Calcul de l'intersection entre listes triées à base de sauts

Lougmiri Zekri*, Faïza Manseur**, Oussama Belhadj***

Université d'Oran1 Ahmed Ben Bella,
Faculté des Sciences Exactes, Département d'informatique
BP 1524, El-M'naouer, 31000, Oran, algérie
*lougmiri@gmail.com, **faiza_inf@hotmail.fr, ***vipvip215@gmail.com

Résumé. Dans les moteurs de recherche, la réalisation des index passe par le calcul de l'intersection entre les documents alors que l'évaluation des requêtes est le fruit de l'intersection entre ces requêtes et les index. Ce problème de calcul de l'intersection entre des ensembles triés a suscité beaucoup d'intention depuis 1971, date d'apparition du premier algorithme. L'optimisation du nombre de comparaisons et des temps de calcul est le principal objectif que les algorithmes doivent atteindre. Dans ce contexte, nous présentons un nouvel algorithme de type diviser-pour-régner pour le calcul de l'intersection entre des listes ordonnées. Le prétraitement sur les listes est présenté en détail. Il permet à notre algorithme d'éviter de comparer des parties qui ne seront pas forcément partagées par ces listes. Les expérimentations menées montrent que notre solution est performante.

1 Introduction

Le calcul de l'intersection entre listes ordonnées est à la base des processus de composition des index et l'évaluation de requêtes, principalement les requêtes conjonctives dans les moteurs de recherche. L'importance de ce calcul revient au fait que les systèmes informatiques actuels sont larges où la production des documents augmente en volume chaque jour. Le traitement des requêtes doit être aussi rapide que possible d'autant plus que le flux des messages circulant dans le système est important. Les moteurs de recherche doivent gérer des listes inversées immenses afin de répondre en quelques millisecondes à des milliers de requêtes. Le calcul de l'intersection date de 1971 avec les travaux de Hwang et Lin (1971, 1972). Les auteurs ont proposé et étudié un algorithme dit linéaire de fusion entre deux tableaux. Cet algorithme est dit aussi naïf dans la mesure où il scanne de façon séquentielle deux tableaux triés dans l'ordre croissant. Les tests se font sans index et sans la définition de mécanismes pouvant éviter des tests inutiles. Depuis, plusieurs travaux ont vu le jour. Certains travaux ont amélioré le temps moyen de l'intersection et d'autres ont implémenté des procédés qui exploitent les propriétés hard des machines sur lesquelles ils s'exécutent. Le calcul parallèle offert par les nouvelles technologies est aussi un moyen efficace pour calculer cette intersection.

La réduction du nombre de comparaisons et l'accélération de ces comparaisons sont le cheval de bataille de tous ces algorithmes et ces mécanismes. Considérons deux listes triées A et B de longueur n, et supposons que l'on veut localiser les éléments de la liste A dans la liste

B. Si on n'introduit pas le hachage, le nombre de comparaisons coûterait O(2n) comparaisons au pire des cas ; il y a exactement (2n-1) opérations de tests. Si on introduit le hachage alors on peut réaliser O(n) comparaisons. Réellement, cette dernière complexité est fausse, car il y a omission de l'opération du hachage des n éléments de la liste B. Donc si on opère n opérations de hachage et n opérations de tests on retrouvera le même nombre de comparaisons. D'une façon générale, nous considérons que nA et nB sont les longueurs respectives de A et B avec nA $\leq$ nB et supposons qu'on veut toujours localiser la petite liste A dans la grande liste B (voir Barbay et al. (2006, 2009)). Dans le but de réduire le nombre de comparaisons, nous proposons un mécanisme permettant de sauter dans les listes A et B lorsqu'on est sûr qu'une certaine séquence de nombres n'est pas partagée. Nous proposons dans le prétraitement de tri, un mécanisme de montage au sein des listes A et B afin d'accélérer la comparaison. Nous montrerons que ce procédé permet de détecter l'intersection dans des temps meilleurs. Comme Schlegel et al. (2011), nous considérons que les listes sont déjà chargées en mémoire. L'algorithme que nous proposons est appelé "Test With Jump", abrégé en TWJ.

Le reste de ce papier se présente comme suit. La section 2 donne un état sur les travaux reliés. Nous présentons les algorithmes Naïf, SvS et Baeza dans la section 3. La section 4 donne les détails de notre solution. La section 5 présente les résultats des expérimentations effectuées. Nous donnons une conclusion dans la section 6.

2 Travaux reliés à cette problématique

Le calcul de l'intersection entre listes est largement étudié vu son importance dans plusieurs domaines, comme la création des index dans les moteurs de recherche et la détection des motifs fréquents. Hwang et Lin (1971) ont proposé un algorithme linéaire pour la fusion de deux listes ordonnées. Ils proposent que si les longueurs des deux listes à comparer sont relativement égales alors une comparaison linéaire serait un excellent moyen pour la fusion alors que si la différence entre les deux longueurs est grande alors le recours, pour localiser les éléments de la petite liste dans la grande liste, à d'autres méthodes, comme la dichotomie ou l'interpolation, accélèrent la fusion. La complexité algorithmique de cet algorithme est donnée. Demaine et al. (2000) ont présenté un algorithme adaptatif pour le calcul de l'intersection. Ces mêmes auteurs ont prouvé que cet algorithme, selon un certain paramètre , est plus performant que l'algorithme standard SvS (pour Small vs Small) Demaine et al. (2001) lorsque les données sont extrêmement larges comme dans les machines de recherche. Les détails de SvS seront donnés dans la section suivante. Demaine et al. (2001) ont présenté un algorithme amélioré du précédent. Barbay et Kenyon (2002) ont présenté un nouvel algorithme avec seuil pour le calcul de l'intersection de plusieurs listes en parallèle. Baeza (2004), en s'inspirant de Bentley (1976), a introduit un nouvel algorithme pour le calcul de l'intersection. Nous référons cet algorithme au nom de son auteur. Baeza suppose deux listes A et B où la différence entre leurs deux longueurs est importantes. De façon récursive, en utilisant la dichotomie, il calcule la médiane de A et tente de la localiser dans B, à l'aide de la dichotomie aussi. S'il ne la trouve pas alors il récupère la position dans laquelle elle devrait se trouver. La médiane de la petite liste et le rang d'insertion de la médiane dans le grand ensemble divisant le problème en deux sous-problèmes. L'algorithme permet de résoudre de manière récursive les instances formées par chaque paire de sous-ensembles, en prenant toujours la médiane du petit sous-ensemble et la chercher dans le grand sous-ensemble. Si l'un des sous-ensembles est vide, il

ne retourne rien. Baeza-Yates et Salinger (2005) ont mené des expérimentations pour comparer Baeza avec d'autres algorithmes. Ils ont montré que si nA est trop petit comparativement à nB alors la dichotomie est un moyen efficace pour le calcul de cette intersection. Bille et al. (2007) ont utilisé le mot mémoire comme unité de représentation. Ils réalisent un prétraitement bien soigné pour écrire les listes dans la RAM. A l'aide d'une fonction de hachage bien définie, il calcule l'intersection entre deux listes. Ding et al. (2009) ont proposé une représentation hiérarchique en mémoire où ils couplent le hachage et le tri pour la détection de l'intersection. Le résultat dépend largement de la représentation en bits des mots mémoire. Schlegel et al. (2011) ont exploité les instructions STTNI (STring and Text processing New Instruction) définies dans le processeur d'Intel SSE 4.2. L'intersection est calculée selon le modèle SIMD sur ce processeur. Leurs expérimentations ont montré que la comparaison de deux tableaux selon des blocs de 8 bits était la meilleure façon pour atteindre des performances élevées. Lemire et al. (2016) ont proposé GALLOPING algorithme qui, à base du modèle SIMD, compare 4 paires d'entiers représentés en 32-bits. Cet algorithme dépend largement de l'architecture du processeur sur lequel il s'exécute. GALLOPING peut comparer des listes de longueurs non égales. Dans le cas contraire, ils utilisent un autre algorithme du type SIMD pour la détection de l'intersection. Tatikonda et al. (2009) ont exploité l'architecture multi-cœurs pour réaliser cette tâche. Zhou et al. (2016) ont proposé un framework efficace qui tourne sur la GPU où les auteurs exécutent un procédé intense à base du modèle SIMD pour réaliser un index pour un moteur de recherche. Les auteurs ont proposé aussi une fonction de hachage binaire pour représenter les termes. Ao et al. (2011) ont présenté des algorithmes parallèles qui exploitent les processeurs graphiques pour la réalisation de l'intersection. Tsirogiannis et al. (2009) ont proposé un partitionnement de l'entrée afin d'avoir un équilibrage de la charge sur une architecture multi-cœurs. Inoue et al (2014) ont proposé un framework qui tourne à base du modèle SIMD pour minimiser les mauvais branchements causés par l'opération de test. L'exécution du programme de test était suivie au niveau bas de la machine. Il s'agit de contrôler l'avancement du compteur ordinal du programme. Les auteurs ont proposé une prédiction en faisant avancer plusieurs pointeurs à la fois. Pugh (1990) a utilisé les skiplists pour représenter les données en mémoire et pour le calcul de l'intersection. Brown et Tarjan (1979) ont utilisé un arbre AVL pour déduire l'intersection avec la même complexité de Hwang et Lin (1972).

3 Les Principaux algorithmes

3.1 L'algorithme naïf

L'algorithme Naïf propose de représenter les données sur un espace linéaire. Il a été proposé par Hawang et Lin (1971, 1972). Cet algorithme est implémenté dans la librairie de C++ pour le calcul de l'intersection entre listes std : :set_intersection. L'algorithme Naïf se présente comme suit :

Cet algorithme prend en entrée les deux listes ordonnées A et B et se pointe vers la première position dans chacune d'elles. Si, à un moment donné, il y a une égalité entre les éléments actuels des deux listes alors l'élément partagé est enregistré sinon Naïf avance le pointeur de lecture dans la liste dont l'élément courant est le plus petit. L'arrêt est provoqué lorsque Naïf atteint la fin de l'une des deux listes au moins.

Algorithme naïf

```
entier intersection (entier[ ] A, entier[ ] B, entier tailleA,
entier tailleB, short* C)
{ entier compt = 0 ; entier i = 0, j = 0 ;
while(i <tailleA && j <tailleB)
{ if(A[i] == B[j]) {C[compt++]=A[i] ; i++ ; j++ ;}
else if (A[i] >B[j]) j++ ;

else i++ ;}
return compt ;}
```

On remarque que le résultat est une liste ordonnée. Un algorithme, tel que Baeza, n'assurant pas cette propriété, serait pénalisé si on l'utilise pour calculer l'intersection entre plusieurs listes, nous reviendrons à plus de détails dans ce papier. Naïf exécute (nA+nB-1) tests dans le pire des cas. Soit alors une complexité de O(nA+nB). Il est conseillé d'utiliser cet algorithme lorsque nA et nB sont presque égaux.

3.2 L'algorithme SvS

SvS (Small versus Small) Barbay et al (2006, 2009) est un algorithme simple et direct largement utilisé pour le calcul de l'intersection entre k listes triées. SvS a été proposé par Hwang et Lin (1971, 1972). Les listes ens[0], ens[1], ..., ens[k], forment l'entrée de SvS, et sont positionnées selon leurs longueurs. SvS compare celles ayant les deux plus petites longueurs. Le résultat sera comparé avec la liste suivante selon son positionnement. La petite liste est considérée la liste candidate. A chaque fois, SvS considère un élément de celle-ci et exécute une recherche binaire pour le localiser dans l'autre liste. Si la recherche n'est pas fructueuse alors il l'élague de la liste candidate. Durant toute l'exécution, SvS sauvegarde la position l de l'élément en cours de localisation pour poursuivre la recherche à partir de cette position l dans l'autre liste. SvS s'arrête lorsque la liste candidate est vide ou s'il épuise les k listes.

Demaine et al (2000) ont considéré la variante Swapping_SvS, où l'élément recherché est choisi dans l'ensemble contenant le moins d'éléments restants, au lieu du premier initialement le plus petit ensemble défini dans SvS.

3.3 L'algorithme Baeza

L'algorithme de Baeza-Yates et Salinger (2005, 2010) était initialement destiné à l'intersection de deux listes triées. Il prend la médiane de la petite liste et tente de la localiser dans la grande liste. L'élément est ajouté à l'ensemble de résultats s'il est présent dans la grande liste. La médiane de la petite liste et le rang d'insertion de la médiane dans le grand ensemble divisent le problème en deux sous-problèmes. L'algorithme permet de résoudre de manière récursive les instances formées par chaque paire de sous-ensembles, en prenant toujours la médiane du petit sous-ensemble et la recherche dans le grand sous-ensemble. Si l'un des sous-ensembles est vide, il ne retourne rien. Pour utiliser cet algorithme sur des instances avec plus de deux listes, Baeza Yates suggère de faire l'intersection des listes deux à deux, en croisant les plus petites listes en premier. La figure 1 montre les parties considérées à chaque étape.

Le pseudo-code peut être consulté dans Yates et al (2010). Comme l'algorithme d'intersection fonctionne pour les listes triées alors que le résultat de l'intersection ne sera pas trié, cet algorithme se trouve dans l'obligation de trier ce résultat avant de chercher l'intersection avec la liste suivante. Cette opération de tri supplémentaire rend Baeza inefficace lorsque le nombre de listes à comparer est trop élevé.

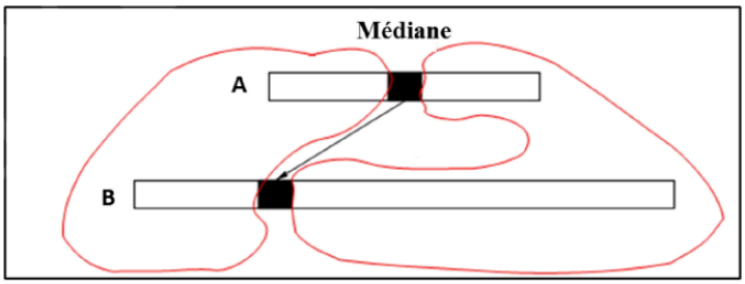

FIG. 1 – *Procédé de découpage dans Baeza Yates et al (2010)*

La comparaison de ces algorithmes se fait selon deux métriques au moins ; à savoir, le temps d'exécution et le nombre d'accès aux listes. Les accès aux listes donnent la complexité de ces algorithmes. Il est démontré théoriquement que leur complexité tourne autour de O(nA*log 2 (nB)). Sauf que ces algorithmes fonctionneraient mal si nA=nB. De point de vue temps d'exécution, il faut étudier la manière d'exécution de ces algorithmes. SvS et Baeza utilisent la dichotomie pour localiser les éléments de façon récursive. On sait que durant le calcul récursif, les résultats intermédiaires sont sauvegardés dans le tas (heap). Donc, si le tas est réduit et si les données sont volumineuses comme dans les moteurs de recherche, alors Baeza et SvS agiraient faiblement sur ces données. Donc un retardement de l'exécution est palpable. A la différence de Naïf et SvS, Baeza retourne un résultat non trié. L'ajout d'une procédure de tri consommera un temps supplémentaire.

En ce qui concerne l'algorithme Naïf, le pire des cas donne une complexité O(nA+nB). Ce cas se produit lorsque l'intersection est vide alors que le meilleur des cas pour cet algorithme donne une complexité O(min (nA,nB). Il faut remarquer que dans le domaine de la recherche d'information, où les listes inversées des index sont trop longues, voire impossible à scanner, les algorithmes auront un comportement asymptotique. C'est pour ce constat que nous considérons le pire des cas. Si le meilleur des cas se produit, tous les algorithmes peuvent avoir un comportement idéal sinon acceptable.

4 L'algorithme Test With Jump TWJ

La solution que nous proposons vise à minimiser le nombre de comparaisons (les tests) effectuées autant que possible. Notre idée est de créer une nouvelle structure à partir des ensembles qu'on souhaite croiser. Dans cette structure on va diviser les tableaux initiaux en fragments appelés séquences. Chaque séquence est précédée par deux champs servants d'identifiants à celle-ci.

SeqID : est l'identifiant de la séquence. Dans le cas des tableaux de chaines de caractères, l'identifiant est la première lettre de chaque séquence. Dans le cas des tableaux d'entiers, cet identifiant est le quotient de la division entière des valeurs du tableau sur la valeur du séquencement (on va détailler sur des exemples dans la partie qui suit).

Nombre de cases : C'est le nombre d'éléments contenus dans une séquence. Ces champs aideront à prévoir le nombre de cases à sauter pour atteindre la prochaine séquence, d'où le gain en nombre de comparaisons entre éléments des deux tableaux comparés.

4.1 Création des séquences

Les séquences doivent être créées d'une manière bien étudiée pour profiter des avantages. La création des séquences diffère selon les types des tableaux :

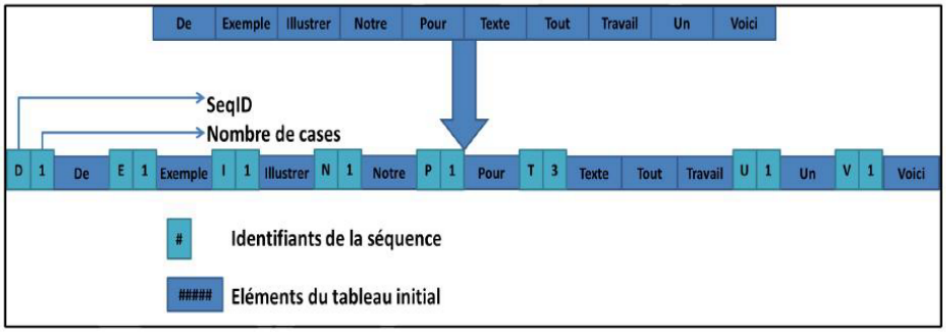

FIG. 2 – *Création de séquences pour un tableau de chaines de caractères*

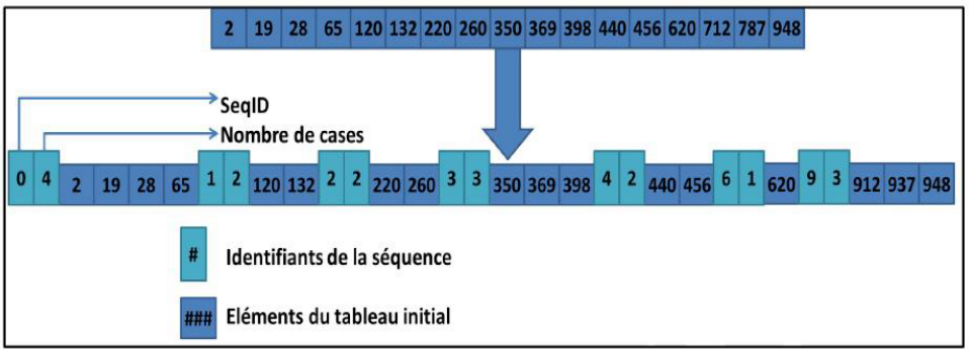

FIG. 3 – *Exemple de création de séquences pour un tableau d'entiers*

Tableaux de chaines de caractères : Dans ce cas de figures, les termes sont triés dans l'ordre lexicographique. Dans chaque séquence, on met la suite de termes commençant par la

même lettre et SeqID recevra cette lettre comme identifiant de séquence. Sur l'exemple de la figure 2, le texte "Voici Un Exemple Pour Illustrer Tout Notre Travail" est mis dans un tableau puis trié. La restructuration donne huit séquences ainsi que les nombre de termes de chaque séquence.

Tableaux d'entiers : Dans ce cas, on utilise une valeur de séquencement. Cette valeur est obtenue d'une manière expérimentale et varie selon la distribution des valeurs, la valeur minimale et la valeur maximale dans le tableau. Chaque séquence contient toutes les valeurs comprises entre (SeqID * valeur de séquencement) et ((SeqID+1) * valeur de séquencement). A titre d'exemple, si la valeur de séquencement est égale à 1000, on obtiendra des séquences 0, 1000, 2000, 3000 etc. La figure 3 présente un exemple avec une valeur de séquencement égale à 100.

4.2 L'algorithme TWJ

L'algorithme Test With Jump TWJ prend en entrée deux listes triées transformées en séquences avec une valeur de séquencement bien spécifiée et retourne les éléments communs entre elles. Les éléments de l'ensemble d'intersection retournés sont eux même triés. L'algorithme TAS traite les séquences comme étant des éléments d'un tableau trié et procède à une recherche comme celle de l'algorithme Naïf entre les séquences de chaque tableau et entre les éléments des séquences possédants le même SeqID. Il compare itérativement deux SeqID de chaque liste, en commençant par la première case des deux ensembles. Chaque fois que les deux SeqID sont égaux, il passe à chercher l'intersection entre ces deux séquences. Quand il atteint la fin d'une des deux séquences comparées, il passe à la prochaine séquence de chaque liste et fait le même traitement à nouveau pour ces deux séquences. Si les valeurs des SeqID comparés sont inégales, alors il passe à la séquence suivante dans l'ensemble ayant le SeqID inférieur et la compare avec la séquence actuelle. L'algorithme se termine lorsqu'il n'y a plus de séquences dans l'un des deux ensembles. L'algorithme TWJ est présenté par le pseudo code précédent pour deux ensembles.

Algorithme TWJ

```
{
Intersection_GTWJ(entier[] SeqA, entier[] SeqB, entier[] C ){
Entier i = 0 ; Entier j =0 ; cp =0 ;
tant que (j <TailleSeqB){
si (SeqIDAi == SeqIDBj) alors{
tant que (i <NSeqAi && j <NSeqBj){
si (SeqA[i] == SeqB[j]) alors{ i++ ; j++ ;C[cp++] = SeqA[i] ;}
sinon si (SeqA[i] <SeqB[j]) alors i++ ;
sinon j++ ; } }
sinon si (SeqIDAi <SeqIDBj) alors i = i+SeqIDAi ;
sinon j = j+SeqIDBj ;}
renvoi C ;}
}
```

La figure 4 présente un exemple de déroulement de TWJ avec 10 comparaisons entre les éléments et 9 comparaisons entre les SeqID (soit 19 en tout) et 12 éléments évités. Naïf effectuera 26 comparaisons pour les mêmes entrées. Le point fort de l'algorithme TWJ vient du fait qu'il a la possibilité d'éviter des séquences, comme les sections : 3, 5, 6, 7, 8 et 9 et des cases, comme dans les deux sections 0 et 1.

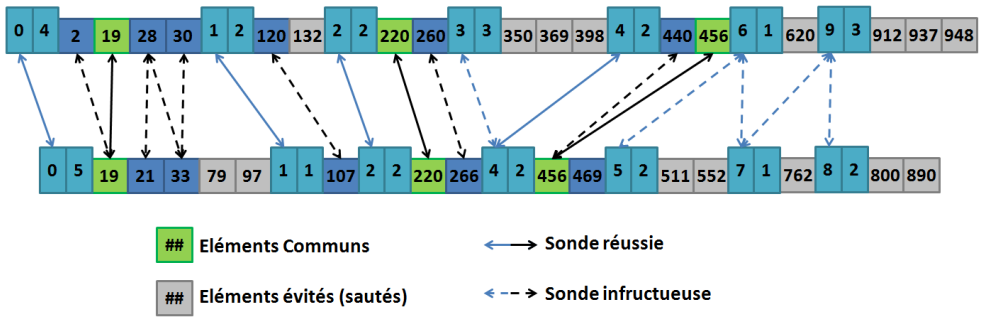

FIG. 4 – *Exemple de déroulement de l'algorithme TWJ sur des entiers positifs*

4.3 Analyse du nombre d'opérations

Comme nous l'avons cité dans la section précédente, l'algorithme Naïf exécute (nA+nB) opérations de comparaisons au pire des cas. Selon le découpage que nous effectuons sur les tableaux, nous avons : chaque tableau A est sectionné en NSeqA sections et chaque section contiens NSeqi éléments donc : $\sum_{i=1}^{N} seqANseq_i$.

Puisque TWJ a la possibilité d'exécuter des sauts dans un tableau A, soit alors NsautA le nombre de cases évitées. D'où lorsque TWJ compare deux tableaux A et B, il aura la possibilité de sauter NS = NsautA + NsautB. Il résulte donc que TWJ exécute (nA+nB-NS) opérations de comparaison. Ce qui donne que TWJ est borné par Naïf, autrement dit : nA+nB-NS $\leq$ nA+nB.

Nous évitons l'utilisation des restrictions sur les longueurs des listes. Selon Yates et al (2010), si les écarts sont trop grands alors la dichotomie serait un excellent moyen pour calculer l'intersection. On sait que (nA+nB) est le nombre d'opérations exécutées par Naïf alors que (nA*log2nB) est le nombre d'opérations exécutées si on adopte la dichotomie. Avoir (nA+nb)<(nA*log2nb) =>(nA<(log2nB/(log2nB-1)).

Cette restriction ne peut être garantie dans le cas général lorsqu'on l'on veut réaliser des index larges comme dans les moteurs de recherche. L'algorithme que nous proposons est de la même complexité que Naïf, puisqu'il est impossible de calcul NS en fonction des tailles des tableaux en entrée. Dans la partie expérimentation, nous montrerons que TWJ est plus rapide que Naïf lorsqu'il y a effectivement des parties communes entre les tableaux d'entrée, comme lors de la composition des index dans les moteurs de recherche. L'expérimentation rend en fait un temps moyen.

Nous rappelons qu'il s'agit dans ce contexte de la comparaison entre listes triées ; donc le prétraitement lui-même ne rentre pas dans les temps de calcul. Ce procédé est courant dans les différents domaines de recherche.

5 Expérimentation

Nous avons expérimenté les algorithmes sur une machine dotée d'un processeur Intel ® Core i3, à 2.30 GHz avec un cache de niveau L1=32 ko, L2= 265 ko et L3=3Mo et 4 Go de Mémoire RAM sous Windows 7 64 bits. Le programme est implémenté en Java sous l'environnement NetBeans 8.0.2.

5.1 Méthode et données des expérimentations

Nous comparons la performance des algorithmes, TWJ, Naïf, SvS et Baeza, sur des paires d'ensembles triées générées de la même manière que dans Yates et al (2004). Nous générons des séries de nombres entiers aléatoires uniformément répartis dans l'intervalle [1, 10^7]. On peut varier l'intervalle de génération librement. Les algorithmes de ce type de domaines sont très rapides de façon qu'il est difficile de toucher à la différence entre leurs performances à petite échelle, il s'agit de temps en microsecondes. Pour réaliser une comparaison correcte, nous avons fait tourner ces algorithmes 100 fois sur des tailles de tableaux variables, puis nous avons calculé la moyenne des temps d'exécution pour chacun en microsecondes. Cette procédure a la faculté de faire éviter les piques aléatoires de l'exécution. Ces piques sont dues à la mise et à la libération du cache du système.

Comme méthodologie des expérimentations, nous avons varié la taille du premier tableau jusqu'à une taille maximale avec un saut prédéfini, puis nous avons varié la taille du deuxième tableau jusqu'à une taille maximale. Enfin, nous avons forcé le contenu du plus petit tableau de façon à rendre ses valeurs égales ou plus petites que celles du deuxième.

Les deux premiers points nous permettent de toucher à l'impact de la variation des longueurs des tableaux sur les mesures de comparaison. Remarquons qu'à chaque exécution, c'est le tableau de la plus petite longueur qui sera comparé à celui ayant la plus grande. Le troisième point permet de montrer que Naïf et TWJ agissent de façon meilleure que SvS et Baeza-Yates lorsque les tableaux possèdent la même longueur. Le quatrième point permet de montrer que TWJ agit plus efficacement selon le contenu (distribution des valeurs à l'intérieur du tableau) que les autres algorithmes.

5.2 Variation de la taille des séquences

L'utilité de cette expérimentation est d'observer les changements de la valeur de séquencement dans TWJ. Cette opération permet de donner la meilleure valeur de séquencement. Nous avons fixé la taille du premier tableau à 1000 et celle du deuxième à 100000. La valeur de séquencement varie de 250 à 10000 avec un pas égal à 250. Les résultats de cette expérience sont présentés dans les Figures 5, et 7.

La Figure 5 montre que plus la valeur de séquencement est petite plus les temps d'exécution augmentent. Cette figure montre que le meilleur séquencement varie entre 1000 et 2000. On remarque aussi que lorsque la séquence est élevée alors les temps d'exécution commencent à s'élever. La Figure 6 justifie bien le résultat obtenu dans la figure précédente, l'allure des deux courbes est semblable. On remarque alors que lorsque les séquences sont trop petites ou trop larges, les nombres de comparaison augmentent. La valeur 1500 était le meilleur compromis. Pour le reste des expérimentations, nous avons considéré l'intervalle [1000, 2000] car ses résultats étaient prometteurs.

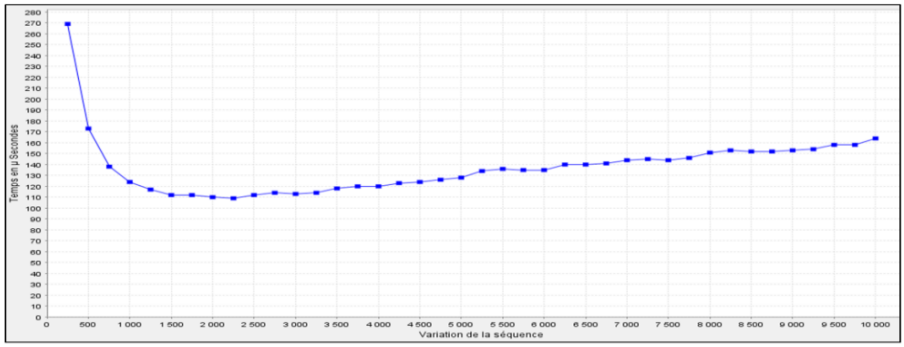

FIG. 5 – *Les temps d'exécution en variant la séquence*

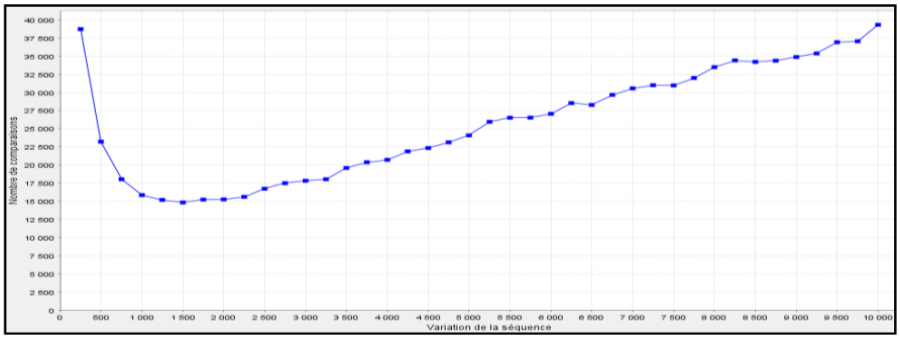

FIG. 6 – *Les comparaisons effectuées par TWJ selon la taille de la séquence*

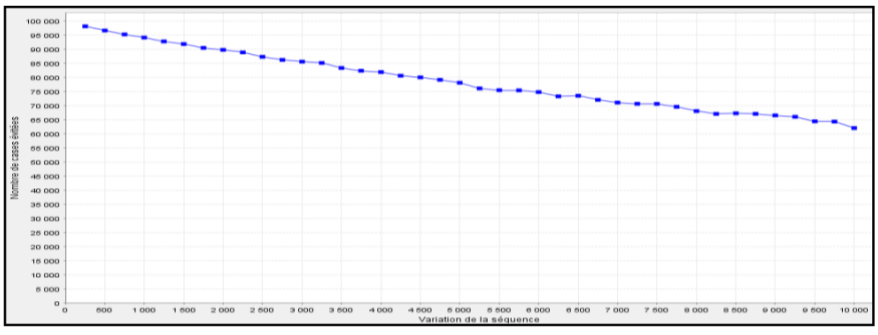

FIG. 7 – *Les nombres de cases évitées par TWJ en variant la taille de la séquence*

La Figure 7 illustre la courbe de NS le nombre de cases sautées. On remarque alors que plus la taille des séquences augmente, moins on évite des cases à comparer. Ceci confirme l'interprétation précédente ; plus la valeur de séquencement augmente plus notre programme se rapproche vers Naïf.

5.3 Variation de la taille du premier tableau

Dans cette partie on va comparer TWJ avec les autres algorithmes selon les nombres de comparaisons et les temps d'exécution.

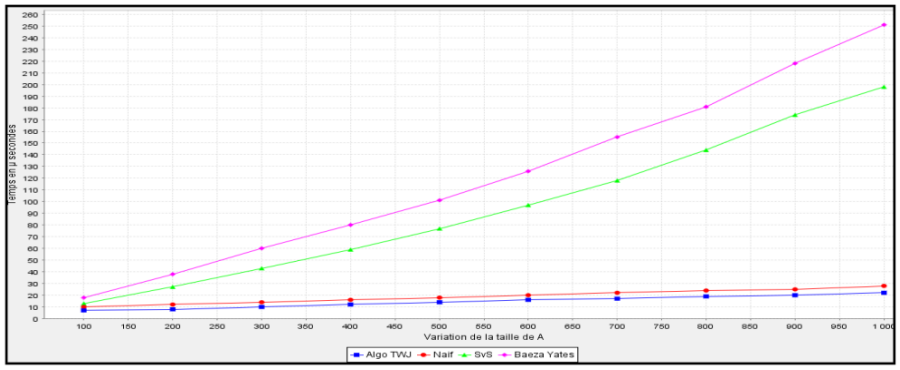

FIG. 8 – *Les temps d'exécution en variant la taille du premier tableau*

Comme un premier test, on va varier la taille du premier tableau de 100 à 1000 avec un pas de 100. La taille du deuxième tableau est fixée à 1000 et la valeur de séquencement pour TWJ est fixée à 1000. Cette étude nous permettra de toucher à l'influence de la variation des tailles des tableaux d'entrées sur les algorithmes étudiés. Les figures 9,10 et 11 montrent les résultats obtenus.

D'après les courbes de la Figure8, les temps d'exécutions de TWJ et Naïf sont proportionnels avec les nombres de comparaisons de chacun. Ils ont donné de meilleurs temps d'exécution par rapport à SvS et Baeza. On note aussi que plus la taille du premier tableau augmente plus le temps d'exécution du SvS et Baeza augmente, et ceci est expliqué dans la littérature du fait que ces deux algorithmes fonctionnent mieux quand la taille du premier tableau est considérablement inférieure à la taille du deuxième tableau. L'augmentation du temps d'exécution de Baeza et SvS est directement liée à la manière avec laquelle s'exécute l'algorithme de dichotomie. Cet algorithme est récursif d'où il passe un temps considérable à l'empilement des valeurs dans le tas (Heap) d'exécution. Donc l'entassement et le dépilement des valeurs augmentent leurs temps d'exécution. Cette figure montre que TWJ a fourni une meilleure performance que les autres algorithmes. Pour les nombres de comparaisons, la Figure 9 montre que Baeza a effectué moins de comparaisons pour les tableaux dont les longueurs sont inférieures ou égales à 400. Dépassant cette valeur, TWJ met moins de d'opérations. Pour renforcer notre étude, nous avons répété la même expérimentation en fixant la taille du deuxième tableau à 10000 au lieu de 1000. Le résultat est présenté dans la Figure 10.

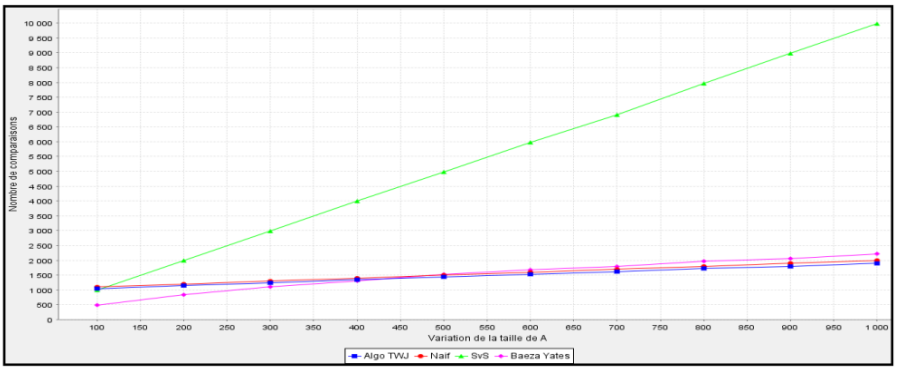

FIG. 9 – *Nombres de comparaisons en variant la taille du premier tableau*

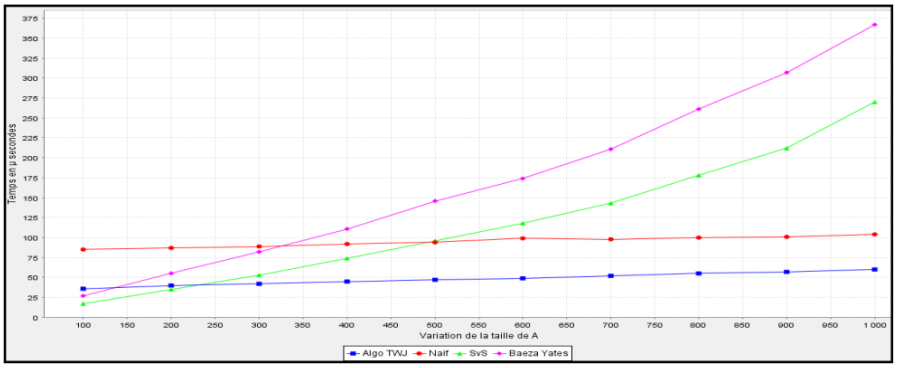

FIG. 10 – *Les temps d'exécution en augmentant la taille du 2ième tableau*

Les temps de ces exécutions sont donnés en microsecondes comme dans tous les travaux de références. La Figure 10 montre que pour de petites valeurs du tableau candidat, SvS consomme moins de temps que les autres. Les deux courbes de SvS et Baeza prennent la même allure puisqu'elle utilise la dichotomie dans leur code. Pour le cas de Naïf, on remarque qu'il commence sa bonne performance à partir de 500 éléments alors que TWJ a présenté la meilleure performance. Si Baeza est pénalisé c'est à cause de la restitution du contexte de l'exécution qui consomme son temps d'exécution. Baeza lui-même a présenté des pires cas desquels pourrait souffrir son algorithme (voir Baeza et al (2010), page 50). Baeza est conçu pour présenter un meilleur cas moyen.

On note sur la Figure 11 que Baeza a exécuté un nombre moindre de comparaisons mais sa courbe a tendance à augmenter en fonction de la taille du plus petit tableau. SvS a démarré avec un nombre minimal, puis ce nombre a commencé à s'élever de façon à avoir la plus mauvaise performance. On remarque que TWJ a exécuté plus d'opérations mais sa courbe n'a pas fait

de piques. Si on agrandit encore plus les tailles il est sûr qu'il va être meilleur que les autres à partir d'un certain seuil. On rejoint alors le résultat de la Figure 9.

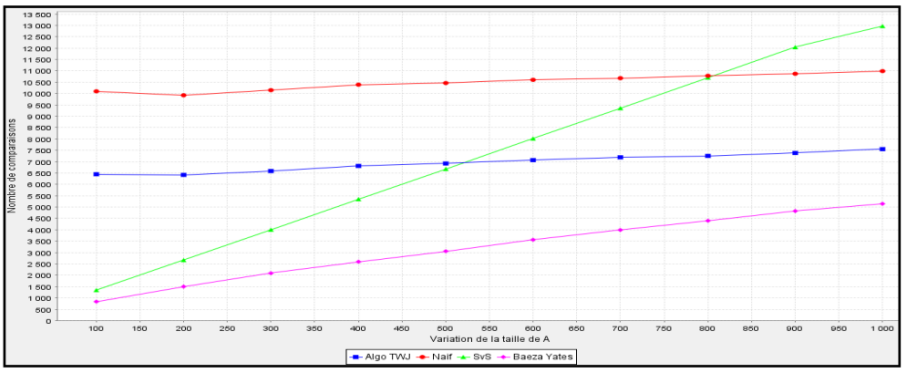

FIG. 11 – *Les nombres de comparaisons en augmentant la taille du 2ième tableau*

5.4 Redistribuer les valeurs dans les tableaux

Les résultats précédents nous ont guidés à penser à la redistribution des valeurs dans les tableaux. A cet effet, nous avons forcé le contenu du plus petit tableau de façon à rendre ses valeurs plus petites ou égales à celles du second. On suivant le même procédé des expérimentations, nous avons obtenus les résultats figurants sur les Figures 12 et 13.

Les résultats de cette expérimentation ont révélé que les temps d'exécution de TWJ et Naïf ont diminué considérablement par rapport à la même expérimentation sur des intervalles de valeurs égales. Les courbes du SvS et de Baeza ont subi des changements légers ; ceci revient au fait que ces deux algorithmes sont sensibles à la taille des entrées. Ces résultats indiquent que la distribution des valeurs sur les tableaux joue un rôle important pour l'exécution du Naïf et TWJ d'où notre algorithme TWJ fonctionnera mieux quand les valeurs ne sont pas répartis uniformément. TWJ arrête l'exécution très tôt puisque la valeur maximale du premier tableau pourrait être touchée très tôt, encore plus, elle pourrait appartenir à une séquence non partagée avec le deuxième tableau. La Figure12 montre un résultat que nous cherchions, nous remarquons que SvS a présenté une meilleure performance que Baeza. Ce cas de figure est constaté dans (Barbay et al (2009).

La Figure 13 montre que les nombres de comparaison des trois algorithmes Naïf, TWJ et Baeza ont diminué après la réduction des valeurs des contenus des tableaux. Quand les valeurs d'un tableau sont inférieures à celle du deuxième tableau, Naïf et TWJ ont tendance à terminer rapidement, c'est pour cette raison qu'on note que leurs nombres de comparaisons ont diminué rapidement. La courbe de Baeza a diminué aussi rapidement contrairement à son temps d'exécution, on peut expliquer ça du fait qu'il a fait beaucoup de divisions en sous tableaux (comparaisons sur les deux côtés des tableaux initiaux) ce qui le rend lent en termes de temps d'exécution.

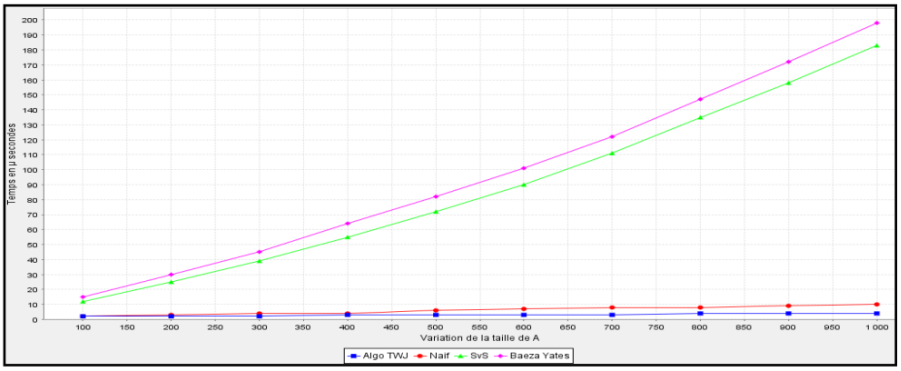

FIG. 12 – *Les temps d'exécution en limitant la valeur maximale du premier tableau*

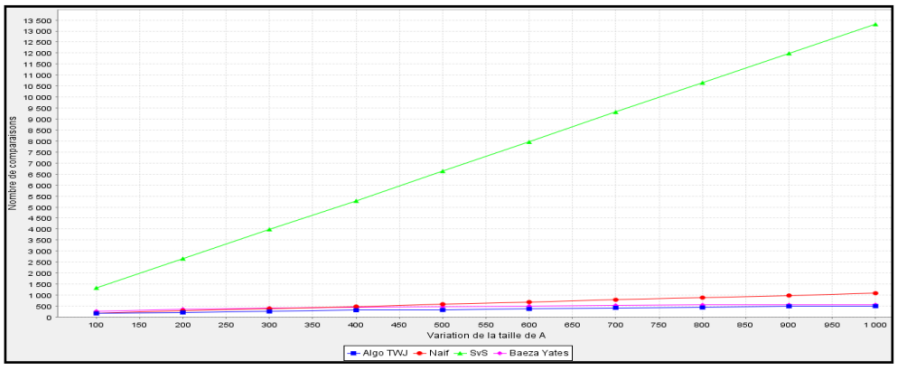

FIG. 13 – *Les comparaisons en limitant les valeurs maximales du premier tableau*

6 Conclusion

Les moteurs de recherche sont au cœur de l'internet aujourd'hui. Ils jouent un rôle important dans notre vie actuelle. Cette notoriété est due à l'incroyable effort pour évaluer des milliers de requêtes par minute. Afin de retourner des réponses, les moteurs de recherche manipulent les index, les données et les requêtes. Les index sont obtenus en calculant l'intersection entre les documents alors que les réponses sont obtenues en comparant les requêtes aux index ou aux documents. Ce qui s'impose donc c'est qu'il faut implémenter des algorithmes efficaces pour calculer ces intersections efficacement. Dans ce contexte, nous avons présenté l'algorithme TWJ. Nous proposons un certain découpage des listes. Ces découpages permettent d'éviter de tester des cellules inutiles. Nous avons présenté aussi les algorithmes qui forment l'état de l'art. Ils sont commentés en détails. Les expérimentations ont porté sur

des données synthétiques. Nous avons pris les temps de réponses et le nombre de comparaisons comme métriques de comparaison. Ces expérimentations ont montré que notre solution a présenté de meilleures performances comparativement aux autres algorithmes. Actuellement, nous travaillons sur la parallélisation de TWJ sur une architecture GPU.

Références

Ao, N., F. Zhang, D. Wu, D. S. Stones, G. Wang, X. Liu, J. Liu, et S. Lin (2011). Efficient parallel lists intersection and index compression algorithms using graphics processing units. *Proceedings of the VLDB Endowment 4*(8), 470–481.

Baeza-Yates, R. et A. Salinger (2005). Experimental analysis of a fast intersection algorithm for sorted sequences. In *International Symposium on String Processing and Information Retrieval*, pp. 13–24. Springer.

Baeza-Yates, R. et A. Salinger (2010). Fast intersection algorithms for sorted sequences. In *Algorithms and Applications*, pp. 45–61. Springer.

Barbay, J., A. López-Ortiz, et T. Lu (2006). Faster adaptive set intersections for text searching. In *International Workshop on Experimental and Efficient Algorithms*, pp. 146–157. Springer.

Barbay, J., A. López-Ortiz, T. Lu, et A. Salinger (2009). An experimental investigation of set intersection algorithms for text searching. Volume 14, pp. 7. ACM.

Bille, P., A. Pagh, et R. Pagh (2007). Fast evaluation of union-intersection expressions. In *International Symposium on Algorithms and Computation*, pp. 739–750. Springer.

Brown, M. R. et R. E. Tarjan (1979). A fast merging algorithm. *Journal of the ACM (JACM) 26*(2), 211–226.

Demaine, E. D., A. López-Ortiz, et J. I. Munro (2000). Adaptive set intersections, unions, and differences. In *In Proceedings of the 11th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA*. Citeseer.

Demaine, E. D., A. López-Ortiz, et J. I. Munro (2001). Experiments on adaptive set intersections for text retrieval systems. In *Workshop on Algorithm Engineering and Experimentation*, pp. 91–104. Springer.

Ding, S., J. He, H. Yan, et T. Suel (2009). Using graphics processors for high performance ir query processing. In *Proceedings of the 18th international conference on World wide web*, pp. 421–430. ACM.

Hwang, F. K. et S. Lin (1971). Optimal merging of 2 elements with n elements. *Acta Informatica 1*(2), 145–158.

Hwang, F. K. et S. Lin (1972). A simple algorithm for merging two disjoint linearly ordered sets. *SIAM Journal on Computing 1*(1), 31–39.

Lemire, D., L. Boytsov, et N. Kurz (2016). Simd compression and the intersection of sorted integers. *Software : Practice and Experience 46*(6), 723–749.

Pugh, W. (1990). A skip list cookbook. Technical report.

Schlegel, B., T. Willhalm, et W. Lehner (2011). Fast sorted-set intersection using simd instructions. In *ADMS@ VLDB*, pp. 1–8.

Tatikonda, S., F. Junqueira, B. B. Cambazoglu, et V. Plachouras (2009). On efficient posting list intersection with multicore processors. In *Proceedings of the 32nd international ACM SIGIR conference on Research and development in information retrieval*, pp. 738–739. ACM.

Tsirogiannis, D., S. Guha, et N. Koudas (2009). Improving the performance of list intersection. *Proceedings of the VLDB Endowment 2*(1), 838–849.

Zhou, J., Q. Guo, H. Jagadish, W. Luan, A. K. Tung, Y. Yang, et Y. Zheng (2016). Generic inverted index on the gpu. *arXiv preprint arXiv :1603.08390*.