tâche *reduce* pour établir l'alignement. Chaque tâche *reduce* établit le produit cartésien des entités du bloc qui lui a été affecté. Dans le cas des grands blocs, chaque tâche reduce établit le produit cartésien entre deux sous-blocs du bloc original, afin de s'assurer que toutes les comparaisons nécessaires soient effectuées.
- Avant d'affecter les blocs aux tâches *reduce*, la méthode *BlocSplit* calcule le nombre de comparaisons qui vont avoir lieu durant le *matching* pour chaque bloc. Ceci est garanti par le fait que les tailles des différents blocs sont calculées dans la matrice *BDM*. Ensuite, les blocs sont affectés aux tâches reduce selon une heuristique gloutonne de *load balancing*. L'idée de cette heuristique consiste à affecter les blocs aux tâches *reduce* selon l'ordre décroissant de leurs tailles (i.e., les grands blocs sont affectés avant les petits blocs). Cela garantira que l'équilibrage du temps d'exécution des différentes tâches *reduce*.

Le principal inconvénient de cette méthode, est que malgré la mise en place de l'heuristique d'équilibrage de la charge des blocs, il est toujours possible que les différences entre les tailles des blocs soient significatives. Cela est dû au fait que la division des grands blocs se fait par rapport au nombre de partitions *M*, ce permet rarement d'avoir des blocs de mêmes tailles.

Méthode 'PairRange'

La méthode *PairRange* a été proposée pour remédier au problème de déséquilibre des charges des blocs qui apparait dans la méthode *BlockSplit*. Le principe du *load balancing* dans *BlockSplit* été basé sur la notion de bloc. Par contre, dans la méthode *PairRange*, les auteurs ont implémenté un mécanisme d'équilibrage plus sophistiqué basé sur le nombre de paires qui doivent être affectées à chaque tâche *reduce*. En effet, l'idée est d'affecter le même nombre d'entités aux différentes tâches *reduce* afin d'obtenir des temps d'exécution équivalents. Cette méthode est basée sur deux idées fondamentales:

- La méthode *PairRange* établit une énumération de toutes les entités existantes et des comparaisons (les paires) qui vont avoir lieu en se basant sur le contenu de la matrice BDM. Cela est nécessaire pour déterminer la tâche *reduce* à laquelle on doit affecter une entité donnée [3] ;
- Pour établir le *load balancing*, *PairRange* divise l'ensemble des entités en *R* sous-ensembles de tailles (presque) égales, tel que *R* est le nombre de tâches *reduce* dont on dispose. Ensuite, chaque sous-ensemble est affecté à la tâche reduce qui dispose du même rang.

Il a été prouvé à travers les expérimentations établies dans [3] que la méthode PairRange dépasse la méthode BlockSplit et la méthode basique d'ER (sans load Balancing) en termes de temps d'exécution pour des facteurs différents (le nombre de tâches reduce utilisées et le nombre de nœuds utilisés).

3.2 Alignement collectif des entités

Les approches d'alignement des entités sont classées en deux grandes familles, les approches dites conventionnelles et celles dites collectives. Les approches conventionnelles, comme les deux méthodes BlockSplit et PairRange que nous avons présenté précédemment, sont des méthodes qui font l'alignement des entités à l'aide de mesures de similarités sans l'utilisation des relations qui peuvent exister entre les entités. Ces méthodes sont relativement moins couteuses en termes de temps et moins performantes en termes de précision par rapport aux méthodes collectives [2]. La supériorité des méthodes collectives en termes de précision de l'alignement vient du fait que ces dernières exploitent en plus des mesures de similarités, les relations qui peuvent exister entre les différentes entités. Cela permet de faire face au problème d'omission de certaines comparaisons, qui peut avoir lieu dans les méthodes conventionnelles. Par contre, ces méthodes souffrent d'une faible scalabilité dans le cas de jeux de données volumineux. Pour remédier à ce problème, V. Rastogi et al. ont proposé une technique collective d'alignement d'entités que nous allons présenter dans ce qui suit. Mais avant, nous allons faire un rappel de l'exemple des articles scientifiques qui va nous servir pour expliquer cette méthode.

Nous disposons d'un ensemble d'entités E. Cet ensemble est divisé en deux sous-ensembles P et A tel que P représente l'ensemble des entités de type article et A l'ensemble des entités de type auteur. Nous disposons également d'un ensemble de relations R= {Authored, Cites, Coauthor} tel que Authored (a, p) signifie que a est un auteur de p, Cites (p_1, p_2) signifie que p_1 cite p_2 et Coauthor (a_1, a_2) signifie que a_1 et a_2 ont coécrit un article.

Dans ce qui suit, nous allons présenter la méthode à l'aide de l'exemple d'alignement des entités de type auteur (voir figure 2) que les auteurs de la méthode ont utilisé.

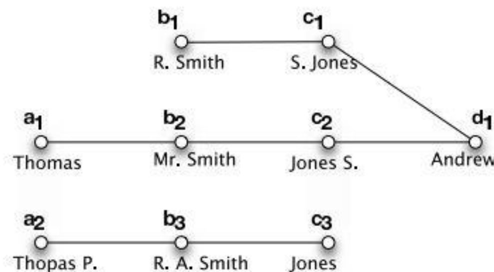

Fig. 2. Un exemple du problème d'*entity matching* : un sommet représente un auteur et un arc représente une relation de Coauteur entre deux sommets [2]

Alignement d'entités base sur les réseaux logiques de Markov (RLM).

Comme on a indiqué précédemment dans l'introduction, les approches de *matching* collectives se basent sur des outils de *machine learning* tels que les réseaux *Bayésiens,* les *réseaux logiques de Markov, etc.* Les auteurs de la méthode que nous sommes en train de présenter ont choisi comme outil d'alignement les *RLM.* La méthode proposée est capable d'utiliser d'autres types d'outils autres que les *RLM*, car le but des auteurs était de proposer une méthode générique qui permet d'implémenter n'importe quel outil d'alignement collectif.

Maintenant nous allons expliquer le principe de fonctionnement du *matching* collectif basé sur les *RLM* en se servant de l'exemple de la figure 2. A partir de la figure on peut tirer par exemple les relations suivantes : *Coauthor* (c_1, d_1), *Coauthor* (c_2, d_1), etc. La relation *Coauthor* est utilisée en plus d'une fonction *Similar* pour définir une autre relation que les auteurs ont appelée *Match* qui permet d'indiquer si deux entités doivent être alignées ou pas. La fonction *Similar* est une fonction qui permet d'indiquer si deux chaines de caractères sont similaires ou pas. D'une manière intuitive, deux entités (auteurs) sont alignées si elles ont le même nom, et elles partagent le même coauteur. Cela peut être exprimé à l'aide de ces deux règles :

R1: Similar (x, y) => Match (x, y)

R2: Similar (x1, y1) ∧ Coauthor (x1, x2) ∧ Coauthor (y1, y2) ∧ Match (x2, y2) => Match (x1, y1)

L'utilisateur pourra écrire d'autres règles autres que *R1* et *R2*. Ensuite le système va affecter un poids à chaque règle suite à une étape d'apprentissage. Ces poids sont utilisés pour attribuer un score à chaque ensemble d'entités à aligner. Le score de l'ensemble est la somme des poids des règles appliquée sur les paires d'entités du groupe, et qui sont mises à vrai.

Pour pouvoir expliquer la méthode, les auteurs ont supposé que le système a attribué à la règle *R1* un poids de -5 et à *R2* un poids de 8. Le poids négatif de *R1* signifie que cette règle ne peut pas être utilisée toute seule pour affirmer que deux entités sont alignées. Par contre, si pour une paire d'entités, les deux règles *R1*et *R2* sont respectées, alors le poids total sera égal à 8 – 5 = 3, ce qui implique que les deux entités seront alignées du fait que le score total est supérieur à zéro. On considère l'exemple des entités c_1 et c_2 de la figure 2. Le système commence par générer deux ensembles intermédiaires $\{c_1, c_2\}$ qui signifie que c_1 et c_2 vont être alignées, et l'ensemble vide {} qui signifie que c_1 et c_2 ne vont pas être alignées. Ensuite pour chacun des deux ensembles, le système applique les deux règles *R1* et *R2*, et calcule le score obtenu. L'ensemble qui va être gardé est celui qui aura le plus grand score.

On commence par le premier cas, où c_1, c_2 sont supposées alignées. La relation *R1* est mise à vrai du fait que les noms des entités sont similaires (cela est vérifié par la fonction *Similar de R1*). Donc on ajoute au compteur des scores le poids de *R1* qui est égal à -5. La règle R2 est aussi vérifiée car c_1 *et* c_2 possèdent le même nom d'auteur et partagent le même coauteur qui est d_1. Donc on ajoute au compteur le poids de *R2* qui est égal à 8. Donc au total, le compteur de $\{c_1, c_2\}$ sera égal à +3. Pour le deuxième ensemble { }, les deux relations *R1* et *R2* ne sont pas vérifiées, donc le score attribué à cet ensemble est 0. Donc le système décidera que c_1 et c_2 sont alignées du fait que le score de l'ensemble $\{c_1, c_2\}$ est supérieur à celui de l'ensemble {}. Continuons avec

l'exemple de la figure 2, mais en prenant cette fois comme exemple, les deux entités b_1 et b_2. Ces deux entités respectent la règle *R,* donc le score -5 sera ajouté au compteur (initialement égal à 0). Par contre, si on considère que l'alignement de cette paire $\{b_1, b_2\}$ se fait indépendamment du premier résultat obtenu (i.e., c_1 est alignée avec c_2), alors le système conclura que la règle *R2* n'est pas vérifiée pour b_1 et b_2 (le système ne sait pas que *Match* (c_1, c_2) est vraie). Dans ce cas le système va conclure que b_1 et b_2 ne sont pas alignées malgré qu'en réalité elles représentent la même entité. C'est dans ce type de cas qu'on constate l'importance de faire l'alignement des entités d'une manière collective. En effet, pour le cas de la paire $\{b_1, b_2\}$, le système pourra conclure que ces deux entités sont alignées s'il utilise le résultat de l'alignement de $\{c_1, c_2\}$.

Après avoir présenté le principe de l'alignement collectif des entités, nous allons maintenant présenter la méthode proposée par les auteurs *V. Rastogi et al.* [2] qui permet de dérouler ce type d'alignement sur des clusters de machines. L'idée repose sur deux notions clés : la notion de voisinages (*neighborhoods*) et la notion de passage de messages.

Voisinages

L'ensemble *E* d'entités en entrée est divisé en sous-ensembles appelés voisinages. Une entité peut appartenir à plusieurs voisinages à la fois. Une couverture (*cover*) est un ensemble de voisinages dont l'union est égale à *E* (voir figure 3). La figure 3 représente un exemple de couverture obtenue à partir du schéma de la figure 2. La notion de couverture représente une extension de la notion de *blocking*, du fait que le regroupement des entités dans la notion de couverture ne se fait pas juste à l'aide des mesures de similarité, mais aussi à l'aide des relations qui les relient.

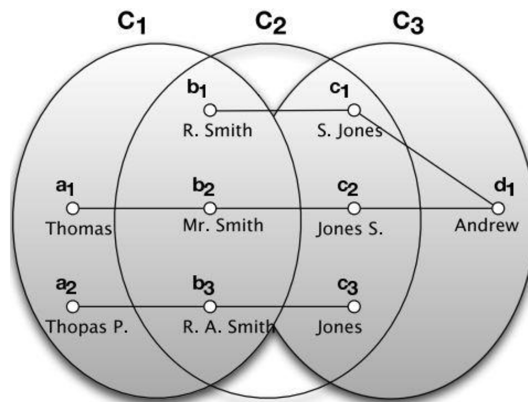

Fig. 3. Exemple de couverture issue du schéma de Fig.2 [2]

Après avoir créé les voisinages, le système lance en parallèle un nombre d'instances (selon le modèle *MR*) du programme d'alignement égal au nombre de voisinages qui existent (une instance pour chaque voisinage). Dans notre exemple, le nombre d'instances est égal à 3, car la couverture est constituée de trois voisinages

C_1, C_2 et C_3. Chaque voisinage sera affecté à une instance du programme afin d'établir l'alignement des entités qui y sont incluses.

Dans un premier temps, on considère que l'alignement de chaque voisinage se fait indépendamment des autres voisinages. Ainsi, dans le voisinage C_3, les entités qui seront alignées sont $\{c_1, c_2\}$. Par contre dans les voisinages C_1 et C_2 il n'y aura pas d'entités alignés. En revanche, si l'instance sur laquelle se déroule l'alignement de C_3 aurait passé les paires qu'elle a aligné à C_2, il aurait été possible d'aligner les deux entités b_1 et b_2. Pour cette raison, les ont appliqué un mécanisme qui de passage de messages entre les différentes instances du programme d'alignement qui se déroulent en parallèle.

Passage de messages

Le passage de message est un mécanisme important qui permet d'établir des communications entre les différents voisinages de la couverture. Il existe deux types de messages, les messages simples et les messages maximaux.

Un message simple consiste à l'envoie des entités alignées dans un voisinage aux autres voisinages. Cela permettrait à ces voisinages d'utiliser ces résultats pour aligner quelques entités. Si on prend l'exemple de la figure 3, il est impossible que les entités b_1 et b_2 soient alignées sans que le voisinage C_3 informe C_2 que les entités c_1 et c_2 ont été alignées. Par contre malgré l'application du passage de messages simples, il reste impossible d'aligner des paires d'entités comme $\{b_2, b_3\}$ et $\{a_1, a_2\}$.

Pour cela, la notion de message maximal a été introduite. En effet, b_2 et b_3 partagent le même nom d'auteur, et de même a_1 et a_2. En plus de cela, il existe une relation de coauteur entre a_1 et b_2 et entre a_2 et b_3. Dans ce cas, le système conclura que l'alignement de $\{b_2, b_3\}$ dépond de l'alignement $\{a_1, a_2\}$ et vice versa. Ainsi il considère les entités b_2 et b_3 comme alignées et de même les entités a_1 et a_2.

Malgré que la méthode présentée remédie au problème de la faible scalabilité des algorithmes de matching collectifs, grâce à leur exécution en mode distribué (la notion de couverture et de passage de messages), il reste quelques problèmes qui n'ont pas été abordés et qui peuvent affecter la performance en termes de temps d'exécution. Le principal problème qui peut être détecté, est le problème de la synchronisation de l'exécution des différentes instances du programme d'alignement. En effet, l'alignement des entités appartenant à un voisinage peut dépondre des résultats de l'alignement des entités d'autres voisinages, ce qui nécessite l'envoi de messages entre les différentes instances sur lesquelles se déroule l'alignement des voisinages concernés. Par contre, rester bloqué en attente des résultats des autres voisinages pourra allonger le temps d'exécution, et pourra également mener à des situations d'inter-blocage. Un autre problème qu'il faut prendre en considération est l'équilibrage des tailles des voisinages (*load balancing*).

Dans cette section, nous avons présenté les principales approches existantes qui font de l'*ER* avec *MR*. Les deux premières méthodes que nous avons présenté (*BlockSplit* et *PairRange*) avaient comme objectif principal d'améliorer le temps d'exécution en mettant en place des mécanismes pour équilibrer la charge d'entités affectée aux différentes tâches *reduce* (phase de *matching*). En revanche, ces mé-

thodes peuvent souffrir d'un problème d'omission de comparaisons d'entités. La troisième méthode que nous avons présentée, avait traité l'alignement collectif des entités. L'objectif de cette méthode était d'améliorer la scalabilité des algorithmes de *matching* collectif, qui malgré leur grande précision, souffrent d'un problème de faible scalabilité [2]. La méthode que nous avons proposée (*MREER*) et qui sera présentée dans la prochaine section, s'intéresse aux deux aspects fondamentaux qui ont été traité par les trois méthodes que nous avons proposées, et qui sont l'amélioration du temps d'exécution (matérialisé par le *load balancing*) et la précision de l'alignement qui représente le point fort des méthodes collectives. Notre objectif est d'obtenir une performance en termes de temps de calcul proche de celle de la méthode *PairRange* de *L. Klob et al.* [3] et une performance en termes de précision proche de celle des méthodes collectives. Par contre, notre solution n'est pas d'appliquer le mécanisme du *load balancing* sur les méthodes collectives, mais c'est de proposer une approche de *matching* à la *MR* qui implémente le *load balancing* (amélioration du temps de calcul) et un mécanisme de filtrage itératif (amélioration de la qualité de l'alignement) moins couteux en termes de temps que le mécanisme de passage de message de la méthode de *V. Rastogi et al.* [2].

4 L'approche MREER

Dans cette section nous détaillons notre approche itérative d'*ER* que nous avons appelée MREER pour Map Reduce Et Entity Resolution. La figure 4 représente une vue globale de l'approche. MREER est une approche basée sur *MR*, destinée à faire l'alignement d'entités sémantiques issues de sources distribuées. Le principe de notre approche repose sur deux points fondamentaux, l'aspect itératif de l'alignement avec changement de la clé de *blocking* et l'aspect de parallélisation des calculs avec mise en place du mécanisme du *load balancing*.

Chaque itération du processus d'alignement est constituée d'un ensemble de traitements que nous détaillerons par la suite dans cette section. L'aspect itératif de l'approche permet de détecter le maximum d'entités correspondantes et de remédier au problème de l'omission de comparaisons qui apparaît souvent dans les approches d'ER basées sur le modèle MR.

Le deuxième aspect important de l'approche, qu'est la parallélisation des calculs, permet de tirer avantage du modèle MR en termes de performance. Un mécanisme de *load balancing* inspiré de la méthode PairRange de *L. Klob et al.* est mis en place afin de remédier au problème de déséquilibre entre les charges des blocs qui constitue une faiblesse naturelle des solutions d'*ER* basées sur le modèle *MR* [3].

4.1 Principe

L'algorithme MREER permet d'effectuer l'alignement entre des paires d'entités regroupées au sein de blocs. Ces blocs sont construits à partir d'un ou plusieurs attributs servant de clés de bloc. L'idée de MREER est de prendre une clé de bloc différente à chaque itération, et de retirer à chaque itération les entités alignées, et de garder les

entités non alignées qui feront l'objet de l'entrée d'une nouvelle itération du processus d'alignement. Pour ce faire, différentes étapes sont nécessaires et sont représentées sur la figure 4.

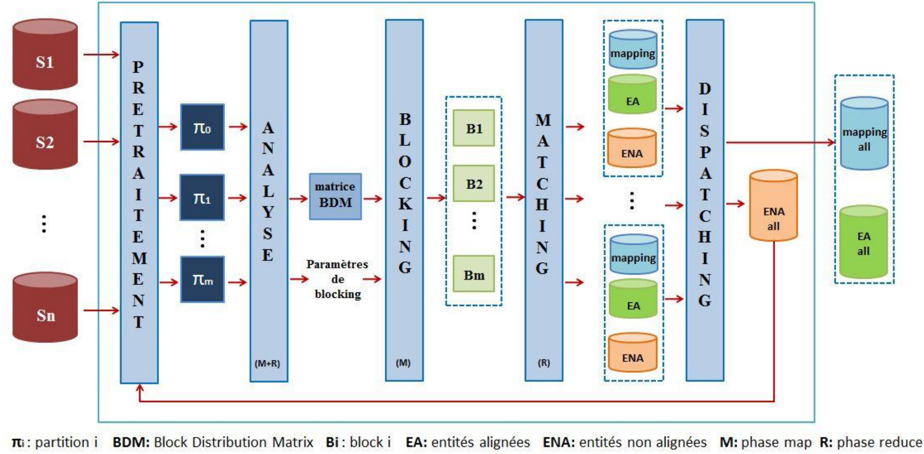

Fig. 4. Vue globale de l'approche MREER

Nous considérons l'ensemble $C = \{c_1, c_2, \dots, c_{n_c}\}$ de concepts représentées sous forme d'entités présentent dans l'ensemble des sources $S = \{S_1, S_2, \dots, S_{n_S}\}$. n_c représente le nombre total de concepts représentés dans l'ensemble des n_s sources fournissant des entités. Chaque source S_i fournit l'ensemble d'entités $E_i = \{e_1^i, e_2^i, \dots, e_{m_i}^i\}$ où e_j^i est la $j^{ème}$ entité de la source S_i et m_i le nombre d'entités fournies par la source S_i. Nous posons $E = \bigcup_{i=1}^{ns} Ei$ l'ensemble de toutes les entités disponibles à travers les sources.

Par la suite, nous faisons les hypothèses suivantes. Chaque entité présente dans une source représente un concept de C. En considérant la fonction $c: E \rightarrow C$, nous avons $\forall\, i, j, \exists\, k$ tel que $e_i^i = c_k \in C$. Aucune source de S ne dispose de doublons parmi ses entités et chaque entité d'une même source représente des concepts différents. Autrement dit, $\forall\, k \in [1, n_s], \forall\, i, j \in [1, m_k], i \neq j \Rightarrow e_i^k \neq e_j^k \, et \, c(e_i^k) \neq c(e_j^k)$. On suppose également que les schémas des différentes sources sont homogènes de manière à nous focaliser sur l'hétérogénéité sémantique des entités au niveau instance et non structurelle. Nous posons $S_c = \{a_1, a_2, \dots, a_{n_a}\}$ le schéma des entités où a_i représente le $i^{ème}$ attribut du schéma composé de n_a attributs.

Analyse du jeu de données .

La phase d'analyse se décompose en deux traitements. Le premier traitement consiste en la création d'un treillis résultant de l'analyse du schéma des entités.

On note par T ce treillis et *T(att)* le nœud/élément du treillis composé de l'attribut ou de l'ensemble d'attributs *att*. Cette première phase d'analyse du schéma des entités

s'effectue une seule fois à l'initialisation de l'application. Ce treillis permet de représenter les $n_t = 2^{n_a}$ sous-ensembles d'attributs (voir figure 5).

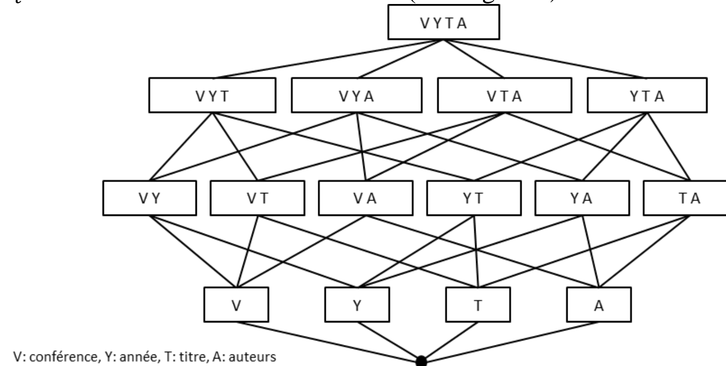

Fig. 5. Structure du treillis

Le deuxième traitement d'analyse de la distribution des entités est matérialisé par le calcul de la matrice de distribution de blocs (BDM) pour chaque élément du treillis sauf pour le premier élément qu'est la racine.

La matrice BDM est une matrice *B x M* qui indique le nombre d'entité que contient chaque bloc appartenant à une partition donnée, où *M* est le nombre de partitions existantes, et *B* représente le nombre de blocs dans chaque partition. Un élément *(i, j)* de la matrice BDM, représente le nombre d'entités que contient le bloc i de la partition j, tel que $1 \leq i \leq B$ et $1 \leq i \leq M$.

Le calcul de la matrice *BDM* se fait à travers un processus MR constitué d'une phase *map* et d'une phase *reduce*. Chaque fonction *map* lit une partition d'entités $\boldsymbol{\pi_i}$, et pour chaque entité de la partition, elle détermine sa clé de *blocking* et génère en sortie une paire *(clé de blocking **.** numéro de la partition, 1)*. Le numéro de la partition est rajouté à la clé de blocking de l'entité afin que les entités appartenant à une partition donnée, restent dans la même partition après la phase de *blocking*. Après que le *blocking* soit établit, chaque bloc est affecté à une tâche *reduce* dans laquelle se fait la somme de tous les compteurs intermédiaires du bloc concerné, et à la fin cette fonction génère un triplé de la forme *(clé de blocking**,** numéro de la partition, nombre d'entités)*.

Les fonctions *map* et *reduce* correspondant au calcul de la *BDM* sont représenté par les pseudo-codes :

```
map (int clé_in,  List valeur_in)
// clé_in : numéro de la partition
// valeur_in : liste des entités de la partition
Pour chaque entité de la partition :
Générer (clé de blocking. Numéro de la partition, 1) ;
```

Pseudo-code 3: fonction map

```
reduce (int clé_int,  List valeur_int)
// clé_int : clé de blocking. Numéro de la partition
// valeur_int : liste des compteurs intermédiaires
Int nbr_entites =0 ;
Pour chaque compteur de la liste valeur:
nbr_entites = nbr_entites + compteur ;
Générer (clé de blocking, Numéro de la partition, nbr_entites) ;
```

Pseudo-code 4 : fonction reduce

Sélection d'une clé .

Le choix de la clé de *blocking* est un aspect crucial pour le bon déroulement de l'alignement. Généralement, pour obtenir une bonne qualité de l'alignement, il est préférable d'imposer une clé de blocking stricte composées de plusieurs attributs de l'entité. Ce choix permet de réduire le nombre de comparaisons inutiles, en ne comparant que les entités qui sont très proches les unes des autres. Mais ce choix pourra emmener à un problème d'explosion du nombre de blocs (un grand nombre de blocs de petites tailles). Par contre, si on choisit une clé de *blocking* moins stricte, plusieurs comparaisons inutiles peuvent avoir lieu, et des comparions utiles peuvent être omises. En effet, on donne l'exemple suivant :

Nous avons un ensemble d'articles scientifiques, tel que chaque article a la forme suivante <année, auteur(s), titre, conférence/journal>. Les articles sont les suivants :

P_1 (année$_1$, auteur$_1$, titre$_1$, conf$_1$), P_2 (année$_2$, auteur$_1$, titre$_1$, conf$_1$), P_3 (année$_3$, auteur$_1$, titre$_1$, conf$_1$), P_4 (année$_1$, auteur$_4$, titre$_4$, conf$_4$) et P_5(année$_1$, auteur$_5$, titre$_5$, conf$_5$).

Donc si on choisit comme clé de blocking l'année, on obtiendra les blocs suivants : $B_1 = \{P_1, P_4, P_5\}$, $B_2 = \{P_2\}$, $B_3 = \{P_3\}$. Dans ce cas les entités qui seront comparées sont seules les entités appartenant au même bloc. On remarque que les entités du bloc B_1 n'ont en commun que l'attribut année, et malgré ça elles se trouvent dans le même bloc. Par contre, les entités P_1, P_2 et P_3 ont trois attributs en commun et qui sont l'auteur, le titre et la conférence, et malgré ça elles ne seront pas comparées.

Pour remédier à ce problème, nous avons opté dans notre approche MREER pour un alignement itératif avec changement de la clé de blocking pour chaque itération du processus d'alignement. A travers le changement de la clé de blocking, il se peut que des entités qui n'ont pas été alignées lors d'une itération donnée, soient alignées dans l'itération qui la suit, à condition que ces entités soient dans le même bloc avec l'application de la nouvelle clé de regroupement. Reprenons l'exemple des articles scientifiques présenté précédemment dans cette section. Lorsque nous somme partit avec comme clé de blocking l'année, nous avons constaté que les articles P_1, P_2, et P_3 ne seront pas comparés puisqu'ils sont retrouvé dans des blocs différents malgré qu'ils disposent de trois attributs similaires. Donc, si on applique une nouvelle itération d'alignement avec comme clé de blocking la conférence, on obtient les blocs

suivants : $B_1 = \{P_1, P_2, P_3\}$, $B_2 = \{P_4\}$, $B_3 = \{P_5\}$. Ainsi les entités P_1, P_2, et P_3 seront comparées puisqu'elles sont retrouvées dans le même bloc.

Pour choisir une clé de blocking parmi les n_t éléments du treillis, nous avons proposé la stratégie BILBOH (*Best In Lattice and Based On Height*) qui associe à chaque élément du treillis sa hauteur. La hauteur est usuellement considérée comme la longueur du plus long chemin entre la racine du treillis et l'élément considéré. Ainsi, la racine du treillis est considérée de hauteur zéro et les feuilles treillis ont la hauteur maximale. Il est important de noter que plus la hauteur de l'élément choisi comme clé est petite, plus les contraintes de sélection seront importantes sur les entités ce qui intuitivement réduit le nombre d'entités par bloc et augmente le nombre de blocs.

La stratégie BILBOH consiste à considérer pour l'itération i les éléments de de hauteur i où $h(T)$ est la hauteur du treillis T. Ainsi, à une itération i, la clé de blocking est constituée de $n_a - i$ attributs (pour mémoire n_a correspond au nombre total d'attributs du schéma S_c), et correspond à l'élément qui donne le plus de comparaisons d'entités par rapport aux autres éléments qui disposent de la même hauteur.

Création des blocs ('blocking').

Cette étape permet de regrouper les différentes entités existantes selon la clé de blocking précédemment choisie. Dans cette phase, et en se basant sur la matrice BDM qui a été créée lors de la phase d'analyse, on cherche à créer des blocs de paires d'entités en s'inspirant du principe de la méthode PairRange proposée par *L. Klob et al.* [3].

Dans les solutions d'alignement d'entités basées sur le paradigme MR, la phase *map* sert généralement pour établir le blocking des entités, en déterminant pour chaque entité sa clé de blocking, ce qui permet par la suite à la fin de la phase *map* de regrouper les entités qui partagent la même clé en un seul bloc. Par contre, les blocs obtenus peuvent être de tailles différentes, ce qui affecte le temps pris pour établir l'alignement, vu que le temps global de l'alignement sera égal au temps de traitement du plus grand bloc (traitement parallèle). Pour cette raison, la méthode PairRange a été proposée. L'idée de cette méthode est de déterminer les paires d'entités qui doivent être comparées ensembles, ensuite créer R groups de paires d'entités de tailles équivalentes (pour mémoire R est le nombre de tâches reduce à utiliser). Dans notre contexte, où nous cherchons à aligner des entités issues de sources différentes, nous avons la contrainte supplémentaire qu'une paire ne peut pas contenir deux entités provenant de la même source. Se baser sur cette solution nous semble judicieux dans le sens où elle nous permet de gérer efficacement et simplement l'équilibrage de charge entre les différentes tâches *reduce* (voir section Travaux connexes).

Calcul des similarités.

Cette étape correspond à la phase d'exécution des comparaisons entre entités et à la phase de décision pour déterminer si deux entités sont équivalentes. Nous supposons disposer d'une mesure de similarité *mSim : (String; String) double* qui pour deux chaines de caractères en entrée retourne un score compris entre [0,1]. Plus le score est

proche de 1, plus la similarité entre les deux chaînes de caractères est importante. Pour chaque paire d'entités à comparer, nous appliquons cette mesure de similarité à chaque valeur d'attribut hormis les attributs constituant la clé de blocking. La similarité Sim_i calculée à l'itération i entre deux entités issues de sources différentes est exprimée par l'équation suivante :

$$\forall\, \boldsymbol{h}, \boldsymbol{k}, \forall \boldsymbol{e} \in \boldsymbol{E_h}, \boldsymbol{e'} \in \boldsymbol{E_k}, \boldsymbol{Sim_i}(\boldsymbol{e}, \boldsymbol{e'}) = \frac{\sum_{att \in A_i} mSim(e[att], e'[att])}{|A_i|}$$ **Équation 1**

Où l'ensemble A_i correspond à l'ensemble des attributs composant le schéma des entités privé des attributs constituant la clé de blocking, et e[att] *(resp.* e'[att]*)* représente la projection de l'entité *e (resp. e')* sur l'attribut *att*. Pour décider si les deux entités comparées sont équivalentes, on suppose qu'un seuil $theta_{sim}$ est préalablement défini par l'utilisateur et pour tout score de similarité entre deux entités au-delà de ce seuil permet de décider que les entités sont équivalentes. Les calculs de similarité sont effectués aux sein de tâches Reduce et il est important de noter que le traitement de chacun des blocs à l'itération i permet d'obtenir en sortie trois ensembles (EA_i , M_i , ENA_i) où *EA* représente l'ensemble des entités alignées, M_i les correspondances trouvées(mappings), et ENA_i les entités non alignées.

Préparation de l'itération suivante.
Cette étape permet de générer le jeu de données qui fera l'objet de l'entrée de l'itération suivante. Logiquement, les entités en entrées d'une itération i doivent être les entités non alignées lors de l'étape qui la précède (i.e., ENA_{i-1}). Comme indiqué précédemment, à la fin de chaque itération i du processus d'alignement des entités, trois types de sorties sont générées par chaque tâche reduce les EA_i, les M_i et ENA_i. Ces sorties, ainsi que les sorties des autres tâches *reduce* sont regroupées dans les ensemble EA_{all} , les M_{all} et ENA_{all} qui représentent respectivement l'ensemble des entités alignées, l'ensemble de correspondances trouvées et l'ensemble des entités non alignées depuis de le début du processus d'alignement.

4.2 Algorithme

Après avoir énoncé le principe et motivé certains choix, nous présentons de manière plus détaillée l'algorithme présenté dans 1. La fonction MREER prend en entrée l'ensemble *D* des entités à traiter, le schéma *S* des entités, l'entier T_M (respectivement T_R) correspondant au nombre maximum de tâches de type map (respectivement reduce) supportées par le système, l'entier $iter_{max}$ le nombre d'itérations maximum accepté par l'utilisateur et, *mSim* l'ensemble des mesures de similarité exploitées lors de la comparaison entre les entités. L'ensemble *Mappings* permet de stocker les correspondances entre entités trouvées à chaque itération.

Les lignes 2 et 3 correspondent à la phase d'analyse du jeu de données. Dans un premier temps (ligne 2), le schéma *S* est analysé pour générer l'ensemble *SubSetAtt* des $m = 2^{card(s)} - 2$ sous-ensembles d'attributs possibles (l'ensemble vide et l'ensemble *S* ne sont pas considérés).

Ensuite (ligne 3), un processus Map Reduce est lancé pour construire la matrice BDM pour chacun des *m* ensembles d'attributs.

La phase map construit pour chacun des m sous-ensemble d'attributs, les ensembles de valeurs distinctes prises par les entités et le nombre d'entités prenant ces valeurs et ce, au sein de chaque partition. La phase reduce agrège les résultats obtenus au niveau des différentes partition en fusionnant les ensembles de valeurs et en faisant la somme cumulée des nombres d'entités prenant ces valeurs, puis en calculant la cardinalité de chaque ensemble de valeurs pour construire les *m* matrices BDM.

Algorithm 1 *Function MREER*

1 : **function** *MREER (D, S, TR, TM, mSim, iter, Mappings)*
2*: SubSetAtt* ← *EnumSubSet(S);*
3*: DataDesc* ← *ReduceMergeAndCount(MapEntityV alues(D, SubSetAtt,* T_M *),* T_R*);*
4*: AttrFiltered* ← *filter(S,DataDesc);*
5*: BlockingKey* ← *chooseKey (AttrFiltered);*
6*: (AE,M,NAE)* ← *ReduceMatching(MapBlocking(BlockingKey,A,* T_M *),mSim,* T_R*);*
7*: Mappings* ← *Mappings* ∪ *M ;*
8: **if** *(iter > 0)* **then**
9: *FAE* ← *representantSelecting (AE);*
10*: NewDataset* ← *NAE* ∪ *FAE;*
11*: (newT*$_R$*, newT*$_M$ *)* ← *getTasks(*T_R*,* T_M*,NewDataset);*
12*: iter* ← *iter - 1;*
13*: MREER(NewDataset, S , newTR, newTM , mSim , iter, Mappings);*
14: **else**
15: **return** *Mappings ;*
16: **end if**
17: **end function**

Les lignes 4 et 5 correspondent à la phase de sélection de la clé de blocking. La ligne 6 correspond au processus Map Reduce qui effectue le calcul de comparaison des entités au sein de chacun des blocs préalablement constitués. La ligne 7 permet d'enrichir l'ensemble Mappings des nouvelles correspondances trouvées lors de la phase de matching. Les lignes de 8 à 14 correspondent à la préparation de l'itération suivante qui s'effectuera par l'appel récursif à la fonction MREER en ligne 14. Ligne 8 exprime la condition d'arrêt des appels récursif, en l'occurrence quand le nombre d'itération fixée par l'utilisateur est atteint. Cette préparation de l'itération suivante débute ligne 9. Les représentants d'entités sont sélectionnés pour chacun des groupes d'entités alignées selon la stratégie souhaitée (i.e. choix aléatoire, l'entité avec le moins d'attributs non valués, l'entité la plus similaire en moyenne). Ligne 10 permet de construire le nouvel ensemble fourni en entrée de l'itération suivante à partir des entités non précédemment alignées et les représentants d'entités sélectionnés dans l'ensemble FAE. Le nombre d'itération est ensuite décrémentée pour pouvoir lancer l'appel récursif de la fonction MREER.

5 Validation expérimentale

Dans cette section nous allons évaluer notre approche MREER en termes de qualité de l'alignement ainsi qu'en termes de performance. Pour ce faire, nous avons implémenté notre approche ainsi que la méthode PairRange en se basant sur la description de la méthode présentée dans [3]. L'objectif de l'implémentation de PairRange, est de nous permettre de comparer ses résultats avec les résultats de notre approche MREER dont l'objectif est d'améliorer la qualité de l'alignement à travers l'aspect itératif et le passage à l'échelle à travers l'adoption du principe de distribution des calculs basé sur le paradigme Map-Reduce.

En premier temps, nous allons décrire le protocole expérimental que nous avons utilisé pour valider et tester les deux approches. Ensuite nous présentons les différents résultats que nous avons obtenus pour les trois expérimentations que nous avons effectuées.

Dans la première expérimentation nous comparons l'approche MREER avec PairRange en termes de qualité d'alignement et en termes de performances. L'objectif principal de cette première expérimentation est bien de démontrer le passage à l'échelle de notre approche, en la comparant avec une approche distribuée reconnue comme obtenant de bonnes performances en l'appliquant sur des jeux de données volumineux [3]. Par contre, dans la deuxième expérimentation, nous avons déroulé deux processus d'alignement d'entités avec l'approche MREER. Chacun des processus d'alignement est constitué de deux itérations. L'objectif de cette deuxième expérimentation est de mettre en évidence l'importance de l'aspect itératif de notre approche dans la résolution du problème d'omission de comparaisons. Enfin, dans la troisième expérimentation, nous présentons les résultats de l'évaluation et de la comparaison des deux approches en termes de performances.

5.1 Protocole expérimental

Nous avons effectué nos tests sur un cluster du département constitué de 4 machines en utilisant Hadoop 1.7.0_51. Chaque machine du cluster dispose d'une RAM de 7.5 Go, d'un processeur doté de deux cœurs et d'un espace disque égal à 30 Go.
Nous avons utilisé deux *datasets* réels disponibles sur internet sur le site de l'université de Leipzig [10]. Ces jeux de données sont constitués d'ensembles réels d'entités portant sur des articles scientifiques.

Chacune des entités des deux *datasets* a la forme suivante : <id, title, authors, venue, year>. Le premier *dataset* est constitué d'environ 2 600 articles scientifiques issus de ACM. Par contre, le deuxième *dataset* est constitué d'environ 65 000 articles issus de DBLP.

Pour comparer deux entités données, nous calculons la similarité entre leurs titres, avec comme mesure de similarité utilisée, *Jaro* avec un seuil supérieur ou égal à 0.8.

Comme nous avons indiqué dans l'introduction de cette section, nous allons comparer notre approche MREER avec la méthode PairRange en termes de qualité de l'alignement et en termes de performance.

Les mesures que nous utilisons pour déterminer la meilleure méthode en termes de qualité de l'alignement sont le Rappel, la Précision et la F-mesure. Ces mesures peuvent être définis de la manière suivante :

$$\boldsymbol{Rappel} = \frac{\boldsymbol{NCCT}}{\boldsymbol{NCC}}$$ **Équation 2**

$$\boldsymbol{Pr\acute{e}cision} = \frac{\boldsymbol{NCCT}}{\boldsymbol{NCT}}$$ **Équation 3**

$$\boldsymbol{F} = \frac{2\times(\boldsymbol{Pr\acute{e}csion} \times \boldsymbol{Rappel})}{(\boldsymbol{Pr\acute{e}cision}+\boldsymbol{Rappel})}$$ **Équation 4**

Où : *NCCT* représente le nombre de correspondances correctement trouvées, *NCC* le nombre de correspondances correctes (fournies par l'expert) et *NCT* le nombre de correspondances trouvées.

Ainsi, le taux de rappel sera élevé si le nombre de correspondances trouvées par l'utilisateur est proche du nombre de correspondances présentes dans le liste des correspondances fournie par l'expert. Aussi, la précision est élevée, cela signifie que peu de correspondances inutiles sont proposées par le système et que ce dernier peut être considéré comme "précis". La F-mesure est la pondération des deux mesures précédentes. Ainsi, la méthode qui dispose de la F-mesure la plus élevée sera celle qui est jugée meilleur en termes de qualité.

Par contre pour évaluer la performance, nous utilisons comme mesures le temps de calcul et le nombre de comparaisons d'entités établies.

5.2 Comparaison de MREER et PairRange en termes de qualité de l'alignement

A travers cette première expérimentation, nous cherchons à évaluer et à comparer les deux approches MREER et PairRange en termes de qualité d'alignement matérialisée par la mesure du Rappel et de la Précision et la F-mesure pour chacune des deux méthodes.

Les figures 6 et 7 représentent respectivement la Précision et le Rappel obtenus pour chacune des méthodes MREER et PairRange pour trois processus d'alignement d'entités. Dans le premier processus, nous avons utilisé comme clé de *blocking* l'année, dans le deuxième nous avons utilisé la conférence, par contre dans le troisième, nous avons utilisée l'année et la conférence.

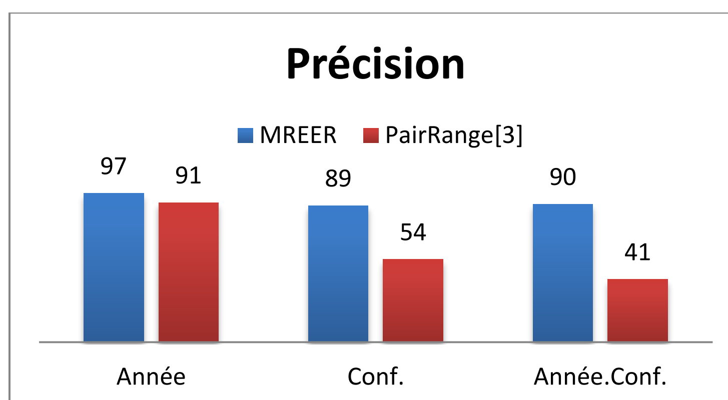

Fig. 6. La précision obtenue pour MREER et PairRange pour 3 processus d'alignement d'entités.

Dans la figure 6, l'axe des abscisses représente la clé de blocking adopté pour chacune des trois exécutions du processus de l'alignement et l'axe des ordonnées représente précision de l'alignement exprimée en pourcent.

A travers la figure 6, on constate que notre approche MREER est meilleure que PairRange en termes de Précision de l'alignement car pour les trois processus et pour des clés de blocking différentes on a toujours obtenu une précision supérieure à celle de PairRange. Pour la première exécution du processus d'alignement, l'écart entre les précisions des deux méthodes n'était pas très grand (97% pour MREER et 91% pour PairRange). Par contre, dans les deux autres exécutions, l'écart dépasse les 30% ce qui représente un écart significatif.

La supériorité de MREER en termes de précision de l'alignement par rapport à PaireRange, est due au fait quand dans MREER on diminue le nombre de comparaisons inutiles effectuées entre les entités de plus de 75% en moyenne (voir la troisième expérimentation) par rapport au nombre de comparaisons établies dans PairRange.

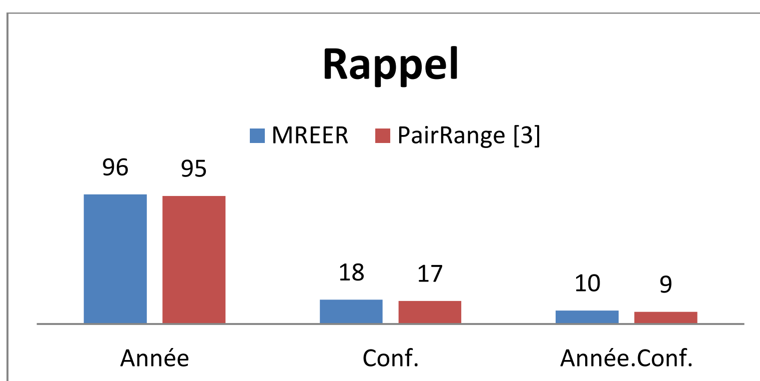

Fig. 7. Le Rappel obtenu pour MREER et PairRange pour 3 processus d'alignement d'entités.

A travers la figure 7 qui représente le Rappel obtenu pour les trois exécutions du processus d'alignement, on constate que les méthodes MREER et PairRange sont presque équivalentes en termes de Rappel. Cette équivalence est due au fait que dans les méthodes on obtient presque le même nombre de correspondances correctement alignées (NCCA).

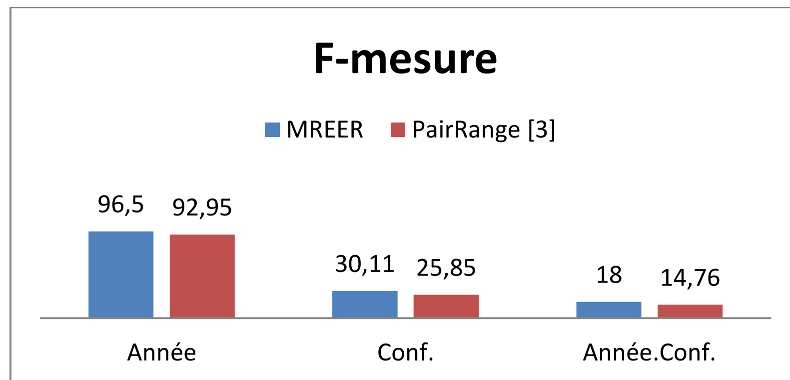

Fig. 8. La F-mesure obtenue pour MREER et PairRange pour 3 processus d'alignement d'entités.

Globalement, à travers cette première expérimentation, nous constatons qu'avec notre approche MREER, on obtient une meilleure qualité de l'alignement des entités (meilleure F-mesure) par rapport à PairRange.

5.3 Evaluation de l'aspect itératif de MREER

Le but de cette expérimentation est de montrer l'impact de l'aspect itératif de MREER sur le Rappel. Pour ce faire, nous avons déroulé deux processus itératifs d'alignement d'entités avec notre approche MREER, où chacun des processus est constitué de deux itérations. Pour le premier processus, la clé choisie pour la première itération est constituée de l'année et de la conférence. Par contre, pour la deuxième itération, l'année a été choisie pour être clé de blocking. Concernant le deuxième processus, dans la première itération, on a utilisé comme clé de blocking l'année et la conférence, et dans la deuxième itération, nous avons utilisé la conférence comme clé de blocking.

Le tableau 2 représente les résultats obtenus pour cette expérimentation.

Tableau 2: Le Rappel obtenu pour les deux processus d'alignement des entités

	Itération 1	Itération 2
Processus (Année.Conf ; Année)	10	95
Processus (Année.Conf ; Conf)	10	17

A travers les résultats obtenus dans le tableau 2, on constate que grâce à l'aspect itératif de l'alignement nous avons gagné en termes de Rappel. En effet, en sortie de la première itération des deux processus d'alignement que nous avons testés, le rappel était égal à 10%. Par contre, en sortie de la deuxième itération, le Rappel était égal à 95% et 17% pour le premier et le deuxième processus respectivement. Le gain en Rappel signifie que lors de la deuxième itération, nous avons pu détecter des correspondances qui n'ont pas été détectées (omises) lors de la première itération. Ce qui signifie que l'aspect itératif permet de remédier au problème de l'omission de comparaisons.

5.4 Comparaison de MREER et ParirRange en termes de performance

Dans cette partie, nous établissons une comparaison entre l'approche MREER et PairRange en termes de performances en se basant sur le nombre de comparaisons d'entités établies par chaque méthode et le temps de calcul, obtenus avec la première expérimentation.

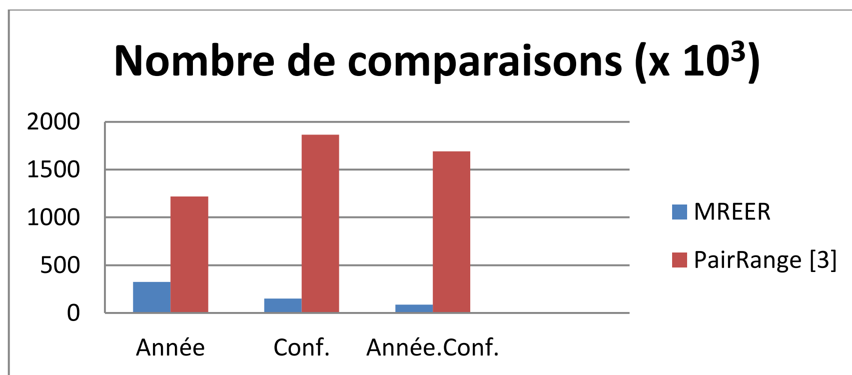

Fig. 9. Nombre de comparaisons obtenus pour les méthodes MREER et PairRange.

A travers les résultats de la figure 9, nous constatons que notre approche MREER est largement meilleure que la méthode PairRange en termes de performances, car en moyenne, notre approche permet d'effectuer 75% de comparaisons en moins par rapport à la méthode PairRange. La réduction du nombre de comparaisons, nous a permis de gagner aussi en termes de temps de calcul comme le montre la figure 10.

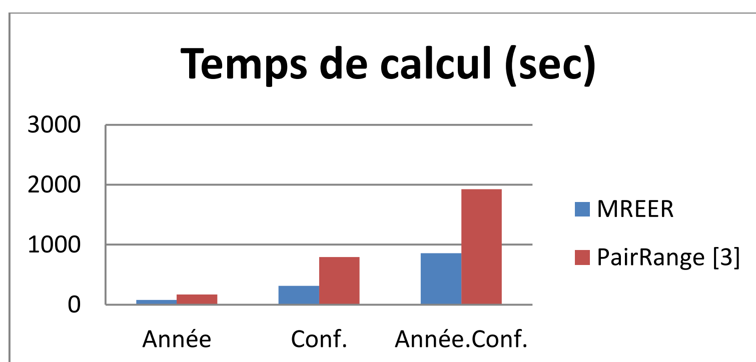

Fig. 10. Temps de calcul obtenus pour les méthodes MREER et PairRange.

6 Conclusion et perspectives

Dans cet article nous avons proposé une méthode d'alignement d'entités qu'on a nommée MREER. L'approche proposée est itérative et distribuée et basée sur le changement de la clé de blocking à chaque itération du processus d'alignement. Nous avons également définit un mécanisme de choix automatique de la clé de blocking basé sur une analyse du schéma des entités et sur des statistiques sur la distribution des entités.

Notre approche MREER ainsi qu'une méthode existante qui est PaiRange de L. Klob et al. [3] ont été implémentées et testées sur une grappe de machines. Les résultats des expérimentations ont montrés que MREER dépasse PairRange en termes de qualité de l'alignement. En plus de cela, il a été démontré également que MREER permet d'éliminer plus de 75% de comparaisons inutiles d'entités par rapport à PairRange, ce qui traduit la supériorité de notre approche en termes de performance. Une dernière expérimentation nous a montré l'impact de l'aspect itératif sur la qualité de l'alignement. En effet, il a été montré à travers cette expérimentation que l'aspect itératif avec le changement de la clé de blocking permettent d'améliorer la qualité de l'alignement à travers la détection de nouvelles correspondances parmi les entités qui n'ont pas été alignées lors d'une itération précédente.

Comme perspectives, nous envisageons dans un premier temps, faire des évaluations à plus large échelle, en déroulant nos tests sur des jeux de données plus volumineux et sur un plus grand cluster de machines. Nous envisageons également de mettre en place d'autres techniques de choix de la clé de blocking en se basant sur la structure du treillis.

Finalement, je tiens à préciser que cette expérience de stage de recherche m'a beaucoup aidée à m'améliorer sur le plan personnel ainsi que sur le plan professionnel. En effet elle m'a permis de collaborer avec des chercheurs brillants, ce qui m'a permis d'acquérir de nouvelles connaissances scientifiques et pédagogiques. J'ai bénéficié dans ce stage aussi pour améliorer mes compétences dans le domaine de calcul distribué en travaillant sur une thématique qui s'insère dans le cadre du BigData. Ma formation de Master 2 recherche IADE, m'a permis d'être rapidement opérationnel

sur quelques aspects de mon stage, et notamment en ce qui concerne la fouille de données.

References

[1] S. Melnik, H. Garcia-Molina and E. Rahm, “Similarity Flooding: A Versatile Graph Matching Algorithm and its Application to Schema Matching”, Proceedings of the 18th International Conference on Data Engineering, Page 117, 2002.
[2] V. Rastogi, N. Dalvi and M. Garofalakis, “Large-Scale Collective Entity Matching”, Proceedings of the VLDB Endowment, Homepage archive Volume 4 Issue 4, Pages 208-218, January 2011.
[3] L. Kolb, A. Tohr and E. Rahm, “Load-Balancing for MapReduce Entitiy Resolution”, Proceedings of the 2012 IEEE 28th International Conference on Data Engineering, Pages 618-629, Aug 2011.
[4] T. Kirsten et al. “Data Partitioning for Parallel Entity Matching”, PVLDB 3(1): 484-493 (2010).
[5] J. Dean and S. Ghemawat, “MapReduce : Simplified Data Processing on Large Clusters”. OSDI 2004: 137-150.
[6] Hadoop : Le Framework MapReduce Libre : https://hadoop.apache.org/
[7] H. Pasula, B. Marthi, B. Milch, S. Russell, and I. Shpitser. “Identity uncertainty and citation matching”. InNIPS, pages 1401–1408, 2002.
[8] P. Singla and P. Domingos. “Entity resolution with markov logic”. In ICDM, pages 572–582, 2006.
[9] J. Dean and S. Ghemawat, “MapReduce : Simplified Data Processing on Large Clusters”. OSDI 2004: 137-150.
[10] Benchmark datasets for entity resolution : http://dbs.uni-leizig.de/de/research/projects/object_matching/fever/benchmark_datasets_for_entity_resolution
[11] E. Ioannou, N. Rassadko, and Y. Velegrakis. On Generating Benchmark Data for Entity Matching. Journal on Data Semantics, 2(1):37–56, 2013.