國立臺灣海洋大學
資訊工程學系
碩士學位論文

指導教授：張雅惠　博士

探討 XML 關鍵字查詢有效輸出之研究
Identifying Relevant Matches for XML Keyword Search with and-or Operators

研究生：李俊毅　撰

中華民國 99 年 1 月

探討 XML 關鍵字查詢有效輸出之研究

Identifying Relevant Matches for XML Keyword Search with and-or Operators

研 究 生：李俊毅　　Student：Chun-Yi Li

指導教授：張雅惠　　Advisor：Ya-Hui Chang

國 立 臺 灣 海 洋 大 學
資 訊 工 程 學 系
碩 士 論 文

A Thesis
Submitted to Department of Computer Science and Engineering
College of Electrical Engineering and Computer Scienece
National Taiwan Ocean University
In Partial Fulfillment of the Requirements
for the Degree of
Master of Science
in
Computer Science and Engineering
January 2010
Keelung, Taiwan, Republic of China

中華民國 99 年 1 月

摘要

在本論文中，我們探討關鍵字以 “AND” 和 “OR”方式結合作查詢時，輸出結果應該要符合的性質，以及制定輸出結果時條件的篩選。其中性質為單調性(monotonicity)和一致性(consistency)，兩種性質用來說明當文件或查詢句更新時，查詢結果個數或內容該有的變化。針對篩選條件 (AO_Contributor)的定義，我們將查詢句分作項次 (clause) 和關鍵字兩種類型，定義出項次包含與關鍵字包含共四種情況。首先依據每個節點底下所有的關鍵字，和兄弟節點底下的關鍵字，來判斷自己與兄弟節點所佔有的項次與關鍵字，再依據四種情況，與兄弟節點比較來研判此節點是不是該輸出，以達到 and 和 or 組成的查詢句的語意。我們會證明由此篩選條件組成的演算法，會滿足上述的單調性和一致性。

最後我們實作演算法，並進行一系列實驗，來探討此作法的有效性及效率。

Abstract

In this thesis, we discuss how to produce the desirable results for queries which consist of keywords combined by the “AND” operator and the “OR” operator. We consider two properties: monotoncity and consistency, which describe the desirable change to the number of query results or the content of query results upon update to the data or query. We also propose a corresponding algorithm. It first divides a query into clauses and keywords. According to the four cases for matches under each node, it then compares the clause set and keyword set of each node with its sibling to decide whether to output or not.

We have implemented the algorithm, and performed a series of experiments to show the effectiveness and efficiency of the proposed approach.

誌謝

首先要感謝指導教授張雅惠博士，在學生修業之期間耐心指導，解決學生論文上的諸多疑點，使學生順利的完成論文。同時也要感謝系上林川傑博士和台大趙坤茂博士百忙中抽空參予論文審查工作，提供許多寶貴意見和建議。

最後要感謝台大容任學長，以及本實驗室的治中學長、佳臻同學、育旻同學、品銜同學、榮彬同學和誠正學弟的幫忙，謝謝你們精神上的鼓勵以及課業上的指點迷津。最後感謝父母在背後的付出與支持。在此一併致上謝意，謝謝你們。

目錄

圖目錄

表目錄

第一章 緒論

1.1 研究動機與目的

近年來，全球資訊網 (WWW) 已經成為資訊分享的主要平台，但是以 HTML 表示的網頁資料，並不適合自動化處理。為此，W3C 制定了 XML (Extensible Markup Language) ，允許使用者自訂文件所需之標籤 (tag) 和結構。XML 的語法十分類似 HTML，不過，其標籤的作用並不是編排內容，而是用來描述資料本身的涵義，以便我們去了解 XML 資料且使用它。

由於 XML 文件已經成為表現 Web 資料的標準格式，因此查詢 XML 文件是一個重要的議題。W3C 先後提出 XPath 和 XQuery 兩種語言，供使用者針對 XML 文件下達查詢句。此二語言定義如何描述元素間的結構限制，以達到精準的搜尋結果。然而，當使用者不知道 data schema、或者 schema 太複雜以至於使用者不容易組成查詢句、或者使用者不知道如何使用 XPath 或 XQuery 時，使用者都會希望能夠直接進行關鍵字的搜尋。所以，允許使用者只下達關鍵字，但能夠回傳符合 XML 結構的資料，也變成一個相當重要的研究課題。

因此，本論文是希望能夠針對 XML 文件直接以關鍵字下達查詢句，並支援關鍵字能夠以 “AND” 和 “OR” 的型式結合，提供有效率的查詢處理。我們以目前之獻上提出最佳作法的論文 [LC08] 為例。該論文中，查詢句只能以 “AND” 方式來連接關鍵字。譬如針對圖 1.2 NBA 的球員資料，若我們想找「球員 Fisher 的進攻位置」，查詢句為「Fisher and position」。不過，該論文並沒有探討同時使用 “AND” 和 “OR” 的查詢處理。，因此我們延伸[LC08]的作法，希望能夠直

接處理「(Fisher or Ariza) and (position or nationality)」這類以 “AND” 和 “OR” 的型式結合關鍵字的查詢句。

1.2 研究方法和貢獻

首先，我們先說明 MaxMatch [LC08]，如何處理關鍵字以 “AND” 的方式相連的查詢句。MaxMatch 針對輸出結果提出 Monontonicity 和 Consistency 兩個性質，Monontonicity 針對查詢結果的個數，會隨著資料或查詢句的更新有所改變，Consistency 針對查詢結果的內容，會隨著資料或查詢句的更新有所改變。例如圖 1.1 的 Q1 與 Q2，[Q1, D1] 和 [Q2, D1]，Q1 回傳結果為以 player (1.1.3.1)、player (1.1.3.2)和 player (1.1.3.3)當作根節點的三棵子樹，但 Q2 新增關鍵字 Swiss 後，只會回傳 player (1.1.3.3)為根節點的子樹，查詢結果個數由 3 變為 1，此為 query monotonicity。接著考慮[Q3, D1]，則回傳結果個數為 0，若增加資料形成文件 D2，則回傳以 player (1.1.3.5) 為根節點的子樹，回傳結果個數為 1，此性質為 data monotonicity。接下來討論 query consistency，隨著關鍵字增加，回傳結果也包含新增的關鍵字。考慮[Q1, D1] 和 [Q2, D1]，在多加關鍵字 Swiss 之後，player (1.1.3.3)也多了 nationality (1.1.3.3.4)和關鍵字 Swiss (1.1.3.3.4.1)。最後討論 data consistency，若在新增節點後，有額外的子樹變成回傳結果的一部份，則此棵額外的子樹應該要包含新增的節點。考慮[Q3, D1] 和 [Q3, D2]，額外的子樹為以 player (1.1.3.5)作為根節點的子樹，且此棵子樹包含新增的節點 nationality (1.1.3.3.4)和關鍵字 Swiss (1.1.3.3.4.1)。

接著，我們說明 MaxMatch 如何實作。MaxMatch 首先使用[XP05]提出的方式，找出滿足所有關鍵字都至少出現一次的最小子樹根節點 (SLCA) 的集合，接著將所有關鍵字分配到所屬的 SLCA 底下形成各自的 group，接著將每個 group

建立成樹，並依據 Contributor 的概念選擇如何輸出節點，也就是節點底下的關鍵字集合與自己兄弟節點底下的關鍵字集合，若沒有被包含，則輸出此節點。為了改善 MaxMatch 的效率，[LCC10]提出 SingleProbe 的作法。該系統直接由關鍵字往上建立成樹，接著從根節點往下尋找 SLCA，再由 SLCA 開始以 Contributor 的概念選擇如何輸出節點。

Q1	guard ∧ number 尋找後衛的球衣號碼
Q2	guard ∧ Swiss ∧ number 尋找瑞士的後衛的球衣號碼
Q3	center ∧ Spain 尋找西班牙的中鋒

圖 1.1　And 查詢句範例

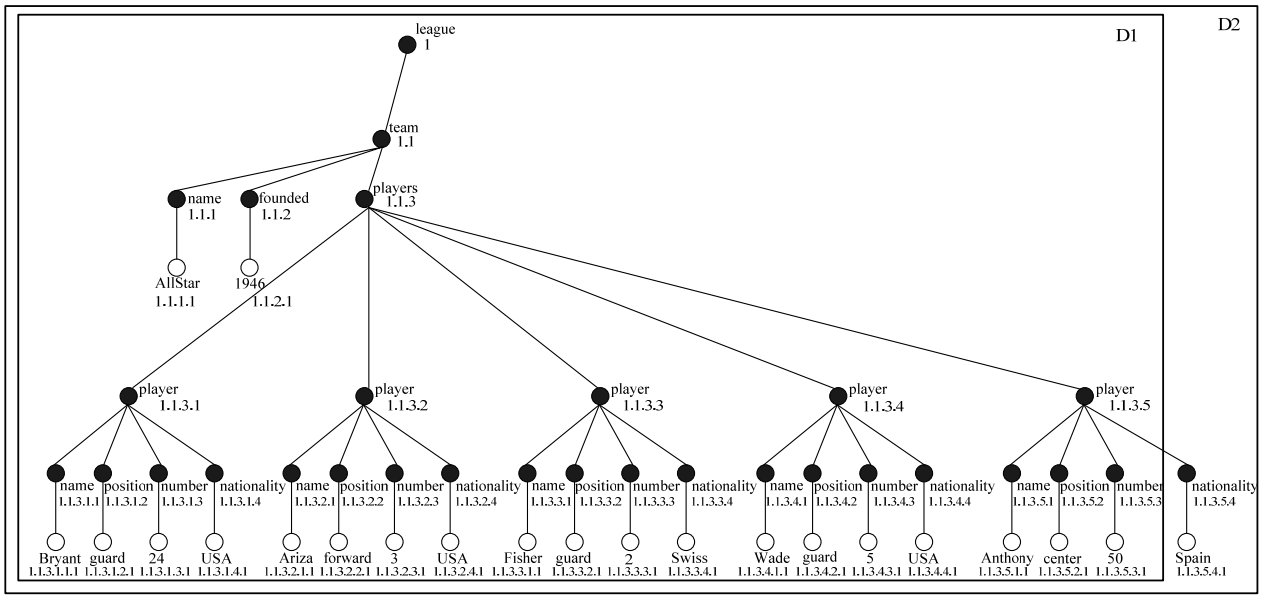

圖 1.2　XML 樹

本論文提出兩種作法。第一種 AO_MaxMatch 系統參考 MaxMatch 的方式，支援查詢句以 “AND” 和 “OR” 的型式結合關鍵字，並將查詢句以 And 做分界定義項次 (clause)和關鍵字兩種類型，提出項次包含和關鍵字包含，我們修改並擴充 Monotonicity 和 Consistency 兩性質，使得 And-Or 查詢句輸出結果更加合理化。最後並重新制定 AO_contributor 的條件作為輸出節點的方法。第二種 AO_SingleProbe 系統參考 SingleProbe 的方式，同樣支援查詢句以 “AND” 和 “OR” 的型式結合關鍵字，也以 AO_contributor 的條件作為輸出節點的方法。

針對本論文主要貢獻，總結如下所示：

1. 我們將查詢句分作項次 (clause)和關鍵字兩種類型，定義出項次包含與關鍵字包含共四種情況，當作輸出結果的判斷，並修正和擴充 [LC08]的四個性質：data monotonicity、data consistency、query monotonicity 和 query consistency，使得 And-Or 查詢句的輸出結果能夠更合理。
2. 本論文首先提出 AO_MaxMatch 方法，該方法擴充[LC08]的做法，進一步支援處理以 “AND” 和 “OR” 的型式結合的查詢句，另外參考[LCC10]的作法，提出 AO_SingleProbe 方法，一樣支援以 “AND” 和 “OR” 的型式結合的查詢句，並捨棄了[XP05]提出的找 SLCA 的方法。
3. 我們針對此兩種方法進行一系列的實驗，得知我們定義出的篩選條件能夠適當的反應出使用者下的查詢句，所需要回傳的資訊。並且發現，關鍵字頻率高低差大約在 1：100 內時，適合使用 AO_SingleProbe 方法，反之，則採用 AO_MaxMatch 方法。

1.3 相關研究

隨著XML資料越來越廣泛的被應用，XML的查詢處理成為一個重要的議題，

而由於 XML 資料具有文件的特性，所以 IR 的技術也被考慮，主要探討如何同時符合 XML 查詢句的結構限制和關鍵字限制，以及 scoring function 的設計。在研究[AYJ03]中，作者給與每個元素一個評分(score)，並在評分時同時考慮其下的子孫元素，作者提出了 TIX (a bulk algebra)及以 stack 為基本的 TermJoin 跟 PhraseFinder 演算法，TIX 提供了將 IR 型式的查詢併入一般 XML 資料庫的可能，而以 stack 為基本的 TermJoin 及 PhraseFinder 能快速有效的將評分算出。至於在研究[GSBS03]中，作者將整個 XML 文件採用 Dewey 編碼並紀錄於 inverted list 中，如此一來在搜尋關鍵字時，可以由 Dewey 編碼的前序排列得到符合的路徑，也可以快速地得知任兩個元素之間是否有父子或祖孫關係。這份研究也會針對文件中的元素計算出個別的 ElemRank，其公式考慮到 XML 文件 hyperlink 及 nested 的結構特性，和 keyword proximity。[YLP04]在處理一個 XPATH 查詢句時，會先將其轉成樹的結構，針對每一對節點的結構關係（如父子或祖孫關係），從 XMLtree 中將符合的資料找出來。接下來計算每一個答案的結構罰責，並和 contain 的罰責相加，根據罰責分數的高低即可回傳前 k 個最相關的資料。研究[KKNR04]則提出將 Structure Index 以及 Inverted List 結合的架構，其作法先將 Branching Path Expression 依分支的點將其分解成一條一條無分支的路徑，由 Structure Index 取出符合該結構的元素編碼，最後再透過 Inverted List 取得值符合的元素，並將其與之前所處理出的 Structure 比對看是否符合，若是則此為正確答案。研究[TSW05]中在 XML 的結構部分則是利用 PreOrder 和 Preorder 對元素進行編碼，並將該編碼存放於 Inverted List 中。作者並修改基本的 top-K algorithm，使其能針對 XML 文件結構上的關係進行 TOP K 的處理。研究[AKMD+05]則是希望提出的分數能同時反映 XML 文件的結構和內容。該論文參考 twig scoring，path scoring，binary scoring 的架構，並利用 query relaxation 的方式，使得越精確的答案分數越高，同時也提出一個 Directed acyclic graph (DAG) 結構，加快 XML 回傳 Top-K 的結果。

有些研究允許使用者只給予關鍵字的限制，然後系統會根據 XML 文件的結構自動回傳最相關的資料。研究[XP05] 考慮在搜尋多個關鍵字時，找出符合結果最小子樹的集合，也就是 SLCA (Smallest Lowest Common Ancestor)。該論文承襲研究[GSBS03]的編碼方式，提出了 The Indexed Lookup Eager Algorithm(IL)和 The Scan Eager Algorithm，其中 IL 演算法特別適用於關鍵字變化的頻率較大時。研究[ACD06]則將符合個別關鍵字的元素存放於 SCU table 中，然後在處理複雜的條件句，如限定不同關鍵字的順序，該論文會先找出符合所有關鍵字的 LCA，接著再進行 Full-Text Predicates 的運算。研究[LC07] 設計 XSeek 系統，其作法如[XP05]一般，會先根據 keyword match 到的節點找到 VLCA，但是 Xseek 會再由 VLCA 往下走訪與 keyword 有關係的子樹。該作法分辨資料中可能的 entity、attribute 或 connection node，並分辨出 keyword 中可能是 search 的限制或是回傳的項目，來決定所要回傳的子樹範圍。研究[LC08]認為關鍵字或資料改變時，回傳的結果應該要符合 monotonicity 和 consistency 兩個特性。其中 monotonicity 是針對回傳結果數量的改變，而 consistency 則是探討回傳結果內容的改變。為了符合此兩種性質，該論文定義了 MaxMatch，也就是在 SLCA 之下符合 contributor 特性的節點才必須輸出。其中 contributor 節點之下包含關鍵字的子孫，會有些不在兄弟節點的子孫中。而 [LC07] 和 [LC08]的差別，在於是否只回傳使用者所需要的資料，還是直接回傳整棵樹而不做篩選。研究[LCC10]則改善[LC08]的效率，提出 MinMap 和 SingleProbe 兩種做法，主要在於減少去 index 存取的次數，而 SingleProbe 系統，在關鍵字頻率高低差越接近時，表現會較 MinMap 好。

另一方面，研究[LOF+08]則提出稱作EASE的方法，以便對Unstructured、Semi-structured和Structured Data，皆可做關鍵字的查詢。EASE的作法是將data表示成graph，然後利用adjacency matrix，解決r-radius Steiner graph的問題。該論

文也提出一個混合DB跟IR的觀點的scoring function。研究[TJM+08]是希望讓使用者在不需要知道複雜的schema以及query language的情況下，建構出自己的查詢句template。該論文首先將keyword match到schema graph上，並附上每個資料來源的權重值，然後在schema graph上找到top k的steiner tree，以產生top k的query。該系統會進一步針對所產生的查詢句提供答案以及對應的資料來源，以便使用者提供feedback給系統，讓系統可以學習並改變權重值。 [SM08]則是希望讓使用者不需要知道確切的XML結構，但仍然能夠對XML中的relation data做查詢。 該論文定義tree relation，允許所謂的「阿米巴結構」，該系統會自動取出所有對應到的不同的XML結構，並利用functional dependencies去掉不符合XML語意的結構。至於研究[ZC08]中，主要是探討如何快速的找出top-k的answer。該論文利用skyline所建立出來的layer，以及各layer中資料值的關連性，來建立dominant graph。該論文提出的走訪演算法，可將查詢的範圍侷限在skyline points的個數上，所以非常的有效率。

1.4 論文架構

本論文其餘各章節的架構如下：在第二章我們將簡介在此論文中所需要用到的相關定義，如：XML 文件、對 XML 文件編碼、And 和 Or 的查詢句，並對本論文所解決的問題進行定義以及範例的說明。在第三章中我們將介紹查詢句輸出的結果必須滿足的性質，並說明 AO_MaxMatch 符合輸出的性質並給予證明。在第四章中我們將說明 AO_MaxMatch 以及 AO_SingleProbe 的整體系統架構，我們將解說在系統中的每個模組，並說明該模組所使用之演算法。在第五章中，我們將以實驗分析兩系統的效率，以及回傳結果的品質測試。最後在第六章我們將提出結論，並指出此論文未來的研究方向。

第二章 相關定義

在本章中，我們將說明 XML 基本結構、本論文處理的查詢句表示法和欲輸出的結果，並針對我們所解決之問題給予範例說明與定義。

2.1 XML 基本架構

首先，我們介紹 XML 文件的基本架構。XML 文件以元素作為資料表示的基本單位，如 <name>AllStar </name>表示了一個 “name”元素，其內容為 “AllStar”。另外，元素間必須具有嚴謹的巢狀包含關係，譬如：

```
<team>
    <name> AllStar </name>
    <founded>1946</founded>
</team>
```

表示了 “team” 元素中包含了兩個元素，分別是 “name”和 “founded”。所以，我們通常將一份 XML 文件表示成一棵樹狀結構，如圖 2.1 所示。其中，實心圓表示一般元素節點，旁邊的文字是標籤名稱，例如 team 是一個標籤名稱。空心圓表示內容節點，其下的文字則是元素內容。直線則表示元素間的巢狀關係。在圖 2.1 的 XML 樹中，league 代表了文件的根元素，其下的 team 子元素，表示了 All Star 球隊的資料，包含其球隊名 (name)、創立年份 (founded)、和球員名單 (players)。其中球員名單 (players) 下的球員 (player) 的資料包含有球員名 (name)、球員位置 (position)、球衣號碼 (number)、以及國籍 (nationality)。注意到，該圖表示了樹 D1 和樹 D2，其中 D2 比 D1 多了一個元素 nationality。這兩棵樹將作為本論文的範例。

為了可以快速地推斷元素間的結構關係，常見的方法是替每一個元素進行編碼。在圖 2.1 中，我們標示的是「杜威編碼」(dewey code)，該方法替每個元素指定的編碼，主要是依照其父親的編碼，再加上自己是父親的第幾個子元素。譬如，league 為根節點，沒有父節點，其編號為“1”。它的子節點為 team，元素編碼為“1.1”，而 team 的三個子元素，由左至右依序為 “1.1.1”、“1.1.2”、“1.1.3”，依此類推。此編碼的好處，是可以由自己的編碼，立即得知自己所有祖先的編碼。注意到，內容節點也會被編號。

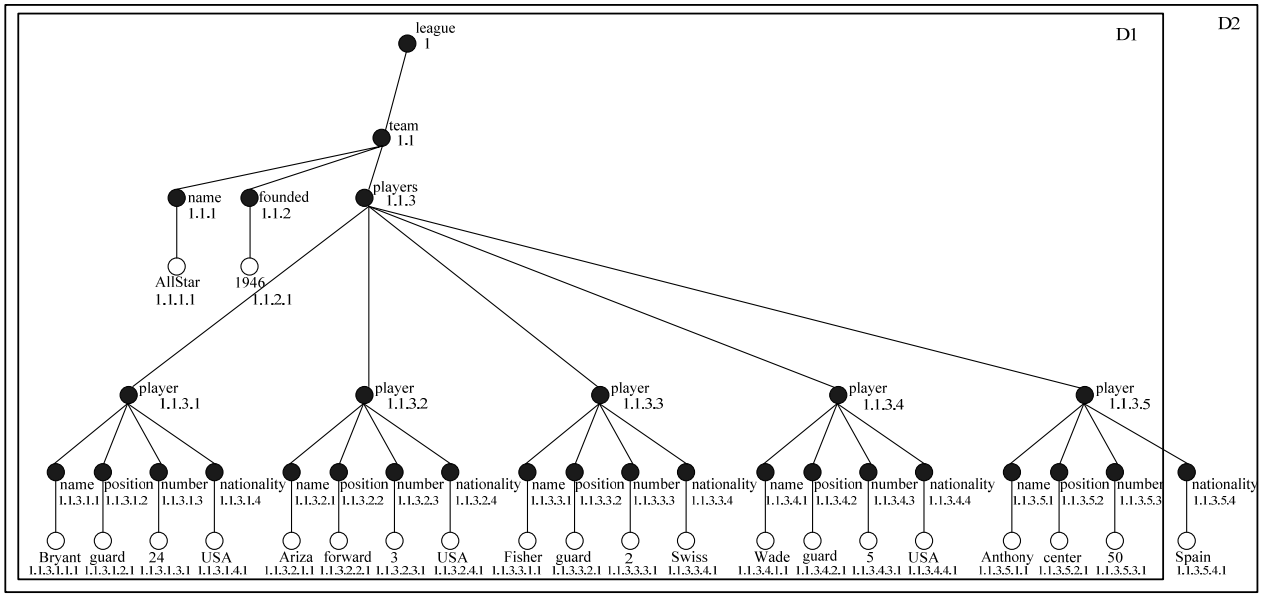

圖 2.1　XML 樹

2.2　And 查詢句的回傳結果

目前的研究，大多允許使用者同時列舉數個關鍵字，然後要求輸出的資料必須同時包含所有這些關鍵字。我們稱這類的查詢句為“AND”查詢句。在相關的研究中，我們以 MaxMatch [LC08]作為對照組。以下說明 MaxMatch 的輸出範例，

以及相關的定義。

假設使用者欲針對圖 2.1 XML 樹的 D2 找出「Ariza 這位球員所穿著的球衣號碼」，則可輸入 Q4 查詢句： (“Ariza” ∧ “number”)。而其方法只會回傳球員 “Ariza” 所穿著的球衣號碼為 “3” 號，如圖 2.3 所示，而不會輸出其他節點，如 position 或 nationality。

Q4	Ariza ∧ number　Ariza 這位球員所穿著的球衣號碼
Q5	AllStar ∧ Ariza ∧ number　AllStar 中 Ariza 這位球員所穿著的球衣號碼

圖 2.2 “AND” 查詢句範例

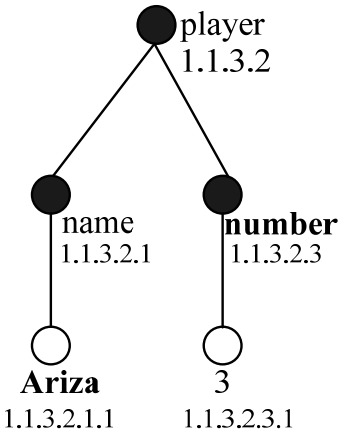

圖 2.3　Q4 的回傳結果 [MaxMatch]

以下我們介紹 MaxMatch 中的相關定義：

[定義 2.1]：對應 (Match)：若關鍵字 k 與 XML 文件中節點 u 名稱(或內容)相同，則稱節點 u 對應關鍵字 k。

[範例 2.1]：[Q4, D2]：給關鍵字 “Ariza”， 則 match 到圖 2.1 中 XML 樹 D2 的節點 1.1.3.2.1.1。若給關鍵字 “number”，則 match 到的節點為{ 1.1.3.1.3, 1.1.3.2.3, 1.1.3.3.3, 1.1.3.4.3, 1.1.3.5.3 }。

[定義 2.2]：Lowest Common Ancestor (LCA)：給予一群節點的集合 S1,…,Sn，每個 Si 為對應到關鍵字 ki 的集合，假如存在 v1 ∈ S1 ,…, vk ∈ Sk 使得 v = lca (v1,…, vk)，其中函式 lca 計算節點 v1,…, 節點 vk 最低的共同祖先，則所有節點 v 形成的集合，稱為 S1,...,Sn 節點集合的 LCA。

[範例 2.2]：參考範例 2.1，若 S1 為 Ariza 節點的集合，也就是 S1 = { 1.1.3.2.1.1 }，S2 為 number 節點的集合，也就是 S2 = { 1.1.3.1.3, 1.1.3.2.3, 1.1.3.3.3, 1.1.3.4.3, 1.1.3.5.3 }。此時取 v1 = Ariza (1.1.3.2.1.1)，v2 = number (1.1.3.1.4.1)，則其 lca 為 players (1.1.3)。若取 v2 = number (1.1.3.2.3)，則其 lca 仍為 player (1.1.3.2)。其餘的 number 節點，與 Ariza (1.1.3.2.1.1)的 lca 皆為 players (1.1.3)，所以，S1 和 S2 的 LCA 為 { 1.1.3, 1.1.3.2 }。

接下來，我們介紹最小最低共同祖先 (SLCA) 的定義，以代表一棵包含所有關鍵字的最小子樹，如下所述：

[定義 2.3]：Smallest Lowest Common Ancestor (SLCA)：若一個節點 v 屬於 LCA (S1,…, Sk)，而且對於所有 u 屬於 LCA (S1,…, Sn)，v 不是 u 的祖先，則所有節點 v 形成的集合，稱為 S1,...,Sn 節點集合的 SLCA。

[範例 2.3]：延續範例 2.2，在 LCA (S1, S2)中，因為 players (1.1.3)為 player (1.1.3.2)的祖先，所以只有 player (1.1.3.2)屬於 SLCA。

最後，我們定義 MaxMatch 的輸出：

[定義 2.4]：查詢結果：對於一棵 XML 樹 D 和 AND 組成的查詢句 Q，產生的查詢結果 R 的表示式為 R = (Q, D)，R 中的每一個查詢結果可表示成一棵樹，表示為 r = (t, M)。其中 t 為根節點，t 底下所有和關鍵字有關的節點 (relevant match) 我們稱作 M，其中 M 至少包含每個關鍵字一次。由查詢結果組成的樹，為在 D 中從 t 連接到 M 中的所有關鍵字節點，所形成的路徑。而查詢結果的個數，也等於 (t, M) 的個數。

我們舉例何為 relevant match：

[範例 2.4]：[Q5, D2]：此時有意義的查詢結果為 t = team (1.1)，M = { AllStar (1.1.1.1), Ariza (1.1.3.2.1.1), number (1.1.3.2.3) }。在此可發現，在 team 的底下的 nunber (1.1.3.1.1)節點不應該為 relevant match，而且不在 M 的集合內，因為它是 Bryan 的球員號碼，而不為 Ariza。最後查詢結果組成的樹，為 team 到 M 中所有節點組成的路徑，也包含節點的 value，其結果如圖 2.4 所示。

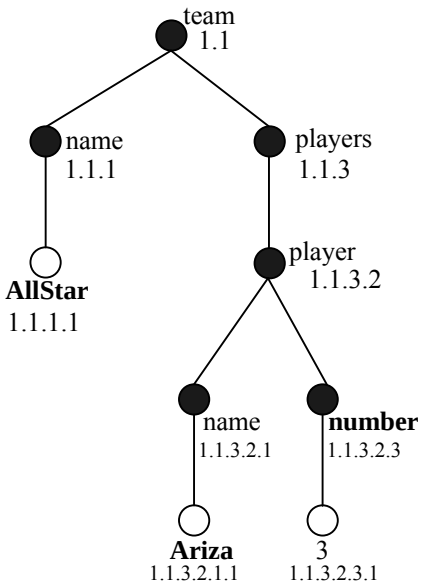

圖 2.4　Q5 的回傳結果 [MaxMatch]

2.3 問題定義

本論文所討論的查詢句，可以將關鍵字以 “AND” 或 “OR” 的 方式連結，譬如，如果我們要查詢國籍為 Swiss 或 Spain 的球員名字，針對 D2 可以下查詢句 Q7：name ∧ (Swiss ∨ Spain)，而其合理的查詢結果如圖 2.5 所示：

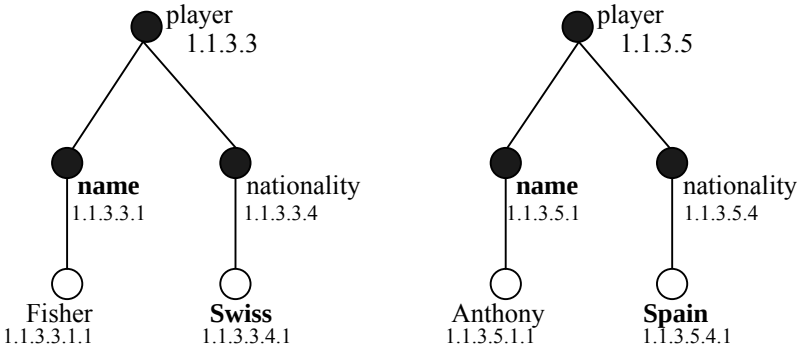

圖 2.5　Q7 的回傳結果

不過，此查詢結果並不符合 MaxMatch 的定義，如之下範例 2.5 所示。為了方便後續的討論，我們統一將查詢句以 CNF (Conjunctive Normal Form)表示，並將每個以 “AND” 隔開的 clause 稱作「項次」，譬如，Q7 共有兩個項次。

[範例 2.5]：[Q7, D2]:在此範例中，我們可以發現，樹根 player (1.1.3.3) 和 player (1.1.3.5) 底下並沒有同時擁有 “name”, “Swiss” , “Spain” 這三個關鍵字，但卻擁有查詢句 Q7 中的兩個項次。

因此，我們修正定義 2.2 中的 Sn 如下：

[定義 2.5]：Lowest Common Ancestor (LCA)：參考定義 2.2，但修正 Si 為對應到第 i 項次的集合。

所以針對 Q7，此時 S1 為對應到第一個項次 (name 節點) 的集合，也就是 S1 = { 1.1.3.1.1, 1.1.3.2.1, 1.1.3.3.1, 1.1.3.4.1, 1.1.3.5.1 }，S2 為對應到第二個項次，也就是 Swiss 節點和 Spain 節點的集合，S2 = { 1.1.3.4.1, 1.1.3.5.4 }，而得到的 SLCA 為 player (1.1.3.3) 和 player (1.1.3.5)，所以如圖 2.5 所示，輸出了兩個查詢結果。

接下來，我們同樣修正定義 2.4，重新定義查詢結果如下：

[定義 2.6]：查詢結果：對於一棵 XML 樹 D 和 AND-OR 查詢句 Q，產生的查詢結果 R 的表示式為 R = (Q, D)，R 中的每一個查詢結果可表示成一棵樹，表示為 r = (t, M)。其中 t 為根節點，t 底下所有的 relevant match 我們稱作 M，其中 M 至少包含每個項次內的某一個關鍵字。由查詢結果組成的樹，為在 D 中從 t 連接到 M 中的所有關鍵字節點，所形成的路徑。而查詢結果的個數，也等於 (t, M) 的個數。

定義 2.6 與定義 2.4 的差別在於，t 底下的 M 為至少包含每個項次內的某一個關鍵字，而不是包含查詢句內的每個關鍵字。

原始的 MaxMatch，提出如何判斷「相關的節點 (relevant match) 」，在本論文之中，我們將擴充相關的定義和演算法，以處理 And-Or 查詢句。以下我們針對輸出蘊含最多的資訊概念，舉一個範例：

[範例 2.6]：參考範例 2.5，在 D2 中的 player (1.1.3.1)、player (1.1.3.2)、player (1.1.3.3)、player (1.1.3.4)、player (1.1.3.5)的底下都有關鍵字 name，然而 player (1.1.3.3)底下多了 Swiss，而 player (1.1.3.5)底下多了 Spain，佔了 Q7 的第一和第二項次，因此我們認為 player (1.1.3.3)和 player (1.1.3.5)底下蘊含的資訊，多過於其他三個 player 底下所擁有的資訊，所以我們會選擇輸出 player (1.1.3.3)和 player (1.1.3.5)。

第三章 AO_MaxMatch

在本章中，我們首先描述我們輸出結果欲符合的性質，接著提出 AO_MaxMatch 的作法，最後證明該作法會符合上述的性質。

3.1 性質描述

MaxMatch 的作法提出 Monotonicity 以及 Consistency 兩個性質，用以描述當資料或查詢句的更新時，查詢結果個數的改變或查詢結果內容的改變，必須符合的性質。不過該論文查詢句內的關鍵字之間只能以“AND”方式連接，而本論文則允許關鍵字也能以“OR”連接，因此我們提出以下的擴充性質。在以下的描述中，我們會先描述每個性質，給予範例，再提出正式的定義。

Q6	player ∧ name ∧ (forward ∨ center) 尋找前鋒或後衛的球員名字
Q7	name ∧ (Swiss ∨ Spain) 尋找瑞士和西班牙球員的名字
Q8	name ∧ (Swiss ∨ Spain ∨ USA) 尋找瑞士或西班牙或美國的球員名字
Q9	All Star ∧ position ∧ (Bryant ∨ Ariza) 尋找 AllStar 中 Bryant 和 Ariza 的進攻位置
Q10	players ∧ name ∧ (forward ∨ center) 尋找前鋒或後衛的球員名字

圖 3.1 “AND-OR”查詢句範例

Data Monotonicity：若我們增加節點到資料中，則資料內容變的更加豐富，因此查詢結果的個數為非嚴格性的單調遞增。

[範例 3.1]：[Q7, D1], [Q7, D2]：[Q7, D1]想要找出「國籍為瑞士或是西班牙的球

員名字」。在 D1 中，合理回傳的子樹為以 player (1.1.3.3)為根節點的子樹，此時查詢結果的個數為 1，且此顆子樹底下 match 到的節點為 name (1.1.3.3.1)、Swiss (1.1.3.3.4.1)。現在我們想要查詢的資料為圖 2.1 XML 樹的 D2，此時除了得到上述的子樹外，我們會再得到一棵以 player (1.1.3.5)為根節點，查詢結果的個數增加至 2，而底下 match 到的節點為 name (1.1.3.5.1)和 Spain (1.1.3.5.4.1)的子樹。

然而在增加節點到資料後，我們仍可能得到相同數目的查詢結果。譬如：考慮圖 3.1 中 Q7：player ∧ name ∧ (forward ∨ center)，針對圖 2.1 XML 樹的 D1 或 D2，回傳的結果仍然是圖 2.4 的兩顆子樹，查詢結果的個數都是 2，此情況仍然符合性質中所提到的「非嚴格性」遞增。

[定義 3.1]：Data Monotonicity：給予 AND-OR 查詢句 Q，和兩份 XML 文件 D, D'，其中 D' = D ∪ {n}，n 為 XML 的節點且 n ∉ D。則在文件 D'中的查詢結果的個數，不應少於在文件 D 中的查詢結果的個數。

Query Monotonicity：若我們增加關鍵字到項次中 (or)，則此查詢句條件變的較不嚴格，因此，查詢結果的個數為非嚴格性單調遞增。但是若增加關鍵字使得項次 (and) 增加，則此查詢句條件變的較嚴格，因此，查詢結果的個數為非嚴格性遞減。

[範例 3.2]：[Q7, D2], [Q8, D2]：在範例 3.1 中針對 D2，所回傳的子樹分別是以 player (1.1.3.3)和 player (1.1.3.5)為根節點的兩顆子樹，查詢結果的個數為 2。接著我們考慮 Q8：name ∧ (Swiss ∨ Spain ∨ USA)，此查詢句是在 Q7 的第二個項次中多加了 USA 這個國籍。此時回傳的子樹分別是以 player (1.1.3.1)、player (1.1.3.2)、player (1.1.3.3)、player (1.1.3.4) 和 player (1.1.3.5) 為根節點的五棵子

樹，查詢結果的個數為 5。

若增加關鍵字到“AND”條件中時，則此查詢句條件變的較嚴格，因此，查詢結果的個數為非嚴格性單調遞減。由於這種情況在 MaxMatch 中已經提出，因此在本論文中不再多加贅述。

[定義 3.2]：Query Monotonicity：給予一份 XML 文件 D，和兩組 AND-OR 查詢句 Q, Q’，其中 Q’ = Q ∪ {k}，k 為新增在某一項次中的關鍵字，且 k ∉ Q。則在文件 D’中的查詢結果的個數，不會少於在文件 D 中的查詢結果的個數。若 k 為新增一項次，則在文件 D’中的查詢結果個數，不會多於在文件 D 中的查詢結果個數。

Monotonicity 性質描述查詢結果的個數隨著資料或是查詢句的更新，而有所改變。consistency 性質則描述查詢結果的內容隨著資料或是查詢句的更新，而有所改變。在描述此性質前，我們需先定義「差別樹」，所謂的差別樹，就是在兩棵查詢結果樹中，存在一棵最大子樹只屬於新增加的查詢結果樹，而不會在原來的查詢結果樹中出現。在此我們沿用原始 MaxMatch 中給予差別樹的定義如下：

[定義 3.3]：差別樹(Δ)：假設查詢句 Q 在資料 D 中，回傳結果為 R，而增加關鍵字到 Q 或增加節點到 D 後的回傳結果為 R’。一棵以 n 為根節點的差別樹 r’ ∈ R’ 的定義如下: 假設 desc-or-self (n, r’) ∩ R = ∅，而且 desc-or-self (parent (n, r’), r’) ∩ R ≠ ∅，則這棵以 n 為根節點的子樹稱為差別樹，R 與 R’的差別樹表示式為 Δ(R, R’)。

[範例 3.3]：參考範例 3.1，此時以 player (1.1.3.5)為根節點的子樹為一棵差別樹，

因為它是一棵只會在 Q8 於 D2 的查詢結果中會出現的最大的子樹，但卻不會在 D1 的查詢結果中出現。

Data Consistency：在增加資料之後，若有額外的子樹變成查詢結果的一部份，則此子樹應該包含新增的節點。

[範例 3.4]：參考範例 3.3，我們會再得到一棵以 player (1.1.3.5)為根節點的子樹，另外，此棵差別樹包含了新增的節點 nationality (1.1.3.5.4)和 Spain (1.1.3.5.4.1)，所以符合 Data Consistency。

[範例 3.5]：[Q6, D1], [Q6, D2]：此時得到的查詢結果皆為以 player (1.1.3.2)和 player (1.1.3.5)為根節點的子樹。由於這種情況沒有差別樹，因此 Data Consistency 直接成立。

[定義 3.4]：Data Consistency：給予 And-Or 查詢句 Q，和兩份 XML 文件 D, D'，其中 $D' = D \cup \{n\}$，n 為 XML 的節點且 $n \notin D$。若 $\Delta(R(Q, D), R(Q, D')) \neq \varnothing$，則每一棵差別樹必定包含節點 n。

Query Consistency：若我們增加關鍵字到某一項次中或成為新的項次，若有額外的子樹變成查詢結果的一部份，則此子樹應該至少包含一個新增的關鍵字節點。

[範例 3.6]：在範例 3.2 中，Q7 在 D2 中回傳的子樹分別是以 player (1.1.3.3)和 player (1.1.3.5)為根節點的兩顆子樹。Q8 在 D2 中回傳的子樹分別是以 player (1.1.3.1)、player (1.1.3.2)、player (1.1.3.3)、player (1.1.3.4) 和 player (1.1.3.5)為根節點的五棵子樹，比 Q7 多了以 player (1.1.3.1)、player (1.1.3.2) 和 player (1.1.3.4) 為根節點的三棵子樹。根據定義 3.3，多出的三棵子樹為差別樹，這三棵子樹底下也分

別多了 nationality (1.1.3.1.4, 1.1.3.2.4, 1.1.3.4.4) 和 USA (1.1.3.1.4.1, 1.1.3.2.4.1, 1.1.3.4.4.1)，其中 USA 為新增的關鍵字節點，表示差別樹包含了新增的關鍵字節點，符合 Query Consistency。

另外，若新增的關鍵字增加新的項次，此狀況在 MaxMatch 中已討論過，所以在此不再舉例。

[定義 3.5]：Query Consistency：給予一份 XML 文件 D，和兩組 And-Or 查詢句 Q 和 Q'，其中 Q' = Q ∪ {k}，且 k ∉ Q。若 Δ(R(Q, D), R(Q', D)) ≠ ∅，則每一棵差別樹必定至少包含一個關鍵字 k。

3.2 相關節點 (Relevant Matches) 的選擇

在很多時候，我們不能認為根節點底下的子樹其所有節點都是 relevant matches。因此，MaxMatch 作法，是認為每個節點若其底下 matches 的資訊不少於自己兄弟節點底下 matches 的資訊，則此節點為「contributor」，並認定此節點為 relevant match 而輸出。但由於本論文想要解決的查詢句為 "AND" 和 "OR" 組成的關鍵字，MaxMatch 提出的「contributor」定義無法滿足我們的需求，因此我們定義「AO_contributor」，若某節點符合「AO_contributor」的定義，我們則認定此節點為 relevant match 並輸出。在定義「AO_contributor」之前，我們先描述 MaxMatch 中的「Descendant Match」以及「contributor」，接著再擴充相關定義，以處理 And-Or 查詢句。另外，在定義 2.6 中提到的查詢結果，表示式為 r = (t, M)，以下我們將把 t 認為是 SLCA，這一節將討論 M 的輸出。

[定義 3.6]：Descendant Match：給予一棵 XML 樹 D 以及 And-Or 查詢句 Q，其

中節點 n ∈ D 的 descendant match 表示為 n 底下針對 Q 的關鍵字節點的集合，表示式為 dMatch (n)。

[定義 3.7]：contributor：給予一棵 XML 樹 D 以及查詢句 Q，其中節點 n ∈ D 對於 Q 若為 contributor，則必須滿足條件如下：

(i) n 有祖先 n1 在 SLCA 的集合中，或著 n 本身就在 SLCA 的集合中。

(ii) n 不能有兄弟 n2，使得 dMatch (n) ⊂ dMatch (n2)。

[範例 3.7]：[Q10, D2]：查詢句 Q10 共有四個關鍵字，我們從左到右依序將 players 定為第一關鍵字，name 為第二關鍵字，forward 為第三關鍵字，center 為第四關鍵字，考慮 player (1.1.3)，dMatch (1.1.3) = {1, 2, 3, 4}。而 players 底下的五位 player，dMatch (1.1.3.1) = {2}，dMatch (1.1.3.2) = {2, 3}，dMatch (1.1.3.3) = {2}，dMatch (1.1.3.4) = {2}，dMatch (1.1.3.5) = {2, 4}。我們可得到：dMatch (1.1.3.1) = dMatch (1.1.3.3) = dMatch (1.1.3.4) ⊂ dMatch (1.1.3.2)以及 dMatch (1.1.3.1) = dMatch (1.1.3.3) = dMatch (1.1.3.4) ⊂ dMatch (1.1.3.5)。所以 contributor 為 player (1.1.3.2) 和 player (1.1.3.5)。

接下來，我們提出新的定義：

[定義 3.8]：Descendant Clause Match：給予一棵 XML 樹 D 以及 And-Or 查詢句 Q，其中節點 n ∈ D 的 descendant clause match 表示為 n 底下針對 Q 所佔項次的集合，表示式為 dcMatch (n)。

[範例 3.8]：[Q10, D2]：查詢句 Q10 共有三個項次，我們從左到右依序將 players 定為第一項次，name 為第二項次，(forward ∨ center)定為第三項次。接著考慮圖 2.1 中的 D2，考慮 players (1.1.3)，dcMatch (1.1.3) = {1, 2, 3}。另外 players 底下

的五位 player，dcMatch (1.1.3.1) = {2}，dcMatch (1.1.3.2) = {2, 3}，dcMatch (1.1.3.3) = {2}，dcMatch (1.1.3.4) = {2}，dcMatch (1.1.3.5) = {2, 3}。我們可得到：dcMatch (1.1.3.1) = dcMatch (1.1.3.3) = dcMatch (1.1.3.4) ⊂ dcMatch (1.1.3.2) = dcMatch (1.1.3.5)。我們可觀察到，其包含關係和範例 3.7 不同。

在提出 AO_contributor 的定義之前，我們還必須重新定義 SLCA 如下：

[定義 3.9]：SLCA：對於 XML 文件 D 以及 And-Or 查詢句 Q，所產生的 Smallest lowest common ancestor (SLCA) 的集合，我們定義寫法為 SLCA(Q, D)。此集合由 t ∈ D 組成，並滿足下列兩個條件：

(i) dcMatch(t) 包含 Q 中的所有的項次。

(ii) 不存在 t 的子孫 t’，使得 dcMatch(t’) 包含 Q 中所有的項次。

[定義 3.10]：AO_contributor：給予一棵 XML 樹 D 和一組 AND-OR 查詢句 Q，對應 Q 在文件 D 中的節點 n 稱為 AO_contributor，必須滿足下列條件：

(i) n 的祖先必須屬於 SLCA 的集合，或者 n 自己就是 SLCA。

(ii) n 的子孫中必須有任一的關鍵字節點。

(iii) 對於 n 與 n 的任意兄弟節點 n2，其下子孫的關鍵字節點以及關鍵字節點佔有的項次作比較，則 n 與 n2 會有下列四種情形：

Case1： 若 dcMatch (n) ⊂ dcMatch (n2)，但 dMatch (n) ⊄ dMatch (n2)，則 n 為 AO_contributor。

Case2： 若 dcMatch (n) ⊂ dcMatch (n2)，且 dMatch (n) ⊂ dMatch (n2)，則 n 不為 AO_contributor。

Case3： 若 dcMatch (n) ⊄ dcMatch (n2)，且 dMatch (n) ⊄ dMatch (n2)，則 n 為 AO_contributor。

Case4： 若 dcMatch (n) ⊄ dcMatch (n2)，但 dMatch (n) ⊂ dMatch (n2)，

則 n 為 AO_contributor。

若一個節點 n 已經滿足定義 3.10 中的(i)和(ii)後，針對(iii)我們將 AO_contributor 的四種 case 以下面的狀態圖表示。注意到，MaxMatch 中的 contributor 定義，符合 AO_contributor 中的 case2 和 case3，因為在其查詢句中，項次即等於關鍵字，所以可以把 contributor 看作是 AO_contributor 的一種特例。接下來我們說明四種 case 是否為 AO_contributor 的理由：首先，若節點 n 其底下的項次與關鍵字皆被兄弟節點包含 (case2)，則可確切了解 n 底下所擁有的資訊，其兄弟節點已經擁有，甚至更多，所以 n 不為 AO_contributor。相反的，若節點 n 其底下的項次與關鍵字皆不被兄弟節點 n2 包含 (case3)，則可確切了解 n 底下所擁有的資訊，其兄弟節點皆不完全擁有，因此 n 為 AO_contributor。而當節點 n 只有項次 (case1) 或只有關鍵字 (case4) 被兄弟節點包含時，我們認為 n 在項次或關鍵字上，所擁有的資訊仍勝過自己的兄弟節點，因此認定 n 為 AO_contributor。注意到的是，我們提到的包含，不包括等於的情況，因為當兩個節點底下所佔有的項次和關鍵字都相同時，我們認為這兩節點底下仍可能代表著兩筆不同的資訊。所以注意到，case 4 的情況，會是關鍵字被包含，而項次等於的情形。

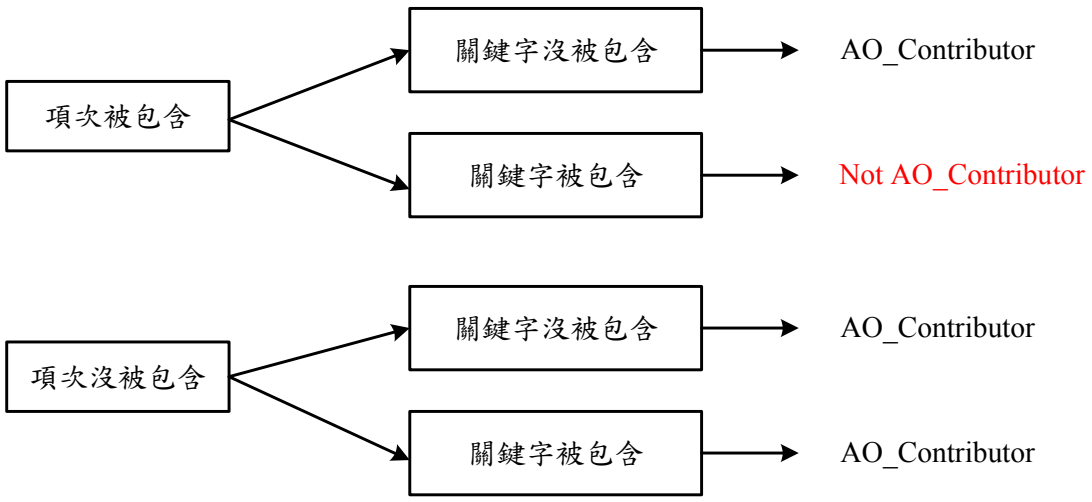

圖 3.2　是否為 AO_contributor 四種 case 狀態圖

Q11	players ∧ (5 ∨ 7) ∧ (guard ∨ forward) ∧ (Spain ∨ USA) ∧ name 尋找球員名字，球衣號碼為 5 號或 7 號，進攻位置為後衛或前鋒，國籍為西班牙或美國
Q12	players ∧ (center ∨ guard) ∧ USA ∧ name 尋找球員名字，進攻位置為中鋒或後衛，國籍為美國
Q13	Spain ∧ (center ∨ forward) ∧ (3 ∨ 13) ∧ name 尋找球員名字，國籍為西班牙，進攻位置為中鋒或前鋒，球衣號碼為 3 號或 13 號

圖 3.3　判斷 AO_contributor 前三種 case 的查詢句範例

在定義完 AO_ontributor 後，我們針對四種 case 分別給予範例說明：

[範例 3.8]：[Q11, D2]：根據 SLCA 的定義，此時可得到的 SLCA 節點為 players (1.1.3)。如圖 3.4，若我們觀察 player (1.1.3.2)以及 player (1.1.3.4)底下的 matches，dcMatch (1.1.3.2) = { 3, 4, 5 }，dMatch (1.1.3.2) = { 5, 7, 8 }。而 dcMatch (1.1.3.4) = { 2, 3, 4, 5 }，dMatch (1.1.3.4) = { 2, 3, 7, 8 }。可以發現 dcMatch (1.1.3.2) ⊂ dcMatch (1.1.3.4)且 dMatch (1.1.3.2) ⊄ dcMatch (1.1.3.4)，符合 case1，player (1.1.3.2) 跟兄弟 player (1.1.3.4) 的比較結果為 AO_contributor。就查詢句來看，此查詢句想要找的球員，搜尋的條件為「球衣號碼」、「進攻位置」和「國籍」。就回傳結果來看，player (1.1.3.2)在球衣號碼這條件上雖然都沒有滿足，但卻擁有 forward 這個進攻位置，是 player (1.1.3.4)所沒有的，因此我們仍認為 player (1.1.3.2)為 AO_contributor。

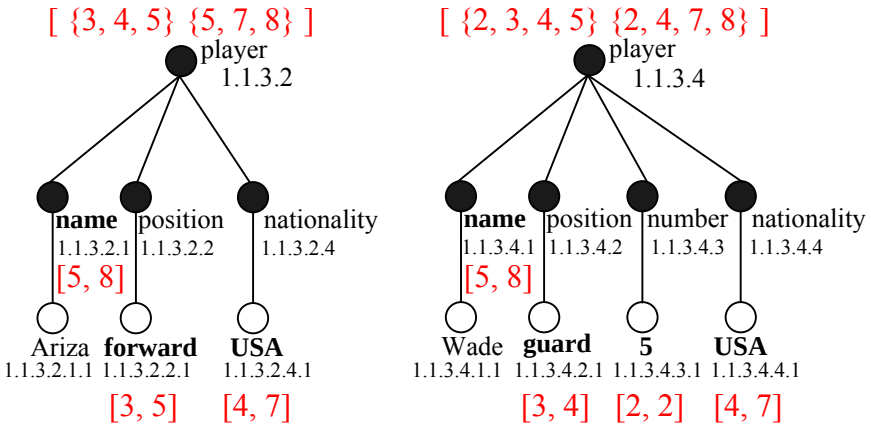

圖 3.4　player (1.1.3.2) 以及 player (1.1.3.4)，其中 match 到的關鍵字為粗體，關鍵字節點底下[a, b]，a 表示所佔的項次，b 表示所佔的關鍵字。而 player [{c}, {d}]，集合 c 表示底下所佔的項次，集合 d 表示底下所佔的關鍵字

[範例 3.9]：[Q12, D2]：再來考慮查詢句 Q12。如圖 3.5，若我們觀察 player (1.1.3.1) 以及 player (1.1.3.2)底下的 matches，dcMatch (1.1.3.1) = { 2, 3, 4 }，dMatch (1.1.3.1) = { 3, 4, 5 }。而 dcMatch (1.1.3.2) = { 3, 4 }，dMatch (1.1.3.2) = { 4, 5 }。可以發現 dcMatch (1.1.3.2) ⊂ dcMatch (1.1.3.1)且 dMatch (1.1.3.2) ⊂ dMatch (1.1.3.1)，符合 case2，所以 player (1.1.3.2) 跟兄弟 player (1.1.3.1) 的比較結果不為 AO_contributor。就查詢句來看，此查詢句想要找的球員，搜尋的條件為「進攻位置」和「國籍為 USA」。就回傳結果來看，player (1.1.3.2)雖然國籍為 USA，但進攻位置卻不符合搜尋條件，而其兄弟節點 player (1.1.3.1)符合進攻位置的條件，且亦為 USA 的國籍，所含的資訊皆蓋過 player (1.1.3.2)所擁有的，因此我們認為 player (1.1.3.2)不為 AO_contributor。

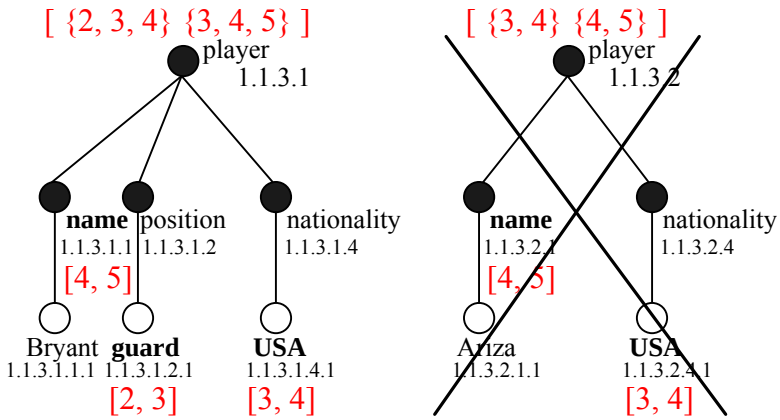

圖 3.5　player (1.1.3.1)以及 player (1.1.3.2)

[範例 3.10]：[Q13, D2]：接著考慮查詢句 Q13。如圖 3.6，我們觀察 player (1.1.3.2)以及 player (1.1.3.5)底下的 matches，dcMatch (1.1.3.2) = {2, 3, 4}，dMatch (1.1.3.2) = { 3, 4, 5 }。而 dcMatch (1.1.3.5) = { 1, 2, 4 }，dMatch (1.1.3.5) = { 1, 2, 5 }。可以發現 dcMatch (1.1.3.2) ⊄ dcMatch (1.1.3.5) 且 dMatch (1.1.3.2) ⊄ dMatch (1.1.3.5)，符合 case3，因此 player (1.1.3.2)跟兄弟 player (1.1.3.5)的比較結果為 AO_contributor。就查詢句來看，此查詢句想要找的球員，搜尋的條件為「進攻位置」、「西班牙的國籍」和「球衣號碼為 3 號或 13 號」。就回傳結果來看，player (1.1.3.2)與 player (1.1.3.5)皆滿足進攻位置的搜尋條件，而西班牙的國籍和球衣 3 號則分散在兩個 player 底下，因此我們認為兩個 player 擁有的資訊是不能互相取代的，因此 player (1.1.3.2)相對於 player (1.1.3.5)為 AO_contributor，反之，player (1.1.3.5) 相對於 player (1.1.3.2)亦為 AO_contributor。

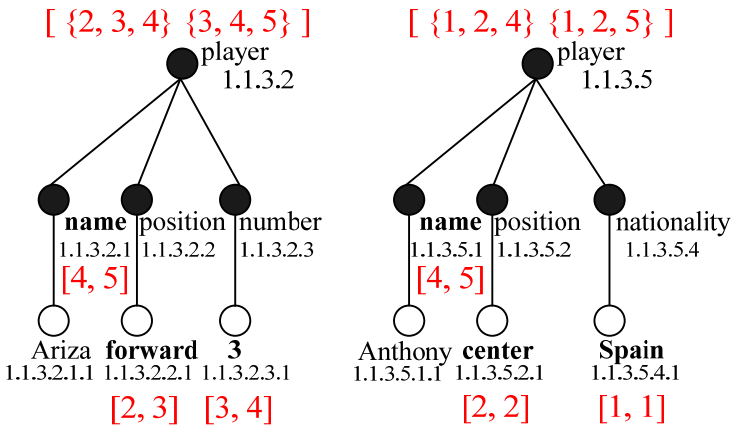

圖 3.6　player (1.1.3.2)以及 player (1.1.3.5)

[範例 3.11]：[Q14, 圖 3.7]：由於第四種 case 以圖 2.1 的文件較難表達，所以我們以圖 3.7 的文件，並考慮查詢句 Q14，輸出結果如圖 3.9。圖 3.9 中，若我們觀察 paper (1.1.3.1)以及 paper (1.1.3.2)底下的 matches，dcMatch (1.1.3.1) = { 2, 3 }，dMatch (1.1.3.1) = { 2, 4 }。dcMatch (1.1.3.2) = { 2, 3 }，dMatch (1.1.3.2) = { 2, 3, 4 }。可以發現 dcMatch (1.1.3.1) $\not\subset$ dcMatch (1.1.3.2) 且 dMatch (1.1.3.1) $\subset$ dMatch (1.1.3.2)，符合 case4，因此 paper (1.1.3.1)跟兄弟 paper (1.1.3.2)的比較結果為 AO_contributor。就查詢句來看，此查詢句想要找的 paper，搜尋的條件為「分類為 XML，作者名字為 John 或 Tom」。就資料性質來看，一個作者是很有可能撰寫多篇 paper 的。就回傳結果來看，由於 John 跟 Tom 是在 Or 條件內，我們認為 paper (1.1.3.2)滿足了 Or 內的所有關鍵字，而 paper (1.1.3.1)雖然只滿足 Or 的一個關鍵字，但因為項次與 paper (1.1.3.2)相同，再加上以 Or 的語義來看，paper (1.1.3.1)是輸在同一個項次內的 Or 的關鍵字，因此我們認為 paper (1.1.3.1)仍為 AO_contributor。

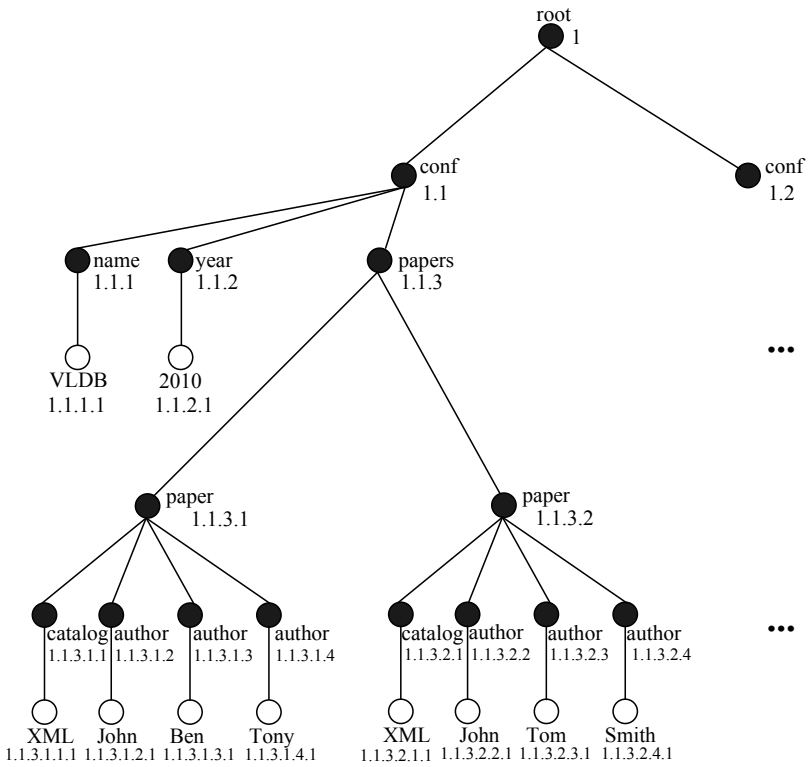

圖 3.7　XML 樹

Q14	VLDB ∧ XML ∧ (John ∨ Tom) 尋找 VLDB 會議中，分類為 XML，作者名字為 John 或 Tom 的 paper

圖 3.8　判斷 AO_contributor 第四種 case 的查詢句範例

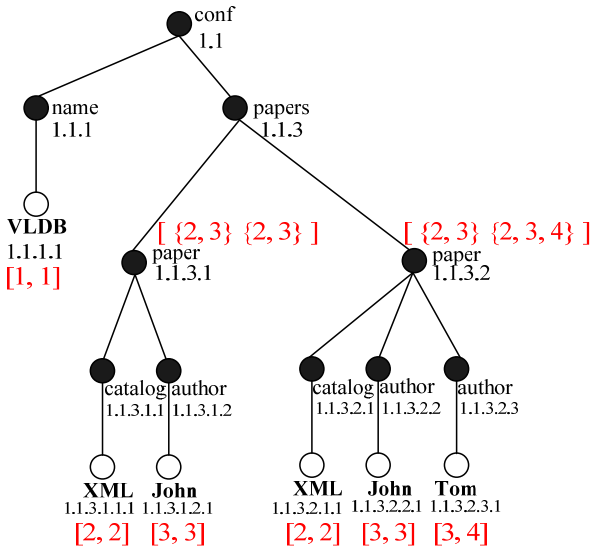

圖 3.9　paper (1.1.3.1)以及 paper (1.1.3.2)

3.3　AO_MaxMatch 符合性質

根據 AO_contributor 的定義，我們提出 AO_MaxMatch 的作法。根據 Relevant Match 和查詢結果的定義，以下我們將對 AO_MaxMatch 能夠符合 Monotonicity 以及 Consistency 提出證明。

首先，其 Relevant Match 的定義如下：

[定義 3.11]：Relevant Match：給予一棵 XML 樹 D 和查詢句 Q，其中節點 n ∈ D 針對 Q，若為 relevant Match，則必須滿足下列條件：

(i)　n 有祖先 n1 ∈ SLCA(Q, D)，或者 n ∈ SLCA(Q, D)。

(ii)　從節點 n1 至節點 n 組成的路徑，路徑上每個節點都必須為 AO_contributor。

[性質 3.1]：(t, M) 表示一個查詢結果，其中 t ∈ SLCA(Q, D)，M 為 t 底下 relevant match 的集合，則 (t, M) 符合定義 2.6 的查詢結果。

證明：我們必須證明 M 至少包含 Q 中的每一個項次至少一次。假設 M'為 t 底下所有的關鍵字節點，根據定義 3.9，得知 M'至少包含 Q 中的所有項次至少一次。接著我們移除 M'中不相關的關鍵字節點產生了 M，若有一個不相關的關鍵字節點 m1 對應到關鍵字 k，且被刪除掉，那麼 m1 的祖先 n1 就不是 AO_contributor，所以，n1 一定存在一個兄弟節點 n2 且為 AO_contributor。那麼 dcMatch (n1) ⊂ dcMatch (n2) (根據定義 3.10 的 case2)，也就是 k 所佔的項次在 dcMatch (n2) 中一定存在。最後根據歸納法，必定存在一個以 t 為根節點的子樹，其底下的關鍵字節點一定包含 Q 中所有項次，最後 M 一定包含所有項次的關鍵字至少一個。

根據 releveant match 的正式定義，我們修正定義 2.6 如下：

[定義 3.12]：AO_MaxMatch 的查詢結果：給予一棵 XML 樹 D 和 And-Or 查詢句 Q，每一個由 AO_MaxMatch 產生的查詢結果我們定義為 r = (t, M)，其中每一個 t ∈ SLCA(Q, D)，以 t 為根節點，底下相對於 Q 的 relevant match 稱為 M。一個查詢結果樹 r，為由 t 開始到底下的 M 中的 relevant match 組成的路徑。

[性質 3.2]：AO_MaxMatch 滿足 data monotonicity

證明：給予 AND-OR 組成的查詢句 Q，和兩份 XML 文件 D, D'，其中 D' = D ∪ {n}，n 為 XML 的節點且 n ∉ D。根據定義 3.12，由 AO_MaxMatch 產生查詢結果的個數相等於 SLCA 的個數。對於任意的 t ∈ SLCA(Q, D)，則有下面情況。其中 SLCA(Q, D)表示查詢句 Q 在文件 D 中產生的 SLCA；|R(Q, D)|表示查詢句 Q 在文件 D 中產生的查詢結果個數：

- t ∈ SLCA(Q, D')，則個數不會變動。
- t ∉ SLCA(Q, D')，dcMatch(t) 仍包含 Q 的所有項次，但新增了 n 後，t 不再是最低的 SLCA，那麼至少存在一個 t 的子孫為 SLCA 節點，且在 SLCA(Q, D')的集合裡，則 |R(Q, D')| ≥ |R(Q, D)|。

因此，$|R(Q, D')| \geq |R(Q, D)|$成立。

[性質 3.3]：AO_MaxMatch 滿足 query monotonicity

證明：給予一份 XML 文件 D，和兩組 AND-OR 的查詢句 Q, Q'，其中 $Q' = Q \cup \{k\}$，且 $k \notin Q$：

(i) 若 k 為項次內的關鍵字，對於任意的 $t \in SLCA(Q, D)$，則有下面情況：

- $t \in SLCA(Q', D)$，則個數不會變動。
- $t \notin SLCA(Q, D')$，dcMatch(t) 仍包含 Q 的所有項次，但新增了 n 後，t 不再是最低的 SLCA，那麼至少存在一個 t 的子孫為 SLCA 節點，且在 SLCA(Q, D')的集合裡，則 $|R(Q, D')| \geq |R(Q, D)|$。

因此，$|R(Q', D)| \geq |R(Q, D)|$成立。

(ii) 若 k 為新增項次的關鍵字，對於任意的 $t \in SLCA(Q, D)$，則有下面情況：

- $t \in SLCA(Q', D)$，則個數不會變動。
- $t \notin SLCA(Q', D)$，表示查詢句新增項次後，t 不再包含所有的項次，而且根據定義 3.9，t 的子孫不會在 SLCA(Q', D)裡面，或者最多只有一個 t 的祖先為 SLCA 節點，並且會在 SLCA(Q', D)的集合中，則$|R(Q', D)| \leq |R(Q, D)|$。

因此，$|R(Q', D)| \leq |R(Q, D)|$成立。

[性質 3.4]：AO_MaxMatch 滿足 data consistency

證明：給予 AND-OR 的查詢句 Q，和兩份 XML 文件 D, D'，其中 $D' = D \cup \{n\}$，n 為 XML 的節點且 $n \notin D$。則有下列兩種情況：

(i) 若 $\Delta(R(D, Q), R(D', Q)) = \varnothing$，則 AO_MaxMatch 直接成立。

(ii) 若 $\Delta(R(D, Q), R(D', Q)) \neq \varnothing$，假設 n1 是 $\Delta(R(D, Q), R(D', Q))$差別

樹的根節點，n2 是 n1 的父親節點，根據定義 3.3 和定義 3.10，n2 為 D 跟 D'的 AO_contributor，但 n1 是 D'的 AO_contributor 而不為 D 的 AO_contributor。此時，存在一個 n1 的兄弟節點 n3，使得在 D 中，dcMatch (n1) ⊂ dcMatch (n3) 且 dMatch (n1) ⊂ dMatch (n3)成立 (根據定義 3.10 的 case2 得知)，但在 D'中不成立。由此可知，在 D'中，新增的節點 n 使得 dcMatch (n1) ⊄ dcMatch (n3) 或 dMatch (n1) ⊄ dMatch (n3) (滿足定義 3.10 的 case1 或 case3 或 case4)，表示這棵以 n1 為根節點的差別樹必定包含新增的節點 n (而且 n 必定為查詢句中的某個關鍵字)。

[性質 3.5]：AO_MaxMatch 滿足 query consistency

證明：給予一份 XML 文件 D，和兩組 AND-OR 的查詢句 Q, Q'，其中 Q' = Q ∪ {k}，且 k ∉ Q。則有下列兩種情況:

(i) 若 Δ(R(D, Q), R(D, Q')) = ∅，則 AO_MaxMatch 直接成立。

(ii) 若 Δ(R(D, Q), R(D, Q')) ≠ ∅，假設 n1 是 Δ(R(D, Q), R(D, Q'))差別樹的根節點，n2 是 n1 的父親節點。根據定義 3.3 和定義 3.10，n2 為 Q 在 D 以及 Q'在 D 的 AO_contributor，但 n1 是 Q'在 D 的 AO_contributor 而不為 Q 在 D 的 AO_contributor。此時，存在一個節點為 n1 的兄弟節點 n3：

- 若 k 為對於 Q 新增項次的關鍵字：Q 在 D 中，dcMatch (n1) ⊂ dcMatch (n3) 且 dMatch (n1) ⊂ dMatch (n3) (根據定義 3.10 的 case2 得知)，但 Q'在 D 中不成立。由此可知，新增的關鍵字 k 使得 dcMatch (n1) ⊄ dcMatch (n3) 且 dMatch (n1) ⊄ dMatch (n3) (因為新增項次的關鍵字，就等同於新增項次和新增關鍵字，所以 n3 必定沒有此項次和關鍵字)，滿足定義 3.10 的 case3，且此

差別樹必定包含新增的關鍵字 k。

- 若 k 為在 Q 的項次內新增的關鍵字：Q 在 D 中，dcMatch (n1) ⊂ dcMatch (n3) 且 dMatch (n1) ⊂ dMatch (n3) (根據定義 3.10 的 case2 得知)，但 Q'在 D 中不成立。由此可知，新增的關鍵字 k 使得 dMatch (n1) ⊄ dMatch (n3) (滿足定義 3.10 的 case1 或 case3)，所以此差別樹必定包含新增的關鍵字 k。

第四章 AO_MaxMatch和AO_SingleProbe演算法

4.1 AO_MaxMatch 系統架構

AO_MaxMatch 系統的架構如圖 4.1。首先，使用者輸入一個 “AND” 和 “OR” 組成的查詢句，經過 FindMatch 模組，透過對 XML 文件編 dewey id 建置而成的索引，得到所有關鍵字的 dewey id。接著，在 FindSLCA 模組中，採用[XP05] Scan Eager 的方式，處理所有的 dewey id，得到此查詢句的所有 SLCA。下一步，GroupMatches 模組將所有關鍵字的 dewey id 依序對應到所屬的 SLCA，形成與 SLCA 相同數量的 group。然後將所有的 group 送進 ConstructSLCATree 模組中，建立成以 SLCA 為根節點的樹，樹的 path 為所有關鍵字至 SLCA 所有的節點連接而成。最後在 PruneAndOutput 模組中，將以 SLCA 為根節點的樹，採 preorder 方式造訪所有節點，並由 AO_contributor 當作最後節點輸出的篩選條件。另外，由於空間的限制，我們將 XML 文件編 dewey id、Dewey id to Tag 的 index 和 Tag to Dewey id 的 index 之演算法，置於附錄 A、附錄 B 和附錄 C。

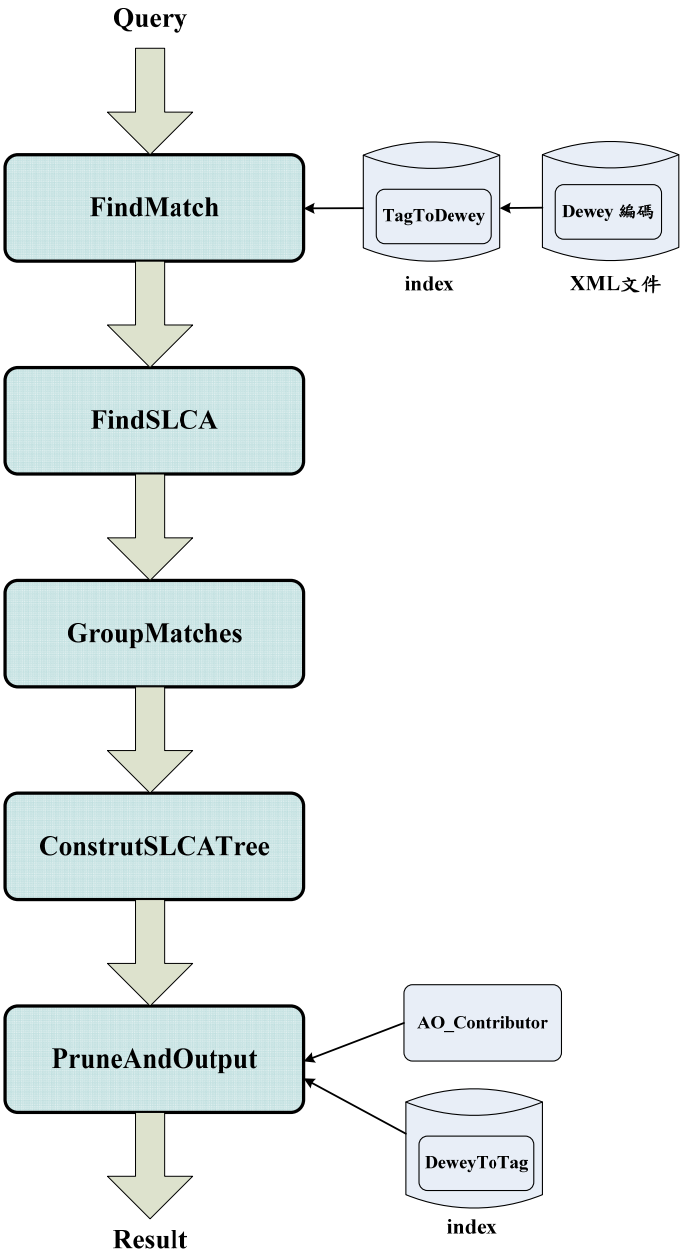

圖 4.1　AO_MaxMatch 系統架構

4.2　FindMatch、FindSLCA 和 GroupMatches 模組

首先我們將 XML 文件節點編好 dewey id，建立好索引，索引的 key 為每個節點的標籤，索引的 value 為對應到該標籤所有節點的 dewey id。接著在 FindMatch 模組中，我們將使用者輸入的查詢句，依照查詢句的項次為分界，將

項次內所有的關鍵字當作索引的 key，去索引查詢得到所有關鍵字的 dewey id，稱作 KwMatch，同時我們將每一個項次蒐集到的 dewey id 的集合稱作一個 S。下一步，在 FindSLCA 模組中，將所有 S 內 dewey id 數量最少的集合當作 S1，其餘依序為 S2…Sn，採用[XP05]的 Scan Eager 方法，得到此查詢句的所有 SLCA。最後在 GroupMatches 模組中，將 KwMatch 中所有 dewey id 依所屬的 SLCA 分配，形成與 SLCA 個數相同的 group 個數，且每個 group 內容為 SLCA 以及底下的關鍵字。這裡基本上與[XP05]的演算法相同。其中 L05 使用的排序演算法為 C++ 標準函式庫中 (STL)的 introsort algorithm，而時間複雜度方面，average case 與 worst case 皆為 O(n log(n))。

[範例 4.1]：以 Q12 查詢句為例，在 L01 抓取所有關鍵字的 dewey id，kwMatch = { 1.1.3, 1.1.3.1.2.1, 1.1.3.3.2.1, 1.1.3.4.2.1, 1.1.3.5.2.1, 1.1.3.1.4.1, 1.1.3.2.4.1, 1.1.3.4.4.1, 1.1.3.1.1, 1.1.3.2.1, 1.1.3.3.1, 1.1.3.4.1, 1.1.3.5.1 }。接著在 L02 和 L03，分別取 players = S1 = { 1.1.3 }，(center or guard) = S2 = { 1.1.3.1.2.1, 1.1.3.3.2.1, 1.1.3.4.2.1, 1.1.3.5.2.1 }，USA = S3 = { 1.1.3.1.4.1, 1.1.3.2.4.1, 1.1.3.4.4.1 }，name = S4 = { 1.1.3.1.1, 1.1.3.2.1, 1.1.3.3.1, 1.1.3.4.1, 1.1.3.5.1 }，送進 L04，得到 SLCA 節點為{ 1.1.3 }。在 L05 中將 KwMatch 依 dewey id 作好排序後，將 KwMatch 和 SLCA 節點送進 L06，得到 group[1] = { t = 1.1.3, M = { 1.1.3, 1.1.3.1.1, 1.1.3.1.2.1, 1.1.3.1.4.1, 1.1.3.2.1, 1.1.3.2.4.1, 1.1.3.3.1, 1.1.3.3.2.1, 1.1.3.4.1, 1.1.3.4.2.1, 1.1.3.4.4.1, 1.1.3.5.1, 1.1.3.5.2.1 } }。

演算法名稱：AO_MaxMatch
輸　　入：(k1 or k2 or ...) and...(...or kn)
輸　　出：tag’s name of n

```
L01   kwMatch ← FindMatch(k1…kn)
L02   {S1 is the least number of dewey id of each clause, the rest of clause are
L03   S2…Sn, where n is the number of clause}
L04   SLCA ← FindSLCA(S1…Sn)
L05   Sort(KwMatch)
L06   group ← groupMatches(kwMatch, SLCA)
L07   for each group[j] = (t, M) do
L08      ConstrutSLCATree(t, M)
L09      PruneAndOutput(t)
```

圖 4.2　AO_MaxMatch 演算法

4.3　ConstructSLCATree 模組

在介紹 ConstrucSLCATree 模組前，我們先介紹需要使用到的資料結構。在建樹的過程中，新增的每個節點擁有三種布林陣列，分別是 clauseMatch、keyMatch、ClauseKeySet，其中 ClauseKeySet 為 struct 型態的陣列，成員有兩個，其中一個為 cbit，型態為布林值，另一個為 KeyMatchSet，型態為布林陣列。

clauseMatch：此布林陣列用來記錄節點對應到的查詢句的項次，並在對應位置記錄 bit 值為 1，陣列大小為查詢句項次的個數，初始值為 0。

keyMatch：此布林陣列用來記錄節點對應到的查詢句的關鍵字，並在對應位置記錄 bit 值為 1，陣列大小為查詢句關鍵字的個數，初始值為 0。

ClauseKeySet：此陣列為 struct 型態，陣列大小為 $2^{clauseMatch.size}$，用來將此節點所有小孩的 clauseMatch 轉成十進位值後，在對應位置記錄成員 cbit 值為 1 (cbit 初

始值皆為 0)。而另一個成員為 KeyMatchSet，陣列大小為 $2^{keyMatch.size}$，初始值為 0，用來將此節點所有小孩的 keyMatch 轉成十進位值後，在相對應的項次位置底下的成員 KeyMatchSet 的對應位置記錄 bit 值為 1。當我們在決定某節點是否為 AO_contributor 時，必須去跟此節點的所有兄弟節點作比較。此時去父親節點的 ClauseKeySet，從自己的 cbit 值為 1 的位置，往後檢查成員 cbit 為 1 的位置值，再將自己的 clauseMatch 的 bit 值與此位置值的二進位值作 and 運算，若得到的值與自己的 clauseMatch 的 bit 值相同，表示項次被包含。接著檢查成員 KeyMatchSet，同樣從自己的 keyMatch 的十進位值的位置，往後檢查布林值為 1 的位置，再將自己的 keyMatch 與此位置值的二進位值作 and 運算，若得到的值與自己的 clauseMatch 的 bit 值相同，表示關鍵字被包含。

[範例 4.2]：延續範例 3.9 [Q12, D2]，並參考圖 3.5。兩個 player (1.1.3.1)其底下對應到的項次分別為{ 2, 3, 4 }，關鍵字為{ 3, 4, 5 }，因此 player (1.1.3.1)將在 clauseMatch 的第 2、3 和 4 的位置記錄為 1，bit 值為 1110，keyMatch 的第 3、4 和 5 的位置記錄為 1，bit 值為 11100。同理類推，player (1.1.3.2)的 clauseMatch 值為 1100，keyMatch 值為 11000。接著在兩個 player 的父親節點 players (1.1.3)，記錄此兩個小孩的資訊，所以在 players 的 ClauseKeySet 的第 14 個位置 (為 player (1.1.3.1)的 clauseMatch bit 值 1110)，紀錄 cbit 值為 true，同樣位置的另外一個成員 KeyMatchSet 的第 38 個位置 (為 player (1.1.3.1)的 keyMatch bit 值 11100 轉十進位值為 38)， 紀錄為 true。同理類推，在 ClauseKeySet 紀錄 player (1.1.3.2)的資訊，因此在第 12 個位置記錄 cbit 為 true，同樣位置的另外一個成員 KeyMatchSet 的第 24 個位置紀錄為 true，設定後的結果如圖 4.4 (a)所示。最後，若 player (1.1.3.2)在檢查是否被兄弟包含時，只需從父親 player (1.1.3)的 ClauseKeySet 的第 13 的位置開始尋找成員 cbit 值為 true 的位置，會發現在第 14 個位置 cbit 為 true (此為兄弟節點 player (1.1.3.1)的資訊)，接著將自己的 clauseMatch bit 值與 14 的二進位

值 and 運算，得到 1100，與自己的 clauseMatch bit 值相同，表示項次被包含，同理類推檢查關鍵字，也可發現被包含，結果如圖 3.5 所示，player (1.1.3.2)不為 AO_contributor。

接著我們介紹 ConstructSLCATree 演算法，在 ConstructSLCATree 模組中，將會一一接受上述的 group，並且建立成以 SLCA 為根節點的子樹。對照圖 4.3 ConstructSLCATree 演算法，首先，在 L03，關鍵字節點以 reverse preorder 順序 (也就是 bottom-up, rightmost-child-first)，依序新增從關鍵字節點到 start 中的每個節點，並連接成路徑。接著，在 L05，若為 leaf 節點，也就是關鍵字節點，則先檢查對應查詢句的 clause 以及 keyword，然後在 clauseMatch 以及 keyMatch 的相對位置記錄 bit 值為 1，若新增的節點為非關鍵字節點，則執行 L10，將這條路徑的小孩紀錄的 clause 和 keyword 的 bit 值作 OR 運算紀錄給自己。下一步執行 L14，若此節點不為下一個關鍵字節點的祖先，表示下一個關鍵字節點所建立的路徑中的所有節點將不可能為此節點的小孩，也就是此節點的小孩已經都造訪過了，此時將自己所有小孩的 clauseMatch 和 keyMatch 的 bit 值由二進位轉成十進位，並在自己的 ClauseKeySet 陣列紀錄 clauseMatch 轉十進位後的相對位置設為 true，ClauseKeySet 的成員 KeyMatchSet 陣列紀錄 keyMatch 轉十進位後的相對位置設值為 true，完成此節點對自己所有小孩的紀錄。

另外，當 L04 for 迴圈結束後，表示這個關鍵字至 start 這條路徑的節點已經走訪完畢，下一個 start 值的選擇方式，則在 L25 由下一個關鍵字與下下一個關鍵字的 LCA 當作新的 start 值。而當只剩最後一個關鍵字時，由於沒有下一個關鍵字可以作 LCA 運算來決定 start 值，因此執行 L23，由此棵樹的根節點 SLCA 當作最後一個 start 值。

演算法名稱：ConstructSLCATree	
輸　　入：t, M　/* t is the SLCA, M is the sorting array of those matches under t */	
輸　　出：SLCATree	
L01	i ← M.size -1
L02	start ← t
L03	**while** i ≥ 0　/*reverse preorder*/
L04	**for** each node n on the path from M[i] to start do /*從 memory pool 中取出節點*/
L05	**if** n is the leaf node of this path
L06	**if** (n matches clause[j])
L07	then set the j^{th} bit of n.clauseMatch to 1
L08	**if** (n matches keyword[k])
L09	then set the k^{th} bit of n.keyMatch to 1
L10	**else**　/*n is internal node on this path*/
L11	nc ← n.child on this path
L12	n.clauseMatch ← n.clauseMatch OR nc.clauseMatch
L13	n.keyMatch ← n.keyMatch OR nc.keyMatch
L14	**end if**　/*L05*/
L15	**if** (IsAncestor(n, M[i-1]) == false)　/*n 的小孩已經都送完 bit*/
L16	**for** (all child of n)　/*開始紀錄 n 所有小孩的資訊*/
L17	n.ClauseKeySet[DecimalClause].cbit ← true
L18	n.ClauseKeySet[DecimalClause].KeyMatchSet[DecimalKey] ←true
L19	**end for**　/*L16*/
L20	**end if**　/*L15*/
L21	**end for**　/*L04*/
L22	i ← i − 1

```
L23     if (i == 0)
L24         start ← t
L25     else
L26         start ← LCA(M[i], M[i-1])
L27   end while   /*L03*/
```

圖 4.3　ConstructSLCATree 演算法

在此說明與 MaxMatch 不同的地方。MaxMatch 中每個節點都必須去與查詢句的關鍵字做比對，也就是 L06 的動作，在此我們作改善，將路徑中的節點分為 leaf node 和 internal node，也就是 L05 和 L10 的判斷，因為每條路徑都是從關鍵字節點往上造訪，因此只需要將這條路徑的關鍵字節點，也就是 leaf node，與查詢句做比對，並紀錄 bit 值即可。此外，我們除了紀錄節點的關鍵字 bit 值，也額外記錄了項次的 bit 值。另外，MaxMatch 在紀錄小孩資訊時，每個節點走訪時皆會做此動作，也就是每個節點皆會執行 L17。我們認為，這樣會造成每個節點不只紀錄了小孩的資訊，連子孫的資訊皆紀錄進去，這樣在決定 AO_contributor 時，每個節點在與兄弟節點比較時，有可能與此兄弟節點的子孫也做了無謂的比較動作。因此我們加了 L15 的條件，確定此節點小孩都走訪完了，才開始紀錄小孩資訊。

[範例 4.3]：延續範例 4.1，我們將 group[1] = { t = 1.1.3, M = { 1.1.3, 1.1.3.1.1, 1.1.3.1.2.1, 1.1.3.1.4.1, 1.1.3.2.1, 1.1.3.2.4.1, 1.1.3.3.1, 1.1.3.3.2.1, 1.1.3.4.1, 1.1.3.4.2.1, 1.1.3.4.4.1, 1.1.3.5.1, 1.1.3.5.2.1 } }送進圖 4.2 AO_MaxMatch 演算法的 L08，進入圖 4.3 ConstuctSLCATree 演算法中。在 L01 得到 i 的值為 M 的 size = 13，L02 初始值 start 為 t = 1.1.3，在 L04 處理的路徑為 1.1.3.5.2.1 到 1.1.3 所有的節點，

也就是 center (1.1.3.5.2.1)、position (1.1.3.5.2)、player (1.1.3.5) 和 players (1.1.3)。接著在 L05~L09 中，對應查詢句，紀錄關鍵字節點 center (1.1.3.5.2.1)擁有的 clause 和 keyword 的 bit 值，分別為 0010 以及 00010，此紀錄 bit 的方式為對應查尋句的項次與關鍵字的相對位置上，紀錄為 1，與範例 4.2 相同。而節點 1.1.3.5.2 至 1.1.3 為 internal node，則執行 L11~L13，由自己小孩的 bit 值往上送，並和自己的 bit 值作 or 運算，由此可得到自己與其小孩的 bit 值。因此 1.1.3.5.2 至 1.1.3 中的所有節點 clause 和 keyword 的 bit 值分別為 0010 以及 00010。在 L15~L18，因為 position (1.1.3.5.2)不為下一個關鍵字 name (1.1.3.5.1)的祖先，表示此節點不會再有小孩需要造訪，因此，開始將此節點的小孩的 bit 值轉成十進位記錄在自己 ClauseKeySet 上，而節點 player (1.1.3.5)為下一個關鍵字 name (1.1.3.5.1)的祖先，所以暫不先紀錄 player (1.1.3.5)的資訊，直到最後一個小孩 name (1.1.3.5.1)走訪完，才將 player (1.1.3.5)的小孩 bit 值轉十進位記錄在自己 ClauseKeySet 上。

最後決定下一個 start 值，在 L26 中由下一個關鍵字 name (1.1.3.5.1)與下下一個關鍵字 USA(1.1.3.4.4.1)得到新的 start 值為 players (1.1.3)，因此下一條走訪的路徑為 name (1.1.3.5.1)至 players (1.1.3)的所有節點。當所有節點送完 bit 後如圖 4.4(b) 所示，而圖 4.4(a) 為所有節點走訪完後，players (1.1.3)底下五位小孩 player 的 ClauseKeySet 的資訊。

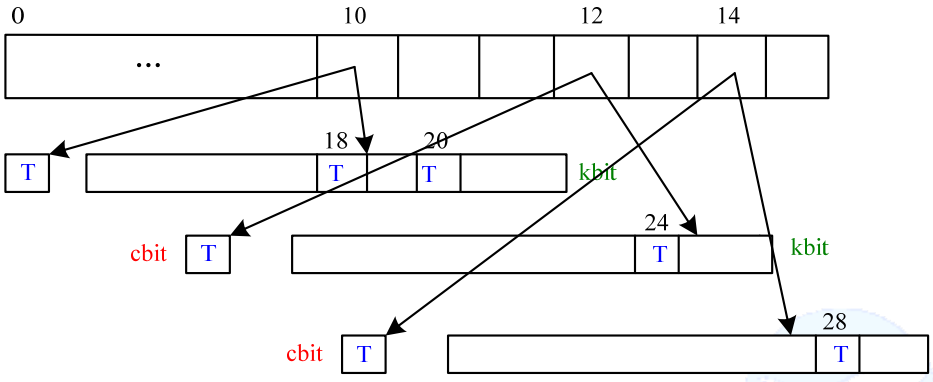

圖 4.4(a)　players (1.1.3)的 ClauseKeySet

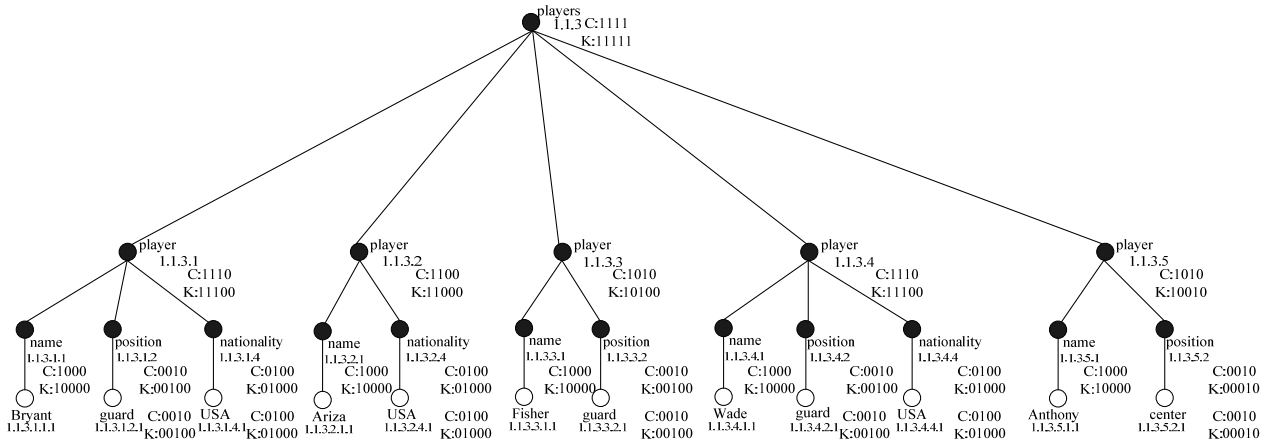

圖 4.4(b)　ConstructSLCATree 範例，其中 C 表示 clauseMatch 值，k 表示關鍵字 keyMatch 值

4.4　PruneAndOutput 模組

結束了 ConstructSLCATree 模組後，回到圖 4.2 AO_MaxMatch 演算法 L09，將 SLCA 節點送至 PruneAndOutput 模組中，透過 AO_contributor 的篩選來輸出節點。在圖 4.5 的 PruneAndOutput 演算法中，因為每棵子樹根節點皆為 SLCA，因此初始的節點 n 為 SLCA。接著造訪 n 的小孩，在 L01 將 n 當作 np，L02 中 n 的小孩為 n，將 np 與 n 送進 L03 演算法 Checking_AO_contributor 中去判斷 n 是否符合 AO_Contributor 四種 case 中該輸出的節點，若不符合輸出條件，則表示 n 底下的所有節點皆不為 AO_Contributor，此時結束對 n 底下節點的造訪。若 n 為 AO_Contributor，則輸出節點 n 的 tag name，並往下執行 L07，對 n 的小孩作演算法 PruneAndOutput 的遞迴運算。

演算法名稱：PruneAndOutput	
輸　　入：n　/*initial n is SLCA*/	
輸　　出：tag's name of n	
L01	np ← n
L02	n ← n.child
L03	**if** (n == NULL || Checking_AO_contributor (np, n) == false)
L04	return
L05	**else**
L06	output n
L07	**if** (n ≠ NULL)
L08	**for** each child nc of n do
L09	PruneAndOutput(n)

圖 4.5　PruneAndOutput 演算法

在 Checking_AO_contributor，首先在 L01 和 L02 將自己紀錄在父親節點的位置找出來，也就是將自己的 clause 和 keyword 所佔的 bit 值轉成十進位。接著在 L03 中，在自己的父親從自己 clause 的位置開始往後，若找到 ClauseKeySet[j].cbit 為 true，表示有自己的兄弟節點有可能在 clause 所佔的集合包含了自己，此時要將自己的 clause 所佔的 bit 與 j 的二進位值做 AND 的運算，若得到的值為自己的 clause bit 值，則表示在 clause 方面是被兄弟所包含的，因此在 L05 中，將 clauseFlag 設為 false，表示項次被包含。在確定完 clause 若被某一個兄弟節點包含後，緊接著在 L06 中測試關鍵字所佔的 bit 是否也被此兄弟節點所包含，測試方法跟上述雷同，將自己的節點的關鍵字所佔的 bit 轉十位後為 k，在此兄弟節點的 KeyMatchSet 從 k+1 的位置開始比對，所比對的位置 m 若有

布林值為 true，則一樣將關鍵字 bit 與 m 的二進位值做 AND 的運算，若此節點的關鍵字也被兄弟所包含，則在 L08 中將 keyFlag 設為 false，表示關鍵字被包含，此時已可證明有兄弟包含了自己的資訊，可立刻跳出此項次的關鍵字包含的檢查的迴圈，也就是 L06 的 for 迴圈。若關鍵字不被包含，表示已經檢查了此項次包含的兄弟節點的關鍵字關係，所以將 keyFalg 設為 true。最後在 L14 中檢查 clauseFlag 和 keyFlag 的值，若皆為 false，則符合定義 3.10 中的 case 2，因此不為 AO_contributor，回傳 false，其餘為定義 3.10 的 case 1 或 case 3 或 case 4，則回傳 true。

演算法名稱：Checking_AO_contributor	
輸　　入：np, n　/* np is n's father */	
輸　　出：true or false	
L01	i ← Decimal(n.clauseMatch)
L02	k ← Decimal(n.keyMatch)
L03	**for** j ← i + 1 to $2^{clause.size} - 1$ do
L04	**if** (np.ClauseKeySet[j].cbit = true && ANDbit(i, j) = i)
L05	n.clauseFlag ← false
L06	**for** m ← k+1 to $2^{key.size} - 1$ do
L07	**if** (np.ClauseKeySet [j].KeyMatchSet [m] = true && ANDbit (k, m) = k)
L08	n.keyFlag ← false
L09	Break
L10	**else**
L11	n.keyFlag ← true
L12	**end if** /*L07 */
L13	**end for** /*L06 */

```
L14      else
L15          n.clauseFlag ← true
L16      end if    /*L04 */
L17   end for    /*L03 */
L18   if (n.clauseFlag = false && n.keyFlag = false)    /*項次與關鍵字皆被包含*/
L19      return false
L20   else
L21      return true
```

圖 4.6　Checking_AO_Contributor 演算法

[範例 4.4]：承範例 4.3，由 SLCA 節點 players (1.1.3)開始造訪，首先由第一個小孩 player (1.1.3.1) 開始判斷，在 PruneAndOutput 演算法的 L03 進入 Checking_AO_Contributor 演算法。在 Checking_AO_Contributor 演算法中，發現 player (1.1.3.1) 所佔的 clause bit 為 1110，轉十進位會在 players 的 ClauseKeySet[14].Clause 設為 true，然後開始往第 15 個位置開始尋找為 true 的陣列位置，此時發現已經沒有 true 的值，則跳至 L15 判定 clauseFlag 為 true，接著不符合 L18 的條件，因此 player (1.1.3.1)為 AO_contributor。回到 PruneAndOutput 演算法的 L08，name (1.1.3.1.1)、position (1.1.3.1.2)與 nationality (1.1.3.1.4)也分別在 L09 中進入 PruneAndOutput 判斷是否輸出，步驟與上述相同，可判定 name (1.1.3.1.1)、position (1.1.3.1.2)與 nationality (1.1.3.1.4)皆為 AO_contributor，與其小孩一併輸出。

造訪完 player (1.1.3.1)底下的節點後，接著來到 player (1.1.3.2)這個節點。此節點的 clause bit 為 1100，轉十進位會在 players 的 ClauseKeySet 第 12 個位置設

為 true，然後開始往第 13 個位置開始尋找為 true 的陣列位置，會發現在第 14 個位置會有設 true 的布林值，因此將 1100 與 1110 作 AND 的運算得到 1100，表示 player (1.1.3.2)項次被兄弟包含，clauseFlag 為 false。接著檢查關鍵字部分，player (1.1.3.2) keyword bit 十進位值為 24，因此從 ClauseKeySet[14].KeyMatchSet[25] 的位置開始尋找布林值為 true 的位置，可找到在 ClauseKeySet[14].KeyMatchSet[28]的位置值為 true，因此將 24 的二進位值 11000 與 28 的二進位值 11100 作 AND 運算，可得到 11000，與自己相同，此判斷方式與範例 4.2 中相同，因此 keyFlag 為 false，表示關鍵字被包含。由於 clauseFlag 與 keyFlag 皆為 false，因此 player (1.1.3.2)不為 AO_contributor，而其底下的節點也不需要再造訪。依照上述步驟檢查 player (1.1.3.3)、player (1.1.3.4)和 player (1.1.3.5)，可發現 player (1.1.3.3)也不為 AO_contributor，此範例最後回傳結果如圖 4.7 所示。

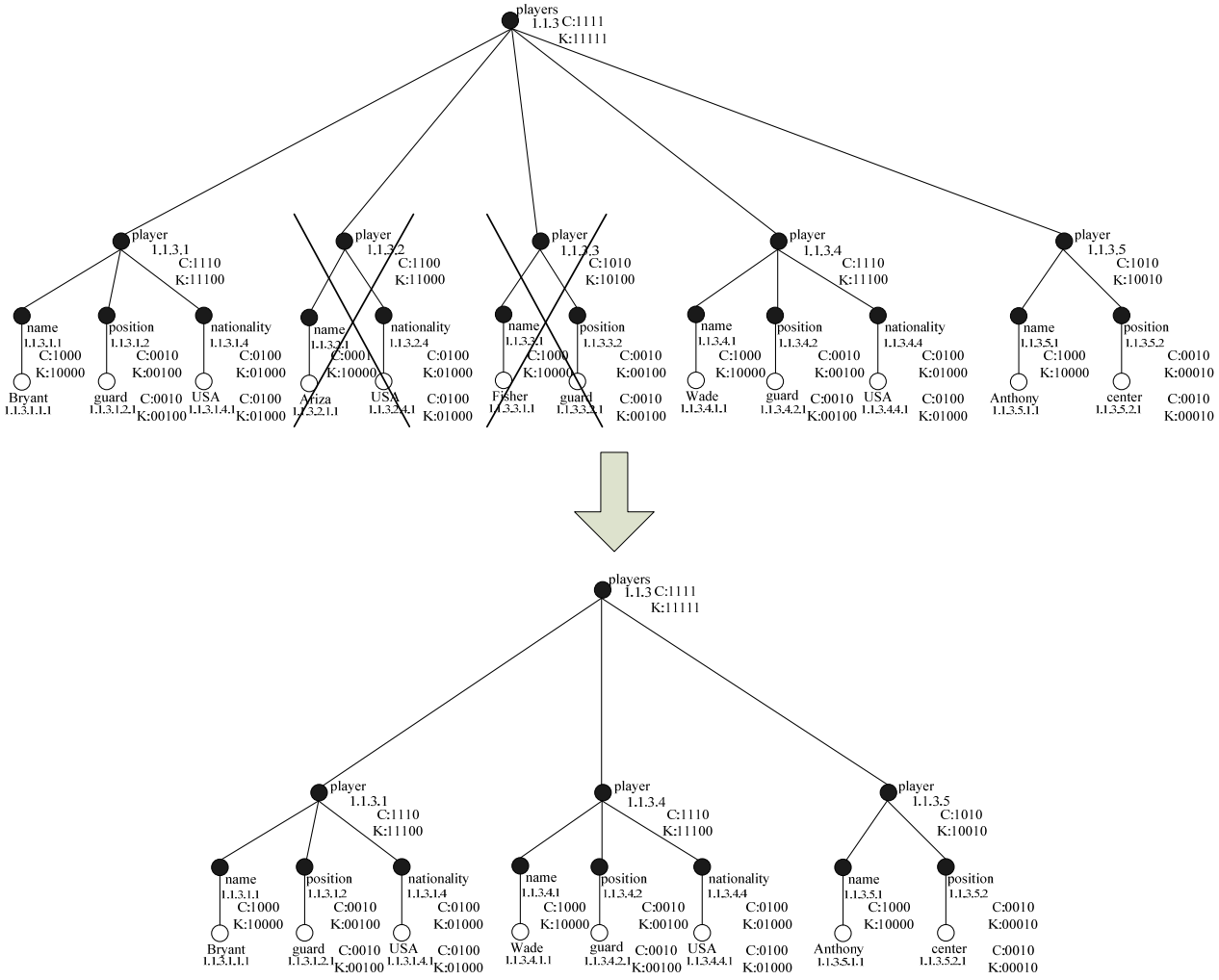

圖 4.7　Q12 最後回傳結果

4.5　AO_SingleProbe 系統架構

AO_SingleProbe 系統的架構如圖 4.8 所示。我們參考 SingleProbe[LLC20]的作法，與 AO_MaxMatch 系統不同的地方在於省略了 FindSLCA 和 GroupMatches 的動作，將所有關鍵字節點在 FindMatch 模組擷取出來後，直接在 ConstructSLCATree 模組由關鍵字節點以 reverse preorder 順序建立樹，接著在 SetSLCA 模組中找到 SLCA 節點，最後將 SLCA 節點依序送至 PruneAndOutput 模組中，根據 AO_contributor 作篩選並輸出最後的結果。

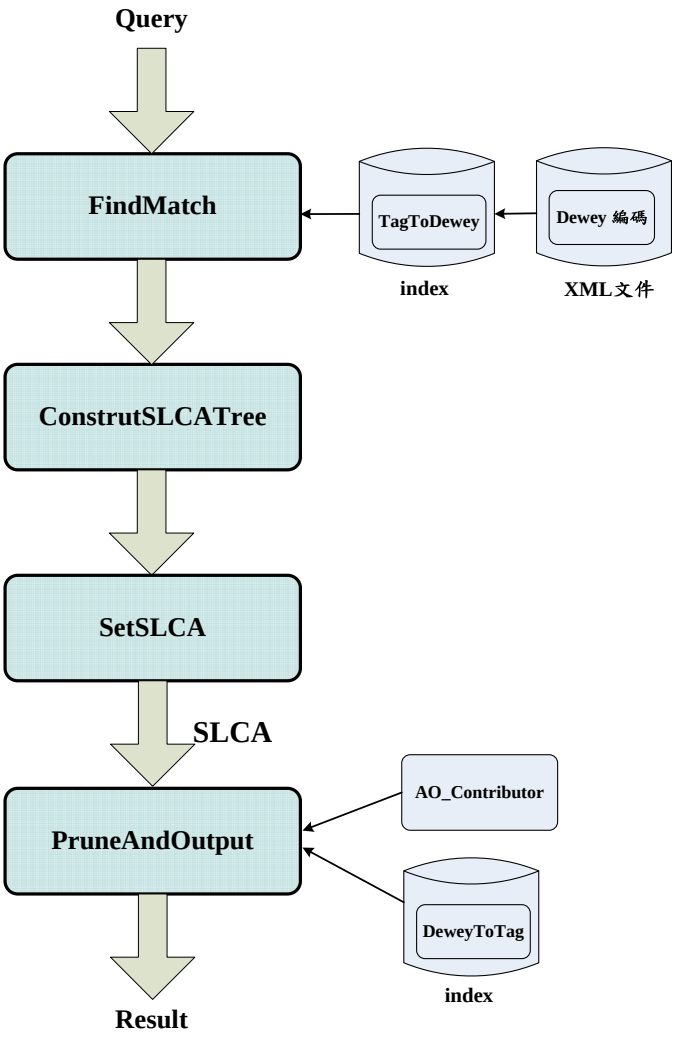

圖 4.8　AO_SingleProbe 系統架構

在圖 4.9 AO_SingleProbe 演算法，我們在 L03 直接將關鍵字節點集合 KwMatch 送進 ConstructSLCATree 中。建好樹後，我們將此樹的根節點，也就是原始文件 XML 樹的根節點，送進 L04 的 SetSLCA 中。在圖 4.10 SetSLCA 演算法中，我們從根節點開始造訪，並檢查所有小孩來找到 SLCA 節點，且每個節點皆須造訪自己的小孩，來找到 SLCA 節點。而判斷 SLCA 節點的條件如 SingleProbe 的 Observation，其 Observation 如下：

[Observation 4.1]： 在建完 ConstructSLCATree 後，其底下節點 n 若為 SLCA 節

點，若且為若滿足下列兩個條件：

(i) n 的 clause bit = SLCAbit，變數 SLCAbit，其 bit 個數與 clause 個數相等，且 bit 值全為 1。

(ii) 對於 n 的所有小孩 nc，nc 的 clause bit ≠ SLCAbit。

在 SetSLCA 演算法中，我們首先在 L01 設定一個變數 result，其初始值為 0，用來判斷每個節點 n 是否為 SLCA，當不為 0 時，表示節點 n 有小孩違反 Observation 的條件(ii)。接著在 L02 檢查 n，若不與 SLCAbit 相同，表示其小孩皆不可能會是 SLCA 節點，因此回傳 0，若與 SLCAbit 相同，則滿足 Observation 的條件(i)，接著執行 L04。在 L04 中檢查 n 的所有小孩，並作 SetSLCA 的遞迴運算，若小孩皆不為 SLCA 節點，則小孩皆會滿足 L02 的條件，並都會回傳 0，表示 n 滿足 Observation 的條件(ii)，且 n 在 L05 中得到最後的 result 值為 0。滿足 Observation 條件(i)和(ii)，表示 n 為 SLCA，因此最後執行 L07 的 PruneAndOutput。

演算法名稱：AO_SingleProbe	
輸　　入：(k1 or k2 or ...) and...(...or kn)	
輸　　出：tag's name of n	
L01	kwMatch ← FindMatch(k1…kn)
L02	Sort(KwMatch)
L03	ConstrutSLCATree(KwMatch)
L04	SetSLCA(root)

圖 4.9　AO_SingleProbe 演算法

演算法名稱：SetSLCA	
輸　　入：n	
輸　　出：1 or 0　/*return 1 if n is a SLCA, otherwise return 0*/	
L01	result ← 0
L02	**if** (n.clauseMatch ≠ SLCAbit)
L03	return 0
L04	**for** each child nc of n do
L05	result = result + SetSLCA(n)
L06	**if** (result == 0)
L07	PruneAndOutput (n)
L08	return 1

圖 4.10　SetSLCA 演算法

[範例 4.5]：仍以 Q12 查詢句為例，參考範例 4.1 得到 kwMatch = { 1.1.3, 1.1.3.1.2.1, 1.1.3.3.2.1, 1.1.3.4.2.1, 1.1.3.5.2.1, 1.1.3.1.4.1, 1.1.3.2.4.1, 1.1.3.4.4.1, 1.1.3.1.1, 1.1.3.2.1, 1.1.3.3.1, 1.1.3.4.1, 1.1.3.5.1 }。接著在 L03 中，將 KwMatch 送進 ConstrutSLCATree 演算法，建立好的樹如圖 4.11 所示。接著我們在 L04 將根節點 league (1)送進 SetSLCA 演算法中。在 SetSLCA 中，首先我們在 L02 檢查 league (1)的 clause bit 值，發現與 SLCAbit 相等。接著在 L04 進入 SetSLCA 的遞迴，檢查 league 所有的小孩，也就是 team (1.1)，發現 team(1.1)與 SLCAbit 相同，因此回傳值為 1，使得 result 值為 1，因此 league (1)不為 SLCA 節點，不執行 L07 的 PruneAndOutput。同理，team (1.1)也因為小孩 players (1.1.3)的 clause bit 值等於 SLCAbit，因此不為 SLCA 節點。最後，造訪 players (1.1.3)，發現其 clause bit 值等於 SLCAbit，且其小孩 player (1.1.3.1)、player (1.1.3.2)、player (1.1.3.3)、player

(1.1.3.4)和 player (1.1.3.5)其 clause bit 值皆不等於 SLCAbit，所以 players (1.1.3)為 SLCA 節點。接著 players (1.1.3)執行 L07 的 PruneAndOutput，最後輸出結果與圖 4.7 一致。

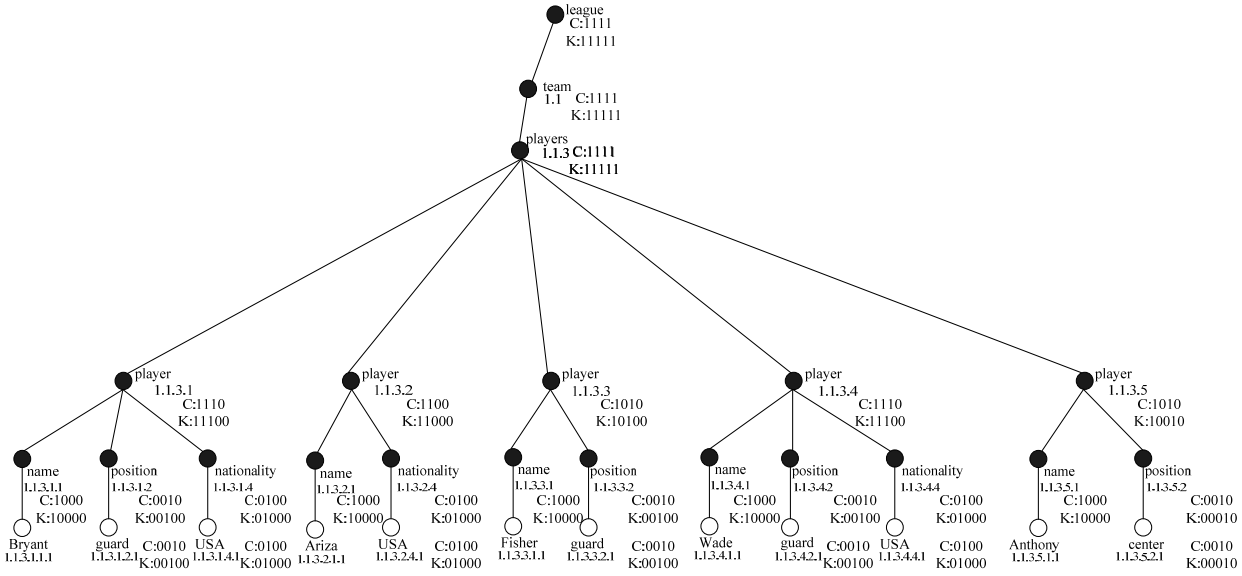

圖 4.11　AO_SingleProbe 的 ConstructSLCATree 範例

第五章 實驗

在本章中，我們將藉由回傳結果來測試 precision、recall 以及 F-measure，來評估我們所提出之系統回傳結果的準確度，並比較兩個系統，AO_MaxMatch 和 AO_SingleProbe，在何種情況下，會各有其最好的效能。首先，先說明我們所進行實驗的環境，我們以個人電腦作為實驗的環境，其 CPU 為 Core 2 Duo 6320，CPU 的每一顆核心時脈是 1.86 GHz，其記憶體為 1.5GB，而其採用的作業系統則為 Windows XP SP3，此外實作兩系統的工具，皆為 Visual C++ 2008，並且使用 Oracle Berkeley DB 建立擷取杜威碼的索引 (TagToDewey Index)和擷取元素名字和 value 的索引 (DeweyToTag Index)。

我們測試三種 data sets，分別為 Baseball、Mondial 和 Reed。Baseball 是有關於 North American baseball league 中，紀錄隊伍裡所有球員的資料，資料大小為 1.01MB。Mondial 是有關於世界地理的資料，資料大小為 2.13MB，而 Reed 則為紀錄 Reed College 內所有課程的資料，資料大小為 276KB。三種 data sets 其文件範例與 DTD 範例置於附錄 D、附錄 E 和附錄 F。我們將針對三種資料分別測試八個查詢句，另外為了測試 data monotonicity 和 data consistency 的有效性，我們將在查詢句前面註明「＊」，表示測試此查詢句時，我們將會對資料作修改，例如：QR5、QB3 和 QM2。另外，為了測試 query monotonicity 和 query consistency 的有效性，我們將會設計相似的查詢句，而相似查詢句之間的差別在於多新增一個關鍵字，例如：QR5 和 QR6，QR7 和 QR8。

為了比較兩個系統在何種情況下，會各有其最好的效能，我們將會針對查詢句內的項次所 match 到的節點數，改變其高低差的比例。而作此實驗的動機，在於[XP05]也作了關鍵字頻率高低差的實驗。最後，我們將三種 data set，分別將

內容重複複製成 100MB、200MB、300MB 和 400MB 四種 data size，來測試兩個系統的 scalability。

5.1 搜尋品質之實驗

為了測試搜尋的品質，根據每個查詢句，我們需要先針對原始文件，評定哪些節點為相關節點 (relevant nodes (R))，接著我們使用 precision、recall 和 F-measure 根據我們系統回傳的節點 (System Output nodes (S))，來評量我們系統搜尋的品質。其中 precision 用來量測「系統回傳的結果中，佔有多少百分比是我們想要的」，recall 則量測「我們想要回傳的節點中，佔有多少百分比是系統所回傳的」，而 F-measure 則量測 precision 和 recall 的調和平均數，公式如下所示：

$$\text{Precision} = \frac{|R \cap S|}{|S|}$$

$$\text{Recall} = \frac{|R \cap S|}{|R|}$$

$$\text{F-measure} = \frac{(1+\alpha) \times precision \times recall}{\alpha \times precision \times recall}$$

三份文件測試的查詢句，其 precision 和 recall 分別如表 5.1、表 5.2 和表 5.3 所示。首先觀察表 5.1 reed 文件的結果，查詢句 QR1 想要尋找 110 或 120 或 121 這三間教室的上課時間，然而系統會將課程編號 (crse) 為 110 或 120 或 121 的上課時間也回傳，因此造成 precision 降低。查詢句 QR2 想要尋找 Th (Thursday) 或 M (Monday)的課程名稱，而對於 M-W 和 T-Th 這種天數的寫法的課程名稱，則無法回傳，同造成 recall 降低。而 QR3 想要尋找 M (Monday) 或 F (Friday)的課程名稱，然而回傳結果會將部門編號 (sect)為 F 的課程名稱也回傳，造成 precision 降低，而 M-W-F 這種天數也無法回傳，造成 recall 降低。

接著討論 Baseball 文件的結果，其查詢句 QB1~QB8 的 precision 和 recall 值皆為 100%，其原因在於文件內容規律，且沒有歧異性，因此回傳結果符合需求。

最後討論 Mondial 文件，觀察查詢句 QM6，此查詢句想要找出宗教信仰為 Buddhist 或 Bahai 的 country，然而在 Mondial 文件中，province 節點和 city 節點都有小孩為 country，這兩種 country 皆會回傳，造成 precision 降低。觀察查詢句 QM7，此查詢句想要找出種族為 Turkish 或 Gypsy 的 country，同樣的因為 province 節點和 city 節點都有小孩為 country，所以造成 precision 降低。

為了測試 data monotonicity 和 data consistency 的有效性，我們在測試 QR5 時，將原本 Reed 文件中，授課老師為 Kaplan 的節點移除。測試 QB3 時，則將 Baseball 文件中，隊伍 Tigers 底下的 Starting Pitcher 節點移除。測試 QM2 時，則將 Mondial 文件中，宗教信仰為 Jewish 的節點移除，以上測試的回傳結果，precision 與 recall 值皆為 100%。測試 query monotonicity 和 query consistency 的有效性，我們設計了相似的查詢句，如 QR5 和 QR6，QR7 和 QR8，QB1 和 QB2，QB5 和 QB6，QM3 和 QM4，其回傳結果的個數，與 query monotonicity 和 query consistency 性質相符合，其 precision 值與 recall 值如表中所示。

最後我們將這 24 個查詢句各自的 precision 值和 recall 值，分別使用 α 係數 0.5、1 和 2 去計算 F-measure，最後取 24 個查詢句 F-measure 的平均值，如圖 5.1 所示。

QR1：(110 ∨ 120 ∨ 121) ∧ start_time　尋找 110 或 120 或 121 教室開始上課時間				
Relevant nodes (R)	System Output nodes (S)	R ∩ S	precision	recall

434	746	434	58%	100%
QR2：(Th ∨ M) ∧ title　尋找星期四或星期一的開課名稱				
2355	420	420	100%	17%
QR3：(M ∨ F) ∧ title 尋找星期一或星期五的開課名稱				
1578	973	223	23%	14%
QR4：08:00 ∧ (subj ∨ title)　尋找早上八點的上課的學科類別或開課名稱				
80	80	80	100%	100%
＊ QR5：(Geselbracht ∨ Yezerinac ∨ Kaplan) ∧ title 尋找 Geselbracht 或 Yezerinac 或 Kaplan 教授的開課名稱				
145	145	145	100%	100%
QR6：(Geselbracht ∨ Yezerinac ∨ Kaplan ∨ Drumm) ∧ title 尋找 Geselbracht 或 Yezerinac 或 Kaplan 或 Drumm 教授的開課名稱				
165	165	165	100%	100%
QR7：(ANTH ∨ BIOL) ∧ (reg_num ∨ crse) 尋找人類學類或生物學類的註冊號碼或課程編號				
658	658	658	100%	100%
QR8：(ANTH ∨ BIOL ∨ CHIN) ∧ (reg_num ∨ crse) 尋找人類學類或生物學類或中文學類的註冊號碼或課程編號				

714	714	714	100%	100%

表 5.1　reed 文件的 Precision 和 Recall

QB1：Jim ∧ (Outfield ∨ Relief Pitcher) 尋找人名為 Jim，且為外野手或救援投手				
Relevant nodes (R)	System Output nodes (S)	R ∩ S	precision	recall
293	293	293	100%	100%
QB2：Jim ∧ (Outfield ∨ Relief Pitcher ∨ Starting Pitcher) 尋找人名為 Jim，且為外野手或救援投手或先發投手				
225	225	225	100%	100%
＊ QB3：Tigers ∧ WINS ∧ (Starting Pitcher ∨ Relief Pitcher) 尋找老虎隊中，先發投手或救援投手的勝場數				
98	98	98	100%	100%
QB4：Angels ∧ (Relief Pitcher ∨ Third Base) ∧ GIVEN_NAME 尋找天使隊中，救援投手或三壘手的名字				
68	68	68	100%	100%
QB5：White Sox ∧ (First Base ∨ Outfield) ∧ HOME_RUNS 尋找白襪隊中，一壘手或外野手的全壘打數				
48	48	48	100%	100%
QB6：White Sox ∧ (First Base ∨ Outfield) ∧ (HOME_RUNS ∨ DOUBLES)				

尋找白襪隊中，一壘手或外野手的全壘打數或雙殺次數				
63	63	63	100%	100%
QB7：TEAM ∧ (Jason ∨ Tim) ∧ GAMES 所有隊伍中，名叫 Jason 或 Tim 的比賽數				
150	150	150	100%	100%
QB8：TEAM ∧ (Jason ∨ Tim) ∧ (GAMES ∨ POSITION) 所有隊伍中，名叫 Jason 或 Tim 的比賽數或守備位置				
204	204	204	100%	100%

表 5.2　Baseball 文件的 Precision 和 Recall

QM1：(Chinese ∨ Indian) ∧ gdp_ind 尋找中國人或印度人的個人生產毛額				
Relevant nodes (R)	System Output nodes (S)	R ∩ S	precision	recall
148	148	148	100%	100%
＊ QM2：(Jewish ∨ Roman Catholic) ∧ population 尋找猶太教或羅馬天主教的人口數				
6613	6613	6613	100%	100%
QM3：Birmingham ∧ United Kingdom ∧ longitude ∧ latitude 尋找英國的伯明罕市的經度和緯度				
11	11	11	100%	100%

QM4：Birmingham ∧ (United Kingdom ∨ United States) ∧ longitude ∧ latitude 尋找英國或美國的伯明罕市的經度和緯度				
22	22	22	100%	100%
QM5：(Tainan ∨ Hefei) ∧ longitude ∧ latitude 尋找台南或合淝的經度和緯度				
14	14	14	100%	100%
QM6：country ∧ (Buddhist ∨ Bahai) 尋找信奉佛教或巴哈派教的國家				
429	1314	429	32%	100%
QM7：country ∧ (Turkish ∨ Gypsy) 尋找土耳其人或吉普賽人的國家				
358	1006	358	35%	100%
QM8：population_growth ∧ (French ∨ Spanish) 尋找法國或西班牙的人口成長數				
119	119	119	100%	100%

表 5.3　Mondial 文件的 Precision 和 Recall

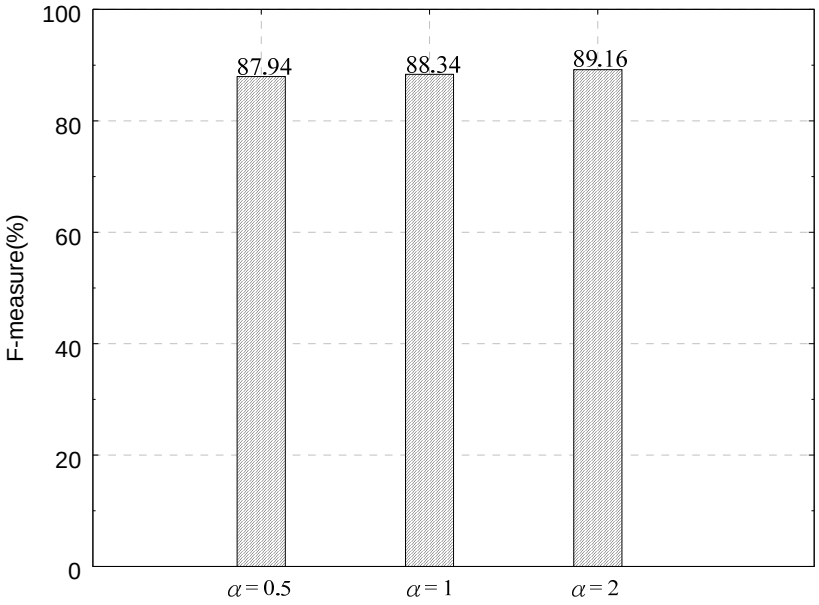

圖 5.1　F-measure 的實驗柱狀圖

最後，我們與[XP05]中的 XKSearch 的輸出結果做比較，稱作 X 系統，X 系統的輸出方式是以 SLCA 為根節點，輸出至底下的所有關鍵字連接而成的路徑。而我們的 AO_MaxMatch 稱作 A 系統，比較的查詢句皆為以 And 方式連接關鍵字，分別針對三種 data set 列出對應的查詢句。比較結果如表 5.4 所示，可以發現 QA1、QA2 和 QA3 在 X 系統回傳結果過多，造成 precision 降低，而 QA6，兩個系統皆因 T-Th 這種天數無法回傳，造成 recall 降低。比較的結果，證明我們的系統輸出結果比較好。

QA1：Jim ∧ Abbott ∧ Outfield 尋找名字為 JIM，小名為 Abbott 的外野手					
Relevant nodes (R)	系統	System Output nodes (S)	R ∩ S	precision	recall
82	A	82	82	100%	100%
	X	455	82	18%	100%

QA2：Tigers ∧ Strarting Pitcher ∧ SURNAME 尋找老虎隊中，先發投手的小名					
37	A	37	37	100%	100%
	X	103	37	35%	100%
QA3：Yankees ∧ Relief Pitcher ∧ GAMES 尋找洋基隊中，救援投手的比賽數					
54	A	54	54	100%	100%
	X	75	54	72%	100%
QA4：White Sox ∧ Outfield ∧ SURNAME 尋找白襪隊中，外野手的小名					
37	A	37	37	100%	100%
	X	115	37	32%	100%
QA5：ANTH ∧ Start_time ∧ days 尋找分類為 ANTH 課程的上課時間與星期幾					
138	A	138	138	100%	100%
	X	138	138	100%	100%
QA6：Th ∧ subj 尋找星期四的上課主題					
1044	A	204	204	100%	19%
	X	204	204	100%	19%
QA7：Muslim ∧ Roman Catholic ∧ gdp_total 尋找國家中信奉回教和天主教國人的總生產毛額					
126	A	126	126	100%	100%

	X	126	126	100%	100%
QA8：Muslim ∧ Chinese ∧ population 尋找信奉回教且有中國人的國家的人口數					
242	A	242	242	100%	100%
	X	242	242	100%	100%

表 5.4　AO_MaxMatch 與 XKSearch 的 precision 與 recall 比較

5.2　項次內的關鍵字頻率比之實驗

在這一節中，我們針對查詢句每個項次內所 match 到的節點數，其最高與最低的比例，測試在何種條件下，AO_MaxMatch 和 AO_SingleProbe 會各有其表現較好的狀況。測試的資料為 Baseball 文件，我們將其內容重複複製成 100MB，欲測試的查詢句如表 5.4 所示，測試的結果如圖 5.2 所示。

由測試的結果我們可以發現，隨著高低頻比率落差越大，約在 1:100 時，AO_SingleProbe 的執行時間開始多於 AO_MaxMatch 的執行時間。因為 SLCA 的個數不會多於最低頻率的項次所擁有的節點數，因此當高低頻落差漸漸增加時，表示會有更多屬於高頻率項次的關鍵字節點，不在 SLCA 底下，但 AO_SingleProbe 的演算法仍需要去走訪這些不在 SLCA 底下的關鍵字節點。相反的，此時 AO_MaxMatch 卻只需要走訪已經計算好的 SLCA 底下的關鍵字節點即可，因此時間將少於 AO_SingleProbe。相反的，當高低頻越接近時，表示 SLCA 底下涵蓋高頻率項次內的關鍵字節點越多，而造訪不在 SLCA 底下的關鍵字節點所花的時間就相對變少，此時 AO_SingleProbe 在走訪節點時，也等同計算並得

到了 SLCA，而且少了 GroupMatch 的時間，因此 AO_SingleProbe 時間將少於 AO_MaxMatch 的時間。

查詢句	項次內關鍵字高低頻比
Q1：GAMES ∧ Watson	1：1000
Q2：GAMES ∧ (Watson ∨ Cradle)	1：500
Q3：POSITION ∧ (Aaron ∨ Luis)	1：100
Q4：Catcher ∧ (Charlie ∨ James ∨ Reggie)	1：10
Q5：TEAM_CITY ∧ (Kevin ∨ Jason)	1：1

表 5.5　各種高低頻率比之查詢句

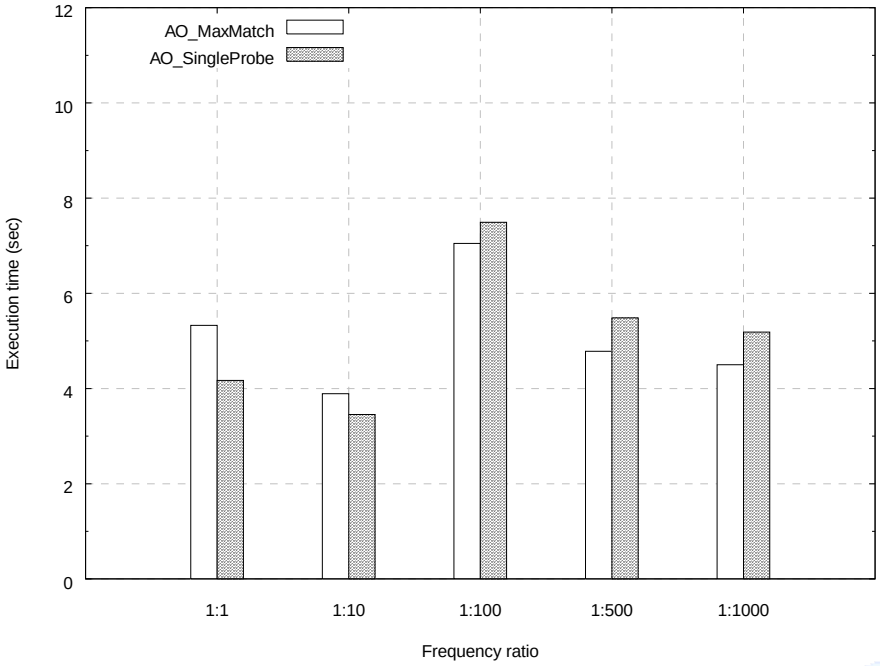

圖 5.2　高低頻率比的實驗柱狀圖

5.3 Scalability 時間之實驗

在這一節中，我們將測試 AO_MaxMatch 與 AO_SingleProbe 兩個演算法的 scalability 時間的實驗，我們將 BaseBall 文件內容重複複製成 100MB、200MB、300MB 和 400MB 作為測試資料，接著分別使用表 5.4 的 Q2 (1:500) 和 Q3 (1:100) 作為測試的查詢句，其結果分別為圖 5.3 和 5.4。從兩張圖可發現，AO_MaxMatch 以及 AO_SingleProbe 隨著資料的增加，其時間皆為線性成長。

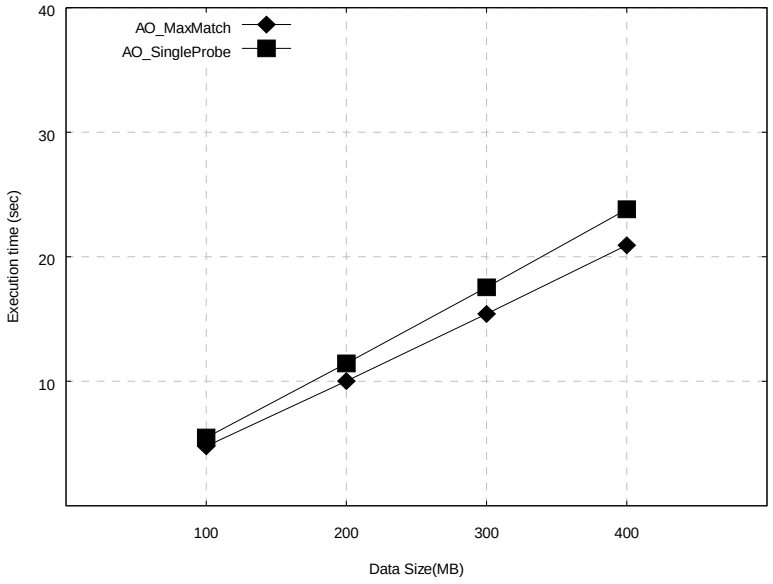

圖 5.3　查詢句 GAMES ∧ (Watson ∨ Cradle) (1:500)

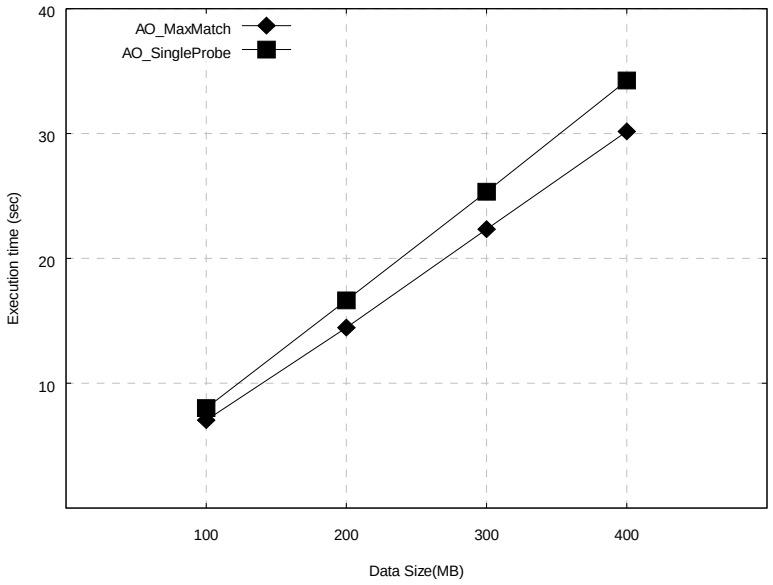

圖 5.4　POSITION ∧ (Aaron ∨ Luis) (1:100)

第六章 結論與未來方向

在本論文中，首先提出 AO_MaxMatch 方法，該方法擴充 MaxMatch 的做法，進一步支援處理以 “AND” 和 “OR” 的型式結合的查詢句，另外提出 AO_SingleProbe 方法，此方法參考 SingleProbe 的做法，一樣支援以“AND” 和 “OR”的型式結合的查詢句，並捨棄了[XP05]提出的找 SLCA 的方法，在建完樹後，尋找 SLCA 並直接輸出。另外我們將 MaxMatch 提出的四個性質：data monotonicity、data consistency、query monotonicity 和 query consistency，予以修改加擴充，並制定 AO_contributor，使得 AND-OR 查詢句的輸出結果更加合理。最後實驗結果，證明在搜尋品質上的有效性，以及 AO_MaxMatch 和 AO_SingleProbe 在何種情況下，執行時間會各有其最好的表現。

本論文未來的研究方向，可以針對回傳的查詢結果再加上 Ranking 的機制，或者支援更複雜的關鍵字結合方式的演算法，如：AND-OR-NOT。

參考文獻

[ACD06] Sihem Amer-Yahia, Emiran Curtmola, Alin Deutsch, "Flexible and Efficient XML Search with Complex Full-Text Predicates", In Proceeding of the SIGMOD Conference, Chicago, Illinois, USA, 2006.

[AYJ03] Shurug Al-Khalifa, Cong Yu, H. V. Jagadish, "Querying Structured Text in an XML Database", In Proceedings of the SIGMOD Conference, Jun. 2003.

[AKMD+05] Sihem Amer-Yahia, Nick Koudas, Amelie Marian, Divesh Srivastava, David Toman, "Structure and Content Scoring for XML", In Proceedings of the VLDB Conference, Pages: 361–372, Trondheim, Norway, 2005.

[GSBS03] Lin Guo, Feng Shao, Chavdar Botev, Jayavel Shanmugasundaram, "XRANK: Ranked Keyword Search over XML Documents", In Proceedings of the SIGMOD Conference, San Diego, CA, June 9-12, 2003.

[KKNR04] Raghav Kaushik, Rajasekar Krishnamurthy, Jeffrey F. Naughton, "Raghu Ramakrishnan: On the Integration of Structure Indexes and Inverted Lists", In Proceedings of the ICDE Conference, 2004.

[LC07] Ziyang Liu, Yi Chen, "Identifying Meaningful Return Information for

XML Keyword Search", In Proceedings of the SIGMOD Conference, 2007.

[LC08] Ziyang Liu, Yi Chen, "Reasoning and Identifying Relevant Matches for XML Keyword Search", In Proceedings of the VLDB Conference, 2008.

[LCC10] Rung-Ren Lin, Ya-Hui Chang, Kun-Mao Chao, "Efficient Algorithm for Searching Relevant Matches in XML DataBases", Submitted to WWW 2010.

[LOF+08] Guoliang Li, Beng Chin Ooi, Jianhua Feng, Jianyong Wang, Lizhu Zhou, "EASE: An Effective 3-in-1 Keyword Search Method for Unstructured, Semi-structured and Structured Data", In Proceedings of the SIGMOD Conference, 2008.

[SM08] Taro L. Saito, Shinichi Morishita, “Relational-Style XML Query”, In Proceedings of the SIGMOD conference, June 9–12, 2008, Vancouver, BC, Canada.

[TJM+08] Partha Pratim Talukdar, Marie Jacob, Muhammad Salman Mehmood, Koby Crammer, Zachary G. Ives, Fernando Pereira, Sudipto Guha, "Learning to Create Data Integrating Queries", In Proceeding of the VLDB Conference, 2008.

[TSW05] Martin Theobald, Ralf Schenkel, Gerhard Weikum, "An Efficient and Versatile Query Engine for TopX Search", In Proceedings of the VLDB Conference, 2005.

[XP05] Yu Xu, Yannis Papakonstantinou, "Efficient Keyword Search for Smallest LCAs in XML Databases", In Proceedings of the SIGMOD Conference, 2005.

[YLP04] Sihem AmerYahia, Laks V.S. Lakshmanan, Shashank Pandit, "FleXPath: Flexible Structure and FullText Querying for XML", In Proceedings of the SIGMOD Conference, Pages: 13-18, Paris, France, 2004.

[ZC08] Lei Zou, Lei Chen, "Dominant Graph: An Efficient Indexing Structure to Answer Top-K Queries", In Proceedings of the VLDB Conference, 2008.

附錄 A： ParsingXMLtoDewey 演算法

演算法名稱：ParsingXMLtoDewey

輸　　入：XML 文件

輸　　出：Dewey id

```
L01   ReadCh(ch)
L02   while(1)
L03   {
L04      if(ch=='<')
L05      {
L06            in>>ch;
L07            if(ch=='!' || ch=='?')
L08            { //如果是'!'或'?'，則一直讀字元直到'>'出現，並忽略這一行
L09                  while(ch!='>') in>>ch;
L10                  in>>ch; continue;
L11            }
L12            ch2 = ch;
L13            for(i=0; ch!='>'; ReadCh(ch))
L14                  buf[i++] = ch;   //一直讀字元直到'>'出現，並把<...>之間的字串存在 buf
L15            buf[i] = 0;
L16            if(ch2=='/') Pop(buf+1);   //如果 buf[0]是'/'，表示是結尾 tag，此時從 stack 取出
L17 一物件
L18            else
L19            {//底下的 for 主要是要取<...>裡邊的第一個字串，因為 tag 有可能長這樣<Lion
L20 Id=15>，此時我們只取"Lion"
```

```
L21                 for(i=1;buf[i];)
L22                 {
L23                     if(buf[i]==' ') buf[i] = 0;
L24                     else i++;
L25                 }
L26                 Push(buf, flag+1); //把字串 push 到 stack，並做編碼，當下一次與他對應的
L27 結尾 tag 出現時就會再被 pop
L28             }   //end else L18
L29             in>>ch;
L30         } //end if L04
L31         else
L32         { //會進到這裡表示不是 tag 而是 content，例如<name>John</name>的"John"
L33             for(i=0;ch!='<';ReadCh(ch)) //把 content 一直讀進來
L34                 buf[i++] = ch;
L35             buf[i] = 0;
L36             Push(buf, flag); //push 到 stack，並做編碼
L37         } //end else L31
L38     } //end while L02
```

演算法名稱：Push

輸　　入：tag or value

輸　　出：Dewey id

變數說明：str /*tag or value*/

StackItem /*儲存 tag or value 的 stack*/

maxbf /*記錄每一層 Dewey 的最大值，就是最大寬度*/

Xelement /*儲存此元素的所有 dewey id，為 link list*/

XCode /*儲存一個杜威碼的所有數字部份*/

```
L01   StackItem *item = new StackItem(str);
L02   if(cur==0) maxbf[0] = item->dcode = 1
L03   else {   //item->dcode 表示此節點(假設 n)在這一層的 dewey number 而[cur-1]->child 表
L04 示 n 的 parent 最靠近自己的 child(也可說是 n 的左邊一個 sibling) 所以把它加 1 就是 n
L05 在這一層的 dewey number*/
L06       item->dcode = ++s[cur-1]->child; //StackItem 中的 dcode 是指自己 n，s[cur-1]則是指
L07                          n 的父母，父母的 child 值取出來，然後加 1，就會是 n 的 dewey 碼
L08   } //end if L02
L09     if((flag&2) && maxbf[cur]<item->dcode) //此時為第一次 parsing，這兩行主要是記錄
L10                                            每一層的 dewey 碼的最大值
L11     maxbf[cur] = item->dcode;
L12   }
L13   /*(flag&2)為 false 時代表第二次 parsing，此時已知最大 dewey 碼。此外，XElement
L14 代表一個 keyword 物件，此 keyword 在 XML tree 可能出現多次，所以用 linked list 來串
L15 起來，並以 head 記錄第一個，tail 記錄目前最後一個*/
L16   if((flag&2)==0) {
L17     XElement *e = hash.AddEle(str);   //利用 keyword 取得一個 hash 空間
```

```
L18     if(e->head==NULL) e->head = e->tail = new XCode(blen); //此行代表第一次遇到此
L19                                                   keyword，所以 head=NULL
L20     else { //else 代表此 keyword 第二次以上的出現
L21        e->tail->next = new XCode(blen);   //配置一 XCode 物件，這一次的節點的 Dewey
L22                                              碼就是準備要放這裡
L23     e->tail = e->tail->next;
L24     DCodeString(d, bits, e->tail->code);//d 陣列是 int 陣列，因為已知各層最大 dewey 碼，
L25   所以利用 DCodeString 重新編碼 (可參考[XP05]第四章) 並儲存在剛配置的 XCode
L26   物件(參考程式部分 DCodeString 函式)
L27  } //end if L16
```

附錄 B： CreateTagtoDeweyIndex 演算法

演算法名稱：CreateTagtoDeweyIndex

輸　　入：tag and dewey id

輸　　出：TagtoDewey index

變數說明：Xelement /*儲存此元素的所有 dewey id，為 link list*/

table /*為陣列，每一格儲存元素名字，和 XElement*/

MYPRIME /*table 的大小*/

blen /*min bytes for all bits[]/*

XCode /*儲存一個杜威碼的所有數字部份*/

```
L01   XElement *cur;
L02   for(i=0;i<MYPRIME;i++) // table 總共有 MYPRIME 個元素，針對每一個元素做底下的
L03 事
L04   {
L05      for(cur= table.ele[i];cur;cur=cur->next) //針對被指定的元素串列
L06      {
L07         length = cur->size*blen; //cur->size 為此 keyword 的 frequency，乘以 blen 後表示要
L08                                  儲存這個 keyword 所需的大小
L09         p = arr = new unsigned char[length]; //配置記憶體
L10         for(XCode *dc=cur->head;dc;dc=dc->next,p+=blen)
L11            memcpy(p, dc->code, blen); //把此 keyword 的所有節點的 dewey number 都複製
L12                                       到剛剛配置的記憶體
L13         Dbt key(cur->str, strlen(cur->str)+1); //設定寫入 index 的 key，也就是 tag name，為
L14                                                Berkeley DB 的 API
L15         Dbt data(arr, length); //設定寫入 index 的 value，也就是 dewey id，為 Berkeley DB
```

L16	的 API
L17	ret = db.put(0, &key, &data, DB_NOOVERWRITE); //寫入 index，為 Berkeley DB
L18	的 API
L19	}//end for L05
L20	}//end for L02

附錄 C： CreateDeweytoTagIndex 演算法

演算法名稱：CreateDeweytoTagIndex

輸　　入：tag name and dewey id

輸　　出：DeweytoTagIndex index

變數說明：Xelement /*儲存此元素的所有 dewey id，為 link list*/

table /*為陣列，每一格儲存元素名字，和 XElement*/

MYPRIME /*table 的大小*/

blen /*min bytes for all bits[]/*

XCode /*儲存一個杜威碼的所有數字部份*/

```
L01   XElement *cur;
L02   for(i=0;i<MYPRIME;i++) // table 總共有 MYPRIME 個元素，針對每一個元素做底下的
L03 事
L04   {
L05      for(cur= table.ele[i];cur;cur=cur->next) //針對被指定的元素串列
L06      {
L07         Dbt data(cur->str, strlen(cur->str)+1); //設定寫入 index 的 value，也就是 tag name，
L08                                                  為 Berkeley DB 的 API
L09         for(XCode *dc=cur->head;dc;dc=dc->next,p+=blen)
L10         {
L11            Dbt key(dc->code, blen); //設定寫入 index 的 key，也就是 dewey id，為 Berkeley
L12                                     DB 的 API
L13            ret = db.put(0, &key, &data, DB_NOOVERWRITE); //寫入 index，為 Berkeley DB
L14                                                          的 API
L15         }//end for L08
```

L16	}//end for L05
L17	}//end for L02

附錄 D：Baseball 文件範例與 DTD 部分定義

來源：

http://www.cafeconleche.org/books/biblegold/examples/baseball/index.html

文件範例：

```
<TEAM>
        <TEAM_CITY>Chicago</TEAM_CITY>
        <TEAM_NAME>White Sox</TEAM_NAME>
        <PLAYER>
          <GIVEN_NAME>Jim</GIVEN_NAME>
          <SURNAME>Abbott</SURNAME>
          <POSITION>Starting Pitcher</POSITION>
          <GAMES>5</GAMES>
          <GAMES_STARTED>5</GAMES_STARTED>
          <WINS>5</WINS>
          <LOSSES>0</LOSSES>
          <SAVES>0</SAVES>
          <COMPLETE_GAMES>0</COMPLETE_GAMES>
          <SHUT_OUTS>0</SHUT_OUTS>
          <ERA>4.55</ERA>
          <INNINGS>31.2</INNINGS>
          <HITS_AGAINST>35</HITS_AGAINST>
          <HOME_RUNS_AGAINST>2</HOME_RUNS_AGAINST>
          <RUNS_AGAINST>16</RUNS_AGAINST>
          <EARNED_RUNS>16</EARNED_RUNS>
```

```
                <HIT_BATTER>1</HIT_BATTER>
                <WILD_PITCHES>0</WILD_PITCHES>
                <BALK>0</BALK>
                <WALKED_BATTER>12</WALKED_BATTER>
                <STRUCK_OUT_BATTER>14</STRUCK_OUT_BATTER>
            </PLAYER>
<TEAM>
```

DTD 部分定義：

```
<!ELEMENT TEAM (TEAM_CITY, TEAM_NAME, PLAYER*)>
<!ELEMENT TEAM_CITY (#PCDATA)>
<!ELEMENT TEAM_NAME (#PCDATA)>
<!ELEMENT PLAYER (GIVEN_NAME, SURNAME, POSITION, GAMES,
HOME_RUNS, DOUBLES…)
<!ELEMENT GIVEN_NAME (#PCDATA)>
<!ELEMENT SURNAME (#PCDATA)>
<!ELEMENT POSITION (#PCDATA)>
<!ELEMENT GAMES (#PCDATA)>
<!ELEMENT HOME_RUNS (#PCDATA)>
<!ELEMENT DOUBLES (#PCDATA)>
```

附錄 E：Mondial 文件範例與 DTD 部分定義

來源：http://www.dbis.informatik.uni-goettingen.de/Mondial/#XML

文件範例：

```
<country car_code="AL" area="28750" capital="cty-cid-cia-Albania-Tirane"
                memberships="org-BSEC org-CE org-CCC org-ECE org-EBRD
org-FAO org-IAEA org-IBRD org-ICAO org-Interpol org-IDA org-IFRCS org-IFC
org-IFAD org-ILO org-IMO org-IMF org-IOC org-IOM org-ISO org-ICRM org-ITU
org-Intelsat org-IDB org-ANC org-OSCE org-OIC org-PFP org-UN org-UNESCO
org-UNIDO org-UNOMIG org-UPU org-WFTU org-WHO org-WIPO org-WMO
org-WToO org-WTrO">
    <name>Albania</name>
    <population>3249136</population>
    <population_growth>1.34</population_growth>
    <infant_mortality>49.2</infant_mortality>
    <gdp_total>4100</gdp_total>
    <gdp_agri>55</gdp_agri>
    <inflation>16</inflation>
    <indep_date>1912-11-28</indep_date>
    <government>emerging democracy</government>
    <ethnicgroups percentage="3">Greeks</ethnicgroups>
    <ethnicgroups percentage="95">Albanian</ethnicgroups>
</country>
```

DTD 部分定義：

```
<!ELEMENT mondial (country*,continent*,organization*,
                   mountain*,(sea*,river*,lake*,desert*,island*)*)>
<!ELEMENT country (name,encompassed+,
                   ethnicgroups*,religions*,languages*,border*,
                   province*,city*)>
<!ATTLIST country car_code ID #IMPLIED
                  capital IDREF #IMPLIED
                  memberships IDREFS #IMPLIED
                  id ID #REQUIRED
                  name CDATA #REQUIRED
                  datacode CDATA #IMPLIED
                  continent CDATA #IMPLIED
                  area CDATA #IMPLIED
                  total_area CDATA #IMPLIED
                  population CDATA #IMPLIED
                  population_growth CDATA #IMPLIED
                  infant_mortality CDATA #IMPLIED
                  gdp_agri CDATA #IMPLIED
                  gdp_ind CDATA #IMPLIED
                  gdp_serv CDATA #IMPLIED
                  inflation CDATA #IMPLIED
                  gdp_total CDATA #IMPLIED
                  indep_date CDATA #IMPLIED
                  government CDATA #IMPLIED>
<!ATTLIST province id ID #REQUIRED
```

```
                name CDATA #REQUIRED
                country IDREF #REQUIRED
                capital IDREF #IMPLIED
                area CDATA #IMPLIED
                population CDATA #IMPLIED>
<!ATTLIST city id ID #REQUIRED
                is_country_cap CDATA #IMPLIED
                is_state_cap CDATA #IMPLIED
                longitude CDATA #IMPLIED
                latitude CDATA #IMPLIED
                country IDREF #REQUIRED
                province IDREF #IMPLIED>
```

附錄 F：Reed 文件範例與 DTD 部分定義

來源：

http://www.cs.washington.edu/research/xmldatasets/www/repository.html#courses

文件範例：

```
<course>
    <reg_num>10624</reg_num>
    <subj>BIOL</subj>
    <crse>431</crse>
    <sect>F01</sect>
    <title>Field Biology of Amphibians</title>
    <units>0.5</units>
    <instructor>Kaplan</instructor>
    <days>T</days>
    <time>
        <start_time>06:10PM</start_time>
        <end_time>08:00</end_time>
    </time>
    <place>
        <building>PHYSIC</building>
        <room>240A</room>
    </place>
</course>
```

DTD 部分定義：

```
<!ELEMENT root (course*)>
```

<!ELEMENT course (reg_num,subj,crse,sect,title,units,instructor,days,time,place)>

<!ELEMENT reg_num (#PCDATA)>

<!ELEMENT subj (#PCDATA)>

<!ELEMENT crse (#PCDATA)>

<!ELEMENT sect (#PCDATA)>

<!ELEMENT title (#PCDATA)>

<!ELEMENT units (#PCDATA)>

<!ELEMENT instructor (#PCDATA)>

<!ELEMENT days (#PCDATA)>

<!ELEMENT time (start_time,end_time)>

<!ELEMENT start_time (#PCDATA)>

<!ELEMENT end_time (#PCDATA)>

<!ELEMENT place (building,room)>

<!ELEMENT building (#PCDATA)>

<!ELEMENT room (#PCDATA)>