一個支援串流查詢之 XML 文件壓縮技巧

An XML Document Compression Technique Supporting Stream Query

賈坤芳　　周健平　　丁正文
國立中興大學資訊科學與工程系
{kfjea, phd9402, s9456039}@cs.nchu.edu.tw

摘要

串流XML文件的特性在於XML文件樹狀的半結構化特性與串流本身特殊的傳輸型態。使用串流的方式傳送 XML 文件時，會使客戶端在接收的過程中，無法預測後續的文件內容，亦無法重覆讀取串流文件，故許多傳統加速查詢的技術－例如索引，皆不適用於串流 XML 文件之查詢。

本研究提出一個壓縮 XML 文件的技巧，以加速串流 XML 文件之查詢。藉由處理 XML 文件結構中重複的路徑，可縮短文件長度、簡化文件結構，使資料傳輸的時間縮短；並且由於文件結構的簡化，使查詢的複雜度降低，處理速度因而加快。此外，我們設計的查詢演算法僅需針對查詢的相關部分作解壓縮，避免完全解壓縮文件所造成的負擔。

關鍵詞：XML、串流 XML 查詢、壓縮、編碼

一、簡介

近年來，網際網路與無線通訊的使用量大增，行動電話、個人數位助理（PDA）等手持行動設備已經成為許多人生活上用來接收訊息的必備工具。在此情境之下，產生兩個重要的問題：第一是網路上各種系統平台有著不同資料格式，造成資料交換的困難；第二是行動客戶的數量龐大，往往超出伺服器所能負荷。為了解決這兩個問題，XML 資料串流管理系統（Data Stream Management System）如[6][7]應運而生，理由是 XML[4]已經成為網路上資料交換的標準格式，而採用串流系統可以分散伺服器的負擔，讓行動客戶端在有限的記憶體與運算能力的情況下，擷取所需要的資訊。所謂 XML 資料串流是指在 XML 文件上循序瀏覽所取得的一連串資料，如果就 XML 文件本身的結構關係來看，可視為一連串的開始與結束標籤，如：*<Sigmod><issue><articles>…</articles></issue><issue>…</issue></Sigmod>*。

設計 XML 串流查詢系統有兩項主要考量的因素。第一項因素是資料傳輸的時間。因為串流傳輸的限制，使用者必須等待整份 XML 文件傳送結束，才能結束查詢，造成頻寬與客戶行動設備電力的浪費。第二項考量因素是客戶端的查詢效能。查詢的速度若太慢，會來不及處理串流資料，進而影響查詢的正確性。為了加速客戶端查詢的速度，傳統的方法是由伺服器對文件進行編碼[9][10]，輔助客戶端的查詢，但是編碼會增加文件額外的內

容，等於增加資料傳輸的時間，不利於第一項因素。

根據上述兩項設計串流查詢系統所需考量的因素，本研究針對 XML 文件格式以及串流傳輸的特性，設計了資料壓縮與查詢壓縮文的演算法。我們選擇在伺服器端壓縮 XML 文件，然後廣播給每個客戶，由客戶進行壓縮文的查詢處理。我們所提出的查詢演算法支援文件部份解壓縮（partial decompression），讓使用者僅需要將與其查詢相關的 XML 節點做部份解壓縮，即可大幅減輕查詢的負擔；並又為了增進查詢的效率，我們設計了適合 XML 串流查詢的編碼方式。經由壓縮的技巧及支援部分解壓縮的查詢演算法，可縮短 XML 串流傳輸時間，提高的查詢速度。本研究所提出的方法特色與技巧如下：

1. 除了 XML 文件本文（text）之外，特別針對 XML 文件的結構（structure）進行壓縮。我們先找出串流 XML 文件中重複的標籤路徑，然後用一個單一節點來取代，達到資料壓縮的目的。

2. 設計適合 XML 串流查詢的編碼方式（labeling scheme），可加速查詢，且不會對串流文件的傳輸造成過多的負擔。

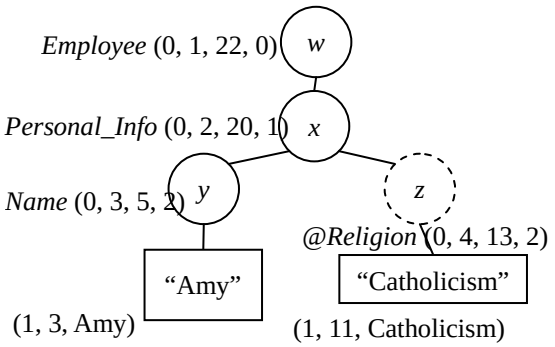
3. 提出兩階段查詢方法，能夠將不需要處理的 XML 節點略過，降低客戶端查詢的時間成本。

二、相關研究

本節回顧目前著名的 XML 資料串流查詢系統 Expedite[6][7]，以及可查詢式 XML 壓縮器 XGRIND[12]。

2.1 XML 串流查詢系統

EXPedite[6][7]系統為一架構在無線網路環境下的主從式 XML 查詢系統，支援路徑查詢（path query）及分支查詢（twig query）。EXPedite 伺服器將 XML 文件內的元素與內容節點（text content）分別進行編碼，元素的編碼格式為(*Flag, T, Size, Depth*)，其中 *Flag*＝0 代表此編碼格式屬於元素或屬性節點，以與內容節點作區別。*T* 是可對應到唯一元素名稱的數字，*Size* 表示此節點的子樹大小，*Depth* 表示節點所在的樹的階層數（根節點的階層數為 0）；內容節點的編碼格式為(*Flag, Length, Text*)，其中 *Flag*＝1，*Length* 是指內容字串的大小，*Text* 為內容字串。



圖一　EXPedite 編碼範例

圖一為 EXPedite 對一份 XML 文件的編碼範例，其中實線圓形代表元素節點，虛線圓形代表屬性節點，方塊代表文字內容節點。以節點 *x* 為例，其編碼為(0, 2, 20, 1)，第一個欄位 0 表示 *Flag*；第二個欄位 2 表示利用字典編碼法（dictionary coding method）將節點 *x* 的標籤 *Personal_Info* 編碼成唯一的識別碼 2；第三個欄位 20 表示 *Size*；第四個欄位 1 表示 *Depth*。以節點 *y* 內容編碼(1, 3, Amy)為例，第一個欄位表示 *Flag*，第二個欄位 3 表示內容節點的大小，第三個欄位值 Amy 為內容字串。

伺服器將編碼過後的 XML 文件廣播給使用者，XML 文件串流中的元素編碼

(*Flag, T, Size, Depth*)代表文件的結構，被客戶端接收到之後，逐一被轉換成區間編碼（range-based labeling scheme）[9][10]格式：(*Flag, T, Start, End, Level*)，以利查詢。其中 *Start*＝客戶端目前所接收到的總位元組數；*End* ＝ *Start* ＋ *Size*；*Level* ＝ *Depth*。EXPedite 查詢方法是利用對應於查詢式標籤所分別建立的堆疊(稱為"查詢堆疊")，處理與輸出查詢結果，演算法類似於 Holistic twig joins[5]。

2.2 XML 文件壓縮

XGRIND[12] 屬於一種半適應（semi-adaptive）壓縮方法，須先讀取文件一遍，以得到統計資料後，再一次讀取文件來產生壓縮文。XGRIND 將 XML 文件分成三種不同類型的節點：標籤、列舉型屬性、非列舉型屬性或內容，以進行壓縮。XGRIND 壓縮粒度（granularity）為 XML 節點，保留了節點之間的結構關係，並能利用一般路徑查詢的語法對查詢所需要的部分進行解壓縮。

第一種是標籤的壓縮，將每一個 XML 開始標籤（start tag）以"T"加上一個唯一的元素識別碼來表示，例如：T0、T1 分別代表兩種不同的開始標籤。結束標籤（end tag）則以"/"來取代。第二種是列舉型（enumerated-type）屬性壓縮，例如：公司裡的部門、國家的省分，屬於固定的屬性集合，則採用字典編碼法將固定出現的屬性集合分別編成唯一的數字，來達到壓縮的效果。第三種是非列舉型屬性或內容，採用霍夫曼編碼（Huffman encoding）[8]的方式來壓縮。

圖二為對圖一的範例進行壓縮的結果。每個標籤節點都用唯一的代號取代，例如 T0 代表 *Employee*，T1 代表 *Personal_Info*；屬性名稱之做法類似，如 A0 代表*@Religion*。內容節點採用霍夫曼編碼，以 *huff*(Amy)表示之；列舉屬性採用字典編碼法，如 "Catholicism" 以 *enum*(Catholicism)函式表示。

```
T0
  T1   A0   enum(Catholicism)
  T2   huff(Amy) /
  /
/
```

圖二　XGRIND 壓縮範例

XML 文件壓縮技術依照壓縮的粒度，可區分為兩類[11]。第一類是壓縮整份 XML 文件，不能經由部分解壓縮或其他方式來進行查詢，必須整份文件解壓縮之後才能夠進行查詢，例如傳統的 RAR 或 ZIP 的壓縮方式。另一類是壓縮 XML 文件中的節點，如圖二，經過 XGRIND 壓縮之後，我們仍可輕易地辨識壓縮文中的節點結構關係，使客戶可以直接在壓縮文上進行查詢，或僅需部份文件的解壓縮。一般而言，第一類技術的壓縮率比較高，但是在串流系統中，由於在客戶端無法保留全部的 XML 文件，故亦無法接收整份壓縮文件後再做解壓縮，所以並不適用。本論文著重於第二類可查詢壓縮文的壓縮方法之研究，以支援串流查詢系統。

三、問題與方法

3.1 環境及問題定義

我們的系統環境是在主從式架構之下，伺服器端利用單一頻道，以串流的形式將 XML 文件廣播出去；而客戶端所持的是手機或其他行動裝置，其運算能力有限。我們的系統支援 XPath[3]查詢語言的核心，包括子代軸線（child axis）與後代軸線（descendant axis）查詢。

在串流系統中無法像一般環境，可以利用索引等技巧來輔助查詢。倘若客戶端的查詢演算法效率不佳，就無法及時處理接收到的大量連續之串流資料，造成客戶端須使用大量的緩衝區，勢必消耗行動裝置有限的記憶體以及電力。所以如何縮短整體資料傳輸的時間，以及提高客戶端串流資料查詢演算法之效能，就是我們要探討的研究問題。

3.2 XSC 壓縮技巧

我們提出一個 XSC 壓縮方法以支援 XML 文件串流查詢，可分成三個部份來說明。首先在 XML 文件中，將多次重複出現的結構路徑找出，稱為頻繁樣式路徑（FPP），然後利用 FPP 去取代重複的路徑，達到壓縮文件的目的；其次，利用編碼與其他壓縮技巧，進一步增加壓縮率及提升查詢的效率；最後是針對 XSC 壓縮文件所設計的查詢演算法。

3.3 尋找頻繁樣式路徑

資料串流的環境下，大量 XML 文件從資料伺服器依序不斷被傳送給客戶端。對客戶端而言，所接收到的 XML 文件資料，可視為一連串的標籤（tag）序列。圖三為一份範例 XML 文件，其中 A、B、C、D、E、F 為標籤名稱。當資料在單一頻道上循序傳送出去時，按深度優先搜尋（depth-first search）的順序傳送，如 <A><B><C><D>Asia</D></C><E>NCS2007</E></B><B><C><D>Dec.20</D><F>Taichung</F></C></B></A>。我們可從資料串流觀察到：(1) 所有 XML 資料串流的頻繁路徑樣式，皆為 XML 文件某個具有分支的節點與其第一子節點（first child node）所遞迴組成的路徑。例如在前述的資料串流中，路徑樣式<B><C><D>重複出現了兩次；(2) 資料串流中重複敘述 XML 文件結構的頻繁路徑都會被內容節點（text）所切斷，無法形成很長的頻繁路徑。於是，我們可搜尋到的最長頻繁路徑樣式，將不會超過 XML 文件的最大深度。

```
<A>
 <B>
  <C>
    <D>Asia</D>
  </C>
  <E>NCS2007</E>
 </B>
 <B>
  <C>
    <D>Dec.20</D>
    <F>Taichung</F>
  </C>
 </B>
 </A>
```

圖三　XML 文件範例

基於上述的觀察，為了達到壓縮 XML 文件中重複結構的目的，我們將搜尋頻繁子樹樣式這個複雜的問題，簡化成搜尋有限長度的頻繁路徑樣式。根據文獻[2]，一般 XML 文件的深度平均在 4，且絕大部分的文件的深度都不超過 8，所以搜尋有限長度的頻繁路徑樣式，其計算量不高。

搜尋頻繁路徑樣式的步驟簡述如下：首先利用 SAX（simple API for XML）分析整份 XML 文件，將文件中有分支的節點的第一個分支節點（即第一個子節點）以遞迴的方式取出，形成有序的路徑，即路徑樣式（path pattern），各個路徑樣式以文件中的內容節點或標籤結束符號作為分隔。接著拆解路徑獲得子序列路徑，例如：A~B~C 代表一條會經過節點標籤依序為 A、B、C 的路徑，A~B~C 路徑須分解出 A~B 與 B~C 子序列路徑（沒有 A~C，因為路徑必須由連續的節點所組成）。以動態

配置方式儲存這些路徑並累計它們出現的頻率，最後利用預先設定的門檻值（min_supp）來判斷是否成為頻繁路徑樣式（FPP）。尋找 FPP 演算法步驟如圖四。

Algorithm XSC_Find_FPP
輸入：XML 文件 *D*，參數 *Y*：FPP 最小門檻值
輸出：List *L*：存放所有的 FPP 之串列
初始設定：暫存串列 *TL* 為 Null
步驟 1：用SAX分析文件*D*，依序產生的每一個事件$event_i$。
步驟 2：如果$event_i$是開始標籤，則將$event_i$加入暫存串列*TL*中，回到步驟 1；若$event_i$是內容節點或結束標籤，則繼續步驟 3。
步驟 3：*TL*中所有開始標籤$event_1 \ldots event_{TL.length}$形成一個序列路徑*P*，呼叫 Update_L(*P*)；將*P*分解出所有長度大於 1 的子序列路徑P_{sub}，呼叫 Update_L(P_{sub})。清空*TL*。
步驟 4：重複步驟 1 到 3，直到文件 *D* 結束。
步驟 5：將低於門檻值 *Y* 的序列路徑從 *L* 中刪除後，輸出 *L*。

Procedure Update_L(*f*)
步驟：若發現序列路徑 *f* 已存在於串列 *L* 中，則累加 *f* 的出現頻率；否則，新增 *f* 於 *L* 中，並將 *f* 的出現頻率設為 1。

圖四　XSC－尋找 FPP 演算法

每一個FPP都有唯一的識別碼，用來置換在XML文件中相同的路徑，被置換的路徑則放置一個FPP節點來表示，達到壓縮文件結構的效果。例如：令FPP_1＝<B><C><D>，則圖三XML範例文件的資料串流會壓縮為：<A><FPP_1>Asia</FPP_1><E>NCS2007</E><FPP_1>Dec.20</FPP_1><F>Taichung</F></A>。我們將所有的FPP資訊記錄於FPP表格中，包括FPP的識別碼以及其所取代的序列路徑。

3.4　XSC 壓縮方法

利用串流資料必須循序讀取的特性，我們捨棄傳統區間編碼需給定每個節點一組編碼(*Start, End, Level*)來表示文件結構的方式，而僅保留 *Level*，大幅降低編碼所額外需要的儲存空間。藉由持續觀測各節點 *Level* 的變化，推算出各個節點的子樹範圍，藉此判定文件的結構關係。如圖一的範例，假設客戶端循序讀取到節點 *x* 時，其階層為 1，則之前已讀取到的節點，若其階層小於或等於 1，必不屬於節點 *x* 的子樹範圍內，例如根節點 *w*（階層為 0）；而後陸續接收到節點如 *y*、*z*，階層皆大於 1，且自節點 *x* 以來，尚未出現其他階層小於或等於 1 的節點，則可斷定節點 *y*、*z* 皆屬於節點 *x* 的子樹範圍內。另外，我們也刪去文件中每個標籤節點的結束標籤，以節省更多的空間，且不影響 XSC 方法用 *Level* 來判斷文件結構的正確性。

除了結構上的編碼法之外，XSC 尚運用字典編碼法進一步去壓縮文件。我們將每個標籤都編一個唯一可識別的數字，例如 *Employee*、*Personal_Info* 分別對應到數字 1、2，來節省文件的空間。同樣地，節點的屬性名稱也利用字典編碼法去壓縮。至於內容節點的處理則與 XGRIND[12]相同，採用霍夫曼編碼（Huffman encoding）[8]來進行壓縮。

Algorithm XSC_Compression
//產生壓縮 XML 文件
輸入：XML 文件 *D*，FPP 表格 *FPP_table*
輸出：壓縮之 XML 文件 *CXML*
初始設定：暫存串列 *TL* 為 Null
步驟 1：用SAX分析文件*D*，依序產生的每一個事件$event_i$。
步驟 2：如果$event_i$是開始標籤，則將$event_i$加入*TL*中，回到步驟 1；若是內容節點或結束標籤，則繼續步驟 3。
步驟 3：*TL*中的所有的事件$event_1 \ldots event_{TL.length-1}$形成一個序列路徑P，呼叫Output(P)；若$event_{TL.length}$為內容節點，則呼叫OutputHuffman($event_{TL.length}$)。清空*TL*。
步驟 4：重複步驟 1 到 3，直到文件 *D* 結束。
步驟 5：輸出 *CXML*。

Procedure Output(*f*)
//將壓縮節點輸出至壓縮文
輸入：*f*：由多個標籤組成的序列路徑
步驟：若序列路徑 *f* 存在於 *FPP_table*，則將 *f* 所對應的 *FPP_id* 與 *f* 的第一個標籤節點的 *Level* 寫入檔案 *CXML*；若 *f* 不存在於 *FPP_table*，則將 *f*（包括 *f* 內每個標籤的 *Level*）寫入檔案 *CXML*。

圖五　XSC－壓縮演算法

XSC 壓縮（包含編碼）演算法如圖五。當我們對 XML 文件壓縮完畢，壓縮後的 XML 文件會含有三種類型節點的編

碼：分別是 FPP 節點、一般節點與內容節點。各種節點的資料格式如：FPP 節點為(*Flag, FPP_id, Level*)、一般節點為(*Flag, Node_id, Level*)、內容節點為(*Flag, Length, Text*)。*Flag* 用來區分三種不同格式的節點，*FPP_id* 或 *Node_id* 分別表示節點的識別碼，*Length* 與 *Text* 的意義與 EXPedite [6][7]編碼相同，差別在於 *Text* 是以霍夫曼編碼壓縮（圖五之 OutputHuffman 函式）。內容節點的 *Flag*＝0；若 *Flag*＝1，則需額外識別 *FPP_id* 或 *Node_id*，進一步判斷是屬於 FPP 節點或一般節點。

3.5 XSC 壓縮文件之查詢演算法

支援XSC壓縮文件之查詢包括XPath子代軸線與後代軸線查詢[3]。XSC查詢方法分為兩個階段：第一階段根據使用者查詢式預先處理FPP表格內所有的FPP，演算法步驟如圖六(a)。首先將FPP表格傳送給客戶端，客戶端根據查詢式需要的特定標籤節點，從FPP表格中找出符合查詢式的FPP，並將這些FPP節點及相關資訊存入客戶端的查詢表格*Q_Table*中（步驟 2）。查詢表格紀錄FPP節點取代的路徑上的標籤與查詢式相符標籤的對應位置。舉例來說，若查詢式為//A/B/D，而 FPP_1 = <B><C><D>，則查詢表格中會新增一筆紀錄(FPP_1, 0, 1, 3)，分別表示FPP識別碼與查詢式中三個標籤的對應位置。第二個欄位 0 代表查詢式的A標籤並未出現於FPP_1；第三個欄位 1 代表查詢式的B標籤出現於FPP_1第 1 個位置；第四個欄位 3 代表查詢式的D標籤出現於FPP_1第 3 個位置。

第二階段是在XSC壓縮文件上做查詢，XSC查詢演算法主要的概念來自於Holistic twig joins[5]，並融入我們設計的FPP技巧，其演算法步驟如圖六(b)所示。以下舉例說明演算法相關的術語與符號。XPath查詢式可視為一個查詢樹，各節點都有對應的查詢堆疊。假設查詢式為//A[/B]/C/D，則對應的查詢堆疊分別為 S_A、S_B、S_C、S_D，其中S_A稱為“根查詢堆疊”，S_B、S_D稱為“葉查詢堆疊”，而S_D稱為“結果堆疊”，是查詢式所需要傳回的結果。查詢處理過程中，暫存於查詢堆疊裡的元素以q代表，如q_A、q_B、q_C、q_D。

Algorithm：XSC_Query_Phase1
輸入：FPP表格 *FT*、查詢*Q*及*Q*所含的標籤 $t_1t_2…t_n$
輸出：查詢表格 *Q_Table*
步驟 1：逐一取出*FT*中的FPP，稱為FPP_i。
步驟 2：若FPP_i有節點與$t_1t_2…t_n$的標籤相同，則將FPP_i加入查詢表格*Q_Table*，並標記FPP_i與查詢式相符標籤的對應位置。

(a) 第一階段

Algorithm XSC_Query_Phase2
輸入：查詢*Q*，查詢表格*Q_Table*，壓縮XML文件*D*每個節點產生的事件：$event_1…event_j$
輸出：符合查詢式之節點內容
初始設定：事件順序*i*=1，根據查詢*Q*建立對應的查詢堆疊$S_1…S_k$，及對應的暫存結果緩衝區$C_1…C_k$
步驟 1：讀取$event_i$，根據$event_i.Level$判斷查詢堆疊$S_1…S_k$可被丟棄（pop）的節點，將其丟棄並輸出已成立之查詢結果。
步驟 2.1：如果$event_i$為一般節點，令*x*=Condition(*Q*, $event_i$)，若*x*＞0 則將$event_i$放到對應的查詢堆疊中。跳到步驟 3。
步驟 2.2：如果$event_i$是FPP節點且*Q_Table*中有符合FPP的紀錄，則逐項分析FPP路徑上每個事件標籤$event_z$，令 *x*=Condition(*Q*, $event_z$)，若*x*＞0 則將$event_z$放到對應的查詢堆疊S_x中。跳到步驟 3。
步驟 2.3：如果$event_i$是內容節點且為結果堆疊S_k內節點的內容，則將$event_i$放入C_k。跳到步驟 3。
步驟 3：令 *i*＝*i*+1；若 *i*＞*j* 則程式結束，否則回到步驟 1。

Function Condition(*Q*, $event_i$)
輸入：查詢*Q*與查詢堆疊：$S_1…S_k$，事件$event_i$
輸出：數字 *x*，0≦*x*≦*k*
步驟：判斷節點是否符合查詢式。若是，傳回符合的查詢堆疊編號 *x*；若否，傳回 0。

(b) 第二階段

圖六　XSC－查詢演算法

查詢處理 XSC 壓縮文的節點分為三種情形：一般節點、FPP 節點以及內容節點。首先如圖六(b)步驟 1，根據節點的階層（*Level*）判斷，刪除查詢堆疊 *S* 中可被

丟棄（pop）的節點 q，以節省暫存空間。接著，若節點屬一般節點，且符合查詢式之堆疊的結構及標籤名稱（由步驟 2.1 之 Condition 函式判斷），則放入（push）查詢堆疊中。若屬 FPP 節點，則搜尋客戶端的查詢表格是否存在此節點，若是，則分解 FPP 節點以進行 Condition 函式判斷；否則可略過（步驟 2.2）。若節點屬內容節點，且為結果堆疊內節點的內容（步驟 2.3），則需要暫存起來。圖六所有副程式的詳細演算法請參考論文[1]。

舉圖三的XML文件壓縮後為例，其中有FPP節點FPP_1＝<B><C><D>。若查詢式為 //A/B/D，在第一階段查詢時，客戶端先接收到FPP表格並搜尋發現FPP_1有符合查詢式//A/B/D，故將(FPP_1, 0, 1, 3)存入查詢表格。第二階段開始進行查詢，演算法首先依據查詢式建立三個查詢堆疊S_A、S_B、S_D，並陸續接收XML文件節點。第一個接收到的是A節點，符合查詢式，故將A節點放入查詢堆疊S_A中；其後是FPP_1，位於查詢表格中，故可查表得知FPP_1的第 1 及第 3 個標籤（B、D）符合查詢第 2 及第 3 個標籤，於是解開FPP_1獲得B、D節點並放入查詢堆疊S_B、S_D。此時查詢堆疊S_A、S_B、S_D已經全部都包含節點，代表查詢式已完全被滿足，故輸出結果S_D，並隨即將之丟棄。而後E節點與B節點陸續出現，使S_B、S_D堆疊內的節點也被丟棄。如此不斷地接收串流資料並立即做查詢處理，直到查詢結束為止。

四、實驗

我們進行三個實驗，模擬 XSC 方法在壓縮與查詢上的效能，並且對高壓縮率所帶來的額外負擔也做了評比。模擬實驗的環境為個人電腦（中央處理器 Intel Core 2 T5600 1.83GHz，1GB 記憶體），Windows XP 作業系統，程式用 Microsoft C# .Net 撰寫。實驗文件特徵如表一所示，XMark 是用網站 http://monetdb.cwi.nl/xml/ 提供的程式與 DTD 綱要所產生的人造 XML 文件，其餘為真實文件，取自公開於網際網路上的專案，如 GSFC/NASA XML project、Penn Treebank project 等。實驗 1 比較 XSC、XGRIND 及 EXPedite 三種演算法對文件的壓縮率。實驗 2 量測上述三種演算法查詢壓縮文件的時間。實驗 3 量測三種演算法壓縮文件所需要的壓縮與編碼的時間。

表一　XML 實驗測試資料

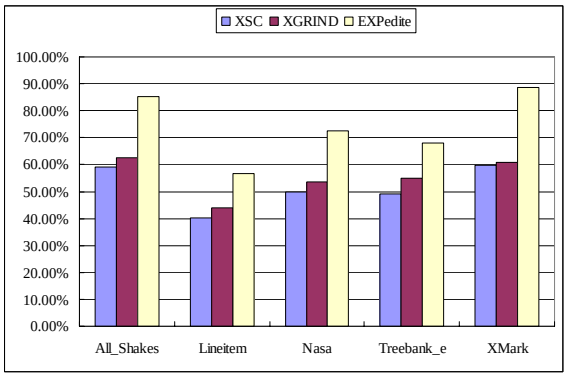
	節點數	文件大小	屬性數	最大階層	平均階層	結構對內容的比率
All_Shakes	180K	7.7MB	0	6	4	0.25: 1
Nasa	477K	24MB	56317	8	5.58	0.56: 1
Lineitem	1023K	31MB	1	3	2.94	3.7: 1
Treebank_e	2438K	84MB	1	36	7.87	0.451: 1
XMark	1666K	115MB	381878	11	4	0.32: 1

4.1 實驗 1：壓縮率

實驗 1 進行整份文件與單就結構部份的壓縮率比較。進行模擬實驗時，以各份文件總節點數的 1%來當作頻繁路徑模式的門檻值，故出現次數超過該文件節點數百分之一的序列路徑，即視為 FPP。實驗程序是將表一之五份 XML 文件檔案讀取之後，利用我們實做出的三種不同的方法去壓縮，輸出壓縮後的文件檔案。圖七為整份文件的壓縮率比較圖，圖八為文件結構部份的壓縮率比較圖。圖中的壓縮率計

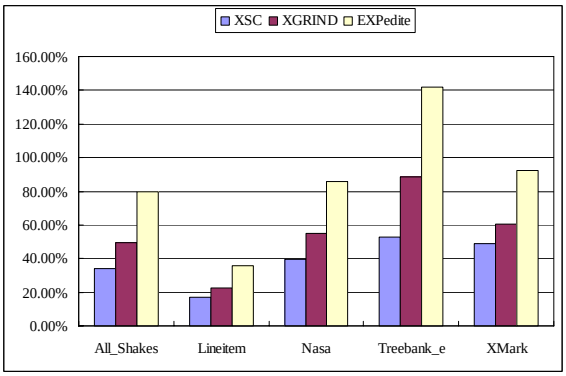
算公式為＝（壓縮後文件的大小）/（原始文件大小）x100%，所以壓縮率愈低表示壓縮的效果愈好。

圖七的實驗結果顯示，XSC 與 XGRIND 都可將文件壓縮至 65%以下，壓縮率優於 EXPedite 約 13%~28%的差異。EXPedite 雖對文件加入額外的編碼資訊，但編碼後文件仍然比原始文件小，主要原因是它利用字典編碼法，將標籤節點取代為唯一的識別碼，所省下的空間多於額外的編碼資訊，故達到壓縮的效果。



圖七　XML 文件壓縮率比較圖

以圖七整份文件壓縮率而言，XSC 優於 XGRIND 平均約 3.5%的差異。由於 XSC 壓縮技巧主要是針對結構上重複出現的路徑，一般而言，結構重複性愈高，出現 FPP 路徑愈長，則預期 XSC 效果會愈好。以 Treebank_e 文件為例，最大的階層數為 36，XSC 能搜尋出更多、更長的 FPP 路徑，故有較好的壓縮率（49.26%），優於 XGRIND 方法（54.99%）；反之，以 XMark 文件為例，其結構與內容的比例為 0.32：1，階層數平均為 4，其結構較 Treebank_e 單純，可預期壓縮率不佳；實驗結果顯示，XSC 對 XMark 的壓縮率僅優於 XGRIND 1.21%的差異。然而 XSC 壓縮率高低非單一因素可決定，以 Lineitem 文件為例，結構與內容比率為五份實驗文件中最高者（3.7：1），但壓縮率卻不如 Treebank_e，原因在於 Lineitem 的標籤結構十分簡單，XSC 只能找到一組 FPP。總體分析發現，當文件的結構複雜、節點的標籤長，以及結構與內容的比率高的文件，XSC 的壓縮率會比較高。



圖八　XML 文件結構壓縮率比較圖

若以圖八僅比較結構部份的壓縮率而言，XSC 表現最好仍然是在 Treebank_e 文件，壓縮率 52.87%，優於 XGRIND 的 88.79%，差異有 35.92%之多；Lineitem 壓縮率 17.11%，優於 XGRIND 的 22.37%，差異最小僅 5.26%。從實驗數據可知，XSC 針對結構的壓縮表現良好，優於 XGRIND 差異平均有 16.61%。

4.2 實驗 2：查詢時間

實驗 2 比較三種方法在串流查詢上所花費的時間。我們分別採用三份 XML 文件：Nasa、Lineitem、Treebank_e 進行實驗，利用個人電腦模擬串流環境，循序讀取壓縮後的 XML 文件，然後將查詢所得的結果直接輸出在螢幕上，並重複執行實驗十次以獲得平均查詢時間。三份實驗文件的大小、結構對內容的比率以及結構複雜程度都不相同，藉此驗證 XSC 方法運用在各類文件上的優缺點。

由於 XGRIND[12]論文中並沒有詳細描寫所使用的查詢演算法，所以我們根據

其壓縮文件格式，自行編寫查詢程式，稱為 XGRIND'，以利進行實驗比較。實驗 2 所使用的查詢集如表二，混合了長度不等的路徑查詢與分支查詢；實驗結果如圖九(a)、(b)、(c)所示。

表二　實驗 2 查詢集

Nasa

查詢一：//source/other/title
查詢二：//field/name
查詢三：//field[/name]//definition
查詢四：//history/ingest[/affiliation]/creator/lastName

Lineitem

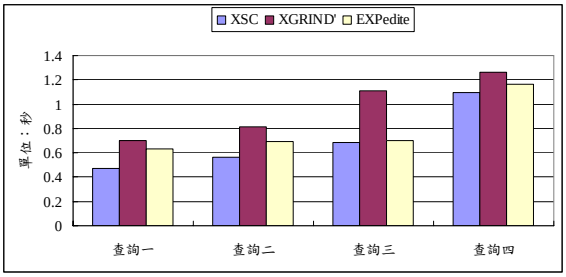
查詢一：//T/L_ORDERKEY
查詢二：//T/L_PARTKEY
查詢三：//T[/L_PARTKEY]/L_SUPPKEY

Treebank_e

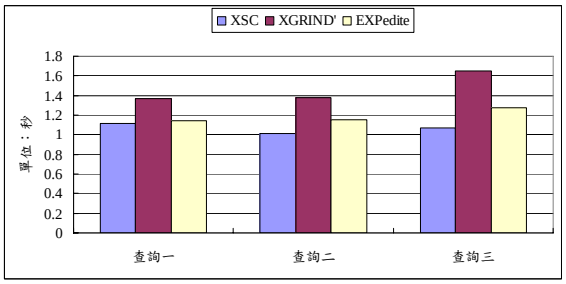
查詢一：//NP/JJ
查詢二：//VBP/VP/VBN
查詢三：//NP[//NN]/JJ
查詢四：//NP[//VBN//VB]/NN/PP/IN

圖九(a)為 Nasa 文件的查詢時間比較圖，EXPedite 由於有區間編碼的輔助，所以查詢時間比 XGRIND'短，而 XSC 不僅利用編碼的 *Level*（階層），更採用兩階段查詢，略過不需要的 FPP 節點，所以實驗結果平均最佳。在查詢三的數據中，XSC 的查詢時間最接近 EXPedite，原因是查詢三所獲得的內容節點較多，XSC 在進行霍夫曼編碼的解壓縮時間增加，故相對於查詢一、二、四而言，能改善的幅度較少，僅優於 EXPedite 0.019 秒，約 2.7%。

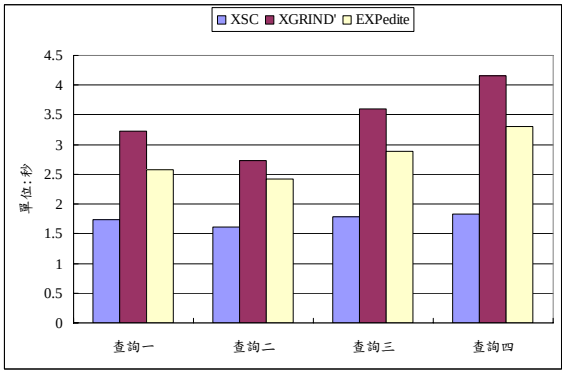
圖九(b)為 Lineitem 文件的查詢時間比較圖。由於 Lineitem 文件的結構十分簡單，文件最大階層只有 3，故我們僅設計三個查詢（兩個路徑查詢，一個分支查詢）來進行模擬實驗。根據實驗結果，XSC 具有最短的查詢時間。Lineitem 標籤結構簡單，並沒有許多 FPP 節點可以取代，XSC 於查詢一、二中，改進 EXPedite 查詢效能約 3%、12%；對於分支查詢三，XSC 效能最好，改進 EXPedite 效能約 16%。



圖九(a)　Nasa 查詢時間比較圖



圖九(b)　Lineitem 查詢時間比較圖

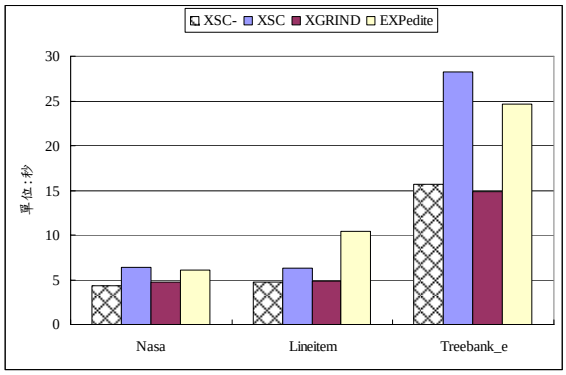


圖九(c)　Treebank_e 查詢時間比較圖

圖九(c)為 Treebank_e 文件的查詢時間比較圖，顯示 XSC 查詢效能仍然是優於 EXPedite，而 EXPedite 優於 XGRIND'。並且改進效能明顯比前面兩份文件 Nasa、Lineitem 要高，因為 Treebank_e 文件結構複雜，XSC 能夠用來壓縮的 FPP 節點也較多，故在兩階段查詢中，可預先於第一階段摒除不合查詢的 FPP 節點，但 EXPedite 及 XGRIND'並沒有這種預先過濾的動作。Treebank_e 模擬實驗共四個查詢式，XSC 平均查詢時間為 1.7 秒，EXPedite 2.7 秒，XGRIND' 3.4 秒；XSC 改進 EXPedite 查詢效能平均達 37%。

4.3 實驗 3：壓縮與編碼時間

實驗 3 比較三種方法在壓縮文件上所花費的時間成本。如同實驗 2，我們採用三份文件：Nasa、Lineitem、Treebank_e 進行效能比較。圖十顯示 XSC-、XSC、XGRIND 及 EXPedite 壓縮文件所需要的時間，XSC-代表 XSC 整體壓縮時間扣掉搜尋 FPP 的時間，而 XSC 代表整體壓縮時間。



圖十　壓縮時間比較圖

就壓縮文件的速度而言，XGRIND 與 XSC、EXPedite 相比，在三份文件上的壓縮時間都是最短的，因為 XSC、EXPedite 都加入了編碼等輔助查詢的機制，故使整體壓縮速度變慢。另外，EXPedite 在 Nasa 與 Lineitem 文件上壓縮時間低於 XSC，但在 Lineitem 文件高於 XSC；原因是 XSC 方法必須處理 FPP，增加了壓縮的時間，而由於 Lineitem 文件簡單，FPP 數量少，故處理 FPP 佔整體壓縮時間比例小，於是 XSC 方法可優於 EXPedite。若將搜尋 FPP 的時間去除掉，即用 XSC-與 EXPedite 相比，則發現，XSC-在壓縮文件的時間上皆優於 EXPedite，且與 XGRIND 不相上下，原因是 XSC 採用了更簡單的編碼，大幅縮短了編碼所需額外的時間所致。

在串流系統中，編碼及壓縮的時間只在伺服器中執行一次，除非文件時常更新，否則不需重複執行。故在文件沒有更新的情況下，我們壓縮及編碼的時間，可算是一次性的成本。當 XML 串流資料被大量且多次傳送至客戶端，壓縮成本相對於查詢所節省下來的時間而言，更顯得微不足道。另一方面，若文件內容有更新，而結構不變的情形之下，XSC 只需對更改的內容重新做霍夫曼編碼，不需重新搜尋 FPP，亦不須重新編碼，可節省約 25~44% 的壓縮時間（以實驗 3 為例）。

五、結論與未來工作

本研究主要貢獻在於，提出一個適用於主從式串流環境的 XML 文件壓縮演算法，以及特殊的兩階段查詢方法，提升查詢串流壓縮文件的效能。壓縮演算法保留原始 XML 文件特性，可供客戶端直接在壓縮的文件上作查詢，不用暫存整份壓縮文件，節省了大量的記憶體需求；兩階段查詢方法可以讓客戶端預先在第一階段過濾掉不可能為查詢結果的文件片段，避免因解壓縮非查詢文件片段所增加的計算成本，減少比對符合查詢式的節點數，進而提高 XML 串流查詢的速度。實驗數據顯示，雖然我們提出的 XSC 方法的壓縮時間略高於實驗比較的對象 EXPedite 和 XGRIND，但是在壓縮率與查詢的時間方面，皆有較好的表現。

在未來工作方面，我們認為在不影響查詢效能的原則下，可嘗試利用文件綱要（如 DTD 或 XML Schema）去獲得語義方面的資訊，達到更高的壓縮率。另外，目前 XSC 方法適用於單一文件，未來將朝向相似綱要之多文件壓縮演算法進行研究。

致謝

本論文由國科會研究經費補助，計畫編號為 NSC-95-2221-E-005-048，特此致謝。

六、參考文獻

[1] 丁正文，一個支援串流查詢之 XML 文件壓縮技巧，國立中興大學資訊科學系，碩士論文 (指導教授：賈坤芳)，2007 年。

[2] D. Barbosa, L. Mignet, and P. Veltri, “Studying the XML Web: Gathering Statistics from an XML Sample,” World Wide Web, Volume 8, Number 4, pp.413-438, 2005.

[3] A. Berglund *et al.*, XML Path Language (XPath) 2.0, W3C Recommendation, Available at http://www.w3.org/TR/xpath20/, 2007.

[4] T. Bray *et al.*, Extensible Markup Language (XML) 1.1, W3C Recommendation, Available at http://www.w3.org/TR/xml11/, 2006.

[5] N. Bruno, N. Koudas, and D. Srivastava, “Holistic Twig Joins: Optimal XML Pattern Matching,” Proceedings of ACM SIGMOD Conference, pp.310-321, 2002.

[6] Y. Chen, G. A. Mihaila, S. B. Davidson, and S. Padmanabhan, “EXPedite: a System for Encoded XML Processing,” Proceedings of the 13th ACM International Conference on Information and Knowledge Management, pp.108-117, 2004.

[7] Y. Chen, G. A. Mihaila, S. B. Davidson, and S. Padmanabhan, “Efficient Path Query Processing on Encoded XML,” Proceedings of International Workshop on High Performance XML Processing, 2004.

[8] D. A. Huffman, “A Method for the Construction of Minimum-Redundancy Codes,” Proceedings of the Institute of Radio Engineers, Volume 40, pp.1098-1101, 1952.

[9] H. Jagadish *et al.*, “TIMBER: A Native XML Database,” Very Large Data Bases, Volume 11, Issue 4, pp.274-291, 2002.

[10] Q. Li and B. Moon, “Indexing and Querying XML Data for Regular Path Expressions,” Proceedings of the 27th International Conference on Very Large Data Bases, pp.361-370, 2001.

[11] W. Ng, L.W. Yeung, and J. Cheng, “Comparative Analysis of XML Compression Technologies,” World Wide Web Journal, Volume 9, Issue 1, pp.5-33, 2006.

[12] P. M. Tolani and J. R. Haritsa, “XGRIND: A Query-friendly XML Compressor,” Proceedings of 18th International Conference on Database Engineering, pp.225-234, 2002.