XML 資料分群之查詢處理設計
Efficient Group-by Query Processing for XML data

黃治中 簡伯先 張雅惠
國立台灣海洋大學資訊工程研究所
{ M95570052, M98570022, yahui }@ntou.edu.tw

***摘要*—XML 資料的查詢處理，已經是重要的研究議題，而針對分群運算的處理，目前也受到大家的重視。先前研究所提出的作法，必需要輸入整份 XML 文件。而本論文提出的處理方式，則設計內容索引，快速地將元素分群，再利用主實體索引，將該群的資料取出。實驗結果顯示，當分群或輸出的運算式數量較少時，我們的系統會比先前的研究所提出的方法有著更佳的效率。**

***Abstract*- XML query processing has become an important research issue, and many researches now focus on the functionality of group-by expressions. Previous researches need to process the entire XML document. We propose a new method in this paper based on indices. The value index is designed to efficiently identify the group of each element, and the master-entity index is designed to efficiently retrieve the required data for a particular group. We have performed a series of experiments, and the results show that the proposed method is more efficient than the existing ones when the grouping or return expressions are few.**

***關鍵詞*—XML、分群運算、查詢處理、索引。**

***Keywords*—XML, grouping, query processing, index。**

一、緒論

近年來網際網路（WWW）發展迅速，已經成為資訊分享的重要工具，而自從 XML 被提出來做為 WWW 文件的標準後，許多企業紛紛藉由WWW作為資訊交換的管道。因此如何針對 XML 資料提高查詢效率，已經變成一個重要的研究議題。

針對 XML 文件的查詢方式，W3C 制定了 XPath 與 XQuery，其中 XPath 是路徑語言，用來定址 XML 文件中的某部份，而 XQuery 是建立在 XPath 基礎上面，具有複雜的查詢功能，但是在 2007 年制定的 1.0 版標準中，並沒有定義明確的 Group-By 語法，而是利用巢狀的 return 表示式去達到分群的效果。

Group-By 語法主要的目的，是依據資料的內容，將資料進行分群，以便算出群組的統計資料，如平均、總和…等。在 SQL 語言中具有重大的地位。很多研究者認為明確的分群語法，更有助於提升查詢的語意和效率，所以提議把該語法加入。本論文針對 XML 文件的分群運算，研究如何提昇查詢效率，我們的貢獻主要有下列幾點:

1. 我們設計內容索引（Value Index），將擁有同樣標籤的元素，利用其內容值建立索引，以快速得知同一個標籤下，哪些元素的內容值相同，屬於同一群。
2. 我們設計主實體索引（Master-Entity Index），利用主實體的觀念來表示元素的群組，以便將同一群組中對應到輸出的元素內容抓出。
3. 我們進行一系列的實驗，並且和一個著名的系統[3]比較。結果顯示，當分群或是輸出的運算式數量較少時，我們的系統會比論文[3]的效率更佳。

本論文其餘各節的架構如下：在第二節中我們簡介分群運算的基本定義。第三節介紹本系統中會使用的資料結構。第四節中說明整體系統架構，並將解說系統中每個模組及其使用之演算法。第五節將以實驗來分析此系統的效率。第六節將介紹與本論文主題有關之相關研究。最後在第七節提出結論以及未來展望。

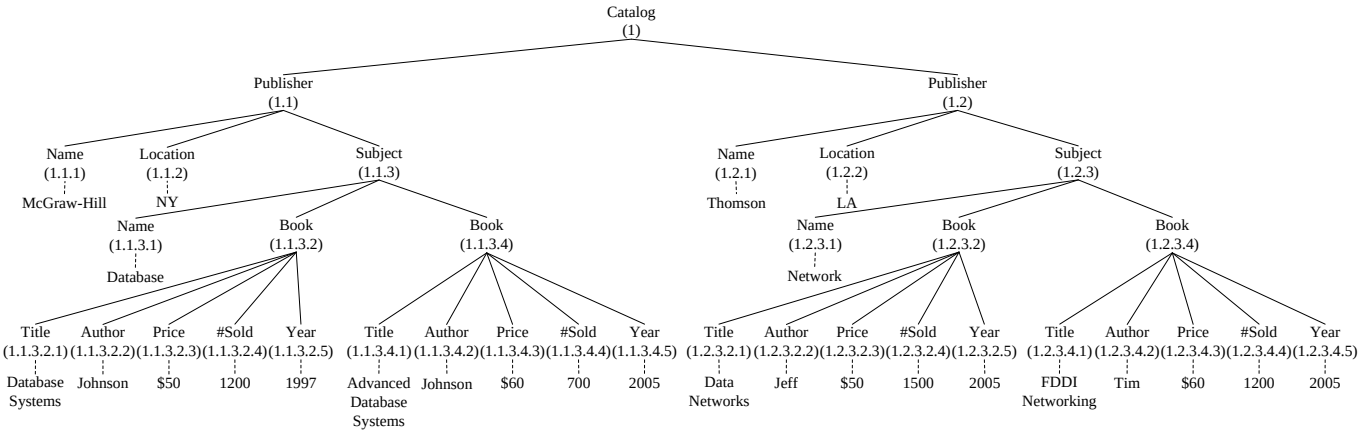

圖 1：XML 樹

二、問題定義

在本節中，我們會說明分群查詢語法及範例，以及對應的輸出結果。

2.1. XML 資料表示

首先，我們介紹 XML 文件的表示法。XML 文件以元素作為資料表示的基本單位，如 <Name> Database </Name > 表示了一個“Name” 元素，其內容為 “Database” ，而元素間必須具有嚴謹的巢狀包含關係，如 <Subject>< Name > Database </Name ></Subject> 表示了 Subject 元素中包含了一個 Name 子元素。所以，我們通常將一份 XML 文件表示成一棵樹。圖 1 的 XML 樹表示了兩個出版商（Publisher）的資料，並且將每個出版商出版的書（Book），以學科（Subject）分類表示。另外，元素內容則表示成元素的小孩，以虛線連接。其中，我們稱每條路徑最深之節點但不包含元素內容為「葉節點」（Leaf Node）如 Price 元素。至於每一個元素旁邊標示的編碼，將在第三節說明。

2.2. 分群查詢的語法和範例

本論文使用論文[1]提出的分群語法，以清楚地表達所有和分群有關的表示式。圖 2 列出了一個具有兩層巢狀的分群樣版，而圖 3 的 Q1 為一個對應的範例。以下利用 Q1 做為範例說明樣版裡的各個運算式：

L01	**for** α
L02	**group by** $\beta_1^{outer} \ldots \beta_k^{outer}$
L03	**having** λ^{outer} **order by** ω^{outer} **rank** ε^{outer}
	return （
L04	$\beta_1^{outer} \ldots \beta_k^{outer}$, $agg(\gamma_1^{outer}) \ldots agg(\gamma_k^{outer})$
L05	**then by** $\beta_1^{nest} \ldots \beta_k^{nest}$
L06	**having** $agg(\lambda^{nest})$ **order by** ω^{nest} **rank** ε^{nest} **return** （
L07	$\beta_1^{nest} \ldots \beta_k^{nest}$, $agg(\gamma_1^{nest}) \ldots agg(\gamma_k^{nest})$
L08	） ）

圖 2： 分群查詢句樣版

L01	**for** //Book
L02	**group by** //Price
L03	**having** avg（#Sold）> 700 **order by** //price **rank** < 10
	return （
L04	//Price, avg（#Sold）
L05	**then by** //Year **return** （
L06	//Year, count（*）
L07	） ）

圖 3： Q1 查詢句

1. 基準運算式（α）：列於 for 子句的 XPath 表示式，如 Q1 的//Book。此表示式是一個絕對表示式，用以直接指定 XML 文件裏對應的元素。
2. 分群運算式（β）：列於 group by 和 then by 子句內的內容，以其值做為分群的依據，如 Q1 裡的//Price 和//Year。

3. 限制運算式（λ）：列於 having 的內容，用以挑選符合特定限制的群組，如 Q1 的 avg（#Sold）> 700。
4. 排序運算式（ω）：列於 order by 子句的內容，用以做為排序群組的依據。為了確保每一個群組具有唯一的值，只有 β expression 才能列在ω運算式裏。
5. 分級運算式（ε）：列於 rank 子句的內容，只有排序過後符合 rank 限制的結果才會輸出。以 Q1 來說，只有前 9 名的群組才會輸出。
6. 回傳運算式（γ）：列於 return 子句內除了β運算式的內容，用以指定將內容輸出的元素，一般會套用 aggregation function，譬如 Q1 裏的 avg（#Sold）和 count（*）。

對於每一個查詢句，我們會如同論文[3]，將其結果表示成一個答案樹（Answer Tree）。對應 Q1 的答案樹如圖 4 所示。該樹以 XML 文件的根節點標籤，做為答案樹的根節點標籤。另外，答案樹的結構會對應到原始查詢句的巢狀結構，每一層的子樹根節點（Subtree Root），其標籤是將該層β運算式內所有的標籤串接起來，最後加上“-group”。該根節點其下的樹葉節點對應到β和γ運算式，其下虛線為對應的元素內容。以圖 4 為例，最外層有兩顆子樹，分別對應到 Price 為 50 和 Price 為 60 的群組。其餘的子節點則為巢狀的子樹根節點。至於同一層子樹的順序，則是根據該層的ω運算式的值，比較小的排在左邊。

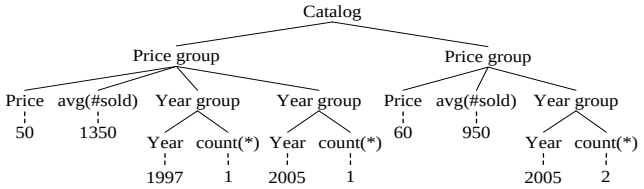

圖 4： Q1 的答案樹

```
L01  for//Subject
L02  group by anc::Publisher/Name, //Name return （
L03          anc::Publisher/Name, //Name, count（*）
             then by dec::Book/Year return （
L04                                  dec::Book/Year,
L05                          count（*）
L06          ） ）
```
圖 5: Q2 查詢句

注意到，雖然此例只表示兩層的巢狀關係，但是在本論文中，我們可以處理任意層數的巢狀結構。我們並稱最外層為第一層，依此類推。

此下我們再列出一個查詢句，顯示分群運算式和α運算式不同的結構關係，如圖 5 的 Q2 所示。如果一個運算式限定是α的祖先，則前面加上“anc::”，若是子孫則在前面加上“dec::”，或是“//”，父親則加註“par::”，小孩則直接以一條斜線“/”表示。以 Q2 為例，外層的第一個β運算式，限定是α（Subject 元素）的祖先 Publisher 下的子元素 Name，而內層β運算式，則限定是α的子孫 Book 元素下的子元素 Year。為了明確顯示依祖先分群的效果，我們另外設計一個 XML 樹，如圖 6 所示，而對應 Q2 的答案樹如圖 7 所示。

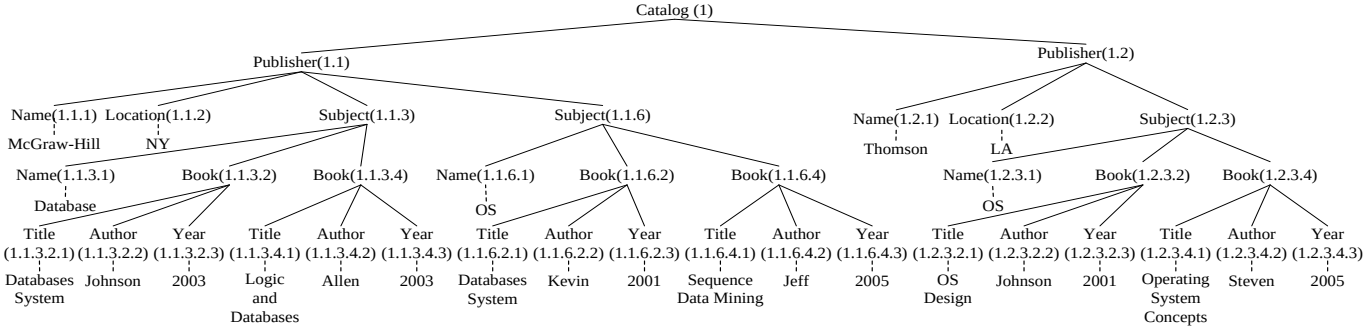

圖 6: XML 樹

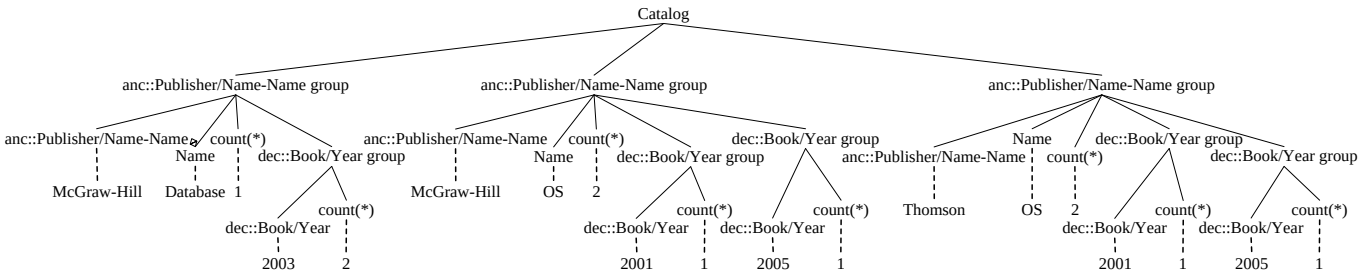

圖 7: Q2 的答案樹

三、資料表示法

在本節中，我們介紹如何表示 XML 資料，以加快查詢的速度。首先我們介紹元素的編碼方式和「Master-Entity」的定義，最後介紹本系統使用的資料結構。

3.1. 延伸杜威編碼

在本論文中，我們採用論文[5]提出的延伸杜威編碼（Extended Dewey Code）替 XML 元素編碼，如圖 1 所示。舉例來說，左邊的 Publisher 元素編碼為 1.1。此編碼的好處是可以直接將該編碼轉換成樹根到樹葉的標籤路徑（Labeled path），譬如 1.1.1 會轉換成 /Catalog/Publisher/Name。此編碼必須用到 DTD 的資訊，因為論文空間的關係，完整的編碼公式請參照原論文。

3.2. 主實體定義

在進行查詢句的分群處理時，我們必須先從 XML 資料中，確定對應到α運算式的元素，再從其相關結構關係處理其他運算式。根據觀察，對應到α運算式的元素，通常其在 DTD 的定義中，都會利用到“ * ”或“ +”符號，我們稱呼其為「*-node」，如圖 8 中的 Publisher、Subject、Book、Author 等。此節點的特性為，在 XML 文件中，此元素可以與兄弟節點具有相同的標籤。所以，我們沿用論文[4]的用法，提出以下定義：

```
<!ELEMENT catalog （publisher*）>
<!ELEMENT publisher （name, location, subject*）>
<!ELEMENT subject （name, book*）>
<!ELEMENT book （title, author+, price, sold, year）>
<!ELEMENT author （name, city）>
<!ELEMENT city （#PCDATA）>
<!ELEMENT name （#PCDATA）>
<!ELEMENT location （#PCDATA）>
<!ELEMENT title （#PCDATA）>
<!ELEMENT price （#PCDATA）>
<!ELEMENT sold （#PCDATA）>
<!ELEMENT year（#PCDATA）>
```

圖 8: 對應到圖 1 XML 文件的 DTD

【定義 1】實體（Entity）:若一個元素在 DTD 的定義中為一個 *-node，則稱其為實體。

由於進行查詢處理時，其它運算式，如β或γ運算式， 通常會依據最接近的祖先實體加以分群，所以我們提出第二個定義:

【定義 2】主實體（Master-Entity）: 針對一個 XML 中的節點 N，若節點 A 是 N 的祖先，為一個實體，且離此節點最近（可包含自己），則稱節點 A 為節點 N 的主實體。

以圖 1 的 XML 資料和元素 Price（1.1.3.2.3）為例，在符合實體定義的元素中，Book 元素（1.1.3.2）和該元素祖先關係最靠近，所以 Price 元素的主實體是 Book 元素。

注意到，若其他運算式和α的結構關係是 des 或 child，則其主實體可能會正好對應到α運算式。以 Q1 為例，return 子句中的 Price 和＃Sold

是 Book 的子元素，所以其對應的主實體會正好是α運算式取出的元素。但是，若結構關係為 par 或 anc 時，則主實體會是α的祖先。譬如，在 Q2 中，α運算式為//Subject，而對應到β運算式 anc::Publisher/Name 元素的主實體，會是 Publisher 元素，為α元素的祖先。

3.3. 元素編碼表和索引結構

在本篇論文中，我們會替 XML 元素以標籤名稱分類，建立各自的元素編碼表（EC-Table），但是根元素不予建立。針對每個元素，我們會給予以下三種資料:

1. 元素編碼（ID）：記錄該元素在 XML 樹中的延伸杜威編碼。
2. 主實體編碼（Master-Entity ID）：記錄該元素的主實體的延伸杜威編碼。
3. 內容值（Value）：若元素的節點型態是葉節點（Leaf Node），則記錄其元素內容。

對應圖 1 而標籤為 Name 的元素編碼表，如圖 9 所示

由於在做查詢處理時，我們會以元素值做為分群的依據，所以我們進一步建立「內容索引」（Value-Index），將屬於相同標籤的元素值收集起來，建立成 B^+-tree。注意到此索引以 Value 為 Key，但是回傳對應的主實體，而非擁有該值的葉元素，以便後續進行分群的處理。圖 10 是根據圖 1 替 Name 元素建立的內容索引。

另一方面，分群之後，我們必須輸出該群中符合輸出要求的元素內容值。所以我們另外建立「主實體索引」(Master-Entity Index)，針對每個標籤表示出主索引和元素值的對應。注意到，不同於內容索引，每個 Key 值（主索引編碼）可能會對應到很多個元素值。圖 11 是對應到 Name 的主實體索引。

ID	Master-Entity ID	Value
1.1.1	1.1	McGraw-Hill
1.1.3.1	1.1.3	Database
1.2.1	1.2	Thomson
1.2.3.1	1.2.3	Network

圖 9： Name 的元素編碼表

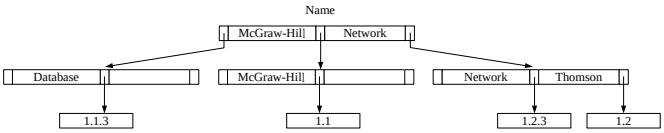

圖 10： Name 的內容索引

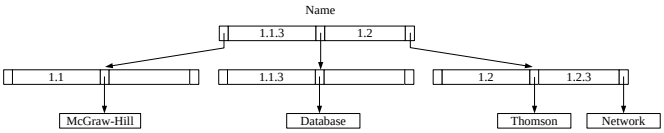

圖 11： Name 的主實體索引

四、查詢句處理

在本節中，我們首先介紹整體系統架構，接著會對其中幾個重要的模組做更詳細的說明。

4.1. 系統架構

本查詢系統的整體架構如圖 12 所示。首先，使用者輸入一個查詢句，經過 Parse Query 模組，建立成 Query List 將查詢句分層表示。接著，在 BuildABList 模組中，會依序對 Query List 中的每層子查詢句，去處理排序運算式（ω）和分群運算式（β）。此模組利用內容索引先取出符合個別運算式的元素，接著兩兩比對關係之後，組合成一個完整的 ABList（All-By List）並且依序排好，同時會將下一層子查詢句產生的分群結果，串接起來。下一步，CreateAnswerTree 模組會依據 ABList 的結構建立答案樹，並透過主實體索引抓取對應到回傳運算式的資料。當回傳運算式是 Aggregation 節點，此模組會同時做對應的計算。

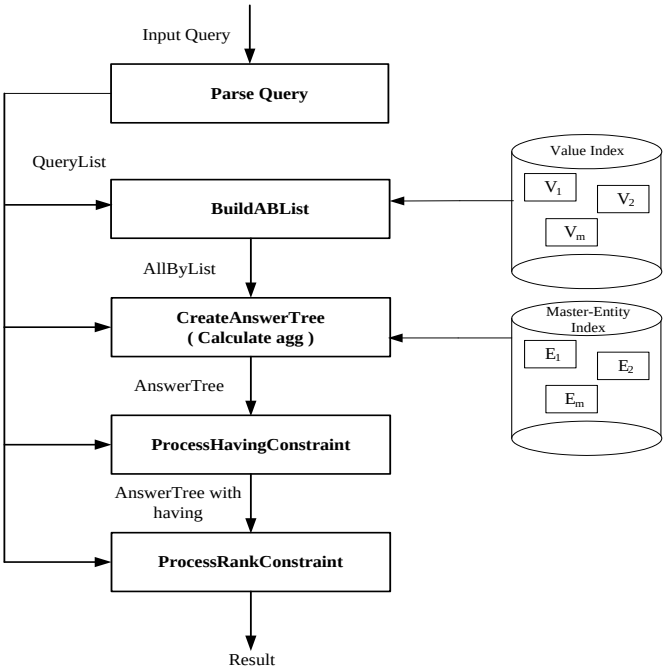

圖 12： 系統架構

得到初步的答案樹後，ProcessHaving-Constraint 模組，會依據 Query List 裡面的每一個限制運算式，刪除答案樹裡不符合限制的子樹，最後透過 ProcessRankConstraint 模組，刪除不符合 Rank 限制的子樹，得到最後的答案樹。

4.2. ABList 結構

本論文提出 ABList 的結構用以紀錄處理完分群運算和排序的結果。ABList 是由一個個 ABNode 所串接起來，每個節點內部又儲存了 4 項資料，如圖 13 所示，以下一一說明：

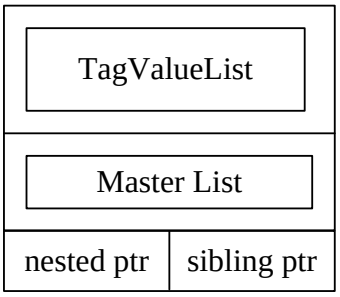

圖 13: ABNode

1. TagValueList：對應查詢句裡的分群運算式，紀錄某一群元素的標籤和內容值。以 Q1 和圖 1 為例，第一層分群運算式會產生兩個節點，其 TagValueList 分別是（Price, 50）和（Price, 60）。注意到，若有多個分群運算式，必須將其值進行配對組合（Cartesian Product）。
2. Master List：記錄符合該 Tag-Value 限制的元素。為了進行以群組為基礎的運算，我們會記錄該元素的主實體編碼。
3. sibling ptr：其目的是將同層的分群運算式所產生的所有 ABNode 連接起來，並進行排序運算的處理，內容較小的在左邊。若有多個排序運算式，會以第一個為主，依此類推。
4. nested ptr：連接到下一層 query 所產生的 ABList 的第一個節點。

以 Q1 和圖 1 的 XML 文件為例，所產生的 ABList 如圖 14 所示。

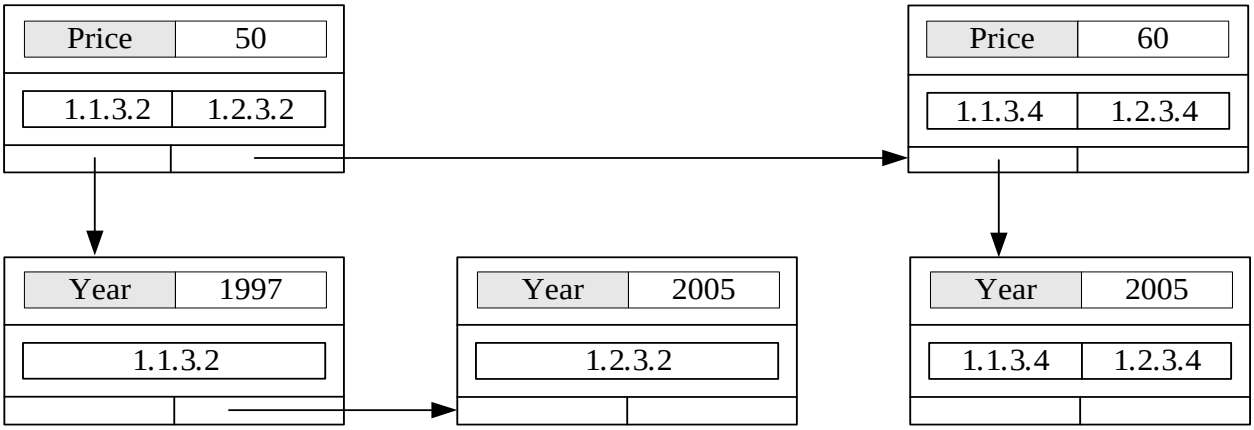

圖 14: 對應 Q1 中分群運算式 Price 和 Year 的 ABList

4.3. 建立 ABList

本小節說明建立 ABList 之演算法，如圖 15 所示，此演算法的第一個參數 QList，由一個個 QNode 所組成，每一個 QNode 表示了每一層查詢句的所有運算式。第二個參數 MList 則是前次運算完的 ABNode 的 Master List。我們並設計此演算法為一個遞迴函式，其透過第三個參數 Level 取得下一層子查詢句產生的 ABList，以建立巢狀結構。

以下我們做進一步的說明，首先我們取得此層查詢句的資料，並透過 Level=0 取得基準運算元（α）。第 5 行的 For 迴圈是依序處理每個分群和排序運算式，針對每個運算式，我們將其和 α 運算式組合起來，至 DTD 中取出所有符合的路徑（L6），若有多個路徑，我們選擇最上層的路徑，以其主實體的 Level 做為後續的判斷（L7）。同時，我們利用其標籤，到對應的內容索引取出符合的編碼。注意到，如果 MList 不為空，則主實體編碼也必須出現在 MList 中（或作祖孫的判斷），取出的編碼資料存在 NewABList 裏（L8）。

接下來，我們必須合併先前產生的 ABList 和剛產生的 NewABList。此兩者的結構關係，是透過 PreMELevel 和 NewMELevel 來判斷，當兩個 List 的主實體在 XML Tree 中位於同層，會呼叫演算法 FindTheSameLevelME 進行比對。如果 PreMELevel 比 NewMELevel 低，也就是 ABList 中的主實體會是 NewABList 中的主實體的祖先，我們呼叫演算法 FindDescendantME 進行比對，反之則呼叫演算法 FindAncestorME。最後，我們更新 PreMELevel 的值，如果還有下一層子查詢句的話，我們會依序針對 ABList 中的每一個節點進行遞迴處理，以建立巢狀結構。

以下我們說明演算法 FindTheSameLevel-ME，如圖 16 所示。此演算法會輸入兩個 ABList，分別稱作 List1 和 Lsit2。此二個 List 的

```
Function BuildABList
Input: QList
       MList          /*上一層 ABNode 的 MasterList*/
       Level          /*目前處理到的 QList 的位置*/
Output: ABList //處理完 QList,每一棵子樹 ABList 結構
變數說明:
       PreMELevel     /*先前 β 在 DTD 的 Level*/
       NewMELevel     /*目前 β 在 DTD 的 Level*/
       PathArray      /*所有符合跟 α 關係的路徑*/
1.  QNode = QList[Level]
2.  GroupRoot = QList[0].Group.α.Tag
3.  ABList = NULL
4.  PreMELevel = 0
5.  For（each Tag in QNode.OrderByArray or
                                  in QNode.ByArray）{
6.     PathArray = DTD.GetIndexName
                                  （GroupRoot, Tag）
7.     （IndexName, NewMELevel ）=
                                  PathArray.GetTop（）
8.     NewABList = GetABList（IndexName, MList）
9.     If （PreMELevel – NewMELevel == 0）{
10.       ABList = FindTheSameLevelME（ABList,
                                  NewABList）
11.    }Else If（PreMELevel – NewMELevel < 0）{
12.       ABList = FindDescendantME（ABList,
                                  NewABList）
13.    }Else{
14.       ABList = FindAncestorME（ABList,
                                  NewABList）
15.    }
16. }
17. PreMELevel = NewMELevel
18. While（Level < QList.size）{
19.    If（QList.size == 1）{
20.       return ABList
21.    }Else{
22.       For（each ABNode in ABList）{
23.          MList = ABNode.MasterList
24.          ABNode.Nest = BuildABList （QList, MList,
                                  Level+1）
25.       }
26.    }
27. }
28. return ABList
```

圖 15: BuildABList 之演算法

差別，是 List1 的資料和 List2 比對後，仍然保持同樣的順序。此演算法基本上將二個 List 的所有節點一一比對，由於每一個節點裡的 MList 中，元素編碼都由小排到大，且因為根據主實體的定義，每個編碼在每個 MList 裏都會唯一，所以我

```
Function FindTheSameLevelME
Input:   List1, List2                    /*ABList 結構*/
Output: MatchList
變數說明: MList1, MList2,
        NotMatchList /*MasterList 結構*/
        start, end /*代表第一個和最後一個 deweyid*/
        MatchNode, Node1, Node2 /*ABNode*/
        ptr1, ptr2              /*MaterList 的指標*/
        AnswerListPtr /*AnswerList 為 ABNode 結構*/
1.  While （Node1 != null）{
2.    MList1 = Node1.MasterList
3.    For （each Node2 in List2）{
4.      Create a new MatchNode and a NotMatchList
5.      MList2 = Node2.MasterList
6.      If MList1.end < MList2.start{
7.        continue
8.      }Else If MList1.start > MList2.end{
9.        continue
10.     }Else{
11.       while( ptr1 in MList1 and ptr2 in MList2 are
                                    not null）{
12.         If *ptr1 == *ptr2{
13.           MatchNode.MasterList.insert（*ptr1）
14.           ptr1++, delete ptr2, ptr2++
15.         }Else If Node2 == List2.end and
                    ptr2 == Node2.end{   //*ptr1≠*ptr2
16.         ptr1++
      //到 List2 的底,代表沒找到*ptr1,所以丟掉*ptr1
17.         }ElseIf *ptr1 < *ptr2{
18.           NotMatchList.MasterList.insert（*ptr1）
19.           ptr1++
20.         }Else{                  //*ptr1>*ptr2
21.           ptr2++
22.         }
23.       }    //end while
24.     }      //end Else
25.     If MatchNode.MasterList is not null{
26.       MatchNode.TagValueList.insert
                                  （Node1.TagValueList）
27.       MatchNode.TagValueList.insert
                                  （Node2.TagValueList）
28.       MatchList.insert（MatchNode）
29.     }
30.   } //end for
31.   If NotMatchList is not null{
32.     Node1.MasterList = NotMatchList
33.   }Else{
34.     Node1 = Node1.next}
35. }
36. return MatchList
```

圖 16: FindTheSameLevelME 之演算法

們會依據 Merge- join 的精神，將兩個 MList 中共同的主實體編碼取出。

L01	**for** //Book
L02	**group by** //Price, //Year
L03	**return** （//Price, //Year, avg（#sold））

圖 17： Q3 查詢句

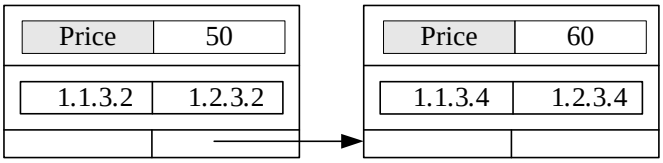

圖 18: //Price 的 ABList

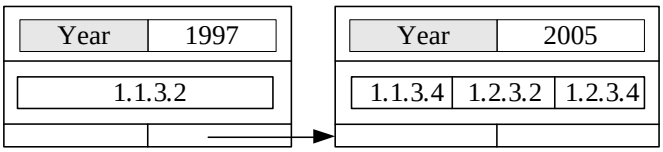

圖 19: //Year 的 ABList

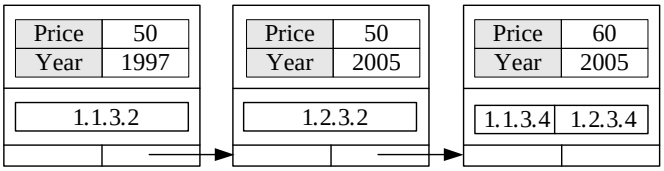

圖 20: 合併後的 ABList

以圖 17 的 Q3 和圖 1 的 XML 樹為例，處理第一個的分群運算式//Price 時，GetABList 建立的 ABList 如圖 18，處理第二個//Year 也則到圖 19。由於這兩個 ABList 裡的主實體的 Level 都是 4，所以會呼叫 FindTheSameLevel-ME 進行比對。注意到，每一個 ABNode 中的 MasterList 都是遞增排序，而 Price 第一個 node 中的兩個主實體，在 Year 的 ABList 中是存放在不同的 ABNode 中，所以我們利用演算法外層的迴圈（L01）循序處理 List1 的每一個 ABNode（Node1），而利用內層的迴圈 L03 循序處理 List2 的每一個 ABNode 節點（Node2），一一比對，最後得到的結果如圖 20 所示。

接下來我們說明如何比對 Node1 中的 MasterList（MList1）和 Node2 中的 MasterList（MList2）。如果 MList1 的最後一個編碼比

MList2 裡的第一個編碼小，或者 MList1 的第一個編碼比 MList2 的最後一個編碼大，則表示此兩個 List 不會有交集，則我們就跳過 MList2，繼續處理 List2 的下一個節點（L6-L8）。在 L11-L25 會對 MList1 跟 MList2 做交集比對。如果得到相同的編碼，表示比對成功，會將該編碼放入 MatchNode 的 Master List 中，並將 MList1 和 MList2 的指標往後移。我們會同時刪除 MList2 的此筆編碼，避免下次重覆比對。如果已經比對到 List2 的最後一個節點，都沒有找到共同的編碼，代表在 List1 的這個編碼沒有出現在 List2 中，則移動 List1 的指標處理下一筆資料。最後一種狀況是移動編碼比較小的指標，繼續比對（L17-L22），但是若是 List1 中的編碼較小，我們會將其放到 NotNatchNode 中，準備與下一個 Node2 比對（L18）。最後，如果 MatchNode 裡面的 MasterList 有資料，代表 Node1 和 Node2 有共同的資料，所以要將 Node1 和 Node2 的 TagValue 存放到 MatchNode 中以便輸出（L30-L33）。如果 NotMatchList 裏有資料，表示有元素在 Node1 而不在 Node2 中，此 List 會重新取代原來節點的 MList，繼續跟 List2 中的下一個節點比較。這樣的設計，可確保比對後的元素仍然以 List1 排序。

因為論文空間的關係，我們不詳列另兩個演算法，而以範例說明，如何針對不同結構關係的 ABList 進行合併。以圖 5 的 Q2 和圖 6 的 XML 樹為例，首先利用標籤對應的內容索引，取出每一筆主實體編碼，並由其編碼轉成的 Labeled Path，判斷 Name 和 Subject 的關係是否為叔姪關係，就是符合 anc::Publisher/Name 的表示式才輸出該主實體的編碼，如圖 21 所示。而第

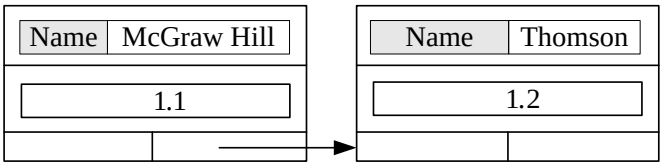

圖 21: anc::Publisher/Name 的 ABList

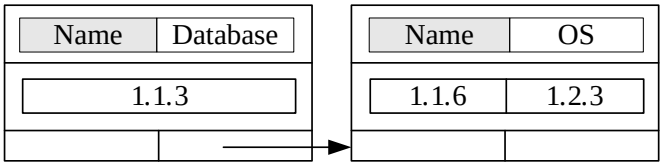

圖 22: //Name 的 ABList

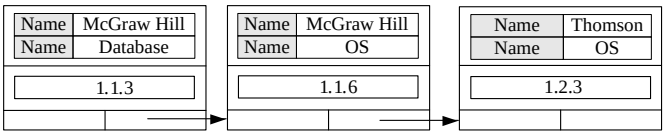

圖 23: by anc::Publisher/Name and Name

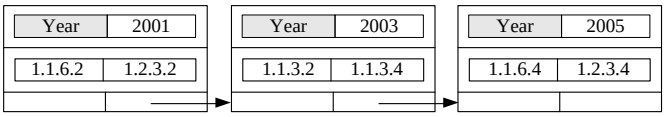

圖 24: dec::Book/Year 的 ABList

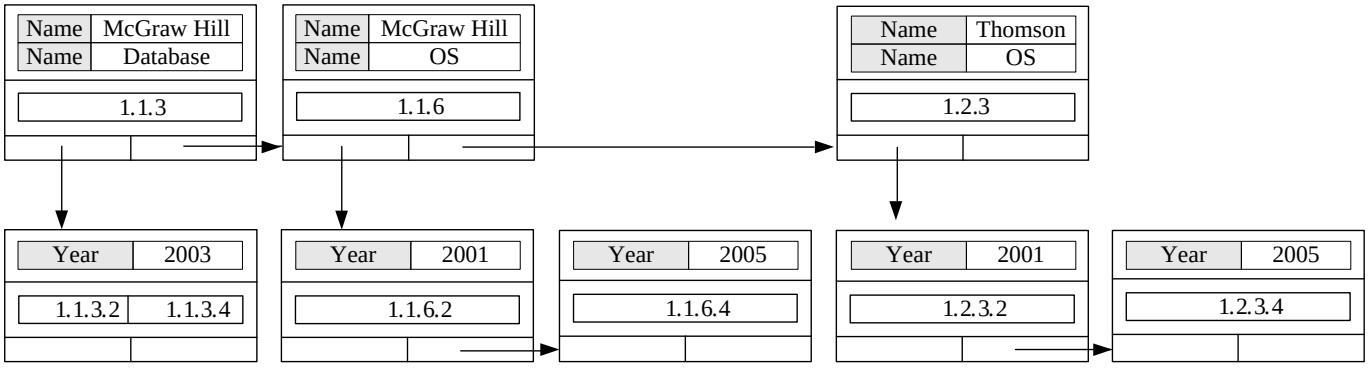

圖 25: by anc::Publisher/Name 、 Name and dec::Book/Year

二個分群運算式 //Name 要求元素是 Subject 的子孫，同樣符合該限制式的元素，會從內容索引取出，如圖 22。由於第一個 ABList 儲存 Subject 的祖先 publisher，而第二個 ABList 儲存 Subject，所以利用 FindDesendant 演算法比對兩個 List，得到圖 23。注意到，1.1 會比對到 1.1.3 和 1.1.6，而 1.2 會比對到 1.2.3。我們在此會記錄子孫的編碼。接下來 Nested 的 ABList 必須以上一層 ABList 的資料再下去分群。對應到 dec::Book/Year 的 ABList 則如圖 24。以圖 23 的 1.1.3 跟圖 23 的 ABList 比對，會得到巢狀結構的下一群 1.1.3.2 和 1.1.3.4。三個 ABList 比對的結果如圖 25 所示。

4.4. 建立答案樹

接下來的模組，會根據 ABList，建立出答案樹的結構。

以 Q1 為例，其最終 ABList 如圖 14，當處理到第一個 ABNode 節點，會建立 SubTreeRoot 節點 Price-group，再依 TagValue 建立節點 Price，內容為 50。接下來根據回傳運算式建立節點標籤存 avg(#sold)，其內容則利用主實體索引，取出該群的#sold 元素值，並計算其平均值。處理完第一層，接著對下一層重複剛剛的建立順序，得到如圖 4 的結果。

建立完初步的答案樹後，接著便處理 Having 限制。首先由第一棵 Price-group 子樹，對子樹下的節點名稱 avg（Price）判斷內容是否符合限制內容，而此節點內容為 1350 大於 750，符合限制的條件，接著往下一層 Year-group 去判斷 Having 限制，在此查詢句中沒有 Having 限制。如果某子樹不符合 Having 限制，則移除目前的 SubTreeRoot，同時停止遞迴處理。

接著處理 Rank 限制，Rank 語法是用來限制要傳回前、後 N 棵子樹。處理 Rank 的方法與處理 Having 部份相同，不同在於 Rank 不需要找子樹及其葉節點的節點名稱和內容，而是利用 Count 計算 SubtreeRoot 的順序，若該節點順序不符合 Rank 限制，則移除 SubtreeRoot 以下的整棵子樹。以 Q1 為例，最後的結果仍然如圖 4 所示。

五、實驗

在本節中我們將設計數個實驗來評估我們所提出之系統的效能。我們所進行實驗的環境是 CPU 為 Core 2 Quad 6600 的個人電腦，其記憶體為 4 GB，而採用的作業系統則為 Windows Server 2003。此外在實作此系統的工具上，我們採用的為 Visual C++ 2008。在以下的幾個實驗中，我們將採用 DBLP 之 Data Set，大小為 100MB。

我們並且和論文[3]中的 NGB_Disk 之方法進行比較，NGB_Disk 之方法會將整個 XML 文件輸入，在 scan 文件的過程中，根據元素和查詢句的關係一一處理。譬如，如果元素標籤是 α 節點，則更新 count（*）資料。如果遇到 β 或 γ 節點，且其值沒有出現在目前的答案樹的子樹中，則建立新的子樹，否則更新擁有同樣值的 Count 節點。但是若其結構關係為 anc 或 par，則建立成 DummyTree，之後再移至答案樹中。若遇到 γ 節點且無法直接計算其 Aggregation 的值時，則將其值儲存入檔案中，等文件全部讀完再一起計算。為了便於區隔，我們的系統會叫做「Index」，而論文[3]的作法將稱作「Scan」。

5.1. 回傳運算式數量之實驗

我們以 Q1~Q4 分別代表回傳運算式 1~4 個之查詢，結果如圖 26。我們可觀察到，Index 系統所需要的時間，皆低於 Scan 系統，但是隨著運算式數量的增加，時間的增加會比 Scan 系統增加的快。

5.2. 分群運算式數量之實驗

本節的實驗是控制分群運算式的數量，Q1~Q4 分別代表分群運算式的個數為 1-4，結果如圖 27。如上，Index 系統的效率仍然比 Scan 系統好。另一方面，由於分群運算式越多，答案樹的結構越複雜，所以 Scan 系統受到的影響也和 Index 系統類似。

5.3. 巢狀層數之實驗

在此實驗中主要控制查詢句巢狀層數，Q1~Q4 之數量分別為 0、1、2、3。圖 28 則為該實驗之執行時間。我們可看到 Index 系統仍然快於 Scan 系統，但由於 Index 系統針對巢狀的處理必須遞迴建立 ABList，所以時間受到此因素的影響較大。

綜合而言，Scan 系統在上述實驗中，效率都比本論文提出的 Index 系統差，不過其優點是受到查詢句複雜度的影響比 Index 系統小。但是，一般查詢句的運算式數量都不會太多，所以我們的 Index 系統在實際表現上會優於 Scan 系統。

六、相關研究

本節討論與分群查詢有關的論文。目前 W3C 所制定的 XQuery 標準語法[7]，雖然沒有明確定義 Group By 的語法，但透過巢狀結構的語法，可以實作出 Group By 的效果，Timber 系統 [2] 即是一個代表。該系統將 XQuery 轉成 GTP（Generalized Tree Patterns），然後外層 GTP 對應的 witness tree 和內層 GTP 對應的 witness tree，會利用 Left outer join 進行合併。不過該系統額外定義一個 Group-By operation，若使用者的 Query 符合特定限制，可利用該 operation 提升效率 [6]。

另一方面，論文[1]提出明確的 XQuery 的 GroupBy 語法，包含 Rank 的語法，及如何表示每一群的內容限制。論文[3]則根據該語法對 XML 資料做 Group-By 的運算，該論文將 Query 中的節點分成數類，並利用 Hash 結構，判斷遇到的資料有無出現在目前處理的樹，以決定是否要建立新的子樹，等處理完整棵 XML Tree 才去計算答案，如果是一般的 Aggregation（如 min、max 和 sum 等等），就直接在 Tree 中計算結果。在第 5 節中，我們已經和該系統的效率比較。

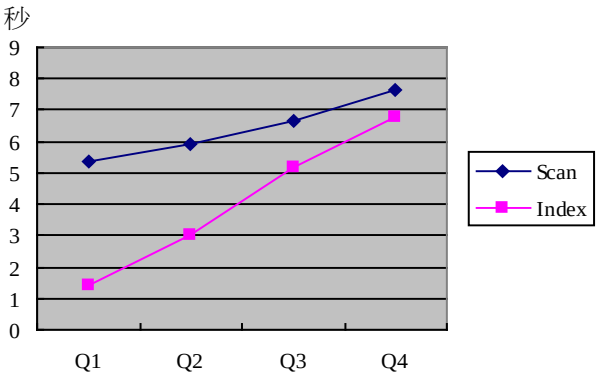

圖 26: 回傳運算式數量之實驗曲線圖

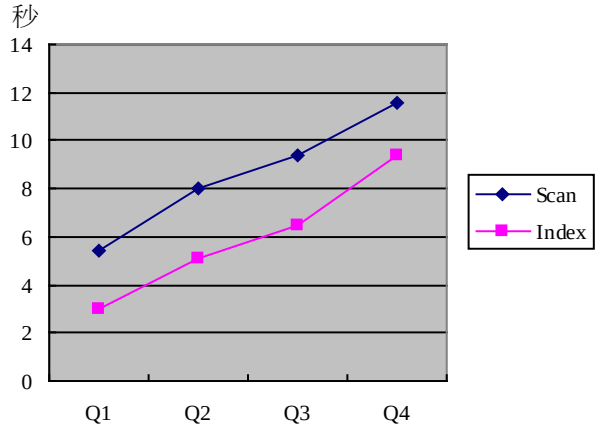

圖 27: 分群運算式數量之實驗曲線圖

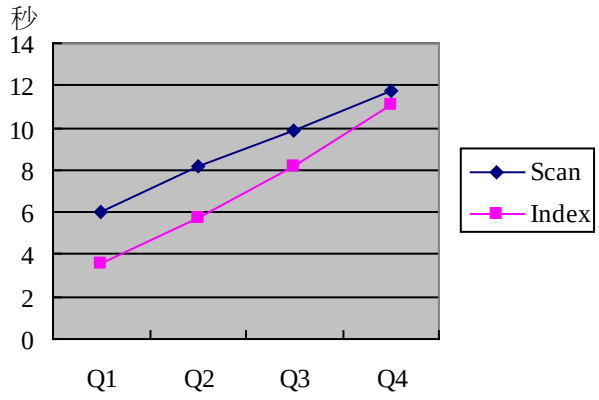

圖 28: 巢狀層數之實驗曲線圖

七、結論與未來方向

在本論文中，我們討論如何提升分群查詢句在 XML 資料處理上的效能。其基本精神是利用內容索引，取出對應分群運算式的 MList，再根據結構關係兩兩比對建立成 ABList。該資料結構已將資料分群且分層表示，所以可以很容易地轉成答案樹的結構。接下來，我們可以透過主實體索引快速地抓取樹中每一層所需要的回傳資料，實驗結果顯示，我們比目前一個著名的系統表現更好。未來，我們希望可以改善抓取回傳資料的效率，並且討論如何對分群查詢處理進行最佳化。

誌謝

此計畫由國科會贊助，編號為：

NSC97-2221-E-019-028

八、參考文獻

[1] K. Beyer, D. Chamberlin, L. S. Colby, F. Özcan, H. Pirahesh, Y. Xu, “Extending XQuery for Analytics”, Proceedings of the 24th International Conference on SIGMOD Conf, Maryland, USA, 2005.

[2] Z. Chen, H.V. Jagadish, L. V. S. Lakshmanan, S. Paparizos, “From Tree Patterns to Generalized Tree Patterns: On Efficient Evaluation of XQuery”, Proceedings of the 29th International Conference on VLDB Conf, Berlin, Germany, 2003.

[3] C. Gokhale, N. Gupta, P. Kumar, L.V.S. Lakshmanan, R. Ng, B.A. Prakash, "Complex Group-By Queries for XML", Proceedings of ICDE Conference, Pages: 646-655, April 15-20, Istanbul, 2007.

[4] Z. Liu, Y. Chen, "Identifying Meaningful Return Information for XML Keyword Search", Proceedings of the SIGMOD Conference, Beijing, China, 2007

[5] J. Lu, T. W. Ling, C. Chan, T. Chen, “From Region Encoding To Extended Dewey: On Efficient Processing of XML Twig Pattern Matching”, Proceedings of the 31st International Conference on VLDB Conf, Trondheim, Norway, 2005.123

[6] S. Paparizos, S. Al-Khalifa, H. V. Jagadish, L. Lakshmanan, A. Nierman, D. Srivastava, Y. Wu, “Grouping in XML”, Proceedings of the International Conference on EDBT 2002 Conf, Prague, Czech Republic, 2002.

[7] World Web Consortium（W3C）, “XQuery 1.0: an XML Query Language”, January 2005. http://www.w3.org/TR/xquery/