Application des arbres de radicaux a des algorithmes parallèles du datamining

stage de DEA encadré par :
Christophe Cérin
Michel Koskas

Jean-Sébastien Gay

LARIA
- Université de Picardie *Jules Verne* -

22 juillet 2004

Table des matières

Table des figures

Introduction

Le *Grid Computing* ou calcul sur grille est un domaine de recherche très récent dans le monde de l'informatique. Les premières applications apparues au début des années 90 avaient pour but de partager le temps de calcul des ordinateurs. Le processus était simple, un ordinateur se connectait à un serveur pour obtenir des données à calculer. La première utilisation de ce type de système fut d'abord le cassage de clés cryptographiques. Par la suite de nombreux projets ont vu le jour dont le plus célèbre reste SETI@home. La réussite de ce projet a montré qu'il était possible de réunir une puissance de calcul énorme (évaluée à environ 20 TeraFlops pour le projet SETI) et une grande communauté d'utilisateurs. Avec l'arrivée des systèmes pair-à-pair d'échanges de fichiers, les projets de Grid Computing se sont enrichis de nouvelles perspectives : le partage de toutes les ressources disponibles (CPU, RAM, disque). C'est dans ce contexte que vient s'inscrire le projet *Grid Explorer*.

Le projet Grid Explorer dispose d'un financement d'environ 1 million d'euros et se trouve porté par le CNRS, l'INRIA et le ministère. D'autres laboratoires comme le LARIA y sont aussi fortement impliqués. Le projet Grid Explorer fait lui-même partie du projet *Grid 5000* qui vise à réunir les ressources de 5000 processeurs répartis en dix sites de confiance dans toute la france. On entend par sites de confiance, des sites qui restent connectés avec un débit fiable et dont les principales pannes restent les chutes de disques ou autres pannes matériels. Le fonctionnement est le suivant, des ordinateurs sont connectés à internet et sont prêts à partager une partie ou la totalité de leurs ressources. Pour ce faire, les ordinateurs envoient des informations régulières sur leur disponibilité, celles-ci sont ensuite enregistrées dans plusieurs bases de données hétérogènes. À partir de l'analyse de ces données, il faudra réussir à anticiper le disponibilité des ressources pour gérer au mieux l'ordonnancement des tâches. La partie qui nous intéressera plus particulièrement ici concerne le stockage à grande échelle.

Les logiciels de base de données actuels (Oracle, Postgres...) ne sont pas encore adaptés pour les grilles d'ordinateurs. De ce fait, il existe différentes approches pour réaliser un système de base de données adapté aux grilles. La première d'entre elles, revient à réaliser une interface permettant d'intégrer les systèmes de base de données actuels aux grilles. Les bases de cette solution sont détaillés dans [Wat03]. Cette première approche est l'objet du stage de DEA d'Emmanuel Marty. La seconde méthode consiste à reconstruire entièrement un gestionnaire SQL répondant aux caractéristiques définies dans [UW02]. Dans [Kos04], une partie des opérations fondamentales d'un tel gestionnaire (union, intersection d'ensembles, ...) a été conçue en séquentielle à l'aide d'une nouvelle approche qui tient en l'utilisation d'arbres de radicaux pour implémenter une partie de ces opérations. Cette façon de faire les choses a montré de bonnes performances en passant le TPC (*Transaction Performance council*) par rapport à des gestionnaires SQL commerciaux. Ces gains dus à l'utilisation d'un noyau de programmation pour la gestion des arbres de radicaux s'avèrent très prometteurs.

Notre stage vise au développement de ce noyau en intégrant une gestion multithreadée d'arbres, en analysant systématiquement les choix d'implémentation, en proposant différents langages cibles (C++ et Java) et en le validant sur une application de taille plus raisonnable qu'un gestionnaire complet de base de données : la recherche de règles d'association. Notre but est clairement accès, ici, sur la recherche de performances dans les opérations sur les arbres de radicaux. Ceci passe notamment par le parallélisme mais aussi par des

techniques de compilation que nous rappelerons dans la suite de ce document. Le parallélisme reste cependant la solution principale dans la mesure ou le projet Grid Explorer sur lequel va tourner la bibliothèque est composé de nœuds SMP (*Symmetric Multi-Processing*). Par ailleurs, les fabriquants commencent à annoncer la fabrication de machines quadri et octo processeurs et l'on attend, pour 2005/2006, le lancement des premiers processeurs dual-core (2 processeurs sur le même substrat, partageant la même mémoire). La technique du multi-threading a donc de l'avenir à notre sens.

Ce document sera organisé en quatre parties, de la manière suivante. Dans une première partie, nous ferons une présentation des différents concepts qui vont être abordés tout au long de ce rapport. Nous commencerons par une rapide présentation des arbres de radicaux, puis nous continuerons par une présentation des principaux algorithmes de recherche d'itemsets fréquents. Dans une deuxième partie, nous reviendrons sur les arbres de radicaux. Nous étudierons notamment certaines opérations sur ces arbres, ainsi que leurs avantages par rapport à l'utilisation d'autres structures de données. Dans la troisième partie, nous proposerons différentes politiques de parallélisation et nous évoquerons des pistes pour optimiser une telle structure en temps d'exécution. Dans la quatrième et dernière partie, nous verrons comment se servir d'arbres de radicaux dans les algorithmes de recherche d'itemsets fréquents et nous proposerons un nouvel algorithme qui est l'objet d'un article [CGKM04]. Ce rapport comporte également une conclusion et une annexe dans laquelle nous pouvons trouver une présentation plus technique de notre bibliothèque sur les arbres de radicaux ainsi que notre article [CGKM04].

Chapitre 1

Présentation

Les arbres de radicaux font partie d'une famille de structure de données destinées aux traitements de chaînes de caractères. Cette famille est notamment composée d'arbres plus connus tels que les arbres de préfixes ou les arbres de suffixes mais compte aussi dans ses rangs les tries, les arbres "PATRICIA" ou encore les "digital search trees" et bien d'autres encore... Nous verrons dans un premier temps une présentation de la structure des arbres de radicaux. Puis, dans un second temps, nous verrons une présentation générale des objectifs de la famille de données ainsi que trois algorithmes fondamentaux de cette discipline.

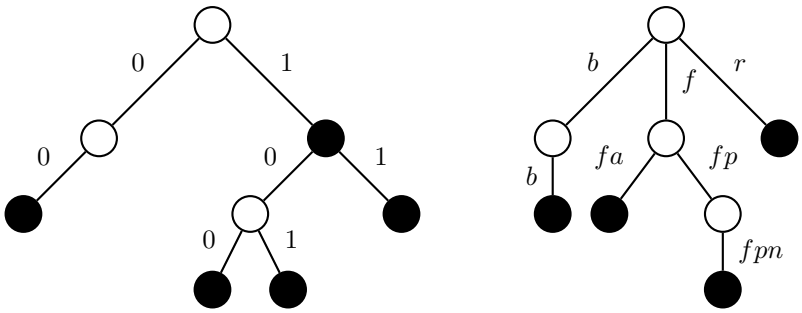
1.1 Arbres de radicaux

La présentation des arbres de radicaux nécessite l'introduction des définitions suivantes :

Préfixe : On dit qu'une chaîne w est un préfixe d'une chaîne x, si $x = wy$ pour une certaine chaîne $y \in \Sigma^*$.

Suffixe : On dit qu'une chaîne w est un suffixe d'une chaîne x, si $x = yw$ pour une certaine chaîne $y \in \Sigma^*$.

Les arbres de radicaux sont utilisés pour stocker des chaînes de caractères sur n'importe quel alphabet [Mei02]. Par exemple, nous pouvons stocker des chaînes de caractères sur un alphabet binaire $\{0, 1\}$ comme des chaînes de caractères sur un alphabet latin $\{a, b, c, \cdots, z\}$ (voir figure 1.1).



a - Alphabet binaire b - Alphabet latin

FIG. 1.1 – Exemple d'arbres de radicaux

Sur la figure 1.1 a, nous voyons que chaque branche gauche représente le caractère 0 et que chaque branche droite représente le caractère 1. Nous constatons aussi la présence de deux types de nœuds à savoir les nœuds blancs et les nœuds noirs. Les nœuds noirs indiquent qu'une chaîne de caractères est stockée à cet endroit-là de l'arbre et respectivement un nœud blanc indique qu'aucune chaîne n'y est stockée. Par

exemple, si l'on cherche la valeur 101 dans l'arbre figure 1.2 alors nous commençons par suivre la branche 1 (droite), puis de là, la branche gauche (0) et enfin la branche droite. Nous vérifions ensuite la couleur de ce nœud, celui-ci étant noir, la chaîne de caractères 101 est donc présente dans l'arbre. À l'inverse, la valeur 10 n'est pas présente dans cet arbre. L'arbre -a- de la figure 1.2 contient donc les chaînes $00, 100, 101$ et 11.

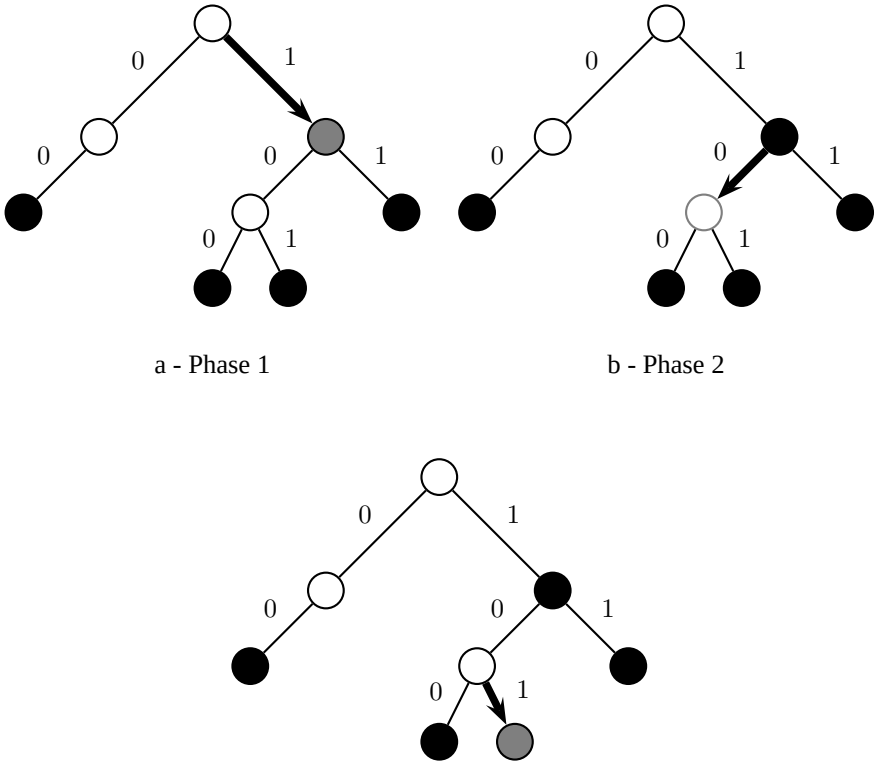


FIG. 1.2 – Recherche de la valeur 101 dans un arbre de radicaux

On peut donc remarquer tout de suite que toutes les feuilles seront forcément de couleur noire et que seuls les nœuds internes pourront avoir l'une de ces deux couleurs. En effet, si une feuille est blanche, cela veut dire que la chaîne qui part de la racine et qui va jusqu'à cette feuille n'existe pas. On peut donc la raccourcir s'il existe une chaîne plus courte ayant le même préfixe dans l'arbre voire la supprimer, si aucune chaîne ayant le même préfixe n'existe. (voir figure 1.3)

Il est nécessaire de préciser que si l'on décide de travailler avec des chaînes de tailles fixes alors le système de couleurs (nœuds blancs, nœuds noirs) devient inutile. En effet, le système de couleurs (ou d'états dans d'autres implémentations) sert à stocker tout en distinguant des mots pouvant être préfixes d'autres mots comme 1 et 11 sur la figure 1.1 a-. Or si les chaînes insérées dans l'arbre ont toutes la même taille alors une chaîne ne peut plus être préfixe d'une autre, elles sont équivalentes. En reprenant la définition de préfixe donné un peu plus haut, si $x = wy$ et $|x| = |w|$ alors $y = \varepsilon$ et $x = w$.

Maintenant que nous avons vu une rapide présentation des arbres de radicaux, nous pouvons isoler quelques remarques d'ordre général et propriétés de ces structures.

- Les arbres de radicaux permettent de stocker des chaînes de caractères sur n'importe quel alphabet.
- Ces chaînes sont stockées de manières ordonnées.

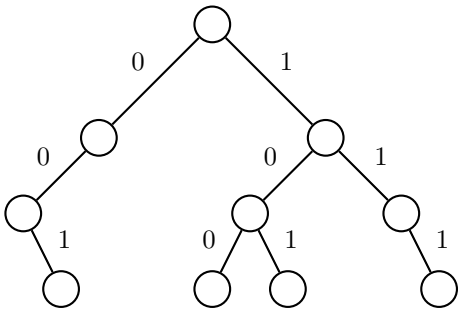


FIG. 1.3 – Exemple d'arbres de radicaux avec chaînes de taille fixe

– Les opérations de recherche et d'insertion dépendendent de la taille de la chaîne de caractères à insérer ou à rechercher.

Dans tout le reste du document, nous travaillerons avec des chaînes de taille fixe, codées sur un alphabet binaire $\{0, 1\}$. Nos arbres ressembleront donc à celui de la figure 1.3. Nous pouvons enrichir les observations précédentes en notant que l'opération d'insertion s'effectura cette fois-ci en un temps constant qui sera égal à la taille de la chaîne à insérer. En ce qui concerne l'implémentation de cette structure de données, nous distinguerons deux types de nœuds : les nœuds internes qui contiennent simplement deux pointeurs vers deux autres fils et les nœuds de type feuille qui se trouvent aux extrémités des branches et qui ne contiennent qu'un pointeur sur un objet quelconque. Dans la partie 2, nous réflechirons à l'emploi d'une telle implémentation et nous proposerons d'autres pistes.

Un autre point intéressant mais qui est plus lié, cette fois, à la structure arborescente d'un arbre est l'utilisation du parallélisme. Nous verrons dans une autre partie comment se servir d'une telle approche mais l'on peut d'ores et déjà remarquer que si deux nœuds ne sont pas sur une même branche alors ils peuvent être considérés comme indépendants l'un de l'autre. Ceci revient à dire que chaque branche droite de l'arbre est indépendante de la branche gauche et réciproquement, d'où la possibilité de traiter les deux sous-branches en parallèle.

1.2 Datamining

Le datamining est une étape du "Knowledge Discovery in Database" (KDD) ou extraction de données qui consiste en la fouille de celles-ci. Il englobe la recherche de règles d'association. C'est cette partie qui nous intéressera principalement ici. Le principe de la recherche de règles d'association est d'extraire des données d'une base de règles en fonction de leur fréquence d'apparition. Ces extractions permettent d'établir des relations et de dégager par conséquent des informations potentiellement utiles. Nous commencerons par rappeler quelques définitions ainsi que certaines notions puis ensuite, nous étudierons trois des algorithmes les plus connus pour la recherche de règles d'association. Un de ces algorithmes se déroule en séquentiel tandis que les deux autres se déroulent en parallèle.

1.2.1 Définitions

Le problème de recherche de règles d'association peut se formaliser de la façon suivante [ZPL97] :

Transaction : Soit $\mathcal{I} = \{i_1, \cdots, i_m\}$ un ensemble de m articles (items) distincts, une $transaction$ est un sous-ensemble de $\mathcal{I}$ et chaque transaction $\mathcal{T}$ possède un identifiant unique dans la base des transactions $\mathcal{D}$. Une transaction est donc de la forme $< TID, i_1, \cdots, i_k >$ et nous appellons $i_1, \cdots, i_k$ un ensemble d'articles ou un $k - ensemble$. Pour une meilleur compréhension, nous utiliserons les

mots anglais à savoir $itemset$ pour ensemble d'objets ou d'articles et $k-itemset$ pour un ensemble de k objets.

Support : On dit qu' un $itemset$ possède un support de s si $s\%$ des transactions de $\mathcal{D}$ contiennent cet itemset. Autrement dit, un ensemble d'objets a un support de s quand la base de données des transactions contient $s\%$ de cet ensemble d'objet.

Règle d'association : Une *règle d'association* est une expression de la forme $A \Rightarrow B$ où $A, B \subset \mathcal{I}$ et $A \bigcap B = 0$.

Confiance : La $confiance$ d'une règle d'association est tout simplement la probabilité conditionnelle qu'une transaction contienne B, sachant qu'elle contient déjà A. Ceci est calculé par le rapport : $support(AB)/support(A)$.

Étant donné m items (articles), il y a potentiellement 2^m itemsets dont le support se situe au dessus d'un support donné. Il est donc inconcevable de devoir tous les énumérer. Cependant, en pratique, il n'existe qu'une toute petite fraction de ces ensembles répondant à cette condition. Ceci requiert donc une attention particulière pour réduire les coûts mémoires et les coûts d'entrées/sorties dans les algorithmes utilisés. Nous allons voir dans la suite de cette partie trois algorithmes pour résoudre ce problème de recherche de règles d'association.

1.2.2 Algorithme séquentiel : A priori

L'algorithme séquentiel *Apriori* est à la base [AIS93] de tous les algorithmes parallèles de recherche de règles d'association [Zak99, JA]. Il utilise le fait qu'un sous-ensemble d'itemsets fréquents est aussi un itemset fréquent. Donc à partir de ce principe, seuls les candidats retenus au cours d'un premier parcours sont utilisés pour générer de nouveaux candidats.

Cet algorithme comporte trois étapes importantes. Ces trois étapes sont réitérées tant que de nouveaux candidats sont générés :

- Construction de l'ensemble des nouveaux candidats.
- Évaluation du support pour chaque nouveau candidat.
- Suppression des candidats dont le support n'est pas suffisant. Le support minimum est quant à lui choisi arbitrairement.

L'algorithme séquentiel complet est le suivant :

The Apriori algorithm

$L_1 = \{$ frequent 1-itemset$\}$;
for ($k = 2$; L_{k-1} ; $\neq \varnothing$; $k++$)
  $C_k =$ Set of new Candidates ;
  for all transaction $t \in D$
    for all k-subsets s of t
      if ($s \in C_k$) s.count++ ;
  $L_k = \{c \in C_k \mid c.\text{count} \geq \text{minimum support}\}$;
Set of all frequent itemsets $= \bigcup_k L_k$;

Remarque 1.2.1 *Dans cet algorithme, la base entière est lue et re-lue à chaque itération de l'algorithme (voir l'instruction* **pour** `tous` k`-sous-ensembles` s `de` t*). Par conséquent, les performances ne peuvent pas être bonnes sur une grosse base de données comme celle que nous allons traiter.*

1.2.3 Algorithmes parallèles

Count Distribution

L'algorithme parallèle *Count Distribution* est simplement une version parallèle de l'algorithme *Apriori*. Chaque processeur possède une portion de la base de données et chacun d'eux calcule des candidats "locaux", évaluent des supports locaux et les transmettent (broadcast all-to-all) aux autres processeurs qui calculent chacun le support global de l'itemset. Étant donné que seules les valeurs des supports doivent être transmises à un processeur dédié, les coûts de communication sont minimisés dans cet algorithme.

Les performances des algorithmes *Apriori* ou *Count Distribution* sont limitées, même en parallèle, pour diverses raisons. Tout d'abord, ces différents algorithmes (qui pour rappel ont la même base) nécessitent des lectures de la base entière à chaque itération. En outre, ils énumèrent tous les itemsets candidats autant de fois qu'ils sont trouvés dans la base de données même si les transactions sont identiques.

En plus de tout ceci, la base de données de transaction est considérée comme ayant une conception horizontale : une transaction possède un identifiant, suivi des items qu'elle contient. Il apparaît que ce type d'organisation de données n'est pas très adapté pour la phase d'évaluation du support de ces algorithmes. En effet, si l'on doit chercher un k-sous ensemble d'une transaction de taille s, cela implique de tester les $\begin{pmatrix} s \\ k \end{pmatrix}$ sous ensembles de la transaction. Un autre algorithme utilisant une transformation verticale de la base de données élimine ce problème comme nous allons le voir maintenant.

Eclat

Un des avantages de l'algorithme *Eclat* [ZPL97] comparé à l'algorithme *Count Distibution* est qu'il ne fait que deux parcours seulement de la base de données. Il parcourt une première fois la base pour construire les 2-itemsets puis une seconde fois pour la mettre sous forme verticale. L'algorithme Eclat est composé de trois phases principales :

- La phase d'initialisation : construction globale des 2-itemsets.
- La phase de transformation : partitionnement de l'ensemble des 2-itemsets fréquents et distribution de ces partitions aux autres processeurs. Transformation verticale de la base.
- La phase asynchrone : construction des k-itemsets fréquents.

Voici une présentation formelle de l'algorithme :

The Eclat algorithm

Initialisation phase :

Scan local database partition
Compute local counts for all 2-itemsets
Construct global L_2 count

Transformation phase :

Partition L_2 into equivalence classes
Schedule L_2 over the set of processors P
Transform local database into vertical form
Transmit relevant tid-lists to other processors

Asynchronous Phase :

for each equivalence class E_2 in local L_2
  ComputeFrequent(E_2)

Final Reduction Phase :

Cet algorithme utilise un partitionnement de la base de données en classes d'équivalences. Les classes d'équivalences sont basées sur un préfixe commun. Il faut considérer le fait que les itemsets sont triés dans l'ordre lexicographique. Par exemple, AB, AC et AD sont dans la même classe d'équivalence S_A car ils possèdent tous le préfixe commun A.

Les itemsets candidats peuvent être générés en joignant les membres d'une même classe d'équivalence. Sur notre exemple, les prochains candidats, de longueur 3, nommés C_3 sont : ABC et ABD. Nous pouvons remarquer que des itemsets produits par une classe d'équivalence sont toujours différents de ceux produits par une classe d'équivalence différente, de plus le partitionnement par classe d'équivalence peut être utilisé pour répartir le travail sur les processeurs. Cette méthode est utilisée dans d'autres algorithmes tels que *Candidate Distribution* et dans un autre algorithme que nous verrons un peu plus loin.

La phase de transformation est souvent considérée comme étant la phase la plus coûteuse de l'algorithme. En fait, les processeurs doivent diffuser à tous les autres processeurs leur liste locale correspondant aux identifiants de transactions pour tous les itemsets.

Dans le chapitre 3, nous verrons comment se servir des arbres de radicaux pour représenter les items et les transactions. Ceci nous amènera à proposer un nouvel algorithme pour la recherche d'itemsets fréquents. Avant ça, il est necessaire d'examiner plus en détail les arbres de radicaux.

Chapitre 2

Implémentation des arbres de radicaux

Après avoir vu au début de ce document une brève présentation des arbres de radicaux, nous allons maintenant étudier plus en profondeur cette structure de données. Dans un premier temps, nous étudierons des opérations avancées telles que l'union, l'intersection et l'élagage. Dans un second temps, nous examinerons l'avantage des arbres de radicaux par rapport à d'autres structures telles que les listes chaînées.

2.1 Opérations sur les arbres de radicaux

Dans cette partie nous allons voir en détail différentes opérations sur les arbres de radicaux contenant des chaînes de caractères de taille fixe et sur un alphabet binaire. Nous donnerons également pour chaque opération, les complexités. Dans la partie suivante, nous établirons une comparaison des arbres de radicaux avec d'autres structures de données.

2.1.1 Union

Mathématiquement, un arbre de radicaux binaire peut se formaliser ainsi :

$$\mathcal{A} = \varnothing \cup (\mathcal{A}, \mathcal{A})$$

L'opération d'union de deux arbres de radicaux est définie par :

$$\begin{aligned} \mathcal{A} \cup \varnothing &= \mathcal{A} \\ \varnothing \cup \mathcal{A} &= \mathcal{A} \\ \mathcal{A}_1 \cup \mathcal{A}_2 &= (\mathcal{A}_{1_g}, \mathcal{A}_{1_d}) \cup (\mathcal{A}_{2_a}, \mathcal{A}_{2_d}) \\ &= (\mathcal{A}_{1_g} \cup \mathcal{A}_{2_g}, \mathcal{A}_{1_d} \cup \mathcal{A}_{2_d}) \end{aligned}$$

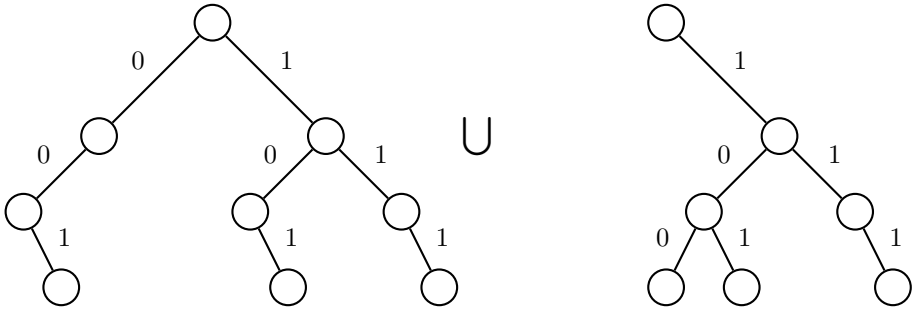
Voici l'algorithme qui en découle :

Algorithme Arbre *Union*(Arbre A1, arbre A2)

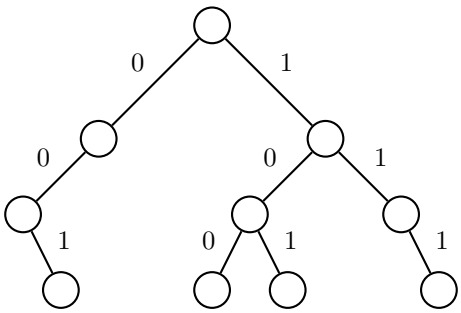
Arbre res
si (A_1 == vide)
alors
    retourner A_2
sinon
    si (A_2 == vide)
    alors
        retourner A_1
    sinon
        res.fg = *Union*(A_1.fg, A_2.fg)

res.fd = *Union*(A_1.fd, A_2.fd)
retourner res
fin si

L'algorithme d'union consiste à parcourir les deux arbres en même temps et sur les mêmes branches. À chaque nœud parcouru, nous en créons un nouveau dans l'arbre résultat. Si une branche existe dans un arbre mais pas dans l'autre alors nous l'ajoutons dans l'arbre résultat. Ce qui fait que le pire des cas consiste à avoir deux arbres totalement disjoints. Dans ce cas, nous serons obligés de parcourir la totalité des deux arbres. La complexité de l'opération d'union est donc en $\mathcal{O}(m+n)$ avec m et n le nombre de nœuds de chaque arbre.



a - Arbre 1 b - Arbre 2



c - Résultat

FIG. 2.1 – Union

On peut remarquer sur la figure 2.1 que l'union de deux arbres de radicaux représentant des ensembles donne un arbre de radicaux représentant l'union de ces ensembles. En effet, sur l'exemple, l'arbre 1 contient l'ensemble $[001, 101, 111]$, l'arbre 2 contient l'ensemble $[100, 101, 111]$ et l'arbre résultat contient bien l'ensemble $[001, 100, 101, 111]$ qui est la réunion des ensembles representés par les arbres 1 et 2.

2.1.2 Intersection

En réutilisant la définition mathématique d'un arbre de radicaux binaire donnée dans la partie précédente, nous pouvons écrire l'opération d'intersection de deux arbres de radicaux.

Mathématiquement l'opération d'intersection de deux arbres de radicaux est définie par :

$$\begin{aligned} \mathcal{A} \cap \varnothing &= \varnothing \\ \varnothing \cap \mathcal{A} &= \varnothing \\ \mathcal{A}_1 \cup \mathcal{A}_2 &= (\mathcal{A}_{1_g}, \mathcal{A}_{1_d}) \cap (\mathcal{A}_{2_g}, \mathcal{A}_{2_d}) \\ &= (\mathcal{A}_{1_g} \cap \mathcal{A}_{2_g}, \mathcal{A}_{1_d} \cap \mathcal{A}_{2_d}) \end{aligned}$$

Voici l'algorithme qui en découle :

Algorithme Arbre *Intersection*(Arbre A_1, Arbre A_2)

Arbre res
si (A_1 == vide) ou (A_2 == vide)
alors
    retourner vide
sinon
    res.fg = *Intersection*(A_1.fg, A_2.fg)
    res.fd = *Intersection*(A_1.fd, A_2.fd)
    retourner res
fin si

L'algorithme d'intersection consiste à parcourir deux arbres en même temps et sur les mêmes branches. Seules les branches communes feront partie du résultat. Donc le cas où l'opération est la plus coûteuse est celui où les arbres de radicaux auront le plus de branches communes. La complexité dépend alors du plus petit des deux arbres. L'opération d'intersection est en $\mathcal{O}(min(m, n))$ avec m et n, le nombre de nœuds des deux arbres.

Comme pour l'union, l'intersection de deux arbres de radicaux représentant des ensembles a pour résultat un arbre de radicaux représentant l'intersection de ces ensembles. Sur l'exemple de la figure 2.2, l'arbre 1 contient l'ensemble $[001, 101, 111]$, l'arbre 2 contient l'ensemble $[100, 101, 111]$ et l'arbre résultat contient bien l'ensemble $[101, 111]$ qui est l'intersection des ensembles des arbres 1 et 2.

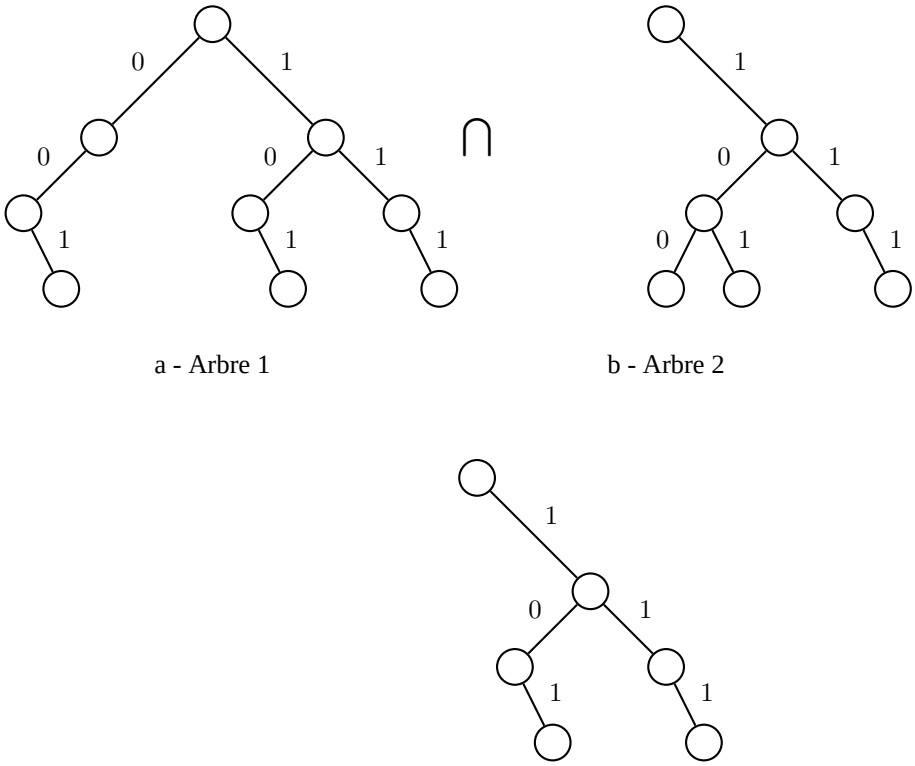
2.1.3 Élagage

L'opération d'élagage consiste à supprimer les n premières feuilles d'un arbre en considérant que celui-ci est toujours complet (voir figure 2.3 p.13). L'élégage revient donc à rechercher le(s) sous-arbre(s) à supprimer. Cette opération, nous sera très utile par la suite dans la mise au point d'un algorithme de data-mining.

Algorithme Arbre *Elaguer*(Arbre A, Entier n, Entier h)

si $(n \neq 0)$
alors
    $i = 1$
    tant que $(n \geq |alphabet|^{h-1} \times i)$
        supprimer(A.fils[i])
        $i \leftarrow i + 1$
    fin tant que
    $Elaguer(A.\text{fils}[i], n - |alphabet|^{h-1} \times i, h - 1)$
fin si

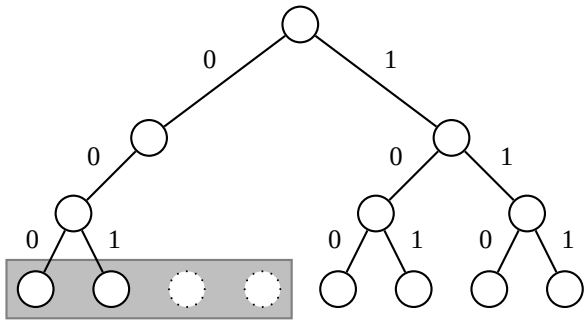
Cet algorithme a été conçue pour un alphabet de taille quelconque. La complexité de l'élagage est proche de celle d'une recherche d'un élément. Nous ne sommes pas ici en temps constant car le sous-arbre a éliminé peut être trouvé très rapidement. Par exemple, si nous disposons d'un arbre binaire de hauteur 3, il dispose d'au maximum 8 feuilles et si nous voulons supprimer les 4 premières feuilles (en considérant qu'il est complet et non par rapport au nombre d'éléments dans l'arbre), il suffit alors d'un seul tour pour



a - Arbre 1 b - Arbre 2

c - Résultat

FIG. 2.2 – Intersection



On considère l'arbre complet lors de l'élagage.

FIG. 2.3 – Élagage des 4 premières feuilles

supprimer le sous-arbre gauche. Au pire, l'algorithme nécessite d'aller jusqu'à une feuille. La complexité est donc en $\mathcal{O}(h)$ où h est la hauteur de l'arbre. La hauteur étant égale à $log_{|alphabet|}(n)$ avec n, la valeur à insérer, une autre expression de la complexité est alors $\mathcal{O}(log_{|alphabet|}(n))$.

2.2 Avantages des arbres de radicaux

Un des premiers avantages liés à l'utilisation des arbres de radicaux se trouve au niveau des coûts mémoires. Grâce à cette structure, nous pouvons stocker de manière simple et ordonnée des nombres codés sur n'importe quelle taille. Imaginons, que l'on veuille stocker le nombre 4 sur 5 bits, nous aurons un arbre de hauteur 4 avec la chaîne 00100. Nous pouvons procéder de la sorte pour coder les nombres sur 4 bits comme sur 40. Si nous comparons à une liste chainée ou un tableau, nous aurons alors un nombre par nœud ou indice. En appelant h, la taille du codage et n, le nombre d'éléments dans la liste, la taille mémoire nécessaire sera alors de $h \times n$. Avec un arbre de radicaux, les préfixes communs ne seront stockés qu'une seule fois. Par exemple, sur 4 bits, 4 et 5 vont donner une branche 0010 avec ensuite deux feuilles 0 et 1. Nous économisons donc 4 bits.

Dans le cas optimal où l'arbre de radicaux est complet (chaque nœud interne possède 2 fils), la taille mémoire nécessaire pour stocker l'arbre sera égale à $2n - 1$ (nombre de nœuds dans un arbre binaire complet). L'arbre complet est le meilleur des cas car la ré-utilisation des préfixes est maximale. $2n - 1$ est plus avantageux que $h \times n$ dès que l'on dépasse une taille de codage égale à 2 bits.

Dans le cas où l'arbre n'est pas complet, il est dur de chiffrer précisement les gains car le nombre d'éléments insérés ne peut nous renseigner sur le nombre total de nœuds dans l'arbre. En revanche, on est sûr que dans le pire des cas, nous économisons un bit à chaque fois que le nombre de valeurs insérées est égal à 1 plus une puissance de 2. Ainsi, si nous rentrons 2 valeurs, l'économie est de 0. Avec 3 valeurs, on économise un bit (on se ressert d'un préfixe la racine n'ayant que 2 fils). Avec 5 valeurs, 2bits etc... En pratique, les valeurs sont mieux réparties, ce qui nous permet d'économiser de l'espace mémoire beaucoup plus vite.

Maintenant que nous avons vu l'avantage des arbres de radicaux sur l'espace mémoire, il est temps d'analyser les coûts des opérations.

Nous avons déjà vu précédemment que les opérations d'insertion et de recherche s'effectuaient en temps constant sur les arbres de radicaux. Ce temps nous avions dit est égal à la hauteur de l'arbre, soit en $\Theta(h)$. La recherche dans un ensemble est en revanche en $\mathcal{O}(n)$ si celui-ci n'est pas trié et en $\mathcal{O}(log(n))$ si celui-ci est trié. Les arbres de radicaux sont donc plus adaptés à la recherche d'éléments dans un ensemble.

Considérons maintenant l'opération d'intersection. Nous nous arrêterons pour ce rapport à l'opération d'intersection car c'est la plus importante pour les algorithmes de recherche de règles d'associations. Si l'on dispose de deux ensembles non triés comportant respectivement n et m éléments, l'intersection la plus efficace consiste à trier l'un des deux ensembles et de rechercher tous les éléments de l'ensemble non trié dans l'ensemble trié. Le coût du tri de l'ensemble de taille n est en $\mathcal{O}(nlog(n))$ et la recherche des m éléments dans le premier ensemble est en $\mathcal{O}(mlog(n))$. Le coût total est donc en $\mathcal{O}(nlog(n)+mlog(n)) = \mathcal{O}((n+m)log(n))$. Cette méthode est la plus efficace si n est supérieur à m.

Pour les arbres de radicaux, nous avons vu que la complexité est en $\mathcal{O}(min(a_1, a_2))$ mais avec a_1 et a_2 étant le nombre de noeuds respectifs dans les arbres. Pour pouvoir comparer avec la méthode d'intersection précédente, prenons le pire des cas : l'intersection de deux arbres complets. Celle-ci est alors en $\Theta(a_1)$ avec $a_1 = a_2$. Comme les arbres sont complets, si l'on appelle n, le nombre d'éléments stockés dans l'arbre, a_1 est alors égal à $2n - 1$. La compléxité en fonction du nombre de feuilles dans le pire des cas (les deux arbres complets) est en $\Theta(2n - 1)$. Si nous comparons le même cas avec la méthode vue précédemment, la compléxité de l'intersection est en $\Theta(2nlog(n))$ avec $log(n)$ est égal à la taille du codage en base 2,

soit la hauteur de l'arbre. Par exemple, si l'on choisit une taille de codage égale à 24, on pourra stockée 2^{24} valeurs. La complexité de l'intersection d'ensembles sera alors en $\Theta(48n)$ contre $\Theta(2n-1)$ pour l'intersection d'ensemble par arbres de radicaux, soit un coefficient divisé par 24. De cette comparaison, on peut déduire deux propriétés valables quelque soit le nombre d'éléments dans les ensembles :

- Plus on augmente le nombre d'éléments, plus les arbres de radicaux seront performants par rapport aux autres structures.
- Plus on augmente la taille du codage, plus les arbres de radicaux seront performants par rapport aux autres structures.

Chapitre 3

Parallélisation et Optimisations

Aprés avoir étudié les opérations génériques sur les arbres de radicaux, nous pouvons présenter des méthodes pour les paralléliser. Dans une première partie, nous étudierons donc différentes méthodes pour paralléliser certaines opérations sur les arbres. Puis dans une seconde partie, nous essaierons de découvrir de nouvelles pistes pour optimiser les temps d'exécution de ces opérations.

3.1 Parallélisation

Dans la première partie, lors de la présentation des arbres de radicaux, nous avions soulevé le fait qu'une structure arborescente possédait des particularités qui semblaient se prêter avantageusement au parallélisme. Avant d'examiner plus en détail ces particularités, il convient de rappeler brièvement notre choix d'implémentation. Un arbre est composé de deux sortes de nœuds : les nœuds internes qui contiennent chacun deux pointeurs sur un nœud (l'un représentant la branche 0 et l'autre la branche 1) et de nœuds de type feuille qui ne contiennent chacun qu'un pointeur sur un objet.

3.1.1 Rappels

Propriétés

Afin de pouvoir analyser au mieux nos méthodes de parallélisation, il convient de rappeler quelques propriétés sur les arbres.

Propriété 3.1.1 *Soit $\mathcal{A}$, un arbre de radicaux, complet, de hauteur h et codé sur un alphabet binaire alors le nombre de feuilles dans $\mathcal{A}$ est égale à 2^h.*

► **Preuve :** Si $h = 0$ alors l'arbre possède $2^0 = 1$ feuille.
La propriété est vraie pour $h = 0$.
Supposons que la propriété soit vraie pour une hauteur p, on a alors :
nombre de feuilles dans $\mathcal{A}$ égal à 2^p.
Au rang suivant $p + 1$, toutes les 2^p feuilles auront 2 feuilles chacunes,
ce qui se traduit par $2^p \times 2 = 2^{p+1}$.
La propriété est donc vérifiée au rang $p + 1$.

Propriété 3.1.2 *Soit $\mathcal{A}$, un arbre de radicaux, complet, de hauteur h et codé sur un alphabet binaire alors le nombre de branches dans $\mathcal{A}$ est égale à $\sum_{i=1}^{h} 2^i$.*

► **Preuve :** Si $h = 1$ alors l'arbre possède $\sum_{i=1}^{1} 2^i = 2^1 = 2$ branches.
La propriété est vraie pour $h = 1$.
Supposons que la propriété soit vraie pour une hauteur p, on a alors :
nombre de branches dans $\mathcal{A}$ égal à $\sum_{i=1}^{p} 2^i$.
Au rang suivant $p + 1$, toutes les 2^p feuilles auront 2 feuilles chacunes,
l'arbre aura donc 2^{p+1} feuilles au rang $p + 1$ (propriété 3.1.1).
Il y aura donc 2^{p+1} branches pour rattacher les 2^{p+1} feuilles.
Ce qui se traduit par :
$\sum_{i=1}^{p} 2^i + 2^{p+1} = 2^0 + 2^1 + \cdots + 2^p + 2^p + 1 = \sum_{i=1}^{p+1} 2^i$
La propriété est donc vérifiée au rang $p + 1$.

Thread

Pour réaliser la parallélisation de ces opérations, nous avons utilisé des *threads*. Les threads peuvent se traduire par "fil d'exécution" ou encore par "processus léger". Les threads exécutent un programme donné en parallèle. Si une machine ne dispose que d'un seul processeur, le système se charge alors d'ordonnancer les différents processus pour donner l'impression d'un travail en paralléle ou pour masquer les latences mémoires. Les threads d'une même application partagent un même espace d'adressage, il faut donc faire attention aux accès concurrents lors d'écritures dans des variables qui leurs sont communes.

Dans un premier temps, nous allons présenter une première idée, nous ne nous fixerons pas de limites sur le nombre de threads disponibles ni même sur le nombre de processeurs. Puis, dans un second temps, nous verrons comment adapter cette méthode à un environnement disposant de ressources plus limitées. Enfin, nous donnerons une autre méthode de gestion des threads.

3.1.2 Première méthode

Lorsque l'on observe un arbre de radicaux complet (chaque nœud interne possède deux fils), on peut remarquer une symétrie entre les branches de l'arbre. Deux branches réunies par un même père sont identiques mais n'ont aucun autre lien entre elles. On pourrait presque dire qu'elles sont indépendantes l'une de l'autre (figure 3.1).

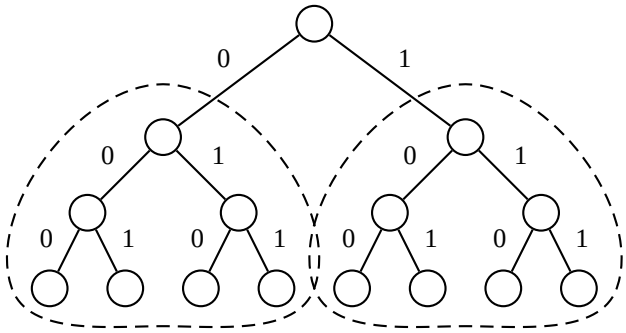


FIG. 3.1 – Parallélisation

L'idée de base pour paralléliser les opérations d'union, d'intersection et de copie d'arbres consiste à se servir le plus simplement de cette propriété. Si l'on prend le cas de l'union par exemple, nous pouvons dire que l'union de deux arbres peut se voir comme l'union de leur racine puis comme l'union de leur sous-arbre gauche et de leur sous-arbre droit. L'union des sous-arbres gauches et l'union des sous-arbres droits sont deux opérations indépendantes l'une de l'autre, elles peuvent donc être effectuées en parallèle. Étant donné que l'opération d'union, tout comme les deux autres s'effectue de manière récursive, on peut réitérer ce procédé de parallélisation des tâches pour chaque nœud interne de l'arbre. Dans la pratique, nous allons confier chaque sous-arbre à un thread comme le montre la figure 3.2 ci-dessous.

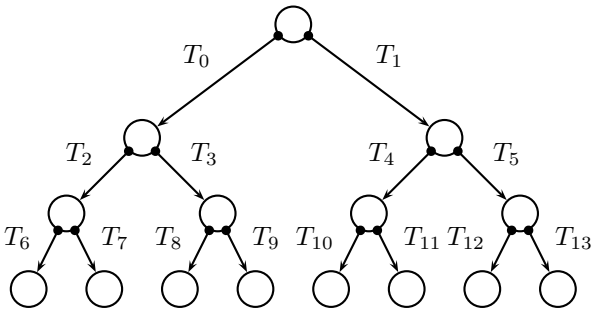


FIG. 3.2 – Première méthode

Grâce aux propriétés précédentes, il devient facile de calculer le nombre de threads que nous pouvons lancer sur un arbre binaire complet et sans limitation de ressources. Étant donné que nous lançons un thread sur chaque branche de l'arbre, le nombre de threads total est égal au nombre de branches de cet arbre, soit $\sum_{i=1}^{h} 2^i$ (h, hauteur de l'arbre) .

Après avoir vu ce qu'il se passait dans le cas où nous travaillons dans des environnements disposant de ressources illimitées, nous pouvons nous demander ce qu'il adviendrait dans un environnement plus proche de la réalité. Après tout, les ordinateurs ne sont pas tous équipés de multi-processeurs et nous pouvons légitimement nous interroger sur l'utilité de lancer 40 threads si l'on ne dispose que de 2 processeurs. Nous allons donc maintenant étudier ce qu'il se passe dans le cas où le nombre de processus dont nous disposons est limité.

Lorsque notre nombre de processus est limité, il n'y a en fait pas grand chose à faire pour que notre programme de base s'adapte à une telle contrainte. Il suffit, de créer une variable qui puisse renseigner à tout instant du nombre de threads disponibles. Cette variable est bien sûr accessible par tous les threads, elle est donc sujette aux problèmes d'exclusions mutuelles. il existe différentes solutions pour résoudre ce problème (mutex, sémaphores, $\cdots$). Pour le reste, à chaque fois que l'on veut lancer un thread, c'est à dire pour chaque sous-arbre d'un nœud, le programme demande si il y en a un de disponible. Si il ne peut en disposer, ce sera à lui de faire le travail qu'aurait du faire le processus fils.

Par exemple, si l'on veut faire une copie d'un arbre et que l'on ne dispose que d'un seul thread, la démarche sera la suivante. Le programme principal va d'abord copier la racine, puis il va ensuite demander à lancer un thread sur le sous-arbre gauche. Comme un thread est disponible, il obtiendra une réponse positive, le thread sera alors lancé sur le sous-arbre gauche de la racine. Le programme principal ne disposant plus de threads, il devra faire lui-même la copie du sous-arbre droit. Le thread créé ne pouvant plus lancer d'autres threads faute de ressource (rappel 1 thread dispo), il devra faire lui-même la copie de tout son sous-arbre (figure 3.3). À chaque passage de niveau dans l'arbre, nous vérifions qu'aucun thread n'est à nouveau disponible. En effet, dans la majorité des cas, les arbres ne sont pas complets, par conséquent, un thread à des chances de finir son exécution bien avant la fin de l'opération complète. Il est donc judicieux de chercher à le ré-utiliser.

La solution détaillée précédemment marche sans problème quelque soit le type d'arbres et de ressource(s). Pour autant, elle présente quelques inconvénients sur certains points. Dans les prochains paragraphes, nous détaillerons ces faiblesses et nous introduirons une nouvelle méthode pour paralléliser nos opérations.

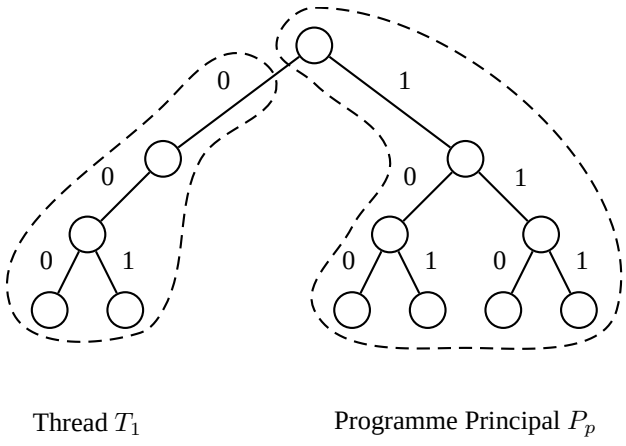


FIG. 3.3 – Exemple avec 1 thread disponible

Parmi les inconvénients de cette méthode, voici le premier : admettons que l'on dispose d'une machine comportant trois processeurs et que l'on veuille réaliser une copie d'un de nos arbres à l'aide de trois threads. L'exécution se déroulera ainsi : le programme principal va faire une copie de la racine puis lancer chacun des deux threads sur chacun des sous-arbres. À partir de cet instant, le programme principal va se mettre en attente de la fin de ses deux threads fils.

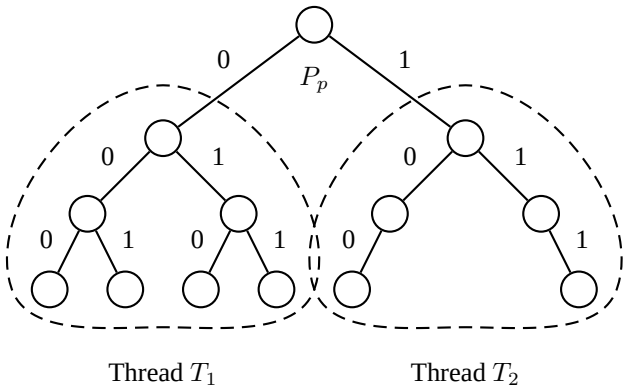


FIG. 3.4 – Problème d'équilibrage de charges

L'équilibrage des charges n'est donc pas optimal puisque notre programme principal va occuper un processeur à attendre la fin de ses processus fils alors qu'il pourrait utiliser une partie de son temps d'attente et un peu du temps de son processeur pour copier une autre partie de l'arbre. Il est donc nécessaire d'améliorer les charges de travail des différents processus. Le second point à prendre en compte est l'espace mémoire. Nous voulons pouvoir travailler à terme sur des arbres de très grandes tailles (plusieurs GO par exemple) et réaliser des opérations coûteuses en terme de mémoire, il est donc important de pouvoir en économiser le plus possible. Or si l'on re-considère le cas d'un arbre de radicaux binaires complet sans aucune limitation de ressources (figure 3.2 p.18), le nombre de threads lancés est de $\sum_{i=1}^{h} 2^i$ avec h égal à la hauteur de l'arbre. Ce nombre est assez conséquent lorsque l'on pense aux coûts mémoires associés à chaque création de threads. Il est en plus dommage de devoir réserver cette mémoire pour des processus qui réalisent une seule opération (dans notre cas théorique) et qui passe le reste de leur temps à attendre la mort de leur fils. Si l'on reprend l'exemple de la copie d'arbre, chaque nœud doit attendre la mort de ses processus fils. Si l'on se met à la place de la racine, en appelant Δt le temps moyen pour qu'un nœud effectue son travail et en négligeant les coûts de synchronisation, la racine va travailler pendant un temps

Δt et va attendre $h \times \Delta t$. On déduit le temps d'attente précedent du fait que tous les nœuds d'un même niveau effectuent leurs opérations en parallèle et que toutes ces opérations se réalisent en temps Δt. Il serait intéressant de trouver un moyen de ré-utiliser au maximum les threads qui attendent ou de pouvoir libérer les ressources qu'elles mobilisent.

3.1.3 Deuxième Méthode

Afin d'éviter les inconvénients de la méthode exposée précédemment, nous avons opté pour une nouvelle stratégie. Le principe de celle-ci est d'obliger chaque processus à travailler jusqu'à ce qu'il atteigne au moins une feuille (voire plus en fonction du nombre de threads disponibles). Pour ce faire, nous avons choisi arbitrairement de lancer les threads uniquement sur les sous-arbres droits (figure 3.5). De cette manière, les processus pères passent beaucoup moins de temps à attendre la fin de leurs processus fils puisqu'ils doivent s'occuper du sous-arbre gauche avant d'attendre.

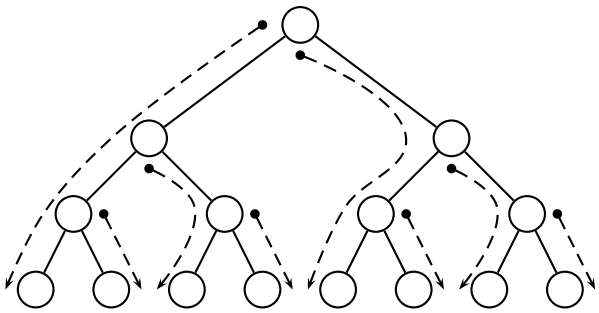


FIG. 3.5 – Deuxième méthode

Le nombre de threads dans notre cas théorique est divisé par deux puisqu'on lance un thread de moins à chaque niveau de l'arbre ce qui nous donne $\sum_{i=1}^{h} 2^{i-1}$ threads. Si l'on reprend l'exemple du paragraphe précédent avec la machine à trois processeurs et deux threads, dans notre cas, le programme principal (P_p) s'occupe de faire la copie de la racine, lance un thread (T_1) sur le sous-arbre droit et continue de faire la copie du sous-arbre gauche. Au niveau suivant, soit le thread, soit le programme principal lance la dernière thread disponible (T_2) sur son sous-arbre droit pendant que lui-même continue sur son sous-arbre gauche (figure 3.6 p. 21). Dans ce cas là, nos trois processus vont travailler et aucun ne passera la majorité de son temps à attendre.

Si cette méthode s'avère plus efficace en théorie, il reste certains cas pratiques où la première méthode reste préférable. Imaginons le cas où un arbre possède un seul fils gauche et un sous-arbre droit complet (figure 3.7 p.21). Si nous ne disposons que d'un seul thread alors dans notre deuxième méthode, le thread s'occupera du sous-arbre droit entier tandis que le processus père s'occupera du sous-arbre gauche (figure 3.3 p.19). Dans la première méthode, c'est l'inverse qui se produira. Si le fils s'occupe du sous-arbre droit complet, il y a de grandes chances qu'il finisse longtemps après le père qui n'a qu'une simple branche à copier, on va donc se retrouver dans un cas ou le père doit passer une grande partie de son temps à attendre son fils et où l'on ne pourra pas ré-utiliser de threads.

Une partie des ressources est gâchée. On peut se dire qu'on a juste à lancer les threads sur le sous-arbre gauche mais on tomberait dans un cas symétrique. Le problème vient du fait qu'on choisisse à l'avance et pour tout l'arbre sur quelle branche sera lancé un thread. On ne connait rien de la structure de l'arbre. Or en théorie, on sait que si l'on veut faire travailler un processus père et un processus fils ensemble, alors il vaut mieux lancer le processus fils sur le sous-arbre qui représente le moins de travail. Ceci permet de pouvoir réutiliser le processus fils quand il se sera fini et on limitera les temps d'attentes du processus père.

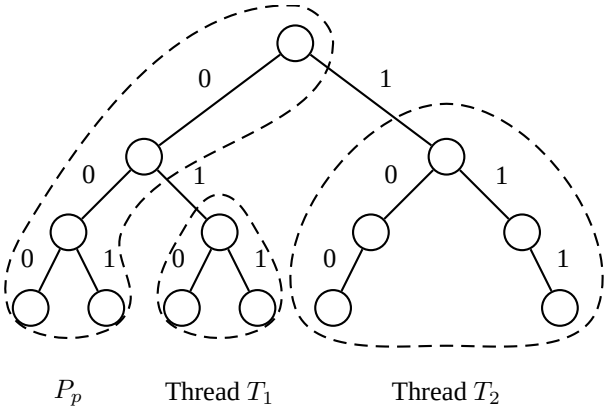


FIG. 3.6 – Avantages

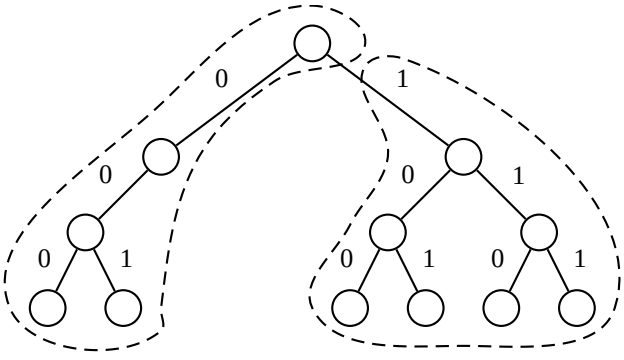


FIG. 3.7 – Problème d'équilibrage de charges

Voici donc des pistes pour améliorer encore une fois notre manière de faire. La première consiste à utiliser des heuristiques qui renseignent sur le poids des branches pour choisir qui du processus fils ou du processus père doit s'occuper de la sous-branche droite ou gauche. La seconde méthode, quant à elle, est une méthode exacte mais qui peut poser certains problèmes de réalisation. Il s'agirait de stocker dans chaque nœud interne, le nombre de nœuds de son sous-arbre gauche ainsi que le nombre de nœuds de son sous-arbre droit. Quand on aura la possibilité de lancer un thread, on le lancera sur le sous-arbre comportant le moins de nœuds. Le processus père s'occupera ainsi du sous-arbre le plus gros.

Cette technique nécéssite malheureusement d'allourdir la structure de données. Imaginons qu'on est un arbre complet de hauteur h, la racine devra posséder deux entiers qui pourront chacun contenir une valeur égale à $1 + \sum_{i=1}^{h-1} 2^i$. Dans le cas d'un arbre de hauteur, $h = 40$, le coût mémoire engendré risque donc d'être assez lourd. Il est par conséquent nécessaire, avant d'utiliser une telle méthode de bien analyser les besoins en fonction des ressources disponibles.

3.2 Optimisations

L'architecture des ordinateurs ayant tellement évolué ces dernières années, il est important si l'on recherche des performances ou que l'on possède une grosse masse de données à traiter de ne pas la négliger. En effet, de nombreuses possibilités d'améliorer les performances sont offertes si l'on prend la peine d'utiliser aux mieux les différents éléments d'un ordinateur. Nous verrons dans cette partie quelques pistes pour se servir pleinement de la puissance d'une machine et ainsi améliorer les résultats de nos opérations sur les arbres. Puis nous verrons comment adapter la structure de nos arbres pour pouvoir bénéficier de ces techniques.

3.2.1 Présentation

Rappels

Les ordinateurs tels que les avait définis autrefois Von Neumman en 1945 devaient comporter cinq unités essentielles :

- L'unité arithmétique et logique (UAL).
- L'unité de commande.
- La mémoire centrale.
- L'unité d'entrée.
- L'unité de sortie.

De tels systèmes devaient, pour être efficaces, fonctionner électronniquement et utiliser la numérotation binaire. Ces principes sont encore à la base de la conception des ordinateurs modernes. Cependant, de nos jours, la vitesse des processeurs (UAL) a tellement augmenté par rapport à la vitesse de la mémoire que nous arrivons face à un nouveau problème que l'on appelle le "mur mémoire". Ce principe est illustré sur la figure 3.8. Sur cette figure, nous pouvons voir que la courbe qui représente la vitesse des processeurs au cours de ces dernières années augmente très vite, alors que celle représentant la vitesse du bus mémoire augmente de manière lente creusant ainsi un fossé conséquent entre les deux.

Une bonne optimisation passe donc obligatoirement de nos jours par une bonne gestion de la mémoire. Par conséquent, une bonne connaissance de celle-ci s'avère indispensable pour une meilleure utilisation de cette dernière. Il existe plusieurs types de mémoires qui vont de la plus rapide à la plus lente, la plus petite à la plus grosse et de ce fait, de la plus chère à la moins chère. Ces types de mémoires sont :

- Les registres.
- Les caches.
- La Mémoire "vive" ou RAM.
- La Mémoire "morte" ou ROM.

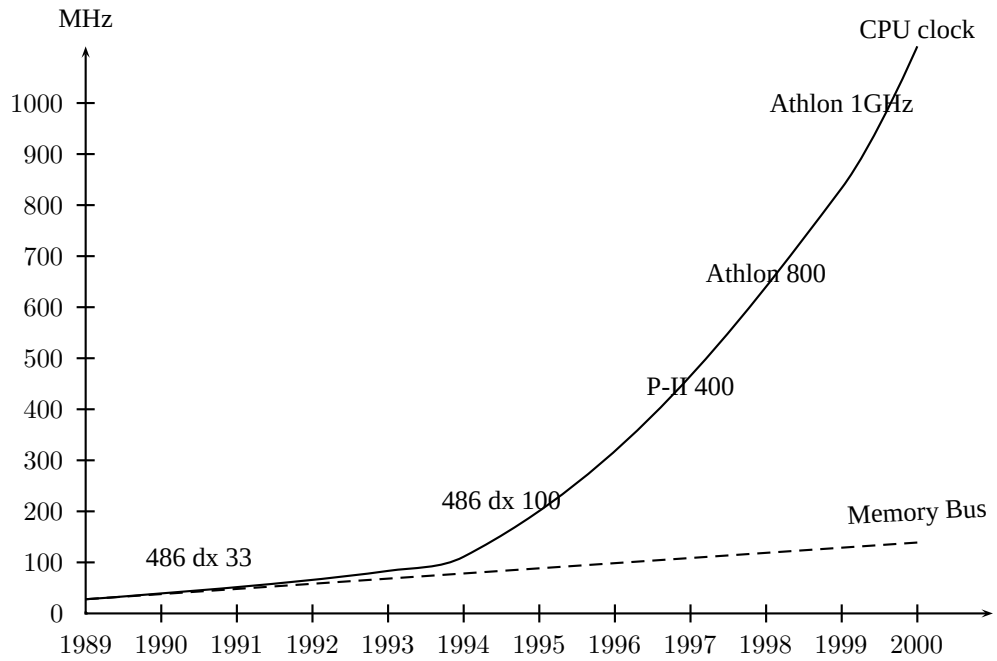


FIG. 3.8 – Mur mémoire

Le but est de maintenir les données utiles le plus près possible du processeur. Dans les prochaines sous-parties, nous nous intéresserons d'abord à la mémoire cache puis aux registres.

Mémoire cache

Afin d'utiliser au mieux la mémoire cache, il convient d'abord d'en examiner la structure. La mémoire cache est découpée en lignes que l'on nomme plus communément segments. Ces segments sont associés à d'autres segments qui appartiennent eux, à la mémoire vive. Il existe cependant plusieurs types d'associations entre les segments de mémoire cache et les segments de mémoire vive :

Direct Mapped : 1 segment mémoire est associé à un emplacement fixé dans le cache.

k-associative : 1 segment mémoire est associé à k emplacements fixés dans le cache.

Fully associative : 1 segment mémoire correspond à n'importe quel segment de la mémoire vive.

Si une donnée requise par le processeur se trouve dans la mémoire cache au moment où il en a besoin alors le processeur la lit directement. On appelle cela *cache hit*. En revanche, si la donnée ne s'y trouve pas, il va être nécessaire de la charger dans la mémoire cache pour que le processeur puisse la lire, on appelle cela : *cache miss*. Le but de l'optimisation au niveau de la mémoire cache consiste à essayer de minimiser au maximum le nombre de cache miss. Pour ce faire, il existe deux techniques principales : la localité spaciale qui consiste à mieux ordonner les données dans le cache et la localité temporelle qui consiste, quant à elle, à ré-utiliser au maximum les données déjà présentes dans le cache. Il existe plusieurs outils, dont nous ne parlerons pas ici, pour effectuer ce genre de choses. Les compilateurs tentent de nos jours d'effectuer eux-mêmes ce genre d'optimisations. Il existe des bibliothèques de calcul les utilisant parfaitement comme "BLAS" ou "LAPACK".

Registres

Depuis le passage des ordinateurs au statut d'outil multimédia grand public et le renforcement de la concurrence sur le marché des fabriquants de processeurs, de nouveaux registres et de nouvelles instructions

déstinés aux fonctions multimédias ont été intégrés aux processeurs en plus des registres habituels. C'est ainsi que l'on a d'abord vu apparaître le MMX (MultiMedia eXtensions) en 1997 chez Intel puis toute la gamme des SSE dont nous arrivons actuellement à la troisième génération et du 3DNow ! chez AMD. L'utilisation de ces registres est donc la base de l'optimisation à ce niveau.

Les registres permettent une vraie parallélisation, pour un processeur, des opérations. Ils offrent la possibilité d'exécuter des instructions vectorielles tirant pleinement partie du pipelining des processeurs modernes. On parlera ici de *SIMD*, Simple Instruction Multiple Data. Ceci signifie qu'il est possible d'effectuer une même opération sur des données différentes de manière parallèle. Ainsi, si l'on s'interesse aux registres MMX, ils sont aux nombres de 8 dans un ordinateur et possèdent une taille de 64 bits. Ce qui permet de stocker dans un registre soit :

- 1 mot de 64 bits.
- 2 mots de 32 bits.
- 4 mots de 16 bits.
- 8 mots de 8 bits.
- etc.

Si l'on peut stocker 8 mots de 8 bits alors nous réaliserons une opération sur 8 données en une seule fois. En utilisant donc au maximum les registres MMX, nous pourrions accroître de manière significative la vitesse de nos opérations. Les registres et instructions, SSE et 3dNow !, sont quant à eux déstinés aux opérations sur les nombres décimaux, ils disposent pour cela d'une taille de 128 bits. Pour accéder à ces registres, il est souvent nécessaire de passer par du code assembleur et les guides d'instructions fournis par le fabriquant. Mais depuis 2002, il existe la librairie "mmintrin.h" qui offre une interface aux programmeurs C pour utiliser les registres MMX.

3.2.2 Applications

Afin de pouvoir bénéficier des optimisations décrites ci-dessus, il est nécessaire de disposer d'une bonne repésentation des données. Avec une implémentation standard des arbres de radicaux, c'est à dire avec un bloc mémoire et 2 pointeurs pour symboliser un nœud, on ne peut accéder à celles-ci. En effet, le compilateur réservera de la mémoire n'importe où et se débrouillera pour la charger sans que l'on puisse réellement interférer sur la manière de faire. Les données se trouveront dans de nombreuses lignes de caches différentes, ce qui rendra la localité spatiale et temporelle très mauvaise. Quant au problème d'utilisations des registres pour améliorer nos opérations d'ensembles, notre structure se révéle tout aussi inadaptée. Les seuls structures de données vraiment aptes pour ce genre d'opérations sont les tableaux. Ceux-ci étant stockés de manière contigues en mémoire, le compilateur va les charger correctement dans le cache (par lignes complètes de cache). Le problème consiste donc à trouver une représentation par tableau efficace de nos arbres de radicaux.

Dans [Kos04], une première méthode, à la base destinée au stockage sur disque permet de convertir un arbre de radicaux en tableau de bits sans coût mémoire supplémentaire. La technique consite à indiquer par un 1 si une branche existe et par un 0 sinon. On applique cette technique en effectuant un parcours préfixe sur l'arbre. Par exemple, si l'on transforme l'arbre de la figure 1.3 (p.6), nous obtiendrons la séquence : 110101111101. Le problème de cette approche est que l'on ne peut à partir de cette suite de bits définir la forme de l'arbre. Par conséquent, les opérations sur les arbres sont difficilement envisageables.

La deuxième méthode consiste à compléter la suite précédente par des 0 de manière à obtenir une représentation de l'arbre complet. Dans ce cas, le coût mémoire devient alors catastrophique car les arbres n'étant en pratique que très rarement complets, le surplus d'informations généré risque de devenir non négligeable dans les calculs. Si nous prenons un arbre de hauteur 40, nous devrons avoir un tableau de taille $2 \times 2^{40} - 1 = 2^{41} - 1$ bits et ce quelque soit le nombre d'éléments présents dans l'arbre.

Nous arrivons donc à la troisième méthode qui consiste à supprimer le plus d'informations tout en gardant la structure utilisable pour les techniques d'optimisation. Pour y arriver, nous avons conclut que les données dans les branches n'étaient pas vraiment utiles, que seules les feuilles contenaient les informations. Nous avons donc décider d'employer un tableau tout simple, représentant toutes les feuilles, l'indice i du tableau, représentant la valeur i de l'arbre. Si un 1 se trouve à l'indice i du tableau alors la valeur i existe dans l'arbre. Dans ce cas, une intersection d'ensembles consiste alors à appliquer un "ET" logique entre les tableaux ; une recherche, quant à elle consiste simplement à la vérification de l'indice i. Le principal problème de cette solution se trouve dans le fait que la structure manque de souplesse. Nous verrons par la suite qu'il sera parfois nécessaire de construire des arbres de radicaux dans chaque feuille d'un arbre de radicaux. Nous n'avons pas encore trouvé de solutions adéquates pour réaliser cela avec les tableaux de bits.

En ce qui concerne le coût des opérations, les gains envisagés lors d'une opération d'intersection par tableau de bits ne sont pas aussi évident que cela par rapport à une implémentation standard des arbres de radicaux (un nœud possèdant 2 pointeurs). L'intersection consiste ici, à parcourir 2 tableaux de la même taille (puisque nous représentons toutes les feuilles de l'arbre) et à effectuer un "ET" logique entre chaque élément. La complexité est donc en $\Theta(n)$ (avec n nombre de feuilles de l'arbre complet). Or nous avons vu que notre intersection était en $\mathcal{O}(min(m, p))$ (pour m et p, nombre de nœuds de l'arbre). Donc si notre nombre de nœuds dans l'arbre est inférieur au nombre de feuilles dans l'arbre complet, alors les gains s'il y en a seront peut être négligeables. De plus suivant les configurations des arbres, une opération d'intersection peut se terminer immédiatement grâçe aux préfixes (voir figure 3.9). Sur la figure, nous pouvons voir que les deux arbres n'ont aucun préfixe commun. Par conséquent, le résultat est immédiat, il s'agit de l'arbre vide.

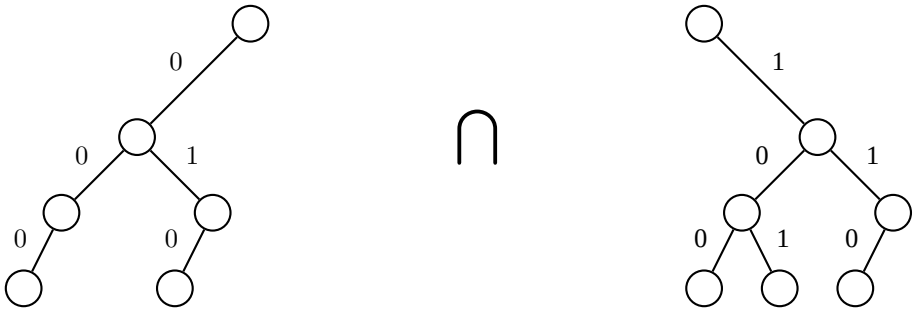


FIG. 3.9 – Exemple d'intersection en 2 tours.

Si cette configuration est un peu exagérée, elle montre qu'en comparant les préfixes, on peut effectuer de grandes coupes dans les arbres et ainsi gagner un temps considérable dans les traitements.

Maintenant que nous avons vu l'éfficacité des opérations sur les arbres de radicaux, nous allons voir comment s'en servir dans les algorithmes de recherche de règles d'association.

Chapitre 4

Application des arbres de radicaux aux algorithmes du datamining

Le but de cette partie est de montrer comment les arbres de radicaux peuvent être utiles dans les algorithmes du datamining et plus particulièrement dans les algorithmes de recherche d'itemsets fréquents. Pour cela nous allons d'abord voir dans quelle mesure les arbres de radicaux peuvent être utiles dans le stockage sur disque. Puis nous allons voir comment se servir des arbres de radicaux pour représenter les candidats et enfin comment en générer de nouveaux.

4.1 Stockage sur disque

La taille des bases de données à analyser dans les algorithmes de datamining est tellement énorme qu'il est inconcevable de pouvoir la faire tenir entièrement en mémoire vive. Il est donc nécessaire de trouver des méthodes efficaces pour stocker les données sur disque. Ces méthodes se doivent d'être d'autant plus efficaces que nous recherchons les performances et que les accès disques seront fréquents.

Dans [Kos04] (à paraître), une méthode pour stocker et indexer les données sur disque a l'aide d'arbres de radicaux a été introduite. Cette méthode repose sur une organisation structurée de l'espace disque. Le problème principal lorsque l'on décide de stocker une base de données sur disque est le fait de pouvoir retrouver avec une vitesse acceptable une donnée demandée. Si, l'on se contente d'une organisation toute simple, telle que, un objet par fichier, rechercher une donnée revient alors à rechercher le fichier qui la contient. Le problème dans notre cas est que le nombre de fichiers atteint, risque de mettre à mal le système d'exploitation. Imaginez devoir retrouver un fichier parmi des millions, tous stockés dans un même répertoire. Même si les systèmes de fichiers ont fait beaucoup de progrès ces dernières années, ils ne sont pas encore capables de répondre à de telles demandes dans des temps corrects. Si l'on décide de regrouper toutes les données dans des fichiers trop gros, c'est alors le problème de la mise à jour de ces fichiers qui devient une véritable contrainte.

La méthode que nous utilisons propose une structure arborescente de répertoires avec dans chacun de ces répertoires, de petits fichiers facilement ré-actualisables. Dans [Kos04], les articles de la base de données sont indexés grâce à leur identifiant et chaque répertoire contient trois fichiers. Ces trois fichiers sont (voir figure 4.1) :

- Un fichier contenant le thésaurus des items et pour chaque item, l'offset d'un champ de bits contenu dans le second fichier (fichier 1).
- Un fichier contenant les champs de bits des identifiants pour le niveau n (fichier 2).
- Un fichier contenant une permutation de tous les mots donnant l'ordre lexicographique (fichier 3).

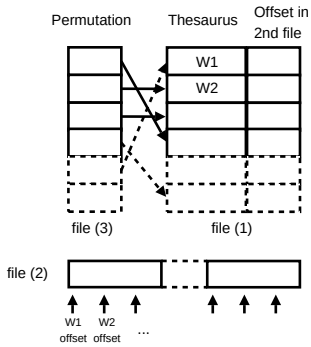


FIG. 4.1 – Organisation de fichiers

Afin de stocker une ligne de la base de données associée à un mot, nous attribuons un identifiant sous forme de champs de bits de taille n (rappel les arbres de radicaux contiennent un ensemble de taille 2^n) à chaque mot. Chacun de ces champs de bits désigne le chemin vers le prochain répertoire où chercher. La taille d'un champ de bits étant n, un répertoire peut donc contenir jusqu'à 2^n sous-répertoires.

Le fichier de permutation (fichier 3), contient une permutaion, p, des éléments du thésaurus les donnant dans l'ordre lexicographique. Soit t, la cardinalité du thésaurus, p est une permutation de $[0, t-1]$ sur $[0, t-1]$. Cette permutation est telle que le i^{th} mot contenu dans l'ordre lexicographique correspond au $p[i^{th}]$ dans le fichier de permutation. Cette méthode nous permet de pouvoir rajouter facilement des mots dans le thésaurus car seul, le fichier de permutation doit être ré-écrit. De plus la recherche d'un mot dans le fichier de permutation est équivalente à la recherche d'un mot dans un tableau trié, la complexité est donc en $\mathcal{O}(log(n))$ avec n=t, soit le nombre de mots présents dans le thesaurus.

Pour rechercher une ligne de la base de données où un item particulier apparaît, nous commençons par consulter le fichier de permutations (fichier 3). Celui-ci nous donne la position du mot dans le thesaurus (fichier 1) auquel est associé l'offset du champ de bits contenu dans le fichier 2. Le champ de bits nous donnera ensuite le chemin vers le prochain répertoire à visiter. Une fois dans le répertoire suivant, l'opération se répete à nouveau, ceci jusqu'à ce que l'on atteigne le dernier niveau où la ligne est contenue. La figure 4.2 montre un exemple de hiérarchie de fichiers sur quatre niveaux.

Les avantages de pouvoir indexer une base de données à l'aide d'arbres de radicaux sont notamment un gain de place, le fait de pouvoir bénéficier d'opérations à temps constant et peu coûteuses ainsi qu'une organisation de fichiers offrant un bon compromis entre taille de fichiers et nombre de fichiers par répertoire. Le gain de place se traduit par le fait que les prefixes communs des identifiants ne sont stockés qu'une seule fois. En effet deux identifiants ayant le même prefixe partageront le même repertoire. En ce qui concerne les opérations sur les arbres de radicaux, nous avons pu étudier, dans une partie précédente, leur compléxité. Nous pouvons quand même rappeler que l'insertion et la recherche se font en temps constant. Ces temps dépendent du nombre de niveau sur lequel nous voulons stocker les données et par conséquent de la taille du champ de bits représentant un identifiant. Les arbres de radicaux sont utilisés avec succés dans [Kos04] pour construire un gestionnaire de bases de données séquentiel tel qu'il est défini dans [UW02].

4.2 Représentation des candidats

Pour se servir des arbres de radicaux dans les algorithmes de recherche de règles d'association, la première idée consiste à représenter les données sous forme d'arbres. Nous allons donc commencer par stocker les candidats dans un arbre de radicaux. Si par exemple, nous avons quatre articles, nous les stockerons comme le montre la figure 4.3. À chacun de ces articles, nous associerons la liste des transactions,

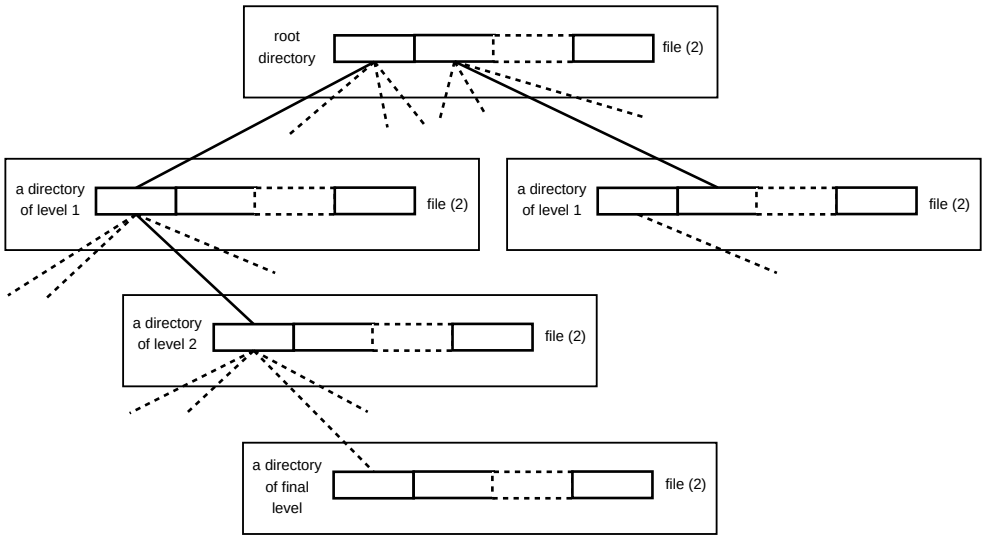


FIG. 4.2 – Hiérarchie de fichiers

toujours sous forme d'arbres de radicaux, dans lesquelles ils apparaîssent. Cet arbre s'appelera l'arbre des transactions. Cette façon de représenter les choses simplifie grandement la phase de calcul du support. Celle-ci consiste dorénavant à effectuer des intersections entre les arbres de transactions et à compter le nombre de feuilles.

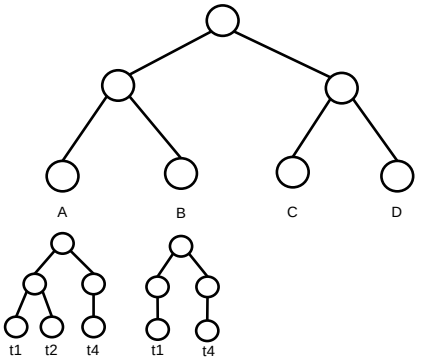


FIG. 4.3 – Représentation des candidats

Les intersections d'arbres de radicaux sont très avantageuses par rapport aux autres structures de données comme nous l'avons vu un peu plus tôt dans ce document. Mais un des avantages les plus interessants reste qu'il est possible de construire l'arbre des candidats en un seul parcours de la base de données. À titre de comparaison, l'algorithme A priori (chapitre1) perd beaucoup de temps dans cette phase (un scan de base par itération). De plus pour améliorer encore un peu les performances, nous pouvons envisager de calculer des supports partiels locaux à la manière de l'algorithme Count Distribution. Maintenant que nous avons vu comment se servir des arbres de radicaux pour repésenter les données, il nous reste à voir comment l'on peut générer de nouveaux candidats.

4.3 Génération de candidats

La génération de nouveaux candidats s'effectue par jonction des membres d'une même classe d'équivalences. En représentant les itemsets par des arbres de radicaux comme décrit sur la figure 4.3, tous les membres d'une classe d'équivalence sont dans le sous-arbre de l'itemset définissant la classe. Par exemple, tous les membres de la classe d'équivalence S_A seront dans le sous-arbre enraciné en A.

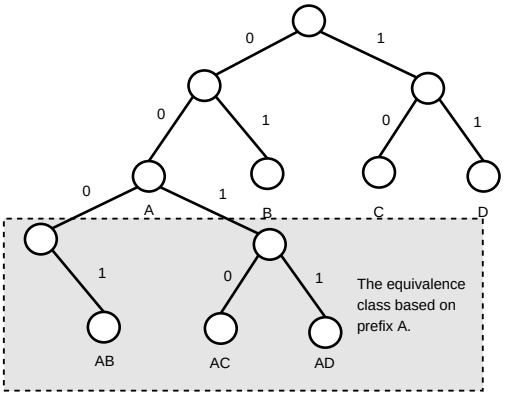


FIG. 4.4 – Représentation d'une classe d'équivalence

La génération de nouveaux candidats consiste donc à réaliser pour toutes les feuilles d'un arbre la jonction avec les membres de la classe d'équivalences (voir figure 4.5). Cette opération nécessite d'éliminer tous les doublons, pour cela, nous pourrons utiliser notre opération d'élagage définie dans le chapitre 2. Il faut considérer comme doublon, toutes permutations d'un itemset. Par conséquent, les itemsets AB et BA sont les mêmes donc seul le premier nous intéressera.

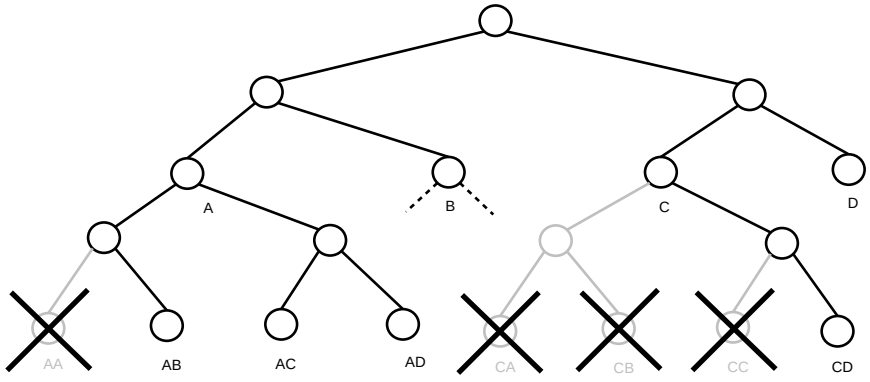


FIG. 4.5 – Génération de candidats

En répétant le procédé jusqu'à ce qu'on ne puisse plus obtenir de candidats, nous obtenons l'arbre de la figure 4.6.

La représentation des données par arbres de radicaux facilite certaines opérations. Nous avons déja dit dans la partie précédente qu'un seul parcours de la base était nécessaire pour construire les arbres. Mais le calcul des supports des nouveaux candidats devient tout aussi simple puisqu'il nécessite qu'une intersection des arbres de radicaux contenant les identifiants. De plus, nous pouvons, à la manière de l'algorithme count distribution répartir la base sur plusieurs processeurs et calculer des supports locaux. La seule communication entre les différents processeurs consistera alors en l'échange des supports pour en calculer un global. Pendant ces phases de communications, il est de plus envisageable que les processeurs ne perdent

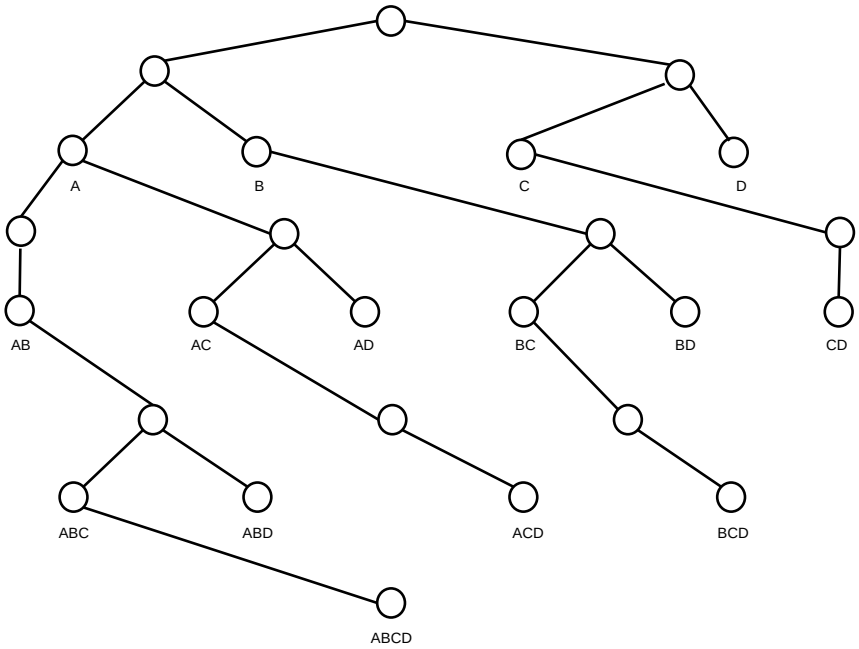


FIG. 4.6 – Génération de candidats (complet)

pas de temps en commençant déjà à éliminer les candidats invalides de leurs arbres locaux. Ceci conduit à un entrelacement des calculs et potentiellement à un gain sur le temps d'exécution. Les candidats sont invalides si un des items de l'itemset n'est pas présent dans la base locale. Par exemple, on sait que si l'article A n'est pas dans la base locale alors AB n'y est pas non plus. Les opérations de suppression sont en plus de faibles coûts. Ces améliorations apportées par les arbres de radicaux ont permis la mise au point d'un nouvel algorithme s'inspirant des algorithmes parallèles présentés dans la partie 1.

4.4 Notre nouvel Algorithme de recherche de règles d'association

L'étude des algorithmes basés sur A priori ainsi que l'étude de l'algorithme Eclat et l'approche offerte par les arbres de radicaux nous ont permis de proposer un nouvel algorithme essayant de combiner les avantages de ces derniers.

Algorithm executed on each Proc. $0 \leq i \leq p$**.**

/ Initially, each processor has locally* n/p *lines of the transaction file where* n *is the total number of lines.*/*

Scanning local part of the database for construction of 1-itemset tree.

do

Broadcast supports.

/* This part can be de-synchronized */

/* to perform overlapping (see above) */

Wait for all supports from others.

Perform the sums reductions.

Elimination of unsufficient itemsets support.

$L_k =$ rest of C_k

Construction of new candidates sets C_{k+1}.

while ($C_{k+1} \neq \varnothing$)

frequent itemsets $= \bigcup L_k$

Comme nous l'avons vu précédemment, cet algorithme offre plusieurs avantages. Premièrement, il permet de n'effectuer qu'un seul parcours de la base de données. Deuxièment, le comptage des feuilles peut s'effectuer lors de la construction de l'arbre de transactions. Ceci nous permet de bénéficier très vite de support locaux. À partir de là, nous pouvons éliminer les candidats invalides. L'élimination se fait alors par la suppression d'une branche qui est une opération en $\mathcal{O}(h)$ où h est la hauteur de l'arbre, ce qui est très efficace. Troisièment, la génération de nouveaux candidats consiste en l'enracinement d'arbres élagués aux feuilles de l'arbre des candidats. Ceci passe par des opérations de coûts relativement faibles. Comparé aux algorithmes existant, nous avons le faible nombre de parcours de l'agorithme Eclat (1 seul) allié au faible coût des communications de l'algorithme Count Distribution (nous ne diffusons que les supports locaux).

Conclusion

Le but de ce stage de DEA était la mise au point d'une bibliothèque proposant des opérations multhreadées sur les arbres de radicaux. Cette bibliothèque a pu être réalisé à la fois en langage C++ et en langage JAVA (voir Annexe). L'étude de l'utilisation des arbres de radicaux dans les algorithmes de recherche d'itemsets fréquents a donné lieu à la création d'un nouvel algorithme sur le sujet et à la publication d'un article [CGKM04]. Notre algorithme est actuellement implémenté en MPI (Message Passing Interface) qui est un standard de la programmation parallèle.

Le déploiement sur grille pose cependant plusieurs problèmes. En effet, même si nous disposons de sites de confiance, nous ne sommes pas à l'abri d'une panne technique. Étant donné la taille du projet Grid explorer, les pannes matérielles passent désormais du statut d'évènements exceptionnels au statut d'évènements chroniques. Il est donc important d'étudier les moyens de rendre notre algorithme (au moins) tolérant aux pannes. Une des pistes à explorer est le projet MPICH-V qui se définit comme "a Scalable Fault Tolerant MPI for Volatile Nodes" [Hér04]. Ce projet étudie le problème de la tolérance aux pannes automatique (sans intervention humaine) et transparente (sans modification du code existant) dans le cadre des applications MPI. Il est composé de plusieurs architectures dont MPICH-V3 qui est un travail en cours et dont le but est d'introduire la tolérance aux fautes dans les grilles de clusters, en tolérant toutes les fautes, d'une machine dans un cluster à un cluster dans la grille.

Un autre point qui nécessite une attention toute particulière est la sécurité des données, il est évident que si nous possédons des données à analyser, alors celles-ci doivent être sévèrement protégées pour que seul le détenteur puisse y avoir accès ou du moins en saisir le contenu réél. Si la sécurité des communications est de nos jours bien avancées, il n'en va pas tout à fait de même pour la sécurité des données stockées.

Ces deux points sont actuellement à la base de nombreux travaux de recherche, ce qui reflète l'importance qu'ils ont dans l'informatique d'aujourd'hui et de demain.

Je tiens à remercier Michel Koskas et Cristophe Cerin pour leur disponibilité et pour leurs aides précieuses tout au long de ce stage et je tiens à remercier également Gaël Le Mahec avec qui j'ai travaillé pour la rélisation du travail présenté ici.

Annexe A

Bibliothèque

Le but de cette annexe est de faire une rapide présentation de la bibliothèque développée pendant ce stage.

A.1 Présentation

A.1.1 Technique

La bibliothèque d'utilisation des radix trees a été conçue dans deux langages orientés objets : le C++ et le JAVA. La nécessité de disposer d'un code robuste et ré-utilisable nous a donc confortés dans le choix de ce type de langage. La version C++ est considérée comme la version principale étant données les performmances de ce langage par rapport à celles obtenues par le biais du langage JAVA. C'est d'ailleurs pour cette raison que nous étudierons principalement cette version. Il est possible de sortir une documentation complète du code au format html et LaTeX grâce à l'outil de documentation doxygen (doxygen et javadoc pour la version java). Afin de pouvoir utiliser la bibliothèque C++, il est nécessaire de disposer de la librairie boost disponible sur `http ://www.boost.org`. Cette librairie fournit notamment des bitsets dynamiques et une implémentation réputée plus portable des threads.

A.1.2 Conceptuelle

Pour représenter les arbres de radicaux binaires, nous avons considéré ceux-ci comme des ensembles de noeuds. Ces noeuds peuvent être de deux types : internes ou externes (autrement appelés feuilles). Les différents objets du programme se sont alors imposés d'eux mêmes :

Objet Radix Tree : C'est l'objet principal, il représente l'ensemble des noeuds. Cet objet contient en fait uniquement la racine de l'arbre qui est un objet de type noeud interne.

Objet virtuel Noeud : Les noeuds possèdent tous les mêmes méthodes. Par contre, suivant leur type (interne ou feuille), ils ne l'effectuent pas de la même manière. Le fait de passer par un objet virtuel Noeud permet de simplifier les choses en disant qu'à partir du moment où l'on demande à un objet de type Noeud d'executer une de ses méthodes, il la réalise peu importe son type (interne ou feuille). La distinction entre l'action d'un noeud interne et d'une feuille se fera alors par l'objet lui-même.

Objet Noeud Interne : Les objets de type Noeud interne héritent simplement de l'objet virtuel Noeud. Son rôle principal est de définir le comportement des méthodes héritées de Noeud.

Objet Feuille : Les objets de type Feuille ont le même rôle que les objets de type Noeud Interne.

Objet Donnée : L'objet donnée est contenu dans les feuilles. Il nous permettra de créer des arbres de radicaux avec n'importe quoi aux feuilles. Comme le C++ ne possède pas de véritable classe objet, nous essayons de la simuler par cette classe.

Afin de rendre le code plus lisible, nous avons essayé d'adopter au maximum, quelque soit la langage, les principes de la standardisation JAVA. Nous avons donc un fichier par classe et chaque fichier a le même nom que la classe qu'il définit.

Maintenant que nous avons vu une rapide présentation de la bibliothèque, examinons la manière de s'en servir.

A.2 Utilisation

Avant de parler de notre bibliothèque, il est indipensable de fournir les options de compilation nécessaires à l'utilisation de la librairie boost : `-pthread` ainsi que `-lboost_thread-gcc-mt`. Nous avons fourni notre bibliothèque C++ sous 3 formes. La première habituelle est sous forme de sources, il suffit de régénérer les .o à l'aide de la commande make et d'inclure le fichier `RadixTree.hh` dans le fichier où l'on veut utiliser nos arbres de radicaux. La deuxième est sous forme de librairie statique, il suffit d'inclure le fichier `librtree.hh` et de compiler le programme avec l'option `-lrtree`. Enfin la troisième est sous forme de librairie partagée et s'utilise comme la librairie statique. Pour la version JAVA, nous n'avons que les fichiers class pour le moment, nous pensons tout regrouper sous forme de point Jar par la suite. Maintenant que nous avons vu les différents moyens d'accéder aux fonctions de notre bibliothèque, nous allons faire une rapide présentation de ces fonctions celles-ci :

`RadixTree()` : Constructeur par défaut, il crée un arbre de hauteur 24 (contenant des valeurs sur 24 bits) disposant de 0 thread.

`RadixTree(int nthread, int bsize)` : Constructeur qui permet de choisir les paramètres de l'arbre (nombre de threads et taille du codage).

`RadixTree(RadixTree * r)` : Construit simplement un arbre à partir des carcatèristiques d'un autre arbre passé en paramètre. Attention, on ne copie pas les valeurs stockées dans l'arbre.

`RadixTree * clone()` : Cette fonction permet de copier un arbre entier (caractèristiques et valeurs contenues).

`void insert(unsigned long int value, DataType * data)` : permet d'insérer une valeur dans l'arbre et une donnée dans la feuille nouvellement construite. Il existe une autre version permettant de passer un dynamic_bitset au lieu d'un entier.

`bool search(const boost : :dynamic_bitset<> * bs)` : Cette fonction nous permet de vérifier si une valeur est stockée dans l'arbre. Dans la version JAVA, on passe une valeur à la place d'un bitset.

`void show()` : Cette fonction effectue un parcours préfixe de l'arbre et affiche la suite de bits correspondantes.

`void showValues()` : Cette fonction affiche les valeurs en base 10 des nombres stockés dans l'arbre de radicaux.

`RadixTree * op_and(RadixTree * r)` : Opération d'intersection sur les arbres de radicaux. Renvoie un nouvel arbre resultant de l'opération d'intersection.

`RadixTree * op_or(RadixTree * r)` : Opération d'union sur les arbres de radicaux. Renvoie un nouvel arbre resultant de l'opération d'union.

`RadixTree * copyPrune(int nbleaf)` : Opération d'élagage sur les arbres de radicaux. Cette version effectue une copie élaguée de l'arbre.

`void prune(int nbleaf)` : Opération d'élagage sur les arbres de radicaux (sans faire de copie).

Les opérations de la bibliothèque JAVA portent exactement le même nom et s'utilise de la même manière.

A.3 Résultats

Nous avons testé la bibliothèque C++ sur un bi-opteron. Pour cela nous avons crée un programme qui génère 60 arbres aléatoires avec un nombre de valeurs par arbres prédéfinis. Le programme réalise 600 intersections avec ces arbres. Afin d'obtenir un point de comparaison, nous avons réalisé la même chose avec des listes chainées (sauf que nous devons trier les listes avant d'en faire l'intersection). Les valeurs obtenues se trouvent dans le tableau A.1 p.35.

nb valeurs	listes	arbres
40000	13,69 sec.	12,25 sec.
50000	19,73 sec.	13,12 sec.
60000	26,44 sec.	18,02 sec.
70000	38,28 sec.	21,49 sec.
80000	45,09 sec.	23,81 sec.
100000	63,32 sec.	27,91 sec.
150000	111,56 sec.	40,69 sec.

FIG. A.1 – Tableau de résultats

Lors des tests, nous avons lancé les opérations sur les arbres avec un thread de disponible. Sans thread, 600 intersections d'arbres de 150000 éléments nous a pris 58,39 secondes. Voici la représentation sous forme de graphique des résultats obtenues précédemment (fig. A.2).

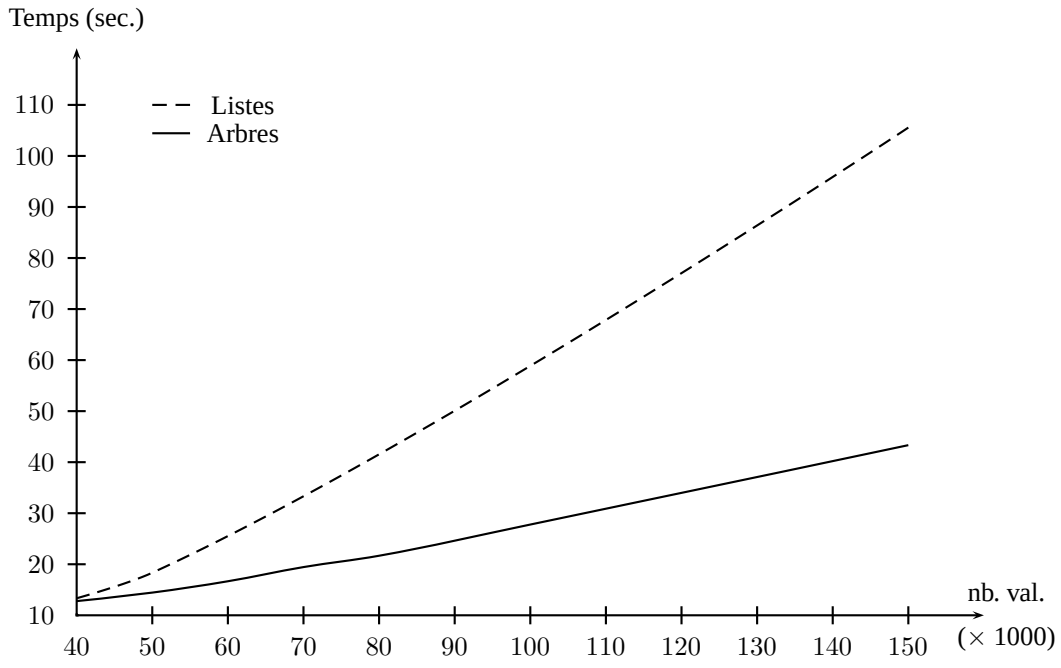


FIG. A.2 – Résultats de l'opération d'intersection

Les gains obtenus s'expliquent par la ré-utilisation des prefixes. Plus on augmente le nombre de valeurs dans les structures, plus ce phénomène de "ré-utilisation" est fréquent. Le nombre de valeurs reste cependant assez faible dans nos tests comparé à une hauteur d'arbre égale à 24 (2^{24} valeurs possibles). Ceci nous permet donc de montrer qu'une implémentation par arbres de radicaux devient vite plus avantageuse qu'une implémantation par listes, l'écart ne faisant que se creuser par la suite.

Bibliographie

[Abe01] Karl Aberer. P-Grid : A self-organizing access structure for P2P information systems. *Lecture Notes in Computer Science*, 2172 :179–192, 2001.

[AIS93] 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.

[BM72] Rudolf Bayer and Edward M. McCreight. Organization and maintenance of large ordered indices. *Acta Informatica*, 1 :173–189, 1972.

[BS97] Bentley and Sedgewick. Fast algorithms for sorting and searching strings. In *SODA : ACM-SIAM Symposium on Discrete Algorithms (A Conference on Theoretical and Experimental Analysis of Discrete Algorithms)*, 1997.

[Buy99a] Rajkumar Buyya. *High Performance Cluster Computing, Volume 1 : Architectures and Systems*. P T R Prentice-Hall, Englewood Cliffs, NJ 07632, USA, 1999.

[Buy99b] Rajkumar Buyya. *High Performance Cluster Computing, Volume 2 : Programming and Applications*. P T R Prentice-Hall, Englewood Cliffs, NJ 07632, USA, 1999.

[CGKM04] Cristophe Cerin, Jean-Sébastien Gay, Michel Koskas, and Gaël Le Mahec. Efficient data-structures and parallel algorithms for association rules discovery, 2004.

[Hér04] Thomas Hérault. Mpi tolérant aux fautes. In *DRUIDE 2004 : actes*, pages 195–196, Mai 2004.

[JA] Ruoming Jin and Gagan Agrawal. An efficient association mining implementation on clusters of SMP. pages 156–156.

[KAS02] Magdalena Punceva Karl Aberer, Manfred Hauswirth and Roman Schmidt. Improving data access in p2p systems. *IEEE Internet Computing*, 2002.

[Kos04] Michel Koskas. A hierarchical database management algorithm, 2004.

[Mei02] Marina Meilă. Radix trees. May 2002.

[PRR99] Plaxton, Rajaraman, and Richa. Accessing nearby copies of replicated objects in a distributed environment. *MST : Mathematical Systems Theory*, 32, 1999.

[Toi96] Hannu Toivonen. Sampling large databases for association rules. In T. M. Vijayaraman, Alejandro P. Buchmann, C. Mohan, and Nandlal L. Sarda, editors, *In Proc. 1996 Int. Conf. Very Large Data Bases*, pages 134–145. Morgan Kaufman, 09 1996.

[UW02] Jeffrey D. Ullman and Jennifer D. Widom. *First Course in Database Systems, A, 2/e*. Prentice Hall, 2002.

[Wat03] Paul Watson. *Databases and the grid, Making the Global Infrastructure a reality*. F. Berman and A. Hey and G. Fox, 2003.

[Zak99] Mohammed J. Zaki. Parallel and distributed association mining : A survey. *IEEE Concurrency*, 7(4) :14–25, /1999.

[Zak02] M. Zaki. Efficiently mining frequent trees in a forest, 2002.

[Zak25] Mohamed J. Zaki. Parallel and distributed association mining : A survey. *IEEE Concurrency*, Vol. 7, No. 4, October-December 1999, pp. 14-25.

[ZPL97] 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