輸出 XML 關鍵字查詢有效節點之研究

The Research on Identifying Contributors for XML Keyword Search

研 究 生：簡伯先　　Student：Po-Hsien Chien

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
July 2011
Keelung, Taiwan, Republic of China

中華民國 100 年 7 月

摘要

針對 XML 文件進行關鍵字查詢時，如何以精簡的回傳結果來滿足使用者的資訊需求一直是項重要的研究課題。 [LC08]定義了以關鍵字的 SLCA 到其下所有相關對應節點之路徑上的所有節點來做為回傳結果，這樣的定義方式受到了其他研究廣泛地討論以及應用。本論文的目的，即為探討如何有效率地產生 [LC08]所定義的回傳結果。

我們提出了 TDPrune 以及 LevelPrune 此兩個系統，透過減少處理的節點數來加快查詢效率。此外，我們提出一系列的改進方法，透過資料儲存和壓縮的技術，來減少計算量以加快查詢處理所需的時間。我們根據提出的演算法實作了 TDPrune 和 LevelPrune 兩個系統，另外並以 LevelPrune 為基礎加進壓縮技術而實做出 LevelPrune+。

我們針對以上系統和 MaxMatch 進行一系列的實驗比較，來探討各系統之優缺點，並分析各系統所適用的情況。最後得知，我們提出的系統明顯地比 MaxMatch 有效率，同時，除了在少數極端的情況下，TDPrune 會較 LevelPrune+為快之外，在大部份的情況下，LevelPrune+皆有最好的執行效率。

Abstract

It is important to satisify one's information needs with compact returning results when users input keyword queries. [LC08] proposed the MaxMatch system which has been widly discussed and applied. In this thesis, we will discuss how to efficiently generate the results defined by [LC08].

Two systems, TDPrune and LevelPrune, are designed to improve the performance by reducing the nodes to process. We also propose several improvements based on storage and compression techniques to further reduce the query processing time. We then implement the two systems and LevelPrune+, which improves the LevelPrune system by applying the proposed storage and compression techniques.

Finally, we design a series of experiments to compare the three systems with MaxMatch and analyze the pros and cons of each system. The experiment results show that all our systems are more effiecient then MaxMatch, and LevelPrune+ is usually the most effiecient one.

目錄

摘要 ..... **i**

Abstract ..... **ii**

目錄 ..... **iii**

圖目錄 ..... **v**

表目錄 ..... **vii**

第一章 緒論 **1**

1.1 研究動機與目的 1

1.2 研究方法與貢獻 2

1.3 相關研究 4

1.4 論文架構 6

第二章 相關定義 **8**

2.1 XML 資料表示法 8

2.2 關鍵字查詢之輸出結果 10

2.3 MaxMatch 演算法介紹 13

第三章 TDPrune 演算法 **19**

3.1 系統架構 19

3.2 FindMatch 模組和 FindSLCA 模組 20

3.3 GroupMatchRange 模組 22

3.4 TDPruneMatches 模組 26

3.5 TDPrune 效率分析 40

第四章 LevelPrune 演算法 46

4.1 系統架構 46

4.2 LevelPruneMatches 模組 47

4.3 LevelPrune 效率分析 51

4.4 編碼行導向儲存 51

4.5 kwMatch 壓縮 53

4.6 反轉索引壓縮 56

第五章 實驗 58

5.1 各式查詢之實驗 59

5.2 查詢句關鍵字數量之實驗 63

5.3 對應節點數量之實驗 66

5.4 輸出節點數量之實驗 69

5.5 貢獻比率之實驗 70

5.6 同層節點數量之實驗 73

第六章 結論及未來方向 75

參考文獻 76

附錄 A： groupMatches 演算法 **78**

附錄 B： pruneMatches 演算法 **79**

圖目錄

圖 2.1 範例 XML 樹 8

圖 2.2 *Q1* 之輸出結果 13

圖 2.3 MaxMatch 演算法 14

圖 2.4 第一次追蹤後之 *dMatch* 和 *dMatchSet* 17

圖 3.1 TDPrune 系統架構 19

圖 3.2 Node 結構 20

圖 3.3 計算 dmatch_range 前之 Node 結構狀態 21

圖 3.4 *Q1* 之 *kwMatch* 及兩個 SLCA 計算 dmatch_range 後之 Node 結構狀態 23

圖 3.5 GroupMatchRange 演算法 24

圖 3.6 節點 “1.2.2” 之 Node 結構狀態 25

圖 3.7 TDPruneMatches 演算法 29

圖 3.8 *Q2* 之 3 個 *kwMatch* 及 SLCA 節點 “1” 之 Node 結構 30

圖 3.9 節點 “1” 執行第一次 *FindNextChild* 函式後之 *dmatch_range* 狀態 30

圖 3.10 節點 “1” 執行第二次 *FindNextChild* 函式後之 *dmatch_range* 狀態 31

圖 3.11 節點 “1” 執行第三次 *FindNextChild* 函式後之 *dmatch_range* 狀態 31

圖 3.12 節點 “1” 執行第四次 *FindNextChild* 函式後之 *dmatch_range* 狀態 31

圖 3.13 *tmplist* 與其中節點之子孫對應關鍵字集合 34

圖 3.14 *Q2* 之輸出結果 36

圖 3.15 FindNextChild 演算法 38

圖 3.16 *Q2* 中所有對應節點及其祖先節點 43

圖 4.1 LevelPrune 系統架構 46

圖 4.2 LevelPruneMatches 演算法 49

圖 4.3 行導向儲存之 $kwMatch_3$ 52

圖 4.4 列導向儲存之 $kwMatch_3$ 52

圖 4.5 節點 “1.2.2” 與節點 “1.2.3” 之記憶體位置 53
圖 4.6 於壓縮前之 *kwMatch*$_3$ 指定“1.2” 的 *dmatch_range* 所需進行之檢查動作 ... 54
圖 4.7 壓縮前後之 *kwMatch*$_3$ 55
圖 4.8 於壓縮後之 *kwMatch*$_3$ 指定“1.2” 的 *dmatch_range* 所需進行之檢查動作 ... 56
圖 4.9 壓縮索引所節省之儲存空間 57
圖 5.1 *TQ1 ~ TQ8* 之實驗柱狀圖 (Baseball) 60
圖 5.2 *TQ9 ~ TQ16* 之實驗時間柱狀圖 (Mondial) 61
圖 5.3 *TQ17 ~ TQ20* 之實驗柱狀圖 (DBLP) 64
圖 5.4 *TQ21 ~ TQ24* 之實驗柱狀圖 (SwissProt) 64
圖 5.5 *TQ25 ~ TQ27* 之實驗柱狀圖 (DBLP) 67
圖 5.6 *TQ28 ~ TQ30* 之實驗柱狀圖 (SwissProt) 67
圖 5.7 輸出節點數量的實驗柱狀圖 (DBLP) 69
圖 5.8 查詢句 *TQ35 ~ TQ38* 之輸出結構圖 71
圖 5.9 貢獻比率的實驗柱狀圖 (DBLP) 71
圖 5.10 custom 系列資料集之結構 73
圖 5.11 *TQ39 ~ TQ42* 之實驗柱狀圖 74

表目錄

表 2.1 查詢句 *Q1* 10

表 3.1 查詢句 *Q2* 30

表 5.1 資料集之各項參數 58

表 5.2 壓縮前後索引大小 58

表 5.3 查詢句 *TQ1 ~ TQ16* 60

表 5.4 查詢句 *TQ1 ~ TQ16* 之各式節點統計 62

表 5.5 查詢句 *TQ17 ~ TQ24* 63

表 5.6 查詢句 *TQ17 ~ TQ24* 之各式節點統計 65

表 5.7 查詢句 *TQ25 ~ TQ30* 66

表 5.8 查詢句 *TQ25 ~ TQ30* 之各式節點統計 68

表 5.9 查詢句 *TQ31 ~ TQ34* 69

表 5.10 查詢句 *TQ31 ~ TQ34* 之各式節點統計 70

表 5.11 查詢句 *TQ35 ~ TQ38* 70

表 5.12 查詢句 *TQ35 ~ TQ38* 之各式節點統計 72

表 5.13 *TQ35* 各系統對各層所有節點之總處理時間 72

表 5.14 查詢句 *TQ39 ~ TQ42* 73

第一章 緒論

在此章，說明本論文的研究動機、目的和研究的方法，以及提出本論文的貢獻，並介紹相關的研究，最後說明本論文的架構以及各章節的內容。

1.1 研究動機與目的

隨著網際網路 (Internet) 的日益發達，其便利以及快速的特性已使其逐漸成為了資訊交流的管道，全球資訊網 (WWW) 更發展成為網際網路上資訊分享的平台。然而，在全球資訊網上所廣泛使用的 HTML 文件，由於其格式的特性較不易使用程式對其進行大量自動化的處理，使得可再利用性大大地減低。因此，W3C 提出了 XML (Extensible Markup Language) 這種可讓使用者利用標籤與結構自行定義文件型態的文件以供電子資料交換。

針對 XML 文件的處理，W3C 提出 XPath 與 XQuery 此兩種包含結構關係的查詢語言供使用者進行查詢處理。此兩種語言可以透過指定元素之間的結構關係來限定回傳的資料集合，進而更快速地滿足使用者的資訊需求。然而，在進行此類的查詢時，使用者除了必需學習這兩種語言的語法外，還需要知道該 XML 文件中元素之間的結構關係。在無法滿足以上兩個條件的情況下，使用關鍵字來進行查詢便成為了一個重要的研究課題。

由於關鍵字查詢缺乏了對 XML 文件結構的限制，其回傳結果的資料量相對之下便較為龐大，查詢結果也常常包含許多與使用者的資訊需求不相關的資料。因此，許多有關選擇適當回傳資料，以及過濾不必要回傳資料的研究便油然而生。其中[LC08]利用比較元素之間所包含關鍵字的數量來判斷是否輸出該元素，其篩選回傳結果的方式廣泛地獲得其他研究應用。因此，本論文希望依據[LC08]所定義的規則，提出更有效率的方法來進行篩選運算，以期縮短查詢所花費的時間以及在計算過程中所使用的記憶體空間。

1.2 研究方法與貢獻

首先，我們先說明[LC08]所定義的關鍵字查詢輸出結果。[LC08]所定義的輸出結果為 *n* 個答案樹，*n* 等於所有關鍵字之最小最低共同祖先 (Smallest Lowest Common Ancestor, SLCA) 的個數。而每一個答案樹則包含了一個 SLCA，以及該 SLCA 到其下所有相關對應節點 (relevant match) 之路徑上的所有節點。相關對應節點的判定則是利用到了貢獻節點 (contributor) 的觀念；若包含一個或一個以上關鍵字的節點到 SLCA 之路徑上的所有節點皆為貢獻節點，則該節點為相關對應節點。因此，為了產生最後的答案樹，必需對大量的節點進行判斷是否為貢獻節點的動作。

接著我們說明[LC08]所提出的 MaxMatch 系統如何實作。MaxMatch 利用[XP05]所提出的方法找出所有關鍵字的 SLCA，並將每一個 SLCA 視作一個群。再將所有樹中對應到關鍵字的對應節點分配到和其有著祖孫關係的 SLCA 所屬的群中後加以排序，再分別對每一群進行兩次節點的追蹤 (traversal) 以判別貢獻節點，進而產生答案樹。其中，第一次的追蹤為後序追蹤 (postorder traversal)，MaxMatch 利用杜威編碼的特性產生所有對應節點到 SLCA 之路徑上的所有節點，並建立完整的樹狀關係，再由下而上 (bottom-up) 地為樹中的每個節點記錄其及其子孫節點中出現過的關鍵字集合，以供後續判別貢獻節點之用。第二次的追蹤則為先序追蹤 (preorder traversal)，由上而下 (top-down) 地對樹中每個節點檢查該節點是否為貢獻節點。最後，再檢查所有對應節點是否符合相關對應節點的條件，進而產生輸出結果。

本論文提出兩個系統來產生[LC08]所定義的關鍵字查詢輸出結果，分別為 TDPrune 和 LevelPrune。兩個系統皆可由預先產生的反轉索引 (inverted index) 中獲得足以判別某一節點是否為貢獻節點的資訊。因此，兩種方法皆僅需對樹中的節點進行一次追蹤來產生樹狀結構以及進行貢獻節點的辨別。在節點的追蹤方式

上，TDPrune 由每一群的 SLCA 開始，找出所有為貢獻節點的孩子節點，並對這些孩子節點逐一遞迴地執行前述找出為貢獻節點的孩子節點之動作。在過程中，TDPrune 會在處理某一節點前將該節點輸出，得到與[LC08]相符的輸出結果。而 LevelPrune 對節點的追蹤方式，則是一次性地處理所有在樹中位於同層的節點，找出這些節點各自為貢獻節點的孩子節點，如此逐層地產生節點，最後在確認相關對應節點後，再依相關對應節點與 SLCA 的相對結構關係產生輸出結果。

針對本論文主要貢獻，總結如下所示：

1. 本論文提出兩個系統來建立[LC08]所定義之關鍵字查詢的答案樹，第一個系統 TDPrune，利用對應節點的反轉索引隱含樹狀結構和各節點及其孩子所包含的關鍵字集合的關係，減少追蹤答案樹中之節點的次數。同時，利用遞迴由上而下 (top-down) 地產生答案樹，此建立方式能儘早排除非貢獻節點之節點，減低運算量，進而改進產生輸出結果的效率。

2. 第二個系統 LevelPrune，不同於 TDPrune，LevelPrune 採分層式計算；先產生該層中的所有節點的孩子節點，並加以判斷是否為貢獻節點，在記錄完所有確定為相關對應節點的對應節點之後，再計算下一層。直到該層中沒有任何節點為止。最後，再依照所記錄的相關對應節點來建立答案樹。此方法能避免遞迴產生的額外參數堆疊，降低動態記憶體配置的次數以及記憶體的使用量，並利用記憶體空間區域性 (spacial locality) 的特性來增加執行效率。

3. 應用編碼行導向儲存、編碼壓縮和反轉索引壓縮此三種改善方式來降低線上執行查詢句時反轉索引的讀取量，並且降低創立答案樹中單一節點所需的運算量來進一步提升效率。此三種改善方式可套用至 TDPrune 以及 LevelPrune 之上，而我們將其實作在 LevelPrune 上，並將該系統稱作 LevelPrune+。

4. 我們針對 TDPrune、LevelPrune、LevelPrune+和 MaxMatch 進行一系列的實驗，實驗結果顯示本論文所提出的 TDPrune、LevelPrune 和 LevelPrune+系

統在各種情況下皆花費較 MaxMatch 少的時間來產生查詢結果，而又以貢獻節點被排除之比率較大的情況下，所提升的效率更為明顯。而三個系統之間，除了在少數極端的狀況下，TDPrune 會較 LevelPrune+為快之外，在大部份的情況下，LevelPrune+皆有著較快的執行效率。

1.3 相關研究

在關鍵字查詢方面，有許多針對選擇回傳結果所進行的研究。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，該論文定義了相關對應節點 (Relevant Matches) 來限制在最小最低共同祖先 (SLCA) 之下所需回傳的節點類型，並提出了 MaxMatch 來刪減子樹中不必要輸出的節點，進而產生答案樹。[LCC10]針對[LC08]所定義的關鍵字查詢之答案樹範圍提出了兩種不同的演算法 MinMap 與 SingleProbe。前者與 MaxMatch 類似，先找出所有對應節點的最小最低共同祖先。最後在刪減子樹中不輸出的節點時，藉由減少讀取索

引的次數來改善其執行效率。而後者則是先進行整個由對應節點所建立出的子樹進行刪減動作後，再找出其最小最低共同祖先，最後再將符合的結果依所屬最小最低共同祖先分群後輸出。

在考慮回傳符合查詢條件的結果上，如何在避免大量計算下找出前 k 個結果的 Top-k 研究，近來也是受到關注的課題。研究[TSW05]希望找出 Top-k 個包含所給定關鍵字的 XML 文件：在預先將所有包含相同標籤的節點依照 BM25 的大小依序存放於 Inverted List 中後，對所有符合關鍵字的 Inverted List 以 round-robin 的方式，依據 Inverted List 中節點的資訊來更新 XML 文件整體的分數，過程中並配以一個大小為 k 的 Top-k 陣列，以及一個 candidate 陣列來進行判斷，在把前 k 個 xml 文件放入 Top-k 陣列後，每次讀到一個 XML 文件便把其放入 candidate 陣列中，並維護 candidate 陣列中所有 XML 文件分數的門檻值，當 candidate 陣列中所有 XML 文件分數的門檻值皆小於 Top-k 陣列中 XML 文件的分數時，便把 top-k 陣列中的 XML 文件輸出。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。[VOPT08]則針對分散的環境，希望找出前 K 個和查詢句最相關的資料庫，以避免不必要的查詢處理。其方法是將 query 內的 Keyword 建立成 join connection tree，其中 node 代表 keyword，若表示 keyword 的 tuple 可 join 則建立 edge，然後同樣替每個資料庫根據關鍵字和其之間是否可 join 的關係建立 keyword relationship graph(KRG)，但是 edge 會額外標示權重以代表兩點之間做 join 需要的次數。查詢的時候，會利用此權重來計算 KRG 中 subgraph 的分數，然後其中包含最多關鍵字且分數最低的 subgraph 稱作 candidate graph。最後從各個資料庫中所建立的 candidate graph 中，取出前 k 個包含最多 keyword 的 candidate graph 所屬的資料庫。

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。針對 XML keyword search，論文[CP10]研究如何快速找出前 top-k 個符合查詢關鍵字的 SLCA。利用其創造的編碼格式 J-Dewey，能夠以完整編碼中的任一個數字來代表它是該層的第幾個節點，再配上依分數排序的關鍵字 inverted lists，便能以 bottom-up 的方式去比對每一層的所有節點，由於 SLCA 的定義為子孫節點必需包含所有關鍵字，因此必需在找到同時出現在各個關鍵字的 inverted list 之中的元素，且其分數總和大於門檻值後才能將該元素判定為 SLCA 並加以輸出。論文[ABBP10]希望找出前 k 個與輸入的查詢樹最為相似的樹，其研究利用 k 的值來推算結果子樹的大小，並依動態規劃(dynamic programming)的方式來計算所有符合條件的子樹與原查詢樹的 edit distance，在過程中，使用了 prefix ring buffer 來減少運算時所需的記憶體空間。[LLZW11]中提出兩個利用 top-k 原理對 Probabilistic XML document 尋找前 top-k 個 SLCA 的演算法，第一個是以 bottom-up 方式掃描過文件中的所有節點之後再計算一個節點成為 SLCA 的機率的 PrStcak 演算法，第二個是以傳統 Eager 演算法為基礎，並以各種 distribution node 的性質所推導出數個性質在運算中過濾掉確定機率過小不符合 Top-k 的節點，最後回傳 top-k 結果的 EagerTopK 演算法。

1.4 論文架構

本論文其餘各章節的架構如下：第二章介紹本論文所會使用到的相關定義，包括 XML 的資料表示法、XML 文件的編碼、關鍵字查詢的輸出結果，藉以對本論文所期望解決的問題做詳細的定義，並介紹本論文所欲改善的對象，也就是研究[LC08]所提出的 MaxMatch 系統。在第三章及第四章分別說明我們所提出的 TDPrune 以及 LevelPrune，說明其整體架構以及個別模組的演算法，並對

LevelPrune 套用了編碼行導向 (column-oriented) 的儲存機制、kwMatch 壓縮和反轉索引壓縮的改進，提出了 LevelPrune+。系統改善前後效率上的差異則會在第五章中以實驗進行比較，此外，與 MaxMatch 的比較也會一併於第五章中進行。最後，在第六章提出結論以及本論文未來的研究方向。

第二章 相關定義

我們在此章說明相關的定義，包括 XML 資料表示法，關鍵字查詢之輸出。接著，我們定義了本論文所針對的問題，並介紹其他研究針對相同問題的演算法 MaxMatch。

2.1 XML 資料表示法

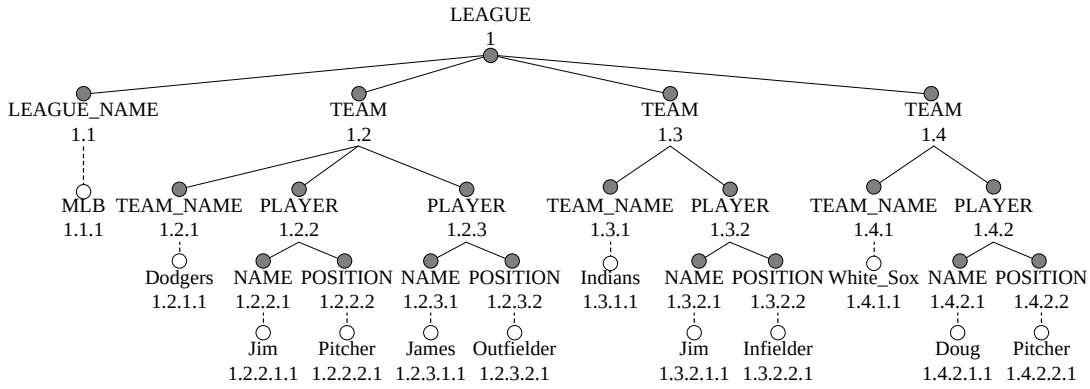

圖 2.1 範例 XML 樹

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>Jim</NAME>表示了一個 “NAME” 元素，其內容為 “Jim” 。另外，元素間必須具有嚴謹的巢狀包含關係，譬如：

<PLAYER>

<NAME>Jim</NAME>

<POSITION>Pitcher</POSITION>

</PLAYER>

表示 “PLAYER” 元素中包含了兩個元素，分別是 “NAME” 和 “POSITION” 。我們通常將 XML 文件以樹狀結構來表示，如圖 2.1 所示。因此，元素或內容也可被稱為一個 “節點” 。其中，實心圓表示一般元素節點，旁邊的文字是 “標籤” ，例如 NAME 是一個標籤。空心圓表示內容節點，其下的文字

則是“元素內容”。節點間的直線則表示兩者之間的巢狀關係。 在此範例樹中，“LEAGUE”表示該文件的根節點，其下“LEAGUE_NAME”元素記錄該聯盟的名稱，三個“TEAM”元素分別記錄三個不同球隊的資料，每個“TEAM”元素下又各自包含了一個“TEAM_NAME”元素表示該隊伍的名稱，和若干個記錄球員資料的“PLAYER”元素。而每個“PLAYER”元素則記錄著球員的名字(NAME) 以及守備位置 (POSITION) 。在往後的章節中，我們便以此 XML 樹做為各查詢句查詢的對象。

為了可以快速辨認特定的節點，將各個節點加上獨特的編碼是一種常見的做法，在圖 2.1 中的 XML 樹中，我們採取的編碼方式為杜威編碼 (Dewey Order Encoding) [1]，該法為根節點給定編碼“1”，而其他的節點的編碼則是依照父親節點的編碼，再連接上自己是父親節點的第幾個子節點，並以“.”分隔。如根節點“LEAGUE”之編碼為“1”，其四個孩子節點的編碼由左至右分別為“1.1”、“1.2”、“1.3”、“1.4”。由於每個孩子節點都記錄著其父親節點的編碼，我們可以輕易的得知任一節點的所有祖先節點編碼，如編碼“1.2.2.1”的“NAME”節點，其所有祖先節點由上至下分別為“1”的“LEAGUE”節點、“1.2”的“TEAM”節點、以及“1.2.2”的“PLAYER”節點。同時，針對多個節點，我們可以由這些節點之杜威編碼的最前方開始，擷取最長的相同部份，來很快速地找到這些節點的最低共同祖先 (Lowest Common Ancestor, LCA) 。以“1.2.1.1”的“Dodgers”節點、“1.2.2.1.1”的“Jim”節點、和“1.2.2.2”的“POSITION”節點為例，其 LCA 即此三個節點的編碼由最前方開始之最長的相同部份“1.2”，即“TEAM”節點。在本論文中，我們便以節點的杜威編碼來表示該節點。

[1] http://www.oclc.org/dewey/

2.2 關鍵字查詢之輸出結果

在此節中，我們參考[XP05]和[LC08]的論文，將本論文會使用到的定義說明如下：

[定義 2.1]：對應節點 (Match) ：若關鍵字 k 與 XML 文件中節點 n 相同，則稱節點 n「**對應**」關鍵字 k，或 v 為一個關鍵字 k 的「**對應節點**」。並將所有對應到關鍵字 k 的節點蒐集成一個「**對應節點集合**」，以 S_k 表示。

[範例 2.1]：考慮圖 2.1 中的 XML 樹，關鍵字 “POSITION” 的「**對應節點集合**」$S_{POSITION}$ 為 { 1.2.2.2, 1.2.3.2, 1.3.2.2, 1.3.3.2, 1.3.4.2 } 。

[定義 2.2]：節點集合之最低共同祖先 (Lowest Common Ancestor, LCA)：給予 i 個關鍵字 $k_1, \cdots, k_i$，及其對應節點集合 $S_1, \cdots, S_i$，假如存在 $n_a, n_1 \in S_1, \cdots, n_i \in S_i$ 使得 n_a 為 $n_1, \cdots, n_i$ 的最低共同祖先，則稱 n_a 為 $S_1, \cdots, S_i$ 的一個「**節點集合之最低共同祖先**」，記為 $n_a \in lca(S_1, \cdots, S_i)$。

編號	關鍵字	資訊需求
Q1	Jim 、 POSITION 、 TEAM_NAME	球員 Jim 的所屬隊伍 (TEAM_NAME) 以及其守備位置 (POSITION)

表 2.1 查詢句 *Q1*

[範例 2.2]：考慮表 2.1 中的查詢句 *Q1*，關鍵字 “Jim” 的對應節點集合 S_{Jim} = { 1.2.2.1.1, 1.3.2.1.1 } ，關鍵字 POSITION 的對應節點集合 $S_{POSITION}$ = { 1.2.2.2, 1.2.3.2, 1.3.2.2, 1.4.4.2 } ，關鍵字 TEAM_NAME 的對應節點集合 S_{TEAM_NAME} = { 1.2.1, 1.3.1, 1.4.1 } ，則 S_{Jim}、$S_{POSITION}$ 與 S_{TEAM_NAME} 的「**節點集合之最低共同祖先**」$lca(S_{Jim}, S_{POSITION}, S_{TEAM\_NAME})$ 為 { 1.2, 1, 1.3 } 。

[定義 2.3]：最小最低共同祖先 (Smallest Lowest Common Ancestor, SLCA)：若節

點 $n_1 \in lca(S_{k1}, \cdots, S_{kn})$，$\forall\ n_2 \in lca(S_{k1}, \cdots, S_{kn})$，$n_1$ 不是 n_2 的祖先，則 n_1 為 $S_1, \cdots, S_n$ 的一個「**最小最低共同祖先** (SLCA) 」，記為 $n_1 \in slca(S_1, \cdots, S_n)$。

[範例 2.3]：考慮以查詢句 *Q1* 進行查詢，S_{Jim}、$S_{POSITION}$ 與 S_{TEAM_NAME} 的最低共同祖先集合為{ 1.2, 1, 1.3 }，其中 1 為 1.2 的祖先，因此 $1 \notin slca(S_{Jim}, S_{POSITION}, S_{TEAM\_NAME})$，最後得到所有「**最小最低共同祖先** (SLCA) 」$slca(S_{Jim}, S_{POSITION}, S_{TEAM\_NAME})$ 為 { 1.2, 1.3 }。

[定義 2.4]：子孫對應關鍵字集合 (descendant match) ：對於一棵 XML 樹 *D* 以及查詢句 *Q*，其中節點 $n \in D$ 的「**子孫對應關鍵字集合**」為 *Q* 中滿足下述條件的關鍵字的集合，表示為 $dMatch(n)$，則$\forall\ k \in dMatch(n)$, 以 *n* 為根節點所形成之子樹的所有節點中至少有一個節點對應 *k*。

[定義 2.5]：貢獻節點 (contributor) ：對於一棵 XML 樹 *D* 以及包含關鍵字 $k_1, \ldots, k_i$ 的查詢句 *Q*，若節點 $n \in D$ 對於 *Q* 為「**貢獻節點**」，則 *n* 必須滿足條件如下：

(i) 存在一個 *n* 的祖先節點 $n_a \in slca(S_{k1}, \ldots, S_{ki})$，或 $n \in slca(S_{k1}, \ldots, S_{ki})$。

(ii) *n* 不能有兄弟 n_s，使得 $dMatch\ (n) \subset dMatch(n_s)$。

[範例 2.4]：考慮以查詢句 *Q1* 進行查詢，得到 $slca(S_{Jim}, S_{POSITION}, S_{TEAM\_NAME})$ = { 1.2, 1.3 } ，欲判別節點 1.2.3.2 對於 *Q1* 是否為「**貢獻節點**」，節點 1.2.3.2 之兄弟節點只有一個，為 1.2.3.1，1.2.3.2 的「**子孫對應關鍵字集合**」$dMatch$ (1.2.3.2) = { POSITION } ，1.2.3.1 的子孫對應關鍵字集合 $dMatch$ (1.2.3.1) = ϕ。因為存在一個 1.2.3.2 的祖先節點 $1.2 \in slca(S_{Jim}, S_{POSITION}, S_{TEAM\_NAME})$，且 $dMatch(1.2.3.2) \not\subset dMatch(1.2.3.1)$ ，所以節點 1.2.3.2 對於 *Q1* 為「**貢獻節點**」。欲判別節點 1.2.3 對於 *Q1* 是否為貢獻節點，因為存在一個節點 1.2.3 的兄弟節點 1.2.2，$dMatch$ (1.2.3) = { POSITION } ，$dMatch$ (1.2.2) = { Jim, POSITION }，使得 $dMatch(1.2.3) \subset dMatch(1.2.2)$，所以節點 1.2.3 對於 *Q1* 不是貢獻節點。

[定義 2.6]：相關對應節點 (relevant match)：對於一棵 XML 樹 D 以及包含關鍵字 $k_1, \ldots, k_i$ 的查詢句 Q，若一個對應節點 $m \in D$ 對於 Q 為「**相關對應節點**」，則必須滿足下述條件之一：

(i) $m \in slca(S_{k1}, \ldots, S_{ki})$。

(ii) 存在一個 m 的祖先節點 $n_a \in slca(S_{k1}, \ldots, S_{ki})$，且 m 到 n_a 之路徑上的所有節點皆為貢獻節點。

[範例 2.5]：考慮以查詢句 *Q1* 進行查詢，得到 $slca(S_{Jim}, S_{POSITION}, S_{TEAM\_NAME})$ = { 1.2, 1.3 }，所有第一個 SLCA 節點 1.2 的子孫節點中，是貢獻節點的節點共有 1.2.1、1.2.2、1.2.2.1、1.2.2.1.1、1.2.2.2、1.2.3.2。其中，對應節點 1.2.1 與節點 1.2 路徑上的所有節點 (1.2.1、1.2) ，對應節點 1.2.2.1.1 與節點 1.2 路徑上的所有節點 (1.2.2.1.1、1.2.2.1、1.2.2、1.2) ，以及對應節點 1.2.2.2 與 1.2 路徑上的所有節點 (1.2.2.2、1.2.2、1.2) 皆為貢獻節點，因此這三個對應節點 1.2.1.1、1.2.2.1.1、1.2.2.2 對於 *Q1* 為「**相關對應節點**」。

[定義 2.7]：查詢結果：對於一棵 XML 樹 D 和包含關鍵字 $k_1, \ldots, k_i$ 的查詢句 Q，產生一組查詢結果的集合 R，表示為 $R = (Q, D)$ ，R 中的查詢結果個數為 $slca(S_1, \cdots, S_n)$ 中的 SLCA 節點個數，而每一個查詢結果為一棵樹，以 $r = (t, M)$ 表示。$t \in slca(S_1, \cdots, S_n)$，$M$ 則是所有 t 的子孫節點，且為相關對應節點的節點而形成的集合，M 至少包含每個關鍵字一次。每一個查詢結果 r 所形成的樹，包含所有在 D 中從 t 連接到 M 中的所有相關對應節點所形成的路徑上所經過的全部節點，若相關對應節點包含形態為內容節點的孩子節點，則將其一併納入輸出結果。

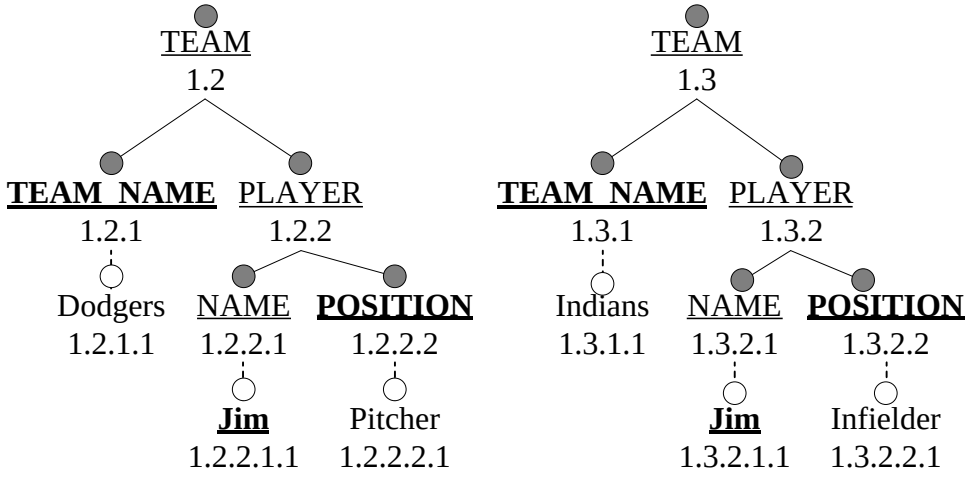

圖 2.2 *Q1* 之輸出結果

[範例 2.6]：考慮以查詢句 *Q1* 進行查詢，得到 $slca(S_{Jim}, S_{POSITION}, S_{TEAM\_NAME})$ = { 1.2, 1.3 } ，最後得到查詢結果集合 $R = \{r_1, r_2\}$，其中 $r_1 = (t_1, M_1)$ = (1.2, {1.2.1, 1.2.2.1.1, 1.2.2.2}) ，$r_2 = (t_2, M_2)$ = (1.3, {1.3.1, 1.3.2.1.1, 1.3.2.2}) 。r_1 和 r_2 所代表的答案樹分別如圖 2.2 中左樹和右樹所示。其中，所有的貢獻節點為 1.2、1.2.1、1.2.2、1.2.2.1、1.2.2.1.1、1.2.2.2、1.3、1.3.1、1.3.2、1.3.2.1、1.3.2.1.1、1.3.2.2，即圖中標籤名稱加底線標示之節點。所有的相關對應節點為 1.2.1、1.2.2.1.1、1.2.2.2、1.3.1、1.3.2.1.1、1.3.2.2，即圖中標籤名稱以粗體字標示之節點。而 1.2.1.1、1.2.2.2.1、1.3.1.1、1.3.2.2.1 則因為是相關對應節點所包含之內容節點，故將這四個節點一併輸出。

2.3 MaxMatch 演算法介紹

[LC08]針對其所定義的輸出結果，提出了 MaxMatch 演算法來判別元素是否為貢獻節點。我們以查詢句 *Q1* 進行查詢來說明 MaxMatch 演算法及其各個模組的功用，並將 *groupMatches*、*pruneMatches* 模組的完整演算法列於附錄 A 和附錄 B。

演算法名稱：MaxMatch 輸　　入：*keyword*[*w*] (*w* 個關鍵字的查詢句) 輸　　出：所有最小最低共同祖先到其下所有相關對應節點的路徑上的所有元素
L01　*kwMatch* ← *findMatch*(*keyword*) L02　*SLCA* ← *findSLCA*(*kwMatch*) L03　group ← *groupMatches*(*kwMatch, SLCA*) L04　**for all** *group*[*j*] **do** L05　　*pruneMatches*(*group*[*j*])

圖 2.3 MaxMatch 演算法

MaxMatch 主要分為四個步驟，如圖 2.3 所示。首先，在 L01 利用 *findMatch* 模組讀取查詢句中各個關鍵字的對應節點，接著在 L02 以 *findSLCA* 模組來計算出這些對應節點的最小最低共同祖先，然後在 L03 以 *groupMatches* 模組將每個最小最低共同祖先節點與其所屬的對應節點分群，並建立出完整的樹狀結構，最後在 L04-L05 對每一群以 *pruneMatches* 判斷逐一所有節點是否為貢獻節點，進而產生輸出結果。

對於包含一組 *w* 個關鍵字的查詢句，*findMatch* 模組從預先建立好的反轉索引 (inverted index) 逐一為各關鍵字讀取在 XML 文件中對應節點的杜威編碼並存至 $kwMatch_j$，$1 \le j \le w$。其中反轉索引建立的格式為以節點名稱為鍵值 (key)，所有對應該節點名稱之節點的杜威編碼做為資料。*findSLCA* 模組接著採用[XP05]所提出的演算法來找出這些節點群的最小最低共同祖先，並將結果存至 *SLCA* 中。

[範例 2.7]：考慮以查詢句 *Q1* 進行查詢，讀取 *Q1* 中的三個關鍵字 Jim、POSITION、TEAM_NAME 之所有對應節點的杜威編碼，分別為 {1.2.2.1.1, 1.3.2.1.1} 、

{1.2.2.2, 1.2.3.2, 1.3.2.2, 1.4.2.2} 、 {1.2.1, 1.3.1, 1.4.1} 。*findSLCA* 模組接著對此三個節點集合找出其最小最低共同祖先 $slca(S_{Jim}, S_{POSITION}, S_{TEAM\_NAME})$ = {1.2, 1.3} 。

groupMatches 模組接著將所有對應節點其所屬的最小最低共同祖先分群並依其在 XML 文件中出現的順序進行排序；若該對應節點為 SLCA 節點 *t* 的子孫，或該對應節點即 *t*，則將兩者歸類為同一群。每一群以 $G_i = (t_i, M_i)$ 表示，其中 *i* 代表群的編號，用以辨認該群。t_i 為該群之 SLCA 節點，M_i 則為所有滿足上述條件的對應節點所形成的集合。

[範例 2.8]：考慮以查詢句 *Q1* 進行查詢，由於最小最低共同祖先有兩個節點，分別為 1.2、1.3，因此會將符合條件的對應節點分為兩群並排序後得到 $G_1 = (t_1, M_1)$ = (1.2, {1.2.1, 1.2.2.1.1, 1.2.2.2, 1.2.3.2}) , $G_2 = (t_2, M_2)$ = (1.3, {1.3.1, 1.3.2.1.1, 1.3.2.2}) 。

最後，MaxMatch 會對每一群以 *pruneMatches* 模組將該群的所有節點依照相互之間的結構關系建立完整的樹狀結構。並對樹中的每一個節點給定一個大小為關鍵字個數 *k* 的布林陣列 (Boolean array) 以記錄子孫對應關鍵字集合 (descendant match) ，名為 *dMatch*，用以後續判別貢獻節點之用。*dMatch* 的記錄規則為，若某一節點 *n* 的任一子孫節點包含查詢句中的第 *i* 個關鍵字，則將 *dMatch*(*n*)中由右至左的第 *i* 個布林值設為 1。

[範例 2.9]：考慮以查詢句 *Q1* 進行查詢，*Q1* 包含 Jim、POSITION、TEAM_NAME 三個關鍵字，故 *dMatch* 的大小為 3。以 player (1.2.2) 為例，由於其子孫包含了 Jim、POSITION 此兩個關鍵字，分別為 *Q1* 中的第 1 個和第 2 個關鍵字，因此將該節點之 *dMatch* 由右至左的第 1 個和第 2 個布林值設為 1，得到 *dMatch*(1.2.2) = “011” 。

在判別貢獻節點方面，依據定義 2.5，若節點 n 有兄弟 n_s，使得 $dMatch(n) \subset dMatch(n_s)$，則 n 不為貢獻節點。而 *groupMatches* 模組採取的策略則是將給定所有節點一個大小為 2^w 的布林陣列 (boolean array) *dMatchSet*，其中 w 為關鍵字個數，用來記錄所有孩子節點的子孫對應關鍵字集合資訊，並藉以判別孩子節點是否為貢獻節點。具體而言，考慮節點 n，將 $dMatch(n)$ 的值視為一個二進位的數字，將其轉為十進位後的數字並令之為 i，則將 n 的父親節點 n_p 的 $dMatchSet[i]$ 設為 1。當 $n_p.dMatchSet[i]$ 已依照其所有孩子節點更新後，便可以使用 $n_p.dMatchSet[i]$所記錄的資料配合以下的規則來判別 n 是否為貢獻節點：

對一個節點 n 來說，節點 n_p 為 n 的父親節點，若 $dMatch(n)$ 轉為十進位後的數為 i，則當以下條件皆成立時，n 不為貢獻節點：

1. 存在 j，使得 $n_p.dMatchSet[j]$為 1
2. $AND(i, j) = i$
3. $i < j$

亦即存在一個 n 的兄弟節點 n_s，$dMatch(n_s)$ 轉為十進位後的數為 j，$dMatch(n) \subset dMatch(n_s)$ ($dMatch(n)$中所有的關鍵字皆屬於 $dMatch(n_s)$，且 $dMatch(n)$包含的關鍵字比 $dMatch(n_s)$少)。

[範例 2.10]：考慮以查詢句 *Q1* 進行查詢，TEAM (1.2)包含三個孩子節點，其孩子節點的 *dMatch* 由左至右分別為 100、011、010，故將 TEAM (1.2)的 *dMatchSet*[4]、*dMatchSet*[3]、和 *dMatchSet*[2]分別設為 1。考慮 TEAM (1.2)的第三個孩子節點 PLAYER (1.2.3)，其 *dMatch* = 010，因為存在 011 以十進位表示的數值 3，使得 TEAM (1.2)的 *dMatchSet*[3]為 1，*AND*(010,011) = 010 且 010 < 011，因此 PLAYER (1.2.3) 不為貢獻節點。

groupMatches 模組在對每一群進行處理時，會對所有節點進行一次後序追蹤 (postorder traversal) 以及一次前序追蹤 (preorder traversal) 。在第一次追蹤時，

會利用所有對應節點的杜威編碼產生至樹根節點路徑上的所有節點，並建立完整的樹狀結構關係，再對樹中的每一個節點利用其所有孩子節點的 *dMatch* 來計算該節點的 *dMatch*，並依據 *dMatch* 更新該節點之父親節點的 *dMatchSet*，而第二次的追蹤則會由樹根節點開始對已確定為貢獻節點其所有的孩子節點進行判別是否為貢獻節點的動作。

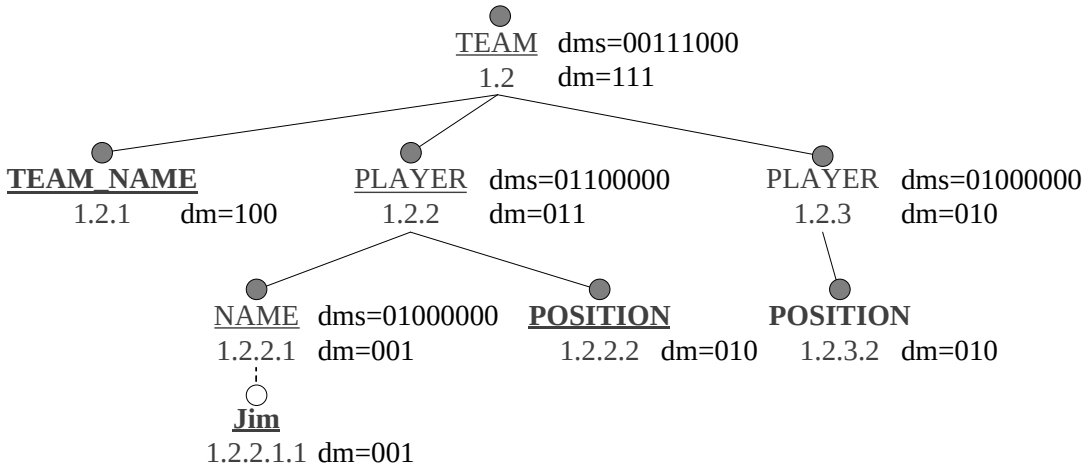

圖 2.4 第一次追蹤後之 *dMatch* 和 *dMatchSet*

[範例 2.11]：考慮以查詢句 *Q1* 進行查詢，對第一群 G_1 進行第一次的追蹤得到如圖 2.4 所示的 *dMatch* (dm) 和 *dMatchSet* (dms)資料，第二次的追蹤則由根節點開始，依據前次追蹤所得的資料，以 pre-order 的方式逐一對樹中所有父親是貢獻節點的節點檢查其是否為貢獻節點。注意到，因為 1.2.3 並非貢獻節點，因此並不會替其孩子節點 1.2.3.2 進行判別是否為貢獻節點的動作，我們最後得到的貢獻節點共有 1.2、1.2.1、1.2.2、1.2.2.1、1.2.2.1.1、1.2.2.2，如圖 2.4 中標籤名稱以底線標示之節點。

最後，MaxMatch 會對每一群 $G_i = (t_i, M_i)$ 中，檢查 M_i 所記錄的對應節點是否已在前一步驟被判斷為貢獻節點。注意到，由於在前一步驟我們由樹根開始，只檢查父親是貢獻節點的節點是否為貢獻節點，所以若一個對應節點被標記為貢獻節點，則其所有祖先節點必皆為貢獻節點，符合 MaxMatch 所定義的相關對應節點的條件。因此我們可以判定該對應節點為相關對應節點，進而利用其杜威編

碼來產生該相關對應節點至樹根節點路徑上的所有節點及其完整樹狀結構。

[範例 2.12]：考慮以查詢句 *Q1* 進行查詢，第一群 $G_1 = (t_1, M_1) = (1.2, \{1.2.1, 1.2.2.1.1, 1.2.2.2, 1.2.3.2\})$ 中，M_1 所記錄的四個對應節點 1.2.1、1.2.2.1.1、1.2.2.2、1.2.3.2 中，被標記為貢獻節點的節點共三個，為 1.2.1、1.2.2.1.1、1.2.2.2。利用杜威編碼產生這些節點與 $t_1 = 1.2$ 之間的樹狀結構關係，即可得到如圖 2.2 中左樹之輸出結果。

第三章 TDPrune 演算法

在本章中，我們介紹 TDPrune 系統的架構以及其各個子模組，並針對 TDPrune 系統中佔用最多執行時間的 *TDpruneMatches* 模組進行時間複雜度以及空間複雜度上的效率分析，以及將此模組與 MaxMatch 中所對應的模組進行比較。

3.1 系統架構

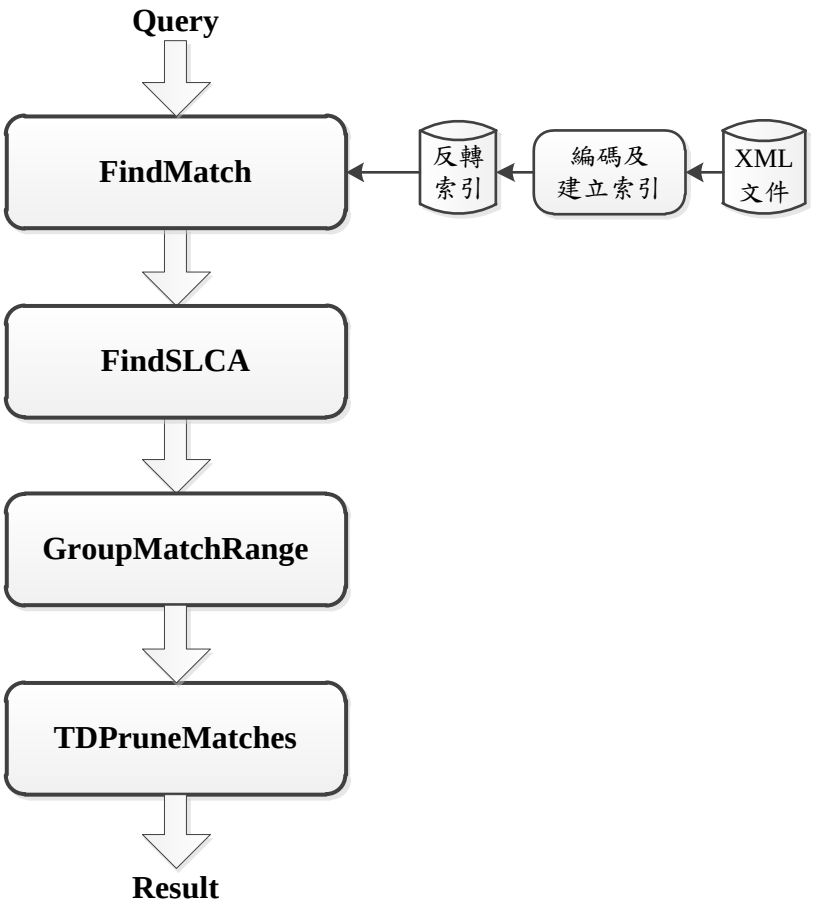

圖 3.1 TDPrune 系統架構

TDPrune 主要分為四個模組，如圖 3.1 所示。在進行線上查詢前，先為 XML 文件進行杜威編碼以及建立「反轉索引」 (inverted index) 的工作，反轉索引的格式與 MaxMatch 中的反轉索引相同，以節點名稱為鍵值 (key)，將所有對應該

節點名稱之節點的杜威編碼依節點在 XML 出現順序排序後做為回傳資料。在進行線上查詢時，依查詢句的關鍵字由 *FindMatch* 模組從索引中讀入資料後，交由 *FindSLCA* 模組計算出所有 SLCA 節點，再經由 *GroupMatchRange* 模組將 SLCA 節點與對應節點配對分群，最後由 *TDPruneMatches* 模組計算並輸出查詢結果。

3.2 FindMatch 模組和 FindSLCA 模組

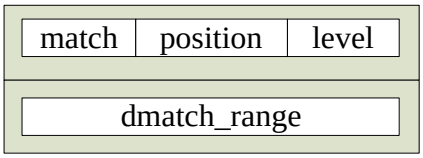

圖 3.2 Node 結構

如同 MaxMatch 一樣，TDPrune 首先以 *FindMatch* 模組，從索引中逐一讀出對應到查詢句中不同關鍵字的所有對應節點，並將其編碼存放至對應的 $kwMatch_i$ 中，其中 "*i*" 代表第 *i* 個關鍵字。每一個 $kwMatch_i$ 為一個 $r_i \times c_i$ 的二維數字陣列，r_i 為所有對應到該關鍵字之節點的個數，c_i 則為所有對應到該關鍵字之節點中，層數最深之節點的層數。節點之編碼存放的方式，則是將第 *j* 個對應到第 *i* 個關鍵字的節點之杜威編碼中每一層的數字，依序存放於 $kwMatch_i$ 之第 *j* 行的不同欄位中。而若該節點的層數小於 c_i，則將未存放任何資訊的欄位存以 0。

接著，*FindSLCA* 模組找出這些對應節點的所有 SLCA 節點，並對每一個 SLCA 節點建立一個結構如圖 3.2 所示的 Node 結構。其中，*dmatch_range* 為一個大小為關鍵字個數的兩倍之整數陣列，用以記錄該 Node 之下所有為對應節點的子孫節點，陣列中所有元素於節點初建立時皆設定為 0，我們將會在 3.3 節中說明其記錄的方法。*match*、*positon* 和 *level* 為三個整數，記錄著用以還原該節點之杜威編碼的資訊；由於所有 SLCA 皆為反轉索引中某一節點的祖先，因此我們只要記錄該節點為第 *match* 個關鍵字的反轉索引中的第 *positon* 個對應節點位於第 *level* 層的祖先，便可以以該節點之下的這三個整數配合反轉索引而得到其完

整的杜威編碼，做為最後輸出答案樹之用。在此我們將 SLCA 節點的 *match* 值皆設為 1。藉由這樣的記錄杜威編碼之方式，我們可以使 Node 結構的大小為一個固定值，以方便後續的處理。並能在建立層數大於 3 之節點的 Node 時，節省用以記錄杜威編碼的空間。

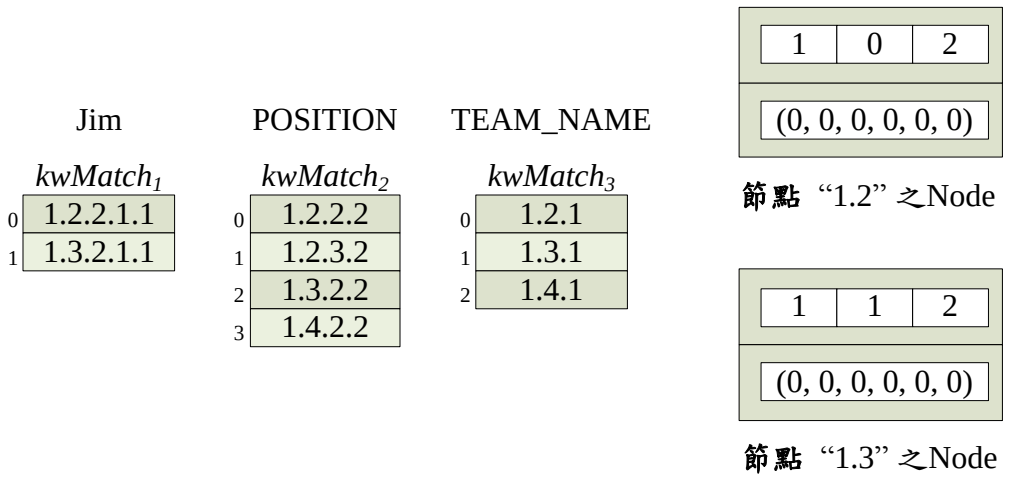

圖 3.3 計算 dmatch_range 前之 Node 結構狀態

[範例 3.1]：考慮以查詢句 *Q1* 進行查詢，對查詢句 *Q1* 中的三個關鍵字 Jim、POSITION、TEAM_NAME 讀取其所有對應節點的杜威編碼，分別為 $kwMatch_1$ = { 1.2.2.1.1, 1.3.2.1.1 } 、$kwMatch_2$ = { 1.2.2.2, 1.2.3.2, 1.3.2.2, 1.4.2.2 } 、$kwMatch_3$ = { 1.2.1, 1.3.1, 1.4.1 } ，其二維陣列結構如圖 3.3 左方所示。*findSLCA* 對此三群節點找出其 SLCA { 1.2, 1.3 } ，並分別對 1.2、1.3 建立 Node 結構。節點 "1.2" 為 $kwMatch_1$ 中，記錄於位置 0 的對應節點 (即 1.2.2.1.1) 位於第 2 層的祖先，因此我們設定節點"1.2" 的 Node 裡之 *match* = 1，*positon* = 0，*level* = 2。節點 "1.3" 為 $kwMatch_1$ 中，記錄於位置 1 的對應節點 (即 1.3.2.1.1) 位於第 2 層的祖先，因此我們設定節點"1.3" 的 Node 裡之 *match* = 1，*positon* = 1，*level* = 2。*Q2* 中共有 3 個關鍵字，因此所有 Node 裡的 *dmatch_range* 皆為一大小為 6 的整數陣列，且陣列中各元素於節點初建立時皆皆為 0，兩個 Node 結構之狀態如圖 3.3 右方所示。

在後續的計算中，我們會利用反轉索引中所記錄的對應節點資訊來產生出

SLCA 到其下對應節點之路徑上的某些節點，並使用 Node 結構來記錄其相關資訊。由於這些節點皆會為某一對應節點的祖先，因此我們一樣可以利用 *match*、*positon* 和 *level* 來還原該節點之杜威編碼。

3.3 GroupMatchRange 模組

GroupMatchRange 模組接著將所有對應節點與其所屬的 SLCA 分群，並將結果存至該 SLCA 之 Node 結構中。與 MaxMatch 不同的是，MaxMatch 利用一個節點的串列來記錄每一群。串列中的內容為所有關鍵字的對應節點，各節點並依照其於 XML 文件中出現的順序混合排序。其中，節點所紀錄的資訊為該節點的杜威編碼。而 TDPrune 為了配合後續的處理，並不會將所有關鍵字的對應節點混合排序，而是依關鍵字分開記錄於 Node 結構中，在考慮實際記錄的方式之前，我們先介紹以下的定理：

[定理 3.1]：考慮一個記錄對應節點杜威編碼的佇列 l，若 l 已依在 XML 文件中出現的順序排序，則對任一個節點 v_a，若佇列 l 中包含 v_a 的子孫節點，則在佇列 l 中所有 v_a 的子孫節點必位於連續的位置。

[證明]：考慮任意兩個為 v_a 子孫且位於佇列 l 的節點 v_1、v_2，$v_1 \neq v_2$，且 v_1 與 v_2 不位於連續的位置，而 v_a 的深度為 d_{va}。假設存在一個在佇列 l 中位於 v_1 與 v_2 之間的節點 v_3，且 v_3 不為 v_a 的子孫節點，因為 v_1 與 v_2 皆為 v_a 的子孫節點，所以 v_1 與 v_2 之編碼的前 d_{va} 位數與 v_a 相同。而 v_3 不為 v_a 的子孫節點，故 v_3 之編碼的前 d_{va} 位數與 v_a 不同，則 v_1、v_2 與 v_3 未依其在 XML 文件中出現的順序排序，與命題矛盾，故先前之假設不成立。也就是，不存在任何一個在佇列 l 中位於 v_1 與 v_2 之間的節點 v_3，且 v_3 不為 v_a 的子孫節點。即在佇列 l 中所有 v_a 的子孫節點必位於連續的位置。■

由於定理 3.1 的關係，對於代表任一節點的 Node 結構來說，我們可以利用

其 *dmatch_range* 記錄該節點之下對應到各個關鍵字的子孫節點在其 *kwMatch* 陣列的開始出現的位置與最後出現的位置，便能記錄該節點之下所有對應到各個關鍵字的子孫節點。更詳盡地說，對一個包含 w 個關鍵字的查詢句來說，TDPrune 利用 FindMatch 讀出 w 個對應到不同關鍵字的陣列 $kwMatch_1, \ldots, kwMatch_w$，*dmatch_range* 為一個 $2 \times w$ 大小的陣列，對於代表任一節點的 Node 來說，該 Node 的 *dmatch_range* 陣列中，位於第 $2 \times (n-1)$ 個位置的數字 r 代表該節點之下對應第 n 個關鍵字的子孫節點在 $kwMatch_n$ 中開始出現的位置 (第 r 列)，而第 $2 \times (n-1)+1$ 個位置的數字則代表代表該節點之下對應第 n 個關鍵字的子孫節點在 $kwMatch_n$ 中最後出現的位置。

藉由前述的記錄方式，我們只需對每一個 SLCA 建立一個 Node，並正確計算該 Node 之 *dmatch_range*，便可以以該 Node 表示一個包含著一個 SLCA 節點以及其下所有對應到各個關鍵字之子孫節點的群。

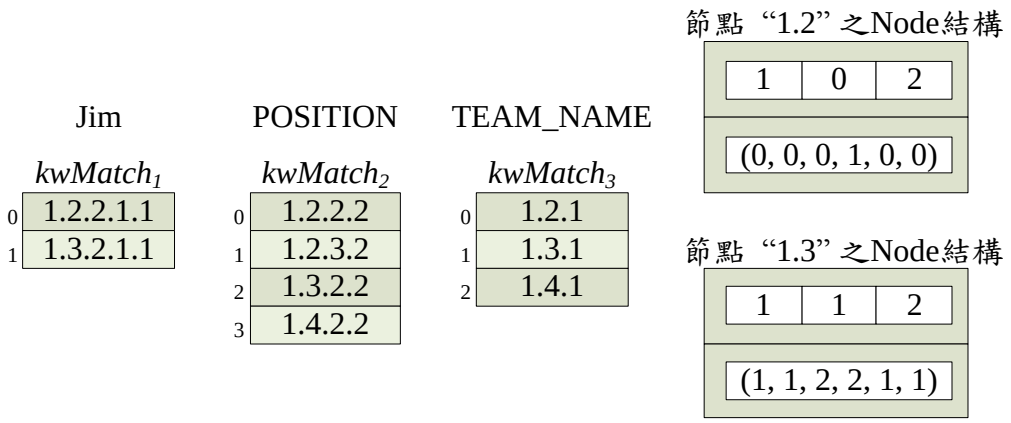

圖 3.4 *Q1* 之 *kwMatch* 及兩個 SLCA 計算 dmatch_range 後之 Node 結構狀態

[範例 3.2]：考慮以查詢句 *Q1* 進行查詢，對查詢句 *Q1* 中的三個關鍵字 Jim、POSITION、TEAM_NAME 讀取其所有對應節點的杜威編碼，結果如圖 3.4 左方所示。*findSLCA* 對此三群節點找出其 SLCA { 1.2, 1.3 } ，並分別對 1.2、1.3 建立 Node 結構。節點 "1.2" 之 Node 的 *dmatch_range* 為 (0, 0 , 0, 1 , 0, 0) 。括號中第 1 與第 2 個數字 0，0 分別表示所有 "1.2" 之下對應第 1 關鍵字 "Jim"

的子孫節點在 $kwMatch_1$ 開始出現的位置與最後出現的位置，第 3 與第 4 個數字 0，1 分別表示所有“1.2”之下對應第 2 個關鍵字“POSITION”的子孫節點在 $kwMatch_2$ 開始出現的位置與最後出現的位置，而第 5 與第 6 個數字 0，0 則分別表示所有“1.2”之下對應第 3 個關鍵字“POSITION”的子孫節點在 $kwMatch_3$ 開始出現的位置與最後出現的位置。節點“1.3”之 Node 的 *dmatch_range* 為（1, 1 , 2, 2 , 1, 1 ），記錄的方式如同前述，最後結果如圖 3.4 右方所示。

演算法名稱：GroupMatchRange

輸　　入：Node 串列 *nl_slca*

變　　數：*w*：查詢句中關鍵字之個數

position：記錄目前 *kwMatch* 讀取的位置

說　　明：更新 *nl_slca* 中每一個 *Node* 節點的 dmatch_range 值。

```
L01  for int i ← 0; i < w; i++ do
L02    position[i] ← 0;
L03  while ( (cur_n ← nl_slca.GetNode() ) != 0 )
L04    for int i ← 0; i < w; i++ do
L05      for ; position[i] < kwMatch_i.Size(); position[i]++ do
L06        if ( IsAncestor( cur_n, kwMatch_i[ position[i] ] ) )
L07          cur_n.dmatch_range[2*i] ← position[i]
L08          while ( position[i] < kwMatch_i.Size() ||
                       IsAncestor( cur_n, kwMatch_i[ position[i] ] ) )
L09            position[i]++;
L10          if ( position[i] = kwMatch_i.Size() )
L11            cur_n.dmatch_range[2*i] ← position[i]
L12          else cur_n. dmatch_range [2*i+1] ← position[i] - 1
```

圖 3.5 GroupMatchRange 演算法

完整的 GroupMatchRange 演算法如圖 3.5 所示。L01~L02 將 *position*[*i*]指定為 0，表示由反轉索引位置 0 開始讀取。L03 ~ L12 的迴圈不斷從記錄著所有 SLCA 節點的 Node 串列 *nl_slca* 中取出 SLCA 節點來進行處理。針對每一個取出的 SLCA 節點 *cur_n*，L04 ~ L12 的迴圈會執行 *w* 次，用以更新 *cur_n.dmatch_range* 中，代表該關鍵字開始出現以及最後出現的位置之數字。L05 ~ L12 從第 *i* 個反轉索引中的第 *position*[*i*]個位置開始檢查 *cur_n* 是否為該位置所記錄之節點的祖先。若是，則於 L07 設定其開始出現之位置，並利用 L08 ~ L12 來設定其最後出現之位置。注意到，若任一對應節點為某一 SLCA 節點的子孫節點，則其必不另一 SLCA 節點的子孫節點。因此在 L05~L12 的迴圈中，我們利用 *position*[*i*]來記錄前次迴圈終止時，所發現之第一個不為前次 SLCA 節點的對應節點之位置，於下次迴圈執行時，僅需由該位置開始進行檢查即可。因此，*GroupMatchRange* 僅需讀取反轉索引中所記錄的所有對應節點一次後，便可完成所有 SLCA 節點之 *dmatch_range* 之值的設定。

根據定理 3.1，除了 SLCA 節點以外，針對 SLCA 節點到其下對應節點之路徑上的任一節點，我們一樣可以使用 *dmatch_range* 來記錄該節點之下所有對應到各個關鍵字的子孫節點。

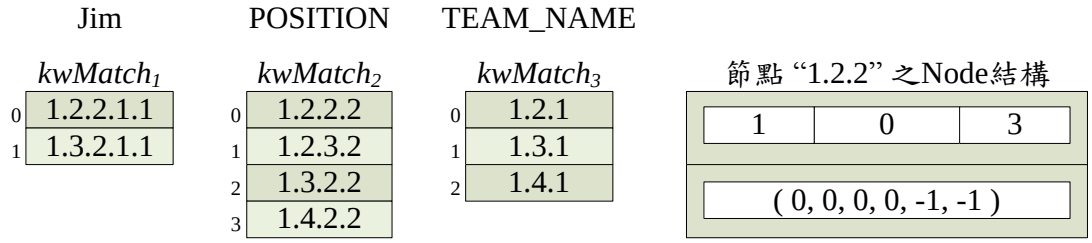

圖 3.6 節點 “1.2.2” 之 Node 結構狀態

[範例 3.3]：考慮以查詢句 *Q1* 進行查詢，得 SLCA 為{ 1.2, 1.3 }，針對 SLCA “1.2” 到其下對應節點 “1.2.2.1.1” 之路徑上的節點 “1.2.2” ，可以利用如圖 3.6 所示

的 Node 結構記錄其相關資訊，其 *match* = 1，*positon* = 0，*level* = 3，代表其為 $kwMatch_1$ 中，記錄於位置 0 的對應節點 (即 1.2.2.1.1) 位於第 3 層的祖先。其 *dmatch_range* 為 (0, 0 , 0, 1 , -1, -1) ，記錄著其下所有對應節點在各個 kwMatch 中開始出現以及最後出現的位置。

3.4 TDPruneMatches 模組

在 *GroupMatchRange* 將資料分群之後，*TDPruneMatches* 模組接著將每個群建立答案樹。其基本精神為，由每一群的 SLCA 節點開始以由上而下(top-down)的方式，利用該節點的 *dmatch_range* 所限定的反轉索引區間內的資訊，產生該節點的所有有可能成為貢獻節點的孩子節點，並在過程中將確認為貢獻節點的節點直接輸出。詳細地說，由於我們的目的在輸出 SLCA 到相關對應節點之路徑上的所有節點，因此這些節點必定是某一對應節點的祖先節點，或是該節點本身即為對應節點。所以，在由上而下地建立樹的過程中，針對每個節點，我們僅需考慮其所有為某一對應節點之祖先的孩子節點，而不需考慮其他孩子節點。而對於這些有可能成為貢獻節點的孩子節點，我們可以利用反轉索引中所記錄的資訊來產生。再配合用以限定反轉索引區間的 *dmatch_range* 來進一步減少計算量。

根據定義 2.7 所定義的輸出結果，我們必需在檢查完所有節點是否為貢獻節點後，將所有祖先皆為貢獻節點的對應節點輸出。這樣的方式使得我們在輸出節點時，必需針對所有對應節點及其與根節點路徑上的所有節點進行額外的檢查。因此，我們介紹以下的定理來避免此步驟：

[定理 3.2]：對於一棵 XML 樹 D、查詢句 Q 及節點 n、n_a，n_a 為一個 Q 中所有關鍵字的 SLCA，n 為 n_a 的子孫節點。若 n 為一個貢獻節點且 n 不為對應節點，則 n 必至少有一個孩子節點為貢獻節點。

[證明]：考慮 n 所有孩子節點 $n_1,\ldots, n_k$ 的子孫對應節點集合 (descendant match) ，

以 $dmatch(n_1),\ldots, dmatch(n_k)$表示。首先，因為 $n_1,\ldots, n_k$ 皆為 n 的孩子節點，所以 $n_1,\ldots, n_k$ 皆滿足定義 2.5 第一項。接著，若 $n_1,\ldots, n_k$ 此中存在任一個節點 n_a，使得 $dmatch(n_a)$不與其他節點的子孫對應節點集合有真包含關係，則 n_a 滿足定義 2.5 第二項，即 n_a 為貢獻節點。若 $dmatch(n_1),\ldots, dmatch(n_k)$皆互相有真包含關係，不失一般性假設 $dmatch(n_1) \subset dmatch(n_2) \subset \ldots \subset dmatch(n_k)$，則 n_k 滿足定義 2.5。■

[定理 3.3]：對於一棵 XML 樹 D、查詢句 Q 及節點 n， n 為一個 Q 中所有關鍵字的 SLCA，考慮 n 的任一子孫節點 n_d，若 n 與 n_d 路徑之上所有節點皆為貢獻節點 (包括 n_d) ，則必滿足以下兩種條件之一：

- n_d 為一個相關對應節點 (relevant match)。
- 存在一個對應節點 n_{dk}，使得 n_{dk} 為 n_d 的子孫節點，且 n_{dk} 與 n_d 路徑上的所有節點皆為貢獻節點，即 n_{dk} 為一相關對應節點。

[證明]：若 n_d 為對應節點，因為 n 與 n_d 路徑之上所有節點皆為貢獻節點，則 n_d 為一相關對應節點，命題成立。若 n_d 不為對應節點，由定理 3.2 與我們可知必存在一個為其孩子節點，且為貢獻節點的節點 n_{d1}。仿照前述動作，若 n_{d1} 不為相關對應節點，則必存在一個為 n_{d1} 的孩子節點，且為貢獻節點的節點 n_{d2}。假設每次產生出的節點 $n_{d1}, n_{d2}, \ldots$皆不為相關對應節點，則 n_d 有無限多個子孫節點，與事實矛盾。因此所有產生出的節點 $n_{d1}, n_{d2}, \ldots$中，必有一節點為相關對應節點，命題亦成立。■

當我們可以確認在某一個 SLCA 節點 n 下的任一節點 n_d 為貢獻節點，且 n 與 n_d 路徑上的所有節點皆為貢獻節點時，由於定理 3.3 的關係，我們可得知 n_d 必為最後答案樹所包含的一個節點。如此，在由上而下建立答案樹的過程中，由於樹中所有的節點皆為貢獻節點，當我們確認任何一個欲檢查的節點為貢獻節點時，便可直接將其輸出。

演算法名稱：TDPruneMatches

輸　　入：Node *n*

變　　數：*n_child_cnt*：記錄 *n* 的孩子節點個數

n_c：新產生之孩子節點

w：查詢句中關鍵字之個數

n_c*_dmatch*：記錄 n_c 之子孫對應關鍵字集合狀態的一個二進位整數

tmplist：所有有相同子孫對應關鍵字集合狀態的節點串列

tmplist_record：依序記錄每個節點放入哪一個 *tmplist* 的整數陣列

is_contributor：記錄目前節點是否為貢獻節點的布林值

is_contributor_list：記錄該 *tmp_list* 所存放之節點是否為貢獻節點

說　　明：輸出 *n* 後，產生 *n* 的所有為貢獻節點的孩子節點，並對這些節點遞迴地呼叫此函式藉以輸出所有祖先皆為貢獻節點之節點。

```
L01  output DeweyId(n)
L02  n_child_cnt = 0
L03  for n_c ← FindNextChild(n); n_c ≠ Null; n_c ← FindNextChild(n) do
L04    for int i ← 0; i < w; i++ do
L05      if n_c.dmatch_range[ 2 × i ] ≠ -1 then
L06        set the (i-1)^th bit of n_c_dmatch to 1
L07    append n_c into tmp_list[num(n_c_dmatch)] //num is the function converting
                                                   a binary number to a decimal number
L08    tmplist_record[n_child_cnt] ← num(n_c_dmatch)
L09    n_child_cnt++
L10  if n_child_cnt = 0 then do nothing
L11  else if n_child_cnt = 1 then
L12    v ← get the only node from tmplist[n_c_dmatch]
```

```
L13      TDPruneMatches(v)
L14   else if n_child_cnt ≦ 2^w
L15      for each int i in tmplist_record do
L16         is_contributor ← true
L17         for int j ← i + 1; j < 2^w; j++ do
L18         if tmp_list[j].size > 0 && AND(i, j) = i then     // i.e. j subsumes i
L19            is_contributor ← false
L20            break;
L21         if is_contributor = true then
L22            v ← get a node from tmplist[i]
L23            TDPruneMatches(v)
L24   else     // n_child_cnt > 2^w
L25      for int i ← 0; i < 2^w; i++ do
L26         is_contributor_list[i] ← true
L27         for int j ← i+1; j < 2^w; j++ do
L28            if tmp_list[i] is a proper set of tmp_list[j] then
L29               is_contributor_list[i] ← false
L30      for each int i in tmplist_record do
L31         if is_contributor_list[i] = true then
L32            v ← get a node from tmplist[i]
L33            TDPruneMatches(v)
```

圖 3.7 TDPruneMatches 演算法

TDPruneMatches 模組利用遞迴的方式從 SLCA 節點開始由上而下找出所有父親亦為貢獻節點的貢獻節點，詳細演算法如圖 3.7 所示。L01 將節點本身的杜威編碼輸出。L03 ~ L09 的迴圈不斷呼叫 *FindNextChild* 函式來取出目前節點 *n* 的

孩子節點，直到 n 沒有其他孩子節點為止。*FindNextChild* 函式利用 n 之 *dmatch_range* 找出下一個為對應節點之祖先的孩子節點 n_c，並在計算 n_c 之 *dmatch_range* 後，調整 n 之 *dmatch_range*，和排除與 n_c 之 *dmatch_range* 重複之部份。藉由調整 n 的 *dmatch_range*，使得再次對 n 呼叫 *FindNextChild* 函式時，可以找到下一個為對應節點的孩子節點。如此，我們可以得到 n 的所有有可能成為貢獻節點的孩子節點。我們將 *FindNextChild* 演算法置於本節末，並對其進行詳細說明。在此，我們先利用範例 3.4 說明該演算法產生的效果。

編號	關鍵字	資訊需求
Q2	MLB、James、POSITION	MLB 聯盟中，所有名叫 James 的球員之守備位置(POSITION)

表 3.1 查詢句 *Q2*

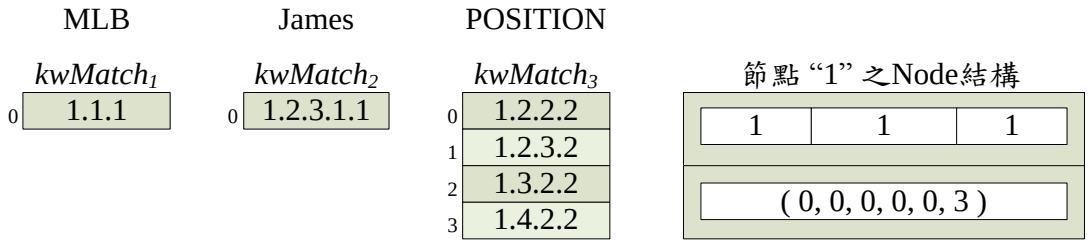

圖 3.8 *Q2* 之 3 個 *kwMatch* 及 SLCA 節點 “1” 之 Node 結構

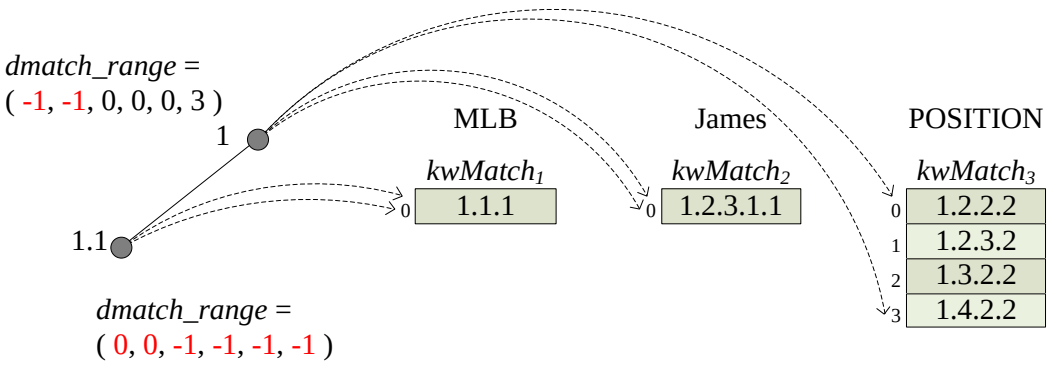

圖 3.9 節點 “1” 執行第一次 *FindNextChild* 函式後之 *dmatch_range* 狀態

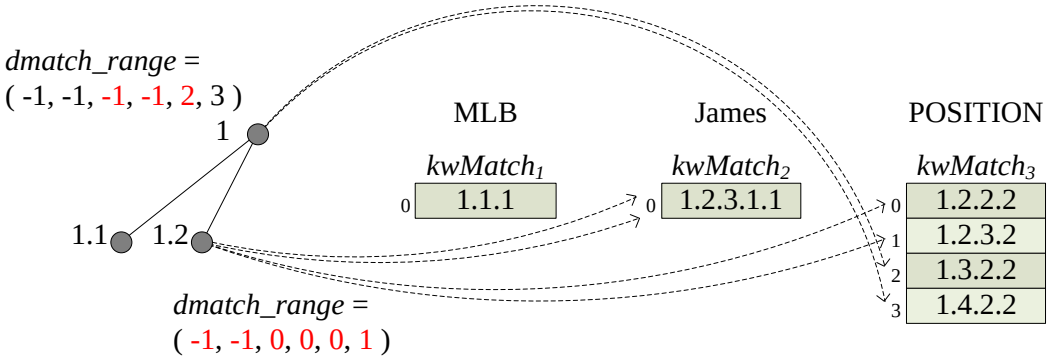

圖 3.10 節點“1”執行第二次 *FindNextChild* 函式後之 *dmatch_range* 狀態

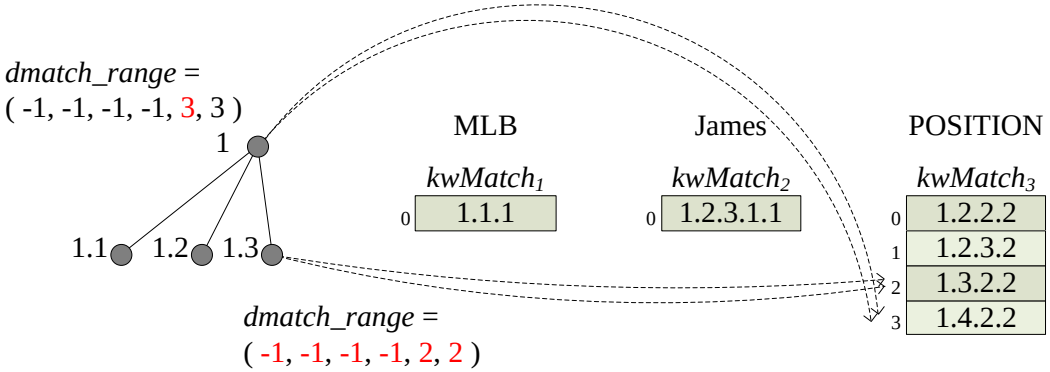

圖 3.11 節點“1”執行第三次 *FindNextChild* 函式後之 *dmatch_range* 狀態

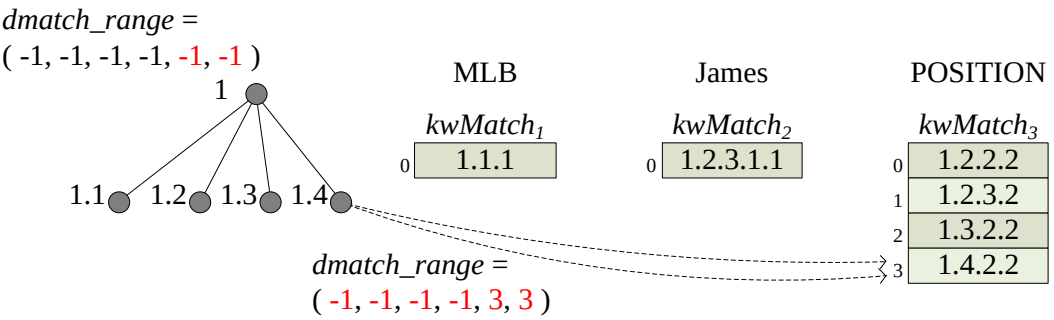

圖 3.12 節點“1”執行第四次 *FindNextChild* 函式後之 *dmatch_range* 狀態

[範例 3.4]：考慮以表 3.1 中的查詢句 *Q2* 進行查詢，得到如圖 3.8 所示之 *kwMatch* 以及唯一之 SLCA 節點“1”。輸出節點“1”之杜威編碼後，反覆呼叫 *FindNextChild* 函式以找出其所有孩子。一開始，節點“1”之 *dmatch_range* 為 (0, 0 , 0, 0 , 0, 3) 。執行第一次 *FindNextChild* 函式後，得到孩子節點“1.1”，其 *dmatch_range* 被設定為 (0, 0, -1, -1, -1, -1) ，表示在 $kwMatch_1$ 中，從位置 0 開始到位置 0 結束所記錄的元素皆為 1.1 的子孫節點，而在 $kwMatch_2$ 與 $kwMatch_3$

中並無祖先為“1.1”的對應節點。而原先節點“1”的 *dmatch_range*，則在排除剛剛找到的節點“1.1”的 *dmatch_range* 後，被更新為 (-1, -1, 0, 0, 0, 3)，其中第 3 至第 6 個元素不為 -1，表示在 $kwMatch_2$ 與 $kwMatch_3$ 中尚記錄著能產生其他孩子節點的資料。如圖 3.9 所示，圖中的虛線代表該節點的 *dmatch_range* 中，非-1 的值所記錄的 kwMatch 之位置。仿上述過程，在執行第二次 *FindNextChild* 函式後，產生節點“1.2”，其 *dmatch_range* 為 (-1, -1, 0, 0, 0, 1)，而節點“1”的 *dmatch_range* 則被更新為 (-1, -1, -1, -1, 2, 3)，如圖 3.10 所示。執行第三次 *FindNextChild* 函式後，產生節點“1.3”，其 *dmatch_range* 為 (-1, -1, -1, -1, 2, 2)，而節點“1”的 *dmatch_range* 則被更新為 (-1, -1, -1, -1, 3, 3)，如圖 3.11 所示。執行第四次 *FindNextChild* 函式後，產生節點“1.4”，其 *dmatch_range* 為 (-1, -1, -1, -1, 3, 3)，而節點“1”的 *dmatch_range* 則被更新為 (-1, -1, -1, -1, -1, -1)，如圖 3.12 所示。最後，因為節點“1”之 *dmatch_range* 被更新為 (-1, -1, -1, -1, -1, -1)，所以在執行第五次 *FindNextChild* 函式時，無法獲得任何孩子節點，於是便停止對節點“1”呼叫 *FindNextChild* 函式。至此，已取出節點“1”之所有有可能成貢獻節點之孩子節點“1.1”、“1.2”、“1.3”和“1.4”。

接著，我們必須檢查這些取出的節點是否為貢獻節點，與 MaxMatch 不同的是，MaxMatch 在建立完樹狀結構後，對節點執行兩次追蹤，第一次追蹤時，更新所有節點的 *dmatch* 與其父親節點的 *dmatch_set*，在第二次追蹤時，對所有父親為貢獻節點的節點進行檢查是否為貢獻節點的動作，檢查時可以直接利用前次追蹤所產生的父親節點之 *dmatch_set* 來進行判斷。相對而言，TDPrune 僅執行一次追蹤，也就是我們採取的策略為在產生出某一節點的所有孩子節點後，立即對所有孩子節點進行是否為貢獻節點的判斷。詳細地說，對某一節點 *n* 來說，我們在 L03 ~ L09 的迴圈內不斷重複執行以下動作直到無法取出任何孩子節點為止：我們先產生出一個 *n* 的孩子節點 n_c，並於 L04 ~ L06 計算 n_c*.dmatch*，n_c*.dmatch* 為一大小為關鍵字個數的布林陣列，若 n_c*.dmatch_range* 陣列中位於位置 $2 \times (i$ -

1）的數字不為 -1，表示 n_c 或 n_c 的子孫節點對應查詢句中的第 *i* 個關鍵字，則將 n_c*_dmatch* 中由右至左的第 *i* 個布林值設為 1，如此我們可以得到與 n_c 之子孫對應關鍵字集合 (descendant match) 所相呼應的一個獨特的數字 *num*(n_c*_dmatch*)，接著再於 L06 將 n_c 放至 Node 串列 *tmplist*[n_c*_dmatch*]內，便可以確保任一個 *tmplist* 內的所有節點皆有著相同的子孫對應關鍵字集合。另外，因為我們將節點放入不同的 *tmplist* 之中，所以在取出節點時，若僅依序由各個 *tmplist* 取出其中所有節點，則會破壞其原本放入的順序。因此我們於 L07 利用 *tmplist_record* 此一整數陣列來記錄每次產生的節點放入之 *tmplist* 的編號中，在後續將節點從 *tmplist* 取出時，僅需依照 *tmplist_record* 所記錄的 *tmplist* 編號逐一取出節點，便能保持原本的順序。

接下來，我們於 L10 ~ L33，根據 *n* 的孩子節點的個數，選擇適合的方式來判斷這些節點是否為貢獻節點，若 *n* 僅有一個孩子節點，我們便在 L12 ~ L13 直接予以輸出。若孩子節點的個數不超過 2^w，則採用第一種方式 (L14 ~L23) ，反之則採用第二種方式 (L24 ~L33) 。第一種方式針對每個所產生的孩子節點逐一進行檢查，根據 *tmplist_record* 所記錄的資訊，按照節點產生的順序檢查其子孫對應關鍵字集合是否被其他兄弟節點之子孫對應關鍵字集合所真包含，藉以判斷該節點是否為貢獻節點。若是，則對其遞迴地呼叫 *TDPruneMatches* 模組。這也基本上是 MaxMatch 所使用之方法。

第二種方式對剛剛放置孩子節點的所有 Node 串列 *tmplist*[]進行其所包含的節點是否為貢獻節點的判斷。與第一種方法不同的是，因為我們將所有有著相同子孫對應關鍵字集合的孩子節點放置於相同的 Node 串列中，所以這裡進行比較的單位是 Node 串列，其至多 2^w 個串列。因此，我們根據節點個數和 2^w 的數值大小來決用採用第一種方法還是第二種方法。在 L30 ~ L33 中，利用 *tmplist_record* 配合記錄檢查結果之 *is_contributor_list*，依照節點放入 *tmplist* 的順序來逐一判斷

該節點是否為一貢獻節點，若是，則將其取出並對其遞迴地呼叫 *TDPruneMatches* 模組。

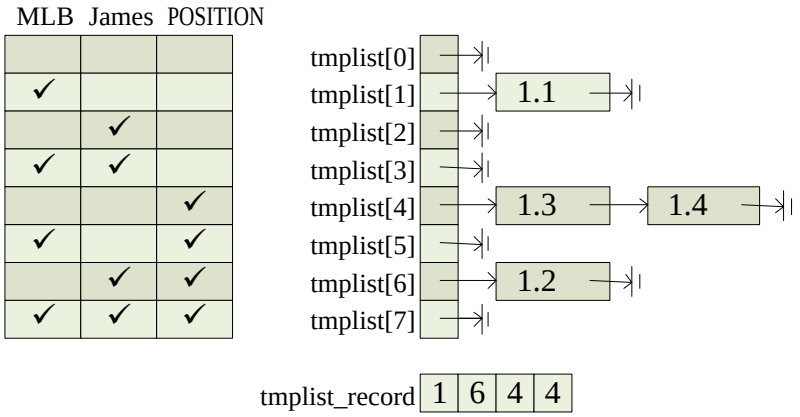

圖 3.13 *tmplist* 與其中節點之子孫對應關鍵字集合

[範例 3.5]：考慮以查詢句 *Q2* 進行查詢，由於 *Q2* 中有 3 個關鍵字，因此對任一節點來說，其子孫對應關鍵字集合至多有 2^3 種不同的狀態，我們以 *tmplist*[0] ~ *tmplist*[7] 來分別記錄包含不同狀態之子孫對應關鍵字集合的節點。首先，對唯一的最低最小共同祖先 “1” 執行 TDPruneMatches 模組，我們找出其下具有關鍵字的所有孩子節點 “1.1” 、“1.2” 、 “1.3” 、 “1.4”， 對於節點 “1.1” ，其 *dmatch_range* 為 (0, 0, -1, -1, -1, -1)，因為其記錄 $kwmatch_1$ 開始出現位置 (即 *dmatch_range*[2×(1-1)]) 的值為 0 ≠ -1，我們可以知道其下包含第 1 個關鍵字，所以將其放至 *tmp_list*[2^{1-1}]，即 *tmp_list*[1]中，並將 *tmplist_record* 更新為 (1)。對於節點 “1.2” ，其 *dmatch_range* 為 (-1, -1, 0, 0, 0, 1)，因為其記錄 $kwmatch_2$ 及 $kwmatch_3$ 的開始出現位置皆為 0 ≠ -1，我們可以知道其下包含第 2 個關鍵字和第 3 個關鍵字，所以將其放至 *tmp_list*[$2^{2-1} + 2^{3-1}$]，即 *tmp_list*[6]中，並將 *tmplist_record* 更新為 (1, 6)。對於節點 “1.3” ，其 *dmatch_range* 為 (-1, -1, -1, -1, 2, 2) ，因為其記錄 $kwmatch_3$ 開始出現位置的值為 2 ≠ -1，我們可以知道其下包含第 3 個關鍵字，所以將其放至 *tmp_list*[2^{3-1}] 即 *tmp_list*[4] 中，並將 *tmplist_record* 更新為 (1, 6, 4)。對於節點 “1.4” ，其 *dmatch_range* 為 (-1, -1, -1, -1, 3, 3) ，因為其記錄 $kwmatch_3$ 開始出現位置的值為 3 ≠ -1，我們可以知道其

下包含第 3 個關鍵字，所以將其放至 *tmp_list*[2^{3-1}]即 *tmp_list*[4]中，並將 *tmplist_record* 更新為 (1, 6, 4, 4)。最後所有 *tmplist* 的狀態如圖 3.13 右方所示，我們並在圖中左方以表格的方式表示該 *tmplist* 所包含的節點之子孫對應關鍵字集合。

接下來，由於 *tmp_list* 中共有 4 個有可能成為貢獻節點之孩子節點，少於 8 個 (即 2^{w})，因此選用第一種方式來判別其是否為貢獻節點，*tmplist_record* 中的第一個數字為 1，代表該節點之子孫對應關鍵字集合為 {MLB} ，所有子孫對應關鍵字集合真包含 {MLB} 的節點所放置的 *tmplist* (*tmplist*[3]，*tmplist*[5]，*tmplist*[7]) 皆為空，因此該節點 “1.1” 為貢獻節點。*tmplist_record* 中的第二個數字為 6，代表該節點之子孫對應關鍵字集合為 {James, POSITION}，子孫對應關鍵字集合真包含{James, POSITION}的節點所放置的 *tmplist*[7]為空，因此該節點 “1.2” 為貢獻節點。*tmplist_record* 中的第三個數字為 4，代表該節點之子孫對應關鍵字集合為 {POSITION}，而放置子孫對應關鍵字集合為 {James, POSITION} 的節點之 *tmplist*[6]不為空，代表節點 “1.3” 不為貢獻節點，*tmplist_record* 中的第四個數字為 4，判斷過程如上，得到節點 “1.4” 不為貢獻節點。此外，我們另行對此範例以第二種方式來判斷，藉以演示第二種方式實際執行的方法如下：考慮所有不為空的 *tmp_list*，*tmp_list*[1]、*tmp_list*[4]、和 *tmp_list*[6]，因為在 *tmp_list*[4]中的節點對應第 2 個關鍵字，*tmp_list*[6]中的節點對應第 2 及第 3 個關鍵字，*tmp_list*[4]中的節點對應的關鍵字為 *tmp_list*[6]中的節點對應的關鍵字所包含，所以 *tmp_list*[4]中的節點不為貢獻節點。最後我們得到所有在 *tmp_list*[1]、和 *tmp_list*[6]中的所有節點皆為貢獻節點。

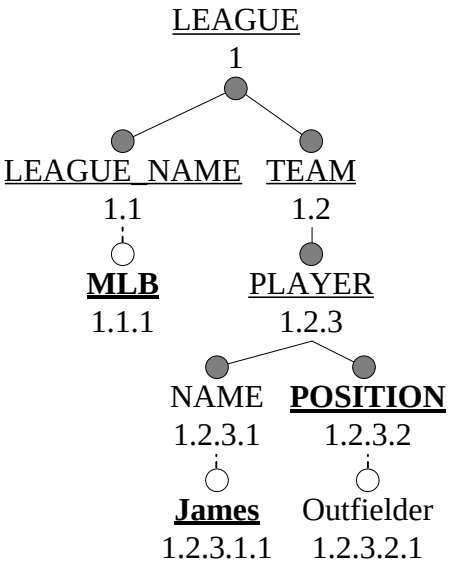

圖 3.14 *Q2* 之輸出結果

[範例 3.6]：考慮以查詢句 *Q2* 進行查詢，對唯一的最低最小共同祖先“1”執行 TDPruneMatches 模組；輸出節點本身，即“1”後，我們找出其所有為貢獻節點的孩子節點“1.1”、“1.2”，並分別對之遞迴地進行呼叫 TDPruneMatches 模組，最後產生如圖 3.14 之輸出結果。

演算法名稱：FindNextChild

輸　　入：Node *n*

輸　　出：*n* 所記錄之 *dmatch_range* 中，第一個為某對應節點的祖先，且為 *n* 之孩子節點的節點

變　　數：*least_lbl* (least_label) ：*n* 在 *dmatch_range* 的所有孩子節點中，最後一位編碼最小的節點的最後一位編碼。

n_c：新產生之孩子節點

w：查詢句中關鍵字之個數

說　　明：回傳 *n* 的下一個編碼最小的孩子節點，並調整 *n* 之 *dmatch_range*，排除與上述節點之 *dmatch_range* 重複的部份。

```
L01   least_lbl ← 0
L02   n_c.level ← n.level + 1
L03   for int i ← 0; i < w; i++ do
L04     if dmatch_range[2*i] != -1
L05       if least_lbl=0 or least_lbl > kwMatch_i.GetLabel(n.dmatch_range[2*i],
          n_c.level) // kwMatch_i.GetLabel(pos, lvl) is the function to retrieve the
          label of level lvl of the node at the position pos of kwMatch_i.
L06         least_lbl ← kwMatch_i.GetLabel(n.dmatch_range[2*i], n_c.level)
L07   if least_lbl = 0
L08     return Null
L09   else
L10     for i ← 0; i < w; i++ do
L11       if n.dmatch_range[2*i] != -1
L12         if least_lbl = kwMatch_i.GetLabel(n.dmatch_range[2*i], n_c.level)
L13           n_c.match ← i
```

```
L14             n_c.position ← n.dmatch_range[2*i]
L15             n_c.dmatch_range[2*i] ← n.dmatch_range[2*i]
L16             if kwMatch_i.GetLabel(n_c.dmatch_range[2*i], n_c.level) =
                   kwMatch_i.GetLabel(n.dmatch_range[2*i+1], n_c.level)
L17                n.dmatch_range[2*i] ← -1
L18                n_c.dmatch_range[2*i+1] ← n.dmatch_range[2*i+1]
L19             else
L20                for int j←n.dmatch_range[2*i]; j<=n.dmatch_range[2*i+1]; j++ do
L21                   if kwMatch_i.GetLabel(j, n_c.level) != least_lbl
L22                      n.dmatch_range[2*i] ← j
L23                      n_c.dmatch_range[2*i+1] ← j-1
L24                      break
L25     return n_c
```

圖 3.15 FindNextChild 演算法

接下來，我們簡略說明 *FindNextChild* 演算法如下：依據前述杜威編碼的特性，每一個節點的編碼皆記錄著其祖先的編碼。假設目前節點 *n* 的深度為 *n.level*，則 *n* 的所有孩子節點為編碼前 *n.level* 位數與 *n* 相同，且編碼總位數為 *n.level* + 1 的節點。因此，我們只要檢查所有 *kwMatch* 裡所記錄的對應節點，找出符合上述條件且編碼最後一位最小的節點，便可得到目前 *dmatch_range* 裡為 *v* 的第一個孩子節點 n_c。至於調整 *n* 與 n_c 的 *dmatch_range* 的方法，為將 *n* 中符合上述條件的 *dmatch_range* 排除，並新增至 n_c 的 *dmatch_range* 中。而若 n_c 不是 *kwMatch* 中任一節點的祖先時，則將其在對應該 *kwMatch* 之 *dmatch_range* 的開始出現位置與最後出現位置設為 -1。

完整的演算法如圖 3.15 所示。首先，在 L01 我們將用以記錄孩子節點之編碼最後一位的變數 *least_lbl* 設為 0。在 L03 ~ L06 的迴圈中，我們對分別記錄著對應到不同關鍵字的節點之 $kwMatch_i$，逐一檢查其位於位置 *dmatch_range*[2**i*] 之元素的編碼中第 n_c.*level* 層的數字，以找出最小的數字，即為 *n* 的所有孩子節點中，最後一位編碼最小的節點 n_c 的最後一位編碼之值。接著於 L07 我們則利用變數 *least_lbl* 的值來檢查前項動作是否有找到最小的編碼，若無 (以 *least_lbl* = 0 代表) ，則表示節點 *n* 沒有任何其下表示對應節點的孩子節點，回傳 *Null*。若有，則繼續於 L10 ~ L24 進行調整 *n* 與 n_c 的 *dmatch_range* 之動作。

我們在對 L10 ~ L24 的迴圈中，先對每一個 $kwMatch_i$ 檢查其位置 *n.dmtach_range*[2**i*]所記錄的節點是否為 n_c 的子孫節點。若是，則於 L13 ~ L24 調整 *n* 與 n_c 的 *dmatch_range*。首先在 L13 ~ L14 依 *n* 的資訊來設定用以還原 n_c 的杜威編碼的資訊，即變數 *match* 及 *position*，接著在 L15 設定 $kwMatch_i$ 中，n_c 的子孫節點開始出現的位置。L16 則檢查 $kwMatch_i$ 中，*n* 的子孫節點最後出現的位置上之值是否與 n_c 的子孫節點開始出現之位置上的值相同。若是，則代表 n_c 為 *n* 唯一的孩子節點，即 n_c 的子孫節點最後出現的位置與 *n* 的子孫節點最後出現的位置相同，於 L17 ~ L18 進行相關設定。注意到，因為我們可以直接利用 *dmatch_range*[2**i*] 是否為 -1 來判斷節點是否還有孩子節點，而不需檢查 *dmatch_range*[2**i*+1] (如 L04、L11)，因此若 n 不會再有其他的孩子節點，我們僅需設定其 *dmatch_range*[2**i*] 之值為 -1 (L17) ，而不需另行設定其 *dmatch_range*[2**i*+1]之值。接著，若 *n* 的子孫節點最後出現的位置上之值與 n_c 的子孫節點開始出現之位置上的值不同，代表除了 n_c 以外，*n* 尚有其他孩子節點，則利用 L20 ~ L24 的迴圈，從 *n* 的子孫節點開始出現的位置開始，逐一檢查該節點是否亦為 n_c 的子孫節點，直到找到第一個為 *n* 的子孫節點但卻不是 n_c 的子孫節點為止，並於 L22 ~ L23 進行相關設定。最後，於 L25 將已正確設定 *dmatch_range* 的 n_c 回傳。

3.5 TDPrune 效率分析

我們定義下列表示式，並針對 MaxMatch 系統與 TDPruneMatches 系統中相異的子模組進行分析比較，分別為 *groupMatches* 和 *GroupMatchRange*，以及 *pruneMatches* 模組和 *TDpruneMatches* 模組：

時間複雜度：

M_{min}：最小單一關鍵字的對應節點的數量

M_{max}：最大單一關鍵字的對應節點的數量

M：查詢句中所有關鍵字的所有對應節點之個數

|D|：XML 文件之節點個數

d：XML 文件之最大深度

w：查詢句中的關鍵字個數

MaxMatch 與 TDPrune 分別採用 *groupMatches* 與 *GroupMatchRange* 進行將節點分群的工作：

- MaxMatch.*groupMatches* 模組： $O(M \times \log w + M \times d)$
- TDPrune.*GroupMatchRange* 模組： $O(M \times d)$

兩個模組同樣必需完整的對反轉索引內記錄之每一個對應節點與 SLCA 進行判別是否為其子孫節點的動作，而每一次判別花費 $O(d)$的計算量，因此總計算量為 $O(M \times d)$。除此之外，由於 *groupMatches* 模組需先對所有節點進行混合排序，因此另需外費 $O(M \times \log w)$之計算量。

MaxMatch 與 TDPrune 分別採用 *pruneMatches* 與 *TDPruneMatches* 進行建立答案樹的工作：

- MaxMatch.*pruneMatches* 模組：
  - 第一次追蹤： $O(\min\{|D|, M \times d\})$。
  - 第二次追蹤： $O(\min\{|D|, M \times d\} \times 2^w)$
  - 合計：$O(\min\{|D|, M \times d\} \times 2^w)$
- TDPrune.*TDPruneMatches* 模組： $O(\min\{|D|, M \times d\} \times 2^w)$

我們分析 *TDPruneMatches* 的運算量如下：*TDpruneMatches* 模組使用 *FindNextChild* 函式來產生孩子節點，我們將其分為兩個部份，第一個部份為 L01 ~ L06，包含第一個 for 迴圈，用以找出所有孩子節點中，最後一位編碼最小的節點的最後一位編碼，上界為 O(w)。第二部份為 L07 ~ L25，包含第二個與第三個 for 迴圈所構成的一個巢狀迴圈，用以判斷 *dmatch_range* 的範圍，其計算量為該節點之 *dmatch_range* 所限定的區間大小，即 O(M)。因此，單一次呼叫 *FindNextChild* 計算量之上界為 O($w + M$)。注意到，對位於同一層的所有節點而言，由於位於同一層的所有節點之 *dmatch_range* 所限定的區間大小的總和至多為 M，因此對這些節點呼叫 *FindNextChild* 以找出下一層的所有 k 個節點之總計算量為 O($k \times w + M$)而非 O($k \times w + k \times M$)。

同樣地，我們也將 *TDPruneMatches* 分為 L01 ~ L09 及 L10 ~ L33 兩個部份來討論。首先，每一次執行 *TDPruneMatches* 時，皆會執行 L03 ~ L09 的迴圈若干次以找出其孩子節點，因此我們可以知道 L03 ~ L09 之程式片段的總執行次數之上界為 SLCA 以下之節點的總數量，即 $O(\ min\{|D|, M \times d\}\ )$，針對其中的 *FindFirstChild* 函式，其第一部份的計算量為 $O(w)$乘上節點總數 $O(\ min\{|D|, M \times d\}\ )$，即 $O(\ min\{|D|, M \times d\} \times w)$，而第二部份我們採取逐層的方式計算其計算

量，同一層所有節點的 *dmatch_range* 之總和不超過 M，因此對任一層的所有節點而言，其 *FindFirstChild* 函式之第二部份的計算量為 $O(M)$，乘上層數 d 後便得到所有節點的計算量為 $O(M \times d)$，因此針對 *FindFirstChild* 函式，其總計算量亦為 $O(\ min\{|D|,\ M \times d\} \times w\ +\ M \times d) = O(\ min\{|D|,\ M \times d\} \times w)$。接下來，*TDPruneMatches* 中 L04 ~ L06 的迴圈之計算量為 $O(w)$，乘上計算次數 $O(\ min\{|D|,\ M \times d\}\ )$，得到其總計算量 $O(\ min\{|D|, M \times d\} \times w\ )$。因此 TDPruneMatches 的第一部份計算量為 $O(\ min\{|D|, M \times d\} \times w)$。

接下來，我們討論 *TDPruneMatches* 第二部份的計算量，若孩子節點的數量小於 1，花費 $O(1)$。若孩子節點的數量 c 不超過 2^w，則花費 $O(c \times 2^w) \leqq O(2^w \times 2^w) = O(2^{2w})$。若孩子節點的數量超過 2^w，則花費 $O(2^w \times 2^w) = O(2^{2w})$，也就是說，對任一個節點之$|n_c|$個孩子節點至多花費 $O(|n_c| \times 2^w)$，而所有節點之孩子數量的總合不超過 $min\{|D|, M \times d\}$，因此 *TDPruneMatches* 第二部份的總計算量為 $O(\ min\{|D|, M \times d\} \times 2^w)$。因此，*TDPruneMatches* 的總計算量為 $O(\ min\{|D|, M \times d\} \times w + min\{|D|, M \times d\} \times 2^w) = O(min\{|D|, M \times d\} \times 2^w)$，與 MaxMatch 之 *pruneMatches* 模組相同。

接下來，我們討論 MaxMatch 系統與 TDPrune 系統的總計算量，由於兩者在計算 SLCA 皆需花費 $O(M_{min}wd\log M_{max})$ ，而兩者在分群時的計算量皆不為支配性的，因此兩系統有著相同的總計算量 $O(M_{min}wd\log M_{max} + min\{|D|, M \times d\} \times 2^w\ )$ 。然而，在實際的查詢情況下，*TDPruneMatches* 卻有很大的機會可以在對樹中較少的節點進行檢查的動作的情況下產生輸出結果，進而減少整體的運算量以達到較佳的執行效率。我們將兩個系統於計算過程中會對之執行運算的節點分為以下三類，以說明減少計算量的原理：

v_c：所有對應節點及其祖先節點中，父親節點與本身皆為貢獻節點之節點

v_{pc}：所有對應節點及其祖先節點中，本身非貢獻節點但父親節點為貢獻節點

的節點

v_{pnc}：所有對應節點及其祖先節點中，父親節點非貢獻節點的節點

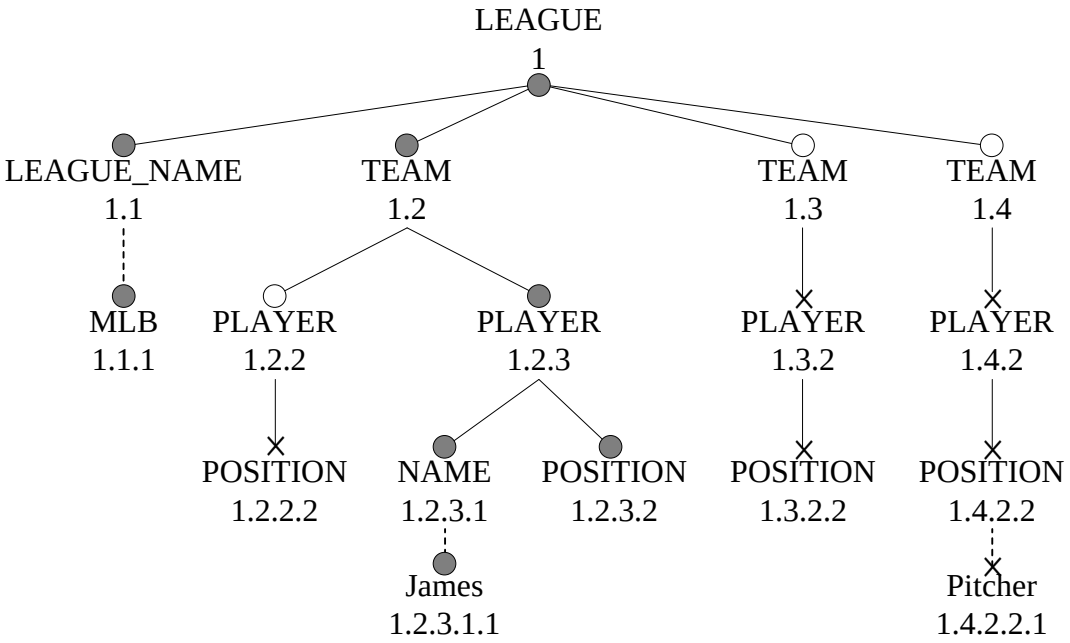

圖 3.16 *Q2* 中所有對應節點及其祖先節點

[範例 3.7]：考慮以表 3.1 中的查詢句 *Q2* 進行查詢，所有對應節點及其祖先節點圖 3.16 所示。其中 v_c、v_{pc}、與 v_{pnc} 分別以實心圓、空心圓以及叉叉表示。

MaxMatch 的 *pruneMatches* 與 TDPrune 的 *TDPruneMatches* 模組僅對父親為貢獻節點的節點進行判斷其是否為貢獻節點的動作。因此，同樣只需對於 v_c 與 v_{pc} 此兩類節點進行處理。然而，在判斷節點是否為貢獻節點時，MaxMatch 的 *pruneMatches* 模組必需利用第一次追蹤時，所更新 *dmatch* 及 *dmatchset* 來做為判斷的依據，而其第一次追蹤時，則會針對所有對應節點及其祖先節點，即 v_c、v_{pc}、以及 v_{pnc} 三類節點進行處理。因此，相較於 TDPrune 的 *TDPruneMatches* 模組，MaxMatch 的 *pruneMatches* 模組必需額外對 v_{pnc} 這類節點進行處理。

我們觀察 v_{pnc} 這類節點，在圖 3.16 中，1.3.2.2、1.4.2.2 這兩個屬於 v_{pnc} 的 POSITION 節點分別為不同的 TEAM 節點之下，某一個 PLAYER 節點的孩子節

點，其實際代表的意義則為某一個球隊中的某一個球員的守備位置。然而，在現實狀況下，一個球隊往往不止一個球員，而每一個球員皆會有著一個 POSITION 節點來記錄其守備位置。假設某一個不為貢獻節點的 TEAM 節點，其下共有 k 個 PLAYER 節點，便會產生 k 個為 v_{pnc} 的 POSITION 節點，進而對 MaxMatch 造成龐大的計算量。

此外，在檢查節點是否為貢獻節點方面，*TDPruneMatches* 在某些情況下亦可以以較少的運算量完成檢查。若一個節點的孩子數量 $c > 2^w$，相較於 *pruneMatches* 需花費 $O(c \times 2^w)$，*TDPruneMatches* 僅需花費 $O(2^w \times 2^w)$便可完成檢查。

空間複雜度：

MaxMatch 與 TDPrune 分別採用 *pruneMatches* 與 *TDPruneMatches* 進行建立答案樹的工作。*pruneMatches* 於第一次追蹤時，為所有節點建立了一個大小為 w 的 *dmatch* 布林陣列及一個大小為 2^w 的 *dmatchset*，第二次追蹤時僅使用一些配合檢查是否為貢獻節點的變數。*TDPruneMatches* 則進行最深為 d 層遞迴呼叫，每次呼叫需保存 2^w 個 *node* 串列資料及一些配合檢查是否為貢獻節點的變數。此外，針對所保存的 *node* 串列，其中包含的 *node* 數總共最多為 M 個，每個 *node* 記錄的資訊量為 $O(w)$，因此上界為 $O(d \times 2^w + M \times O(w)) = O(d \times 2^w + M \times w)$：

- MaxMatch. *pruneMatches* 模組：
  - 第一次追蹤： $O(\min\{|D|, M \times d\} \times 2^w)$。
  - 第二次追蹤： $O(1)$
  - 合計：$O(\min\{|D|, M \times d\} \times 2^w)$

◆ TDPrune.*TDPruneMatches* 模組： $O(d \times 2^w + M \times w)$

考慮 TDPrune.*TDPruneMatches* 的上界：$O(d \times 2^w + M \times w) < O(d \times 2^w + M \times 2^w) = O((M + d) \times 2^w) < O(\min\{|D|, M \times d\} \times 2^w)$，即相較於 *pruneMatches* 模組的上界 $O(\min\{|D|, M \times d\} \times 2^w)$，在空間複雜度上 TDPrune 亦有著較好的表現。

第四章 LevelPrune 演算法

在本章中，我們介紹 LevelPrune 演算法的系統架構。並如同上一章，以表 3.1 中的 *Q2* 對圖 2.1 中的 XML 樹進行查詢做為範例來演示 LevelPrune 執行的過程，最後並分析其時間與空間複雜度。

4.1 系統架構

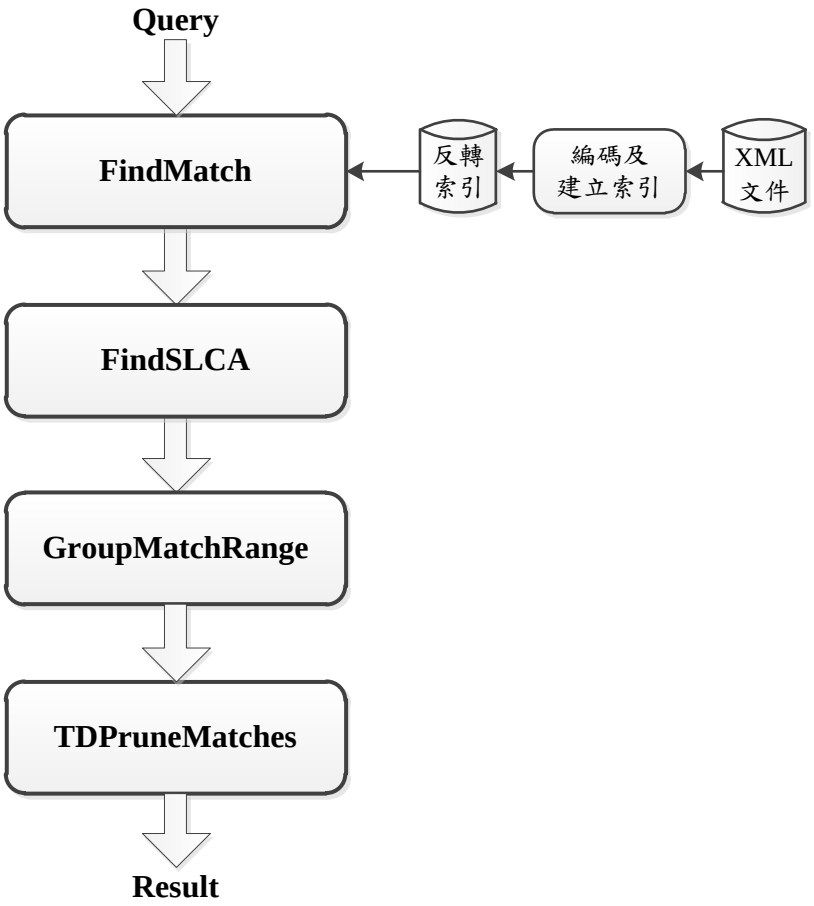

圖 4.1 LevelPrune 系統架構

LevelPrune 系統分為四個主要模組，如圖 4.1 所示。其中使用 2.3 章所介紹的 *FindMatch* 模組讀入索引資料後，以 *FindSLCA* 模組計算 SLCA，再由 3.3 章所介紹過的 *GroupMatchRange* 模組將 SLCA 節點與對應節點配對分群，最後由 *LevelPruneMatches* 模組計算並輸出查詢結果。由於前三個模組與 LevelPrune 相

同，因此我們僅介紹第四個模組，即 *LevelPruneMatches*。

4.2 LevelPruneMatches 模組

與前章所述之 TDPrune 系統的 *TDPruneMatches* 模組不同的是，針對所有位於第 *d* 層的節點 $n_1, \ldots, n_k$，*TDPruneMatches* 模組在產生 n_1 之所有為貢獻節點的孩子節點後，便以深度優先 (Depth-First Search) 的方式，直接對 n_1 位於第 *d*+1 層的孩子節點進行處理，過程中，第 *d* 層的其他節點 $n_2, \ldots, n_k$ 皆需被保存在記憶體中以供後續處理之用。同樣地，在處理到第 *d*+*e* 層時，*d*, *d*+1, … , *d*+*e*-1 層皆有可能有節點需要被保存以供後續之用。相反地，針對所有位於第 *d* 層的節點 $n_1, \ldots, n_k$，*LevelPruneMatches* 模組則是以廣度優先 (Breadth-First Search) 的方式，在 *n1*, … ,n_k 之所有為貢獻節點的孩子節點皆被產生後，才會逐一對上述動作所產生的節點進行處理，因此，在處理第 *d* 層的過程中，僅需記錄所有位於 *d*+1 層之新產生的節點，而當所有第 *d* 層的節點被處理完後，再對第 *d*+1 層中的所有節點進行如同上述的處理。

整體而言，*LevelPruneMatches* 模組對每一個 *GroupMatchRange* 模組所產生的群採取以下策略：由 SLCA 節點所在的層開始，逐層產生該層中的所有節點的孩子節點並加以判斷是否為貢獻節點，直到該層中沒有任何節點為止。若前述動作產生的節點為貢獻節點則將其加入至一個記錄下一層所有節點的 Node 陣列 *tmplist* 中，過程中如有發現對應節點，則該對應節點為相關對應節點，將其記錄。最後再依照所記錄的相關對應節點來建立答案樹。

演算法名稱：LevelPruneMatches
輸　　入：Node *n* (SLCA)
輸　　出：所有最小最低共同祖先到其下所有相關對應節點的路徑上的所有元素

變　　數：*nl_cur_level* (Node List of Current Level)：

記錄位於目前層數之所有節點的 Node 陣列

nl_next_level (Node List of Next Level)：：

記錄位於下一層之所有節點的 Node 陣列

n_c：新產生之孩子節點

w：查詢句中關鍵字之個數

n_c*_dmatch*：記錄 n_c 之子孫對應關鍵字集合狀態的一個數字

tmplist：所有有相同子孫對應關鍵字集合狀態的節點串列

tmplist_record：依序記錄每個節點放入哪一個 *tmplist* 的數字陣列

is_contributor：記錄目前節點是否為貢獻節點的布林值

is_contributor_list：記錄該 *tmp_list* 所存放之節點是否為貢獻節點

```
L01  append n into Node list nl_cur_level
L02  while nl_cur_level is not empty do
L03    for each node cur_n in nl_cur_level
L04      output DeweyId(cur_n)
L05      for n_c ← FindNextChild(cur_n); n_c≠Null; n_c ← FindNextChild(cur_n) do
L06        for int i ← 0; i < w; i++ do
L07          if n_c.dmatch_range[ 2 × i ] ≠ -1 then
L08            set the (i-1)^th bit of n_c_dmatch to 1
L09        append n_c into tmp_list[num(n_c_dmatch)]   //num is the function
                                          converting abinary number to a decimal number
L10        tmplist_record[n_child_cnt] ← num(n_c_dmatch)
L11        n_child_cnt++
L12      if n_child_cnt = 0 then do nothing
L13      else if n_child_cnt = 1 then
```

L14 get the only node from *tmplist*[n_c_*dmatch*] and append to *nl_next_level*)

L15 **else if** *n_child_cnt* $\leqq$ 2^k

L16 **for** each int *i* in *tmplist_record* **do**

L17 *is_contributor* $\leftarrow$ *true*

L18 **for** int *j* $\leftarrow$ *i* + 1; *j* < 2^k; *j*++ **do**

L19 **if** *tmp_list*[*j*].size > 0 && *AND*(*i*, *j*) = i **then** // i.e. *j* subsumes *i*

L20 *is_contributor* $\leftarrow$ *false*

L21 **break**;

L22 **if** *is_contributor* = *true* **then**

L23 get a node from *tmplist*[*i*] and append to *nl_next_level*)

L24 **else if** *n_child_cnt* > 2^k

L25 **for** int *i* $\leftarrow$ 0; *i* < 2^w; *i*++ **do**

L26 *is_contributor_list*[*i*] $\leftarrow$ *true*

L27 **for** int *j* $\leftarrow$ *i+1*; *j* < 2^w; *j*++ **do**

L28 **if** *tmp_list*[*i*] is a proper set of *tmp_list*[*j*] **then**

L29 *is_contributor_list*[*i*] $\leftarrow$ *false*

L30 **for** each int *i* in *tmplist_record* **do**

L31 **if** *is_contributor_list*[*i*] = *true* **then**

L32 get a node from *tmplist*[*i*] and append to *nl_next_level*)

L33 empty all *tmp_list*

L34 *nl_cur_level* $\leftarrow$ *nl_next_level*;

L35 *nl_next_level* $\leftarrow$ *Null*;

圖 4.2 LevelPruneMatches 演算法

LevelPrune 對每一個群執行 *LevelPruneMatches* 模組，演算法如圖 4.2 所示。該演算法利用 *nl_cur_level, nl_next_level* 這兩個 Node 串列來記錄每一層的貢獻節點及其下一層的所有貢獻節點。首先，在 L01 將該群的 SLCA 節點加入至 *nl_cur_level*，並由 L02 ~ L33 的迴圈開始逐層對所有節點進行處理，L03 由記錄著該層所有節點的 *nl_cur_level* 中取出一節點 *cur_n*。於 L04 將節點 *cur_n* 的杜威編碼輸出，L05 ~ L11 則利用 *FindNextChild* 函式以及迴圈來不斷取出 *cur_n* 之孩子節點來進行處理，直到 *cur_n* 沒有其他孩子節點為止。L12 ~ L32 採用如同 *TDPruneMatches* 的方式，根據 *cur_n* 之孩子節點的數量來選擇適合的方式進行處理，其處理的方式與 *TDPruneMatches* 不同的地方是，並不馬上針對節點進行遞迴處理，而是把節點放入 *nl_next_level* 中，等到該層所有其他的節點皆處理完後，再對下一層的節點進行處理。當對 *nl_cur_level* 中所有節點完成前述動作後，藉由 L34 ~ L35 將 *nl_next_level* 中所有節點移入 *nl_cur_level*，做為下一次執行 L02 ~ L33 之迴圈時檢查的目標。最後，程式會於無法產生任何為貢獻節點的孩子節點時停止執行。過程中，會以由上至下，由左至右的方式，輸出所有貢獻節點。

[範例 4.1]：考慮以查詢句 *Q2* 進行查詢，對唯一的最低最小共同祖先 “1” 執行 LevelPruneMatches 模組；將 “1” 加入 *nl_cur_level* 後，我們找出 *nl_cur_level* 中的所有節點之所有為貢獻節點的孩子節點 “1.1” 、“1.2”，並將其加入 *nl_next_level* 中。接下來，我們將 *nl_next_level* 中的所有節點移入 *nl_cur_level* 中，再重覆執行上述產生孩子節點的動作，藉以找出每一層中所有為貢獻節點的節點直到無法再產生任何為貢獻節點的孩子節點為止。最後，我們產生並輸出的節點依序如下，第一層：“1”，第二層：“1.1” 、“1.2”，第三層：“1.1.1” 、“1.2.3”，第四層：“1.2.3.1” 、“1.2.3.2”，第五層：“1.2.3.1.1”，如圖 3.14 所示之輸出結果。

4.3 LevelPrune 效率分析

我們於此節對 *LevelPruneMatches* 進行效率分析：

時間複雜度：

同樣地，我們將 *LevelPruneMatches* 分為兩個部份來討論其運算量，第一部份為 L05 ~ L11 用以產生孩子節點的迴圈，與 TDPruneMatches 相同，其總運算量為 O(min{|*D*|, *M* × *d*} × *w*)。而第二部份 L12 ~ L32 的總運算量為為節點數量 O(*min*{|*D*|, *M* × *d*})乘上判別每一個點的計算量 O(2^w)，即 *O*(*min*{|*D*|, *M* × *d*} × 2^w)。整體而言，其總運算量亦為 *O*(*min*{|*D*|, *M* × *d*} × 2^w)。

- TDPrune.*TDPruneMatches* 模組： *O*(*min*{|*D*|, *M* × *d* } × 2^w)
- LevelPrune.*LevelPruneMatches* 模組： *O*(*min*{|*D*|, *M* × *d* } × 2^w)

空間複雜度：

LevelPruneMatches 除使用 3 個至多包含 *M* 個 Node 的 Node 串列來儲存此層節點、和相關對應節點外，還使用 2^w 個 *tmp_list* 串列來暫存孩子節點，所有 *tmp_list* 串列中包含的 *node* 數總共最多為 *M* 個，而每個 *node* 記錄的資訊量為 O(*w*)。

- LevelPrune.*LevelPruneMatches* 模組：*O*(2^w + *M* × *w*)

之前我們曾分析 TDPrune 之 *TDPruneMatches* 模組所需空間為 *O*(*d* × 2^w + *M* × *w*)，所以可以看出來 LevelPrune 較省空間。

4.4 編碼行導向儲存

我們考慮 LevelPrune 的 *LevelPruneMatches* 模組，對一節點 *v*，假設其深度為 *d*(*v*)，在對 *v* 產生其孩子節點時，會檢查 *kwMatch* 中，*dmatch_range* 所限定的

區間，來產生深度為 $d(v)+1$ 的節點，而產生出的節點其編碼的前 $d(v)$層數字與 v 的編碼相同，僅第 $d(v)+1$ 的資訊需由 *kwMatch* 中取得。

由於前述特性，在反轉索引及 *kwMatch* 中我們可將節點的編碼改以行導向 (column-oriented) 的方式儲存：對於存有 n 個對應某關鍵字的節點的 *kwMatch*，其對應節點的最大深度為 d，則 *kwMatch* 為一大小為 $n \times d$ 的整數陣列，在記憶體中，我們將第 i 個對應節點的編碼中第 j 層的整數，記錄於 *kwmatch* 的第 $i + (j \times n)$ 個位置，即 *kwMatch* 中位置 $0 \sim n\text{-}1$ 的整數分別為此 n 個節點第 1 層的整數，位置 $n \sim 2n\text{-}1$ 的整數分別為此 n 個節點中第 2 層的整數，…，依此類推。每一列的第 $0 \sim (d - 1)$ 行記錄著一個節點第 $1 \sim d$ 層的編碼。

POSITION

$kwMatch_3$

0	1	2	2	2
1	1	2	3	2
2	1	3	2	2
3	1	4	2	2

記憶體存放狀態

… | 1 | 1 | 1 | 1 | 2 | 2 | 3 | 4 | 2 | 3 | 2 | 2 | 2 | 2 | 2 | 2 | …

圖 4.3 行導向儲存之 $kwMatch_3$

POSITION

$kwMatch_3$

0	1.2.2.2
1	1.2.3.2
2	1.3.2.2
3	1.4.2.2

記憶體存放狀態

… | 1 | 2 | 2 | 2 | 1 | 2 | 3 | 2 | 1 | 3 | 2 | 2 | 1 | 4 | 2 | 2 | …

圖 4.4 列導向儲存之 $kwMatch_3$

[範例 4.2]：考慮圖 2.1 的 XML 樹中，標籤為 “POSITION” 的節點共 4 個，其編碼分別為 “1.2.2.2” 、 “1.2.3.2” 、 “1.3.2.2” 、 “1.4.2.2”，最大深度為 4，若以行導向的方式儲存 *kwMatch*，則該 *kwMatch* 之內容為 { 1.1.1.1, 2.2.3.4, 2.3.2.2, 2.2.2.2 }，如圖 4.3 所示。若以列導向的方式儲存 *kwMatch*，則其內容如圖 4.4 所示。

行導向儲存的好處在於，在將索引讀取至 *kwMatch* 後，對所有同層的節點來說，其編碼在記憶體中會儲存在連續的位置。而 *LevelPruneMatches* 呼叫 *FindNextChild* 函式以產生節點 v 的所有孩子節點時，會對 *kwMatch* 中符合 *dmatch_range* 限制的所有節點取出第 $d(v)+1$ 層 (即第 $d(v)$ 行) 的編碼來進行比對，因為上述所有節點位於第 $d(v)+1$ 層的編碼在記憶體中皆存放於連續的位置，符合記憶體的空間區域性 (spatial locality)，因此可以降低從記憶體讀取至快取記憶體的次數，進而提升演算法的整體效率。

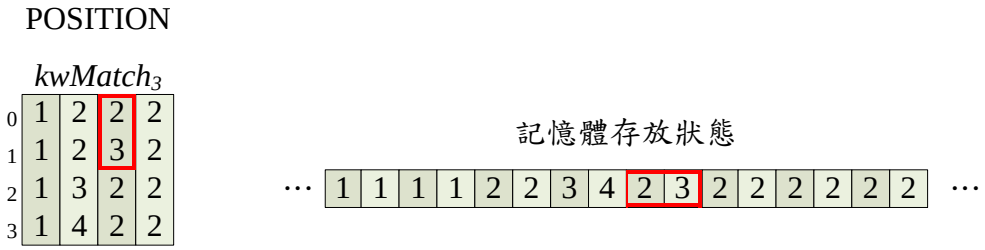

圖 4.5 節點 “1.2.2” 與節點 “1.2.3” 之記憶體位置

[範例 4.3]：考慮以查詢句 *Q2* 進行查詢，過程中，當 *LevelPruneMatches* 模組對節點 “1.2” 取出其所有孩子節點時，會逐次對 $kwMatch_1$ ~ $kwMatch_3$ 中所記錄符合 *dmatch_range* 限制的節點檢查其第 3 層的編碼。以存放關鍵字 POSITION 的對應節點的 $kwMatch_3$ 而言，其所記錄的節點個數為 4，內容為 { 1.1.1.1, 2.2.3.4, 2.3.2.2, 2.2.2.2 } ，此時限定 $kwMatch_3$ 範圍的 *dmatch_range* 為(0, 1)，因此 *LevelPruneMatches* 會取得 $kwMatch_3$ 中第 2 列的第 0 行以及第 2 列第 1 行的數字，在記憶體中，此兩個數字會分別位於第 8 以及第 9 這兩個於記憶體中相鄰的位置，分別為 “2” 及 “3” ，如圖 4.5 所示。再配合 “1.2” 本身的編碼，得到 “1.2.2” 、“1.2.3” 此兩個 POSITION 的祖先節點。

4.5 kwMatch 壓縮

在 *FindNextChild* 函式中，當我們利用 *kwMatch* 產生 v_p 的一個位於 $d(v_c)$ 層的孩子節點 v_c 後，我們必需依 *kwMatch* 及 v_p 的 *dmatch_range* 所限定的區間，來

指定 v_c 的 *dmatch_range*。我們發現對於某些在區間中連續出現的對應節點，其位於 $d(v_c)$ 層的祖先節點常常為同一個節點，若 *n* 個連續出現的對應節點符合上述狀況，TDPrune 便需在一一檢查這 *n* 個對應節點後，直到在檢查出下一個對應節點位於 $d(v_c)$ 層的編碼不同，或是發現已超出 *dmatch_range* 的範圍後，才能確認出 v_c 的 *dmatch_range*。

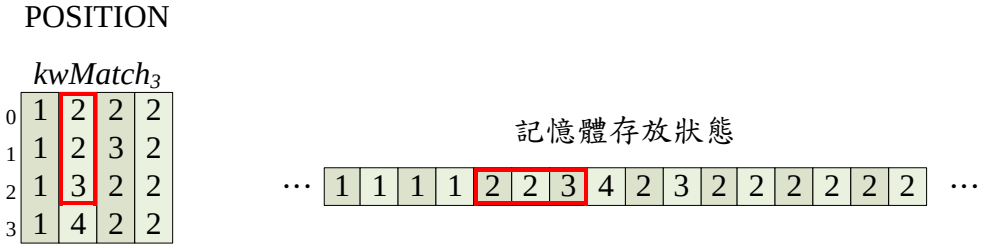

圖 4.6 於壓縮前之 $kwMatch_3$ 指定“1.2” 的 *dmatch_range* 所需進行之檢查動作

[範例 4.4]：考慮以查詢句 *Q2* 進行查詢，當找出節點 “1” 的第一個孩子節點 “1.1” 後， “1” 的 *dmatch_range* 被更新為(-1, -1, 0, 0, 0, 3)，在產生第二個孩子節點 “1.2” 時，必需計算 “1.2” 的 *dmatch_range*，當在計算 $kwMatch_3$ 的開始出現及結束出現的位置時，必需針對記錄於 $kwMatch_3$ 中第 0 列至第 3 列中的所有對應節點，檢查其祖先節點是否為 “1.2” ，來指定節點 “1.2” 記錄 $kwMatch_3$ 的位置之 *dmatch_range*。此時 $kwMatch_3$ 中第 0 列的節點，其編碼的第 1 行的數字為 “2“，表示其為 “1.2” 的子孫節點，我們接著對 $kwMatch_3$ 中第 1 列的節點進行同樣的檢查，檢查結果亦符合，直到對 $kwMatch_3$ 中第 2 列的節點進行檢查時，發現其第 1 行的數字為 “3“，才能確認 “1.2” 的 *dmatch_range* 為 (-1, -1, 0, 0, 0, 1)。過程中，為了指定 “1.2” 的 *dmatch_range* 的第 5 個及第 6 個整數的值，我們對於 $kwMatch_3$ 中共進行了 3 次的檢查動作，即圖 4.6 以方框標示之位置。

為了降低設定 *dmatch_range* 的檢查次數，我們在行導向的 *kwMatch* 中記錄節點連續出現的次數，也就是若節點連續出現的次數大於 1，則於其第二次出現的位置記錄其之後總共出現的次數。詳細規則如下：

對任一串從第 r_s 列第 c 行開始至第 r_e 列第 c 行連續出現的整數 v，若從第 0 行開始至第 c 行的每一行，其第 r_s 列到第 r_e 列之數值於未被取代前皆相同，則將第 $r_s + 1$ 列第 c 行的值以 $r_s - r_e$ 取代。

壓縮的詳細公式如下：考慮壓縮後的第 i 列第 j 行之數字 $new\_value(i,j)$：

$$new\_value(i,j)=\begin{cases} -k & \text{if } full(i\text{-}1, j) = full(i, j) = full(i\text{+}1, j) = \ldots = full(i\text{+}k\text{-}1, j), \\ & \quad full(i\text{-}2, j) \neq full(i\text{-}1, j), full(i\text{+}k\text{-}1, j) \neq full(i\text{+}k, j) \\ value(i,j), & \text{otherwise} \end{cases}$$

其中 $value(i,j)$ 表示未壓縮前第 i 列第 j 行的編碼。

而 $full(i,j)$ 則表示第 i 行之對應節點位於第 j 層的祖先節點之完整編碼。

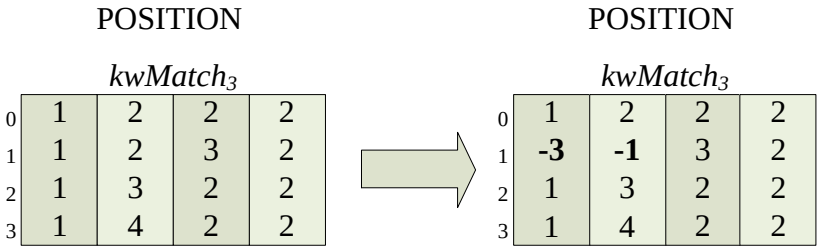

圖 4.7 壓縮前後之 $kwMatch_3$

[範例 4.5]：考慮圖 2.1 的 XML 樹中，標籤為 “POSITION” 的節點共 4 個，其編碼分別為 “1.2.2.2” 、 “1.2.3.2” 、 “1.3.2.2” 、 “1.4.2.2” ，最大深度為 4，若以行編碼並格式化其 *kwMatch*，則其內容為 { 1.**-3**.1.1, 2.**-1**.3.4, 2.3.2.2, 2.2.2.2 }，其中，$new\_value(1,0) = -3$，代表從 $value(0,0)$ 之後同一行的 3 個值 $value(1,0)$，$value(2,0)$， $value(3,0)$，皆等於 $value(0,0)$，即 1。$new\_value(1,1) = -1$，代表從 $value(0,1)$ 之後同一行的 1 個值 $value(1,1)$等於 $value(0,1)$，即 2，如圖 4.7 右方所示。

若改以上述格式儲存 *kwMatch*，由於 *kwMatch* 記錄著節點連續出現的次數，

因此在設定任一位於第 $d(v_c)$ 層之節點 v_c 的 *dmatch_range* 時，僅需依其父親節點 v_p 的 *dmatch_range* 中記錄的開始位置 i，檢查 *kwMatch* 中第 i 列第 $d(v_c)$ - 1 行的數字以及第 i + 1 列第 $d(v_c)$ - 1 行的數字，便可完成設定。

POSITION

kwMatch$_3$

0	1	2	2	2
1	**-3**	**-1**	3	2
2	1	3	2	2
3	1	4	2	2

圖 4.8 於壓縮後之 *kwMatch*$_3$ 指定“1.2” 的 *dmatch_range* 所需進行之檢查動作

[範例 4.6]：考慮以查詢句 *Q2* 進行查詢，當找出節點 “1” 的第一個孩子節點 “1.1” 後， “1” 的 *dmatch_range* 被更新為(-1, -1, 0, 0, 0, 3)，在產生第二個孩子節點 “1.2” 時，必需計算 “1.2” 的 *dmatch_range*，當在計算 *kwMatch*$_3$ 的開始出現及結束出現的位置時，我們首先檢查 *kwMatch*$_3$ 中第 0 列的節點，其編碼的第 2 層 (第 1 行) 的數字為 “2” ，表示其為 “1.2” 的子孫節點，我們接著對 *kwMatch*$_3$ 中第 1 列的節點進行同樣的檢查，其編碼的第 2 層 (第 1 行) 的數字為 “-1” ，表示由此位置開始總共連續出現 1 個 “2” ，因此我們可以知道第 3 列的第 2 層 (第 1 行) 的數字不為 “2” ，進而直接推算出 “1.2” 的 *dmatch_range* 為 (-1, -1, 0, 0, 0, 1)。過程中，為了指定 “1.2” 的 *dmatch_range* 的第 5 個及第 6 個整數的值，我們對於 *kwMatch*$_3$ 中共進行了 2 次的檢查動作，即圖 4.6 以方框標示之位置。

4.6 反轉索引壓縮

我們可以在進行線上查詢之前，便先行對 XML 原始資料進行處理來產生如上格式之數字陣列，再將其存至索引中。而如前述，行編碼常會出現連續出現的相同節點，因此在儲存索引時，我們可以僅記錄該節點的標籤以及重覆出現的次數，於讀取至 *kwMatch* 時再將其還原即可，如此，在線上查詢時，便可藉由降

低讀取的資料量來提升速度。

POSITION

$kwMatch_3$

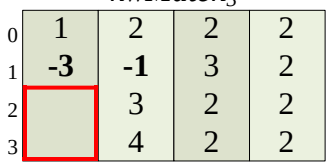

圖 4.9 壓縮索引所節省之儲存空間

[範例 4.7]：考慮存放關鍵字 POSITION 的對應節點的數字陣列而言，其所記錄的內容為 { 1.-3.1.1, 2.-1.3.4, 2.3.2.2, 2.2.2.2 } 此一 4 × 4 的二維整數陣列，共 16 個數字，在儲存至索引中時，我們僅以一維陣列的方式儲存 "1、-3、2、-1、3、4、2、3、2、2、2、2、2、2" 此 14 個數字，即省略 *new_value*(2,0)、*new_value*(3,0) 即足夠於讀取至 *kwMatch* 時，還原為原始的二維數字陣列，如圖 4.9 所示。詳細地說，因為 *new_value*(1,0) = -3，代表從 *value*(0,0) 之後同一行的 3 個值 *value*(1,0)，*value*(2,0)， *value*(3,0)，皆等於 *value*(0,0)，即 1，所以 *new_value*(2,0) = 1，*new_value*(3,0) = 1。

第五章 實驗

在本章中，我們進行實驗來比較 TDPrune、LevelPrune、LevelPrune+、和 MaxMatch 效率上的差異，其中 LevelPrune+為 LevelPrune 加入 4.4 節至 4.6 節所討論的儲存和壓縮方式後改進而成的系統。首先，先說明我們進行實驗的環境，我們以個人電腦作為實驗的環境，其 CPU 為四核心的 Intel i7 2600，其中每個核心的時脈為 3.4GHz，記憶體為 16GB，所採用的作業系統為 Windows 7 企業版 SP1。此外，MaxMatch、TDPrune、LevelPrune 與 LevelPrune+皆以 Microsoft Visual C++ 2010 進行實作，並且採用 Oracle Berkeley DB[2]來儲存索引。

名稱	大小	節點數	最大深度	平均深度	最大寬度	平均寬度
Baseball	10.1MB	51.6 萬	8	7.46	約 25.2 萬	約 3.8 萬
Mondial	17.2MB	51.4 萬	7	5.2	約 12.6 萬	約 6.5 萬
DBLP	820MB	3800 萬	6	3.4	約 1780 萬	約 1232 萬
SwissProt	109MB	499 萬	6	4.02	約 203 萬	約 97.6 萬

表 5.1 資料集之各項參數

名稱	壓縮前索引大小	壓縮後索引大小	壓縮比率
Baseball	16.1MB	7.01MB	43.54%
Mondial	33.2MB	13.4 MB	40.36%
DBLP	1587MB	928 MB	58.48%
SwissProt	110MB	82.2 MB	74.73%

表 5.2 壓縮前後索引大小

[2] http://www.oracle.com/technetwork/database/berkeleydb/

我們測試所使用的資料集包括 Baseball[3]、Mondial[4]、DBLP[5]以及 SwissProt[6]。其中，我們將 Baseball 及 Mondial 的原始內容複製 10 倍，以便在實驗時突顯計量執行時間上的不同。DBLP 為一記錄歷年來資訊領域相關的期刊和會議所發表的論文以及其相關資料，我們使用 2011 年 4 月 13 日的版本，SwissProt 為一記錄蛋白質序列及其相關資訊之資料集。此外，所有資料集之各項參數如表 5.1 所示。我們並將索引在套用 4.6 節所述之壓縮前後之大小列於表 5.2。

最後，說明我們測量執行時間的方式為：對每個查詢句連續執行三次，量取系統第二次以及第三次執行所花費的時間的平均值做為比較。

5.1 各式查詢之實驗

編號	資料集	關鍵字
TQ1	Baseball	Jim, Abbott, Outfield
TQ2	Baseball	Jim, Abbott, James, Baldwin, Starting Pitcher
TQ3	Baseball	PLAYER, Abbott, Baldwin
TQ4	Baseball	Tigers, Starting Pitcher, SURNAME
TQ5	Baseball	Tigers, Starting Pitcher, Outfield, SURNAME
TQ6	Baseball	Cordero, First Base
TQ7	Baseball	Cordero, First Base, Tigers
TQ8	Baseball	1998 Abbott TEAM
TQ9	Mondial	United States, Birmingham, population

[3] http://www.ibiblio.org/xml/books/biblegold/examples/baseball/

[4] http://www.cs.washington.edu/research/xmldatasets

[5] http://dblp.uni-trier.de/xml/

[6] http://www.cs.washington.edu/research/xmldatasets/

TQ10	Mondial	United States, United Kingdom, Birmingham, population
TQ11	Mondial	Tasmania, Sardinia, Gotland, Area
TQ12	Mondial	ethnicgroups, Chinese, Indian, Capital
TQ13	Mondial	mondial, country, Muslim
TQ14	Mondial	country, Muslim
TQ15	Mondial	Asia, China, government
TQ16	Mondial	organization, name, member

表 5.3 查詢句 *TQ1 ~ TQ16*

我們首先使用 MaxMatch[LC08]中做為評測的 16 個查詢句來對各個系統作測試，藉由第三方所定立之多樣化的查詢句來比較各系統的效率。其中 *TQ1 ~ TQ8* 為其針對 Baseball 資料集所設計之查詢句，而 *TQ9 ~ TQ16* 則為其針對 Mondial 資料集所設計之查詢句，如表 5.3 所示。

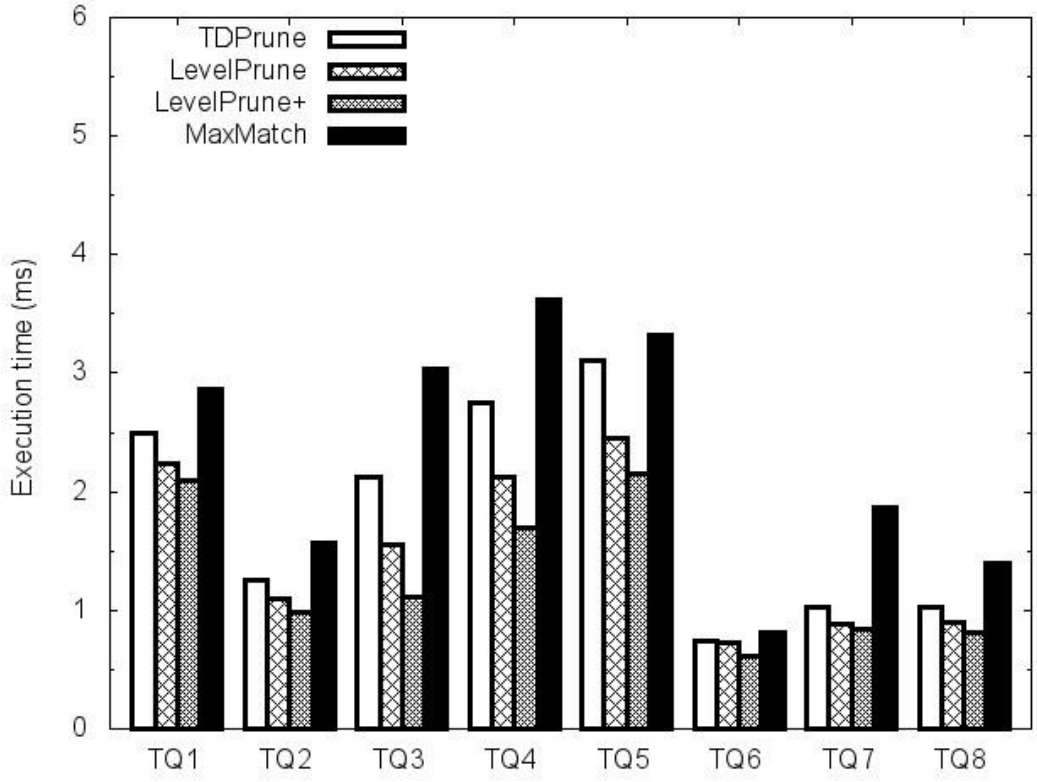

圖 5.1 *TQ1 ~ TQ8* 之實驗柱狀圖 (Baseball)

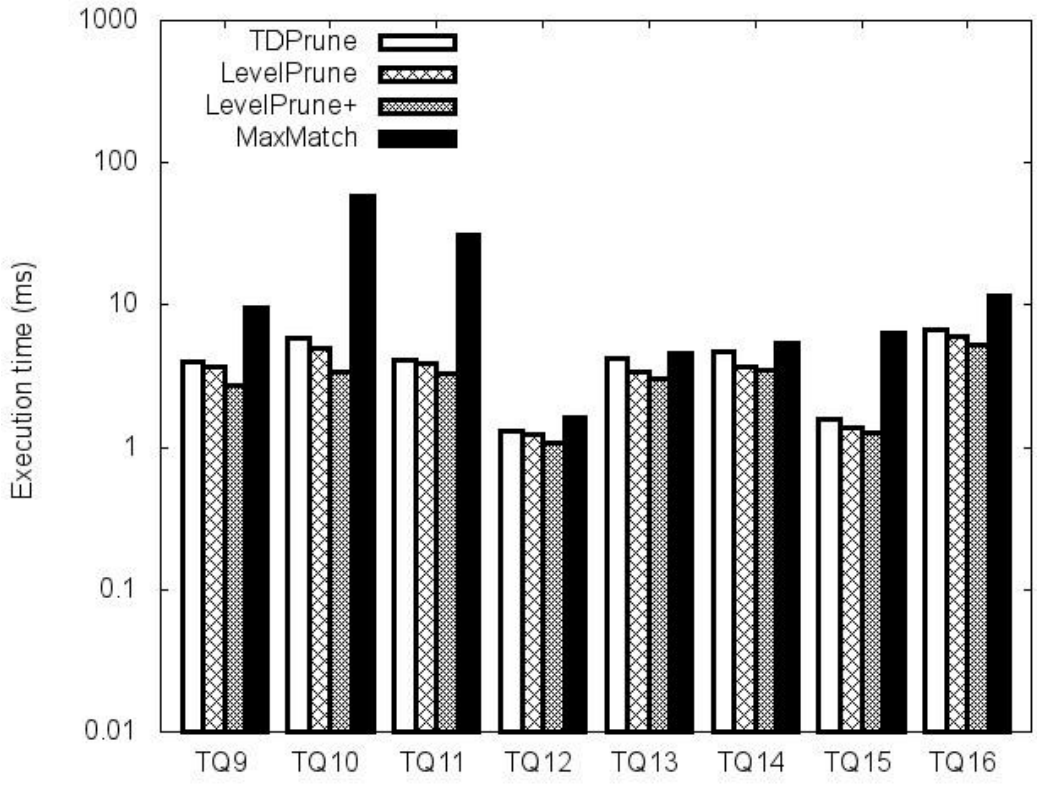

圖 5.2 *TQ9 ~ TQ16* 之實驗時間柱狀圖 (Mondial)

編號	貢獻量	$\|v_{all}\|$ ($\|v_c\| + \|v_{pc}\| + \|v_{pnc}\|$)	貢獻比率
TQ1	920	1060	86.79%
TQ2	210	350	60%
TQ3	400	400	100%
TQ4	640	970	65.98%
TQ5	850	1110	76.58%
TQ6	50	50	100%
TQ7	210	590	35.59
TQ8	380	530	71.7%
TQ9	640	8030	7.97%
TQ10	3980	85450	4.66%
TQ11	7080	37580	18.84%

TQ12	270	270	100%
TQ13	4410	4410	100%
TQ14	3000	3000	100%
TQ15	2430	4780	50.84%
TQ16	5080	5080	100%

表 5.4 查詢句 *TQ1 ~ TQ16* 之各式節點統計

圖 5.1 及圖 5.2 分別表示 *TQ1 ~ TQ8* 以及 *TQ9 ~ TQ16* 之實驗時間的柱狀圖。此外，我們並將 SLCA 節點至其下對應節點之路徑上的所有節點依 3.5 節所述之類別分類並統計各類之總數量如表 5.4 所示，其中，貢獻量表示所有 v_c 類以及 v_{pc} 類節點數量，即$|v_c| + |v_{pc}|$，貢獻比率表示貢獻量與所有 v_c 類、v_{pc} 類和 v_{pnc} 類節點數量之比率，即$(|v_c| + |v_{pc}|) / (|v_c| + |v_{pc}| + |v_{pnc}|)$。我們可以由圖 5.1 及圖 5.2 中觀察到，在各式的查詢中，本論文所提出的三個系統在執行時間上皆較 MaxMatch 有著不同程度的改進。其中，相較於 LevelPrune 以一個 Node 串列一次處理整層所有節點，TDPrune 必需多次動態創建與刪除用於保存節點及其兄弟節點形成的 Node 串列，即圖 3.7 中之 *tmplist*。因此，在執行時間上，LevelPrune 優於 TDPrune。而 LevelPrune+又因為節省了載入反轉索引的時間而優於 LevelPrune。注意到，我們將圖 5.2 中的執行時間 (Y 軸) 改以對數座標呈現，針對 *TQ9 ~ TQ16*，LevelPrune+在執行時間上依然較 LevelPrune 有著一定程度的改善。此外，MaxMatch 因為必需對所有 v_{all} 節點進行計算其 *dMatch* 及 *dMatchSet* 的動作，因此在 *TQ9*、 *TQ10*、*TQ11* 貢獻比率較小的情況下，其執行效率與我們所提出的系統有明顯差別，差距接近 1 個 order。

5.2 查詢句關鍵字數量之實驗

編號	*k*	*M*	資料集	關鍵字
TQ17	3	10 萬	DBLP	1996、editor、note
TQ18	4	10 萬	DBLP	editor、note、1991、incollection
TQ19	5	10 萬	DBLP	note、CoRR、publisher、proceedings、1985
TQ20	6	10 萬	DBLP	CoRR、publisher、isbn、proceedings、1988、series
TQ21	3	20 萬	SwissProt	Ref、DB、DOMAIN
TQ22	4	20 萬	SwissProt	Ref、Gene、TRANSMEM、Eukaryota
TQ23	5	20 萬	SwissProt	Ref、PIR、BY SIMILARITY、CARBOHYD、DISULFID
TQ24	6	20 萬	SwissProt	Species、Features、PIR、BY SIMILARITY、CARBOHYD、DISULFID

表 5.5 查詢句 *TQ17 ~ TQ24*

在本節中，我們控制對應節點的總個數，並操縱關鍵字的個數，設計了 *TQ17 ~ TQ24* 以觀察關鍵字個數對執行時間的影響。其中，*TQ17 ~ TQ20* 為針對 DBLP 之查詢句，對應節點總個數固定為 10 萬 ± 5%，關鍵字個數分別為 3 ~ 6 個，*TQ21 ~ TQ24* 為針對 SwissProt 之查詢句，對應節點總個數固定為 20 萬 ± 5%，關鍵字個數分別為 3 ~ 6 個，如表 5.5 所示。

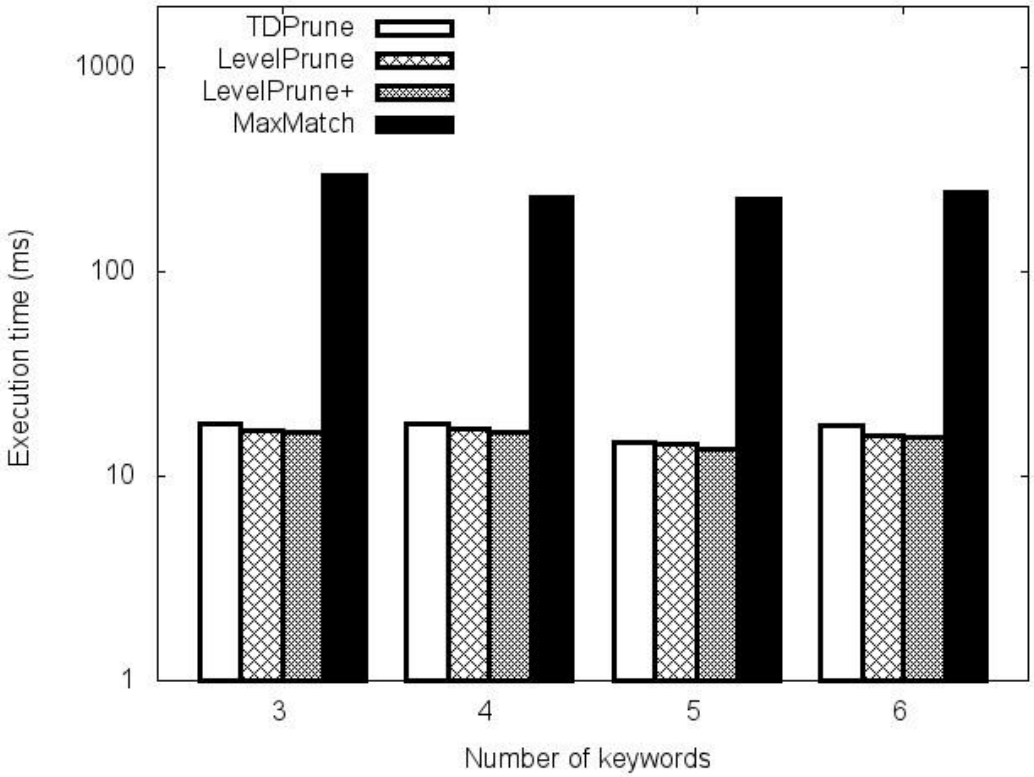

圖 5.3 *TQ17 ~ TQ20* 之實驗柱狀圖 (DBLP)

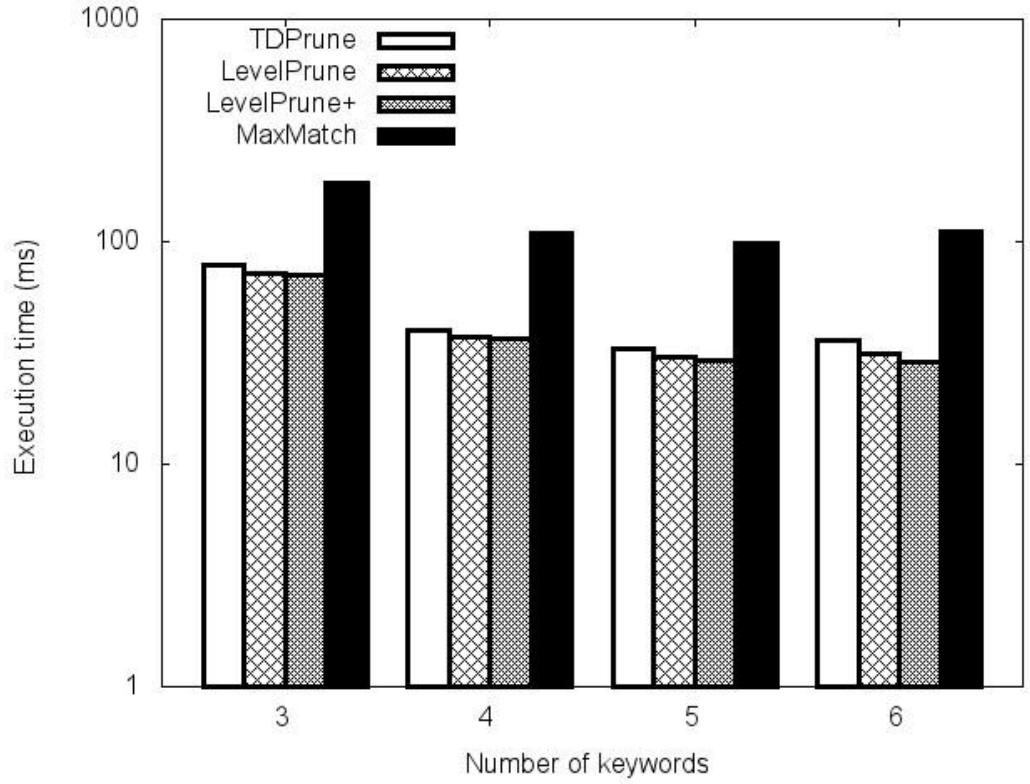

圖 5.4 *TQ21 ~ TQ24* 之實驗柱狀圖 (SwissProt)

編號	貢獻量	$\|v_{all}\|$ ($\|v_c\| + \|v_{pc}\| + \|v_{pnc}\|$)	貢獻比率
TQ17	75606	202029	37.42%
TQ18	79819	181716	43.93%
TQ19	57883	178918	32.35%
TQ20	60278	186334	32.35%
TQ21	115786	115786	100%
TQ22	54790	54790	100%
TQ23	34343	35155	97.69%
TQ24	31726	32538	100%

表 5.6 查詢句 *TQ17 ~ TQ24* 之各式節點統計

TQ17 ~ TQ20 以及 *TQ21 ~ TQ24* 的實驗結果以及其各類節點之總數量分別如如圖 5.3、圖 5.4 和表 5.6 所示。本論文所提出之三個系統在執行時間上皆大幅快於 MaxMatch，且所有系統的執行時間並不隨著查詢句中關鍵字的數量而上升。接著我們可以由圖 5.3 中看到，由於 MaxMatch 需對所有 v_{all} 節點進行處理，因此在 *TQ17 ~ TQ20* 之 v_{all} 節點總數量相差不多的情況下，MaxMatch 執行四個查詢句的時間亦沒有相差太多。而在圖 5.4 中，v_{all} 節點總數量反而隨著關鍵字的增加而減少，因此MaxMatch 系統所減少的執行時間之比例亦與其大致相符。此外，由於本論文所提出之三個系統皆僅需對 v_c 以及 v_{pc} 類節點進行處理，而在 *TQ17 ~ TQ20* 中之貢獻比率皆較在 *TQ21 ~ TQ24* 中的貢獻比率為小，因此在 *TQ17 ~ TQ20* 中，本論文所提出之三個系統與 MaxMatch 所花費時間之差距較 TQ21 ~ TQ24 為大。另外，由於 *TQ21 ~ TQ24* 中各類節點數量較其於 *TQ1 ~ T16* 中增加，因此各系統在計算貢獻節點時所花費的時間佔總執行時間的比率亦隨之升高。相對而言，索引壓縮所能改善的 I/O 時間佔總執行時間的比例便降低。因此，LevelPrune+ 較 LevelPrune 節省的時間不若其在 *TQ1 ~ T16* 中明顯，但依然有著 5% ~ 10%不同程度的改善。

5.3 對應節點數量之實驗

編號	*M*	*k*	資料集	關鍵字
TQ25	低	3	DBLP	180、IUI、157
TQ26	中	3	DBLP	cdrom、INTERSPEECH、sub
TQ27	高	3	DBLP	2002、4、editor
TQ28	低	3	SwissProt	PROBABLE、Liliopsida、Respiratory chain
TQ29	中	3	SwissProt	Transmembrane、SIGNAL、Proteobacteria
TQ30	高	3	SwissProt	Cite、Comment、Species

表 5.7 查詢句 *TQ25 ~ TQ30*

在本節中，我們控制關鍵字的個數，並操縱對應節點之總個數，設計了 *TQ25 ~ TQ30* 以觀察對應節點之總個數對執行時間的影響。其中，*TQ25 ~ TQ27* 為針對 DBLP 之查詢句，關鍵字個數固定為 3，對應節點總個數分別為 $\alpha_1 \times 10^2$、$\alpha_2 \times 10^3$、$\alpha_3 \times 10^4$ 萬，*TQ28 ~ TQ30* 為針對 SwissProt 之查詢句，關鍵字個數固定為 3，對應節點總個數分別為 $\alpha_4 \times 10^2$、$\alpha_5 \times 10^3$、$\alpha_6 \times 10^4$ 萬，$\alpha_1 \sim \alpha_6$ 皆為大於 1 且小於 3，如表 5.7 所示。

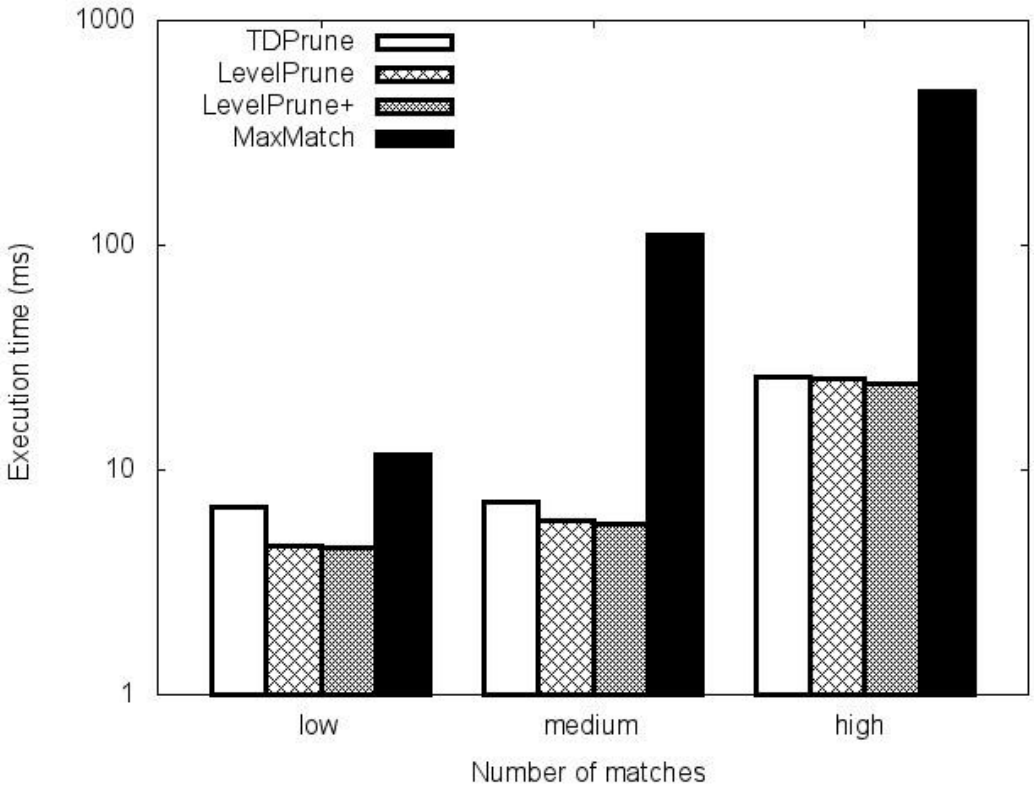

圖 5.5 *TQ25 ~ TQ27* 之實驗柱狀圖 (DBLP)

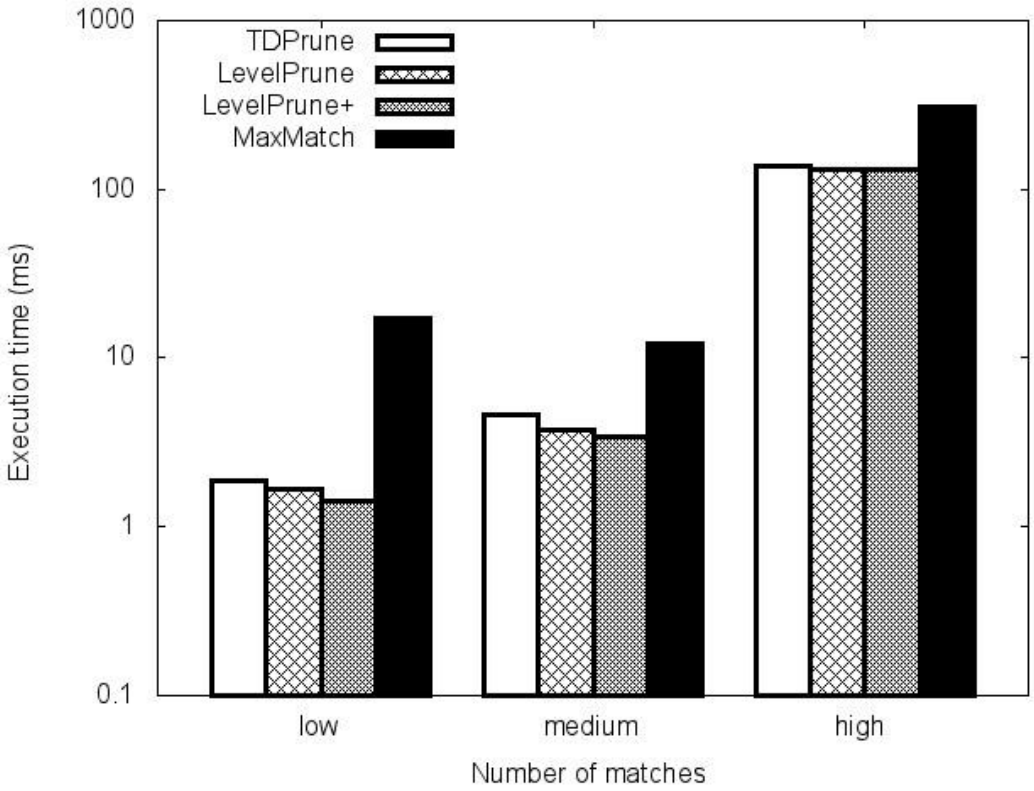

圖 5.6 *TQ28 ~ TQ30* 之實驗柱狀圖 (SwissProt)

編號	貢獻量	$\|v_{all}\|$ ($\|v_c\| + \|v_{pc}\| + \|v_{pnc}\|$)	貢獻比率
TQ25	8959	8959	100%
TQ26	27557	70587	39.04%
TQ27	111359	331361	35.33%
TQ28	2704	10032	26.95%
TQ29	1141	1141	100%
TQ30	228223	246169	92.71%

表 5.8 查詢句 *TQ25 ~ TQ30* 之各式節點統計

TQ25 ~ TQ27 和 *TQ28 ~ TQ30* 的實驗結果，以及其各類節點之總數量分別如圖 5.5、圖 5.6 和表 5.8 所示。圖 5.5 中，*TQ25 ~ TQ27* 因為 v_{all} 節點總數量的上升，MaxMatch 執行時間的確隨之上升。然而，在圖 5.6 中，*TQ29* 由於 v_{all} 節點總數量較 *TQ28* 為少，MaxMatch 的執行時間反而下降。因此，對應節點總數量上升雖有機會使得$|v_{all}|$一併上升，但對應節點總數量並不直接影響執行時間。

另外，我們觀察到在 *TQ30* 中，本論文所提出之系統與 MaxMatch 之差距較其他查詢句為小，我們分析其原因如下。MaxMatch 系統必需在第一次追蹤時，計算所有節點之 *dMatch* 及 *dMatchSet*，在第二次追蹤時，記憶體中已放置了所有節點之相關資訊以供判斷是否為貢獻節點。相反地，本論文所提出的系統，在動態產生節點及計算 *dMatchRange* 並在另外產生某一數量的節點後，隨即利用先前產生之節點來進一步產生其他節點，即節點產生之第一次存取到用之產生其他節點之第二次存取之間隔較 MaxMatch 來得小，因此，在$|v_c| + |v_{pc}|$較小時，本論文提出的系統有較大的機會能利用快取記憶體的特性來提升執行效率，即針對每一個系統需處理的節點，本論文提出的系統對單一節點能以較少的平均時間去處理。換言之，當$|v_c| + |v_{pc}|$較大時，本論文提出的系統對單一節點的處理時間便與 MaxMatch 差距不大，因此在 *TQ30* 兩系統需處理的節點數差距亦不大時，其執

行時間的的差距便縮小。

5.4 輸出節點數量之實驗

編號	*M*	$\|v_c\|$	資料集	關鍵字
TQ31	10002	30004	DBLP	dblp、1987
TQ32	22065	66196	DBLP	dblp、1987、1988
TQ33	31159	93478	DBLP	dblp、1995
TQ34	41927	125782	DBLP	dblp、1998

表 5.9 查詢句 *TQ31 ~ TQ34*

在本節的實驗中，我們選用根節點 *dblp* 以及代表年份的數字做為關鍵字。因為年份與根節點 *dblp* 的層數差為 3 層，且所有節點皆為貢獻節點，因此貢獻節點的個數會大約等於對應節點總個數之 3 倍。藉由這樣的方法，我們可以透過控制對應節點總個數來控制貢獻節點的總個數，進而觀察各系統的執行效率。*TQ31 ~ TQ34* 如表 5.9 所示。

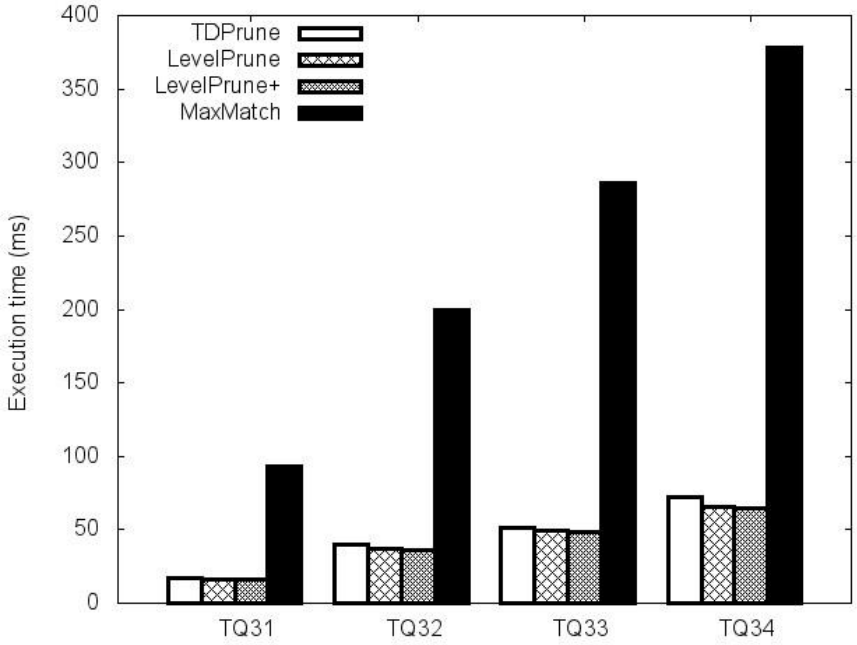

圖 5.7 輸出節點數量的實驗柱狀圖 (DBLP)

編號	$\|v_c\|$	$\|v_{pc}\|$	$\|v_{all}\|$	貢獻比率
TQ31	30004	0	30004	100%
TQ32	66196	0	66196	100%
TQ33	93478	0	93478	100%
TQ34	125782	0	125782	100%

表 5.10 查詢句 *TQ31 ~ TQ34* 之各式節點統計

TQ31 ~ TQ34 的實驗結果及各類節點統計分別如圖 5.7 及表 5.10 所示。我們可以看到，所有系統所需計算的節點數量皆為$|v_{all}|$，而各系統的執行時間亦隨著$|v_{all}|$數量的上升成線性的上升。

5.5 貢獻比率之實驗

編號	對應年份之節點總個數	year 節點之總個數	year 成為貢獻節點之比率	關鍵字
TQ35	31161	1629795	1.91%	dblp、year、1995
TQ36	162869	1629795	9.99%	dblp、year、2009
TQ37	294284	1629795	18.06%	dblp、year、2008、2010
TQ38	419951	1629795	25.77%	dblp、year、2008、2007、2006

表 5.11 查詢句 *TQ35 ~ TQ38*

在本節的實驗中，我們選用 dblp 此一根節點及其孩子節點 year，並利用 year 的內容節點來控制 year 節點成為貢獻節點 ($|v_c|$) 的比率，進而控制總貢獻比率並觀察其對四個系統所造成的影響，*TQ35 ~ TQ38* 如表 5.11 所示。

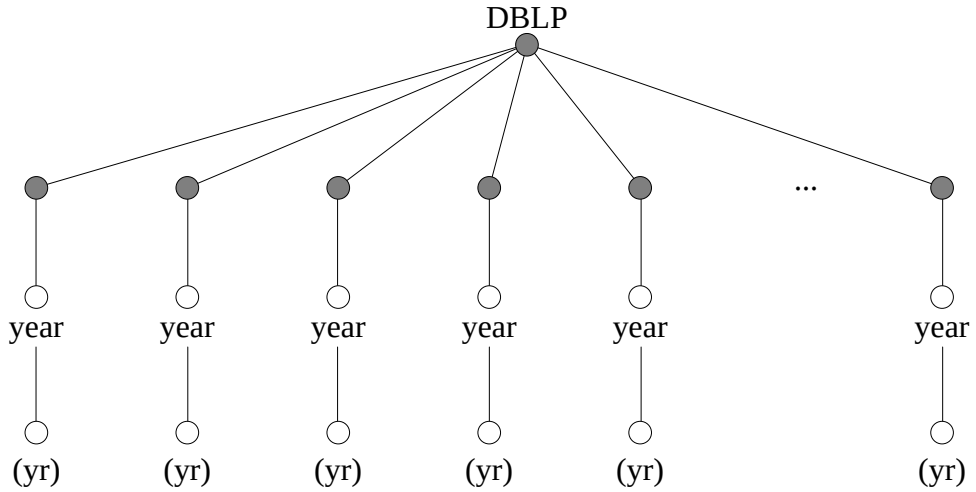

圖 5.8 查詢句 *TQ35 ~ TQ38* 之輸出結構圖

此外，我們選用的關鍵字 year 為根節點之差距 2 層的子孫節點，而代表著年份的數字為其內容節點，因此我們可以預期 SLCA 僅有根節點一個，且結構如圖 5.8 所示，其中 (yr) 為代表年份之關鍵字。

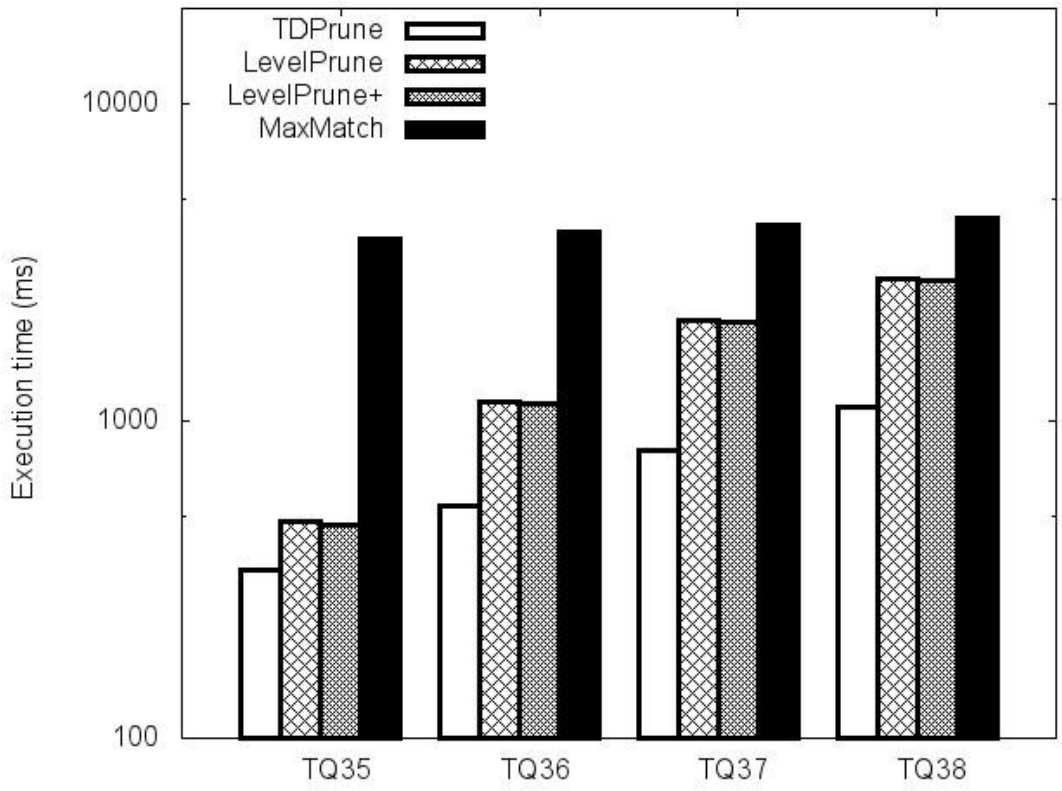

圖 5.9 貢獻比率的實驗柱狀圖 (DBLP)

編號	貢獻量	$\|v_{all}\|$	貢獻比率
TQ35	1692111	3290757	51.42%
TQ36	1954575	3423417	57.09%
TQ37	2216267	3555974	62.33%
TQ38	2467628	3681622	67.03%

表 5.12 查詢句 *TQ35 ~ TQ38* 之各式節點統計

TQ35 ~ TQ38 的實驗結果及各類節點統計分別如圖 5.9 及表 5.12 所示。我們可以發現，MaxMatch 系統由於$|v_{all}|$沒有太大的變動，因此 MaxMatch 系統的執行時間亦大致相同，本論文所提出的三個系統則隨著$|v_c| + |v_{pc}|$的上升，而使得執行時間有顯著的上升。三個系統與 MaxMatch 系統之間執行時間的差距隨著$(|v_c| + |v_{pc}|) / |v_{all}|$的比率上升而變小。

	各層所有節點之總處理時間		
系統	第 1 層	第 2 層	第 3 層
TDPrune	263.22	28.9	24.4
LevelPrune	263.24	87.47	110.7
LevelPrune+	262.88	85.34	108.36

表 5.13 *TQ35* 各系統對各層所有節點之總處理時間

此外，我們觀察到，在這系列的實驗中，LevelPrune 的執行時間相較 TDPrune 有著較大幅的增加。因此，我們針對 *TQ35* 將所有需被計算的節點分層統計其花費時間，如表 5.13 所示。我們發現，針對第 1 層的根節點，三個系統花費大約相同的時間以產生其孩子節點。然而，針對第 2 層及第 3 層的節點，LevelPrune 與 LevelPrune+系統卻花費了數倍於 TDPrune 的時間來處理。我們發現到，此答案樹在第二層和第三層皆只有一個孩子節點 (扇出為 1) ，所以 TDPrune 在處理第 2 層和第 3 層的任一節點 *v* 時，在代表 *v* 的 Node 結構及其 *dmatch_range* 被產

生後遂即被用以產生下一層的節點。相較之下 LevelPrune 與 LevelPrune+則需將同層的所有節點的 Node 結構皆被產生完後，才從該層之第一個節點開始逐一產生下一層的節點，所以花費較多時間。在下一節，我們針對此類型的答案樹進行更進一步的分析。

5.6 同層節點數量之實驗

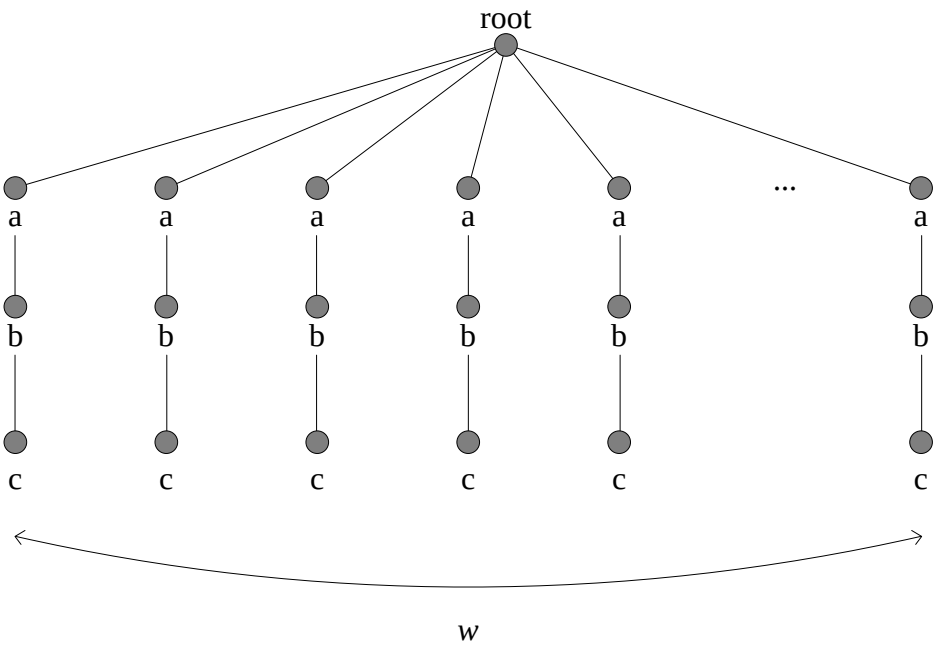

圖 5.10 custom 系列資料集之結構

編號	資料集	根節點之扇出	樹的寬度 (*w*)	關鍵字
TQ39	custom60k.xml	6 萬	6 萬	root、c
TQ40	custom70k.xml	7 萬	7 萬	root、c
TQ41	custom80k.xml	8 萬	8 萬	root、c
TQ42	custom90k.xml	9 萬	9 萬	root、c

表 5.14 查詢句 *TQ39 ~ TQ42*

針對如圖 5.10 的答案樹，此節比較 TDPrune 與 LevelPrune 系列系統在不同「同層節點數量」之情況下的表現，此實驗使用了 custom60k.xml、custom70k.xml、

custom80k.xml 和 custom90k.xml 此四個資料集，其結構皆如圖 5.10 所示。四者之差別僅在於根節點 root 之扇出 (fan-out) 值，分別為 6 萬，7 萬，8 萬和 9 萬。我們藉由控制根節點 root 之扇出值來控制樹的最大「同層節點個數」*w*，並以 root 及 c 做為關鍵字，分別對上述四個資料集進行查詢，如表 5.14 中之 *TQ39* ~ *TQ42* 所示。

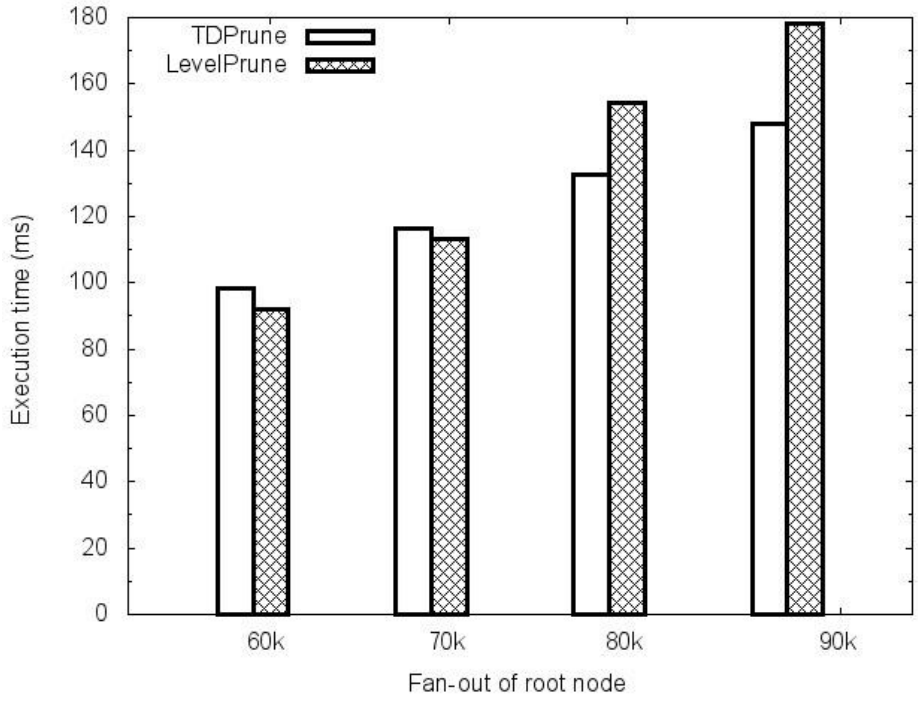

圖 5.11 *TQ39* ~ *TQ42* 之實驗柱狀圖

TQ39 ~ *TQ42* 的實驗結果如圖 5.11 所示。我們可以看到，TDPrune 的執行時間隨著樹的寬度呈現約略線性的成長，而 LevelPrune 卻在寬度為 8 萬的時候急速成長，也就是在同層節點個數為 6 萬及 7 萬時，LevelPrune 在執行時間上優於 TDPrune。而在同層節點個數為 8 萬及 9 萬時，LevelPrune 反而較 TDPrune 為慢。所以，樹的寬度會影響到系統的效率。

第六章 結論及未來方向

在本論文中，我們提出了 TDPrune 與 LevelPrune 此兩種系統來計算[LC08]針對其所定義的關鍵字查詢輸出結果，並提出以多項改進方法套用至 LevelPrune 而成之 LevelPrune+進一步提升效率。在判斷是否為貢獻節點方面，三系統皆可以直接利用儲存在反轉索引內的對應節點資料來做為判別之用。如此便可降低所有處理的節點的總數量。我們並額外提出了與貢獻節點相關的性質，使得在由上往下地產生任一個節點後，若能確認該節點為貢獻節點，則可直接將其輸出，而不需再使用額外的空間來記錄這些節點是否為貢獻節點後，留待所有節點皆檢查完後再依對應節點的資訊來判斷要輸出的節點。最後的實驗結果顯示，三系統皆能在大部份的情況下減少所需計算的節點數量而較 MaxMatch 在執行時間上有著不同程度的效能提升。一般而言，LevelPrune+有著最快的執行效率，而僅在少數極端的情況下，TDPrune 會較 LevelPrune+為快。

本論文未來的研究方向，期望能針對回傳的結果加上 ranking 的機制，使得系統能夠支援 Top-k 的關鍵字查詢。

參考文獻

[ABBP10] Nikolaus Augsten, Denilson Barbosa, Michael Böhlen, Themis Palpanas, "TASM: Top-k Approximate Subtree Matching", In Proceeding of the ICDE Conference, 2010.

[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.

[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.

[CP10] Liang Jeff Chen, Yannis Papakonstantinou, "Supporting Top-K Keyword Search in XML Databases". In Proceeding of the ICDE Conference, 2010.

[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.

[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 Proceeding of the 34th International Conference on VLDB Conf, Auckland, New Zealand, 2008.

[LCC10] Rung-Ren Lin, Ya-Hui Chang, Kun-Mao Chao, "Efficient Algorithm for Searching Relevant Matches in XML DataBases", In Proceedings of DEXA ,2010.

[LLZW11] Jianxin Li, Chengfei Liu, Rui Zhou, Wei Wang "Top-k Keyword Search over Probabilistic XML Data", In Proceedings of ICDE Conference, Pages: 193–204, Norway, 2011.

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

[VOPT08] Quang Hieu Vu, Beng Chin Ooi, Dimitris Papadias, Anthony K. H. Tung, "A Graph Method for Keywordbased Selection of the topK Databases", In Proceeding of SIGMOD Conference, 2008.

[XP05] Y. Xu and Y. Papakonstantinou, "Efficient Keyword Search for Smallest LCAs in XML Databases", In SIGMOD, 2005.

[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： groupMatches 演算法

演算法名稱：groupMatches 說　　明：merge *kwMatch* in Dewey ID order to a list *match* in DeweyID order
L01: *match*[*v*] ← *merge*(*kwMatch*$_0$, …, *kwMatch*$_{w-1}$)
L02: //group matches
L03: $i \leftarrow 0, j \leftarrow 0$
L04: **while** ($i \neq u$) or ($j \neq v$) **do**
L05: group[i].t ← SLCA[i]
L06: **if** *isAncestor*(*group*[*i*].*t*, *match*[*j*]) **then**
L07: *group*[*i*].*M* = *group*[*i*].*M* ∪ *match*[j]
L08: $j \leftarrow j + 1$
L09: **else if** *group*[*i*].*M*≠∅ **then**
L10: $i \leftarrow i + 1$
L11: **else**
L12: $j \leftarrow j + 1$

附錄 B： pruneMatches 演算法

演算法名稱：pruneMatches

說　　明：identify relevant matches in *M* and output query result trees

```
L01:  i ← M.size
L02:  start ← t
L03:  //set dMatch and dMatchSet during a post-order tree traversal
L04:  while i ≥ 0 do
L05:     for each node n on the path from M[i] (exclusively) to start do
L06:        if n matches keyword[j] then
L07:           set the j^th bit of n.dMatch to 1
L08:        n_p ← n.parent, n_c ← n.child on this path
L09:        if n_c ≠ Null then
L10:           n.dMatch ← n.dMatch OR n_c.dMatch
L11:        n.last ← i //record the last descendant match of n
L12:        n_p.dMatchSet[num(n.dMatch)] ← true //num is the function converting
            abinary number to a decimal number
L13:     i ← i - 1
L14:     start ←LCA( M[i], M[i + 1] )
L15:  //print query result during a pre-order tree traversal
L16:  i ← 0
L17:  start ← t
L18:  while i ≤ M.size do
L19:     for each node n from start to M[i] do
L20:        if isContributor(n) = false then
```

```
L21:            i ← n.last + 1 //skip the matches in the subtree rooted at n
L22:            break
L23:        else
L24:            output n
L25:    i ← i + 1
L26:    start ← LCA( M[i - 1], M[i] )
```

演算法名稱：isContributor

說　　明：return true if n is a contributor, otherwise false

```
L01:  n_p ← n.parent
L02:  i ← num(n.dMatch)
L03:  for j ← i + 1 to 2^w - 1 do
L04:      if n_p.dMatchSet[j] = true && AND(i, j) = i then
L05:          return false
L06:  return true
```