Maintenance des motifs fermés fréquents dans un flux

Dame Samb[1,2], Petko Valtchev[1], Robert Godin[1]

[1]Dept d'Informatique, UQAM, CP 8888
Succ. Centre-Ville, Montréal H3C 3P8, Canada
valtchev.petko@uqam.ca, godin.robert@uqam.ca

[2]Université de Thiès, BP A967, Thiès, Sénégal.
dsamb@univ-thies.sn

RÉSUMÉ. Étant l'une des tâches les plus importantes de la fouille d'associations, le calcul des motifs fermés fréquents a été le sujet de nombreuses études et plusieurs algorithmes, travaillant essentiellement avec les bases (statiques) de transactions traditionnelles ont été produits. Aujourd'hui, il existe une forte demande pour la conception d'algorithmes de fouille travaillant avec des données de type flux ($data\ stream$) du fait de leur importance sans cesse croissante. Dans cet article, nous proposons de faire un état de l'art sur les algorithmes de maintenance de motifs fermés fréquents dans un flux de données. Nous donnons une description des algorithmes les plus représentatifs du domaine puis, nous les analysons sous divers angles.

ABSTRACT. As one of the most important tasks of association rule mining, frequent closed itemsets mining has been the subject of many studies and several algorithms which mainly take static databases as input have been produced. Today we have a strong demand for the proposal of mining algorithms over dynamic databases or data streams, resulting of the ever increasing prominence of this type of data. In this paper, we propose a survey of algorithms maintaining frequent closed itemsets over a data stream.

MOTS-CLÉS : fouille de flux, incrémentalité, fouille motifs fermés fréquents.

KEYWORDS : stream mining, incrementality, frequent closed pattern mining.

1. Introduction

La fouille de motifs fréquents ou de leurs ensembles réduits (générateurs minimaux, motifs fermés et motifs maximaux) constitue un problème fondamental dans la fouille d'associations. Ces dernières années, l'expansion rapide observée dans diverses sphères d'activités comme le Web, le commerce électronique et les télécommunications a donné un nouvel élan à la fouille de motifs fréquents. En effet, on assiste à une importante production de données qui se présentent sous la forme de flux ($stream$, en anglais) avec comme corollaire, l'apparition de nouvelles applications telles que la détection de fraudes, l'analyse de trafic et la détection d'intrusions dans un réseau de données, l'analyse des activités et l'organisation des sites web. Dans ces applications, la connaissance des motifs fréquents ou de leurs ensembles réduits est d'une grande utilité.

Les données de type flux sont de nature dynamiques, i.e., elles sont continuellement mises à jour. Elles arrivent à une grande vitesse et de manière désordonnée. Ces contraintes rendent extrêmement difficile l'extraction de motifs fréquents, d'une part, à cause de l'impossibilité de faire tenir toutes les données en mémoire et, d'autre part, à cause de l'explosion en terme de coût (calcul et E/S) que l'utilisation récurrente des algorithmes traditionnels (par lot) de fouille occasionnerait. Partant de l'observation que lors d'une mise à jour des données de la base, les motifs fréquents changent rarement de statut (en passant de l'état infréquent à l'état fréquent ou l'inverse), on peut se servir des informations tirées des précédentes opérations de fouille pour réduire les calculs à effectuer. Cette approche est dite incrémentale et pour des mises à jour de taille raisonnable, il est plus avantageux d'y recourir. Puisque, avec les algorithmes par lot, tout le travail (génération de candidats, calcul des fréquences, etc) est à refaire en cas de mise à jour, alors que parfois, il suffit juste d'actualiser les fréquences des motifs déjà calculés.

Les performances des algorithmes incrémentaux dépendent fortement de la taille des données fournies en entrée. Si pour des données de taille réduite, toutes les opérations peuvent se dérouler en mémoire, tel n'est pas le cas pour de grands volumes de données. En effet, il devient impraticable de stocker en mémoire toute l'information utile à la fouille des motifs lorsque les données ont théoriquement une taille infinie, comme c'est le cas dans un flux. Dans un tel contexte, les algorithmes verront naturellement leurs performances se dégrader. Cette contrainte amène à considérer non pas toutes les données d'un flux mais uniquement celles qui se trouvent dans un intervalle (de taille fixe) qu'on appelle fenêtre. Cette approche suscite beaucoup d'intérêts d'autant plus que dans certaines applications, l'attention est portée sur les données les plus récentes. En fouille de motifs fréquents dans une fenêtre de flux, le principal défi reste le choix des données à maintenir. En effet en plus des motifs fréquents qui sont l'objet la fouille, on doit choisir et conserver des informations issues des précédentes opérations de fouille, pouvant aider dans le futur, à diminuer la quantité de travail à effectuer. Ces informations supplémentaires doivent être assez informatives pour permettre de calculer les motifs fréquents dans la fenêtre et être de taille réduite pour pouvoir tenir en mémoire.

Le reste de l'article est organisé comme suit : Dans la section 2, nous présentons un certain nombre de concepts et de définitions importantes en fouille de motifs fermés fréquents dans un flux. Dans la section 3, nous présentons quelques uns des algorithmes les plus connus du domaine. Dans la section 4, nous analysons sous divers angles les algorithmes présentés. Nous terminons cet article par une conclusion qui résume l'ensemble du travail et par une proposition de perspectives de recherche.

2. Préliminaires

2.1. Fouille d'associations

Une application typique de la fouille d'associations est l'analyse du panier de la ménagère ($Basket\ Market\ Analisys$ [1]) où les produits achetés représentent les attributs et les transactions, les achats effectués dans un commerce. Il existe une formulation mathématique élégante du problème que nous reproduisons dans les lignes qui suivent.

Soit $\mathcal{I}$ = {i_i, i_2, ...i_m} un ensemble de m attributs distincts appelés $items$. Un $itemset$ *X* est un sous-ensemble d'items, i.e., $X \subseteq \mathcal{I}$. Une $transaction\ t = (TID, X)$ est un couple où *TID* est un identifiant unique associé à t et *X* un itemset. La figure 1 (à gauche) montre l'exemple d'une base de transactions ($\mathcal{D}$) où $\mathcal{I}$ = $\{a, b, c, d, e\}$ et les itemsets des transactions sont $\{abc, acd, bcde, bcd, abcde\}$.

Le $support$ absolu (resp. relatif) d'un itemset X, noté $sup_{\mathcal{D}}(X)$ (ou *sup(X)* s'il n y a pas d'ambiguité), est le nombre (resp. pourcentage) de transactions dans la base $\mathcal{D}$ contenant X. Dans notre exemple, *sup(a) = 60%*, *sup(ab) = 40%* et *sup(bc) = 80%*.

Un itemset, X est fréquent si $sup(X) \geq minsup$ où $minsup$ est un seuil fixé par l'utilisateur. La figure 1 (à droite) montre tous les motifs fréquents de la base de transactions $\mathcal{D}$ (à gauche), pour $minsup = 3/5$.

TID	Items
1	a, b, c
2	a, c, d
3	b, c, d, e
4	b, c, d
5	a, b, c, d, e

Itemset	support	Itemset	support
a	3/5	b	4/5
c	5/5	d	4/5
ac	3/5	bc	4/5
bd	3/5	cd	4/5
bcd	3/5		

Figure 1. *TDB $\mathcal{D}$ (à gauche) et ses motifs fréquents (à droite) pour $minsup = 3/5$.*

Une $association$ ou règle d'association est une implication de la forme $X \Rightarrow Y$, où $X \subset \mathcal{I}$, $Y \subset \mathcal{I}$ et $X \cap Y = \emptyset$. Deux mesures sont définies pour cette règle associative : le $support$ et la $confiance$. Le support de règle est donné par $sup(X \cup Y)$ et sa confiance par $sup(X \cup Y)/sup(X)$, i.e., la probabilité conditionnelle qu'une transaction contienne Y sachant qu'elle contient X. Une règle est fréquente si son support est supérieur ou égal à $minsup$ et elle est sûre si sa confiance est supérieure ou égale à $minconf$ où $minsup$ et $minconf$ sont les seuils fixés par l'utilisateur. La fouille d'associations consiste à trouver toutes les règles fréquentes et sures d'une base de transactions donnée. Le problème peut être décomposé en deux sous problèmes : i) Trouver tous les itemsets fréquents ; ii) Dériver à partir des itemsets fréquents toutes les règles fréquentes et sures.

2.2. Notions de flux et de fenêtre

Un flux ($stream$) de données $\mathcal{D} = (T_1, \ldots, T_i, \ldots)$ est une séquence continue de transactions classées selon leur ordre d'arrivée. Cette contrainte rend impossible le maintien de toutes les données d'un flux en mémoire. C'est pourquoi dans la fouille d'associations dans un flux, on travaille avec une fenêtre de données, $F_k = < T_i, T_{i+1}, \ldots, T_j >$ et on considère uniquement les données qui s'y trouvent à l'instant t. Il existe deux modèles de fenêtres : les fenêtres coulissantes ($sliding\ window$) et les fenêtres à point de repère ($landmark\ window$). Dans une fenêtre coulissante, le nombre de transactions (taille de

la fenêtre) est fixé à l'avance. Ce qui fait que lorsque la fenêtre a atteint sa taille maximale, l'ajout de nouvelles transactions dans la fenêtre entraine la suppression d'un nombre égal de transactions. Une fenêtre à point de repère autorise uniquement l'ajout de nouvelles transactions ainsi, dans la fenêtre courante, il y a toutes les transactions ajoutées à la base de la première à la plus récente. Les ajouts et suppressions dans la fenêtre peuvent se faire individuellement ou par lot et à n'importe quelle position. Si on désigne par T_k, la dernière transaction ajoutée, la fenêtre courante F_k dans le cas d'une fenêtre à point de repère est $F_k = < T_1, T_2, \ldots, T_k >$. Dans le cas d'une fenêtre coulissante, la fenêtre courante F_k est donnée par $F_k = < T_{k+1-w}, T_{k+2-w}, \ldots, T_k >$, où w désigne la taille de la fenêtre. La figure 2 montre un exemple des deux types de fenêtre. Pour la fenêtre coulissante (à gauche), $w = 4$ et la première fenêtre (F_1) est constituée des quatre premières transactions de la base. Quand on passe à la deuxième fenêtre (F_2), la première transaction est supprimée et la cinquième ajoutée.

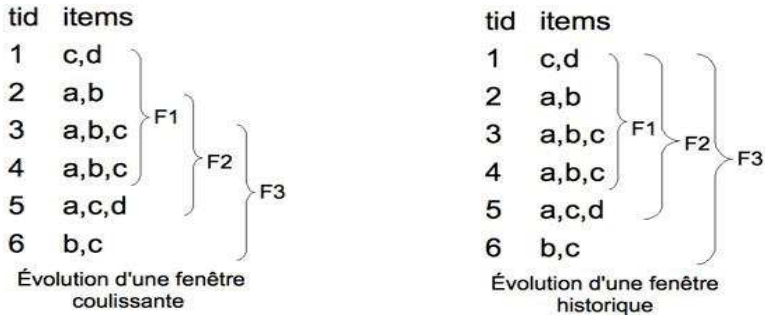


Figure 2. *Evolution d'une fenêtre coulissante et d'une fenêtre à point de repère.*

Dans une fenêtre F_k, le $support$ absolu (resp. relatif) d'un itemset X, notée *sup(X)* est le nombre (resp. pourcentage) de transactions contenant X dans F_k. Dans la fenêtre F_1 de l'exemple, *sup(a) = 75%*, *sup(cd) = 25%* et *sup(bc) = 50 %*. X est un *itemset fréquent* dans la fenêtre F_k si son support est supérieur ou égal à *minsup*, où *minsup* est un seuil fixé par l'utilisateur. Dans la fenêtre F_1 de l'exemple, pour $minsup = 2$, l'ensemble des itemsets fréquents est : F = {a :3, b :3, c :3, ab :3, ac :2, bc :2, abc :2}.

2.3. Itemsets fermés fréquents

Dans cette section, nous définissons les notions de contexte de fouille de données, de connexion de Galois, fermeture de Galois et d'itemsets fermés fréquents [2, 3].

Contexte : Un contexte de fouille de données est un triplet $\mathcal{K} = (\mathcal{O}, \mathcal{A}, \mathcal{I})$ où $\mathcal{O}$ est un ensemble fini d'objets, $\mathcal{A}$, un ensemble fini d'attributs et $\mathcal{I} \subseteq \mathcal{O}$ x $\mathcal{A}$, une relation binaire (d'incidence) entre les objets et les attributs. Par extension, nous appellerons une base de transaction, une contexte de fouille. Exemple dans la figure 1, $\mathcal{A} = \{a, b, c, d, e\}$, $\mathcal{O} = \{1, 2, 3, 4, 5\}$ et $(2, c) \in \mathcal{I}$.

Connexion de Galois : Soit un contexte $\mathcal{K} = (\mathcal{O}, \mathcal{A}, \mathcal{I})$, on définit deux fonctions f et g résumant les liens entre les sous-ensembles d'objets et les sous-ensembles d'attributs induit par $\mathcal{I}$.

– $f : \mathcal{P}(\mathcal{O}) \rightarrow \mathcal{P}(\mathcal{A}), f(X) = X' = \{a \in \mathcal{A} | \forall o \in X, oIa\}$

– $g : \mathcal{P}(\mathcal{A}) \rightarrow \mathcal{P}(\mathcal{O}), g(Y) = Y' = \{o \in \mathcal{O} | \forall a \in Y, oIa\}$

La fonction f associe à un ensemble d'objets, tous les attributs partagés tandis que g est la fonction duale sur les attributs. Par exemple, pour le contexte de la figure 1, $f(135) = bc$ et $g(bcd) = 345$. Les fonctions f et g induisent une connexion de Galois entre $\mathcal{P}(\mathcal{A})$ et $\mathcal{P}(\mathcal{O})$.

Fermeture de Galois : Les fonctions composites $f \circ g(Y) = Y''$ définie dans $\mathcal{P}(\mathcal{A})$ et $g \circ f(X) = X''$ définie dans $\mathcal{P}(\mathcal{O})$ sont des opérateurs de fermeture de Galois.

Itemset fermé fréquent : Un itemset, X, est fermé si $X = X''$, et il est fréquent si $sup(X) \geq minsup$.

3. Algorithmes de fouille de motifs fermés dans un flux

Dans cette section, nous décrivons un certain nombre d'algorithmes de fouille de motifs fermés fréquents dans un flux qui, à notre avis constituent un échantillon assez représentatif des différentes approches que l'on trouve dans la littérature.

3.1. L'algorithme Moment

$Moment$ [4, 5] est, à notre connaissance, le premier algorithme proposé pour la fouille de motifs fermés fréquents dans une fenêtre coulissante pour des données de type flux. C'est un algorithme incrémental qui utilise un arbre de préfixe appelé CET (Closed Enumeration Tree) qui stocke toute l'information nécessaire pour déterminer à chaque instant les motifs fermés fréquents. Chaque nœud n_I du CET représente un itemset I qui appartient à l'une des catégories suivantes : les *infréquents* (représentés sous forme d'elipse en pointillé) qui sont les nœuds infréquents du CET dont les parents sont des nœuds fréquents , les *peu prometteurs* (représentés sous forme de rectangle en pointillé) qui sont les nœuds fréquents (mais pas fermé) du CET dont la fermeture est leur prédécesseur dans le CET, les *intermédiaires* (représentés sous forme d'elipse en trait plein) qui sont les nœuds fréquents (mais pas fermé) du CET dont la fermeture est un descendant et les *fermés fréquents* (représentés sous forme de rectangle en trait plein). Les nœuds infréquents et peu prometteurs sont des feuilles dans le CET, car leur sous-arbre ne peuvent pas contenir d'itemsets fermés fréquents.

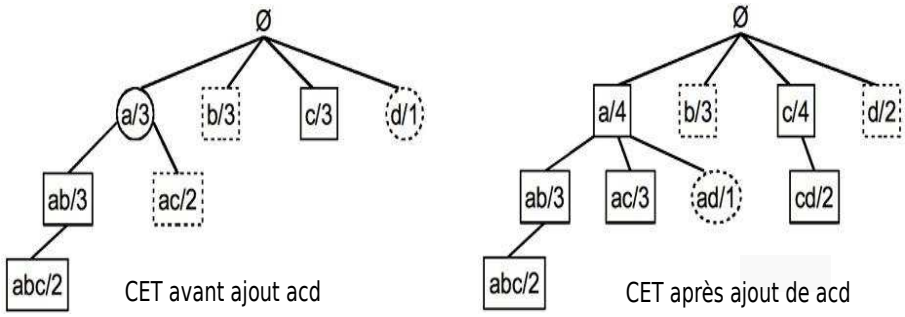


Figure 3. *CETs de la base (fenêtre) F_1(fig 2) avant et après l'ajout de acd ($minsup = 2$).*

Lorsqu'une transaction est ajoutée, pour chaque nœud en relation avec la transaction, le support, et éventuellement le statut sont mis à jour. Les changements de statut concernent les nœuds infréquents qui peuvent devenir fréquents et générer de nouveaux nœuds et les nœuds fréquents non fermés (peu prometteur et intermédiaire) qui peuvent

devenir fermés ou intermédiaires. Par exemple, lorsque la transaction acd est ajoutée au CET de la figure 3 (à gauche), le support et le statut des itemsets inclus dans la transaction sont mis à jour. Ainsi, $c/3$ devient $c/4$ et reste fermé ; $a/3$ devient $a/4$ et n'étant plus contenu dans un descendant de même support, il devient fermé (changement de statut et de support pour a) ; $ac/2$ devient $ac/3$ et n'étant plus contenu dans un nœud prédécesseur de même support, il devient fermé (changement de statut et de support pour ac) ; $d/1$ devient $d/2$ (fréquent), son sous-arbre doit être regénéré et il doit être combiné avec ses frères gauches, ce qui donne les nœuds ad et cd. ad est un nœud infréquent alors que cd est fermé. À noter que le nœud b n'est pas concerné par les modifications. Les mises à jour apportées donnent le CET à droite de la figure 3.

Lorsqu'une transaction est supprimée, pour chaque nœud en relation avec la transaction, le support, et éventuellement le statut sont mis à jour. Les changements de statut concernent les nœuds fréquents qui peuvent devenir infréquents ou non prometteurs. Ils peuvent également passer du statut de fermé à intermédiaire. Par exemple, lorsque la transaction cd est supprimée du CET (à gauche) de la figure 4, le support et le statut des itemsets inclus dans la transaction sont mis à jour. Ainsi, $c/4$ devient $c/3$ et étant inclus dans $ac/3$ qui le précède, il est marqué comme nœud peu prometteur ; $d/2$ devient $d/1$ (infréquent) et son sous-arbre de même que les nœuds obtenus par combinaison avec ses frères gauche sont elagués. Ainsi, ac et cd sont supprimés de l'arbre. Les autres nœuds ne sont pas concernés par les modifications. Les mises à jour apportées donnent le CET à droite de la figure 4.

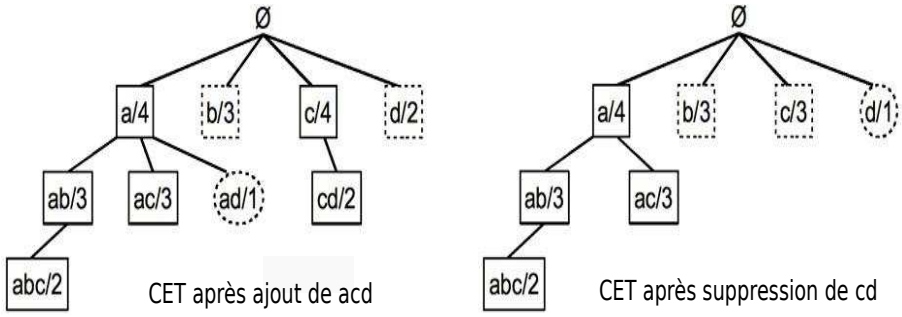


Figure 4. *CETs de la base (fenêtre) F_1(fig 2) après l'ajout de acd et la suppression de cd.*

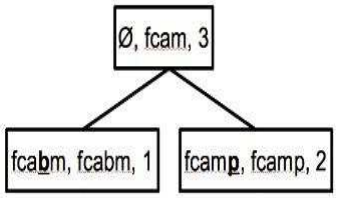
3.2. L'algorithme NewMoment

À la suite de $Moment$, d'autres algorithmes de la même famille ont été proposés [6, 7, 8]. Ces algorithmes concentrent leurs efforts sur les stratégies à mettre en œuvre pour réduire au minimum, l'information nécessaire à la fouille, qui doit être stockée en mémoire. Bien que réduisant l'espace mémoire utilisé, ils n'ont pas reussi à améliorer de manière significative les performances de $Moment$. $NewMoment$ [7, 8] est de la même famille que $Moment$. Il utilise un vecteur de bit pour représenter les items de la fenêtre et leur support ce qui permet d'avoir une représentation compacte des données et facilite les calcul de support. $NewMoment$ utilise également un arbre de préfixe comme arbre d'énumération des itemsets. Dans cet arbre, on ne maintient que les motifs fermés fréquents et les items (avec leur support sous forme de vecteurs de bit) de la fenêtre courante. L'arbre d'énumération (NewCET) est construit à l'aide d'une procédure qui visite en profondeur tous les itemsets de l'arbre. Un nœud fils n_J est obtenu en ajoutant un

nouveau item à I tel que $I <_L J$. Lorsqu'une transaction est supprimée, tous les vecteurs de bit représentant les supports des items concernés sont décalés à gauche et lorsqu'une transaction est ajoutée, le bit le plus à droite du vecteur est mis 0 si la transaction ne contient pas l'item et à 1 dans le cas contraire.

3.3. L'algorithme GC-Tree

$GC\text{-}Tree$ [11, 12] exploite une relation d'ordre total sur les itemsets de l'espace de recherche. L'usage d'une technique de saut de fermeture basée sur cet ordre total permet de détecter efficacement les duplications dans les calculs de fermeture. Ainsi chaque itemset fermé a un générateur unique et il n'est généré qu'une seule fois. L'algorithme fonctionne avec une fenêtre coulissante et les itemsets fermés sont stockés dans une structure de données (arbre) en mémoire appelée GC-Tree (Generator and frequent Closed itemsets Tree). Chaque nœud du $GC\text{-}Tree$ a le format suivant : $< gen, eitem, Clos >$ où gen est le générateur de la fermeture, $eitem$ l'item utilisé pour étendre le générateur et $Clos$ l'itemset correspondant au nœud. Les noeuds de l'arbre sont ordonnées lexicographiquement et pour chaque nœud, il y a un chemin spécifique partant de la racine. La figure 5 montre un exemple de CG-Tree.



TID	Items
1	f, c, a, m, p
2	f, c, a, m, p
3	f, c, a, b, m
4	f, b
5	c, b, p

Figure 5. *GC-Tree (à gauche) obtenu pour la fenêtre <* t_1, t_2, t_3*> de la base* $\mathcal{D}$ *(à droite).*

Pour l'ajout d'une nouvelle transaction, l'algorithme examine chaque nœud de l'arbre. Si la transaction existait dans la base, il n y a pas de nouveaux nœuds à insérer dans l'arbre, il suffit simplement de mettre à jour le support des sous-ensembles (itemsets) de la transaction. Si la transaction est nouvelle dans la base et si le générateur du nœud courant n'est pas un de ses sous-ensembles, on appelle une procédure $CheckSubTree$ qui visite les nœuds du sous-arbre pour calculer leur intersection avec la nouvelle transaction. Les noeuds du sous-arbre peuvent potentiellement donner des fermés qu'il faudra insérer à l'endroit approprié s'ils ne sont pas dans l'arbre ou mettre à jour leur support s'ils y sont. Si le générateur du nœud courant est inclus dans la transaction, on calcule l'intersection (du fermé avec la transaction) qui peut s'agir d'un itemset modifié (intersection identique au fermé) ou d'un nouveau itemset (intersection différente du fermé). Si on est en présence d'un itemset modifié, on procède à une mise à jour du support et on appelle récursivement la procédure d'ajout sur les fils du nœud courant dont le générateur inclus dans la transaction et la procédure $CheckSubTree$ sur les fils du nœud courant dont le générateur n'est pas inclus dans la transaction. Si on est en présence d'un nouveau itemset, un nouveau nœud est crée. Ce noeud à le même itemset fermé que son géniteur (le noeud à partir duquel il a été obtenu) mais ils ont un générateur différent. et qui sera sont son parent dans l'arbre. Les fils du géniteur qui préservent l'ordre (par rapport au générateur du nouveau nœud) vont être détachés et rattachés au nouveau nœud. Le géniteur devient parent du nouveau nœud, son itemset fermé prend comme nouvelle valeur, la valeur de l'intersection et on procède à une mise à jour de son support (s'il y a lieu) et on appelle

récursivement la procédure d'ajout sur les fils du noeud géniteur.
Par exemple pour ajouter la transaction $t_4(fb)$, on commence par la racine $< \emptyset, fcam, 3 >$. Comme t_4 n'existait pas dans la base et que le générateur de la racine est un de ses sous-ensembles ($\emptyset \subseteq fb$), on calcule l'intersection entre le fermé de la racine $fcam$ et la transaction t_4. On obtient un nouveau fermé ($f = fcam \cap fb \neq fcam$). On crée un nouveau $< fc, fcam, 3 >$. Les fils du géniteur $< fcabm, fcabm, 1 >$ et $< fcamp, fcamp, 2 >$ préservent l'ordre par rapport au nouveau, ils vont être détachés et devenir des fils du nouveau. Le géniteur (la racine) devient parent du nouveau, son itemset fermé prend comme nouvelle valeur, la valeur de l'intersection (f) et son support passe à 4 ($< \emptyset, f, 4 >$) et on appelle récursivement la procédure d'ajout sur les fils du nœud géniteur. $< fc, fcam, 3 >$, le fils du géniteur a un générateur fc qui n'est pas un sous-ensemble de la transaction fb donc on visite les nœuds de son sous-arbre. La première intersection donne un nouveau itemset ($fb = fcabm \cap fb \neq fcabm$) qui va être inséré dans l'arbre. La deuxième intersection donne l'itemset ($f = fcamp \cap fb \neq fcamp$) qui existe déjà avec un support à jour. Le résultat de l'insertion de la transaction t_4 est donnée par la figure 6.

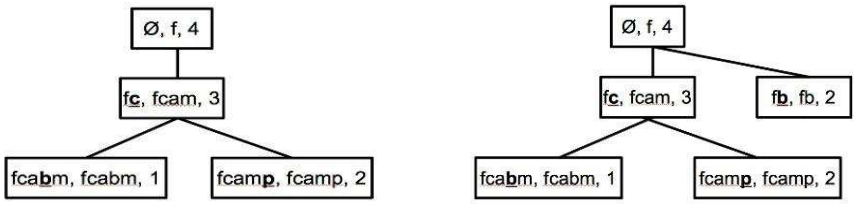


Figure 6. *Étapes de l'insertion de la transaction $t_4(fb)$ dans le GC-Tree de la figure 5.*

3.4. L'algorithme CFI-Stream

$CFI\text{-}Stream$ [9] est un autre algorithme incrémental pour la fouille de motifs fermés fréquents dans un flux avec fenêtre coulissante. Mis à part les motifs fermés et leur support, il ne maintient aucune autre information. Il utilise une structure de données en mémoire appelée DIU-Tree (DIrect Update Tree) pour représenter les motifs fermés. Le DIU-Tree est en réalité un arbre dont les nœuds respectent un ordre lexicographique. Au niveau k de l'arbre, on trouve les k-itemsets fermés fréquents. Chaque nœud de l'arbre contient un itemset fermé fréquent, son support et les liens vers son nœud-père et ses nœuds-fils. Lorsqu'une transaction est ajoutée ou supprimée, l'arbre est maintenu incrémentalement en utilisant les informations sur les fermés qui s'y trouvent déja. Pour certains nœuds de l'arbre, il faudra alors vérifier s'ils restent fermés ou non. Cette vérification se fait à la volée en faisant un seul parcours de la base.

Pour l'ajout d'une nouvelle transaction t, les motifs qui ne sont pas ses sous-ensembles ne changent pas de statut, i.e., s'ils étaient fermés, ils le restent et s'ils ne l'étaient pas, ils ne le deviennent pas. Pour les sous-ensembles de t, deux cas de figure peuvent se présenter :

1) t était présente dans la base : t et ses sous-ensembles gardent leur statut.

2) t n'était pas présente dans la base : t est un nouveau fermé et les fermetures de ses sous-ensembles présents dans la base sont obtenues par intersection des anciennes avec t.

Pour la suppression d'une transaction t, les motifs qui ne sont pas ses sous-ensembles ne changent pas de statut. Pour les sous-ensembles de t, deux cas de figures peuvent se présenter :

1) t n'était pas unique dans la base : les sous-ensembles de t ne changent pas de statut.

2) t était unique dans la base : les sous-ensembles non fermés ne changeront pas de statut mais ceux qui étaient fermés peuvent ne plus l'être.

3.5. L'algorithme CloStream

$CloStream$ [10] est un algorithme qui vient améliorer $CFI\text{-}Stream$ en partant du constat que la génération de tous les sous-ensembles de la transaction ajoutée peut se révéler extrêment couteuse particulièrement pour les bases denses. Comme $CFI\text{-}Stream$, *CloStream* ne maintient que les itemsets fermés. Pour cela, deux structures (table de hachage) sont utilisées. Une première table, $ClosedTable$ est reservée aux itemsets fermés où chaque enregistrement représente l'information sur un itemset fermé. Il y a trois champs dans cette table : cid qui est un identifiant attribué à l'itemset, CI qui est la valeur de l'itemset et $support$ qui représente le support de l'itemset. La seconde table, $CidList$ sert d'index aux items de la base. En effet, pour pouvoir retrouver facilement les itemsets fermés en relation avec la transaction ajoutée, on se sert de cette table qui associe à chaque item i la liste des itemsets fermés qui sont ses sur-ensembles. Chaque enregistrement de la table comporte deux champs : $item$ qui représente la valeur de l'item et $cidset$ qui contient les cid des itemsets fermés associés à l'item. La figure 7 montre un exemple de base de transaction (à gauche) ainsi que les ClosedTable (au centre) et CidList (à droite) associés. $CloStream$ se sert également (temporairement) d'une troisième table, où les itemsets obtenus par intersection entre les itemsets fermés de la base courante et la transaction ajoutée sont enregistrés et où leur vrai support est calculé avant leur insertion dans la première table.

TID	Items
1	c, d
2	a, b
3	a, b, c
4	a, b, c
5	a, c, d

Cid	CI	Sup	Cid	CI	Sup
0	$\emptyset$	0	1	cd	2
2	ab	3	3	abc	2
4	c	4	5	acd	1
6	a	4	7	ac	3

Item	CidSet
a	2, 3, 5, 6, 7
b	2, 3
c	1, 3, 4, 5, 7
d	1, 5

Figure 7. *Un TDB (à gauche), le ClosedTable (au centre) et le CidList (à droite) associés.*

L'ajout d'une nouvelle transaction comporte deux phases. Dans la première phase, on cherche (par intersection) tous les itemsets qui doivent être mis à jour, on les met dans la table temporaire avec le cid de leur fermeture (dans l'ancienne base). Dans la seconde phase, on met à jour leur support puis les tables ClosedTable et CidList. Par exemple pour ajouter la transaction bc, on commence par placer l'itemset bc (il est fermé) dans la table temporaire avec 0 comme cid de sa fermeture. Puis en se servant de la table CidList, on détermine les fermés qui doivent être utilisés dans les intersections. Les items b et c ont respectivement pour Cidset, les ensembles $\{2, 3\}$ et $\{1, 3, 4, 5, 7\}$ dont la fusion donne l'ensemble $\{1, 2, 3, 4, 5, 7\}$. Cet ensemble correspond aux itemsets fermés $\{cd, ab, abc, c, acd, ac\}$. La première intersection est $c = cd \cap bc$, qu'on place dans la table temporaire avec 1 comme cid de sa fermeture. Ensuite c'est $b = ab \cap bc$ qu'on place

dans la table temporaire avec 2 comme cid de sa fermeture. Ensuite $bc = abc \cap bc$ est déjà dans la table temporaire donc, des fermés ayant permis de le générer, il aura le cid de celui ayant le plus grand support. Le cid de la fermeture de bc sera donc 3. Le même travail est fait pour les fermés c, acd et ac. Dans la seconde phase, on met à jour le support des intersections de la table temporaire (avant de les insérer dans la table ClosedTable) et la table CidList.

4. Analyse et discussion

Dans cette section, nous analysons les algorithmes sous divers angles : les types de fenêtre et d'opérations supportées, la qualité des résultats, les données maintenues, la représentation et le maintien des données de la base, le conteneur des itemsets fermés, la technique de calcul des fermetures, les performances, les forces et les faiblesses.

Types de fenêtre et d'opérations supportées : Tous les algorithmes étudiés supportent les opérations d'ajout et de suppression, i.e., un modèle de fenêtre coulissante sauf $CloStream$ qui ne supporte que l'ajout, i.e., un modèle de fenêtre à point de repère. Qu'il s'agisse d'un ajout ou d'une suppression, une seule transaction est concernée.

Qualité des résultats : Les algorithmes étudiés retournent un résultat complet, i.e., tous les itemsets fermés fréquents sont présents. $NewMoment$ et $Moment$ retournent un résultat exact, i.e., uniquement les itemsets fermés fréquents tandis que $GC\text{-}Tree$, $CFI\text{-}Stream$ et $CloStream$ retournent en plus des fréquents, les itemsets fermés infréquents, ce qui fait qu'une opération de filtrage est nécessaire si on s'intéresse aux itemsets fréquents uniquement.

Données maintenues : $CFI\text{-}Stream$ maintient dans le DIU-Tree tous les itemsets fermés (fréquents et infréquents). $Moment$ maintient dans le CET tous les itemsets fermés fréquents, certains itemsets non fréquents et d'autres fréquents mais non fermés. Il maintient également une table de hachage pour accélérer les tests de fermeture. $NewMoment$ maintient dans le NewCET les itemsets fermés fréquents et les items de la base. Il maintient également une table de hachage pour accélérer les tests de fermeture. $GC\text{-}Tree$ maintient dans l'arbre tous les itemsets fermés (fréquents et infréquents) ainsi que leurs générateurs. $CloStream$ maintient dans le ClosedTable tous les itemsets (fermés fréquents et infréquents). Il maintient également une table qui permet d'indexer les fermés.

Représentation et maintien des données de la base : $NewMoment$ utilise et maintient une projection verticale de la base de données sous forme de vecteur de bit. Cette forme de représentation facilite les calculs de support en plus de compresser la base de données. $Moment$ utilise et maintient un $FP\text{-}Tree$ légèrement modifié pour compresser la base. $GC\text{-}Tree$, $CFI\text{-}Stream$ et $CloStream$ travaillent sur une représentation horizontale de la base mais les données ne sont pas maintenues. Une transaction est oubliée dès qu'elle est traitée.

Conteneur des itemsets fermés : Sauf $CloStream$ qui utilise une table de hachage, tous les algorithmes utilisent un arbre de préfixe pour stocker les itemsets fermés.

Techniques de calcul des fermetures : $GC\text{-}Tree$ et $CloStream$ calculent les fermetures par intersection alors que $NewMoment$ et $Moment$ le font par fusion d'items en combinant chaque nœud de l'arbre dénumeration avec ses frères de droite. $CFI\text{-}Stream$ énumèrent tous les itemsets qui sont des sous-ensembles de la transaction et effectue un test de fermeture sur eux.

Performances : $CFI\text{-}Stream$ est plus rapide que $Moment$ et utilise beaucoup moins d'espace mémoire que lui lorsque le support est relativement faible. Les deux algorithmes ont sensiblement les mêmes performances et occupe la même quantité d'espace mémoire lorsque le support est relativement élevé. $NewMoment$ utilise moins d'espace mémoire que $Moment$ mais, il est légèrement plus rapide que lui. $GC\text{-}Tree$ est plus rapide que $CFI\text{-}Stream$ mais il occupe légèrement plus d'espace mémoire. $CloStream$ est plus rapide que $CFI\text{-}Stream$ et il occupe moins d'espace mémoire.

Forces et Faiblesses : Bien que le coût de la mise à jour soit faible, $Moment$ doit maintenir un nombre important de nœuds dans le CET pour un nœud fermé fréquent (le ratio est 30 pour 1). Donc, s'il y a un nombre important de fermés, la mémoire risque d'être congestionnée et le coût pour l'exploration et la vérification du statut des nœuds relativement élevé. Le maintien d'autres structures de données telles que le FP-Tree et la table de hachage des fermés augmente d'avantage le coût de l'algorithme.

$CFI\text{-}Stream$ calcule les fermés sans utiliser aucune autre information, ce qui, dans certains cas peut occasionner un gain de temps et une réduction considérable de l'espace mémoire. Cependant, il est obligé d'énumérer tous les sous-ensembles possibles de la transaction ajoutée/supprimée, ce qui, peut ne pas être efficace dans le cas de contextes denses.

$NewMoment$ réduit l'espace mémoire et accélère les calculs de support avec l'utilisation des vecteurs de bits. Cependant, les informations issues des précédentes opérations de fouille ne sont pas suffisamment exploitées, car avec juste le maintien des items de la base, il faudra regénérer tous les motifs fréquents non fermés et éventuellement recalculer leur support.

$CloStream$ réduit considérablement les intersections à réaliser avec l'utilisation d'un index sur les fermés. C'est d'ailleurs l'une des raisons qui fait qu'il soit plus rapide que $CFI\text{-}Stream$. Le nombre de tables de hachage utilisés et leur maintien peut cependant poser problème d'autant plus l'utilisation de la table intermédiaire ne se justifie pas.

$GC\text{-}Tree$ ne maintient aucune autre structure car toute l'information utile se trouve dans son arbre d'énumération. Ceci constitue un avantage par rapport à d'autres algorithmes en terme d'utilisation de l'espace mémoire. Le gain d'espace mémoire aurait pu être beaucoup plus important si l'algorithme ne maintenait pas les générateurs des fermés et les fermes infréquents.

5. Conclusion

Dans cet article, nous avons fait un survol de l'état de l'art sur la fouille de motifs fermés fréquents dans un flux. Nous avons décrit les algorithmes les plus connus du domaine et nous les avons analysés sous plusieurs angles. En résumé, il en ressort que les algorithmes de fouille de motifs fermés fréquents dans un flux vont maintenir des informations complémentaires aux motifs fermés. Cette redondance va se manifester de façon complète (tous les fréquents pourront être maintenus dans certains cas et avec ou sans une partie des infréquents) ou partielle (sans pour autant fournir une caractérisation précise de l'ensemble des itemsets qui sera maintenu à chaque instant qui serait indépendante de l'algorithme) entraînant des coûts supplémentaires dont on n'a aucune certitude de la nécessité. Tous ces constats doivent pousser à examiner de plus près l'évolution des structures mises en jeux. En cela, les résultats provenant des études sur l'évolution des treillis de concepts pourraient être mis à profit.

6. Bibliographie

[1] R. AGRAWAL , T. IMIELINSKI , A. SWAMI, « Database Mining : A performance perspective », *IEEE Transactions on Knowledge and Data Engineering*, vol. 5, n° 6, 1993.

[2] N. PASQUIER, Y. BASTIDE, R. TAOUIL, L. LAKHAL, « Discovering frequent closed itemsets for association rules », *International Conference on Database Theory (ICDT)*, 1999.

[3] B. GANTER, R. WILLE, « Formal concept analysis, mathematical foundations », *Springer-Verlag*, 1999.

[4] Y. CHI , H. WANG , P.S YU , R.R MUNTZ, « MOMENT : Maintaining closed frequent itemsets over a stream sliding window », *Proceedings of the 4th IEEE international conference on data mining*, 2004.

[5] Y. CHI , H. WANG , P.S YU , R.R MUNTZ, « Maintening closed frequent Itemset over a data stream sliding window », *Knowledge and information systems*, 2006.

[6] J. REN, C. HUO, « Mining closed frequent itemsets in sliding window over data streams », *3rd International Conference on Innovative Computing Information and Control (ICICIG)*, 2008.

[7] HUA-FU. LI , LI, C. HO , F. KUO , S. LEE, « A new algorithm for maintaining closed frequent itemsets in data streams by incremental updates », *IICDM Workshops*, 2006.

[8] HUA-FU. LI , LI, C. HO , S. LEE, « Incremental updates of closed frequent itemsets over continuous data streams », *Expert Systems With Applications (ESWA)*, vol. 36, n° 2, 2009.

[9] N. JIANG, L. GRUENWALD, « CFI-Stream : mining closed frequent itemsets in data streams », *ACM SIGKDD intl. conf. on knowledge discovery and data mining*, 2006.

[10] SHOW-JANE YEN, YUE-SHI LEE, CHENG-WEI WU, CHIN-LIN LIN, « An Efficient Algorithm for Maintaining Frequent Closed Itemsets over Data Stream », *Lecture Notes in Computer Science*, vol. 5579, 2009.

[11] J. CHEN, S. LI, « GC-tree : A fast online algorithm for mining frequent closed itemsets », *Proc. PAKDD Workshop of HPDMA*, 2007.

[12] J. CHEN, B. ZHOU, L. CHEN, X. WANG, Y. DING, « Finding Frequent Closed Itemsets in Sliding Window in Linear Time », *IEICE - Transactions on Information and Systems* vol. E91-D, 2008.