串流 XML 資料之路徑查詢處理

陳耀輝　黃文彬　何銘啟　余美伶　曾柏銜
國立嘉義大學資訊工程學系
{ychen, s0930302, s0920206, s0930314, s0930305}@mail.ncyu.edu.tw

摘要

XML 近年來已經成為網際網路上資料表示及資料交換的標準格式。在許多網路應用中，XML 資料不再僅以單一文件的形式傳送，而是以一種連續性的串流資料形式傳輸。傳統的資料庫系統已經無法支援此種串流形式的資料型態，一般的查詢處理演算法也無法處理大量的串流 XML 資料。因此，本論文為 XPath 查詢語法提出新的查詢處理演算法以解決串流 XML 資料之路徑查詢處理問題。我們利用開始標籤及結束標籤引導查詢處理的進行，用計數器記錄標籤個數以快速決定輸入的資料是否符合查詢式，並以 Stack 及指標結構儲存暫時的中間結果。當要取得結果時，僅需遵循指標即可由 Stack 中重建正確的結果。我們提出的演算法不僅可以處理單一路徑查詢，也可以處理分支路徑查詢。實驗結果證明，使用此法在效能上優於現有索引結構或狀態機的查詢處理機制。

關鍵字：串流 XML 資料、XPath、查詢處理、分支路徑查詢

1. 簡介

伴隨著網際網路應用的興起，XML 資料可以用一種連續性的資料串流（Data Stream）形式傳輸，如：股票行情、線上拍賣、與感應器的資料等等。因為資料是不斷地產生，傳統的資料庫系統已無法支援此種串流形式的新資料。該如何有效處理針對串流 XML 資料的查詢式，便是我們所要探討的問題。

在傳統的查詢處理技術中經常藉由建置資料索引來加快查詢處理的速度，目前已有許多針對XML資料建立索引機制的文獻發表 [1][3][6][8][9][10][14][16][17][18][21][22][23]。Holistic Twig Join [3] 在XML文件上建立索引結構及利用多重Stack來處理節點之間的關係，以避免產生過多的中間處理結果(Intermediate Results)。如 Figure 1 所示，在資料上先建立索引結構（LeftPos: RightPos, LevelNum），再根據查詢式將資料各自放到Stack中並用指標指向前一個Stack，以建立彼此間的關係，最後，依據指標取得查詢結果。因為C_1指向B_2，而B_2指向A_2，則得到一個結果 [A_2, B_2, C_1]。而A_1在A_2之下，表示A_1為A_2之祖先節點，所以 [A_1, B_2, C_1] 也是結果之一。此外，[A_1, B_1, C_1] 亦是其中之一的結果。然而，因為A_2在A_1之上，而B_1是指向A_1，所以 [A_2, B_1, C_1] 並不是此查詢的結果。

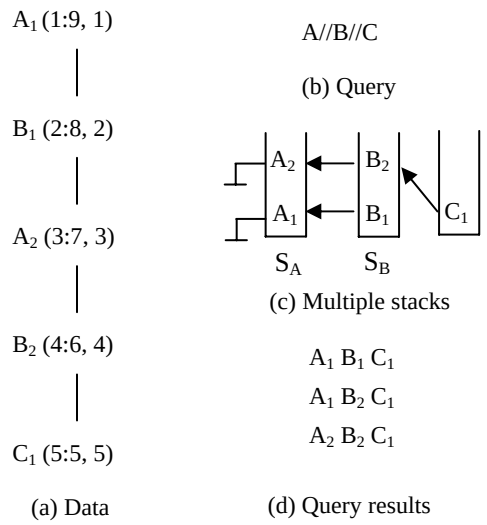
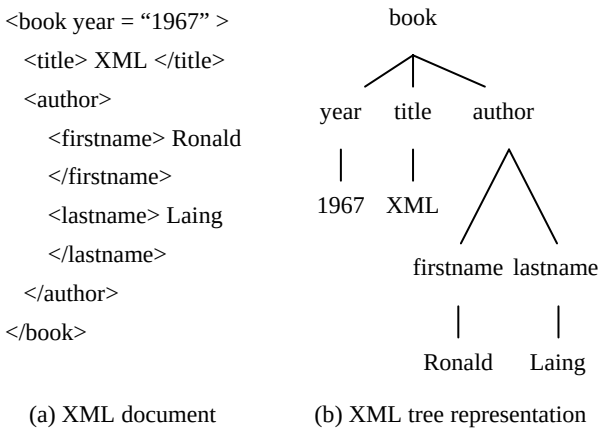


Figure 1. Compact encoding of answers using stacks.

但在串流型態下，除非事先將資料建立索引結構，不然則需在接收串流資料的過程中建立索引，以便查詢之處理。而 [4] 所提出的 Index-Filter，利用已建置索引結構的串流資料，配合其所提出的三種情況來刪除不必要的資料並找出所有結果。但其僅能處理單一路徑查詢，若針對分支路徑查詢則需要非常耗時的 join 運算。[6] 提出 BLAS 系統，分別對分支路徑查詢式上的"/"和"//"作 P-labeling 及 D-labeling，以減少 join 次數和 join 最佳化。另外，[21][22] 皆提出在查詢式和資料上建立索引結構，以避免分支路徑查詢耗時的 join 運算，但卻不適用於串流資料。

由於在串流查詢處理時資料多數並未建立索引結構，因此要有不同的查詢處理方式。目前已有許多用 Finite Automata 處理查詢的論文被提出 [2][11][12][13][19][20]，通常它們可以一次處理大量的單一路徑查詢，但是卻需要龐大的記憶體空間。[5] 利用查詢式中的 "/" 運算子來建立索引結構，以減少不必要的比對動作；而 [20] 則使用一個 Hierarchical Pushdown Transducer (HPDT) 來處理路徑查詢；[7] 則對 XPath 利用多重 Stack 來暫存符合的資料，以便找出最後結果；[8][9] 整合 DBMS 及串流 XML 資料之間的瓶頸，在 DBMS 部份先將串流 XML 資料編碼後才傳輸，而接收端只要配合 DBMS 的編碼方式即可很快找出所要的結果。



(a) XML document (b) XML tree representation

Figure 2. XML document and tree representation.

XML 查詢主要是在處理路徑比對的問題，而其困難點在於分支路徑查詢。目前多數解決分支路徑查詢的方法是先將分支路徑分解成多條單一路徑，然後分別對單一路徑做處理，等所有單一路徑查詢結束後，再將結果做 join 運算，以取得分支路徑查詢的所有正確資料。此方法的缺點在於無法確保每條單一路徑查詢在做 join 運算時能比對成功，若多數單一路徑查詢結果在做 join 運算時被刪除，則會浪費過多時間在處理不符合結果的單一路徑查詢上。

查詢路徑是有順序性的，我們利用深度優先搜尋法（Depth First Search）取得查詢元素的順序。在我們的方法中查詢式上的每個節點各自擁有一個計數器及 Stack 來暫存符合的資料，Stack 之間則用指標連接，以便重建出正確結果。而計數器主要是用來記錄標籤的個數以決定是否符合查詢元素之用。然而，為了處理分支路徑查詢及避免 join 的運算，則必須先暫存部份符合的資料，但是符合的資料又必須遵守查詢式的條件要求，例如：給予一分支路徑查詢式“//A[//B]//C”，則表示 A 元素必須先符合之後，B 元素才會滿足條件，最後，C 元素才可接受並得到結果，但是符合 C 元素的資料不可被包含在 B 元素之內。因此，我們根據不同的查詢結構給予不同的指標，以便於重建正確的結果。本篇論文的主要貢獻是提出一種串流 XML 資料之路徑查詢處理方法並且：

- 僅需利用串流 XML 資料中開始標籤及結束標籤，就能引導查詢處理的進行。
- 針對單一路徑查詢，僅需使用計數器、Stack 及單一指標結構即可取得查詢之結果。
- 針對分支路徑查詢，除了使用計數器及 Stack 外，根據不同的查詢式結構給予不同的指標，以區別符合的串流資料彼此之間的關係。
- 實驗結果證明，我們的方法無需任何索引結構，即可處理 XML 串流資料的查詢，並且在效能上更優於使用 Automata 的機制。

本文的文章結構如下，第二節介紹 XML 資料型態及路徑查詢式，第三節說明路徑查詢處理的方法，第四節為實驗評估結果，最後第五節為結論。

2. XML 資料型態及路徑查詢式

一份 XML 文件是由許多的元素及屬性所構成，元素之間可組成一個巢狀式結構，並且存有父子關係及祖孫關係，有了這樣的關係，可以輕易地將 XML 文件轉換成樹狀結構。此樹狀的根為所有底下節點的祖先，而當有兩節點存在同一條路徑上時（由根到葉節點），則較上層節點者稱為祖先節點，較下層節點則為子孫節點。若兩節點之間只相差一個階層，則上層者稱為父節點，下層者則稱為子節點。如 Figure 2 所示，為一簡單的 XML 文件及其樹狀結構。

串流 XML 資料是在 XML 文件上循序瀏覽所取得的一連串資料。如果不考慮屬性與內容，可以將它們視為一連串的開始與結束標籤，如：<A> <B>…</B> <C>….</C> </A>。但是成對的開始及結束標籤是不可以相互交錯的，例如一連串的<A> <B> </A> </B>是不允許出現在 well-formed 的 XML 文件中的。因此，我們可以利用開始與結束標籤來處理路徑查詢式。

XPath 查詢語言 [28] 讓使用者用路徑表示法從 XML 樹狀結構中找出相對應的資料。例如：給予一查詢式“book[@year]//author[firstname]/lastname”，表示要在 XML 樹狀結構上找出具有屬性 year 的 book，且 book 具有子孫節點 author，而此 author 又必須具有兩個子節點 firstname 及 lastname 的路徑結果。其中“//”表示元素之間為祖孫關係，“/”表示元素之間為父子關係。而“[ ]” 表示為分支路徑，若其中包含“@”則表示為具有屬性的元素，而不是分支路徑。

3. 路徑查詢處理

XPath 查詢式是以路徑表示法來呈現所要查詢資料的位置。路徑是一連串循序的元素，其每個元素各自擁有標籤名稱。查詢式的條件通常是尋找出某些 XML 文件路徑上擁有相同名稱的節點，這些標籤名稱之節點排列順序及節點間的關係也必須符合查詢式的要求。在處理串流 XML 資料的查詢時，可以利用 SAX [27] 將 XML 文件剖析成各個節點的開始及結束標籤。而其節點間的關係則利用階層數來區分，如：兩個節點的階層數相減為 1，表示此兩節點具有父子關係；若兩節點的階層數相減大於 1，則表示此兩節點為祖孫關係。有了這些初始條件，就可以提出不同的演算法來解決複雜的路徑查詢處理問題。

3.1 單一路徑查詢處理

我們提出的單一路徑查詢處理方法與 Figure 1 所表示的 Holistic Twig Join [3] 方法類似，都是使用多重 Stack 及指標來記錄查詢的中間處理結果，以節省儲存空間。但我們的方法不需要在 XML 資料上建立索引結構，而是使用開始和結束標籤及計數器來引導查詢處理的進行，所以特別適合處理串流 XML 資料。我們透過深度優先搜尋法取得查詢元素的順序，並且給予查詢式的每個元素各自一個計數器及 Stack 來記錄符合的資料，Stack 彼此間則利用指標來建立關係。計數器是用來判斷來源資料符合查詢式的哪些元素，當計數器為 0 時，表示該 Stack 目前無符合查詢式的資料；若不為 0 時，表示目前已有符合的資料。Figure 3 為單一路徑查詢演算法，完整演算法請參照 [14]。

Figure 4 為PATH演算法的例子。輸入資料如Figure 4(a)所示，Figure 4(b)為一路徑查詢式“A//B/A”，Figure 4(c) 為讀取開始及結束標籤的處理過程。首先讀取 <A_1>，A_1雖符合查詢式中的兩個元素A，但因還未讀取到元素B，故無法符合查詢式中的第二個A。因此將 $Count_1$ 加 1 並將A_1存入$Stack_1$中，但因$Count_2$為 0 故不將A_1存入$Stack_3$中。接下來讀取<B_1>，B_1符合查詢式中的元素B且其前一個計數器$Count_1$不為 0，因此將$Count_2$ 加 1 並將B_1存入對應的$Stack_2$中，再建立指標指向前一個$Stack_1$。第三步驟讀取<A_2>，因A_2符合查詢式A//B/A中的兩個元素A，又因$Count_2$不為 0 且符合B/A的父子關係，故將A_1存入$Stack_1$和$Stack_3$中，其他和步驟二同。因A_2符合查詢式的最後一個元素，所以可以沿著指標所指到的Stack依序輸出結果。在此只將標籤輸出而不將其移出Stack，因這些標籤還可能和未到達的資料流產生符合查詢式的結果。第四步驟讀取<A_3>，A_3符合查詢式中的兩個元素，且B_1已在$Stack_2$，$Count_2$不為 0，但B_1與A_3不符合查詢式中B/A的父子路徑關係，因此只將A_3存入$Stack_1$中。接下來的步驟都是讀取結束標籤，因此只需將相關的計數器減 1，並將標籤從對應的Stack中取出即可。

3.2 分支路徑查詢處理

目前查詢處理的作法大多專注在處理單一路徑的查詢上，但對多重路徑的查詢卻沒有太多貢獻。若遇上這方面的問題，一般的作法是將分支路徑分成一條條單獨路徑做處理，最後再將所有符合查詢條件的單一路徑組合成分支路徑的查詢結果。然而，較好的方式應當視整個分支路徑為一體，並利用路徑之間的關係來執行查詢。在我們提出的分支路徑查詢處理方法中，查詢式的每一個元素都有各自的 Stack 及計數器以便記錄查詢的中間處理結果。單一路徑查詢處理只要考慮一條路徑，所以當接收到路徑的最後一個元素時就可以跟隨反向的指標(由較大編號的 Stack 指向較小編號的 Stack)來重建結果。在分支路徑查詢時，完全符合一條路徑並不能保證會符合全部的分支路徑，但前面符合的路徑要保留起來以便後來的資料使用。一直要等到分支查詢式的第一個元素結束後才可以用追蹤

```
//輸入：循序的 XML 資料，輸出：所有結果
01  For (每個 SAX Parser 輸出的節點) {
02    If (節點符合查詢元素 i) {
03      If (節點為開始標籤) {
04        If (節點為查詢式的第一個元素) {
05          將節點i存入Stack1中; Count1 ++;
06          If (節點為查詢式的最後一個元素) {
07            將此節點輸出為結果;
08            getResults (元素 i-1, 節點指標); } }
09        Else { /* 不是第一個元素 */
10          If (Count_{i-1} != 0) { /*父節點符合查詢式*/
11            If (satisfyPathRelationship(節點,元素 i)){
12                將節點i存入Stack_i中; Count_i ++;
13                建立指標指向Stack_{i-1}的頂端;
14                If (節點為查詢式的最後一個元素) {
15                  將此節點輸出為結果;
16                  getResults (元素 i-1, 節點指標); }
17                } } } }
18        Else { /* 節點為結束標籤 */
19          If (Count_i != 0) { /* 此節點符合查詢式 */
20            Count_i--，將節點從Stack_i移出; }
21        } } }

Function satisfyPathRelationship(節點, 元素 i)
//輸入：開始標籤的節點及查詢元素 i，輸出：布林值
01    If (元素 i 與元素 i-1 關係為“//”) {
02      傳回 true; }
03    Else {
04      If (元素 i 與元素 i-1 關係為“/”) {
05        If (節點與Stack_i中最頂端節點level數差 1) {
06          傳回 true; }
07        Else 傳回 false;
08      } }

Procedure getResults (元素 i, 節點指標)
//輸入：元素 i 及節點指標，輸出：所有結果
01.  If (元素 i 非查詢式中第一個元素) {
02.    If (元素 i 與元素 i+1 間的關係為“/”) {
03.      利用指標取出指向Stack_i的節點並輸出為結果;
04.      getResults (元素 i-1, 節點指標); }
05.    Else {
06.      利用指標從上至下取出指向Stack_i的節點並依序輸出為結果;
07.      getResults (元素 i-1, 節點指標); } }
08.  Else { /* 為第一個元素 */
09.    If (元素 i 與元素 i+1 間的關係為“/”) {
10.      利用指標取出指向Stack_i的節點並輸出為結果; }
11.    Else {
12.      Stack_i中指標指到的節點至底部節點依序輸出為結果; } }
```

Figure 3. Algorithm PATH.

指標的方式來重建結果。由於要保留之前符合的路徑，我們採用正向的指標(由較小編號的 Stack 指向較大編號的 Stack)並用它們記錄分支結構。

查詢處理時首先透過深度優先搜尋法取得在分支路徑查詢式上元素的順序並依序給予元素編號。例如一個分支路徑查詢式“//A[//B]//C”，透過深度優先搜尋法依序取得 A、B、C 三個查詢元素。表示在處理分支查詢時 A 元素必須先符合之後，B 元素才會滿足條件，最後 C 元素才可接受並得到結果，但是符合 C 元素的資料不可被包含在 B 元素之內。我們歸納出串流資料上的每個標籤需符合以下條件才可運用：

- Root 元素：

  無限制條件，其為查詢式的第一個元素，當然只要符合第一個元素的資料標籤即符合條件。如 Figure 5(a) 中的 A 元素。

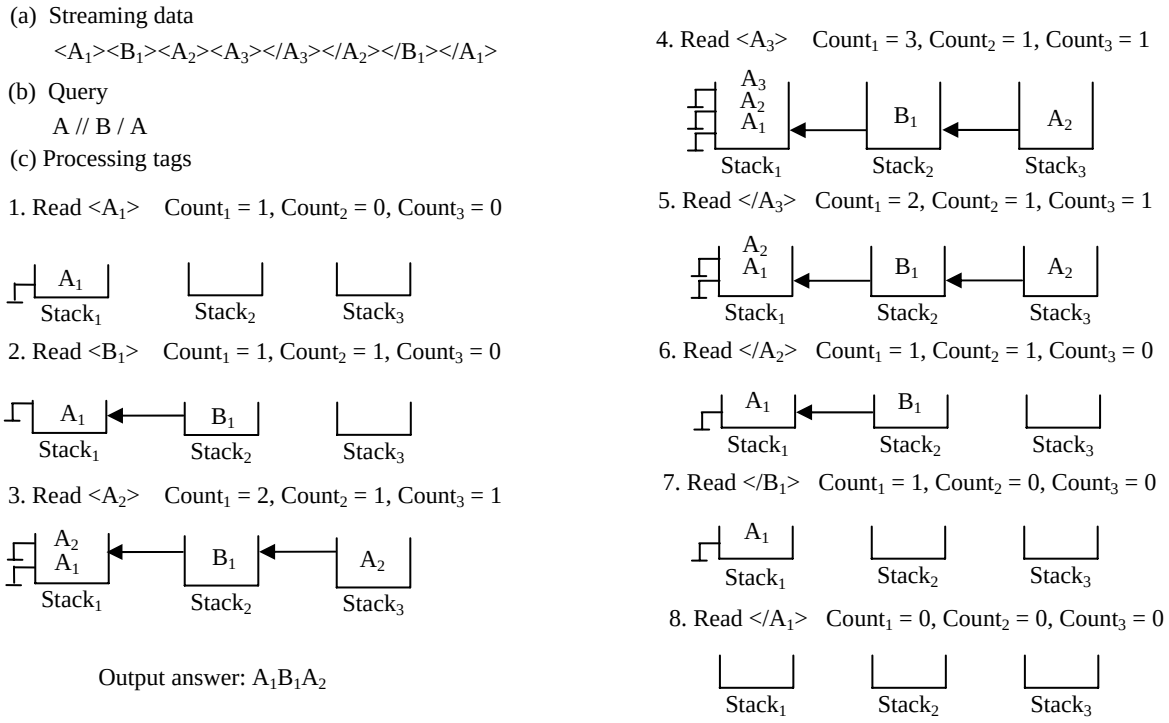
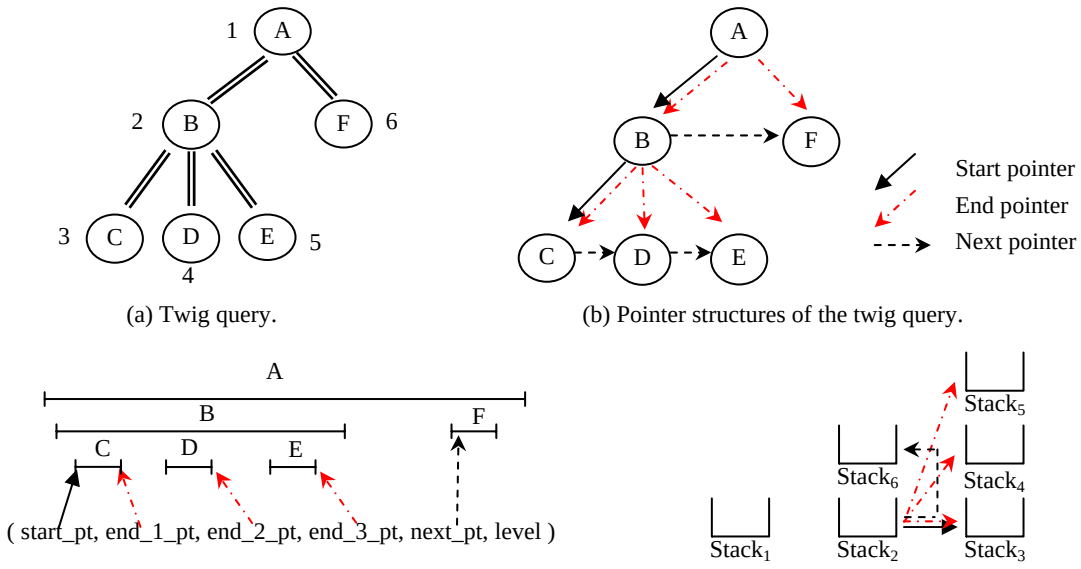


Figure 4. An example for algorithm PATH.



(a) Twig query.

(b) Pointer structures of the twig query.

(c) Pointer structures for the B element.

(d) Stack relationship for the B element.

Figure 5. Pointer structures and stack relationship for the twig query.

● 左路徑元素：

此資料標籤必須在前一標籤的範圍之內，也就是其父元素已在開始狀態且尚未收到結束標籤，則此資料標籤必落在其範圍之內，為其子元素。故此條件為前一元素是開始狀態才會符合。如 Figure 5(a) 中的 B、C 元素。

● 右路徑元素：

此資料標籤必須符合在其父元素的範圍之內，且其左兄弟必須是已在結束狀態，才會滿足條件。如 Figure 5(a) 中的 D、E、F 元素。

我們的目的是要找出所有符合查詢式的結果，因此利用計數器來判斷節點是否仍在開始狀態，而用 Stack 來暫存符合的結果。由於分支路徑查詢式的每個元素具有以上歸納出來的條件，故定義每個 Stack 中的節點指標結構為(start_pt, end_1_pt, end_2_pt, …, end_n_pt, next_pt, level)，其中 level 為此節點在 XML 文件中的階層數，主要是被用來區分父子與祖孫關係。next_pt 為右兄弟的開始位置，start_pt、end_n_pt 分別表示其路徑下的開始位置及第 n 條路徑的結束位置。換言之，一個節點的指標結構（start_pt, end_n_pt）表示其所包含的子節點範圍。舉例來說，如 Figure 5(b) 中的 B 元素，其本身是分支節點(有三個子節點)且具有右兄弟，故其指標結構為(start_pt, end_1_pt, end_2_pt, end_3_pt, next_pt, level)。將元素 B 以範圍來表示並且搭配指標結構就如 Figure 5(c)所示，可以清楚看到 start_pt 所指的位置為元素 C 的開始位置，end_1_pt 為元素 C 的結束位置，end_2_pt、end_3_pt 分別為元素 D、E 的結束位置，而 next_pt 所指的位置為其右兄弟元素 F 的開始位置。Figure 5(d) 為元素 B 搭配指標結構及 Stack 的表示圖。因此，我們可以使用計數器、Stack 並搭配指標結構來處理複雜的分支查詢式。

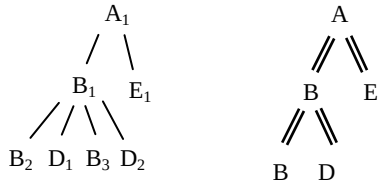
查詢時，先將查詢式做深度優先搜尋，依序給予每個元素各自一個計數器及 Stack 來記錄符合的資料。Stack 彼此間則利用指標建立關係，共有開始指標、兄弟指標和結束指標三種。Figure 6 為 TWIG 演算法，完整的演算法請參照 [14]。

Figure 7 為TWIG演算法的例子。Figure 7(a)為輸入資料，Figure 7(b) 為分支路徑查詢式“A[//B[//B]//D]//E”，Figure 7(c)為讀取開始及結束標籤的處理過程。首先讀取<A_1>，此時A_1符合查詢式路徑的根元素，將$Count_1$加 1，並存入$Stack_1$中，建立start_pt指向$Stack_2$。讀取<B_1>後，因為B_1符合查詢式路徑的兩個元素，而B_1指向B_1為無效的，故$Stack_3$中的B_1會被直接捨棄，只將其存入$Stack_2$中並將$Count_2$加 1。讀取<B_2>也符合兩個查詢式路徑的元素，故B_2會存入$Stack_2$ 和 $Stack_3$ 中，並將 $Count_2$ 和 $Count_3$ 加 1，

```
//輸入：循序的 XML 資料，輸出：無
01.   For (每個 SAX Parser 輸出的節點) {
02.     If (節點符合查詢元素 i) {
03.       If (節點為開始標籤)
04.         processStartTag(節點, 元素 i);
05.       Else /* 節點為結束標籤 */
06.         processEndTag(節點, 元素 i); } }

Procedure processStartTag(節點, 元素 i)
//輸入：為開始標籤的節點及查詢元素 i，輸出：無
01.   If (節點為查詢式的第一個元素) {
02.     Count1++; 將此節點存入Stack1中;
03.     建立 start_pt 指標; }
04.   Else { /*節點非查詢式的第一個元素*/
05.     If (Countparent != 0 ) { /* 父元素的count不為 0 */
06.       If (節點在左路徑)
07.         pushNodeIntoStack(節點, 元素 i);
08.       Else {  /*節點在右路徑*/
09.         If (有 next_pt 從左兄弟指向此節點)
10.           pushNodeIntoStack(節點, 元素 i); } } }

Procedure pushNodeIntoStack(節點, 元素 i)
//輸入：為開始標籤的節點及查詢元素 i，輸出：無
01.   If (satisfyPathRelationship(節點, 元素 i)) {
02.     Counti++; 將此節點存入Stacki中;
03.     If (在查詢式的單一路徑上此節點非葉元素)
04.       建立 start_pt 指向 child stack;
05.     Else {
06.       If (元素 i 與元素 i-1 關係為“/”)
07.         調整其父節點的 start_pt 指標; } }

Procedure processEndTag (節點, 元素 i)
//輸入：為結束標籤的節點及查詢元素 i，輸出：所有結果
01.   For (尋找包含此節點的Stacki) {
02.     Counti--;
03.     If (節點非查詢式第一個元素) {
04.       If (在查詢式的單一路徑上此節點為葉元素) {
05.         If (在查詢式中此節點有右兄弟元素)
06.           建立 next_pt 指標指向右兄弟的 Stack; }
07.       Else { /* 節點非葉元素 */
08.         If (指標所指 children stacks 位置不為空) {
09.           刪除不符合查詢式的資料;
10.           建立 end_n_pt 指標指向 children stacks;
11.           If (節點在查詢式中有右兄弟元素)
12.             建立 next_pt 指標指向右兄弟的 Stack; }
13.         Else /* 不符合查詢式 */
14.           移除Stacki最上端節點或調整此節點
              start_pt; } }
15.     Else { /* 節點為查詢式第一個元素 */
16.       If (指標所指 children stacks 位置不為空) {
17.         刪除不符合查詢式的資料;
18.         建立 end_n_pt 指標指向 children stacks;
19.         If Count1 = 0, getTwigResults (元素i, null, null,
            0); }
20.       Else /* 不符合查詢式 */
21.         移除Stacki最上端節點或調整此節點start_pt; } }
```

Figure 6. Algorithm TWIG.

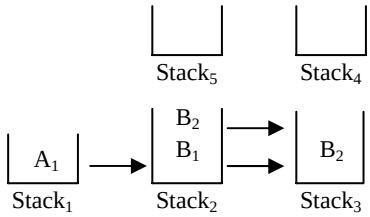
(a) Streaming data tree (b) Twig query tree.



(c) Algorithm TWIG example.

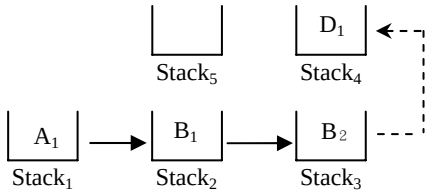
Read <A_1>、<B_1>、<B_2>

$Count_1 = 1, Count_2 = 2, Count_3 = 1, Count_4 = 0, Count_5 = 0$



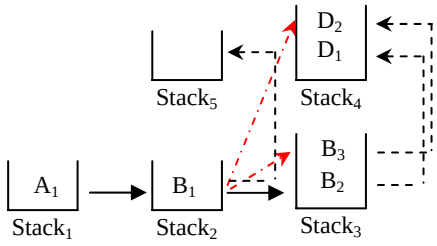
Read </B_2>、<D_1>、</D_1>

$Count_1 = 1, Count_2 = 1, Count_3 = 0, Count_4 = 0, Count_5 = 0$



Read </B_1>

$Count_1 = 1, Count_2 = 0, Count_3 = 0, Count_4 = 0, Count_5 = 0$



Get answers:

"$A_1B_1B_2D_1E_1$, $A_1B_1B_2D_2E_1$, $A_1B_1B_3D_2E_1$"

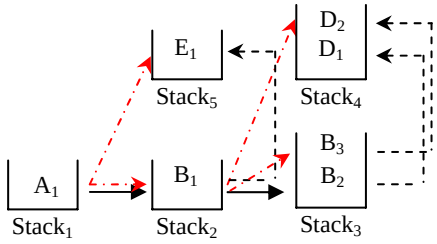


Figure 7. An example for algorithm TWIG.

然而$Stack_2$中B_2的start_pt指向$Stack_3$的B_2是無效的，故將此指標指向$Stack_3$中空的位置。接下來讀取</B_2>，由於為結束標籤且屬於查詢路徑的葉元素，表示查詢式的左路徑已結束，將建立next_pt指向$Stack_4$，需注意上一步驟中B_2指向$Stack_3$空的位置之start_pt，已變成無效的中間產物，所以會被捨棄，並將$Count_2$減 1。接著讀取<D_1> </D_1>，因D_1在查詢式中為葉元素且無右兄弟元素，於是將D_1存入$Stack_4$。後續的處理方式和之前的步驟相似。讀取</B_1>時，由於有右兄弟路徑，則會建立next_pt指向$Stack_5$，需注意的是此時的$Count_2$為 0，表示其子查詢路徑已經結束，故B_1會建立end_1_pt及end_2_pt分別指向$Stack_3$最上層的標籤B_3和$Stack_4$最上層的標籤D_2。讀取<E_1> </E_1> </A_1>時則將E_1存入$Stack_5$中，建立A_1的end_1_pt及end_2_pt分別指向$Stack_2$的B_1和$Stack_5$的E_1。最後依照每個標籤的指標範圍，可取得查詢結果[$A_1B_1B_2D_1E_1$]，[$A_1B_1B_2D_2E_1$]及[$A_1B_1B_3D_2E_1$]。

4. 實驗評估

4.1 環境設定

評估效能實驗平台的作業系統是 Windows 2003 Server，CPU 為 Intel Pentium 4 2.4 GHz，主記憶體 1GB，使用 JAVA 做為撰寫實驗的程式語言，並利用 Borland JBuilder 9 程式專案管理軟體協助實驗程式的開發。實驗資料使用人造資料和實際資料各兩種。Synthetic 人造資料是由 IBM XML Generator [24] 所產生，資料量約為 116MB。另一個 Auction 人造資料是由 XML-benchmark project [29] 所提供，其資料量大約為 111MB。NASA 實際資料為 GSFC/NASA XML Project [26] 所提供的天文資料，資料大小約為 23MB。另一個 SigmodRecord 實際資料則由 ACM SIGMOD Record: XML Version [24] 所提供，資料量大約為 704KB。

實驗主要的評估對象為 XSQ [20] 及 XMLTK [12]。XSQ 是一個利用 pushdown transducers 來處理 XPath 查詢式的串流 XML 資料處理機制。其實驗環境與我們的環境相同，亦使用 JAVA 做為其程式語言。但其所提供的程式無法處理分支路徑上左路徑的祖孫關係及較長的左路徑，因此，在處理分支路徑時 XSQ 查詢結果並不正確。XMLTK 亦是處理串流 XML 資料的路徑查詢處理機制，使用懶散式 Deterministic Finite Automaton (DFA) 來處理 XPath 查詢式。而 [12] 所提供的程式，其實驗環境為 RedHat Linux 9.0，Intel CPU P4 1.5GHz，主記憶體為 512MB，使用 C 程式語言。[12] 提供的程式無法處理分支路徑，不過其在處理單一路徑上效能比 XSQ 佳 [7]，故我們亦拿來當比較的對象之一。

為了評估串流 XML 資料的查詢時間（秒），實驗中每個演算法皆執行 30 次，並將查詢時間取平均值計

Table 1. Execution time (seconds).

Dataset	XPath Queries	PATH	TWIG	XSQ	XMLTK
Auction (111MB)	site/people/person/name	11.69	11.71	27.56	21.84
	site/people/person[@id]/name	11.66	11.79	27.24	24.04
	open_auctions/open_auction/bidder/increase	10.85	11.06	30.80	25.85
	person[address]/profile[interest]/business	N/A	10.81	29.55	N/A
	person[@id][address]/profile[@income][interest]/business	N/A	10.72	N/A	N/A
NASA (23MB)	dataset//reference//holding	2.73	2.78	11.74	6.59
	dataset//reference[@type]//holding[@role]	2.68	2.74	11.69	N/A
	dataset[title]/descriptions[abstract]/details	N/A	3.41	11.13	N/A
	dataset[descriptions/abstract]/history/ingest/creator/lastName	N/A	4.12	N/A	N/A
	dataset[title]/descriptions[@xml:lang][abstract]/details	N/A	3.34	N/A	N/A
SigmodRecord (704KB)	issue//articles//author	0.17	0.18	0.49	0.43
	issuesTuple//articleTuple/title	0.15	0.17	0.44	0.39
	Issues//articles//author[@AuthorPosition]	0.17	0.18	0.51	N/A
	issuesTuple[volume]//articlesTuple[title]//author	N/A	0.25	1.12	N/A
	issues//articles[//title]//authors/author	N/A	0.34	N/A	N/A
Synthetic (116MB)	nation//country//city//company//manager//department//place //employee//email	16.28	16.68	105.98	22.21
	countries//city[@cityID]/place//countries/country	11.19	11.55	43.93	24.46
	company//countries[//city[@cityID]//department//name]//place	N/A	33.39	N/A	N/A
	countries[place]//country//place	N/A	12.41	61.48	N/A

算，但測量時間不包含查詢式的剖析時間。此外，我們還有評估單一路徑的最大記憶體使用量。

4.2 實驗結果

在 Table 1 中，XMLTK 不支援查詢式的最後一個元素具有屬性及分支路徑查詢式。PATH 亦不支援分支路徑查詢。而 XSQ 不支援左路徑較長、左路徑上有祖孫關係、以及具有屬性與左路徑在一起的情況。這些不支援的查詢式，在 Table 1 中皆以 N/A 來表示。

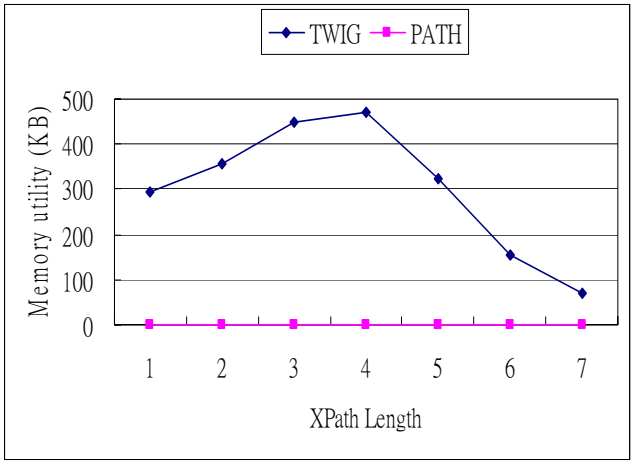


Figure 8. Memory utility for synthetic data.

Table 1 所示，對單一路徑查詢式而言，PATH 演算法和 TWIG 演算法皆優於 XSQ 及 XMLTK，因為 PATH 演算法只有利用計數器的加減運算及堆疊器來記錄資料而已。TWIG 比 PATH 稍差的原因在其必須等到符合查詢式第一個元素的結束標籤時才取出所有正確結果，故所花的時間比 PATH 還久一點。而 XSQ 及 XMLTK 處理效能上比 PATH 差，其原因是 Automata 產生的狀態過多。TWIG 演算法比 XSQ 佳的原因在於 XSQ 除了處理 pushdown transducers 外，其 HPDT 在處理祖孫關係時需要迴圈的判斷，而 TWIG 只需比較階層數即可。至於 XMLTK 執行效能優於 XSQ 的原因是 XMLTK 除了使用 DFA 外，另外也利用 NFA 來加快處理速度。對於分支路徑查詢式而言，TWIG 演算法可以適用在任何查詢式，而且具有相當不錯的效能。

在空間花費上如 Figure 8 所示，評估 Synthetic 人造資料，在單一路徑查詢式長度變化下的最大記憶體使用量。Figure 8 顯示 PATH 演算法幾乎沒有使用到記憶體，原因在於其不用記錄已經過去的資料。而隨著查詢式的長度變化，TWIG 演算法所需要的記憶體使用量則不同。TWIG 演算法需要使用較多的記憶體，主因是必須記錄將來可能還會被使用到的歷史資料。而由於所查詢的資料分佈不同，記憶體的使用量並沒有隨著路徑的變化呈線性成長。

5. 結論與未來研究

本文提出的路徑查詢處理機制，適用於串流 XML 資料，並利用循序而至的開始、結束標籤及有順序限制的路徑查詢元素，配合計數器、堆疊器與不同的路徑節點給予不同的指標結構，來達到查詢處理之目的。對於單一路徑的查詢處理，給予查詢式上的各個元素一計數器及堆疊器，依據計數器將符合的資料暫存至相對應的堆疊器中並與其父元素的堆疊建立指標。當要取得結果時，僅需遵循指標及堆疊器的特性即可獲得正確結果。對於分支路徑查詢處理，除了依序給予計數器及堆疊器之外，也針對不同的路徑節點給予不同的指標結構，最後根據這些指標結構來重建正確的結果。實驗結果證明，我們的方法無需索引結構且效能上更優於使用狀態機的處理機制。在未來研究方面，我們將提出適用於多個查詢或多個來源的串流資料環境下的查詢處理方法。

6. 誌謝

本研究部分經費由國科會計畫 NSC93-2213-E-415-007 及 NSC94-2213-E-415-004 補助。

7. 參考文獻

[1] S. Al-Khalifa, H. V. Jagadish, N. Koudas, J. M. Patel, D. Srivastava, and Y. Wu, “Structural Joins: A Primitive for Efficient XML Query Pattern Matching,” ICDE, 2002.

[2] M. Altinel and M. J. Franklin, “Efficient Filtering of XML Documents for Selective Dissemination of Information,” VLDB, 2000.

[3] N. Bruno, N. Koudas, and D. Srivastava, “Holistic Twig Join: Optimal XML Pattern Matching,” SIGMOD, 2002.

[4] N. Bruno, L. Gravano, N. Koudas, and D. Srivastava, “Navigation- vs. Index-Based XML Multi-Query Processing,” ICDE, 2003.

[5] C. Y. Chan, P. Felber, M. Garofalakis, and R. Rastogi, “Efficient Filtering of XML Documents with XPath Expressions,” ICDE, 2002.

[6] Y. Chen, S. B. Davidson, and Y. Zheng, “BLAS: An Efficient XPath Processing System,” SIGMOD, 2004.

[7] Y. Chen, S. B. Davidson, and Y. Zheng, “ViteX: A Streaming XPath Processing System,” ICDE, 2005.

[8] Y. Chen, G. A. Mihaila, S. B. Davidson, and S. Padmanabhan, “EXPedite: A System for Encoded XML Processing,” CIKM, 2004.

[9] Y. Chen, G.. A. Mihaila, S. B. Davidson, and S. Padmanabhan, “Efficient Path Query Processing on Encoded XML,” Proc. of Int’l Workshop on High Performance XML Processing, 2004.

[10] S. Chien, Z. Vagena, D. Zhang, V. J. Tsotras, and C. Zaniolo, “Efficient Structural Joins on Indexed XML Documents, ” VLDB, 2002.

[11] Y. Diao, P. Fischer, M. J. Franklin, and R. To, “YFilter: Efficient and Scalable Filtering of XML Documents,” ICDE, 2002.

[12] T. J. Green, G. Miklau, M. Onizuka, and D. Suciu, “Processing XML Streams with Deterministic Automata,” Proc. of the 9th Int’l Conf. on Database Theory, 2003.

[13] A. K. Gupta and D. Suciu, “Stream Processing of XPath Queries with Predicates,” SIGMOD, 2003.

[14] M. C. Ho,” Efficient Evaluation of XPath Queries on Streaming XML Data,” Master Thesis, Dept. of Computer Science and Information Engineering, National Chiayi University, 2005.

[15] H. Jiang, H. Lu, W. Wang, and B. C. Ooi, “XR-Tree: Indexing XML Data for Efficient Structural Joins,” ICDE, 2003.

[16] H. Jiang, W. Wang, H. Lu, and J. X. Yu, “Holistic Twig Joins on Indexed XML Documents,” VLDB, 2003.

[17] R. Kaushik, R. Krishnamurthy, J. F. Naughton, and R. Ramakrishnan, “On the Integration of Structure Indexes and Inverted Lists,” SIGMOD, 2004.

[18] Q. Li and B. Moon, “Indexing and Querying XML Data for Regular Path Expressions,” VLDB, 2001.

[19] M. Onizuka, “Light-Weight XPath Processing of XML Stream with Deterministic Automata,” CIKM, 2003.

[20] F. Peng and S. S. Chawathe, “XPath Queries on Streaming Data,” SIGMOD, 2003.

[21] P. Rao and B. Moon, “PRIX: Indexing And Querying XML Using Prufer Sequences,” ICDE, 2004.

[22] H. Wang, S. Park, W. Fan, and P. S. Yu, “ViST: A Dynamic Index Method for Querying XML Data by Tree Structures,” SIGMOD, 2003.

[23] C. Zhang, J. Naughton, D. DeWitt, Q. Luo, and G. Lohman, “On Supporting Containment Queries in Relational Database Management Systems,” SIGMOD, 2001.

[24] ACM SIGMOD Record: XML Version, http://www.dia.uniroma3.it/Araneus/Sigmod/, 2005.

[25] A. Diaz and D. Lovell, XML Generator, http://www.alphaworks.ibm.com/tech/xmlgenerator, 1999.

[26] NASA, http://xml.gsfc.nasa.gov, 2003.

[27] SAX 2.0, http://www.saxproject.org, 2002.

[28] W3C Recommendation, “XML Path Language (XPath) Version 2.0,” http://www.w3.org/TR/2005/WD-xpath20-20050404, 2005.

[29] XMARK the XML-benchmark project, http://monetdb.cwi.nl/xml/index.html, 2001.