Algorithme distribué pour l'extraction des fréquents maximaux

Nicolas Hanusse, Sofian Maabout, Radu Tofan

Algorithme distribué pour l'extraction des fréquents maximaux

Nicolas Hanusse[1] and Sofian Maabout[1] and Radu Tofan[1]

[1] *LaBRI, CNRS & INRIA (Cepage), Université Bordeaux 1 33400 Talence, France*

L'extraction des ensembles fréquents maximaux est un problème clef en fouille de données. Nous présentons dans cet article un algorithme distribué qui réalise cette tâche. Il s'agit du premier algorithme distribué avec des garanties de performance prouvées théoriquement.

Keywords: calcul distribué, fouille de données

1 Description du problème traité

Soit $\mathbb{I} = \{i_1, \ldots, i_N\}$ un ensemble d'objets (nous parlerons aussi d'items). Soit $\mathcal{T} = \{I_1, \ldots, I_M\}$ un multi-ensemble avec $I_j \subseteq \mathbb{I}$ un ensemble d'objets (nous l'appellerons aussi *itemset*). $\mathcal{T}$ est appelée *base de transactions*. Soit $I \subseteq \mathbb{I}$ un itemset, $|I|_{\mathcal{T}}$ désigne le nombre d'occurences de I dans $\mathcal{T}$. On notera simplement $|I|$ quand $\mathcal{T}$ est fixée. $t = |\mathcal{T}|$ désigne le nombre d'éléments de $\mathcal{T}$. La *fréquence* $f(I)$ de I dans $\mathcal{T}$ est égale à $\frac{|I|}{|\mathcal{T}|}$. Un item I est *s-fréquent* si $f(I) \geq s$. L'objectif de ce travail est de proposer un calcul distribué des éléments maximaux appelés aussi bordure :

Définition 1 (Fréquent maximal)**.** *I est un itemset s-fréquent* maximal *si aucun de ses sur-ensembles n'est s-fréquent. La bordure $B(s)$ contient tous les itemsets s-fréquent maximaux.*

Exemple 1. *Soit $\mathbb{I} = \{A,B,C,D\}$. Soit $\mathcal{T} = \{ABD, AB, ABD, CD, AD\}$. AD est $2/5$-fréquent car $f(AD) = 3/5$ alors que CD ne l'est pas puisque $f(CD) = 1/5$. On a que $B(2/5) = \{ABD\}$.*

On rencontre le problème du calcul des itemsets fréquents dans divers domaines : par exemple, on peut être intéressé par les chemins entre routeurs (items) les plus utilisés en considérant pour tout routage (transaction) de la liste des routeurs (itemsets) traversés. De manière générale, l'extraction des itemsets fréquents est extrêmement coûteux car une bordure peut atteindre 2^N (si $s = 0$ mais pas seulement). Aussi, concevoir un algorithme d'extraction des fréquents nous oblige à tenir compte de deux ecueils qui sont le temps pour calculer les fréquences (la table $\mathcal{T}$ peut être parcouru plusieurs fois) ainsi que la quantité de mémoire utilisée à chaque itération. Pour illustrer notre propos, considérons les deux variantes suivantes :

1. Pour chaque $I \in \mathcal{T}$, pour chaque $J \subseteq \mathbb{I}$, si $J \subseteq I$ alors incrémenter le nombre d'occurences de J.
2. Pour chaque $J \subseteq \mathbb{I}$, pour chaque $I \in \mathcal{T}$, si $J \subseteq I$ alors incrémenter le nombre d'occurences de J.

La première variante consiste à parcourir une seule fois $\mathcal{T}$, par contre elle nécessite de garder en mémoire un compteur pour chacun des 2^N itemsets. La deuxième variante, quant à elle, parcourt $\mathcal{T}$ 2^N fois par contre elle n'a besoin que d'un seul compteur à la fois en mémoire. Dans les deux cas, le temps de calcul de fréquence est cependant $\Theta(2^N M)$. Comme $\mathcal{T}$ est généralement de taille trop grande pour être entièrement chargée en mémoire, il faut en limiter les parcours pour réduire les accès disque très gourmands en temps. Par ailleurs, les itemsets *candidats* peuvent être trop nombreux, il faut limiter leur nombre lors d'un même parcours. Nous sommes donc en présence de deux contraintes contradictoires.

Pour limiter le nombre de candidats à tester lors de chaque parcours, les solutions proposées utilisent la propriété *d'antimonotonicité* de la fréquence : si I est fréquent alors tous ses sous-ensembles sont fréquents, et par conséquent, si I n'est pas fréquent alors tous ses sur-ensembles ne le sont pas. En procédant par

niveaux (un niveau correspond au nombre d'items dans un itemset), on peut utiliser cette propriété pour ne pas tester les itemsets pour lesquels on a trouvé lors des phases précédentes qu'un de leurs sous-ensembles n'est pas fréquent. C'est le principe de l'algorithme *A Priori* proposé dans [AS94]. A partir des maximaux, on peut générer tous les fréquents. Comme les maximaux sont moins nombreux, il est intéressant de considérér des algorithmes spécialement conçus pour ces derniers. Plusieurs algorithmes ont été proposés dans la littérature [Bay98, GZ05, BCG01]. Tous essayent d'exploiter au mieux l'antimonotinicité de la fréquence, i.e ne pas tester un itemset si un de ses sous-ensembles n'est pas fréquent **et** ne pas tester un itemset si un de ses sur-ensembles est fréquent.

2 Algorithmes existants et contribution

2.1 Hypothèses et modèles

Un maître répartit le calcul d'une bordure sur k processeurs esclaves $p_1, p_2, \ldots, p_k$ de manière synchronisée : à chaque ronde, il échange des informations : une liste des itemsets candidats dont la fréquence est à calculer, etc. Dans une ronde, chaque processeur esclave ne fait qu'un parcours de la table $\mathcal{T}$. Nous supposons que chaque processeur possède une copie de la table $\mathcal{T}$.

2.2 Comparaison

Les différents algorithmes existants et notre contribution sont résumés dans le tableau 1. Les mesures de performance sont :

- la mémoire requise pour chaque processeur (maître et esclaves) ;
- le nombre de rondes exprimant le nombre de parcours de tables par un processeur ;
- le temps $T = T_e + T_m + T_c$ correspondant au : (1) temps T_e de calcul des fréquences fait par les *esclaves* exprimé par le produit le nombre de lignes lues par le coût de calcul par ligne ; (2) temps de calcul du *maître* T_m de composition des résultats partiels en résultat global - principalement calcul de moyennes ; (3) temps de communications T_c exprimé en taille de messages échangés.

Il est à noter que jusqu'a présent, aucun modèle de temps n'avait été proposé dans la littérature. Notre proposition permet de comparer d'un point de vue théorique les différents algorithmes. De ce point de vue, APriori peut sembler aussi compétitif que MAFIA ou MAX-Miner mais ces deux derniers algorithmes s'avèrent bien plus performants en pratique car ils utilisent des heuristiques dont une analyse de pire cas (comme l'analyse présente) ne peut tenir compte.

Algorithme	Mémoire	ronde	T_e	T_m	T_c
APriori [AS94]	$O(2^N/\sqrt{N})$	N	$O(t2^N)$		
MAFIA/GenMAX [BCG01, GZ05]	$O(2^N/\sqrt{N})$	$N/2$	$O(t2^N)$		
PickBorders [HMT09]	$O(b)$	$O(2^N)$	$O(2^N t)$		
Parallel/MAX-Miner [CL08]	$O(2^N/\sqrt{N})$	$N/2$	$O(\frac{2^N t}{k})$	$O(2^N)$	$O(2^N)$
Notre contribution	$O(b)$	$2N$	$O(\frac{2^N t}{k})$	$O(Nb)$	$O(2^N)$

TAB. 1: Contribution ($b = |B(s)|$)

Pour indication, les complexités des versions séquentielles sont indiquées en supposant que tout le calcul est fait par un processeur. Les versions parallèles existantes (principalement Parallel MAX-Miner) partent du principe que chaque processeur est responsable d'une proportion $1/k$ de $\mathcal{T}$ et que le maître s'occupe de composer les résultats. Les inconvénients sont multiples : les quantités de calculs effectués et la taille des messages échangés sont indépendantes de la taille de la bordure. Nous avons proposé dans [HMT09] l'algorithme PickBorders qui tient compte de la taille de la bordure, qui est donc plus performant dès que $b = |B(s)| = o(\frac{2^N}{N})$. Cependant, PickBorders n'est pas aisément parallélisable et nécessite une re-écriture totale. Nous donnons quelques indications sur la conception de Parallel PickBorders. Par manque de place, nous ne pouvons donner l'algorithme complet.

3 Calcul des bordures

Nous présentons tout d'abord quels sont les calculs de fréquences nécessaires pour déterminer la bordure puis l'ordonnancement des itemsets à calculer en parallèle.

3.1 Calculs de fréquences

Nous supposons que les items sous ordonnés par un ordre total $<_{\mathbb{I}}$ et que les itemsets sont ordonnés lexicographiquement (ordre du "dictionnaire") avec l'ordre $<_{lex}$. Dans ce qui suit, nous supposons que $A <_{\mathbb{I}} B <_{\mathbb{I}} C <_{\mathbb{I}} \ldots$. Nous pouvons définir l'arbre lexicographique des itemsets avec les relations suivantes pour un itemset I :

- Fils(I) contient les itemsets sur-ensembles de I de cardinalité $|I|+1$ et plus grand lexicographiquement ;
- Pere(I) contient l'itemset sous-ensemble de I de cardinalité $|I|-1$ et plus petit lexicographiquement.

En résumé, l'algorithme PickBorders revient à parcourir l'arbre lexigraphique à l'aide d'un parcours en profondeur en sautant/élaguant les sous-arbres lorsqu'on peut déterminer que les itemsets correspondants ne sont pas fréquents. A chaque instant, la bordure contient la liste des itemsets maximaux visités. Si l'itemset courant n'est pas maximal au moment de son traitement, il est nécessaire de calculer sa fréquence. L'avantage de cette méthode est que la mémoire requise ne sert qu'à stocker la bordure qui ne peut que croître avec le temps. La partie la plus coûteuse pour le calcul de $\mathcal{B}$ est l'exécution de la fonction **frequence**(**I**) qui nécessite le parcours de $\mathcal{T}$. Nous avons mis en évidence le nombre maximum de fois que cette fonction est appelée.

Lemme 1. *La fonction* **frequence** *est appelée au maximum* 2^N *fois.*

La figure ci-dessous illustre l'exécution de cet algorithme lors du calcul d'une bordure $B(3/7)$ pour la table de transactions $\mathcal{T} = \{CDE, ABDE, ABE, ABCE, DE, CE, ABE, CDE\}$. On a calculé la cardinalité de 14 itemsets (en dehors de la racine dont la taille est connue) sur les 32 pour obtenir la bordure $\{ABE, CE, DE\}$. Pour certains itemsets, il n'est pas nécessaire de calculer la fréquence (élagage ou étant sous-ensemble d'un élément de bordure comme AE ou B).

L'algorithme précédent présente l'avantage de réduire le nombre de parcours fait sur $\mathcal{T}$ mais ne calcule à chaque fois qu'une seule fréquence.

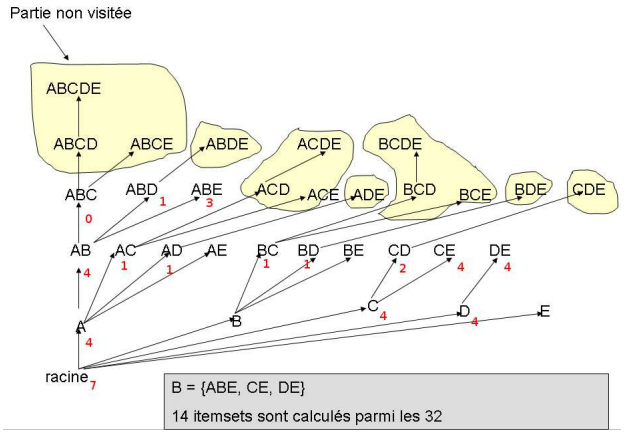

FIG. 1: Arbre lexicographique élagué (cardinalité indiquée en rouge).

ronde	**itemsets**	*nb itemsets*
0	$\emptyset$	1
1	A	1
2	AB	1
3	ABC, B	2
4	$ABCD, AC, BC$	3
5	ABD, ACD, BCD, C	4
6	AD, BD, CD	3
7	D	1

FIG. 2: Ordonnancement des calculs d'itemsets

3.2 Version distribuée

Dans notre version distribuée, le maître affecte à chaque ronde une liste d'itemsets pour chaque processeur dont la fréquence est à calculer. Pour cela, nous définissons la fonction `round`(I) qui détermine la

ronde dans laquelle un itemset $U = u_1u_2 \dots u_k$ avec $u_i <_I u_{i+1} \in I$ peut être potentiellement calculée. Nous devons d'abord définir la fonction `rank`. Pour chaque item u_i de I, $\texttt{rank}(u_i) = j$ si u_i est le j-ième élément de I. Pour chaque itemset $U = u_1u_2 \dots u_k$, $\texttt{round}(u) = 2\texttt{rank}(u_k) - k$.

Plus précisément, à la fin de la ronde R, le serveur est capable de calculer la frontière $B_R(s)$ pour chaque ensemble d'éléments dont la ronde est inférieure ou égale à R. Au début de la ronde $R+1$, le serveur possède une liste d'éléments à considérer (ceux de ronde $R+1$ dont le calcul a été décidé dans les rondes précédentes). Pour un élément de ronde $R+1$, nous avons deux cas : soit il existe un élément de $B_R(s)$ qui est un sur-ensemble et l'itemset courant ne peut être maximal (et il n'y a pas besoin de le calculer) ou le calcul de fréquence est requis. C'est la raison pour laquelle chaque ronde se décompose en 2 phases :

– Le maître détermine les itemsets fréquents *non maximaux*, c'est-à-dire dont le calcul de taille est requis.
– Le maître distribue équitablement le calcul de fréquences de ce sous-ensemble aux k esclaves et

L'algorithme distribué est basé sur les lemmes suivants :

Lemme 2. *Soient U et V deux itemsets tels que U est un sur-ensemble de V et $U <_{lex} V$ alors $\texttt{round}(V) = \texttt{round}(U+1)$.*

Lemme 3. *Tous les sur-ensembles de cardinalité $|U|+1$ plus petits dans l'ordre lexicographique d'un itemset U de la ronde R sont traités dans la même ronde précédente (ronde $R-1$).*

Il est à noter que le lemme **??** implique que le nombre total de calcul de fréquences est $O(2^N)$ si les itemsets sont considérés les uns aprés les autres. Nous donnons quelques indications pour montrer que l'algorithme distribué ne fait pas plus de calcul de fréquences au total. Considérons l'arbre lexicographique des itemsets. Chaque itemset I a deux types de sur-ensembles : ceux qui en découlent dans l'arbre (notés $Suiv(I)$ et ceux qui le précedent dans l'arbre (notés $Prec(I)$). Par exemple, $BCE \in Suiv(B)$ par contre $AB \in Prec(B)$. L'ordonnancement des calculs d'itemsets repose sur l'observation suivante : pour un itemset I, dès que tous les itemsets de $Prec(I)$ ont été traités, il n'est pas besoin d'attendre plus longtemps pour traiter I. Le lemme 3 garantit cette propriété. Autrement dit, soit il existe $J \in Prec(I)$ t.q J est fréquent et auquel cas, I l'est aussi soit aucun ne l'est donc il faut calculer $f(I)$. Le principe de l'amélioration consiste à, lors d'un même parcours de $\mathcal{T}$ calculer la fréquence de tous les itemsets I dont les parents dans $Prec(I)$ ont déjà été traités. L'ordonnancement fixé par ronde garantit cette propriété.

Exemple 2. *Pour $\mathbb{I} = \{A,B,C,D\}$, voir l'affectation des rondes dans le tableau 2.*

Théorème 1. *Par un algorithme distribué sur k processeurs, il est possible de calculer les éléments maximaux $B(s)$ d'une base de transactions $\mathcal{T}$ avec un nombre maximum de parcours de $\mathcal{T}$ de $2N$, une mémoire $O(|B(s)|)$ par processeur et un nombre maximum de calculs de fréquences par processeur égal à $\frac{2^N}{k}$.*

Faute de place, nous ne donnons pas la description précise de l'algorithme. Il serait bien entendu intéressant de vérifier en pratique le comportement du calcul distribué par rapport à l'existant.

Références

[AS94] Rakesh Agrawal and Ramakrishnan Srikant. Fast algorithms for mining association rules in large databases. In *VLDB Proceedings*, 1994.

[Bay98] Roberto Bayardo. Efficiently mining long patterns from databases. In *SIGMOD Proceedings*, 1998.

[BCG01] Douglas Burdick, Manuel Calimlim, and Johannes Gehrke. Mafia : A maximal frequent itemset algorithm for transactional databases. In *ICDE Proceedings*, 2001.

[CL08] Soon M. Chung and Congnan Luo. Efficient mining of maximal frequent itemsets from databases on a cluster of workstations. *Knowledge Inf. Systems*, 16 :359–391, 2008.

[GZ05] Karam Gouda and Mohammed J. Zaki. GenMax : An efficient algorithm for mining maximal frequent itemsets. *Data Mining and Knowledge Discovery*, 11(3) :223–242, 2005.

[HMT09] Nicolas Hanusse, Sofian Maabout, and Radu Tofan. A view selection algorithm with performance guarantee. In *Proceedings of EDBT/ICDT*, 2009.