Étude de l'algorithme FP-Growth pour la recherche de regles d'associations et de sa parallélisation via le framework Map-Reduce

May 30, 2012

Contents

Introduction

Considérons un ensemble d'objets E (*items*). On appelle *transaction* tout sous ensemble de E. L'objectif de ce document est de présenter une méthode d'exploration de bases de données de transactions, afin de chercher un lien entre la présence des différents objets.

À partir du résultat de cet algorithme, nous devrions être en mesure de pouvoir deviner, avec un certain niveau de confiance, la présence de certains items en fonction de celle d'autres.

Dans un premier temps, nous aborderons le principe général de cette recherche, puis nous présenterons un algorithme nommé **Frequent Pattern Growth**. Enfin, nous étudierons une méthode de parallélisation des traitements par l'intermédiaire de *Map-Reduce*.

1 Méthode générale

La recherche de regles d'association s'effectue habituellement en deux temps. En premier lieu, il est nécessaire de déterminer *l'ensemble des motifs d'items fréquents* : si on fixe un seuil minimum ζ, on cherchera tous les sous ensembles d'items présents dans au moins ζ transactions.

La recherche de regles d'association se fait dans ces ensembles : on considère un motif fréquent formé de p items, et nous cherchons à en selectionner certains comme prémisses et d'autres comme conclusion, de manière à ce que la règle soit vérifiée plus de ζ fois.

L'algorithme **FPGrowth**, comme son nom l'indique, a pour objectif la génération des motifs fréquents. Son interêt majeur est qu'il compresse la base de données afin d'éviter de la parcourir dans son intégralité, ce qui n'est pas négligeable étant donné qu'en pratique, elle est extrêmement volumineuse.

2 Algorithme de recherche FP Growth

L'algorithme comporte deux étapes. Dans un premier temps, il explore la base de données afin de compresser les transactions sous la forme d'un arbre (*Frequent Pattern Tree*), avant de l'explorer pour en inférer les motifs.

2.1 Construction de l'arbre de préfixes

Considérons la base de transactions. La première étape consiste à déterminer tous les singletons fréquents, c'est à dire tous les items présents dans plus de ζ enregistrements. Ce travail nécessite un premier parcours de la base de données dans son intégralité.

La seconde étape consiste à, pour chaque transaction t :

- On construit la transaction t', où nous supprimons de t les items qui ne sont pas fréquents (qui apparaissent moins de ζ fois), puis nous ordonnons les objets qui la composent par ordre décroissant de fréquence d'apparition.
- On insère t' dans l'arbre que nous construisons, où chaque noeud est un item et chaque chemin une transaction, de manière à ce que deux transactions ayant un préfixe commun soient dans le même sous arbre. Il est donc nécessaire de reparcourir une seconde fois la base de transactions.
- On associe au dernier item de t' (le moins fréquent) la branche de l'arbre. Ainsi, pour chaque item i, nous connaissons l'ensemble des branches qui se terminent par i.

Ainsi, la compression de la base de données sous la forme d'un **FP-Tree** se fait en la parcourant deux fois ($O(n)$). Une fois que ce dernier est construit, pour chaque item fréquent a, nous cherchons les motifs fréquents contenant a de la manière suivante.

2.2 Exploration de l'arbre et génération des motifs fréquents

La méthode utilisée pour l'exploration de l'arbre est un cas particulier du sempiternel *Divide et impera*. Voici comment nous déterminons les ensembles fréquents terminant par un motif m, dans un arbre r :

- On considère le sous ensemble de transactions formé par l'ensemble des branches contenant m : son cardinal représente le support du motif. On le rejette s'il est inférieur à ζ. Dans le cas contraire, on passe à l'étape suivante.
- Parmis toutes les branches que nous avons considéré (*l'arbre conditionnel*), nous les élaguons à partir du préfixe m. L'arbre résultant représente l'ensemble des transactions, sachant qu'elles contiennent m (*arbre conditionnel*).
- Pour chaque noeud a de l'arbre résultant, nous l'ajoutons au motif m pour obtenir $m' = m \cup \{a\}$ et nous itérons sur le même arbre, et le motif m'.

2.3 Application sur un exemple

Supposons que nous connaissons trois transactions et que nous cherchons les ensembles fréquents pour $\zeta = 2$. Après filtrage des items non fréquents et avoir ordonné les transactions restantes par fréquence d'apparition décroissante, la base de de données contient : ["fca", "fb", "cab"] qui peut être compressée sous la forme de l'arbre de préfixes suivant :

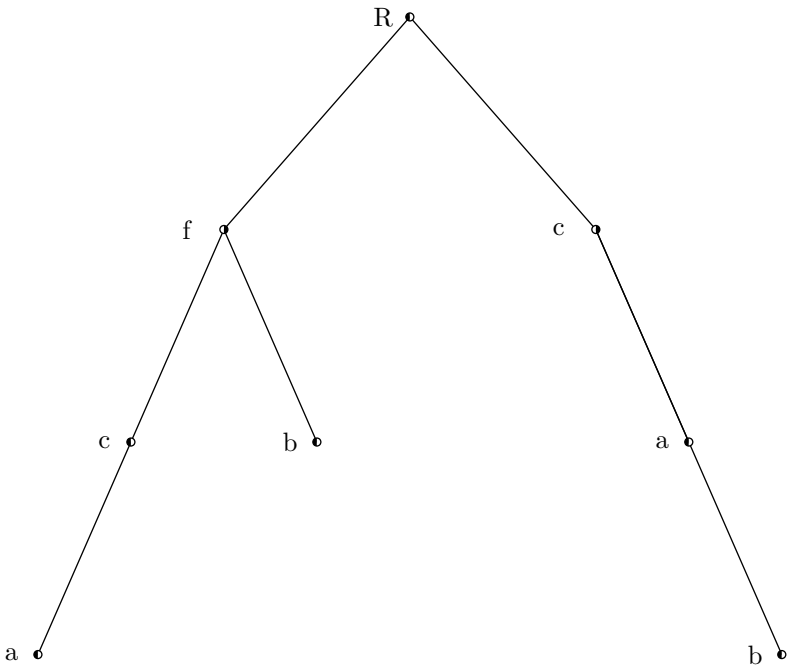

Figure 1: FP-Tree pour la base ["fca", "fb", "cab"]

Nous commençons donc l'exploitation de l'arbre pour en extraire les motifs fréquents. Cherchons par exemple tous les ensembles fréquents contenant $m = \{c\}$. Nous récupérons dans un premier temps toutes les branches contenant le motif m. Il y en a deux, donc le motif est fréquent :

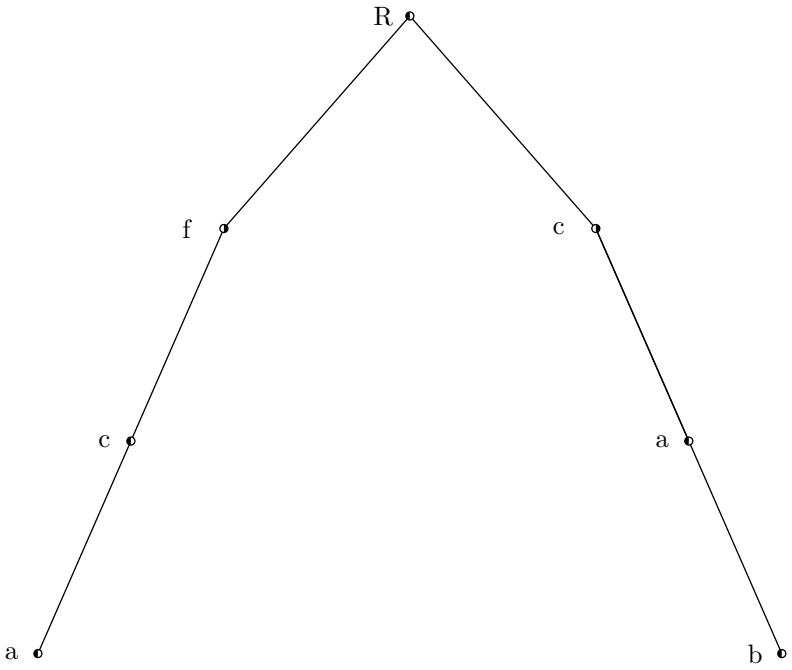

Figure 2: FP-Tree conditionnel pour $m = \{c\}$

On élague l'arbre à partir du suffixe. Il ne reste que la branche contenant l'item "f"

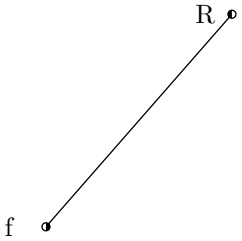

Figure 3: FP-Tree conditionnel élagué pour $m = \{c\}$

On itère donc en cherchant toutes les transactions contenant $m' = \{fc\}$, ce qui revient à chercher l'ensemble des branches contenant f dans l'arbre conditionnel. Il n'y en a qu'une, donc sa fréquence d'apparition est inférieure à ζ. On arrête la recherche, m' n'est pas fréquent, et donc aucun motif contenant m' ne peut l'être.

La recherche débutant par c est donc terminée, le seul motif fréquent trouvé est

donc $m = \{c\}$.
Supposons maintenant que nous cherchons des motifs contenant a ($m = \{a\}$). Nous construisons l'arbre conditionnel à m (qui est identique au précédent dans ce cas).

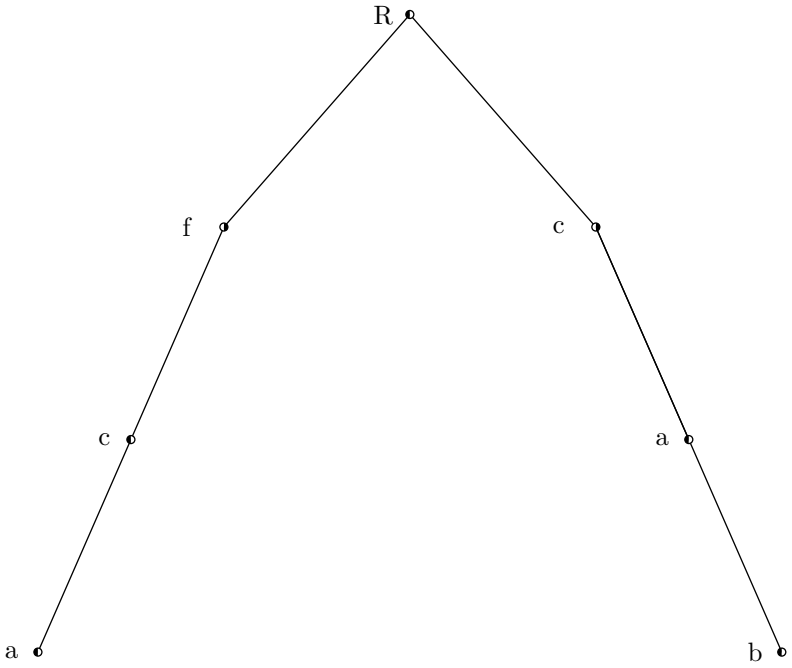

Figure 4: FP-Tree conditionnel pour $m = \{a\}$

Comme il possède 2 branches, le motif est considéré comme fréquent (car $\zeta = 2$). On poursuit donc la recherche, en explorant les motifs de taille supérieure. Pour cela, nous élaguons l'arbre à partir du suffixe $m = \{a\}$:

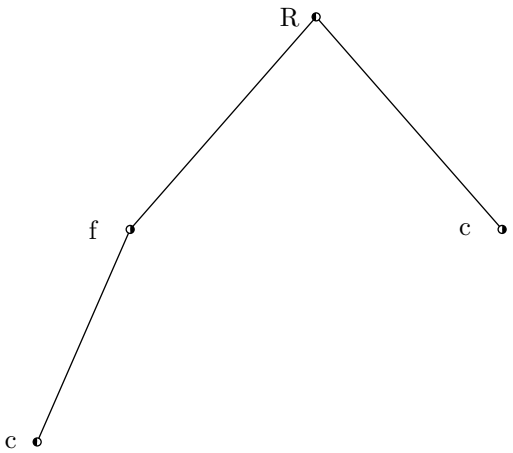

Figure 5: FP-Tree conditionnel élagué pour $m = \{a\}$

Nous devons à présent recommencer l'opération pour les motifs $m_1 = \{fa\}$ et $m_2 = \{ca\}$ (donc considérer les items f et c dans l'arbre conditionnel).

Considérons l'item f (donc le motif m_1). Il n'y a qu'une seule branche de l'arbre conditionnel qui le contient : son support[1] est donc inferieur à ζ. Ainsi $m1_1$ n'est pas fréquent et il est inutile de poursuivre la recherche à ce niveau.

Considérons à la place l'item c, et donc le motif $m_2 = \{ca\}$. Le motif est présent dans les deux branche, et est donc considéré comme fréquent. Nous poursuivons l'exploration dans l'arbre conditionnnel au motif m_2 :

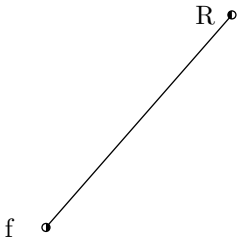

Figure 6: FP-Tree conditionnel pour $m_2 = \{ca\}$

Le seul item que nous pouvons choisir dans l'arbre conditionnel est f (nous considérons donc le motif $m_3 = \{fca\}$). Comme il n'est présent que dans une seule branche, il n'est pas considéré comme fréquent.

En pratique, nous démarrons l'algorithme en recherchant tous les suffixes formés par les singletons fréquents, ce qui permet de balayer à coup sûr toutes les possibilités

[1] Le support est la fréquence d'apparition.

tout en évitant d'explorer des transactions sans interêt.

3 Étude de la parallélisation de l'algorithme FP Growth

3.1 Présentation du framework Map-Reduce

Le framework *Map-Reduce* permet de simplifier grandement la parallélisation d'algorithmes réalisant des traitements identiques pour toutes les données d'une base d'enregistrements. Il utilise en pratique deux notions :

- Le **Map**, qui est une fonction prennant en paramètre un enregistrement, et retournant une liste de couples (clef, valeur). Nous pourrions le formaliser ainsi :

```
Map :: Record -> [ (Key, Value) ]
```

- Le **Reduce** est une fonction qui sera appellée avec les resultats du Map. Elle récupèrera, pour une clef donnée, la liste des valeurs associées. Ainsi, nous pourrions formaliser son type :

```
Reduce :: (Key, [Value]) -> [ (Key, Value) ]
```

Map-Reduce se charge de répartir les *Map* et les *Reduce* entre les différentes machines et processus disponibles, de la mémoire partagée ainsi que de la gestion des erreurs.

3.2 Version parallélisée de l'algorithme