Tạp chí Tin học và Điều khiển học, T.23, S.3 (2007), 260–271

THUẬT TOÁN KHAI THÁC TẬP THƯỜNG XUYÊN HIỆU QUẢ DỰA TRÊN KỸ THUẬT PHÂN LỚP DỮ LIỆU

NGUYỄN HỮU TRỌNG

Khoa Công nghệ Thông tin - Trường Đại học Nha Trang; Email: trongnh@ntu.edu.vn

Abstract. Find all association rules is a basic work in data mining. This problem is solved in two main steps: First, find all the frequent itemsets follow the given S0 threshold. Second, based on frequent itemset, find the association rules. All difficulties on the problem are focused on Step 1. Researches about association rules exploitation centralize processing speed improvement, memory capacity and the number of time access the hard disk. In this article, we propound the solution for the upper problem by divided data into n classes. Each of class is archived in one file on the external storage independently and offers the SPP_Mining algorithm to exploit the frequent itemset with S0 threshold and to parallel processing on n computers.

Tóm tắt. Tìm các luật kết hợp là công việc cơ bản trong khai thác dữ liệu. Bài toán được giải theo hai bước chính: Bước một, tìm tất cả các tập thường xuyên theo ngưỡng S_0 cho trước. Bước hai, dựa vào các tập thường xuyên, tìm các luật kết hợp. Tất cả khó khăn của bài toán tập trung ở bước một. Các nghiên cứu về khai thác luật kết hợp đều tập trung cải tiến tốc độ xử lý, dung lượng bộ nhớ và số lần truy cập đĩa. Trong bài báo này, chúng tôi đề xuất phương án giải bài toán trên bằng cách phân hoạch dữ liệu thành n lớp, mỗi lớp được lưu trữ độc lập thành một file trên bộ nhớ ngoài và đề xuất thuật toán SPP_Mining để khai thác các tập thường xuyên với ngưỡng S_0 tùy ý và được xử lý song song n trên máy.

1. MỞ ĐẦU

Quá trình nghiên cứu về khai thác dữ liệu được tổng kết bởi Q. Zhao trong [1]. Từ thuật toán AIS lần đầu tiên được Agrawal. R giới thiệu năm 1993 trong [2], thuật toán Apriori năm 1996 [3], rồi từng bước được các tác giả chính và nhóm nghiên cứu công bố: FP-Tree của J. Han năm 2000 [4], DCI của C. Lucchese năm 2005 [5], CHARM của M. J. Zaki năm 2005 [6], LCM của T. Uno năm 2006 [7], BFS của V. Choi năm 2006 [8], tất cả đều đưa toàn bộ dữ liệu vào bộ nhớ trong để xử lý. Hạn chế của các thuật tán này là không thể giải được khi dữ liệu lớn. Khái niệm phân lớp dữ liệu đã được Ashok Savesere và đồng nghiệp đưa ra năm 1995 [9] và được Shakil Ahmed hoàn thiện năm 2006 trong [10] nhằm khắc phục hạn chế trên. Bài báo này đề xuất một thuật toán mới để tìm tập thường xuyên dựa vào kỹ thuật phân lớp dữ liệu, có tốc độ xử lý nhanh hơn các thuật toán đã được công bố.

1.1. Mô tả bài toán

Cho $I = \{x_1, x_2, ..., x_n\}$ là tập hợp các mục dữ liệu. Một tập con $t = \{x_{i1}, x_{i2}, ..., x_{ik}\} \subseteq I$ gọi là một giao tác trên I. Một tập hợp gồm m giao tác $T = \{t_1, t_2, ..., t_m\}$ gọi là một cơ sở dữ liệu (CSDL) giao tác trên I. Mỗi tập hợp $X \subseteq I$ gọi là tập mục dữ liệu (itemset),

mỗi tập con $S \subseteq T$ gọi là tập định danh giao tác (tidset). Độ hỗ trợ (support)của một tập mục dữ liệu X, ký hiệu $\text{Supp}(X)$ là số các giao tác trong T chứa X. Để đơn giản, quy ước dùng $\text{Supp}(x_i)$ thay cho $\text{Supp}(\{x_i\})$ khi tập chỉ có một mục dữ liệu. Cho S_0 là một số nguyên, ta nói X là tập mục dữ liệu thường xuyên (frequent itemset) theo ngưỡng S_0 nếu $\text{Supp}(X) \geqslant S_0$. Một biểu thức. $X_1 \to X_2$, với $X_1, X_2 \subseteq I$ và $X_1 \cap X_2 \neq \phi$ được gọi là một luật kết hợp (Association rule) trên T. Độ hỗ trợ của luật kết hợp $X_1 \to X_2$, ký hiệu $\text{Supp}(X_1 \to X_2) = \text{Supp}(X_1 \cup X_2) = \text{Supp}(X_1X_2)$. Luật kết hợp $f : X_1 \to X_2$ trên T là luật thường xuyên theo ngưỡng S_0 nếu $\text{Supp}(X_1 \to X_2) \geqslant S_0$. Độ tin cậy (Confidence) của luật $X_1 \to X_2$, ký hiệu $\text{Conf}(X_1 \to X_2)$, là tỷ số: $p = \text{Conf}(X_1 \to X_2) = \text{Supp}(X_1X_2)/\text{Supp}(X_1)$. Với $C_0 \in P(0, 1]$, ta nói $f : X_1 \to X_2$ là luật tin cậy (Confident rule) theo ngưỡng S_0 và C_0 nếu f là luật thường xuyên theo ngưỡng S_0 và $\text{Conf}(X_1 \to X_2) \geqslant C_0$.

Bài toán khai thác luật kết hợp: Cho CSDL giao tác T trên tập mục dữ liệu I với $\|T\| = m$ (số phần tử của T), $\|I\| = n$, ngưỡng độ hỗ trợ tối thiểu S_0 và ngưỡng độ tin cậy tối thiểu C_0. Hãy tìm tất cả các luật kết hợp trên T thỏa ngưỡng S_0 và C_0.

1.2. Các hướng tiếp cận giải toán luật kết hợp

Tất cả khó khăn của bài toán tập trung ở bước tìm các tập mục dữ liệu thường xuyên. Đến nay, có ba hướng tiếp cận chính để giải quyết bài toán tìm tập mục dữ liệu thường xuyên [1].

Hướng thứ nhất: Gọi là hướng tiếp cận Apriori, dựa vào thuật toán Apriori được Agrawal. R và đồng nghiệp giới thiệu lần đầu tiên năm 1993 trong [2] và từng bước được các nhà nghiên cứu dày công cải tiến trong [4–8]. Nội dung cơ bản của thuật toán này gồm các bước:

1) Duyệt cơ sở dữ liệu lần thứ nhất để xác định tập một mục dữ liệu thường xuyên:

$$L_1 = \{\{x_i\} | x_i \in I, \text{Supp}(x_i) \geqslant S_0\}.$$

2) Lặp bước 2 cho đến khi $L_{k+1} = \phi$;

Từ $L_k = \{X \subseteq I \| \|X\| = k,\ \text{Supp}(X) \geqslant S_0\}$ với $k = 1, 2, ...,$ lập tập ứng viên C_{k+1}:

$C_{k+1} = \{X \subseteq I | \|X\| = k + 1,\ Y \subseteq X$ thỏa $\|Y\| = k$ thì $Y \in L_k\}$;

$L_{k+1} = \{X \in C_{k+1} | \text{Supp}(X) \geqslant S_0\}$. Hạn chế của hướng tiếp cận này là:

1) Sự bùng nổ các tập mục dữ liệu ứng viên cần tính độ hỗ trợ, dẫn đến không đủ thời gian thực và bộ nhớ của máy tính để xử lý.

2) Quá nhiều lần duyệt qua cơ sở dữ liệu.

Hướng thứ hai: Gọi là hướng tiếp cận FP_Tree, dựa vào thuật toán FP_Tree do Han J., Pei H., Yin Y. đưa ra năm 2000 [3]. Nội dung cơ bản của thuật toán này gồm các bước:

1) Duyệt cơ sở dữ liệu lần thứ nhất để tính độ hỗ trợ cho mỗi mục dữ liệu và tìm ra được tập thường xuyên một mục dữ liệu.

2) Tạo FP_Tree. Tạo ra nút gốc Pt của FP_Tree, gắn nhãn Root. Duyệt cơ sở dữ liệu lần thứ hai để xây dựng FP_Tree: Ứng với mỗi giao tác trong T, lấy ra những mục dữ liệu thường xuyên và sắp xếp các mục này thành một danh sách có thứ tự giảm dần của độ hỗ trợ. Phân các mục dữ liệu trong danh sách thành hai phần $[x|L]$, với x là mục dữ liệu thường xuyên đầu tiên trong danh sách và L là phần còn lại, gọi thủ tục Insert_Tree$([x|L], Pt)$.

Thủ tục Insert_Tree$([x|L], Pt)$ làm việc như sau: Nếu Pt có một nút con N sao cho $N.\text{ItemName} = x$ thì tăng số lần đếm nút N lên 1, ngược lại, tạo một nút N mới là con của Pt và gán $N.\text{ItemName} = x$ với lần đếm nút N là 1, tạo con trỏ liên kết từ nút cùng tên xuất hiện trước đến nó. Thủ tục Insert_Tree được gọi đệ quy Insert_Tree$([y|L_1], N)$ với y là

mục dữ liệu đầu tiên của L và L_1 là phần còn lại. Thủ tục Insert_Tree được gọi đệ quy cho đến khi L là rỗng.

3) Dùng thuật toán FP_Growth để tìm các tập mục dữ liệu thường xuyên.

Hướng tiếp cận này giải quyết được cơ bản hai vấn đề tồn tại của hướng tiếp cận Apriori. Thứ nhất, quá trình xây dựng FP_Tree không xét đến số lượng các tập thường xuyên mà chỉ xét các mục dữ liệu thường xuyên trong từng giao tác, do đó số tập mục dữ liệu được xét là số giao tác của dữ liệu nên không xảy ra việc bùng nổ các tập ứng viên. Thứ hai, trong quá trình xây dựng FP_Tree chỉ có hai lần duyệt qua cơ sở dữ liệu, lần thứ nhất duyệt để tìm các mục dữ liệu thường xuyên (giống như hướng tiếp cận Apriori) và lần thứ hai là duyệt qua cơ sở dữ liệu để xây dựng FP_Tree.

Hạn chế của hướng tiếp cận này là:

1) Trong khai thác tập thường xuyên, cần phải tính toán với nhiều ngưỡng độ hỗ trợ tối thiểu khác nhau. Khi cần tìm tập thường xuyên với một ngưỡng tối thiểu S khác, hướng tiếp cận này phải làm lại từ đầu.

2) Thuật toán FP_Growth dùng để tìm các tập thường xuyên phải gọi đệ quy thủ tục FP_Growth($Tree, \alpha$) liên tiếp trên các nhánh cây con. Khi dữ liệu lớn, việc gọi đệ quy sẽ chiếm một bộ nhớ rất lớn và chạy rất chậm.

3) Cách giải quyết bài toán của Han J. là đưa toàn bộ dữ liệu vào bộ nhớ trong khi xây dựng FP_Tree. Điều này rất khó thực hiện khi dữ liệu lớn.

Hướng thứ ba: Phân hoạch dữ liệu. Hướng này sử dụng cho dữ liệu lớn, không thể đưa toàn bộ dữ liệu vào bộ nhớ trong. Ý tưởng đầu tiên của hướng này do Ashok Savasere và đồng nghiệp tại Viện Tính toán trường Đại học kỹ thuật Georgia, Atlanta, Hoa Kỳ đề xuất năm 1995 với thuật toán Partition [9], sau đó được Shakil Ahmed và đồng nghiệp thuộc khoa Khoa học Máy tính trường Đại học Liverpool hoàn thiện năm 2006 với thuật toán Partition-P-tree [10]. Nội dung cơ bản của thuật toán này gồm các bước:

1) Xác định một thứ tự cho các mục dữ liệu thường xuyên.

2) Chia danh sách các mục dự liệu đã có thứ tự thành k nhóm và đánh số $1, 2, 3, ..., k$.

3) Phân chia cơ sở dữ liệu giao tác thành h phân đoạn.

4) Ứng với mỗi phân đoạn, xây dựng k PP-Tree trên bộ nhớ trong, cuối cùng lưu kết quả ra bộ nhớ ngoài. Quá trình xây dựng này chỉ cần một lần duyệt cơ sở dữ liệu gốc.

5) Ứng với nhóm 1, đọc cây PP1 của mọi phân đoạn vào bộ nhớ rồi áp dụng thuật toán Apriori-TFP tạo một $T - Tree$ để tìm các tập mục dữ liệu thường xuyên trong phân hoạch này. Giai đoạn này, trên mỗi phân đoạn chỉ đọc các cây PP1 đúng một lần. $T - Tree$ giữ trong bộ nhớ trong suốt quá trình xử lý, cuối cùng được lưu lên đĩa.

6) Lặp lại bước 5 cho các nhóm $2, 3, , k$.

Hướng tiếp cận này có nhiều ưu điểm, tuy vẫn còn hạn chế:

1) Khi xây dựng các PP-Tree trên mỗi phân đoạn dữ liệu phải đếm số lần xuất hiện của tất cả các tập mục dữ liệu trong từng nhóm để đưa vào cây, dẫn đến việc xây dựng các PP-Tree là phức tạp.

2) Khi tìm tập các tập thường xuyên ở bước 5 bằng cách dùng thuật toán Apriori-TFP tạo $T - Tree$, cây $T - Tree$ phải lưu giữ trong suốt quá trình tính toán, điều này gặp khó khăn trong trường hợp dữ liệu lớn.

3) Đặc biệt, hướng tiếp cận này không thể xử lý song song để có kết quả nhanh.

Để khắc phục hạn chế của các cách tiếp cận trên, tác giả đề xuất thuật toán SPP để giải bài toán tìm các tập mục dữ liệu thường xuyên dựa trên kỹ thuật phân lớp dữ liệu.

2. THUẬT TOÁN SPP

(Simple Partition Pattern)

2.1. Nội dung của thuật toán

Nội dung của thuật toán này bao gồm các bước:

1) Duyệt qua cơ sở dữ liệu lần thứ nhất để tính độ hỗ trợ của tất cả các mục dữ liệu.

2) Duyệt qua cơ sở dữ liệu lần thứ hai để phân hoạch các giao tác: Ứng với mỗi giao tác, tạo một danh sách gồm các mục dữ liệu trong giao tác và sắp xếp các mục này theo thứ tự giảm dần của độ hỗ trợ, x_i là mục dữ liệu đầu tiên trong danh sách. Lưu danh sách này vào tập tin FPD_i.

3) Phân hoạch các tập mục dữ liệu: Ứng với mỗi tập tin FPD_i, xây dựng cấu trúc SP_Tree, là cấu trúc cây nhị phân để chuyển tập tin FPD_i vào bộ nhớ trong. Duyệt qua cây để tìm tất cả các hậu tố của từng x_j và lưu hậu tố vào tập tin FPP_j, là tập tất cả các hậu tố của x_j, $1 \leqslant j \leqslant n$.

4) Dùng thuật toán SPP_Mining để khai thác tập thường xuyên theo ngưỡng S_0. SPP_Mining dùng thuật toán tựa Apriori (Thuật toán Apriori trên tập dữ liệu giao tác có trọng số) để tìm tất cả các tập thường xuyên trong từng FPP_j, là tập hậu tố của x_j. Nối x_j vào các tập thường xuyên trong FPP_j ta được tập các danh sách mục thường xuyên bắt đầu x_j của T. Tập các danh sách mục dữ liệu thường xuyên trên toàn cơ sở dữ liệu là hội tất cả các tập danh sách mục dữ liệu thường xuyên bắt đầu x_j với $1 \leqslant j \leqslant n$.

2.2. Phân hoạch cơ sở dữ liệu giao tác

Thực hiện bước 1 và 2, phân dữ liệu vào các tập tin FPD_i ở bộ nhớ ngoài. Cấu trúc FPD:

$$FPD_i = \{[x_{i_1}, x_{i_2}, ..., x_{i_k}] | \text{Supp}(x_{i_1}) \geqslant \text{Supp}(x_{i_2}) \geqslant ... \geqslant \text{Supp}(x_{i_k}) \text{ và}$$

$$\exists t \in T,\ t = \{x_{i_1}, x_{i_2}, ..., x_{i_k}\}\}.$$

Procedure Partition_Data(T, I, F, ItemList);
Input: T - Cơ sở dữ liệu giao tác trên $I = \{x_1, x_2, ..., x_n\}$;
Output: $F = \{FPD_1, FPD_2, ..., FPD_n\}$, n tập tin phân hoạch các giao tác;
ItemList: Mảng chứa độ hỗ trợ của các mục dữ liệu đã được sắp thứ tự giảm dần;
Method:

```
For each j ∈ {1, ..., n} do ItemList[j] := 0;
For each t ∈ T do
    For each j ∈ {1, ..., n} do
        If x_j ∈ t then
            ItemList[j] := ItemList[j] + 1;
        EndIf;
    EndFor;
EndFor;
Sort_List(ItemList); // Sắp xếp ItemList theo thứ tự giảm dần của Supp(x_i) = ItemList[i].
```

For each $j \in \{1, ..., n\}$ do
    Create_File(FPD_j);
EndFor;
For each $t \in T$ do
    $t' := Sort\_Item(t) = [x_i, x_{i_2}, ..., x_{i_k}]$ với
        $\text{Supp}(x_i) \geqslant \text{Supp}(x_{i_2}) \geqslant ... \geqslant \text{Supp}(x_{i_k})$;
    Write(FPD_i, t');
EndFor;

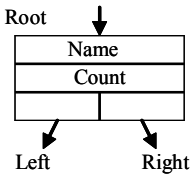
2.3. Phân hoạch các tập mục dữ liệu (Partition Patterns)

Cho danh sách các mục dữ liệu $X = [x_i, x_{i_2}, ..., x_{i_k}]$. Ta gọi $Y = [x_i, x_{i_2}, ..., x_{i_k}]$ là hậu tố của x_i nếu $\text{Supp}(x_i) \leqslant \text{Supp}(x_{i_2}) \leqslant ... \leqslant \text{Supp}(x_{i_k})$.

Việc phân hoạch các mục dữ liệu là phân chia các hậu tố của các x_j vào các tập tin FPP_j, dựa vào kỹ thuật xây dựng và duyệt cây nhị phân SP_Tree.

2.3.1. Xây dựng SP_Tree

Cây SP_Tree là một cây nhị phân đơn giản. Mỗi nút của cây con bao gồm bốn trường: Trường thứ nhất có tên là Name, chứa tên của mục dữ liệu, trường thứ hai có tên là Count, chứa số đếm các giao tác chứa các mục dữ liệu trong nhánh cây này, trường thứ ba và bốn có tên là Left và Right, là hai con trỏ nối đến nút con của nó. Các nút của cây gồm hai loại: Loại một: Trường Name $\neq \phi$ và Count $\neq 0$, đây là loại nút bình thường. Loại hai: Trường Name $= \phi$ và Count $= 0$, ta gọi là nút rỗng, dùng để mô tả nút đầu của một nhánh con trong cây.



Hình 1. Một nút của SP_Tree

Với mỗi tập tin FPD_i đã được phân hoạch ở mục trước, ta xây dựng một cây nhị phân SP_Tree tương ứng.

Thủ tục Xây dựng SP_Tree

Đầu tiên tạo nút gốc có tên là Root. Duyệt qua tập tin FPD_j, ứng với mỗi phần tử L của tập tin, ta phân L thành hai phần $L = [x|L_1]$ với x là mục dữ liệu đầu tiên của L và L_1 là phần còn lại, gọi thủ tục Insert($Left, x, L_1, Root$) để chèn L vào cây.

Thủ tục Insert($Left|Right, x, L, P$) dùng để chèn các mục dữ liệu trong một danh sách vào cây. Có hai trường hợp phải chèn với hai tham số trong lời gọi thủ tục. Trường hợp một, gọi với tham số $Left$ để chèn nút thông thường vào nhánh trái của cây: Insert($Left, x, L, P$). Trường hợp hai: Gọi với tham số $Right$ để tạo nhánh phải của cây: Insert($Right, x, L, P$).

1) Trường hợp một: Insert($Left, x, L, P$).

a) Nếu $P =$ Null thì tạo một nút mới N, gán N.Name $:= x, N$.Count $:= 1, N.Left :=$ Null, $N.Right :=$ Null, nối P vào nút N. Nếu $L \neq \phi$ thì phân L thành hai phần: $L = [y|L_1]$ với y là mục dữ liệu đầu tiên của L và L_1 là phần còn lại, gọi đệ quy Insert($Left, y, L_1, N.Left$).

b) Nếu $P \neq$ Null, nghĩa là P chỉ đến nút N và N.Name $= x$, tăng N.Count lên 1 đơn vị. Nếu $L \neq \phi$ thì phân L thành hai phần $L = [y|L_1]$ với y là mục dữ liệu đầu tiên của

L và L_1 là phần còn lại. Nếu $N.Left$ = Null thì gọi đệ quy Insert$(Left, y, L_1, N.Left)$, ngược lại, nghĩa là $N.Left$ trỏ đến nút M, ta lại xét tiếp: Nếu M.Name $= y$ thì gọi đệ quy Insert$(Left, y, L_1, N.Left)$, ngược lại, ta gọi đệ quy Insert$(Right, y, L_1, N.Right)$.

2) Trường hợp hai: Insert$(Right, x, L, P)$.

a) Nếu P = Null, thực hiện hai công việc: Đầu tiên, tạo một nút E rỗng, nút đầu của nhánh cây con, gán E.Name $:= \phi$, E.Count $:= 0$, $E.Left :=$ Null, $E.Right :=$ Null, trỏ P đến E; tiếp theo gọi đệ quy Insert$(Left, x, L, E.Left)$.

b) Nếu $P \neq$ Null, nghĩa là P đã trỏ đến một nút đầu của một nhánh con. Nếu x trùng với tên nút đầu tiên của nhánh con trái: $P.Left^{\wedge}$.Name $= x$, ta gọi đệ quy Insert$(Left, x, L, P.Left)$, ngược lại ta gọi đệ quy Insert$(Right, x, L, P.Right)$.

Việc gọi thủ tục Insert$(Left|Right, x, L, P)$ cho đến khi L rỗng.

Procedure SP_Tree$(FPD_i, Root)$;
Input: FPD_i: Tập tin lưu các danh sách bắt đầu x_i;
Output: Root: Một SP_Tree;
Method:
    $Root :=$ Null;
    For each $L \in FPD_i$ do
        $L := [x|L_1]$;
        Insert$(Left, x, L_1, Root)$;
    EndFor;

Procedure Insert$(Kind, x, L, P)$;
Input: Kind $\in \{Left, Right\}$; x: Một mục dữ liệu;
    L: Danh sách các mục dữ liệu;
    P: Con trỏ trỏ đến nút trong cây con;
Output: P - Một nhánh trong SP_Tree;
Method:
    If Kind $= Left$ then
        If $P =$ Null then
            Tạo nút Nvà gán N.Name $:= x$,
            N.Count $:= 1$; các nhánh con là Null; Nối P đến nút N;
            If $L \neq \phi$ then
                $L := [y|L_1]$;
                Insert$(Left, y, L_1, P.Left)$;
            EndIf
        Else //P trỏ đến nút N có N.Name $= x$;
            P.Count $:= P$.Count $+ 1$;
            If $L \neq \phi$ then
                $L := [y|L_1]$;
                If $(P.Left =$ Null) or $(P.Left^{\wedge}$.Name $= y)$ then
                Insert$(Left, y, L_1, P.Left)$
            Else
                Insert$(Right, y, L_1, P.Right)$;
            EndIf;
        EndIf;
    EndIf;

Else //$Kind = Right$
  If P = Null then
    Tạo nút E rỗng;
    Nối P đến nút E;
    Insert($Left, x, L, E.Left$);
  Else
    If $P.Left^{\wedge}$.Name = x then
      Insert($Left, x, L, P.Left$)
    Else
      Insert($Right, x, L, P.Right$);
    EndIf;
  EndIf;
EndIf;

2.3.2. Thủ tục SP_Mining

Trong quá trình xây dựng cây SP_Tree ta thấy: Trên mỗi nhánh, một mục dữ liệu tại một nút luôn có số đếm nhỏ hơn hay bằng số đếm của mục dữ liệu tại các nút tổ tiên của nó. Dựa vào tính chất này, ta lọc ra các danh sách các mục dữ liệu và độ hỗ trợ của nó như sau: Tại mỗi nút N chứa mục dữ liệu x_j của cây SP_Tree, ta lấy ra một danh sách các mục dữ liệu $X = [x_j, x_{j_1}, ..., x_{j_k}]$ với $x_{j_1}, ..., x_{j_k}$ lần lượt là các mục dữ liệu theo thứ tự tại các nút tổ tiên của N, nên ta có Supp$(x_j) \leqslant$ Supp$(x_{j_1}) \leqslant ... \leqslant$ Supp(x_{j_k}) suy ra $Y = [x_{j_1}, ..., x_{j_k}]$ là hậu tố của x_j. Đặt Sup_$Y = N$.Count và lưu (Y, Sup_Y) vào FPP_j

$FPP_j = \{(Y, \text{Sup\_}Y)\}$ với $Y = [y_{j_1}, ..., y_{j_k}]$ thỏa Supp$(x_j) \leqslant$ Supp$(y_{j_1}) \leqslant ... \leqslant$ Supp(y_{j_k}) và $\exists t \in T : \{x_j, y_{j_1}, ..., y_{j_k}\} \subseteq t$ (File Partiton Patterns).

Thủ tục SP_Mining duyệt qua cây SP_Tree để tìm hậu tố của tất cả các mục dữ liệu và lưu vào các tập tin FPP_j với $1 \leqslant j \leqslant n$.

Procedure SP_Mining(Root);
Input: Root: Một SP_Tree
Output: $F = \{FPP_1, FPP_2, ..., FPP_n\}$: n tập tin phân hoạch tập mục dữ liệu;
Method:
  Gọi Filter_Pattern($Z, Root$);
Procedure Filter_Pattern(X, P);
Input: X: Danh sách các mục dữ liệu; P là con trỏ trỏ đến một nút trong cây;
Output: $F = \{FPP_1, FPP_2, ..., FPP_n\}$.
Method:
  If $P \neq$ Null then
    If P.Name $\neq \phi$ then
      $x_i :=$ P.Name;
      $Su :=$ P.Count;
      Write ($FPP_i, (X, Su)$);
      $X := [x_i|X]$;
    EndIf;
    Filter_Pattern($X, P.Left$);
    Filter_Pattern($X, P.Right$);
  EndIf;

2.3.3. Thủ tục Partition_Pattern

Thủ tục Partition_Pattern tuần tự chuyển các phân hoạch giao tác FPD_i thành các phân hoạch mục dữ liệu FPP_j, $1 \leqslant i, j \leqslant n$.

Procedure Partition_Pattern(PD, F);
Input: $PD = \{FPD_1, FPD_2, ..., FPD_n\}$: n tập tin phân hoạch các giao tác;
Output: $F = \{FPP_1, FPP_2, ..., FPP_n\}$: n tập tin phân hoạch các mục dữ liệu;
Method:
For each $x_i \in I = \{x_1, x_2, ..., x_n\}$ do
Create_File(FPP_i);
EndFor;
For each $x_i \in I = \{x_1, x_2, ..., x_n\}$ do
SP_Tree($FPD_i, Root$);
SP_Mining($Root$);
EndFor;

2.3.4. Thủ tục SPP_Mining

Khai thác các tập thường xuyên của cơ sở dữ liệu giao tác T theo ngưỡng S_0.

Sau khi phân hoạch xong các mục dữ liệu, việc giải bài toán ban đầu được chuyển thành giải n bài toán độc lập trên n tập dữ liệu $\{FPP_1, FPP_2, ..., FPP_n\}$ với cùng một ngưỡng tối thiểu S_0. Với n tập tin FPP_i và S_0 được xử lý trên n máy độc lập. Kết quả của bài toán là hội của các kết quả xử lý trên n máy.

Procedure SPP_Mining(F, S_0, ItemList, Pre_DB);
Input: $F = \{FPP_1, FPP_2, ..., FPP_n\}$ là n tập tin phân hoạch mục dữ liệu;
S_0: Ngưỡng hỗ trợ tối thiểu; Mảng ItemList với ItemList$[i]$ = Supp(x_i);
Output: Pre_DB = $\{(X, \text{Supp}_X) | X \geqslant I$ và $\text{Supp}_X = \text{Supp}(X) \geqslant S_0\}$.
Method:
Pre_DB = $\{(\{x_i\}, \text{ItemList}[i]) | x_i \in I$ và ItemList$[i] \geqslant S_0\}$; //Tập một mục DLTX
For each FPP_i in F do
If ItemList$[i] \geqslant S_0$ then
LM:= Tập các mục DL có trong FPP_i;
$k := 1$;
$L_1 := \{\{x_i\} | x_i \in LM$ và ItemList$[i] \geqslant S_0\}$.
While $L_k \neq \phi$ do
$C_{k+1} := \{X \subseteq LM | \|X\| = k+1, \forall Y \subseteq X$ thỏa $\|Y\| = k \Rightarrow Y \in L_k\}$;
$L_{k+1} := \phi$;
For each $X \in C_{k+1}$ do
$Sup := 0$;
For each $(Z, Sup\_Z) \in FPP_i$ do
If $X \subseteq Set(Z)$ then // $Set(Z)$: tập hợp các mục dữ liệu có trong Z.
$Sup := Sup + Sup\_Z$;
EndIf;
EndFor;
If $Sup \geqslant S_0$ then
$L_{k+1} := L_{k+1} \cup \{X\}$;

$X := \{x_i\} \cup X$;
Pre_DB := Pre_DB $\cup\{(X, Sup)\}$;
EndIf;
EndFor;
$k := k + 1$;
EndWhile;
EndIf;
EndFor;

Ví dụ 2. Cho cơ sở dữ liệu giao tác:

TID	Mục dữ liệu
1	f, a, c, d, g, i, m, p
2	a, b, c, f, l, m, o
3	b, f, h, j, o
4	b, c, k, s, p
5	a, f, c, e, l, p, m, n

Bước một: Phân hoạch các giao tác

- Tính độ hỗ trợ của từng mục dữ liệu và xếp thứ tự giảm dần theo độ hỗ trợ:

ItemList =	f:4	c:4	a:3	b:3	m:3	p:3	l:2	o:2	d:1	e:1	g:1	h:1	i:1	j:1	k:1	n:1	s:1

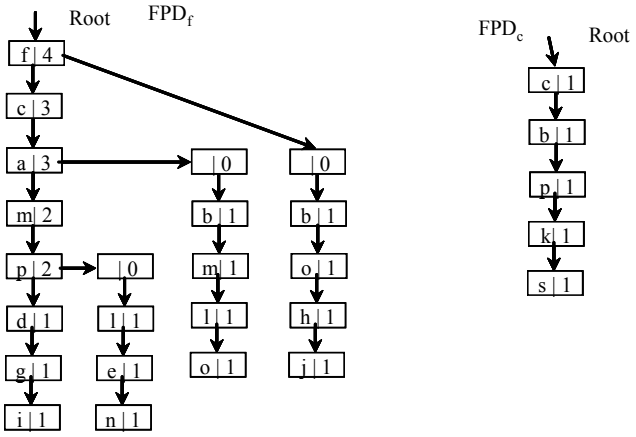
- Phân hoạch dữ liệu thành hai lớp:

FPD_f
f, c, a, m, p, d, g, i
f, c, a, b, m, l, o
f, b, o, h, j
f, c, a, m, p, l, e, n

FPD_c
c, b, p, k, s

Bước hai: Phân hoạch các mục dữ liệu

- Xây dựng SP_Tree cho hai lớp trên (Hình 2).
- Phân hoạch mục dữ liệu:



Hình 2. SP_Tree

Với SP_Tree ở Hình 2, các tập mục dữ liệu được phân hoạch và lưu thành 17 file như sau:

Bảng 1. Các file FPP_i của cây ở Hình 2

FPP_f	FPP_c	FPP_a	FPP_b	FPP_m	FPP_p	FPP_l	FPP_o	FPP_d
	f:3	c,f:3	a,c,f:1	a,c,f:2	m,a,c,f:2	p,m,a,c,f:1	l,m,b,a,c,f:1	p,m,a,c,f:1
			f:1,c:1	b,a,c,f:1	b,c:1	m,b,a,c,f:1	b,f:1	

FPPe	FPPg	FPPh	FPP_i	FPP_j	FPP_k	FPP_n	FPP_s
l,p,m,a,c,f:1	d,p,m,a,c,f:1	o,b,f:1	g,d,p,m,a,c,f:1	h,o,b,f:1	p,b,c:1	e,l,p,m,a,c,f:1	k,p,b,c:1

Bước ba: Áp dụng thuật toán FP_Mining để tìm tất cả các tập thường xuyên theo ngưỡng $S_0 = 2$.

Với $S_0 = 2$, ta chỉ tìm trong FPP_f, FPP_c, FPP_a, FPP_b, FPP_m, FPP_p, FPP_l, FPP_o. Để đơn giản cách viết, dùng ký hiệu *abc* thay cho $\{a, b, c\}$ và (X, Su) có nghĩa là $\text{Supp}(X) = Su$. Gọi Pre_FPP_i là các tập mục dữ liệu thường xuyên trong file FPP_i.

Với FPP_f ta có Pre_Ff = {(f, 4)}.

Với FPP_c ta có Pre_Fc = {(c, 4), (cf, 3)}.

Với FPP_a ta có Pre_Fa = {(a, 3), (ac, 3), (af, 3), (acf, 3)}.

Với FPP_b ta có Pre_Fb = {(b, 3), (bc, 2), (bf, 2)}.

Với FPP_m ta có Pre_Fm = {(m, 3), (ma, 3), (mc, 3), (mf, 3), (mac, 3), (maf, 3), (macf,3)}.

Với FPP_p ta có Pre_Fp = {(p, 3), (pm, 2), (pa, 2), (pc, 3), (pf, 2), (pma, 2), (pmc, 2), (pmf, 2), (pac, 2), (paf, 2), (pcf, 2); (pmac, 2), (pmaf, 2), (pacf, 2), (pmacf, 2)}.

Với FPP_l ta có Pre_Fl = {(l, 2), (lm, 2), (la, 2), (lc, 2), (lf, 2), (lma, 2), (lmc, 2), (lmf, 2), (lac, 2), (laf, 2), (lcf, 2), (lmac, 2), (lmaf, 2), (lacf, 2), (lmacf, 2)}.

Với FPP_o ta có Pre_Fo = {(o, 2), (ob, 2), (of, 2), (obf, 2)}.

Suy ra, Pre_DB = {(f, 4), (c, 4), (cf, 3), (a, 3), (ac, 3), (af, 3), (acf, 3), (b, 3), (bc, 2), (bf, 2), (m, 3), (ma, 3), (mc, 3), (mf, 3), (mac, 3), (maf, 3), (macf, 3), (p, 3), (pm, 2), (pa, 2), (pc, 3), (pf, 2), (pma, 2), (pmc, 2), (pmf, 2), (pac, 2), (paf, 2), (pcf, 2), (pmac, 2), (pmaf, 2), (pacf, 2), (pmacf, 2), (l, 2), (lm, 2), (la, 2), (lc, 2), (lf, 2), (lma, 2), (lmc, 2), (lmf, 2), (lac, 2), (laf, 2), (lcf, 2), (lmac, 2), (lmaf, 2), (lacf, 2), (lmacf, 2), (o, 2), (ob, 2), (of, 2), (obf, 2) }.

3. ĐÁNH GIÁ THUẬT TOÁN

Để đánh giá thuật toán, chúng tôi đã tạo ra các ma trận giao tác ngẫu nhiên với số hàng (tổng số giao tác) và số cột (số mục dữ liệu) chọn trước với độ dày của ma trận (tỷ số giữa tổng số ô có giá trị 1 và tổng số ô của ma trận) tùy ý. Dữ liệu này bảo đảm tương ứng với dữ liệu bán hàng của một siêu thị.

Ba thuật toán Apriori, thuật toán FP_Tree và thuật toán SP_Tree đã được cài đặt và chạy trên máy IBM ThinKPad T43. Hiệu quả của chúng được đánh giá thông qua việc so sánh thời gian thực hiện (giây) trên cùng một tập dữ liệu được thể hiện ở hai bảng dưới đây.

Bảng 2. So sánh thời gian xây dựng cây FP_Tree và SP_Tree (giây)

STT	Tổng số giao tác	Số mục DL	FP_Tree	SP_Tree	Tỷ lệ giảm
1	10000000	25	483	203	2.4
2	5000000	50	68875	307	224.3
3	1000000	100	Tràn bộ nhớ	413	
4	2000000	100	Tràn bộ nhớ	789	
5	10000000	250	Tràn bộ nhớ	3272	

Ta nhận thấy, khi số mục dữ liệu lớn, SP_Tree rất hiệu quả. Khi dữ liệu đủ lớn thì FP_Tree không thể thực hiện được vì tràn bộ nhớ (hàng 3, 4, 5 trong Bảng 2), SP_Tree vẫn thực hiện được, điều này có nghĩa là SP_Tree tốn ít bộ nhớ hơn rất nhiều, có khả năng xử lý những dữ liệu lớn.

Bảng 3. So sánh thời gian chạy của hai thuật toán Apriori và SPP_Mining

STT	Tổng số giao tác	Số mục DL	Ngưỡng tối thiểu S_0	Tổng số mục DLTX	Thời gian tính (giây)		Tỷ lệ giảm
					Apriori	SP_Mining	
1	65.000	50	4.000	863	15	5	3.00
			3.000	1977	51	8	6.38
			2.000	4250	141	16	8.81
			1.000	18295	1908	53	36.00
2	1.000.000	50	40.000	2066	598	171	3.50
			30.000	3841	1263	251	5.03
			20.000	9907	2577	413	6.24
			10.000	31023	15608	1251	12.48
3	2.000.000	25	100.000	270	69	6	11.50
			50.000	691	234	7	33.43
			30.000	1386	404	12	33.67
			10.000	6045	1806	42	43.00
4	10.000.000	25	100.000	2527	3866	45	85.91
			50.000	5801	8943	108	82.81
			30.000	10606	13383	190	70.44
			20.000	16426	22170	294	75.41

Ta nhận thấy rằng, khi dữ liệu lớn, thuật toán SPP_Mining càng hiệu quả.

4. KẾT LUẬN

Với kỹ thuật xây dựng SP_Tree và thuật toán SPP_Mining khai thác các tập thường xuyên, cơ bản giải quyết được những tồn tại của ba hướng tiếp cận kể trên.

1) Thuật toán chỉ có hai lần duyệt qua cơ sở dữ liệu. Sau khi xây dựng xong các SP_Tree, một lần duyệt qua cây là xác định được tất cả các độ hỗ trợ của các mục dữ liệu trong phân hoạch, không phải đếm số lần chúng xuất hiện trong từng giao tác.

2) Trong quá trình giải bài toán, khi phân hoạch dữ liệu ở bước 2 và 3, thuật toán SPP chưa xét đến độ hỗ trợ, đến bước 4, tìm các tập mục dữ liệu thường xuyên mới xét đến độ hỗ trợ và chỉ xử lý trên các tập dữ liệu phân hoạch. Do đó, trong quá trình khai thác dữ liệu với các độ hỗ trợ khác nhau, nếu dữ liệu chưa thay đổi thì không phải phân hoạch lại dữ liệu, giảm thiểu thời gian đáng kể.

3) Thuật toán SPP chia nhỏ dữ liệu và xử lý trên tập tin nên độ lớn dữ liệu tùy thuộc

vào dung lượng bộ nhớ ngoài, do đó, khi giải bài toán không cần đến máy tính có bộ nhớ trong lớn.

4) Mỗi tập FPP_j là một bộ (X, Sup_X), với Sup_X là trọng số của X trong T, số phần tử của FPP_j (là tập các hậu tố của x_j, không có thuộc tính x_j) nhỏ hơn nhiều so với T. Hơn nưa, có nhiều x_j không có hậu tố, tức FPP_j rỗng, khi áp dụng thuật toán Apriori để khai thác tập thường xuyên sẽ giảm khối lượng tính toán rất lớn.

5) Khi tìm các tập mục dữ liệu thường xuyên, SPP tìm trên từng tập FPP_i riêng rẽ nên dễ dàng cài đặt chương trình song song thực hiện trên các máy tính đa bộ xử lý hoặc trên n máy độc lập, do đó tốc độ xử lý nhanh chóng hơn, hiệu quả hơn.

Tuy nhiên, thuật toán SPP chỉ xử lý trên dữ liệu xác định, chưa xử lý dữ liệu tăng trưởng. Trong những bài báo tiếp theo, tác giả sẽ đề nghị thuật toán khai thác luật kết hợp trên cơ sở dữ liệu tăng trưởng.

TÀI LIỆU THAM KHẢO

[1] Qiankun Zhao, Sourav S. Bhowmick, Association rule mining: A survey, *Technical Report, CAIS*, Nanyang Technological University, Singapore, No. 2003116, 2003.

[2] R. Agrawal, T. imielinski, A. swami, Mining associations between sets of items in massive databases, *Proc.of the 1993 ACM-SIGMOD Intl Conf. on Management of Data* 1993 (207–216).

[3] R. Agrawal, H. Mannulla, R. Srikant, H. Toivonen, A. I. Verkamo, Fast discovery of association rules, *Advances in Knowledge Discovery and Data Mining*, AAAI Press, 1996 (307–328).

[4] J. Han, H. Pei, and Y. Yin, Mining frequent patterns without candidate generation, *Proc. Conf. on the Management of Data (SIGMOD00, Dallas, TX)* ACM Press, New York, NY, USA 2000 (1–12).

[5] Claudio Lucchese, Salvatore Orlando, Raffaele Perego, Fast and memory efficient algorithm to mine frequent closed itemsets, *IEEE Transaction On Knowledge and Data Engineering* **18** (1) (January 2006) 21–36.

[6] J. Mohammed Zaki and Ching-Jui Hsiao Charm, Efficient algorithm for mining closed itemsets and their lattice structure, *IEEE Transactions On Knowledge And Data Engineering* **17** (4) (April 2005). http://www.cs.rpi.edu/_zaki.

[7] Takeaki Uno, Masashi Kiyomi, Hiroki Arimura Lcm, ver.2: Efficient mining algorithms for frequent/closed/maximal itemsets, *IEEE ICDM04 Workshop FIMI04 (International Conference on Data Mining, Frequent Itemset Mining Implementations)* (2004) http://research.nii.ac.jp/ũno/papers/0411lcm2.pdf.

[8] Vicky Choi, Faster algorithms for constructing a concept (Galois) lattice, *arXIV:cs.DM/0602069* **2** (1) (Jun 2006).

[9] A. Savesere, E. Omiecinski,, and S. Navathe, An efficient algorithm for mining association rules in large databases, *Proceedings of 20th International Conference on VLDB*, 1995 (432–444)

[10] Shakil Ahmed, Frans Coenen, and Paul Leng, Tree-based partitioning of data for association rule mining, *Knowledge and Information Systems* **10** (3) (2006) 315–331.

Nhận bài ngày 6 - 3 - 2007

Nhận lại sau sửa ngày 25 - 7 -2007