動態交易資料庫中線上關聯法則探勘法之研究

A Study of Mining Online Association Rules in Dynamic Transaction Databases

李建億
國立台南師範學院資訊教育研究所
leeci@ipx.ntntc.edu.tw

陳可欣
國立台南師範學院資訊教育研究所
@benz.ntntc.edu.tw

吳孟淞
國立台南成功大學資訊工程研究所
p7890109@ccmail.ncku.edu.tw

摘　要

由於資料庫應用範圍與所處理資料量的遽增，如何從這些資料中找出有用的隱藏資訊，已成為一個重要的議題。在關聯法則探勘的應用上，大部份都著重在靜態的資料庫，為了能夠在動態資料庫上挖掘出即時的關聯法則，稱之為線上關聯法則資料探勘然目前所提出的作法都有著需求額外大量的記憶體空間之缺點。因此，在本文中，將提出一種較有效率的方法，稱之為 OLDIC (OnLine Dynamic Counting) 演算法，其作法是利用分段掃瞄資料庫來產生所需資訊，儲存並且重複利用此已產生的資訊，以提高演算法的執行效率。本方法不僅可提供線上動態資料庫關聯法則和線上動態最小支持度的關聯法則探勘，且能不受限於記憶體的容量，即可以在線上完成所有的探勘執行過程。再者，因為項目集在探勘過程中具有週期的特性，在資料庫和最小支持度經常異動情況下，可以減少讀取資料庫的時間。從實驗的數據，可以發現處裡資料量大的資料庫時，OLDIC 演算法仍有很好的執行效率。

關鍵詞：線上動態資料庫、動態最小支持度、線上關聯法則

一、簡介

關聯法則探勘 (Association Rule) 主要是發掘出交易資料庫 (transaction databases) 中所包含的項目 (item) 之間所隱含的關聯性。所謂交易資料庫是由一些交易的紀錄所集合而成的，交易 (transaction) 則是由一些項目的紀錄所集合而成，而項目代表消費者所購買的物品。舉例而言，假設 $I = \{ i_1, i_2, i_3, \ldots, i_n \}$ 為所有項目 i_j (其中 $1 \leqq j \leqq n$) 組成的集合。給定一個交易資料庫 D，具有 $\| D \|$ 筆交易，交易資料庫中的每一筆交易 $T \in D$ 均為 I 的次級集合，而關聯法則就是以 $X \rightarrow Y$ 的形式來表示，其中的 X 與 Y 都是 I 的次級集合，而且 $X \cap Y = \varnothing$。如果在所有包含 X 的交易當中有 c %同時也包含了 Y 的話，則我們可以稱關聯法則 $X \rightarrow Y$ 以信賴度 (confidence) c 的水準成立於交易資料庫 D 中。此外，如果有 s % 的交易同時包含了 X 與 Y，則稱關聯法則 $X \rightarrow Y$ 以支持度 (support) s 的水準成立於交易資料庫 D 中。一個集合中若含有 k 個 items 則稱為 *k-itemset*。若支持度的門檻值設定為 s %，此門檻值稱為最小支持度 (minimum support)，而且在交易資料庫 D 之中至少有 s %的交易含有同樣的 *k-itemset*，則可將此 *k-itemset* 稱為 *large k-itemset*。

近年來，許多關聯法則的相關研究，大都是運用於靜態資料庫 (static database) [3, 4, 7, 12, 13]，主要是把所有符合使用者所設定之最低信賴度 (minimum confidence) 和最小支持度的關聯法則找出來。但是當更新現有的資料庫時，必須對整個更新後的資料庫重新探勘而降低效率。因此，

一些學者提出漸進式關聯法則探勘 [5, 9, 10, 11, 14, 15] 來改善此缺點。然而，不管是靜態資料庫的關聯法則探勘或漸進式關聯法則資料探勘，都無法提供線上交易系統的即時性 (real-time) 關聯法則探勘。線上與一般關聯法則探勘演算法主要的不同在於：線上的資料庫資料變動頻率會隨著應用層面的不同而改變，若應用於資料高度變動的線上資料探勘演算法，就必須將及時更新資料庫的因素考慮進來。例如，異動資料庫資料的時間對資料探勘的即時性關係以及資料更動所花費的探勘時間和記憶體需求成本的降低等因素。此外，線上關聯法則探勘還需要能動態調整最小支持度，來符合使用者的需求，亦即對資料庫隨時新增、刪除的反應機制，和對使用者所預設之最小支持度做動態調整的功能。

為了符合線上資料探勘演算法所需功能，並解決目前所提出的線上資料探勘演算法[1, 2, 6, 8]在記憶體上的限制以及在線上探勘前必須產生相關資訊的缺點，在本文中提出 OnLine Dynamic Itemset Counting (OLDIC) 的演算法，來進行線上關聯法則的探勘，以 Dynamic Itemset Counting (DIC) 演算法 [7] 為基礎，加以改良以增進其效能。

DIC 演算法是應用在批次關聯法則資料探勘上，它將資料庫分成多個區段，在處理一個區段後，產生在此區段中的 large itemsets，藉此產生 candidate itemsets，減少資料庫掃瞄次數。在線上資料探勘時，可以在原始資料庫執行的同時，對動態更新資料及動態調整最小支持度做出反應，不用等到原始資料庫整個探勘完成才對使用者的需求做出調整，增加了演算法的彈性和即時性。同時，DIC 演算法一次掃預設筆數的作法，符合實際資料庫更新方式。另外，OLDIC 演算法利用項目集生命週期的特性，可以在一個高度變動的資料庫或最小支持度常常轉換，而且需要即時探勘的應用上，重複利用已知資訊，減少掃瞄資料庫的時間，達到提高效率的目的。

二、文獻探討

在本章中，主要針對線上關連法則探勘相關演算法做一簡單的敘述說明。Amir [2] 和 C.C. Aggarwal [6] 等人所提出的線上關聯法則探勘法，主要是將資料庫的資料直接載入於記憶體，雖然兩者的作法能有效降低執行時間，卻受限於記憶體的容量，導致無法挖掘出完整的結果。除此之外，兩者的作法均需要事先建立所需的項目集絡 (lattice)，再將項目集絡放置於記憶體中供使用者進行線上挖掘，雖然可以降低線上的挖掘時間，卻不夠彈性，且當資料資料異動時，需重新建構一次項目集絡。而 C. Hidber 所提出的 Continuous Association Rule Mining Algorithm (Carma) [8] 演算法，雖然解決了記憶體和需預先在批次時建立項目集絡的問題，但只能在第一部份做資料庫異動和動態改變最小支持度。另外，Reduced Itemset Lattice (RIL) 演算法 [1] 雖可以動態調整資料庫和最小支持度且異動資料時不需要重新建構項目集絡，然而，仍需在 offline 時建立項目集絡，且所花費的成本很大。

為了能對資料庫隨時新增刪除，以及對使用者所預設之最小支持度做動態調整，並解決上述線上資料探勘演算法的缺點，在下一章中，將提出 OnLine Dynamic Itemset Counting (OLDIC) 演算法，來提昇線上關聯法則的探勘效率。

三、線上關聯法則資料探勘 (OLDIC 法)

在本章中，將針對 OLDIC 法可解決線上關聯法則探勘、線上動態資料庫的關聯法則探勘及線上動態最小支持度的關聯法則探勘等三議題，分別加以探討。

（一）線上關聯法則探勘

線上關聯法則是直接在線上產生 large itemsets，對完成掃瞄所有資料庫的項目集集合定義如下：完成整個資料庫掃瞄且為 large itemset 的

項目集集合，稱之 FL_k (Final Large *k*-itemset) 及完成整個資料庫掃瞄卻為 small itemset 的項目集集合，稱之 FC_k (Final Candidate *k*-itemset)。另外，對尚未完成整個資料庫掃瞄的項目集集合定義如下：未超過最小支持度的項目集集合，稱之 C_k (Candidate *k*-itemset)，以及超過最小支持度的 large itemset，稱之 L_k。則對任何一個已經產生的 *k*-itemset，不管資料庫更動和最小支持度的轉換，都是在 FC_k、C_k、L_k 和 FL_k 這些集合間轉換，此種情形稱之為「項目集的生命週期」。也就是說，當任何一個項目集被產生後就會一直存在，從 C_k 型態成為 L_k 型態，到最後成為 FL_k 或 FC_k 型態的過程中，其項目集都不會被刪除，只在這些型態之間轉換。因此，利用此特性可以減少掃瞄資料庫的時間，達到提高效率的目的。有關 OLDIC 演算法常用的符號，詳列於表 1 所示。

在執行線上資料探勘前，使用者需要事先設定 *minsup*，並且將 *DB* 每 *M* 筆資料分成一個區塊，若未滿 *M* 筆，仍然視為一個區塊，至於 *M* 的設定值，則根據 DIC 演算法 *M* 為經驗值。例如 $||DB||$ 為 12，若以 5 筆交易為一個區段，則可分為 3 個區段，其編號為 (*p1 - p2*)、(*p2 – p3*) 及 (*p3 - p4*)。

首先，將所有的 1-itemset 當作 C_1，掃瞄 *M* 筆資料後，判斷 C_1 的出現次數是否超過 *minsup*，若成立，則加入 L_1 中並且和集合中的其他 L_1 結合成 C_2 (和 Apriori 演算法[4]相同)。接著，繼續掃瞄下一個 *M* 筆的資料，同理，若 C_1 超過 *minsup*，則加入 L_1 中並且和已經成為 L_1 的 itemsets 結合成 C_2。但 $k \geqq 2$ 時，則必須有 $k+1$ 個子集合同時為 L_k 或 FL_k 才能結合成 L_{k+1}，若有任何一個子集合不屬於 L_k 或 FL_k，則必須刪除，以去除多餘的 candidate itemsets 增加演算法的執行效率。假設 L_2 = { *AD, DE* }，C_2 = (*AB*) 超過 *minsup* 則加入 L_2，但不能與 (*AD*) 結合成 (*ABD*)，因為 *k* = 2 且 (*ABD*) 的子集合有 (*AB*)、(*AD*) 與 (*BD*)，其中 (*BD*) 不屬於 L_2，所以 (*ABD*) 不能成為 C_3，必須刪除。

在掃瞄資料庫的過程中，每個 itemset 都需要記錄從第幾個 pass 開始掃資料庫以及那個切點編號開始計算出現次數，記為 *start_p = (pass, p)*，已掃瞄過的資料庫區塊編號，記為 *scan_block*。例如 (*A*) 由第一個 pass 切點編號 *p1* 開始計算出現次數，同時已經掃過編號 (*p1 - p2*) 和 (*p2 - p3*) 的資料庫區塊，則 *start_p* 為 (1, 1)，*scan_block* 記錄為 (1, 3)。任何一個 itemset 掃瞄過一次的 *DB*，則不需要繼續計算出現次數，亦即，若此時 C_k 沒有過門檻值，則存於 FC_k 中，相反的，若原本在 L_k 中的 itemset，並且已經掃瞄資料庫一次則加入 FL_k，若 (*AB*) 的 *start_p* 為 (1, 2)，則 (*AB*) 需要在 *pass* = 2 且 *p* = 2 時結束計算。同理，繼續執行，直至所有的 C_k 及 L_k 裡的 itemsets 都掃完一次資料庫為止。舉例說明，假設有一資料庫如表 2 所示，其項目為 *ABCDE*。*Minsup* 為 0.2，且 *M* = 3，所以可切為 3 個區段，也就是 *(p1 - p2)*，*(p2 - p3)* 和 *(p3 - p4)*，同時 *p4* 可視為 *p1*，如圖 1 所示。首先 *minsup* = 0.2，則 *minsup_tran* = ⌜ 0.2 * 8 ⌝ = 2，而 *pass* 的初始值為 1 且 C_1 = { *A, B, C, D, E* }。C_1 在 *p1* 開始掃瞄，如圖 1 可知，先掃瞄資料庫前三筆資料並計算出現次數，在 *p2* 時，可得 C_1 的出現次數，由表 3 知道，count(*B*) = 3，count(*C*) = 2 及 count(*D*) = 3，{ *B, C, D* } 的出現次數超過 *minsup_tran* 成為 L_1，並可結合成 C_2，C_2 = { *BC, BD, CD* }，這些 C_2 的 *start_p* = (1,2)，*count* = 0。同理，繼續掃瞄資料庫，在 *p4* 時可視為下一個 pass 的 *p1*，這個時候，所有的 1-itemset 都必須停止計算出現次數，如表 3 所示，所有的 C_1 都成為 FL_k，而新加入的 C_2 = { *AE, BE, CE, DE* }。此時，C_k 除了有上述新加入的 C_2 外尚有還未完成計算的 itemsets，如表 3 所示，所以必須繼續執行演算法。(*AD*) 在 *pass* = 2 且 *p* = 3 時的出現次數超過 *minsup_tran*，雖然可以和 FL_k 裡的 (*BD*) 結合成 (*ABD*)，但是 (*ABD*) 其中的一組子集合 (*AB*) 並未超過 *minsup_tran*，所以必須刪

除。接著，(*AB*) 和 (*AC*) 已經掃瞄資料庫一次並且其出現次數未達到 *minsup_tran*，所以 (*AB*) 和 (*AC*) 必須放入 FC_k，同時 C_k 尚有未完成計算的 itemset，同理，演算法必須繼續執行。當掃瞄資料庫第三次 *p2* 時，由於 C_k 裡的 (*BCD*) 的出現次數大於 *minsup_tran*，而且完成資料庫掃瞄成為 FL_k。同時 C_k 及 L_k 中已經沒有尚未完成計算出現次數的 itemset，所以演算法結束，FL_k 為針對使用者所定義的 *minsup* 而產生出來的 large itemsets。

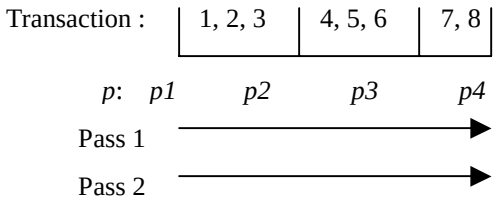

圖 1、資料庫區塊圖 I

表 1、符號表 I

參數名稱	意義
N	資料庫中總共有 *n* 個單項項目(item)
k-itemset	由 *k* 個單項項目所組合的複合項項目，如：(*AB*) 即 2-itemset，而 (*DEF*) 則為 3-itemset
C_k, $k = 1..n$	large *k*-itemset 的候選集合，而且目前這些 *k*-itemset 都還在做資料庫出現次數的計算
count(k-itemset)	為 *k*-itemset 在資料庫所出現的次數
total_trans	資料庫的總交易數
Minsup	最小支持度，亦即 *Count*(C_k) ≧ *minsup*，則稱之 L_k
Minsup_tran	符合最小支持度的交易數
L_k, $k = 1..n$	Large *k*-itemset 的集合，記為 L_k
FC_k, $k = 1..n$	small *k*-itemset 的集合，記為 FC_k
FL_k, $k = 1..n$	已經掃完整個資料庫的 large *k*-itemset 的集合
DB	原始的資料庫
‖ *DB* ‖	原始資料庫的總交易數
M	將資料庫每 *M* 筆劃分成一個區段，若筆數不到 *M* 筆仍看成一個區段
p	區塊切點編號
pass	掃瞄資料庫的次數

表 2、資料庫範例

TID	Items
1	*ABCD*
2	*BCD*
3	*BD*
4	*C*
5	*DE*
6	*AD*
7	*BCE*
8	*BCD*

（二）線上動態資料庫關聯法則探勘

上一節所提到的線上關聯法則探勘，只針對原始資料庫產生 L_k，並不能直接在線上更新資料庫。因此，本節中提出一些方法加以改良，使 OLDIC 法可以進行線上動態資料庫關聯法則探勘，其常用的相關參數如表 4 所示。

動態資料庫包括新增資料和刪除資料，而線上動態資料庫資料探勘則是在線上探勘的同時進行動態增刪資料的動作，於是分為 insert db^+ 和 delete db^- 兩部份來討論。

PART 1： insert db^+ 可以分成二種情況處理：

情況一：在尚未掃瞄完第一個 pass 的 *DB* 前新增 db^+。

當演算法執行到 *p* 點時，需要檢查是否有 db^+ 要新增。若有 db^+ 要新增時，*minsup_tran* 會跟著改變，亦即 *minsup_tran = minsup * total_trans = minsup * (||DB|| + || db^+ ||)*。此時，原本已經過門檻值的 itemset，由於 *minsup_tran* 的改變而成為 C_k，所以必須重新檢查 L_k 的出現次數。若 *count (L_k) ≧ minsup_tran*，則維持原狀；相反的，*count (L_k) < minsup_tran*，則必須由 L_k 中刪除丟回 C_k，根據 downward closure 的特性可知，其超集合也會隨著這樣的機制刪除。再者，要將 db^+ 分掃瞄區塊，區塊的起始切點編號為上一個資料庫的切點結束編號，而新加入的 db^+ 要分成幾個區塊，則視 「 || db^+ || / *M* 」。若「 || db^+ || / *M* 」= 1，而前一個資料庫結束編號的為 *p3*，則新加入的區塊編號為 (*p3 - p4*)。當原始資料庫掃瞄過一次後，亦即掃瞄到切點編號為 *p3* 時，則需要從 db^+ 接下去掃瞄，db^+ 掃瞄完後才重新由原始資料庫繼續執行，其他步驟，和線上關聯法則探勘的方法相同。

情況二：在已掃瞄完第一個 pass 的 *DB* 後新增 db^+。

當演算法掃瞄第二次以上的資料庫時，在 *p* 點時發現有 db^+ 要新增。同理，有 db^+ 要新增時，*minsup_tran* 會改變，也就是 *minsup_tran = minsup * total_trans = minsup * (|| DB || + || db^+ ||)*。此時，已有 *k*-itemset 已經掃完 *DB* 一輪，但其出現次數未超過原本的門檻值而存於 FC_k，然而 db^+ 的新增，這些 FC_k 有可能翻身成為 L_k，所以丟入 C_k 掃瞄 db^+ 即可，但不是每一個 FC_k 都必須掃瞄 db^+，只有在 *count(FC_k) +* || db^+ || ≧ *minsup_tran* 時，才需要重新加入 C_k 掃瞄 db^+。此外，不同於情況一的是 FL_k 裡的 itemsets，FL_k 是已經掃瞄 *DB* 一輪完成計算出現次數，這些 FL_k 只需要丟入 C_k 或 L_k 掃瞄 db^+ 即可，至於是要加入 C_k 或 L_k，則是依據出現次數有沒有過門檻值而定。假設，*count(FL_k) ≧ minsup_tran*，則重新加入 L_k 並從 FL_k 中刪除；若 *count(FL_k) < minsup_tran*，則必須由 FL_k 中刪除加入 C_k，由 downward closure 的特性可知，其超集合也可以根據這樣的機制在 FL_k 中不會出現。至於目前正在做次數計算的 L_k，就必須重新檢查 L_k 的出現次數，若 *count(L_k) ≧ minsup_tran*，則維持原狀。相反的，*count(L_k) < minsup_tran*，則必須由 L_k 中刪除加入 C_k。當然在這樣的過程中，可能會發現某些 itemset，在資料庫中的某幾個區塊已經掃瞄過，當然也就不需要對這些區塊重複掃瞄。且原始資料庫掃瞄過一次後，需要從 db^+ 接下去掃瞄，db^+ 掃瞄完後才重新由原始資料庫繼續執行，其他的步驟，同線上關聯法則探勘。

PART 2： delete db^-

刪除一個資料庫 db^-，可以視為新增一個 db^-，只是這個 db^- 出現的 itemsets 所要做的是出現次數的減少，而不是增加。亦即，在作法上可以如同 insert db^+ 一樣，只要在計算 *count(C_k)* 或 *count(L_k)* 的時候，將增加出現次數改成減少出現次數，若該 itemset 為 L_k 且同時出現次數低於 *minsup_tran*，則該 itemset 從 L_k 中刪除加入 C_k 即可。這個部分同樣可以分成二種情況處理：

情況一：在尚未掃瞄完第一個 pass 的 *DB* 前刪除 db^-。

當演算法執行到 *p* 點時，需要檢查是否有 db^- 要刪除。若有 db^- 要刪除時，*minsup_tran* 會跟著改變，亦即 *minsup_tran = minsup * total_trans = minsup * (||DB|| - || db^- ||)*。此時，除了對 db^- 計算次數時，將增加出現次數改為減少出現次數，若該 itemset 為 L_k 且同時出現次數低於 *minsup_tran*，則該 itemset 從 L_k 中刪除，加入 C_k 即可。其他的步驟，同線上關聯法則探勘。

情況二：在已掃瞄完第一個 pass 的 *DB* 後作刪除 db^-。

當演算法掃瞄第二次以上的資料庫時，在 *p*

點時發現有有 db^- 要刪除。同理，有 db^- 要刪除時，*minsup_tran* 會跟著做下降，也就是 *minsup_tran* = *minsup* * *total_trans* ＝ *minsup* * (|| *DB* || - || db^- ||)。此時，已有 *k*-itemset 已經掃完 *DB* 一次，但其出現次數未過原本的門檻值而存於 FC_k，然而 db^- 的刪除，這些 FC_k 有可能翻身成為 L_k，所以丟入 C_k 掃瞄 db^- 即可，但不是每一個 FC_k 都必須掃瞄 db^-，只有在 $count(FC_k) \geqq minsup\_tran$ 時，才需要重新加入 C_k 掃瞄 db^-。此外，在 FL_k 裡的 itemsets 已經掃瞄 *DB* 一輪完成計算出現次數，這些 FL_k 只需要丟入 C_k 或 L_k 掃瞄 db^- 即可，至於是要加入 C_k 或 L_k，則是依據出現次數有沒有過門檻值而定。假設，$count(FL_k) \geqq minsup\_tran$，則重新加入 L_k 並從 FL_k 中刪除；若 $count(FL_k) < minsup\_tran$，則必須由 FL_k 中刪除加入 C_k，其超集合也可以根據這樣的機制在 FL_k 中不會出現。至於目前正在做次數計算的 L_k，就必須重新檢查 L_k 的出現次數，若 $count(L_k) \geqq minsup\_tran$，則維持原狀不做任何處理。相反的，$count(L_k) < minsup\_tran$，則必須由 L_k 中刪除加入 C_k。當然在這樣的過程中，可能會發現某些 itemset，其實在資料庫中的某幾個區塊已經掃瞄過，當然也就不需要對這些區塊重複掃瞄。且原始資料庫掃瞄過一次後，需要從 db^- 接下去掃瞄，db^- 掃瞄完後才重新由原始資料庫繼續執行，此時，除了對 db^- 計算次數時，改為將增加出現次數變成減少出現次數，若該 itemset 為 L_k 且同時出現次數低於 *minsup_tran*，則該 itemset 從 L_k 中刪除，加入 C_k 即可。其他的步驟，和線上關聯法則探勘相同。以表 2 的資料庫範例，並且加入表 5 的新增資料庫範例。假設 *minsup* = 0.2 而 *minsup_tran* = ⌜ 0.2 * (8 + 3) ⌝ = 3，而新增資料庫的切點編號為 *(p4 – p5)*，同時 *p5* 可視為 *p1*，如圖 2 所示。若新增資料庫的加入時間，是原始資料庫執行過程中的 *pass* = 1 且 *p* = 2，根據表 3 所示，需要重新檢查集合內的 itemset，只有 L_k = { *B, C, D* }，其中 count(*C*) 小於因為新增資料庫而改變的 *minsup_tran*，所以在此時 (*C*) 必須從 L_1 中刪除重新加入 C_1，接下來的步驟，和線上關聯法則探勘一樣，整個演算法的執行情形和結果如表 6 所示。另外，若加入的時間改為 *pass* = 2 且 *p* = 2，如圖 3 所示，所需重新檢查的集合包括 L_k 、 FL_k 及 FC_k。據表 3 可知，FL_k 中的 { *A, B, C, D, E, BC, BD, CD* }，因為資料的新增必須加入 C_k 重新計算 db^+，又 { *B, C, D, BC, BD, CD* } 仍然有超過改變後的 *minsup_tran*，所以必須從 FL_k 中刪除加入 L_k，亦即因新增資料後的 C_k = { *A, E, AB, AC, AD, AE, BE, CE, DE, BCD* }，L_k = { *B, C, D, BC, BD, CD* }，而 FL_k 及 FC_k 都是空集合。又{ *A, B, C, D, E, BC, BD, CD* }已經掃瞄過資料庫一次，其 *scan_block* 記錄為 (1, 4)，所以這些 itemset 在區塊 (*p1 - p4*) 間並不再重複計算次數。之後的步驟和線上關聯法則探勘相同，整個演算法的執行情形和結果如表 7 所示。

（三）線上動態最小支持度關聯法則探勘

動態最小支持度關聯法則探勘包括了提高最小支持度和降低最小支持度，其相關的參數如表 8 所示。

PART 1： 提高最小支持度，這個部分可以分呈二種情況處理：

情況一：在第一個 pass 的 *DB* 尚未掃瞄完前提高最小支持度。

當演算法執行到 *p* 點時，程式會檢查使用者是否要提高最小支持度。若需要提高最小支持度，*minsup_tran* 會跟著上升，亦即 *minsup_tran* = *change_minsup* * *total_trans*。原本已經過門檻值的 itemset 有可能變成不過門檻值，所以必須重新檢查 L_k 的出現次數，若 $count(L_k) \geqq minsup\_tran$，則維持原狀不做任何處理。相反的 $count(L_k) < minsup\_tran$，則必須由 L_k 中刪除，重新加入 C_k，接下來的步驟和線上關聯法則探勘一樣。

表 3、OLDIC 法的執行狀態（ 範例一 ）

pass	1			2			3	
p	1	2	3	1	2	3	1	2
C_k	*A(0) B(0) C(0) D(0) E(0)*	*A(1) E(0)* ***BC(0)BD(0) CD(0)***	*E(1)* ***AB(0) AC(0) AD(0)*** *BC(0) BD(0) CD(0)*	*AB(0) AC(0) AD(0)* ***AE(0)*** *BD(1)* ***BE(0)*** *CD(1)* ***CE(0) DE(0)***	*AB(1) AC(1) AD(1) AE(0) BE(0) CE(0) DE(0)* ***BCD(0)***	*AE(0) BE(0) CE(0) DE(1) BCD(0)*	*BCD(1)*	Φ
L_k	Φ	*B(3) C(2) D(3)*	*A(2) B(3) C(3) D(5)*	*BC(2)*	Φ	Φ	Φ	Φ
FC_k	Φ	Φ	Φ	Φ	Φ	*AB(1) AC(1)*	*AB(1) AC(1) AE(0) BE(1) CE(1) DE(1)*	*AB(1) AC(1) AE(0) BE(1) CE(1) DE(1)*
FL_k	Φ	Φ	Φ	*A(2) B(5) C(5) D(6) E(2)*	*A(2) B(5) C(5) D(6) E(2) BC(4) BD(4) CD(3)*	*A(2) B(5) C(5) D(6) E(2) AD(2) BC(4) BD(4) CD(3)*	*A(2) AD(2) B(5) BC(4) C(5) BD(4) D(6) CD(3) E(2)*	*A(2) AD(2) B(5) BC(4) C(5) BD(4) D(6) CD(3) E(2) BCD(3)*

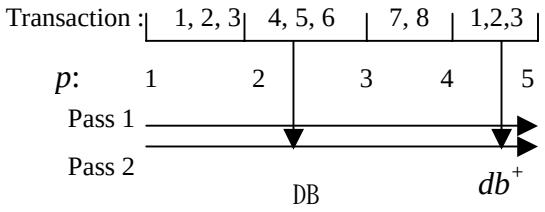

圖 2、資料庫區塊圖 II

表 4、符號表 II

參數名稱	意義
Db^+	新增的資料庫
Db^-	刪除的資料庫
$\|\| db^+ \|\|$	新加入資料庫的交易數
$\|\| db^- \|\|$	刪除的資料庫的交易數

表 5、新增資料庫範例

TID	ITEM
1	*BCD*
2	*D*
3	*CD*

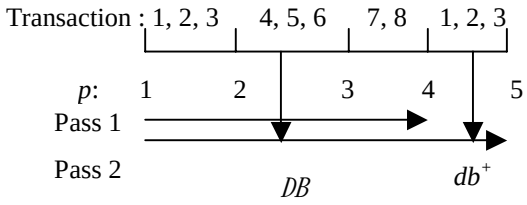

圖 3、資料庫區塊圖 III

表 8、符號表 III

參數名稱	意義
change_minsup	改變後的 *minsup*

表 6、OLDIC 法的執行狀態

pass	1				2				3	
p	1	2	3	4	1	2	3	4	1	2
C_k	*A(0) B(0) C(0) D(0) E(0)*	*A(1)C(2) E(0)**BD(0)***	*A(2) E(1) **BC(0)**BD(0) **CD(0)***	*A(2) E(2) BC(2)BD(1) CD(1)*	*BD(2)*	***BCD(0)***	*BCD(0)*	*BCD(1)*	*BCD(2)*	Φ
L_k	Φ	*B(3) D(3)*	*B(3) C(3) D(5)*	*B(5) C(5) D(6)*	*BC(3) CD(3)*	*BC(5) CD(5)*	Φ	Φ	Φ	Φ
FC_k	Φ	Φ	Φ	Φ	*A(2) E(2)*	*A(2) E(2)*	*A(2) E(2)*	*A(2) E(2)*	*A(2) E(2)*	*A(2) E(2)*
FL_k	Φ	Φ	Φ	Φ	*B(6) C(7) D(9)*	*B(6) C(7) D(9) BD(5)*	*B(6)C(7) D(9) BC(5) BD(5) CD(5)*	*B(6)C(7) D(9) BC(5) BD(5) CD(5)*	*B(6)C(7) D(9) BC(5) BD(5) CD(5)*	*B(6)C(7) D(9) BC(5) BD(5) CD(5) BCD(4)*

表 7、OLDIC 法的執行狀態

Pass	1			2				3	
p	1	2	3	1	2	3	4	1	2
C_k	*A(0) B(0) C(0) D(0) E(0)*	*A(1) E(0) **BC(0) BD(0) CD(0)***	*E(1) **AB(0) AC(0) AD(0)** BC(0) BD(0) CD(0)*	*AB(0)AC(0) AD(0)**AE(0)** BD(1)**BE(0)** CD(1)CE(0) DE(0)*	*A(2) E(2) AB(1)AC(1) AD(1)AE(0) BE(0)CE(0) DE(0) **BCD(0)***	*A(2) E(2) AB(1)AC(1) AD(2)AE(0) BE(0)CE(0) DE(1) BCD(0)*	*A(2) E(2) AB(1)AC(1) AD(2)AE(0) BE(1)CE(1) DE(1) BCD(1)*	*BCD(2)*	Φ
L_k	Φ	*B(3) C(2) D(3)*	*A(2) B(3) C(3) D(5)*	*BC(2)*	*B(5) C(5) D(6) BC(4) BD(4)CD(3)*	B(5) *C(5) D(6)BC(4) BD(4)*CD(3)	B(5) *C(5) D(6)BC(4) BD(4)*CD(3)	Φ	Φ
FC_k	Φ	Φ	Φ	Φ	Φ	Φ	Φ	*A(2) E(2) AB(1) AE(0) AC(1) AD(2) BE(1)* CE(1) *DE(1)*	*A(2) E(2) AB(1)AC(1) AD(2)AE(0) BE(1)*CE(1) *DE(1)*
FL_k	Φ	Φ	Φ	*A(2)* B(5) *C(5) D(6) E(2)*	Φ	Φ	Φ	B(6) *C(7) D(9) BC(5) BD(5) CD(5)*	B(6) *C(7) D(9) BC(5) BD(5) CD(5) BCD(4)*

情況二：在第一個 pass 的 *DB* 掃瞄完後提高最小支持度。

當演算法掃瞄第二次以上的資料庫時，在 *p* 點時程式會檢查是否要提高最小支持度。若需要提高最小支持度， *minsup_tran* 會跟著改變，即 *minsup_tran* = *change_minsup* * *total_trans*。此時 FL_k 和 L_k 雖然已經超過於原始的 *minsup*，因為最小支持度的改變，這些 FL_k 和 L_k 內的 itemsets 有可能小於 *minsup_tran*，所以重新檢查 FL_k 和 L_k。亦即，FL_k 和 L_k 內的 itemsets 若超過 *minsup_tran*，則維持原狀；反之，則必須從 FL_k 或 L_k 中刪除丟入 FC_k 或 C_k，接下來的步驟同線上關聯法則探勘。

PART 2： 降低最小支持度，這個部分可以分呈二種情況處理：

情況一：在第一個 pass 的 *DB* 尚未掃瞄完前降低最小支持度。

當演算法執行到 *p* 點時，程式檢查是否要降低最小支持度。最小支持度需要降低時，*minsup_tran* 會跟著下降，即 *minsup_tran* = *change_minsup* * *total_trans*。此時，所有的 L_k 都不會產生問題，所以不需要檢查 L_k，同線上關聯法則探勘。

情況二：在第一個 pass 的 *DB* 掃瞄完後降低最小支持度。

當演算法掃瞄第二次以上的資料庫時，在 *p* 點程式會檢查是否要降低最小支持度。最小支持度需要降低時，*minsup_tran* 會跟著下降，即 *minsup_tran* = *change_minsup* * *total_trans*。此時，所有的 L_k 及 FL_k 內的 itemsets 都不會產生問題，所以不需要檢查 L_k 及 FL_k。但是 FC_k 有可能翻身成為 FL_k，*count*(FC_k) < *minsup_tran*，則維持原狀不做任何處理；若 *count*(FC_k) ≧ *minsup_tran*，則加入 FL_k 和同一層的 L_k 或 FL_k 做結合 (同 Apriori join [3])，亦即，FC_1 和 L_1 或 FL_1 結合成 C_2。接下來的步驟，同線上關聯法則探勘。

四、效能評估

在簡介中所提到的四種線上關聯法則資料探勘演算法當中，Amir [2] 及 C.C. Aggarwal [6] 所提出的兩個演算法，是將資料庫裡的相關資訊載入記憶體來運作，這與本線上資料探勘法的主要訴求有相當大的差異。RIL 演算法 [1] 則是需要在 offline 時事先建立相關資訊載入記憶體來運作，這與本線上資料探勘法的直接線上探勘也有不同。Carma 演算法 [8] 雖然不必事先在 offline 建立項目集絡，卻無法完全支援動態更新資料庫及動態最小支持度。因此本研究中，只針對本線上資料探勘法進行實作模擬。

（一）產生模擬資料

本文中所使用的資料是透由 IBM 所提供的模擬資料庫產生器來產生[1]。其相關參數意義列於表 9。設定 $N = 1000$、$|T| = 20$ 以及 $|I| = 4$。原始資料庫交易數為 10,000 筆以及 100,000 筆如表 10 所示，另外，更動資料庫的交易筆數分別為原始資料庫的 10%，亦即 1,000 筆和 10,000 筆如表 11 所示。最小支持度的區間則設為 3 % ~ 0.3 %。*M* 分別為 1,000 和 10,000。由於 OLDIC 演算法可以允許不同的時間點新增或刪除資料庫，所以，模擬實驗的演算法執行時間，皆由亂數產生新增或刪除資料庫的時間點，再以五次的平均作為演算法所花費的時間。

[1] 下載網址 http : // www.almaden.ibm.com /cs/quest/syndata.html

表 9、資料庫參數涵意

參數名稱	意義
$\|T\|$	交易包含的平均項目個數
$\|I\|$	資料庫平均 large itemset 的項目個數
$\|D\|$	資料庫包含的交易筆數
$\|N\|$	單項項目個數

表 10、原始資料庫各項參數設定值

資料庫種類	$\|T\|$	$\|I\|$	$\|D\|$	資料庫大小
T20.I4.D10K	10	4	10K	893KB
T20.I4.D100K			100K	5.0MB

表 11、更動資料庫各項參數設定值

資料庫種類	$\|T\|$	$\|I\|$	$\|D\|$	資料庫大小
T20.I4.D1K	10	4	1K	90KB
T20.I4.D10K			10K	893KB

（二）OLDIC 演算法模擬結果與比較

由圖 4 可以發現，10,000 筆和 100,000 筆的資料庫分別新增 1,000 筆和 10,000 筆的資料，隨著最小支持度的門檻降低，兩者的執行時間也隨之增加，尤其以 *minsup* = 0.005 到 *minsup* = 0.003 的執行時間最為明顯，這是因為門檻降低後，candidate itemsets 與 large itemsets 的數量也隨之增加所致。此外，兩者的執行時間差異並不大，這可以顯示資料量的多寡對演算法的執行效率而言，並沒有太大的影響。圖 5 主要在說明新增資料庫的個數對演算法執行時間的影響，根據此圖可以發現，演算法隨著新增資料個數的增加，執行時間不會呈等比例成長，亦即演算法對新增資料庫個數的敏感度會隨著個數的增加而減少。這是由於項目集的生命週期影響所致，演算法產生出來的 itemsets 不會因為任何的改變而消失，只會在不同的型態間轉換，如此一來可以降低重新計算出現次數的時間。由圖 6 可以發現演算法在 10,000 筆和 100,000 筆的資料庫分別刪除各 1/10 資料量時，會隨著最小支持度的門檻降低，其執行時間也隨之增加，尤其以 *minsup* = 0.005 到 *minsup* = 0.003 的執行時間最為明顯，與新增資料庫的結果相似，同樣的，這是因為門檻降低後，candidate itemsets 與 large itemsets 的數量也隨之增加所致。除此之外，兩者的執行時間差異並不大，這也顯示資料量的多寡對演算法的執行效率而言，並沒有太大的影響。從圖 7 可以得知，演算法隨著刪除資料個數的增加，執行時間不會呈等比例成長，亦即演算法對資料庫個數的敏感度會隨著個數的增加而減少。

綜上所述，可以得知 OLDIC 演算法的執行效率不會因為刪除或新增資料庫的資料量和個數而有所明顯差異。

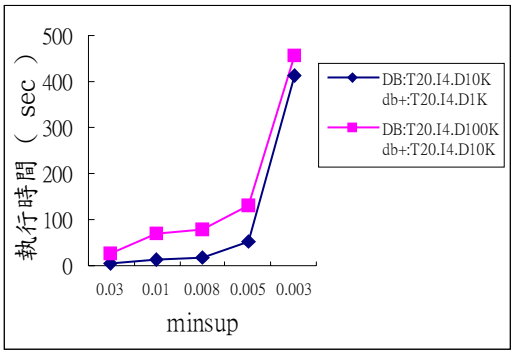

圖 4、T20.I4.D100K 與 T20.I4.D10K 資料庫之效能比較 （新稱資料庫個數為 10）

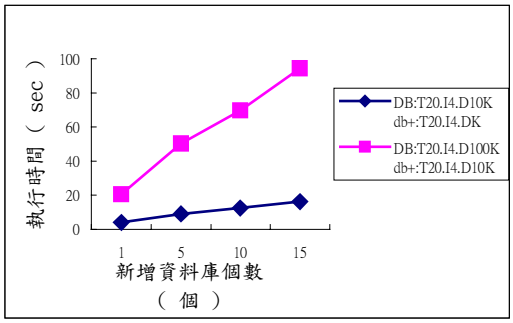

圖 5、新增資料庫之效能比較

（minsup = 1%）

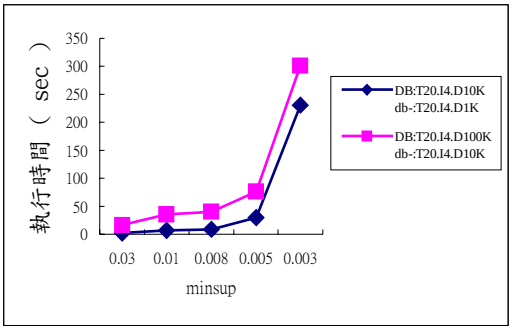

圖 6、T20.I4.D100K 與 T20.I4.D10K 資料庫之效能比較（ 刪除資料庫個數為 3）

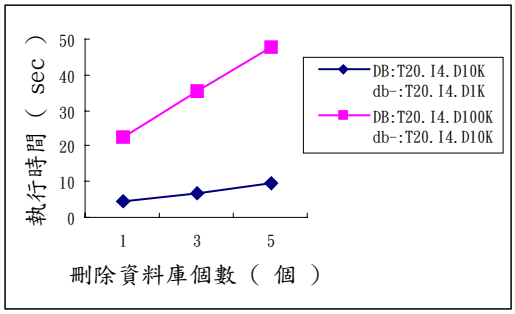

圖 7、刪除資料庫個數之效能比較（ minsup = 1% ）

五、結論與未來研究方向

由於電腦的普及和資料庫應用的廣泛，累積了大量的資料，除了資料管理的功能外，更希望可以從中獲得有用的資訊。所謂資料探勘，正是由資料庫中探勘有用的資訊，而關聯法則為資料探勘研究領域之一，主要是發掘出交易資料庫中所包含的項目之間所隱含之關聯性。大部分關聯法則探勘演算法，都應用於靜態資料庫，然而，在某些應用上則需要在動態資料庫上發掘出即時性的資訊。

目前線上關聯法則演算法的方法，是將資料庫裡的資料直接載入記憶體中，雖然提高了執行效率卻受限於記憶體容量造成探勘結果的不完整。除此之外，有些演算法需要在執行線上探勘前產生相關資訊，如此一來，使用者必須等待一段時間才能進行線上關聯法則探勘。根據上述的缺點，本文提出了 OLDIC 演算法，解決了記憶體容量造成探勘結果不完整的問題，同時所有演算法的執行過程都可以在線上完成，又因為項目集生命週期的特性，在資料庫或最小支持度的變動率很高的情況下，可以有效減少重複掃瞄資料庫的執行時間，提高演算法的效率。再者，OLDIC 演算法可以允許使用者在任何時候更新資料庫或修改最小支持度的設定，相當有彈性。從實驗的結果，可以發現 OLDIC 演算法對於多筆數的資料量依然有很高的執行效率。未來的研究方向，擬針對本演算法加以修改，運用於 multiple-level 的關聯法則探勘、加入權重（ weight ）因子的關聯法則探勘，或是對量值屬性（ quantitative attribute ）項目的關聯法則探勘等研究領域。

六、參考文獻

[1] 許智豪（民 87）:在動態資料庫上作線上挖掘關聯式法則，*國立中興大學資訊科學研究所碩士論文*。

[2] A. Amir, R. Feldman, R. Kashi, A New and Versatile Method for Association Generation, ”Principles of Data Mining and Knowledge Discovery, First European Symposium, *PKDD’97*, 1997, pp. 221-231.

[3] R. Agrawal, T. Imiclinski, A. Swami. Mining Association Rules between Sets of Items in Large Database. *In Proceedings of the ACM SIGMOD*, 1993, pp. 207-216.

[4] R. Agrawal, R. Srikant. Fast Algorithms for Mining Association Rules. *In Proceedings of the 20th VLDB Conference, Santiago, Chile*, 1994. pp. 487-499.

[5] N. F. Ayan, A. U. Tansel, E.Arkun, An Efficient Algorithm to Update Large Itemsets With Early Pruning, *KDD-99*, pp. 287-291.

[6] C.C. Aggarwal, P.S. Yu, Online Generation of Association Rules, *Proc. Of the IEEE*

ICDE'98, 1998, pp. 402-411.

[7] S. Brin, R. Motwani, J. Ullman, S. Tsur, Dynamic Itemset Counting and Implication Rules for Market Basket Data. *SIGMOD-97*, 1997, pp. 233-264.

[8] C. Hidber, Online Association Rule Mining, *Proc. Of the ACM SIGMOD*, 1999, pp. 145-156.

[9] David W. Cheung, J. Han, Vincent T. Ng, C.Y. Wong, Maintenance of Discovered Association Rules in Large Databases: An Incremental Updating Technique, *In Proceedings of the 12th ICDE, New Orleans, Louisiana*, February 1996, pp. 106-114.

[10] David W. Cheung, S.D. Lee, Benjamin Kao, A General Incremental Technique for Maintaining Discovered Association Rules, *Proceedings of the Fifth International Conference on Database Systems for Advanced Application*, Melbourne, Australia, April 1-4,1997, pp. 185-194.

[11] K.-K. Ng, W. Lam, Updating of Association Rules Dynamically, Database Applications in Non-Traditional Environments, 1999. *(DANTE '99) Proceedings 1999 International Symposium on*, 2000, pp. 84 -91.

[12] J. S. Park, M.-S. Chen, P. S. Yu, An Effective Hash-Based Algorithm for Mining Association Rules. *IEEE Transactions on Knowledge and Data Engineering*, 9(5), 1997, pp. 813-825.

[13] A. Savasere, E. Omiecinski, S. Navathe, An Efficient Algorithm for Mining Association Rules in Large Databases. *In Proceedings of the Very Large Data Base Conference*, September 1995, pp. 432-444.

[14] N.L. Sarda, N. V. Srinivas, An Adaptive Algorithm for Incremental Mining of Association Rules, *In Proc. DEXA Workshop*, 1998, pp. 240-245.

[15] S. Thomas, S. Bodagala, K. Alsabti, S. Ranka, An Efficient Algorithm for the Incremental Updation of Association Rules in Large Databases, *Proc. of the 3rd International Conference on Knowledge Discovery and Data Mining (KDD97)*, 1997, pp. 263-266.