支援 XML 文件查詢的索引結構

張雅惠 吳俊頡 謝璨隆
國立台灣海洋大學資訊科學系
202 基隆市北寧路 2 號
yahui@cyber.cs.ntou.edu.tw

摘要

近年來，在網際網路上從事商業行為，特別是處理企業與企業間大量資料的交換，成為一個非常重要的課題，而 XML (eXtensible Markup Language) 的提出受到很大的重視與廣泛的支持，我們預期未來將有很多企業將公司資料以 XML 格式存放。本篇論文討論管理眾多 XML 文件的方式，透過所設計的 B^+-Tree 與元素定位點兩種索引結構，可提高資料查詢的速度。我們並做了多項實驗，測試這兩種資料結構個別的效率，而且討論同時使用這兩種結構來查詢時所需的時間。

關鍵字：全球資訊網，XML 技術，查詢處理

一、簡介

近年來，在網際網路 (Internet) 上從事商業行為，也就是所謂的電子商務，正蓬勃發展中，而當前電子商務一個很受重視的課題，便是處理企業與企業間 (B2B) 的資料交換，XML (eXtensible Markup Language) 的提出則加速了相關的發展。XML 是由 W3C (World Wide Web Consortium) 所訂定的一個描述 Web 文件結構的標準規範，在 1996 年 11 月有了最先的雛形，並於 1998 年 2 月確立 XML1.0 規格書。XML 為 SGML (Standard Generalized Markup Language) 的子集，是一個結構性的、以文字格式儲存的標註語言 (markup language)。XML 具有可擴充性，各個團體可以依 XML 的規範訂定符合自己需求的資料結構，而該團體的使用者依其規則建立出來的資料便具有一致的表示方式，使資料的交換與流通更為方便。

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。

因為預見未來大量的企業資料會以 XML 表示，各大資料庫廠商如微軟、IBM、甲骨文 (Oracle) 等，都在其既有的關聯式資料庫系統裡，擴充了處理 XML 的功能。但是，這些資料庫軟體本身相當昂貴，技術也很複雜，所以另一種作法是直接管理 XML 文件，如微軟提出 XML 的解析器(parser)，用來解析 XML 文件，以便於針對 XML 文件開發應用程式。本篇論文提出的作法，是利用微軟處理 XML 文件之軟體開發套件，配合自行設計的特殊索引結構，包括元素定位點 (offset) 與 B^+-Tree，以加快查詢 XML 文件時的速度。元素定位點是針對單一 XML 文件，表示其資料所在的位置，B^+-Tree 則可以協助我們從多份 XML 文件中快速找到所需要的文件。經過多方的實驗與測試，顯示我們所設計的索引結構的確有效地加速了查詢的速度。

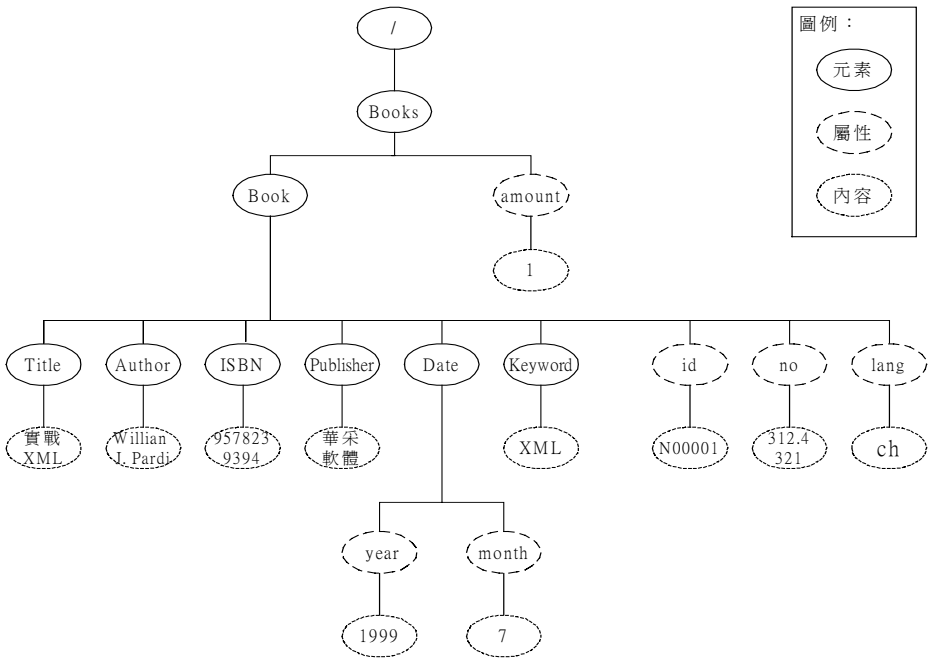
本篇論文的基本架構介紹如下：我們於第二節中，首先簡介 XML，說明其特殊的資料儲存格式及定義的方式，微軟提供的分析文件的方法也一並介紹之；在第三節中，介紹我們為 XML 資料查詢處理所設計的資料結構，包括元素定位點及 B^+-Tree 索引結構；我們對這兩個資料結構的效能所做的實驗，則在第四節中討論；最後，在第五節裡，我們舉出相關的研究報告以做比較，並在第六節的結論裡，指出未來的研究方向。

```
L1  <?xml version="1.0" encoding="Big5" ?>
L2    <!DOCTYPE Books SYSTEM "Books.dtd">
L3     <Books amount="1">
L4       <Book id="N00001" no="312.4321" lang="ch">
L5           <Title>實戰 XML</Title>
L6           <Author>Willian J. Pardi</Author>
L7           <ISBN>9578239394</ISBN>
L8           <Publisher>華采軟體</Publisher>
L9           <Date year="1999" month="7" />
L10          <Keyword>XML</Keyword>
L11       </Book>
L12    </Books>
```

圖一：XML 範例

```
L1  <?xml version="1.0" encoding="Big5"?>
L2  <!ELEMENT Books (Book*)>
L3  <!ATTLIST Books amount NMTOKEN "1">
L4  <!ELEMENT Book (Title, Author+, ISBN, Publisher, Date, Keyword+)>
L5  <!ATTLIST Book id ID #REQUIRED no NMTOKEN #REQUIRED lang (ch|en|jp|hz) "ch">
L6  <!ELEMENT Title (#PCDATA)>
L7  <!ELEMENT Author (#PCDATA)>
L8  <!ELEMENT ISBN (#PCDATA)>
L9  <!ELEMENT Publisher (#PCDATA)>
L10 <!ELEMENT Date EMPTY>
L11 <!ATTLIST Date year NMTOKEN #REQUIRED month NMTOKEN #REQUIRED>
L12 <!ELEMENT Keyword (#PCDATA)>
```

圖二：DTD 範例



圖三：DOM 結構範例

```
L1  TCOMIXMLDOMDocument2 DOMPtr = CoDOMDocument30::Create();
L2  DOMPtr ->async = false;
L3  DOMPtr ->load( “example.xml” );
L4  TCOMIXMLDOMElement pRoot = DOMPtr ->documentElement;
L5  TCOMIXMLDOMNodeList pBooksList = pRoot->childNodes;
L6  TCOMIXMLDOMElement pBooksNode = pBooksList ->get_item( 0 );
L7  TCOMIXMLDOMNodeList pBookItemList = pBooksList ->childNodes;
L8  TCOMIXMLDOMElement pBookNode = pBookItemList ->get_item( 0 );
L9  AnsiString type = pRoot->text;
```

圖四：DOM 範例程式

二、XML 簡介

我們首先簡單說明 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。圖一為一個 XML 文件的範例，圖中左欄的 Li 代表行數，其中 L1 行為 XML 宣告，說明此檔案採用的編碼為 Big5。L3 行的 Books 為本文件中的根元素，一個 XML 文件中只能有一個根元素。L4 行的 Book 是直屬於 Books 這個母元素下的子元素，它另外定義了三個屬性，屬性等號後面的值則為屬性值，譬如 id 屬性的值為 N00001，no 屬性的值為 312.4321，lang 屬性的值為 ch。而 L5 至 L10 定義的子元素如 Title 等，則是進一步提供 Book 元素的相關資料。

另外在圖一 L2 行裡使用了 Books.dtd，其檔案的作用是檢查此文件是否正確。DTD (document type definition) 是用來定義某種 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) 。圖二即為 Books.dtd 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。圖二中的 L4 行說明，Book 這個元素包含了 Title、Author 等子元素，而在 L6 行中進一步註明 Title 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、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，由 L5 行開始，定義了 Book 這個元素具有 id、no、lang 這三個屬性，id 屬性的類別為 ID，表示該屬性值在同一個 XML 文件中不可重複，預設行為的描述為 #REQUIRED ，表示該屬性值必須存在。

為了分析 XML 文件，W3C 通過一種叫做 DOM (document object model) 的規格，提供了 XML 文件架構模組的文件物件模型。它指出一份文件均從一個根節點 (root node) 開始往下發展成一個樹狀架構。圖三的 DOM 範例對應到圖一的 XML 文件，高度為 2 的節點為 XML 文件之根元素 Books，根元素之下再建立其他節點，包括子元素 Book 及屬性 Amount，如此依序往下發展為一個樹狀結構。DOM 的內容包含了邏輯結構和實體結構，邏輯結構就如同圖中所示的每一個內部節點，它說明這份文件應該包含那一些元素及屬性，並規定了這一些元素及屬性的順序，而實體結構則為文件的真實資料也就是內容，如圖中所示的每一個外部節點。透過 DOM 模型，我們可以清楚地知道一個 XML 文件內部資料的相關性，以便於加以處理取出所需要的資料。

我們所使用的 DOM 是 Microsoft 提供的 MSXML 3.0 軟體開發套件，其功用如同 W3C 所制定的，可以將 XML 文件分析並建立成如圖三的樹狀結構，它並進一步提供各種方法以便取得元素內容或是屬性值，以下則以 C++ 的程式碼說明之。

在圖四中的 L1 行，我們首先宣告 XML DOM 物件指標，指向一個新建立的 DOM 物件；接著在 L2 行，我們設定此 DOM 物件的行為，是先讀取完整個 XML 檔案再分析該文件。假定圖一內的 XML 範例所對應的檔案名

00 01	**00 61**	**00 50**	**00 44**	**00 66**	**00 7E**	**00 9E**	**00 BD**
unit **1**	unit **2**	unit **3**	unit **4**	unit **5**	unit **6**	unit **7**	unit **8**

圖五：元素定位檔範例

稱為 example.xml，在 L3 行，我們以 DOM 物件將此 XML 文件檔案載入，接下來我們就可直接從該 DOM 物件中把所要的元素取出。譬如，在 L4 行，我們取得 DOM 物件的 root element，也就是 Books 元素；而在 L5 行，則取得 Books 元素下所有子元素的清單，在此範例中是所有 Book 元素所成的集合。同樣的，我們可用不同的方法 (method) 去取得更下層的元素。譬如在 L6 行中，我們取得 Book 元素集合內的第 1 個元素，L7 行則取得第 1 個 Book 元素下的所有子元素清單，在這裡指的是 Title, Author, …等元素所成的集合；L8 行，取得第 1 個 Book 元素下子元素集合的第 1 個子元素，在這裡指的就是 Title 元素；最後的 L9 行，就可以直接取得 Title 元素的元素值，也就是 XML 文件 example.xml 內第 1 個 Book 元素下的第 1 個 Title 元素的值。

三、索引結構

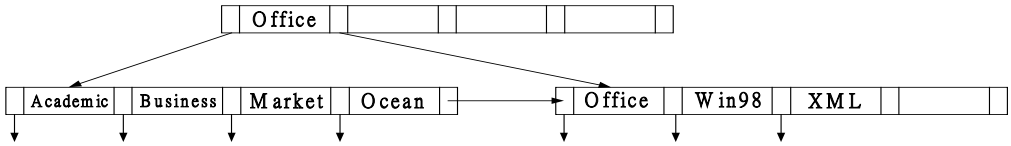
本節說明為支援快速查詢 XML 文件所設計的資料結構，首先討論提供查詢單一 XML 文件的索引結構。由於 XML 檔案內的元素具有固定的順序，所以我們利用此順序性建立元素定位點 (offset)，以記錄元素在檔案內的位置，如此則可省略利用 DOM 分析整份 XML 文件的步驟，快速搜尋到某元素的資料。我們設計讓一個 XML 資料檔案有一個對應的元素定位檔 (offset file)。檔案內每兩個位元組 (byte) 為一個單元 (unit)，第一個單元記錄此檔案之記錄 (record) 個數，之後的單元則組成一筆筆記錄。元素定位檔內的每一筆記錄對應到 XML 資料檔裡每一個主要元素的所有資料，所謂的“主要元素”是根元素之下的第一個子元素。以圖一為例，根元素為 Books，而主要元素為 Book。每一筆記錄的第一個單元，記錄其對應的主要元素資料的位置，也就是在 XML 資料檔案中與前一個主要元素資料之距離。而其後之單元，則記錄各子元素與主要元素的距離。

圖五表示的元素定位點資料對應到圖一的 XML 範例檔案，其中 unit 1 表示檔案內有一個 record，這是因為圖一內的主要元素 Book 只出現一次。unit 2 到 unit 8 則組成本檔案內的唯一一筆記錄。unit 2 是表示第一個主要元素(Book 元素)為從檔頭開始第 97（61h）byte，在檔案內我們以 16 進位表示，unit3 表示第一個子元素 Title 的資料距 Book 元素 45 (2Dh) bytes，unit 4 表示第二個子元素 Author 的資料元素距 Book 元素 68 (44h) bytes，unit 5 到 unit 8 也是依相同的方式表示。當子元素沒有資料，以 0 (00h) bytes 來表示。我們記錄元素間的相對位置，而非在檔案內的絕對位置，如此可使定位點資料不會因為 XML 檔案的資料修改而需頻頻修正。

接下來我們介紹為多份 XML 文件所設計的索引結構。一般企業若將資料以 XML 檔案存放時，由於資料量極大，所以多會表示成數個固定格式的 XML 檔案，也就是都符合同一個 DTD 檔案的定義。不過，若要查詢某筆特定資料時，必須循序搜尋所有檔案，效率會非常不理想。因此，我們針對幾個比較重要的元素，建立查詢值 (search key) 與 XML 檔案對應的索引結構，如此便可直接開啟所需要的 XML 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裡最常見的 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。如圖六所示，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。

B^+-Tree 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



圖六：B^+-Tree 結構

```
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)>
```

圖七：B^+-Tree 節點檔案之 DTD

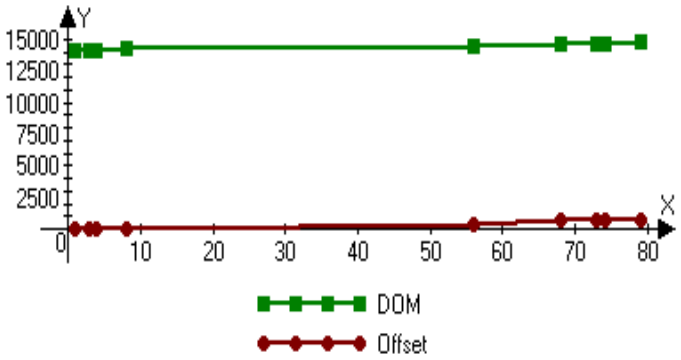
```
L1  <?xml version="1.0" encoding="Big5" ?>
L2  <!DOCTYPE BTree-Node SYSTEM "Btree.dtd">
L3  <Node type="ex">
L4     <Pointer key="Office">B0001.3,</Pointer>
L5     <Pointer key="Win98">B0002.2,</Pointer>
L6     <Pointer key="XML">B0001.1,</Pointer>
L7     <Next>B3.bt</Next>
L8  </Node>
```

圖八：B^+-Tree 節點之範例檔案

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 B^+-Tree 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 B^+-Tree 之高度不大於$\left| \log_{\lceil n/2 \rceil} K \right|$，這裡 K 為查詢值的個數。假設查詢值為 10,000 筆，當 order 為 20 時，則 B^+-Tree 之高度只有 4，所以查詢的速度相當快。圖六的 B^+-Tree 其 order 為 5。

我們的設計是每一個 B^+-Tree 的節點存成一個 XML 檔案，其 DTD 如圖七所示。在該圖中，L2 行表示 Node 元素為本文件之根元素，內含多個 B^+-Tree 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 B^+-Tree 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 元素代表下一個外部節點的檔名，以方便處理範圍查詢。

圖八為 B^+-Tree 節點的一個範例檔案，對應到圖六右下角節點，其格式遵循圖七之 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，B0001.3 則表示完整的真實資料為 B0001 這個檔案中的第三筆資料。值得注意的是，我們不只指出對應的檔案名，也一併記錄在該檔案裡的位置，如此可省卻比較檔案內容的步驟，更進一步加快查詢的速度。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 B^+-Tree，其根節點的檔名，以作為查詢的起點，此類資料存放在系統設定檔內，該檔案也是 XML 格式。



圖九：元素定位點與 DOM 查詢速度的比較

表一：元素定位點與 DOM 查詢到大量資料筆數的效能比較

資料筆數	DOM 所需時間	元素定位點所需時間
1948	28101	17996
1928	28190	17986
1925	27800	17605
1894	27780	17485

於是，利用 B^+-Tree 索引結構和元素定位檔，從多份 XML 文件中查詢到特定資料的處理步驟大致如下：

1. 利用系統設定檔，找出所查詢元素，如 Title 或 Keyword 所對應的 B^+-Tree 根節點為何檔案，開啟該檔案。

2. 若開啟的檔案對應到 B^+-Tree 的內部節點，則比較查詢值的大小，繼續走到樹的下一層，也就是開啟另一個 B^+-Tree 節點檔案。

3. 若開啟的檔案對應到 B^+-Tree 的外部節點，且真實資料存放的位置假設為 Bxxxx.n，則先開啟 Bxxxx.xml 所對應的元素定位檔 Bxxxx.of，取得第 n 筆資料位於 Bxxxx.xml 內的位置所在，然後再開啟 Bxxxx.xml 資料檔案，利用之前得到的位置資訊，直接取出第 n 筆元素資料，處理後傳回結果。

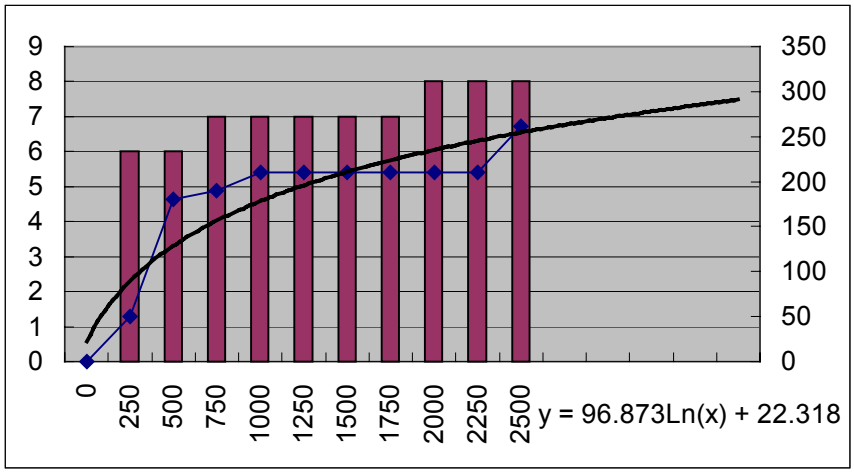
四、實驗結果

在本節中我們設計數個實驗，來評估所提出的索引結構的效能。我們使用的實驗環境為 PentiumIII650 的個人電腦及 256M 的記憶體 (RAM)，加上 Win2000 Server 的作業系統。實驗中所用來測試的資料，是以電腦亂數的方式產生，而實驗數據的收集是根據關鍵字的查詢。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，進行多次的實驗，取得平均值。以下針對各實驗作仔細的說明。

(一) 元素定位點與 DOM 的效能比較

在本實驗中，我們利用所設計的元素定位點，與微軟所提供的 DOM 元件，比較對單一 XML 文件查詢的速度。該文件表示了 50000 筆不同的書籍資料。

如圖九所示，找到資料的筆數為 X 軸，查詢所需時間為 Y 軸，以微秒 (ms) 為單位。元素定位點 (offset) 所對應到的折線是以圓圈連起來，位於圖的下方，而 DOM 則是對應到以方塊連起來的折線，位於圖的上方。我們可觀察到，DOM 所需的時間，遠遠地超過元素定位點，這是因為 DOM 必須先分析完 50000 筆資料，以建立出如圖三的樹狀結構，而這段時間花費甚鉅。



圖十：檔案個數與利用 B^+-Tree 查詢所需時間之對照表

另外，我們也發現到，不管使用何種方法，查詢所需的時間會隨著找到的資料筆數增加，但元素定位點顯然較 DOM 增加得還快。為了瞭解當找到的資料筆數非常大時，使用元素定位點查詢是否仍然較 DOM 快，我們做了如表一的實驗。由表一所示，在找到的資料筆數將近兩千筆時，雖然兩者的差距較圖九的大為減少，但是 DOM 的時間仍然約為元素定位點的一倍半。因此我們再做了一個特殊的實驗，也就是比較兩者將所有書籍資料 (50000 筆) 找出來所需的時間，在這個實驗中，使用元素定位點查詢時所需的時間為 1138654ms，而使用 DOM 查詢時所需的時間為 1102456ms，兩者的差距並不大。因此我們可以得知，使用元素定位點查詢確實較 DOM 快，其差距則隨著找到的資料筆數越多而遞減。

(二) B^+-Tree 與多檔案查詢效能的關係

本實驗是測試當 XML 檔案很多時，B^+-Tree 索引比循序查詢更有效率。我們將 20 筆書籍資料存放在一個檔案，分別針對擁有不同檔案個數的資料做關鍵字的查詢，關鍵字對應到的 B^+-Tree 之 order 則定為 4。

如圖十所示，X 軸為檔案個數，右側的 Y 軸刻度表示查詢所需的時間，這裡的時間只記錄找到資料所在的檔案。根據查詢的演算法，該時間應該就是走過 (traverse) B^+-Tree 的時間，所以我們在左側的 Y 軸刻度同時表示 B^+-Tree 的高度，並利用柱狀圖代表各種不同檔案個數所對應到的 B^+-Tree 高度。

而圖十中，由菱形方塊連成的折線圖記錄各種不同檔案個數查詢所需的時間。我們發現，當檔案個數增加時，查詢的時間並沒有隨之線性成長，由此可見針對多個檔案的查詢，B^+-Tree 提供了比循序查詢更有效的方法。我們也根據折線圖找到一條對數的趨近線，以茲對照。

同時我們發現，當 B^+-Tree 的高度增加時，查詢時間不一定隨之增加，這是因為查詢時間除了和 B^+-Tree 高度有關外，亦與每個關鍵字在 B^+-Tree 節點內所在的位置有關，也就是關鍵字的分布會讓搜尋時間有差異。目前我們所使用的是線性搜尋的演算法，若是改用其它更快速的搜尋演算法，如二元演算法，可讓此現象較不明顯。

(三) Cost Model 的建立

在本節中，我們希望建立出一個時間公式 (cost model)，可以用來推算利用我們的 B^+-Tree 與元素定位點，去多個 XML 檔案中查詢資料時，所需要的時間。我們以 I/O cost 做為 cost model 的衡量標準，並不考慮 CPU time，而處理一個 block 的時間視為一個 I/O cost。由於我們使用的 Win2000 作業系統磁碟分割格式為 NTFS，其一個 block 的大小為 512 bytes，為了簡化起見，我們讓 2 筆書籍資料存放在一個 XML 檔案，所以存取一個 XML 檔案約為一個 block I/O。

雖然在這樣的設計下，每個 XML 檔案、offset 檔案、及 B^+-Tree 的所有節點檔案，大

表二：實際時間與預測時間對照表

實驗	A	B	C	D	E
實際時間	24.14	25.94	28.94	30.14	32.45
預測時間(I/O cost)	45. 88	49. 88	53. 88	57. 88	61. 88
比率(實際/預測)	0.5262	0.5200	0.5371	0.5207	0.5244

實驗	F	G	H	I	J
實際時間	26.64	26.44	30.05	25.74	31.24
預測時間(I/O cost)	53. 88	53. 88	60.86	51. 88	62.86
比率(實際/預測)	0.4944	0.4907	0.4938	0.4961	0.4970

小皆約為一個 block，但是 B^+-Tree 的節點檔案都必須被 DOM 解析(parse)，而由先前的實驗得知，它會耗費相當大的時間，不可忽略。為了得到每個 B^+-Tree 節點檔案比純粹對一個 block 做 I/O 時多花的額外時間，我們加以實驗，結果得到處理一個 XML 檔案及其相對應的一個 offset 檔案共需要 1.0525ms，加以平均得到處理一個檔案的時間，也就是一個 block I/O 需要花 0.5263ms。另外我們得到處理一個 B^+-Tree 節點檔案需要 3.6716 ms，相除後得知處理每個 B^+-Tree 節點檔案大約需要 6.98 個 block I/O 的時間。根據此結果，我們有下列推論：

cost model

＝B^+-Tree 所需之時間+元素定位點所需之時間+XML 檔案所需之時間

＝6.98×B^+-Tree 之高度+找到資料所對應的元素定位點檔案個數+找到資料所對應的 XML 檔案個數

為了驗證我們所訂定的 cost model 之正確性，我們針對 5000 個 XML 檔案加以實驗，根據關鍵字的查詢，將實際所需的時間記錄在表二中的第二欄。表中的預測時間 (第三欄)，則是利用 cost model 的公式計算出來。由該公式得知，實際時間與預測時間的比率，就是對一個 block 實際做 I/O 所需的時間，我們由表二的第四欄可以發現，大部份的實驗結果都與之前得到的 0.5263 相當接近，由此可知我們的 cost model 相當正確。以下對各個不同環境的實驗做進一步的說明。

表二中的實驗 A 到 E 以具有 2000 個關鍵字的 10000 筆書籍資料做為實驗範本，B^+-Tree 的 order 為 4，其所建立的 B^+-Tree 高度為 6，而根據關鍵字的查詢所找到的資料，由實驗 A 到 E 分別為 2、4、6、8、10 筆。觀察表二中的數據，我們發現由於 B^+-Tree 的高度為定值，所以查詢時間並不受 B^+-Tree 高度的影響，而是隨著找到資料筆數的增加而增加，此結果與我們的 cost model 吻合。

而在表二的實驗 F 、G 及 H，則分別以具有 1000、2000 和 5000 個關鍵字的 10000 筆書籍資料做為實驗範本，其所建立的 B^+-Tree 高度各為 6、6 和 7，我們取根據關鍵字查詢所找到的資料為 6 筆的查詢時間。我們發現當找到的資料筆數固定時，查詢時間隨著 B^+-Tree 的高度而改變，這個結果也符合我們 cost model 的設計。同時我們發現，當實驗的資料來源及環境完全相同時，如實驗 C 及實驗 G 皆是以擁有相同 2000 個關鍵字的 10000 筆書籍資料做為實驗範本，而使用不同的關鍵字做查詢時，雖然找到的資料筆數相同，可是其實際的查詢時間卻不一樣，這是因為關鍵字在 B^+-Tree 節點內的分布會影響搜尋時間。

最後，表二中的實驗 I 及 J，以分別具有 3000 和 5000 個關鍵字的 10000 筆書籍資料做為實驗範本，其所建立的 B^+-Tree 高度各為 6 和 7，而找到的資料筆數則為 5 和 7 筆，也就是所有的環境皆不同。但觀察實驗數據，我們發現實際的查詢時間與 cost model 所預測的 I/O cost 非常接近，由此可知我們所訂的 cost model 是正確的。

五、相關研究

隨著 XML 日益受到重視，眾多研究者也開始進行相關的研究。例如，有研究者針對 XML 資料實作了 Java 翻譯引擎(rendering engine)，以便處理具有複雜特殊符號的文件 (Z Specifications)，並使其可在網際網路上快速傳遞與呈現 [5] 。 XML-GL [3]，則為針對 XML 文件的圖形化查詢語言，可提供傳統關聯式資料庫的基本查詢功能。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 [11]，以下則針對和我們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 [9]。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 (path expression) 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 [10]。

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 [2]。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 [7]。有的研究者則替 XML 查詢提供了一個代數的方法，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 [4]。

另外一個與本論文相關的議題是眾多 XML 文件或資料可能產生的效能問題。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 [8]。至於在 [6] 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。

六、結論

電子商務是未來的趨勢，在網際網路上從事企業間的資料交換，是其中重要的課題，而 XML 的推出更加速了這方面的推展，我們相信未來有很多企業將會以 XML 表示其資料。本篇論文的目的，是討論加快眾多 XML 文件查詢速度的方式，我們設計了 B^+-Tree 及元素定位點的特殊資料索引結構，並建立測試資料測量效能。根據我們的實驗數據，不管是針對單一 XML 文件，或是多份 XML 資料，我們所設計的索引結構都提供相當好的查詢效率。

本研究未來可能改進的方向有下列幾點：

1. 我們目前設計的 B^+-Tree 索引結構，由於是 XML 格式，所以仍然需要透過微軟提供的 DOM 發展套件來分析該 XML 文件，耗時極大，未來希望能加以改善。

2. 目前所設計的索引結構只支援針對某一特定元素的查詢，隨著更多 XML 查詢語言結構的提出，我們希望能擴充目前的設計，以支援更複雜的查詢功能。

誌謝

此計畫由國科會贊助，編號為：NSC

90-2213-E-019-011

七、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] K. Bohm, “On extending the XML engine

with query-processing capabilities", In Proceedings of IEEE Advances in Digital Libraries, pages: 127 –138, 2000.

[3] S. Ceri, S. Comai, E. Damiani, P. Fraternali and L. Tanca, "Complex Queries in XML-GL", ACM Symposium on Applied Computing, volume 2, pages: 888 – 893, 2000.

[4] 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.

[5] P. Ciancarini, F. Vitali, C. Mascolo, "Managing Complex Documents Over the WWW: A Case Study for XML", IEEE Transactions on Knowledge and Data Engineering, volume 11, issue 4, pages: 629 – 638, 1999.

[6] 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.

[7] 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.

[8] 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.

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

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

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