幾個快速挖掘關聯規則的資料探勘方法

陳彥良　　趙書榮　　陳禹辰

中央大學資訊管理系　中央大學資訊管理系　東吳大學企業管理系

摘要

關聯規則的挖掘，是目前最重要的資料挖掘問題之一，它的目的是要從銷售的交易資料庫中，發現商品項目間的關聯。在過去已經有相當多挖掘關聯規則演算法被提出來，當中 FP-tree 演算法可說是最主要的演算法之一，並以高執行效率著稱。它的主要概念是不產生 candidate itemsets，而將資料庫壓縮在 FP-tree 的結構中，以避免多次的高成本資料庫掃瞄。在本文中，我們針對原本的 FP tree 演算法，更進一步改進其所用的資料結構以提高挖掘效率。

在本文中共建立了三種資料結構：(一)FP-tree_tail 演算法，也就是在 head table 中增加一個 tail 欄位，(二)FP-tree_hash 演算法，乃是以 hash function 計算出每個 node 所在位置方式建立 FP-tree，(三)FP-tree_hash+tail 演算法，為結合(一)、(二)之優點，所完成之演算法.。作者將以上三個演算法與傳統 FP-tree 演算法一起比較，以找出各演算法之優缺點。經由各種實驗數據發現，傳統 FP-tree 演算法所需花費之時間，為三個改良 FP-tree 演算法的數十倍。

關鍵詞：資料挖掘、關聯規則、交易資料庫

Several Improved Data Mining Algorithms For Finding Association Rules

Yen-Liang Chen and Shu-Rong Zhao

Department of Information Management
National Central University

Yu-Chen Chen
Department of Business Administration
Soochow University

Mining association rules is one of the most important problems in data mining. Its aim is to discover the associations between items in a large database of sales transactions. In the past, a large number of algorithms for mining association rules have been proposed, and the FP-tree algorithm is one of the most famous ones, known for its efficiency. Unlike the traditional approach that requires many phases of candidate itemsets generation and database scan, the FP-tree algorithm compresses and stores the entire database into a sophisticated tree structure, called FP-tree, by which all the associations can be found by two database scans. In this paper, we attempt to further improve the standard structure of the FP-tree such that the mining performance can be improved. To this end, three variants of the improved FP-tree algorithm are proposed. The first variant is called FP-tree_tail, which adds a tail pointer into the head table of the original FP-tree structure. The second is named as FP-tree_hash, which adds a hash table into every node of the FP-tree. Finally, we call the last FP-tree_hash+tail, which is a combination of the first two improvements. Finally, a performance evaluation is done to compare their performances. The result indicates that the three proposed algorithms are about 20-50 times faster than the original FP-tree algorithm.

Keywords：Data mining, Association rule, Transaction database

壹、 緒論

現今是一個資訊爆炸的時代，隨著各種組織機構的全面電腦化，加上網際網路的蓬勃發展，資料的產生及流通皆快速地成長。如何從這些龐大的資料量中，快速有效地找出有用的資訊並加以利用，已成為當前管理者的重要課題。以傳統交易性資料庫為例，每天均有相當大量的交易發生，經年累月累積下來的資訊頗為可觀，根本無法由人工進行分析來找出商品交易記錄間的相關性，然而這些交易記錄事實上可能隱含了許多有用的資訊在裡面，如果能運用適當的方法將它找出來，便可能發現商機，創造利潤。

因為使用者很難從資料庫中的大量資料找到可以利用的隱含資訊，挖掘資料間關聯規則的演算法，就在這種需求驅使下產生出來了。關聯規則的挖掘，便是在龐大的資料中，找出不同資料項目間的相關性。關聯規則挖掘的第一步驟就是找出 large itemset，其作法則可以分為：要產生 candidate itemsets 與不產生 candidate itemsets 兩種類型，而 FP-Tree 是不產生 candidate itemsets 作法的代表。從文獻中得知，因為不用產生 candidate itemsets，所以只需掃瞄資料庫兩次，可以避免多次的高成本的資料庫掃瞄，節省了大量 I/O 的時間，因此整體的效率相當不錯，大幅領先要產生 candidate itemsets 的演算法。

有三個原因使得 FP-Tree 的績效能大幅領先原有的演算法：（一）它把原資料庫的所有資訊巧妙的儲存在樹狀結構與指標鏈結串列中；（二）它設計了一個有效率的方法將關聯規則從資料結構中粹取出來；（三）它只需掃描資料庫兩次。而在本文中，我們將針對第一點，即針對其原先的資料結構做進一步改良，以尋求其效率的提昇。本研究在這個基礎之上，以兩種不同資料結構技術來改良 FP-Tree 演算法，並發展三種實做方法，分別是：

1) 在每個 head node 加入一個 tail 欄位，演算法名稱為 FP-tree_tail
2) 以 hash 方式建立 FP-tree 演算法，名稱為 FP-tree_hash
3) 綜合上述 tail 與 hash 方式所完成之演算法，名稱為 FP-tree_hash+tail

本研究進行各種實驗，比較 FP-tree_hash、FP-tree_hash+tail、FP-tree_tail 以及單純樹狀結構 FP-Tree 演算法四者執行時間效能之優劣。結果顯示，單純樹狀結構 FP-Tree 演算法的執行時間，大幅落後於以前述三種新的資料結構完成之演算法達數十倍之多。單純樹狀結構 FP-Tree 演算法執行時間，完全無法與三個改良版之 FP-tree 演算法相抗衡。

貳、文獻探討

在挖掘 association rules 的領域中，主要可以分成兩大類：(1)利用 Apriori-like 的方法產生 candidate set，並找出符合 minimum support 的 large itemsets，再依據 large itemsets 產生 association rules；(2)使用 Non Apriori-like 的方法，找出 large itemsets。

第(1)類的方法中是以 Apriori 演算法**[1]**為基礎，它們共同的特點是第一次的 candidate set（以 C_1 表示）是直接利用掃描資料庫一次直接得到，其他的 C_k（$k>1$）產生方式都包含了兩個主要步驟，第一個是合併產生 candidate set，第二個則是將這些 itemsets 中，含有不是前一次作業的 large itemsets 者去除，然後針對這些留下來的 candidate itemsets，以掃描資料庫的方式獲取其 support 值，再將未滿足 support 值要求的 itemsets 去除掉，即得到所謂的 large itemsets。

由於 Apriori 反覆處理 candidate itemsets 要花費很多時間，之後陸續有一些改良 Apriori 演算法的方式提出來，DHP**[2]**利用 hash function 建立 hash table 來達到減少 candidate itemsets 數量的目的；Partition**[3]**將交易分割成一些沒有重疊的 partitions 在主記憶體運作以增快速度；DIC**[4]**（Dynamic Itemset Counting）把 itemsets 存入 lattice 中並分割 transaction sequence 來達到儘早決定 large itemsets 的目的；Random sampling**[5][6]**則以抽樣的方式找出合適大小的樣本，再針對這些樣本迅速找出 large itemsets。

由於第(1)類的方法在檢驗 candidate set 是否為 large 時，都需要重新掃描資料庫一次，需要大量 I/O 的時間，即使做了一些改良，所能提昇的效率仍然有限，

所以最近的研究都不利用這種方式，改以直接快速的方法來得到 frequent itemsets，我們將它們都歸類為第(2)類的方法。第(2)類的方法不利用 Apriori 的原理來找出 frequent itemsets，Max-Miner**[7]**運用 set-enumeration tree 的搜尋架構配合 subset infrequency 和 superset frequency 作 pruning 來迅速找出 maximal frequent itemsets；Close Algorithm**[8]**利用 closed itemset lattice 遠小於 itemset lattice 的特性，以它取代 itemset lattice 進行 pruning，可以減少 database 存取次數和 CPU 的負擔；graph-based approach**[9]**以 vertical 的方式用 bit vector 建構 association graph，並由 graph 產生所有的 large itemsets；TreeProjection**[10]** 建構一個 lexicographical tree 做 candidates 的 counting 並提供快速且有彈性的挑選策略；FP-growth**[11]**掃描資料庫兩次建立 FP-tree 後，從 FP-tree 可快速找出 frequent patterns。

在第(2)類方法中，TreeProjection**[10]**和 FP-growth**[11]**是目前我們看到的文獻中最快的兩種方法，其中又以 FP-tree（Frequent Pattern Tree）的結構結合 FP-growth algorithm 的作法效率最高，所以 FP-growth 可說是目前我們所看到的研究中最新、最有效率也最為著名的方法， DBMINER 便是實際運用 FP-tree 的產品。於挖掘單一維度及布林值 association rules 的領域中，FP-tree 是相當有用的結構，在進行 association rule、max-patterns、sequential patterns 以及 Constrained Frequent Pattern 挖掘，都可以結合 FP-tree 演算法，達到很好的執行效率**[12][13][14]**。

FP-tree 演算法的作法可分為兩個階段，第一階段建立 FP-Tree，第二階段是 mining FP-Tree。主要步驟如下：

1)建立 FP-Tree：

1. 第一次掃瞄資料庫，找出符合 minimum support 的 large 1-itemset。
2. 將每一筆紀錄中 large items 依其出現在資料庫中頻率的次數，作降冪排列。
3. 第二次掃瞄資料庫時，建立 FP- tree。

2)mining FP-Tree：

1. 對 FP-tree 中的每一個分支 node，建立 conditional pattern base。
2. 再對每一個 conditional pattern base 分別建立其 conditional FP-tree。
3. 再對 conditional FP-tree 進行挖掘，並逐次增加包含在 conditional FP-tree 的 frequent pattern。
4. conditional FP-tree 中有包含一條路徑，就可列舉出所有 pattern。

參、問題說明與方法

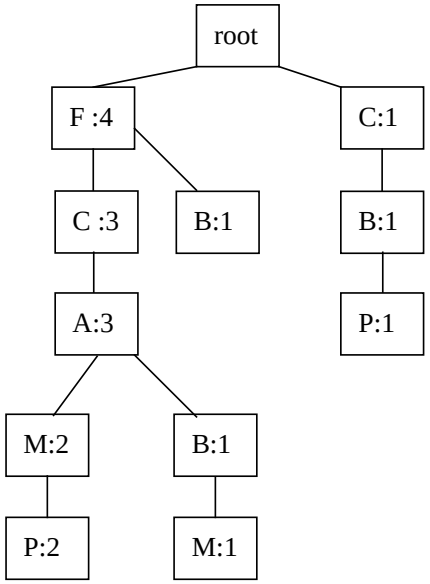
一、新增 Head table 串列 item 問題

在 FP-Tree 作者的原文中, 會產生出一顆 FP-tree 與一個 head table；head table 記錄著每一個排序過的 large 1 item, 每一個 large 1 item 再以 linked-list 將 FP-tree 中相同的 item 串聯起來，以建立其 conditional FP-tree，而後再對 conditional FP-tree 進行挖掘。

在上述以 linked-list 串聯相同 item 的過程中有一個問題產生, 亦即當每次在 FP-tree 中找到與 head table 有相同的 item 時，必須回到 head table 中，循序找出最後一個 linked-list 的 item 後, 再將新找到的 item 加入 linked-list 中。若在 linked list 中有 n 個 item, 則這個做法所花費之時間為 O(n), 當資料量大的時候效率並不佳。

圖一所示即為 FP-tree 建立之過程。首先自 transaction database 中針對每筆 record 找出來相對應的 large 1 itemset （即圖一之 Ordered frquent items 欄位之 large 1 itemset), 再依 Ordered frquent items 欄位之 large 1 itemset 逐筆建立 FP-tree 之各分支，並記錄每一個分支 node 之 item 出現的次數，即可完成如圖一所示的 FP-tree。

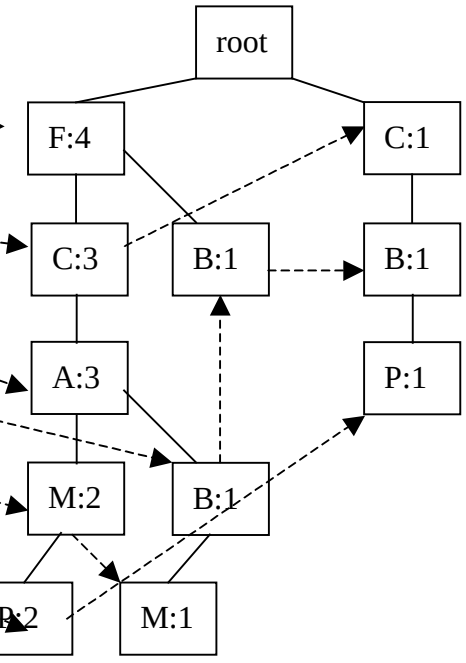
PID	Items	Ordered frquent items
100	f,a,c,d,g,I,m,p	f,c,a,m,p
200	a,b,c,f,l,m,o	f,c,a,b,m
300	b,f,h,j,o	f,b
400	b,c,k,s,p	c,b,p
500	a,f,c,e,l,p,m,n	f,c,a,m,p



圖一：原先 FP-tree 的結構

事實上，在第二階段把資料加入 FP-tree 的同時，我們會建立 head table，及 head table 相對應 item 的 linked-list 鏈結，如下圖二中所示：

Item	Head of node links
F	
C	
A	
B	
M	
P	



圖二：原先的 FP-tree 加入指標串列

如果此時要新增一筆資料為 a, b, m 到圖二的 FP-tree，則我們必須找到 a 的 linked list 的尾巴才能加入新的 item a，同理必須找到 b 的 linked list 的尾巴才能加入新的 item b，必須找到 m 的 linked list 的尾巴才能加入新的 item m，如此才能完成此一資料的新增。所以我們在這過程中，必須從頭到尾循序走完這三個 linked list，不然我們就無法完成更新。

在 head table 的 node 中加上一個 tail 欄位，就可以解決循序搜尋 linked list 的浪費。tail 欄位中記錄著每一個 item 的最後一個值，當 head table 要加入一個新找到的 item 時，就不需要循序一個一個搜尋以找出最後一個 linked list ，而可以直接將新增的 item 放到 tail 欄位中，並調整 tail 欄位及 next 指標即可。

二、以樹狀結構建立 FP-Tree 問題

建立 FP-Tree 的過程存在另外一個問題：當我們要在 FP-tree 樹狀結構中插入一筆新的資料時，例如要插入 f, c, a, p 到圖二中，我們必須逐層往下搜尋，從 root 走到節點 F:4，從 F:4 往下走到 C:3，從 C:3 往下走到 A:3，最後在 A:3 我們發現它沒有辦法再往下。在這逐層往下搜尋的過程中，我們必須比較所有子節點，若有某一個子節點的 item 跟我們所要找的相合，則往該子節點繼續搜尋，不然就必須新增一個新的子節點。如果某一個節點有 n 個子節點，則每一次為了要確定是不是有相同 item 的子節點，我們必須花費之比較時間為 O(n)，效率不佳。

擬解決此問題，可改利用 hash function 來算出來每一個 node 位置。hash 方式可以省去循序的一個一個比對建立之時間，而直接計算得出該 node 所在之位置，其所花費之時間為 O(1)，可以節省相當多的時間。

三、演算法

依據上述兩種問題，設計了三個不同演算法：（一）FP-tree_tail，（二）FP-tree_hash，（三）FP-tree_hash+tail，描述如下。

FP-tree_tail 演算法跟傳統的 FP-tree 演算法基本上一切都相同，唯一的差別是我們在 head table 的每一個 item 欄位中，除了原有的 head 指標外，另外增加了一個 tail 指標。當我們要增加一筆資料到 FP-tree 中時，我們必須往下逐層搜尋，若在某一節點上，若發現它的所有子節點的 item 都和要新增的 item 不合時，我們就必須把該新增的 item 串接到此一 item 的 linked list 的尾巴。在傳統的做法，我們會必須走完該 linked list 的之後，才能找到尾巴並加以串接。但在 FP-tree_tail 演算法中，透過 tail 指標，我們可以立即找到尾巴，並更正相關的指標位置。

FP-tree_hash 演算法跟傳統的 FP-tree 演算法基本上一切都相同，唯一的差別是我們在每個 FP-tree 樹上的節點都建置了一個 hash table，因此會提高我們在 FP-tree 中巡行的效率。當我們要增加一筆資料到 FP-tree 中時，我們必須往下逐層搜尋，也即在每一節點上，我們都必須確定它有沒有一個子節點其 item 是和我們要新增的 item 相同的；若有相同的子節點，則往該子節點繼續搜尋，不然我們就必須把該新增之 item 串接到其相關的 linked list。傳統的做法因為沒有 hash 結構，必須是逐一比較所有的子節點。但在我們的演算法中，我們可以透過 hash 函數直接搜尋。

我們在每一個節點都建置了一個可以儲存 32 個指標的 hash table，而每一個 item 的後五位 bit 數值即構成它的 hash value。因此在一個節點上，如果要確認它有沒有一個子節點的 item 是 a，我們就用 a 去求算其 hash value，接著我們就可以往該子節點去檢查看 a 是否存在於該子節點上。如果有多於一個以上的 item 被 hash 到同一子節點，我們就以 linked list 方式再加以串接。

FP-tree_hash+tail 演算法則是綜合了上述的兩個方法。

四、實驗模擬

一、實驗設計

本研究進行一系列的實驗以評估前述演算法的效率。實驗的環境為 windows 2000 professional，硬體為 Intel Pentium 4-1.8GHz CPU 及配備 1024 MB DDR-RAM，此外程式是以 Boland C++ Builder 5 撰寫。

實驗用的測試資料使用模擬真實環境的交易資料，每一個 item 以一個數字代表，本文是以[1]文中所使用的方式來產生測試資料，此方法為多數挖掘關聯規則的研究所援用。在模擬資料中我們定義了以下的參數：

D：交易資料之總筆數。

T：每筆交易紀錄之平均長度。

N：交易資料庫中 item 的個數。

I：潛在 large itemset 的平均長度

一般而言，下列四方面因素會影響執行時間：(一)交易長度變化，(二) 不同的資料量，(三) minimum support 門檻值大小，(四)潛在 large itemset 的平均長度。這四個原因都會使 FP-tree 的節點增加。本研究針對這四點，測試比較 FP-tree 、FP-tree_tail、FP-tree_hash、FP-tree_hash+tail 等四個演算法之執行時間效率。

由圖三之實驗數據發現，當輸入資料量只有 400K 時，在五個不同 minimum support 值測試下，FP-tree 演算法的計算時間均落後給其他三個改良版，至少有 22 倍，最多時達 55 倍之多。由圖三可以看出，FP-tree 演算法效能完全不及 FP-tree_tail、FP-tree_hash、FP-tree_hash+tail 三個演算法，其所花費之時間均超過 5 分鐘以上；相對而言，本研究的三個演算法之執行時間僅需數十秒。故本文後續之實驗只針對 FP-tree_tail、FP-tree_hash、FP-tree_hash+tail 三個演算法相互比較，而排除傳統以樹狀結構所做之 FP-tree 演算法。

	各演算法花費時間（秒）				FP-tree 與其他三演算法之比值(FP-tree/other)		
minsup	FP-tree_hash+tail	FP-tree_hash	FP-tree_tail	FP-tree	FP-tree_hash+tail	FP-tree_hash	FP-tree_tail
0.25	22.06	24.69	23.16	1076.21	48.78	43.60	46.46
0.5	19.78	21.95	20.98	1032.39	52.20	47.03	49.21
0.75	16.55	18.12	18.37	912.09	55.10	50.35	49.66
1	13.19	14.37	15.56	716.94	54.38	49.89	46.07
1.5	7.34	7.87	9.53	215.90	29.41	27.43	22.65

圖三： N=1000 T=10 I=4 D=400K

以下的模擬實驗，若未特別說明，採取下列標準參數設定：交易資料庫中 item 的個數 N=1000，每筆交易紀錄之平均長度 T=10，潛在 large itemset 的平均長度 I=4，輸入資料量=500K，minimum support=1%。本文所列之測試實驗名稱，及相關測試點，如圖四所示。

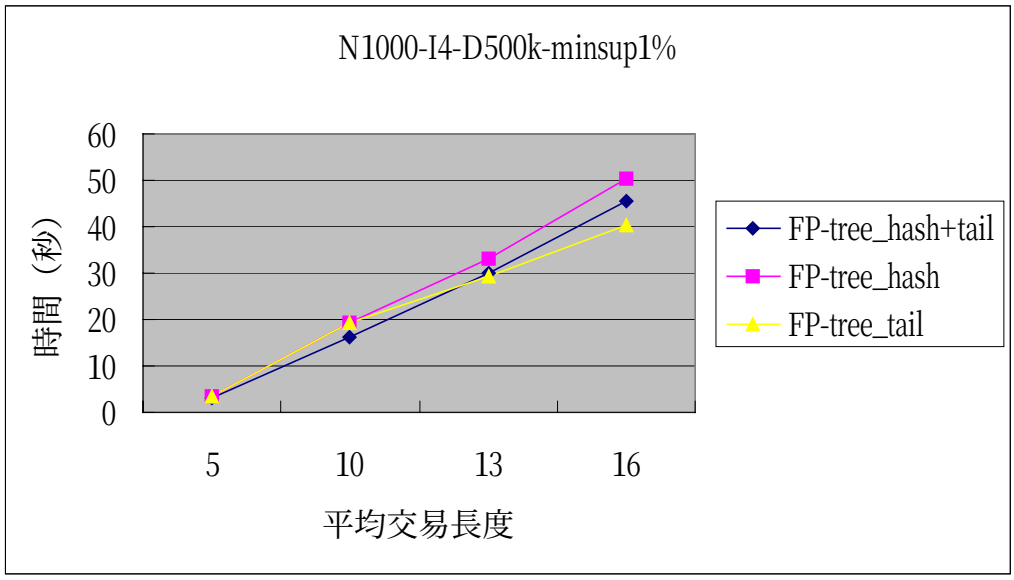
實驗名稱	測試點 1	測試點 2	測試點 3	測試點 4	測試點 5
平均交易長度	5	10	13	16	
不同的資料量	400K	500K	600K	700K	
不同的 minimum support	0.25	0.5	0.75	1	1.5
不同的的平均長度 I	2	4	6	8	

圖四：模擬測試實驗設定

二、結果分析

依據圖四所設定的四個不同實驗，產生出下列圖五~圖八之結果：

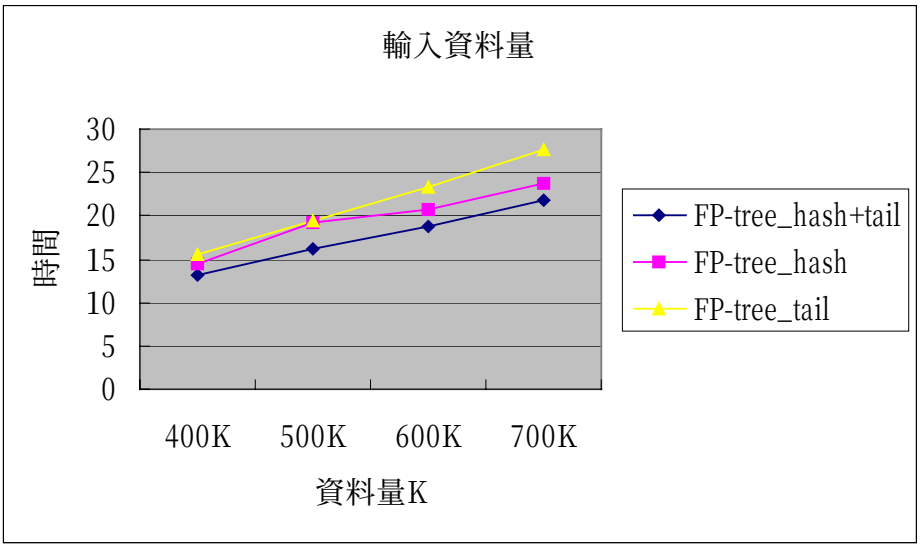
實驗一　不同平均交易長度之三者比較



圖五：不同平均交易長度之比較

本次實驗中資料庫的參數為 N=1000, I=4, D=500K, minimum support=1%。由圖五可得知，FP-tree_hash+tail 在平均交易長度十以內的效率，優於 FP-tree_hash，FP-tree_tail 演算法，但超過長度 10 以後，在主記憶體為 1024MB 的環境中，FP-tree_hash，FP-tree_hash+tail 演算法因節點數目大幅增加，效率就明顯變差。原因在於一筆長度為 n 的交易記錄，就有 $(2^n - 1)$ 個子集合，每一個子集合都需要一個節點來表示，所以可以得知，當交易長度愈長，就代表交易紀錄的子集合越多，故 FP-tree_hash＋tail、FP-tree_hash、FP-tree_tail 演算法之節點均增多，但是建構 FP-tree_hash+tail 與 FP-tree_hash 所需節點增多時，因為每一個節點都要維持一個 hash table，因此所佔記憶體的空間也越多，而 FP-tree_tail 不需要每一個節點都要維持一個 hash table，所以 FP-tree_hash+tail 、FP-tree_hash 在交易長度大時執行效率就不及 FP-tree_tail 演算法。

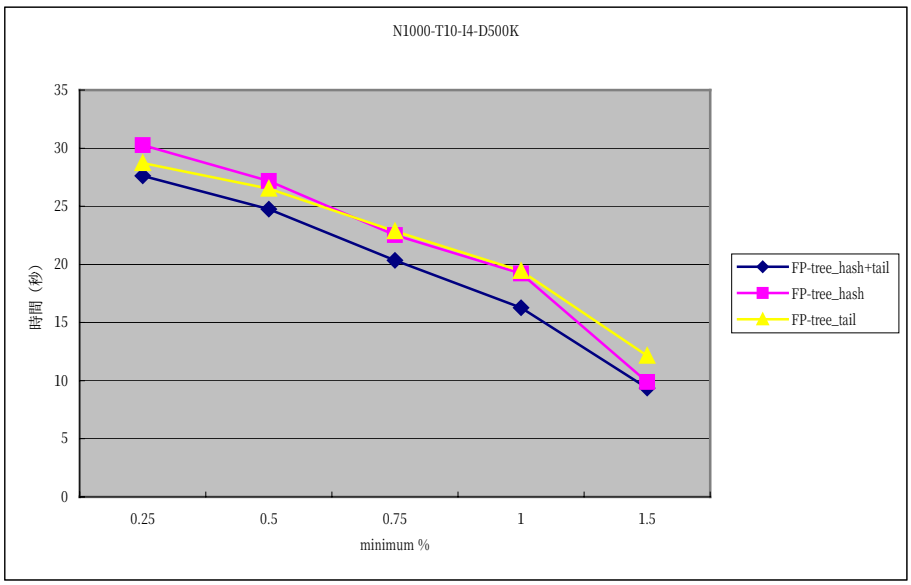
實驗二 不同的資料量之三者比較



圖六：不同資料量之比較

本次實驗中資料庫的參數為 N=1000，T=10，I=4，minimum support=1%。在平均交易長度固定情況下，節點並不會以指數方式大幅上升並逼近 1024MB 主記憶體之上限。因此，在記憶體足夠的情況下，交易資料量時增加，FP-tree_hash、FP-tree_hash+tail 演算法，均優於 FP-tree_tail 演算法；而 FP-tree_hash+tail 因為有 tail 欄位，可以省去在 head table 中循序搜尋之時間，故 FP-tree_hash+tail 在記憶體足夠的情況下均優於 FP-tree_hash 演算法。

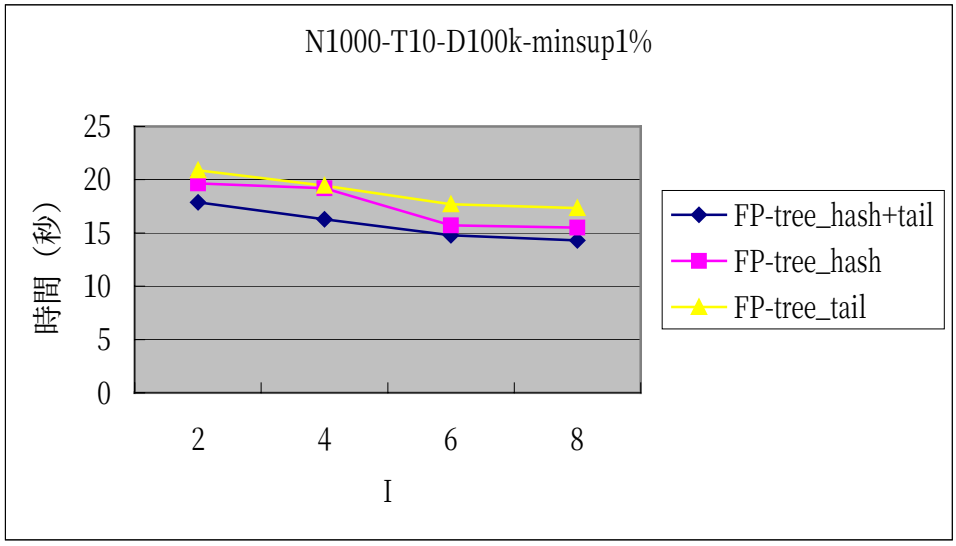
實驗三　不同的 minimum support 之三者比較



圖七：不同的 minimum support 之比較

由於在建構 FP-tree 時設定 minimum support 愈小，則所產生之 item 節點就愈多，樹也越大。因此一方面讓 hash 所提供的直接存取功能更加寶貴，但一方面也讓 hash table 的維護成本越高，兩個因素相互抵銷下，我們發現在 1024MB 記憶體範圍內，所有的情況下 FP-tree_hash+tail 均優於 FP-tree_hash 與 FP-tree_tail 演算法。

實驗四　不同潛在 large itemset 的平均長度 I 之三者比較



圖八：不同潛在 large itemset 的平均長度 I 之比較

當潛在 large itemset 的平均長度 I 變大時，會使得 FP-tree 變得越大。因此一方面使 hash 所提供直接存取更加有效，但一方面也讓 hash table 的維護成本變高，兩相抵銷下，我們發現所有的情況下 FP-tree_hash+tail 均優於 FP-tree_hash 與 FP-tree_tail 演算法。

三、實驗結論

由以上實驗得知，若只使用 FP-tree 原作者所述之單純樹狀結構，未用任何資料結構作改善的情況時，則在各種參數之下，其執行效率均不及 FP-tree_tail、FP-tree_hash，FP-tree_hash+tail 等三種修正後之演算法。主要的原因就是當每次在 FP-Tree 加入新的分支 Item 時，必須以循序比對搜尋方式來建樹，所花費之時

間為 O(n)。另外再建立 Head table 之連結時，亦須到 Head Table 中，先找出相同的 item，然後循序依每個節點的 link-list 所記錄之內容，一個一個去找出來，直到找出最後一個 Link-List 的 Item 後再加入，而此步驟所花費之時間亦為 O(n)。當資料量稍微增大的時候，FP-tree 的效率就非常不好。

FP-tree_tail，在 Head Table 的 Node 中加上一個 Tail 欄位，此欄位之功能在資料量少的時後效能比較看不出來，因為整個 FP-tree 之 node 不多，相對應在 head table 中的 item 也不多，因此只要比對幾次就會找到。但是當找出 pattern 數很多的時候，每次新加入一個 item 時，只要在 head table 中，找到 tail 欄位所指到的位置（最後一個 node），新的 item 再加入即可，不必再循該 item 之 link-list 找到最後一個，所以可大幅提升效率。相較於 FP-tree 演算法，只在 Head Table 的 Node 中多一個 Tail 欄位，memory 並不會佔用太多。

FP-tree_hash，利用 hash function 直接計算得出該 node 所在之位置，不必經過循序比對可以節省時間。但是與 FP-tree_tail 相比，較浪費 memory，因為建立 FP-tree 時之每一個 node，都要維持一個 hash table，相較於非 hash 樹狀結構 FP-tree，hash 方法是以空間換時間，但是在 head table 部份沒有增加 tail 欄位，所以在 pattern 數多的時候，效率與 FP-tree_tail 互有領先。

FP-tree_hash+tail，為結合 FP-tree_tail、FP-tree_hash 兩個演算法之優點，當電腦如果記憶體有足夠容量，則 FP-tree_hash+tail 在各種參數測試條件下為最好的演算法，但是平均交易長度 T>10 情況下，效率比 FP-tree_tail 差，因為 FP-tree_hash+tail 有以空間換時間之 hash 結構，所以當平均交易長度 T>10，就有 $\left(2^n-1\right)$ 個子集合產生（n 為平均交易長度），因而需要大量 memory，所以 FP-tree_hash+tail 執行時間效率，就不及 FP-tree_tail，但仍然比 FP-tree_hash 好。

總結來說，關於 FP-tree_hash+tail，FP-tree_hash，FP-tree_tail 三個演算法相互之比較，本文以 N=1000，T=10，I=4，D=400K~700K 情況下所產生之 pattern 個數作為比較基礎，得到結果如圖五至圖八所示，從這些測試結果可以歸納出下

列狀況,

(1) FP-tree_hash+tail 在 T 小於 10 的各種參數情況下，其效率優於 FP-tree_hash+tail，FP-tree_tail。

(2) FP-tree_tail 在 T 大於 10 以上, 效率優於 FP-tree_hash+tail, FP-tree_tail。

五、結論與建議

本文首先指出傳統 FP-tree 演算法執行時間較長的兩個狀況：一是建立 FP-tree 時以循序比對方式建 FP-tree 樹；另一個是在建 FP-tree 樹時, 到 head table 中，循序比對找對應之 item 並加入 head table 之 Link list 中。針對這兩種情形, 本文提出了 FP-tree_tail、FP-tree_hash、FP-tree_hash+tail 等三個改良的演算法, 經由本文的實驗模擬可得知，FP-tree_tail、FP-tree_hash、FP-tree_hash+tail 的三個改良的演算法於各種情況下，均顯著地優於 FP-tree 演算法（執行時間快了至少有 22 倍）。能大幅改善傳統 FP-tree 演算法執行時間較長的缺點。

從實驗數據中，可以規納出在記憶體足夠、T<=10，交易資料量增加的情況下，FP-tree_hash＋tail 演算法在各種參數測試下均明顯優於 FP-tree_hash、FP-tree_tail 演算法，在大部分測試參數下，它都是四種演算法（FP-tree、FP-tree_tail、FP-tree_hash、FP-tree_hash+tail）中最佳之演算法，但是在執行的效率上仍有改善空間，可以從二方面來改善：

1. hash function 之決定：如何快速算出 node 所在位置，並減少在記憶體發生碰撞之機會。
2. hash table 之大小與管理：可以有效減少記憶體所佔的空間，能擴大適用的資料庫範圍。

參考文獻

[1] R. Agrawal and R. Srikant. Fast Algorithms for Mining Association Rules. Proc. Int'l Conf. Very Large Data Bases, 487-499 (September 1994).

[2] J.S. Park, M.S. Chen, and P.S. Yu. An Effective Hash-Based Algorithm for Mining Association Rules. Proc. ACM-SIGMOD Int'l Conf. Management of Data, 175-186 (May 1995).

[3] A. Savasere, E. Omiecinski, and S. Navathe. An Efficient Algorithm for Mining Association Rules in Large Databases. Proc. Int'l Conf. Very Large Data Bases, 432-444 (Sept. 1995).

[4] S. Brin, R. Motwani, J. Ullman and S. Tsur. Dynamic Itemset Counting and Implication Rules for Market Basket Data. In Proc. of the 1997 ACM-SIGMOD Conf. on Management of Data, 255-264 (1997).

[5] Mohammed Javeed Zaki, Srinivasan Parthasarathy, Wei Li and Mitsunori Ogihara. Evaluation of sampling for data mining of association rules. Technical Report 617, Computer Science Dept., U. Rochester, (May 1996).

[6] G. Gunopulos, H. Mannila and S. Saluja. Discovering All Most Specific Sentences by Randomized Algorithms. In Proc. of the 6th Int'l Conf. on Database Theory, 215-229 (1997).

[7] J. Roberto and Jr. Bayardo. Efficiently Mining Long Patterns from Databases. In Proc. of the ACM-SIGMOD Int'l Conf. on Management of Data, 85-93 (1998).

[8] Nicolas Pasquier, Yves Bastide, Rafik Taouil and Lotfi Lakhal. Efficient mining of association rules using closed itemset lattices. Information Systems, Volume: 24, Issue: 1, 25-46 (March 1999)

[9] S.J. Yen and A.L.P. Chen. An Efficient Approach to Discovery Knowledge from Large Database. Proceeding of the IEEE/ACM International Conference on

Parallel and Distributed Information Systems, 8-18 (1996).

[10] R. Agarwal, C. Aggarwal, and V. V. V. Prasad. A tree projection algorithm for generation of frequent itemsets. In J. Parallel and Distributed Computing, (2000).

[11] Jiawei Han , Jian Pei and Yiwen Yin. Mining Frequent Patterns without Candidate Generation. Proc. 2000 ACM-SIGMOD Int. Conf. Management of Data (SIGMOD'00), 1-12 (May 2000).

[12] J. Pei and J. Han. Can We Push More Constraints into Frequent Pattern Mining? Proc. 2000 Int. Conf. on Knowledge Discovery and Data Mining (KDD'00), Boston, MA, (August 2000).

[13] J. Han and J. Pei. Mining Frequent Patterns by Pattern-Growth: Methodology and Implications. ACM SIGKDD Explorations (Special Issue on Scaleble Data Mining Algorithms), 2(2) (December 2000).

[14] J. Pei, J. Han, H. Pinto, Q. Chen, U. Dayal, and M.-C. Hsu. PrefixSpan: Mining Sequential Patterns Efficiently by Prefix-Projected Pattern Growth. Proc. 2001 Int. Conf. on Data Engineering (ICDE'01), Heidelberg, Germany, (April 2001).