FIDS : Extraction efficace d'itemsets dans des flots de données

Chedy Raïssi[*,**], Pascal Poncelet[**] et Maguelonne Teisseire[*]

[*]LIRMM UMR CNRS 5506, 161 Rue Ada, 34392 Montpellier cedex 5 - France
{raissi,teisseire}@lirmm.fr
[**]EMA-LGI2P/Site EERIE, Parc Scientifique Georges Besse, 30035 Nîmes Cedex, France
{Chedy.Raissi,Pascal.Poncelet}@ema.fr

Résumé. La recherche d'itemsets dans des flots de données (*data stream*) est un nouveau problème auquel se trouve confrontée la communauté Fouille de Données. Dans cet article, nous proposons une nouvelle approche pour extraire des itemsets maximaux de ces flots. Elle possède les avantages suivants : une nouvelle représentation des items et une nouvelle structure de données pour maintenir les itemsets fréquents couplée avec une stratégie d'élagage efficace. A tout moment, les utilisateurs peuvent ainsi connaître quels sont les itemsets fréquents pour un intervalle de temps donné.

1 Introduction

Pour de nouvelles applications émergentes (transactions financières, navigation sur le Web, téléphonie mobile, ...), il s'avère que les techniques classiques de fouille de données ne sont plus adaptées. En effet, dans ce contexte, les données manipulées sont des séquences d'items obtenues en temps réel, de manière continue et ordonnée (*data streams*). Ainsi, nous nous trouvons confrontés à des flux très importants, potentiellement infinis, de données (paquets TCP/IP, transactions, clickstreams, capteurs physiques, ...) et l'accès rapide à l'intégralité des données devient impossible. Un tel contexte impose alors que l'extraction de connaissances se fasse de manière très rapide (si possible en temps réel) car les résultats doivent pouvoir être utilisés pour réagir rapidement (adaptation en temps réel à des utilisateurs dans le cas de clickstreams, détection de fraude, supervision de processus, ...). Récemment, de nouveaux travaux de recherche, appelés "*Data Stream Mining*", s'intéressent à la définition d'algorithmes de fouille pour prendre en compte ces nouveaux besoins [9, 4, 3, 8, 12]. Comme les algorithmes traditionnels nécessitent plusieurs passes ou bien que le contenu entier de la base soit accessible ils ne sont plus adaptés et l'un des challenges important est de trouver et maintenir les itemsets fréquents des flots [6, 7]. Dans cet article, nous proposons une nouvelle approche appelée FIDS (*Frequent Itemsets mining on Data Streams*) pour extraire les itemsets fréquents dans des flots continus de données. L'une des originalités de notre approche est d'utiliser une nouvelle structure de données, couplée à une stratégie d'élagage rapide, pour maintenir les itemsets. Cette dernière offre la possibilité de réduire l'espace de recherche efficacement dans la mesure où nous sommes à même de retrouver rapidement quel est l'ensemble minimal d'itemsets à considérer lorsque de nouvelles données arrivent dans le flot. Nous reconsidérons également la

problématique de manipulation des itemsets en proposant une nouvelle représentation de ces derniers par des nombres premiers. L'avantage de cette représentation est de traduire les itemsets en une seule valeur par une multiplication de nombres premiers et d'utiliser certaines propriétés arithmétiques associées pour optimiser les traitements. Enfin, nous intégrons la notion de tilted-time windows table, précédemment introduite dans [4], pour permettre à l'utilisateur final de connaître quels sont les itemsets fréquents pour n'importe quelle période de temps. En effet, étant donné les contraintes existantes sur les flots, il n'est plus possible de donner une réponse exacte. Nous pouvons toutefois garantir que notre approche propose une réponse *approximative*, la plus proche possible du réel, par égard à un seuil d'erreur fixé, aux requêtes de l'utilisateur.

Le reste de l'article est organisé de la manière suivante. La Section 2 présente plus formellement le problème. Dans la Section 3, nous proposons un survol des travaux liés et nos motivations pour une nouvelle approche. La Section 4 présente notre solution. Les expérimentations sont décrites dans la Section 5 et une conclusion est proposée dans la Section 6.

2 Problématique

La problématique de la recherche d'itemsets fréquents a initialement été définie par [1] : Soit $I = \{i_1, i_2, \ldots i_m\}$ un ensemble d'items. Soit une base de données DB de transactions, où chaque transaction t_r identifiée de manière unique est un ensemble d'items tel que $t_r \subseteq I$. Un ensemble $X \subseteq I$ est aussi appelé un *k-itemset* et est représenté par $(x_1, x_2, \ldots x_k)$ avec $x_1 < x_2 < \ldots < x_k$. Le support d'un itemset X, noté $supp(X)$, correspond au nombre de transactions dans lesquelles l'itemset apparaît. Un itemset est appelé *fréquent* si $supp(X) \geq \sigma \times |DB|$ où $\sigma \in (0, 1)$ correspond au support minimal spécifié par l'utilisateur et $|DB|$ à la taille de la base de données. Le problème de la recherche des itemsets fréquents consiste à rechercher tous les itemsets dont le support est supérieur ou égal à $\sigma \times |DB|$ dans DB.

Les définitions précédentes considèrent que la base de données est statique. Considérons, à présent, que les données arrivent de manière séquentielle sous la forme d'un flot continu. Soit *data stream* $DS = B_0^1, B_1^2$, ..., B_{n-1}^n, ..., un ensemble infini de batches où chaque batch est estampillé selon un intervalle de temps $[t, t+1]$, i.e. B_t^{t+1}, et n est l'identifiant du dernier batch B_{n-1}^n. Chaque batch B_a^b correspond à un ensemble de transactions : $B_a^b = [T_1, T_2, T_3,$ ..., $T_k]$. Nous supposons également que les batches n'ont pas nécessairement la même taille. La longueur L du data stream est définie par $L = |B_0^1| + |B_1^2| + ... + |B_{n-1}^n|$ où $|B_a^b|$ correspond à la cardinalité de l'ensemble B_a^b.

B_0^1	T_a T_b	(1 2 3 4 5) (8 9)
B_1^2	T_c	(1 2)
B_2^3	T_d T_e	(1 2 3) (1 2 8 9)

FIG. 1 – *Les ensembles de batches B_0^1, B_1^2 et B_2^3*

Le support d'un itemset X à un instant donné t est défini comme étant le ratio du nombre de clients qui possèdent X dans leur fenêtre de temps sur le nombre total de clients. Ainsi, étant donné un support minimal spécifié par l'utilisateur, le problème de la recherche des itemsets dans un flot de données consiste à rechercher tous les itemsets fréquents X, i.e. vérifiant $\sum_{t=0}^{L} \text{support}_t(X) \geq \sigma \times L$, dans le flot.

Exemple 1 *La Figure 1 illustre l'ensemble de tous les batches. Supposons que le support minimal soit fixé à 50%. A partir des transactions de B_0^1, i.e. pour le premier intervalle de temps [0-1], nous pouvons extraire les deux itemsets maximaux suivants : (1 2 3 4 5) et (8 9). Si nous considérons, à présent, l'intervalle de temps [0-2], i.e. batches B_0^1 et B_1^2, l'ensemble des itemsets maximaux est réduit à : (1 2). Finalement, en considérant tous les batches, i.e. pour l'intervalle [0-3], nous obtenons l'ensemble d'itemsets suivant : (1 2), (1) et (2).*

3 Travaux liés

La première approche d'extraction d'itemsets dans des flots de données a été proposée par [9] avec un algorithme, en une passe, basé sur la propriété d'antimonotonie du support. Pour cela, les auteurs utilisent une structure à base d'arbres pour représenter les itemsets. L'autre originalité de cette approche est de proposer des résultats approximatifs en introduisant le concept de ϵ-*deficient function*. En effet, comme nous l'avons précisé en introduction, étant donné la quantité de données et la vitesse à laquelle celles-ci arrivent, il n'est plus possible d'offrir à l'utilisateur une réponse exacte. Une telle réponse, si elle était possible, nécessiterait d'une part un trop grand espace de stockage et d'autre part de parcourir très rapidement un grand nombre de données. Li et al. [8] proposent d'extraire les itemsets fréquents en partant des plus grands aux plus petits et utilisent une extension d'une représentation basée sur des arbres préfixés. Dans [4], les auteurs proposent d'utiliser une structure de type FP-tree [5] pour rechercher des itemsets fréquents à différents niveaux de granularité temporelle. Pour cela, ils mettent en place une nouvelle technique appelée *tilted-time windows table*. Cette notion a initialement été introduite dans [2] et est basée sur l'intuition suivante : les utilisateurs s'intéressent plus souvent aux récents changements avec une granularité fine et aux changements à long terme avec une granularité plus large.

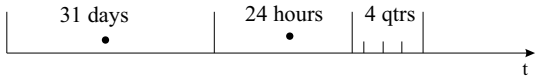


FIG. 2 – *Structure de Tilted-Time Window Naturelle*

La figure 2, extraite de [4], illustre une table de tilted-time windows : les 4 quarts d'heure plus récents, puis les dernières 24 heures et enfin 31 jours. A partir de ce modèle, nous pouvons représenter les informations apparues pendant la dernière heure avec une précision d'un quart d'heure, le dernier jour avec une précision d'une heure, etc. Dans le cas de notre problématique, nous pouvons associer à chaque itemset une tilted time windows table, il est possible de détecter et de connaître leurs supports avec les intervalles de temps associés. Dans [4], les

auteurs étendent ces fenêtre en *tilted-time windows table logarithmique* pour offrir une représentation temporelle plus compacte en espace mémoire tout en introduisant en contrepartie un mécanisme d'approximation : soient $\tau_0, \tau_1, \ldots, \tau_n$ les unités de temps déjà passées. Soit $n \geq b, a \geq 0$ et $b > a$, la fenêtre de temps T_a^b correspond à l'intervalle de temps $\tau_a, \ldots, \tau_b$. Une *table de fenêtres de temps logarithmique* (ou tilted-time windows table logarithmique) est une partition des unités temporelles passées. Chacune de ces partitions représente un niveau de granularité et consiste en une collection de fenêtres de temps consécutives. Chaque niveau contient au plus $max_L \geq 2$ fenêtres. Par exemple, le premier niveau de taille $max_L = 2$ dans une table de fenêtres de temps logarithmique est :

$$\underbrace{T_{a_n}^{b_n}, T_{a_n-1}^{b_n-1}}_{niveau0}, \ldots, T_{a_0}^{b_0}$$

La mise à jour des fenêtres temporelles au sein de la table se fait grâce a une opération de décalage et de compactage sur chacun des niveaux de granularités en commençant par la granularité la plus fine. Le processus s'arrête lorsqu'il n'est plus possible de décaler de nouvelles fenêtres.

Comme nous venons de le constater, l'extraction d'itemsets dans des flots est difficile car de nombreuses contraintes doivent être considérées. La notion de tilted-time windows table offre une solution efficace pour permettre de poser des requêtes sur n'importe quel intervalle de temps et en garantissant une réponse "approximative" la plus proche possible du réel, par égard à un seuil d'erreur fixé. Néanmoins, dans un contexte aussi dynamique, deux problèmes persistent :

(a) Comment retrouver efficacement les itemsets fréquents précédents afin de mettre à jour leur tilted-time windows ? Idéalement, nous souhaiterions éviter de naviguer dans tous les itemsets stockés ou, en d'autres termes, nous souhaiterions réduire l'espace de recherche aux seuls itemsets "intéressants". Dans [10], nous avons proposé une solution basée sur une notion de région qui s'avère bien adaptée pour les motifs séquentiels. Dans ce cadre, la question devient : est-ce qu'une solution basée sur des régions est adaptée aux itemsets ?

(b) Comment vérifier efficacement si un itemset est un sous ensemble d'un autre itemset ? Plus précisément, pouvons nous trouver une nouvelle représentation pour les itemsets nous permettant de vérifier l'inclusion de manière efficace ?

Nous répondons à ces différentes questions dans la section suivante.

4 L'approche FIDS

Dans cette section, nous proposons l'approche FIDS pour extraire des itemsets dans des flots de données. Tout d'abord nous présentons un survol de l'approche. Nous examinons ensuite une nouvelle représentation des itemsets. Finalement, nous décrivons les algorithmes.

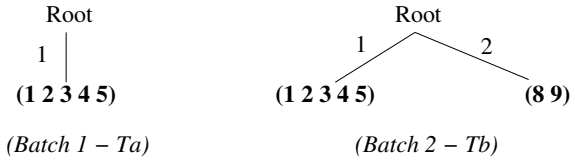
4.1 Aperçu de l'approche

De manière à illustrer le fonctionnement de FIDS, reprenons l'exemple de la Figure 1. Tout d'abord, nous traitons la première transaction T_a dans B_0^1 en stockant T_a dans un treillis

Items	Tilted-T W.	Reg.	Root$_{Reg}$
1	$[t_0^1 = 1]$	1	(1 2 3 4 5)
2	$[t_0^1 = 1]$	1	(1 2 3 4 5)
3	$[t_0^1 = 1]$	1	(1 2 3 4 5)
4	$[t_0^1 = 1]$	1	(1 2 3 4 5)
5	$[t_0^1 = 1]$	1	(1 2 3 4 5)

Itemsets	Taille	Tilted-T W.
$(1\ 2\ 3\ 4\ 5)$	5	$[t_0^1 = 1]$

FIG. 3 – *Les items (gauche) et les itemsets (droite) après la transaction T_a*



(Batch 1 – Ta) *(Batch 2 – Tb)*

FIG. 4 – *$LatticeReg$ après le premier batch*

($LatticeReg$). Ce treillis possède les caractéristiques suivantes : chaque chemin dans le treillis est apparenté à une *région*, les itemsets correspondent aux nœuds et les itemsets stockés sont ordonnées en respectant la propriété d'inclusion. Par construction, tous les sous-ensembles d'un itemset appartiennent à la même région. Ce principe de région est utilisé afin réduire l'espace de recherche lors des étapes de comparaison et d'élagage.

Nous ne stockons que les "itemsets maximaux" dans $LatticeReg$. Ces derniers sont en fait : soit les itemsets, i.e. les transactions, directement extraits du batch, soit leurs sous-ensembles maximaux. Un *sous-ensemble maximal*, dans notre contexte, correspond au plus grand sous-ensemble de la transaction qui peut être construit à partir des items appartenant à une même région. En ne stockant que ces itemsets, notre but est de stocker le minimum d'information pour pouvoir répondre aux requêtes des utilisateurs. A la fin de l'analyse de T_a, nous disposons donc d'un ensemble d'items {1,2,3,4,5}, d'un itemset **(1 2 3 4 5)** et de $LatticeReg$ mis à jour. Les items sont stockés comme illustré Figure 3 (gauche). L'attribut "*Tilted-T W*" correspond au nombre d'occurrences de l'item dans le batch dans l'intervalle de temps considéré. L'attribut "$Root_{Reg}$" est un pointeur vers la racine de région dans $LatticeReg$. Une *racine de région* correspond à l'itemset maximal pour une région donnée. Remarquons que pour une région, il ne peut exister qu'une racine et qu'un item peut apparaître dans plusieurs régions. L'attribut *Val.* correspond à un entier identifiant la région. Pour les itemsets (voir Figure 3 (droite)), nous stockons à la fois les tailles et les tilted-time windows tables associées. Cette dernière information sera utilisée lors de l'étape d'élagage. La partie gauche de la Figure 4 illustre comment le treillis $LatticeReg$ est mis à jour lorsqu'on considère T_a.

Examinons à présent la seconde transaction T_b du batch B_0^1. Comme T_b n'est pas incluse dans T_a, elle est directement insérée dans $LatticeReg$ dans une nouvelle région (voir sous-arbre $(8\ 9)$ dans la Figure 4).

Le batch B_1^2 est simplement réduit à la transaction T_c. T_c étant un itemset maximal nous devons l'inclure dans $LatticeReg$ (C.f. Figure 6). Cependant, l'itemset $(1\ 2)$ est inclus dans

Itemsets	Taille	Tilted-T W.
(1 2 3 4 5)	5	$[t_0^1 = 1]$
(8 9)	2	$[t_0^1 = 1]$
(1 2)	2	$[t_0^1 = 1]$, $[t_1^2 = 1]$

Itemsets	Taille	Tilted-T W.
(1 2 3 4 5)	5	$[t_0^1, 1]$
(8 9)	2	$[t_0^1 = 1]$
(1 2)	2	$[t_0^1 = 1]$, $[t_1^2 = 1]$, $[t_2 = 1]$
(1 2 3)	3	$[t_2^3 = 1]$

FIG. 5 – *Les itemsets mis à jour après B_1^2 (gauche) et après T_d de B_2^3 (droite)*

(1 2 3 4 5), i.e. T_a, ce qui veut dire que quand T_a est intervenu dans les batches précédent, il en est de même pour T_c. Les tilted-time windows tables associées doivent donc être mises à jour (voir Figure 5 (gauche)).

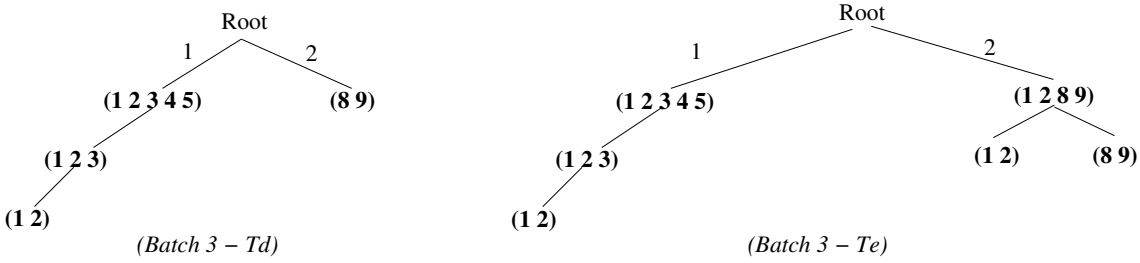


FIG. 6 – *Le treillis des régions après le dernier batch*

La transaction T_d est considérée de la même manière que T_c. Les modifications sur les structures sont décrites Figure 6 et Figure 5 (droite). Considérons à présent la transaction T_e. Cet itemset n'est pas inclus dans l'ensemble de tous les itemsets. Nous remarquons que les items 1 et 2 sont dans la région de valeur 1 alors que 8 et 9 appartiennent à la région de valeur 2. Nous remarquons également que l'itemset (8 9) existe déjà et est inclus dans T_e. Nous incluons donc T_e comme racine de région pour la valeur 2.

Examinons à présent comment les itemsets stockés peuvent être effacés en fonction d'une valeur de support minimale spécifiée par l'utilisateur. En prenant une valeur de σ =50%, pour les transactions incluses dans le batch B_0^1 un itemset est fréquent si nous avons : $support_{B_0^1}(X) \geq \sigma|B_0^1|$. Pour le batch B_1^2, en examinant l'ensemble de tous les itemsets, nous trouvons que seul **(1 2)** vérifie la contrainte de support minimal. Tous les autres itemsets non fréquents sont élagués du treillis en navigant dans $LatticeReg$ en fonction des régions des items.

Ce processus est répété après chaque nouveau batch de manière à n'utiliser que le minimum de mémoire.

4.2 Une représentation efficace des items

Comme nous l'avons précisé dans la section concernant les travaux liés, l'un des problèmes important est de pouvoir calculer rapidement l'inclusion entre deux itemsets. Cette opération coûteuse peut toutefois être améliorée avec une nouvelle représentation des items dans les transactions. Ansi, nous représentons les items à l'aide de nombres premiers (C.f. Figure 7).

Items	**1**	**2**	**3**	**4**	**5**	...	**8**	**9**	...
Nombres premiers	2	3	5	7	11	...	19	23	...

FIG. 7 – *Transformation en nombres premiers*

Remarquons qu'une représentation similaire mais dans un autre contexte a également été proposée de manière indépendante dans [11]. Chaque itemset peut maintenant être représenté par le produit de tous les nombres premiers correspondant aux items inclus. Par définition, le produit de nombres premiers étant décomposable d'une manière unique, nous pouvons rapidement vérifier l'inclusion de deux itemsets (e.g. $X \preceq Y$) en réalisant une opération de modulo sur chacun des itemsets ($Y \ mod \ X$). Si le reste est 0, nous savons que $X \preceq Y$, autrement X n'est pas inclus dans Y. Par exemple, Figure 8, $T_c \prec T_a$, puisque $T_a \ mod \ T_c = 0$.

T_a	(1 2 3 4 5)	$2 \times 3 \times 5 \times 7 \times 11$	2310
T_b	(8 9)	19×23	437
T_c	(1 2)	2×3	6
T_d	(1 2 3)	$2 \times 3 \times 5$	30
T_e	(1 2 8 9)	$2 \times 3 \times 19 \times 23$	2622

FIG. 8 – *Transactions transformées*

4.3 L'algorithme FIDS

Algorithm 1: L'algorithme FIDS

Data: Un ensemble infini de batches B=B_0^1, B_1^2, ... B_{n-1}^n ... ; σ le support minimum défini par l'utilisateur ; ϵ le seuil d'erreur maximal défini par l'utilisateur.

Result: L'ensemble des itemsets maximaux sur le flot.

$REGION \leftarrow \emptyset$; $ITEMS \leftarrow \emptyset$; $ISETS \leftarrow \emptyset$;$region \leftarrow 1$;

repeat

  foreach $B_a^b \in B$ **do**

    UPDATE(B_a^b, $ITEMS$, $ISETS$);

    PRUNE($ITEMS$, $ISETS$, σ, ε);

until *plus de batches*;

Tant que des batches sont disponibles, nous examinons les itemsets pour mettre à jour nos structures (UPDATE). Nous élaguons alors les itemsets non fréquents pour conserver les structures en mémoire (PRUNE). Par la suite, nous considérons que nous disposons de trois structures. Chaque valeur d'$ITEMS$ est un tuple ($labelitem$, $\{time, occ\}$, $\{regions, Root_{Reg}\}$) où $labelitem$ est l'identifiant de l'item, $\{time, occ\}$ est utilisé pour stocker le nombre d'occurrences de l'item au cours des différents batches, et pour chaque région dans $\{regions\}$ nous stockons l'itemset racine de région ($Root_{Reg}$) de la structure $LatticeReg$. La structure

$ISETS$ est utilisée pour stocker les itemsets. Chaque valeur de $ISETS$ est un tuple ($itemset$, $size(itemset)$, $\{time, occ\}$). Finalement, la structure $LatticeReg$ est un treillis où chaque nœud est un itemset stocké dans $ISETS$ et où les arcs correspondent aux régions associées.

Algorithm 2: L'algorithme UPDATE

Data: Un batch $B_a^b = [T_1, T_2, T_3, \ldots, \$T_k]$;
Result: $ITEMS$ et $ISETS$ mis-à-jour.
foreach $transaction\ T\ \in B_a^b$ **do**
  $LatticeMerge \leftarrow \emptyset$; $DelayedInsert \leftarrow \emptyset$; $Candidates \leftarrow$ GETREGIONS(T);
  if $Candidates = \emptyset$ **then**
    INSERT(T,Nouvelle région) ;
  else
    foreach $region\ reg\ \in Candidates$ **do**
      $FirstItemset \leftarrow$ GETROOT$_{Reg}$(Val) ; // On récupère la racine de la région candidate à l'insertion du nouvel itemset
      $NewIts \leftarrow$ GCD(Seq,$FirstItemset$) ; // On calcule l'itemset maximal commun (avec notre représentation, un simple gcd)
      if $(NewIts == T)||(NewIts == FirstItemset)$ **then**
        $LatticeMerge \leftarrow Val$;
      else
        // Un nouvel itemset doit être introduit
        INSERT($NewIts, reg$) ; UPDATETTW($NewIts$) ;
        $DelayedInsert \leftarrow NewIts$;
    // Si l'itemset à introduire ne peut pas être introduit dans une région
    //déjà existante, on en crée une nouvelle
    if $|LatticeMerge| = 0$ **then**
      INSERT(T,Nouvelle région) ; UPDATETTW(T);
    else
      if $|LatticeMerge| = 1$ **then**
        INSERT($T, LatticeMerge[0]$) ; UPDATETTW(T);
      else
        // Un itemset maximal va fusionner 2 ou plusieurs régions
        MERGE($LatticeMerge, T$);

Examinons à présent l'algorithme Update (voir Algorithme 2). Nous considérons chaque transaction T incluse dans le batch et cherchons tout d'abord les régions où tous ses items apparaissent(GETREGIONS). Si les items n'ont pas encore été examinés nous insérons la transaction T dans $LatticeReg$ avec une nouvelle région. Autrement, nous examinons l'ensemble des différentes valeurs de région associées aux items de T. Pour chaque région la fonction GETROOT retourne la racine de région, $FirstItemset$, associée à $Root_{Reg}$, i.e. l'itemset maximal de la région reg, qui sera utilisé dans les étapes suivantes. Comme nous représen-

tons les items par des nombres premiers, la recherche des itemsets maximaux est simplement réalisée par le calcul du plus grand commun dénominateur (PGCD) entre T et $FirstItemset$ (fonction GCD). Cette fonction retourne l'ensemble vide lorsqu'il n'y a pas d'itemsets maximaux ou si les itemsets sont réduits à un seul item. S'il n'existe qu'un seul itemset, i.e. la cardinalité de NewIts est 1, nous savons que celui-ci est soit racine de région, soit la transaction T elle même. Nous la stockons alors dans un tableau temporaire (*LatticeMerge*) utilisé par la suite pour éviter de créer une nouvelle région inutile. Autrement, nous savons que nous avons un nouvel itemset que nous insérons dans $LatticeReg$ (INSERT) et nous effectuons les mises à jour dans les tilted-time windows tables associées (UPDATETTW). Les itemsets sont également stockées dans un tableau temporaire (*DelayedInsert*). S'il existe plus d'un seul sous-itemsets (venant de GCD), nous les insérons dans les branches de $LatticeReg$ correspondantes aux régions concernées. Nous les stockons également avec T dans *DelayedInsert* de manière à retarder leur insertion en tant que nouvelle région. Si *LatticeMerge* est vide, nous savons qu'il n'existe aucun itemset de T qui soit inclus dans des itemsets de $LatticeReg$ et nous pouvons donc inclure T dans $LatticeReg$ dans une nouvelle région. Si la cardinalité de $LatticeMerge$ est supérieure à 1, nous avons forcément un itemset qui sera une nouvelle racine de région et l'insérons.

Par manque de place, nous ne détaillons pas la fonction PRUNE. Son principe est le suivant. Tout d'abord les tilted-time windows des itemsets sont mis à jour en utilisant la même approche que [4]. Les itemsets ne vérifiant pas la contrainte de support sont stockés dans un tableau temporaire ($ToPruned$). Nous considérons alors les items de $ITEMS$. Si un item n'est plus fréquent nous naviguons dans $LatticeReg$ pour supprimer cet item des itemsets et pour supprimer les itemsets qui apparaissent dans $ToPruned$. Cette fonction tire avantage de la propriété d'antimonotonie ainsi que de l'ordre des itemsets dans $LatticeReg$.

Par manque de place, nous ne présentons pas la complexité de l'algorithme FIDS. Toutefois, dans [10], nous calculons la complexité d'une approche basée sur des régions pour rechercher des motifs séquentiels.

5 Experimentations

Les flots de données utilisés sont générés à partir de ceux du ECML/PKDD Challenge 2005. A partir des données initiales pré-traitées, nous avons considéré une durée de batches de 30 secondes et un taux d'erreur de 0.1 (soit 10% d'erreur), les batches contenant à peu près 5000 itemsets. Toutes ces transactions ont été considérées comme entrée standard pour notre programme. Ce dernier a été développé en Java. Toutes les expérimentations ont été réalisées sur un Pentium 3 (1200Mhz) fonctionnant sous Linux avec 512 Mb de RAM.

A chaque traitement des batches, les informations suivantes ont été analysées : la taille de la structure $LatticeReg$ en Bytes et le temps de traitement pour tous les itemsets. L'axe des x représente le numéro du batch. Les Figures 9 illustrent le temps de traitement pour les itemsets. Tous les deux batches ($max_L = 2$ dans notre implémentation), l'algorithme a besoin d'un temps supplémentaire pour calculer les itemsets. Ce temps est en fait lié a l'opération de "compactage" des tilted-time windows tables, réalisé dans nos expérimentations tous les deux batches, dans la structure $LatticeReg$. Deux conditions se posent pour l'extraction d'itemsets sur les flots :

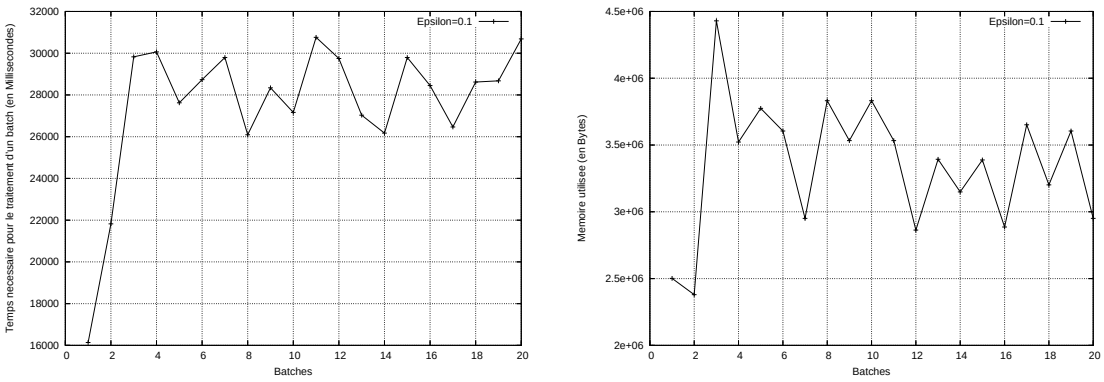


FIG. 9 – *Temps de traitement et mémoire nécessaire pour Fids*

1. Borner la mémoire utilisée. Cette condition est respectée avec cette implémentation de FIDS, puisque la mémoire nécessaire pour la gestion des structures ne dépasse pas les 4,5Mo.
2. Borner le temps de calcul entre chaque batch. Cette condition est aussi respectée, puisque pour chaque étape de calcul, l'implémentation actuelle de FIDS ne dépasse pas les 30 secondes (qui est aussi la taille du batch)

6 Conclusion

Dans cet article, nous avons abordé la problématique de la recherche d'itemsets dans des flots de données. Nos contributions principales sont les suivantes. Premièrement, en utilisant des nombres premiers pour représenter les items du flots nous améliorons la vérification des inclusions d'itemsets et donc améliorons le processus global (cette représentation nécessite quand même de borner le nombre d'items possibles). Deuxièmement, nous avons montré qu'une nouvelle structure de représentation basée sur la notion de région était non seulement adaptée à la recherche de motifs [10] mais également aux itemsets car elle permet de rapidement trouver les itemsets stockés soit pour en extraire les itemsets inclus soit pour élaguer la structure, ce qui évite, grâce à l'indexage, de parcourir des branches ou des parties entières du treillis contenant les itemsets maximaux. Finalement, en stockant seulement un nombre minimal d'itemsets (i.e. les itemsets maximaux) couplée avec la notion de tilted time windows table nous pouvons fournir une réponse (avec un taux d'erreur borné) à l'utilisateur tout en respectant les contraintes de support et de temps fixées. Avec FIDS, l'utilisateur final peut, à tout moment, connaître les itemsets fréquents sur un intervalle de temps ou détecter les changements de fréquences.

Références

[1] Rakesh Agrawal, Tomasz Imielinski, and Arun N. Swami. Mining association rules between sets of items in large databases. In Peter Buneman and Sushil Jajodia, editors,

SIGMOD Conference, pages 207–216. ACM Press, 1993.

[2] Yixin Chen, Guozhu Dong, Jiawei Han, Benjamin W. Wah, and Jianyong Wang. Multi-dimensional regression analysis of time-series data streams. In *VLDB*, pages 323–334, 2002.

[3] Y. Chi, H. Wang, P.S. Yu, and R.R. Muntz. Moment : Maintaining closed frequent itemsets over a stream sliding window. In *Proc. of ICDM'04 Conference*, 2004.

[4] C. Giannella, J. Han, J. Pei, X. Yan, and P. Yu. Mining frequent patterns in data streams at multiple time granularities, 2002.

[5] Jiawei Han, Jian Pei, Behzad Mortazavi-Asl, Qiming Chen, Umeshwar Dayal, and Mei-chun Hsu. Freespan : frequent pattern-projected sequential pattern mining. In *KDD*, pages 355–359, 2000.

[6] C. Jin, W. Qian, C. Sha, J.-X. Yu, and A. Zhou. Dynamically maintaining frequent items over a data stream. In *Proc. of CIKM'04 Conference*, 2003.

[7] R.-M. Karp, S. Shenker, and C.-H. Papadimitriou. A simple algorithm for finding frequent elements in streams and bags. *ACM Transactions on Database Systems*, 28(1) :51–55, 2003.

[8] H.-F. Li, S.Y. Lee, and M.-K. Shan. An efficient algorithm for mining frequent itemsets over the entire history of data streams. In *Proc. of the 1st Intl. Workshop on Knowledge Discovery in Data Streams*, 2004.

[9] G. Manku and R. Motwani. Approximate frequency counts over data streams. In *Proc. of VLDB'02 Conference*, 2002.

[10] C. Raissi, P. Poncelet, and M. Teisseire. Need for speed : Mining sequential patterns in data streams. In *Proc. of the Bases de Données Avancées Conference (BDA 2005)*, 2005.

[11] S.N. Sivanandam, D. Sumathi, T. Hamsapriya, and K. Babu. Parallel buddy prima - a hybrid parallel frequent itemset mining algorithm for very large databases. In *www.acadjournal.com, Vol.13*, 2004.

[12] W.-G. Teng, M.-S. Chen, and P.S. Yu. A regression-based temporal patterns mining schema for data streams. In *Proc. of VLDB'03 Conference*, 2003.