Incremental Structured Index Update Mechanism and Path Expression Query for XML Documents

Student: Chien-Che Hung Advisor: Dr. Hao-Ren Ke, Dr. Wei-Pang Yang

Institute of Computer and Information Science
National Chiao Tung University

ABSTRACT

XML documents contain abundant structural information that can be employed to better understand the XML documents. Structured queries are queries that search the structural information of XML documents. In order to support structured queries, k-ary trees are used in this thesis as an index structure to store structural information. Based on this index structure, this thesis proposes an incremental structured index update mechanism and implements a few functions of a SQL-like path expression query language, called Lorel.

K-ary trees are a well-known index structure can be used to store structural information of XML documents, where k is a predefined maximum number of branches that a tree node can have. When an XML document is modified, its corresponding k-ary tree has to be updated as well for reflecting the modification. However, when a modification makes the number of branches of a tree node exceeds k, the k-ary tree is no longer workable and the whole k-ary tree has to be reconstructed. Reconstructing the whole k-ary tree is time-consuming when the structure information changes frequently. This thesis proposes an incremental structured index update mechanism called the *Split Method* to avoid reconstructing the k-ary tree when the number of branches of a node exceeds k. Through splitting

nodes and defining an equivalent relationship, this mechanism can update k-ary trees efficiently. Several evaluations were conducted to assess the Split Method and the k-ary tree reconstruction. The evaluation shows that the Split Method only wastes a little more space than tree reconstruction, but it is more efficient than tree reconstruction in updating k-ary trees.

In addition, based on the k-ary tree structured index, this thesis implements a few functions of a SQL-like path expression query language, called Lorel. Path expression queries facilitate user to issue structural queries when the document structure is irregular or unknown in advance. The results of queries are formatted as XML documents.

Keywords： Incremental Structured Index Update, k-ary Tree, Path Expression Query, Split Method, XML

XML 文件漸進式結構化索引更新與路徑表示式檢索之研究

研究生：洪健哲　　指導教授：柯皓仁博士，楊維邦博士

國立交通大學資訊科學研究所

摘要

XML 文件中隱含了豐富的結構化資訊，為了能利用這些結構化資訊提供結構化檢索(Structured Query)，本論文利用 k-ary 樹狀結構的特性來建立結構化資訊索引(Structured Index)。基於此索引結構，本論文提出一套漸進式結構化索引更新機制來避免文件結構更新時需重新建置結構化資訊索引，此機制稱為 *Split Method*。此外，本論文實作了一套 SQL-like 的路徑表示式檢索(Path Expression Query)，可讓使用者在結構資訊不規則或不明確的情況下透過結構化資訊索引找出滿足檢索路徑條件的資料，並將檢索結果以 XML 呈現。

在結構化文件索引更新中，若加入資料的節點分支數超出索引結構之最大分支數 k 時，傳統的方法係將結構索引重建，當結構資料時常變更時，此索引結構重建的步驟相當耗時。有鑑於此，本論文乃提出一個漸進式結構化索引更新的機制來避免索引重新建置。此方法主要利用資料分屬不同文件的概念，即分裂的節點所屬之文件編號(Document Identifier, DID)與原始資料之 DID 的不同。當加入資料的節點超出索引結構之最大分支數時，從該節點分裂出一個節點，而分裂的節點另屬於不同的 DID，並定義一個對等關係來記錄該節點與分裂節點間的關連性。

從效益評估結果顯示，雖然本論文提出之 Split Method 需要較多的索引空間花費，以本論文中的實驗為例，索引空間花費的 Overhead 約介於 2.38%與 17.03%之間，然而在索引更新的效能上大大優於索引重建的方法，且對於實際檢索的效能，我們的方法與原來索引重建的方法相較之下並未相差太大。

關鍵字： 漸進式結構化索引更新，k-ary 樹狀結構，路徑表示式檢索，分裂方法，可擴展標示語言

誌謝

時光荏苒，回想起兩年的研究生涯，在柯皓仁教授與楊維邦教授的帶領下，讓我有如沐春風之感。雖然 Meeting 比別人多、課業比別人忙，但若沒有經過這段時間的琢磨與煎熬，相信我不會像現在一樣，感到充實與快樂。於此，特別感謝柯教授，在這段期間除了理論研究的指導之外，還為我安排了一些工作，讓我在實務經驗累積上更為豐碩。最終的碩士論文得以完成，除了一再地與柯教授反覆討論外，楊教授亦是悉心指點論文研究時的種種盲點，在此，健哲由衷感謝您。同時，也感謝口試委員黃明居教授與陳昭珍教授的蒞臨指導，讓本篇論文得以更加完善。

兩年歲月，因為有實驗室學長們的照顧；同學間的相互關心、鼓勵，加上可愛的學弟妹們，讓我在交大的日子裡，感受到同儕間的溫暖與情誼。在此，謝謝鄭培成、蔡政容與李正吉學長們在生活上的關照；黃夙賢學長在學業上的解惑；鎮源、玉旻、家豪、繼弘、元琳等幾位好友在各方面的支持與協助，在我灰心與氣餒之時，你們猶如一盞明燈，點亮我眼前的路，也讓我燃起一線希望，謝謝你們！

此時此刻，心中最想感謝的是我的父、母親。因為你們，讓我在成長與求學的過程中，能不畏艱難地勇往直前、日求精進。另外，支持我達到此目標的人，除了父、母親之外，還有我的閨中密友--歆怡，在我最無助、最沮喪時，她不吝嗇地給予我支助、安慰與包容，讓我有勇氣面對種種困難、有決心度過層層難關，謝謝妳—歆怡。職是之故，僅將此篇論文獻給我最愛的家人與閨中密友。

June 20, 2002

目錄

表目錄

圖目錄

方程式目錄

第一章　緒論

第一節　研究動機

可擴展標示語言(Extensible Markup Language，XML) [Roy00]、[XML98]、[Zisman00] 逐漸成為網際網路上資料溝通、交換的一套標準。XML 與 HTML 都屬於標示語言，但 XML 比 HTML 支援更豐富的功能，如允許使用者自訂標籤與定義標籤之間的結構化關係，並於文件中使用自訂的標籤來描述對應的資料內容。同時，XML 將資料呈現的方式從資料內容中分離出來，資料呈現主要是透過可擴展樣式語言(eXtensible Stylesheet Language，XSL) [XSL01]來呈現。如此設計使得 XML 文件更簡潔、更具模組化，使得一份內容除了在 WEB 上瀏覽外也可以放在不同設備上瀏覽，諸如 PDA 或無線裝置等。

由於 XML 具有的彈性與韌性，使其應用的層面很廣，如透過 XML 來做資料交換、資料庫間的互動溝通、文件發佈傳播等等；不同程式之間可利用 XML 當作共通語言來進行資料交換，或者做為異質性系統間溝通的橋樑。由於 XML 具有允許自訂標籤的特性，使其本身亦扮演著母語言(Metalanguage，亦稱超語言、元語言) 的角色。因此，許多領域乃以 XML 為基礎定義出適用的標示語言。如[Zisman00]中所提及的 Mathematical Markup Language (MathML) [MathML]與 Chemical Markup Language (CML) [CML]等標示語言就是針對數學與化學領域定義其標籤，以方便表示數學與化學式子。

XML 的彈性設計亦使得網路資訊的搜尋與擷取更快、更有效率，因為透過自訂標籤來描述資料內容能使得資料的意義更明確。例如球隊的名稱可自訂一對標籤為"<*TEAM_NAME*>*球隊名稱*</*TEAM_NAME*>"，如此，搜尋引擎即可利用標籤資訊有效且準確的找出使用者要的資料。舉例來說，使用者可發出一個檢索條件為「找出球隊名稱為『Dodgers』的球隊資訊」，搜尋系統則利用自訂標籤的

特性尋找並比對標籤<TEAM_NAME> … </TEAM_NAME>間的資料(即球隊名稱)是否為『Dodgers』，若球隊名稱符合時即將其相關的球隊資訊列出。

另外，XML 能夠較明確地表示文件資料內容的意涵。因為 XML 的標籤可以視為文件內容的詮釋資料(Metadata)，若直接利用 XML 來當作資料交換的格式，更能表達資料內容的實際意義。以圖書館為例，通常利用 ISO 2709 來做資料交換的標準，資料內容為一連串資料的編碼，對館員來說，這樣的格式內容觀看不易，必須經過進一步資料剖析的動作方能得知文件內容。相對地，若利用 XML 的標籤來標記書目的相關資料，透過標籤即可明瞭資料對應內容為何。例如:『作者』的資料就會出現在一個對應的 XML 標籤<Author>作者資料</Author>內，如此必定讓處理交換資料的館員一目了然。

XML 具有如上所述的多方面優點，我們可以預期未來將會有越來越多的應用會基於 XML 來發展、越來越多的文件會利用 XML 來編寫。因此，當越來越多應用以 XML 為基礎當作資料傳播、資料交換的格式時，如何有效管理 XML 資料已成為一個值得研究的方向。

在[Larson01]中針對 XML 資料管理系統提出了幾個不同的架構，大體上可區分為三類：

1. 直接針對 XML 的特性來設計、建構一套適合 XML 文件的資料管理系統;
2. 透過關連式資料庫來管理；
3. 透過物件導向資料庫來管理。

無論利用哪種管理架構，資料索引與資料檢索皆為重點議題。對於 XML 文件中所隱含的結構化資訊，若能為其建置索引並有效地維護，必能提供更多元、更有效率的檢索，如內容檢索(Content Query)與結構化檢索(Structured Query)。為了有效率地進行結構化檢索，檢索系統必須針對結構化資訊建立結構索引(Structured Index)。對於結構索引，[Lee96]提出如何利用 k-ary 完整樹狀(k-ary

Complete Tree)結構來保存結構資訊：由於每個結構化文件都能夠展開成一個階層式的樹狀結構，故該論文將一份結構化文件對應至一個 k-ary 完整樹狀結構上，並以該文件最大的分支數目作為此樹狀結構的分支數 k；將結構化文件對應到完整樹狀結構後，再根據此樹狀結構上的節點依序配置每個節點一個唯一元素識別號(UID)，透過節點之 UID 只需用簡單的運算即可得知該節點的父節點與子節點之 UID。該論文即利用此概念來記錄結構間的資訊。

對於一份結構化文件，若在結構資訊上有變動時，對應的索引資訊亦會受到影響。例如：一份結構化文件可能會新增、修改或刪除某個標籤資料，對應到 k-ary 完整樹狀結構上，則表示有新的節點會加入索引結構、原有的節點資訊會被修改成更新的資訊，而欲刪除的資料則會從索引結構中移除。索引資料更新的狀況中，影響最大的情況在於新增資料時，有可能使完整樹狀結構的分支數超出原有的最大分支數 k 值，此時加入的節點已無法適當地配置 UID。針對此問題，目前的解決方法乃是將新標籤資料加入結構文件中後，再重新將此結構文件對應到一個新的 k-ary 完整樹狀結構上。然而，當資料更新頻率非常頻繁時，此方法非常耗時，尤其當資料量大時更凸顯了此缺點。

至於 XML 檢索語言方面，迄今有許多研究針對 XML 文件提出各種檢索語言。在眾多的檢索語言當中，目前尚未有一套標準的檢索語言。本論文中則實作了一套可親性(User Friendly)高的 SQL-like 檢索語言，並將其應用於 XML 文件檢索。

第二節　研究目的

1.2.1　避免結構索引重建

本論文中，針對結構索引乃以 k-ary Tree 的樹狀結構為索引基礎。基於此索引結構，如果 XML 文件資料的變動非常頻繁，相對地結構資訊需時常變更，但只要每次加入新節點時，若超出原有的完整樹狀結構的最大分支數 k，即須重建

一個新的完整樹狀結構，此方法非常耗時，尤其當資料量非常龐大時，更顯示出索引重建的效率不彰。有鑑於此，本論文提出一個漸進式結構化索引更新的方法，避免結構索引整個重新建置，以改善索引維護的效能。

1.2.2 實作 XML 檢索語言

我們利用 k-ary Tree 的樹狀結構做為 XML 結構化資訊的索引之用。有了資料索引後，如何從中取出所需的資料乃為一個重要的議題。目前許多研究已提出多種 XML 檢索語言，諸如：Lorel [Abiteboul97]、XML-QL [Deutsch98]、XML-GL [Ceri99]…等，但這些檢索語言中，除了 Lorel 有說明如何透過其所使用的資料模型(Data Model)來達到資料檢索外，其餘皆只提出檢索語法卻未提及細部實作的方法。因此，本論文擬針對某個檢索語言透過我們使用的索引結構來實作檢索功能。由於[Abiteboul97]提出的檢索語言 Lorel 乃是透過路徑表示式(Path Expression)並定義了 SQL-like 的檢索語法，對於熟悉 SQL 語法的使用者來說，更能快速熟悉 Lorel 檢索語法並運用它以達到 XML 檢索的目的。因此，我們實作了 Lorel 的 SQL-like 檢索語言，透過路徑表示式來檢索 XML 文件。

第三節　本論文內容與架構

本論文第二章介紹 XML 相關技術背景、Lorel 檢索語言簡介與本論文採用的索引結構，第三章說明我們提出的漸進式結構化索引更新方法並概述實作的 XML 文件檢索系統架構，第四章則對實作的路徑條件檢索做詳細說明，第五章展示實作的系統與漸進式結構化索引更新方法的效益評估分析，第六章歸納出結論與未來研究方向。

第二章　相關研究

本章說明與本論文相關的研究工作，主要分成三個部分，首先介紹 XML 技術背景，包括 XML 文件類型、DOM 資料模型等。第二部分介紹針對 XML 文件所提出的檢索語言，如[Abiteboul97]、[Deutsch98]等論文。最後會描述本論文所採用的資料索引結構 — k-ary 完整樹狀結構。圖 1 分別列出這些相關研究工作之大類別及發表年代(詳見本論文最後面之參考文件)。

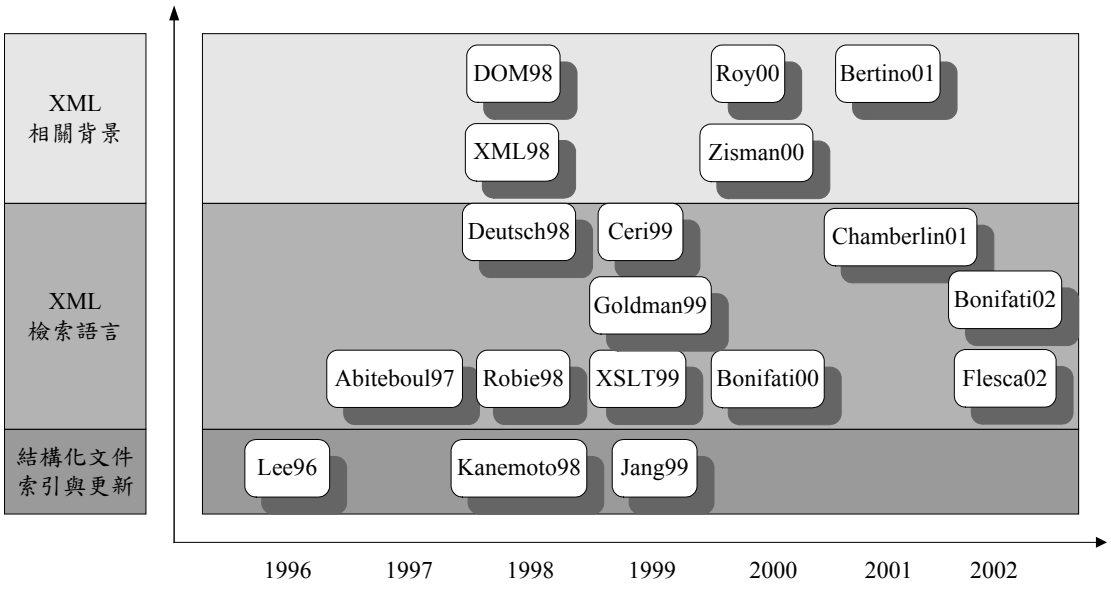

圖 1：相關研究工作

第一節　XML 技術背景介紹

2.1.1　概要

XML (Extensible Markup Language) [XML98] [Zisman00] [黃中杰 00]是近年來備受矚目的標示語言，它繼承了 SGML (Standard Generalized Markup Language) [Bryan92]的強大可擴充能力，提供使用者自訂標籤、自訂文件結構，並以純文字方式表示各式各樣的文件資料；除此之外，XML 還儘可能地降低 SGML 的複

雜性與困難度，使其本身簡單且易於使用。圖 2 為以 XML 資料格式來表示一個棒球隊及其球員資料。

```
<?xml version="1.0"?>
<DIVISION>
  <TEAM>
      <TEAM_CITY>Los Angeles</TEAM_CITY>
      <TEAM_NAME>Dodgers</TEAM_NAME>
      <PLAYER>
          <SURNAME>Darren</SURNAME>
          <GIVEN_NAME>Hung</GIVEN_NAME>
          <POSITION>Starting Pitcher</POSITION>
          <GAMES>8</GAMES>
          <HITS>168</HITS>
      </PLAYER>
      <PLAYER>
          <SURNAME>Carolina</SURNAME>
          <GIVEN_NAME>Kao</GIVEN_NAME>
          <POSITION>Relief Pitcher</POSITION>
          <GAMES>7</GAMES>
          <WINS>5</WINS>
      </PLAYER>
      ...
  </TEAM>
  <TEAM>
  ...
  </TEAM>
</DIVISION>
```

圖 2：XML 球隊資料範例

XML 對於使用者自行定義的標籤、標籤間的關係以及文件結構，是定義在所謂的 DTD (Document Type Definition 或 Document Type Declaration)中。根據 DTD 是否包含於 XML 文件內，可將 DTD 分成兩種：若 DTD 另屬一個文件則稱此 DTD 為外部 DTD (External DTD)；若包含在 XML 文件中則稱之為內部 DTD (Internal DTD)。圖 3 為圖 2 球隊資料範例之 XML 檔對應的 DTD。

```
<!ELEMENT DIVISION (TEAM+)>
<!ELEMENT TEAM (TEAM_CITY, TEAM_NAME, PLAYER+)>
<!ELEMENT TEAM_CITY (#PCDATA)>
<!ELEMENT TEAM_NAME (#PCDATA)>
<!ELEMENT PLAYER (SURNAME, GIVEN_NAME, POSITION, GAMES, HITS?, WINS?)>
<!ELEMENT SURNAME (#PCDATA)>
<!ELEMENT GIVEN_NAME (#PCDATA)>
<!ELEMENT POSITION (#PCDATA)>
<!ELEMENT GAMES (#PCDATA)>
<!ELEMENT HITS (#PCDATA)>
<!ELEMENT WINS (#PCDATA)>
```

圖 3：DTD 範例

圖 3 的 DTD 定義 XML 檔以 DIVISION 作為根部節點，其子節點至少出現一個 TEAM 節點，而 TEAM 節點下須有一個 TEAM_CITY、TEAM_NAME 與一個以上的 PLAYER 子節點；其中 PLAYER 這個子節點下又須有 SURNAME、GIVEN_NAME、POSITION、GAMES、HITS 與 WINS 這五個子節點，而 HITS 與 WINS 子節點可在節點 PLAYER 中出現一次或不出現。沒有子節點的節點在範例中設定它們的資料型態為可被解讀的字元資料 (以#PCDATA 表示)。詳細的 DTD 規範可參考 XML 規格書 [XML00]。

2.1.2 資料為主(Data-Centric)與文件為主(Document-Centric)的 XML 文件

XML 文件依其資料內容取向可區分成兩類：以資料為主(Data-Centric)與以文件為主(Document-Centric) [Bertino01]。資料為主的 XML 文件係利用 XML 格式來儲存資料以方便於網路上資料傳輸，這類型的文件在結構(Structure)與內容(Content)上都較具規則且較著重於結構間的資料內容。圖 2 的 XML 球隊範例即屬於此類型的文件。另一方面，當我們將整個 XML 文件視為一個應用程式相關的物件(Application-Relevant Objects)時，此類的 XML 文件則屬於以文件為主的 XML 文件，這類的文件在結構上較不規則，內容也比較複雜。圖 4 為一個文件為主的 XML 範例。

```
<Product>
<Name>Turkey Wrench</Name>
<Developer>Full Fabrication Labs, Inc.</Developer>
<Summary>Like a monkey wrench, but not as big.</Summary>
<Description>

<Para>The turkey wrench, which comes in both right- and left-handed versions (skyhook optional), is made of the finest stainless steel. The Readi-grip rubberized handle quickly adapts to your hands, even in the greasiest situations. Adjustment is possible through a variety of custom dials.</Para>

<Para>You can:</Para>
<List>
<Item><Link URL = "Order.html" >Order your own turkey wrench</Link></Item>
<Item><Link URL = "Wrenches.htm" >Read more about wrenches</Link></Item>
<Item><Link URL = "catalog.zip" >Download the catalog</Link></Item>
</List>

<Para>The turkey wrench costs just $19.99 and, if you order now, comes with a hand-crafted shrimp hammer as a bonus gift.</Para>

</Description>
</Product>
```

圖 4：以文件為主的 XML 文件

2.1.3 XML 文件物件模型(DOM)

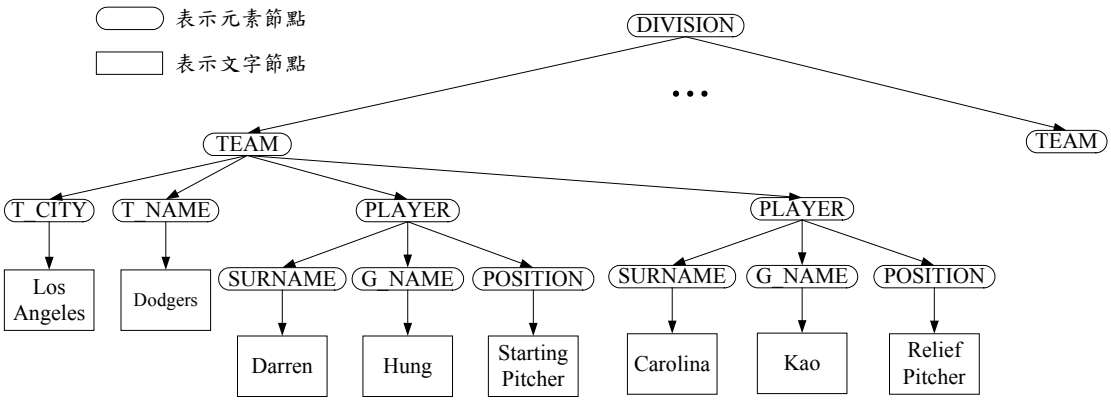

圖 5：DOM 樹狀結構表示法

DOM [DOM98] [Roy00] [Zisman00]為 Document Object Model 的縮寫，中文譯為文件物件模型。DOM 是以樹狀結構為基礎的資料模型，乃針對 HTML 與 XML 設計的應用程式介面(API)。DOM 定義了結構文件的邏輯架構並提供許多方法(Method)讓使用者存取與操作樹狀結構內的資料。以圖 2 的 XML 球員資料

為例，若以 DOM 樹狀結構表示則如圖 5 所示，其中方形節點代表文字節點(Text Node)，橢圓形節點表示元素節點(Element Node)，而最上層的節點為根節點(Root Node)。在 DOM 樹狀表示法中的每一個節點皆為一個物件。

在對 XML 文件操作時，可先將 XML 文件對應到 DOM 樹狀結構，之後透過 DOM 提供的方法(Method)來走訪(Traverse)整個樹狀結構，甚至可在樹狀結構上新增、刪除某個節點或更新節點資料。由於 DOM 樹狀結構上的所有操作皆在記憶體中進行，一旦記憶體中的資料釋放掉後，若欲再次存取 XML 文件內的資料時，需重新將 XML 對應到 DOM 樹狀結構。

為避免前述之狀況，我們透過 DOM 將 XML 轉成內部資料表示法存放於磁碟中。當欲對 XML 文件操作時，即可透過對應的內部表示法有效地取得 XML 各部分的資料，而不需每次重新剖析 XML 產生對應的 DOM 樹狀結構。此外，若 XML 文件檔案過大時，對應的 DOM 樹狀結構所佔據的記憶體空間亦相對的提高，導致整體操作效能降低。但若利用對應的內部表示法則只需將必要的部分載入記憶體中即可，而不需將所有資料放入記憶體中。

第二節　XML 檢索語言

隨著 XML 逐漸成為網際網路上資料呈現與資料交換的一套標準，越來越多針對 XML 文件之檢索語言相繼提出。如[Abiteboul97]提出之 Lorel 有著 SQL-like 的語法；有些檢索語言的設計概念則是源自於 XML 發展出來，如[Deutsch98]提出之 XML-GL。本論文所實作之 XML 路徑檢索乃基於 Lorel 檢索語言，以下乃針對其特色來做說明，並簡介其他檢索語言。

Lorol 是由史丹福大學(Stanford University)針對半結構化(Semi-Structured)資料提出之檢索語言，經過適當的修正已可應用於 XML 文件檢索[Abiteboul97] [Goldman97]。Lorel 的特色在於它為 SQL-like 的語法，並透過強大的路徑表示式(Path Expression)來檢索 XML 文件，尤其當對於文件的結構化資訊不明瞭

時，更能顯示出其強大的檢索能力。以圖 2 的 XML 球隊範例來說，若欲找出所有球隊中名為「Darren」的球員資訊，則對應的 Lorel 檢索語法如圖 6 所示。

```
Select DIVISION.TEAM.PLAYER
Where DIVISION.TEAM.PLAYER.G_NAME = 'Darren'
```

圖 6：Lorel 檢索範例一

對文件結構不規則甚至未知的結構，Lorel 提供萬用字元(Wild Cards)與常規表示式(Regular Expression)來檢索文件。舉例而言，假設 XML 文件中某些球員的姓名定義在「NAME」節點下，而某些球員的名字以標籤節點「FIRST_NAME」或「GIVEN_NAME」命名。圖 7 的範例即可在結構資訊不明確的情況下，找出符合名為「Darren」的球員資訊，其中的「.NAME」節點後面出現的符號「?」表示「NAME」節點在路徑表示式中可出現或不出現。而萬用字元「%」會找出以 NAME 字串結尾的節點，因此節點標籤「FIRST_NAME」或「GIVEN_NAME」內的資料若為「Darren」，則兩者皆會滿足路徑檢索條件。

```
Select DIVISION.TEAM.PLAYER
Where DIVISION.TEAM.PLAYER(.NAME)?.%NAME = 'Darren'
```

圖 7：Lorel 檢索範例二

其他檢索語言如 XML-QL [Deutsch98]的設計概念源自於 XML，其檢索語法如圖 8 所示。此檢索範例的語法在於找出 TEAM 名稱為「Dodgers」下所有球員的資訊。

```
WHERE   <DIVISION>
          <TEAM>
              <T_NAME>Dodgers</T_NAME>
              <PLAYER> $p </PLAYER>
          </TEAM>
        </DIVISION> IN "www.a.b.c/team.xml"

CONSTRUCT $p
```

圖 8：XML-QL 檢索範例

從 Lorel 與 XML-QL 的檢索範例可看出，Lorel 使用的語法較容易讓人理解。若能提供 Lorel 這類的高階檢索語言，對於習慣 SQL 語法的使用者，更能快速地上手與應用。

定義了 Lorel 等檢索語言後，如何透過各種資料模型來實作 XML 文件檢索亦是重要的一環。

第三節　結構化文件索引

對於結構化文件的檢索，除了一般的內容檢索(Content Query)之外，亦可提供結構化檢索(Structured Query)。以圖 2 為例，我們可以尋找「TEAM」中是否有名為「Darren」的「PLAYER」，其結構化的檢索條件可以路徑表示法(Path Expression)來表示，如「TEAM.PLAYER.GIVEN_NAME = 'Darren'」。

要達到有效地結構化檢索功能，首先必須要對結構化文件的結構資訊作索引。在結構化文件索引的建立中，如何快速得知結構間的關係則取決於索引結構的設計。針對此點，[Lee96]中所提出的 k-ary Tree 索引結構可利用簡單的數學運算來達到目的。k-ary Tree 乃是一個完整的樹狀結構(Complete Tree Structure)，而 k 代表文件樹狀結構中的最大分支數。[Lee96]配置文件樹狀結構中的每個節點一個「唯一元素識別號(Unique Element Identifier, UID)」，代表該節點在階層式架構中的位置，透過節點的 UID 快速計算後即可得知其父節點或子節點的 UID，進而取得相關資訊。以下茲針對[Lee96]提出之方法做進一步說明。

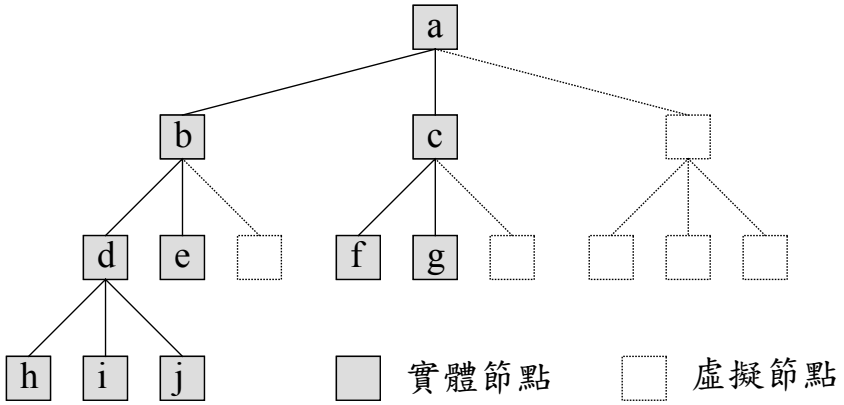

圖 9：3-ary 文件樹狀結構

為了達到結構化檢索，[Lee96]設計了 UID 來包含結構資訊。首先，作者將結構化文件表示成一個 k-ary 的完整樹狀結構，並以文件的最大分支數當作樹狀架構的分支數 k。如圖 9 所示為一個 3-ary Tree，其中每個節點的最大分支數為 3，由實線繪成的方框為實體節點(Real Node)，表示節點資料實際存在原始文件中；相反地，若節點不存在原始文件中，則為虛擬節點(Virtual Node)，以虛線的方框表示之。為維持樹狀結構的完整性，必須將虛擬節點加入一起進行 UID 配置。接著依據階層順序由左至右配置每個節點一個 UID，最後實體節點的 UID 配置結果如表 1 所示。

表 1：UID 配置表

Element	UID	Element	UID
a	1	f	8
b	2	g	9
c	3	h	14
d	5	i	15
e	6	j	16

由於 k-ary Tree 為完整的樹狀結構，故階層架構中父節點與子節點間的關係可透過方程式 1 與方程式 2 快速計算得知。方程式 1 為計算父節點 UID 的公式，方程式 2 為計算子節點 UID 的公式。方程式 1 中的 *i* 為節點本身的 UID，

k 則為此 k-ary 樹狀結構的分支數。在方程式 2 中，i 與 k 的意義如前所述，而 $j(1 \le j \le k)$ 表示該節點為其父節點的第幾個分支。

$$parent(i) = \left\lfloor \frac{i-2}{k} + 1 \right\rfloor$$

方程式 1：計算父節點 UID 之公式

$$child(i, j) = k(i-1) + j + 1$$

方程式 2：計算子節點 UID 之公式

在此我們利用圖 9 與表 1 的範例來說明上述兩個方程式的正確性。由圖 9 我們得知 d 節點為 b 節點的第一個分支，又 d 節點為 h 節點的父節點。我們知道 b 節點的 UID 為 2 即 i = 2，d 節點為 b 節點的第一個分支，因此 j = 1，將 i、j 與 k 代入方程式 2 可得知 d 節點的 UID 為 child(2,1) = 3(2-1) + 1 + 1 = 5。相反的，若我們知道 h 節點的 UID 為 14，透過方程式 1 可計算出其父節點 d 的 UID 為 $parent(14) = \left\lfloor \frac{14-2}{3} + 1 \right\rfloor = 5$，的確符合 d 節點之 UID。

第三章　漸進式結構化索引更新方法之設計

結構化索引能夠輔助使用者快速地從結構化文件中找到符合結構化條件的資訊。當結構資料經常更動時，如何有效率地更新索引，以提升整體檢索效能，乃是本論文的目標。我們於本章中提出一套有效率的漸進式結構化索引更新方法，首先，第一節概要描述系統架構及其運作流程，第二節則詳細描述結構化資訊索引的建置流程，最後會對所提出的漸進式索引更新方法做詳細說明。

第一節　XML 文件檢索系統架構

為了有效率地更新結構化索引，本論文在結構化資料索引中加入了漸進式索引更新，並發展了一套結構化檢索系統雛形，當結構化資料更新時可利用本論文提出的索引更新方法來有效率地更新索引資料，系統架構如圖 10 所示：

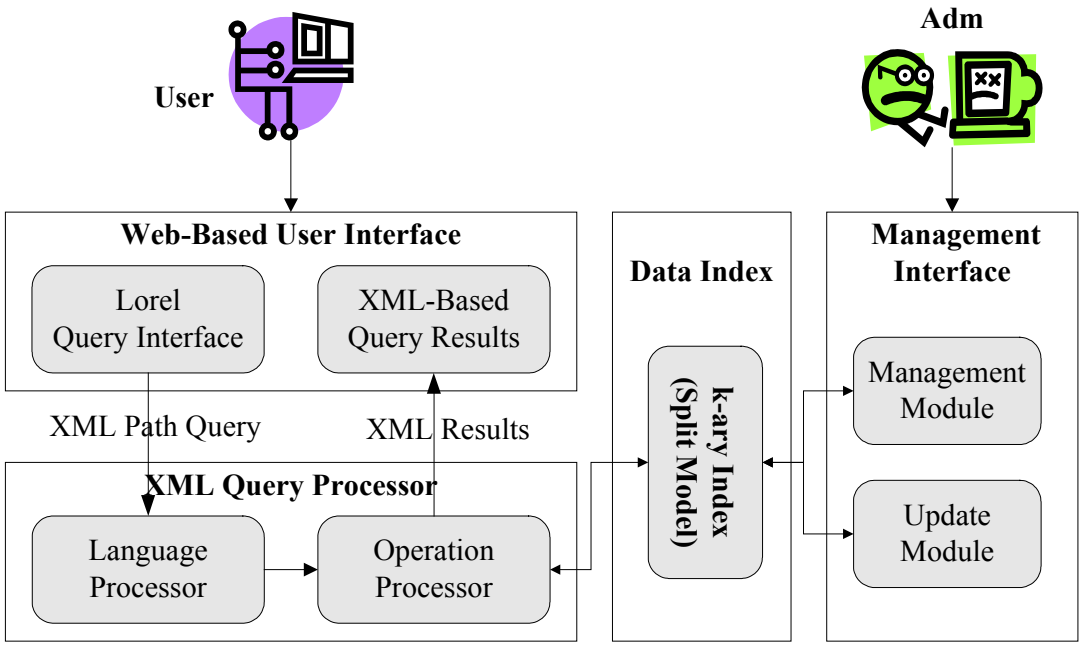

圖 10：XML 文件檢索系統架構圖

使用者在***使用者介面(Web-Based User Interface)***輸入 SQL-like 的路徑表示式檢索條件，檢索條件會被傳送到***路徑檢索處理器(XML Query Processor)***。路徑檢

索處理器中的***語言處理器(Language Processor)***會將路徑檢索條件轉換成對應的資料存取機制，接著***資料操作處理器(Operation Processor)***會處理各個資料存取機制，並從***資料索引(Data Index)***中的 k-ary 樹狀結構找出符合路徑條件的資料，並將結果以 XML 的方式呈現給使用者。

資料管理員則可透過***管理模組(Management Module)***來決定要對那些 XML 文件建立 k-ary Tree 結構化資訊索引。另外，透過***更新模組(Update Module)***可讓資料管理員來新增、刪除或更新 XML 中的資料，同時也會更新資料索引中對應的索引資料。

下節將詳細說明結構化資料索引的建置流程與本論文所使用的內部資料結構。

第二節　XML 結構化資訊索引建置流程

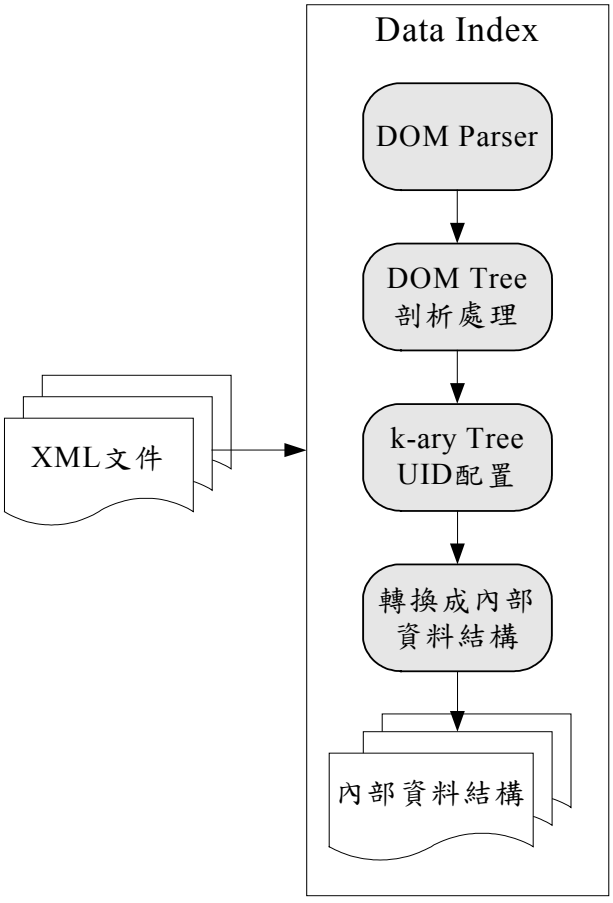

圖 11：系統索引建置流程圖

XML 結構化資訊索引建置流程如圖 11 所示，其目的在於將 XML 文件的結構資訊粹取出來建構成 k-ary 樹狀結構，透過此樹狀表示法提供給使用者進行結構化檢索。本論文中的 XML 文件屬於以資料為主的 XML 文件。以下詳細說明結構化資訊索引的建置步驟：

1. DOM Parser：先將 XML 文件的階層式架構轉換成文件樹狀結構。在此我們使用 Apache Xerces DOM Parser 來做為結構文件剖析處理的工具，先利用 Xerces DOM Parser 將 XML 文件轉成一個 DOM 樹狀結構。
2. DOM Tree 剖析處理：由於步驟 1 中的文件樹狀結構會包含一些非資料節點，我們在本步驟中將之刪除。
3. k-ary Tree UID 配置：利用 k-ary 樹狀結構的概念來對文件樹狀結構做深度優先搜尋法(Depth First Search)，剖析出每個節點的資訊並配置 UID。UID 配置演算法如圖 12 所示。
4. 轉換成內部資料結構：將剖析過的節點資訊存於本論文使用的內部資料結構中。

```
pUID = 1;
Level = 1;
Function Traverse (Node p, long pUID, int level) {
   NodeList children = p.getChildNodes( );
   k = children.getLength( );  //k 代表節點 p 的子節點個數
   UID = 0;
   //對節點 p 下的每個子節點再往下層配置 UID
   For (I = 0; I < k; I++) {
        Node child = children.item(I);
        NodeList childNodeNo = child.getChildNodes( );
        childno = childNodeNo.getLength( );
        UID = MaxBranch * (pUID –1) + (I+1) + 1;
        Traverse (child, UID, level+1);
  }
}
```

圖 12：UID 配置演算法

圖 13 為圖 2 XML 球隊範例資料中一個球員的資料，我們以此 XML 簡例來詳細說明配置節點 UID 的過程。

```
<?xml version="1.0"?>
<DIVISION>
    <TEAM>
        <TEAM_CITY>Los Angeles</TEAM_CITY>
        <TEAM_NAME>Dodgers</TEAM_NAME>
        <PLAYER>
            <SURNAME>Darren</SURNAME>
            <G_NAME>Hung</G_NAME>
            <POSITION>Starting Pitcher</POSITION>
        </PLAYER>
   </TEAM>
</DIVISION>
```

圖 13：XML 文件簡例

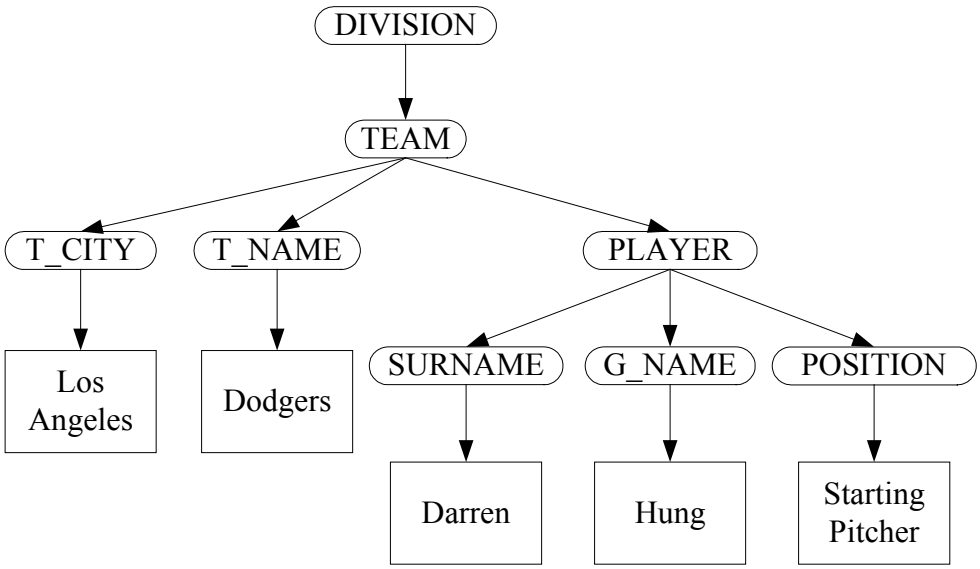

圖 14：圖 13 文件簡例之 DOM 樹狀結構

圖 13 的 XML 文件會先經由 XML Parser 轉換成圖 14 的 DOM 樹狀結構，其中方框的節點表示實際資料內容，其餘節點為標籤元素的資訊。轉成 DOM 之後，再根據每個節點在文件中的階層位置取出其對應的資料並將之轉存到 k-ary

Tree 中。在此XML 範例中「DIVISION」為根節點、「TEAM」為其子節點而「TEAM」節點下又有其所屬的子節點與資料。依據這樣的關係，XML Parser 從根節點開始做深度優先搜尋法並配置 UID。由圖 14 的文件樹狀結構可經由 XML Parser 分析出文件樹的最大分支為 3，接著做深度優先法則走訪文件樹中的每一個節點並將之存到 3-ary Tree 的對應位置。

依據 k-ary Tree 的特性，若取得一個節點的 UID 即可依據其子節點所在的分支位置，透過計算子節點的公式 child(i, j) = k(i-1) + j +1 來算出子節點的 UID，其中 i 代表此節點的 UID，j 為子節點所在的分支位置，而 k 為 k-ary Tree 的最大分支數。以「PLAYER」節點為例，其 UID 為 7，則其第一個子節點「SURNAME」的 UID 為 3(7-1) + 1 + 1 = 20。由此方式即可配置完所有節點的 UID，以圖 13 的 XML 文件為例，則其 UID 配置完的結果如表 2 所示。

表 2：UID 配置結果

Element	UID	Element	UID
DIVISION	1	SURNAME	20
TEAM	2	Darren	59
T_CITY	5	G_NAME	21
Los Angeles	14	Hung	62
T_NAME	6	POSITION	22
Dodgers	3	Starting Pitcher	65
PLAYER	7		

配置完所有的 UID 後，本論文使用(SeqId, DID, UID, Level, ElementType, Value, ChildNo)的資料結構來將所有分析過的節點資訊儲存起來，資料結構中各個欄位資訊如表 3 所示：

表 3：內部資料結構中各欄位資訊

欄位資訊	欄位意義
SeqId	每個節點的流水編號
DID	XML 文件識別號
UID	唯一元素識別號
Level	節點位於 k-ary Tree 中的層級
ElementType	k-ary Tree 中的節點型態
Value	k-ary Tree 中節點對應的資料
ChildNo	節點擁有的子節點個數

以圖 14 為例，最後真正儲存的索引資料如表 4 所示：

表 4：索引資料表

SeqId	DID	UID	Level	ElementType	Value	ChildNo
1	1	1	1	Root	DIVISION	1
2	1	2	2	Element	TEAM	3
3	1	5	3	Element	T_CITY	1
4	1	14	4	Text	Los Angeles	0
5	1	6	3	Element	T_NAME	1
6	1	17	4	Text	Dodgers	0
7	1	7	3	Element	PLAYER	3
8	1	20	4	Element	SURNAME	1
9	1	59	5	Text	Darren	0
10	1	21	4	Element	G_NAME	1
11	1	62	5	Text	Hung	0
12	1	22	3	Element	POSITION	1
13	1	65	5	Text	Starting Pitcher	0

第三節　漸進式結構化索引更新(Incremental Structured Index Update)

當一份結構化文件的結構資訊有變更時，一個好的結構化索引機制應能有效地同步更新索引資訊。我們將索引更新分為三種情況：更新節點內容、刪除節點與新增節點。當資料更新造成 k-ary Tree 的最大分支數 k 值改變時，若每次超過 k 值皆須重建索引以維護節點 UID 的正確性，則將耗費大量時間來維護索引的正確性，尤其當索引資料量很大時，更凸顯出索引重建的缺點。有鑑於此，漸進式

地更新結構化索引以降低維護索引的時間乃有其必要性。以下乃針對這三種狀況來詳細描述資料更新時，索引資料採取的對應更新動作。

3.3.1 更新節點內容(Update Element Content)

只需從索引資料中找出舊有資料的所在位置，並以新的資料替代即可完成索引更新。以圖 13 的 XML 範例來說，一個球員在球隊中原本擔任的角色為「Starting Pitcher」，若將其角色改為「Relief Pitcher」，那麼索引更新時，必須先從索引資料中找出「Starting Pitcher」的對應 DID 與 UID，從表 4 中可以得知文字節點(Text Node)—「Starting Pitcher」的 DID = 1 而 UID = 65，接著再透過 DID 與 UID 將「Starting Pitcher」以新的資料「Relief Pitcher」替代。

以此類推，若改變的資料是元素節點的節點資訊，亦是先從索引資料中找出其對應的 DID 與 UID 後，再透過 DID 與 UID 將舊的節點資料以新的資料替代。這種類型的資料更新並不會改變原有 k-ary Tree 的最大分支數，因此是資料更新中最簡單的一類。

3.3.2 刪除節點(Delete Element)

當欲刪除節點資料時，如同更新節點資料，必須先由索引資料中找出欲刪除節點的所在位置，之後再將該節點及其所有下屬節點一併從索引資料中刪除。同樣以圖 13 的 XML 範例來說，當欲將球員(PLAYER)的資料從球隊中刪除時，透過表 4 得知球員的 DID = 1 與 UID = 7，而其下屬節點的資訊可從球員的 DID、UID 與該球員所擁有的子節點個數來一一找出，進而將欲刪除的節點資料從索引資料中刪除。

刪除節點的動作雖不會造成節點分支數超過最大分支數 k，但會造成 k-ary 樹狀結構中該節點後面的兄弟節點(Sibling Nodes)及各兄弟節點的下屬節點需往前遞移才能維持完整樹的特性，因為需將 UID 往前遞補以彌補刪除節點之

缺。因此，當欲刪除的節點位於樹狀結構中越上層且越靠左的分支位置，則刪除節點時所需重新計算 UID 的節點越多，相對地所花的時間越長。

為減少刪除節點所花的時間，本論文使用一個欄位資訊 Delete 來記錄節點是否已被刪除，若已被刪除則 Delete 為 1，否則為 0，而在資料搜尋時會利用此資訊判斷該節點是否已被刪除，若節點已被刪除則此節點資訊不會被挑選出來。待日後需索引資料重建時，再將這些刪除的節點確實刪除。為此，我們將原始的索引資料結構修改為(SeqId, DID, UID, Level, ElementType, Value, ChildNo, Delete)，其中新增 Delete 欄位來記錄節點為實體節點或虛擬節點，實體節點代表節點未被刪除，虛擬節點則表示節點已被刪除。以表 4 的索引資料表為例，假設欲刪除名為「Darren」的球員，則刪除該節點後的索引資料表如表 5 所示。由表 5 可看出球員節點下的所有下屬節點亦會被刪除，因此 Delete 皆為 1。

表 5：包含 Delete 資訊之索引資料表

SeqId	DID	UID	Level	ElementType	Value	ChildNo	Delete
1	1	1	1	Root	DIVISION	1	0
2	1	2	2	Element	TEAM	3	0
3	1	5	3	Element	T_CITY	1	0
4	1	14	4	Text	Los Angeles	0	0
5	1	6	3	Element	T_NAME	1	0
6	1	17	4	Text	Dodgers	0	0
7	1	7	3	Element	PLAYER	3	1
8	1	20	4	Element	SURNAME	1	1
9	1	59	5	Text	Darren	0	1
10	1	21	4	Element	G_NAME	1	1
11	1	62	5	Text	Hung	0	1
12	1	22	3	Element	POSITION	1	1
13	1	65	5	Text	Starting Pitcher	0	1

雖然利用此方法來維護結構索引，每個節點需比實際將節點刪除時多一個 Delete 欄位的儲存空間，且還需儲存每個被刪除的節點。但在索引更新時，只需

花費常數時間(Constant Time)即可標示已被刪除的節點。

透過上述方法使得這類的資料更新亦不會改變 k-ary Tree 的最大分支數，並能夠很快地執行索引資料更新。

3.3.3 新增節點(Insert Element)

針對新增節點的情形，由於新增的節點可能影響到 k-ary Tree 的最大分支數，因此新增節點時必須分成兩種情況來討論。

首先，要新增資料的節點稱之為「被插入節點」。在插入新增節點之初，先透過索引資料取得被插入節點的 DID、UID 與 ChildNo 三個資訊，由 ChildNo 可得知被插入節點是否已經含有 k 個子節點。根據被插入節點是否已含有 k 個子節點，新增節點時相對應的動作如下：

1. 插入節點時，最大分支數不會超出 k：直接將新增的節點加入索引資料中。此時，新增節點之 DID 即為被插入節點的 DID，而 UID 可透過公式 child(i, j) = k(i-1) + j +1 計算得知，其中 i 為被插入節點的 UID，j 為 ChildNo + 1 表示新增節點為被插入節點的第 ChildNo + 1 個子節點。
2. 插入節點時，最大分支數會超出 k：當我們由 ChildNo 發現，若加入節點後會使得索引結構的最大分支數超過 k 時，表示欲新增之節點的 UID 已無法在原有的 k-ary 完整樹狀結構中透過被插入節點的 UID 計算出來。此時，最簡單的方法是將新增的資料加入原本的 XML 檔後，再重新透過 XML Parser 剖析整個 XML 文件並配置適當的 UID，以建構一個新的 k-ary Tree。

雖然我們將資料更新分成三種情形來討論，但前述兩類在資料變更時並不會影響到整個索引結構的最大分支數。而第三類中，若新增節點後造成 k-ary Tree 的最大分支數超出 k 值，此時採取重新建立索引結構的步驟會導致索引維護的整

體效率大打折扣，尤其是當 XML 檔案很大且經常發生資料變更時。針對此點，本論文提出一個漸進式的索引更新方法，稱之為 Split Method，來避免上述情況。

3.3.4 漸進式索引更新方法 — Split Method

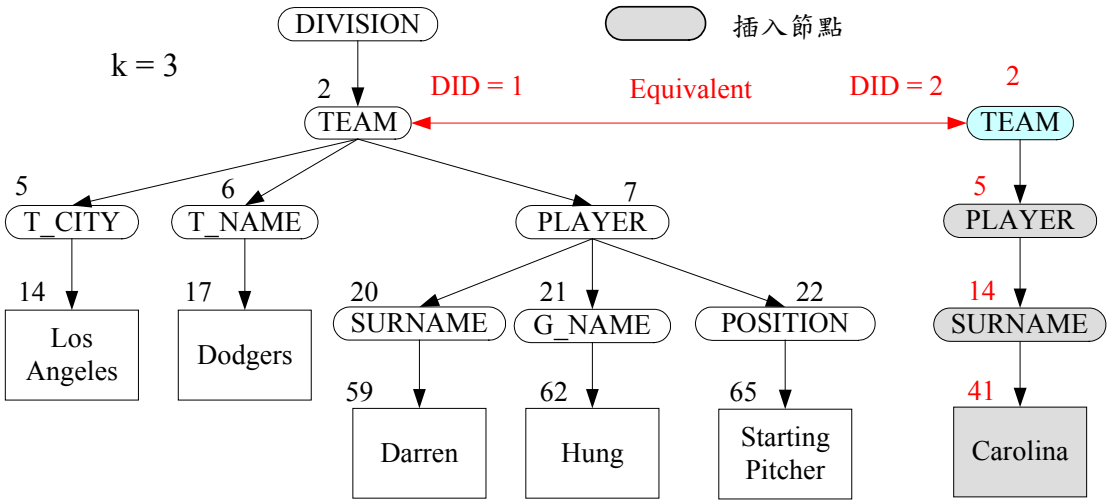

圖 15：在圖 14 的 TEAM 節點下新增一筆 PLAYER 資料時的分裂結果

顧名思義，Split Method 是指新增節點後，若造成 k-ary Tree 的最大分支數超過 k，則從被插入節點 A 處分裂出一個新的節點 A'。同時，定義一個對等關係(Equivalent Relation)來記錄兩個節點間的關聯性，並利用 DID 的概念視分裂出來的節點 A'及其底下的子節點為另一個新的 XML 文件之對應 k-ary Tree。為表示對等關係，我們在索引結構中新增 EQF (Equivalent Forward)與 EQB (Equivalent Back)兩個資訊，以前述 A 和 A'而言，A 的 EQF 為 A'，表示節點 A 分裂出來的節點為 A'，A'的 EQB 為 A，表示節點 A'乃從節點 A 分裂出來。在此即以圖 13 的 XML 範例來說明。在節點「TEAM」下若新增一名「PLAYER」其「SURNAME」為「Carolina」時，會導致節點「TEAM」下的分支數超過最大分支數 k = 3，則經過 Split Method 分裂後的結果如圖 15 所示。

圖 15 中灰底的節點代表新增之節點，由於「SURNAME」為「Carolina」的「PLAYER」插入「TEAM」後造成此節點的分支數超過 k，因此被插入的節

點「TEAM」隨即分裂出一個對等的節點，即圖中右手邊的『TEAM』節點。在此，被插入的「TEAM」節點所屬的 DID = 1，分裂出來的『TEAM』節點所屬的 DID = 2，至於「TEAM」與『TEAM』節點的 UID 皆為 2。對於新增的「PLAYER」節點其父節點即為分裂出來的『TEAM』節點，該節點的 DID 與其父節點同為 2，而 UID 則依照方程式 2 透過其父節點之 UID 與其所在之分支位置與最大分支數計算得知為 5。以此類推可得知元素節點「SURNAME」與文字節點「Carolina」之 UID 分別為 14 與 41，而兩節點之 DID 亦為 2。

表 6：Split Method 下的索引資料表

SeqId	DID	UID	Level	ElementType	Value	ChildNo	Delete	EQF	EQB
1	1	1	1	Root	DIVISION	1	0	0	0
2	1	2	2	Element	TEAM	3	0	14	0
3	1	5	3	Element	T_CITY	1	0	0	0
4	1	14	4	Text	Los Angeles	0	0	0	0
5	1	6	3	Element	T_NAME	1	1	0	0
6	1	17	4	Text	Dodgers	0	0	0	0
7	1	7	3	Element	PLAYER	3	1	0	0
8	1	20	4	Element	SURNAME	1	1	0	0
9	1	59	5	Text	Darren	0	1	0	0
…									
14	2	2	2	Element	TEAM	1	0	0	2
15	2	5	3	Element	PLAYER	1	0	0	0
16	2	14	4	Element	SURNAME	1	0	0	0
17	2	41	5	Text	Carolina	0	0	0	0

由於 Split Method 定義了被插入節點與分裂節點間的對等關係，此對等關係必須定義在索引結構中，因此我們將原本索引的資料結構修改成(SeqId, DID, UID, Level, ElementType, Value, ChildNo, Delete, EQF, EQB)。

對於每個節點的 EQF 與 EQB 其初始值皆為 0，當節點 A 發生分裂時，EQF 會紀錄分裂節點 A'的 SeqId，而分裂節點 A'的 EQB 則會紀錄被插入節點的

SeqId。Split Method 下的索引資料表如表 6 所示。從表 6 中可看出 SeqId = 14 的『TEAM』節點，其 EQB = 2，表示此節點是從 SeqId = 2 的「TEAM」節點分裂出來的。相反地，我們也可以從 SeqId = 2 的「TEAM」節點之 EQF = 14 得知分裂節點為 SeqId = 14 的節點。

第四章　路徑表示式檢索實作說明

為了支援 XML 文件檢索，在眾多的 XML 檢索語言中，我們以 Lorel 檢索語言為基礎，透過第三章提出的索引機制來達到 XML 文件檢索，並將符合條件之節點資料以 XML 文件的方式呈現給使用者。本章即針對我們實作的檢索功能詳細說明如何達到路徑表示式檢索。圖 16 為資料搜尋演算法。

```
1    Function StartQuery (String select, String from, String where){
2      //where 條件由 queryPath + queryOperator + queryText 組成
3      queryPath = GetQueryPath(where);
4      queryOperator = GetQueryOperator(where);
5      queryText = GetQueryText(where);
6      //找出所有滿足 queryText 的 Text 節點
7      Vector nodesSeqId = SelectQueryText(from, queryOperator, queryText);
8      //找出 select 與 queryPath 兩個路徑條件間相同 subPath 中的外部節點
9      elmSelectPattern = GetExternalElement(FindComSubPath(select, queryPath));
10
11     For ( I = 0; I < NumberOfNodesSeqId; ){
12       //檢查所有滿足 queryText 的節點是否滿足 queryPath
13       //取出滿足 queryText 條件之 Text 節點的 DID 與 UID
14       ResultSet node = SelectNode(nodesSeqId(I));
15       DID = node.getInt("DID");
16       cUID = node.getBigDecimal("UID");
17       //透過節點的 UID 來找出其父節點的 UID
18       pUID = (cUID-2)/MaxBranch + 1;
19       //開始檢查是否滿足 queryPath
20       PathCheck(DID, pUID, queryPath, elmSelectPattern);
21       I++;
22     }
23   }
```

圖 16：資料搜尋演算法

進行資料檢索前，檢索系統會先取得使用者輸入的 SQL-like 路徑檢索條件，如圖 17 範例所示。進行資料檢索時，系統將路徑檢索條件中的 Select、From 與

Where 這三部分的條件資料剖析出來，而這三個部分所代表的意義如下：

```
Select SEASON.LEAGUE.DIVISION.TEAM.PLAYER
From SourceUID
Where SEASON.LEAGUE.DIVISION.TEAM.PLAYER.G_NAME = ‘Darren’
```

圖 17：檢索條件範例

- Select：欲挑選出來的節點資訊。

範例中的 Select 路徑條件「SEASON.LEAGUE.DIVISION.TEAM.PLAYER」表示要挑選出 PLAYER 節點的資訊，其中 PLAYER 節點須從節點 SEASON 依路徑條件經過 LEAGUE 等節點到達。圖 16 資料搜尋演算法中的第 9 行將欲挑選出來的節點資訊存到 elmSelectPattern 的字串變數中，待後續路徑條件比對時查看路徑條件中是否有出現欲挑選的節點，在滿足路徑條件的情況下，將符合條件且為欲挑選出的節點之 SeqId 紀錄下來，以利後續產生 XML 結果。

由於某些情況下，使用者欲挑選的節點不見得會出現在 Where 的路徑條件下，因此，可先從 Select 與 Where 兩路徑條件中找出共同的最長路徑，並視其中最外部的節點為使用者欲挑選的節點。此例中兩條件下的最長共同路徑剛好為 Select 的路徑條件「SEASON.LEAGUE.DIVISION.TEAM.PLAYER」，因此，最外部的節點「PLAYER」為欲挑選的節點。

- From：從哪個索引資訊中進行資料搜尋。

From 條件中的 SoureUID，即代表某個 XML 檔案對應的索引資料表。因此，資料搜尋時只需到 SourceUID 進行路徑條件比對即可，以增進搜尋的效能。

- Where：路徑檢索條件。

Where 檢索條件乃由路徑條件(queryPath)、運算元(queryOperator)與檢索條件值(queryText)三個部分所組成。以圖 17 的檢索條件為例，其中的路徑檢索條件如下：

「SEASON.LEAGUE.DIVISION.TEAM.PLAYER.G_NAME = ‘Darren’」

此例中，運算元為「=」，檢索條件值為「Darren」而路徑條件為

「SEASON.LEAGUE.DIVISION.TEAM.PLAYER.G_NAME」。

從上述Select、From與Where三條件中剖析出需要的資訊後，首先利用Where條件中的運算元與檢索條件值到From條件的SourcdUID取出滿足檢索條件值的文字節點，接著再透過迴圈來比較每個符合檢索條件值的文字節點是否滿足路徑條件。此部分的演算法如圖 16 所示，其中第 20 行 PathCheck(DID, pUID, queryPath, elmSelectPattern)代表的意思為進行文字節點之上層節點路徑檢查。對於路徑條件檢索，可分為 Simple Path Query 與 General Path Query 兩類，接著即對這兩類檢索做更詳細說明。

第一節　簡易路徑檢索(Simple Path Query)

此類檢索在於使用者可以限定一個明確的檢索路徑從某個節點經過樹狀結構中連續的節點到達另一個節點。在此模式下支援了兩種搜尋：精確搜尋(Exact Search)與近似搜尋(Proximate Search)。

- 精確搜尋：文字節點的資料須與檢索條件值完全符合且路徑條件的運算元為「=」。舉例來說，路徑檢索條件若為

  「SEASON.LEAGUE.DIVISION.TEAM.PLAYER.G_NAME = ‘Darren’」，

  表示某個球員的名字必須為「Darren」才算符合路徑檢索條件。

- 近似搜尋：檢索條件值只需出現於文字節點中即符合路徑檢索條件，且路徑條件的運算元為「LIKE」。依檢索條件值出現於文字節點的位置又可分成三種情況：

  - 出現於任意位置，以*%檢索條件值%*表示。如：

    「SEASON.LEAGUE.DIVISION.TEAM.PLAYER.G_NAME LIKE

‘%John%’」。

- 出現於字首，以*檢索條件值%*表示。如：

「SEASON.LEAGUE.DIVISION.TEAM.PLAYER.G_NAME LIKE ‘John%’」。

- 出現於字尾，以*%檢索條件值*表示。如：

「SEASON.LEAGUE.DIVISION.TEAM.PLAYER.G_NAME LIKE ‘%John’」。

第二節　一般路徑檢索(General Path Query)

這類的檢索支援了萬用字元(Wild Cards)與常規表示式(Regular Expressions)，因此在路徑條件的檢索上比 Simple Path Query 來的強大。對於萬用字元，我們實作了「%」與「#」的功能，其中「%」的意義如同 Simple Path Query 的近似搜尋，但情況發生於元素節點上，如圖 13 中的節點 PLAYER 下有兩個子節點為 SURNAME 與 G_NAME，當路徑條件下某個節點為%NAME 時，則 SURNAME 與 G_NAME 皆會被挑出以進行路徑比對。而另一個符號「#」則表示從某個節點開始，中間可經過 0 個以上的節點以到達另一個節點，「#」符號常用在使用者不知道文件結構的情況下。如路徑條件「SEASON.#.PLAYER.G_NAME」則表示從節點「SEASON」開始，不管中間經過幾個節點，只要最後節點路徑為「PLAYER.G_NAME」即滿足條件。

至於常規表示式，我們實作了 Optional Node —「(.node)?」的功能，表示路徑中可出現 0 次或 1 次的「node」節點。舉例來說，若路徑條件為「PLAYER(.NAME)?.G_NAME」，則進行路徑檢索時，須比對兩個路徑「PLAYER.NAME.G_NAME」或「PLAYER.G_NAME」是否滿足條件。下列即對「#」、「%」與「(.node)?」三種功能之演算法做說明。

- 「#」：演算法如圖 18 所示。

```
IF (pathNodeElement = "#" ){
  //取出 queryPath 中 wildcard 前的 Element 節點資訊
  nodeElementBfWildCard = GetElementBfWildCard(queryPath);

  //透過 DID 與 UID 來找出其父節點資訊，直到父節點與WildCard 前
  的節點資訊相同時再做最後的 PathCheck
  While (nodeValue != nodeElementBfWildCard){
       IF (nodeValue= elmSelectPattern){  //判斷是否為欲挑選的節點
          //若 matchedEleSeqId 中沒出現過 node SeqId
          //則將 nodeSeqId 加入 matchedEleSeqId
          IF (!CheckContain(matchedEleSeqId, nodeSeqId))
               matchedEleSeqId.addElement(nodeSeqId);
       }
       pUID = (UID-2)/MaxBranch + 1;
       ResultSet node = SelectNode(DID, pUID);
       nodeSeqId = node.getBigDecimal("seqId");
       nodeValue = node.getString("value");
  }
  subPath = GetSubQueryPathBfWildCard(queryPath);
  PathCheck(DID, pUID, subPath, elmtPattern);
}
```

圖 18：WildCardOperation("#") 路徑檢查演算法

若路徑條件為「SEASON.LEAGUE.#.G_NAME」，當進行路徑比對中發現目前的節點為「#」時，此時將萬用字元前的節點以 nodeElementBfWildCard 記錄之，接著遞迴進行上層父節點比對直到該節點為萬用字元前的節點後，再做最後的路徑比對以檢查是否滿足路徑條件。往上層父節點尋找的過程中，若有出現欲挑選的節點時，則須將節點的 SeqId 記錄之。以前述之路徑條件而言，以下即為處理符號「#」的步驟：

1. 開始進行路徑比對時，由滿足檢索條件值的 UID 得知其父節點為 G_NAME，路徑條件中最外部的節點確實為 G_NAME。滿足路徑條件後繼續進行上一層路徑「SEASON.LEAGUE.#」比對。

2. 出現符號「#」，此時 nodeElementBfWildCard 設為 LEAGUE。接著由節點 G_NAME 之 UID 往上層父節點推導，直到節點為 LEAGUE 止。

3. 接著，再由節點 LEAGUE 進行更上層路徑「SEASON.LEAGUE」比對。

- 「%」：演算法如圖 19 所示。

```
IF (percentOccur){
  IF (percentAtHead) {
      //WildCard %出現在 Element 首與尾，如%NAME%
    pathNodeElement = RemovePercentAtHead(pathNodeElement);
    IF (pathNodeElement End With percent) {
        pathNodeElement = RemovePercentAtTail(pathNodeElement);
        IF (nodeValue Contains Substring pathNodeElement){
      //目前節點資料滿足路徑條件則繼續遞迴比較上一層父節點是否
      亦滿足
            pUID = (UID-2)/MaxBranch + 1;
            subPath = GetSubQueryPath(queryPath);
            PathCheck(DID, pUID, subPath, elmtPattern);
          }
      } ELSE IF (nodeValue Ends With Substring pathNodeElement){
        //WildCard %出現在 Element 首，如%NAME
        //目前節點資料滿足路徑條件則繼續遞迴比較上一層父節點是
        否亦滿足
          pUID = (UID-2)/MaxBranch + 1;
          subPath = GetSubQueryPath(queryPath);
          PathCheck(DID, pUID, subPath, elmtPattern);
      }
    } ELSE {
      //WildCard %出現在 Element 尾，如 NAME%
    pathNodeElement = SubStringRemoveWildCard(pathNodeElement);
      IF (nodeValue Begins With pathNodeElement) {
      //目前節點資料滿足路徑條件則繼續遞迴比較上一層父節點是否
      亦滿足
        pUID = (UID-2)/MaxBranch + 1;
        subPath = GetSubQueryPath(queryPath);
        PathCheck(DID, pUID, subPath, elmtPattern);
      }
    }
}
```

圖 19：WildCardOperation("%") 路徑檢查演算法

路徑比對的過程中，若路徑條件中的節點出現符號「%」時，只有當該節點滿足近似搜尋所述的三種情況下，才會繼續進行上一層的路徑條件比對。舉例而言，路徑條件若為「DIVISION.PLAYER.%NAME」，則開始進行路徑比對時，由滿足檢索條件值的 UID 得知其父節點為 G_NAME，該節點確實由 NAME 結尾。因此滿足路徑條件接著繼續進行上一層路徑「DIVISION.PLAYER」比對。

- 「(.node)?」：演算法如圖 20 所示。

```
IF (questionMarkOccur){      //出現 Optional 的情況，如 a(.b)?.c
   //分兩種情形繼續遞迴搜尋
   subPath = GetPathContainOptionalNode(queryPath);
   PathCheck(DID, pUID ,subPath, elmtPattern); //Optional 的節點有出現
   subPath = GetPathNotContainOptionNode(queryPath);
   PathCheck(DID, pUID, subPath, elmtPattern); //Optional 的節點沒有出現
}
```

圖 20：OptionalNodeOperation( ) 路徑檢查演算法

當 Optional Node 的條件出現時，路徑條件將分成包含 Optional Node 與不包含 Optional Node 的情況繼續進行上一層的路徑比對。舉例而言，假設路徑條件為「DIVISION.PLAYER(.NAME)?.G_NAME」，當比對到節點 G_NAME 時，發現路徑條件中出現 Optional Node –「NAME」。此時，路徑條件可為兩種情形，一條路徑為包含 Optional Node，即「DIVISION.PLAYER.NAME.G_NAME」，另一條則否，即「DIVISION.PLAYER.G_NAME」。接著再以這兩種路徑條件個別進行路徑比對。

第三節　路徑條件檢查

路徑檢查是整個路徑條件檢索最核心的部分，檢索系統開始之初透過索引資料表找出滿足檢索條件值的文字節點，接著對每個滿足條件的文字節點進行路徑條件比對，路徑條件檢查乃透過遞迴的方式，當目前的路徑節點符合索引資料表中的元素節點資訊時，再繼續進行上一層路徑條件檢查。圖 22 所示的路徑條件檢查演算法為整合前兩節所述之 Simple Path Query 與 General Path Query 下的各種狀況。當前述的萬用字元(%、#)或常規表示式((.node)?)的情形出現時，則會採取對應的處理，如圖中的第 14 行會處理萬用字元「#」出現的情況。

圖 22 的演算法乃針對 Split Method 下所使用的索引資料結構而設計，若要適用於原始的資料結構，只須將圖中第 8 行處的 FindSplitedNode( )函數移除即可。FindSplitedNode( )函數是在節點分裂的狀況下，進行路徑條件檢查時，透過對等關係中的 EQB 來找出被插入之節點所在位置，從而進行其父節點資訊之比對。對應的演算法如圖 21 所示。

```
EQB = node.getBigDecimal("EQB");
//EQB 不為 0 表示此節點為"分裂"出來的節點，因此需透過 EQB 找出其對等的節點
While (EQB != 0){
      nodeSeqId = EQB;
      ResultSet node = SelectSeqId(EQB);
      DID = node.getInt("DID");
      EQB = node. getBigDecimal("EQB");
}
```

圖 21：FindSplitedNode( ) 分裂節點搜尋演算法

```
1  Vector matchedElmtSeqId;
2  Function PathCheck (int DID, long UID, String queryPath, String elmSelectPattern){
3    //根據 DID 與 UID 找出對應的 Element 節點
4   ResultSet node = SelectNode(DID, UID);
5    //從找到的 Element 節點中取出其 sequence Id 與節點資訊
6    nodeSeqId = node.getBigDecimal("seqId");
7    nodeValue =node.getString("value");
8    FindSplitedNode( );   //執行圖 21 的演算法
9    //從 queryPath 中由外層的節點往根部節點取出節點資訊以便進行路徑比對
10  IF (queryP  ath Is Not To Root){   //queryPath 的節點並未到根節點
11     //取得 queryPath 的最外部節點資訊
12     pathNodeElement = GetExternalElement(queryPath);
13     IF (pathNodeElement = = "#"){
14        WildCardOperation("#");   //執行圖 18 的演算法
15     }
16     IF (percentOccur){
17        WildCardOperation("%");   //執行圖 19 的演算法
18     } ELSE {
19        //Element 節點未出現 Wild Card %，則以 Original 的方式進行路徑條件比對
20        IF (pathNodeElement = = nodeValue){
21           //目前節點資料滿足路徑條件則繼續遞迴比較上一層父節點是否亦滿足
22           IF (nodeValue= elmSelectPattern){     //判斷是否為欲挑選的節點
23            CheckAddNodeSeqIdOrNot(  matchedEleSeqId, nodeSeqId);
24           }
25           pUID = (UID-2)/MaxBranch + 1;
26           IF (questionMarkOccur){     //出現 Optional Node 的情況，如 a(.b)?.c
27            OptionalNodeOperation( );     //執行圖 20 的演算法
28           } ELSE {
29            subPath = GetSubQueryPath  (queryPath);
30            PathCheck(DID, pUID, subPath, elmSelectPattern);
31           }
32        } ELSE {     //路徑節點資料不等於目前節點資料，則表示路徑條件不滿足
33           pathMatch = 0;      //queryPath Not Match!!
34        }
35     }
36   } ELSE { // queryPath  的節點到了根節點
37    IF (pathNodeElement =   = nodeValue)    {
38          IF (nodeValue= = elmSelectPattern){      //判斷是否為欲挑選的節點
39            CheckAddNodeSeqIdOrNot(matchedEleSeqId, nodeSeqId);
40          }
41          pathMatch = 1;   //queryPath Match!!
42     }
43    ELSE
44          pathMatch = 0;   //queryPath Not Match!!
45   }
46 }
```

圖 22：路徑條件檢查演算法

第四節　XML 輸出結果

路徑條件比對的過程中，我們將符合路徑條件且是使用者欲挑選的節點之 SeqId 記錄下來，最後再透過圖 24 的 XML 輸出演算法從索引資料表中將節點的資料以 XML 文件格式輸出。若在原本的資料結構下要輸出 XML 檢索結果，只須將圖 24 中的 FindSplitNode( )函數刪除即可達到目的。FindSplitNode( )函數的功能在於當欲從被插入節點的地方找出分裂的節點時，只需透過對等關係中的 EQF 即可找到分裂的節點，進而將其資料以 XML 輸出。FindSplitNode( )的演算法如圖 23 所示。

```
EQF = node.getBigDecimal("EQF");
//EQF 不為 0 表示此節點有"分裂"的節點，因此需透過 EQF 找出由此
分裂出的節點，再繼續將底下的子節點資料取出
While (EQF != 0){
      ResultSet node = SelectSeqId(EQF);
      DID = node.getInt("DID");   //分裂出來的節點 DID 與原本不同
      childNo = node.getInt("childrenno");
      EQF = node.getBigDecimal("EQF");
      sOut = getChildData(sOut, DID, pUID, treeLevel++, childNo);
}
```

圖 23：FindSplitNode( ) 分裂出之節點搜尋演算法

```
Function produceXML(VectormatchedEleSeqId){
   String sOut = "<?xml version=\"1.0\" encoding=\"BIG5\"?>;
      SOut += " <QueryResults>";
      treeLevel = 1;

      //將欲顯示的節點透過迴圈一一將其資料與其子節點的資料全部找出
      For ( I = 0; I < NumberOfMatchedEleSeqId; I++ ){
            ResultSet node = SelectNode(matchedEleSeqId[I]);
            DID = node.getInt("DID");
            UID = node.getBigDecimal("UID");
            nodeValue = node.getString("value");
            childNo = node.getInt("childrenno");
            sOut += "<" + nodeValue + ">";
            sOut = getChildData(sOut, DID, UID, treeLevel++, childNo);
            FindSplitNode();
            sOut += "</"+ value+ ">";
      }
      sOut += "</QueryResults>";
}

Function getChildData(String sOut, int DID, long pUID, int treeLevel, int childNo){
      levelNow = treeLevel;

      For ( I = 1; I <= childNo; I++){
            childUID = MaxBranch*(pUID-1) + I +1;
            ResultSet node = SelectNode(DID, childUID);
            elementType =node.getString("elementType");
            nodeValue =node.getString("value");
            childrenno = node.getInt("childrenno");

            IF (elementType= "TextNode"){
            sOut += nodeValue;
         }    ELSE {
               sOut += "<"+ nodeValue + ">";
               sOut = getChildData(sOut, DID, childUID, levelNow++, childrenno);
               FindSplitNode();
               sOut += "</"+ nodeValue + ">";
         }
      }
      return sOut;
}
```

圖 24：XML 輸出演算法

第五章　XML 檢索系統實作與結果分析

本章將敘述本論文實作的 XML 檢索系統並對提出的漸進式索引更新方法進行效益評估與結果分析。第一節簡介本論文實作之系統，第二節說明評估索引更新效能的方法與標準，第三節則是敘述本論文之漸進式結構化索引更新方法的評估結果與分析。

第一節　實作系統展示與說明

本論文開發出一套 Web-based 的 XML 檢索系統，可讓使用者透過瀏覽器來對 XML 文件執行路徑表示式查詢，經由輸入路徑表示式的檢索條件，系統即能搜尋出滿足路徑條件的資料，並以 XML 文件顯示結果。系統介面如圖 25 所示，圖中的文字區域可讓使用者輸入 SQL-like 的路徑檢索條件。本系統的 XML 資料來源為 1998 年美國大聯盟各區球隊統計資料，總計共有 26 個球隊，每支球隊各有 21 名以上的球員。XML 資料經過 DOM Tree 剖析處理得知 DOM Tree 的高度為 7，最大分支數 k 為 23，共有 13682 個元素節點與 12918 個文字節點。檢索系統的硬體配備為 Intel Celeron 600 的 CPU，記憶體空間為 256MB；作業系統為 Microsoft Windows 2000 Server，使用 Java 2 SDK Standard Edition v.1.3.1 作為軟體發展的工具。

當使用者欲從球隊的 XML 文件找出球員「PLAYER」的「GIVEN_NAME」或「SURNAME」中出現以「John」開頭的球員資料，而某些球員的「GIVEN_NAME」與「SURNAME」不是直屬於該節點下的子節點，而是間接透過一個「NAME」節點才知該球員的「GIVEN_NAME」與「SURNAME」，加上若不考慮根節點到球員節點間出現的節點，那麼路徑檢索條件即可以「SEASON.#.PLAYER(.NAME)?.%NAME LIKE 'John%'」表示。圖 26、圖 27 與圖 28 為對應的檢索結果，圖 26 中「PLAYER.SURNAME」為「Johnson」、

圖 27 中「PLAYER.SURNAME」為「Johnstone」、「PLAYER.GIVEN_NAME」為「John」，而圖 28 中之「PLAYER.NAME.SURNAME」為「Johnson」，由這三個球員資料可看出檢索系統確實有將滿足路徑檢索條件的球員資料挑選出來。

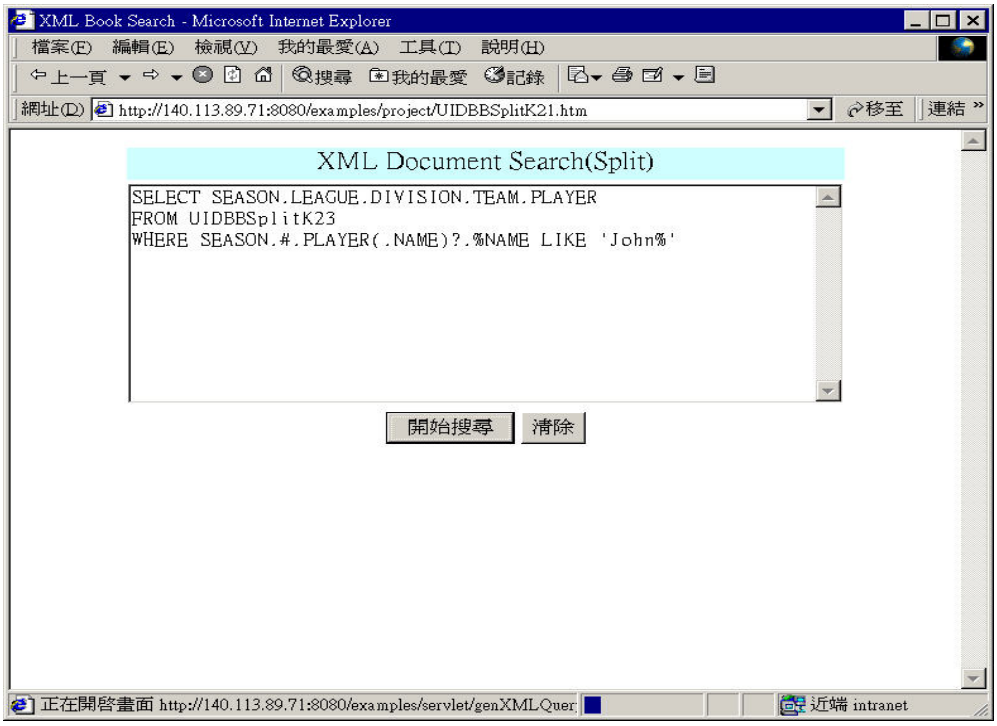

圖 25：系統介面圖

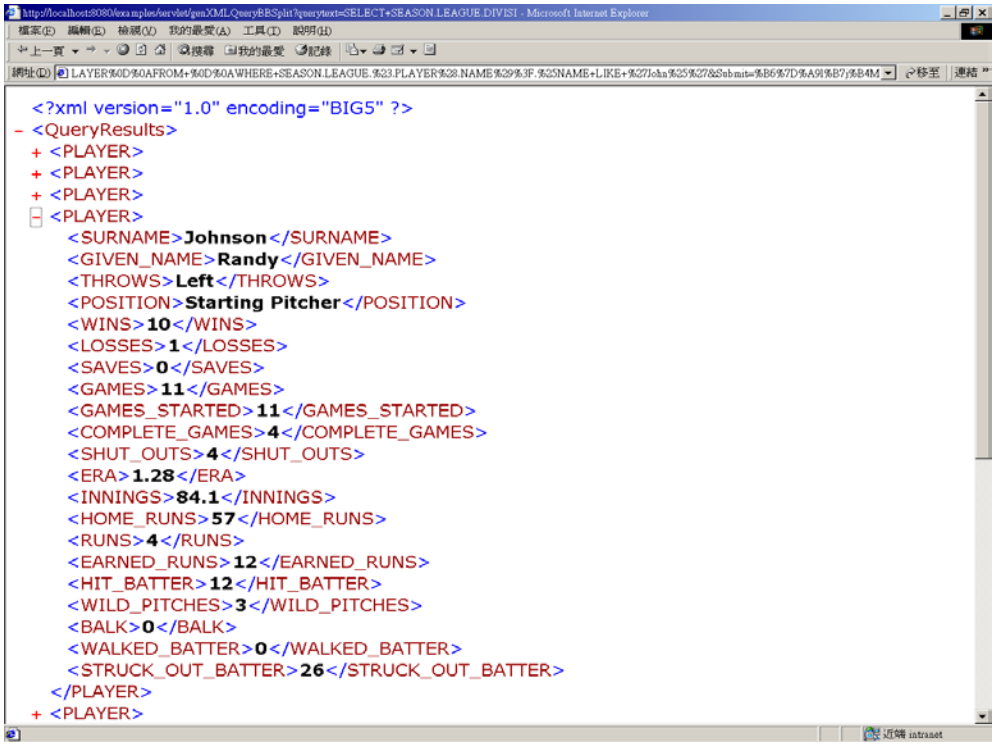

圖 26：檢索結果一

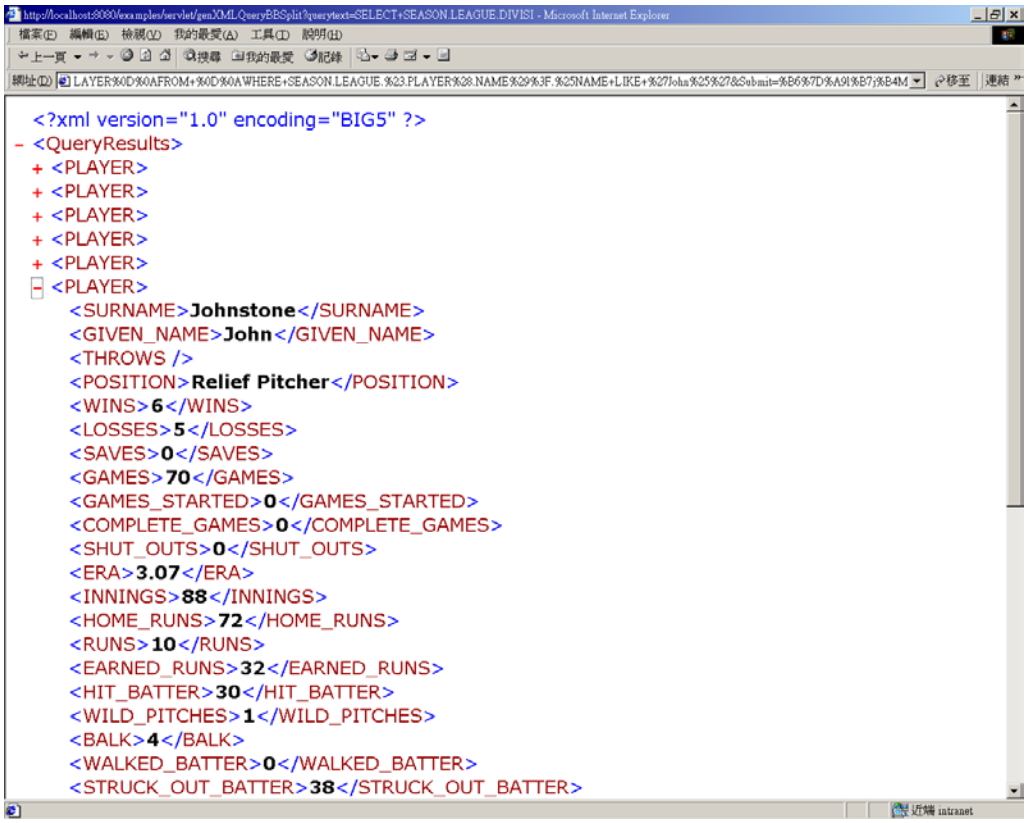

圖 27：檢索結果二

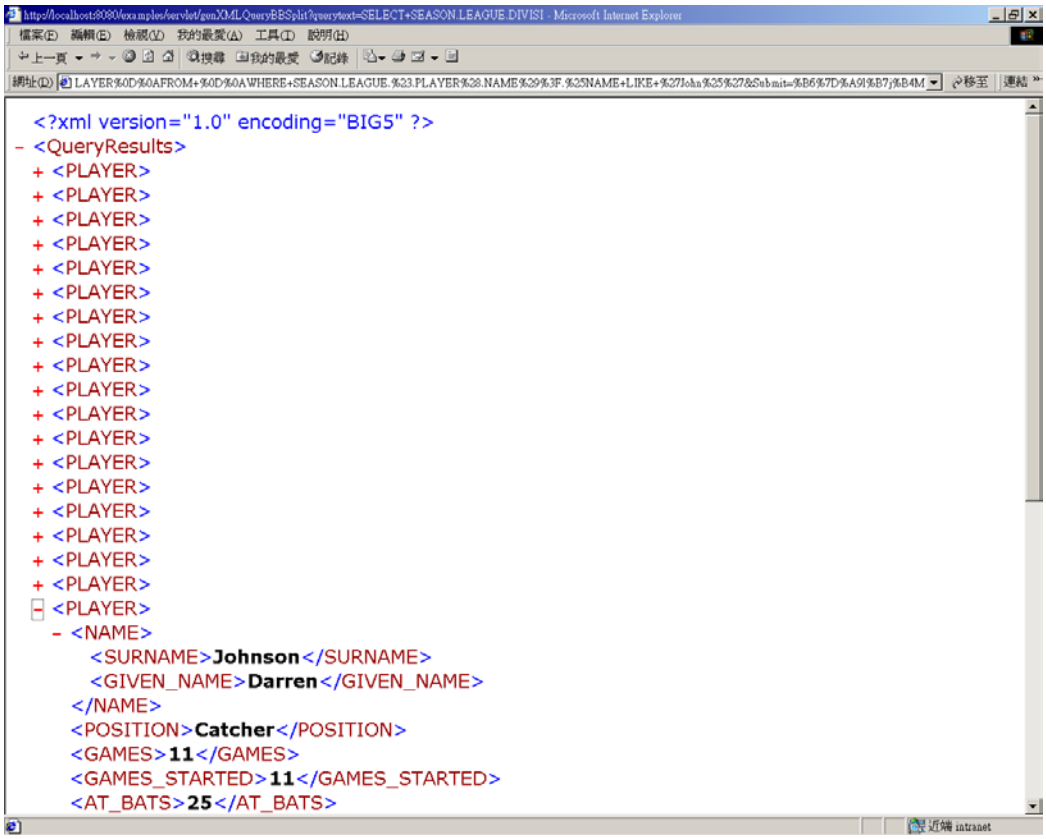

圖 28：檢索結果三

第二節　效益評估方法

針對索引重建下的索引結構與本論文提出之 Split Method 下所用的索引結構，本節依據索引空間花費(Index Size Cost)、索引更新時間花費(Update Time Cost)與檢索時間花費(Retrieval Time Cost)三個準則來評估比較兩者的效能。對於原始的索引結構與 Split Method 下的索引結構，我們分別以 UID 與 UIDSplit 表示之。以下茲說明這三種準則。

5.2.1　索引空間花費

表 7：索引空間花費比較表

方法	空間花費公式
UID	$N + \sum_{i=1}^{M} A_i$ 個實體節點 $\Rightarrow (N + \sum_{i=1}^{M} A_i) * S$
UIDSplit	$N + \sum_{i=1}^{M} A_i + M'$ 個實體節點 $\Rightarrow (N + \sum_{i=1}^{M} A_i + M') * (S+E)$

對於索引空間之花費，我們以加入資料節點時造成 k-ary Tree 超出最大分支數 k 的次數為評比標準。在此假設在原始的索引結構下，於某特定節點加入 M 次資料時，發生 M 次超出最大分支數，其中每次加入的節點數量為 A_i，$i = 1 \sim M$。而原始文件的資料節點數為 N，依據原始的資料結構(SeqId, DID, UID, Level, ElementType, Value, ChildNo, Delete)，我們定義每個節點所佔的空間為 S，若以我們提出之 Split Method 下的資料結構(SeqId, DID, UID, Level, ElementType,

Value, ChildNo, Delete, EQF, EQB)，則每個節點佔的空間會比原始的資料結構多出 E，E 為定義對等關係所需的空間大小，對等關係 E 係由(EQF, EQB)兩資訊所組成。因此，Split Method 下的節點大小為 S + E。依據前述評比標準，兩方法空間花費之比較如表 7 所示。

基於前述的評比標準，每加入一次資料時，索引重建會將新增的節點資料加入原始資料中。因此，加入 M 次資料後共有 $N+\sum_{i=1}^{M} A_i$ 個節點，對應的索引空間即為$(N+\sum_{i=1}^{M} A_i) * S$。在 Split Method 下，每次加入資料時，亦是將新增的節點資料加入原始資料中，但加入 M 次資料時只發生 M'次(M'<M)超出最大分支數 k 值。此情況下在插入資料的節點處會分裂出 M'個對等節點，因此，加入 M 次資料後共有 $N+\sum_{i=1}^{M} A_i + M'$個節點，對應的索引空間即為$(N+\sum_{i=1}^{M} A_i + M') * (S + E)$。

5.2.2　索引更新時間花費

表 8：索引更新比較表

方法	索引更新時間花費公式
UID	M 次索引重建所需走訪的節點總數 $\sum_{i=1}^{M} \text{Traverse}(N+\sum_{j=1}^{i} A_j)$
UIDSplit	走訪 M 次加入的節點資料量 $\sum_{i=1}^{M} \text{Traverse}(A_i)$，其中 i = 1~M

由於新增資料時結構索引的更新，原始的方法與本論文 Split Method 之處理方式有所不同，前者會將新增的資料加入原始資料中後再重新建立索引資料，而

後者只需判斷是否需要分裂即可避免整個索引資料重建。雖然處理方法不同，但在配置節點 UID 時兩者皆須走訪文件樹狀結構中欲配置 UID 的節點。因此，我們以所需走訪的節點總數為基準來比較兩者在索引更新時間上的花費。兩者對於索引更新所需走訪的節點數量如表 8 所示。

比較索引更新時，我們仍假設在原始的索引結構下，於某特定節點連續加入 M 次資料時，發生 M 次超出最大分支數，其中每次加入的節點數量為 A_i，i = 1 ~ M。針對原始的方法，一旦插入資料造成最大分支數超過 k 時，整個 k-ary Tree 須重建以符合完整樹的特性。因此第一次加入 A_1 個節點時造成最大分支數超過 k，需走訪 $(N + A_1)$ 個節點；第二次加入 A_2 個節點時亦造成最大分支數超過 k，因此需走訪 $(N + A_1 + A_2)$ 個節點；以此類推，第 M 次加入 A_M 個節點時，需走訪 $(N + \sum_{i=1}^{M} A_i)$ 個節點，連續加入 M 次資料所需走訪的節點總數為$(N + A_1) + (N + A_1 + A_2) + \ldots + (N + \sum_{i=1}^{M} A_i) = \sum_{i=1}^{M} (N + \sum_{j=1}^{i} A_j)$；而 Split Method 下，每次只需走訪每次加入的節點個數 A_i。因此，連續加入 M 次資料共需走訪 $\sum_{i=1}^{M} A_i$ 個節點。當加入資料時，若沒有造成 k-ary Tree 超出最大分支數 k，則兩者所走訪的節點數目相同為 A_i。

5.2.3 檢索時間花費

對於資料檢索，我們將其分成資料搜尋與產生 XML 結果兩個部分來比較原始方法與 Split Method 在檢索時間上的花費。第一個部分為找出滿足使用者檢索條件的節點所需花費的時間，另一個部分則是將找到的節點透過索引結構來產生 XML 結果所需花費的時間。以下茲針對這兩個部分來做說明。

第一部分資料搜尋步驟與時間花費分析如下：

1. 資料檢索時需先尋找滿足檢索字串的節點。

假設共有 X 個節點滿足，對於原始的方法，此步驟每一次所花費的時間為 log T，其中 T 為經過 M 次加入節點，發生 M 次超過最大分支數，重建資料索引後的所有節點數目，即$(N + \sum_{i=1}^{M} A_i)$。

2. 檢查該節點是否滿足路徑條件。

   針對一個滿足檢索條件值的節點，需要檢查該節點是否滿足路徑條件。接下來由該節點的 DID 與 UID 即可透過方程式 1 快速計算出其父節點之對應 DID 與 UID，此步驟所需的時間為常數。當計算出其父節點之 UID 後，必須再經過 log T 的時間從索引配置表中找到其實際位置。假設路徑條件的長度為 L，則找出路徑條件中的每個節點所需的時間皆為 log T。所以，判斷一個節點是否滿足路徑檢索條件所需花費的時間為 (L + 1) * log T。

3. 最後，找出所有滿足路徑檢索條件的節點所需的時間共為 X * (L + 1) * log T。

找到滿足檢索條件的節點後，系統需走訪每個節點下之所有子孫節點以產生 XML 結果。假若有 Y 個節點滿足檢索條件，則第二部分產生 XML 所花的時間為走訪這 Y 個節點的時間總和，亦即$\sum_{i=1}^{Y} \text{Traverse}(Y_i\text{的子樹})$。

表 9：資料搜尋比較表

方法	資料搜尋時間花費公式
UID	X * (L + 1) * logT
UIDSplit	X * (L + M' + 1) * log(T + M')

對於資料搜尋 Split Method 與原始方法的差別在於搜尋時多了 M'個分裂出來的節點。因此，在尋找滿足檢索字串的節點需花費的時間為 log (T + M')、檢

查一個滿足檢索條件值的節點是否滿足檢索路徑條件所需花的時間為(L + 1) * log (T + M')。而符合檢索條件值的節點若落在分裂後的子樹上時，則需透過對等關係中的 EQB 來尋找被插入節點，最差情況為符合檢索條件值的節點落在第 M' 次分裂的子樹上。因此，需多花 M' * log (T + M')的時間來找到被插入節點。所以，當欲檢查之 X 個滿足檢索條件值的節點都發生在最差的情況時，所需花費的時間即為 X * ((L + 1) * log (T + M') + M' * log (T + M')) = X *(L + M' + 1) * log (T + M')。

如同索引重建，假設滿足路徑檢索條件的節點有 Y 個，Split Method 產生 XML 結果也須花$\sum_{i=1}^{Y}$Traverse(Y_i的子樹) 的時間，但若節點落在曾發生分裂的點上時，則必須再花 M' * log (T + M')來找到分裂節點的位置，所以 Split Method 在產生 XML 結果時總需花費的時間為$\sum_{i=1}^{Y}$Traverse(Y_i的子樹) + M' * log (T + M')。

針對資料搜尋部分，兩者的比較結果如表 9 所示，而兩種方法對於產生 XML 結果的比較如表 10 所示。

表 10：產生 XML 結果比較表

方法	產生 XML 結果時間花費公式
UID	$\sum_{i=1}^{Y}$Traverse(Y_i的子樹)
UIDSplit	$\sum_{i=1}^{Y}$Traverse(Y_i的子樹) + M' * log(T + M')

雖然 Split Method 的檢索時間花費較大，但一般來說，文件的總節點數量 T 會遠大於分裂節點量 M'。因此，兩者在時間上的差距不會很大。

第三節　索引更新方法之效益評估

本節針對原始的索引機制與 Split Method 這兩種方法，就索引空間、索引更

新時間與檢索時間三個面向來探討 Split Method 的成效。

表 11：每筆資料的節點個數

加入資料的次數	每筆資料的節點數量
一	42, 42, 43, 43, 43
二	43, 43, 43, 43, 42
三	43, 43, 43, 43, 43
四	43, 43, 42, 43, 43
五	43, 43, 43, 43, 43, 43, 43

實驗之初，原始資料中共有 26600 個節點資料，為比較上述兩索引更新方法，我們於某特定節點依序加入五、五、五、五、七筆資料來比較兩者對於空間與時間的效能花費，每筆資料中含有的節點數量如表 11 所示。對於原始的方法，在這五次的實驗中，每加入一筆即造成該節點分支數超過最大分支數 k；而 Split Method 下，在第一次與第五次時各發生一次節點分支數超過最大分支數。原始方法與 Split Method 在此分別以 UID 與 UIDSplit 表示。

5.3.1 索引空間花費比較

表 12：節點空間上限下兩方法的空間花費比較表

資料累加筆數	UID 下的節點數	UIDSplit 下的節點數	UIDSplit 比 UID 多花費的空間(bytes)	Overhead
五	26813	26814	1 * 345 + 26813 * 8 (Split 一次)	2.38%
十	27027	27028	1 * 345 + 27027 * 8 (Split 一次)	2.38%
十五	27242	27243	1 * 345 + 27242 * 8 (Split 一次)	2.38%
二十	27456	27457	1 * 345 + 27456 * 8 (Split 一次)	2.38%
二十七	27757	27759	2 * 345 + 27757 * 8 (Split 二次)	2.38%

依前述的實驗步驟，兩個方法下對於索引空間花費如表 12 所示。表格中第二欄與第三欄分別代表兩個方法下，加入資料後索引資料的節點總數，而第四欄則表示 Split Method 比原始方法多花費的空間量。

在此我們以第一次加入五筆資料的情形來說明兩者間空間花費的狀況。原始的方法在加入五筆資料時發生五次超出最大分支數 k，經過五次索引重建後，索引資料表中共有 26813 個節點資料。而 Split Method，在加入五筆資料時造成一次節點分裂，此情況下會多一個分裂節點，因此總共有 26814 個節點資料。

對於索引空間配置，Split Method 下每個節點空間會比原始方法下的節點空間多出對等關係的空間花費。因此，Split Method 加入五筆資料後所佔的空間會比原始方法多出一個節點空間與對等關係的空間，總計多 1 * 345 + 26813 * 8 bytes，其中對等關係 EQF 與 EQB 兩資訊各佔 4bytes，每個節點因此需多花費 8bytes，總體而言，本論文提出之 Split Method 的索引空間 Overhead 為 2.38%。此實驗乃基於節點空間的上限來比較原始方法與 Split Method 的索引空間花費，兩方法下的資料結構空間配置分別如表 13 與表 14 所示。

表 13：原始方法下節點空間上限的空間配置表

原始方法下節點空間上限大小 (bytes)							
SeqId	DID	UID	Level	ElementType	Value	ChildNo	Delete
4	4	9	4	10	300	4	2

表 14：Split Method 下節點空間上限的空間配置表

Split Method 下節點空間上限大小 (bytes)									
SeqId	DID	UID	Level	ElementType	Value	ChildNo	Delete	EQF	EQB
4	4	9	4	10	300	4	2	4	4

計算整體空間 Overhead 時，節點空間配置的大小適當與否會大大影響計算

的結果。表 13 中，原始方法下節點的 Value 欄位配置空間為 300 bytes，使得整個節點大小為 337 bytes，其中 Value 欄位的空間佔整個節點空間的比例很高。因此，計算 Overhead 時原始方法下的空間花費無形中提升許多，然而 Split Method 下所分裂出的節點空間花費量相較之下顯的渺小。所以，相對於原始方法，Split Method 整體空間花費的 Overhead 很低。

表 15：原始方法下節點空間下限的空間配置表

原始方法下節點空間下限大小 (bytes)							
SeqId	DID	UID	Level	ElementType	Value	ChildNo	Delete
4	4	9	4	10	10	4	2

表 16：Split Method 下節點空間下限的空間配置表

Split Method 下節點空間下限大小 (bytes)									
SeqId	DID	UID	Level	ElementType	Value	ChildNo	Delete	EQF	EQB
4	4	9	4	10	10	4	2	4	4

表 17：節點空間下限下兩方法的空間花費比較表

資料累加筆數	UID 下的節點數	UIDSplit 下的節點數	UIDSplit 比 UID 多花費的空間(bytes)	Overhead
五	26813	26814	1 * 55 + 26813 * 8 (Split 一次)	17.03%
十	27027	27028	1 * 55 + 27027 * 8 (Split 一次)	17.03%
十五	27242	27243	1 * 55 + 27242 * 8 (Split 一次)	17.03%
二十	27456	27457	1 * 55 + 27456 * 8 (Split 一次)	17.03%
二十七	27757	27759	2 * 55 + 27757 * 8 (Split 二次)	17.03%

在此，另外以實際的節點空間花費下限來比較兩者在索引空間的花費，兩方法下實際測得的節點空間平均大小如表 15 與表 16 所示。若以此節點空間標準來比較整體索引空間花費，兩方法之索引空間花費比較如表 17 所示。當原始方法的節點空間大小降低為 47 bytes 時，Split Method 下索引空間花費的 Overhead 則提升到 17.03%。因此，從表 12 與表 17 可獲得一個結論，即當原始節點配置的空間越大，則本論文提出之 Split Method 在整體空間的耗費上會相對地降低。

從前面的節點空間上限與節點空間下限的條件下，Split Method 所花費的索引空間雖然比原始的方法來的多，但整體空間的花費比例並不會相差太大。可是在結構索引的維護時間花費上，Split Method 卻較原始的方法減少許多，以下即針對索引更新時間來比較兩者間效能上的差異。

5.3.2 索引更新時間花費比較

索引更新的三種類型中，資料更新與資料刪除並不會影響整個索引結構的最大分支數，因此兩個方法在這兩類的索引更新效能上相當接近。但對於新增資料時，在不影響索引結構的最大分支數下，原始方法與 Split Method 皆只需走訪加入的節點量即可。若影響到索引結構的最大分支數時，對於原始的方法，假設每加入一筆資料即造成超出最大分支數 k，此時必須將新增的資料加入原始資料中後，再重新建置一個新的 k-ary Tree 以滿足 k-ary Tree 的特性。由此可預見，若避免掉索引重建會使得索引更新效能大大提升。下列的實驗數據的確印證了本論文提出的 Split Method 在索引更新的效能上確實優於原始方法。

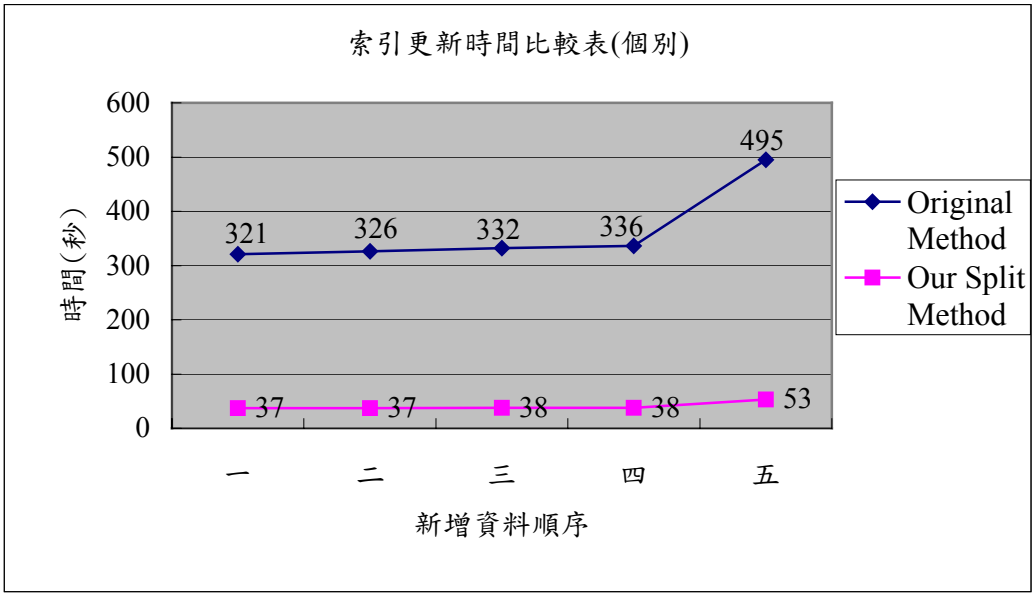

圖 29：索引更新時間比較圖(個別)

在五次加入資料的實驗中，我們以三種角度來比較原始方法與 Split Method 在索引更新時間上的效能。圖 29 的結果是比較兩種方法在五次個別加入資料的情形下索引更新所花費的時間。第一次加入五筆資料時，原始方法發生五次超出最大分支數，需重建五次 k-ary Tree，因此需花費 321 秒才能完成索引更新。而 Split Method 在第一次加入資料時發生一次節點分裂，但只需花費 37 秒即可完成索引更新。由此可明顯看出原始方法需比 Split Method 花接近 6 倍的時間才能完成索引更新。

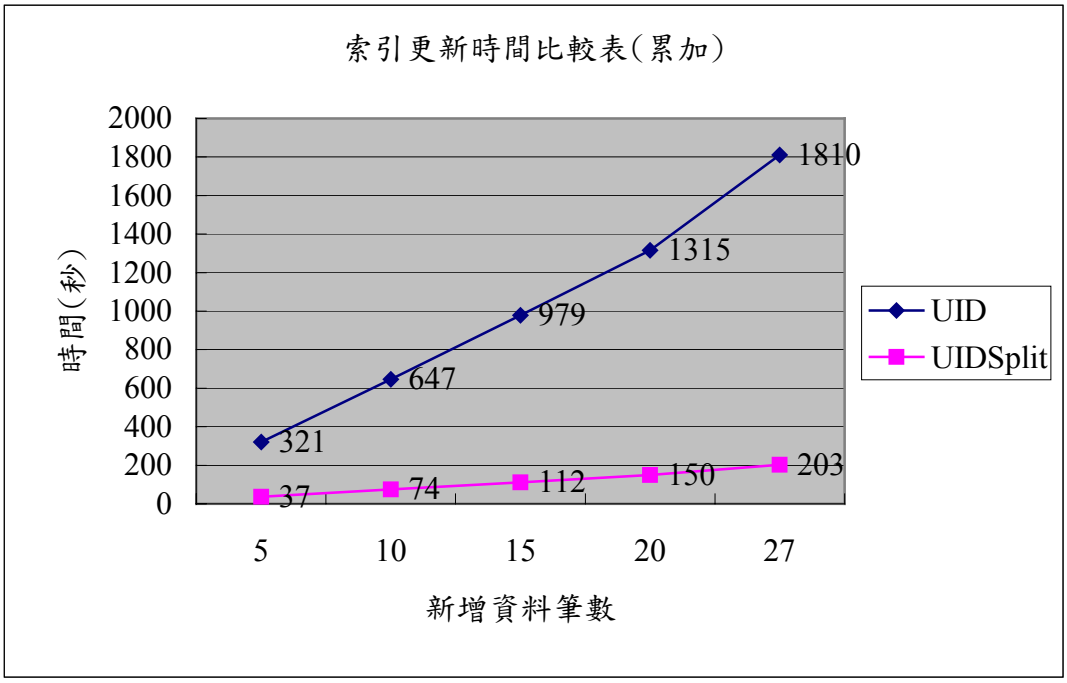

圖 30：索引更新時間比較圖(累加)

圖 30 的實驗乃基於圖 29 的實驗，差別在於當資料加入的量由小變大時，更能看出兩方法間索引更新效能的差距。若第一次加入五筆時，原始方法發生五次超出最大分支數，在第二、三、四、五次加入資料時，總計累加發生十、十五、二十及二十七次超出最大分支數。相對地，Split Method 在第一次加入資料時發生一次節點分裂，而在第二、三、四次加入資料時並未發生節點分裂，只有在第五次時又發生一次節點分裂。因此，當一次加入的節點越多時，原始方法花於索引重建的時間會高出 Split Method 甚多。圖 30 即為兩方法對於此實驗的比較結果，從圖中可看出原始方法當加入的資料越多時，整個時間曲線爬升的速度比我們提出之方法高出許多。因為原始方法在發生超出最大分支數須將先前加入之節點反覆重新走訪。

第三個實驗，乃依據5.2.2 節所述的評估標準來計算連續加入二十七筆資料時，索引更新所需走訪的節點總數。圖 31 為兩方法在五個階段中完成索引更新所需走訪的節點數量比較圖。

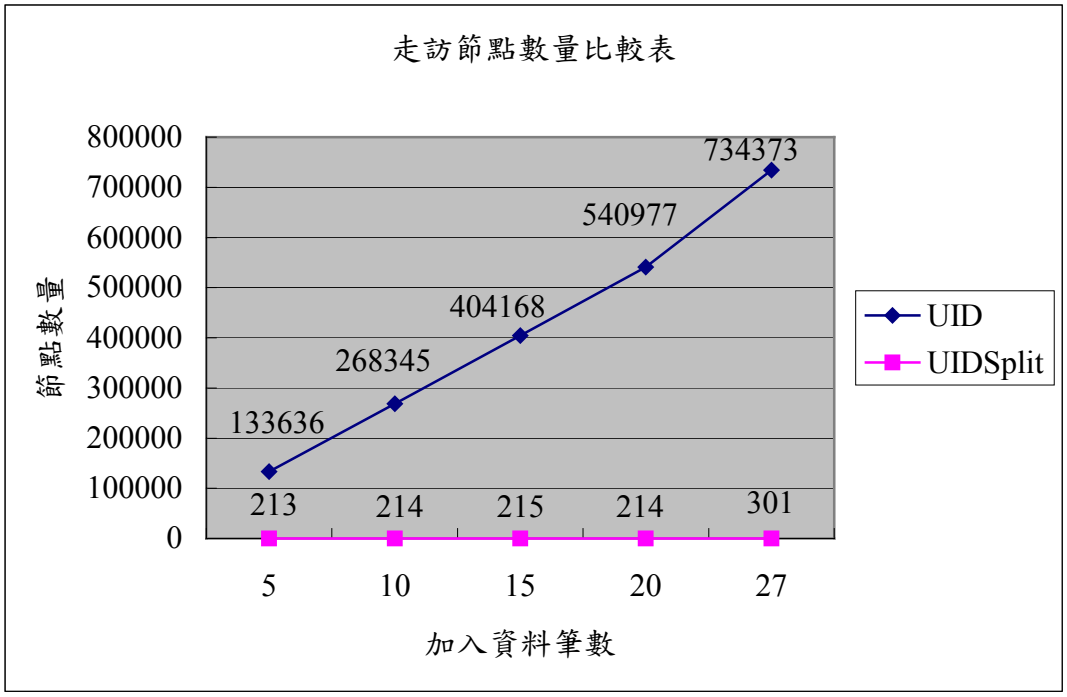

圖 31：走訪節點數量比較圖

此結果顯示 Split Method 於每次加入資料時，不管是否會造成 k-ary Tree 超出最大分支數，只需走訪加入的資料節點量即可；但原始的方法每加入一次資料即造成 k-ary Tree 超出最大分支數，因此需將新增的資料加入原始資料中再重新對所有的資料做深度優先搜尋以配置 UID。如此，會造成前面所加入的節點到下次加入資料時，仍需反覆重新配置 UID，因此，加入的資料越多則完成索引資料更新的時間亦越多。此實驗結果亦與圖 30 之實驗結果相互呼應。

5.3.3 檢索時間花費比較

依據5.2.3 節的評估標準，我們在檢索時間上分成兩個部分來進行比較：第一部分為搜尋出滿足路徑檢索條件的節點所需花費的時間，第二部分為將滿足條件的節點以 XML 文件結果呈現所需花費的時間。在此我們又以兩種檢索狀況來比較兩方法在時間上的花費。檢索狀況一為一般的檢索條件，即節點資料不是出現在發生分裂過的節點上；而檢索狀況二為資料出現在發生分裂過的節點上。針對檢索狀況一與狀況二，我們依據滿足路徑檢索條件之節點量的不同下了多個檢

索條件以求平均檢索時間。

表 18：兩檢索狀況下之對應路徑檢索條件範例

狀況	路徑檢索條件範例
一	SELECT SEASON.LEAGUE.DIVISION.TEAM FROM SourceUID WHERE SEASON.LEAGUE.DIVISION.TEAM.PLAYER.GIVEN_NAME = 'Darren'
二	SELECT SEASON.LEAGUE.DIVISION.TEAM FROM SourceUID WHERE SEASON.LEAGUE.DIVISION.TEAM.PLAYER.GIVEN_NAME = 'Sal'

舉例而言，檢索條件如表 18 所示。

- 狀況一的檢索條件想要找出名為「Darrin」的球員，但此實驗中發生分裂的「TEAM」節點下並無名為「Darrin」的球員。這樣的條件下只有 1 個「TEAM」節點滿足此路徑檢索條件。
- 狀況二的檢索條件下，發生分裂的「TEAM」節點底下含有名為「Sal」的球員。共有 1 個「TEAM」節點滿足此路徑檢索條件。

這兩類的檢索主要比較兩方法對於資料出現於發生分裂的節點中與否，是否會造成檢索時間上的影響。

根據 Split Method 的特性，我們提出以下兩個關於資料檢索時間的推論：

1. 資料不是出現於發生分裂的節點中，則 Split 的方法不需透過對等關係找出對等的分裂節點，因此對於狀況一，Split 的方法與原始的方法在資料檢索所花費的時間應該非常接近；
2. 但對於狀況二，Split 的方法需透過對等關係找出分裂的節點後才能進一步取得分裂節點下的子節點，但原始的方法則沒有分裂上的問題，因此，對於狀況二，原始的方法在檢索效能上會略比 Split Method 好。

由表 19、表 20、表 21 與表 22 的實驗數據顯示 Split Method 在檢索時間的花費上略比索引重建多，此結果印證了前述的推論並與5.2.3 節的理論相呼應。

表 19 與表 20 中對於狀況一的第一個檢索條件，只有一個 TEAM 滿足條件，索引重建與 Split Method 下找出滿足路徑條件的節點所需花費的時間分別為 0.70 秒與 0.83 秒，而產生 XML 結果對於原始方法與 Split Method 下所需的時間分別為 75.15 秒與 75.70 秒。

表 19：狀況一下節點搜尋時間比較表

狀況一:資料不出現在分裂節點上		
滿足條件的 TEAM 個數	UID 的平均時間花費(秒)	UIDSplit 的平均時間花費(秒)
1	0.70	0.83
2	1.34	1.53
3	2.08	2.28
4	2.66	2.87
5	3.81	4.16

表 20：狀況一下產生 XML 的時間比較表

狀況一:資料不出現在分裂節點上		
滿足條件的 TEAM 個數	UID 的平均時間花費(秒)	UIDSplit 的平均時間花費(秒)
1	75.15	75.70
2	156.94	157.03
3	251.25	258.32
4	328.43	330.45
5	445.99	453.55

表 20 與表 21 中，在狀況二第一個檢索條件下，只有一個 TEAM 滿足條件，索引重建與 Split Method 在節點搜尋的時間上分別花費 0.70 秒與 0.88 秒。在產生 XML 所花費的時間，索引重建與 Split Method 分別花費 179.82 秒與 184.25 秒。如前面推論 Split Method 所需花費的時間確實比原始方法略多一些。

表 21：狀況二下節點搜尋時間比較表

狀況二:資料出現在分裂節點上		
滿足條件的 TEAM 個數	UID 的平均時間花費(秒)	UIDSplit 的平均時間花費(秒)
1	0.70	0.88
3	2.29	2.54
4	3.24	3.60
5	3.89	4.28
6	5.72	6.44

表 22：狀況二下產生 XML 的時間比較表

狀況二:資料出現在分裂節點上		
滿足條件的 TEAM 個數	UID 的平均時間花費(秒)	UIDSplit 的平均時間花費(秒)
1	179.82	184.25
3	373.52	380.05
4	482.13	490.61
5	591.87	596.40
6	727.99	734.74

5.3.4 索引重建

由於 Split Method 在資料更新時，若發生節點分支數超過最大分支數 k 則會造成節點分裂。但每次節點分裂只會多出一個分裂節點，從5.3.1 節的索引空間分析得知，就整體而言分裂節點的數量在所有的節點中所佔的比例很小。因此，從新增節點的角度無一個衡量標準來決定結構索引重新建構的時機。因為不管在空間花費或實際進行資料檢索時，分裂節點皆不會造成整體效能上太大的影響。

從刪除節點的角度來看，雖然資料刪除時並不會影響最大分支數，但刪除節點時需要調整該節點在樹狀結構中後面的兄弟節點以維持完整樹的特性。針對刪除節點的狀況，有兩個方法來更新索引資訊。方法一為將欲刪除之節點實際從索引結構中刪除，並更新該節點後面所有兄弟節點的 UID；方法二為3.3.2 節所述，利用 Delete 變數來記錄節點是否已被刪除，以節省方法一中調整節點 UID 的時

間花費。方法一乃利用時間來換取空間上的效能，而方法二則反之。以下乃針對這兩種兩方法來提出索引重建的準則。

- 實際刪除節點：

若欲刪除的節點為 N，則進行資料刪除時，需將節點 N 下的所有下屬節點與該節點分裂出的節點子樹一併刪除。資料刪除後需將該節點其後的兄弟節點往前挪動以維持 k-ary Tree 完整樹的特性。假設欲挪動的節點數量為 S，而所有的資料節點數量為 T，當 S / T ≧ α 時索引即可重建，其中 α 為使用者設定之節點移動參數。方程式 S / T ≧ α 代表挪動的節點數與索引重建所需重新走訪的節點數量比例若大於使用者設定的參數 α ，則 k-ary Tree 可重新建置。因為刪除資料後完成索引資料更新所需花費的時間與索引重建的時間接近，此時是重建索引的最佳時機。

- 利用 Delete 變數：

假設刪除的節點數量共有 X 個，因此會有 X 個節點的 Delete 變數會為 1。若所有的資料節點數量為 T，當 X / T ≧ β 時索引即可重建，其中 β 為使用者設定之節點刪除參數。方程式 X / T ≧ β 代表已刪除的節點數量佔全部節點數量的比例若大於使用者設定的參數 β ，則 k-ary Tree 可重新建置。因為索引結構中有一定比例的節點是虛擬節點，因此可透過索引重建來實際刪除節點。

上述兩種方法各有其優缺點，何種方法適用於索引重建乃由資料管理者來取決。

第六章　結論與未來研究方向

本章總結本論文以及敘述未來研究方向，第一節說明本論文提出之漸進式索引更新方法的優點與不足之處，第二節則說明本論文未來可能的研究發展方向。

第一節　結論

當越來越多的 XML 文件在網際網路上傳播與交換時，若能對這些具有結構化資訊的文件建構索引並有效地加以維護，勢必能提供使用者從結構多樣化的文件中找到感興趣的資料。對於結構化資訊經常變更時，本論文所提出之 Split Method 具有以下的優點：

1. ***避免索引資料結構重建***。由於原始 k-ary Tree 的方法在加入資料時，若遇到節點分支數超出完整樹的最大分支數，必須花費龐大的時間來重新建構整個資料索引，以維持完整樹的特性。而我們提出之方法則利用節點分裂避免了索引重建。

2. ***檢索效能與原始 k-ary Tree 重建的方法接近***。雖然我們需比原始 k-ary Tree 重建的方法多花費一些空間來避免結構索引重建，但這並不會造成檢索效能上太大的影響。

3. ***支援路徑表示式檢索***。透過我們使用的索引資料結構亦可支援強大的路徑表示式檢索。

本論文提出之方法相較於原始 k-ary Tree 重建之方法不足的地方在於索引空間上的花費，這本是時間與空間上取捨(Trade-off)的問題，雖花費較多索引空間但卻能換得時間上的效率。

第二節　未來研究方向

本論文提供之 SQL-like 路徑表示式檢索尚未將 Lorel 中更強大的功能實作出來，如在檢索語言中使用標籤變數(Tag Variable)、路徑變數(Path Variable)、依某種條件來排序檢索結果(Ordering the results)或群聚檢索結果(Grouping the results)以及 Join 等操作。未來可繼續研究如何透過本論文使用的索引結構來支援更豐富的檢索功能。此外，基於本論文使用的索引結構進行資料檢索時，若能對檢索條件進行最佳化處理，以產生最佳的檢索機制，使檢索效能更加提升，亦是未來的研究方向。

此外，本論文支援的 XML 文件檢索目前僅止於全文檢索。當資料量日趨龐大時，如何更準確地找出使用者感興趣的資料亦為目前資訊搜尋上的議題。因此，如何將資訊擷取(Information Retrieval)的技術整合進檢索機制中，以期找出更符合使用者感興趣的資訊亦值得進一步研究。

參考文獻

1. [Abiteboul97] S. Abiteboul, D. Quass, J. McHugh, J. Widom, J. L. Wiener. *"The Lorel query language for semistructured data,"* International Journal on Digital Libraries, Volume 1, Issue 1, 1997, pages 68-88.
2. [Bertino01] E. Bertino, B. Catania. *"Integrating XML and Databases,"* IEEE Internet Computing, Volume 5, Issue 4, July-Aug. 2001, pages 84-88.
3. [Bonifati02] A. Bonifati, D. Braga, A. Campi, and S. Ceri. *"Active XQuery,"* Data Engineering, 2002. Proceedings. 18th International Conference on, 2002, pages 403-412.
4. [Bonifati00] A. Bonifati and S. Ceri. *"Comparative Analysis of Five XML Query Languages,"* SIGMOD RECORD, Volume 29, Number 1, 2000, pages 68-79.
5. [Bryan92] M. Bryan. *"An Introduction to the Standard Generalized Markup Language (SGML),"* 1992, http://www.personal.u-net.com/~sgml/sgml.html
6. [Ceri99] S. Ceri, S. Comai, E. Damiani, P. Fraternali, S. Paraboschi, L. Tanca. *"XML-GL: A Graphical Language for Querying and Restructuring XML Documents,"* Proc. WWW8, Toronto, Canada, May 1999.
7. [Chamberlin01] D. D.Chamberlin. *"Query languages and XML,"* Database Engineering & Applications, 2001 International Symposium on, 2001, pages 297-300
8. [CML] Chemical Markup Language (CML™) http://www.xml-cml.org/
9. [Deutsch98] A. Deutsch, M. Fernandez, D. Florescu, Alon Levy and D. Suciu. *"XML-QL: A Query Language for XML,"* In Proc. of the Query Languages workshop (QL98), Cambridge, Mass., December 1998, http://www.w3.org/TR/NOTE-xml-ql
10. [DOM98] Document Object Model (DOM). http://www.w3.org/DOM/
11. [Flesca02] S. Flesca, F. Furfaro, and S. Greco. *"A Graphical XML Query Language,"* Data Engineering, 2002. Proceedings. 18th International Conference on, 2002, pages: 264.
12. [Goldman99] R. Goldman, J. McHugh, and J. Widom. *"From Semistructured Data to XML: Migrating the Lore Data Model and Query Language,"* In 2nd ACM SIGMOD Int. Workshop on The

Web and Databases, 1999, pages 25-30.

13. [Jang99] H. Jang, Y. Kim, and D. Shin. *"An effective mechanism for index update in structured documents,"* Proceedings of the eighth international conference on Information knowledge management November 1999, pages 383-390.
14. [Kanemoto98] H. Kanemoto, H. Kato, H. Kinutani, and M. Yoshikawa. *"An efficiently updatable index scheme for structured documents,"* Database and Expert Systems Applications, 1998. Proceedings. IEEE Computer Society, Ninth International Workshop on, 1998, pages 991-996.
15. [Larson01] P. Larson. *"XML Data Management Go Native or Spruce up Relational Systems?,"* ACM SIGMOD 2001 Santa Barbara, California, May 21-24, 2001 (panel abstract)
16. [Lee96] Y. K. Lee, S. J. Yoo, and K. Yoon, *"Index Structures for Structured Documents,"* Proceedings of the 1st ACM international conference on Digital libraries, 1996, pages 91-99.
17. [MathML] Mathematical Markup Language (MathML™) http://www.w3.org/TR/REC-MathML/
18. [Robie98] J. Robie, J. Lapp and D. Schach. *"XML Query Language (XQL),"* In Proc. of the Query Languages workshop, Cambridge, Mass., Dec. 1998, http://www.w3.org/TandS/QL/QL98/pp/xql.html
19. [Roy00] J. Roy, A. Ramanujan. *"XML: data's universal language,"* IT Professional, Volume 2, Issue 3, May-June 2000 pages 32-36.
20. [XML98] Extensible Markup Language (XML) 1.0 http://www.w3.org/XML
21. [XML00] Extensible Markup Language (XML) 1.0 (Second Edition) W3C Recommendation 6 October 2000 http://www.w3.org/TR/2000/REC-xml-20001006
22. [XSL01] The Extensible Stylesheet Language (XSL) Version 1.0, Oct. 2001, http://www.w3.org/TR/xsl
23. [XSLT99] XSL Transformations (XSLT) Version 1.0, Nov. 1999, http://www.w3.org/TR/xmlt
24. [Zisman00] A. Zisman. *"An overview of XML,"* Computing and Control Engin. J., Volume 11, Aug. 2000, pages 165-167.
25. [黃中杰 00] 黃中杰、王天利，「XML 新網頁語言開發手冊」，知城數位，2000 年十二月。