



# **Utilisation des arbres de radicaux pour les algorithmes de Data-Mining sur grille de calcul.**

*Stage de DEA en Informatique Parallèle Répartie et Combinatoire..*

Gaël Le Mahec  
encadré par C. Cérin et M. Koskas

Laboratoire de recherche en informatique d'Amiens  
- Université de Picardie Jules Verne -

24 juillet 2004

## Table des matières

<b>1</b>	<b>Introduction</b>	<b>4</b>
1.1	Grilles et bases de données. . . . .	4
1.2	Extraction de Connaissances à partir de Données. . . . .	5
1.3	Règles associatives. . . . .	5
1.3.1	Obtenir une règle d'association à partir d'un motif. . . . .	6
<b>2</b>	<b>Analyse du problème.</b>	<b>7</b>
2.1	Formalisation . . . . .	7
2.2	Les principaux algorithmes de recherche d'itemsets fréquents. . . . .	7
2.2.1	L'algorithme 'Apriori'. . . . .	7
2.2.2	L'algorithme 'AprioriTID'. . . . .	8
2.2.3	L'algorithme 'Count Distribution'. . . . .	8
2.2.4	L'algorithme 'Data Distribution'. . . . .	8
2.2.5	L'algorithme 'Eclat'. . . . .	8
2.2.6	Synthèse. . . . .	8
2.3	Propriétés des itemsets. . . . .	9
2.4	Treillis d'itemsets. . . . .	9
2.5	Itemsets fermés. . . . .	9
<b>3</b>	<b>Résolution du problème.</b>	<b>10</b>
3.1	Apriori . . . . .	10
3.1.1	L'algorithme . . . . .	10
3.1.2	Génération des candidats dans Apriori . . . . .	11
3.1.3	Complexité. . . . .	12
3.2	Eclat . . . . .	12
3.2.1	Partitionnement en classes d'équivalences. . . . .	12
3.3	L'algorithme 'Eclat'. . . . .	13
3.3.1	Phase d'initialisation . . . . .	13
3.3.2	La phase de transformation. . . . .	13
3.3.3	La phase asynchrone. . . . .	14
3.3.4	La phase de réduction finale. . . . .	15
3.3.5	Complexité. . . . .	15
3.4	Conclusion. . . . .	15
<b>4</b>	<b>Les arbres de radicaux.</b>	<b>15</b>
4.1	Présentation . . . . .	15
4.2	Opérations sur les arbres de radicaux. . . . .	16
4.3	Parallélisation des opérations sur les arbres de radicaux. . . . .	18
4.4	Stockage sur disque à l'aide d'arbre de radicaux. . . . .	20
<b>5</b>	<b>Utilisation des arbres de radicaux pour la recherche des itemsets fréquents.</b>	<b>20</b>
5.1	Insertion d'un item dans un arbre de radicaux. . . . .	20
5.2	Insertion d'ensemble dans les arbres de radicaux. . . . .	21
5.2.1	Produit d'ensembles. . . . .	21
5.3	Treillis d'itemsets. . . . .	22

5.3.1	Composition sans répétition des éléments d'un ensemble. . . . .	22
5.3.2	Élagage d'arbre de radicaux . . . . .	23
5.4	Génération de candidats. . . . .	24
5.4.1	Construction du treillis des itemsets. . . . .	24
5.5	Transformation verticale de la base. . . . .	25
5.6	Construction du treillis des itemsets fréquents. . . . .	25
5.7	Recherche des itemsets fréquents. . . . .	26
5.8	Recherche parallèle des itemsets fréquents. . . . .	27
5.9	L'algorithme parallèle de recherche des itemsets fréquents. . . . .	27
<b>6</b>	<b>Conclusion.</b>	<b>29</b>

L'informatique s'est, à ses débuts, construite autour du concept d'ordinateur central regroupant toutes les ressources. De ce concept sont nés les supercalculateurs permettant d'obtenir une grande puissance de calcul (Cray, Deeper Blue etc.). Les premiers réseaux étant conçus sur ce modèle, il ne permettaient que l'interrogation d'un serveur par un logiciel client (modèle client/serveur). Si TCP/IP, la couche de protocole supportant Internet, (exploitée pour la première fois à grande échelle en 1983) offrait déjà une symétrie permettant à un logiciel d'être à la fois client et serveur, cette capacité n'était que peu exploitée. Les protocoles les plus utilisés reposaient sur le modèle client/serveur (FTP, HTTP etc.).

Depuis le milieu des années 90, le concept de système pair-à-pair s'est largement développé avec l'explosion du nombre de connections à internet et l'apparition de logiciels de partage de fichiers (Napster, Kazaa, Gnutella etc.). Dans un système pair-à-pair, chaque noeud est à la fois client et serveur et est "l'égal" des autres (on parle aussi de système "d'égal à égal"). Il supporte le système sans pour autant être indispensable à celui-ci. On peut comparer ce type de système à un réseau ad-hoc supportant les disparitions et apparitions de noeuds et de liens entre ceux-ci. Cependant, des systèmes plus centralisés sont en général considérés comme systèmes pair-à-pair. Ainsi dans le cas de Napster, c'est un serveur central qui gère les partages, la recherche et l'insertion d'informations bien que celles-ci ne transitent ensuite qu'entre les pairs. Ces systèmes sont beaucoup plus vulnérables que les systèmes décentralisés car la déconnexion du serveur les paralyse intégralement. À l'inverse, les systèmes décentralisés comme Gnutella sont beaucoup plus robustes mais rendent l'obtention de l'information plus difficile et sans garantie. Sur ce type de réseau, le fait de ne pas trouver une information ne signifie pas que celle-ci n'est pas présente et les recherches nécessitent de nombreux messages. Des protocoles basés sur les tables de hachage distribuées ont été développés pour limiter ce nombre de messages et garantir une plus grande fiabilité de recherche des données et sont implémentés dans Chord [18], Freenet [13], Pastry [6] par exemple...

D'autres systèmes optent pour une architecture intermédiaire confiant certaines tâches utiles au réseau comme l'indexation des données à des super-pairs (superpeers), noeuds du réseau réputés plus fiables que les autres (Le protocole FastTrack utilisé par KaZaA par exemple). Si les systèmes de partage de fichiers sont les applications les plus connues des systèmes pair-à-pairs, d'autres systèmes permettent le partage de ressources telles l'espace disque (OceanStore [22], PAST [14], Us [34]...) ou le temps processeur (Seti@Home, Distributed.net...). Le succès de certains de ces systèmes comme Seti@Home qui a fédéré des centaines de milliers d'utilisateurs pour réaliser une tâche commune (l'analyse d'onde extra-terrestre) a montré la viabilité du concept de calcul distribué ou calcul global sur Internet. La capacité de calcul de ce système est considérable (jusqu'à 20 Teraflops dépassant ainsi la très grande majorité des supercalculateurs dans le monde) et valide la possibilité d'utilisation à grande échelle de ce type de systèmes.

On distingue parfois les systèmes de calcul global des systèmes pair-à-pair étant donné leur architecture très centralisée et l'inexistence de communications entre les pairs. Il reste néanmoins de nombreux points communs entre ces deux types de systèmes : l'échelle de déploiement, la volatilité des noeuds et la connexion du système à travers Internet. Le Grid Computing ou calcul sur grille se différencie encore légèrement de ces systèmes par son échelle de déploiement (plusieurs milliers de processeurs contre des centaines de milliers), la nature des ressources partagées (des systèmes rarement dotés de plus d'un processeur pour le calcul global) et le type de connections réseaux utilisées. C'est plus sur une architecture de type grid computing qu'est basé le travail de ce mémoire, mais avec

la perspective de faire évoluer les algorithmes proposés vers une architecture plus hétérogène et plus grande échelle.

## 1 Introduction

Dans la première partie de ce document, on présentera brièvement les principaux thèmes développés par la suite ainsi que le contexte dans lequel on veut les appliquer à savoir les grilles de calcul. La deuxième partie présente le problème de Data-Mining auquel nous nous sommes intéressés durant ce stage. La troisième partie présente les solutions “classiques” proposées à ce problème. La quatrième partie introduit les arbres de radicaux, structure de donnée utilisée pour la réalisation d’un algorithme original de Data-Mining présenté dans la cinquième partie.

### 1.1 Grilles et bases de données.

Les premières applications sur les grilles étaient pour la plupart basées sur une organisation des données en fichiers. Cependant, destinées à un spectre beaucoup plus large d’applications, aussi bien scientifiques que commerciales, on doit disposer d’applications qui offrent les fonctionnalités d’une organisation en base de données (requêtes, transactions etc.). Étant donné la diversité des gestionnaires de bases de données aussi bien dans leur paradigme (objet, relationnel) que dans les fonctionnalités qu’ils peuvent offrir, l’adaptation de ceux-ci aux grilles s’avère être un large problème aujourd’hui loin d’être résolu. On souhaiterait à terme bénéficier de tous les outils dont on dispose dans les gestionnaires plus “traditionnels” pour les bases de données distribuées sur les grilles. On aimerait de plus, fédérer des données de plusieurs sources distribuées, indépendamment des choix du créateur de ces données quant à l’organisation de celles-ci. Il est donc primordial de disposer d’un “middleware” générique pour la fédération des données stockées sur les grilles. Les fonctionnalités nécessaires à un système de gestion de bases de données sur les grilles doivent recouvrir celles offertes par les SGBD courants ([39]), à savoir :

- Gestion des requêtes et mises à jour.
- Interface de programmation.
- Indexation.
- Haute disponibilité.
- Gestion des erreurs.
- Gestion de multi-versions.
- Evolution
- Accès uniforme aux données et schémas de données.
- Contrôle de concurrence d’accès.
- Gestion des transactions.
- Bulk-loading.
- Archivage.
- Sécurité.
- Pérennité.
- Notification des changements (gestion des déclencheurs).

Pour réaliser cet objectif, on peut tenter d’adapter les systèmes de bases de données actuels aux grilles comme décrit dans [37], approche ne permettant pas de tirer plein parti de la puissance de stockage et de calcul de cette architecture, ou reconstruire un gestionnaire SQL conçu spécialement pour les grilles. Dans [21], M. Koskas décrit une nouvelle approche de gestion des données basée sur

l'emploi des arbres de radicaux. C'est cette approche qu'on aimerait utiliser pour l'implémentation d'un gestionnaire de bases de données sur grille, étant donné que celle-ci a montré des résultats supérieurs aux gestionnaires commerciaux traditionnels dans des phases de test par le TPC (Transaction Performance Council). Dans ce mémoire, on s'intéressera à un algorithme de Data-Mining illustrant les capacités offertes par la gestion des données dans des arbres de radicaux. Cet algorithme original faisant intervenir des arbres de radicaux à plusieurs niveaux fait l'objet d'une publication à paraître dans la prochaine conférence internationale Parallel Computing Systems PCS'04, Colima, Mexique.

## 1.2 Extraction de Connaissances à partir de Données.

Le Knowledge Discovery in Database (KDD) ou Extraction de Connaissances à partir de Données (ECD) consiste en l'extraction d'informations implicites, non connues et potentiellement utiles stockées dans des bases volumineuses [20].

L' ECD est un domaine de recherche en plein essor. En effet, les quantités de données collectées dans les divers domaines de l'informatique deviennent de plus en plus importantes et leur analyse de plus en plus demandée. Ce domaine de recherche se situe à la croisée de l'intelligence artificielle, des statistiques et des bases de données. On distingue en général plusieurs domaines dans l'ECD (classification, regroupement, découverte de règles associatives...). Le Data-Mining ou fouille de données, souvent vu comme équivalent à l'ECD, est plutôt à considérer comme une étape de celle-ci, néanmoins, il s'agit d'une étape essentielle et non triviale. Il s'agit d'extraire des données, des motifs nouveaux dont on tirera des informations potentiellement utiles.

“Le data mining est le descendant et, selon certains, le successeur des statistiques telles qu'elles sont utilisées actuellement. Statistiques et data mining ont le même but, qui est de réaliser des modèles compacts et compréhensibles rendant compte des relations liant la description d'une situation à un résultat (ou un jugement) concernant cette description.” [2] Les connaissances recueillies au terme du processus de recherche sont présentées sous forme de *motifs* définit comme suit dans Frawley et al. [38] :

“Un motif est une expression dans un langage qui décrit des relations dans un sous-ensemble de données avec une certaine certitude, telle que l'expression est plus simple (en un certain sens) que l'énumération de tous les faits de la base de données.”

Une connaissance est donc un *motif intéressant* vis à vis d'une mesure d'intérêt laissée à l'utilisateur. Si la mesure d'intérêt est la fréquence, les connaissances sont des *motifs fréquents*. Cette découverte de motifs fréquents est un problème important du Data-Mining et est l'objet de nombreuses recherches. De ces motifs on tirera des règles d'associations entre attributs ou items de la base par des mesures simples de fréquences et de probabilités. Ainsi, les règles obtenues sont de la forme : Si on rencontre l'item A alors il y a x% de chances de rencontrer l'item B. Ces règles ont une grande importance dans de nombreux domaines. Dans le marketing avec le célèbre exemple de la corrélation de l'achat de couche et de bière le samedi (et pas les autres jours). Mais aussi dans des domaines très variés (l'association du syndrome de Reyes et la prise d'aspirine chez les enfants...).

C'est à ce problème et aux algorithmes qui en découlent que je me suis principalement intéressé lors de ce stage de DEA.

## 1.3 Règles associatives.

Partant d'une base de transaction (par exemple un ensemble de tickets de caisse), on souhaite obtenir des corrélations de présences d'objets différents dans les transactions. Étant donné la taille des bases à analyser, on limite en général la recherche des règles d'associations aux objets les plus

courants de la base. On entend par “courant” les objets ou ensembles d’objets qui répondent à un critère minimum de présence choisi par l’utilisateur. Ce critère appelé “support minimum” est le taux de présence minimum exigé pour prendre en compte l’objet ou l’ensemble d’objets dans la recherche de motifs. Une règle associative, écrite sous la forme  $A \implies B$  désigne une relation entre des conjonctions d’attributs  $A$  et  $B$ , pondérée par une valeur de “confiance”. Dans cette expression,  $A$  désigne la prémisse de la règle et  $B$  sa conclusion. Par exemple, la règle vue en introduction s’écrit :

- (Samedi)  $\wedge$  (Couches)  $\implies$  (Bière). (C’est bien l’ordre relevé par l’étude.)

L’utilisateur choisit des valeurs seuil pour déterminer quelle règle est intéressante et éliminer les règles ayant peu d’intérêt par leur caractère de fréquence insuffisamment affirmé.

### 1.3.1 Obtenir une règle d’association à partir d’un motif.

Pour trouver les règles d’associations, on commence par chercher des motifs fréquents dans la base de données. On considère un motif comme fréquent à partir d’une présence minimum dans la base (appelée support minimum) donné ‘a priori’ (d’où le nom de l’algorithme fondamental de recherche des règles d’associations) par l’utilisateur. Ainsi, si le support minimum est fixé à 10 %, les ensembles d’items apparaissant dans moins de 10 % des transactions sont éliminés des motifs fréquents. A partir de ces motifs et leurs supports, on calcule la valeurs de confiance des règles que l’on peut en déduire. Si par exemple le motif  $ABC$  est retenu comme motif fréquent, on pourra en tirer les règles :

- $AB \implies C$
- $AC \implies B$
- $BC \implies A$
- $A \implies BC$
- $B \implies AC$
- $C \implies AB$

Pour calculer la valeur de confiance de la première règle, on effectue le calcul  $\frac{\text{support}(ABC)}{\text{support}(AB)}$  ce qui correspond en fait à la probabilité conditionnelle de rencontrer  $C$  dans une transaction sachant que celle-ci contient  $A$  et  $B$ . Le résultat est bien sûr en général différent pour chacune des règles déduite d’un motif. En effet, il est probable que de nombreux célibataires achètent de la bière le samedi sans acheter pour autant des couches (alors qu’apparemment, l’achat de couche implique bien souvent l’achat de bière...). Ainsi, l’utilisateur peut encore filtrer les résultats obtenus sur la valeur de confiance d’une règle afin d’éliminer celles qui ne sont pas réellement significatives.

On appelle arbitrairement “fréquent” un itemset qui apparaît un nombre minimum de fois dans la base, ce nombre minimum étant laissé à la définition de l’utilisateur. S’il s’agit de trouver tous les itemsets de la base afin de découvrir toutes les règles d’associations valables, il suffit de choisir pour support minimum  $\frac{1}{|\mathcal{T}|}$ ,  $|\mathcal{T}|$  étant le nombre de transactions de la base de transactions  $\mathcal{T}$ . Ainsi, chaque itemset apparaissant dans la base est considéré comme fréquent. On voit ici, que le principal problème pour trouver des règles d’associations consiste en la découverte des itemsets fréquents. En effet, ceux-ci sont potentiellement au nombre de  $2^m$  pour une base contenant  $m$  items. Du choix du support minimum dépend énormément le nombre d’itemsets fréquents, mais pas le nombre de règles d’associations valables. On peut, sur une base de taille importante, ne jamais trouver de corrélation entre la présence d’un item et la présence d’un autre même si les items choisis ont une présence importante dans la base. Alors même que la présence d’un item par ailleurs très peu représenté dans la base peut entraîner systématiquement la présence d’un autre item. Par exemple (ces exemples sont purement fictifs) :

- De nombreuses personnes achètent du pain et de nombreuses personnes achètent des piles (support important), cependant, on trouve une valeur de confiance peu élevée pour la règle

(Pain)  $\implies$  (Piles).

- Peu de personnes achètent un appareil photo numérique, peu de personnes achètent des cartes mémoires (support faible), mais la valeur de confiance de la règle (Appareil photo numérique)  $\implies$  (Carte mémoire) est élevée.

La recherche de règles d'associations valables nécessite donc en général l'emploi d'une valeur de support minimum faible pour ignorer un minimum de règles intéressantes. Or, plus le support minimum choisi est faible, plus les itemsets fréquents sont nombreux et donc plus le travail à fournir est important. C'est pour ces raisons, qu'il est indispensable de bénéficier d'une puissance de calcul et d'espace mémoire important pour réaliser la tâche de rechercher les itemsets fréquents. C'est ce qui a conduit au développement d'algorithmes parallèles. Étant donné la taille des bases à traiter, un calcul sur grille semble fournir une solution adéquate à la puissance de calcul nécessaire à la réalisation de cet objectif. De plus, les bases de données dont on veut tirer les motifs fréquents sont en général de tailles très importantes et nécessitent une capacité de stockage que peut offrir le calcul sur grille.

## 2 Analyse du problème.

### 2.1 Formalisation

Une formalisation du problème a été donné dans [40] :

Soit  $\mathcal{I} = \{i_1, i_2, \dots, i_m\}$  un ensemble de  $m$  attributs distincts également appelés *items*. Chaque transaction  $\mathcal{T}$  de la base de transaction  $\mathcal{D}$  possède un identifiant unique et *contient* un ensemble d'items (appelé *itemset*) tel que  $\mathcal{T} \subseteq \mathcal{I}$ , i.e. chaque transaction est de la forme  $\langle \text{TID}, i_1, i_2, \dots, i_k \rangle$ . Un itemset de  $k$  éléments est appelé  $k$ -itemset, un sous-ensemble de taille  $k$  est appelé  $k$ -subset. On dit qu'un itemset a un support de  $s$  si  $s\%$  des transactions de  $\mathcal{D}$  contiennent cet itemset.

Une règle d'association est une expression de la forme  $A \implies B$ , avec les itemset  $A, B \subset \mathcal{I}$  et  $A \cap B = \emptyset$ . La confiance d'une telle règle d'association est donnée par le rapport  $\frac{\text{support}(AB)}{\text{support}(A)}$ , ce qui correspond à la probabilité qu'une transaction de  $\mathcal{D}$  contienne  $B$  sachant qu'elle contient  $A$ .

### 2.2 Les principaux algorithmes de recherche d'itemsets fréquents.

Les algorithmes brièvement présentés, présentent deux approches différentes : ceux basés sur Apriori et l'algorithme Eclat qui se démarque dans le sens où il distribue le travail aux processeurs de manière à ce que ceux-ci trouvent les itemsets fréquents par simple intersection d'ensembles de transactions de manière indépendante. Apriori et ses dérivés quant à eux, travaillent sur des ensembles d'items et pas de transactions.

#### 2.2.1 L'algorithme 'Apriori'.

Cet algorithme fondamental proposé par R. Agrawal et R. Srikant en 1994 [28] est la base de nombreux algorithmes de recherche de règles d'association. Il repose sur le fait qu'un ensemble fréquent d'éléments a pour sous-ensembles des ensembles fréquents d'éléments. En effet si l'ensemble  $\{A, B\}$  est fréquent dans la base, les ensembles  $\{A\}$  et  $\{B\}$  sont eux-mêmes fréquents dans la base. L'algorithme fonctionne en trois phases :

- Calcul des fréquences des ensembles d'un seul élément (les objets de la base) et élimination des ensembles dont la fréquence n'atteint pas le support minimum.

- On génère ensuite les ensembles candidats de deux éléments par jointure de l'ensemble des éléments fréquents obtenu à l'issue de la phase précédente.
- On répète ce processus afin d'obtenir tous les ensembles fréquents.

Cet algorithme dans sa version originale nécessite  $k$  passes sur la base s'il existe des ensembles fréquents de taille  $k$ .

### 2.2.2 L'algorithme 'AprioriTID'.

Cet algorithme, amélioration de 'Apriori' a été proposé par Agrawal & Srikant en 1994 [28], il cherche à garder le contexte en mémoire afin de limiter les accès à la base. À supposer (ce qui est en général le cas sur des données réelles) que les ensembles de candidats décroissent en même temps que la taille des candidats, il s'avère bien plus efficace qu'Apriori.

### 2.2.3 L'algorithme 'Count Distribution'.

Proposé par R. Agrawal et J.C Shafer en 1996 [27], cet algorithme est une version parallélisée de l'algorithme Apriori. Chaque processeur traite sa portion locale de la base de transactions, il minimise le coût des communications, mais est freiné par la structure de données utilisée qui, lors du traitement d'une base de données de grande taille impose de coûteuses opérations d'entrée/sortie. Seuls les supports des candidats sont transmis lors d'une opération de Broadcast *All-to-all*. Nous analyserons cet algorithme en détail dans la suite de ce document.

### 2.2.4 L'algorithme 'Data Distribution'.

Il s'agit encore d'un algorithme issu d'Apriori et proposé par R. Agrawal et J.C. Shafer en 1996 [27], il diffère de Count Distribution dans le sens où, les processeurs se partagent le travail non plus par portion de base de données mais par candidats. Chaque processeur traite un ensemble de candidats différents, ce qui impose d'accéder aux portions de la base de transaction des autres processeurs, ce qui augmente drastiquement la charge des communications. Il s'avère meilleur que Count Distribution lorsque la base contient beaucoup d'items distincts et lorsqu'on choisit un support minimal peu élevé. Afin de réduire la charge en communication de cet algorithme, une version considérant les processeurs comme étant sur un anneau logique a été créée, c'est l'algorithme 'Intelligent Data Distribution'.

### 2.2.5 L'algorithme 'Eclat'.

Introduit par M. J. Zaki en 1997 [41], cet algorithme repose sur le découpage de la base en classes d'équivalences et distribution de la charge de travail sur tous les processeurs. On considère que deux itemsets (ensemble d'items) sont dans la même classe d'équivalence s'ils, désignés par les items qu'ils contiennent dans l'ordre lexicographique, possèdent un préfixe commun. Par exemple, les itemsets ABC et ABD sont dans la classe d'équivalence AB. Au lieu de transmettre des supports locaux ou des portions de base de données comme dans les principaux algorithmes dérivés d'Apriori, cet algorithme fonctionne en transmettant les listes de transactions correspondant à chaque classe d'équivalence au processeur qui s'occupe de celle-ci. Cet algorithme sera détaillé par la suite.

### 2.2.6 Synthèse.

Les algorithmes ici présentés fonctionnent suivant deux approches différentes, les algorithmes basés sur Apriori étudient les transactions, c'est à dire des ensemble d'items, alors qu'Eclat étudie des

ensembles de transactions par classe d'équivalence d'items. On parle ici de placements respectivement horizontal et vertical de la base tels que les illustre la figure 1

transactions \ items	items			
	A	B	C	D
T <sub>1</sub>	1	0	1	1
T <sub>2</sub>	0	0	1	1
T <sub>3</sub>	1	0	1	0

forme horizontale de la base

forme verticale de la base

FIG. 1 – Formes horizontale et verticale de la base.

Par l'utilisation d'arbre de radicaux, nous proposerons plus loin, un algorithme qui essaye de tirer partie des avantages des deux approches.

### 2.3 Propriétés des itemsets.

**Propriété 1 :** Tout sous-ensemble d'un ensemble fréquent est fréquent.

Soit  $A$  itemset fréquent. C'est à dire que  $support(A) > MinSupport$ .

Supposons qu'il existe  $A' \subset A$  tel que  $support(A') \leq MinSupport$ . On a alors :

$support(A') \geq support(A)$  puisque  $A' \subset A$ .

Ce qui implique  $support(A) \leq support(A') \leq MinSupport$  et donc  $support(A) \leq MinSupport$  et  $A$  est un itemset infrequent ce qui contredit l'hypothèse.

**Propriété 2 :** Tout sur-ensemble d'un ensemble infrequent est infrequent.

Soit  $A$  un itemset infrequent. C'est à dire que  $support(A) \leq MinSupport$ .

Supposons qu'il existe  $A'$  tel que  $A \subset A'$  et  $support(A') > MinSupport$ . On a :

$support(A') \leq support(A)$  puisque  $A \subset A'$ .

Ce qui implique  $support(A) \geq support(A') > MinSupport$  et donc  $support(A) > MinSupport$  et  $A$  est un itemset fréquent ce qui contredit l'hypothèse.

### 2.4 Treillis d'itemsets.

On appelle treillis un ensemble ordonné  $(\mathcal{T}, \leq)$  dans lequel toutes les paires de  $\mathcal{T}$  possède une borne inférieure et supérieure. L'ensemble des parties d'un ensemble  $\mathcal{I}$  muni de l'opération d'inclusion  $\subseteq$  est un treillis : Il admet une borne inférieure  $\emptyset$  et une borne supérieure  $\mathcal{I}$ .

Tous les itemsets des transactions d'une base peuvent être représentés par un treillis. Nous nous servirons de cette propriété, pour générer les candidats à l'aide de radix trees dans notre algorithme.

### 2.5 Itemsets fermés.

On appelle itemset fermé un itemset maximal commun à un ensemble de transactions. Par exemple, pour les transactions  $\{A, B, C, E, F, H\}$  et  $\{A, C, E, F\}$ , l'itemset  $ACEF$  est un itemset fermé car

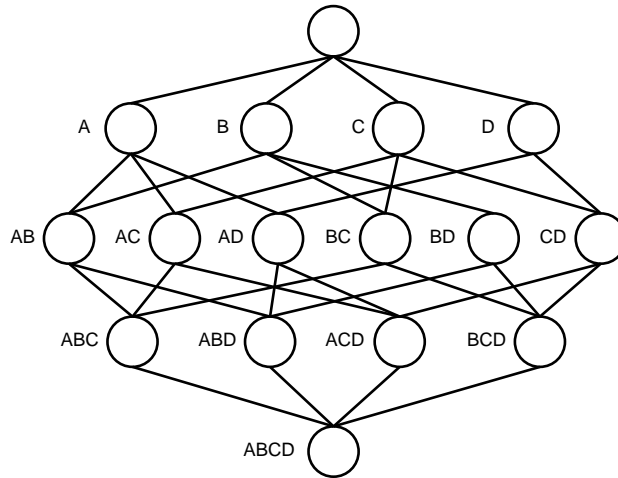


FIG. 2 – Treillis d’itemsets pour 4 items.

c’est le plus grand ensemble commun aux deux transactions. Par contre, les itemsets  $A, B, C, E, F, H, AC$  et  $ACE$  ne sont pas fermés pour ces deux transactions.

On appelle alors itemset fermé fréquent un itemset fermé  $A$  tel que  $support(A) > MinSupport$ . Ce type d’itemset se retrouve alors comme les noeuds terminaux du treillis des itemsets fréquents (voir figure 3).

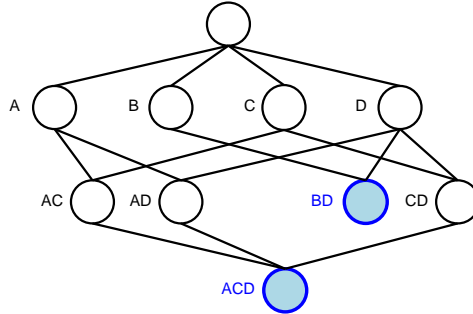


FIG. 3 – Itemsets fermés fréquents dans le treillis des itemsets fréquents.

### 3 Résolution du problème.

Nous allons ici étudier en détail deux algorithmes fondamentaux dans la recherche des itemsets fréquents : Apriori et Eclat qui proposent deux approches légèrement différentes.

#### 3.1 Apriori

##### 3.1.1 L’algorithme

On a vu que tout sur-ensemble d’un ensemble infrequent est un ensemble infrequent. C’est partant de cette constatation que l’algorithme Apriori a été proposé [28]. En effet, en cherchant d’abord les

itemsets fréquents de plus petite taille, on peut envisager d'éliminer les candidats qui contiennent des sous-ensembles infréquents avant même d'évaluer s'ils sont eux-mêmes fréquents.

```

 $L_1$  = Ensemble des itemsets fréquents de taille 1.
pour ( $k = 2$ ;  $L_{k-1}$ ;  $k++$ )
 $C_k$  = Ensemble des nouveaux candidats.
pour toutes les transactions  $t \in \mathcal{D}$ 
pour tous les sous-ensembles  $s$  de  $t$ 
si ( $s \in C_k$ )  $s.count++$ ;  $L_k = \{c \in C_k \mid c.count \geq \text{support minimum}\}$ ;
Ensemble résultat des itemsets fréquents =  $\bigcup_k L_k$ ;

```

L'algorithme commence par construire l'ensemble  $L_1$  des items (itemsets de taille 1 ou 1-itemsets) fréquents. Puis, afin d'obtenir l'ensemble complet des itemsets fréquents, l'algorithme construit les ensembles de tailles croissantes en éliminant ceux qui contiennent des sous-ensembles infréquents.

Les candidats sont stockés dans un arbre de hachage pour faciliter l'évaluation des supports. Un noeud interne de l'arbre au niveau  $l$  contient une table de hachage qui pointe vers les noeuds de niveau  $l + 1$ . Les itemsets sont stockés au niveau des feuilles de l'arbre. Pour insérer un nouvel itemset, on effectue un hachage sur les items contenus dans l'itemset qui sont classés en ordre lexicographique, jusqu'à aboutir à la feuille qui le contiendra. L'algorithme maintient un compteur pour chaque itemset qui est incrémenté à chaque rencontre de celui-ci dans une transaction. La dernière étape consiste à éliminer les itemsets qui n'ont pas un support suffisant vis à vis du support minimum défini par l'utilisateur.

### 3.1.2 Génération des candidats dans Apriori

Pour fournir les nouveaux candidats, l'algorithme utilise la fonction Apriori-gen qui, pour construire les candidats de taille  $k$  prend en paramètre les itemsets fréquents de taille  $k - 1$  ordonnés dans l'ordre lexicographique. Ainsi, on applique la propriété des itemsets qui veut que tout sous-ensemble d'un itemset fréquent est fréquent et on élimine des candidats tout sur-ensembles d'items infréquents.

```

algorithme Apriori-gen( $L_{k-1}$ )
 $C_k \leftarrow \emptyset$ 
pour tout ensemble  $I_1$  d'items de  $L_{k-1}$ 
pour tout ensemble  $I_2 \neq I_1$ 
si  $I_2$  et  $I_1$  ont un préfixe commun de taille  $k - 2$  et  $I_1[k - 1] < I_2[k - 1]$ 
 $C_k \leftarrow C_k \cup (I_1 \cup I_2[k - 1])$ 
pour tout itemset  $c \in C_k$ 
pour tout sous-ensemble de taille  $k - 1$  de  $c$ 
si ( $s \notin L_{k-1}$ )
Supprimer  $c$  de  $C_k$ 

```

Lors de la génération des candidats, on joint d'abord tous les ensembles de  $L_{k-1}$  entre eux. Il y a alors potentiellement  $C_n^k$  itemsets possibles desquels on va supprimer ceux qui contiennent un sous-ensemble de taille  $k - 1$  qui n'appartient pas à  $L_{k-1}$ . Dans un ensemble de taille  $k$ , il y a  $C_k^{k-1} = k$  sous-ensembles de taille  $k - 1$ . On réalise donc l'opération d'élimination en  $k \times C_n^k$  opérations au

maximum. La génération complète des candidats est alors réalisée en  $C_n^{k-1} \times C_n^{k-1} + k \times C_n^k$  opérations au maximum.

### 3.1.3 Complexité.

Soit  $\mathcal{T}$  une base de transactions et notons  $|\mathcal{T}|$  sa taille, une transaction  $t \in \mathcal{T}$  est de taille  $|t|$  (i.e. le nombre d'items apparaissant dans cette transaction.). Chaque transaction  $t$  contient  $C_{|t|}^k$  sous-ensembles de taille  $k$ . Apriori et ses dérivés ont pour principal désavantage de vérifier à chaque itération, si tout sous-ensemble de toute transaction appartient à la liste des candidats. Ainsi, à l'itération  $k - 1$ , on cherche tous les ensembles de taille  $k$  de chaque transaction. Il faut donc, pour chaque transaction  $t$ , générer les  $C_{|t|}^k$  sous-ensemble de celle-ci afin de vérifier leur présence dans la liste des candidats. Si on procédait à l'inverse par la recherche des candidats dans les transactions, on devrait potentiellement vérifier la présence des  $C_n^k$  candidats avec  $n \geq |t|$  quelque soit la transaction. Ceci amènerait à une complexité encore beaucoup plus grande pour le calcul du support des candidats. On

effectuera donc en tout, lors du déroulement de l'algorithme,  $n \times \sum_{t \in \mathcal{T}}^{max(|t|)} \sum_{k=2} C_{|t|}^k$  vérification dans le cas où les  $2^n$  itemsets sont fréquents. On voit ici, que le temps de recherche des itemsets est exponentiel vis-à-vis de la taille des transactions (une transaction de taille 10 contient 252 ensembles de taille 5, dans une transaction de taille 11, il y en a 462...). En terme d'Entrées/Sorties, il faut parcourir la base complète  $N$  fois si  $N$  est la taille du plus grand itemset fréquent. La version parallèle d'Apriori ("Count Distribution") limite les communications entre processeurs à la transmission des supports locaux de chaque itemset. Ainsi, on transmet au maximum un vecteur de  $C_n^k$  entiers à l'itération  $k - 1$ . La quantité d'information à transmettre dépend ici du nombre d'items de la base.

## 3.2 Eclat

L'algorithme Eclat [41] utilise une répartition en classe d'équivalence des itemsets. Le coût engendré par la redistribution des itemsets est amorti dans la suite des itérations. Chaque processeur peut ensuite agir indépendamment des autres puisqu'il n'y a jamais recouvrement entre les classes d'équivalences.

### 3.2.1 Partitionnement en classes d'équivalences.

En définissant une application bijective  $\phi$  de l'ensemble des items vers l'ensemble des entiers, on peut définir un ordre total sur l'ensemble des items. En effet si  $\phi$  est bijective, alors chaque item dispose d'un identifiant unique et chaque identifiant est propre à un item. Dans la suite pour clarifier la notation, on désignera un item par une lettre majuscule et l'ordre utilisé sera l'ordre lexicographique. Un ensemble d'items sera représenté par une suite de lettres majuscules qu'on ordonnera donc suivant l'ordre lexicographique. Disposant d'un ordre sur les items, on peut définir des classes d'équivalences sur les itemsets basées sur les préfixes communs de l'écriture de deux itemsets.

Les items  $A, B, C$  et  $D$  sont tels que  $\phi(A) < \phi(B) < \phi(C) < \phi(D)$  alors, les itemsets s'écrivent dans l'ordre croissant de la valeur par  $\phi$  de leurs éléments. Ainsi, l'itemset contenant les items  $B, A$  et  $C$  s'écrit  $ABC$ , celui contenant les items  $D, B$  et  $A$  s'écrit  $ABD$  et ces deux itemsets appartiennent à la classe d'équivalence  $E_{AB}$  définie par leur préfixe commun de taille  $|itemset| - 1, AB$ .

### 3.3 L'algorithme 'Eclat'.

**Phase d'initialisation**

Scanne la partition locale de la base.  
Calcul des comptes locaux des itemsets de taille 2.  
Construction des comptes globaux de  $L_2$ .

**Phase de transformation**

Partitionnement de  $L_2$  en classes d'équivalences.  
Distribution de  $L_2$  sur les processeurs par classe d'équivalence.  
Transformation de la base locale en base verticale.  
Envoi des listes de transaction aux autres processeurs.  
 $L_2$  **local** = listes de transaction des autres processeurs.

**Phase asynchrone**

**pour** chaque classe d'équivalence  $E_2$  dans  $L_2$  **local**  
    *Construction*( $E_2$ );

**Phase finale de réduction**

Regroupement des résultats et calcul des associations.

#### 3.3.1 Phase d'initialisation

L'algorithme commence par scanner la base afin de construire les itemsets fréquents de taille 2. En effet, il est possible de générer ces ensembles avec peu de coût supplémentaire par rapport à la génération des itemsets fréquents de taille 1, profitant ainsi du gain obtenu en évitant de scanner la base 2 fois. Cependant, l'auteur de l'algorithme précise que si la base de données contient un grand nombre d'items, il est peut-être préférable de scanner la base deux fois afin d'éliminer les items inféquents avant de générer les itemsets de taille 2. A ce stade, on dispose de l'ensemble des itemsets fréquents  $L_2$ .

- On scanne la base de données.
- On construit un tableau de deux dimensions indexé par les items sur la hauteur et la largeur.

items	A	B	C	D
A	-	-	-	-
B	1	-	-	-
C	1	0	-	-
D	0	1	1	-

Dans cet exemple pour une base contenant 4 items, on rencontre les itemsets  $AB$ ,  $AC$ ,  $BD$  et  $CD$  chacun respectivement dans au moins une transaction de la base.

- Chaque processeur calcule le support local des itemsets puis effectue une réduction de somme des résultats des autres processeurs afin de construire les support globaux de chaque itemset de  $L_2$ .

#### 3.3.2 La phase de transformation.

L'algorithme commence par partitionner  $L_2$  en classes d'équivalences qui seront redistribuées sur les processeurs avec une politique d'équilibrage de charge basée sur une heuristique. On effectue alors une transformation de la base afin d'obtenir, non plus une liste d'items par transaction mais une liste de transactions par item (transformation verticale de la base.)

- Partitionnement de  $L_2$  en classes d'équivalences.
  - Calcul de la charge de travail pour chaque classe d'équivalence.
- La mesure est effectuée en fonction du nombre d'éléments  $s$  de la classe d'équivalence. On considère toutes les paires à traiter par la suite. Ainsi, la charge est calculée par la valeur  $C_s^2$ . Par exemple, si dans la classe d'équivalence  $[A]$  on trouve les itemsets  $AB$ ,  $AC$  et  $AD$ , la charge calculée sera  $C_3^2 = 3$ . Il s'agit alors de répartir les tâches à effectuer sur les processeurs en fonction d'une heuristique sur les charges calculées : On assigne toutes les classes d'équivalences par charge décroissantes sur le processeur qui a la charge la plus petite au moment de l'assignation.

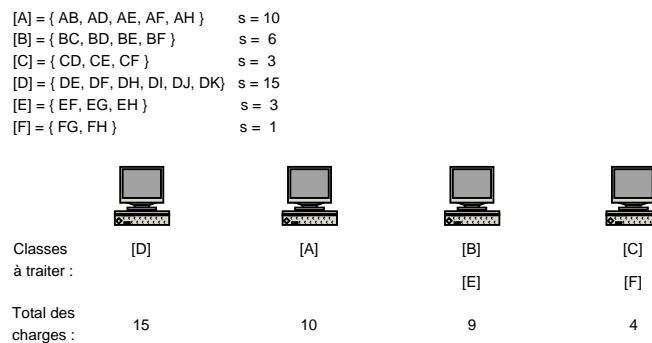


FIG. 4 – Répartition des charges de travail.

Chaque processeur peut effectuer cette tâche indépendamment des autres puisqu'ils disposent alors tous des supports globaux de  $L_2$ .

- Phase de transformation verticale de la base.
- Chaque processeur scanne sa portion locale de la base afin de construire les listes de transactions correspondant aux classes d'équivalence de  $L_2$ . Il faut alors transmettre respectivement les listes aux processeurs chargés de la classe d'équivalence correspondante.

### 3.3.3 La phase asynchrone.

**algorithme**  $Construction(E_{k-1})$   
**pour** tous les itemsets  $I_1$  et  $I_2 \in E_{k-1}$   
**si**  $((I_1.transactions \cap I_2.transactions) \geq \text{support minimum})$   
 Ajouter  $(I_1 \cup I_2)$  à  $L_k$   
 Partitionner  $L_k$  en classes d'équivalences.  
**pour** chaque classe d'équivalence  $E_k \in L_k$   
 $Construction(E_k)$

Les processeurs effectuent concurremment la construction des itemsets de tailles croissantes par intersection des listes de transactions des éléments de chaque classe d'équivalence entre eux. Éliminant les itemsets de support insuffisant, on réduit rapidement le travail à effectuer en même temps qu'on augmente la taille des itemsets construits. C'est du moins ce qu'il se passe sur des données réelles.

### 3.3.4 La phase de réduction finale.

La dernière tâche de l'algorithme consiste en l'accumulation et la réunion des résultats de chaque processeur.

### 3.3.5 Complexité.

La phase d'initialisation, qui consiste en la création de l'ensemble  $L_2$  des itemsets fréquents de taille 2, nécessite une lecture complète de la base locale  $\mathcal{T}$  et  $C_n^2$  opérations pour construire les potentiels  $C_n^2$  itemsets fréquents de taille 2 avec  $n$  le nombre d'items distincts dans  $\mathcal{T}$ . On effectue alors la communication des supports locaux aux autres processeurs en envoyant au maximum un vecteur de  $C_n^2$  entiers. Les processeurs dans la construction de  $L_2$ , nécessite un espace mémoire de l'ordre de  $2^n$  afin de construire la table de fréquences des itemsets de taille 2.

Durant la phase de transformation, si on a pris soin de trier les itemsets de  $L_2$ , on partitionne l'ensemble en  $n$  classes au maximum en temps linéaire par rapport à la taille de  $L_2$  (on parcourt une seule fois  $L_2$  afin de trouver les indices de début et de fin de chaque classe d'équivalence). En  $n$  étapes au maximum, on aura attribué les classes d'équivalence de chaque processeur. On effectue alors une seconde lecture de la base afin de construire les listes de transactions par item (transformation verticale de la base) ceci est réalisé en  $N$  opérations,  $N$  étant le nombre total d'apparition de tous les items dans  $\mathcal{T}$ . On transmet alors les listes de transactions de chaque classe d'équivalence aux autres processeurs. On transmet donc potentiellement des vecteurs de  $|\mathcal{T}|$  identifiants à chaque processeur. Les listes reçues sont alors réunies et triées afin de faciliter la suite des opérations.

## 3.4 Conclusion.

La tâche la plus coûteuse de l'algorithme Eclat est en général la transmission des listes de transactions de chaque item. En effet, sur des bases où les items sont repartis de manière homogène, chaque processeur doit transmettre des listes de tailles importantes à tous les autres. C'est là le point faible de l'algorithme Eclat que nous cherchons à éviter dans notre algorithme. L'algorithme Apriori et ses dérivés quant à eux souffrent de la perte d'informations entre chaque traitement d'itemset dû au fait qu'il n'utilise pas de transformation verticale de la base. Nous verrons plus loin que l'on peut tirer profit des arbres de radicaux pour effectuer cette transformation en même temps que nous construisons les candidats.

## 4 Les arbres de radicaux.

### 4.1 Présentation

Un arbre de radicaux ou arbre des préfixes, est une structure de données pour la représentation des ensembles. Partant de la racine, on parcourt les noeuds de l'arbre en fonction du préfixe de l'élément que l'on recherche (ou de son codage). Les arbres de radicaux sont des arbres  $n$ -aires avec  $n$  la taille de l'alphabet sur lequel on a codé les éléments. Ainsi, l'organisation d'un dictionnaire de la langue anglaise (qui n'utilise pas d'accents) en arbre de radicaux s'effectuera sur un alphabet de 26 lettres et chaque noeud de l'arbre pourra compter 26 fils.

Dans cette représentation, on voit que la présence d'un élément est indiquée par la couleur du noeud : Noir si l'ensemble contient l'élément, blanc sinon. Pour représenter des objets dont le codage est de taille fixe (un entier en mémoire d'un ordinateur par exemple), on peut s'abstenir de maintenir

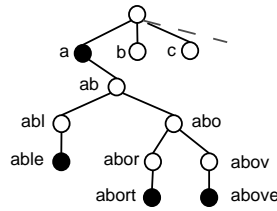


FIG. 5 – Un arbre de radicaux représentant un lexique d’anglais.

cette information puisque chaque objet est désigné par une feuille de l’arbre et sa seule présence signifie que l’objet appartient à l’ensemble.

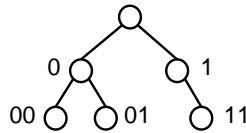


FIG. 6 – Un arbre de radicaux représentant l’ensemble {0, 1, 3} en codage binaire sur 3 bits.

Dans la suite de ce rapport, on ne fera plus référence qu’à des arbres de radicaux sur des codages de taille fixe.

Dans la figure 6, la représentation est effectuée sur un arbre de radicaux binaire. On peut facilement coder des entiers sur un arbre de radicaux  $2^n$ -aire afin de réduire sa profondeur et profiter des capacités de calcul des processeurs lors de la réalisation des opérations de bases que nous verrons plus loin. La figure 7 présente une telle représentation.

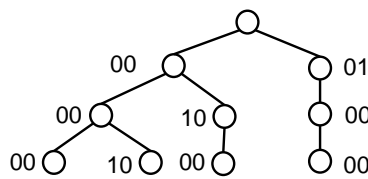


FIG. 7 – Un arbre de radicaux représentant l’ensemble {0, 2, 3, 8, 16} codé sur 6 bits.

## 4.2 Opérations sur les arbres de radicaux.

On a vu que les arbres de radicaux représentent des ensembles. On peut dès lors effectuer les opérations ensemblistes d’union et d’intersection par des opérations sur les arbres de radicaux. Un arbre de radicaux est un  $n$ -uplet de la forme :

$$\mathcal{A} = \begin{cases} \emptyset \\ (a_1, a_2, \dots, a_n) \end{cases}$$

avec  $a_i$  un arbre de radicaux.

On définit ainsi l'opération d'union de deux arbres de radicaux sur un alphabet  $n$ -aire :

$$\mathcal{A} \cup \emptyset = \mathcal{A}$$

$$\emptyset \cup \mathcal{A} = \mathcal{A}$$

$$\mathcal{A} \cup \mathcal{B} = (a_1, \dots, a_n) \cup (b_1, \dots, b_n) = (a_i \cup b_i | i \in \{1, \dots, n\})$$

**Propriété 1 :** L'union de deux arbres de radicaux représentant deux ensembles est l'arbre de radicaux représentant l'union des deux ensembles.

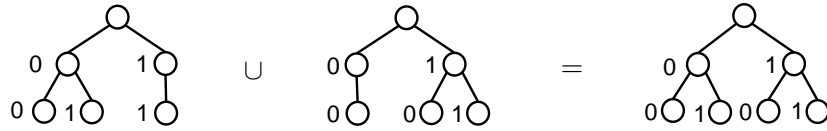


FIG. 8 – Union de deux arbres de radicaux.

Dans la figure 8, on effectue l'union de deux arbres représentant les ensembles  $\{0, 1, 3\}$  et  $\{0, 2, 3\}$ . Ces arbres sur l'alphabet binaire  $\{0, 1\}$  sont, dans la formalisation donnée plus haut, de la forme :

$$\mathcal{A} = (((\emptyset, \emptyset), (\emptyset, \emptyset)), (\emptyset, (\emptyset, \emptyset)))$$

$$\mathcal{B} = (((\emptyset, \emptyset), \emptyset), ((\emptyset, \emptyset), (\emptyset, \emptyset)))$$

Et en appliquant l'opération d'union, on obtient un arbre de la forme :

$$\mathcal{A} \cup \mathcal{B} = (((\emptyset, \emptyset), (\emptyset, \emptyset)), ((\emptyset, \emptyset), (\emptyset, \emptyset)))$$

Ce qui correspond bien à la représentation de l'ensemble  $\{0, 1, 2, 3\}$  résultat de l'union des deux ensembles.

**Propriété 2 :** L'intersection de deux arbres de radicaux représentant deux ensembles est l'arbre de radicaux représentant l'intersection des deux ensembles.

On peut définir de même que pour l'opération d'union, l'opération d'intersection de deux arbres de radicaux, toujours sur un alphabet  $n$ -aire :

$$\mathcal{A} \cap \emptyset = \emptyset$$

$$\emptyset \cap \mathcal{A} = \emptyset$$

$$\mathcal{A} \cap \mathcal{B} = (a_1, \dots, a_n) \cap (b_1, \dots, b_n) = (a_i \cap b_i | i \in \{1, \dots, n\})$$

Dans la figure 9, on effectue l'intersection des deux arbres dont on a réalisé l'union précédemment et le résultat dans la formalisation est bien  $(((\emptyset, \emptyset), \emptyset), (\emptyset, (\emptyset, \emptyset)))$ , représentant l'ensemble  $\{0, 3\}$ .

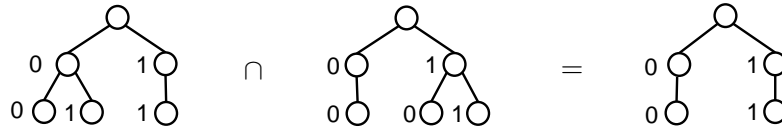


FIG. 9 – Intersection de deux arbres de radicaux.

### 4.3 Parallélisation des opérations sur les arbres de radicaux.

On déduit des formalisations des opérations d'union et d'intersection d'arbres de radicaux telles qu'on les a définies en 4.2, deux algorithmes itératifs présentés ici :

**algorithme** *Union*(Arbre A, Arbre B)

Arbre C =  $(c_1, \dots, c_n)$

**si**  $(A = \emptyset)$

**retourner** B

**si**  $(B = \emptyset)$

**retourner** A

**pour**  $i \leftarrow 1$  à  $n$

$c_i \leftarrow Union(a_i, b_i)$

**retourner** C

**algorithme** *Intersection*(Arbre A, Arbre B)

Arbre C =  $(c_1, \dots, c_n)$

**si**  $(A = \emptyset)$  **ou**  $(B = \emptyset)$

**retourner**  $\emptyset$

**pour**  $i \leftarrow 1$  à  $n$

$c_i \leftarrow Intersection(a_i, b_i)$

**FinPour**

**retourner** C

Dans ces algorithmes, on voit que la boucle **pour** réalise  $n$  opérations indépendantes que l'on peut envisager de paralléliser. On peut effectuer simplement la distribution du travail à réaliser sur  $p$  processeurs. On obtient ainsi l'algorithme d'intersection suivant : (le même principe étant appliqué à l'union) :

**algorithme** *IntersectionParallèle*(Arbre A, Arbre B)

Arbre C =  $(c_1, \dots, c_n)$

**si**  $(A = \emptyset)$  **ou**  $(B = \emptyset)$

**retourner**  $\emptyset$

**pour**  $i \leftarrow 1$  à  $\lceil \frac{n}{p} \rceil$

**pour**  $j \leftarrow 1$  à  $p$  **en parallèle**

$c_{i \times p + j} \leftarrow Intersection(a_{i \times p + j}, b_{i \times p + j})$

**retourner** C

Si cette parallélisation est optimale dans le cas où les arbres sont complets, de nombreux pro-

cesseurs ont des périodes d'attente longues lorsque ce n'est pas le cas. En effet, si un seul fils de la racine de  $C$  par tranche de  $p$  fils n'est pas vide, à chaque itération de la première boucle **pour**,  $p - 1$  processeurs attendrons la fin du travail du seul processeur actif. De plus, si  $p > n$ ,  $p - n$  processeurs ne travaillent jamais. On cherche donc un algorithme qui, à chaque niveau de l'arbre vérifie, si un processeur est inactif et l'utilise le cas échéant. Plutôt que de paralléliser l'algorithme par "sous-arbre", on va utiliser une stratégie par "branche". Il s'agit d'utiliser les processeurs sur la hauteur de l'arbre à chaque appel récursif de l'opération. On va utiliser une stratégie de parcours en profondeur du travail de chaque processeur.

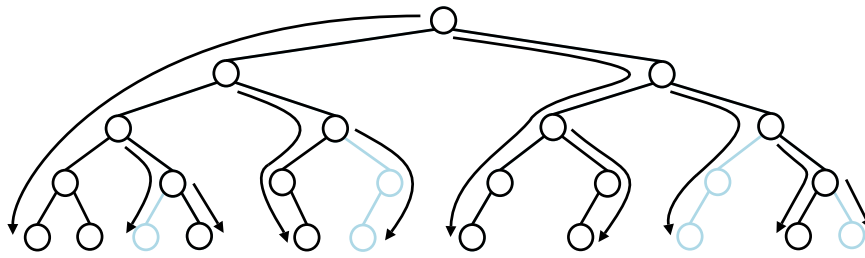


FIG. 10 – Stratégie de parallélisation de l'algorithme d'union.

La figure 10 présente une stratégie de répartition du travail sur les  $p$  processeurs. Au départ, on répartit le travail sur 2 processeurs, au second niveau, s'il reste des processeurs disponibles, on les utilise pour continuer sur la branche de droite, les deux processeurs du départ continuant leurs parcours par la gauche. On procède ainsi récursivement à chaque niveau. Si à un niveau, aucun processeur n'est disponible pour traiter la branche de droite, c'est le même processeur qui traitera les deux branches. On vérifie systématiquement si un processeur est disponible ainsi, si un processeur doit s'occuper de deux branches, il est possible qu'un autre processeur se libère avant qu'il finisse d'opérer sur l'intégralité des deux branches et ainsi, ce dernier peut alors être appelé par le premier.

On a alors l'algorithme parallèle suivant :

```

algorithme UnionParallèle(Arbre A, Arbre B)
  Arbre C =  $(c_1, \dots, c_n)$ 
  si  $(A = \emptyset)$ 
    retourner B
  si  $(B = \emptyset)$ 
    retourner A
   $k \leftarrow \min(\text{cpuDispos}+1, n)$ 
  pour  $i \leftarrow 2$  à  $k$  en parallèle
     $c_i \leftarrow \text{UnionParallèle}(a_i, b_i)$ 
  pour  $i \leftarrow k + 1$  à  $n$ 
     $c_i \leftarrow \text{UnionParallèle}(a_i, b_i)$ 
  retourner C
  
```

#### 4.4 Stockage sur disque à l'aide d'arbres de radicaux.

Une méthode d'indexation des bases de données par arbre de radicaux est proposée dans [21]. Cette méthode s'est avérée très efficace lors de tests sur le TPC. On cherche à limiter les accès disques par une organisation des indexes en arbre de radicaux. Prenant la valeur d'indexe associé à un objet de la base, on la découpe en bitsets de taille constante fixée à l'avance en fonction de la capacité de stockage que l'on veut obtenir et des performances souhaitées. On organise les fichiers d'indexe par arborescence de répertoires chaque répertoire contenant trois fichiers.

Un fichier de thésaurus ou catalogue des objets, qui associe à un objet une valeur d'offset dans le second fichier représentant un bitset désignant le prochain répertoire à parcourir. Le troisième fichier étant un fichier de permutation donnant un ordre sur les valeurs d'indexe et permettant d'ajouter des objets au thésaurus par simple ajout à la fin du fichier, tout en conservant la possibilité d'une recherche dichotomique à chaque niveau de répertoire. La recherche d'un objet de la base se résume alors en la consultation des fichiers de thésaurus et de permutation pour trouver le bitset du répertoire suivant jusqu'à obtention de la référence à l'objet recherché. En choisissant des identifiants de 40 bits, découpés en 4 bitsets de 10 bits, on peut indexer  $2^{40}$  objets que l'on retrouvera après un parcours au travers seulement 4 répertoires. Chacun des répertoires contenant un nombre raisonnable de sous-répertoires ( $2^{10}$  soit 1024). La figure 11 représente l'organisation des données dans les fichiers et la figure 12 représente l'organisation des fichiers dans le répertoires.

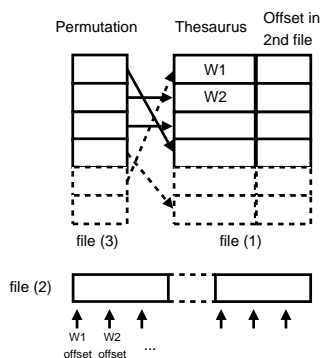


FIG. 11 – Organisation des données dans les fichiers.

## 5 Utilisation des arbres de radicaux pour la recherche des itemsets fréquents.

### 5.1 Insertion d'un item dans un arbre de radicaux.

Disposant d'un codage  $\phi$  de  $E$ , l'ensemble des items, vers l'ensemble des entiers  $\mathbb{N}$ , on peut représenter l'ensemble des items par un arbre de radicaux. On utilise pour cela l'écriture binaire de l'identifiant d'un item et un arbre de radicaux sur un alphabet  $\alpha$  composé des suites binaires de taille  $m$  (si  $m = 1$  alors, l'alphabet est  $\{0, 1\}$ , si  $m = 2$ , il s'agit de l'alphabet  $\{00, 01, 10, 11\}$  etc.). En choisissant un codage sur  $n$  bits avec  $n$  un multiple de  $m$ , on peut décomposer tout identifiant de manière unique en une suite de lettres de  $\alpha$ .

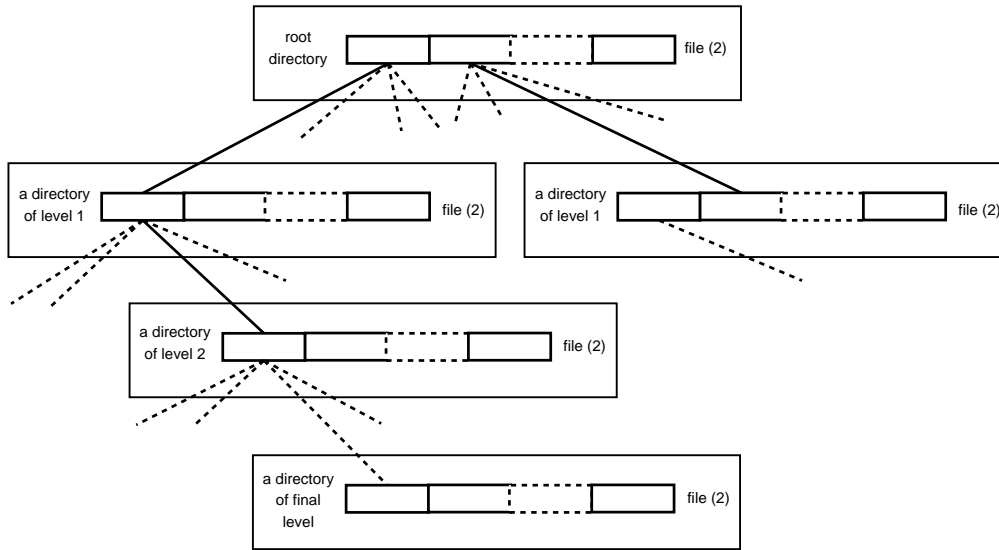


FIG. 12 – Arborescence des répertoires.

Soit  $f$  un noeud de l’arbre de radicaux, on désigne par  $f_i, i \in \{1, \dots, m\}$  les fils de  $f$ . Insérer un item dans l’arbre de radicaux revient à parcourir la décomposition en lettres de  $\alpha$  de son codage, en choisissant le fils  $f_i$  dans lequel on continue l’insertion en fonction des lettres rencontrées. Puisque notre alphabet est une suite binaire, chaque lettre de celui-ci possède une valeur numérique différente de celle de toute autre lettre de  $\alpha$ . Si  $f_i$  est le fils de  $f$  associé à la lettre dont la valeur numérique est  $i$ , on obtient un ordre sur les items induit dans la structure de l’arbre (ordre que j’ai implicitement utilisé depuis le début de ce mémoire). On appellera “position” d’un fils  $f_i$  d’un noeud  $f$  l’indice  $i$  de ce fils et dans les représentations graphiques des arbres, on placera ces fils de gauche à droite dans l’ordre croissant des  $i$ .

## 5.2 Insertion d’ensemble dans les arbres de radicaux.

Disposant d’un codage  $\phi$  de l’ensemble des objets vers l’ensemble des entiers, on peut représenter facilement un ensemble d’objets par un arbre de radicaux (voir 5.1). Un tel codage d’objet est donné en attribuant un identifiant unique à chaque objet, soit explicitement en fixant arbitrairement une valeur différente à chaque objet, soit par hachage avec une fonction adaptée. Les opérations ensemblistes d’union et d’intersection s’effectuent alors, comme nous l’avons vu en 4.2, simplement par union/intersection noeud à noeud des arbres représentant les ensembles. Si ces opérations sont utiles dans les algorithmes de Data-Mining qui fonctionnent sur des ensembles d’items, la génération des candidats nécessite de disposer d’une opération performante de produits d’ensembles.

### 5.2.1 Produit d’ensembles.

Le produit de deux ensembles  $E_1 \times E_2$ , avec  $E_1 = \{a_1, a_2, \dots\}$  et  $E_2 = \{b_1, b_2, \dots\}$  est l’ensemble des couples  $(a_i, b_j)$ .

Étant donné une application bijective  $\phi : E_1 \cup E_2 \longrightarrow \mathbb{N}$ . Le produit de deux tels ensembles codés par  $\phi$  est l’ensemble des couples  $(\phi(a_i), \phi(b_j))$ .

L'enracinement d'un ensemble  $E_2$  représenté par un arbre  $T_2$  à chaque feuille d'un arbre de radicaux  $T_1$  représentant un ensemble  $E_1$  (en utilisant le codage  $\phi$ ), nous fournit un arbre  $T_3$  représentant l'ensemble des couples  $(a_i, b_j)$ .

La figure 13 représente le produit de l'ensemble  $\{A, B, C, D\}$  par lui-même.

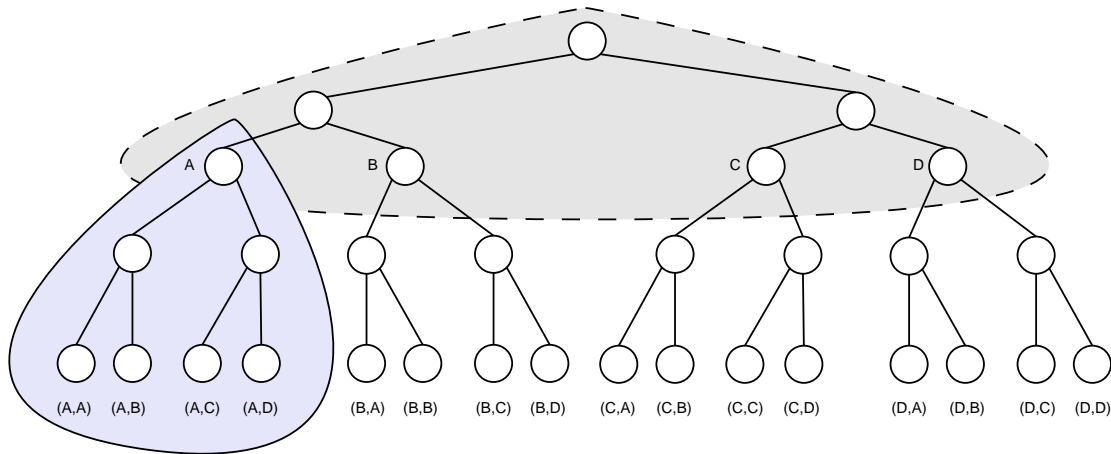


FIG. 13 – Produit d'ensembles par enraccinement.

Pour effectuer un produit d'ensembles, il suffit donc d'enraciner le second membre du produit (pour un produit ordonné) à chaque feuille du premier.

### 5.3 Treillis d'itemsets.

Dans les algorithmes du Data-Mining, on cherche des itemsets, n-uplets d'items sans répétition. Ces itemsets constitueront, un treillis comme nous l'avons vu précédemment. Or, il existe une structure d'arbre de radicaux incluse dans chacun de ces treillis qui recouvre tous les itemsets. La figure 14 présente ce recouvrement du treillis par un arbre de radicaux.

L'arbre qui recouvre le treillis d'itemsets est à identifier à un arbre obtenu par produit d'ensembles. On peut en effet, reconstituer le treillis par produit "partiel" des deux ensembles. Il s'agit d'éliminer les répétitions lors de l'enracinement du second ensemble aux feuilles du premier. Il faut donc procéder à un "élagage" de l'arbre qu'on enraccine : on doit éliminer un certain nombre d'éléments de l'ensemble ajouté à la feuille en fonction de la feuille elle-même. Les itemsets étant sans répétition on doit éliminer de l'arbre construit par enraccinement, les feuilles qui produiront des itemsets équivalents ou contenant plusieurs fois un même item. Ainsi  $ABC$  est, dans la recherche des itemsets fréquents, équivalent aux itemsets  $BAC$ ,  $BCA$ ,  $ACB$ ,  $CBA$  et  $CAB$  tout comme les itemsets  $ABC$  et  $AABC$  etc.

#### 5.3.1 Composition sans répétition des éléments d'un ensemble.

Soit  $E$ , un sous ensemble de  $\{a_1, \dots, a_n\}$ , la composition sans répétition des éléments de  $E$  est l'ensemble  $E^2$  des couples  $(a_i, a_j)$  avec  $a_i, a_j \in E$ ,  $i \neq j$  et  $(a_i, a_j) = (a_j, a_i)$ . La structure d'arbre de radicaux induit un ordre sur les éléments. On supposera, sans perte de généralité, que l'ordre dans l'arbre est l'ordre des indices. Ainsi  $\forall i \in \{1, \dots, n\} i < j \implies a_i < a_j$  suivant l'ordre

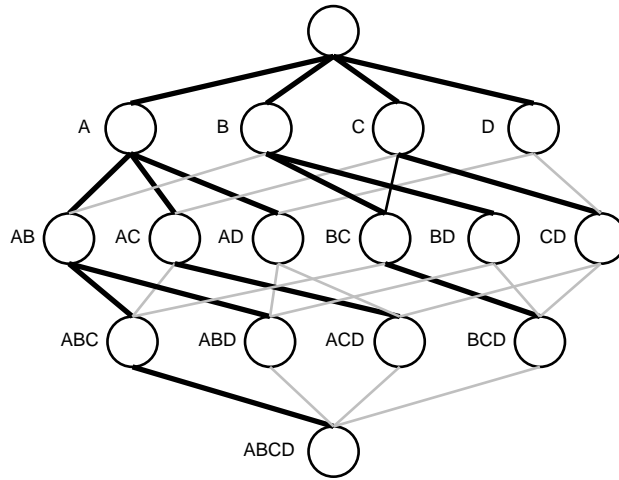


FIG. 14 – Arbre de radicaux recouvrant le treillis d'itemsets.

induit par l'arbre. L'ensemble  $E^2$  est donc constitué des couples  $(a_i, a_j)$  avec  $i < j$ . Ainsi, obtenir la composition des éléments de  $E$  revient à enraciner à chaque feuille  $a_i$  de l'arbre représentant  $E$ , l'arbre privé des  $a_j$  tels que  $j \leq i$ . Il faut donc supprimer  $i$  feuilles de l'arbre complet à l'arbre qu'on enracine en chaque  $a_i$ .

### 5.3.2 Élagage d'arbre de radicaux

On doit utiliser un algorithme qui supprime les  $n$  premiers éléments d'un arbre complet à un arbre quelconque :

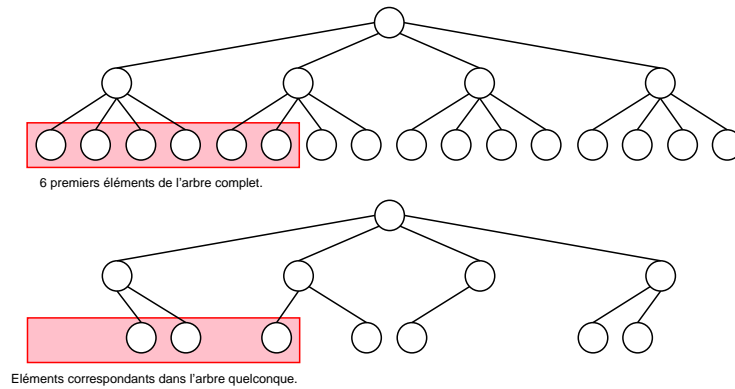


FIG. 15 – Suppression des 6 premiers éléments de l'arbre complet dans un arbre quelconque.

On utilise un algorithme récursif simple prenant en paramètres l'arbre à élaguer, le niveau d'élagage et la hauteur de l'arbre :

```

algorithme Elaguer(Arbre A, entier n, entier l)
  si (n ≠ 0)
    i = 1
    TantQue (n ≥ |alphabet|l-1 × i)
      supprimer(A.fils[i])
      i ← i + 1
    FinTantQue
    Elaguer(A.fils[i], n - |alpha|l-1 × i, l - 1)

```

À la sortie de l'algorithme, l'arbre *A* a été élagué des *n* premières feuilles de l'arbre complet correspondant.

Si *n* est le nombre d'éléments contenus dans l'arbre, la complexité de l'algorithme est en  $O(\log_{|alphabet|}(n))$  (il peut se résumer en la recherche du ou des sous-arbres à supprimer). Lorsqu'on souhaite obtenir une copie élaguée de l'arbre, la complexité est en  $O(n)$ .

**5.4 Génération de candidats.**

**5.4.1 Construction du treillis des itemsets.**

On a vu, que le treillis d'itemsets des candidats contient un arbre de radicaux qui recouvre l'ensemble des itemsets. On va voir, qu'en utilisant l'algorithme d'élagage vu précédemment, on peut construire le treillis des itemsets.

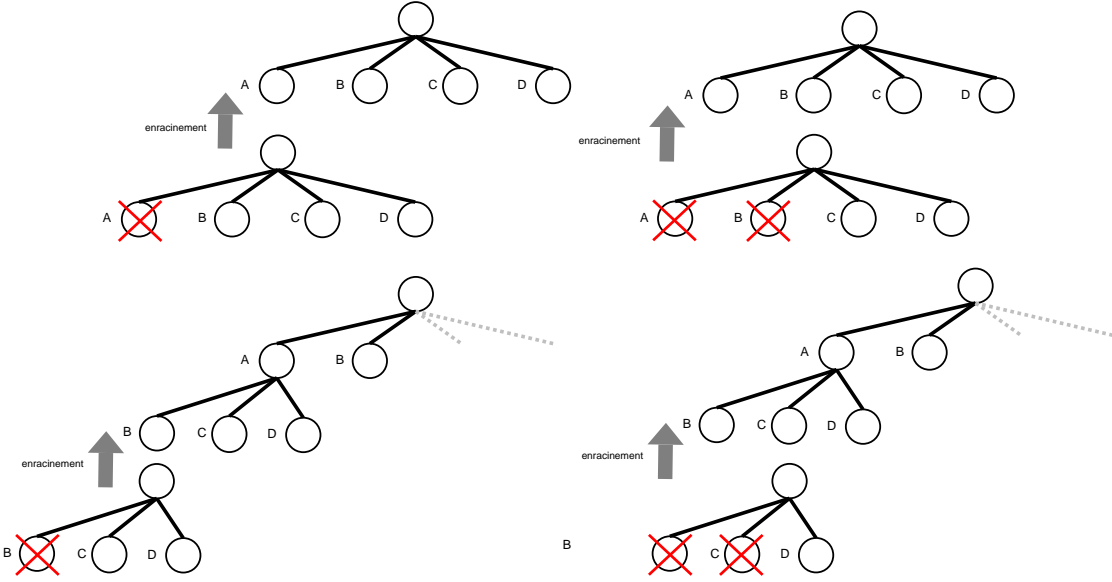


FIG. 16 – Construction du treillis des itemsets.

On construit récursivement le treillis des itemsets, partant de l'arbre des items par l'algorithme suivant, en numérotant chaque feuille de l'arbre par sa position dans l'arbre complet :

```

algorithme Treillis(Arbre A)
pour Chaque feuille f de A
  Enraciner(f, Elaguer(A, f.position, A.hauteur))
  Treillis(f)
FinPour

```

Dans cet algorithme, on enracine à chaque feuille l'arbre contenant la feuille, élagué du nombre de feuille égal à la position de celle-ci si elle se trouvait dans l'arbre complet.

Il est ici, important de noter que dans le cas d'ensembles d'entiers, si on a pris soin de construire la structure de l'arbre de radicaux comme indiqué en 5.1, la position d'un élément dans l'arbre complet est tout simplement la valeur numérique de l'élément.

Puisqu'on enracine des arbres qui contiennent de moins en moins de valeurs, l'opération s'effectue de plus en plus vite.

### 5.5 Transformation verticale de la base.

En se servant des arbres de radicaux, on peut, pour chaque itemset, stocker l'ensemble des transactions dans lesquelles il apparaît. Si on procède ainsi dès l'insertion d'un item dans l'arbre des items qui servira à construire le treillis des itemsets, une seule lecture de la base permet de conserver toutes les informations utiles à la recherche des itemsets fréquents. En effet, une simple intersection des arbres de transactions des différents items d'un itemset nous donne l'arbre de transactions de l'itemset.

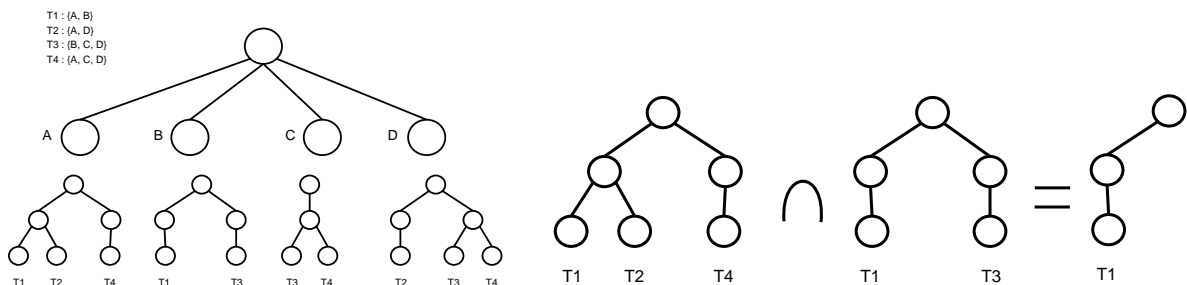


FIG. 17 – Création de l'arbre de transaction d'un itemset.

L'évaluation du support d'un itemset revenant alors à compter le nombre d'éléments de son arbre de transactions, ce qui peut être fait simultanément avec l'intersection. Ainsi, sans augmenter la complexité de l'algorithme, on obtient, après les intersections, l'ensemble des informations nécessaires à la poursuite de la recherche.

### 5.6 Construction du treillis des itemsets fréquents.

En effectuant le filtrage des itemsets non fréquents avant chaque étape d'enracinement de l'algorithme précédent, on obtiendra le treillis des itemsets fréquents.

On ajoute simplement l'évaluation du support et l'élimination des itemsets inférieurs à l'algorithme de construction du treillis des itemsets :

```

algorithme TreillisFréquent(Arbre A)
  pour Chaque feuille  $f$  de A
    si ( $f.support < MinSupport$ )
      Supprimer( $f, A$ )
  pour Chaque feuille  $f$  de A
    Enraciner( $f, Elaguer(A, f.position, A.hauteur)$ )
  pour Chaque feuille  $g$  de  $f$ 
     $g.transactions \leftarrow g.transaction \cap f.transaction$ 
    TreillisFréquent( $f$ )
FinPour

```

Dans cet algorithme, on considère que le calcul du support s'effectue en même temps que l'intersection des arbres de transactions (ce qui se réalise très simplement). On élimine d'abord les itemsets non fréquents afin de limiter la charge de travail le plus tôt possible. Il s'agit ici d'une version simplifiée pour la lisibilité, puisqu'on peut, avec un minimum d'effort, effectuer dans le même temps, l'intersection des arbres de transactions, le calcul du support et l'élimination des itemsets non fréquents.

## 5.7 Recherche des itemsets fréquents.

Afin d'obtenir l'ensemble des itemsets fréquents d'une base de transactions  $\mathcal{T}$ , on doit construire le treillis des itemsets fréquents. On construit ce treillis dynamiquement en ne gardant à chaque niveau que ce qui est nécessaire. En effet, il n'est pas nécessaire de conserver un niveau du treillis après la construction du niveau immédiatement inférieur. On conserve à chaque fois les itemsets fréquents et leurs supports en éliminant la structure du treillis qui ne servira plus.

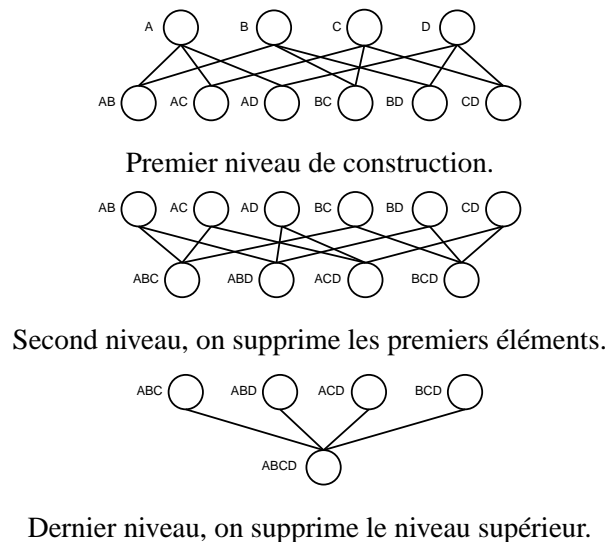


FIG. 18 – Suppression progressive des éléments inutiles du treillis des itemsets.

Pour réaliser la recherche des itemsets fréquents sur une base de transaction  $\mathcal{T}$ , on commence par parcourir la base complète afin de construire l'ensemble de items fréquents. Pour cela, on insère chacun des identifiants d'items dans un arbre de radicaux, accompagné de l'identifiant de la transaction

dans laquelle il apparaît qui sera inséré dans l'arbre de transaction de l'item. On commence alors la construction du treillis d'itemsets fréquents en effectuant à chaque étape le calcul des supports afin d'écartier les itemsets de support insuffisant. Cette opération s'effectue en réalisant l'intersection des arbres de transactions des itemsets que l'on compose en  $O(n)$  avec  $n$  le nombre d'éléments de l'intersection des arbres de transactions de deux itemsets fréquents. Ce nombre est au maximum égal à  $|\mathcal{T}|$  (lorsque l'item apparaît dans toutes les transactions). On élimine alors les itemsets dont le support n'est pas suffisant et on construit les nouveaux candidats par enracinement d'arbres élagués. On a à chaque étape au maximum,  $C_n^{k-1}$  arbres à enraciner pour la construction des itemsets fréquents de taille  $k$  si la base contient  $n$  items. On procède ainsi jusqu'à l'obtention de tous les itemsets fréquents. On élimine au fur et à mesure le niveau supérieur du treillis afin de ne jamais conserver en mémoire que les informations nécessaires à la construction des candidats.

### 5.8 Recherche parallèle des itemsets fréquents.

Comme dans "Count Distribution", chaque processeur ou site travaille sur sa partie locale de la base de transactions. On procédera ensuite aux seuls échanges de supports d'itemsets. À chaque étape, on calcule le support local des itemsets. Si un itemset a un support nul, on procède immédiatement à son élimination. En effet, si un itemset  $A_1A_2 \cdots A_k$  n'apparaît pas dans la base locale, jamais un itemset  $A_1A_2 \cdots A_kA_{k+1}$  n'apparaîtra dans la base... Pour les autres cas, on peut procéder, dans l'attente des supports distants, à une anticipation sur la construction des candidats. Ainsi, si dans la base locale, un itemset a un support supérieur au support minimum, on sait qu'il ne sera pas éliminé et on peut même ignorer ses supports distants. En procédant à un classement des supports au moments de leur envoi aux autres sites, on peut anticiper la construction des candidats par ordre décroissant de support local. Si jamais, on a construit une suite de candidats dont le support global n'était pas suffisant, il suffit de supprimer l'arbre qu'on a enraciné pour construire cette suite de candidats.

### 5.9 L'algorithme parallèle de recherche des itemsets fréquents.

Cet algorithme proposé dans [4] se déroule en deux phases : la phase d'initialisation qui effectue une lecture de la base et la phase de construction des itemsets fréquents.

**Sur chaque processeur** $k \leftarrow 0$ 

Lecture de la base locale et construction de l'arbre des itemsets et de leurs arbres de transactions.

**Faire** $k \leftarrow k + 1$ 

Envoie des supports locaux aux autres processeurs.

**/\* Cette partie peut être désynchronisée afin de prendre de l'avance sur les calculs. \*/**

Reception des supports des autres processeurs.

Calcul des supports globaux.

Élimination des itemsets dont le support est insuffisant dans le treillis.

 $L \leftarrow L \cup \text{dernier niveau du treillis}$ 

Construction du niveau  $k + 1$  du treillis.

Élimination du niveau  $k$

**TantQue** (Le niveau  $k + 1 \neq \emptyset$ )

La première étape de l'algorithme nécessite une lecture complète de la base. La construction des ensembles d'items et de leurs ensembles de transactions s'effectue en temps linéaire par rapport à la taille de la base. En effet, l'insertion d'une valeur dans un arbre de radicaux s'effectue en temps constant (fonction de la taille du codage choisie qui est une constante pour l'exécution de l'algorithme). On construit l'arbre des items et leurs arbres de transactions en parcourant la base et en effectuant deux insertions par item rencontré dans chacune des transactions (l'insertion de l'item et l'insertion de la transaction).

La seconde étape est effectuée  $K$  fois,  $K$  étant la taille du plus grand itemset fréquent de la base. Ce nombre est au maximum égal à  $n$  (le nombre d'items de la base) lorsque les  $2^n$  itemsets sont fréquents. Durant cette étape, chaque processeur attend les supports distants des  $k$ -itemsets qu'il a rencontré dans sa base locale à l'étape  $k$ . Le nombre de messages à cette étape est borné par  $C_n^k$  par processeur. C'est pendant cette étape qu'on peut anticiper la construction des candidats. Les éliminations de candidats s'effectuent en temps constant puisqu'il s'agit d'éliminer des éléments d'un arbre de radicaux de hauteur fixe. La construction des candidats réalisée également à cette étape consiste en un simple enracinement (voir 5.4)

À la sortie de l'algorithme, l'ensemble  $L$  contient l'ensemble des itemsets fréquents.

## 6 Conclusion.

Les bases de données réparties sur les grilles, dans le cadre d'un stockage à grande échelle, offrent un contexte particulier pour l'analyse de données. L'échelle de stockage aussi bien que la diversité possible des sources, nécessitent de très hautes performances, pour la gestion des requêtes et pour l'accès aux données. L'emploi d'arbres de radicaux, structure qui offre d'excellents résultats pour la gestion des opérations ensemblistes ainsi que pour l'indexation des données, s'est avéré particulièrement efficace dans le cadre des algorithmes de recherche de règles d'associations.

L'algorithme proposé dans ce mémoire, qui a donné lieu à la publication d'un article ([4]) utilise une telle structure de données et en tire avantage en plusieurs points. Les opérations ensemblistes de bases (union, intersection) offrent une meilleure complexité par l'emploi de cette structure [21] et sont facilement parallélisables. La génération des candidats par enracinement s'effectue en un nombre toujours minimal d'opérations et permet d'envisager de bénéficier des optimisations physiques des machines puisqu'il s'agit essentiellement de réaliser des copies mémoires.

La distribution des données sur les grilles conduit à envisager plusieurs problèmes. En effet, sur des ensembles regroupant plusieurs milliers de machines, on ne peut plus considérer les pannes et erreurs comme événements exceptionnels. Dans un tel contexte, ceux-ci prennent un caractère chronique et nécessitent d'être pris en compte dès la conception des systèmes. MPICH-V [15] toujours en cours de développement propose une implémentation du Message Passing Interface tolérant aux fautes et permet de gérer celles-ci de manière transparente pour le programmeur aussi bien que pour l'utilisateur. Cette interface permettra à terme une automatisation des gestions de pannes aussi bien d'une seule machine au sein d'un cluster que d'un cluster complet dans une grille. La sécurité des données est un autre problème, loin d'être résolu. En effet, l'interrogation de bases de données nécessite de pouvoir accéder à celles-ci rapidement. Or si les données sont cryptées sur le serveur, on ne peut pas directement les interroger et envoyer l'intégralité des données cryptées à travers le réseau est inenvisageable. Les données consultées par les algorithmes du data mining sont parfois confidentielles et nécessitent la prise en compte de ces problèmes. La gestion des bases de données sur les grilles nécessitera encore beaucoup de travail afin d'offrir les fonctionnalités courantes des SGBDs "traditionnels".

Je tiens à remercier Christophe Cérin et Michel Koskas pour l'attention qu'ils ont portée à mon travail ainsi que pour leur disponibilité tout au long de ce stage. Je remercie également Jean-Sébastien Gay avec qui j'ai collaboré dans la réalisation des travaux présentés dans ce document.

## Table des figures

1	Formes horizontale et verticale de la base. . . . .	9
2	Treillis d'itemsets pour 4 items. . . . .	10
3	Itemsets fermés fréquents dans le treillis des itemsets fréquents. . . . .	10
4	Répartition des charges de travail. . . . .	14
5	Un arbre de radicaux représentant un lexique d'anglais. . . . .	16
6	Un arbre de radicaux représentant l'ensemble {0, 1, 3} en codage binaire sur 3 bits. . . . .	16
7	Un arbre de radicaux représentant l'ensemble {0, 2, 3, 8, 16} codé sur 6 bits. . . . .	16
8	Union de deux arbres de radicaux. . . . .	17
9	Intersection de deux arbres de radicaux. . . . .	18
10	Stratégie de parallélisation de l'algorithme d'union. . . . .	19
11	Organisation des données dans les fichiers. . . . .	20
12	Arborescence des répertoires. . . . .	21
13	Produit d'ensembles par enraccinement. . . . .	22
14	Arbre de radicaux recouvrant le treillis d'itemsets. . . . .	23
15	Suppression des 6 premiers éléments de l'arbre complet dans un arbre quelconque. . . . .	23
16	Construction du treillis des itemsets. . . . .	24
17	Création de l'arbre de transaction d'un itemset. . . . .	25
18	Suppression progressive des éléments inutiles du treillis des itemsets. . . . .	26

[]

## Références

- [1] Rakesh Agrawal, Tomasz Imielinski, and Arun N. Swami. Mining association rules between sets of items in large databases. In P. Buneman and S. Jajodia, editors, *Proceedings of the 1993 ACM SIGMOD Int. Conf. on Management of Data*, pages 207–216, Washington, D.C., 26–28 1993.
- [2] Christophe Le Bret. Connaissez-vous le data mining. *Science Tribune*, Octobre 1997.
- [3] G. Bernot C. Bobineau B. Gennaro B. Finance F. Jouanot Z. Kedad D. Laurent F. Tahy G. Vargas-Solar T.-T Vu C. Collet, K. Belhjjame and X. Xue. Towards a mediation system framework for transparent access to largely distributed sources. In *Actes de DRUIDE*, pages 145–158. LSR-IMAG Lab., INP Grenoble, May 2004.
- [4] M. Koskas C. Cérin, J.-S. Gay and G. Le Mahec. Efficient data-structures and parallel algorithms for association rules discovery. *Conference Parallel Computing System PCS'04*, 2004.
- [5] F. Cappello. Calcul global et desktop grids. In *Actes de DRUIDE*, pages 3–10. INRIA, May 2004.
- [6] Miguel Castro, Peter Druschel, Y. Charlie Hu, and Antony Rowstron. Exploiting network proximity in distributed hash tables. In Ozalp Babaoglu, Ken Birman, and Keith Marzullo, editors, *International Workshop on Future Directions in Distributed Computing (FuDiCo)*, pages 52–55, June 2002.
- [7] Miguel Castro, Peter Druschel, Anne-Marie Kermarrec, Animesh Nandi, Antony Rowstron, and Atul Singh. Splitstream : High-bandwidth content distribution in a cooperative environment. In *IPTPS'03*, February 2003.
- [8] Miguel Castro, Peter Druschel, Anne-Marie Kermarrec, Animesh Nandi, Antony Rowstron, and Atul Singh. Splitstream : High-bandwidth multicast in a cooperative environment. In *19th ACM Symposium on Operating Systems Principles (SOSP'03)*, October 2003.
- [9] Miguel Castro, Peter Druschel, Anne-Marie Kermarrec, and Antony Rowstron. One ring to rule them all : Service discover and binding in structured peer-to-peer overlay networks. In *SIGOPS European Workshop*, September 2002.
- [10] Miguel Castro, Peter Druschel, Anne-Marie Kermarrec, and Antony Rowstron. Scribe : A large-scale and decentralized application-level multicast infrastructure. *IEEE Journal on Selected Areas in Communication (JSAC)*, 20(8), October 2002.
- [11] Miguel Castro, Peter Druschel, Anne-Marie Kermarrec, and Antony Rowstron. Scalable application-level anycast for highly dynamic groups. In *Networked Group Communication, Fifth International COST264 Workshop (NGC'2003)*, September 2003.
- [12] Miguel Castro, Michael B. Jones, Anne-Marie Kermarrec, Antony Rowstron, Marvin Theimer, Helen Wang, and Alec Wolman. An evaluation of scalable application-level multicast built using peer-to-peer overlays. In *Infocom'03*, April 2003.
- [13] Ian Clarke, Oskar Sandberg, Brandon Wiley, and Theodore W. Hong. Freenet : A distributed anonymous information storage and retrieval system. *Lecture Notes in Computer Science*, 2009 :46, 2001.
- [14] Peter Druschel and Antony Rowstron. PAST : A persistent and anonymous store. In *HotOS VIII*, May 2001.

- [15] Franck Cappello Samir Djilali Gilles Fédak Cécile Germain Thomas Hérault Pierre Lemari-nier Oleg Lodygensky Frédéric Magniette Vincent Néri-Anton Selikhov George Bosilca, Aurélien Bouteiller. Mpich-v : Toward a scalable fault tolerant mpi for volatile nodes. In *SuperComputing 2002*, November 2002.
- [16] A. Inkeri Verkamo H. Mannila, H. Toivonen. Discovery of frequent episodes in event sequences. *Data Mining and Knowledge Discovery*, 1(3), 1997.
- [17] T. Hérault. Mpi tolérant aux fautes. In *Actes de DRUIDE*, pages 195–196, May 2004.
- [18] D. Karger F. Kaashoek H. Balakrishnan I. Stoica, R. Morris. Chord : A scalable peer-to-peer lookup service for internet applications. In *ACM SIGCOMM 2001*, San Diego, CA, USA, 2001.
- [19] Sitaram Iyer, Antony Rowstron, and Peter Druschel. Squirrel : A decentralized peer-to-peer web cache. In *12th ACM Symposium on Principles of Distributed Computing (PODC 2002)*, pages ?–?, July 2002.
- [20] Mohamed Jemni. *Méthodologie de parallélisation : conception, algorithmique et programmation & Elaboration d'un système adaptatif pour le e-learning*. PhD thesis, Université de Versailles, 2004. Présentation des travaux pour obtenir l'habilitation à diriger des recherches.
- [21] Michel Koskas. A hierarchical database management algorithm. To appear in the annales du Lamsade, 2004.
- [22] John Kubiatawicz, David Bindel, Yan Chen, Patrick Eaton, Dennis Geels, Ramakrishna Gum-madi, Sean Rhea, Hakim Weatherspoon, Westly Weimer, Christopher Wells, and Ben Zhao. Oceanstore : An architecture for global-scale persistent storage. In *Proceedings of ACM AS-PLOS*. ACM, November 2000.
- [23] G. Karypis V. Kumar M. V. Joshi, S. H. Eui-Hong. Parallel algorithms for data mining. *IEEE Transactions on Knowledge and Data Engineering - Large-scale Parallel and Distributed Data Mining, Lecture Notes in Computer Science/Lecture Notes in Artificial Intelligence (LNCS/LNAI)*, 1759, 2000.
- [24] Ratul Mahajan, Miguel Castro, and Antony Rowstron. Controlling the cost of reliability in peer-to-peer overlays. In *IPTPS'03*, February 2003.
- [25] Ayalvadi Ganesh-Antony Rowstron Miguel Castro, Peter Druschel and Dan S. Wallach. Security for structured peer-to-peer overlay networks. In *5th Symposium on Operating Systems Design and Implementaion (OSDI'02)*, December 2002.
- [26] P. Pucheral. Confidentialité des données. In *Actes de DRUIDE*, pages 167–173. Laboratoire PRiSM Versailles & Projet SMIS, INRIA Rocquencourt, May 2004.
- [27] J.C. Shafer R. Agrawal. Parallel mining of association rules. *IEEE Transactions on Knowledge and Data Engineering*, 8(6), Decembre 1996.
- [28] R. Sikrant R. Agrawal. Fast algorithms for mining association rules. *Proc. 20th Int. Conf. Very Large Data Bases VLDB*.
- [29] Antony Rowstron and Peter Druschel. Pastry : Scalable, distributed object location and routing for large-scale peer-to-peer systems. In *IFIP/ACM International Conference on Distributed Systems Platforms (Middleware)*, pages 329–350, November 2001.
- [30] Antony Rowstron and Peter Druschel. Storage management and caching in PAST, a large-scale, persistent peer-to-peer storage utility. In *18th ACM Symposium on Operating Systems Principles (SOSP'01)*, pages 188–201, October 2001.

- [31] Antony Rowstron, Anne-Marie Kermarrec, Miguel Castro, and Peter Druschel. Scribe : The design of a large-scale event notification infrastructure. In Jon Crowcroft and Markus Hofmann, editors, *Networked Group Communication, Third International COST264 Workshop (NGC'2001)*, volume 2233 of *Lecture Notes in Computer Science*, pages 30–43, November 2001.
- [32] V. Kumar S.H. Eui-Hong, G. Karypis. Scalable parallel data mining for association rules. *IEEE Transactions on Knowledge and Data Engineering*, 12(3), May, June 2000.
- [33] V. Kumar M. SteinBach P.-N. Tan S.H. Eui-Hong, M. V. Joshi. High performance data mining. In J. Hernandez V. Palma J. M.L.M., Dongarra and Sousa, editors, *High Performance Computing for Computational Science - VECPAR 2002*, June 2002.
- [34] O. Soyeze. Us : Prototype de stockage pair-à-pair. In *Actes de RenPar 15*, Octobre 2003.
- [35] H. Toivonen. Sampling large databases for association rules. In *Proceedings of the 22th International Conference on Very Large Data Bases*, pages 134–145. Morgan Kaufmann, 1996. ISBN 1-55860-382-4.
- [36] T. V. Raman U. Erlingsson, M. Krishnamoorthy. Efficient multiway radix search trees. *Information Processing Letters*, 60(3), 1996.
- [37] Jeffrey D. Ullman and Jennifer D. Widom. *First Course in Database Systems, A, 2/e*. Prentice Hall, 2002.
- [38] C. J. Matheus W. J. Frawley, G. Piatetsky-Shapiro. Knowledge discovery in databases - an overview. *AI Magazine*, 13.
- [39] Paul Watson. *Databases and the grid*. University of New-Castle, 2001.
- [40] Mohamed J. Zaki. Parallel and distributed association mining : A survey. *IEEE Concurrency*, Vol. 7, No. 4, October-December 1999, pp. 14-25.
- [41] Mohammed Javeed Zaki, Srinivasan Parthasarathy, and Wei Li. A localized algorithm for parallel association mining. In *ACM Symposium on Parallel Algorithms and Architectures*, pages 321–330, 1997.

## **Index**

élagage, 23

Apriori (algorithme), 7, 8, 10–12

Arbre de radicaux, 9, 15–20

classe d'équivalence, 12

codage, 20

confiance, 6

Count Distribution (algorithme), 8

Data Distribution (algorithme), 8

ECD, 5

Eclat (algorithme), 8, 12–15

horizontal (base), 8

indexation, 20

intersection (d'arbres de radicaux), 17

itemset, 7, 9

itemsets fréquents, 6, 25, 26

KDD, 5

motif, 5, 6

parallèle (algorithme), 18–19, 27

produit d'ensembles, 21

Règles associatives, 5

SGBD (fonctionnalités), 4

support, 5

support minimum, 7

treillis, 22, 24, 25

treillis (d'itemsets), 9

union (d'arbres de radicaux), 16

vertical (base), 8, 14, 25