國立臺灣海洋大學

資訊科學系

碩士學位論文

針對多份 XML 文件之路徑查詢研究

Supporting Query Processing Over Multiple XML Documents

指導教授：張雅惠 博士

研究生：吳俊頡 撰

中華民國 九十一 年 六 月

第一章　緒論

1.1 背景與研究動機

近年來全球資訊網（WWW）已經可以說是全世界資訊分享的主要方式，因為它在無遠弗屆的廣域網路環境中，提供一種便利且簡單的方式去存取資料，所以相當多的企業已經將其產品廣告或可分享的資料放在網際網路上。目前 WWW 上所接受的資料型態是符合 HTML 格式的文件，但是 HTML 主要被設計為顯示資料之用，以便將文件內容呈現在使用者面前，而不是表示資訊的內容及它的結構，其缺乏對資訊意涵的描述，所以不利於自動化的資訊傳遞與交流。

針對於此，所謂的可延伸式標註語言（Extensible Markup Language），簡稱 XML，已成為最近 Web 上相當受到重視的格式。XML 是由 W3C 制定的一個有關於描述資訊的 Mata Language（設計語言的語言），其 1.0 版於 1998 年 2 月正式推出。其目的為定義一個描述資料之標準，允許使用者可以自由定義標籤，來描述資料並將其結構化（所謂的半結構化），另外也能夠將資料與使用者介面分離，而提高更大的使用彈性。

另外，XML 是一個公訂的、開放的（人人都可開發，不會被部份公司把持）、以文字為基礎的標籤語言，而 XML 文件架構簡單易讀，容易在異質系統之間傳遞交流，所以 XML 文件已被大家期望在不久的將來，成為不同組織之間在 Web 上資料共享的主要依據。目前有許多組織，包含銀行公會和高科技電子公會，正嘗試利用 XML 訂定符合自己需求的規則，作為資料交換的標準。而在電腦廠商中，康柏電腦主導的 [台威計劃]，設計 RosettaNet 提供全球供應鏈之協同資訊運作架構，即以 XML 來定義康柏與在台供應商之協同商務運作模式。供應鏈中的上下游廠商，只要透過雙方同意的 XML 標準經由網際網路來傳輸資料，就可以快速的建立交易關係或分享訊息。

隨著 XML 資料日益增多，其查詢處理將是一個重要的課題。又由於 XML 文件的格式必須符合 DTD（document type definition）的定義，舉凡 RosettaNet

及各種組織所定義的 XML，都具有多份 XML 文件符合同一個 DTD 的定義之特性。所以我們針對相同 DTD 的多份 XML 文件，找出解決其查詢處理問題的方法。至於使用者所用的查詢語言，我們參考 W3C 制定的 XML 文件之查詢標準，也就是 XQuery。由於 XML 資料其文件內的元素之間有著階層關係，而可以形成一樹狀結構，若搜尋（traverse）該樹而將經過的元素集合起來，則可構成一條的路徑，稱為路徑表示法(path expression)，該表示法為 XQuery 的基礎。

因此，本篇論文將著重在 XQuery 中多個路徑表示法的處理，探討如何對多份相同格式的 XML 文件完成有效的資料查詢。

1.2 相關研究

種種 XML 在資料庫學門所帶來的影響以及可能的研究方向，在文獻中已可見廣泛的討論 [18]，以下則針對和本論文較為相近的研究提出討論。

由史丹佛（Stanford）大學提出的 Lore 計劃，實做一套半結構化資料(semistructured data) 的管理系統，該系統的查詢語言 Lorel [1]，能夠表示比 SQL 更具彈性的查詢句。為了針對此語言的查詢過程做最佳化的處理，相對應的索引結構也同時被提出 [15]。由於 XML 具有半結構化的特性，所以該研究有很大的參考性，而史丹佛大學也逐漸把該系統轉化為 XML 資料的管理系統。他們除了針對 XML 資料可能的路徑表示法提出數個索引結構，另外也提出更複雜的資料庫統計與執行策略，使該系統能夠對 XML 資料提供有效的查詢處理 [16]。

至於在單一 XML 文件的查詢處理方面，有的研究者針對 PAT algebra 提供的四種運算，也就是內容選擇（content-select）、屬性選擇（attr-select）、字串包含和被包含（includes 和 incl-in）等，設計以事件為基礎（event-based）和以樹狀為基礎（tree-based）的處理方式 ，同時討論最佳化處理的方法 [4]。有研究者則設計了聚集（aggregation）函式，並利用傳統資料庫 “pushing selections” 的方法，為查詢處理做最佳化 [12]。有的研究者則替 XML 查詢提供一個代數的

方法，然後依此基礎設計同時使用數個不同資料庫的方式，並達到最佳的執行策略 [7]。

另外，針對路徑表示法的查詢處理也受到很多研究者的重視。在 [14] 中，作者提出一種能快速決定 XML 文件中各元素之間祖孫關係之結構的編碼，並以此編碼為基礎替 XML 資料提供了一個儲存及索引的系統，同時也提供了一系列的演算法，使路徑表示法的查詢處理較傳統的方法快。有的研究者則直接將 XML 文件中所有的路徑以字串的方式編碼，同時以 Patricia trie 為基礎建立了一個稱為 Index Fabric 的特殊索引結構，此特殊索引結構是將 Patricia trie 以 block 為單位分成多個子 trie (subtries) ，再依這些子 trie 建立多層次的索引結構，如此一來便可以顯著的改善查詢時所需之磁碟存取次數 [9]。另外在做查詢最佳化時，我們可能需要知道某一個路徑所需要的花費，所以有研究者探討如何依給定的路徑表示法計算可到達此路徑之最大可能性 (selectivity) [2]。另外作者提出的資料結構簡化原有之 XML 結構，可以佔用較少的記憶體空間，所以便於處理網際網路上大量且複雜的 XML 資料。

處理眾多 XML 文件或資料可能產生的效能問題也是一個重要的議題。XML 綱要目錄 (XML schema directory) ，是一個特殊的索引結構，它將不同 XML 文件的綱要定義做一個統一而有效的管理，所以可以從眾多 XML 文件中找到查詢所需要的文件，而加快 XML 文件的查詢速度 [13]。至於在 [11] 中，則設計了一個多緒 (multi-threaded) 的系統，可以平行處理多個查詢 XML 資料的需求，所以效能可以達到可調整 (scalable) 的目的。

有研究者則針對 XML 文件實作一個管理系統，稱做 Xyleme，整個系統的架構共可分為三大部分，分別為資料儲存、索引結構及使用者介面。此系統將網際網路上大量異質性的 XML 文件整合後定義 view，並提供透過 view 做查詢的方法，系統則會處理實體資料與 view 之間的轉換。另外此系統也提供對 XML 文件做全文檢索、以及處理近似查詢 [3, 8]。

由於 XML 本身具有文件的特色，所以傳統在文件尋找資料的資訊擷取

(Information Retrieval) 領域，也有不少對 XML 的研究。在 [17] 中，作者提供了一個方法，從 XML 文件中找尋近似於查詢句的答案。其方法是將所要查詢的資料和查詢模組化，使得它們同處在一樣的樹狀結構下，再將問題轉到未排序樹包含問題 (unordered tree inclusion problem) 上，來說明查詢和資料間的關係，並提出趨近樹嵌入 (Approximate Tree Embedding) 的演算法，也就是如何在樹中找趨近樹，以便找出近似的答案。

另外，有研究者利用傳統資訊擷取裏常用的 LSI 方法來處理 XML 文件。LSI 是以向量空間為基底，因此必須做大量的向量相乘而花費時間，於是在 [6] 中，作者將 4 位元組的浮點數轉成 1 位元組的整數，並建立 Quick Look-up Table 和 Truncated Cosine Computation 來加快計算。並且，為了避免線性掃描 (Linear Scan) ，使用 R-Tree 的變形來儲存文件的叢集 (Cluster) 以加快搜尋的速度。

1.3 研究目標

XML 的提出只有短短幾年的時間，但是已經受各界的重視，許多研究者分別針對 XML 的特性進行相關的研究，如查詢處理、路徑表示法及效能問題等都有不少研究者涉入，而工業界幾家重要的軟體廠商，如 Oracle、HP、IBM、Microsoft 等，也皆全力的投入，並配合 W3C 的規格，不斷地推出新的產品，我們深信 XML 的重要性絕對是不可忽視，而如何針對大量的 XML 資料做有效率的查詢處理將是一個重要的課題。

對於 XML 資料的查詢處理，因為 XML 本身為樹狀結構，所以路徑表示法佔有重要的地位。另外由於 XML 文件的特性，相同元素在文件中可以重複出現，所以相同的路徑表示法在一份 XML 文件中有可能對應到許多不同的內容，如何依使用者的要求，過濾這些不同的答案，找出符合查詢條件的結果是個很重要的議題。

本論文探討多份符合相同 DTD 之 XML 文件的查詢處理，所以本論文的研

究目標如下：首先，在兼顧查詢效率高與儲存空間小的考量下，根據 DTD 的特性設計路徑表示法的編碼，讓使用者可以利用路徑表示法快速取得 XML 文件中的資料。再者，將 XML 文件以特殊的格式儲存，並保留元素之間的父子關係，以省去解析 (parse) XML 文件的時間。最後，利用上述的資料結構設計可以找出符合使用者要求的查詢結果之演算法。

基於上述的目標，在本論文中我們設計了三個特殊的資料結構。首先，由於 DTD 定義了 XML 文件中元素可以出現的型態，所以由 DTD 中可以找出這些 XML 文件內所有可能出現的路徑表示法，我們將這些路徑表示法編碼並記錄在元素編碼樹 (EN-Tree)。為了避免查詢過程中 XML 文件解析時所花的額外時間，我們設計了元素值編碼表(EV-Table)，它是針對單一 XML 文件，保留文件內元素間的祖孫關係，並將欲查詢的資料記錄下來。文件索引(Document index) 則可以協助我們從多份元素值編碼表檔案中快速找到所需要的檔案。基於這三個資料結構，我們設計了一組正確且有效率的查詢演算法，以加快利用路徑表示法尋找 XML 資料的速度。

為證實我們設計的特殊索引結構及查詢演算法的效率，我們使用 ACM SIGMOD Record 的 XML 文件為測試資料，與 Microsoft SQL Server 2000 關聯式資料庫及史丹佛 (Stanford) 大學提出的 Lore 做比較。實驗結果證實我們所提出的演算法對於任意長度的路徑表示法有較佳的執行效率。

1.4 論文架構

本論文的基本架構介紹如下：除了第一章緒論外，我們於第二章中，首先簡介 XML，說明其特殊的資料儲存格式及定義的方式，然後說明 W3C 所提出之 XML 查詢語言的標準；在第三章中，介紹我們為 XML 資料查詢處理所設計的資料結構及其演算法，包括元素編碼樹、元素值編碼表及文件索引；利用這三個資料結構所做的查詢處理之演算法則在第四章中討論；我們對查詢處理的效能所

做的實驗，則在第五章中討論；最後，在第六章提出結論，並指出未來的研究方向。

第二章　相關定義

在本章中，我們將說明 XML 文件之格式及定義的方式，然後說明 W3C 所提出之 XML 查詢語言的標準，稱為 XQuery。

2.1 XML 的相關定義

XML 描述的資料物件稱為 XML 文件，和 HTML 文件一樣起源於 SGML。但是 HTML 裏的標籤，為控制輸出和排版之用（ 如<table>，<font>，<ol>，<li>…等 ），而 XML 則允許使用者自訂標籤，以說明內容資料之意涵。如圖 2.1 裏的 XML 文件範例，它表示了在 ACM SIGMOD Record 之一般議題論文集（Ordinary Issue Page）中，1999 年 3 月份第 28 冊第 1 號裏的兩篇文章(articles)，每一篇文章則分別描述了文章名稱（title）、開始頁數（initPage）、終止頁數（endPage）、作者（authors）等訊息。由此我們可看出 XML 文件利用適當的標註，可以提供資料的結構及與語意有關的資訊。特別注意的是，該文件只表示“資料”，並無指定“顯示介面”。

接下來，我們將進一步的說明 XML 文件的結構。一份 XML 文件主要是由宣告（ declaration，或稱 prolog ）、處理指令（ processing instructions ）、元素（ element ）以及註解（ comments ）所組成。圖 2.1 中左欄的 Li 代表行數，其中 L1 行為 XML 宣告，說明此檔案採用的編碼為 ISO-8859-1。L3 行的 OIP 為本文件中的根元素（root element），一個 XML 文件中只能有一個根元素。為了方便說明，在本論文中，我們將原始檔案裏的元素名稱 OrdinaryIssuePage 及 sectionListTuple，分別簡化為 OIP 及 sLT。而每個元素中都可以包含其他元素（ 子元素 ）及屬性（ attribute ），所有的屬性之值必須加上單引號或雙引號，如 L10 行的 sectionName 是直屬於 L9 行的 sLT 這個母元素下的子元素，它另外定義了一個屬性，屬性等號後面的值則為屬性值，譬如 id 屬性的值為 000。此外，XML 要求文件必須格式正確（ well-formed ），亦即每一個元素均須包含開始標籤（start-tag）及結束標籤（end-tag），如圖 2.1 的 L3 行之開始標籤 <OIP> 對應到

```
L1  <?xml version='1.0' encoding='ISO-8859-1' ?>
L2  <!DOCTYPE OIP SYSTEM '../DTD/OIP.dtd'>
L3  <OIP>
L4      <volume>28</volume>
L5      <number>1</number>
L6      <month>March</month>
L7      <year>1999</year>
L8      <sectionList>
L9          <sLT>
L10             <sectionName id="000">Articles</sectionName>
L11             <articles>
L12                 <articlesTuple>
L13                     <toArticle>
L14                         <title id="00011000">Editor's Notes.</title>
L15                     </toArticle>
L16                     <initPage>2</initPage>
L17                     <endPage>2</endPage>
L18                     <authors>
L19                         <author id="00">Arie Segev</author>
L20                         <author id="00">Jennifer Widom</author>
L21                         <author id="00">Michael</author>
L22                     </authors>
L23                 </articlesTuple>
L24                 <articlesTuple>
L25                     <toArticle>
L26                         <title id="00028001">Message</title>
L27                     </toArticle>
L28                     <initPage>2</initPage>
L29                     <endPage>2</endPage>
L30                     <authors>
L31                         <author id="00">Won Kim</author>
L32                     </authors>
L33                 </articlesTuple>
L34             </articles>
L35         </sLT>
L36     </sectionList>
L37 </OIP>
```

圖 2.1：XML 文件範例

```
L1  <?xml version="1.0" encoding="Big5"? >
L2  <!ELEMENT OIP ( volume,number,month,year,sectionList ) >
L3  <!ELEMENT volume ( #PCDATA ) >
L4  <!ELEMENT number ( #PCDATA ) >
L5  <!ELEMENT month ( #PCDATA ) >
L6  <!ELEMENT year ( #PCDATA ) >
L7  <!ELEMENT sectionList ( sLT ) * >
L8  <!ELEMENT sLT ( sectionName,articles ) >
L9  <!ELEMENT sectionName ( #PCDATA ) >
L10 <!ATTLIST sectionName id    CDATA    #IMPLIED >
L11 <!ELEMENT articles ( articlesTuple ) * >
L12 <!ELEMENT articlesTuple ( toArticle,initPage,endPage,authors ) >
L13 <!ELEMENT toArticle ( title ) *>
L14 <!ELEMENT title ( #PCDATA ) >
L15 <!ATTLIST title id CDATA    #IMPLIED >
L16 <!ENTITY % Xlink " xml:link CDATA #FIXED 'simple' href CDATA
L17 #IMPLIED inline ( true | false ) #FIXED 'true' ">
L18 <!ATTLIST toArticle %Xlink; >
L19 <!ELEMENT initPage ( #PCDATA ) >
L20 <!ELEMENT endPage ( #PCDATA ) >
L21 <!ELEMENT authors ( author ) * >
L22 <!ELEMENT author ( #PCDATA ) >
L23 <!ATTLIST author id CDATA #IMPLIED >
```

圖 2.2：DTD 範例

L37 行的結束標籤 </OIP>。每個元素的開始標籤與結束標籤須成對，標籤之間不可交錯，即所有元素的排列必須為嚴謹的樹狀 (巢狀) 結構。

另外在圖 2.1 的 L2 行裏使用了 OIP.dtd，其檔案的作用是檢查此文件裏出現的標籤是否正確 (valid) 。DTD (document type definition) 為 W3C 所頒訂用來定義某份 XML 文件的格式，也就是將每一個元素包含哪些子元素或屬性、各元素出現的順序等，清楚地加以定義和規範。用 DTD 定義出來的一套元素組合，通常稱為「語彙」(vocabulary) 。圖 2.2 即為 OIP.dtd 這一份 DTD 文件的內容。

在 DTD 文件中，ELEMENT 標籤之後放的是元素名，接著用小括號括起來的，是它的“內容模型”，也就是在對應的 XML 文件中可以出現的內容。圖 2.2

中的 L2 行說明，OIP 這個元素包含了 volume、number 等子元素，而在 L3 行中進一步註明 volume 元素存放的資料為 #PCDATA (parsable character data) ，其為預先定義的標記，代表可解析的文字資料。ATTLIST 標籤則是宣告元素的屬性，包含了屬性名稱、屬性類別及預設行為的描述，若屬性不只一個時，可以用這三個部分為一個單位一直重複下去。譬如，L10 行定義了 sectionName 這個元素具有 id 這個屬性，id 屬性的類別為 CDATA，表示該屬性值為文字資料，預設行為的描述為 #IMPLIED，表示該屬性是選擇性的，如果處理器沒有讀到指定的值，便會忽略此屬性。

特別注意的是，DTD 允許類似 regular expression 的符號，如 L7 行的星號 (*) 代表 sectionList 元素裏可包含多個 sLT 子元素。相較於結構化的關聯式資料庫 (relational database) ，每筆資料都必須具有固定個數的欄位，XML 提供了一種半結構化的 (semi-structured) 表示資料的方式，允許元素可以不出現，或出現一次以上。本論文將針對這個特性加以討論，其對應的查詢演算法在第四章中將會有詳細地說明。

2.2 XQuery 的相關定義

W3C定義了相當多的輔助技術來處理XML資料。譬如，Xpath是節點位置語言，用來描述XML元素的位置。如果我們要指定一個特定的元素，我們可以把它完整的路徑寫出來，如 " /OIP/sectionList/sLT "；或利用萬用字元取出不限定名稱的所有元素，如 " /OIP/sectionList/* "；或利用函數來做進一步地限制，如 " //*[count (author) =3] "，則是取出擁有三個 [author] 子元素的所有元素。但是為了提供更便利的查詢方式，W3C 在 2001 年的 2 月首先提出了 XML Query Requirements，討論對XML資料做查詢時的需求，隨後也在2001的6月提出了 XQuery 1.0 Working Draft，以作為XML查詢語言的標準。此節則是對XQuery加以說明。

```
L1  <Result>
L2  {
L3      for $a in document ( "http://dblab.cs.ntou.edu.tw/SIGRd1.xml " )
            /OIP/sectionList/sLT/articles/articlesTuple
L4      where contains ( $a/toArticle/title/text ( ) , " Editor's Notes." )
L5      return
L6              $a
L7  }
L8  </Result>
```

圖2.3：XQuery範例：找出文章名稱（title）是Editor's Notes.的書籍資料

XQuery是針對各類XML資料所設計，不論其儲存為文件或是在關聯式資料庫裏。XQuery其資料模型（data model）視一個XML文件為一棵標記（label）樹，並考慮標記樹中每個節點及所有值之順序關係。其特性是路徑表示法的使用，使用者可以利用路徑表示法在XML資料中尋找任意長度路徑的資料。而XQuery使用的路徑表示法，其語法是引用Xpath的語法，在計算XML文件的路徑表示法時會傳回一個有序列的資料，其順序是依照文件中各個元素之間的順序關係而定。另外使用者也可以從資料中取得關於schema的資訊。

XQuery 以 expression 為基礎，每個 expression 可能包含常數和變數，如圖 2.3 中，L3 行中的/OIP/sectionList/sLT/articles/articlesTuple 為一個路徑表示法，而此行表示要指定文件“ SIGRd1.xml ”內，符合此路徑表示法的每條路徑下所包含之元素給$a 這個變數。而 expression 之間可以是巢狀表示，或是互相被結合的，如$a 這個變數結合了 L3 行及 L4 行中的兩個 expression。另外 expression 還可以使用算術運算、包含邏輯或排序等的各種運算子、函數呼叫、元素的建構子（constructor）等，如圖 2.3 的 L4 行中之 contains ( ) 就是屬於函數呼叫，可用以檢查元素的內容值是否包含某個特定字串。

XQuery 主要的功能就是讓使用者下達查詢句，並且決定哪些 XML 元素的查詢結果應該被傳回來。一個 XQuery 的敘述式為一個 FLWR (for-let-where-return) expression，其中四個部分各自有其功能：for 讓變數可以遞迴取得（iterate over)一個 expression 的結果；而 let 則是將變數直接與某一個的 expression 結合

```
L1  <Result>
L2      <articlesTuple>
L3          <toArticle>
L4              <title id="00011000">Editor's Notes.</title>
L5          </toArticle>
L6          <initPage>2</initPage>
L7          <endPage>2</endPage>
L8          <authors>
L9              <author id="00">Arie Segev</author>
L10             <author id="00">Jennifer Widom</author>
L11             <author id="00">Michael J. Franklin</author>
L12         </authors>
L13     </articlesTuple>
L14 </Result>
```

圖 2.4：對應於圖 2.3 之 XQuery 的查詢結果

(binding) ；where 則允許對變數做條件的限制；return 則可以建構新的 XML 元素為查詢的輸出。

接下來我們舉例來說明 XQuery 的查詢句。假設圖 2.1 的 XML 資料存放在網址“ http://dblab.cs.ntou.edu.tw ”下檔名為 SIGRd1.xml 的文件內，若不指定檔名表示查詢句必須處理該網址下所有的 XML 文件，圖 2.3 則表示當我們要從該檔案找出文章名稱是 Editor's Notes.的書籍資料時，所下的 XQuery 查詢句。L1 行及 L8 行指定輸出之 XML 資料的根節點為<Result> ，及其結尾標籤為</Result>。L2 行及 L7 行表示 XQuery 的敘述式要用大括弧 “{” 和 “}”包起來。L3 行指定 “SIGRd1.xml”檔案內的路徑表示法為/OIP/sectionList/sLT/articles/articlesTuple 的每一條路徑下所包含之元素給$a 這個變數，我們同時可以指出文件的所在位置 (URL) 為“ http://dblab.cs.ntou.edu.tw/ ”。L4 行的目的在限定 $a 元素下的 title 子元素必須包含 “Editor's Notes.”這個字串，L5 行及 L6 行表示將符合條件的查詢結果傳回來並印出來，也就是印出/OIP/sectionList/sLT/articles/articlesTuple 的內容。此查詢句傳回來的結果如圖 2.4 所示。

本論文所提出的查詢處理，主要是從多份 XML 文件中，找出符合利用路徑

表示法的 XQuery 查詢句，同時也針對路徑表示法中的元素對應到 XML 文件中之真實資料的元素可能重複出現的情況，也就是如圖 2.2 中 L7 行星號 (*) 的情況，設計了特殊的查詢演算法。

第三章　資料結構

在本章中，我們將介紹為了便利查詢處理所設計之資料結構，及建置這些資料結構的演算法。這些資料結構分別為元素編碼樹（EN-Tree）、元素值編碼表（EV-Table）及文件索引（Document index）。其中，元素編碼樹提供查詢句中路徑表示法與編碼的對應，其做法是根據 DTD 的定義，將所有元素依照其排列形成的樹狀結構加以編碼，然後紀錄下來。元素值編碼表的提出則是為了避免查詢過程中，因解析（parse）XML 文件所花的額外時間，其做法是針對每份 XML 文件，紀錄元素間的祖孫關係，以及元素的內容值。而文件索引則可以協助我們從多份元素值編碼表檔案中快速找到所需要的檔案。這三種資料結構將分別在下列各節中詳細說明。

3.1 元素編碼樹

我們提出元素編碼樹的目的是為了將冗長的路徑表示法轉換成以單一數字表示的編碼。首先我們定義所採用的編碼方式如下：

【定義 3.1】已知一個樹狀結構，若以前序（preorder）的方式對此樹狀結構的每個節點加以編碼，則稱這些編碼為每個節點的***節點編碼***。

另外，為了方便之後的說明，我們也加以定義***祖先節點***。

【定義 3.2】已知一個樹狀結構及一個節點 N，由節點 N 到根節點的路徑上存在的任何節點，為節點 N 之***祖先（ancestor）節點***。

元素編碼樹的做法是針對一份 DTD 文件內所有的元素（在本論文中某元素所包含的屬性皆視為該元素的子元素），依各元素之間巢狀關係所構成的樹狀結構，取得節點編碼後並依照其樹狀結構紀錄下來，其特性如下所列：

(1) 節點由 DTD 中之元素轉換而成，其名稱依照各元素在 DTD 中的名稱命名，元素名稱中有星號（*）者，此星號也一併為節點名稱。(星號表示在 XML 文件中會重複出現的元素。)

(2) 各節點的階層關係依照各元素在 DTD 中出現的順序以及與其它元素之間的

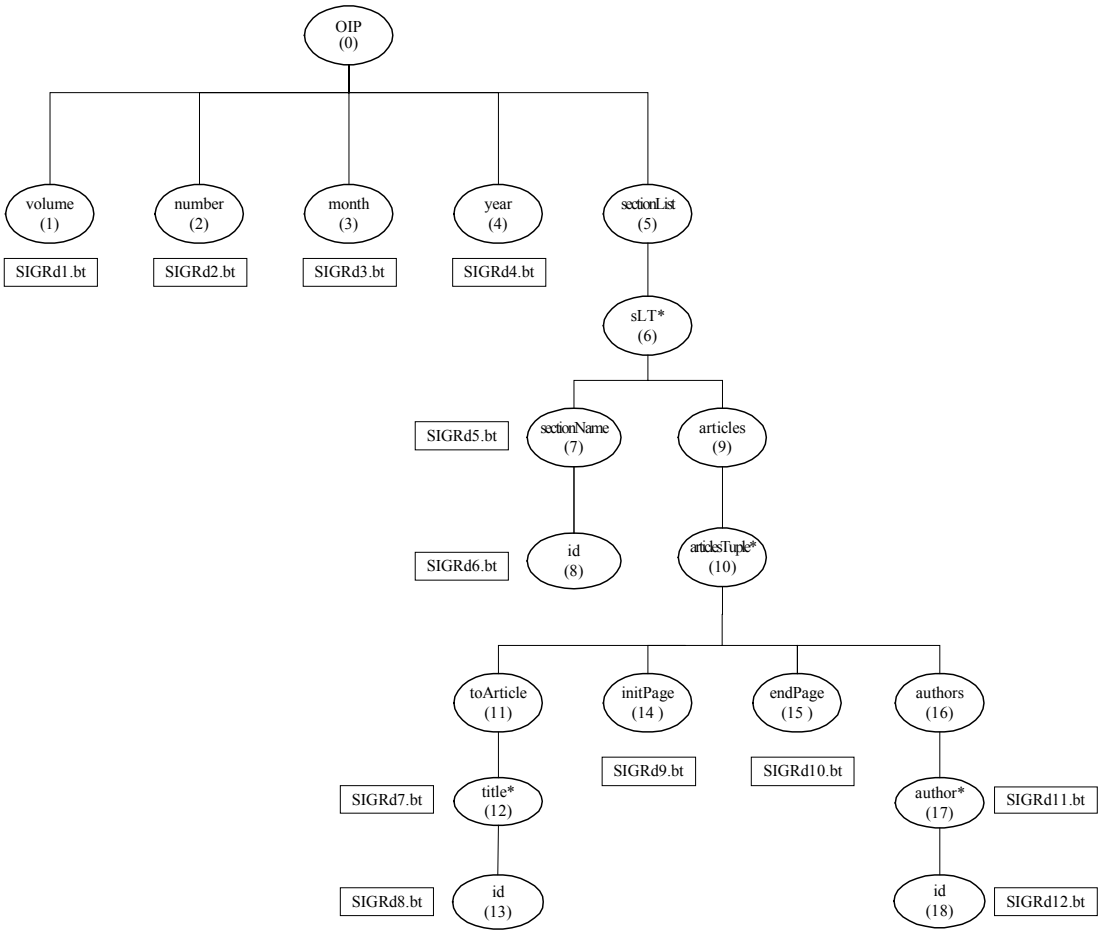

圖 3.1：元素編碼樹結構

父子關係。

(3) 每個節點指定一個節點編碼 (參見定義 3.1)。

(4) 一個特定的節點稱為根節點 (root node) ，是從 DTD 中的根元素轉換來的。

(5) 最下層的所有節點為外部節點 (external node) 又稱為樹葉節點 (leaf node) 。

(6) 由根節點到外部節點之間的所有節點稱為內部節點 (internal node)。

(7) 若某外部節點具有實質內容，則記錄其對應的文件索引的之起始檔案名稱 (於第 3.3 節將有詳細的介紹)。

圖 3.1 中的元素編碼樹是由圖 2.2 的 DTD 轉換過來的。根節點為 OIP，外部節點如 volume、number、id 等，內部節點如 sectionList、sLT、articles 等，而父子關係則如圖中的 OIP 與 volume，OIP 稱為 volume 的父親節點，而 volume 稱

```
L1  <OIP id="0">
L2      <volume id="1">SIGRd1.bt</volume>
L3      <number id="2">SIGRd2.bt</number>
L4      <month id="3">SIGRd3.bt</month>
L5      <year id="4">SIGRd4.bt</year>
L6      <sectionList id="5">
L7          <sLT* id="6">
L8              <sectionName id="7"> SIGRd5.bt
L9                  <id id="8"> SIGRd6.bt </id>
L10             </sectionName>
L11             <articles id="9">
L12                 <articlesTuple* id="10">
L13                     <toArticle id="11">
L14                         <title* id="12">SIGRd7.bt
L15                             <id id="13">SIGRd8.bt </id>
L16                         </title>
L17                     </toArticle>
L18                     <initPage id="14">SIGRd9.bt </initPage>
L19                     <endPage id="15">SIGRd10.bt</endPage>
L20                     <authors id="16">
L21                         <author* id="17">SIGRd11.bt
L22                             <id id="18">SIGRd12.bt </id>
L23                         </author>
L24                     </authors>
L25                 </articlesTuple>
L26             </articles>
L27         </sLT>
L28     </sectionList>
L29 </OIP>
```

圖 3.2：以 XML 表示之元素編碼樹範例

為 OIP 的兒子節點。節點編碼為每個節點名稱下方括號內的數字所示，如 initPage 的節點編碼為 14，而其對應的文件索引的之起始檔案名稱為 SIGRd9.bt。

由於元素編碼樹保留原來 DTD 中元素之間的樹狀結構，所以針對查詢句中的某一條路徑表示法，我們可以很快速地找到對應的節點編碼。同時由於元素編碼樹記錄了對應特定節點的文件索引，所以透過該樹可以一次處理多份 XML 文

件，而不必對每份 XML 文件做重複的路徑搜尋，將查詢的效率顯著的提昇。

圖 3.1 的元素編碼樹可以用圖 3.2 的 XML 檔案來表示。樹中的節點對應到元素名稱，而節點編碼則以該元素的屬性 id 的值來表示。外部節點所對應的文件索引之起始檔名（如 SIGRd1.bt），則表示為該元素的內容值。建立此檔案的演算法表示在圖 3.3。該演算法的輸入資料為所有欲查詢的 XML 文件所符合的 DTD 文件。在演算法中依照此 DTD 內元素之間的父子關係，將所有元素以樹狀結構儲存下來，在這裏我們將各元素的屬性視為該元素的子元素，然後利用 EncodeId 函式為此樹狀結構中的每個節點以前序的方式編碼，EncodeId 函式的演算法如圖 3.4 所示，該演算法的輸入為樹狀結構中的根節點元素名稱，而輸出則為一個以前序的方式指定好節點編碼的樹狀結構，最後再將此指定好節點編碼的樹狀結構輸出成元素編碼樹檔案。而外部節點對應的文件索引檔案，是在建立每條路徑所對應到的文件索引時，才將其根節點檔名稱紀錄進去。

利用建構好的元素編碼樹，讓我們在查詢的過程中可以快速地將一長串的路徑表示法，依照其終止元素轉換成以單一數字表示的編碼，有利於查詢的處理。[*範例 3.1*] 在處理圖 2.3 的查詢時，我們可以利用圖 3.1 的這個元素編碼樹，將 OIP/sectionList/sLT/sectionName 這個路徑表示法，依照其終止元素的節點編碼轉換成為單一數字 7，由於此路徑為查詢句中的 where 條件，我們同時利用檔案名稱 SIGRd5.bt，去對應的文件索引找到適合的文件加以處理。■

同時，採用單一數字編碼來取代一長串的路徑表示法之另一原因，是由於單一數字表示的編碼可以減小元素值編碼表檔案所需的儲存空間，我們將於 3.2 節詳細說明。

Algorithm Construct_EN-Tree
輸入：*d* /* DTD 文件 */
輸出：*EN_Tree* /*元素編碼樹檔案 */
/*變數說明：
nid：
節點編碼。
EP：
在 DTD 文件的每列中依序取得之父元素。
{*EC*}：
在 DTD 文件的每列中依序取得之子元素集合。
TS：
用於儲存 *EP* 及{*EC*}父子關係的樹狀結構。
R：
TS 的根節點。
*/
BEGIN {Algorithm Construct_EN-Tree}
Step 1 /* 初值設定 */
設定 *nid*=0。
Step 2 /* 文件預先處理 */
Step 2.1 在 DTD 文件中，經由循序的掃描，首先將屬性定義中有使用到 ENTITY 的部分都替換掉。
Step 2.2 將所有開頭字串為<!ATTLIST 的列皆轉換成與開頭字串為<! ELEMENT 的列相同的形式來儲存 /* 也就是將<!ATTLIST 列中屬性名稱的部分以同於處理<!ELEMENT EC (#PCDATA) >的方式來處理 (參見 2.1 節) */。
Step 3/* 對所有元素編碼 */
Step 3.1 再次循序掃描 DTD 文件。從文件的每列中依序取得父元素 *EP* 及子元素{*EC*}，並依其父子關係儲存於樹狀結構 *TS* 中。 /* 特別注意的是{*EC*}中有星號 (*) 的元素，此星號也一並儲存於樹狀結構 *TS* 中 */。
Step 3.2 假設 *TS* 的根節點以 *R* 表示。呼叫 EncodeId (*R*, 0) ，為 *TS* 中所有元素編碼。
Step 4 /* 輸出檔案 */
搜尋(traverse) *TS* 將所有元素及編碼寫入元素編碼樹檔案中。
END {Algorithm Construct_EN-Tree}

圖 3.3：Construct_EN-Tree 演算法

Subroutine EncodeId
輸入：*EP* /* 樹節點名稱 */、*nid* /* 節點編碼 */
輸出：一個指定好節點編碼的樹狀結構
/*變數說明：
{*EC*}：
EP 之子節點集合。
*/
BEGIN {Subroutine of EncodeId }
Step 1/* 將 *nid* 指定給 *EP* */
Step 1.1 將 *nid* 值指定給輸入的 *EP*。
Step 1.2 *nid* = *nid*+1。
Step 1.3 若 *EC*= “#PCDATA”，則離開此 Subroutine。
Step 1.4 若 *EC* ! = “#PCDATA”，則針對每個 *EC* 遞迴呼叫 EncodeId。
END {Subroutine of EncodeId }

圖 3.4：EncodeId 函式

3.2 元素值編碼表

為了避免查詢過程中，因解析 XML 文件所花的額外時間，我們提出了以元素值編碼表來取代 XML 文件的方法，也就是在查詢處理之前就預先將所有欲查詢的 XML 文件轉換成元素值編碼表，同時為了減小儲存元素值編碼表所需的空間，我們在元素值編碼表裏利用元素編碼樹中的節點編碼來取代 XML 文件中每個元素的名稱。

元素值編碼表的每一列代表 XML 文件中的一個元素，每個元素值編碼列 (EV-tuple) 紀錄 XML 文件中每個元素值節點編碼 (Index) 、該元素名稱節點編碼(Node id) 、父節點編碼 (Parents) 及元素內容 (Value)，現分述如下：

- 元素值節點編碼 (Index)：一份 XML 文件裏的所有元素，可以根據其巢狀關係以及出現順序呈現出一個樹狀結構，我們依照定義 3.1 的方式，針對樹狀結構中每個元素以前序的方式指定節點編碼。
- 元素名稱節點編碼 (Node id)：利用在元素編碼表中，每個元素在 DTD 樹狀關係中的節點編碼，來表示該元素值在 XML 文件中的元素名稱。
- 父節點編碼 (Parents) ：用來記載 XML 文件中每個元素的父親節點，以其元素值節點編碼表示，若其值為 **-1** 表示此元素為根元素，沒有父親節點。
- 元素內容 (Value) ：用來將 XML 文件中每個不同元素所包含的內容 (值) 記下來。

將一份 XML 文件的所有元素值編碼列集合起來，依照其*元素值節點編碼*排序，則構成一個元素值編碼表，如表 3.1 所示。此表格對應到圖 2.1 的 XML 檔案。其第一欄為每個元素的索引值，同時也表示了在此表中的先後順序，故稱作索引。值得注意的是，在元素編碼樹中有星號 (*) 出現的元素，在此表可能會對應到多個不同的元素值編碼列。譬如，索引 17 和 19 對應到相同的元素編碼。

表 3.1：元素值編碼表

Index	Node id	Parents	Value
0	0	-1	
1	1	0	28
2	2	0	1
3	3	0	March
4	4	0	1999
5	5	0	
6	6	5	
7	7	6	Articles
8	8	7	000
9	9	6	
10	10	9	
11	11	10	
12	12	11	Editor's Notes.
13	13	12	00011000
14	14	10	2
15	15	10	2
16	16	10	
17	17	16	Arie Segev
18	18	17	00
19	17	16	Jennifer Widom
20	18	19	00
21	17	16	Michael
22	18	21	00
23	10	9	
24	11	23	
25	12	24	Message
26	13	25	00028001
27	14	23	2
28	15	23	2
29	16	23	
30	17	29	Won Kim
31	18	30	00

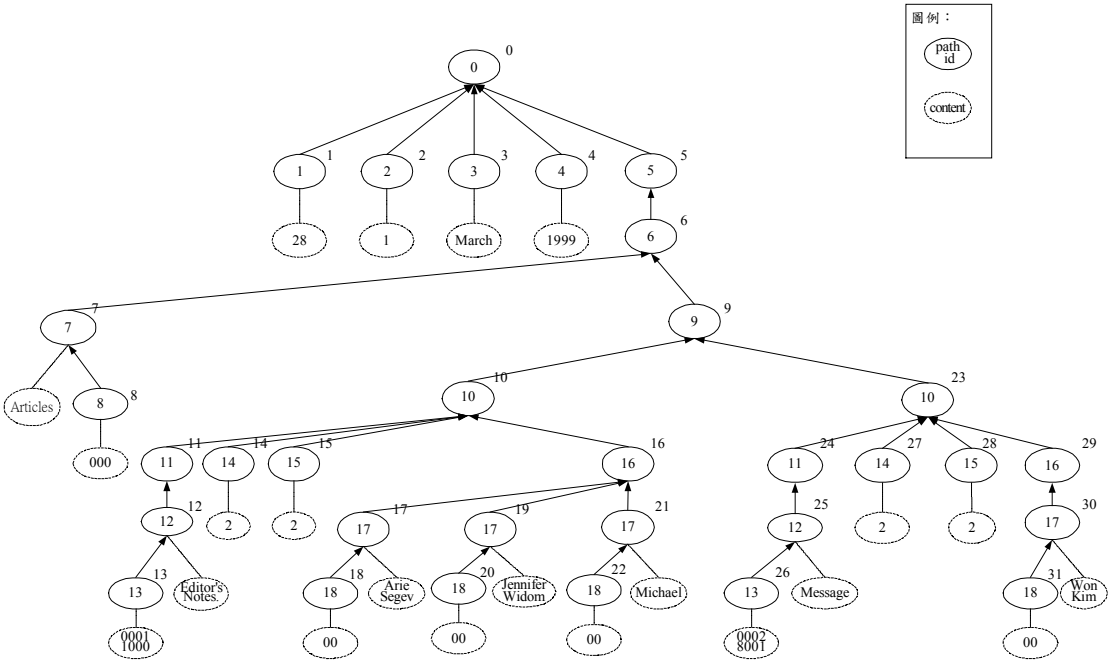

圖 3.5：元素值編碼樹

我們利用表 3.1 的第三欄 (Parent)，將該元素值編碼表所對應的樹狀結構表示成圖 3.5，我們稱此樹狀結構為元素值編碼樹 (EV-Tree) 。在此樹狀結構中每個實線節點對應到元素值編碼表中的每個元素值編碼列，而每個實線節點右上方的數字，代表其*元素值節點編碼*，而實線節點裏的數字，代表其*元素名稱節點編碼*，若此節點對應到的元素有內容值，則在其下方以一個虛線節點來表示。除了根節點之外，每個實線節點都有一條有向的實線指向自己的父親節點，因此可以表示元素值編碼表中的祖孫關係。

使用元素值編碼表來取代 XML 文件的方法與直接針對 XML 文件做處理的方法相比較，其優點是使用元素值編碼表可以省去查詢過程中解析 XML 文件時所花的時間，而利用元素編碼樹中的編碼來取代 XML 文件中的元素名稱，更可以減小元素值編碼表所需的儲存空間，由此可見元素值編碼表的提出確實可以改善一般直接處理 XML 文件時所造成的缺失。

圖 3.6 為建立元素值編碼表的演算法，該演算法的輸入資料為欲查詢的 XML 文件及所對應的元素編碼樹。此演算法首先將一份 XML 文件中的每個元素名稱以元素編碼樹中的節點編碼一一取代，再將每個元素依其在 XML 文件中的父子

Algorithm Construct_EV-Table

輸入：XML 文件及元素編碼樹
輸出：元素值編碼表
/*變數說明：
PE、element：
　　為 XML 文件中之標籤名稱。
nid：
　　為標籤對應的元素名稱節點編碼。
Evalue：
　　為 XML 文件中之*元素內容*。
index：
　　為 XML 文件中之元素值節點編碼，亦為元素值編碼列的索引。
pindex：
　　為元素值編碼列的*父節點編碼*。
value：
　　每個元素值編碼列的*元素內容*。
*/
BEGIN {Algorithm Construct_EV-Table}
　Step 1/* 初值設定 */
　　設定 *index* = 0。
　　設定 *pindex* = -1，且 push *pindex* 到 stack 中。
　Step 2 /* 讀取資料 */
　　掃描 XML 檔案取得下一筆元素 <PE>，若是已經沒有任何元素，則結束程式。
　Step 3/* 在元素值編碼表中寫入*元素值節點編碼*、*元素名稱節點編碼*
　　*父節點編碼*這三個欄位 */
　　利用元素編碼樹找出 PE 之元素名稱節點編碼 *nid*，取出 stack 最上層之 *pindex*，在元素值編碼表檔案中寫入 *index*、*nid*、*pindex*。
　Step 4 /*更新 stack 和 index */
　　將 PE 之 *index* push 到 stack 中。*index* = *index*+1。
　Step 5/* 在元素值編碼表中寫入 *value* 這個欄位 */
　　掃描 XML 檔案取得<PE>之後的內容值 Evalue ，將其寫入元素值編碼表檔案。繼續掃描 XML 檔案。
　　Step 5.1/* PE 處理完畢 */
　　若遇到</PE>，則將 stack 中最上層元素 pop 出來，跳到 Step 2。
　　Step 5.2/*處理 PE 的子元素 */
　　若遇到另一個<element>時，將 element 指定給 PE，跳到 Step 3。
END {Algorithm Construct_EV-Table}

圖 3.6：Construct_EV-Table 演算法

關係以表格的每一列儲存下來，成為一個元素值編碼表。

利用建構好的元素值編碼表，讓我們在查詢的過程中可以省去解析 XML 文件的時間，快速地將某路徑表示法經由元素編碼樹轉換得到的節點編碼，在元素值編碼表中取得符合該節點編碼的*元素內容*。

[*範例 3.2*] 我們可以利用範例 3.1 所得到之*元素名稱節點編碼* 7，配合表 3.1 的元素值編碼表，將編碼 7 的*元素內容*找出來，得到"**Articles**"。 ■

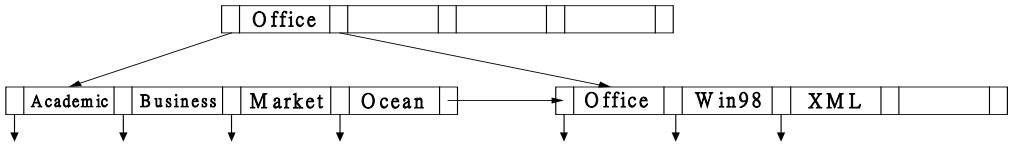

圖 3.7：文件索引結構

最後，若相同的元素名稱節點編碼在元素值編碼表中重複出現時，我們會利用表格中第三個欄位所記載祖孫關係的資訊，經由祖先關係的比較，取得正確的查詢結果。這個查詢處理的過程，我們將在第四章的查詢演算法中說明。

3.3 文件索引

接下來我們介紹為多份元素值編碼表所設計的索引結構[5, 19, 20]。當 XML 檔案的資料量極大時，其所對應的元素值編碼表檔案也會隨著大量增加，若要查詢某筆特定資料時，以循序的方法搜尋所有檔案，效率會非常不理想。因此，我們針對特定的路徑，建立查詢值（search key）與元素值編碼表檔案對應的索引結構，如此便可直接開啟所需要的元素值編碼表檔案，加快查詢的速度。

我們提出的索引結構，是依據目前傳統資料庫裏最常見的 B^+-Tree 樹狀結構所設計。一棵樹包含樹根（root node）、內部節點（internal node）及外部節點(external node)，每個節點皆存放數個查詢值與指標。如圖 3.7 所示，最上層的節點就是樹根，同時它也是此樹唯一的內部節點，會指向其他內部節點或外部節點。而最下層的兩個節點是外部節點，指向真正資料存放的檔案位置。當我們要查詢某一個值時，必須從樹根開始，根據查詢值的大小往下走到最底層，然後循著指標將資料取回。

文件索引的特色在於其樹狀結構是平衡的，也就是從樹根到外部節點的每條路徑都是相同的長度，所以搜尋每個值所需的時間大致相同。另外每個節點的大小有所限制。若文件索引的 order 為 n，則每個內部節點（除了根節點）最多有 n 個指標（或 n-1 個查詢值），最少必須有$\lceil n/2 \rceil$個指標（或$\lceil (n-1)/2 \rceil$個查詢值），所以文件索引之高度不大於$\left| \log_{\lceil n/2 \rceil} K \right|$，這裏 K 為查詢值的個數。假設查

```
L1  <?xml version="1.0" encoding="Big5"?>
L2  <!ELEMENT Node ( Pointer*,Next? ) >
L3  <!ATTLIST Node type ( ex|in ) "ex">
L4  <!ELEMENT Pointer ( #PCDATA ) >
L5  <!ATTLIST Pointer key CDATA #REQUIRED>
L6  <!ELEMENT Next ( #PCDATA ) >
```

圖 3.8：文件索引節點檔案之 DTD

```
L1  <?xml version="1.0" encoding="Big5" ?>
L2  <!DOCTYPE BTree-Node SYSTEM "Btree.dtd">
L3  <Node type="ex">
L4    <Pointer key="Office">SIGRd1</Pointer>
L5    <Pointer key="Win98">SIGRd2</Pointer>
L6    <Pointer key="XML">SIGRd3</Pointer>
L7    <Next>B3.bt</Next>
L8  </Node>
```

圖 3.9：文件索引節點之範例檔案

詢值為 10,000 筆，當 order 為 20 時，則文件索引之高度只有 4，所以查詢的速度相當快。圖 3.7 的文件索引其 order 為 5。

我們的設計是針對特定的路徑建立文件索引。由於在 XQuery 的語法裏，路徑表示法都是由根節點開始，至某元素終止，因此我們可以用該終點元素代表其路徑，更進一步使用其*元素名稱節點編碼*代替。所以，我們針對元素編碼樹中，特定的*元素名稱節點編碼*，將所有元素值編碼表中，符合此*元素名稱節點編碼*的*元素內容*，建立在對應的文件索引中。建立文件索引的演算法，與傳統上建立 B^+-Tree 的方式大致相同，所以在此不再贅述。

每一個文件索引的節點存成一個 XML 檔案，其 DTD 如圖 3.8 所示。在該圖中，L2 行表示 Node 元素為本文件之根元素，內含多個文件索引之指標，以 Pointer 元素表示之。L3 行定義了 Node 的一個屬性 type，用來代表對應節點之型態，若值為 ex 代表外部節點，若值為 in 則代表內部節點。 L4 行及 L5 行分別定義 Pointer 元素的內容及其屬性 key，若此節點為外部節點，則 Pointer 元素的內容存放真實資料之檔名與位置，若為內部節點，則存放下一層文件索引節點之檔名，而屬性

key 值則存放此指標對應的查詢值如 Office。若此節點為外部節點，則 L6 行定義的 Next 元素之內容值代表下一個外部節點的檔名，以方便處理範圍查詢。

圖 3.9 為文件索引節點的一個範例檔案，對應到圖 3.7 右下角節點，其格式遵循圖 3.8 之 DTD。L3 行表示此檔案對應到一個外部節點，L4 行至 L6 行表示了此節點內的三筆資料，第一筆資料其查詢值為 Office，SIGRd1 則表示完整的資料存放在 SIGRd1 這個元素值編碼檔案中。最後 L7 行表示此外部節點的下一個外部節點，是對應到 B3.bt 這個檔案。

此外我們還必須記錄每一個文件索引，其根節點的檔名，以作為查詢的起點，此類資料記錄在元素編碼樹的每條路徑終點節點上，如圖 3.1 所示。

第四章　查詢演算法

在本章中，我們說明利用第三章提出的資料結構所設計的查詢演算法，並證明此演算法的正確性，最後舉例說明此演算法可以解決的查詢句。這個演算法應用我們提出的資料結構之特性，來處理以路徑表示法為基礎的查詢句，快速取得 XML 文件中的資料；並利用比較祖先關係的方法，解決元素值編碼表中，相同*元素名稱節點編碼*同時出現在不同元素值編碼列的問題。在 4.1 節中，我們將首先說明整個查詢處理的架構，於 4.2 節中，說明查詢演算法的執行過程及其優點，並在 4.3 節中舉例來詳細解說查詢演算法的處理步驟及其結果。

4.1 查詢處理的架構

為了解決利用任意長度的路徑表示法，快速查詢多份 XML 文件的問題，我們引用在第三章中介紹的資料結構來設計查詢的處理。在開始介紹整個查詢處理的架構之前，我們將先舉一個例子，來說明由於相同的 XML 元素名稱可能會在一份文件中重複出現，所以在查詢處理的過程中，比較祖先關係的必要性。首先我們先定義以下說明會用到的名詞。

【定義 4.1】令 X_{ij} 表示元素編碼樹（EN-Tree）中第 i 條路徑中的第 j 個節點所對應到之*元素名稱節點編碼*。對於元素編碼樹中的每條路徑 PN_i，假設其長度為 L_i，則該路徑可以用一串有序的*元素名稱節點編碼*$< X_{i1}, X_{i2}, X_{i3}\cdots X_{iL_i} >$來表示，其中 X_{i1} 為根*元素名稱節點編碼*，我們稱此*元素名稱節點編碼*的序列為***元素路徑編碼 (DTD encoding path)***。

[*範例 4.1*] 根據圖 3.1，路徑 OIP/sectionList/sLT/articles/articlesTuple/toArtlce/title 的*元素路徑編碼*為<0, 5, 6, 9, 10, 11, 12>。■

【定義 4.2】令 Y_{ij} 表示元素值編碼樹（EV-Tree）中第 i 條路徑中的第 j 個節點之*元素值節點編碼*。對於元素值編碼樹中的每條路徑 PV_i，假設其長度為 L_i，則該路徑可以用一串有序的*元素值節點編碼*$< Y_{i1}, Y_{i2}, Y_{i3}\cdots Y_{iL_i} >$來表示，其中 Y_{i1} 為根*元素值節點編碼*，我們稱此*元素值節點編碼*的序列為***元素值路徑編碼 (XML***

encoding path) 。

[*範例 4.2*] 根據圖 3.5，<0, 5, 6, 9, 10, 11, 12>為一條合法路徑。■

[*範例 4.3*] 假設要從 SIGRd1.xml 檔案，找出文章名稱為 Editor's Notes.的作者。對應的 XQuery 查詢句如下所示。

```
<Result>
    for $a in document("SIGRd1.xml")//articlesTuple
    where $a/toArtlce/title = " Editor's Notes."
    return
        $a/authors/author
</Result>
```

由於 SIGRd1.xml 已被轉換成如表 3.1 所示的元素值編碼表，而其 DTD 定義則對應到圖 3.1 的元素編碼樹，所以我們可以將查詢句中的兩條路徑表示法，也就是 where 條件的 OIP/sectionList/sLT/articles/articlesTuple/toArtlce/title，及 return 條件的 OIP/sectionList/sLT/articles/articlesTuple/authors/author，分別轉換成*元素路徑編碼*，和*元素值路徑編碼*，如表 4.1 所示。

表 4.1：路徑表示法與元素路徑編碼、元素值路徑編碼對照表

	Where	return
路徑表示法	OIP/sectionList/sLT/articles/ articlesTuple/toArticle/title	OIP/sectionList/sLT/articles/ articlesTuple/authors/author
元素路徑編碼	< 0, 5, 6, 9, 10, 11, 12 >	< 0, 5, 6, 9, 10, 16, 17 >
元素值路徑編碼	W1：< 0, 5, 6, 9, 10, 11, 12 > W2：< 0, 5, 6, 9, 23, 24, 25 >	R1：< 0, 5, 6, 9, 10, 16, 17 > R2：< 0, 5, 6, 9, 10, 16, 19 > R3：< 0, 5, 6, 9, 10, 16, 21 > R4：< 0, 5, 6, 9, 23, 29, 30 >

由表 4.1 得知，一個路徑表示法可以對應到多個*元素值路徑編碼*，但並非所有都是合理的答案。例如利用上述的 XQuery 查詢句與圖 3.5，我們經由 where 條件的查詢值 (Editor's Notes.) 得知，符合條件的*元素值路徑編碼*應該是表 4.1 中的 W1，而 return 條件中符合 where 條件限制的查詢結果應該有三個，分別為

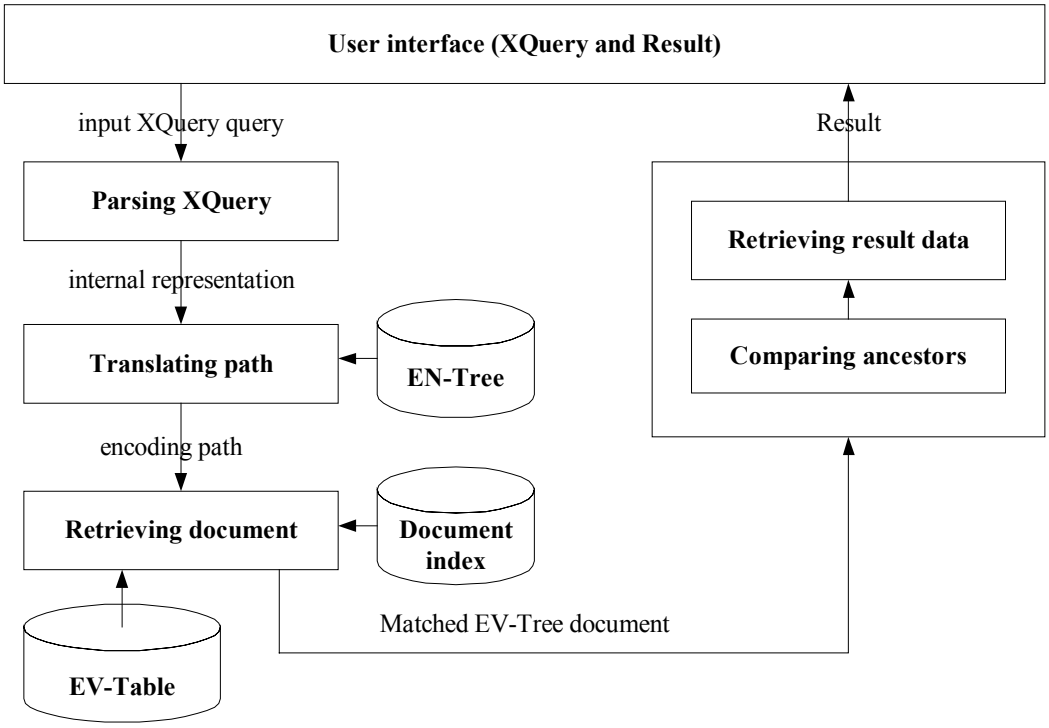

圖 4.1 查詢處理架構

表 4.1 中的 R1、R2 及 R3。這是因為這三個*元素值路徑編碼*和 W1 是屬於同一個 articlesTuple (*元素值節點編碼* 10)，而 W2 及 R4 則是屬於另一個 articlesTuple (*元素值節點編碼* 23)，是不正確的查詢結果。為了排除這些不正確的結果，我們提出了比較祖先關係的演算法。詳細的處理過程將會在 4.2 節中說明。■

接下來，我們將說明整個查詢處理的基本架構。在圖 4.1 中，首先由使用者輸入符合 XQuery 查詢語法的查詢句，經過系統中的查詢解析 (Parsing XQuery) 模組分析後，會將此查詢句以系統內部的表示法表示出來。接下來則是透過路徑轉換 (Translating path) 模組，利用元素編碼樹這個資料結構將所有的路徑表示法轉換成以單一數字表示的編碼，同時在每個編碼中標記其路徑表示法是否可以出現多個對應的*元素值路徑編碼*。再由文件擷取 (Retrieving document) 模組利用 where 條件的任意一個路徑表示法所對應之文件索引，在眾多元素值編碼表檔案中尋找符合此路徑表示法之查詢值的檔案。

然後在這些篩選過的元素值編碼表檔案中，找到符合查詢句要求的結果。若

查詢句的所有路徑表示法，並沒有任何一個被標記會出現多個對應的*元素值路徑編碼*時，則將這些符合查詢條件的元素值編碼表檔案交給查詢結果擷取 (Retrieving result data) 模組處理，而此模組則在每個元素值編碼表檔案中，依照查詢句中的要求將結果取出來回傳給使用者。若查詢句的路徑表示法中，有任一個被標記其路徑表示法會出現多個對應的*元素值路徑編碼*時，則將這些篩選過的元素值編碼表檔案交給祖先關係比較 (Comparing ancestors) 模組處理。此模組會比較所有*元素值路徑編碼*其祖先關係的正確性，如果正確則將這個元素值編碼表檔案交給查詢結果擷取模組處理，將結果取出來回傳給使用者，如果不正確則不輸出結果。關於這個部分詳細的演算法會在 4.2 節中說明。

4.2 查詢演算法的說明

我們設計的查詢演算法是引用在第三章中介紹的資料結構，包括元素編碼樹、元素值編碼表及文件索引，來達到針對 XML 資料利用路徑表示法快速查詢的目的。同時，為了解決相同的*元素名稱節點編碼*會在元素值編碼表中重複出現，而造成一個路徑表示法會對應到多個不同*元素值路徑編碼*的問題，我們在演算法裏提出了比較祖先關係的方法，關於這個方法的處理過程在本節中將有詳細的說明。在開始說明整個查詢演算法之前，我們先定義以下說明會用到的名詞。

【定義 4.3】對於一個*元素路徑編碼*$< X_{i1}, X_{i2}, X_{i3}\cdots X_{iL_i} >$，$X_{iL_i}$為路徑終點之*元素名稱節點編碼*，我們簡稱為***終點元素 (end-point)*** 。

[*範例 4.4*] 對於範例 4.1 裏之路徑，也就是<0, 5, 6, 9, 10, 11, 12>，其*終點元素*為 12。■

【定義 4.4】對於一個*元素值節點編碼*$< Y_{i1}, Y_{i2}, Y_{i3}\cdots Y_{iL_i} >$，$Y_{iL_i}$為路徑終點之*元素值節點編碼*，我們簡稱為***終點元素值 (end-point value)***。

[*範例 4.5*] 對於範例 4.2 裏之路徑，也就是<0, 5, 6, 9, 10, 11, 12>，12 為*終點元素值*。■

值得注意的是，由於*元素路徑編碼*，和*元素值路徑編碼*，都是由根元素開始，所以由*終點元素*和*終點元素值*可以分別指認出 DTD 文件和 XML 文件中唯一的一條路徑。

我們所設計的查詢演算法如圖 4.2 所示，在這個演算法中我們假設可以有多個 where 條件但是只能有一個 return 條件，首先將使用者輸入查詢句中的路徑表示法利用元素編碼樹轉換為*終點元素*，若輸入的路徑表示法在元素編碼樹中對應到的路徑有星號（*）出現時，則將該*終點元素*（*nid*）的旗標參數（nid_{flag}）設為 TRUE，反之則設為 FALSE，其中星號（*）表示某路徑表示法會出現多個對應的*元素值路徑編碼*，需要作進一步的篩選。然後利用查詢句裏任意一個 where 條件的*終點元素*對應到的文件索引之根節點檔案名稱，及此 where 條件中的查詢值來呼叫 TraverseBTree (*BR*, *v_where*) 函式（如圖 4.3 所示），利用文件索引找出符合此 where 條件的所有元素值編碼表檔案。

接下來，則是在每個找到的元素值編碼表檔案中，比較每個 where 條件的*終點元素*及查詢值是否存在，然後檢查 return 條件的*終點元素*是否存在，若有一個不符合則放棄此元素值編碼表檔案；若全部符合且*終點元素*的旗標參數皆為 FALSE 時，則直接將這個元素值編碼表檔案中，對應到 return 條件的*終點元素*之元素值編碼列的*元素內容*輸出為結果；若全部符合且*終點元素*的旗標參數為 TRUE 時，則利用 Select_Path 函式（如圖 4.6 所示)按照*終點元素*節點編碼的大小順序，從小至大依序做如下的祖先關係比較：對於相鄰的兩個*終點元素*，利用 Compare_Ancestors 函式（如圖 4.8 所示），比較兩個*終點元素值*所對應的*元素值路徑編碼*其祖先關係的正確性，如果正確則將這個元素值編碼表檔案中，對應到 return 條件的*終點元素*之元素值編碼列的*元素內容*輸出為結果，如果不正確則放棄這個元素值編碼表檔案，至於祖先關係比較的必要性將在本節後面加以說明。值得一提的是整個處理的過程，每個元素值編碼表檔案只需被開啟一次。

Algorithm QueryProcessing

輸入：{ (*pe_where, v_where)*} /* XQuery 中 where 條件裏的路徑表示法和查詢值集合 */，
pe_return /* XQuery 中 return 條件裏的路徑表示法*/

輸出：{*v_result*} /* 符合查詢條件的結果 */

/*變數說明：

{ (*W_EP*, *v_where*) }：

為輸入中由 where 條件取得之一對變數的集合，其中 *W_EP* 為路徑表示法所轉換的*終點元素*，*v_where* 為查詢值。

R_EP：

為輸入中由 return 條件取得之一個變數，*R_EP* 為路徑表示法所轉換的*終點元素*。

nid_{flag}：

為*終點元素*之旗標參數，TRUE 代表輸入的路徑表示法在元素編碼樹中有星號 (*) 出現，FALSE 代表輸入的路徑表示法在元素編碼樹中沒有星號 (*) 出現。

{ *ev_table* }：

為元素值編碼表的集合。

BR：

為文件索引之根節點檔案名稱。

i：

每個元素值編碼列在元素值編碼表中的索引，亦為其*元素值節點編碼*。*/

BEGIN {Algorithm QueryProcessing}

Step 1/* 初值設定 */

設定所有的旗標參數 nid_{flag} =FALSE。

Step 2/* 將所有輸入的路徑表示法轉為*終點元素* */

利用元素編碼樹將所有輸入的路徑表示法 *pe_where* 及 *pe_return* 轉為 *W_EP* 及 *R_EP*，若輸入的路徑表示法在元素編碼樹中有星號 (*) 出現時，則把該 *W_EP* 或 *R_EP* 的 nid_{flag} 設為 TRUE /* 星號 (*) 代表一個路徑表示法會對應到多個*元素值路徑編碼* */。

Step 3/* 找到文件索引的根節點 */

針對任意一個輸入的路徑 (*W_EP*, *v_where*) ，利用元素編碼樹找到所對應到的 *BR*。

Step 4/* 在元素值編碼表中將正確的查詢結果找出來 */

Step 4.1 依據 *BR* 呼叫 TraverseBTree (*BR*, *v_where*) 得到{ *ev_table* }/* *ev_table* 代表符合一個 where 條件的元素值編碼表 */。

Step 4.2 由{ *ev_table* }依序取得一個 *ev_table*，並在 *ev_table* 中檢查{ (*W_EP*, *v_where*, *i*) }中的每個(*W_EP*, *v_where*)是否符合及 *R_EP* 是否存在。

Step 4.2.1 檢查每一個 (*W_EP*, *v_where*) ：若 (*W_EP*, *v_where*) 存在於 *ev_table* 中，則根據其*元素值節點編碼*得到 (*W_EP*, *v_where*, *i*)，若 (*W_EP*, *v_where*) 不存在於 *ev_table* 中，則跳到 Step 4.2 /* 不必檢查 *R_EP* 是否存在 */。

Step 4.2.2 檢查 *R_EP*：若 *R_EP* 存在於 *ev_table* 中，則根據其*元素值節點編碼*及*元素內容*得到 (*R_EP*, *v_result*, *i*) ，若 *R_EP* 不存在於 *ev_table* 中，則跳到 Step 4.2。

Step 4.2.3 若{ (*W_EP*, *v_where*, *i*) }和(*R_EP*, *v_result*, *i*)中 *W_EP* 和 *R_EP* 的 nid_{flag} = FALSE，則直接將*元素內容 v_result* 印出來為查詢結果。

Step 4.2.4 若{ (*W_EP*, *v_where*, *i*) }中的 *W_EP* 或(*R_EP*, *v_result*, *i*)中的 *R_EP* 之 nid_{flag} = TRUE，則取{ (*W_EP*, *i*) }及(*R_EP*, *v_result*, *i*)，呼叫 Select_Path，把由 Select_Path 傳回來之正確的*元素內容v_result* 印出來為查詢結果 /* 利用 Select_Path 依序選出欲比較祖先關係的*終點元素*交由 Compare_Ancestors 利用*終點元素值*來比較祖先關係的正確性 */。

END {Algorithm QueryProcessing}

圖 4.2：QueryProcessing 演算法

圖 4.3 為利用文件索引找出符合要求的元素值編碼表檔案之函式。該函式的輸入資料為查詢句經元素編碼樹轉換之後，任意一個 where 條件的*終點元素*之查詢值與其對應到文件索引之根節點檔案名稱。在函式中首先會判斷開啟的根節點檔案會對應到文件索引的內部節點還是外部節點，若為內部節點，則比較查詢值和查詢值的大小，繼續走到樹的下一層，也就是開啟另一個文件索引節點檔案，若為外部節點，則循序搜尋此外部節點中的內容，找到與輸入相符的索引值，傳回對應的元素值編碼表檔案名稱為輸出結果。

Subroutine TraverseBTree (*BR*, *v_where*)
輸入：*BR* /* 文件索引之根節點檔案名稱 */，
　　v_where /* 查詢值 */
輸出：{ *ev_table* } /* 符合查詢條件的元素值編碼表 */
/*變數說明：
bpt：
　　一個用來儲存文件索引節點的資料結構，存放如圖 3.9 中所有的資料。
vc：
　　在 *bpt* 中最接近 *v_where* 的查詢值。
*/
BEGIN {Subroutine of TraverseBTree }
　　Step 1/*載入文件索引之根節點檔案 */
　　載入檔案 *BR*，並把它儲存在 *bpt* 中。
　　Step 2/* 搜尋 (traverse) 文件索引找出符合查詢條件的元素值編碼表 */
　　判斷檔案 *BR* 為文件索引中的內部節點或外部節點。
　　Step 2.1 若為外部節點，則循序搜尋 *bpt* 中的查詢值。
　　Step 2.1.1 若找到查詢值與 *v_where* 相符合，則傳回該查詢值的內容 (值)，即為元素值編碼表檔案名稱 /* 查詢值如圖 3.9 L4 行中的 Office，元素值編碼表檔案名稱如圖 3.9 L4 行中的 SIGRd1 */。
　　Step 2.1.2 若找不到查詢值與 *v_where* 相符合，則回傳空集合。
　　Step 2.2 若為內部節點，則循序比較 *bpt* 中的查詢值，找到最接近 *v_where* 的查詢值 *vc*。
　　Step 2.2.1 若 *v_where* < *vc*，則把前一個查詢值之指標指到的文件索引節點檔案名稱與 *v_where* 當做參數遞迴呼叫 TraverseBTree。
　　Step 2.2.2 若 *v_where* >= *vc*，則把此查詢值之指標指到的文件索引節點檔案名稱與 *v_where* 當做參數遞迴呼叫 TraverseBTree。
　　Step 2.2.4 若 *vc* 為最後一個查詢值，則以最後一個指標指到的文件索引節點檔案名稱與 *v_where* 當做參數遞迴呼叫 TraverseBTree。
　　Step 2.3 若不是內部節點也不是外部節點時，則傳回空集合。
END {Subroutine of TraverseBTree}

圖 4.3：TraverseBTree 函式

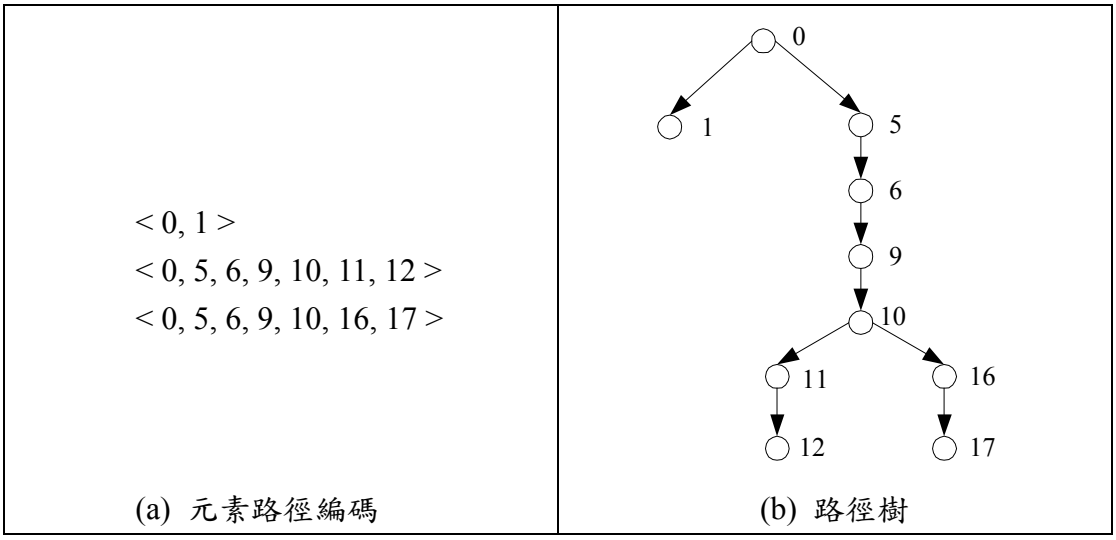

圖 4.4：元素路徑編碼與路徑樹對照圖

接下來說明在眾多符合個別查詢條件的*元素值路徑編碼*中，如何進行祖先關係的比較，以便組合出正確的答案。在開始說明此函式之前，我們先定義以下名詞。

【定義 4.5】假設有兩條*元素路徑編碼*，第 1 條*元素路徑編碼*有 L_1 個編碼，而第 2 條*元素路徑編碼*有 L_2 個編碼。令 X_{mn} 表示第 m 條*元素路徑編碼*中之第 n 個編碼 ($m \in \{1, 2\},\ 1 \le n \le L_m$)。假設對於此兩條*元素路徑編碼*，$\exists l$ 使得 $X_{1l} = X_{2l} = X$ 同時 $X_{1(l+1)}$!= $X_{2(l+1)}$，則稱 X 為此兩*元素路徑編碼*的***終止祖先(terminating_ancestor)***。

[*範例 4.6*] 以表 4.1 中的兩條*元素路徑編碼* < 0, 5, 6, 9, 10, 11, 12 > 與 < 0, 5, 6, 9, 10, 16, 17 >為例，10 即為此兩條*元素路徑編碼*的*終止祖先*。■

【定義 4.6】假設有 k 條*元素路徑編碼*，若是任兩條*元素路徑編碼*依據定義 4.5 找到其*終止祖先*，則所有*終止祖先*所得之集合，稱為***終止祖先集合(terminating_ancestor set)***。

【定理 4.1】假設有 k 條*元素路徑編碼*，將其按照*終點元素*節點編碼的大小排順序，然後由小到大依序比較相鄰兩條*元素路徑編碼*，依據定義 4.5 找到其終止祖先，則可以得到正確的*終止祖先集合*。

為了便於證明定理 4.1，我們將此 k 條*元素路徑編碼*以樹狀結構來表示。在這裏我們以三條*元素路徑編碼*為例，如圖 4.4(a)中的三條*元素路徑編碼*可以用圖 4.4(b)的樹狀結構來表示。正式的定義如下。

【定義 4.7】令一條*元素路徑編碼*的組成編碼依序對應到一條路徑的每個節點，每個節點的名稱並以所對應編碼為名。由多條*元素路徑編碼*所對應到的多條路徑可以構成一個單一根節點的樹狀結構，我們稱為***路徑樹 (Path Tree)***。

觀察圖 4.4 (b)，路徑樹中的三條路徑由左到右依序以 P1、P2、P3 來表示，根據定義 4.6 得知，我們分別找 (P1, P2)，(P1, P3) 和 (P2, P3) 的終止祖先，而得到*終止祖先集合*為 { 0, 10 } ；若依 P1、P2、P3 的順序，也就是依*終點元素*編碼前序的順序，依序考慮相鄰的兩個*元素路徑編碼*，也就是只找(P1, P2) 和 (P2, P3)的終止祖先，亦可以找到*終止祖先集合* { 0, 10 } ；但是若依 P2、P1、P3 的順序，則找到的*終止祖先集合*為 { 0 }。由上述的例子我們發現，依 P2、P1、P3 的順序時，由於不是按照*終點元素*編碼前序的順序來尋找*終止祖先集合*，則有可能產生錯誤的*終止祖先集合*。

依照路徑樹的定義，我們可以將定理 4.1 改寫如下：

【定理 4.1-1】將一個*元素路徑編碼*的集合以路徑樹來表示，在路徑樹中由左到右 (也就是依據*終點元素*前序編碼的順序) 依序考慮相鄰的兩個*元素路徑編碼*，則會找到正確的*終止祖先集合*。

【證明】以反證法，假設有一個正確的*終止祖先*名叫 T，是經由兩條不相鄰的*元素路徑編碼*得出，且無法從任何兩條相鄰的*元素路徑編碼*得出。假設 X_1 和 X_2 是所有可以找出 T 中最接近的兩條不相鄰的路徑，分別為 X_1：$< X_{11}, X_{12}, \cdots X_{1L_1} >$，$X_2$：$< X_{21}, X_{22}, \cdots X_{2L_2} >$，且 $X_{1m} = X_{2m} = T$，$X_{1(m+1)} < X_{2(m+1)}$。

由於此兩條路徑不相鄰，所以在此路徑樹中，必定存在一個節點 X_n，$X_{1(m+1)} < X_n < X_{2(m+1)}$，使得路徑 X_3：$< X_{11}, X_{12}, \cdots X_{1m}, \cdots, X_n >$介於 X_1 和 X_2 中間。X_n 有三種狀況：

1. 亦為 X_m 之子節點：

考慮 X_1 和 X_3 兩條路徑，我們發現此兩條路徑的前 m 個元素相同，而第 m+1 個元素，亦即 $X_{1(m+1)}$和 X_n 不同，也就是 X_1 和 X_3 也可找出*終止祖先* T。同樣 X_2 和 X_3 之*終止祖先*亦為 T。

2. 屬於 $X_{1(m+1)}$之子孫：

假設是 X_{1p} 之子節點，也就是 X_1 和 X_3 之*終止祖先*為 X_{1p}，但 X_2 和 X_3 之*終止祖先*為 T。

3. 屬於 $X_{2(m+1)}$之子孫：

假設是 X_{2q} 之子節點，也就是 X_2 和 X_3 之*終止祖先*為 X_{2q}，但 X_1 和 X_3 之*終止祖先*為 T。

由上面三例，皆違反 X_1 和 X_2 是所有可以找出 T 中最接近的兩條不相鄰的路徑的假設，故得證。

【定義 4.8】假設有兩條欲比較祖先關係的*元素值路徑編碼*，第 1 條*元素值路徑編碼*有 L_1 個編碼，而第 2 條*元素值路徑編碼*有 L_2 個編碼。令 Y_{tu} 表示第 t 條*元素值路徑編碼*中第 u 個編碼($t \in \{1, 2\},\ 1 \le u \le L_t$)。假設對於此兩條*元素值路徑編碼*，$\exists v$ 使得 $Y_{1v} = Y_{2v} = Y$ 同時 $Y_{1(v+1)}$!= $Y_{2(v+1)}$，又 Y 所對應到的*元素名稱節點編碼*為*終止祖先*，則稱 Y 為此兩*元素值路徑編碼*的***正確祖先(correct_ancestor)***。

[*範例 4.7*] 根據圖 3.5，以表 4.1 中的兩條*元素值路徑編碼* < 0, 5, 6, 9, 10, 11, 12 > 與 < 0, 5, 6, 9, 10, 16, 19 > 為例，10 為相同的最大值，由圖 3.5 知 10 對應到的*元素名稱節點編碼*亦為 10。又兩條*元素值路徑編碼*對應到的*元素路徑編碼*為< 0, 5, 6, 9, 10, 11, 12 > 與 < 0, 5, 6, 9, 10, 16, 17 >，10 為此兩條*元素路徑編碼*的*終止祖先*，由上述可知 10 即為此兩條*元素值路徑編碼*的*正確祖先*。■

【定義 4.9】假設有 s 條欲比較祖先關係的*元素值路徑編碼*，可以對應到 t 個*元素路徑編碼*，若是按照其對應的*元素路徑編碼*分成 t 個不同的集合，然後依*終點元素*編碼的大小順序從每個不同的集合中選出一條*元素值路徑編碼*，共選出 t 條*元素值路徑編碼*，若任兩條*元素值路徑編碼*根據定義 4.8 都能找到*正確祖先*，則

表 4.2：元素路徑編碼與元素值路徑編碼對照表

元素路徑編碼	元素值路徑編碼
P1：< 0, 1 >	Pv1：< 0, 1 >
P2：< 0, 5, 6, 9, 10, 11, 12 >	Pv2：< 0, 5, 6, 9, 10, 11, 12 > Pv3：< 0, 5, 6, 9, 23, 24, 25 >
P3：< 0, 5, 6, 9, 10, 16, 17 >	Pv4：< 0, 5, 6, 9, 10, 16, 17 > Pv5：< 0, 5, 6, 9, 10, 16, 19 > Pv6：< 0, 5, 6, 9, 10, 16, 21 > Pv7：< 0, 5, 6, 9, 23, 29, 30 >

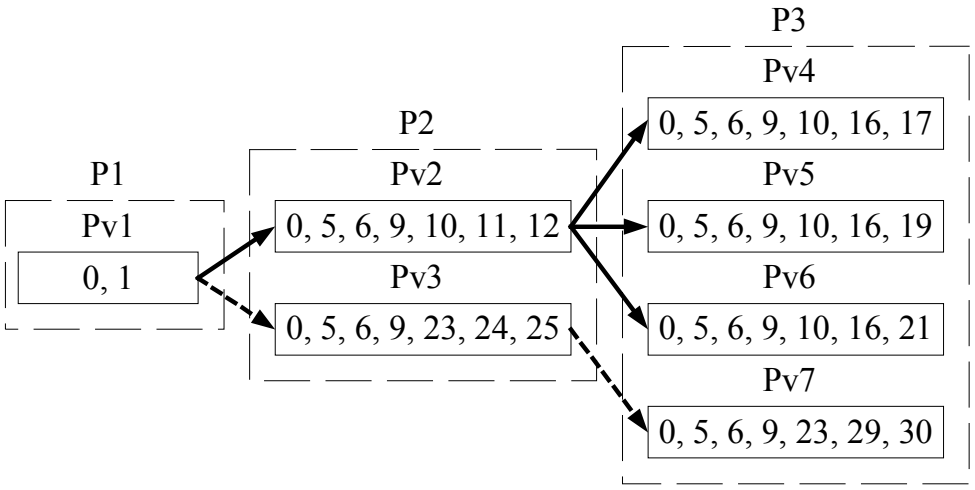

圖 4.5：正確路徑集合圖例

這 *t* 條*元素值路徑編碼*稱為一組***正確路徑集合***。

[*範例 4.8*]我們以圖 4.4 中的三條*元素路徑編碼*為例，由圖 3.5 知 P1、P2、P3 所對應到的*元素值路徑編碼*如表 4.2 所示。我們把表 4.2 中的這 7 條*元素值路徑編碼*（對應到定義 4.9 的 *s*）按照其對應的*元素路徑編碼* P1、P2、P3 分成 3 組（對應到定義 4.9 的 *t*），如圖 4.5 所示，每組找一條來循序比較祖先關係，共選出 3 條*元素值路徑編碼*，若任兩條*元素值路徑編碼*根據定義 4.8 都能找到*正確祖先*，則可以找到*正確路徑集合*。如圖中以實線箭頭連接的路徑集合就是一組*正確路徑集合*，以虛線箭頭連接的路徑集合則是另一組*正確路徑集合*，而 Pv1、Pv3、Pv6 不會有*正確祖先*，所以不是一組*正確路徑集合*。

本節後面處理祖先關係的演算法，就是依循這個方法找到*正確路徑集合*，然

後根據查詢條件將正確的查詢結果找出來。■

處理祖先關係的演算法，包含 Select_Path 函式，Find_TA，和 Compare_Ancestors 函式。

Select_Path 函式的輸入資料包含了 where 條件裏路徑表示法對應的*終點元素*，及其符合條件之*元素值節點編碼*的集合；另外也包含了 return 條件裏路徑表示法對應的*終點元素*、回傳值及*元素值節點編碼*的集合。此函式會將上述 where 條件的*元素值節點編碼*依其對應的*終點元素*以不同的鏈結串列儲存下來，然後將 return 條件的*元素值節點編碼*及回傳值也依其對應的*終點元素*以不同的鏈結串列儲存下來，並根據*終點元素*的大小對所有的鏈結串列排序。此函式會以第一條鏈結串列為基準，將第一條鏈結串列的每個節點做為輸入，呼叫 reSelect 函式，最後取得所有查詢結果。

reSelect 函式會以遞迴的方式，按照每條鏈結串列所對應之*終點元素*的大小順序，由小到大在每條鏈結串列中依序取出一個節點，然後以取出的順序每兩個利用圖 4.8 的 Compare_Ancestors 函式來找出是否具有*正確祖先*，如果具有*正確祖先*，則繼續取出下一個鏈結串列的一個節點與最後取出的節點做祖先關係的比較，依此類推直到做完所有鏈結串列，則將符合 return 條件的*終點元素*所對應的元素值編碼列之*元素內容*回傳。如果不具有*正確祖先*，則在同一個鏈結串列中取下一個節點，直到找出具有*正確祖先*的節點，再繼續取出下一個鏈結串列的一個節點，若在同一個鏈結串列中無法找到具有*正確祖先*的節點，表示沒有正確的查詢結果。

在利用 Compare_Ancestors 函式來找出是否具有*正確祖先*之前，必須先利用圖 4.9 的 Find_TA 函式找出該兩個*終點元素*所對應的之*終止祖先*，以做為比較祖先關係的終止節點。否則因為所有路徑必定有一個共同的祖先，也就是根節點，而*終止祖先*則是兩條路徑中具有*正確祖先*的節點，所以必須先確定*終止祖先*，才能知道所找到的祖先是否為合理的共同祖先。

在找到*終止祖先*之後，我們利用 Compare_Ancestors 函式來比較兩個與不同

*終點元素*配對的*終點元素值*，所對應的*元素值路徑編碼*之間是否擁有*正確祖先*；如果有，表示欲比較祖先關係的兩條*元素值路徑編碼*為合理的路徑，必能找出符合查詢條件的正確答案；如果沒有，表示兩條*元素值路徑編碼*不是合理的路徑，找到的查詢結果為錯誤答案。

Subroutine Select_Path

輸入：{ (*W_EP*, *i*) } /* where 條件的*終點元素及元素值節點編碼*的集合 */，
{ (*R_EP*, *v_result*, *i*) } /* return 條件的*終點元素與元素內容及元素值節點編碼*的集合*/

輸出：{*RValue_Set*} /* 正確的查詢結果 */

/*變數說明：

LL_{where_m}：
用來儲存 where 條件中第 *m* 個*終點元素*所有對應的*元素值節點編碼*的鏈結串列。

m：
表示共有 *m* 個不同*終點元素*之 where 條件的鏈結串列。

LL_{return}：
用來儲存符合 return 條件中的*終點元素，所有的元素值節點編碼與元素內容*的鏈結串列。

LL_j：
所有的鏈結串列皆以 *LL* 來表示，包含了 LL_{where_m} 及 LL_{return} /* *j* 代表第 *j* 個鏈結串列，所有鏈結串列依照*終點元素* (*W_EP* 及 *R_EP*) 來排序，*j* 值愈小表示此鏈結串列的*終點元素*愈小 */。

$Lindex_j$：
鏈結串列 *LL* 中第 *j* 個鏈結串列的索引。

{*RValue_Set*}：
表示查詢結果的集合。

CA_{EP}：
由此函式選出之欲比較祖先關係的*終點元素* (*元素名稱節點編碼*) 。

*/

BEGIN {Subroutine of Select_Path}
Step 1 /* 將輸入的資料存入鏈結串列中 */
Step 1.1 建立 LL_{where_m} 以儲存每個 *W_EP* 和 *i* 的對應。
Step 1.2 建立 LL_{return} 來儲存每個 *R_EP* 的 *v_result* 與 *i*。
Step 2 /* 將所有的鏈結串列排序 */
將所有的鏈結串列依*終點元素*的大小排序，排序後的鏈結串列以 *LL* 來表示。
Step 3 /* 初值設定 */
設定所有鏈結串列的索引 *Lindex* 為 0。
設定{*RValue_Set*} 為 NULL。
Step 4 /* 以 LL_1 為基準，將 LL_1 的每個節點 (node) 做為輸入，呼叫 reSelect 函式取得回傳值 */
取出鏈結串列 LL_1 於 $Lindex_1$ 位置之*終點元素及元素值節點編碼*，得到 (CA_{EP}, *i*) 。
Step 4.1 若 LL_1 屬於鏈結串列 LL_{return}，則將((CA_{EP}, *i*)、LL_2、*v_result*、{*RValue_Set*})做為輸入，呼叫 reSelect 函式得到{*RValue_Set*}，$Lindex_1 = Lindex_1+1$，跳到 Step 4 直到 LL_1 做完。
Step 4.2 若 LL_1 屬於鏈結串列 LL_{where_m}，則將((CA_{EP}, *i*)、LL_2、NULL、{*RValue_Set* }) 做為輸入，呼叫 reSelect 函式，$Lindex_1 = Lindex_1+1$，跳到 Step 4 直到 LL_1 做完。
Step5 回傳{*RValue_Set*}。
END {Subroutine of Select_Path }

圖 4.6：Select_Path 函式

Subroutine reSelect

輸入：(CA_{EP}, i) /* 節點的*終點元素*及其對應之*元素值節點編碼* (*終點元素值*) */，
LL_j /* 所有的鏈結串列 */，
RValue /* 在遞回比較的過程中有碰到 LL_{return} ，則將 *v_result* 暫存在此變數*/，
{ *RValue_Set* } /* 在遞回比較的過程中所有符合條件的回傳值 */

輸出：{ *Rvalue_Set* } /* 在遞回比較的過程結束後所有符合條件的回傳值 */

/*變數說明：

j：

代表第 *j* 個鏈結串列。

(CA_{EP_n}, $i_{\_n}$)：

下一個鏈結串列節點的*終點元素*及其對應之*元素值節點編碼*。

*/

BEGIN {Subroutine of reSelect}

Step 1/* 初值設定 */

設定 $j = 2$。

Step 2/*將輸入的節點(CA_{EP}, i)跟下一個鏈結串列的每個節點(CA_{EP_n}, $i_{\_n}$)做祖先關係比較，如果正確，再將符合的(CA_{EP_n}, $i_{\_n}$)做為輸入，遞迴呼叫 reSelect 函式跟下一個鏈結串列比較*/

將鏈結串列 LL_j 中的每個節點(CA_{EP_n}, $i_{\_n}$)與傳入的節點(CA_{EP}, i)做為輸入，呼叫 Compare_Ancestors 函式。

Step2.1 若傳回 TRUE 且 LL_j 屬於鏈結串列 LL_{return}，則將 *v_result* 暫存在 *RValue*。

Step2.1.1 /* *LL* 做完，因為共有 m+1 個鏈結串列(where 有 m 個而 return 有 1 個) */

若 $j = m+1$，則{*RValue_Set*} = {*RValue_Set* }+ *RValue*。

Step2.1.2 /* *LL* 未做完*/

若 $j < m + 1$，則 $j = j + 1$，將((CA_{EP}, i)、LL_{j+1}、*RValue*、{*RValue_Set*}) 做為輸入，遞迴呼叫 reSelect 函式回傳{*RValue_Set*}。

END {Subroutine of reSelect}

圖 4.7：reSelect 函式

Subroutine Compare_Ancestors

輸入：(CA_{EP}, i) /* *終點元素及其對應之元素值節點編碼 (終點元素值)* */，
(CA_{EP_n}, i_n) /* *終點元素及其對應之元素值節點編碼 (終點元素值)* */

輸出：TRUE 或 FALSE /* TRUE 代表具有正確的祖先節點，FALSE 代表不具有相同的祖先節點 */

/*變數說明：

{ (g_{1_f}, h_{1_f}) }：
表示 CA_{EP} 的第 f 個祖先節點，g_{1_f} *為元素名稱節點編碼*，h_{1_f} *為元素值節點編碼*，f 值較小表示此節點較接近節點 CA_{EP} /*當 f = 0 時表示為 CA_{EP} 節點，當 f = 1 時表示為 CA_{EP} 的第一層祖先節點，其它依此類推 */。

{ (g_{2_j}, h_{2_j}) }：
表示 CA_{EP_n} 的第 j 個祖先節點，g_{2_j} *為元素名稱節點編碼*，h_{2_j} *為元素值節點編碼*，j 值較小表示此節點較接近節點 CA_{EP_n} /* 當 j = 0 時表示為 CA_{EP_n} 節點，當 j = 1 時表示為 CA_{EP_n} 的第一層祖先節點，其它依此類推 */。

terminating_ancestor：
表示 CA_{EP} *和* CA_{EP_n} 的兩個*終點元素*的*終止祖先*節點，亦可做為比較祖先關係的終點節點。

*/

BEGIN {Subroutine of Compare_Ancestors }
Step 1 /* 首先在元素編碼樹中找到 *terminating_ancestor* */
呼叫 Find_TA 將{ g_{1_f} }和 { g_{2_j} }做為輸入，取得 *terminating_ancestor*。
Step 2 /* 初值設定 */
設定 f = 0。
設定 j = 0。
Step 3 /* 比較兩*終點元素*是否具有相同祖先節點 */
Step 3.1 若 $h_{1\_f} = h_{2\_j}$，且 $g_{1\_f} = g_{2\_j}$ = *terminating_ancestor*，則傳回 TRUE。
Step 3.2 若 $h_{1\_f} = h_{2\_j}$，但 g_{1_f} 和 g_{2_j} 不等於 *terminating_ancestor*，則傳回 FALSE。
Step 3.3 若 $h_{1\_f} > h_{2\_j}$，則 $f = f + 1$，跳到 Step 3.1。
Step 3.4 若 $h_{1\_f} < h_{2\_j}$，則 $j = j + 1$，跳到 Step 3.1。
END {Subroutine of Compare_Ancestors}

圖 4.8：Compare_Ancestors 函式

Subroutine Find_TA

輸入：{ g_{1_f} }，{ g_{2_j} } /* 兩個*終點元素*及其祖先節點之集合 */，

輸出：*terminating_ancestor* /* 如定義 4.5 所示 */

BEGIN {Subroutine of Find_TA}
Step 1 /* 初值設定 */
設定 f = 0。
設定 j = 0。
設定 *terminating_ancestor* = 0。
Step 2 /* 找到*元素名稱節點編碼*相同的節點 */
Step 2.1 若 $g_{1\_f} = g_{2\_j}$，則 *terminating_ancestor* = g_{1_f}。
Step 2.2 若 $g_{1\_f} > g_{2\_j}$，則 $f = f + 1$，跳到 Step 2。
Step 2.3 若 $g_{1\_f} < g_{2\_j}$，則 $j = j + 1$，跳到 Step 2。
END {Subroutine of Find_TA}

圖 4.9：Find_TA 函式

4.3 查詢演算法範例

在前節中我們已經詳述了查詢的演算法，在此節中我們舉兩個例子加以說明。我們設計的查詢演算法其處理過程大致可以分成以下四個主要步驟：<1>利用元素編碼樹將路徑表示法轉換成*終點元素*，以便將冗長的路徑表示法轉換成單一數字表示的*終點元素*，並記錄查詢時所需要的資訊。<2>利用文件索引篩選出符合部分查詢條件的元素值編碼表檔案，以節省必須針對每個檔案做檢查的時間。<3>利用*終點元素*從元素值編碼表檔案取出符合查詢條件的結果。這個步驟是將由文件索引找到的元素值編碼表檔案一一開啟，利用查詢句中所有的*終點元素*及 where 條件的查詢值找到符合查詢要求的結果，並依需要做祖先關係的比較，來比較兩個*終點元素值*在元素值編碼樹中是否具有*正確祖先*，以避免因*元素名稱節點編碼*相同而取出不符合限制條件之結果。

為了便於說明，我們將各步驟簡述如下：

《步驟 1》 利用元素編碼樹將路徑表示法轉換成*終點元素*

《步驟 1.1》首先將使用者輸入查詢句中的路徑表示法利用元素編碼樹轉換為*終點元素*。

《步驟 1.2》 若輸入的路徑表示法在元素編碼樹中有星號 (*) 出現時，則以旗標參數將該*終點元素*標記為 TRUE，反之則標記為 FALSE。

《步驟 1.3》 若輸入的路徑表示法為查詢句中的 where 條件，則同時記下此路徑對應到的文件索引根節點檔案名稱。

《步驟 2》 利用文件索引找出符合查詢條件的元素值編碼表檔案

《步驟 2.1》首先找出 where 條件中任意一個*終點元素*及其在查詢句中的查詢值。

《步驟 2.2》 利用步驟 1.3 的文件索引根節點檔案名稱及步驟 2.1 的查詢值利用文件索引找出符合查詢條件的所有元素值編碼表檔案。

《步驟 3》 利用*終點元素*取出符合查詢條件的結果

《步驟 3.1》 首先一一開啟由文件索引找到之元素值編碼表檔案，利用*終點元素*在元素值編碼表檔案中循序比對。

《步驟 3.2》 若所有 where 條件的之查詢值皆符合，且*終點元素*的旗標參數皆為 FALSE，則將 return 條件的*終點元素*所對應到的值傳回為結果。若有任一個 where 條件的查詢值不符合，則放棄此元素值編碼表檔案，繼續處理下一個元素值編碼表檔案。

《步驟 3.3》 若所有 where 條件的查詢值皆符合，且任一*終點元素*的旗標參數為 TRUE，則做祖先關係的比較。若祖先關係正確則將 return 條件的*終點元素*所對應到的值傳回為結果，若不正確則放棄此元素值編碼表檔案，繼續處理下一個元素值編碼表檔案。若有任一個 where 條件的*終點元素*之查詢值不符合，則放棄此元素值編碼表檔案，繼續處理下一個元素值編碼表檔案。

我們將在以下分別舉兩個例子詳細說明。假設所有符合圖 2.2 DTD 的 XML 文件，所轉換之元素值編碼表檔案，擺在網址"http://dblab.cs.ntou.edu.tw/"的地方，而使用者針對這些檔案做查詢。

[範例 4.9]不需比較祖先關係的查詢：

如果我們要找出第 28 冊的論文集是在哪一年出版的，其查詢句如下所示。

```
<OIP>
{
    for $o in document ( "http://dblab.cs.ntou.edu.tw/" ) /OIP
    where contains ( $o/volume/text ( ) , " 28 " )
    return
            $o/year
```

}

</OIP>

在查詢句中 where 條件裏的完整路徑表示法為 OIP/volume = 28，而 return 條件裏的完整路徑表示法為 OIP/year，使用圖 3.1 的元素編碼樹配合文件索引，整個查詢的步驟如下所示。

《步驟 1》 利用元素編碼樹將路徑表示法轉換成*終點元素*

(一) 指定欲轉換的路徑表示法為查詢句中 where 條件的 OIP/volume，則轉換的過程如下。

<1>將 OIP/volume 這個路徑表示法利用元素編碼樹轉換成*終點元素*2。

<2>判斷 OIP/volume 在元素編碼樹中有沒有星號 (*) 出現，此例沒有，所以將*終點元素*2 的旗標參數標記為 FALSE。

<3>因為 OIP/volume 為查詢句中的 where 條件，所以同時記下此路徑對應到的文件索引根節點檔案名稱 SIGRd1.bt。

(二) 欲轉換的路徑表示法為查詢句中 return 條件的 OIP/year，則轉換的過程如下。

<1>將 OIP/year 這個路徑表示法利用元素編碼樹轉換成*終點元素*4。

<2>判斷 OIP/year 在元素編碼樹中有沒有星號 (*) 出現，此例沒有，所以將*終點元素*4 的旗標參數標記為 FALSE。

<3>因為 OIP/year 為查詢句中的 return 條件，所以不必記下此路徑對應到的文件索引根節點檔案名稱。

《步驟 2》 利用文件索引找出符合查詢條件的元素值編碼表檔案

使用在步驟 1(一)得到的文件索引根節點檔案名稱 SIGRd1.bt 和查詢句中 where 條件的查詢值 28，搜尋 (traverse) 文件索引，得到所有符合查詢條件的元素值編碼表檔案，如 SIGRd1.evt。

《步驟 3》 利用*終點元素*取出符合查詢條件的結果

使用得到的元素值編碼表檔案 SIGRd1.evt，配合在步驟 1(一)得到的*終點元素* 2 及其查詢值 28 與在步驟 1(二)得到的*終點元素* 4，則搜尋的過程如下。

<1>開啟元素值編碼表檔案 SIGRd1.evt，在 SIGRd1.evt 中循序比對，找到*終點元素* 2 且其查詢值為 28，繼續找到*終點元素* 4。

<2>檢查所有*終點元素*之旗標參數皆為 FALSE，所以將*終點元素* 4 其對應到的值傳回為結果，也就是 1999。

應該注意的是，由於此範例中所有*終點元素*之旗標參數皆為 FALSE，所以不必做祖先關係的比較。■

[範例 4.10]需要比較祖先關係的查詢：

這類查詢是因為相同的節點編碼會出現在多個不同的元素值編碼列，而造成一個路徑表示法會對應到多個不同的*元素值路徑編碼*，為了解決這個因節點編碼相同而取出不符合查詢要求之結果的問題，我們提出比較祖先關係的方法，此方法將在步驟 4 中說明。假設我們將範例 4.9 查詢句中的 return 條件改為 OIP/sectionList/sLT/sectionName，則此查詢句即成為需要比較祖先關係的查詢，

《步驟 1》 利用元素編碼樹將路徑表示法轉換成*終點元素*

使用圖 3.1 的元素編碼樹資料，並指定欲轉換的路徑表示法為查詢句中 return 條件的 OIP/sectionList/sLT/sectionName，則轉換的過程如下。

<1>將 OIP/sectionList/sLT/sectionName 這個路徑表示法利用元素編碼樹轉換成*終點元素* 7。

<2>判斷 OIP/sectionList/sLT/sectionName 在元素編碼樹中有沒有星號 (*) 出現，此例有，所以將*終點元素* 7 的旗標參數標記為 TRUE。

<3>因為 OIP/sectionList/sLT/sectionName 為查詢句中的 return 條件，所以不必記下此路徑對應到的文件索引根節點檔案名稱。

《步驟 2》 利用文件索引找出符合查詢條件的元素值編碼表檔案

同範例 4.9 的步驟 2，得到元素值編碼表檔案，如 SIGRd1.evt。

《步驟 3》 利用*終點元素*取出符合查詢條件的結果

使用得到的元素值編碼表檔案 SIGRd1.evt，配合範例 4.9 的步驟 1(一)得到的*終點元素* 2 及其查詢值 28 與本範例的步驟 1 得到的*終點元素* 7，則搜尋的過程如下。

<1>開啟元素值編碼表檔案 SIGRd1.evt，在 SIGRd1.evt 中循序比對，找到*終點元素* 2 且其查詢值為 28，繼續找到*終點元素* 7。

<2>檢查所有*終點元素*之旗標參數，發現*終點元素* 7 為 TRUE，所以要做祖先關係的比較。依序比較*終點元素* 2 及*終點元素* 7 對應到的*終點元素值*之祖先關係，得到其*正確祖先為元素值節點編碼* 0，表示其祖先關係正確，將*終點元素* 7 對應到的值傳回為結果，也就是 Articles。

■

第五章　效率評估

在本章中，我們設計數個實驗來評估所提出的資料結構及查詢演算法之效能。我們以 Borland C 5.0 實做演算法程式，以個人電腦做為實驗環境，而為了因應實驗比較對象的需求，所以我們採用不同的作業系統與硬體配備，細節將在各節中說明。實驗中所用來測試的資料，是取自於 ACM SIGMOD Record 中屬於 Ordinary Issue Page 的 XML 文件，一共有 51 個檔案，所有檔案的大小合計有 268KB，而實驗數據的收集是根據各種不同路徑表示法的查詢所得到的時間。為了避免程式執行時有不確定因素影響到實驗結果，我們都是在固定的系統環境下，個別進行 100 次的實驗，取得平均值。在 5.1 節中，我們將說明實驗時所使用的 4 個查詢句範本，於 5.2 節中，說明所提出的方法與傳統關聯式資料庫的實驗評估結果，其中包含了正確性的驗證及查詢效能的比較，最後在 5.3 節中，說明與史丹佛 (Stanford) 大學提出的 Lore 所做的實驗評估結果比較。

5.1 查詢句範本

在本節中，首先我們將說明為了評估所提出的資料結構及查詢演算法之效能，所使用的 8 個查詢句範本，我們以 XQuery 的語法來表示前 4 個查詢句。

查詢句一：找出第 28 冊第 1 號的論文集是在哪一年出版的。

```
L1  <Result>
L2  {
L3      for $o in document("http://dblab.cs.ntou.edu.tw")/OIP
L4      where $o/volume = "28" and $o/number= "1"
L5      return {$o/year}
L6  }
L7  </Result>
```

在查詢句一中，所使用的路徑表示法都不必做祖先關係的比較，也就是此查詢句中任一個路徑表示法只會對應到唯一的*元素值路徑編碼*，所以不會造成找出不符合查詢條件的結果，是本論文的實驗中，查詢處理過程最簡單的查詢句。而

以下查詢句二到查詢句四則用以測出本論文中祖先關係比較的演算法是否正確，更可以與查詢句一比較，得知比較祖先關係的方法之效率如何。

查詢句二：找出文章名稱為 Lore : A Database Management System for Semistructured Data.的作者。

```
L1  <Result>
L2  {
L3      for $a in document("http://dblab.cs.ntou.edu.tw")//articlesTuple
L4      where $a/toArticle/title = " Lore : A Database Management System for
L5                                   Semistructured Data."
L6      return {$a/authors/author}
L7  }
L8  </Result>
```

在查詢句二中，所使用的路徑表示法都會對應到多個*元素值路徑編碼*，所以查詢過程中必須做祖先關係的比較，以過濾掉不符合查詢條件的結果。

查詢句三：找出第 28 冊第 1 號的會議論文集中所有文章的名稱。

```
L1  <Result>
L2  {
L3      for $o in document("http://dblab.cs.ntou.edu.tw")/OIP
L4      where $o/ volume = "28" and $o/ number= "1"
L5      return{$o/sectionList/sLT/articles/articlesTuple/toArticle/title}
L6  }
L7  </Result>
```

查詢句四：找出名稱為 Lore : A Database Management System for Semistructured Data.的文章出現在論文集的哪一冊。

```
L1  <Result>
L2  {
L3      for $o in document("http://dblab.cs.ntou.edu.tw")/OIP
L4      where $o/sectionList/sLT/articles/articlesTuple/toArticle/title = "Lore : A
L5              Database Management System for Semistructured Data."
L6      return{$o/volume}
L7  }
L8  </Result>
```

表 5.1：各查詢句與其相關資料對照表

查詢句	一	二	三	四	五	六	七	八
查詢結果個數	1	5	23	1	1	15	7	17
開啟檔案個數	1	1	1	1	3	15	23	49

在查詢句三中，只有 return 條件中的路徑表示法會對應到多個*元素值路徑編碼*；而查詢句四中，只有 where 條件中的路徑表示法會對應到多個*元素值路徑編碼*。雖然此兩個查詢句並非所有路徑表示法都會對應到多個*元素值路徑編碼*，而為了過濾掉不符合查詢條件的查詢結果，所以在查詢過程中所有路徑表示法都必須做祖先關係的比較。

在所有欲查詢的 XML 文件中，從查詢句一到查詢句四的查詢條件中，都只有一份符合的文件，也就是只需要開啟一個對應的元素值編碼表檔案。為了瞭解查詢時開啟的檔案個數是否會影響整個查詢的時間，我們改寫查詢句四，設計了下列四個查詢句。

查詢句五：找出 1998 年出版的論文集是哪幾冊。

查詢句六：找出論文集的哪幾冊有第 3 號。

查詢句七：找出名稱為 Editor's Notes.的文章出現在論文集的哪幾冊。

查詢句八：找出章節名稱 (sectionName) 為 Articles 的論文集是哪幾冊。

在查詢句五到查詢句八中，每個查詢句在查詢過程中所必須開啟的元素值編碼表檔案個數依序增加，由這 4 個查詢句我們可以測出查詢時間是否與開啟的檔案個數有關係。

表 5.1 為各查詢句的輸出元素個數和查詢過程中必須開啟的元素值編碼表檔案個數之對照表，至於利用所有查詢句得到的實驗結果將在 5.2 節及 5.3 節中說明。

5.2 與傳統關聯式資料庫的效能比較

在與傳統關聯式資料庫做比較的實驗中，我們選擇微軟的 Microsoft SQL Server 2000 資料庫做為評估效能的根據，而該實驗的測試環境為個人電腦，該電腦的中央處理器是 PentiumIII650，搭配 256MB 的記憶體(RAM)，採用 Windows 2000 Server 的作業系統。

本實驗可分為兩個子項目，實驗一是用來驗證我們所設計的資料結構及查詢演算法針對 XML 文件查詢所得到的結果是否正確。實驗二則是用來比較我們所提出的方法與 Microsoft SQL Server 2000 所支援的路徑表示法的查詢（XPath）之執行效率。

『實驗一』我們以實驗一來驗證所設計的資料結構及查詢演算法針對 XML 文件查詢所得到的結果是否正確。在實驗一中，我們將欲查詢的 XML 文件中所有含內容值的元素名稱及屬性名稱（需配合所屬的元素名稱命名，如：元素名稱_屬性名稱），做為關聯式資料庫中欄位的名稱，建立一個名為 OIP 的表格，再將對應的值紀錄到表格中，最後得到一個 12 個欄位 1294 個資料列的大表格。

值得注意的是，上述的方法雖然可以確保資料的正確性，但是卻同時記錄了許多重複的資料，會造成不必要的空間浪費，並不是一個理想的方法。而我們為了查詢上述的 OIP 表格，所以將 5.1 節中的 4 個 XQuery 查詢句改寫成 4 個對應之 SQL 語法的查詢句，如下所示。

查詢句一（SQL）：

```
L1  select distinct year
L2  from OIP
L3  where volume =28 and number= 1
```

查詢句二（SQL）：

```
L1  select author
L2  from OIP
L3  where title = 'Lore : A Database Management System for Semistructured Data.'
```

查詢句三（SQL）：

```
L1  select distinct title
L2  from OIP
L3  where volume = 28 and number= 1
```

查詢句四（SQL）：

```
L1  select distinct volume
L2  from OIP
L3  where title = 'Lore : A Database Management System for Semistructured Data.'
```

實驗的結果證實，利用上述關聯式資料庫的資料與查詢句所得到的查詢結果，與我們所設計的資料結構及查詢演算法所得到的結果完全吻合，由此也驗證了本論文做法之正確性。

『實驗二』我們以實驗二來比較我們所提出的方法與 Microsoft SQL Server 2000 所支援的路徑表示法的查詢（XPath）之執行效率。由於 Microsoft SQL Server 2000 一次只能處理一份 XML 文件，所以此實驗中每個查詢直接針對含有特定資料的單一 XML 文件做查詢。另一方面，我們也將文件索引在我們的方法中所花費的時間扣除，只比較處理 XML 資料所花費的時間。同時，在這個實驗中每個查詢所花費的整體時間都包含了結果輸出的時間。

在 Microsoft SQL Server 2000 提供的數個查詢 XML 資料的方法中，我們利用 OpenXML 函式直接回傳包含 XML 文件資料的資料列集，如此可避免 Client 端處理資料或網路傳輸的時間。圖 5.1 就是將查詢句一改寫成以 OpenXML 函式取得 XML 文件資料的例子。

在圖 5.1 L1 行中宣告的參數@iTree 是用來指定表示 XML 資料的內部節點樹，這個參數包含了經由 L6 行 sp_xml_preparedocument 預存程序回傳的值，也就是利用該預存程序去讀取 XML 資料並檢驗其有效性，然後建立一棵節點樹。L3 行到 L5 行是將 XML 資料指定給 L2 行宣告的參數@xmlDOC，該資料即為欲處理的 XML 資料，在此只列出部分做為代表。

```
L1  DECLARE @iTree INTEGER
L2  DECLARE @xmlDOC VARCHAR(8000)
L3  SET @xmlDOC =
L4  '<OIP><volume>28</volume><number>1</number><month>March</month><
L5  year>1999</year><sectionList>……</OIP>'
L6  EXEC sp_xml_preparedocument @iTree OUTPUT, @xmlDOC
L7  SELECT * FROM
L8     OPENXML(@iTree, '/OIP[volume = 28][number= 1]', 2)
L9     WITH (year text)
L10 EXEC sp_xml_removedocument @iTree
```

圖 5.1：查詢句一（OpenXML 函式）

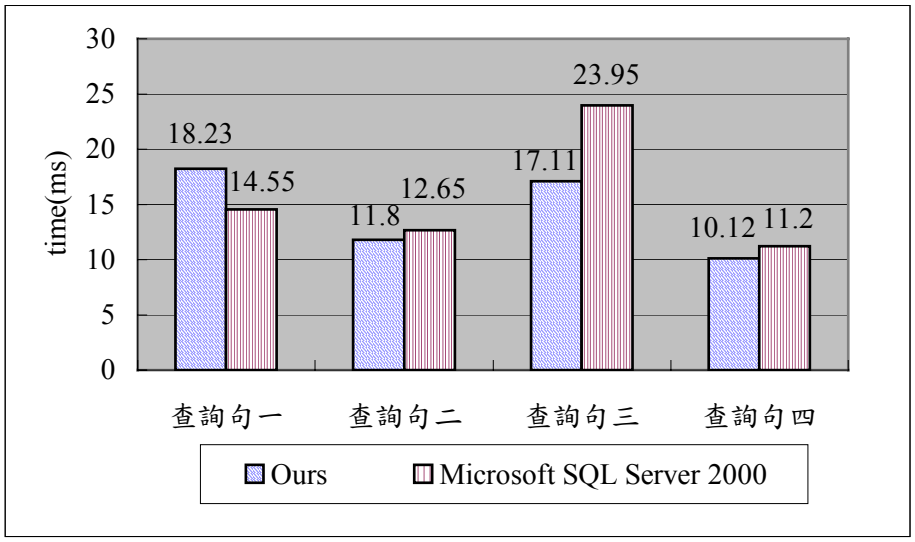

圖 5.2：我們的方法與 Microsoft SQL Server 2000 的執行效率比較圖

在節點樹建立後，就可以使用 L8 行的 OpenXML 函式找尋符合條件的 XML 資料。OpenXML 函式的第一個參數為欲查詢資料之節點樹，第二個參數為 XPath 運算式，用來決定節點樹開始搜尋的節點，和一個限制回傳資料行的標準運算式，概念上與 SQL 語法的 FROM 和 WHERE 子句相同。第三個參數則告訴 SQL Server 以屬性、子元素或兩者為條件而搜尋，此例中的 2 代表以子元素為條件而搜尋。L9 行的 WITH 子句是用來指明將被回傳的資料行，並定義資料列集的結構，概念上與 SQL 語法的 SELECT 子句類似。最後，使用 L10 行的 sp_xml_removedocument 預存程序將節點樹所使用的記憶體收回。

圖 5.2 顯示了這兩種方法的執行效率。圖中以 4 個查詢句為 X 軸，查詢所需

時間為 Y 軸，以毫秒 (ms) 為單位。由圖中可以發現，除了查詢句一外，採用本論文中的方法都有較佳的執行效率。這是因為我們將 XML 文件以元素值編碼表檔案儲存下來，並提出了快速解決祖先關係的演算法，而 Microsoft SQL Server 2000 卻需要將 XML 資料建成一棵節點樹之後再處理，顯然需要花費較多的時間。另一方面，我們的方法必須花費額外的時間來開啟一個元素值編碼表檔案，Microsoft SQL Server 2000 並不需要開啟任何檔案。若是將我們的方法扣除了開啟檔案的時間，則這四個查詢句的平均查詢時間都在 2.84ms 以下。所以，與 Microsoft SQL Server 2000 整體的查詢時間相比，我們的方法確實有非常好的執行效率。

由圖 5.2 中同時發現，我們的方法與 Microsoft SQL Server 2000 的查詢時間都會隨著查詢句中的路徑表示法個數而增加，如查詢句一與查詢句三比其他兩個查詢句多了一個路徑表示法，其查詢時間就比其他兩個查詢句長。而 Microsoft SQL Server 2000 的查詢時間還會因查詢所得的元素個數而變化，如查詢句三中的查詢結果個數較其他查詢句多，所以其查詢時間也隨之增加。

值得注意的是，使用我們的方法時，查詢句一不需要做祖先關係的比較，而查詢句二到查詢句四需要做祖先關係的比較。由實驗結果得知，查詢句一的查詢時間與查詢句二到查詢句四的查詢時間並沒有顯著的差距，由此可知本論文中比較祖先關係的方法之效率非常好。

5.3 與 Lore 的效能比較

由於實驗二主要只是測試在單一 XML 文件中找尋正確資料的效率，為了更進一步瞭解我們系統的整體效率，我們選擇與史丹佛 (Stanford) 大學提出的 Lore 系統做比較。此實驗的測試環境為個人電腦，該電腦的中央處理器是 PentiumII400，搭配 128MB 的記憶體(RAM)，採用的作業系統為 Linux Mandrake release 7.0 (Air) + CLE v0.9 (Yami) Kernel 2.2.14-15mdk on an i686。

表 5.2：資料結構所需儲存空間對照表

	元素編碼樹	文件索引	元素值編碼表	總計
原始 XML 文件	無	無	無	268KB
OURS	260 bytes	92.7 KB	119 KB	211.96KB
Lore	無	無	無	1750.10KB
Lore (Lindex、Vindex)	無	無	無	2176.50KB

表 5.2 為本論文中用來儲存實驗測試資料的所有資料結構各自所需之儲存空間，與 Lore 系統處理相同 XML 資料所需之儲存空間對照表。表中的第二列為 Lore 沒有建立任何索引結構時所需之儲存空間，第三列則為 Lore 有建立 Lindex 及 Vindex 時所需之儲存空間。

由表中可以明顯的發現本論文中使用的所有資料結構之儲存空間，不但比原始 XML 文件所需之儲存空間 268KB 小，也比 Lore 所需之儲存空間小很多。

此節包含三個子項目，實驗三是用來比較針對特定的單一 XML 文件，我們所提出的方法與 Lore 之執行效率。實驗四和實驗五則是用來比較雙方都使用索引結構處理眾多 XML 資料的情況下何者的執行效率較佳。本節中，所有實驗的每個查詢所花費之整體時間都不包含結果輸出的時間。

『實驗三』實驗三與實驗二類似。在 Lore 的資料庫中，我們是依據每個查詢句的需求，一次只匯入一個 XML 文件做為查詢的資料。而我們的方法中則扣掉文件索引的影響，只考慮處理單一 EV-Tree 檔案的時間。

在本論文的所有實驗裏，我們的方法中所使用到之文件索引的 order 都是 128，而文件索引的高度則是隨著查詢值的多寡而變化。根據我們的實驗，查詢句一與查詢句三使用相同的文件索引，其高度為 1，而搜尋此文件索引平均需要花費 0.28 ms；查詢句二與查詢句四使用相同的文件索引，其高度為 2，而搜尋此文件索引平均需要花費 1.12 ms。

在開始比較執行效率之前，我們首先介紹史丹佛大學為 Lore 設計的查詢語言，也就是 Lorel。圖 5.3 就是將查詢句四改寫成以 Lorel 語法表示的例子。

L1	select OIP.volume
L2	from OIP.sectionList.sLT.articles.articlesTuple.toArticle.title X
L3	where concat(X) = "Lore: A Database Management System for Semistructured
L4	Data."

圖 5.3：查詢句四 (Lorel)

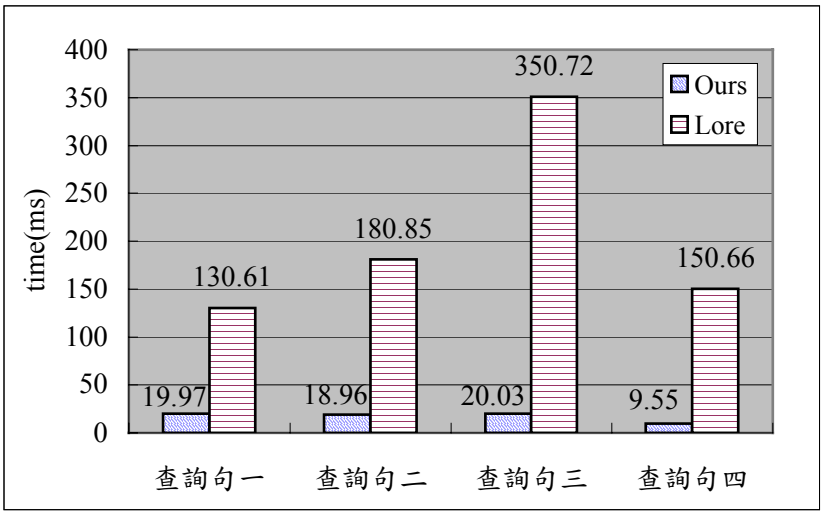

圖 5.4：我們的方法與 Lore 的執行效率比較圖 (單一 XML 文件)

Lorel 的語法是由 OQL 的語法所延伸出來的，與 SQL 語法非常相似，也是包含了 select、from、where 三個子句。在圖 5.3 L1 行的 select 子句是用來將符合 OIP/volume 這個路徑表示法的查詢結果回傳給使用者。值得一提的是，Lorel 將路徑表示法的元素與元素之間以(.)隔開，而 XPath 則是用(/)隔開。

L2 行的 from 子句是用來將路徑表示法 OIP.sectionList.sLT.articles.articles-Tuple.toArticle.title 下的內容以物件 X 來取代，Lore 會將各種資料型態轉換成相同的型態以方便做查詢值的比對，如 L3 行的 where 子句可以比較資料型態為字串的查詢值與物件 X 是否相同，其中 concat 是一個外部函式，用來將包在同一個元素之下的所有文字資料回傳。由於查詢句四中需要比對的查詢值過於冗長，所以需要使用 concat 這個外部函式。

圖 5.4 顯示了這兩種方法的執行效率。圖中以 4 個查詢句為 X 軸，查詢所需時間為 Y 軸，以毫秒 (ms) 為單位。由圖中可以發現，採用本論文中的方法所需的查詢時間都顯著較少，可見我們所提出的演算法有較佳的執行效率。這是因為

在 Lore 裏，XML 的資料對應到一個樹狀結構，而元素名稱都當做樹狀結構的路徑標籤（label）紀錄下來。當元素名稱重複出現在 XML 文件中時，樹狀結構裏則會有很多路徑標籤相同的路徑，在查詢的過程中就必須對每條可能的路徑做搜尋，而花費很多時間。另一方面，我們的方法是利用元素編碼樹將所有可能的路徑紀錄下來並加以編碼，所以處理查詢時直接針對相同編碼的路徑加以處理即可，而產生較好的效率。

我們也發現 Lore 的查詢時間會隨著所取出的 XML 元素多寡而變化，如查詢句三所找出來的查詢結果最多，所以其查詢時間也最長，而路徑表示法的層數也會影響其查詢時間，如查詢句一與查詢句四的查詢結果一樣多，但是查詢句四的路徑表示法之層數較深，其查詢時間也較長。同時元素編碼樹的大小也不會隨著 XML 文件大小而變化，而在 Lore 中樹狀結構的大小會隨著 XML 文件變化，若樹狀結構很大或是分支很多時，查詢時間也會跟著增加。

『實驗四』我們以此實驗來比較我們整個系統與 Lore 系統之執行效率。在實驗四裏，我們將所有 XML 資料匯入 Lore 的資料庫中，實驗數據的收集是根據查詢句一到查詢句八。

由於我們提出的方法是先利用查詢值找到所有可能符合條件的元素值路徑編碼之後，再利用祖先關係的比較來找出正確的資料，與 Lore 提出的幾個索引結構中，由下往上（bottom up）的執行策略較為接近，所以在這個實驗中，我們為 Lore 建立對應的 Lindex，並依據各查詢句中會使用到的元素名稱建立 Vindex，以達到由下往上的執行策略。其中 Lindex 也就是路徑標籤索引（label index），讓我們可以經由路徑標籤找到某節點的父親節點，而 Vindex 也就是查詢值索引（value index），讓我們可以經由路徑標籤找到符合限定條件的查詢值。

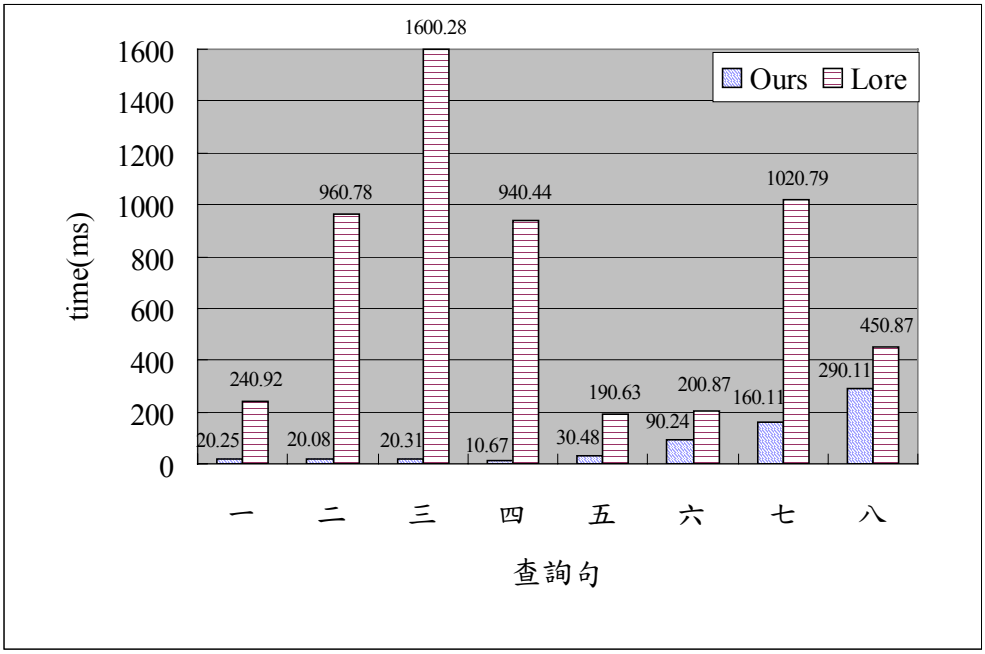

圖 5.5：我們的方法與 Lore 的執行效率比較圖

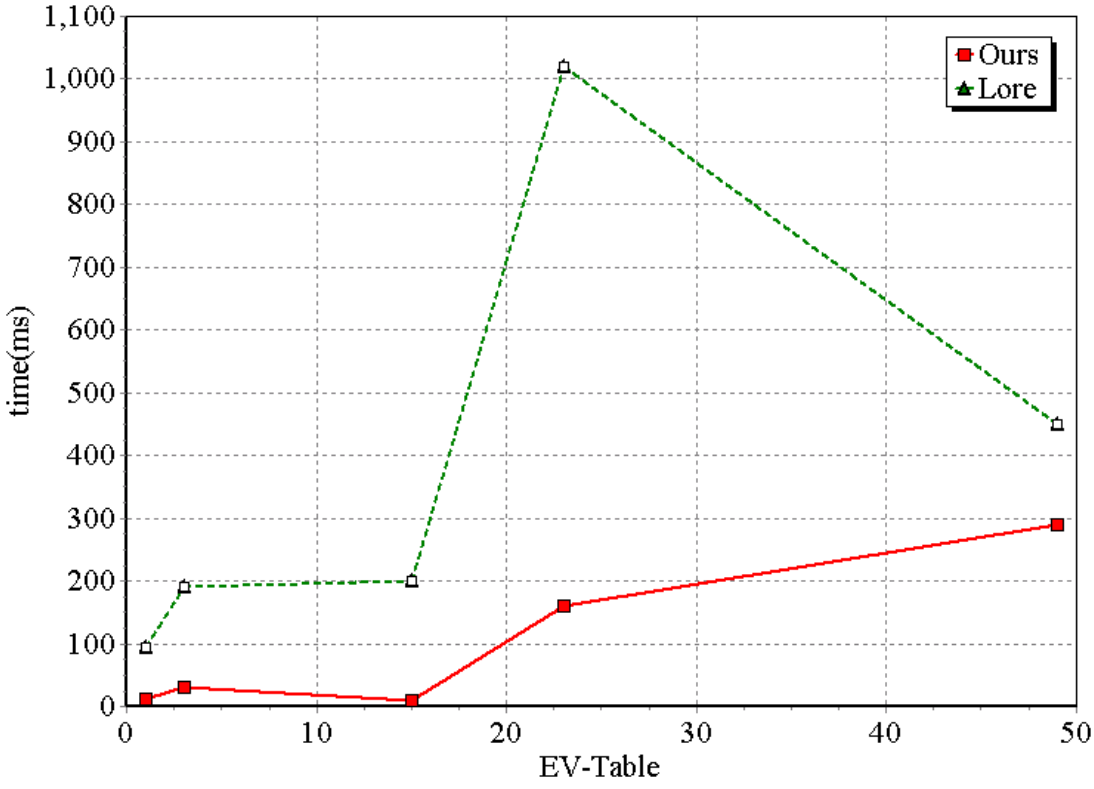

圖 5.6:我們的方法與 Lore 的執行效率比較圖（ 依我們方法中開啟的檔案個數 ）

圖 5.5 顯示了這兩種方法的執行效率。圖中以 8 個查詢句為 X 軸，查詢所需時間為 Y 軸，以毫秒 (ms) 為單位。由圖中可以發現，本論文中的方法有較佳的執行效率，但是我們方法的查詢時間會隨著查詢時所需開啟的元素值編碼表檔案而增加，由圖中的查詢句五到查詢句八可以明顯的看出來。

為了更清楚的看出查詢時間與查詢時所需開啟的元素值編碼表檔案個數之間的關係，我們將實驗結果以曲線圖的方式呈現，如圖 5.6 所示。

圖 5.6 為我們的方法在 51 個欲查詢的 XML 檔案中利用查詢句一到查詢句八，在查詢的過程中需要開啟的 EV-tree 檔案個數與查詢時間之對照圖，圖中以我們的方法在查詢時所需開啟檔案個數為 X 軸，查詢所需時間為 Y 軸，以毫秒 (ms) 為單位。同時我們也將 Lore 利用這八個查詢句得到的查詢時間拿來做比較。由於查詢句一到查詢句四，以我們的方法在查詢的過程中都只會開啟一個元素值編碼表檔案，所以我們取其查詢時間的平均值，做為查詢時需要開啟一個檔案之查詢所需時間。

由圖中我們發現，我們的方法所須之查詢時間，很明顯的隨著所需開啟的 EV-tree 檔案個數而增加，這是因為我們的方法只需要處理經文件索引過濾後符合查詢條件的元素值編碼表檔案。對於一般的文件查詢情況來講，符合限制條件的答案並不會出現在所有文件中，所以我們的方法適用於從大量的 XML 文件中，尋找其中一部分符合限制條件的 XML 資料。

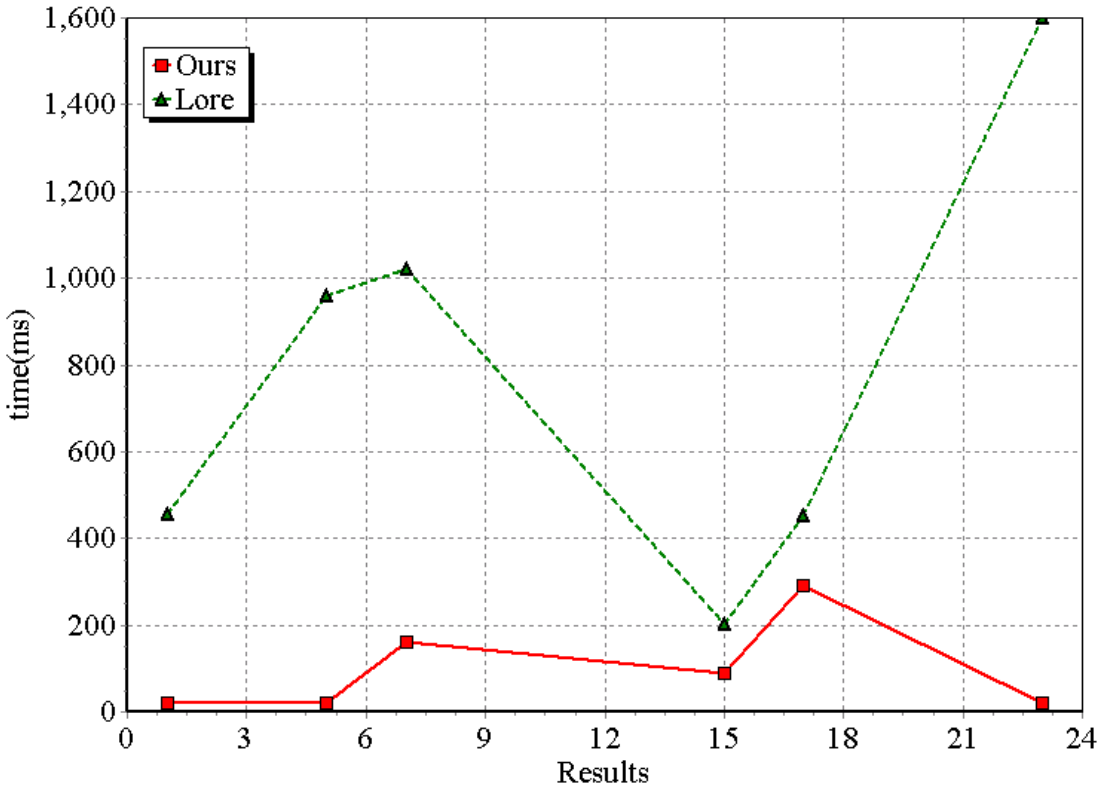

圖 5.7：我們的方法與 Lore 的執行效率比較圖 (依查詢結果個數)

圖 5.7 以查詢的結果個數為 X 軸，查詢所需時間為 Y 軸，以毫秒 (ms) 為單位。由於查詢句一、查詢句四與查詢句五都只會找到一個查詢結果，所以我們取其查詢時間的平均值，做為查詢時會找到一個查詢結果之查詢所需時間。

另一方面，Lore 的效能並不直接和所需開啟的元素值編碼表檔案有關，這是因為 Lore 已經將所有的資料整合在一個樹狀結構中，所以由圖 5.7 可看出，其效率與所取出的元素，也就是符合條件的路徑，有比較直接的關係。不過我們也觀察到 Lore 的查詢時間應該會隨著欲查詢的 XML 文件多寡而變化，也就是匯入 Lore 資料庫中的 XML 資料越多所需查詢時間也越長，我們以實驗五來證實這個說法。

『實驗五』我們以實驗五來證實 Lore 的查詢時間會隨著欲查詢的 XML 文件多寡而變化。在此實驗裏，我們依序將 10、20、30、40、50 個 XML 文件匯入 Lore 的資料庫中，並分別對這些資料建立索引結構。同時，為了減少路徑表示法對本

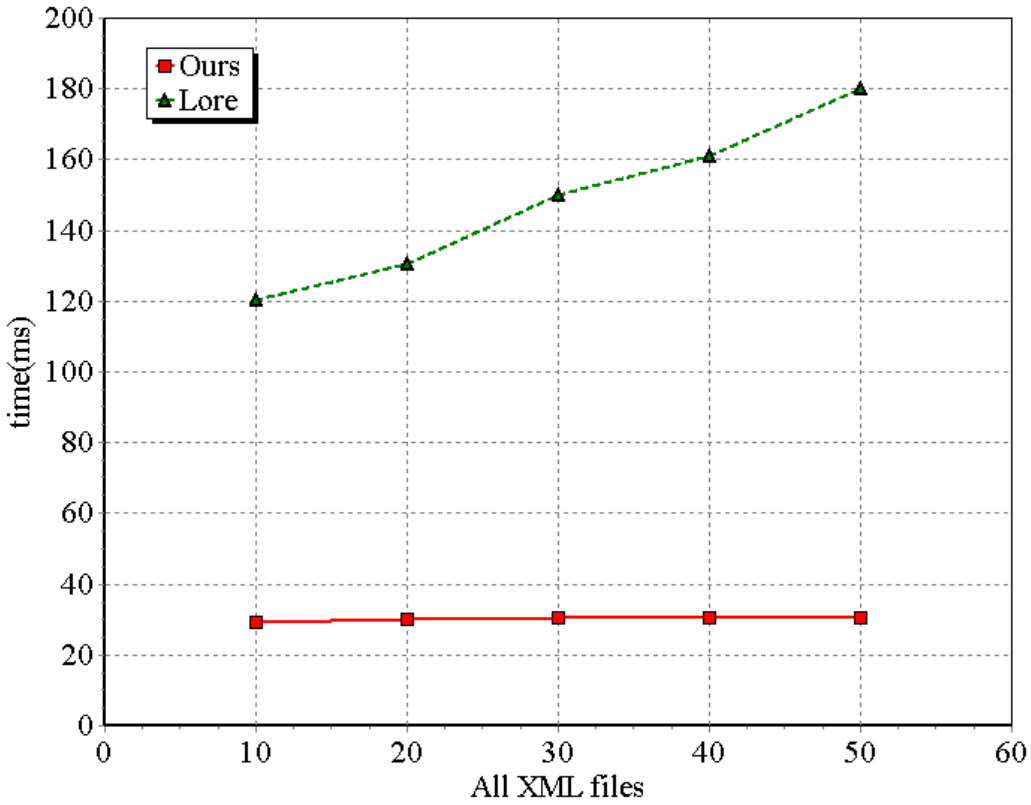

圖 5.8：查詢時間與欲查詢的 XML 文件個數對照圖

實驗的查詢時間所造成的影響，我們選擇路徑表示法的長度最短、個數最少的查詢句，也就是查詢句五，作為實驗數據收集的根據。

圖 5.8 顯示了查詢時間與欲查詢的 XML 文件個數之關係。圖中以欲查詢的 XML 文件個數為 X 軸，查詢所需時間為 Y 軸，以毫秒 (ms) 為單位。由圖中我們發現 Lore 的查詢時間確實與欲查詢的 XML 文件個數有直接的關係，所以當欲查詢的 XML 文件很大量時，Lore 的查詢效率就會降低，而我們的方法顯然與欲查詢的 XML 文件個數沒有直接的關係。

『實驗六』我們以此實驗來驗證路徑表示法的長度會影響 Lore 的查詢效率。為了更清楚地瞭解路徑表示法的長度與 Lore 查詢效率之間的關係，在實驗四裏，我們依據不同長度的路徑表示法設計了以下四個查詢句：

查詢句九：找出第 28 冊的論文集是哪一年出版的。

查詢句十：找出章節名稱（sectionName）為 Articles 的論文集是哪一年出版的。

表 5.3：查詢句與路徑表示法長度對照表

查詢句	九	十	十一	十二
路徑表示法長度	2	4	6	7

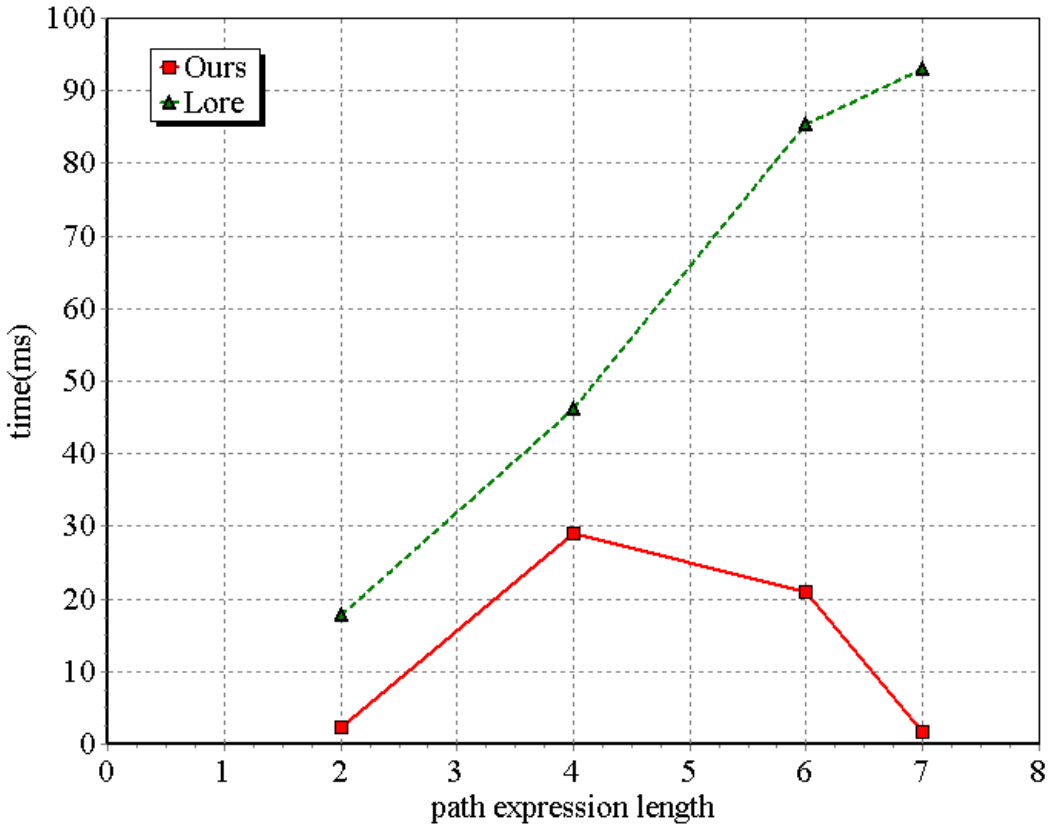

圖 5.9：查詢時間與路徑表示法長度對照圖

查詢句十一：找出擁有起始頁數為 2 的文章之論文集是哪一年出版的。

查詢句十二：找出文章名稱為 Lore : A Database Management System for Semistructured Data.的論文集是哪一年出版的。

在查詢句九到查詢句十二中，我們固定查詢句中 return 條件的路徑表示法 (找出論文集是哪一年出版的) ，只變動 where 條件的路徑表示法，而各查詢句中 where 條件的路徑表示法長度如表 5.3 所示。

圖 5.9 顯示了查詢時間與路徑表示法長度之間的關係。圖中以路徑表示法長度為 X 軸，查詢所需時間為 Y 軸，以毫秒 (ms) 為單位。由圖中我們發現 Lore 的查詢時間會受到路徑表示法長度的影響，當路徑表示法長度增加時，Lore 的

查詢時間也會顯著的增加，而我們的方法受到路徑表示法長度的影響顯然並不大。

第六章　結論與未來方向

在本論文中，我們提出三個資料結構及一系列的查詢演算法，來解決利用任意長度的路徑表示法，快速查詢一份或多份 XML 文件的資料之問題；且針對路徑表示法中的元素，可能對應到 XML 文件中多個元素值的情況加以處理。我們設計的資料結構，包括元素編碼樹、元素值編碼表及文件索引。元素編碼樹是用來紀錄 XML 文件內所有可能出現的路徑表示法，並加以編碼，以便從文件中快速地找出符合查詢條件之路徑表示法。元素值編碼表是針對單一 XML 文件，將元素的資料記錄下來，同時保留文件內元素間的祖孫關係，以避免查詢過程中 XML 文件解析（parse）時所花的額外時間。文件索引則可以協助我們從多份元素值編碼表檔案中快速找到所需要的檔案。而查詢演算法則是利用這三個資料結構找出符合路徑表示法的答案，然後利用祖先關係的比較，來驗證找出的答案是否為使用者要求的查詢結果。

由實驗結果顯示，我們的方法確實能夠正確地找出查詢結果。同時對於任意長度的路徑表示法的查詢，與 Microsoft SQL Server 2000 關聯式資料庫及史丹佛大學提出的 Lore 比較，皆有較佳的執行效率。

本論文未來研究的方向，可以針對 XQuery 的其他表示法詳加探討，提出對於 XML 查詢處理的適當設計，如針對多份 XML 文件的 join 設計其索引結構，最後整合對各個 XQuery 結構所設計的資料結構，完成對 XQuery 查詢處理的設計。

參考文獻

[1] S. Abiteboul, D. Quass, J. McHugh, J. Widom, J. Wiener, The Lorel Query Language for Semistructured Data, International Journal on Digital Libraries, volume 1, number 1, pages: 68 – 88, 1997.

[2] Ashraf Aboulnaga, Alaa R. Alameldeen, Jeffrey F. Naughton, Estimating the Selectivity of XML Path Expressions for Internet Scale Applications, Twenty-Seventh International Conference on Very Large Databases, 2001.

[3] Vincent Aguilera, Sophie Cluet, Pierangelo Veltri, Dan Vodislav, Fanny Wattez, Querying XML Documents in Xyleme, ACM SIGIR 2000 Workshop On XML and Information Retrieval, 2000.

[4] K. Bohm, On extending the XML engine with query-processing capabilities, In Proceedings of IEEE Advances in Digital Libraries, pages: 127 –138, 2000.

[5] Ya-Hui Chang, Ben-Hsu Chen, Chun-Chieh Wu, The Design and Implementation of an Efficient Library Management System by XML, Journal of Internet Technology, volume 2, number 4, 2001.

[6] Chung-Min Chen, N. Stoffel; M. Post, C. Basu, D. Bassu, C. Behrens, Telcordia LSI Engine: implementation and scalability issues, Eleventh International Workshop on Proceedings of Research Issues in Data Engineering, pages: 51 –58, 2001.

[7] V. Christophides, S. Cluet and J. Simeon, On Wrapping Query Languages and Efficient XML Integration, In Proceedings of ACM SIGMOD Conference on Management of Data, pages: 141 – 152, 2000.

[8] Sophie Cluet, Pierangelo Veltri, Dan Vodislav, Views in a Large Scale XML Repository, Twenty-Seventh International Conference on Very Large Databases, 2001.

[9] Brian F. Cooper, Neal Sample, Michael J. Franklin, Gísli R. Hjaltason, Moshe Shadmon, A fast index for semistructured data, Twenty-Seventh International Conference on Very Large Databases, 2001.

[10] D. Florescu, D. Kossmann, A Performance Evaluation of Alternative Mapping Schemes for Storing XML Data in a Relational Database, Rapport de Recherche No. 3680, INRIA, Rocquencourt, France, May, 1999.

[11] T. Grabs, K. Bohm, H.-J. Schek, Scalable distributed query and update service implementations for XML document elements, Eleventh International Workshop on Proceedings of Research Issues in Data Engineering, pages : 35 – 42, 2001.

[12] H. Kato, K. Oyama, M. Yoshikawa, S. Uemura, A query optimization for XML document views constructed by aggregations, Proceedings of 1999 International Symposium on Database Applications in Non-Traditional Environments, pages: 189 –196, 1999.

[13] E. Kotsakis, K. Bohm, XML Schema Directory: A data structure for XML data processing, Proceedings of the First International Conference on Web Information Systems Engineering, volume 1, pages: 62 –69, 2000.

[14] Quanzhong Li, Bongki Moon, Indexing and Querying XML Data for Regular Path Expressions, Twenty-Seventh International Conference on Very Large Databases, 2001.

[15] J. McHugh, J. Widom, Query optimization for semistructured data. Technical report, Stanford University Database Group, February, 1999.

[16] J. McHugh, J. Widom, Query Optimization for XML. Twenty-Fifth International Conference on Very Large Databases, 1999.

[17] Torsten Schlieder, Felix Naumann, Freie Universitat Berlin, Humboldt Universitat zu Berlin, Approximate Tree Embedding for Querying XML Data, ACM SIGIR 2000 Workshop On XML and Information Retrieval, 2000.

[18] J. Widom, Data Management for XML : Research Directions, IEEE Data Engineering Bulletin, volume 22, number 3, 1999.

[19] 張雅惠，吳俊頡，謝璨隆，查詢多份 XML 文件的資料結構設計，廿一世紀數位生活與網際網路科技研討會論文集，2001。

[20] 張雅惠，吳俊頡，謝璨隆，支援 XML 文件查詢的索引結構，中華民國九十年全國計算機會議論文集，2001。