UNIVERSITE MENTOURI CONSTANTINE
Faculté des Sciences de l'Ingénieur
Département d'Informatique

THESE DE DOCTORAT EN SCIENCES

Spécialité: Informatique

MOHAMED EL HADI BENELHADJ

Entrepôt de Données et Fouille de Données
Un Modèle Binaire et Arborescent dans le Processus de Génération des Règles d'Association

Soutenue publiquement le 29 Avril 2012 devant le Jury:

Professeur Zizette Boufaïda	**Présidente**	**Université Mentouri Constantine**
Professeur Nadir Farah	**Examinateur**	**Université d'Annaba**
Professeur Mohamed Chawki Batouche	**Examinateur**	**Université Mentouri Constantine**
Docteur Abdelkamel Tari	**Examinateur**	**Université de Béjaïa**
Professeur Mahmoud Boufaïda	**Rapporteur**	**Université Mentouri Constantine**
Professeur Yahya Slimani	**Invité**	**Université Tunis El Manar**

« Lorsqu'une porte se ferme, il y en a une qui s'ouvre. Malheureusement, nous perdons tellement de temps à contempler la porte fermée, que nous ne voyons pas celle qui vient de s'ouvrir. »

Alexander Graham Bell

Remerciements

Je remercie vivement le Professeur Zizette Boufaïda d'avoir accepté de présider mon jury de thèse. Puisse-t-elle trouver ici l'expression de mon profond respect et de ma reconnaissance.

Je voudrais également remercier chaleureusement le Professeur Nadir Farah, le Professeur Mohamed Chawki Batouche et le Docteur Abdelkamel Tari qui ont bien voulu participer à mon jury de thèse et s'intéresser à mon travail.

Je tiens à remercier tout particulièrement le Professeur Mahmoud Boufaïda, mon Directeur de thèse, qui m'a encadré durant ces longues années. Nous avons eu ensemble des discussions très enrichissantes qui m'ont orienté dans mes travaux de recherche. Le Professeur Mahmoud Boufaïda a également consacré beaucoup de son temps à la relecture minutieuse de ce document et je le remercie pour tous ses conseils avisés et pour sa disponibilité.

Je tiens à exprimer toute ma reconnaissance au Professeur Yahya Slimani, pour son ouverture d'esprit, sa sympathie et pour tout l'intérêt qu'il a manifesté pour mon travail tout au long de ma thèse. Ses compétences dans ce domaine m'ont permis d'une part de découvrir ce domaine fascinant et d'autre part de comprendre et ainsi de mieux appréhender le processus de validation des approches développées durant ma thèse.

Je remercie également, le Docteur Khedija Arour. J'ai particulièrement apprécié les séances de travail que nous avons eu ensembles.

J'ai énormément apprécié le climat dans lequel se sont déroulés mes nombreux passages au sein de l'équipe "Parallélisme, Grid Computing, Distribution et Datamining (PGC2D)" du Département des Sciences de l'Informatique, Faculté des Sciences de Tunis. Merci aux membres de cette équipe pour leur accueil et leur bonne humeur.

Nombreuses sont les personnes, ayant contribué de près ou de loin à l'aboutissement de cette thèse. Je tiens simplement à leur exprimer mes amitiés et mes remerciements, en particulier, un grand merci à Moez, Hayet, Ghada et Islam.

Je ne peux finir sans remercier Mejdi Ayari, Responsable du CNF de Tunis, Mona, Souhaïla, Imen et Boutheïna, pour m'avoir ouvert les portes du CNF et de l'INSAT, et m'avoir accueilli comme l'un des leurs.

Merci à ma famille et à tous mes amis pour m'avoir soutenu et encouragé et surtout de m'avoir supporté durant toutes ces années.

A ceux et celles qui, chacun à leur façon, ont fait ce bout de chemin à mes côtés, merci.

A mes enfants, dont l’existence donne un sens à ma vie. Tous mes remerciements sont à vous. Vous êtes la raison pour laquelle je me lève tous les jours de ma vie.

Une pensée pour mon père, avec qui j'aurai voulu partager ce moment de réussite.

TABLE DES MATIERES

TABLE DES FIGURES

TABLE DES TABLES

INTRODUCTION

INTRODUCTION

L'informatique décisionnelle permet l'exploitation des données d'une organisation afin de faciliter la prise de décision. Elle a connu, et continue de connaître encore aujourd'hui, un essor important. Les Systèmes d'Information (SI) classiques avaient pour simple vocation la production de données. Aussi, l'un des objectifs des entrepôts de données a été la transformation de ces SI en systèmes décisionnels, par la transformation des données de production en des informations stratégiques. Ces dernières sont souvent des données de production agrégées, intégrées et historisées. Elles sont dotées d'outils permettant de regrouper, nettoyer et intégrer les données. Ces outils servent également à établir des requêtes, des rapports et des analyses, à extraire des connaissances (fouille de données ou Data Mining) et à administrer un entrepôt de données (Data Warehouse ou DW) [81, 83].

Les systèmes de gestion de bases de données (SGBDs) traditionnels permettent de faire des opérations sur une base de données (insertion, modification, suppression, consultation) et ont donc un mode de travail transactionnel (OLTP, pour On-Line Transaction Processing). Par contre, les entrepôts de données sont conçus pour l'aide à la décision et utilisent la technologie OLAP (pour On-Line Analytical Processing) avec pour principaux objectifs de regrouper et d'intégrer des informations provenant de sources hétérogènes, de les stocker et de donner une vue orientée métier de ces données, de pouvoir les retrouver et les analyser avec facilité et rapidité [82].

Avec l’augmentation des capacités de stockage, nous avons assisté durant ces dernières années à une croissance importante des moyens de génération et de collection des données. Le besoin d’interpréter et d’analyser ces données afin d’en extraire des connaissances à forte valeur ajoutée, contribuant à la prise de décision, est devenu impérieux. Il s’est ainsi créé une obligation d’acquisition de nouvelles techniques et méthodes intelligentes de gestion. C’est ainsi que l’on a commencé à parler d'Extraction de Connaissances à partir de Données (ECD) ou de Knowledge Discovery in Database (KDD), regroupées sous le vocable de Fouille de Données (Data Mining) [83]. Les techniques de Fouille de Données permettent de découvrir des informations pertinentes *cachées* dans les données. Elles permettent par exemple de trouver les caractéristiques des clients les plus fidèles d’une banque, ou encore les tendances qui se dégagent dans les ventes d'un supermarché. La connaissance de telles informations peut permettre à la

banque ou au supermarché d'élaborer des stratégies commerciales ou de marketing en direction de leurs clients [84, 85, 86].

L'ECD est une discipline récente, se trouvant à l'intersection des domaines des bases de données, de l'intelligence artificielle, des statistiques, des interfaces homme/machine et de l'algorithmique. A partir de données collectées, il s'agit d'extraire des connaissances nouvelles qui enrichissent les interprétations du champ d'application, tout en fournissant des méthodes automatiques pour exploiter ces connaissances. L'ECD est décrite comme un processus interactif de préparation des données, d'extraction de connaissances à l'aide d'algorithmes, de visualisation et d'interprétation des résultats, en interaction avec un expert. Les méthodes d'exploration proposent des solutions aux problèmes de recherche d'associations [78], de classification supervisée [24, 79] et non supervisée [24, 37], etc.

La technique de découverte des règles d'association est une des techniques les plus connues et les plus explorées de la Fouille de données. Introduite par Agrawal [5], cette technique, développée à l'origine pour l'analyse des bases de transactions de ventes, a pour but de découvrir des relations significatives entre les données d'une base. Elle doit sa "notoriété" à son large champ d'applications qui s'étend à divers domaines, tels que le marketing, l'aide au diagnostic médical, les télécommunications, l'analyse de données spatiales, la téléphonie, etc. [13].

Le processus de génération des règles d'association est composé de deux étapes: une étape d'extraction d'ensembles d'attributs les plus fréquents et une étape de génération des règles d'association, à partir de ces ensembles. La phase d'extraction des ensembles fréquents est l'étape la plus couteuse en termes de complexité. Cette complexité est accentuée par la taille des bases de données à explorer. Ainsi, l'espace de recherche est exponentiel par rapport à la taille des bases de données.

La majeure partie des algorithmes de découverte des règles d'association adopte une approche dite "générer et tester", définie par l'algorithme Apriori [5]. L'algorithme Apriori, proposé par Agrawal en 1994, est l'un des algorithmes d'extraction d'ensembles d'attributs fréquents les plus connus. Il fait partie de la classe des algorithmes avec génération d'ensembles de candidats (à devenir fréquents). L'inconvénient majeur de cet algorithme est le grand nombre d'accès à la base de transactions.

D'autres algorithmes adoptent une approche sans génération d'ensembles candidats. L'algorithme FP-Growth [18], par exemple, utilise la structure FP-Tree (Frequent-Pattern tree) pour représenter les transactions. Cette structure permet d'avoir une représentation condensée d'une base de transactions. De plus, l'approche mise en place pour découvrir les itemsets fréquents, à partir d'un FP-Tree, évite la génération d'un grand nombre d'ensembles candidats. Cette famille d'algorithmes utilise une approche de type "diviser pour régner", dans le but de décomposer le processus d'extraction des itemsets fréquents en tâches plus petites, chacune d'elles représentant l'étude d'une partie de la base de données.

L'autre problème de la génération des règles d'association (ou règles associatives) est celui des structures de données utilisées pour représenter une base de transactions. Le but de ces structures est de réduire les coûts des entrées/sorties, engendrés par les accès disque, et de rendre les traitements plus rapides. Ainsi, le choix de la structure de données la plus adéquate possible est fondamental puisque celle-ci influe sur les performances de l'algorithme d'extraction des connaissances.

Le besoin de disposer de structures compactes a motivé de nombreux chercheurs qui ont proposé de nouvelles structures, notamment pour réduire les accès disque. Deux grandes classes de structures émergent: les structures de données pour la représentation de la base de transactions et les structures de données pour la génération et le stockage des ensembles d'attributs candidats. Parmi les structures proposées, nous pouvons citer: FP-Tree [18, 30], Patricia Tries [19, 56], Bitmap [22], Bit-Sliced Bloom-Filtered Signature [27], Transaction Tree [43], COFI-Tree [54] et Prefix Tree [55].

Dans le cadre de ces structures, notre principale contribution se présente sous la forme d'un nouveau modèle binaire et arborescent de représentation condensée d'une base de transactions et un nouvel algorithme ST-Mine d'extraction des ensembles d'attributs fréquents. A chaque transaction, nous associons une signature binaire pour obtenir un ensemble de signatures relatif à l'ensemble des transactions. Nous avons défini ensuite deux variantes: la première consiste à utiliser un fichier de signatures et un arbre de signatures TSF, alors que la seconde n'utilise qu'un arbre de signatures ST-Tree. Chacune des variantes a été ensuite utilisée dans le processus d'extraction des ensembles fréquents et la génération des règles d'association.

L'ensemble de nos contributions est résumé dans ce mémoire de thèse composé de 4 chapitres:

Le premier chapitre présente une synthèse sur les entrepôts de données, la fouille de données et les techniques de fouille de données. Il donne les caractéristiques des entrepôts de données ainsi que les différents modèles de leur représentation. Il définit ensuite les concepts de base de la fouille de données et réserve une grande partie à la technique de génération des règles d'association.

Le deuxième chapitre est divisé en deux parties. La première est consacrée à la présentation des principaux algorithmes d'extraction des ensembles fréquents ainsi que leurs avantages et inconvénients. Cette partie se termine par avec une classification de ces algorithmes selon les différentes stratégies qu'ils adoptent. La deuxième partie de ce chapitre présente les différentes structures de données utilisées pour représenter soit une base de transactions, soit les ensembles candidats.

Le troisième chapitre est consacré à la présentation de nos contributions. Nous commençons par définir les notions de signature binaire, de fichier de signatures et d'arbre de signatures binaires. Nous présentons ensuite les deux variantes que nous avons proposées, à savoir la variante basée sur un arbre de signatures et un fichier de signatures et la seconde variante qui est basée uniquement sur un arbre de signatures.

L'implémentation des algorithmes relatifs aux deux variantes proposées et à l'analyse des résultats obtenus constitueront le quatrième chapitre.

Enfin, le mémoire se termine par une conclusion qui résume l'essentiel de nos contributions et propose quelques perspectives de recherche futures.

CHAPITRE I

DES ENTREPOTS DE DONNEES A LA FOUILLE DE DONNEES

I. DES ENTREPOTS DE DONNEES A LA FOUILLE DE DONNEES

I.1. Introduction

Un entrepôt de données est défini par Bill Inmon comme "une collection de données thématiques (orientée sujet), intégrées, non volatiles et historisées pour le support d'un processus d'aide à la décision [83]. Les données thématiques regroupent les informations des différents métiers et ne tiennent pas compte de l'organisation fonctionnelle des données. Elles proviennent de systèmes sources hétérogènes ce qui nécessite leur intégration. Cette intégration exige une gestion de la cohérence et une normalisation (définition d'un référentiel unique), une maîtrise de la sémantique et une prise en compte des contraintes référentielles et des règles de gestion. L'historisation des données signifie que l'évolution des différentes valeurs des indicateurs (attributs clé) doit être suivie dans le temps. Nous obtenons ainsi plusieurs couches de données. Cette historisation des données entraine une non volatilité des données. En fait, les données ne sont pas supprimées, ce qui permet une traçabilité de toutes les valeurs prises par ces données. Les données historisées sont donc datées et persistent dans le temps. Il y a également la mise en place d'un référentiel temps. L'entrepôt de données donne donc une vision transversale de d'une entreprise [83].

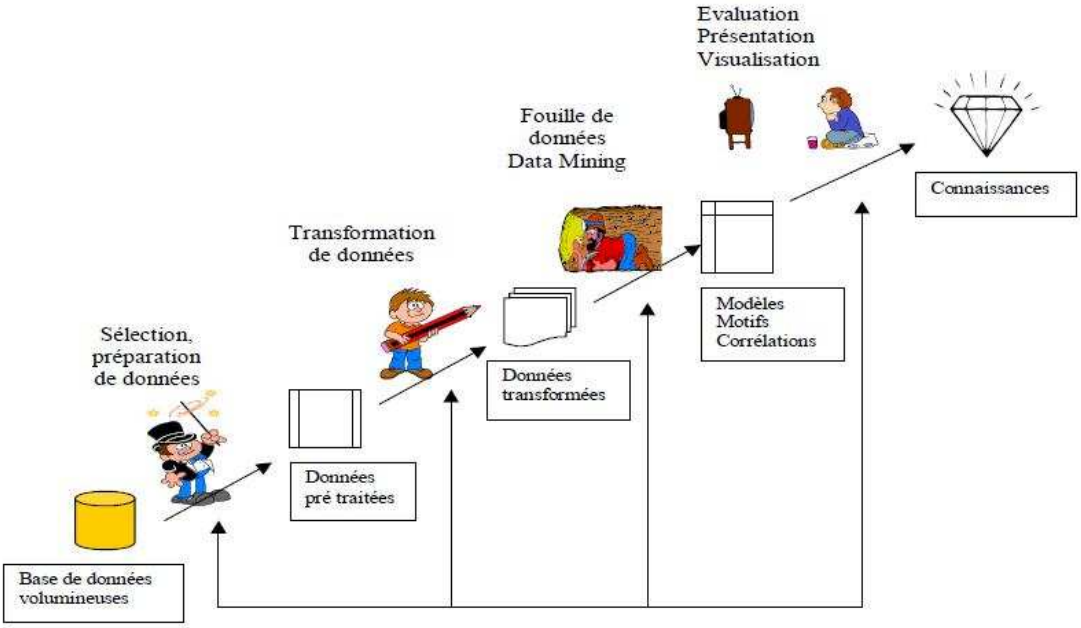


Figure 1. *Processus d'ECD*

L'Extraction de Connaissances à partir de Données (ECD) est une discipline récente [84], à l'intersection des domaines des bases de données, de l'intelligence artificielle, de la statistique, des interfaces homme/machine et de la visualisation. A partir de données collectées par des utilisateurs ou des processus automatiques de collecte, il s'agit d'extraire des connaissances nouvelles qui enrichissent les interprétations d'un champ d'application donné pour exploiter ces connaissances. L'ECD est décrite comme un processus interactif de préparation des données, d'extraction de connaissances à l'aide d'algorithmes, de visualisation et d'interprétation des résultats, lors d'interactions avec un expert (*Figure 1*). Les méthodes d'exploration proposent des solutions aux problèmes de recherche d'associations, de classification supervisée et non supervisée, de regroupement, etc...

Très souvent exprimée sous forme de règles d'association (ou règles associatives), la connaissance extraite requiert la mise au point d'algorithmes efficaces pour prendre en compte les difficultés algorithmiques liés à cette extraction. Les bases de données à explorer sont très larges en terme de volume et contiennent la description de centaines de millions d'objets par des milliers d'attributs et l'espace de recherche est de taille exponentielle en nombre d'attributs [84, 85].

Dans ce chapitre, la section I.2 définit la notion d'entrepôts de données. La section 1.3 donne un aperçu sur la fouille de données et la situe par rapport au processus d'ECD. La section I.4 présente, brièvement, les différentes techniques de fouille de données. La section I.5 détaille la technique de génération de règles d'association.

I.2. Caractéristiques des entrepôts de données

Un entrepôt de données reçoit un flux de données entrant et fournit un flux de données sortant. Le flux entrant passe par un processus d'extraction des données à partir de sources multiples et hétérogènes. Ces données sont ensuite transformées, filtrées, homogénéisées et nettoyées avant d'être stockées dans l'entrepôt de données. La zone de préparation (staging area) est une zone temporaire de stockage qui est souvent détruite après la génération de l'entrepôt de données. La zone de stockage, représentant l'entrepôt de données, est évidemment une zone de stockage permanente des données. La zone de présentation donne accès aux données contenues dans l'entrepôt de données. Elle peut contenir des outils d'analyse programmés (rapports, requêtes, …). Le flux sortant met ces données à la disposition

des utilisateurs. La *Figure 2* illustre l'architecture générale d'un entrepôt de données [82, 83].

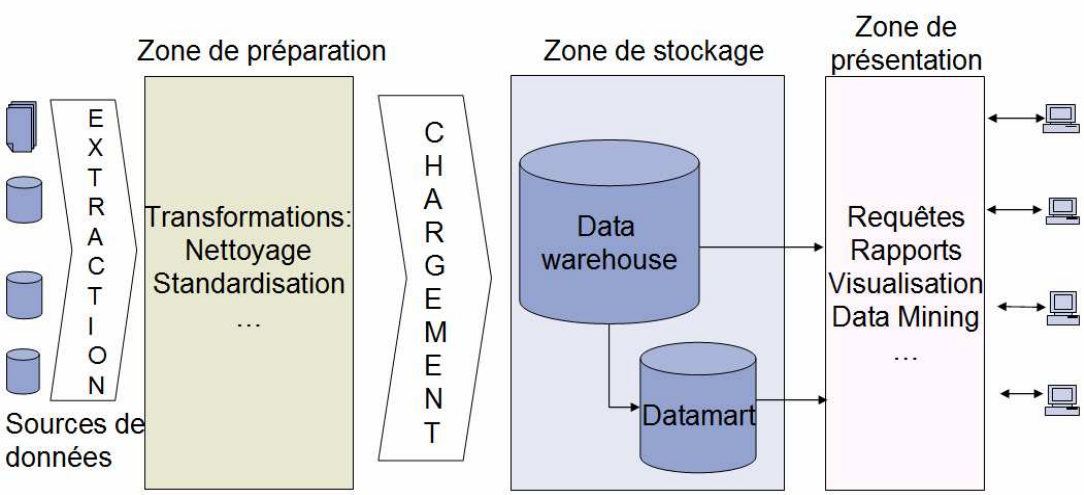


Figure 2. *Architecture générale d'un entrepôt de données*

I.2.1. Datamarts

Les données de l'entrepôt de données étant thématiques, il est donc possible d'extraire des données spécifiques aux besoins d'un secteur ou d'une fonction particulière. Les magasins de données (ou Datamarts) sont des sous-ensembles d'un entrepôt de données contenant des points de vue spécifiques selon des critères métiers. La *Figure 3* donne un exemple de magasins de données (service Marketing, service Ressources Humaines).

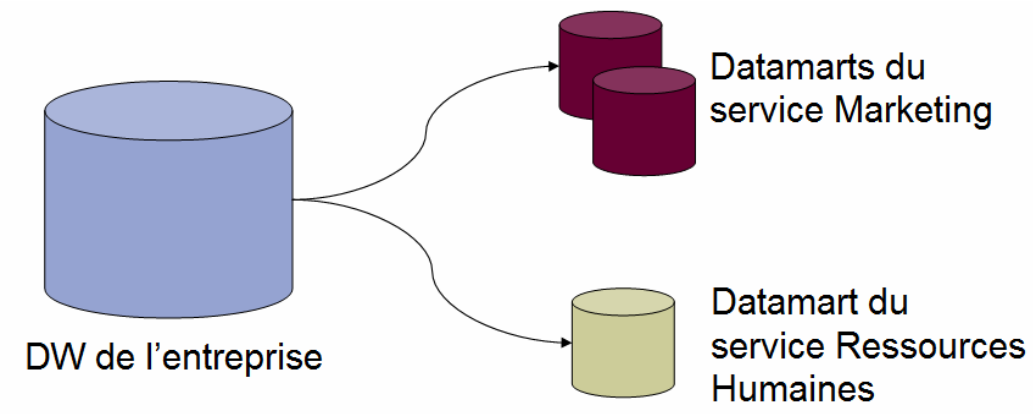


Figure 3. *Exemples de Magasins de données*

L'intérêt des magasins de données est qu'ils offrent un environnement structuré en fonction des besoins d'un métier ou d'un usage particulier. Il contient donc moins de données que l'entrepôt de données, ce qui entraîne une amélioration du temps de réponse aux requêtes [81].

I.2.2. Modélisation des entrepôts de données

Les entrepôts de données ont introduit une nouvelle méthode de conception autour des concepts métiers. On ne parle plus de normalisation au sens relationnel du terme. De nouveaux types de tables ont été également introduits, la table de faits et la table de dimensions. Le modèle en étoile et le modèle en flocon sont de nouveaux modèles de représentation des entrepôts de données.

1.2.2.1. Les tables

La table de faits est la table principale du modèle dimensionnel. Elle contient les données observables (les faits) sur le sujet étudié, selon divers axes d'analyse (les dimensions). La *Table 1* illustre l'exemple d'une entreprise contenant des magasins de vente de produits.

	Table de faits des ventes
Clés étrangères vers les dimensions	Clé date (CE)
	Clé produit (CE)
	Clé magasin (CE)
Faits	Quantité vendue
	Coût
	Montant des ventes

Table 1. *Tables de faits des ventes*

Les faits représentent ce que l'on veut mesurer (quantités vendues, montant des ventes, coût). La table des faits contient également les clés étrangères des axes d'analyse (dimension Date, dimension Produit et dimension Magasin).

Les tables de dimension représentent les axes selon lesquels vont être étudiées les données observables (faits). Chaque table contient le détail sur les faits. La *Table 2* donne un exemple de la dimension Produit [81].

	Dimension produit
Clé de substitution	Clé produit (CP)
Attributs de la dimension	Code produit
	Description du produit
	Famille du produits
	Marque
	Emballage
	Poids

Table 2. *Table dimension Produit*

I.2.2.2. Les types de modèles

Le modèle en étoile et le modèle en flocon sont les deux modèles les plus utilisés pour modéliser un entrepôt de données.

a. Modèle en étoile

Ce modèle est composé d'une table de faits centrale et de tables dimensions. Les dimensions n'ont pas de liens entre elles.

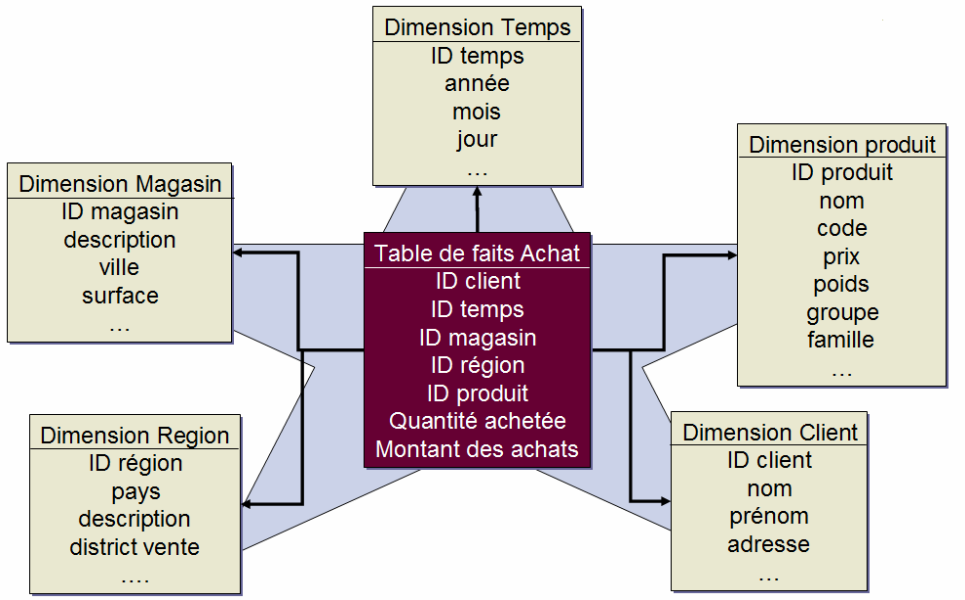


Figure 4. *Modélisation en étoile*

Les avantages de ce modèle sont la facilité de navigation et le nombre restreint de jointures.

Les inconvénients principaux sont la redondance dans les dimensions et le fait que toutes les dimensions ne concernent pas les mesures. La *Figure 4* donne un exemple du modèle en étoile [81].

b. Modèle en flocon

Le modèle en flocon est composé quant à lui d'une table de faits et un ensemble de tables dimension décomposées en sous-hiérarchies. Chaque table dimension a un seul niveau hiérarchique. La table de niveau hiérarchique le plus bas, ayant donc la granularité la plus fine, est reliée à la table de faits

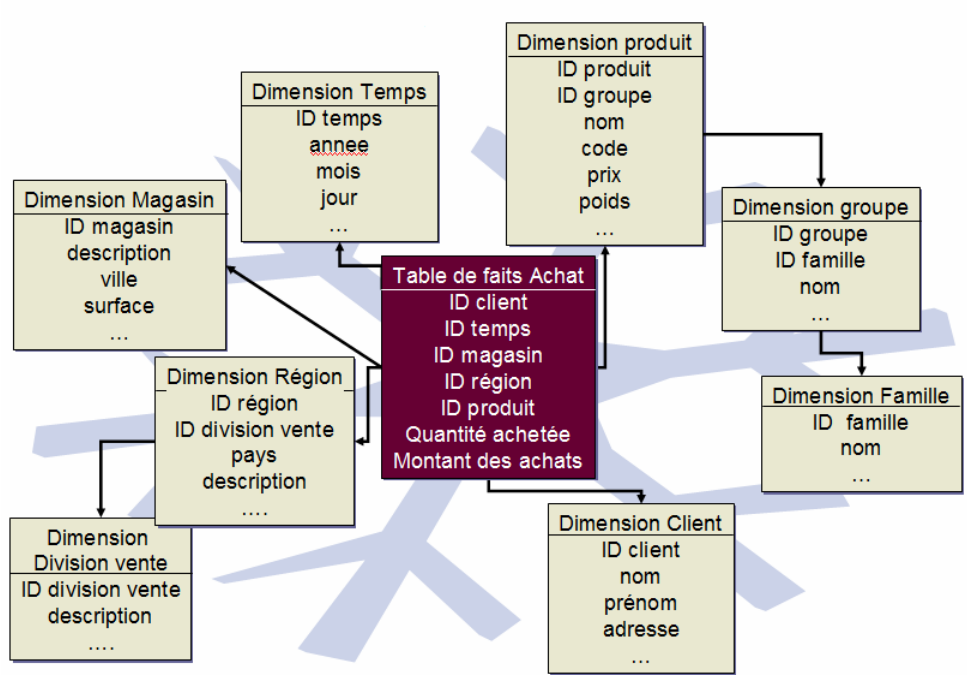


Figure 5. *Modélisation en flocon*

Si ce modèle permet une normalisation des dimensions et une économie d'espace disque, il est plus complexe et nécessite plusieurs jointures dans le traitement des requêtes. La *Figure 5* illustre un exemple du modèle en flocon [81].

I.3. Fouille de données

Frawley et Piatesky-Shapiro définissent la fouille de données comme "*l'extraction d'informations originales, auparavant inconnues, potentiellement utiles à partir de données*" [37]. John Page la considère comme "*la découverte de nouvelles corrélations, tendances et modèles par le tamisage d'un large volume de données*".

Par contre, Kamran Parsaye la définit comme "*un processus d'aide à la décision où les utilisateurs cherchent des modèles d'interprétation dans les données"* [80]. Dimitris Chorafas donne plutôt une définition singulière et étonnante, mais tellement proche de la vérité. Il considère que la fouille de données revient à "*torturer l'information disponible jusqu'à ce qu'elle avoue"* [86].

Le développement de la fouille de données est lié en fait à plusieurs facteurs:

- une puissance de calcul importante.
- un volume sans cesse croissant des bases de données.
- un accès plus simple aux réseaux et un accroissement du débit.

Ces facteurs rendent possible le calcul distribué et la distribution d'information sur un réseau d'échelle mondiale. La fouille de données a aujourd'hui une grande importance économique du fait qu'elle permet d'optimiser la gestion des ressources humaines et matérielles. Elle est utilisée, par exemple:

- dans les organismes de crédit pour décider d'accorder ou non un crédit, selon le profil et l'historique du demandeur de crédit.
- dans l'organisation des rayonnages dans les supermarchés en regroupant les produits qui sont généralement achetés ensemble, pour que les clients n'oublient pas d'acheter un produit situé à l'autre bout du magasin.
- pour l'aide au diagnostic médical.
- dans l'analyse du génome.
- la classification d'objets (astronomie, ...),
- l'analyse des pratiques et stratégies commerciales et leurs impacts sur les ventes.
- Dans les moteurs de recherche sur internet pour définir des profils d'utilisateurs.
- l'extraction de connaissances à partir de texte, d'image, de vidéo.

La fouille de données est le cœur du processus car elle permet d'extraire des connaissances à partir de données de l'information. Néanmoins, c'est souvent une étape difficile à mettre en œuvre, car très coûteuse et dont les résultats doivent être interprétés et relativisés.

Une approche traditionnelle pour découvrir ou expliquer un phénomène est de l'observer, de l'explorer, d'établir un modèle ou une hypothèse, d'essayer de le contredire ou de le vérifier, jusqu'à obtenir une réponse de qualité satisfaisante.

Nous pouvons considérer les outils de fouille de données comme des procédures qui permettent de faciliter ou encore d'automatiser ce processus.

La qualité du modèle obtenu se mesure selon plusieurs critères: rapide à créer, rapide à utiliser, compréhensible pour l'utilisateur, des performances, sa fiabilité, sa capacité à évoluer en fonction de l'évolution des données, etc...

Les choix effectués sont subjectifs, quoique guidés par la disponibilité des méthodes dans les environnements de fouille de données standard. Il va de soi qu'aucune méthode n'aura toutes ces qualités. Il n'existe pas de meilleure méthode de fouille de données. Il faudra faire des compromis selon les besoins dégagés et les caractéristiques connues des outils. Pour une utilisation optimale, une combinaison de méthodes est nécessaire.

Par conséquent, à tout jeu de données et à tout problème correspond une ou plusieurs méthodes. Le choix se fera en fonction de la tâche à résoudre, de la nature et de la disponibilité des données, des connaissances et des compétences disponibles, de la finalité du modèle construit. Pour cela, la complexité de la construction du modèle, la complexité de son utilisation, ses performances, sa pérennité et, plus généralement, l'environnement de l'entreprise, deviennent des critères très importants à prendre en compte [80, 84, 85, 86].

I.4. Techniques de Fouille de Données

Nous distinguons différentes techniques de fouille de données. Ces techniques sont classées en deux grandes catégories: les techniques non-supervisées ou descriptives et les techniques supervisées ou prédictives.

Dans le cas des techniques non supervisées, on cherche à extraire des informations nouvelles et originales à partir d'un ensemble de données dont aucun attribut n'est plus important qu'un autre. Le résultat obtenu doit être analysé afin d'être retenu pour être utilisé ou tout simplement rejeté. Ces ont un rôle principalement descriptif. Elles s'attachent à isoler l'information utile au sein d'un jeu de données. Elles permettent de constituer des groupes homogènes d'objets pour, par exemple, grouper des patients qui ont le même comportement. Selon l'objectif visé, plusieurs techniques existent (Classification hiérarchique, Segmentation des K moyennes, Règles d'Associations ou Règles Associatives, etc…) [77].

Par contre, l'objectif des techniques supervisées est d'apprendre, à l'aide d'un ensemble d'entraînement, des règles qui permettent de prédire (ou « deviner ») certaines caractéristiques de nouvelles observations. Le principe est que, dans tous les cas, on utilise des données « historiques » ou connues pour construire un modèle. Ce modèle est ensuite utilisé dans le but de classer les nouvelles observations (les K plus proches voisins, Arbres de décision, Réseaux de neurones, …).

Les techniques supervisées peuvent aussi servir à faire de la prédiction de manière générale. Un médecin peut ainsi adapter le traitement d'un patient en fonction de ses attributs. Elles permettent d'évaluer la distribution d'une quantité (la taille d'un individu par exemple) sans la mesurer directement, mais en se basant sur des valeurs qui lui sont liées (le poids de la personne par exemple) [79].

Parmi les techniques définies ci-dessus, nous avons utilisé les techniques non supervisées et plus particulièrement la technique de génération des règles d'association.

I.5. Règles d'Association

Le concept de règles d'association a été introduit en 1993 par Agrawal [3]. Les règles d'association sont traditionnellement liées au secteur de la distribution. Leur principale application est ***"l'analyse du panier de la ménagère"*** [93]. L'analyse des tickets de caisse des clients permet de comprendre leurs habitudes de consommation, d'agencer les rayons d'une surface commerciale, d'organiser les promotions, de gérer les stocks, etc. Dans les bases de données de vente, un tuple est une transaction composée d'un ensemble d'articles achetés ou *items*. La base de données est donc un *ensemble de transactions appelé base transactionnelle* ou *base de transactions*. Une règle d'association décrit une corrélation entre des ensembles d'items, ou *itemsets*, dans une base de transactions. Autrement dit, étant donnés deux itemsets I_1 et I_2, l'objectif est de découvrir une transaction contenant I_1 et I_2 [5].

Exemple: *"80% des clients qui achètent un ordinateur achètent aussi une imprimante et un abonnement à Internet"* est une règle d'association associant l'itemset {*ordinateur*} à l'itemset {*imprimante, abonnement à Internet*}.

La technique qui génère des règles d'association peut être appliquée à tout secteur d'activité pour lequel il est intéressant de rechercher des groupements potentiels de produits ou de services: services bancaires, services de télécommunications, par

exemple. Elle peut être également utilisée dans le secteur médical pour la recherche de complications dues à des associations de médicaments ou à la recherche de fraudes en recherchant des associations inhabituelles [13, 14].

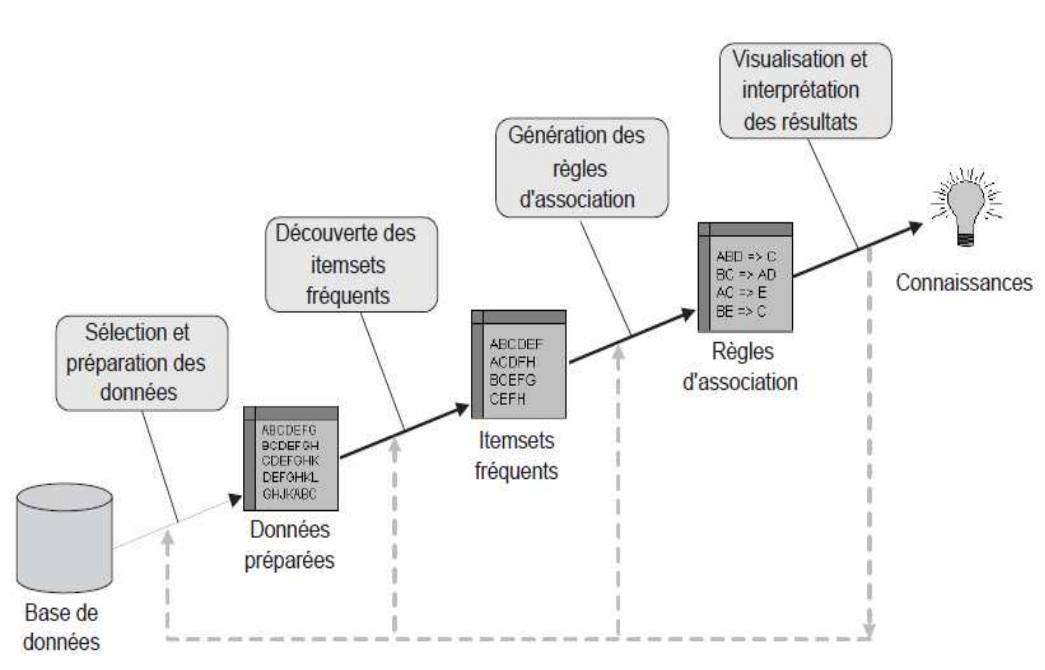


Figure 6. *Etapes du Processus de Génération des Règles d'Association*

Ces règles montrent comment des produits ou des services se situent les uns par rapport aux autres. Elles peuvent ainsi être facilement utilisées dans le système d'information de l'entreprise. Cependant, il faut noter que la méthode, si elle peut produire des règles intéressantes, peut aussi produire des règles triviales ou inutiles. La *Figure 6* illustre les différentes phases de traitements des données pour arriver à l'extraction de la connaissance.

I.5.1. Quelques concepts de base

La technique de génération des règles d'association se base sur un ensemble de concepts. Nous considèrerons, dans notre thèse, que:

- Un item i_i est tout objet, article, attribut, littéral, appartenant à un ensemble fini d'éléments distincts $S = \{i_1, i_2, \ldots, i_n\}$.
- Un itemset I est un sous-ensemble d'items de S. Un k-itemset est un itemset composé de k items.
- Une transaction T_j est un itemset auquel est associé un identificateur unique: le Tid (Transaction Identifier).

- L'ensemble des Tid et des transactions sont stockés dans une base de transactions T.
- La liste des Tid (TidList) associée à un itemset représente l'ensemble des Tid des transactions qui le contiennent.
- Le support d'un itemset I, noté Support(I), est la proportion de transactions de T contenant I. La valeur du support est comprise entre [0,1]. Il peut également être exprimé en pourcentage.
- Un support minimum, Minsup, est une valeur minimale de support choisi par l'utilisateur.
- Un itemset I est dit fréquent si son support Support(I) est supérieur au support minimum Minsup.
- Une règle associative ou règle d'association R, est définie comme une implication de la forme R: $I_1 \rightarrow I_2$, tel que $I_1 \subset S$, $I_2 \subset S$ et $I_1 \cap I_2 = \emptyset$, I_1 est appelé "Condition" et I_2 "Conclusion".
- Le support d'une règle, notée Support(R), correspond au rapport du nombre de transactions, où apparaissent simultanément les items de la condition et du résultat, sur le nombre de transactions total. Le support permet de mesurer la fréquence de l'association.
- La confiance d'une règle, notée Confiance(R), correspond au rapport du nombre de transactions où apparaissent simultanément les items de la condition et de la conclusion sur le nombre de transactions où apparaissent simultanément les items de la condition. La confiance permet de mesurer la force de l'association.
- La règle $I_1 \rightarrow I_2$ est dite "*pertinente*" ou "*satisfaite*", dans l'ensemble des transactions T, si sa confiance est supérieure à une confiance minimale Minconf fixée par l'utilisateur.

I.5.2. Processus de Génération des Règles d'Association

Le processus de génération des règles d'association se divise en deux étapes principales. Une première étape qui consiste à extraire les itemsets fréquents, itemsets ayant un support supérieur ou égal à un support minimum Minsup, et une étape de génération des règles associatives à partir de ces itemsets fréquents. Si X est un k-itemset fréquent, avec k > 1, les règles générées auront comme conclusion un sous-ensemble Y de X, et comme condition l'itemset X-Y. La

confiance de chacune des règles, Confiance(R), est comparée à la confiance minimum Minconf, fixée à priori par l'utilisateur. Seules les règles qui satisfassent la relation Confiance(R) ≥ Minconf sont gardées.

L'étape d'extraction des itemsets fréquents est la phase la plus coûteuse. Elle présente une complexité exponentielle en fonction de la taille de la base des transactions. Ainsi, l'amélioration des performances des algorithmes de découverte de règles associatives passe nécessairement par l'optimisation et la réduction du temps d'exécution de la phase d'extraction des itemsets fréquents.

Un grand nombre d'algorithmes d'extraction des itemsets fréquents a été proposé durant ces dernières années. Le chapitre suivant sera consacré à la présentation des algorithmes les plus connus.

Nous donnons ci-dessous un exemple de génération des règles d'association.

Considérons la base de transactions T suivante:

T = {A C D, A B C E, B C E, B E, A B C E, B C E}

Avec Minsup = 2/6 et Minconf = 2/3

Il existe 32 itemsets candidats possibles:

0-itemset :

{} (Trivial)

1-itemset :

{A}, {B}, {C}, {D}, {E}

2-itemsets :

{AB}, {AC}, {AD}, {AE}, {BC}, {BD}, {BE}, {CD}, {CE}, {DE}

3-itemsets :

{ABC}, {ABD}, {ABE}, {ACD}, {ACE}, {ADE}, {BCD}, {BCE}, {BDE}, {CDE}

4-itemsets :

{ABCD}, {ABCE}, {ABDE}, {ACDE}, {BCDE}

5-itemsets :

{ABCDE}

Le support d'un itemset étant la proportion de transactions contenant l'itemset, le support de l'itemset BC sera égal à Support ({BC}) = |{2, 3, 5, 6}| / |T| = 4/6.

Le support d'une règle d'association est égal au support de l'union de la condition et de la conclusion. Si nous prenons la relation R: BC $\rightarrow$ E, le support de R sera égal à Support(R) = Support({BCE}) = |{2, 3, 5, 6}| / |T| = 4/6.

La confiance d'une règle est égale à la proportion de transactions vérifiant l'implication. La confiance de R sera donc Confiance(R) = Support({BCE}) / Support({BC}) = 1 [(4/6) / (4/6)]

Nous dirons qu'une règle d'association est valide si son support est ≥ à Minsup et sa confiance ≥ Minconf. Les règles, pour être utile du point de vue de la connaissance, doivent concerner une population suffisamment importante et être statistiquement significatives.

La *Table 3* illustre le processus d'extraction des itemsets fréquents de l'exemple précédent.

La génération des k-itemsets candidats se fait à partir des (k-1)-itemsets fréquents. Si nous considérons, par exemple, l'ensemble des 3-itemsets fréquents {ABC, ABD, ACD, ACE, BCD}, la génération des 4-itemsets candidats se fait par jointure des 3-itemsets fréquents de même préfixe.

- ABC et ABD génère ABCD
- ACD et ACE génère ACDE

Tous les sous-ensembles possibles, {ABC}, {ABD} et {ACD}, de l'itemset {ABCD} sont fréquents, donc {ABCD} est fréquent et il est conservé. Le sous ensemble {ADE} de l'itemset {ACDE} est infréquent donc {ACDE} ne peut être fréquent et il supprimé.

Nous définissons un ordre total, noté "<", sur les items d'un ensemble fini I, s'il existe une fonction f: S $\rightarrow$ {1,…, |S|} qui associe à chaque item de S un entier. L'ensemble S muni de l'ordre "<" entre items de I sera noté (S, <).

Nous définirons un treillis par un ensemble ordonnée (S, <), où chaque couple d'itemsets (I_1, I_2) possède une borne inférieure noté $I_1 \wedge I_2$ et une borne supérieure notée $I_1 \vee I_2$. On notera la borne inférieure de S par Inf (I_1, I_2) et la borne supérieure par Sup (I_1, I_2).

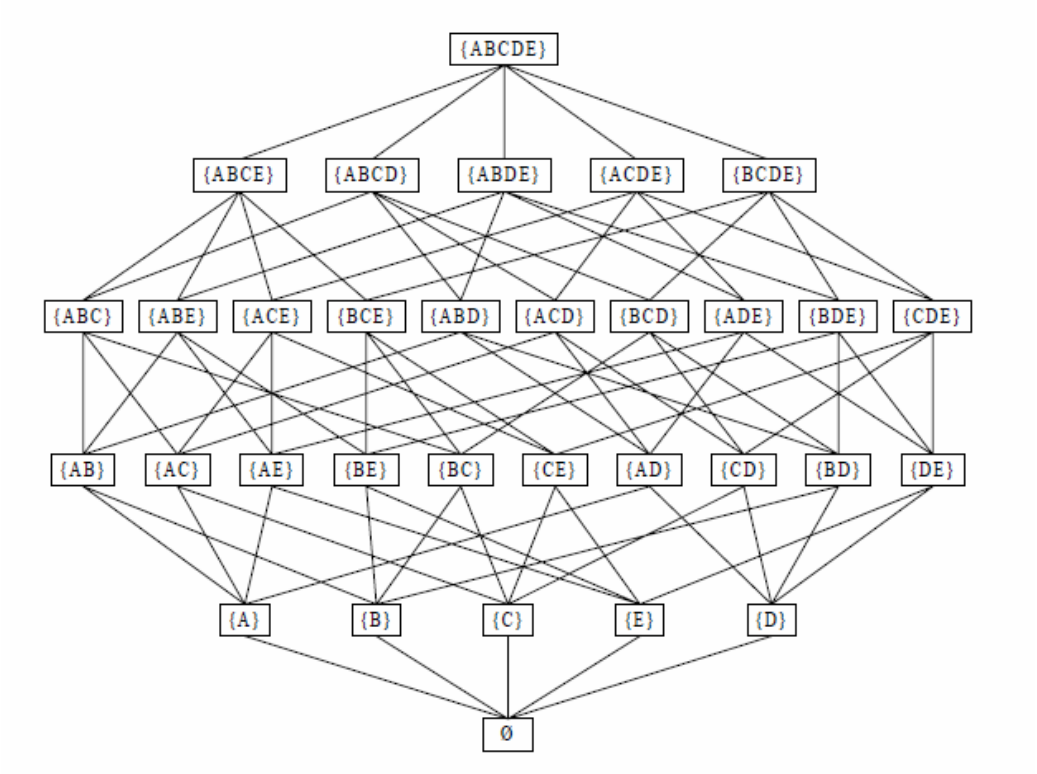


Figure 7. *Treillis des itemsets candidats*

Si nous considérons l'exemple précédent, nous obtenons le treillis des itemsets candidats illustré par la *Figure 7*.

L'élagage dans le treillis des itemsets candidats permet d'éliminer les itemsets non fréquents. Le processus d'élagage commence par éliminer les sur-ensembles d'un itemset infréquent et considérer les sous-ensembles d'un itemset fréquent comme fréquents. {ACE} étant infréquent donc {ABCE}, {ACDE} et {ABCDE} sont infréquents. {ABC} étant fréquent, {AB}, {AC}, {BC}, {A}, {B}, {C} seront fréquents.

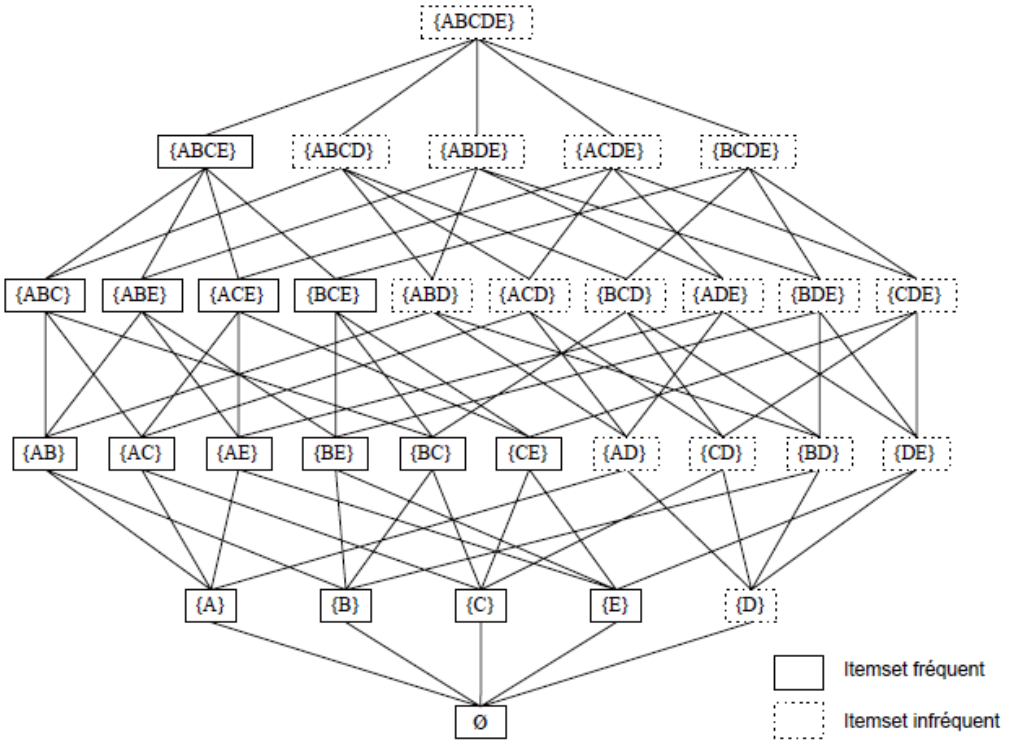


Figure 8. *Treillis des itemsets fréquents*

Il est ensuite nécessaire d'utiliser des méthodes efficaces pour limiter le nombre de balayages du treillis et le nombre d'itemsets examinés. Le parcours du treillis des itemsets candidats doit se faire par niveaux. Ainsi, l'itération k traite les itemsets candidats de taille k. La *Figure 8* illustre le treillis des itemsets fréquents. La *Table 3* montre comment se fait la génération des règles d'association, à partir des itemsets fréquents de taille maximale.

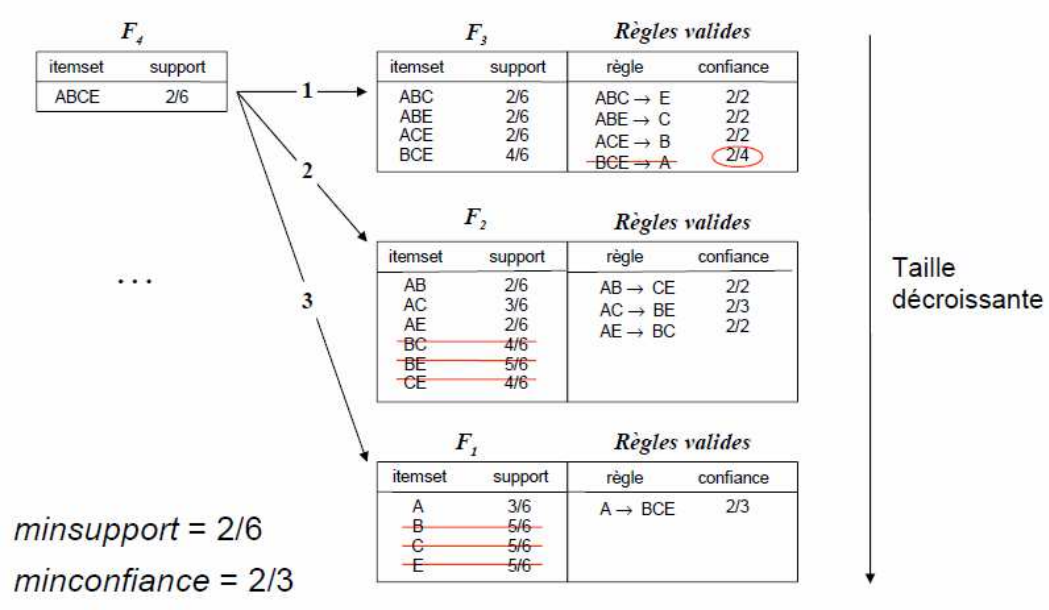


Table 3. *Processus de Génération des Règles d'Association*

I.6. Conclusion

Dans ce chapitre, nous avons donné un aperçu sur les entrepôts de données et la fouille de données. Nous avons également introduit les différentes techniques de fouille de données, en donnant plus de détails sur la technique de génération des règles d'association, qui seront utilisées dans les chapitres suivants de ce manuscrit.

CHAPITRE II

PROCESSUS D'EXTRACTION DES ITEMSETS FREQUENTS

II. PROCESSUS D'EXTRACTION DES ITEMSETS FREQUENTS

II.1. Introduction

Une grande partie des algorithmes réalisés adopte une approche de génération et de test des itemsets candidats tels qu'Apriori. Seulement, cette phase est la plus coûteuse, vu le grand nombre d'accès à la base de transactions. Pour calculer les supports d'une collection d'itemsets, il est nécessaire d'accéder à la base de transactions. Comme les bases de transactions sont généralement volumineuses, une solution permettant d'éviter les accès répétitifs et coûteux, consiste à les représenter par des structures compactes.

Dans ce chapitre, nous donnons un aperçu des algorithmes d'extraction des itemsets fréquents. Nous réservons une grande place à l'algorithme Apriori, son principe et ses limites. Nous présentons également quelques algorithmes améliorant certains aspects d'Apriori et d'autres, comme FP-Growth, qui proposent une approche différente (extraction des itemsets fréquents sans génération de candidats). La deuxième partie du chapitre sera consacrée aux structures de données utilisées soit pour représenter une base de transactions, soit pour le stockage des candidats.

II.2. Algorithmes d'extraction des itemsets fréquents

Nous allons consacrer cette partie à la description de plusieurs algorithmes d'extraction des itemsets fréquents. Nous présenteront les avantages et les inconvénients de chacun d'eux.

II.2.1. Algorithme Apriori

L'algorithme Apriori a été élaboré par Agrawal [5, 6, 8]. Il s'applique à des bases de données contenant des transactions. De cet ensemble de transactions, Apriori est capable de sortir des règles liant les transactions entre elles. Ainsi, à partir de l'ensemble des transactions effectuées dans un centre commercial par exemple, il peut déduire que 80% des personnes qui s'équipent de pneus et accessoires automobiles, s'intéressent aussi aux services offerts pour leurs autos.

Apriori est à la base de plusieurs versions améliorant ses performances, comme il a été utilisé en combinaison avec d'autres techniques (comme la classification ou la segmentation) pour certains problèmes.

Apriori est un algorithme qui parcourt la base des transactions par niveau. Il procède de manière itérative: il commence par trouver les 1-itemsets fréquents, c'est à dire les sous-ensembles d'un seul item qui ont une fréquence supérieure à

un seuil minimum (seuil souvent fixé par l'utilisateur). A partir de cet ensemble, il détermine les 2-itemsets fréquents, c'est à dire les ensembles de deux items ayant une fréquence supérieure au seuil minimum. L'algorithme continue jusqu'à ce qu'il ne trouve plus d'itemsets fréquents [52].

Il repose sur le fait que les ensembles fréquents potentiels forment un treillis possédant des propriétés intéressantes permettant d'éviter une énumération exhaustive des ensembles fréquents potentiels et parcourt ce treillis en largeur [50, 53, 59, 69].

II.2.1.1. Propriétés

L'algorithme Apriori s'appuie essentiellement sur la propriété d'antimonotonicité dans le processus de construction des itemsets candidats. Les itemsets candidats de taille k sont obtenu à partir des itemsets fréquents de taille k-1. Ci qui nous permet de déduire que tous les sous-ensembles d'un ensemble fréquent sont fréquents et que tous les sur-ensembles d'un ensemble non fréquent sont non fréquents. Ce qui permet d'éliminer un ensemble candidat si un seul de ses sous-ensembles est non fréquent.

Si l'itemset I_1 est inclus dans l'itemset I_2 alors Fréquence(I_1) $\geq$ Fréquence(I_2) car toutes les transactions qui contiennent I_2 contiennent nécessairement I_1.

L'algorithme Apriori est composé de trois phases, une phase qui génère un itemset candidat, une phase de parcours de la base de transactions qui calcule le support de l'itemset et une phase qui compare ce support avec le support minimum choisi par l'utilisateur. Ces trois phases sont répétées pour tous les k-itemsets candidats générés.

II.2.1.2. Pseudo-code de L'algorithme

L'algorithme Apriori, ayant en entrée un ensemble d'items, une base de transactions et un support minimum Minsup fixé par l'utilisateur, fournit en sortie un ensemble d'itemsets fréquents.

L'algorithme Apriori utilise une approche itérative par niveaux. Pour cela, le treillis des itemsets candidats est exploré en largeur d'abord. Apriori effectue, à chaque itération k, un accès à la base de transactions afin de calculer le support de chaque k-itemset.

Par la suite, C_k désignera l'ensemble des k-itemsets candidats et F_k l'ensemble des k-itemsets fréquents.

La base T des transactions est d'abord parcourue pour trouver les 1-itemsets fréquents F_1. L'algorithme alterne ensuite génération des itemsets candidats et extraction des itemsets fréquents parmi ces candidats.

Pour cela, à l'itération k, l'ensemble F_{k-1} des (k-1)-itemsets fréquents correspondant aux itemsets de niveau k-1 du treillis (calculé à l'étape précédente), est utilisé pour générer l'ensemble C_k des k-itemsets candidats. La fonction de génération de candidats Apriori-Gen prend en argument F_{k-1} et retourne C_k. Une jointure est d'abord effectuée entre F_{k-1} et F_{k-1}. La génération d'un k-itemset candidat est obtenue, à l'aide d'un ordre lexicographique, par la concaténation de deux (k-1)-itemsets fréquents ayant (k-2) items en commun. La fonction Apriori-Gen s'assure, après avoir généré un candidat de taille k à partir de deux (k-1)-itemsets fréquents, que le nouveau candidat ne contient pas un sous-ensemble peu fréquent, auquel cas le candidat lui même serait peu fréquent selon la propriété d'antimonotonicité. Une fois l'ensemble C_k des candidats de taille k généré, la base de transactions est parcourue transaction par transaction afin de déterminer le support de chaque candidat. La fonction *Subset* (C_k, *t)* recherche parmi les candidats de C_k ceux qui sont contenus dans la transaction t. Si c'est le cas, le support de ces candidats est augmenté. La recherche est optimisée en utilisant un arbre de hachage pour stocker les itemsets candidats. Parmi ces derniers, seuls les candidats fréquents sont gardés dans l'ensemble F_k. L'ensemble F de tous les itemsets fréquents est ensuite mis à jour. C'est cet ensemble qui est retourné à la fin du processus d'extraction des itemsets fréquents d'Apriori. Ci-dessous un exemple du processus de génération des règles d'association à l'aide de l'algorithme Apriori.

Considérons la base des transactions T de la *Table 4*, contenant 9 transactions. Nous supposons que Minsup = 2/9 et Minconf = 70%. La génération des règles d'association nécessite en premier d'extraire les itemsets fréquents et ensuite de générer les règles d'association, à partir de ces itemsets.

Tid	Transactions
T_1	I_1, I_2, I_5
T_2	I_2, I_4
T_3	I_2, I_3
T_4	I_1, I_2, I_4
T_5	I_1, I_3
T_6	I_2, I_3
T_7	I_1, I_3
T_8	I_1, I_2, I_3, I_5
T_9	I_1, I_2, I_3

Table 4. *Exemple de base de transactions T*

Le premier parcours de T a pour objectif de calculer le support des 1-itemsets candidats et ne garder que ceux qui ont un support ≥ Minsup.

1-Itemset	Support		1-Itemset	Support
I_1	6	Comparer les supports avec Minsup →	I_1	6
I_2	7		I_2	7
I_3	6		I_3	6
I_4	2		I_4	2
I_5	2		I_5	2
C1			**L1**	

Table 5. *Extraction des 1-itemsets fréquents*

La *Table 5* illustre cette première étape.

La deuxième étape consiste à générer les 2-itemsets candidats à partir des 1-itemsets fréquents de l'étape précédente, de calculer les supports de chacun d'eux pour ne garder que ceux qui ont un support ≥ Minsup. La *Table 6* Illustre l'étape d'extraction des 2-itemsets fréquents.

2-Itemset	Support		1-Itemset	Support
I_1, I_2	4	Comparer les supports avec Minsup →	I_1, I_2	4
I_1, I_3	4		I_1, I_3	4
I_1, I_4	1		I_1, I_5	2
I_1, I_5	2		I_2, I_3	4
I_2, I_3	4		I_2, I_4	2
I_2, I_4	2		I_2, I_5	2
I_2, I_5	2			
I_3, I_4	0			
I_3, I_5	1			
I_4, I_5	0			
C2			**L2**	

Table 6. *Extraction des 2-itemsets fréquents*

L'étape consiste à générer les 3-itemsets candidats, de calculer les supports associés et d'extraire les 3-itemsets fréquents. Cette étape est représentée par la *Table 7*.

1-Itemset	Support		1-Itemset	Support
I_1, I_2, I_3	2	Comparer les supports avec Minsup →	I_1, I_2, I_3	2
I_1, I_2, I_5	2		I_1, I_2, I_5	2
C3			**L3**	

Table 7. *Extraction des 3-itemsets fréquents*

La génération des 3-itemsets candidats C_3 utilise la propriété d'antimonotonicité d'Apriori. Nous devons, pour cela, calculer la jointure de L_2 avec L_2. Nous obtenons:

$C_3 = \{\{I_1, I_2, I_3\}, \{I_1, I_2, I_5\}, \{I_1, I_3, I_5\}, \{I_2, I_3, I_4\}, \{I_2, I_3, I_5\}, \{I_2, I_4, I_5\}\}$.

Le processus d'élagage, basé sur la propriété d'Apriori " tous les sous-ensembles d'un itemset fréquent sont fréquents", va nous permettre de réduire la taille de l'ensemble C_3.

Considérons l'itemset $\{I_1, I_2, I_3\}$. Nous constatons que tous ses sous-ensembles possibles $\{I_1, I_2\}$, $\{I_1, I_3\}$ et $\{I_2, I_3\}$ sont fréquents. Donc, nous pouvons le garder dans C_3.

Si nous considérons par contre l'itemset $\{I_2, I_3, I_5\}$, on constate qu'au moins un de ses sous-ensembles, l'itemset $\{I_3, I_5\}$ n'est pas fréquent. Après la jointure et l'élagage, C3 contiendra donc $\{\{I_1, I_2, I_3\}, \{I_1, I_2, I_5\}\}$. Le calcul du support de chacun des itemsets candidat donne $L_3 = C_3$.

L'algorithme utilise la jointure de L_3 avec L_3 pour générer l'ensemble des 4-itemsets candidats C_4. $\{I_1, I_2, I_3, I_5\}$ est le seul itemset généré. Cet itemset est élagué puisque son sous-ensemble $\{I_2, I_3, I_5\}$ n'est pas fréquent. Nous obtenons $C_4 = Ø$ et l'algorithme se termine avec l'extraction de tous les itemsets fréquents. Ces itemsets vont servir à générer les règles d'association.

Considérons un des itemset fréquent de taille 3, l'itemset $\{I_1, I_2, I_5\}$. Prenons tous ses sous-ensembles non vides, $\{I_1\}$, $\{I_2\}$, $\{I_5\}$, $\{I_1, I_2\}$, $\{I_1, I_5\}$, $\{I_2, I_5\}$.

Si nous supposons que Minconf = 70%. Nous obtenons les relations suivantes et leurs confiances respectives:

R_1: $I_1, I_2 \rightarrow I_5$ Confiance = 2/4 = 50% $\Rightarrow$ R_1 est éliminée

R_2: $I_1, I_5 \rightarrow I_2$ Confiance = 2/2 = 100% $\Rightarrow$ R_2 est gardée

R_3: $I_2, I_5 \rightarrow I_1$ Confiance = 2/2 = 100% $\Rightarrow$ R_3 est gardée

R_4: $I_1 \rightarrow I_2, I_5$ Confiance = 2/6 = 33% $\Rightarrow$ R_4 est rejetée

R_5: $I_2 \rightarrow I_1, I_5$ Confiance = 2/7 = 29% $\Rightarrow$ R_5 est rejetée

R_6: $I_5 \rightarrow I_1, I_2$ Confiance = 2/2 = 100% $\Rightarrow$ R_1 est gardée

Seules R2, R3 et R6 ont une confiance $\geq$ Minconf.

La *Table 8* ci-dessous illustre le pseudo-code de l'algorithme Apriori.

Algorithme Apriori
Début

1. Calculer le support de chaque item
2. Générer les 1-itemsets fréquents
3. k ← 2
4. **Répéter**
5. Joindre les (*k*-1)-itemsets fréquents pour former les *k*-candidats
6. Supprimer les *k*-candidats ayant un (*k*-1)-subset infréquent
7. **Pour** chaque transaction **faire**
8. **Pour** chaque *k*-itemset candidat inclus dans T **Faire**
9. support++
10. **FinPour**
11. **FinPour**
12. Supprimer les *k*-candidats infréquents
13. *k*++
14. **Tantque** plusieurs *k*-candidats fréquents

Fin.

Table 8. *Pseudo-Code de l'algorithme APRIORI*

II.2.1.3. Problèmes d'Apriori

- une complexité exponentielle dans la mesure où, si l'on considère un ensemble d'items *I* de taille m, le nombre potentiel d'ensembles fréquents est 2^m.
- 10^4 1-itemsets fréquents génèrent 10^7 2-itemsets candidats.
- Pour trouver 100 itemsets on doit générer 2^{100} soit 10^{30} candidats.
- Pour un ensemble fréquent I donné le nombre potentiel de règles d'association est $2|I| - 2$.
- Le calcul des ensemble fréquents impose plusieurs scans de la base de données, On doit faire (n+1) scans pour trouver n-itemsets fréquents, ce qui devient très coûteux.
- Les articles rares sont éliminés à cause du support minimal alors qu'ils sont, dans certains cas, les plus importants. Parmi les règles générées,

beaucoup sont triviales, inutiles ou inexplicables, c'est-à-dire, qu'elles n'apportent rien à son utilisateur.

II.2.1.4. Améliorations d'Apriori

a) Algorithme Apriori TID

L'algorithme Apriori TID est une amélioration d'Apriori proposée par Agrawal et Srikant en 1994 [15], qui cherche à garder le contexte en mémoire afin de limiter les accès à la base de transactions. À supposer que les ensembles de candidats décroissent en même temps que la taille des candidats, il s'avère bien plus efficace qu'Apriori. [12]

b) Algorithme Count Distribution

L'algorithme Count Distribution proposé par R. Agrawal et J.C Shafer en 1996 [21] est une version parallélisée d'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 grande taille impose des opérations d'entrée/sortie coûteuses. Seuls les supports des candidats sont transmis lors d'une opération de Broadcast All-to-all [12].

c) Algorithme Data Distribution

Il s'agit encore d'un algorithme issu d'Apriori et proposé par R. Agrawal et J.C. Shafer en 1996 [21]. 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 des autres processeurs, ce qui augmente 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. [12]

d) Algorithme Sampling

L'algorithme Sampling [94] permet de travailler sur des volumes de données plus conséquents qu'Apriori, car il ne considère qu'un échantillon aléatoire de la base de transactions. Cet algorithme commence par effectuer

un échantillonnage de la matrice binaire de départ. Puis il lance un module d'extraction d'ensembles fréquents de type Apriori sur cet échantillon. La collection d'ensembles alors délivrée n'est peut être pas correcte vis-à-vis de la matrice initiale car elle a été calculée sur un échantillon. Sampling repasse ensuite la matrice en entier afin de vérifier si [23] :

- des conjonctions n'ont pas été oubliées.
- les conjonctions calculées sur l'échantillon sont effectivement fréquentes dans la matrice complète.

e) Algorithme Dynamic Itemset Counting

Proposé par S. Brin, R. Motwani, J. Ulman et S. Tsur en 1997, le but étant toujours de réduire le nombre de passes sur la base de données et minimiser les calculs, cet algorithme se base sur une représentation horizontale des données et un premier parcours en largeur du treillis. C'est en fait, une généralisation de l'algorithme d'Apriori. Comme précédemment, la base de transactions est partitionnée en p partitions de même taille [7]. Le principe de cet algorithme est de diviser les transactions en partitions. La première partition est utilisée par l'algorithme DIC pour calculer les supports de chaque item et générer les 1-itemsets fréquents locaux. L'algorithme DIC parcourt ensuite la deuxième partition pour calculer les supports des 2-itemsets candidats. Le même processus se répète sur les partitions restantes. Le support global d'un itemset est déterminé en parcourant toute la base de transactions.

L'algorithme DIC réduit le nombre de balayages de la base si la plupart des partitions sont homogènes (i.e. possédant les mêmes itemsets fréquents). Cependant, si la base de transactions n'est pas homogène, l'algorithme DIC peut générer des false drop (des itemsets localement fréquents mais non globalement fréquents), ce qui induit des parcours supplémentaires de la base.

f) Algorithme de Partitionnement (Partitionning)

Cet algorithme a été introduit par A.Savasere, E.Omiecinsky et S.Navathe en 1995, dans le but de réduire le nombre de passes sur la base de données initiale [42]. Il se base sur une représentation verticale et un premier parcours en largeur du treillis. Au cours de ce parcours, le treillis des

itemsets est parcouru niveau par niveau, les (k-1)-itemsets sont connus avant de générer les k-itemsets. Autrement dit, il suffit de dénombrer les itemsets selon leur longueur d'abord et par ordre lexicographique ensuite. Son principe est de diviser (d'un point de vue logique) la base de transactions en p partitions P_1,… P_m dont les tailles sont choisies de façon à ce que ces dernières puissent être chargées en mémoire centrale. L'idée sur laquelle se base cet algorithme est que: "si un itemset est fréquent, il doit être fréquent dans au moins l'une de ces partitions".

L'algorithme de partitionnement ("**Apriori partitionné")** se déroule en deux phases. La phase 1 consiste à diviser la base en Q partitions, chercher les sous-ensembles fréquents de chaque partition et ne conserver que ceux qui sont fréquents.

La phase 2 calcule le support de chaque itemset trouvé en parcourant toute la base de transactions. Les supports des itemsets sont calculés par intersection des listes de Tids et non pas par comptage comme dans Apriori.

On peut déduire, par exemple, à partir de l'itemset I_1 ayant pour TidList {1,2,5} et l'itemset I_2 de TidList {1,4,5}, le support de l'itemset {I1, I2} par $\{1,2,5\} \cap \{1,4,5\} = \{1,5\}$, représentant les transactions contenant I_1 et I_2.

En résumé, l'algorithme de Partition réduit le nombre de parcours de la base de données à deux parcours au plus. Dans le cas où la base de données peut être chargée entièrement dans la mémoire principale, un seul parcours est suffisant.

Le problème avec l'algorithme de Partition, bien qu'il nécessite au plus deux parcours, est que, plus le nombre de partitions augmente, plus le nombre d'itemsets localement fréquents augmente. Ces itemsets peuvent se révéler globalement peu fréquents. Ceci peut engendrer une perte de temps à faire des calculs redondants.

II.2.2. Algorithme Bitmap

L'algorithme Bitmap a été proposé par Gardarin et al en 1998 [22]. Il introduit un index binaire bitmap dans lequel chaque couple <ID-transaction, ID-article> est représenté par un bit. Ainsi, dans le bitmap, chaque colonne correspond à un k-itemset donné, tandis que chaque ligne caractérise une transaction. Cette

technique optimise le temps de calcul des supports car, pour calculer le support d'un itemset, il suffit d'appliquer des opérations booléennes (le "Et" logique) à l'ensemble des colonnes concernées dans l'index.

L'algorithme Bitmap se présente sous deux variantes. La variante *Bitmap naïf* utilise un bitmap pour déterminer quelle transaction contient quel item. La variante *Bitmap hiérarchique* se base sur le même principe sauf qu'elle repose sur une indexation à deux niveaux. Cette variante permet d'éliminer les opérations logiques inutiles, lorsque les groupes ne contiennent que des zéros. Elle présente un autre avantage qui est celui de ne pas mémoriser les bitmaps des k-itemsets, pour $k > 1$.

II.2.3. Algorithme FP-Growth

Pour remédier au problème de génération de candidats dont souffrent les algorithmes du même type qu'Apriori, Han et al ont proposé de remplacer la génération de candidats par une opération d'extension sur les itemsets introduisant ainsi un nouveau type d'algorithmes d'extraction des itemsets fréquents sans génération de candidats [18, 30, 31].

L'algorithme FP-Growth génère, en utilisant uniquement deux accès à la base des transactions, un arbre préfixé compact représentant toutes les transactions avec seulement les items fréquents.

L'algorithme FP-Growth ne génère donc pas de candidats mais ne fait que tester les candidats pour essayer d'en trouver des nouveaux, réduisant ainsi les calculs de façon significative. Puis, à partir du FP-Tree, et par une méthode récursive qui crée des projections (arbres conditionnels), l'algorithme FP-Growth découvre tous les itemsets fréquents en employant une technique de recherche basée sur le paradigme "Diviser pour Régner".

Un premier parcours de la base de transactions permet d'extraire la liste des 1-itemsets fréquents (f-liste). Cette liste est ensuite triée par ordre décroissant des supports. Le deuxième parcours de la base de transaction sert à trier le contenu de chacune des transactions suivant l'ordre des supports des 1-itemsets fréquents et d'insérer la transaction dans le FP-Tree en exploitant les préfixes communs. Dans le cas où le FP-Tree ne contient qu'une branche, l'algorithme génère les itemsets fréquents. Pour chaque item i appartenant à f-liste, il forme un nouvel

itemset en lui ajoutant i, génère la base conditionnelle relative à cet itemset et construit la liste des itemsets fréquents ainsi que le FP-Tree relatif à la base conditionnelle. Ce processus se répète tant que le FP-Tree construit est non vide.

Critique

- La construction récursive du FP-Tree affecte les performances de l'algorithme.
- La taille de l'arbre devient très grande pour les bases de données éparses. Elle peut atteindre, pour certaines bases de données, la taille des bases elles-mêmes. De ce fait, l'algorithme sera contraint d'effectuer un nombre important d'opérations pour concaténer les fragments d'itemsets, sans parvenir à trouver des itemsets fréquents.
- FP-Growth nécessite un espace mémoire important pour les structures auxiliaires utilisées (FP-Tree, bases conditionnels, …), surtout dans le cas où le support choisi est faible.

II.2.4. Algorithme DualFilter-Probe

Cet algorithme a été proposé par Lan en 2002 [27]. Une variante des fichiers de signatures, Bit-sliced Signature File, a été utilisée pour représenter de manière compacte la base de transactions. Les fichiers de signatures offrent une représentation pseudo-binaire de la base de transactions.

Principe

Le *DualFilter-Probe* adopte la stratégie "**Filtrer et raffiner**", il se déroule en deux étapes:

Etape 1: Découverte des itemsets fréquents.

L'estimation des supports des itemsets candidats se fait en parcourant le fichier de signatures au lieu de la base de transactions. Le calcul du support d'un itemset est fait en appliquant le *ET* logique à l'ensemble des bit-slices (colonnes) concernés.

Etape 2: Elagage des faux positifs (false drop).

Un faux positif est un itemset reconnu fréquent lors de la première phase alors qu'il ne l'est pas réellement. L'existence de faux positifs s'explique par le fait que les fichiers de signatures sont une représentation approximative de la base.

Réellement, ces deux phases sont intégrées pour détecter et éliminer les faux positifs qui peuvent être à la base de génération d'autres faux positifs.

Critique

Cet algorithme présente un inconvénient majeur qui réside dans la stratégie de génération des itemsets candidats utilisée. En effet, la stratégie utilisée traite des sur-ensembles d'itemsets alors que des sous-ensembles d'itemsets les constituant ne sont pas fréquents.

D'autre part, plus les bases de transactions sont larges plus la taille des bit-slices est importante. Ainsi, le temps de calcul des supports sera important.

II.2.5. Algorithme ITL - Mine

Goplan et al propose une nouvelle structure de données *Item-Trans-Link* (ITL) qui combine la représentation horizontale et verticale des bases de transactions [23]. La structure *Item-Trans Link* (ITL) est formée de deux composantes :

- *ItemTable* est une table qui contient les items et leurs supports et des liens vers la première occurrence de chaque item dans Trans-Link.
- Trans-Link représente les items de chaque transaction de la base.

L'algorithme *ITL - Mine* se base sur cette structure pour découvrir l'ensemble des itemsets fréquents.

Principe

La découverte des itemsets fréquents se fait en trois étapes:

Etape 1

La construction d'*ItemTable* et *Trans-Link* se fait par un seul parcours de la base de transactions. A la fin de cette étape, les *1-itemsets* fréquents seront identifiés dans la table *ItemTable* par calcul de leurs supports.

Etape 2

L'élagage: les items non fréquents seront supprimés de *Trans-Link* puisque ces items ne seront plus utiles dans l'étape d'après.

Etape 3

La découverte de tous les itemsets fréquents. Chaque item dans *ItemTable* sera utilisé comme point de départ pour l'extraction de tous les itemsets fréquents dont il est le préfixe.

II.2.6. Algorithme PatriciaMine

PatriciaMine est un algorithme qui adopte la stratégie de recherche en profondeur d'abord pour découvrir l'ensemble complet des itemsets fréquents. Cet algorithme utilise la structure de données *Patricia Tree* pour la représentation de la base de transactions [19, 56].

Patricia Tree est une condensation du *trie* standard qui est un arbre préfixé. Cette structure offre une représentation efficace du point de vue espace pour les bases denses aussi bien que pour les bases éparses.

Il est à noter que *PatriciaMine* suit une stratégie d'exploration itérative et non récursive, comme la plupart des algorithmes de découverte d'itemsets fréquents.

Principe

L'algorithme PatriciaMine se divise en trois étapes:

Etape 1

- Parcourir la base de transactions pour déterminer L, l'ensemble des items fréquents.

Etape 2

- Parcourir la base de transactions une deuxième fois en triant chaque transaction suivant l'ordre de la liste L et l'insérer dans le Patricia Tree.
- Initialiser l'itemset courant i à "Null".

Etape 3

Pour tout x de L faire:

- Concaténer x avec i.
- Générer i.

II.2.7. Algorithme Transaction Mapping (TM)

Cet algorithme utilise une représentation verticale des données. En effet, il propose une nouvelle technique qui consiste en la compression des Tids et leur transformation en "Listes d'intervalles" pour produire une nouvelle structure de données intitulée "Transaction Tree" (qui s'inspire de la structure FP-Tree). Il utilise aussi un arbre lexicographique pour déterminer les itemsets fréquents. Les itemsets fréquents sont extraits de l'intersection des listes d'intervalles de transactions de l'arbre lexicographique [43].

Principe

L'algorithme est composé de plusieurs étapes:

- Parcours de la base de transactions et détermination des 1-itemsets fréquents.
- Construction du "*Transaction Tree"* qui est un arbre similaire à un FP-Tree mais sans table d'entête ni de liens.
- Construction des listes d'intervalles de transaction pour chaque item. Les listes d'intervalles sont une compression des Tids. En effet, chaque liste d'intervalles correspond à une séquence contigüe de Tids relabellées. Les listes d'intervalles seront fusionnées si les intervalles sont contigus.
- Construction de l'arbre lexicographique en profondeur d'abord. Lors du traitement d'un nœud de l'arbre lexicographique, pour chaque élément de ce nœud, la liste d'intervalle correspondante est calculée. Durant la recherche, les itemsets remplissant la condition sur le support seront produits.

II.2.8. Classification des algorithmes

Malgré que les algorithmes présentés utilisent différentes techniques et adoptent différents points de vue, ils peuvent être classés selon trois aspects

algorithmiques: La stratégie de calcul du support, la direction de recherche, la stratégie de recherche.

II.2.8.1. Stratégie de Calcul du Support

Le calcul du support des itemsets candidats peut se faire selon deux approches:

- Calcul horizontal (comptage direct d'occurrences): Détermine la valeur du support d'un itemset candidat en lui associant un compteur initialisé à 0 et en parcourant la base de transactions, transaction par transaction. A chaque fois que l'on trouve l'itemset recherché dans la transaction courante, le compteur est incrémenté.
- Intersection verticale: une approche pour déterminer les supports des candidats est l'intersection verticale ou intersection des TidLists. Cette méthode est utilisée lorsque la base de transactions est représentée verticalement.

Exemple 2

Soit la base de transactions D suivante:

Tids	Transactions
1	a,b
2	a,c
3	c,d
4	b,c,d
5	a,b,c,d

Table 9. *Exemple de Base de Transactions*

Comme illustration, nous allons calculer le support du 2-itemset {a,b} en utilisant les deux approches:

Calcul horizontal

On parcourt D, et, à chaque fois que l'itemset {a,b} est inclus dans une transaction, on incrémente le compteur. On obtiendra donc support({a,b}) = 2.

Intersection verticale

$\{a,b\} = \{a\} \cup \{b\}$ TidList(a) = {1,2,5} et TidList(b) = {1,4,5}

TidList ({a,b}) = TidList(a) $\cap$ TidList(b) TidList ({a,b}) = {1,5}

Support({a,b}) = |TidList({a,b})| = 2

II.2.8.2. Direction de Recherche

La direction de recherche réfère à la direction dans laquelle l'espace de recherche est traversé. La plupart des algorithmes d'extraction des itemsets fréquents, surtout ceux qui étendent Apriori, adoptent une traversée ascendante de l'espace de recherche, commençant par les 1-itemsets fréquents jusqu'à parvenir aux itemsets fréquents les plus larges (dans le sens de la taille).

L'autre type de traversée est la traversée descendante de l'espace de recherche, commençant par les itemsets fréquents les plus larges jusqu'à l'obtention des 1-itemsets fréquents. Cette stratégie est habituellement utilisée pour l'extraction des itemsets larges maximaux.

II.2.8.3 Stratégie de recherche

Tandis que la direction de recherche guide la manière selon laquelle l'espace de recherche sera utilisé, la stratégie de recherche se réfère à l'ordre dans lequel sont visités les itemsets.

La stratégie de recherche en largeur d'abord (BFS: Breadth First Strategy) opère niveau par niveau. Elle visite tous les itemsets de niveau (k-1) avant de visiter ceux de niveau k. La plupart des algorithmes qui étendent Apriori utilisent une recherche en largeur d'abord (BFS) parce qu'elle facilite l'élagage des candidats. Cette stratégie nécessite plus de mémoire pour garder les sous ensembles fréquents des candidats élagués.

La stratégie de recherche en profondeur d'abord (DFS: Depth First Strategy), quant à elle, visite de façon récursive les descendants d'un itemset.

La *Figure 9* visualise l'espace de recherche pour I = {a, b, c, d} en largeur.

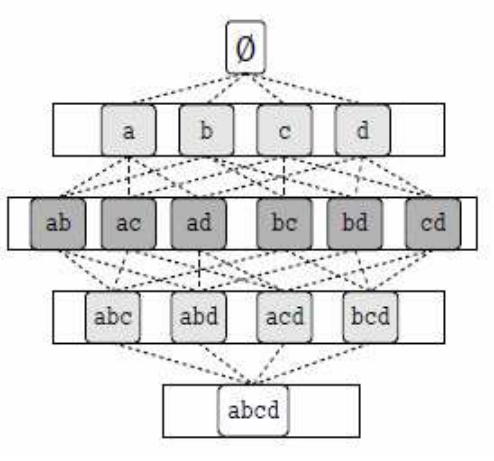


Figure 9. *Stratégie BFS*

La *Figure 10* visualise l'espace de recherche en profondeur.

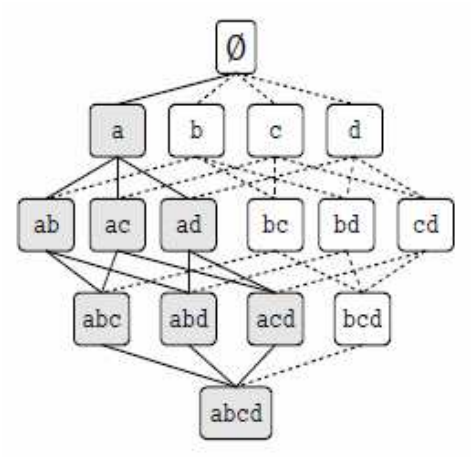


Figure 10. *Stratégie DFS*

La *Table 9* montre la classification des algorithmes d'extraction des itemsets fréquents, précédemment présentés, selon les trois critères ci-dessus.

Algorithme	Stratégie de Calcul du Support	Stratégie de Recherche	Direction de Recherche
Apriori	Calcul Horizontal	BFS	Ascendante
DIC	Calcul Horizontal	BFS	Ascendante
Partition	Intersection Verticale	BFS	Ascendante
Bitmap	Intersection Verticale	DFS	Descendante
FP-Growth	Calcul Horizontal	DFS	Descendante
DualFilter	Intersection Verticale	DFS	Descendante
ITL-Mine	Intersection Verticale	DFS	Descendante
PatriciaMine	Calcul Horizontal	DFS	Descendante
TM	Intersection Verticale	DFS	Descendante

Table 10. *Classification des Algorithmes*

II.2.9. Conclusion

Dans cette partie, nous avons présenté, d'une manière non exhaustive, les principaux algorithmes d'extraction d'itemsets fréquents ainsi qu'une classification de ces algorithmes, selon trois approches algorithmiques: la stratégie de calcul des supports, la direction de recherche et la stratégie de recherche.

Nous avons pu constater que ces algorithmes diffèrent également par les structures de données qu'ils utilisent, que ce soit pour la représentation des itemsets candidats ou pour la représentation de la base de transactions.

Dans la section suivante, nous allons nous intéresser aux principales structures de données utilisées par les algorithmes d'extraction des itemsets fréquents.

II.3. Structures de données pour l'extraction des itemsets fréquents

II.3.1. Introduction

Nous nous sommes intéressés au niveau du chapitre précédent aux algorithmes d’extraction des itemsets fréquents. Nous avons survolé la littérature dédiée à l’extraction des itemsets fréquents présentant quelques uns des algorithmes les plus connus. Ensuite, nous avons présenté une classification de ces algorithmes selon différents critères à savoir: la stratégie de calcul des supports, la direction de recherche et enfin la stratégie de recherche. Dans ce chapitre, nous allons nous intéresser à un autre aspect qui est aussi déterminant pour les algorithmes d’extraction des itemsets fréquents à savoir le choix de la structure de données à utiliser.

Le problème du choix de la structure de données est devenu un problème d’actualité. Le besoin de disposer de structures compactes a motivé de nombreux chercheurs. De nouvelles structures ont été proposées dans le but d’optimiser l’utilisation de la mémoire, de réduire les coûts des Entrées/Sorties et de rendre les traitements plus rapides.

Dans ce chapitre, nous présenterons en premier lieu une classification en deux grandes classes de structures de données utilisées pour l’extraction des itemsets fréquents: structures de données pour la représentation de la base de transactions et structures de données pour le stockage et la génération des itemsets candidats.

II.3.2. Représentation de la Base des Transactions

Pour calculer les supports d'une collection d'itemsets, on a besoin d'accéder à la base de transactions. Comme les bases de transactions sont généralement très larges, une solution, qui permet d'éviter les parcours répétitifs et coûteux de ces bases, peut être de les représenter par des structures compactes afin d'optimiser l'utilisation de la mémoire centrale.

Dans cette section, nous présentons une classification des schémas de représentation des bases de transactions puis nous allons donner des exemples de structures de données.

II.3.2.1. Classification des schémas de représentation

Les chercheurs ont proposé divers schémas de représentation des données pour l'extraction d'itemsets fréquents. Ces schémas peuvent être classés en trois catégories:

1. **Schéma horizontal**: Ce schéma est généralement adopté par les algorithmes basés sur Apriori. Dans ce schéma, la base est représentée comme étant une liste de transactions. Chaque transaction possède un identifiant: Tid, suivi de l'ensemble des items qui la constituent. Pour calculer le support d'un itemset X en utilisant le schéma horizontal, nous devons parcourir la base de transactions (ou la partition) en entier et tester pour chaque transaction T si $X \subset T$. De ce fait, ce format impose des coûts considérables pour le calcul de supports engendrés par les opérations d'entrées/sorties. Ce schéma est plus approprié avec l'exploration ascendante de l'espace de recherche des candidats.
2. **Schéma Vertical** (ou inversé): Dans le schéma vertical, une base de transactions est représentée comme étant un ensemble d'items et chaque item est suivi de sa TidList qui consiste en la liste des identifiants (Tids) de toutes les transactions le contenant. Le coût de calcul des supports des itemsets est beaucoup moins important pour le schéma vertical pour les raisons suivantes:
    - Si les TidLists sont triées dans un ordre croissant alors le support d'un k-itemset candidat est calculé en faisant simplement

l'intersection des TidLists de deux (k-1) sous ensembles de cet itemset.

- Les TidLists contiennent toutes les informations pertinentes concernant un itemset; ce qui permet d'éviter de parcourir la base de transactions `a chaque fois pour calculer le support d'un itemset.
- Plus la taille de l'itemset est grande, plus sa TidList est courte.

3. **Schéma Hybride**: En fait, la Combinaison des deux schémas n'est pas seulement possible mais peut aussi s'avérer très intéressante puisqu'elle profite des avantages des deux schémas. Conceptuellement, une base de transactions peut être représentée par une matrice binaire où chaque ligne représente une transaction et chaque colonne représente un item. La *Table 10* illustre les schémas vertical et horizontal de représentation des données.

Tid	1	2	3	4	5
100	1	0	0	1	0
200	1	0	1	1	1
300	0	1	1	1	1
400	1	1	0	0	1

Tid	1	2	3	4	5
100	1	0	0	1	0
200	1	0	1	1	1
300	0	1	1	1	1
400	1	1	0	0	1

Table 11. *Schémas Horizontal et Vertical de représentation des données*

II.3.2.2. Exemples de Structures de données

1. **Structure Bitmap**

Le Bitmap est un index binaire dans lequel chaque couple <Tid, IdItem> est représenté par un bit. Le support d'un itemset (i_1,i_2,...,i_k) se calcule en appliquant l'opérateur logique "T" à l'ensemble des colonnes associées dans l'index [22]. Gardarin et al ont proposé deux variantes du bitmap:

- **Bitmap Naïf**: A chaque itemset, on associe un vecteur de bit nommé Tidbit. Le bit i est à 1, dans Tidbit, si la transaction i contient l'itemset, 0 sinon.

  Dans un bitmap, chaque colonne correspond à un k-itemset donné. Chaque ligne correspond à une transaction. Puisque l'algorithme

manipule des Tidbits, le codage est important. Chaque 16 bits forment un groupe. Le nombre total de groupes est le nombre total de transactions divisé par 16.

Exemple 4

Soient deux items A et B dont les TidLists respectives {3,5,7,12,25,30} et {5,11,25}. En divisant par 16, on obtiendra que les Tids {3,5,7,12} appartiennent au groupe 0 et que les Tids {25,30} appartiennent au groupe 1 (*Figure 11*).

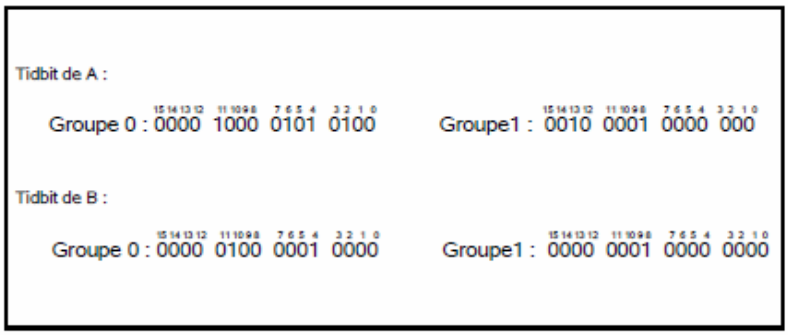


Figure 11. *Tidbits des items A et B*

En Codant en Hexadécimal, nous obtenons:

A.tidbit = {Groupe 0: 0854, Groupe 1: 2100},

B.tidbit = {Groupe 0: 0010, Groupe 1: 0100}.

L'intersection de ces deux Tidbits en hexadécimal donne:

AB.Tidbit = {Groupe 0 : 0010, Groupe 1 : 0100}

Ce qui signifie que la 5ème transaction (5 + 16*0) et la 25ème transaction (9 +16*1) contiennent les items A et B.

- **Bitmap hiérarchique**: Cette variante du bitmap est mieux adaptée aux index binaires creux. Pour résoudre le problème des index creux (contenant beaucoup de bits à 0) et éviter de faire des intersections inutiles, un schéma de compression hiérarchique est appliqué au bitmap. Ainsi, un deuxième index est construit à partir du premier bitmap, où chaque bit indique si un groupe du premier bitmap est vide (ne contenant que des zéros) ou non.

Exemple 5

Soit {3, 12, 36, 55, 84,123} la TidLists de l'item A.

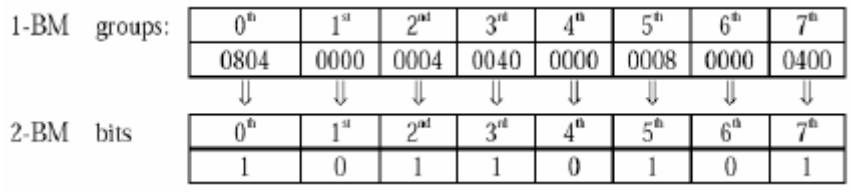


Figure 12. *Bitmap Hiérarchique*

Le Bitmap hiérarchique correspondant à l'item A est représenté par la *Figure 12*.

Pour illustrer les exemples des structures de données qui suivent, nous allons utiliser la base de transactions de la *Table 12* et une valeur de Minsup = 3.

Tid	Transaction
1	1, 2, 4, 5, 6, 7, 8, 9
2	2, 3, 5, 10
3	1, 2, 4, 6, 8, 10
4	1, 2, 3, 4, 6, 7, 10
5	2, 7, 8, 10
6	1, 2, 4, 6, 9

Table 12. *Exemple de Base de Transactions*

2. Structure FP-Tree

Le FP-Tree est une extension de la structure de données Trie qui appartient à la famille des arbres préfixés. Le FP-Tree combine les schémas vertical et horizontal de représentation des données. La proposition du FP-Tree a été un point de départ pour la proposition d'algorithmes d'extraction d'itemsets fréquents sans génération de candidats [18].

Le FP-Tree est une structure compacte constituée:

- D'un Arbre dont la racine porte la valeur "null" et, où chaque noeud, autre que la racine, contient trois informations: l'item représentant le noeud, sa fréquence et le noeud suivant dans l'arbre.

- D'un Index où est stockée la liste des items fréquents. A chaque item est associé un pointeur qui indique le premier noeud de l'arbre où apparaît cet item.

La construction du FP-Tree nécessite deux parcours de la base de transactions et se fait comme suit :

- Le premier parcours de la base permet de déterminer le nombre d'occurrences de chaque item, d'éliminer les items non fréquents et d'ordonner les items fréquents par ordre décroissant de support.
- Durant le deuxième parcours de la base de transactions, les items de chaque transaction sont triés selon l'ordre obtenu durant la première phase (*Table 12*).

Le processus de construction de l'arbre débute par la création d'un noeud racine. Ensuite, on ajoute une branche à l'arbre pour une transaction traitée, tout en exploitant le fait que les transactions ayant un même préfixe partageront le même début de branche de l'arbre.

L'avantage des FP-Tree est plus visible lors de l'utilisation des bases denses et réside dans le niveau de compression atteint en fusionnant les préfixes communs. Mais pour les bases éparses, cette structure perd de son intérêt puisque la taille de l'arbre peut être de l'ordre de la taille de la base et parfois même plus importante, dans le cas de bases fortement éparses.

Tid	Transaction	Items triés sur le support
1	1, 2, 4, 5, 6, 7, 8, 9	2, 1, 4, 6, 7, 8
2	2, 3, 5, 10	2, 10
3	1, 2, 4, 6, 8, 10	2, 1, 4, 6, 10, 8
4	1, 2, 3, 4, 6, 7, 10	2, 1, 4, 6, 10, 7
5	2, 7, 8, 10	2, 10, 7, 8
6	1, 2, 4, 6, 9	2, 1, 4, 6

Table 13. *Base de Transactions triée par rapport au support*

La *Figure 13* illustre le FP-Tree associé à la base de transactions de la *Table 12*.

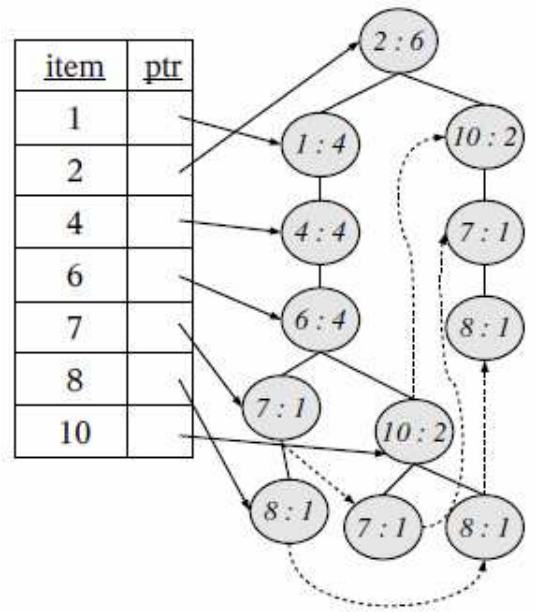


Figure 13. *Exemple de FP-Tree*

II.3.3. Stockage des Candidats

Les algorithmes d'extraction des itemsets fréquents avec génération des itemsets candidats utilisent des structures plus ou moins performantes, selon la manière d'organiser le stockage de ces itemsets candidats. Nous présentons dans cette section quelques unes de ces structures.

III.3.3.1. Arbre de hachage

L'arbre de hachage permet de réduire la taille des candidats des k-itemsets à examiner. Elle consiste à stocker des candidats de niveau k dans un arbre N-aire de hachage.

Les ensembles C_k sont placés dans l'arbre de hachage (hash tree).Un nœud de cet arbre contient:

- soit une liste d'itemsets si le nœud est une feuille,
- soit une table de hachage si le nœud est interne.

Dans un nœud interne, chaque case de la table de hachage pointe sur un autre nœud. On définit la racine de l'arbre à la profondeur 1.Un nœud, de profondeur d, pointe vers des nœuds de profondeur d+1.

Lorsqu'on ajoute un itemset I, on part de la racine pour parcourir l'arbre et atteindre une feuille. Arrivé à un nœud interne de profondeur d, on décide

quelle branche suivre en appliquant une fonction de hachage au $d^{ième}$ objet de l'ensemble I.

Initialement, tous les nœuds sont des feuilles. Quand le nombre d'ensembles placés dans un nœud feuille dépasse une certaine valeur, le nœud est converti en un nœud interne.

En démarrant à la racine, La recherche des candidats contenus dans une transaction T se fait à partir de la racine, de la manière suivante:

- **Au niveau d'une feuille**: en recherchant quels sont les itemsets dans la feuille contenus dans T et on incrémente leurs supports.
- **Au niveau d'un nœud interne**: Ce niveau est atteint en hachant l'itemset I. L'algorithme utilise la fonction de hachage pour « hacher » chaque itemset classé après I dans T. Il applique récursivement la fonction au nœud indiqué par la case correspondante.
- **Au niveau de la racine**: en hachant chaque itemset de T.

La *Table 14* illustre un exemple d'une base de transactions.

Tid	Transaction
1	1, 2
2	1, 3, 5, 6
3	1, 2, 3
4	2, 4
5	2, 3, 6
6	5, 6

Table 14. *Exemple de base de transactions*

Considérons la fonction de hachage suivante:

- Si l'item est égal à 1 ou 4, l'insérer dans la branche gauche
- Si l'item est égal à 2 ou 5, l'insérer dans la branche centrale
- Si l'item est égal à 3 ou 6, l'insérer dans la branche droite

Prenons Minsup = 2

La fonction de hachage choisie nous donnera un arbre de hachage à trois branches.

La *Figure 14* représente l'arbre contenant les 1-itemsets candidats.

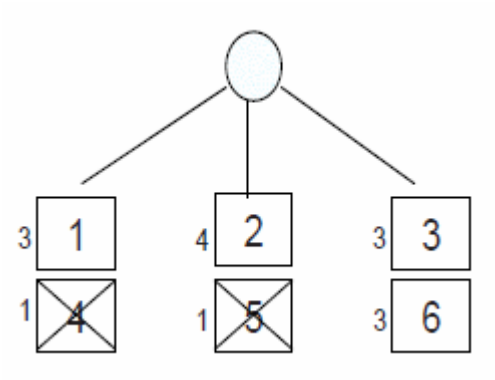


Figure 14. *Arbre de Hachage des 1-Itemsets Candidats et leurs supports*

Un parcours de la base va nous permettre de supprimer les candidats non fréquents et construire l'arbre de hachage des 2-Itemsets candidats (*Figure 15*).

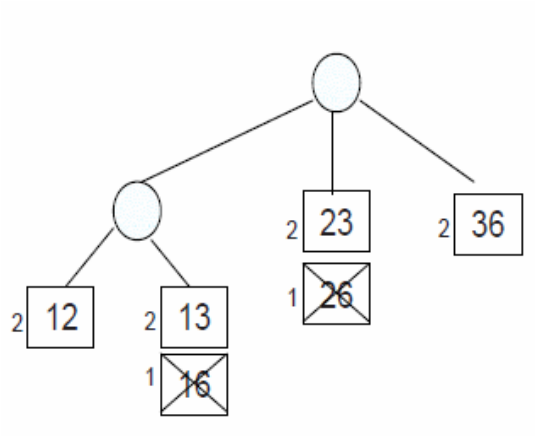


Figure 15. *Arbre de Hachage des 2-Itemsets Candidats et leurs supports*

Le même processus va nous donner l'arbre de hachage des 3-Itemsets candidats de la *Figure 16*.

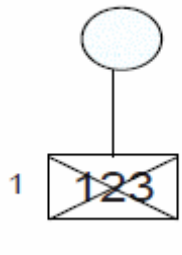


Figure 16. *Arbre de Hachage des 3-Itemsets Candidats*

II.3.3.2. Trie

La structure de données Trie a été introduite à l'origine par De la Briandais et Fredkin pour stocker et récupérer efficacement les mots d'un dictionnaire et a été utilisée pour la première fois pour les algorithmes d'extraction des itemsets fréquents par Mueller. Le Trie est maintenant largement utilisé pour le stockage des candidats.

Le *"Trie"* est un arbre préfixé qui suppose qu'un certain ordre parmi les items ait été défini. La racine est à la profondeur 0. Un nœud à la profondeur d pointe vers des nœuds de la profondeur d+1.

A chaque k-itemset est associé un nœud où est stocké le dernier item de l'itemset qu'il représente et son support.

On prend les transactions une par une et on parcourt récursivement le *"Trie"* jusqu'à atteindre les feuilles contenues dans la transaction traitée. Les compteurs des supports associés à ces feuilles sont alors incrémentés.

La *Figure 17* représente un *Trie* qui stocke les itemsets A, C, F, AC, AF, EF, AEF, en utilisant l'ordre alphabétique.

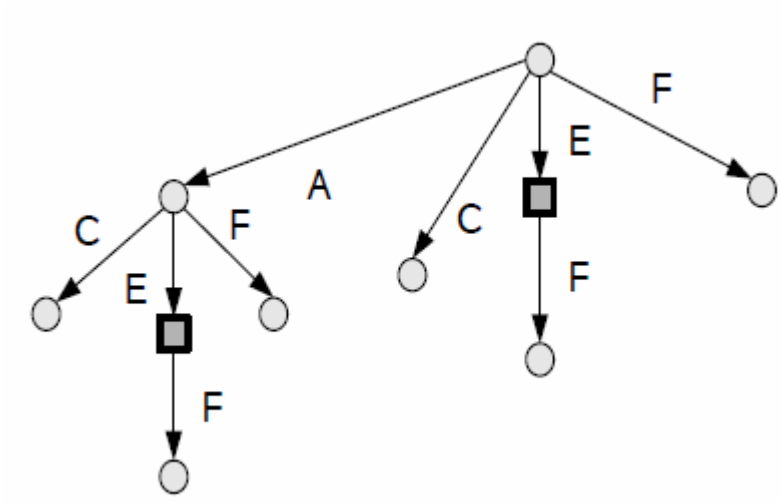


Figure 17. *Un exemple d'Arbre Trie*

II.3.3.3. Arbre Lexicographique

Cette structure a été utilisée dans divers travaux pour faciliter la génération des candidats. Chaque nœud dans l'arbre stocke une collection d'itemsets fréquents. La racine contient tous les 1-itemsets fréquents. Les itemsets à un niveau n sont les n-itemsets fréquents.

Chaque arc dans l'arbre est labellé par un item. Les items au niveau d'un nœud sont stockés comme des singletons sachant que l'itemset actuel contient tous les items rencontrés sur les arcs depuis la racine jusqu'à ce nœud.

L'arbre sera généré en profondeur d'abord. Les candidats générés en profondeur d'abord en utilisant l'arbre lexicographique préfixé sont ceux obtenus par la phase de jointure (sans élagage) de l'algorithme Apriori.

Exemple 7

Prenons un exemple simple d'une base de transactions où seuls les items 1, 2, 3, 4, 5 sont fréquents. Les candidats seront représentés dans l'arbre lexicographique illustré par la *Figure 18*.

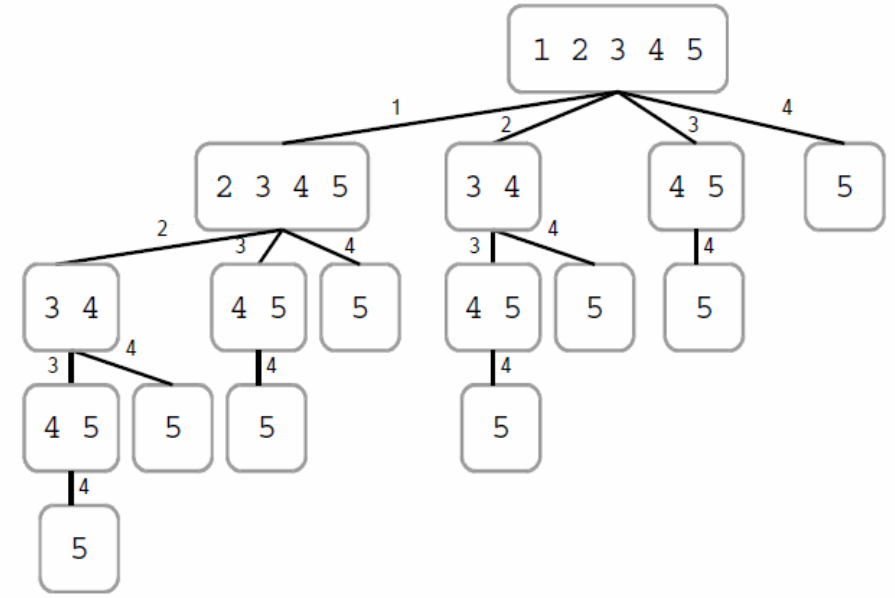


Figure 18. *Exemple d'Arbre Lexicographique*

II.3.4. Conclusion

Nous avons présenté dans ce chapitre les principales structures de données utilisées pour l'extraction des itemsets fréquents. Ces structures appartiennent à deux grandes classes: les structures de données pour la représentation de la base de transactions et les structures de données pour la génération et le stockage des itemsets candidats.

Nous avons constaté à travers notre étude de ces structures qu'elles présentent un certain nombre de faiblesses. Ainsi, pour remédier à ces problèmes, nous avons orienté nos recherches vers la proposition d'un nouveau modèle de

représentation compacte d'une base de transactions qui tend à minimiser les opérations d'accès à la base de transactions.

Dans le chapitre suivant, nous allons définir un nouveau modèle de représentation de la base de transactions, utilisant un arbre de signatures et/ou un fichier de signatures. Nous expliciterons ensuite les algorithmes d’extraction des itemsets fréquents basés sur ce modèle.

CHAPITRE III

UN MODELE DE REPRESENTATION DE LA BASE DE TRANSACTIONS

III. UN MODELE DE REPRESENTATION DE LA BASE DE TRANSACTIONS

III.1. Introduction

Le problème de la découverte des itemsets fréquents est un problème classique de fouille de données. L'espace de recherche des itemsets fréquents est exponentiel.

La majeure partie des algorithmes réalisés, du même type qu'Apriori, adopte une approche de génération et de test des itemsets candidats. Or cette phase de génération de candidats reste très coûteuse, plus particulièrement si le nombre d'itemsets est important et/ou la taille des itemsets est grande.

L'autre problème qui ressort est celui des structures de données utilisées par les algorithmes de découverte des itemsets fréquents. Le but est évidemment d'optimiser l'utilisation de la mémoire, de réduire les coûts des Entrées/Sorties et de rendre les traitements plus rapides. L'amélioration des performances des algorithmes de découverte des règles d'association passe nécessairement par l'optimisation de la phase d'extraction des itemsets fréquents, étant la phase la plus coûteuse. Le choix de la structure de données utilisée dans cette phase devient fondamental puisqu'il influe sur la performance de l'algorithme du point de vue complexité spatiale et temporelle.

Nous proposons comme contribution un nouveau modèle de représentation de la base de transactions. Ce modèle utilise une signature binaire associée à chaque transaction, par application d'une ou de plusieurs fonctions de hachage sur les items composant la transaction. La signature de la transaction est obtenue par combinaison des signatures des items (Ou logique ou Or-ing). L'ensemble des signatures obtenues est représenté sous forme d'arbre binaire ST-Tree (Signatures – Transactions – Tree). L'arbre de signatures ST-Tree jouera le rôle d'index et sera utilisé pour organiser l'accès aux signatures et donc aux transactions dans le processus de génération des itemsets fréquents. La principale originalité de cette approche réside dans le fait qu'elle profite des avantages et de la rapidité des opérations sur bits qu'offrent les signatures pour l'extraction des itemsets fréquents.

Plusieurs variantes de ce modèle ont été définies et utilisées. La première consiste à stocker, dans un fichier, les signatures de transactions et, ensuite, de représenter le fichier de signatures obtenu sous forme d'arbre de signatures. La seconde remplace les pointeurs vers les signatures par les signatures et ajoute également dans les feuilles de l'arbre, le nombre de transactions générant la signature ainsi que les Tids. Donc, cette

variante du modèle n'utilise plus de fichier de signatures mais uniquement l'arbre de signatures.

Dans ce chapitre, nous introduisons la notion de signature, de fichier de signatures et de quelques techniques d'extraction de signatures. Nous présenterons ensuite différentes techniques de représentation physique des signatures. La plus grande partie sera consacrée exclusivement à la présentation détaillée des deux variantes de notre modèle et des résultats obtenus pour chacun, lors du processus de génération des itemsets fréquents et des règles d'association.

III.2. Un Arbre de Signatures pour l'Extraction des Itemsets Fréquents

Dans cette partie, nous allons introduire la notion de signature, de fichier de signatures, d'arbre de signatures et son utilisation dans la représentation des transactions. Deux variantes de modèles de représentation seront présentées:

- Un premier modèle, le modèle de représentation TSF, sera constitué d'un arbre de signatures et d'un fichier de signatures.
- Une second modèle, le modèle ST-Tree, qui ne sera constitué que d'un arbre de signatures.

Nous présenterons également, pour chacun des modèles proposées, l'ensemble des algorithmes permettant une extraction des itemsets fréquents et la génération des règles d'association, à partir de ces itemsets.

III.2.1. Signatures et fichier de signatures

Une signature est une chaîne de bits générée par application d'une fonction de hachage sur une valeur donnée. Comparée à d'autres structures d'index, le fichier de signatures est plus efficace dans le traitement des nouvelles insertions et des requêtes sur les parties de mots. D'autres avantages incluent une simple implémentation et une capacité à supporter les fichiers croissants. Mais, son utilisation génère une perte d'information qui peut être minimisée par une gestion rigoureuse de la méthode d'extraction de signature.

Plusieurs techniques d'extraction de signature existent telles que "Word Signature (WS)", "Superimposed Coding (SC)", "Multilevel Superimposed Coding (MSC)", "Run Length Encoding (RL)", Bit-Block Compression (BC)", "Variable Bit-block Compression (VBC)" [1, 2, 12, 15, 16, 21, 29, 32, 62, 67, 70, 71, 74].

Le processus de codage fixe un nombre m de 1 dans l'intervalle [1, F] où F est la taille de la signature. Le résultat, un vecteur binaire avec un nombre m de bits à 1 et (F- m) bits à 0, est appelé *"Word Signature"*. La signature d'un bloc de texte ou un objet est obtenue par combinaison logique (Superimposed Coding, OR-ing) des signatures de chacun des mots du bloc ou des constituants de l'objet. L'ensemble des signatures constituent le fichier de signatures (Signature File).

La *Table 14* illustre un exemple d'extraction de la signature d'un bloc de texte ("*Information Retrieval"*). Les signatures des mots du bloc sont combinées par superposition ("OU logique") pour donner la signature du bloc.

Information Retrieval	0010 0100 0100 0001
Block Signature	0110 0101

Table 15. *Exemple de Génération d'une Signature de Bloc*

Pour rechercher le mot "*information*", par exemple, nous utilisons la même méthode que celle qui a permis de construire les signatures des blocs. L'étape 1 consiste à calculer la signature du mot "*information*", ensuite de sélectionner les signatures de blocs qui contiennent la signature générée pour finalement vérifier si les blocs sélectionnés contiennent effectivement le mot "*information*".

Cas 1: Résultat correspondant à la requête (Matching Query)

Mot = "*Information*" Signature = 0010 0100

Signature du bloc sélectionné 0110 0101

Le bloc "*Information Retrieval*" contient effectivement le mot "*Information*".

Le bloc est appelé "*Actual Drop*" ou "*True Drop*"

Cas 2: Résultat non correspondant à la requête (False Match Query)

Mot = "*Coding*" Signature = 0010 0100

Signature du bloc sélectionné = 0110 0101

Le bloc "*Information Retrieval*" ne contient pas le mot "*Coding*"

Le bloc est appelé "*False Drop*".

III.2.1.1. Représentations physique du fichier de signatures

Cette partie sera consacrée aux différentes techniques de représentation des fichiers de signatures.

a) Sequential Signature File (SSF)

Dans un fichier SSF, les signatures sont stockées séquentiellement (*Table 15*). La méthode s'implémente aisément, requiert peu d'espace mémoire et un coût de mise à jour bas. Le seul inconvénient est le temps important exigé lors du parcourt du fichier de signatures. Par conséquent, Elle est généralement lente [3].

OIDs	Fichier de Signatures
O1	10101001
O2	01100011
O3	00101101
O4	11101000
O5	00111001
O6	11100010
O7	01010011
O8	01010110

Table 16. *Exemple de SSF*

b) Bit-Sliced Signature File (BSSF)

BSSF stocke les signatures par colonne comme illustré dans la *Table 16*. Ce qui donne F fichiers de bits, un pour chaque bit, pour l'ensemble des signatures.

OIDs	8 Fichiers de Bits							
O1	1	0	1	0	1	0	0	1
O2	0	1	1	0	0	0	1	1
O3	0	0	1	0	1	1	0	1
O4	1	1	1	0	1	0	0	0
O5	0	0	1	1	1	0	0	1
O6	1	1	1	0	0	0	1	0
O7	0	1	0	1	0	0	1	1
O8	0	1	0	1	0	1	1	0

Table 17. *Exemple de BSSF*

Dans le processus de recherche, une partie seulement des F fichiers est parcourue. Ce qui permet d'avoir un coût plus bas que SSF. Seulement, le coût de mise à jour est plus important. En effet, l'insertion d'une nouvelle signature nécessite F accès disque, un pour chaque fichier [3].

c) Compressed Bit-Sliced Signature File (CBSSF)

En choisissant une fonction de hachage propre, nous fixons également le nombre de "1" par signature. Plus la taille de la signature est grande, plus la probabilité d'avoir un false drop est faible. Ce qui donne nécessairement une matrice éparse, facile à compresser. Un moyen simple est de remplacer chaque "1" par son adresse physique. Sur la *Figure 19* ci-dessous, la table de hachage contient une liste de pointeurs qui pointent vers les têtes de listes chaînées [3].

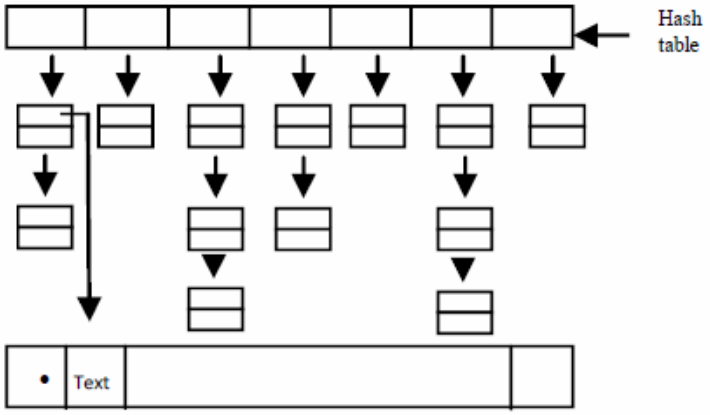


Figure 19. *Exemple de CBSSF*

d) S-Tree

S-Tree est un B+-Arbre dont les feuilles contiennent un ensemble de signatures et leurs OIDs. Les nœuds internes sont obtenus en combinant par "ou logique" les nœuds de niveau inférieur comme illustré sur la *Figure 20* [3].

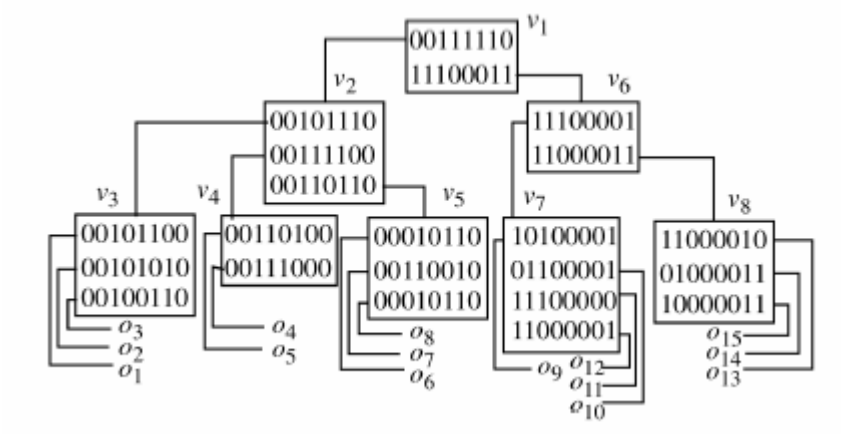


Figure 20. *Exemple de S-Tree*

Par exemple, pour retrouver la signature Sq = 11000000, la recherche se fait de la racine vers les feuilles (Top – Down). A partir du nœud v1, après comparaison, on va vers le nœud v6, ensuite vers v7 et v8 pour terminer en sélectionnant les signatures o11, o12 de v7 et o13 de v8. L'avantage se situe dans le fait que la sélection des signatures n'a pas nécessité le parcours de tout le fichier de signatures.

e) Multilevel Signature File (MSF)

La structure MSF est similaire à la structure S-Tree. Cependant, une signature d'un nœud interne est obtenue par superposition logique à partir de tous les blocs de texte indexés par le sous-arbre pour lequel la signature est la racine. La *Figure 21* représente la représentation MSF correspondante [3].

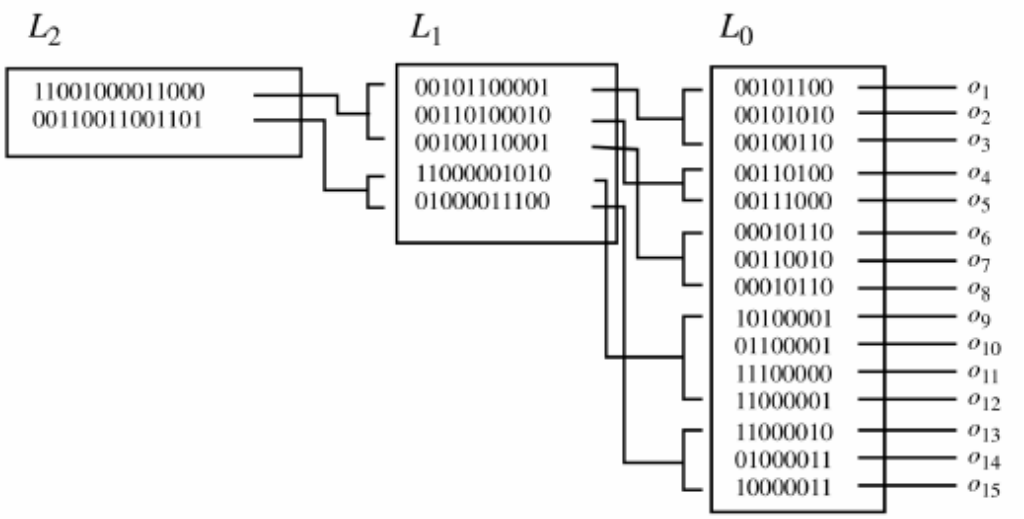


Figure 21. *Exemple de MSF*

f) Signature Graph

Le fichier de signatures est organisé comme un Trie. Cependant, le chemin parcouru dans le graph pour trouver une signature qui correspond à une signature de requête donnée, n'est pas une suite de bits, comme dans Trie, mais un identificateur de signature.

Bien que les signatures soient représentées de manière compacte, la longueur du chemin de recherche n'est pas la même pour toutes les requêtes. Autrement dit, le graphe n'est pas balancé. La *Figure 22* représente un fichier de signatures et le graphe correspondant [3, 16].

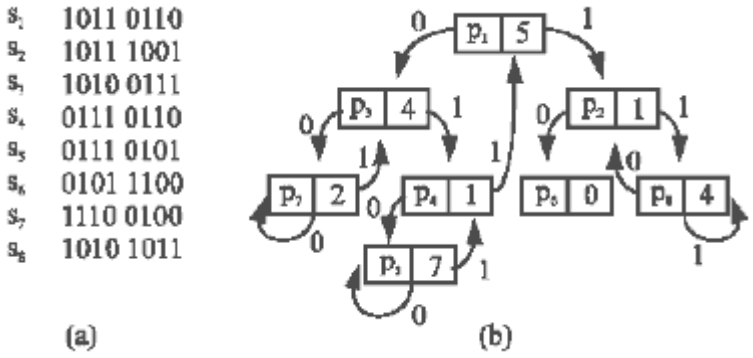


Figure 22. *Fichier de Signatures et Graphe de Signatures*

g) Signature Tree

L'arbre de signatures de la *Figure 23* est un arbre binaire tel que les nœuds internes représentent une position de bit à contrôler. Selon la valeur du bit rencontré, les sous-arbres gauche et droit correspondent respectivement à la valeur binaire "0" ou "1". Chaque signature est identifiée à partir de la racine par test des positions de bits données par les nœuds internes [10, 11, 73].

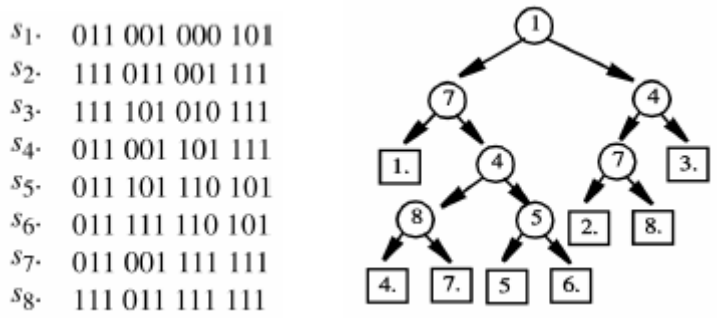


Figure 23. *Fichier de Signatures et Arbre de Signatures*

III.2.2. Variante 1: La structure *TSF* (Transactions – Signatures – File)

La structure *TSF* est une structure compacte qui peut être considérée comme un index associé à la base de transactions. L'idée réside dans l'utilisation combinée d'une structure à deux niveaux: un fichier de signatures et un arbre de signatures. Le fichier de signatures représente la base de transactions, une signature par transaction, et l'arbre de signatures représente un index primaire construit pour accélérer la recherche d'une signature et permettre l'accès au fichier de signatures.

Toutes les signatures du fichier sont différentes les unes des autres i.e. Si $i \neq j$ alors $s_i \neq s_j$. L'arbre de signatures possède autant de feuilles que de signatures. En fait, la génération de signatures s'accompagne de perte d'informations. Ainsi, deux

transactions différentes peuvent générer la même signature (*phénomène de collision*). Le processus de recherche, basé sur les signatures, peut sélectionner des transactions non réellement référencées (*False drop*). Pour résoudre ce problème, nous avons adapté la structure *TSF* pour prendre en charge la présence de signatures identiques dans le fichier de signatures.

Un arbre de signatures *TSF*, représentant un ensemble de signatures $S = s_1,\ldots, s_n$ où $s_i \neq s_j$ pour tout $i \neq j$ et $|s_k| = m$ pour $1 \leq k \leq n$, est un arbre binaire tel que:

- Pour chaque nœud interne de *TSF*, le fils gauche correspond à la valeur *0* et le fils droit à la valeur *1*.
- *TSF* a *N* feuilles. Ces feuilles sont des pointeurs vers les positions des différentes signatures. Pour chaque nœud feuille *nf*, *p(nf)* pointe vers la signature correspondante dans le fichier de signatures.
- A chaque nœud interne *v* est associé un nombre, noté *position(v)*, qui correspond à la position du bit à contrôler dans la signature lors du processus de recherche d'une signature donnée dans *TSF*.

III.2.2.1. Processus de construction de *TSF*

a) Principe de construction de *TSF*

La première étape consiste à générer la signature de chacune des transactions. La transaction étant composée d'un ensemble d'items, il faudra générer la signature de chacun des items et, par composition, obtenir la signature de la transaction.

La deuxième étape construit successivement l'arbre *TSF*. Initialement, l'arbre *TSF* ne contient qu'un nœud initial (la racine qui est un nœud feuille) contenant un pointeur vers la signature de la première transaction. Lors de l'ajout d'une nouvelle signature s, le parcours de l'arbre commence à partir de la racine.

Soit *v* un nœud interne tel que *position(v) = i*. Ce qui signifie que c'est le $i^{ème}$ bit de *s* qui sera contrôlé lors du passage par ce nœud. Si *s[i] = 0*, c'est le fils gauche qui est exploré sinon, c'est le fils droit.

Si *v* est une feuille, on compare la signature *s'* pointée par *v'* avec la signature *s*. Si *s* est égale à *s'*, on l'ajoute dans le fichier signature en créant

un lien avec *s'*. Sinon, s est une nouvelle signature. Soit *k+1* la position du 1er bit sur lequel *s* et *s'* diffère i.e. les bits de *1* à *k* des deux signatures sont identiques. Un nouveau nœud *u* interne est ajouté à l'arbre avec *position(u) = k+1*. Selon la valeur du bit de position *k+1* de *s*, *v* sera le fils gauche de *u* et *p(s)* le fils droit ou inversement. La *Table 17* donne le pseudo-code de l'algorithme de construction de l'arbre *TSF*.

```
Algorithme Construct_TSF;
    Entrée: Fichier de signatures
    Sortie: Arbre de signatures TSF
Début
    Générer un noeud racine r tel que position(r)=1;
    /* r correspond à la première signature du fichier de signatures */
    Pour j = 2 à N Faire
        Insert (sj);
    FinPour
Fin
```

Table 18. *Pseudo-code de Construction de TSF*

Pour chacune des signatures du fichier de signatures, *Construct_TSF* fait appel à une procédure *Insert (s_j)* qui insère la signature s_j dans l'arbre *TSF* (*Table 18*).

```
Procédure Insérer(s)
    Entrée: r racine de TSF
            s signature à insérer
    Sortie: r racine de l'arbre
Début
  Empiler (Pile, r);
  Tantque Pile non vide Faire
     v ← Dépiler (Pile);
     Si v est un nœud interne Alors
       j ← position (v)
       Si s [j] = 1 Alors
          Empiler (Pile, fils_droit)
       Sinon
          Empiler (Pile, fils_gauche)
       Finsi
     Sinon  /* v: nœud feuille = Pointeur vers la signature s' */
        Comparer s et s';
        /* On suppose que les k premiers bits de s et s' sont identiques
           Donc s diffère de s' au (k+1) ième bit */
        Générer un nouveau nœud u avec position(u) = k+1;
        Si s [k+1] = 1 Alors
           s sera le fils droit de u et v son fils gauche;
        Sinon
```

```
                s sera le fils droit de u et v son fils droit;
            Finsi
        Finsi
    FinTantque
Fin
```

Table 19. *Pseudo-Code d'insertion d'une signature dans TSF*

La *Table 21* illustre un exemple de construction de l'arbre de signatures à partir du fichier de signatures de la *Table 20*.

Signatures
S1 0100 1101
S2 1010 1100
S3 1101 1001
S4 0101 1011
S5 1001 0010

Table 20. *Fichier de Signatures*

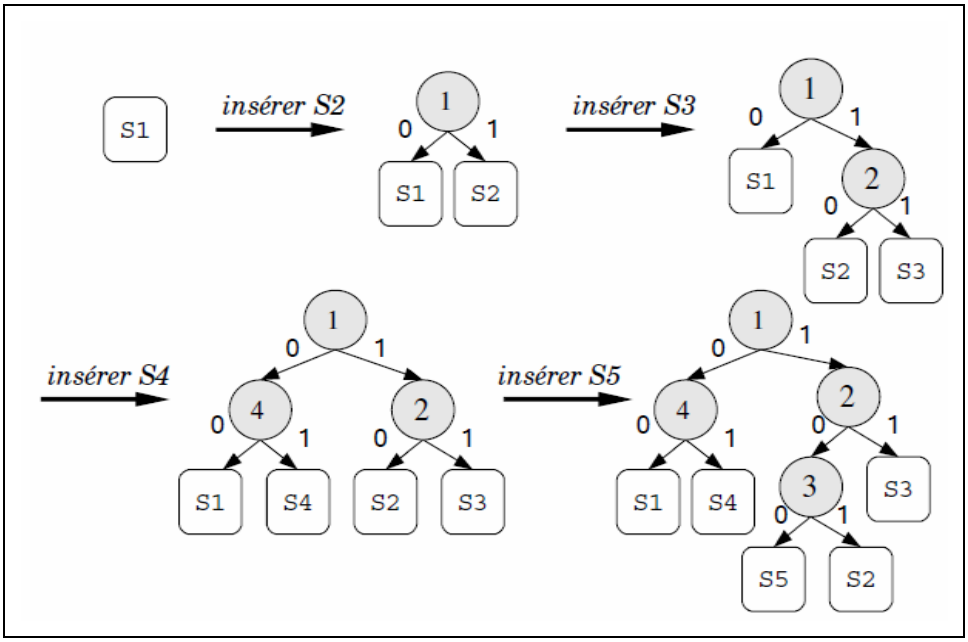


Table 21. *Etapes de Construction de l'arbre TSF*

Proposition 1: L'arbre de signatures *TSF*, associé à un fichier de signatures $S = S_1, S_2, \ldots, S_n$, contient k feuilles, $k \leq n$. Ces feuilles sont utilisées comme pointeurs vers les différentes positions des premières occurrences des signatures dans *S*. Chacune de ces signatures pointe vers la prochaine signature qui lui est identique (si elle existe).

La *Table 22* donne un exemple de base de transactions avec collision de signatures.

Tid	Transaction
1	1, 2, 4, 6, 8, 10
2	1, 2, 6, 10
3	3, 4, 10
4	1, 2, 4, 5, 8
5	1, 3, 4, 6, 10
6	2, 3, 4

Table 22. *Exemple de base de transactions*

La *Figure 24* représente la structure *TSF* relative à la T*able 22*. Nous constatons que les transactions *3* et *6* génèrent la même signature. Donc, S_6 n'apparaît pas dans l'arbre *TSF* mais un lien est créé dans le fichier de signatures, entre S_3 et S_6.

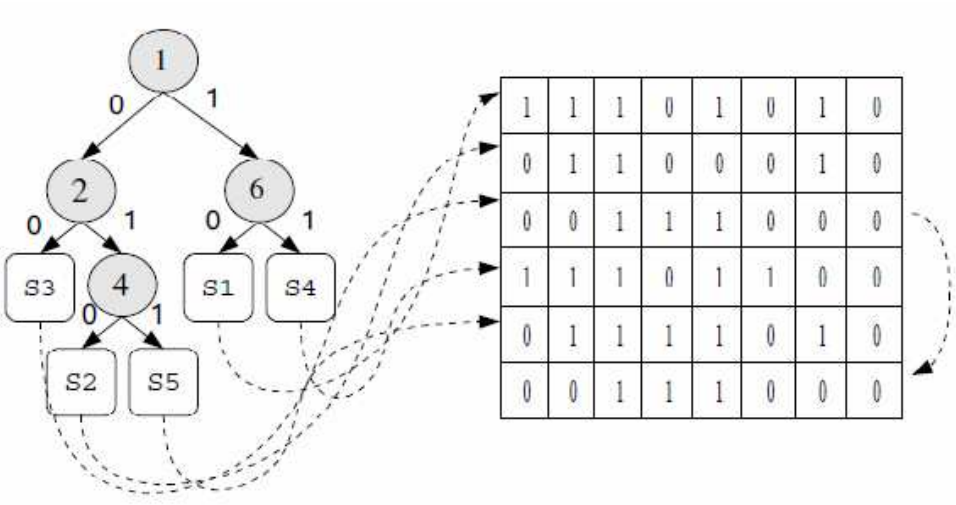


Figure 24. *Structure TSF relative à la table 22*

La rapidité et l'efficacité des opérations logiques sur les bits, ainsi que l'utilisation de la structure *TSF* comme index, permettent d'estimer rapidement le nombre de transactions contenant un itemset donné.

b) Calcul du Support d'un itemset

La génération des signatures n'étant pas bijective, elle s'accompagne donc d'une perte d'informations. La structure *TSF* fournit une représentation approximative de la base de transactions. Ce qui entraîne que les valeurs des supports d'itemsets sont des valeurs estimatives.

Une sur-estimation de la valeur du support est probable. Mais cette valeur ne peut jamais être sous estimée.

La *Table 22* représente le pseudo-code de l'algorithme *Compteur_Support* qui donne l'estimation du support d'un itemset.

```
Algorithme Compteur_Support; /* Estimation du support d'un itemset */
    Entrée: I = {I1, I2, . . ., In} Ensemble d'items.
            N: Nombre de signatures dans le fichier de signatures.
    Sortie: Sup_Estimé, le support estimé de I.
Début
    S_I = Calculer-signature(I);
    /* Vr: Vecteur binaire de taille N. Chaque position de Vr
    correspond à une transaction. Si la transaction contient l'itemset, le
    bit est mis à 1, sinon il reste à 0. */
    Vr = Initialiser(N); /* Met à 0 toutes les positions de Vr */
    S = Signature_Treesearch(S_I);
    Pour chaque (s ∈ S) faire
        Si (S_I ∧ s = S_I) Alors
            Mettre à 1 le bit de Vr correspondant à s;
            Si (s pointe vers une signature identique s_id) Alors
                Mettre à 1 le bit de Vr correspondant à s_id;
            FinSi
        FinSi
    FinPour
    Sup_Estimé = Support (Vr);
    Retourner (Sup_Estimé);
Fin
```

Table 23. *Pseudo-Code de calcul du support estimé d'un itemset*

Le calcul du support estimé d'un itemset nécessite l'appel à plusieurs fonctions:

- *Calculer-signature(I)* : génère la signature S_I de l'itemset *I*.
- *Initialiser(N)*: Initialise le vecteur *Vr* à *0*.
- *Signature_Treesearch(S_I)*: retourne l'ensemble *S* des signatures qui contiennent *I*, en parcourant *TSF*.
- *Support(Vr)* : retourne le nombre de bits égal à *1* dans *Vr*.

La *Figure 25* donne un exemple de l'estimation du support de l'itemset *I = {4, 8}* en utilisant l'algorithme *Compteur_Support* et la structure *TSF*.

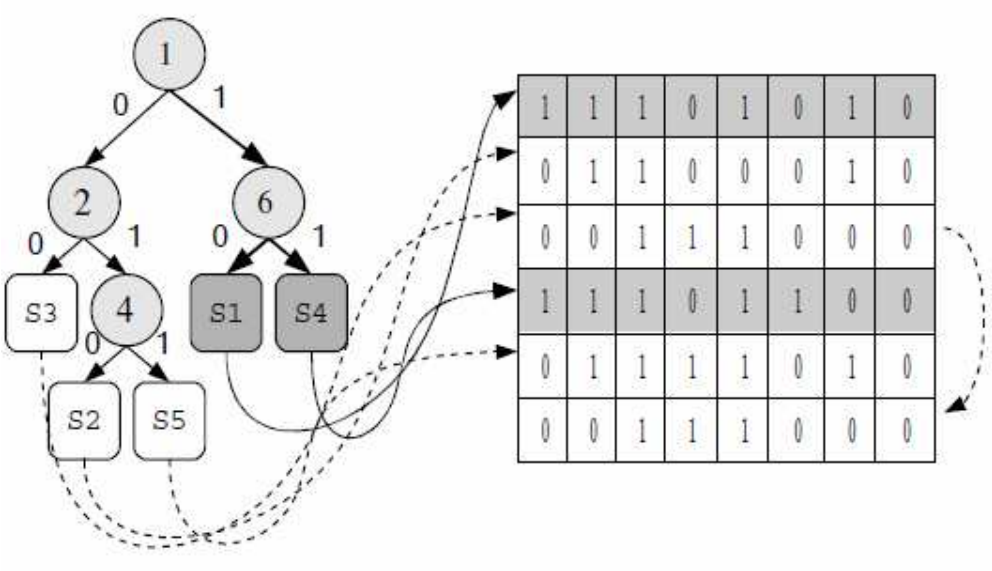


Figure 25. *Exemple de l'estimation du support de l'itemset {4, 8}*

Après l'étape de recherche dans *TSF*, seules les deux signatures S_1 et S_4 vérifient la proposition $S_I \wedge S = S_I$. Donc *Vr = (1, 0, 0, 1, 0, 0)* et le support estimé est égale à *2*.

c) Mécanisme de recherche dans l'arbre de signatures TSF

La procédure de recherche, ayant en entrée la signature S_I d'un itemset *I*, consiste à analyser S_I en parcourant l'arbre de signatures *TSF*. Dans le cas d'un nœud interne, si la valeur du bit associée à la position à contrôler est égale à 1, seul le chemin droit est exploré sinon les deux chemins seront explorés. En fait, lors de la correspondance, pour une position de bit donnée i dans S_I, si le bit est égal à 1, le bit correspondant dans S_I doit être à 1, sinon, le bit correspondant peut avoir soit la valeur 1 soit la valeur 0. Ce qui reflète le processus d'obtention d'une signature de transaction (Ou logique).

La *Table 23* représente le Pseudo-Code de l'algorithme de recherche de la signature d'un itemset en parcourant l'arbre *TSF*.

```
Algorithme Signature_Treesearch;
      Entrée: La signature S_I d'un itemset I */
      Sortie: Ensemble S des signatures susceptibles de contenir S_I */
Début
  Empiler (Pile, racine);
  Tantque pile non vide Faire
      v ← Dépiler (Pile);
```

```
        Si v est un nœud interne Alors
            i ← position (v) /* position du bit à contrôler */
            Si S_I [i] = 1 Alors
                Empiler (pile, droite(v))
            Sinon
                Empiler (Pile, gauche(v))
                Empiler (Pile, droite(v))
            FinSi
        Sinon /* v est un nœud feuille */
            S = S ∪ pointeur (v)
        Finsi
    FinTantQue
    Retourner (S)
Fin
```

Table 24. *Pseudo-Code de Recherche d'une signature dans TSF*

La *Figure 26* donne un exemple de recherche de la signature S_I = *0101 1010* dans l'arbre de signatures de la *Figure 24*.

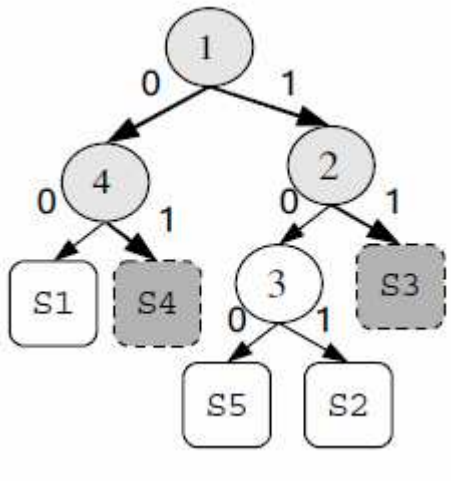


Figure 26. *Recherche d'une signature dans l'arbre TSF*

Le processus de recherche sélectionne les signatures S3 et S4. Ces signatures devront être testées dans l'algorithme de calcul du support estimé.

III.2.2.2. Extraction des Itemsets Fréquents

L'extraction des itemsets fréquents est le processus qui consiste, à partir d'un ensemble d'itemsets candidats, de calculer le support de chacun, *Sup(I)*, et de le comparer à un support minimum, *Minsup*, choisi à priori par l'utilisateur. Un

itemset *I* est dit fréquent si son support *Sup(I)* est supérieur ou égal au support *Minsup*.

Ce processus adopte la stratégie *"Filtrer et Raffiner"*. Cette stratégie est donc constituée de deux phases:

1. Une phase de filtrage pour la découverte des itemsets fréquents probables et
2. Une phase de raffinement pour l'élimination des *false drop*.

Au niveau de l'implémentation, ces deux phases seront intégrées pour déceler et éliminer de manière précoce les *false drop* et éviter ainsi l'effet de ”cascade” (des *false drop* qui peuvent entraîner la découverte d'autres *false drop*).

L'algorithme d'extraction des itemsets fréquents *TSF_Mine* de la *Table 25* utilise la structure TSF, définie plus haut, pour extraire les itemsets fréquents. TSF Mine reçoit en entrée l’ensemble des 1-itemsets fréquents et la valeur du support minimum *Minsup*. Il donne en sortie l’ensemble des itemsets fréquents. *TSF_Mine* opère comme suit:

- Il génère un itemset candidat.
- Il estime son support en utilisant l’algorithme *Compteur_Support*.
- Si le support estimé est ≤ *Minsup*, Il appelle la fonction *CheckCount* pour déterminer si l’itemset est fréquent avec certitude ou s’il peut éventuellement être un *false drop* [27].
- Si l’itemset est reconnu fréquent avec certitude par *CheckCount*, l’algorithme continue la génération de nouveaux candidats.
- Si l’itemset est fréquent avec un certain degré d’incertitude, il faut s'assurer qu'il ne s'agit pas d'un *false drop*, pour éviter l’effet de "*cascade*", déclenchant la découverte d’autres *false drop.* Pour cela, on intègre la phase de raffinement dans celle de filtrage. On détermine le support réel de l’itemset en utilisant la procédure *Refine*.
- Si le support réel est ≤ *Minsup*, l’algorithme continue la génération des itemsets candidats.

```
Algorithme TSF_Mine
    Entrée: Les 1-itemsets fréquents IF et Minsup
    Sortie: L'ensemble d'itemsets fréquents F
Début
    itemset ← NULL;
    count ← 0;
    flag ← 1;
    Filter(IF, itemset, F, count, flag, Minsup);
    retourner (F);
fin
```

Table. 25 *Pseudo-Code de Génération des Itemsets Fréquents*

a) Phase de filtrage

L'objectif de cette phase est de déterminer rapidement un sur-ensemble d'itemsets fréquents en exploitant uniquement l'arbre *TSF*, donc sans avoir accès à la base de transactions.

L'algorithme de filtrage explore l'espace des candidats en profondeur d'abord (DFS). Les descendants d'un item sont visités récursivement avant d'examiner les itemsets qu'on peut obtenir à partir de l'item suivant.

Le support de chaque itemset examiné est estimé en utilisant l'algorithme *Compteur_Support*.

Une fois qu'un itemset est reconnu fréquent (i.e. son support estimé est supérieur ou égal à *Minsup*), il est testé par la procédure CheckCount pour déterminer s'il est fréquent avec 100% de garantie ou avec une certaine incertitude.

La procédure CheckCount, proposée par Lan et al. [27], réduit considérablement le nombre d'itemsets à traiter dans la phase de raffinement.

La *Table 25* représente le Pseudo-Code de l'algorithme *Filter* d'exploration de l'espace des itemsets candidats.

```
Algorithme Filter(IF , itemset, F,count, flag, minsup)
Début
   Si (itemset ≠ NULL) Alors
       F ← F ∪ {itemset};
   FinSi
   Si (IF ≠ NULL) Alors
       IF ← IF - {i} // i un item de IF;
```

```
        Si (Compteur_Support ({i} ∪ itemset) ≥ minsup) Alors
            Check_Count({i}, itemset, count, flag);
            Si (flag ≥ 1) Alors
                Filter(IF, {i} ∪ itemset, F, count, flag);
            Sinon
                /* Soit Vr le vecteur binaire généré par Compteur_Support
                lors de l'estimation du support de ({i} ∪ itemset) */
                Support = Refine(Vr,{i} ∪ itemset);
                Si (support ≥ Minsup) Alors
                    Filter(IF, {i} ∪ itemset, F, count, flag);
                FinSi
            FinSi
        FinSi
    FinSi
Fin
```

Table 26. *Pseudo-Code d'Exploration de l'Espace des Candidas*

b) Phase de raffinement

Dans cette phase, nous procédons à l'élagage des false drop pour déterminer l'ensemble final des itemsets fréquents. Seuls les itemsets reconnus fréquents avec une certaine incertitude sont testés. Nous utilisons une technique qui permet d'´eviter le parcours complet de la base de transactions et d'accéder directement aux transactions pertinentes.

Pour cela, il s'agit d'exploiter pour chaque itemset le vecteur binaire Vr, généré par l'algorithme *Compteur_Support*.

En effet, les bits de Vr élevés à 1 indiquent les positions des transactions concernées. L'accès à ces transactions permettra de vérifier si l'itemset concerné se trouve dans ces transactions ou non.

La *Table 26* représente le Pseudo-Code de l'algorithme *Refine* de calcul du support réel d'un itemset.

```
Algorithme Refine
    Entrée: Le vecteur binaire Vr
            Un itemset I.
    Sortie: Le support réel de l'itemset I.
Début
    Compteur = 0;
    Pour i Allant de 0 à N Faire
        Si (Vr[i] = 1) Alors
            /* la procédure Chercher compare l'itemset I avec la
            transaction Ti et retourne vrai si itemset ⊆ Ti faux sinon */
            Si Chercher (itemset, Ti) Alors
```

```
                Compteur ← Compteur + 1;
            FinSi
        FinSi
    FinPour
    Retourner (Compteur);
Fin
```

Table 27. *Pseudo-Code de Calcul du Support Réel d'un Itemset*

c) Etude de la Complexité

Dans cette section, nous allons nous intéresser au calcul de la complexité théorique de l'algorithme TSF Mine. Pour cela, nous aloons utiliser les notations suivantes.

f: Taille d'une signature (en nombre de bits)

l: Nombre de bits mis à 1 dans la signature s_I de l'itemset I

N: Nombre de transactions (signatures) dans la base de transactions

$n_{items} = |IF|$: nombre d'itemsets fréquents

$n_S = |S|$: nombre de signatures résultats de Signature_Treesearch

$n_{item/tr}$: nombre d'items par transaction

- **Complexité de l'algorithme Compteur_Support**
  - La première ligne génère la signature d'un itemset candidat. La complexité de cette phase dépend de la fonction de hachage utilisée pour la génération des signatures. Nous supposerons que la complexité de cette phase est linéaire et est fonction de la longueur de la signature, soit une complexité de l'ordre de O(f).
  - La deuxième ligne invoque la fonction d'initialisation du vecteur binaire Vr. La complexité de cette étape est de l'ordre de O(N).
  - La troisième ligne est un appel de l'algorithme Signature_Treesearch. Une étude détaillée de la complexité de cet algorithme a été donnée par Chen dans [Che02b, Che04]. A travers cette étude, Chen a démontré que la complexité de cet algorithme est de l'ordre de O(N/2l).
  - La boucle (lignes 6 à 15) itère, sur le nombre de signatures résultat de l'algorithme Signature_Treesearch, un traitement dont la

complexité est de O(1). Donc, la complexité de cette étape est de l'ordre de O(nS).

- L'algorithme fait appel en dernier lieu à l'algorithme *Retourne*r qui retourne le nombre de bits mis à 1 dans Vr. La complexité de cette étape est de l'ordre de O(N).

Donc, la complexité de l'algorithme Compteur_Support sera de l'ordre de O(f +N +N/2l +nS+N) ≈ O(N).

- **Complexité de l'algorithme Check_Count**

Sa complexité est de l'ordre de O(constante) ≈ O(1).

- **Complexité de l'algorithme Refine**

  - La première instruction est une simple affectation. Donc, sa complexité est de l'ordre de O(1).
  - La boucle (lignes 4 à 11) itère N fois l'appel de l'algorithme *Chercher*. Cette fonction recherche un itemset donné dans une transaction. Sa complexité est de l'ordre de $O(n_{item/tr})$.

Il en résulte que la complexité de *Refine* est de l'ordre de $O(N \times n_{item/tr})$.

- **Complexité de l'algorithme Filter**

  - Le coût des instructions, qui apparaissent au niveau des lignes de 3 à 5 est de l'ordre de O(1).
  - La condition de la boucle *TantQue* (ligne 6 à 20) est vérifiée tant qu'il y a des items dans IF. Ainsi, le traitement effectué au niveau du corps de la boucle est itéré n_{item} fois. Pour chaque item, le coût total de cette étape est égal `a la somme des coûts suivants:
    - La ligne 6 est une simple affectation donc sa complexité est de l'ordre de O(1).
    - Le test est un appel à Compteur_Support. La complexité de cette étape est de l'ordre de O(N).
    - La ligne suivante correspond à la complexité de Check_Count, une complexité de l'ordre de O(1).

- ❖ Quand le test de la ligne 10 n'est pas vérifié, il y a un appel à Refine. La complexité est donc de l'ordre de O(N × nitem/tr).
- ❖ Le dernier coût est relatif à l'appel récursif de l'algorithme Filter. Sa complexité est de l'ordre de O((2^nitems + nitems) × (N × nitem/tr)).

- **Complexité de l'Algorithme TSF_Mine**
  - ➢ Les instructions apparaissant au niveau des lignes 3, 4 et 5 sont de simples affectations dont la complexité est de l'ordre de O(1).
  - ➢ La ligne 6 fait appel à l'algorithme récursif Filter.

La Complexité globale de l'algorithme TSF Mine est donc de l'ordre de O((2^nitems + nitems) × (N × nitem/tr)).

Finalement, la complexité de l'algorithme TSF Mine est exponentielle en fonction du nombre d'itemsets fréquents.

III.2.2.3. Conclusion

Dans cette partie, nous avons proposé un modèle de représentation TSF pour représenter une base de transactions. TSF intègre les concepts de fichier et d'arbre de signatures basés sur une représentation binaire compacte. Nous avons présenté l'algorithme TSF_Mine qui exploite le modèle TSF pour l'extraction des itemsets fréquents. Des preuves théoriques ainsi qu'une étude de la complexité théorique ont été fournies.

La section suivante sera consacrée à la présentation de la variante 2 du modèle de représentation d'une base de transactions.

III.2.3. Variante 2: La Structure ST-Tree (Signatures – Transactions – Tree)

L'arbre de signatures *ST-Tree (Signatures – Transactions – Tree)* permet de représenter des signatures de transactions. A chaque transaction est associée une signature (suite de bits) obtenue par application d'une ou de plusieurs fonctions de hachage. *ST-Tree* a l'avantage d'être à la fois une structure compacte (représentation binaire) et dynamique (prise en charge aisée des mises à jour). ST-Tree est composé de deux types de nœuds: un nœud interne et un nœud feuille. Pour chaque nœud interne de ST-Tree, l'arc gauche correspond à la valeur "0" et

l'arc droit à la valeur "1". Chaque nœud de type "feuille" contient trois informations: une signature, le nombre de transactions générant cette signature et la liste des Tids. L'arbre de signatures possède autant de feuilles que de signatures.

III.2.3.1. Processus de Construction de ST-Tree

La première étape consiste à générer la signature de chacune des transactions. La transaction étant composée d'un ensemble d'items, il est nécessaire de générer la signature de chacun des items et, par composition (Or-ing), obtenir la signature de la transaction.

La deuxième étape construit successivement l'arbre ST-Tree. Initialement, l'arbre ST-Tree ne contient qu'un nœud initial (un nœud feuille) composé de la signature de la première transaction, son Tid et un entier égale à 1, signifiant qu'une seule transaction a généré la signature. Lors de l'ajout d'une nouvelle signature s, le parcours de l'arbre commence à partir de la racine. Soit v un nœud interne tel que position(v) = i, signifiant que c'est le $i^{ème}$ bit de s qui sera contrôlé lors du passage par ce nœud. Si s[i] = 0, c'est le fils gauche qui est exploré sinon, c'est le fils droit.

Si v est une feuille, on compare la signature s avec la signature s' contenue dans v. Si s est égale à s', on incrémente le nombre associé à s au niveau de v et on ajoute le Tid correspondant. Sinon, s est une nouvelle signature. Dans ce cas, nous supposons que k soit la position du 1^{er} bit différent entre s et s': ce qui signifie que les bits de 1 à k des deux signatures sont identiques. Un nouveau nœud u interne est ajouté à l'arbre avec position(u) = k+1. Un nouveau nœud feuille v' est également créé. Il contiendra 3 informations: la signature s, le Tid de la transaction générant s et la valeur 1, signifiant qu'une seule transaction a généré la signature s. Selon la valeur du bit de position (k+1), v sera le fils gauche de u et v' le fils droit ou inversement. La *Table 27* donne un exemple de la structure ST-Tree.

Tids	Transactions	Signatures	
T_1	1,5,6,8	S_1	11000110
T_2	2,4,8	S_2	10101000
T_3	5,8	S_3	10000100
T_4	2,3	S_4	00110000
T_5	4,5,7,10	S_5	01001101
T_6	3,10	S_6	00110000
T_7	3,6,7,9	S_7	01010011

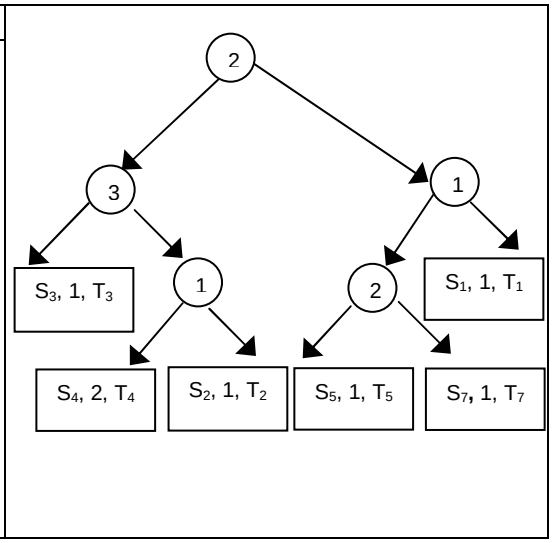


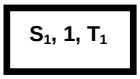
Table 28. *Tids, Transactions, signatures de transactions et ST-Tree*

a) Etapes de construction de ST-TREE

Les étapes de (a) à (g) décrivent le processus de construction de ST-Tree:

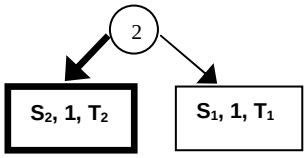
L'étape (a) construit un noeud racine r tel que r soit un noeud feuille contenant la signature S_1, le nombre "1" et T_1 l'identificateur de la première transaction.

(a) Insérer (S_1: 11000110)

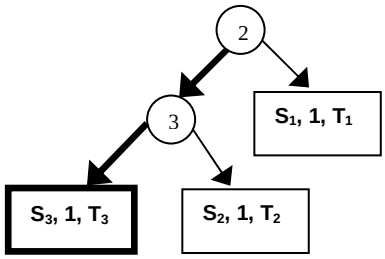


Les étapes (b) à (g) insèrent une nouvelle signature S_i dans le noeud feuille correspondent dans ST-Tree, le parcours se faisant selon la valeur de la position du bit de S_i à contrôler, au niveau de chaque nœud interne rencontré.

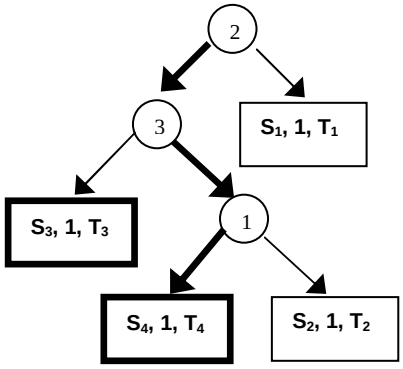
(b) Insérer (S_2: 10101000): le premier bit différent entre S_1 et S_2 est le second bit, $S_1[2] = 1 \neq S_2[2] = 0 \Rightarrow$ créer un nœud interne v avec position(v) = 2 et le nœud feuille $\{S_2, 1, T_2\}$.



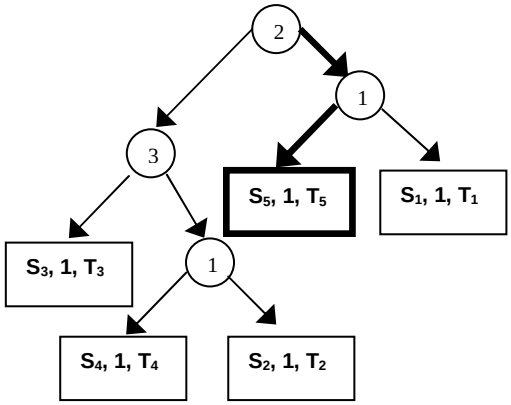
(c) Insérer (S_3: 10000100): $S_3[2] = 0$, le 1^{er} bit différent entre S_3 et S_2 est le $3^{ème}$ bit, $S_2[3]= 1 \neq S_3[3]= 0 \Rightarrow$ créer un nœud interne v avec position(v) = 3 et le nœud feuille $\{S_3, 1, T_3\}$.



(d) Insérer (S_4: 00110000): $S_4[2] = 0 \neq S_4[3] = 1$, le 1er bit différent entre S_4 and S_2 est le 1er bit, $S_4[1] = 0$ et $S_2[1] = 1 \Rightarrow$ créer un nœud interne v avec position(v) = 1 et le nœud feuille $\{S_4, 1, T_4\}$.

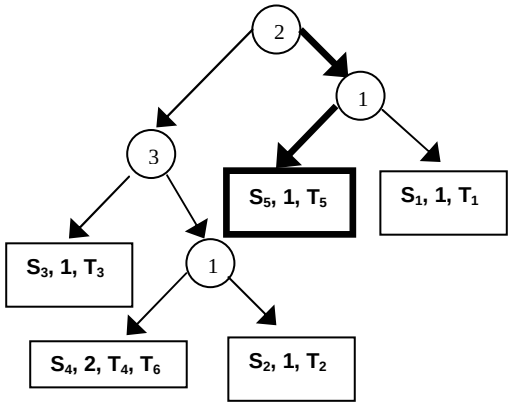


(e) Insérer (S_5: 01001101): $S_5[2] = 1$, le 1er bit différent entre S_1 et S_5 est le 1er bit, $S_1[1] = 1 \neq S_5[1] = 0 \Rightarrow$ créer un nœud interne v avec position(v) = 1 et le nœud feuille $\{S_5, 1, T_5\}$.

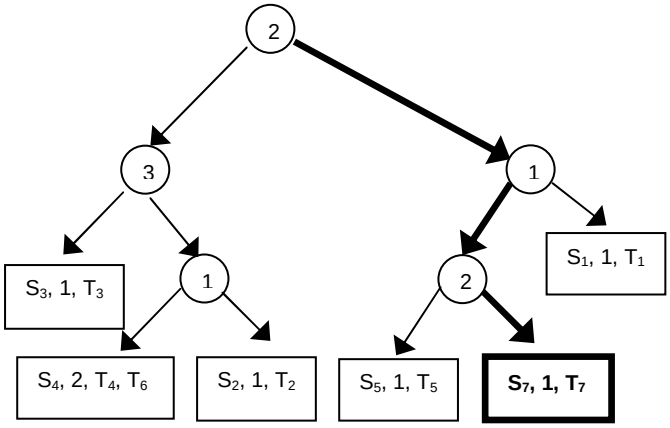


(f) L'étape (f) insère une signature déjà existante.

Insérer (S_6: 00110000): $S_6[2] = 0$, $S_6[3] = 1$, $S_6[1] = 0$. $S_6 = S_4 \Rightarrow$ Incrémenter le nombre de transactions du nœud feuille.



(g) Insérer (S_7: 01010011), $S_7[2] = 1$, $S_7[1] = 0$, le 1er bit différent entre S_5 and S_7 est le 4ème bit, $S_5[4] = 0 \neq S_7[4] = 1 \Rightarrow$ créer un nœud interne v avec position(v) = 4 et le nœud feuille {S_7, 1, T_7}.



b) Algorithme de construction de ST-Tree

L'algorithme *ST-Tree_Construction* de la *Table 28* construit l'arbre relatif à la *Table 27*. Il est composé de deux parties qui se répètent pour chaque transaction:

- Une fonction de hachage *Gen_Signature(T)* qui fournit la signature S_i d'une transaction T_i,
- Une procédure **Insérer** qui insert la signature S_i d'une transaction T_i dans l'arbre ST-Tree.

Nous présentons ci-dessous une description formelle de l'algorithme *ST-Tree_Construction.*

Algorithme ST-Tree_Construction
/* Entrée: Ensemble de transactions Sortie: ST-Tree */
Début
S_1 = Gen_Sig (T_1)
Construire un ST-Tree avec uniquement un nœud racine r. /* r = {S_1, 1, T_1} */

```
Pour i = 2 à n Faire
  Si = Gen_Signature (Ti)
  Insérer (Si)
FinPour
Fin
```

Table 29. *Algorithme de construction de ST-Tree*

Initialement, l'arbre ST-Tree est composé d'un seul nœud qui contient la signature S_1 de la première transaction, la valeur 1 et le Tid T_1 correspondant. La procédure d'insertion de la *Table 29* reçoit en entrée signature S_i d'une transaction T_i et insère S_i dans ST-Tree. Nous parcourons l'arbre à partir de la racine. Soit v un nœud interne avec position(v) = p. Nous allons contrôler $S_i[p]$. Si $S_i[p] = 0$, nous parcourons la branche gauche sinon la branche droite. Si v est un noeud feuille, nous comparons S_i avec la signature S de v. Sil y a égalité, le nombre de transaction correspondant est incrémenté, sinon, nous supposons que les k premiers bits de S_i sont identiques à ceux de S et que S_i et S diffèrent au $(k+1)^{ème}$ bit. Nous créons un nouveau nœud interne u avec position(u) = k+1. La position de v sera occupée par u et v devient un des fils de u. Nous créons également un nouveau nœud feuille v_i, contenant $\{S_i, 1, T_i\}$. Si $S_i[k+1] = 1$, v sera le fils gauche de u et vi le fils droit. Par contre, si $S_i[k+1] = 0$, v sera le fils droit et v_i le fils gauche. Nous présentons ci-dessous la description formelle de l'algorithme Insérer(S_i).

```
Procédure Insérer(Si)
Début
  Pile ← racine
  Tantque Pile non vide Faire
      v ← Dépiler (Pile)
      Si v est un nœud interne Alors
        j ← position (v)
        Si Si[j] = 1 Alors
            Empiler (Pile, fils_droit)
        Sinon
            Empiler (Pile, fils_gauche)
        Finsi
      Sinon  /* v: nœud feuille = S, nt, Tids */
        Si S = Si Alors
            nt ← nt + 1
        Sinon /* Nouvelle signature */
            On suppose que les k premiers bits de S et Si sont identiques
            Donc S diffère de Si au (k+1)ème bit
            Générer un nouveau nœud interne u avec position(u) = k+1
```

```
            Générer un nouveau nœud feuille v' ← {Si, 1, Ti}
            Si Si[k+1] = 1 Alors
              v' sera le fils droit de u et v son fils gauche
            Sinon
              v' sera le fils droit de u et v son fils gauche
            Finsi
          Finsi
        Finsi
    Fintantque
Fin
```

Table 30. *Algorithme d'insertion d'une signature dans ST-Tree*

III.2.3.2. Un support réel pour extraire les itemsets fréquents

L'extraction des itemsets fréquents est le processus qui consiste, à partir d'un ensemble d'itemsets candidats, de calculer le support de chacun, *Support(I)*, et de le comparer à un support minimum, *Minsup*, choisi à priori par l'utilisateur. Un itemset *I* est dit fréquent si son support, *Support(I)*, est supérieur ou égal à *Minsup*.

Le processus d'extraction des itemsets fréquents peut être subdivisé en 3 étapes:

- Le calcul de la signature S_I d'un itemset I, en utilisant la même fonction de hachage que pour la génération d'une signature de transaction,
- Le parcours de *ST-Tree* pour rechercher toutes les signatures qui contiennent S_I. Nous considérons qu'une signature s' contient une signature s si $s' \wedge s = s$.
- Le processus de calcul d'une signature pouvant entraîner un problème de collision (sélection de transactions non réellement référencées: les false drop), il sera nécessaire d'accéder une deuxième fois à la base des transactions pour vérifier si elles contiennent effectivement l'itemset I. Ce qui permet de calculer le support réel de l'itemset I. On dira qu'un itemset I est fréquent si $Support(I) \geq Minsup$.

a) Mécanisme de recherche d'une signature dans ST-Tree

La procédure de recherche, ayant en entrée la signature S_I d'un itemset I, consiste à analyser S_I selon le bit à contrôler au niveau de chaque nœud interne traversé en parcourant ST-Tree. Dans le cas d'un nœud interne, si la

valeur du bit de S_I associée à la position à contrôler est égale à 1, seul le chemin droit est exploré sinon les deux chemins seront explorés. En fait, lors de la correspondance, pour une position de bit donnée i dans S_I, si le bit est à 1, le bit correspondant dans S_I doit être obligatoirement égal à 1. Dans le cas contraire, le bit correspondant peut avoir soit la valeur 1 soit la valeur 0. L'example suivant illustrera de façon plus claire l'idée principale de l'algorithme.

Exemple 1. Soit un itemset I = {5, 8} et sa signature S_I = 10000100. La procédure Rechercher(S_I) visitera les signatures S_1, S_2 et S_3, mais ne gardera que S_1 et S_3 car S_2 ne contient pas S_I. Elle appellera ensuite la fonction Calcul_Support qui vérifiera que les transactions sélectionnées ne sont pas des false drop et ne conservera que les transactions qui contiennent effectivement l'itemset recherché. Dans notre exemple, il n'y a pas de false drop. Elle calculera le support de l'itemset I. Le support de l'itemset I sera donc égal à 2 puisque la somme des nombres de transactions générant S_1 et S_2 est égale à 2.

La *Figure 27* représente en gras le cheminement dans l'arbre ST-Tree lors de la recherche de S_I.

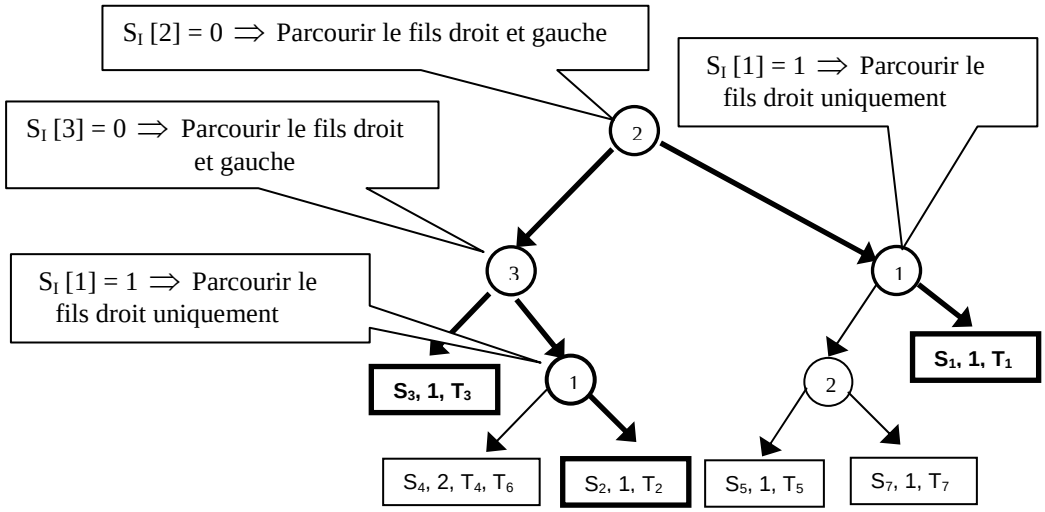


Figure 27. *Processus de recherche de la signature S_I = 10000100*

Nous constatons, qu'à chaque fois que nous rencontrons un bit égale à 1, la branche associée à la valeur 0 est élaguée, seule la branche correspondant à la valeur 1 est parcourue.

b) Algorithme de recherche d'une signature dans ST-Tree

Ayant en entrée une signature S_I, l'algorithme de recherche d'une signature de la *Table 30* consiste à parcourir les branches de l'arbre en se basant, à chaque nœud interne rencontré, sur la position du bit à contrôler dans S_I. Si la valeur de ce bit est égale à 1, seule la branche droite est explorée, sinon, les deux branches sont explorées. Nous présentons ci-dessous la description formelle de l'algorithme Rechercher(I).

```
Algorithme ST-Tree-Search (I)
/* Entrée: Un itemset I */
/* Sortie: Support réel de I*/
Début
   /* Initialement, TidList est vide */
  S_I = Gen_Signature(I)
  Sup_ S_I ← 0
  Empiler (Pile, racine);
  Tantque pile non vide Faire
      v ← Dépiler (Pile);
      Si v est un nœud interne Alors
          i ← position (v) /* position du bit à contrôler */
          Si S_I[i] = 1 Alors
              Empiler (pile, droite(v))
          Sinon
              Empiler (Pile, gauche(v))
              Empiler (Pile, droite(v))
          FinSi
      Sinon /* v est un nœud feuille */
          Comparer S_I avec la signature S' de v
          Si  S'  contient S_I Alors
               TidList ← TidList ∪ {Tids}
          Finsi
      Finsi
  FinTantQue
   Support_I ← Calcul_Support(I,Tidlist)
   Retourner(Support_I)
Fin
```

Table 31. *Algorithme de recherche d'une signature dans l'arbre ST-Tree*

Ce parcours continue jusqu'à arriver à un nœud feuille. Si la signature S' du nœud feuille contient S_I, la liste des Tids correspondant aux transactions qui génèrent S_I est ajoutée à la TidList. Le calcul du support réel de I se fera par appel à une fonction Calcul_Support qui accèdera une deuxième fois à la base des transactions pour éliminer les false drop et calculer le support réel de l'itemset I.

c) Algorithme d'extraction des itemsets fréquents

Le processus d'extraction des itemsets fréquents calcule, pour chaque itemset candidat I, soit un support réel soit un support maximum de l'itemset I. Le support est ensuite comparé au support Minsup fixé à priori par l'utilisateur. L'itemset I sera dit fréquent si son support est supérieur ou égal à Minsup. Nous présentons, dans la *Table 31* ci-dessous, la description formelle de l'algorithme Extraction_IF.

```
Algorithme Extraction_IF
    /* Entrée: 1-Itemsets fréquents */
    /* Sortie: Ensemble des k-itemsets fréquents IF */
Début
    /* Initialement, IF = {1-Itemsets fréquents} */
    Pour i allant de 2 à  n Faire
        Appeler ST-Tree-Search (I_i, Support_I_i)
        Si Support_I_i > Minsup Alors
            IF = IF ∪ {I_i}
        FinSi
    FinPour
    Retourner (IF)
Fin
```

Table 32. *Algorithme d'Extraction des Itemsets Fréquents*

III.2.3.3. Un Support Maximum pour Extraire les Itemsets Fréquents

Pendant le processus de génération des règles d'association, l'étape d'extraction des itemsets fréquents est celle qui prend le plus de temps. Cette étape nécessite plusieurs accès à la base des transactions. L'algorithme Apriori, par exemple, accède à la base autant de fois que d'extraction des k-itemsets fréquents et a une complexité exponentiel. Les autres algorithmes, tel que FP-Growth, utilise une structure FP-Tree. La construction du FP-Tree, basée sur les 1-itemsets fréquents, nécessite deux accès à la base des transactions.

Notre approche, utilisant la structure ST-Tree, une structure binaire, compacte et arborescente, nécessite comme pour FP-Tree, deux accès à la base des transactions, un premier accès pour calculer les signatures de transactions, les représenter sous forme d'arbre de signatures et extraire les 1-.Itemsets fréquents. Seul l'arbre de signatures est utilisé dans la suite du processus d'extraction des itemsets fréquents. Un deuxième accès est nécessaire pour éliminer les false drop (transactions sélectionnées mais non réellement

référencées) et calculer le support réel d'un itemset et le comparer au support minimum choisi par l'utilisateur. Cette approche a été présentée dans la section précédente.

En utilisant la structure ST-Tree, nous avons défini un nouveau concept: le concept de support maximum. Il s'agit du support maximum possible pour un itemset donné. Nous avons associé, au départ, une signature à chaque transaction. Le processus de calcul de signatures peut donner lieu à des collisions. Plusieurs transactions peuvent donc générer la même signature. Nous avons associé, dans notre modèle de représentation de la base des transactions, à chaque signature, le nombre de transactions générant cette signature. Dans le processus d'extraction des itemsets fréquents, nous avons associé à chaque itemset candidat une signature. Le calcul du support nécessite le parcours de l'arbre. Ce parcours va sélectionner toutes les signatures qui contiennent la signature recherchée. Il suffira donc de calculer la somme des nombres associés à chaque signature sélectionnée pour obtenir le ***support maximum*** possible pour chaque itemset candidat I. Plus formellement, nous définiront donc le support maximum MaxSup(I) d'un itemset I comme suit:

$$MaxSup\ (I) = n_1 + n_2 + \ldots + n_m$$

Ce qui nécessite quelques modifications à faire sur les précédents algorithmes. Nous obtenons la description formelle suivante pour l'algorithme de recherche.

```
Algorithme ST-Tree-Search (I)
/* Entrée: Un itemset I */
/* Sortie: Support maximum de I*/
Début
   /* Initialement, TidList est vide */
  S_I = Gen_Signature(I)
  Maxsup ← 0
  Empiler (Pile, racine);
  Tantque pile non vide Faire
      v ← Dépiler (Pile);
      Si v est un nœud interne Alors
          i ← position (v) /* position du bit à contrôler */
          Si S_I[i] = 1 Alors
              Empiler (pile, droite(v))
          Sinon
              Empiler (Pile, gauche(v))
              Empiler (Pile, droite(v))
          FinSi
      Sinon /* v est un nœud feuille */
```

```
        Si S' contient S_I Alors
            Maxsup ← Maxsup + nt
        Finsi
    Finsi
  FinTantQue
   Retourner(Maxsup)
Fin
```

Table 33. *Algorithme de calcul du support maximum*

L'algorithme d'extraction des itemsets fréquents aura la description formelle suivante:

```
Algorithme Extraction_IF
    /* Entrée: 1-Itemsets fréquents */
    /* Sortie: Ensemble des k-itemsets fréquents IF */
Début
    /* Initialement, IF = {1-Itemsets fréquents} */
   1. Générer un itemset candidat I
      Appeler ST-Tree-Search (I, Maxsup)
      Si Maxsup ≥ Minsup Alors
         IF ← IF ∪ {I}
      Finsi
   2. Répéter 1 jusqu'à pas de candidat.
   3. Retourner (IF)
Fin
```

Table 34. *Algorithme d'Extraction des Itemsets Fréquents*

Si nous considérons comme exemple, la signature S_I = 10000100 de l'exemple 1 et Minsup = 2, nous sélectionnons les signatures S_1 et S_3. Le support maximum sera donc égale à 2 ($nt_1 + nt_3 = 1 + 1 = 2$), nt_1 représentant le nombre de transactions générant S_1 et nt_2 représentant le nombre de transactions générant S_3. Nous concluons donc que I est un itemset fréquent (Maxsup = Minsup).

```
Algorithm ST-Mine
    Input: {Transactions}
    Output: {Frequent Itemsets}
Begin
    ST-Tree-construction
    Extraction_FI
End
```

Table 35. *Algorithme ST-Mine*

III.2.3.4. Etude de la complexité théorique

L'algorithme ST-Tree-Construction calcule la signature de chaque transaction et l'insère dans l'arbre. Si nous considérons que le nombre de transactions est égale à n et la taille d'une signature est égale à m, la complexité de cet algorithme est de l'ordre de O(n*m).

L'algorithme Insérer(S_i), quand à lui, requiert un parcours pour l'insertion de la première signature, deux pour la seconde et ainsi de suite jusqu'à la dernière signature. Le nombre de branches parcourue est donc de:

$1 + 2 + ... + n = n (n +1) / 2 = (n^2 + n) / 2$

Sa complexité est de l'ordre de $O(n^2)$.

La complexité de l'algorithme de recherche ST-Tree-Search est de l'ordre de $O(n/2^l)$, où n est le nombre de signatures de transactions et l le nombre de bits à 1 dans une signature. Dans le cas le plus défavorable, la complexité de cet algorithme est de l'ordre de $O(n/2^m) \approx O(n)$

L'algorithme Extraction_FI contient une boucle qui s'exécute p fois (p étant le nombre d'itemsets candidats. Le traitement des itemsets candidats nécessite p fois l'exécution de l'algorithme de recherche. Donc, sa complexité est de l'ordre de $O(p(n/2^m)) \approx O(pn)$.

Finalement, la complexité de l'algorithme ST-Mine est polynomiale et elle est de l'ordre de $O(nm) + O(n^2) + O(pn)$.

III.3. Conclusion

Dans ce chapitre, nous avons présenté notre approche basée sur un nouveau modèle de représentation binaire, compacte et arborescente de la base des transactions. Une présentation détaillée de chacune des sous-approches, basées sur ce modèle, a été faite. La première est basée sur un arbre de signatures et un fichier signatures et la seconde uniquement sur un arbre de signatures. Les signatures sont générées à l'aide d'une fonction de hachage. La notion de support maximum d'un itemset a été introduite et utilisée dans le processus d'extraction des itemsets fréquents. Si, pour la première variante, deux accès sont nécessaires

lors du processus d'extraction des itemsets fréquents, un seul est nécessaire pour la seconde variante.

Le chapitre suivant sera entièrement consacré à l'implémentation des algorithmes définis pour chacune des variantes proposées. Une analyse des résultats obtenus sera faite, par comparaison aux résultats obtenus par l'algorithme Apriori et l'algorithme FP-Growth.

CHAPITRE IV

IMPLEMENTATION ET INTERPRETATION DES RESULTATS

IV. IMPLEMENTATION ET INTERPRETATION DES RESULTATS

IV.1. Introduction

L'implémentation s'appuiera sur les algorithmes d'extraction des itemsets fréquents dans le processus de génération des règles d'association, présentés dans le chapitre précédent.

La première variante consiste à associer une signature binaire à chaque transaction, par application d'une fonction de hachage et à construire au fur et à mesure l'arbre de signatures ainsi que le fichier de signatures. L'extraction des 1-Itemsets fréquents se fait en même temps que la construction de l'arbre. A chaque 2-Itemset candidat, construit à partir de l'ensemble des 1-Itemsets fréquents, est associée une signature obtenue de la même manière que les signatures des transactions. Une recherche dans l'arbre de signatures permettra de sélectionner toutes les signatures de transactions qui contiennent la signature recherchée. Le fichier de signatures sera utilisé pour éliminer les falses drop (transactions sélectionnées mais non réellement référencées). Cette étape nécessite un deuxième accès à la base des transactions.

La deuxième variante, par contre, n'utilise que l'arbre de signatures dans le processus d'extraction des itemsets fréquents. La différence réside dans le contenu des feuilles de l'arbre et dans la manière d'extraire les itemsets fréquents. Si dans la première variante, la feuille ne contient qu'un lien vers le fichier de signatures, elle contient dans la seconde, la signature, le nombre de transactions générant cette signature et l'ensemble des Tids générant cette signature. Ces informations seront différemment utilisées, selon le choix du procédé à appliquer.

Le premier procédé consiste à calculer un support réel. Dans ce cas, les Tids correspondants à chaque signature permettent un deuxième accès à la base des transactions pour éliminer les falses drop et ne garder que les transactions contenant effectivement l'itemset recherché, pour calculer le support.

Le second procédé se base quant à lui sur la notion de *support maximum* possible d'un itemset, procédé ne nécessitant pas d'autres accès à la base des transactions. Le support maximum est égal à la somme des nombres de transactions, se trouvant dans les feuilles sélectionnées.

Le processus sera répété pour tous les itemsets candidats. Il se terminera quand il n'y aura plus d'itemsets candidats. Durant ce processus, chaque k-Itemset candidat créé

sera immédiatement traité. Seuls les k-Itemsets fréquents seront sauvegardés pour construire éventuellement les (k+1)-Itemsets candidats.

Nous présenterons dans ce chapitre les deux architectures de l'implémentation, relatives aux deux variantes définies précédemment. Nous détaillerons par la suite les spécificités des bases de transactions (benchmarks) utilisées pour les tests. Nous terminerons ce chapitre par les résultats expérimentaux obtenus et les analyses de ces résultats.

IV.2. Architecture de l'Implémentation

Nous allons détailler dans ce paragraphe deux architectures d'implémentation, relatives à chacune des variantes présentées dans le chapitre précédent. Ils illustreront l'implémentation du système et sa mise en œuvre.

IV.2.1. Architecture relative à la variante 1

Dans la première variante, nous considérons que la base de transaction est représentée à l'aide d'un arbre de signatures et d'un fichier de signatures (*Figure 28*). Les deux structures obtenues seront utilisées dans le processus d'extraction des itemsets fréquents.

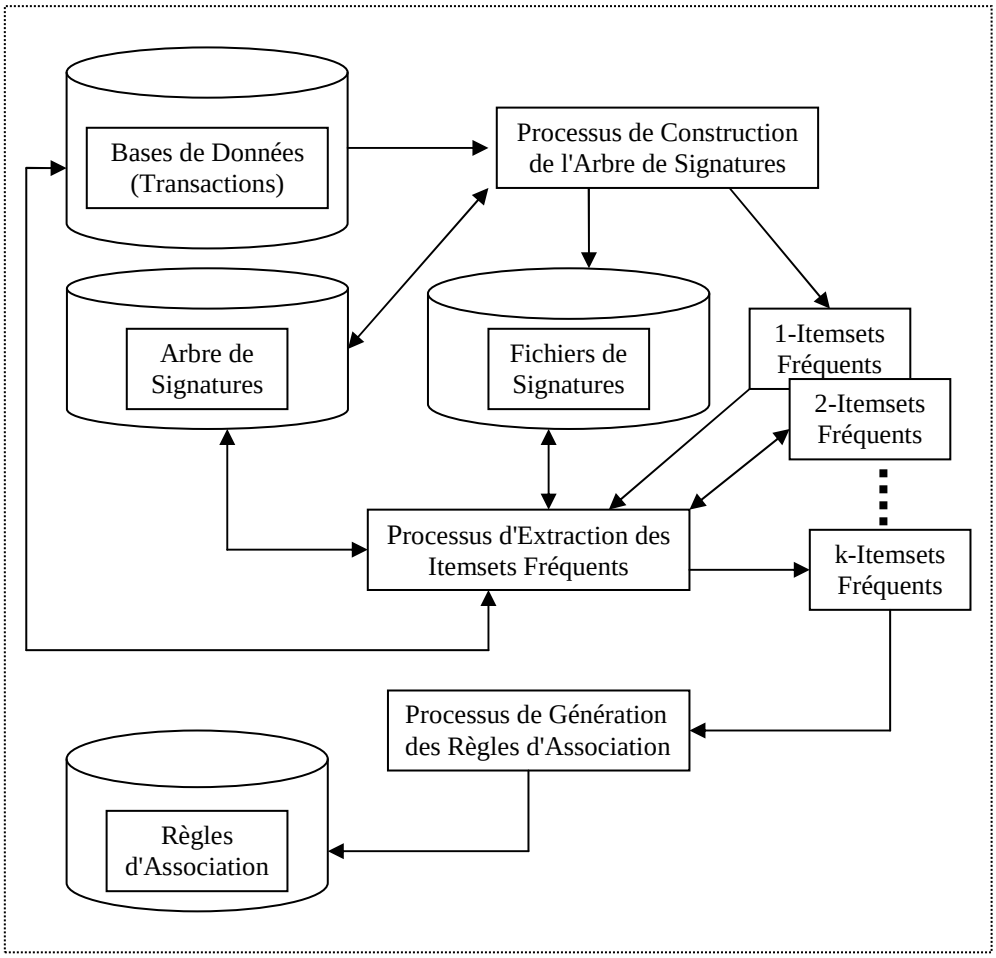


Figure 28. *Architecture de l'Implémentation du Système – Variante 1.*

La base des transactions et le support minimum choisi par l'utilisateur représente l'entrée du système. A chaque transaction est associée une signature binaire obtenue par application d'une fonction de hachage. La première transaction génère une signature S_1 qui est insérée dans le fichier de signatures. Une feuille est créée et sera la racine *r* de l'arbre TSF. Elle contiendra un pointeur vers la position de S_1, dans le fichier signature. La même fonction est appliquée sur la $2^{ième}$ transaction pour générer la signature S_2 qui est insérée dans le fichier de signatures. Une comparaison des deux signatures S_1 et S_2 permet de détecter le premier bit différent entre S_1 et S_2. Un nouveau nœud racine *r* est créé. Ce nœud *r* contiendra la position *k* du 1^{er} bit différent entre les deux signatures S_1 et S_2. Un nouveau nœud feuille est également créé et contiendra le pointeur vers S_2. Si $S2[k] = 0$, alors le nouveau nœud feuille sera le fils gauche de *r* et l'ancienne racine le fils droit ou inversement.

Le calcul et le traitement des autres signatures se feront selon le même procédé. Soit S_i la signature de la transaction T_i. Le processus d'insertion de S_i dans l'arbre TSF exige son parcours, à partir de la racine *r*. L'arbre contient deux types de nœuds: un nœud interne contenant la position *p* du bit à contrôler lors du parcours et un nœud feuille contenant un pointeur vers le fichier de signatures. Selon la valeur du bit à contrôler, on explore soit le chemin gauche, soit le chemin droit. Ce parcours continue jusqu'à arriver à une feuille *f* de l'arbre. Si la signature *S* dans la feuille est égale à S_i (phénomène de collision), la signature S_i est ajoutée au fichier signature et un lien est créé entre les signatures identiques. Sinon, S_i est une nouvelle signature. Une nouvelle feuille *f'* est créée contenant un pointeur vers la position de S_i, dans le fichier de signatures. On suppose que *S* et S_i diffèrent au $k^{ième}$ bit. Un nœud interne "*u*" est créé avec comme position à contrôler "*k*". Si *S[k] = 0*, *f* sera le fils gauche de *u* et *f'* le fils droit ou inversement. Le processus continue jusqu'à l'insertion dans l'arbre TSF de toutes les signatures de transactions.

La construction de l'arbre s'accompagne de l'extraction des 1-Itemsets fréquent. Lors de la lecture des transactions, les itemsets de taille 1 sont extraits et leurs supports calculés au fur et à mesure. Seuls ceux qui ont un support supérieur au support minimum choisi par l'utilisateur seront gardés et utilisés dans la suite du processus d'extraction des itemsets fréquents.

L'implémentation du système nécessite plusieurs phases. La *Figure 29* représente les étapes de l'algorithme d'extraction dans le cas de la variante 1.

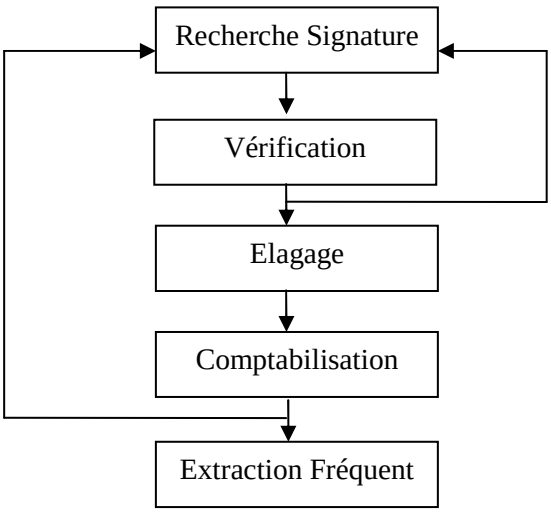


Figure 29. *Différentes Phases de l'algorithme – Variante 1.*

L'algorithme reçoit en entrée la signature S_I d'un itemset I. La recherche de S_I débute au niveau de la racine de l'arbre. Soit *p* la position du bit à contrôler. Si $S_I[p] = 0$, les deux chemins sont explorés, sinon seul le chemin droit est exploré. Le parcours de chacune des branches continue jusqu'aux feuilles. L'étape de *Vérification* consiste à comparer la signature *S* de la feuille à S_I. Si $S_I \neq S$, on revient à l'étape *Recherche Signature* pour traiter une autre branche. Si $S_i = S$, l'étape *Elagage* permet, en accédant à la base des transactions, d'éliminer les falses drop (transactions sélectionnées mais non réellement référencées). L'étape de *Comptabilisation* consiste à calculer le nombre de transactions sélectionnées pour chaque signature, ensuite de revenir à l'étape *Recherche Signature*. La recherche de signatures continue jusqu'à exploration de toutes les branches à partir de chaque nœud non exploré. L'étape *Extraction Fréquent* permet, en comparant le support calculé et le support minimum, de sauvegarder l'itemset s'il est fréquent.

IV.2.2. Architecture relative à la variante 2

Nous utilisons, pour la seconde variante, uniquement l'arbre de signatures ST-Tree pour représenter la base de transactions et extraire les itemsets fréquents (*Figure 25*).

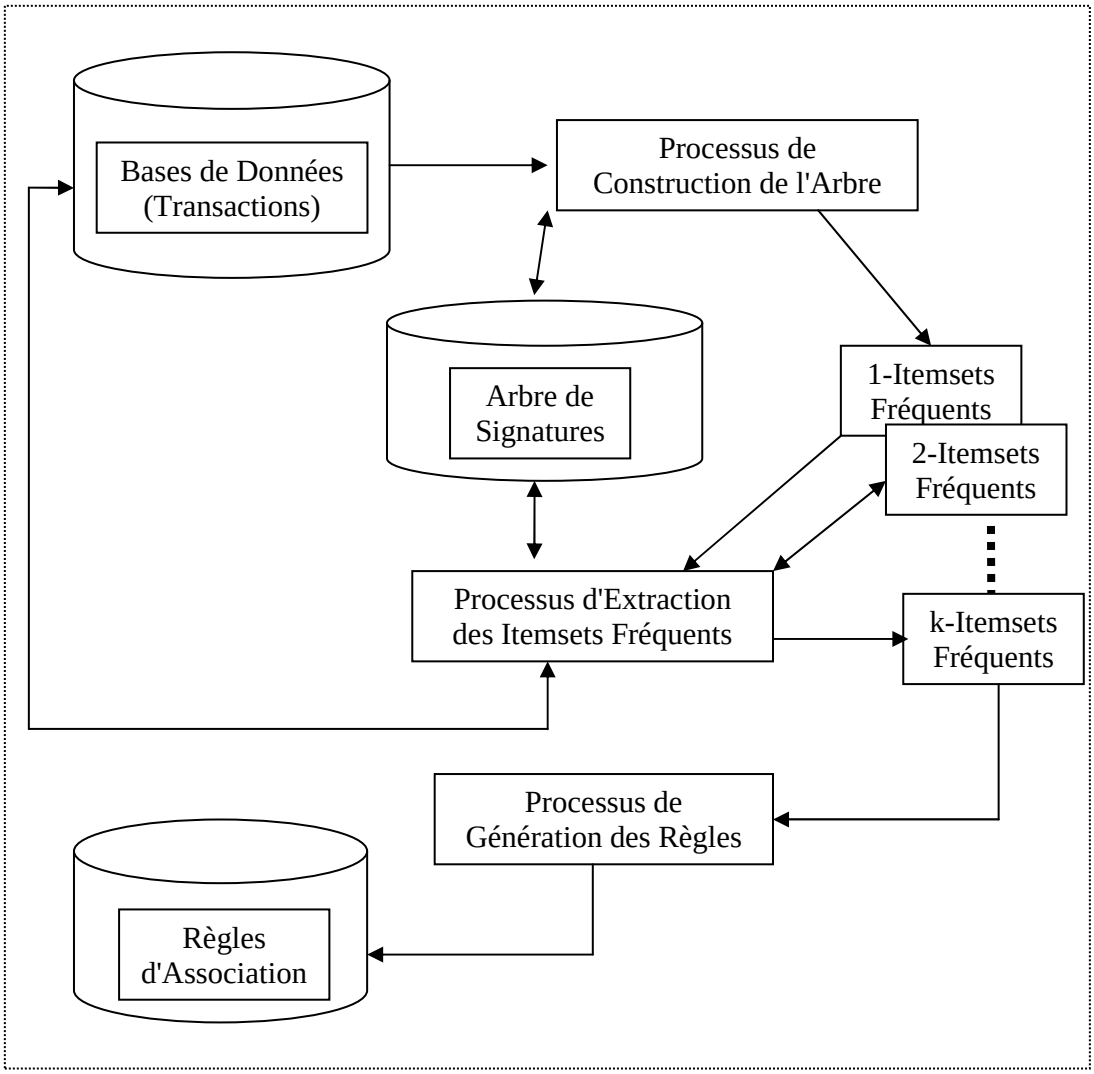


Figure 30. *Architecture de l'Implémentation du Système – Variante2.*

La différence entre la variante 1 et la variante 2 se situe principalement au niveau de la structure de la feuille des deux arbres et de la non utilisation du fichier de signatures dans la variante 2. Ce qui modifie sensiblement le processus d'extraction des itemsets fréquents et les algorithmes associés.

Ce processus reçoit en entrée, comme pour la variante 1, une base de transactions et un support minimum fixé par l'utilisateur. Le processus de construction de l'arbre est également le même sauf pour le contenu des feuilles de l'arbre ST-Tree. Chaque feuille contient une signature, une liste de Tid et le nombre de transactions générant cette signature.

Pour la partie "extraction des itemsets fréquents", l'étape de recherche de la signature d'un itemset est identique à celle de la variante 1.

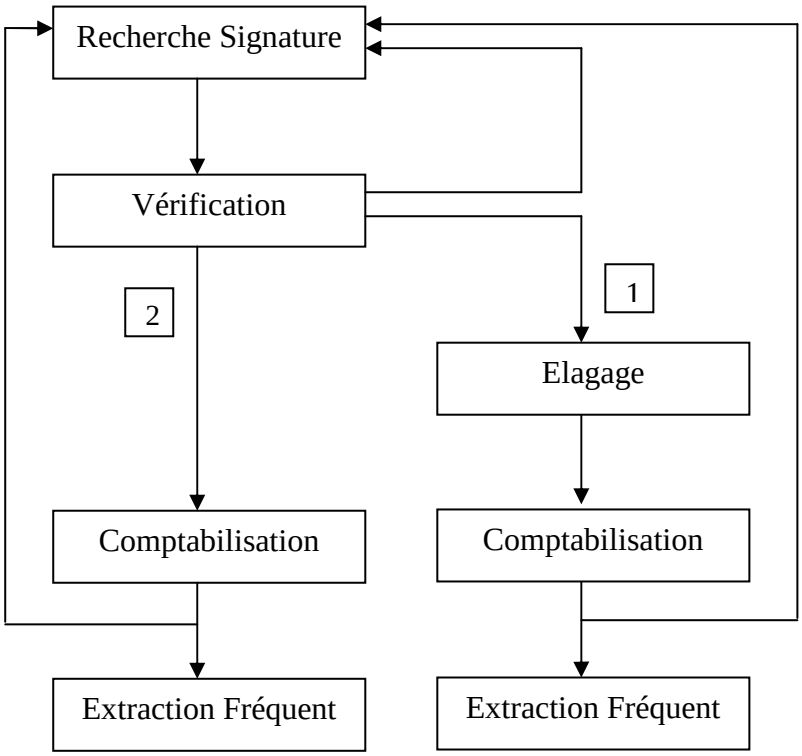


Figure 31. *Différentes Phases de l'algorithme – Variante 2.*

La *Figure 31* représente les étapes de l'algorithme d'extraction dans le cas de la variante 2. Cette étape est exécutée selon deux approches:

1. Calcul d'un support réel. Pour cela, un deuxième accès à la base des transactions est nécessaire afin d'éliminer les falses drop, élaguer les transactions non réellement référencées et ne comptabiliser que les transactions contenant effectivement l'itemset candidat I.

2. Calcul d'un support maximum possible. Pour cela, il y a utilisation du nombre de transactions générant une signature pour calculer le support maximum possible d'un itemset, sans avoir besoin d'un autre accès à la base des transactions.

Les phases *Recherche* et *Vérification* sont semblables à celles de la variante 1. La différence réside dans les phases *Elagage* et *Comptabilisation*. En effet, si $S_I = S$, nous considérons deux cas de figure:

- Le premier cas de figure suppose que toutes les transactions générant S sont susceptibles de contenir l'itemset générant S_I. La cardinalité de l'ensemble des Tids est comptabilisée dans le calcul du support maximum possible de l'itemset candidat.

- Le second cas de figure considère que les transactions générant la signature S ne contiennent pas nécessairement l'itemset candidat I. Un deuxième accès à la base des transactions devient nécessaire pour éliminer les transactions falses drop, en utilisant les Tids se trouvant dans la feuille courante de l'arbre. Seules sont comptabilisées les transactions contenant l'itemset candidat I. Après la phase de Comptabilisation, il y a un retour à la phase Recherche Signature pour traiter les chemins restants.

Dans les deux cas, l'itemset I est fréquent si son support (support réel ou support maximum) est supérieur au support minimum.

IV.3. Spécificités des Bases de Transactions Utilisées pour les Tests

Nous distinguons deux types de bases de transactions: les bases denses et les bases éparses. Nous avons donc choisi pour nos tests des bases représentant les deux types de bases. Le site FIMI offre un ensemble de Benchmarks à la communauté des chercheurs en fouille de données et plus précisément aux tests des travaux liés à l'étape d'extraction des itemsets fréquents, de la technique de génération des règles associatives [92]. Ces Benchmarks sont des bases denses, éparses, réelles et synthétiques. La *Table 32* donne les caractéristiques des bases que nous avons utilisées pour les différents tests effectués.

Nom de la Base	Type de la Base	Nombre de Transactions	Nombre d'Items	Taille Moyenne des Transactions	Taille de la Base
T10I4D100k	Eparse	100 000	870	10,1	3,93 Mo
T40I10D100K	Eparse	200 000	942	39,54	14,8 Mo
Mushroom	Dense	8124	119	23	565 Ko
Chess	Dense	3196	75	37	334 Ko
Retail	Eparse	88162	16469	10,3	4156 Ko
Accident	Eparse	340 184	468	33,8	33,9 Mo
Kosarak	Eparse	990 002	41935	8,1	30,5 Mo

Table 36. *Caractéristiques des Bases de Transactions.*

Les bases *T10I4D100k* et *T40I10D100K* sont des bases synthétiques, générées par le groupe de recherche d'IBM Almaden Quest. Leurs contenus représentent le comportement d’achats des clients dans des magasins.

Mushroom a été proposée par Roberto Bayardo, membre de l'UCI. Elle contient les caractéristiques de 23 espèces différentes de champignons.

*Retai*l a été proposée par Tom Brijs. Elle contient les achats d'un supermarché en Belgique.

Chess renferme des données dérivées des étapes d'un jeu.

Accidents a été proposée par Karolien Geurt. Elle contient des données chiffrées concernant les accidents de la route dans la région de Flander en Belgique entre 1991 et 2000. Ces données appartiennent à l'Institut National des Statistiques Belge (NIS).

Kosarak a été proposée par Ferenc Bodon. Elle contient les données chiffrées liées aux clicks dans un portail Hongrois d'information en ligne. Chaque ligne représente une session (les liens sur lesquels un même utilisateur peut cliquer).

IV.4. Résultats d'expérimentation

Dans cette section, nous présentons les résultats expérimentaux de nos tests qui ont été effectués en utilisant les bases de données synthétiques et réelles de la *Table 27*. Ces tests correspondront aux deux variantes présentées plus haut.

IV.4.1. Résultats relatifs à la variante 1

L’ensemble des expérimentations que nous avons fait ont été effectuées sur un PC muni d’un processeur Pentium IV sous SuSE linux 10.0, de fréquence 3,2 GHz et 512 Mo de RAM.

L’algorithme *TSF-Mine* et les algorithmes *Apriori* et *FP-Growth* ont été implantés en langage C et compilés à l'aide du compilateur *gcc*.

Nous avons utilisé la fonction de hachage *Sha-1* pour générer les signatures. *Sha-1* (Secure Hash Algorithm) est une fonction de hachage cryptographique conçue par l’Agence Nationale de Sécurité des Etats-Unis (NSA). Elle produit comme résultat une signature de 160 bits.

Pour optimiser nos algorithmes, nous avons utilisé deux méthodes: la première méthode est une méthode de filtrage, la seconde une méthode de condensation des transactions.

IV.4.1.1. Optimisation par filtrage

Nous avons commencé par filtrer les transactions en éliminant les items non fréquents et en gardant uniquement les items fréquents. Ensuite, nous avons gardé les transactions pertinentes, transactions contenant au minimum deux items fréquents. Ce qui diminue sensiblement le nombre de false drop au cours du calcul des signatures de transactions, le nombre de signatures dans le fichier de signatures et donc la taille de l'arbre de signatures.

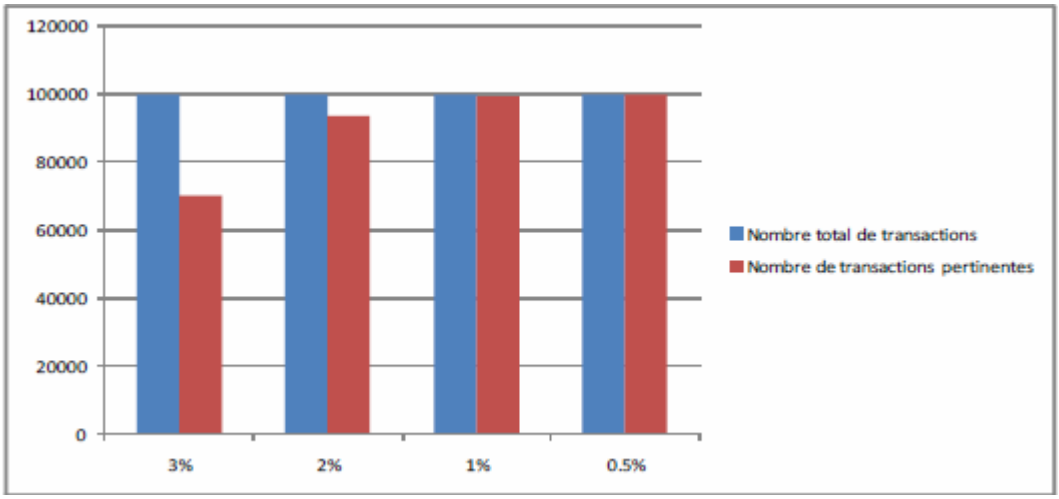


Figure 32. *Filtrage de la base T10.I4.D100K*

La *Figure 32* montre que le nombre de transactions non pertinentes est très important. Nous remarquons que ce nombre devient de plus en plus grand pour des supports > 1.

Nous avons également testé l'influence de ce filtrage sur le nombre de false drop sur la base T10.I4.D100K. La *Figure 33* illustre les deux cas: le nombre de false drop sans filtrage et le nombre de false drop avec filtrage. Nous considérons que le taux de false drop est calculé selon la formule suivante:

$$TFD = \frac{NbrFD}{NbrT} \text{ Où}$$

TFD: Taux de false drop
NbrFD: Nombre de false drop
NbrT: Nombre total de signatures

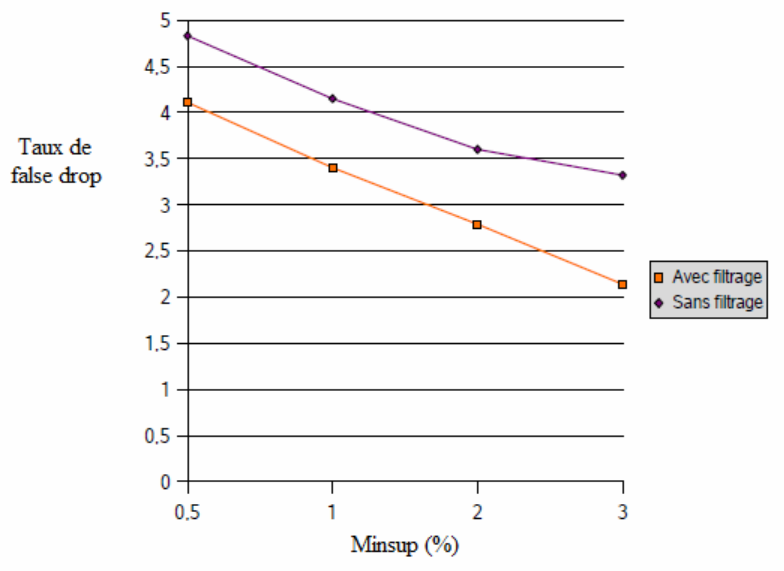


Figure 33. *Impact du Filtrage pour la base T10.I4.D100K*

Nous remarquons, en dehors du fait que le processus de filtrage réduit l'espace mémoire utilisé, que le nombre de false drop diminue également. Cette diminution s'accentue avec l'augmentation de Minsup.

IV.4.1.2. Optimisation par condensation

En dehors du fait que nous pouvons avoir des transactions qui se répètent dans la base des transactions, le processus de filtrage des items non fréquents peut également générer des transactions similaires. Ce qui revient à stocker une même transaction plusieurs fois. Ce stockage, donnant lieu à un surcoût, augmente sensiblement le temps de calcul lors de la phase d'extraction des itemsets fréquents. Bodon propose de représenter chaque transaction une seule fois et de lui associer un compteur d'occurrences. Au cours de l'estimation du support d'un itemset candidat I, si une transaction apparaît n fois, nous vérifions une seule fois si la signature de la transaction contient la signature de I. Dans le cas positif, nous ajoutons n au support de I.

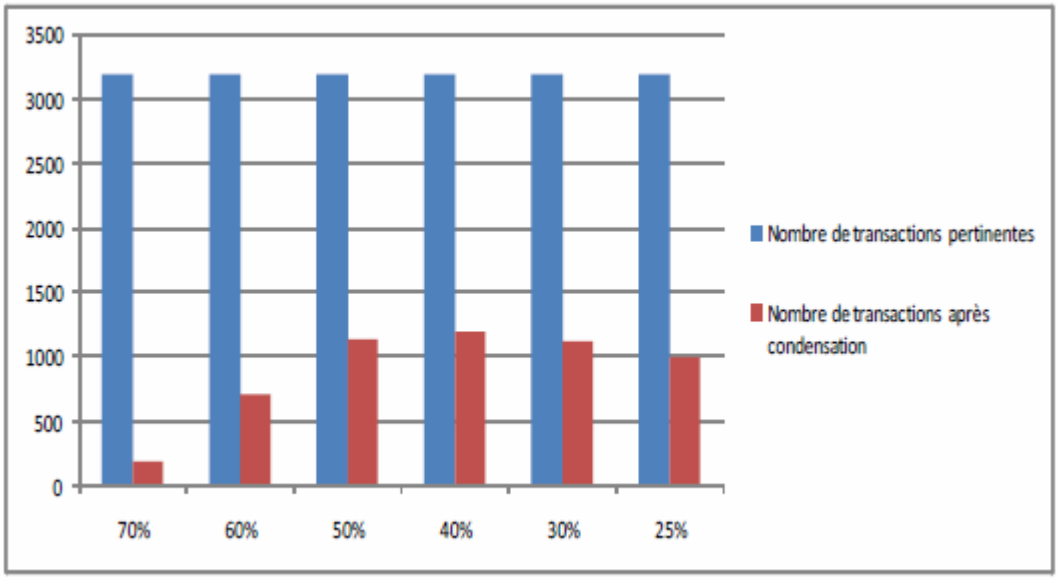


Figure 34. *Minsup vs #Transactions pour la base Chess*

Les *Figures 34, 35 et 36* représentent l'impact du processus de condensation en termes de nombre de transactions pertinentes (après filtrage) et nombre de transactions après condensation.

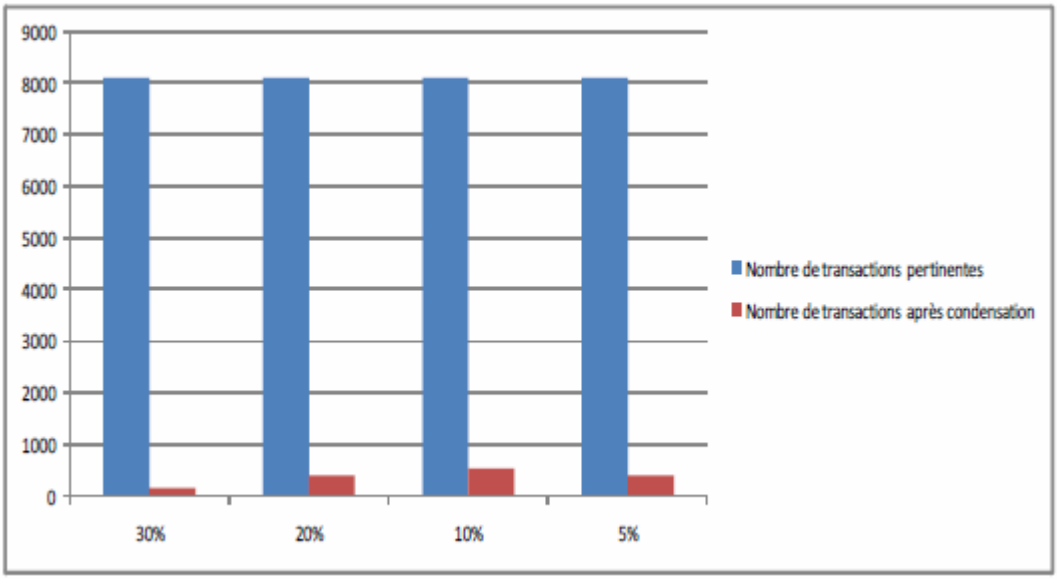


Figure 35. *Condensation des transactions pour la base Mushroom.*

Nous constatons, dans la *Figure 34*, que, pour la base Chess, le nombre de transactions obtenues après le processus de condensation est important. Le taux de condensation est compris entre 62,39 % et 93,83 %, selon le support minimum considéré.

La *Figure 35* montre que le taux de condensation de la base Mushroom est encore plus important que pour la base Chess. Il est compris entre 93,27 % et 97,70 %.

La *Figure 36* montre quand à elle que le taux de condensation pour la base T10.I4.D100K est également aussi important et reste compris entre 56,50 % et 96,60 %.

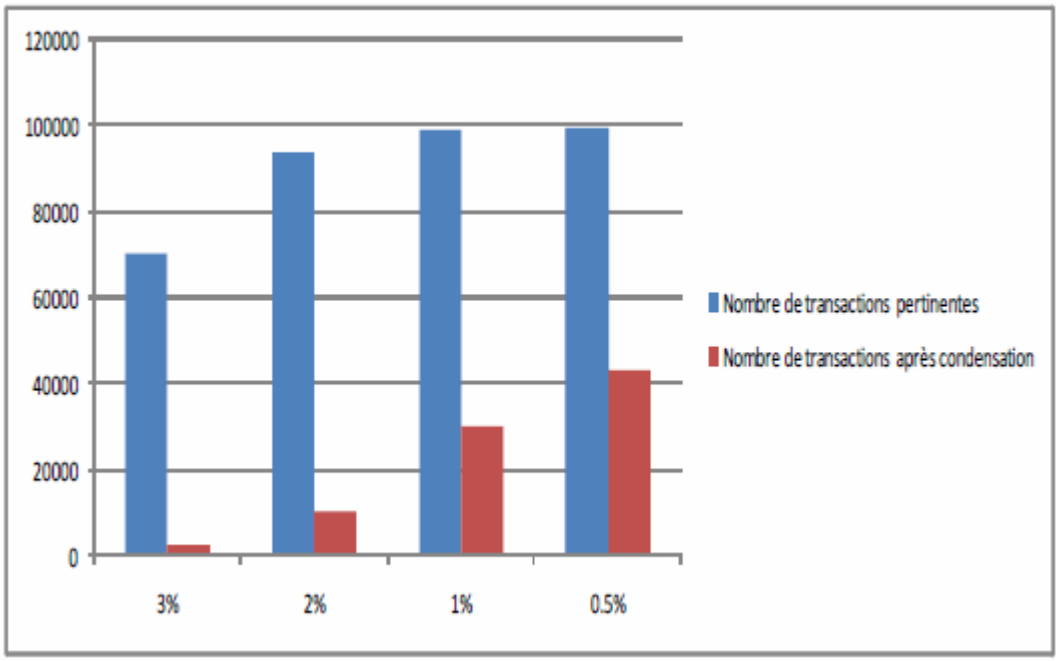


Figure 36. *Condensation des transactions pour la base T10.I4.D100K*

Pour montrer que le processus de condensation a un impact réel sur les performances de l'algorithme TSF_Mine, nous avons utilisé une base dense, la base Mushroom, et une base éparse, la base T10.I4.D100K.

La *Figure 37* illustre les temps d'exécution de l'algorithme TSF_Mine avec condensation de transactions et sans condensation, pour la base Mushroom.

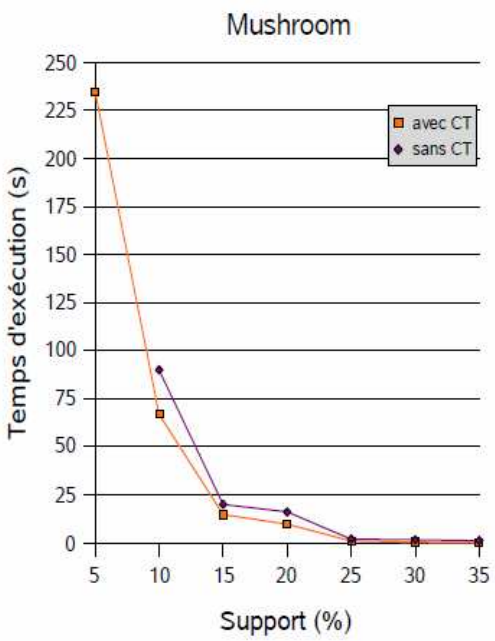


Figure 37. *Minsup vs Temps d'exécution pour Mushroom.*

Nous constatons que le processus de condensation réduit le temps d'exécution. Ce taux varie entre 33,00 % et 97,70 %, avec Minsup ≤ 25%, pour la base Mushroom. Nous remarquons en effet que le temps d'exécution, pour Minsup ≥ 25%, est sensiblement le même avec condensation ou sans condensation. Dans le cas de Mushroom, la condensation, en plus de la réduction du temps d'exécution, permet un gain d'espace mémoire. Nous constatons que, pour un support minimum de 5%, l'algorithme TSF_Mine, sans condensation, se plante. Alors que, si on applique le processus de condensation, il n'y a pas de problème de blocage.

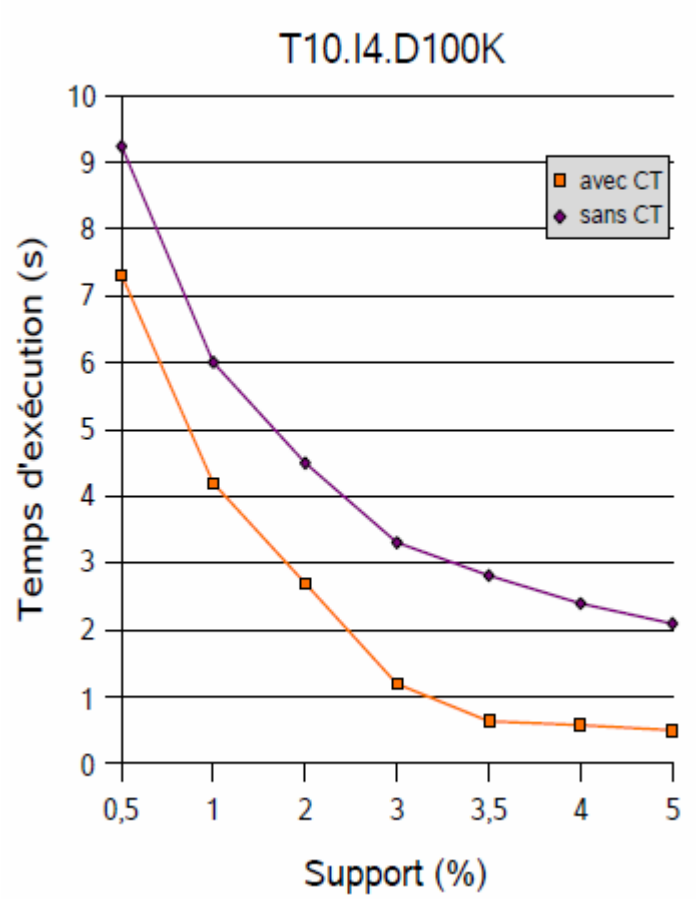


Figure 38. *Minsup vs Temps d'exécution pour T10.I4.D100K*

La *Figure 38* montre par contre que la réduction du temps d'exécution, pour la base *T10.I4.D100K*, varie entre 28,60% et 76,00%. O remarque cependant que cette réduction est plus visible quelque soit la valeur de Minsup, particulièrement pour de petits supports.

IV.4.1.3. Performances de TSF_Mine, Apriori et FP_Growth

Dans cette section, nous comparons les performances de notre algorithme, TSF_Mine, avec celles de l'algorithme Apriori et celles de FP_Growth. Nous pour cela les bases Mushroom, Chess, T10.I4.D10K, T10.I4.D10K et T10.I4.D100K.

Nous constatons sur la *Figure 39*, pour la base Mushroom, que les résultats de FP_Growth sont de loin meilleurs que ceux de TSF_Mine et de Apriori. Nous constatons également que pour des supports inférieurs à 15%, nous obtenons de meilleurs résultats qu'Apriori.

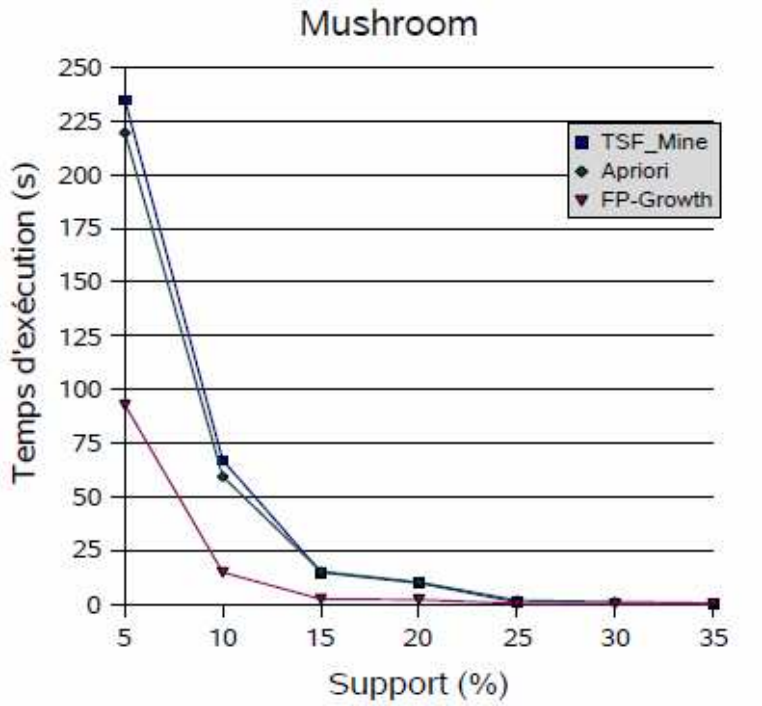


Figure 39. *Minsup vs Temps d'exécution pour Mushroom.*

Sur la *Figure 40*, nous constatons que, pour la base Chess, l'algorithme FP_Growth obtient de meilleurs résultats que les deux autres algorithmes. L'algorithme Apriori donne des temps d'exécution plus intéressants que notre algorithme.

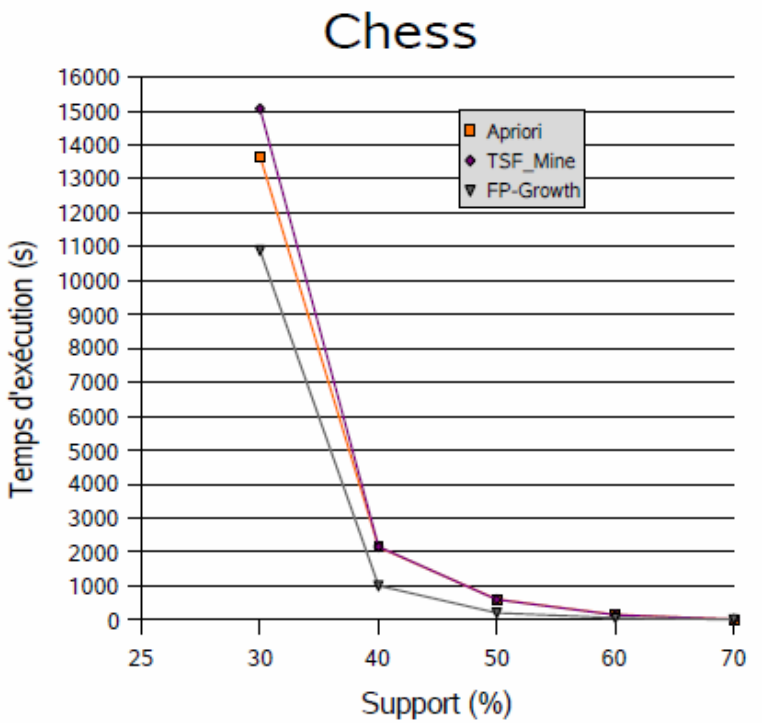


Figure 40. *Minsup vs Temps d'exécution pour Chess.*

La *Figure 41* montre que les performances de l'algorithme TSF_Mine pour la base T10.14.D10K restent non satisfaisantes, spécialement pour de petits supports. Nous constatons en effet que pour des supports qui dépassent 3,5%, nos résultats sont meilleurs que ceux d'Apriori et de FP_Growth.

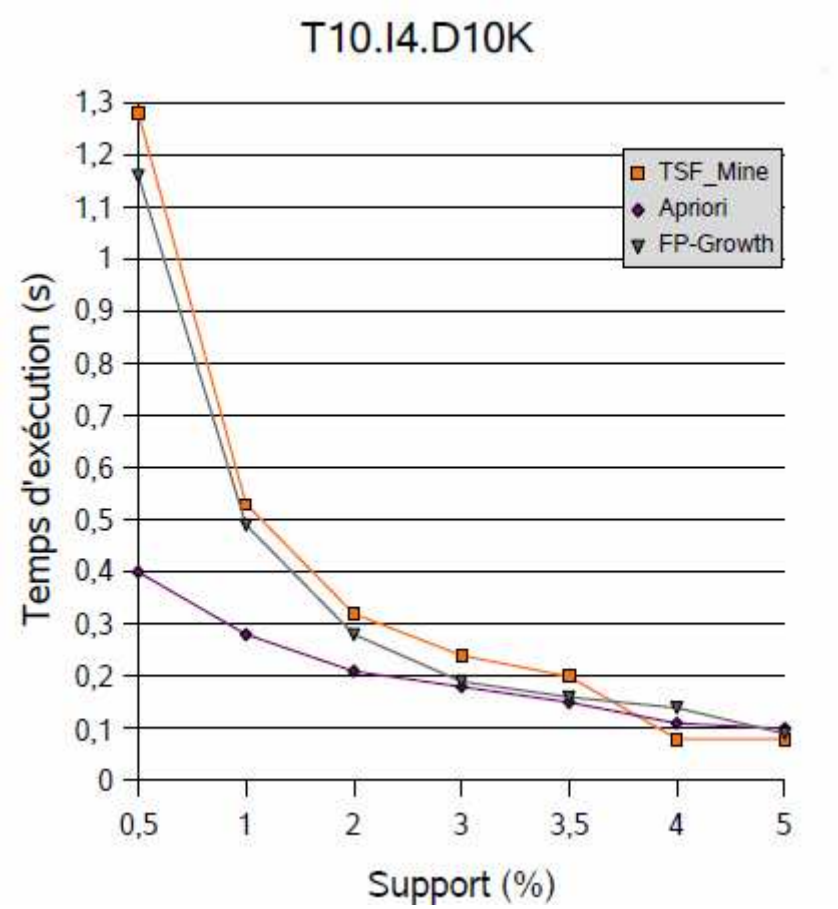


Figure 41. *Minsup vs Temps d'exécution pour T10.14.D10K.*

Par contre, pour la base T10.I4.D40K de la *Figure 42*, nous constatons que pour des supports supérieurs à 2%, l'algorithme TSF_Mine donne des temps d'exécution inférieurs que ceux fournis par FP_Growth. Nous obtenons également de meilleurs résultats par rapport à Apriori, pour des supports supérieur à 3%.

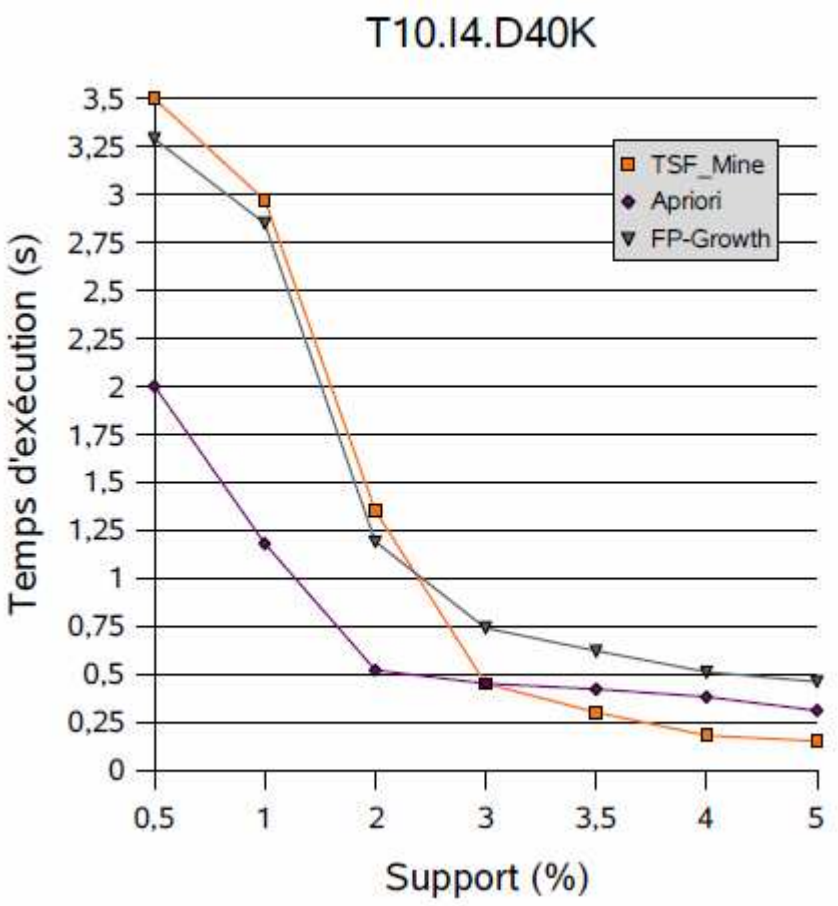


Figure 41. *Minsup vs Temps d'exécution pour T10.I4.D40K.*

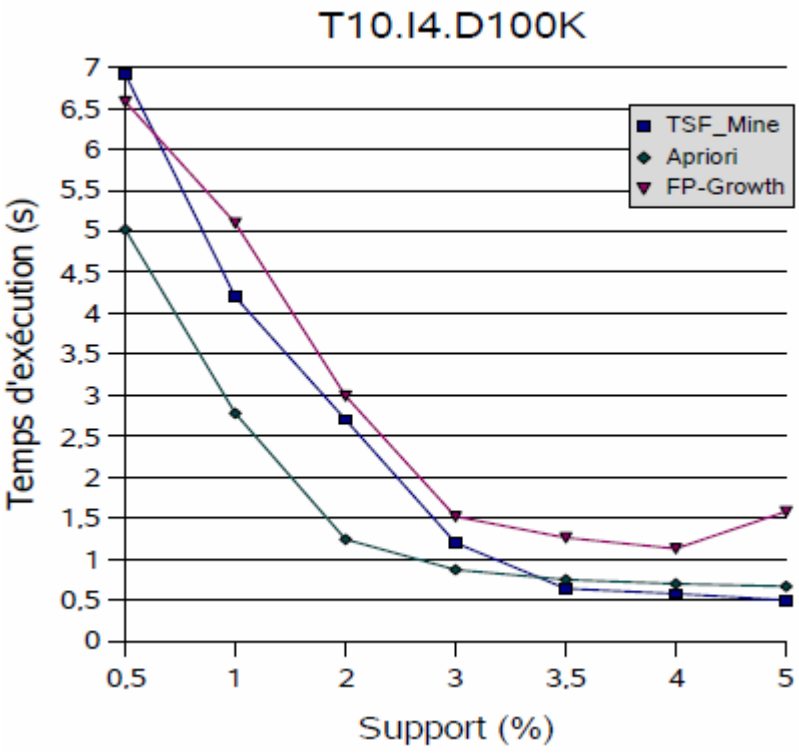


Figure 42. *Minsup vs Temps d'exécution pour T10.I4.D100K.*

Pour la base T10.I4.D100K et à part la valeur de support 0,5%, la *Figure 42* montre que l'algorithme TSF_Mine donne des temps d'exécution plus intéressants que ceux fournis par l'algorithme FP_Growth. Par contre, l'algorithme Apriori donne de meilleurs résultats que notre algorithme pour des supports inférieurs à 3,5%. Les temps d'exécution de TSF_Mine deviennent plus intéressants à partir de 3,5%.

IV.4.2. Résultats relatifs à la variante 2

Nous avons effectué deux séries d'expérimentation. Ces expérimentations ont été réalisées sur un PC portable équipé d'un processeur Intel Core 2 Duo, avec une vitesse d'horloge de 1,80 Ghz, un disque dur de 120 Giga et une mémoire RAM de 2 Giga. Les algorithmes ont été implémentés en langage C++. Les tests ont été effectués sur un ensemble de bases de transactions disponibles sur le site FIMI.

Les tests, relatifs à la première série d'expérimentation, comparent les temps d'exécution obtenus par notre algorithme ST-Mine et l'algorithme Apriori, optimisé par Bodon.

Deux ensembles de tests ont été réalisés lors de la seconde série d'expérimentation. Le premier a concerné l'impact du choix de la fonction de hachage sur le nombre de false drop et le second nous a permis de comparer notre modèle de représentation d'une base de transaction avec la structure FP-Tree. Nous nous sommes particulièrement intéressés à l'espace mémoire utilisé.

IV.4.2.1. Mesure du temps d'exécution de ST-Mine et Apriori

Lors de cette première série de tests, nous avons utilisé plusieurs bases de transactions pour comparer les performances de l'algorithme ST-Mine et l'algorithme Apriori. Nous avons considéré le temps d'exécution requis comme critère de comparaison.

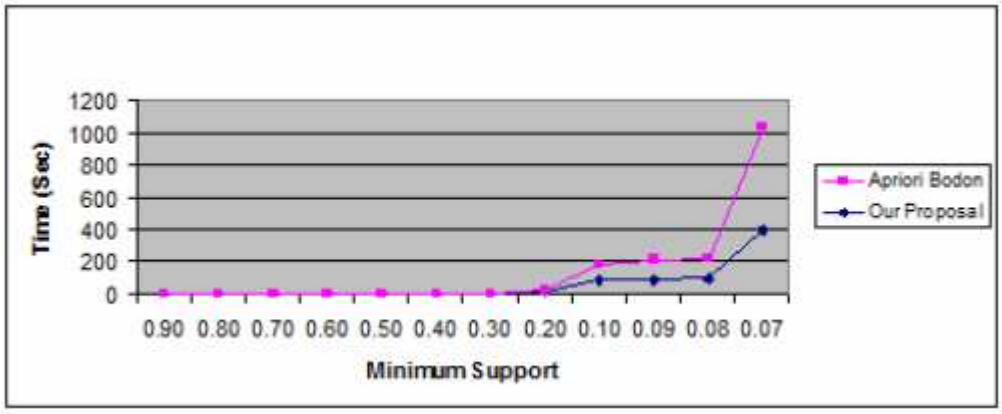


Figure 43. *Minsup vs Temps d'exécution pour Mushroom.*

Nous constatons sur la *Figure 43* que, pour la base Mushroom, les temps d'exécution sont identiques pour des supports ≥ 20%. Pour des valeurs de supports ≤ 20%, l'algorithme ST-Mine donne de meilleurs résultats.

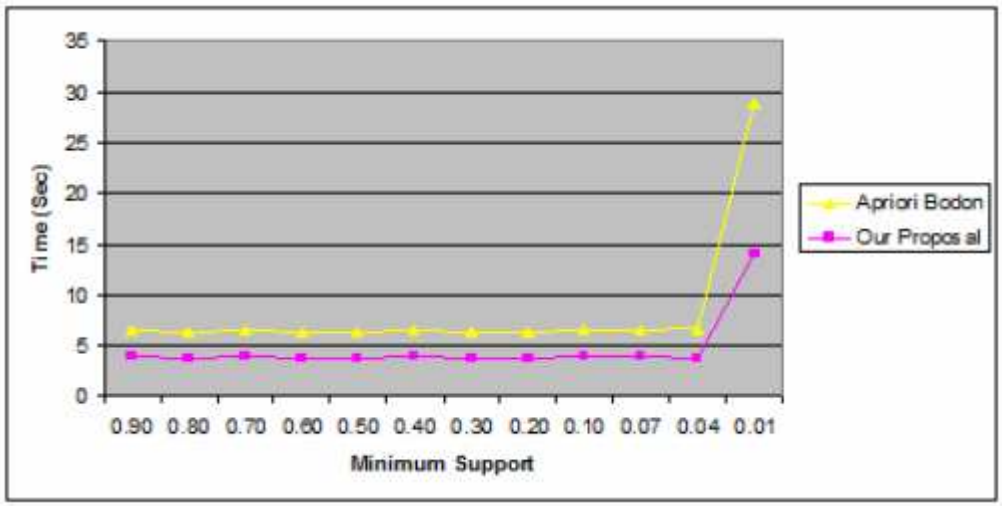


Figure 44. *Minsup vs Temps d'exécution pour Retail*

La *Figure 44* montre que les temps d'exécution fournis par St-Mine pour la base Retail sont également plus petits que ceux fournis par Apriori. Cette différence devient plus importante pour des supports à 4%.

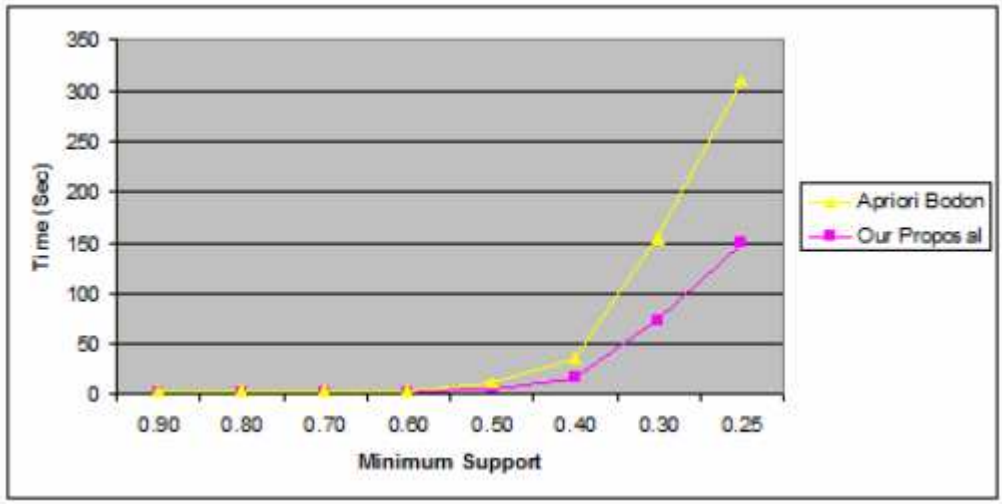


Figure 45. *Minsup vs Temps d'exécution pour Accidents.*

Nous remarquons sur la *Figure 45* que les temps d'exécution sont les mêmes pours des valeurs de supports comprises entre 50% et 90%. Cependant, nos résultats deviennent meilleurs pour des supports ≤ 50%.

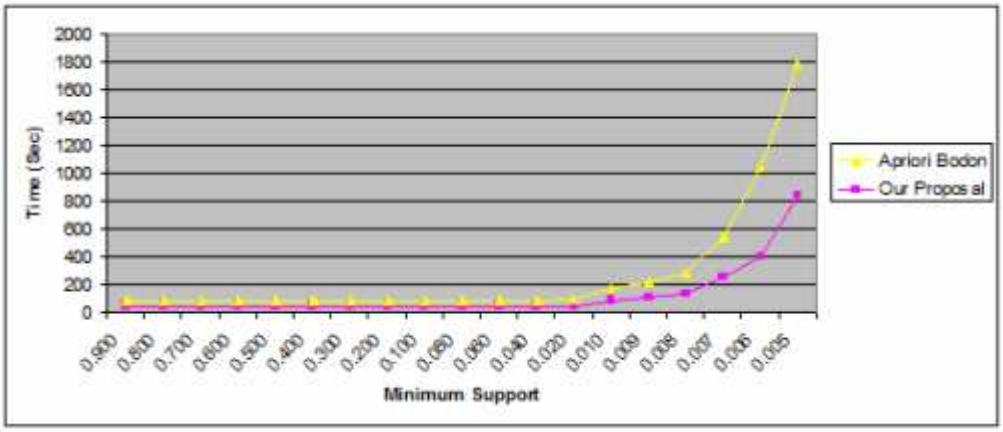


Figure 46. *Minsup vs Temps d'exécution pour Kosarak.*

Sur la *Figure 46*, nous remarquons que les temps d'exécution obtenus sont très proches les uns des autres. La différence devient visible pour des valeurs de supports très petites (≤ 1,5%).

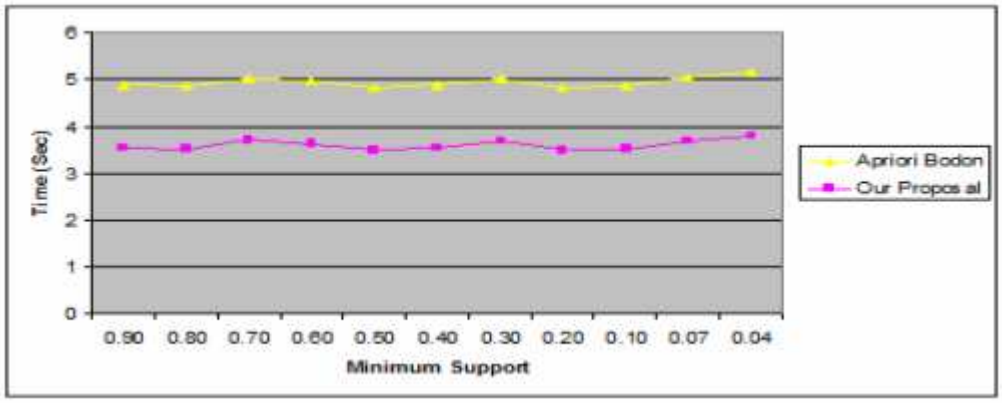


Figure 47. *Minsup vs Temps d'exécution pour T10I4D100K.*

La *Figure 47* montre que les temps obtenus sont plus ou moins linéaires et la différence entre les résultats de l'algorithme ST-Mine et ceux d'Apriori est pratiquement la même entre les deux algorithmes.

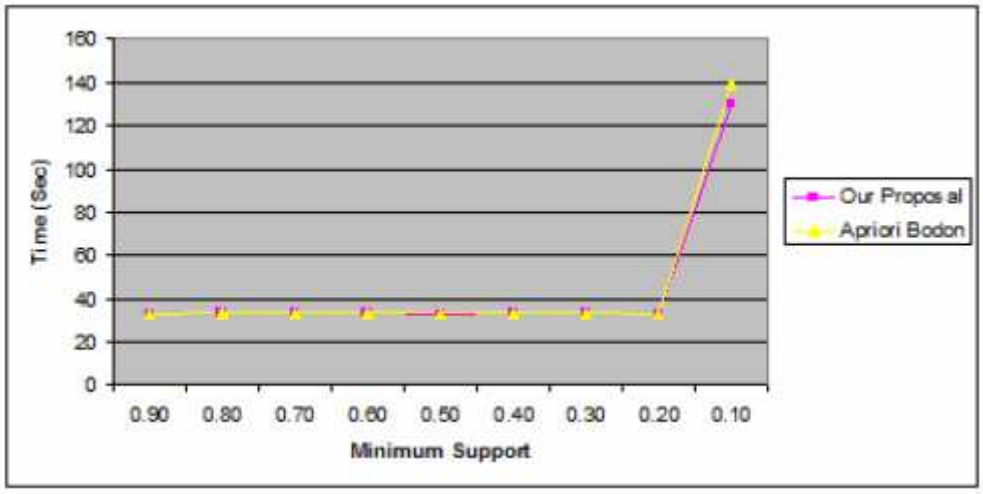


Figure 48. *Minsup vs Temps d'exécution pour T40I4D100K.*

Le seul cas où les résultats sont approximativement identiques est celui de la base T40I4D100K de la *Figure 48*. Nous constatons en effet que les deux algorithmes fournissent un temps linéaire avant d'augmenter brusquement pour des supports ≤ 20%.

En conclusion, nous diront que la série d'expérimentation, réalisée sur différents types de bases de transactions, a montré que notre algorithme ST-Mine est plus performant que l'algorithme Apriori.

La section suivante sera consacrée à l'étude de l'impact du choix de la fonction de hachage sur le nombre de false drop et à l'évaluation de notre algorithme ST-Mine, basé sur le modèle de représentation ST-Tree, et

l'algorithme FP_Growth, basé sur la structure FP-Tree, avec comme critère d'évaluation l'espace mémoire utilisé lors du processus d'extraction des itemsets fréquents.

IV.4.2.2. Impact de la fonction de hachage sur le taux de false drop

Nous allons utiliser, pour l'étude de l'impact du choix de la fonction de hachage, les notations suivantes:

I-Nbre*:* Nombre d'items distincts dans une base de transactions

I-Avg*:* I-Avg: Nombre moyen d'items dans une transaction

H-Fct*:* Fonction de hachage

Size-S*:* Taille d'une signature

A-W*:* Poids moyen d'une signature (Nombre de 1)

Size-B*:* Nombre de transactions dans une base de transactions

A-T*:* Nombre moyen de transactions par feuille de l'arbre

Database	Type	I-Nbre	I-Avg	Size-S	A-W	Size-B	A-T	H-Fct
Chess	Dense	75	37	128	37	3196	1	
Mushroom	Dense	119	23	128	23	8124	1	
T10I4D100K	Sparse	870	10	128	9	100000	1	Modulo
T40I10D100K	Sparse	942	39	128	34	100000	1	
Retail	Sparse	16470	10	128	9	88162	1	128
Accidents	Sparse	468	33	128	32	340183	1	
Chess	Dense	75	37	512	37	3196	1	
Mushroom	Dense	119	23	512	23	8124	1	
T10I4D100K	Sparse	870	10	512	10	100000	1	Modulo
T40I10D100K	Sparse	942	39	512	38	100000	1	
Retail	Sparse	16470	10	512	10	88162	1	512
Accidents	Sparse	468	33	512	33	340183	1	
Chess	Dense	75	37	1024	37	3196	1	
Mushroom	Dense	119	23	1024	23	8124	1	
T10I4D100K	Sparse	870	10	1024	10	100000	1	Modulo
T40I10D100K	Sparse	942	39	1024	39	100000	1	
Retail	Sparse	16470	10	1024	10	88162	1	1024
Accidents	Sparse	468	33	1024	33	340183	1	

Table 37. *Résultats de la fonction de hachage "Modulo"*

La Table 33 montre que, dans le plus mauvais cas, la fonction "Modulo 128" donne une signature de transaction dont 29% des bits sont égaux à "1". Par contre, la fonction "Modulo 512" fournit une signature contenant uniquement 7% de "1" et la fonction "Modulo 1024" 4% de "1".

Database	Type	I-Nbre	I-Avg	Size-S	A-W	Size-B	A-T
Chess	Dense	75	37	512	467	3196	1
Mushroom	Dense	119	23	512	395	8124	1
T10I4D100K	Sparse	870	10	512	237	100000	1
T40I10D100K	Sparse	942	39	512	465	100000	1
Retail	Sparse	16470	10	512	218	88162	1
Accidents	Sparse	468	33	512	455	340183	1

Table 38. *Résultats de la fonction de hachage "MD5+Modulo"*

Dans le cas de la fonction de hachage "MD5+Modulo" (Table 38), nous considérons le résultat fourni par MD5, MD5 fournissant un résultat alphanumérique de taille 64, sur lequel nous appliquons une fonction "Modulo 8" sur chaque caractère pour obtenir une signature de 8 bits par caractère. La concaténation de toutes les signatures donnent une signature de 512 bits. Nous constatons cependant que, dans le meilleur des cas, le poids moyen de la signature reste élevé (43%).

Nous constatons également que, dans les deux cas de figures, nous obtenons un nombre moyen de signatures par feuille égale à 1. Ces résultats montrent que le processus d'application de chacune des fonctions de hachage génère quasiment des signatures de transaction uniques par transaction. Ce qui signifie que quelque soit la base de transactions considérée (dense, éparse, réelle, synthétique), nous n'avons pas de false drop.

IV.4.2.3. Comparaison des performances de ST-Mine et FP-Growth

Pour pouvoir comparer les deux structures ST-Tree et FP-Tree, nous sommes dans l'obligation d'exécuter les deux algorithmes ST-Mine et FP-Growth. Le critère de comparaison considéré est l'espace mémoire utilisé lors du processus d'extraction des itemsets fréquents.

La *Table 39* illustre les résultats de l'exécution de ST-Mine en considérant différentes bases de transactions. Nous constatons que chacune des bases de transactions nécessite un espace mémoire fixe, ne dépendant d'aucun paramètre, même pas du support minimum.

Base de Transactions	Espace Mémoire (Kbytes)
Chess	75
Mushroom	190
T10I4D100K	7966
T40I10D100K	2089
Retail	2342
Accidents	1931

Table 39. *Espace mémoire utilisé par ST-Tree*

Par contre, l'espace mémoire occupé par FP-Tree dépend du support minimum fixé à priori par l'utilisateur et cet espace est différent selon la base de transactions considérée.

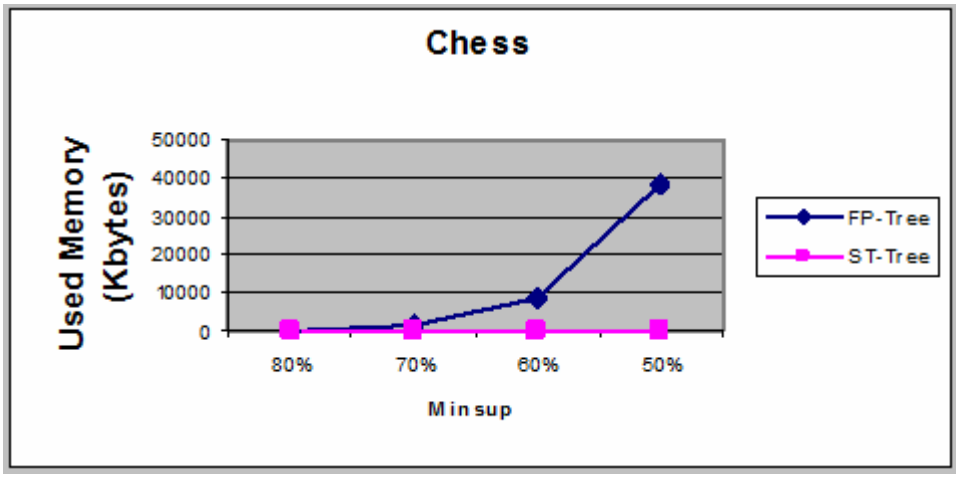


Figure 49. *ST-Tree vs FP-Tree pour Chess*

La *Figure 49* montre la mémoire utilisée pour la base de transactions Chess pour différentes valeurs de Minsup. Nous constatons qu'une différence apparaît à partir de supports ≤ 70%. Celle-ci devient très importante pour les supports ≤ 60%.

Par contre, nous remarquons, sur la *Figure 47*, que les résultats de FP-Tree pour la base Mushroom sont meilleurs dans le cas de supports ≥ 26%, ST-Tree occupant moins d'espace mémoire dans le cas de supports ≤ 26%.

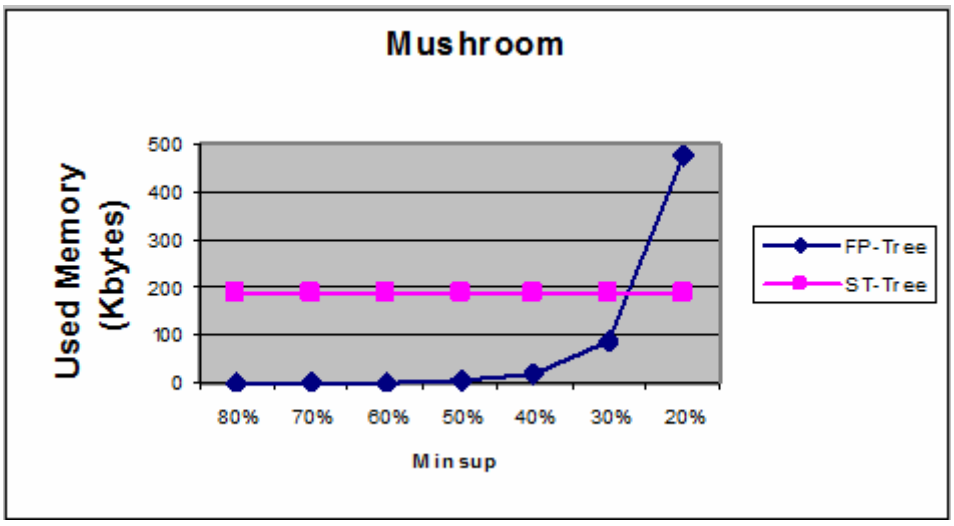


Figure 50. *ST-Tree vs FP-Tree pour Mushroom*

Les résultats illustrés par la *Figure 50* montrent que, pour la base *T10I4D100K*, FP-Tree occupe moins d'espace pour des supports ≥ *1,4%*. ST-Tree devient meilleur à partir de cette valeur de support.

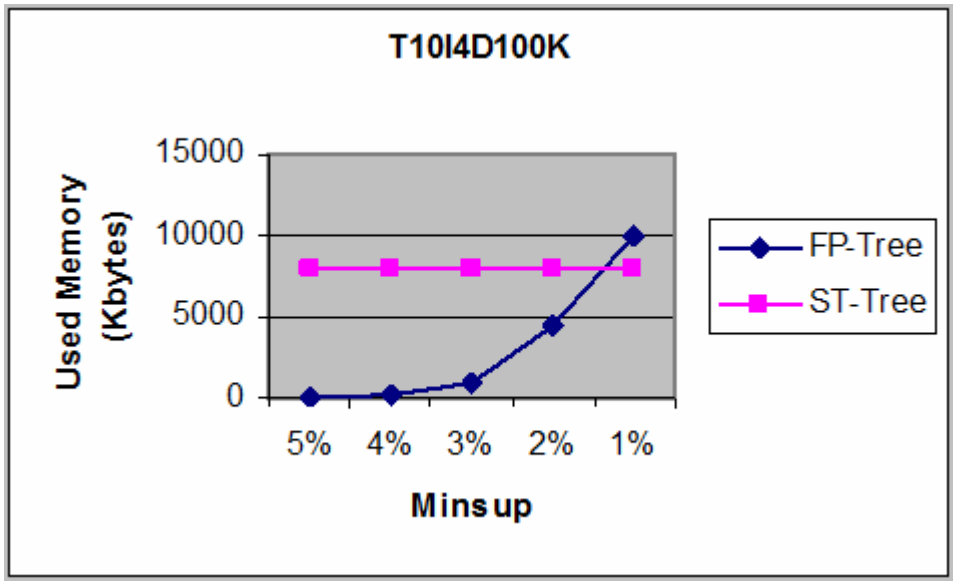


Figure 51. *ST-Tree vs FP-Tree pour T10I4D100K*

La base de transaction T40I4D100K occupe moins d'espace dans le cas de la structure ST-Tree par rapport à FP-Tree, et ce pout tous les supports ≤ 15%. La *Figure 49* montre que la différence devient encore plus évidente quand le support décroît.

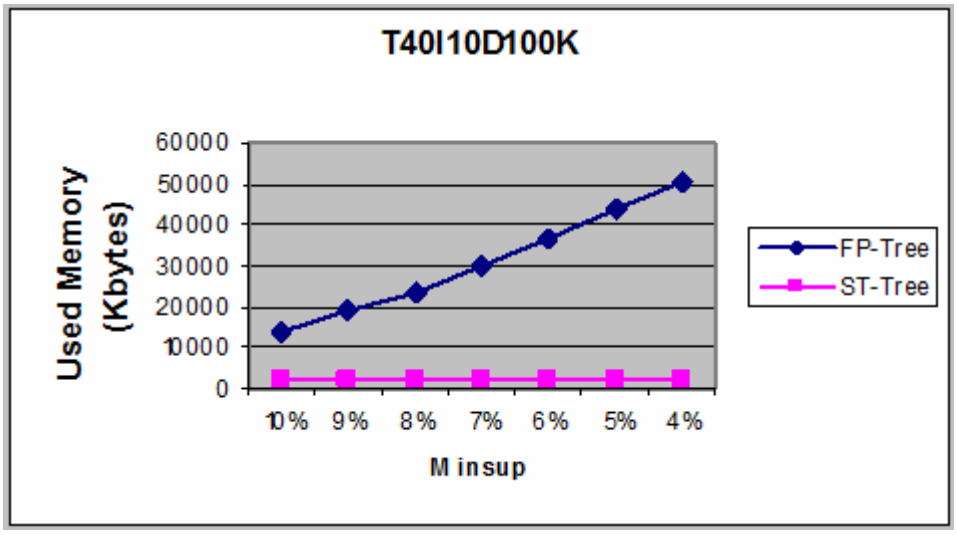


Figure 52. *ST-Tree vs FP-Tree pour T40I4D100K*

Nous notons les mêmes constatations dans le cas de la base de transactions Accidents. ST-Tree et FP-Tree occupent sensiblement le même espace mémoire pour les supports ≥ 55%, FP-Tree occupant plus d'espace pour des supports ≤ 55%.

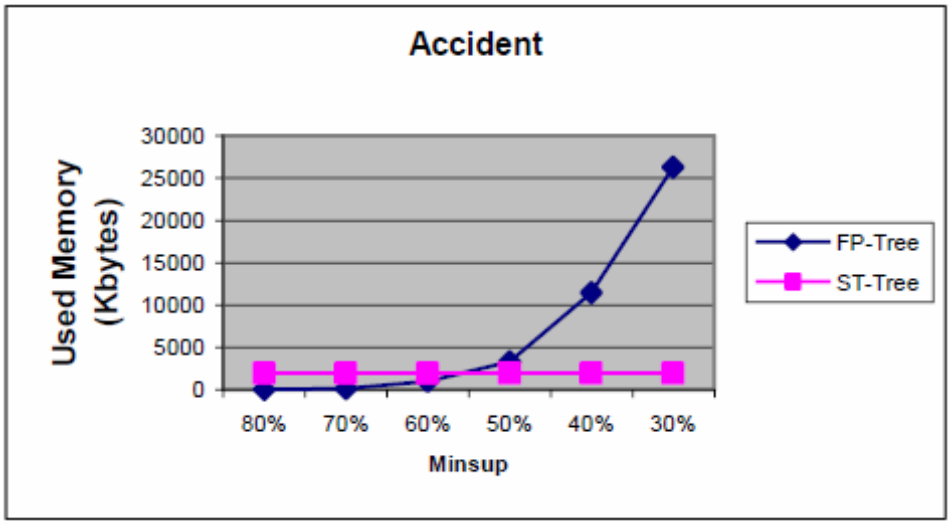


Figure 53. *ST-Tree vs FP-Tree pour Accident*

IV.5. Conclusion

Dans cette section, nous avons détaillé les architectures d'implémentation des deux variantes de notre modèle binaire et arborescent, utilisé dans le processus d'extraction des itemsets fréquents.

La variante 1 consistait à représenter une base de transactions à l'aide d'un arbre de signatures et d'un fichier de signatures, l'arbre de signature servant à rechercher la

signature d'un itemset candidat et le fichier de signature servant d'index permettant d'éliminer éventuellement les false drop et de calculer le support de l'itemset. Nous avons également appliqué deux processus d'optimisation, par filtrage et par condensation, pour diminuer le nombre de false drop et donc améliorer le temps d'exécution. Malgré le fait que cette approche ait donné lieu à des performances acceptables, le temps généré pour extraire les itemsets fréquents restait important. Nous avons remarqué également que la complexité de TSF-Mine était exponentielle.

La variante 2 est une amélioration de la variante 1. Elle consiste à n'utiliser que l'arbre de signatures dans le processus d'extraction des itemsets fréquents. Nous avons prouvé par les différentes expérimentations que cette approche génère un temps d'exécution inférieur à celui généré par l'algorithme Apriori de Bodon et ne nécessite que deux accès à la base des transactions, un accès pour construire l'arbre de signatures et un accès pour éliminer les transactions false drop. Nous avons également introduit un nouveau concept, le concept de support maximum possible pour un itemset candidat. Nous nous sommes basés sur ce support pour extraire les itemsets fréquents. Ce qui nécessite un seul accès à la base de transactions, accès permettant de construire l'arbre de signatures ST-Tree. Nous avons ainsi comparé les performances de l'algorithme ST-Mine et de l'algorithme FP-Growth en considérant l'espace mémoire occupé par l'arbre ST-Tree et l'arbre FP-Tree. Nous avons prouvé que ST-Tree avait deux avantages par rapport à FP-Tree:

1. Chaque base de transactions était représentée par un arbre ST-Tree unique. Alors que FP-Tree dépendait du support minimum choisi.
2. ST-Tree occupe moins d'espace mémoire que FP-Tree, spécialement pour de petits supports minimum.

CONCLUSION ET PERSPECTIVES

CONCLUSION ET PERSPECTIVES

L'Extraction de connaissances à partir de bases de données est définie comme un processus interactif d'extraction de connaissances à partir d'ensembles de données. A partir de ces ensembles, il s'agit d'extraire des connaissances nouvelles qui enrichissent les interprétations du champ d'application d'un expert, tout en fournissant des méthodes automatiques et semi-automatiques pour exploiter ces connaissances. Les techniques de fouille de données permettent de découvrir ces connaissances cachées dans les données. Dans le cadre de notre thèse, nous nous sommes intéressés particulièrement à la technique de génération des règles d'association, qui est très largement utilisée en fouille de données. Cette technique pose deux problèmes principaux: le nombre d'accès à la base de transactions et le temps que prend le processus d'extraction des connaissances.

Il est donc essentiel de minimiser le nombre d'accès à la base de transactions, ce qui conduirait à réduire le temps d'exécution du processus global de génération des règles d'association.

Comme contribution à ce problème de surcoût d'exécution, nous avons proposé un modèle de représentation compacte des transactions, basé sur des signatures binaires. L'ensemble des signatures qui composent ce modèle sont stockées dans un fichier appelé fichier de signatures. Nous avons ensuite représenté le fichier de signatures à l'aide d'un arbre de signatures afin qu'il puisse être exploité dans un processus d'extraction de connaissances. Cette proposition nous a permis de diminuer le nombre d'accès à la base de transactions, mais elle a généré d'autres problèmes, tels que la gestion des false drop (transactions sélectionnées mais non réellement référencées), la taille du fichier de signatures et la taille de l'arbre de signatures. En effet, l'algorithme relatif à cette approche adopte une stratégie "filtrer-et-affiner" qui permet de limiter le nombre d'itemsets à vérifier et de restreindre la vérification uniquement aux transactions pertinentes et non à l'ensemble des transactions. Même si les performances de l'algorithme TSF_Mine, que nous avons développé et expérimenté, se sont révélées encourageantes, des améliorations restent possibles. Elles concernent essentiellement l'impact du choix de la fonction de hachage et de la taille de la signature sur le taux de false drop, la complexité et le temps d'exécution de l'algorithme.

Dans le cadre de ces améliorations, nous avons proposé un nouveau modèle de représentation de la base de transactions qui ne contient plus que l'arbre de signatures représentant les transactions. Ainsi, seul l'arbre de signatures est utilisé dans le processus d'extraction des itemsets fréquents. Nous avons défini également un nouveau concept, à

savoir le concept de support maximum possible pour un itemset. Un seul accès à la base est nécessaire pour extraire l'ensemble des itemsets fréquents, en utilisant le support maximum. Cet accès permet de construire l'arbre et extraire les 1-itemsets fréquents. Le calcul du support réel d'un itemset nécessite un deuxième accès à la base. Une étude détaillée et globale de la complexité des algorithmes proposés a été faite pour chacune des deux variantes. Les algorithmes relatifs à la première variante (arbre et fichier de signatures) ont une complexité exponentielle, alors que la deuxième variante (arbre de signatures uniquement) a donné lieu à une complexité polynomiale.

Nous avons également proposé une architecture relative à l''implémentation de chacune des variantes ainsi qu'une description détaillée des étapes d'exécution des algorithmes associés.

Les résultats d'implémentation ont montré que notre approche donne de meilleures performances qu'une approche basée sur l'algorithme Apriori du point de vue nombre d'accès à la base et temps d'exécution. Nous avons également obtenu de meilleurs résultats que la structure FP-Tree, du point de vue espace de stockage. Cependant, le temps d'exécution, même s'il est très proche de celui de l'algorithme FP-Growth, reste perfectible.

A la suite des résultats que nous avons obtenus, plusieurs perspectives son envisageables.

Comme première perspective, nous envisageons de procéder à l'amélioration des performances de notre algorithme ST-Mine du point de vue temps d'exécution. Cette amélioration passe nécessairement par une exécution parallèle de ST-Mine. Des tests de parallélisation à l'aide des "threads" sont en cours.

La deuxième perspective concerne le type de base de données à explorer. Nous avons remarqué que certains algorithmes donnent de meilleurs résultats sur des bases denses et d'autres sur des bases éparses. Connaître le type d'une base peut donc s'avérer très utile et peut permettre de choisir l'algorithme adapté à la base de transactions à traiter. La définition de plusieurs mesures, notamment la notion de support maximum et l'utilisation de la structure ST-Tree, devrait nous permettre de dire si une base de transactions est dense ou éparse. Ce qui permettrait de choisir l'algorithme approprié qui donnerait les meilleures performances.

Comme dernière perspective, il nous semble qu'en utilisant notre modèle de représentation, il devient possible de diviser une base de transactions en "clusters". Cette architecture en "clusters" pourrait être exploitable pour distribuer ces clusters sur des nœuds d'une architecture répartie ou une architecture de type Cloud

BIBLIOGRAPHIE

BIBLIOGRAPHIE

1. S.R. Ahuja et C.S. Roberts. "An associative/parallel processor for partial match retrieval using superimposed codes". In ISCA'80: Proceedings of the 7th annual symposium on Computer Architecture, pp. 218–227, New York, NY, USA, 1980.
2. D. L. Lee and C. Leng. "Partitioned signature files: design issues and performance evaluation". ACM Transactions on Information Systems Journal (TOIS), Volume 7 Issue 2, April 1989, pp. 158 – 180, New York, NY, USA.
3. E. Shanthi, "Applying SD-Tree for Object-Oriented Query Processing". Informatica 33, 2009, pp. 177-187.
4. M. H. Margahny and A. A. Mitwaly. "Fast Algorithm for Mining Association Rules". AIML 05 Conference, 19-21 December 2005, pp 36-40, CICC, Cairo, Egypt.
5. R. Agrawal et R. Srikant. "Fast algorithms for mining association rules in large databases". In VLDB'94 : Proceedings of the 20th International Conference on Very Large Data Bases, pp. 487–499, San Francisco, CA, USA, 1994.
6. C. Borgelt et R. Kruse. "Induction of association rules: Apriori implementation". In 15th Conference on Computational Statistics, pp. 395–400, Physica Verlag, Heidelberg, Germany, 2002.
7. S. Brin, R. Motwani, J.D. Ullman, et S. Tsur. "Dynamic itemset counting and implication rules for market basket data". In SIGMOD'97: Proceedings of the 1997 ACM SIGMOD International Conference on Management of Data, May 11-15, pp. 255–264, New York, NY, USA, 1997.
8. F. Bodon. A trie-based apriori implementation for mining frequent item sequences. In OSDM '05: Proceedings of the 1st International Workshop on Open Source Data Mining, pp. 56–65, New York, NY, USA, 2005.
9. P. Mahatthanapiwat. "Flexible Searching for Graph Aggregation Hierarchy". In Proceedings of the World Congress on Engineering 2010, Vol I WCE 2010, June 30 - July 2, 2010, London, U.K.
10. Y. Chen. "Building Signature Trees into OODBs". In Journal of Information Science and Engineering 20, pp. 275-304 (2004).
11. Y. Chen. "Signature File and Signature Tree". In Information Processing Letters 82 pp. 213–221 (2002).
12. D. L. Lee, Y. M. Kim and G. Patel. "Efficient Signature File Methods for Text Retrieval". In Journal IEEE Transactions on Knowledge and Data Engineering, Volume 7 Issue 3, pp. 423-435, June 1995.
13. R. Agrawal, T. Imieliński and A. Swami. "Mining association rules between sets of items in large databases". In Proceeding SIGMOD '93 Proceedings of the 1993 ACM SIGMOD International Conference on Management of Data, pp. 207-216, New York, NY, USA, 1993.
14. R. Srikant and R. Agrawal. "Mining Generalized Association Rules". In Proceedings of the 21th International Conference on Very Large Databases, pp. 407-419, September 11-15, Zurich, Switzerland, (1995).

15. W. W. Chang and H. J. Schek. "A signature access method for the Starburst database system". In Proceedings of the 15th International Conference on Very Large Databases, August 22-25, pp. 145-153, Amsterdam, Netherlands 1989.
16. Ya. Chen and Yi. Chen. "Signature File Hierarchies and Signature Graphs: a New Index Method for Object-Oriented Databases". In Proceeding of the 2004 ACM symposium on applied computing, pp. 724-728, Nicosia, Cyprus, March 14-17, 2004.
17. E.Ramaraj and N.Venkatesan. "Bit Stream Mask-Search Algorithm in Frequent Itemset Mining". In European Journal of Scientific Research, Vol. 27 (2), 2009, pp. 286-297.
18. G. Grahne and J. Zhu. "Fast Algorithms for Frequent Itemset Mining Using FP-Trees". In IEEE Transactions on Knowledge and Data Engineering, Vol. 17 (10), October 2005, pp. 1347- 1362.
19. A. Pietracaprina and D. Zandolin. "Mining Frequent Itemsets using Patricia Tries". In Proceedings of the ICDM 2003 Workshop on Frequent Itemset Mining Implementations, 19 December 2003, Melbourne, Florida, USA.
20. C. Faloutsos. "Access methods for text". ACM Computer Survey, Vol. 17(1) ,pp. 49–74, 1985.
21. C. Faloutsos, R. Lee, C. Plaisant, et B. Shneiderman. "Incorporating string search in a hypertext system: User interface and signature file design issues". In Hypermedia Journal, Vol. 2, pp. 183–200, 1990.
22. G. Gardarin, P. Pucheral, et F. Wu. "Bitmap based algorithms for mining association rules". In Proceeding of the 14ème Journées Bases de Données Avancées, pp. 26-30 octobre 1998, Hammamet, Tunisie.
23. R. P. Gopalan et Y. G Sucahyo. "ITL - mine: Mining frequent itemsets more efficiently". In Proceedings of the 1st International Conference on Fuzzy Systems and Knowledge Discovery (FSDK'02), pp. 167 – 171, November 18-22, 2002, Orchid Country Club, Singapore.
24. J. Han, M. Chen, et P. S. Yu. "Data mining: An overview from database perspective". In IEEE Transactions on Knowledge and Data Engineering, Vol. 8(6) pp. 866 – 883, 1996.
25. B. Goethals. "Survey on frequent pattern mining". Technical Report, HIIT Basic Research Unit, Department of Computer Science, University of Helsinki, Finland 2003.
26. J. Hipp, U. Guntzer, et G. Nakhaeizadeh. "Mining association rules: Deriving a superior algorithm by analyzing today's approaches". In PKDD'00: Proceedings of the 4th European Conference on Principles of Data Mining and Knowledge Discovery, pp. 159–168, September 13-16, 2000, Lyon, France.
27. B. Lan, B. C. Ooi and K. L. Tan. "Efficient indexing structures for mining frequent patterns". In Proceedings of 18th International Conference on Data Engineering, pp. 453 – 462, 26 february 26 – March, 2002, San Jose, CA, USA.
28. I. E. Shanthi and R. Nadarajan. "An Index Structure for Fast Query Retrieval in Object Oriented Data Bases Using Signature Weight Declustering". In Information Technology Journal, Volume 8(3), pp. 275-283, 2009.

29. D. Dervos, Y. Manolopoulos and P. Linardis. "Comparison of signature file models with superimposed coding". In Information Processing Letters, Volume 65, Issue 2, January 1998, pp. 101-106.

30. J. Han, J. Pei, Y. Yin and R. Mao. "Mining Frequent Patterns without Candidate Generation: A Frequent-Pattern Tree Approach". In Data Mining and Knowledge Discovery Journal, Vol. 8(1), pp. 53–87, January 2004.

31. J. Han, J. Pei and Y. Yin. "Mining Frequent Patterns without Candidate Generation". In Proceedings of the ACM SIGMOD International Conference on Management of Data (SIGMOD'00), pp. 1-12, Dallas, TX, USA, May 15 - 18, 2000.

32. Y. Ishikawa, H. Kitagawa and N. Ohbo. "Evaluation of signature files as set access facilities in oodbs". In Proceedings of the ACM SIGMOD International Conference on Management of Data (SIGMOD'93), pp 247-256, Washington, D.C., May 26-28, 1993.

33. B. Lan, B. C. Ooi, and K. L Tan. "Efficient indexing structures for mining frequent patterns". In Proceeding of 18th International Conference on Data Engineering (ICDE'02), pp. 453-462, February 26-March 01, San Jose, California, 2002.

34. Y. Liu, J. Pisharath, W. Liao, G. Memik, A. Choudhary and P. Dubey. "Performance Evaluation and Characterization of Scalable Data Mining Algorithms". In Proceedings of the 16th International Conference on Parallel and Distributed Computing and Systems (PDCS), November 9 – 11, MIT Cambridge, USA, 2004.

35. A. Mueller. "Fast sequential and parallel algorithms for association rule mining: a comparison". Technical Report, University of Maryland at College Park College Park, MD, USA, August 1995.

36. Jr. R. J. Bayardo. "Efficiently mining long patterns from databases". In Proceeding of the ACM SIGMOD International Conference on Management of Data (SIGMOD '98), pp. 85–93, June 2-4, Seattle, Washington, USA, 1998.

37. W. J. Frawley, G. Piatetsky-Shapiro and C. J. Matheu. "Knowledge Discovery in Databases: An Overview". In AI Magazine, Volume 13, Number 3, pp. 57-70, 1992.

38. U. Fayyad, G. Piatetsky-Shapiro and P. Smyth. "From Data Mining to Knowledge Discovery in Databases". In AI Magazine Journal, Volume 17, pp. 37-54, 1996.

39. G. Piatetsky-Shapiro. "Knowledge Discovery in Databases: 10 years after". In ACM SIGKDD Explorations Newsletter, Volume 1, Issue 2, pp. 59-61, January 2000, New York, NY, USA.

40. M. J. Norton. "Knowledge Discovery in Databases". School of Library and Information Science, The University of Southern Mississippi, Librarytrends, Vol. 48(1), pp. 9-21**,** Summer 1999.

41. J. H. Su and W. Y. Lin. "CBW: An Efficient Algorithm for Frequent Itemset Mining". In Proceedings of the 37th Hawaii International Conference on System Sciences (HICSS'04), pp. 1-9, 5-8 January 2004, Big Island, HI, USA.

42. A. Savasere, E. Omiecinski and S. Navathe. "An efficient algorithm for mining association rules in large databases". In Proceeding of the 21th International Conference on Very Large Data Bases (VLDB '95), pp. 432-444, Zurich, Swizerland, 1995.

43. M. Song and S. Rajasekaran. "A transaction mapping algorithm for frequent itemsets mining". IEEE Transactions on Knowledge and Data Engineering, Volume 18, Number 4, pp. 472–481, April 2006.

44. R. Srikant and R. Agrawal. "Mining Generalized Association Rules". In Proceedings of the 21st VLDB Conference, pp. 407-419, Zurich, Swizerland, 1995.

45. M. C. Tseng1 and W. Y. Lin. "Maintenance of Generalized Association Rules with Multiple Minimum Supports". In Intelligent Data Analysis Journal, Volume 8, Issue 4, pp. 417 – 436, September 2004, Amsterdam, Netherlands.

46. Y. C. Lee, T. P. Hong and W. Y. Lin. "Mining Association Rules with Multiple Minimum Supports using Maximum Constraints". In International Journal of Approximate Reasoning, Volume 40, Number 1-2, pp. 44-54, July 2005.

47. B. Liu, W. Hsu and Y. Ma. "Mining Association Rules with Multiple Minimum Supports". In *ACM SIGKDD International Conference on Knowledge Discovery& Data Mining* (*KDD-99*), pp. 337 – 341, August 15-18, 1999, San Diego, CA, USA.

48. I. E. Shanthi and R. Nadarajan. "Applying SD-Tree for Object-Oriented Query Processing". In Informatica 33**,** pp. 177-187, 2009.

49. M. J. Zaki, S. Parthasarathy, M. Ogihara and W. Li. "New algorithms for fast discovery of association rules". In Proceeding of KDD'97: 3rd International Conference on Knowledge Discovery and Data Mining, pp. 283–286, August 14–17, Newport Beach, California, USA 1997.

50. F. Bodon. "A Fast APRIORI Implementation". In Proceedings of the IEEE ICDM Workshop on Frequent Itemset Mining Implementations, pp. 16-25, November 2003, Melbourne, Florida, USA.

51. G. Liu, H. Lu, J. X. Yu, W. Wang and X. Xiao. "AFOPT: An Efficient Implementation of Pattern Growth Approach". In Proceedings of the IEEE ICDM Workshop on Frequent Itemset Mining Implementations, pp. 115-124, November 2003, Melbourne, Florida, USA.

52. W. A. Kosters and W. Pijls. "APRIORI, A Depth First Implementation". In Proceedings of the IEEE ICDM Workshop on Frequent Itemset Mining Implementations, pp. 108-114, November 2003, Melbourne, Florida, USA.

53. Christian Borgelt. "Efficient Implementations of Apriori and Eclat". In Proceedings of the IEEE ICDM Workshop on Frequent Itemset Mining Implementations, pp. 26-34, November 2003, Melbourne, Florida, USA.

54. M. El-Hajj and O. R. Zaïane. "COFI-tree Mining: A New Approach to Pattern Growth with Reduced Candidacy Generation". In Proceedings of the IEEE ICDM Workshop on Frequent Itemset Mining Implementations, pp. 135-144, November 2003, Melbourne, Florida, USA.

55. G. Grahne and J. Zhu. "Efficiently Using Prefix-trees in Mining Frequent Itemsets". In Proceedings of the IEEE ICDM Workshop on Frequent Itemset Mining Implementations, pp. 125-134, November 2003, Melbourne, Florida, USA.

56. A. Pietracaprina and D. Zandolin. "Mining Frequent Itemsets using Patricia Tries". In Proceedings of the IEEE ICDM Workshop on Frequent Itemset Mining Implementations, pp. 145-154, November 2003, Melbourne, Florida, USA.

57. A. Gyenesei and J. Teuhola. "Probabilistic Iterative Expansion of Candidates in Mining Frequent Itemsets". In Proceedings of the IEEE ICDM Workshop on Frequent Itemset Mining Implementations, pp. 45-51, November 2003, Melbourne, Florida, USA.

58. C. L., S. Orlando, P. Palmerini, R. Perego and F. Silvestri. "kDCI: a Multi-Strategy Algorithm for Mining Frequent Sets". In Proceedings of the IEEE ICDM Workshop on Frequent Itemset Mining Implementations, pp. 98-107, November 2003, Melbourne, Florida, USA.

59. C. Borgelt. "Recursion Pruning for the Apriori Algorithm". In Proceedings of the IEEE ICDM Workshop on Frequent Itemset Mining Implementations, November 1, 2004, Brighton, UK.

60. M. Anandhavalli, M. K. Ghose and K. Gauthaman. "Fast Association Rule Mining Algorithm for Spatial Gene Expression Data". In Global Journal of Computer Science and Technology, Vol. 9(5), pp. 36-40, January 2010.

61. A. M. J. Md. Z. Rahman, P. Balasubramanie and P. Venkata Krihsna. "A Hash based Mining Algorithm for Maximal Frequent Itemsets using". In Linear Probing INFOCOMP Journal of Computer Science, volume 8, number 1, pp. 14-19, 2009.

62. K. Bicakci, G. Tsudik and B. Tung. "How to construct optimal one-time signatures". In Computer Networks: The International Journal of Computer and Telecommunications Networking, Vol. 43(3), pp. 339–349, October 2003, New York, NY, USA.

63. J. S. Park, M. S. Chen and P. S. Yu. "An effective hash-based algorithm for mining association rules". In Proceedings of the 1995 ACM SIGMOD International Conference on Management of Data, pp. 175 – 186, New York, NY, USA, 1995.

64. N. Mamoulis, D.W Cheung and W. Lian. "Similarity Search in Sets and Categorical Data Using the Signature Tree". In Proceedings of 19th International Conference on Data Engineering, pp. 75 – 86, 5-8 March 2003, Bangalore, India.

65. P. Bagwell. "Ideal Hash Trees". In Infoscience, Technical report, October 2001.

66. M. Kontaki, Y. Manolopoulos and A. Nanopoulos. "Compressing Large Signature Trees". In Lecture notes in computer science (LNCS 2798), pp. 163–177, 2003.

67. A. Kent, R. Sacks-Davis and K. Ramamohanarao. "A Superimposed Coding Scheme Based on Multiple Block Descriptor Files for Indexing Very Large Data Bases". In Proceedings of the 14th VLDB Conference, pp. 351-359, Los Angeles, California 1988.

68. J. S. Park, M. S. Chen and P. S. Yu. "Using a Hash-Based Method with Transaction Trimming_Database Scan Reduction for Mining Association Rules". In IEEE Transactions on Knowledge and Data Engineering Journal, Vol. 9(5), pp. 813-825, September/October 1997.

69. S. Orlando, P. Palmerini and R. Perego. "Enhancing the Apriori Algorithm for Frequent Set Counting". In Proceedings of the Third International Data Warehousing and Knowledge Discovery, DaWaK 2001, pp 71-82, September 5-7, Munich, Germany, 2001.

70. S. Hong, S. Kapralov, H. K. Kim and D. Y. Oh. "Uniqueness of some optimal superimposed codes". In Problems of Information Transmission Journal, Volume 43, Issue 2, pp. 113 – 123, June 2007, New York, NY, USA.

71. S. Yekhanin. "Some new constructions of optimal superimposed designs". In Proceedings of International Conference on Algebraic and Combinatorial Coding Theory, pp. 232-235, 1998.
72. S. Joshi and R. C. Jain. "A Dynamic Approach for Frequent Pattern Mining Using Transposition of Database", In Proceeding of the International Conference on Communication Software and Networks (ICCSN'10), pp 498-501, Singapore, 26-28 February 2010.
73. Ya. Chen, Yi. Chen."On the Signature Tree Construction and Analysis", IEEE Transactions on Knowledge and Data Engineering, vol. 18, Issue 9, pp. 1207-1224, September 2006.
74. C. Faloutsos. "Signature Files: Design and Performance Comparaison of Some Signature Extraction Methods", ACM Sigmod Record, Volume 14, Issue 4, pp. 63 – 82, May 1985.
75. S. Joshi, R S Jadon and R. C. Jain. "An Implementation of Frequent Pattern Mining Algorithm using Dynamic Function", International Journal of Computer Applications, Volume 9, n°9, pp. 37–41, November 2010.
76. S. Zhang, J. Zhang, X. Zhu, Z. Huang. "Identifying Follow-Correlation Itemset-Pairs", In Proceeding of the Sixth International Conference on Data Mining (ICDM), pp. 765-774, Hong Kong, 18-22 December 2006.
77. R. Mikut and M. Reischl. "Data mining tools", Data Mining Knowledge Discovery, Volume 1, pp. 431-443, September/October 2011.
78. J. Han, H. Cheng, D. Xin, X. Yan. "Frequent pattern mining: current status and future directions", Data Mining Knowledge Discovery, Vol. 15(1), pp. 55–86, 2007.
79. S. Folleville et S. Wauquier. "Data Mining supervisé". http://www.ceremade.dauphine.fr/~touati/RUGBY_www/Compte Rendu.pdf
80. B. Espinasse. "Introduction aux méthodes de Fouille de données". http://www.lsis.org/espinasseb/Supports/DWDM-2009/8-MethodesFouille-2009-4p.pdf
81. Y. Grim. "Conception d'un entrepôt de données (Data Warehouse)". http://grim.developpez.com/cours/businessintelligence/concepts/conception-datawarehouse/
82. E. ZIYATI. "Optimisation de requêtes OLAP en Entrepôts de Données Approche basée sur la fragmentation génétique". Thèse de doctorat, Faculté des Sciences Rabat Maroc, Mai 2010. http://toubkal.imist.ma/bitstream/handle/123456789/6969/THESE_ZIYATI.pdf;jsessionid=D5665C1B36E305A9957AF8F321C28460?sequence=1
83. J. F. Desnos. "Entrepôts de données". Cours Master ICA: Ingénierie de la Cognition, de la Création, et des Apprentissages, Universités de Grenoble. http://prevert.upmf-grenoble.fr/SpecialiteIHS/GP/Exocours.pdf
84. K. Zeitouni. "Fouille de données (Data mining)", Cours de Master 2, Université de Versailles Saint-Quentin, 2009.

    http://www.e-campus.uvsq.fr/claroline/backends/download.php?url=L0lfRGF0YU1pbmluZ19JU1RZLnBkZg%3D%3D&cidReset=true&cidReq=FD

85. P. Preux. "Fouille de données, notes de cours", Université de Lille 3, 2008. http://www.grappa.univ-lille3.fr/~ppreux/Documents/notes-de-cours-de-fouille-de-donnees.pdf
86. G. Saporta. Cours intitulé "Introduction au Data Mining et à l'apprentissage statistique", Conservatoire National des Arts et Métiers (CNAM). http://cedric.cnam.fr/~saporta/DM.pdf
87. C. Faloutsos, R. Chan. "Fast Text Access Methods for Optical and Large Magnetic Disks: Designs and performance Comparaison", 14th International Conference on Very Large Data Bases, pp. 280-293, Los Angeles, California, USA, August 29 - September 1, 1988.
88. M. Benelhadj, K. Arour, M. Boufaïda, Y. Slimani. "A New Compact Structure to Extract Frequent Itemsets, International Journal of Database Theory and Application, Vol. 4(4), December 2011, pp. 25-42, ISSN: 2005-4270
89. M. Benelhadj, K. Arour, M. Boufaïda, Y. Slimani. "A Binary Based Approach for Generating Association Rules", International Conference on Data Mining (DMIN'2011), pp 140-146, July 18-21, 2011, Las Vegas, Nevada, USA.
90. M. Benelhadj, M. Boufaïda and K. Arour. "Mining Frequent Itemsets with Tree Signatures", IADIS European Conference on Data Mining (ECDM'2010), pp. 163-165, July 28-29, 2010, Freiburg, Deutschland.
91. K. Arour, G. Mrabet, M. Benelhadj et Y. Slimani. "Arbre de signatures pour l'extraction des itemsets fréquents". In Proceedings of the Eighth International Symposium on Programming and Systems (ISPS'2007), pp. 1-10, May 7-9, 2007, Alger Algeria.
92. FIMI repository, http://fimi.cs.helsinki.fi/data
93. R. Rakotomalala. "Les règles d'Association, market data analysis ou L'analyse du panier de la ménagère". http://eric.univ-lyon2.fr/~ricco/cours/slides/regles_association.pdf
94. H. Toivonen. "Sampling Large Databases for Association Rules". In Proceeding of the 22nd VLDB Conference, pp. 134-145, September 3-6, 1996, Mumbai(Bombay), India.