Vers une solution de Blocking avec sources tierces pour l'Entity Resolution

MANIE Mohamed, Université Claude Bernard Lyon 1

TER encadré par Nicolas LUMINEAU et Fabien DUCHATEAU

Février 2014

Résumé : *L'entity resolution* permet de détecter des correspondances entre différentes entités équivalentes (qui représentent le même objet réel). Cependant, cette recherche a un coût en terme de mémoire et de temps important lorsqu'il s'agit de très grandes collections. L'interêt du *blocking* pour l'Entity *resolution* est de diminuer ce coût de traitement. Dans ce TER, nous proposons donc une solution pour cette étape de *blocking* qui réduit les coûts lors de l'exécution. De plus cette solution intègre le paradigme *MapReduce* pour être par la suite distribuée sur un cluster de machine.

Mots-clés : Entity resolution, Blocking, Matching, Mesures de similarité.

1) Introduction

L'entity resolution permet de découvrir des correspondances entre entités équivalentes. Par exemple, le processus *d'entity resolution* peut permettre entre autre de trouver des pages web qui traitent du même sujet sur internet, de trouver des photos qui contiennent un objet ou une personne précise dans un album ou bien encore de récupérer tout ce qui est en rapport avec le nom d'une personne dans un texte (nom, email contenant le nom, etc). Pour faire cela, on utilise de plus en plus du calcul distribué.

Actuellement, il existe différentes façons de faire de *l'entity resolution*. Une façon de faire simple mais très coûteuse consiste à faire la recherche de similarité entre chaque entité, ce qui revient à faire le produit cartésien de l'ensemble des entités avec lui-même. Une alternative proposée dans la littérature [ELS08] consiste à réaliser un algorithme de *blocking* dans le but de diviser l'ensemble des entités en différents groupes où s'effectuera un *matching* [RAS11] permettant de trouver des similarités à l'intérieur de ces groupes.

Dans le cadre de notre TER, nous nous intéressons à faire de *l'entity resolution* sur de grandes collections d'entités respectant un même schéma. Pour cela, nous testerons dans un premier temps différentes mesures de similarité pour en choisir une efficace dans notre contexte puis nous utiliserons un procédé *d'entity resolution* qui implémente l'étape de *blocking* en s'inspirant de *Map Reduce.*

Dans ce TER nous proposons une solution de *blocking,* en s'assurant qu'elle respecte le paradigme *Map Reduce*. Cette solution a été implémentée et elle a permis d'effectuer différents tests de performances. Dans le reste de ce rapport, la section 2 présente les pré-requis et les travaux connexes à notre proposition. Dans la section 3, nous présentons notre solution de *blocking*. Le détail des expérimentations et des comparatifs de performances sont présentés dans la section 4. La section 5 permet de conclure et de donner des perspectives à notre travail.

2) Pré-requis

2.1) Blocking

Le *blocking* consiste à séparer un très grand nombre d'entités en blocs (cf figure1). Les blocs sont définis en fonction d'un ou plusieurs attributs. Toutes les entités mises dans un bloc ont une valeur commune sur les attributs définissant les blocs. L'intérêt du *blocking* est de diviser les données en plusieurs blocs pour pouvoir exécuter un algorithme de comparaison ou autre défini par l'utilisateur, sur chacun d'eux dans un cluster de machines pour pouvoir augmenter les performances.

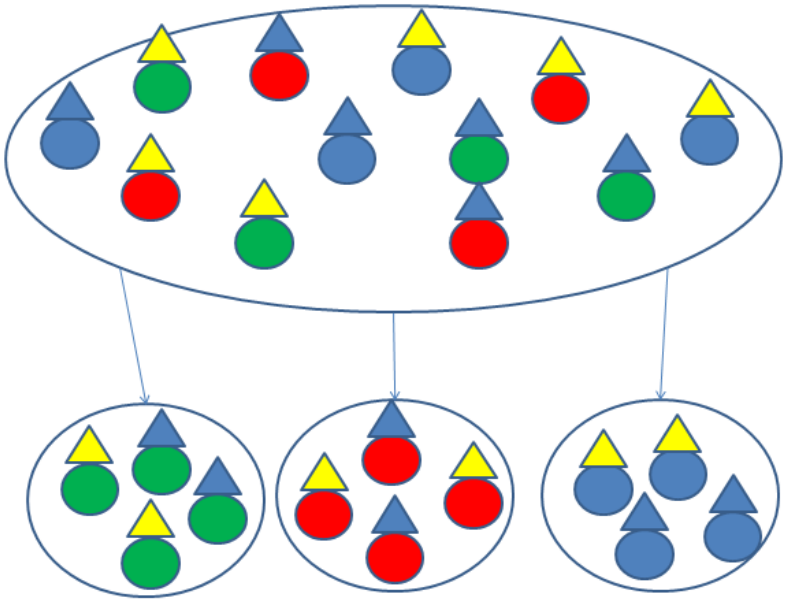

Figure 1 : Blocking effectué sur la couleur du rond

2.2) Matching

Le *matching* sert à déterminer si deux entités issues de sources différentes référencent le même élément/concept. Le processus de *matching* repose donc sur l'utilisation de mesure de similarité.

L'intérêt de ces mesures de similarité est d'avoir un résultat adapté à la comparaison de chaine de caractères. Nous pourrions utiliser une égalité totale avec la fonction **equals()** mais cela nous retournerait FAUX si les deux chaines testées ne sont pas parfaitement égales. Donc cela nous retournerait que les chaines « Maison » et « maison » sont différentes, alors que dans nos tests nous voulons dire qu'elles sont très proches. On peut donc supposer leur égalité. C'est pourquoi nous utiliserons des mesures de similarité qui retournent un score entre 0 et 1 pour pouvoir faire un choix (1 représente l'égalité parfaite).

Différentes mesures de similarité existent [EUZ04]. Dans ce travail de recherche nous faisons le choix de nous focaliser sur la librairie *SecondString* [COH03] largement utilisée par la communauté.

Nous avons choisi de tester les trois méthodes suivantes :

- La distance de **Jaro-Winkler** est la similarité entre deux chaines. Elle est principalement utilisée dans la détection de doublons. Plus la mesure de Jaro-Winkler entre deux chaînes est proche de 1, plus elles sont similaires. Cette mesure est particulièrement adaptée au traitement de chaînes courtes comme des noms ou des mots de passe. Le résultat est normalisé de façon à avoir une mesure entre 0 et 1, le zéro représentant l'absence de similarité.

  Distance Jaro-Winkler : $d_w = d_j + (\ell p(1 - d_j))$ avec dj la distance de Jaro, l la longueur du préfixe commun et p le coefficient permettant de favoriser les chaines avec un préfixe commun.

  Ex : La distance entre «idMaison » et « numMaison » = 0.805

- La distance de **Levenshtein** qui est une distance mathématique donnant une mesure de la similarité entre deux chaînes de caractères. Elle est égale au nombre minimal de caractères qu'il faut supprimer, insérer ou remplacer pour passer d'une chaîne à l'autre.

  Ex : La distance entre «idMaison » et « numMaison » = -3.0

- La distance de **Monge-Elkan** propose de suivre une méthode récursive pour la comparaison de deux chaines de caractères que l'on nommera s et t. La première étape consiste à séparer les chaines s et t en sous chaine s = a1 ……. aK et t = b1 ……. bL. Ensuite la similarité est calculée à l'aide de la fonction suivante : $sim(s,t) = \frac{1}{K}\sum_{i=1}^{K} max_{j=1}^{L} sim'(Ai, Bj)$.

  Avec sim' une fonction pour mesurer la distance entre deux chaines.

  Ex : La distance entre «idMaison » et « numMaison » = 0.75

2.3 MapReduce

Le principe du paradigme MapReduce est de diviser les données à traiter en partitions indépendantes, de traiter ces partitions de manière distribuées puis de combiner les résultats des traitements parallèles. La fonction *Map* consiste à découper puis à distribuer les différentes partitions de données constituées. Généralement, la fonction *Map* est réalisé cluster de nœuds (machines). Chaque nœud en réception se charge alors du traitement sur les données reçues. La fonction *Reduce* est l'opération inverse de la fonction *Map*. Elle consiste à récolter tous les résultats calculés en parallèle et les fusionner en un seul résultat. Chaque partition est composée d'un couple clé/valeur. La clé est utilisée lors de la fusion pour regrouper les valeurs qui vont ensemble. Toutes les valeurs associées à une clé sont donc réunies à la fin de la fonction *Reduce*.

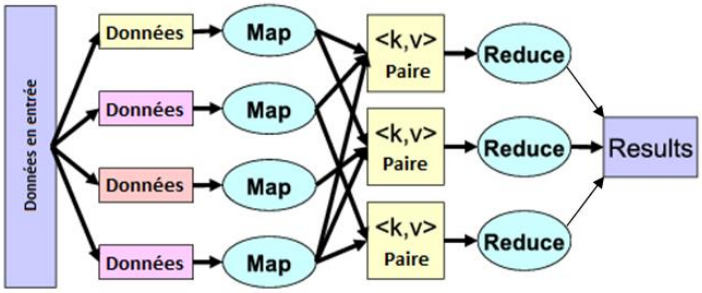

Figure 2 : Exemple du fonctionnement d'un MapReduce

3) Travaux connexes

Des travaux existant ont été réalisés sur le *blocking* et le *matching* de documents. Nous pouvons prendre par exemple une étude [ELS08] qui traite de comparaison entre différents documents.
Dans cette étude, un *blocking* est fait en indexant chaque terme du document en lui attribuant une clé et un couple (poids dans le document/identifiant du document). Suite à cette étape, les index des termes qui reviennent trop souvent sont supprimés pour optimiser le *blocking*.
Une autre étude [KOL12], montre quant à elle comment amélioré *l'entity resolution* avec une approche *Map Reduce.* Pour cela les données à traiter sont analysées puis distribuées en blocs, puis chaque bloc qui est trop grand subit à son tour la même opération.
La première étude [ELS08] ne correspond pas complétement à ce que l'on veut faire car nous nous traitons des entités et donc nous ne pouvons pas supprimer des termes ou attributs qui reviennent trop souvent pour améliorer notre *blocking.* Cependant, l'idée de la deuxième étude [KOL12], qui consiste à crée des blocs de tailles équilibrés est intéressante pour ce TER est peut-être utilisée pour proposer notre solution de *blocking*.

4) MREERaBLOC : *MapReduce* pour *Entity Resolution* avec *Blocking*

Pour notre première étape, chaque entité des collections qui doivent être testées vont faire une requête à la collection de référence pour recevoir une liste de toutes les entités qui possèdent l'attribut demandé. Dans notre cas, la collection de référence sera DBLP et les collections à tester seront ACM et SCHOLAR.

Les deux sources interrogent la collection de référence en envoyant les attributs (conférence, auteur de toutes leurs entités locales …) et obtiennent en retour des entités liées sur la collection de référence. Suite à cela, il y a création des groupes: un groupe par entité.

Exemple avec une requête sur l'année :

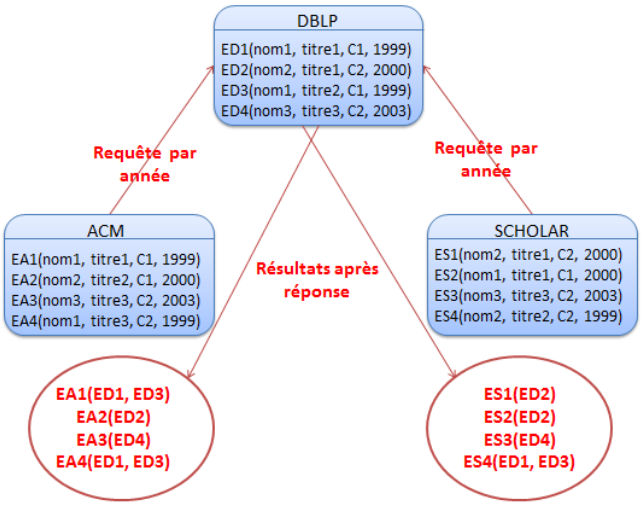

Figure 3 : Schéma représentant un exemple de blocking sur l'année

La Figure2 donne un exemple de *blocking* sur l'année. Nous voulons matcher 2 sources de données, ACM et SCHOLAR. Nous disposons d'une source hautement disponible, DBLP.

L'idée est donc de matcher d'abord ACM avec DBLP, SCHOLAR avec DBLP, et enfin d'en déduire les correspondances entre ACM et SCHOLAR.

Dans l'exemple, nous voyons que l'entité EA1 regroupe les entités tierces ED1 et ED3 car elles possèdent la même valeur pour l'attribut année. Nous faisons de même avec le reste des entités de ACM et SCHOLAR jusqu'à obtenir tous nos groupes.

La deuxième étape est de transformer ces groupes en blocs (cf Figure4) avec les entités de la collection de référence comme clé. Pour cela, nous récupérons la première entité d'un groupe qui devient une clé. Puis nous attribuons à cette clé un bloc qui contient toutes les clés des groupes auxquelles elle appartenait. On obtient donc :

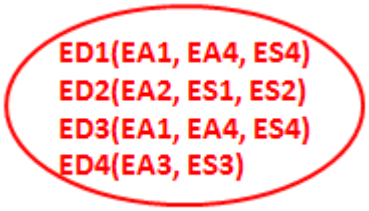

Figure 4 : Blocs obtenu après la seconde étape de notre blocking

Dans le pseudo code ci-dessous qui implémente le cas général (voir figure2), « CollRef » représente la collection d'entités de référence, Coll1 et Coll2 représentent les collections d'entités à tester. « Arg1 » et

« Arg2 » représente la position des attributs sur lesquelles seront faites les requêtes. « Seuil » correspond au seuil à partir duquel la condition sera acceptée et «methode » représente la méthode à utiliser (Jaro, Monge ou Levenshtein).

```
Fonction blocking(CollRef, Coll1, Coll2, Arg1, Arg2, seuil, methode)
   String s1, s2;

   Pour toutes les entités de Coll1
         Pour toutes les entités de CollRef
               S1 = valeur de Arg1 pour entité de Coll1 ;
           S2 = valeur de Arg1 pour entité de CollRef ;
           Si methode.Score(s1, s2) >= seuil
                     On ajoute l'entité courante de CollRef dans le bloc
                  ayant pour clé l'entité courante de Coll1 dans ListBloc.
               Fin Si
         Fin Pour
   Fin Pour

   Pour toutes les entités de Coll2
         Pour toutes les entités de CollRef
               S1 = valeur de Arg1 pour entité de Coll2 ;
           S2 = valeur de Arg1 pour entité de CollRef ;
           Si methode.Score(s1, s2) >= seuil
                     On ajoute l'entité courante de CollRef dans le bloc
                  ayant pour clé l'entité courante de Coll2 dans ListBloc.
               Fin Si
         Fin Pour
   Fin Pour

   Pour tous les blocs de ListBloc
         Pour toutes les entités contenues dans les blocs
                  Transformation des groupes d'entités en blocs avec une
               entité de CollRef pour clé et les stocker dans ListBloc2.
         Fin Pour
   Fin Pour
```

Algorithme1 : Algorithme de blocking proposé

A la fin de ce traitement, l'arrayList « ListBloc2 » sera composé de blocs contenant chacun des entités de Coll1 et Coll2 et pour clé une entité de CollRef. A partir de ce moment-là, le « matching » sera plus facile à réaliser car une sorte de prétraitement aura été effectué.

5) Expérimentation

Cette section «Expérimentations » est divisée en 2 parties. La première présente le protocole d'évaluation et la seconde les expériences.

5.1) Protocole d'évaluation

Nous ferons nos tests sur 3 collections d'entités : ACM, SCHOLAR et DBLP qui sera utilisé comme source tierce hautement disponible. Ces 3 collections suivent le même schéma : (id, title, authors, venue, year).

Dans un premier temps, nous testons le produit cartésien avec les mesures de similarités : JaroWinkler, MongeElkan, Levenshtein et avec 3 seuils d'acceptation (60%, 75% et 90%).

Cela a pour but de déterminer la mesure de similarité que nous utiliserons et de déterminer le seuil pour lequel nous acceptons une correspondance pour la suite des tests. Le choix devra tenir compte du temps d'exécution mais aussi des résultats obtenus.

Ensuite des tests avec notre approche de blocking sont effectués avec différents attributs pour les requêtes (l'année, la conférence puis les deux).

Les mesures prises en compte pour choisir la meilleure méthode seront le rappel, la précision et la f-mesure.

- Dans notre cas, le rappel représente le nombre de correspondances correctement trouvées divisé par le nombre de correspondances correctes (disponibles dans le *perfect mapping* qui contient les correspondances correctes entre les documents testés). Le taux de rappel sera élevé si le nombre de correspondances trouvées par l'utilisateur est proche du nombre de correspondances présentes dans le *perfect mapping*.

- La précision représentera le nombre de correspondances correctement trouvées divisé par le nombre de correspondances trouvées. Si elle 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.
  Sa formule est la suivante :

$$F = \frac{2 \cdot (\text{précision} \cdot \text{rappel})}{(\text{précision} + \text{rappel})}$$

5.2) Comparatifs de résultats

5.2.1) Qualité du « matching »

Nous illustrons dans les graphiques qui suivent, l'évaluation de ces 3 mesures sur les collections ACM et SCHOLAR. Pour cela, les mesures de similarités JaroWinkler, MongeElkan et Levenshtein seront utilisés avec 3 seuils différents.

La Figure5 représente la précision obtenue pour 3 seuils de similarité. L'axe des abscisses représente nos 3 seuils (60%, 75% et 90%) et l'axe des ordonnées représente la valeur de cette précision exprimée en pourcent. Nous observons que plus le seuil pour lequel nous acceptons une correspondance est élevé, plus la précision est élevée. Nous remarquons aussi que pour un seuil de 90%, toutes nos précisions sont à 100% mais pour un seuil de 60%, la précision pour la distance de Levenshtein est très mauvaise.

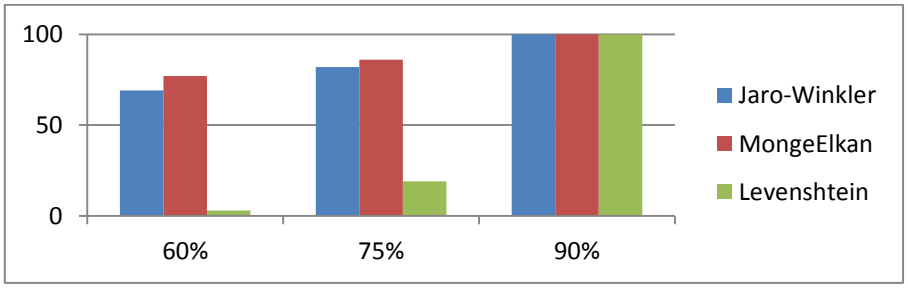

Figure 5 : Précision obtenue pour 3 seuils de similarité

La Figure6 représente le rappel obtenu pour 3 seuils de similarité. L'axe des abscisses représente nos 3 seuils (60%, 75% et 90%) et l'axe des ordonnées représente la valeur de ce rappel exprimée en pourcent. Nous observons que plus le seuil pour lequel nous acceptons une correspondance est élevé, plus le rappel est bas. Cela est dû au fait que plus le seuil est élevé, moins on trouve de correspondances. Nous

remarquons aussi que pour un seuil de 60% le rappel de la distance de Levenshtein est très élevé mais cela est dû au fait que presque toute les paires possible nous sont retournées.

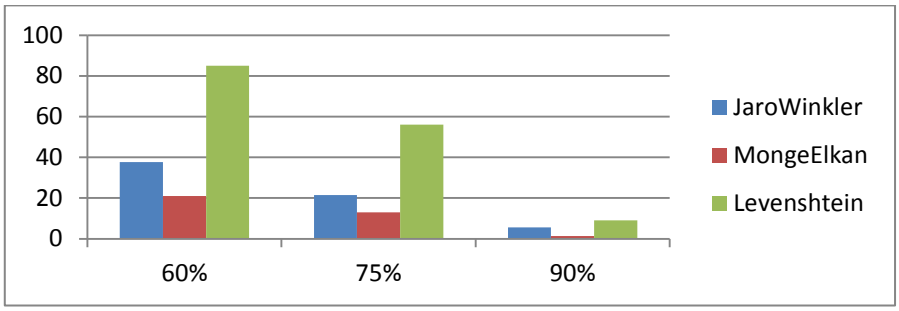

Figure 6 : Rappel obtenu pour 3 seuils de similarité

La Figure7 représente la F-mesure obtenu pour 3 seuils de similarité. L'axe des abscisses représente nos 3 seuils (60%, 75% et 90%) et l'axe des ordonnées représente la valeur de cette F-mesure exprimée en pourcent. Avec la mesure de compromis entre précision et rappel, on remarque que JaroWinkler est la meilleure mesure des 3 avec les seuils à 60% et 75%. Les 3 mesures ont des valeurs très faibles (<20%) avec un seuil à 90%.

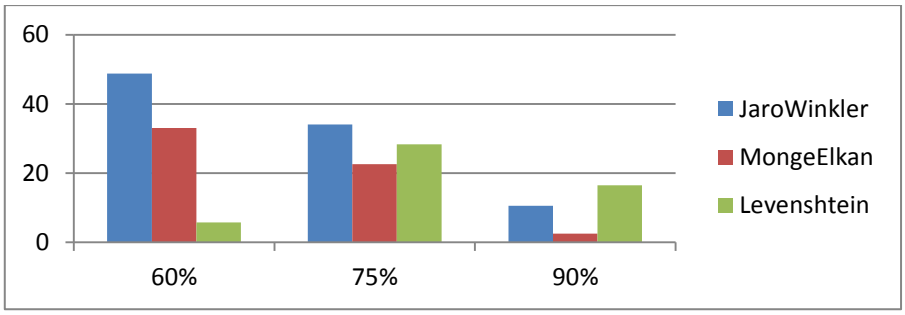

Figure 7 : F-mesure obtenue pour 3 seuils de similarité

La Figure8 représente le temps d'exécution du produit cartésien avec 3 méthodes différentes. On observe que pour MongeElkan ce temps est bien supérieur aux autres. Cela est dû au fait que cette mesure utilise deux autres mesures de similarité pour donner un résultat.

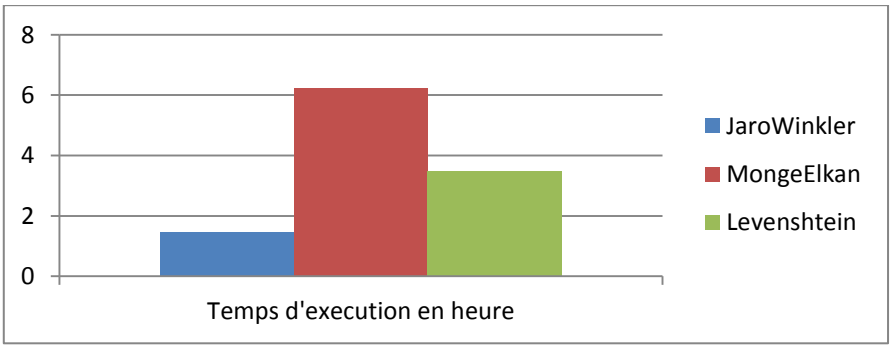

Figure 8 : Temps d'exécution du produit cartésien avec 3 méthodes différentes (en heure)

Nous observons que l'on obtient les meilleurs résultats avec la distance de Jaro-Winkler. En prenant un seuil de similarité à 75%, on obtient un rappel et une précision correcte mais surtout meilleur que les deux autres méthodes. Nous choisissons donc la distance de Jaro-Winkler et un seuil de 75 % pour faire notre *blocking*.

5.2.2) Comparatifs produit cartésien/blocking

La Figure9 représente le nombre de blocs obtenus en fonction des attributs requêtés.
Nous remarquons que le nombre de comparaison est le plus petit lorsque nous combinons notre requête sur 2 attributs (année et conférence).

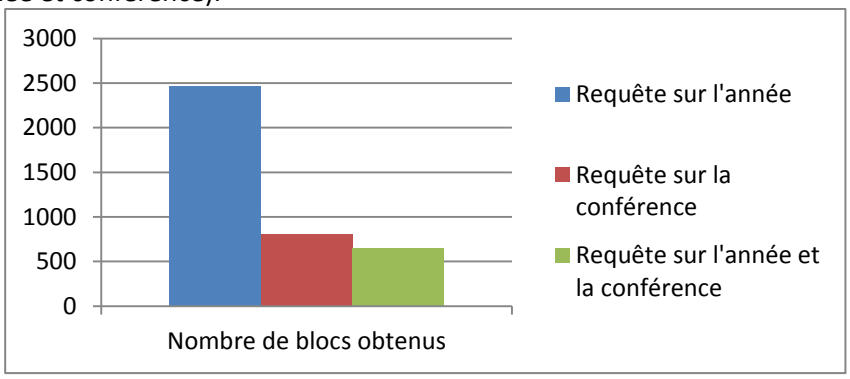

Figure 9 : Nombre de blocs obtenus en fonction des attributs requêtés

La Figure10 représente le nombre moyen d'entité par blocs en fonction des attributs requêtés.
Nous remarquons que ce nombre est faible lorsque nous combinons les 2 attributs pour notre requête. Ce nombre reste quand même conséquent lorsque la requête n'est faite que sur l'année.

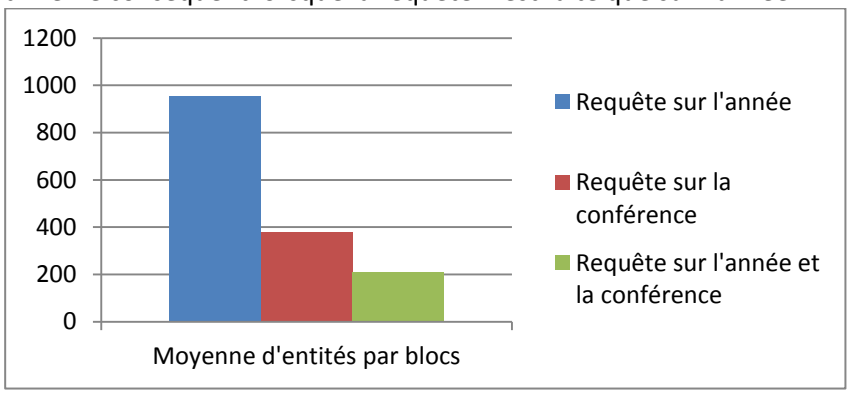

Figure 10 : Moyenne d'entités par blocs en fonction des attributs requêtés

La Figure11 représente le nombre de comparaison à effectuer en fonction des attributs requêtés ou du produit cartésien. Nous remarquons que le *blocking* diminue fortement les comparaisons à réaliser.
Cela devient très intéressant quand nous combinons nos 2 attributs pour la requête, le nombre de comparaison est presque divisé par 1000.

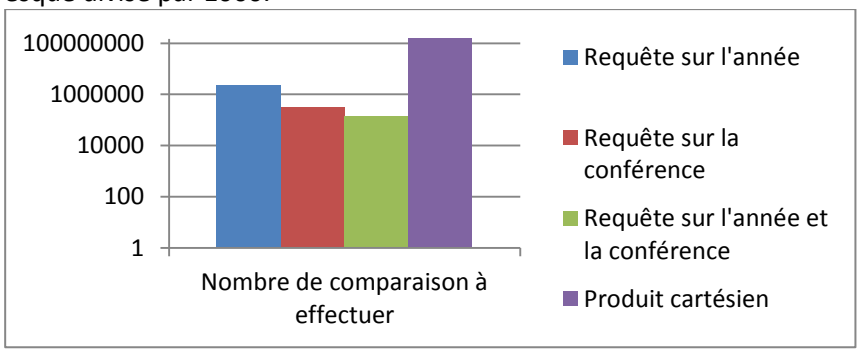

Figure 11 : Nombre de comparaisons à effectuer en fonction des attributs requêtés ou du produit cartésien

6) Conclusion

Dans ce projet nous avons vu comment réaliser un algorithme de *blocking* en respectant le paradigme *MapReduce.* Pour cela, nous faisons des requêtes vers une collection de référence pour pouvoir obtenir des blocs avec une valeur d'attribut en commun pour chaque entité le composant.

Pour faire le meilleur *blocking* possible nous devons choisir des méthodes pour minimiser le temps d'exécution, pour cela le nombre de comparaison doit être diminué au maximum. Aussi, nous devons faire attention au seuil choisi pour accepter une correspondance ou pas pour ne pas éliminer trop de correspondance avant la phase de *matching.*

Après ces étapes de *blocking* et de *matching*, nous aurions pu avec un peu plus de temps, utiliser ces processus à travers un cluster de machine pour mettre en application le paradigme *MapReduce*, nous pourrons aussi tester d'autres mesures de similarité pour tenter d'obtenir de meilleurs résultats.

Ce TER m'a permis de découvrir le monde de la recherche et d'être un peu plus regardant sur les coûts de traitement des programmes tout en m'apportant une première méthode pour minimiser ce coût.

7) Références

[COH03] William W. Cohen, Pradeep Ravikumar, and Stephen E. Fienberg. A comparison of string distance metrics for name-matching tasks. In *IIWEB*, pages 73-78, 2003.

[DEA04] Jeffrey Dean and Sanjay Ghemawat. MapReduce: *simplified data processing on large clusters.* In *OSDI'04: Proceedings of the 6th conference on Symposium on Opearting Systems Design & Implementation*, Berkeley, CA, USA, pages 10--10, 2004. USENIX Association.

[ELS08] Tamer Elsayed, Jimmy Lin, and Douglas Oard: *Pairwise Document Similarity in Large Collections with MapReduce.* Proceedings of the 46th Annual Meeting of the Association for Computational Linguistics (ACL 2008), Companion Volume, pages 265-268, June 2008, Columbus, Ohio.

[EUZ04] Jérome Euzenat. State of the art on ontology alignment. http://www.inrialpes.fr/exmo /cooperation/kweb/heterogeneity/deli/. August, 2004.

[KOL12] Lars Kolb, Andreas Thor, Erhard Rahm*: Load Balancing for MapReduce-based Entity Resolution*. ICDE 2012: 618-629.

[RAS11] Rastogi, Dalvi & Garofalakis. (2011). *Large-Scale collective entity matching. Proceeding of the Very Large Data Base Endowment*, 4(4), 208-218.