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

支援遞迴資料定義之 XQuery 查詢句轉換系統
Translating XQuery in Recursive Schemas

研究生：吳俊賢　撰

中華民國 98 年 7 月

支援遞迴資料定義之 XQuery 查詢句轉換系統

Translating XQuery in Recursive Schemas

研 究 生：吳俊賢　　Student：Chun-Hsien W

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 (Dissertation)
Submitted to Department of Computer Science and Engineering
College of Electrical Engineering and Computer Science
National Taiwan Ocean University
In Partial Fulfillment of the Requirements
for the Degree of
Master of Science
in
Computer Science and Engineering
July 2009
Keelung, Taiwan, Republic of China

中華民國 98 年 7 月

摘要

自從 W3C 組織正式頒布了可擴充式標註語言 XML，XML 已經發展為資料交換的標準格式。而針對 XML 格式的資料，則是以 W3C 組織制訂的 XQuery 來做查詢處理。但是當 XML 資料庫以不同的格式儲存，想要從別的資料庫取出所需要的資料，會因為雙方 XML 結構不同，而導致查詢困難。

本論文的目的，在提出一個針對遞迴資料定義的 XQuery 查詢句轉換系統。為達此目的，本論文設計遞迴資料的對應方式及產生函數的方法。首先須由管理者撰寫正確的對應表示法，再藉由此對應所產生的函數進行查詢句轉換。我們實做一個轉換系統，建立表格記錄函數與查詢句內路徑之間的對應關係，接著藉由查表取得符合的函數解集合，並加以處理，以便將輸入的查詢句轉換成可對另一個資料庫查詢的查詢句。

我們設計了一系列的實驗，以測試不同的狀況下所得到的結果是否符合本論文的設計。實驗證明我們提出的轉換系統，可以正確地保留原查詢句的結構關係，並且產生所有符合原查詢句結構關係的查詢句轉換結果。

Abstract

Extensible markup language (XML) has recently emerged as the standard of data exchange since W3C published it, and the standard query language for XML data is XQuery. How to perform XML query translation between different databases is therefore an important issue.

The purpose of this thesis is to propose a query translation system for translating XQuery which is posed against recursive schemas. This system uses the mapping written by DBA to create functions to support query translating. To achieve this goal, we need to create a new way to represent recursive data mapping and function creating, so that we can easily query different schemas of the databases which have the same data. We also create mapping tables to record the relation between paths in the query and functions, variable and solutions. We implement the whole query translation system based on the mapping information.

We have performed a series of experiments to test the correctness of our system. Experimental results show that our system can produce correct queries which keep the structure represented in the original schema.

致謝

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。同時也要感謝台大教授陳信希博士和系上教授林川傑博士百忙中抽空參與論文審查工作，提供許多寶貴的意見和建議。

感謝父母及家人給我精神上的支持與援助，讓我能無後顧之憂的完成此論文。感謝畢業學長們的不斷支持與鼓勵，以及系上給予協助及照顧的系助理們。最後要感謝吳政儀同學、黃治中學弟、李佳臻學妹和李俊毅學弟的幫忙，在此一併致上謝意，謝謝你們。

俊賢 2009.6.30

章　節　目　錄

圖目錄

表目錄

第一章、 序論

1.1 研究動機與目標

自從 1998 年 W3C 組織正式頒布了可擴充式標註語言(Extensible Markup Language：XML)，由於該語研允許使用者針對所需要的資料，自訂標註(Markup）來描述資訊內容意義，所以已經成為網際網路資料交換的標準格式。

由於資料可能以不同的格式，儲存於不同的資料庫中，當想要從別的資料庫取出所需要的資料時，會因為雙方 XML 結構不同，而導致查詢困難。因此在進行跨資料庫查詢時，必須考慮兩者資料結構上的差異。如何解決資料結構上的差異，以達到跨資料庫查詢的目的，是個重要的課題。本論文提出一個轉換查詢句的轉換系統，希望能夠將針對某一個 XML 資料庫的 XQuery 查詢句，轉換成適合於另一個 XML 資料庫的 XQuery 查詢句。目前 W3C 定義的 XML 查詢句為 XQuery，該語言以路徑表示法（Path Expression）為基礎，但也允許巢狀表示法(Nested Expression)，本論文處理查詢句轉換將著重於跳層路徑表示法的轉換及巢狀表示式的處理。所謂跳層路徑表示法，與直接路徑表示法不同，後者直接明確的指向 XML 中的某一個元素節點，前者是指向所有符合跳層（//）之後的元素名稱的所有元素節點，意即跳層路徑表示法所包含的資訊將大於等於直接路徑表示法所包含的資訊。

本論文的作法，主要是透過對應表示法（Mapping）來建立函數（Skolem Function)，以便利查詢句的轉換，與既有研究的不同之處，在於特別考量跳層、遞迴路徑與巢狀表示法所產生的影響，以及對如何轉換出正確的查詢句作深入的探討。本論文的主要貢獻，總結如下所述：

1. 遞迴路徑的對應表示法與函數的建立：分析資料庫間遞迴路徑的差異，並設計適合的對應表示法，並有效利用函數處理遞迴路徑。
2. 跳層路徑之處理：針對跳層與遞迴路徑，於 XML 最大資料深度條件下，建立查訪表格（Tracing Table)，以便於取得跳層路徑的所有解。
3. 實做查詢句轉換系統：實際開發此轉換系統，經由分析輸出結果，與原查詢

句之結構對照，証明轉換系統可以正確並有效率的轉換出對應的查詢句。

1.2 相關研究

建立本論文之查尋句轉換系統，會牽涉的相關研究可分為對應表格的建立與函數的產生、處理巢狀表示式、跳層與遞迴路徑查詢句轉換之影響，及等價查詢句。

對應表格的建立與函數的產生，論文 [YP04] 針對兩個不同 Schema 的資料，建立變數宣告、Schema 內部連結限制式，與葉元素對應之 Mapping 資訊，再利用 Mapping 產生有結構意義的 Skolem Function，進而利用 Skolem Function 將原始查詢句轉換成目標查詢句。[VMM+05] 提出在進行資料整合時，使用 mapping 來得知如何從 source 獲得 target 所需的資料。mapping 分為兩部份，首先將 source 資訊表示於 foreach 子句中，其中利用 SQL 查詢語法記錄資料從哪些表格取出（from），取出哪些欄位（select），以及 source 內部所需的 join（where）。再將 target 資訊記錄於 exist 子句中，同樣利用 SQL 記錄資料分別存放在哪些表格（from）的哪些欄位（select），以及 target 內部所需的 join（where）。

[DTC+03] 則研究如何轉換巢狀結構、包含複雜路徑的 FLWR 表示式。它使用動態區間編碼，對 XML 文件用深度搜尋演算法（DFS）對每個元素、屬性和文字節點編碼，依此編碼可以找出以某個元素為頭下面整個區間的資料，利用區間編碼的特性，找出元素屬性之間的層次關係，以解決 PC 路徑、AD 路徑和 Wildcard 的問題。

遞迴路徑及跳層，參考論文 [FYL+05] 所設計之遞迴資料結構，與遞迴路徑的處理方法，將迴圈與非迴圈結構，以遞迴路徑的節點做為區分兩者的依據。[KCK+04] 針對 Recursive 的 XML Schema 進行 XML-to-SQL 的查詢句轉換，將 XML to Relational 的對應 Schema S 轉換成 automaton AS，並將對應的 XQuery 轉換成 Finite automaton AQ，而最後出來的 SQL，利用 SQL99 中的 with 子句將各 SQL 子句結合起來。

[ODP+06] 根據輸入的 XQuery queries 與 XQuery views，輸出等價的 XQuery rewriting。其做法分為三個部份：（1）替 query 中的所有變數，找到所對應到的 view，取得 alternate view access path；（2）根據第一部分所得到的 view access path，在 tag 與 groupby 中的變數適當的置換，得到 candidate rewriting （3）將得到的 candidate rewriting 與原來的 XQuery 比較，看是否是等價的。

1.3 論文架構

本篇論文架構如下：在第二章中，介紹並解釋 XML DTD Schema 以及 XML Query，並定義處理的語法範圍，以及定義何謂正確的轉換。第三章說明 [YP04] 所使用的 Mapping 與 Skolem Function 產生方法，並加入本論文所增加的遞迴結構的 Mapping 方式與遞迴結構轉換為 Skolem Function 步驟及範例。第四章是轉換系統模組的介紹，並進一步說明相關的演算法及轉換過程。於第五章列出查詢句轉換前及轉換後的輸出結果，並討論其結構是否維持原有之結構關係。最後在第六章提出本篇論文的結論與未來展望。

第二章、 相關定義

在本章中，首先介紹 DTD Schema 範例及 XQuery 查詢句，進而探討 Schema 有迴圈存在時，將 Target Query 轉換成 Source Query 時，所產生的問題，並定義何謂等價的查詢句。

本論文之所謂的 Target Schema，是使用者所下的查詢句（Target Query）所針對的 Schema，而 Source Schema 是欲取得資料來源所定義的 Schema。必須將 Target query 轉換成對應 Source Schema 的 Source query 才能順利取出資料。

2.1 Target and Source Schema 範例

圖 2.1 的 Target DTD 範例，是記錄大學（dept）的課程（course）與課程之間，及課程與學生（student）之間的關係，每一門課程都有其課號（cno）、課名（title）、先修資訊(prereq)、被哪些學生選修（takenBy）等資訊。先修資訊記錄的是另一門課程的資料，此結構於 DTD 中產生迴圈（見圖 2.2），迴圈是由路徑 //prereq/course 所產生。選修資訊則是記錄了學生（student）的資料，學生資料包含了學號（sno）、姓名（name）、已完成資訊 (qualified）等資訊。已完成資訊記錄的是可能為零筆或多筆的課程資訊，也產生另一個迴圈結構，由路徑 //qualified/course 所產生。為了方便說明，本論文會將 DTD 定義以圖型表示，如圖 2.2 所示，以下定義 DTD Graph 中的節點和線。

【定義 2.1】：DTD Graph 定義

- 根節點（Root Node)：
  - 根節點是對應到 DTD 中的根元素，圖中以單實線方塊表示。
  - Graph 中的所有元素節點都是它的子節點或是後代節點。
  - 根節點為不可重覆元素。
- 內部節點（Internal Node)：
  - 可重覆元素（Repeatable Node)：
    - 在 XML 文件中，針對同一個父元素，可出現零次以上。

- 在 DTD 中以單實線方塊、右上方加上星號“*”表示。

- 不可重覆元素（Un-Repeatable Node)：
  - 在 XML 文件中，針對同一個父元素，只能出現一次。
  - 在 DTD 中以單實線方塊表示。

◆ 葉節點（Leaf Node)：

- 內容節點（Value Node)：
  - 為一有值的元素，DTD 中以單實線橢圓表示。
  - 若為多值，則於橢圓右上方加星號“*”表示。

◆ 實線：

連結元素與其子元素，為一個父子（parent / child）關係。

```
L1  <!ELEMENT dept (course*)>
L2  <!ELEMENT course (cno, title, prereq, takenBy)>
L3  <!ELEMENT prereq (course*)>
L4  <!ELEMENT takenBy (student*)>
L5  <!ELEMENT student (sno, name, qualified)>
L6  <!ELEMENT qualified (course*)>
L7  <!ELEMENT cno (#PCDATA)>
L8  <!ELEMENT title (#PCDATA)>
L9  <!ELEMENT sno (#PCDATA)>
L10 <!ELEMENT name (#PCDATA)>
```

圖 2.1 Target DTD 範例

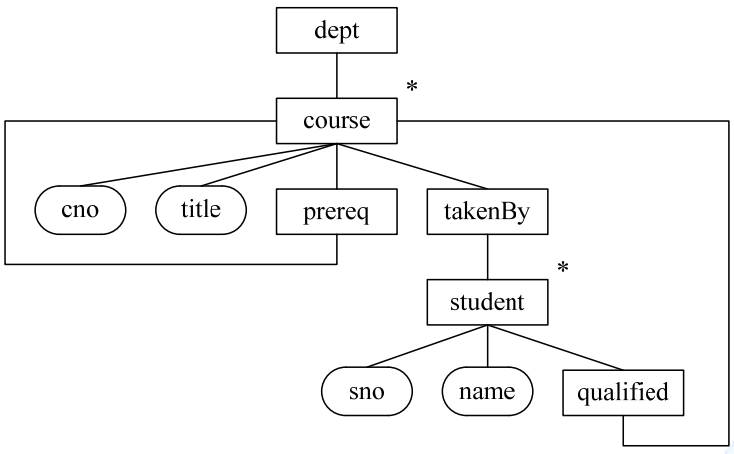


圖 2.2 Target DTD Graph 範例

下圖的 Source DTD 範例，與 Target DTD 相同，是記錄大學（dept）的課

程（course）與學生（student）的資訊。每一門課程包含有課號（cno）、課名（title）、先修資訊（prereq）、被哪些學生選修（takenBy)等資訊。其中所不同的是，被哪些學生選修（takenBy）只記錄了學生的學號（sno），並不如同 Target DTD 一樣記錄了學生的其他資訊。學生資訊（student）是另外獨立的一個集合，其下亦記錄了學號（sno）、姓名（name）、已完成資訊（qualifieds）。與 Target DTD 不同的是，已完程資訊只記錄了課號（cno），而不是記錄了完整的課程資料。

L1	<!ELEMENT dept (course*, student*)>
L2	<!ELEMENT course (cno, title, prereq, takenBy)>
L3	<!ELEMENT prereq (course*)>
L4	<!ELEMENT takenBy (sno*)>
L5	<!ELEMENT student (sno, name, qualifieds)>
L6	<!ELEMENT qualifieds (cno*)>
L7	<!ELEMENT cno (#PCDATA)>
L8	<!ELEMENT title (#PCDATA)>
L9	<!ELEMENT sno (#PCDATA)>
L10	<!ELEMENT name (#PCDATA)>

圖 2.3 Source DTD 範例

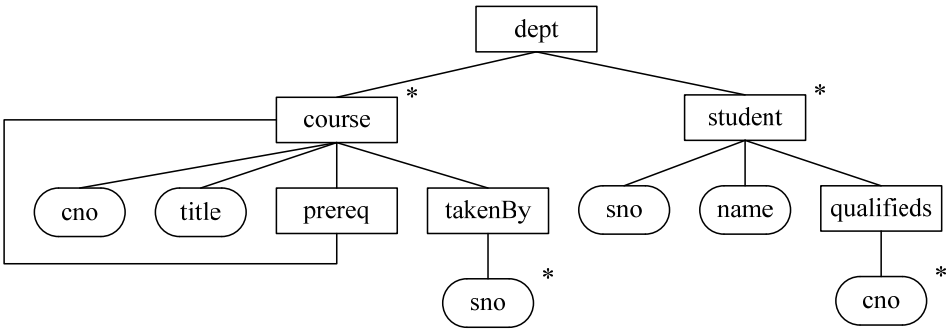


圖 2.4 Source DTD Graph 範例

對應圖 2.3 的 DTD graph，如圖 2.4 所示。該範例是在文件根元素（dept）下分為兩個子樹，左子樹包含了課程（course）資訊。其中 course 為可重覆元素，其子元素 cno、title 為內容節點。內部節點 prereq 下僅有可重覆元素 course；takenBy下則記錄了多值的學號（sno）資訊。右子樹則包含了學生（student）資訊。其中 student 為可重覆元素，其子元素 sno、name 為內容節點。內部節點 qualifieds 下僅有可重覆元素 cno。

2.2 查詢句介紹及 Target and Source Query 範例

XML 文件內的元素間有著階層關係，W3C 組織定義了相關技術來處理 XML 資料，譬如以路徑表示法（Path Expression）為基礎的 XQuery。路徑表示法自根元素（Root Node）透過位址步驟（Location Step）結合 XML 文件中的元素名稱所組成，指出資料在 XML 文件裡的關係位置，結果為一元素集合。在本論文中，位址步驟僅考慮父子與祖孫關係，分別以“/”及“//”符號來表示。屬性則不列入本論文的處理範圍。如圖 2.4 中，course 下的葉元素 cno 的路徑表示法為 /dept/course/cno。

接著，我們介紹 XQuery 查詢句，一個 XQuery 查詢句是由 FOR、LET、WHERE、RETURN，等四個子句所組成，其中 FOR 與 LET 子句最少需出現一個，並有一變數代表其所宣告的路徑所取得的資料集合，其中 FOR 子句是以遞迴的方式取得一個路徑表示法的結果，LET 子句則是一次性的取得所有的結果。WHERE 子句則允許對所宣告的變數所代表的集合內容做條件的限制，依據所下達的限制式內容，過濾取出所需要的資料，限制式可分為條件限制式（定義 2.2）與連結限制式（定義 2.3）。RETURN 子句則是依據子句中所定義的標籤（tag）與變數，或是巢狀表示式（定義 2.4），將建構好的資料回傳給使用者。

【定義 2.2】XQuery 條件限制式：在 XQuery 的 WHERE 子句中，限制式為文字或數字限制，表示法為：(葉節點路徑 = 文字或數字）或（文字或數字 = 葉節點路徑）。

【定義 2.3】XQuery 連結限制式：在 XQuery 的 Where 子句中，限制式代表兩個內部元素間，利用其下相同意義的葉節點來建立連結關係，表示法為：葉節點路徑 = 葉節點路徑。

【定義 2.4】XQuery 之巢狀表示式（Nested expression)：由 Let 所定義的變數，其所代表的資料內容是由另一個查詢句所取得；或於 XQuery 中，由 RETURN 所回傳的資訊中定義的 {…} 資訊，其定義的資訊為一查詢句。以上兩者我們皆稱之為巢狀表示式。

【範例2.2-1】Target Query

```
FOR     $t0 in /dept/course
LET     $t1 :=    FOR     $t2 in $t0/takenBy/student
                  RETURN  $t2/name
WHERE     $t0/cno = "C2"
RETURN    $t1
```

【範例2.2-2】Source Query

```
FOR     $s0 in /dept/course
LET     $s1 :=    FOR     $s2 in $t0/takenBy,
                          $s3 in /dept/student
                  WHERE   $s2/sno = $s3/sno
                  RETURN  $s3/name
WHERE     $s0/cno = "C2"
RETURN    $s1
```

範例 2.2-1 與範例 2.2-2 中的 Query 是以圖 2.1 的 Target Schema 與圖 2.3 的 Source Schema 當作範本所撰寫，上述兩個查詢句皆以取得選修課號為“C2”課程之學生姓名資訊為目的。其中，Target Query 透過條件限制式 $t0/cno = "C2"，以取得 /dept/course/takenBy/student/name 的資料（$t2/name）。在 Source Query 中，LET 所宣告的變數 $t1，其巢狀表示式透過連結限制式 $t2/sno = $t3/sno，且於外層的查詢句限制 $t0/cno = "C2"，以取得 $t1 的資料。

範例 2.2-1 與 2.2-2 的 Target Query 與 Source Query 所取得的 name 集合，在結構意義上，對應到的都是 /dept/course/takenBy/student/name 在限制 /dept/course = "C2" 下，所取得的學生姓名之資料集合，只是 Target Query 直接透過本身的 Schema 結構關係，可直接取得同一 course 下之 student 集合中，所有的 name 集合，而 Source Query 因 Schema 與前者不同，須透過 /dept/course/takenBy/sno 的限制才能取得同一 course 下的 student 集合中，所有的 name 集合。

2.3 問題描述

當 Target DTD 或 Source DTD 中，有遞迴路徑時，且兩者的結構不同，但儲存的資料內容相同，該如何藉由什麼樣的方法，才能於 Source XML 中取得

Target Query 想要取得的資料？但在轉換的過程中，當查詢句所使用的路徑包含了跳層，我們必須保留 Target Query 原有的結構關係，使得經過轉換後的 Source Query 能表達出 Target Query 所代表的語意。當 Target Query 中有複雜的巢狀結構時，保留原有的結構關係將是查詢句轉換是否正確的最大因素。如下述範例 2.3-1 與範例 2.3-2 包含了跳層與巢狀表示式的 Target Query 與 Source Query 。

範例 2.3-1 的查詢句，是查詢課程名稱為 “T1” 的課程資訊中，所有修課學生資訊中的已完成課程的課號。其中，變數 $t0 是指向每一個 /dept/course 及該元素下所有資料，並用條件限制式限制 $t0/title = "T1"，變數 $t1 是指向變數 $t0 下的每一個修課學生，變數 $t2 所指向的是一個巢狀表示式所回傳的資料，變數 $t3 是取出所有的 student 集合及該元素下所有資料，並用連結限制式限制 $t1/sno = $t3/sno，變數 $t4 是對變數 $t3 下的已完成課程資訊，取得課程的名稱集合。最後輸出學生的姓名，與已完成課程的課號的集合。

【範例 2.3-1 】Target Query

```
FOR     $t0 in /dept/course,
        $t1 in $t0/takenBy/student
LET     $t2 :=    FOR     $t3 in //student,
                          $t4 in $t3/qualified/course
                  WHERE   $t1/sno = $t3/sno
                  RETURN  $t4/cno
WHERE       $t0/title = "T1"
RETURN
    <result>
            <student_name> { $t1/name } </student_name>
            <qualifieds>{ $t2 } </qualifieds>
    </result>
```

但若是想在 Source XML 中取得這樣的資訊，我們必須透過範例 2.3-2 查詢句所表示的結構：

【範例 2.3-2】Source Query

```
FOR     $s0 in /dept/course,
        $s1 in $s0/takenBy,
        $s2 in /dept/student
LET     $s3 :=   FOR     $s4 in //student,
                         $s5 in $s4/qualifieds/cno
                 WHERE   $s2/sno = $s4/sno
                 RETURN  $s5
WHERE   $s0/title = "T1" AND $s1/sno = $s2/sno
RETURN
    <result>
            <student_name> { $s2/name } </student_name>
            <qualifieds>{ $s3 } </qualifieds>
    </result>
```

首先在取得學生（student）的集合時，就必須先透過課程（course）之下的選修資訊（takenBy），取得在同一堂課下的學生學號（sno）資料，在此處就必須如範例 2.3-2 一般，透過限制 $s1/sno = $s2/sno，方能取得同一課程下的學生資訊。而原本於 Target Query 中，可直接透過 qualified 元素下的 course，取得已完成課程的課號，但在 Source DTD 中，只能透過 student 下的 qualifieds 取得已完成課號的資訊，所以，於巢狀表示式中的變數所定義之路徑與 RETURN 的回傳值也都需要修正。如範例 2.3-1 與 範例 2.3-2 所示，$t4 in $t2/qualified/course 修正為 $s5 in $s4/qualifieds/cno，而 RETURN 中的 $t4/cno 修正為 $s5。

2.4 所能處理 Query 之 BNF

本論文所能處理的查詢句如圖 2.4 所定義的 BNF 所示，查詢句只允許在 Let 與 Return 中出現巢狀表示式。For 所定義的路徑，只保留了會處理到的變數名稱、跳層與非跳層路徑及元素名稱。Let 僅能定義 Nested Query。Where 可以限制元素與元素、元素與字串或數字。Return 所能處理的資訊除了 Nested Query 外，亦含有：單一元素、由 Tag 所包含的任意上述的合法宣告。

```
FLWORExpr ::= (ForClause | LetClause)+ WhereClause? "return" ReturnClause
        ForClause ::=
            "for" "$" String "in" ExprSingle ("," "$" String "in" ExprSingle)*
        LetClause ::= "let" "$" String ":=" FLWORExpr
        WhereClause ::= "where" AndExpr
//----------------------Where Condition BNF----------------------
            AndExpr ::= ComparisonExpr ( "and" ComparisonExpr )*
                ComparisonExpr ::=
                    "$" String ExprSingle GeneralComp "$" String ExprSingle
                    | "$" String ExprSingle GeneralComp (Digits | String)
                    | (Digits | String) GeneralComp "$" String ExprSingle
                GeneralComp ::= "=" | "!=" | "<" | "<=" | ">" | ">="
//----------------------PATH路徑定義BNF----------------------
ExprSingle ::= (("/" | "//") String)*
//----------------------RETURN定義BNF----------------------
ReturnClause ::= TagReturnBasic | "{" "$" String ExprSingle "}"
    TagReturnBasic ::= "<" String">" ReturnBasic* "</" String ">"
    ReturnBasic ::= TagReturnBasic | EnclosedExpr
        EnclosedExpr ::= "{" Expr | FLWORExpr "}"
            Expr ::= "$" String ExprSingle ("," "$" String ExprSingle)*
//-----Terminal Symbols-----
//-----The following symbols are used only in the definition of terminal symbols;
String ::= [A-Za-z] ([A-Za-z0-9._] | '-')*
Digits ::= [0-9]+
```

圖 2.5 Query BNF

2.5 正確的轉換

在將 Target Query 轉換為 Source Query 後，由於 Target Schema 與 Source Schema 不同，所以分別以 Target Query 與 Source Query 查詢所得到的結果，格式與元素名稱不完全相同。但是，若是一個正確的轉換，Target Query 中所需要的資料，其相互之間的關係，在轉換為 Source Query 後仍會保留。以下我們正式定義何謂正確的轉換。首先我們定義查詢句中一組相關的變數：

【定義 2.4】父變數：假設變數之宣告為 For $Si in $Sj/Path，則稱 $Sj 為 $Si 之父變數。

例如，範例 2.2-1 與範例 2.2-2 ：在 Target Query 查詢句中，原為 AD（

ancestor-descendent) 關係的兩變數 $t0 in /dept/course 與 $t2 in $t0/takenBy/student，若在轉換成 Source Query 時，透過 WHERE 的連結限制式，$s2/sno = $s3/sno，使 $s0 in /dept/course 與 $s3 in /dept/student 在 Source Query 中保持 Target Query 結構所代表之意義。

接著我們定義變數間的結構關係：

【定義 2.5】：變數間的結構關係有下列三種：(1) Parent-Child (以下簡稱 PC 關係)，(2) Ancestor-Descendent (以下簡稱 AD 關係)，(3) 以連結限制式將兩個變數連結起來，使其所取得的資料範圍在限制之下得以固定。

以範例 2.2-1 的 Target Query 為例，變數 $t0 所指向的元素「course」與變數 $t2 所指向的元素「student」，於路徑宣告時，變數 $t2 的父變數 (定義 2.4) 為 $t0，因此我們將變數 $t0 與變數 $t2 視為 PC 關係。然而在轉換為 Source Query 時，因 Source Schema 中的 course 與 student 並非 PC 或 AD 關係，但 course 之下有一元素，takenBy，記錄了 student 的 Primary Key，sno，即可以透過此元素，來聯結 course 與 student 兩個元素，使查詢句中的兩變數仍保有原先設定條件，同一個 course 下的 student 資訊，例如：(1) 範例 2.2-1 的 Target Query 中的變數 $t2 所指向的元素 student，(2) 範例 2.2-2 的 Source Query，變數 $s3 所指向的元素 student。因前者變數宣告時，父變數為 $t0，所以在結構上， $t2 所指向的 student 被 course 限制；而後者因在巢狀表示式中，有連結限制式 $s2/sno = $s3/sno 存在，使得 $s3 所指向的資料被限制在 course 之下。因此，此兩個查詢句在結構意義上是相等的。

所以，我們提出以下定義，規範何謂等價的 (equivalent) 查詢句：

【定義 2.6】：若 Target query 和 Source query，其每一組變數所指向的元素節點，對應的結構關係相符合，則為一個等價的查詢句轉換。

根據定義 2.6，我們規劃設計轉換系統，驗證是否能轉換出正確的查詢句。

第三章、 Mapping 及 Skolem Function

本節介紹將會使用的 Mapping，包括 Mapping 目的、表示法、新增的 Mapping 功能，及 Skolem Function 代表的意義及產生方式。

3.1 既有的 Mapping 的說明

當需要透過另一個 XML 文件查詢資料時，由於兩者 XML DTD Schema 不同，則必須重新撰寫 XQuery 以符合所需要的查詢。目前做法大多是，藉由撰寫 Mapping，並透過程式將 Mapping 轉換成有結構意義的 Skolem Function，如此使得透過 Skolem Function 將原有的 XQuery 轉換成等價的新 XQuery。

本節說明論文 Constraint-Based XML Query Rewriting for Data Integration [YP04] 中，所使用的 Mapping 格式，及產生 Skolem Function（SK）的方法。首先，Mapping 格式及變數說明如下：

Foreach $\$X_0$ in SP_0, …, $\$X_n$ in SP_n
Where *ConditionS1* and *ConditionS2* and …
Exist $\$Y_0$ in TP_0, …, $\$Y_m$ in TP_m
Where *ConditionT1* and *ConditionT2* and …
With $\$Yi/PATH/E = \$Yj/PATH/E$ and …

圖 3.1 論文 [YP04] Mapping 表示法

TP：Target Path，針對 Target Schema，直接定義之完整路徑，或繼承 $\$Y_r$ 而定義的路徑，r：*0~m*

ConditionTk：Target Schema 中的內部連結限制式，k：1, 2, …。

SP：Source Path，針對 Source Schema，直接定義之完整路徑，或繼承 $\$X_r$ 而定義的路徑，r : *0~m*

ConditionSk：Source Schema 中的內部連結限制式，k：1, 2, …。

PATH：0個以上的 Element 所組成的路徑

E：葉元素

該格式透過 Foreach 子句定義 Source Schema 中可重複但非遞迴之元素，並將 Source 中的內部連結限制式撰寫於 Foreach 下的 Where 限制中。透過 Exist 子句定義 Target 中可重但複非遞迴之元素，並將 Target 中的內部限制撰寫於 Exist 下的 Where 限制中。With 中記錄 Target Element 與 Source Element 的對應關係，但若為Target（Source)之內部 Reference Key，則不需於 With 中記錄其對應之 Source（Target）元素。

為方便說明，我們另外給兩個沒有迴圈的 DTD 定義，如圖 3.2 所示之 Target DTD Schema 與 Source DTD Schema，我們畫出圖 3.3 DTD Graph 與圖 3.4 Target 至 Source 的 Mapping。

Target DTD Schema	Source DTD Schema
<!ELEMENT dept (course*)> <!ELEMENT course (cno, title, takenBy)> <!ELEMENT takenBy (student*)> <!ELEMENT student (sno, name)> <!ELEMENT cno (#PCDATA)> <!ELEMENT title (#PCDATA)> <!ELEMENT sno (#PCDATA)> <!ELEMENT name (#PCDATA)>	<!ELEMENT dept (course*, student*)> <!ELEMENT course (cno, title, takenBy)> <!ELEMENT takenBy (sno*)> <!ELEMENT student (sno, name)> <!ELEMENT cno (#PCDATA)> <!ELEMENT title (#PCDATA)> <!ELEMENT sno (#PCDATA)> <!ELEMENT name (#PCDATA)>

圖 3.2 Target and Source DTD

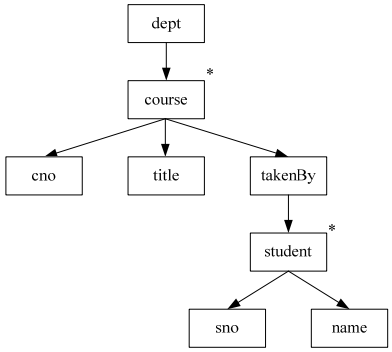
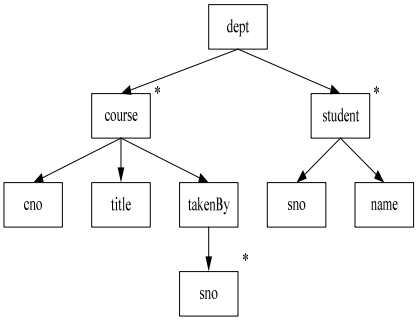
Target DTD Graph	Source DTD Graph
	

圖 3.3 Target and Source DTD Graph

```
foreach   $S0 in /dept/course,
          $S1 in $S0/takenBy,
          $S2 in /dept/student
    where     $S1/sno = $S2/sno
exist     $T0 in /dept/course,
          $T1 in $T0/takenBy/student
with      $S0/cno = $T0/cno,
          $S0/title = $T0/title,
          $S2/sno = $T1/sno,
          $S2/name = $T1/name
```

圖 3.4 Target to Source Mapping

在設定 Mapping 時，我們可將其 Foreach 視為 Source Query 之變數宣告，Foreach 下的 Where 視為 Source Query 之限制條件；另外將 Exist 視為 Target Query 之變數宣告，Exist 下的 Where 視為 Target Query 之限制條件。而 With 所定義之 Foreach 與 Exist 之葉元素（Leaf Element）對應，表示在上述的 Target Query 與 Source Query 中，所取得的每一筆 Leaf Element ，其所代表之意義相等，具有相同的結構性質。

撰寫完 Mapping 之後，藉由程式將 Mapping 轉換為 Skolem Function（SK Function），SK Function 的產生方式為，對 Exist 中定義的每一個可重複元素，依照該元素在 DTD Graph 的位置，以杜威編碼給予編號，並產生 SK Function，SK Function 記錄的資訊，依照該 Target 元素於 Mapping 中，位於該元素下之未經過任何可重複元素之葉元素資訊，與未經過任何可重複元素之可重複元素資訊，並以「[」與「]」區隔上下層元素，以「，」區隔 Sibling 元素。以圖 3.4 的 Mapping 為例，Target 中的元素 course 所產生的 Skolem Function 如下：

```
SKtgt.0.0( )←
      for $S0 in /dept/course
      return [cno = $S0/cno, title = $S0/title,
                        takenBy = [student = SKtgt0.0.3.0( )] ]
```

3.2 新增、修改的 Mapping 格式

本節說明本論文提出針對 Schema 中，迴圈與迴圈對應的 Mapping 表示法，在此之前先提出以下定義：

【定義 3.1】子路徑：任何變數所定義的路徑，其父變數（定義 2.4）之下所定義之由一對或多對 Location Step（“ / ”或“ // ”）與 Element 所組合成之路徑，通稱為子路徑。對於由 Location Step“ / ”與 Element 組合成之子路徑稱之為「一般子路徑」；由“ // ”與 Element 組合成之子路徑稱之為「跳層子路徑」。

如：$b in $a/prereq/course、$b in $a//course、…，我們稱 /prereq/course 為一般子路徑，//course 為跳層子路徑。

【定義 3.2】Recursive Element：於 DTD Schema 中定義於元素 A 之下（PC 或 AD 關係）的元素 B，其定義之下的元素有元素 A 存在，而元素 A 之上層定義並無元素 B 的存在，則稱元素 A 為 Recursive Element。

如圖 2.1 中之 course 元素，我們稱之為 Recursive Element。

【定義 3.3】Recursive Path：若元素 D 為一可重覆元素，且有一 PC 或 AD 關係的元素 A 為 Recursive Element（定義 3.2），則由 D 到 A 的子路徑（定義 3.3）稱為 Recurisve Path。

如圖 2.2 之可重覆元素 course 下之子路徑「/prereq/course」、「/takenBy/student/qualified/course」及可重覆元素 student 下之子路徑「/qualified/course」，我們稱之為 Recursive Path。

在 Mapping 中加入遞迴對應的功能，使得圖 2.1 與圖 2.3 所列出 Schema 中的 Recursive Element 也能透過 Mapping 進行轉換，而不會有資料遺失的問題。

3.1 節 [YP04] 所提出的做法，是將同一個 SK Function 所記錄的節點內容，每一層資料以“ [”與“] ”區隔，在本論文中，我們修改了 SK Function 的表示方式，直接以「子路徑 = 葉元素」或「子路徑 = SK Function」記錄該元素

的對應關係，並只針對多值的元素進行編碼。以圖 3.4 的 Mapping 為例，Target 中的元素 course 所產生的 Skolem Function 如下：

```
SK0
    for $S0 in /dept/course
    return /cno = $S0/cno, /title = $S0/title,
        /takenBy/student = SK0.0
```

在原有的 Mapping 中，雖以“[”與“]”區格每一層的資料，但難免造成閱讀時的困擾，在本論文中修改為上述的表示方示，不僅在閱讀上可以一目了然的看見該 Skolem Function 之下有多少個葉元素的對應，及多少個 Skolem Function 的對應，亦可在建立 Skolem Function 結構時，以“子路徑（定義 2.4）= 對應資訊”的方式直接儲存資訊。

本論文修改 Skolem Function 的編碼方式，不再依照 DTD 中所存在的位置給予 Dewey 編碼，而是依據 Mapping 中，變數所定義之多值元素的順序與上下層關係給予 Dewey 編碼，使得編碼只運用在辨別 Skolem Function，避免編碼過於冗長。但遞迴路徑所產生的 Skolem Function 編碼方式則由範例 3.3-1 與範例 3.3-2 解說。另外，本論文在 Mapping 中加入 Recursive Path to Recursive Path（RR）的對應資訊，以處理在 Target Schema 與 Source Schema 的可重覆之 Recursive Element 的遞迴資料對應。目的是使得 Target Schema 中遞迴的 Schema，在透過 Mapping 轉換後，可以對應到 Source Schema 中的 Recursive Path，並產生與原來意義相同的查詢句。本論文遞迴的路徑限制在最上層的可重覆元素，或是遞迴至元素自己本身的 Recursive Path，以及一對一元素對應的狀況。

關於新增 Recursive Path 對應，以圖 2.1 Target DTD 做為輸入的 Schema，圖 2.3 Source DTD 做為輸出的 Schema，分別寫於圖 3.5 Mapping 中的 Exist 與 Foreach，此 Mapping 必需在定義完圖 3.1 的 Mapping 定義之後，視 Schema 是否有前述的遞迴對應狀況，於 Foreach 與 Exist 中，加入可重複的遞回路徑（Recursive Path）定義。前述之遞迴路徑定義，對 Target Schema 中的遞迴路徑定義於 Mapping 中的 Exist 時，以直接路徑表示，如圖 3.5 之變數 $T1

in $T0//prereq/course；而對 Source Schema 之遞迴路徑定義於 Mapping 中的 Foreach 時，需以跳層子路徑（定義 3.1）表示，如圖 3.5 之變數 $S4 in $S0//prereq/course。

```
Foreach   $S0 in /dept/course,
          $S1 in $S0/takenBy,
          $S2 in /dept/student,
          $S3 in $S2/qualifieds/cno,
          $S4 in $S1//prereq/course
     Where $S1/sno = $S2/sno
Exist     $T0 in /dept/course,
          $T1 in $T0//prereq/course,
          $T2 in $T0/takenBy/student,
          $T3 in $T2/qualified/course
With      $T0/cno = $S0/cno,
          $T0/title = $S0/title,
          $T2/sno = $S2/sno,
          $T2/name = $S2/name,
          $T3/cno = $S3
RR($T1, $T0, $S0, $S4)
```

圖 3.5 有遞迴路徑對應的 Mapping

圖 3.5 的 Mapping 範例中，RR（$T1, $T0, $S0, $S4），以下以 RR（T*y*, T*x*, S*i*, S*j*）代替，簡稱為 RR，其所代表的意義為：有一 Target 的遞迴（Recursive）路徑 Ty，遞迴至非遞迴路徑 T*x* 所指向之元素，透過 Mapping 轉換成 Source 的路徑後，仍為遞迴路徑，且 Target 的遞迴路徑 T*x* 所指向的節點，其子元素與 Mapping 中所定義之非遞迴的可重複節點 T*y* 相同。而 T*x* 所產生的 Skolem Function，繼承 T*y* 所產生的 Skolem Function 的RETURN 資訊後，加入變數宣告 S*j*，再將其 Return 中的所有變數資訊 S*i*，以變數 S*j* 取代之。再將原 Ty 所產生的 Skolem Function 名稱更名為 RRSK，編碼亦需更改。編碼方式則是將原編碼，依照 RR 所宣告的順序 k，k = 0~m，m 為 RR 宣告的個數，將 k 加於原編碼的開頭（範例 3.3-1）。更改 Skolem Function 的編碼與名稱，是為使其成為另一個新的 Skolem Function，目的是區分非遞迴路徑與遞迴路徑所產生的 Skolem Function，並使得此 RR 宣告所產生的 RRSK Function 與

其 Child Skolem Function（CSK Function）、Descendent Skolem Function （DSK Function）自成一個接於 T*y* 所產生的 RRSK Function 之下的 RRSK 迴圈，RRSK Function 迴圈的用途，在於使得函數迴圈能無限的延伸，讓即使再長的 Target 路徑都能被轉換成符合的 Source 路徑。於 3.4 節將有詳細範例說明。

【範例 3.3-1】若原編碼為 a.b.c，而此 RR 宣告為第 k 個 RR 宣告，則新 SK function的 編碼為 k.a.b.c。

而路徑 T*y* 所產生之 Skolem Function，其 CSK 與 DSK，意即與變數 T*y* 為父子或子孫關係之變數所產生的 Skolem Function，亦需再產生新的 Skolem Function。只需將 CSK 與 DSK 中的變數宣告 S*i* 取代為變數 S*j* ，再將其它與 S*i* 相關的變數更改為 S*j*（變數的宣告的父變數或葉元素的對應中有出現 S*i* 者，皆更改為 S*j*），並於原編碼之前加入範例 3.3-1 所說明的編碼方式即可。如範例 3.3-2。

【範例 3.3-2】承範例 3.3-2，若原編碼為 a.b.c 的 Skolem Function A，其 CSK 編碼為 a.b.c.d，則在處理 RR 時，此 CSK Function 所產生的 RRSK Function 編碼為 k.a.b.c.d。

3.3 自 Mapping 產生 Skolem Function 的步驟與範例

產生 SK 的方法，我們依循下列幾個步驟：

1、 自 Mapping 的 Exist 定義，取出變數宣告，判斷是否屬於 RR 宣告的遞迴變數。若不是，則進行下一步驟，繼續產生 SK Function；若是，則跳至步驟 7 產生 RRSK Function。若每一個變數宣告皆處理完畢，則結束產生 SK Function。

2、 依據變數於 Mapping 中的出現順序與上下層關係，我們利用 Dewey 編碼給予其唯一的編碼。

3、 若此變數之路徑宣告有父變數，則繼承父變數所產生之 SK Function 的變數宣告，若無則繼續進行下一步驟。

4、 依據變數於 Mapping 中，Exist 所定義的子變數或子孫變數，加入子

變數與子孫變數宣告路徑的子路徑與 SK Function 的對應，以及 With 所定義的葉元素對應，產生 SK Function 的 RETURN 資料。再加入所使用的 Source 變數至 SK Function 的 FOR 變數宣告中。

5、 最後再依據所使用的 Source 變數，判斷是否需要加入 Mapping 中 Foreach 所定義的 Where 內部連結限制式。若無內部連結限制式則無需考慮。加入的判斷依據是，若連結限制式左右兩邊的變數有任何一者出現於此 SK Function 中，則必需加入。加入的同時，需把另一邊的變數宣告一同加入 SK 的 FOR 變數宣告中。

6、 回到步驟1，繼續處理下一個變數。

7、 依據 RR 的宣告，取得 T*y* 所產生的 SK 的 RETURN 資訊，並加入 S*j* 變數宣告，再將 SK Function 中，該使用到變數 S*i* 者，以 S*j* 取代。最後將 SK 名稱更改為 RRSK，並依照範例 3.3-1 的方式，給予新的編碼。

8、 對 T*y* 所產生的 SK Function 之 CSK Function 與 DSK Function，將其 FOR 之宣告中的 S*i* 以 S*j* 取代，並將其他變數宣告之父變數為 S*i* 者，父變數以 S*j* 取代。最後再將 SK 名稱更改為 RRSK，並依照範例 3.3-2 的方式給予新的編碼。若是 CSK Function 或 DSK Function 本身已經是 RRSK，則不再進行取代與產生新的 RRSK Function。

9、 回到步驟1，繼續處理下一個變數。

以圖 2.1 的 Target DTD 為例，RR（T*x*, T*y*, S*i*, S*j*）宣告中，T*y* 的變數宣告為 $T0 in /dept/course，其所產生的 SK Function 之 Dewey 編碼為 SK0，T*x* 的變數宣告為 $T1 in $T0//prereq/course，則 T*x* 所產生的 RRSK Function的 Dewey 編碼為 RRSK0.0。而 SK0 的 CSK Function 與 DSK Function 有：RRSK0.0、SK0.0、SK0.0.0，因此 S*j* 所產生的 RRSK0.0 的 CSK Function 與 DSK Function 應為 RRSK0.0.0、RRSK0.0.0.0，與自己本身的 RRSK0.0。在此論文中，我們使 Recursive Element 所產生的 Recursive Skolem Function，與其

下之 CSK、DSK 自成一獨立的迴圈，如圖 3.6：

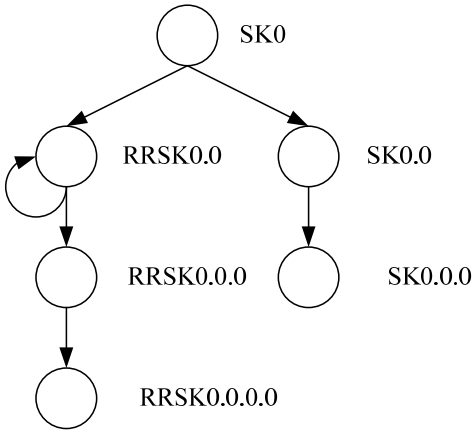


圖 3.6 Skolem Function Graph

以圖 3.6 所產生的 RRSK0.0 迴圈，可以針對循環的路徑，如：//prereq/course/…/prereq/course，進行路徑轉換，關於將路徑轉換為由 Step("/" 或 "//")與 SK Function 所組成的 SKPath 方法，將在圖 4.10 TranslatePath2SKPath 演算法介紹。

以下我們列舉由圖 3.5 之 Mapping 所產生的 SK Function 與 RRSK Function：並提出其中兩個 Function 做為代表加以解釋。

由變數 $T0 in /dept/course 所產生的 Skolem Function，SK0，內容如下：

```
for   $S0 in /dept/course
return /cno = $S0/cno, /title = $S0/title, /prereq/course = RRSK0.0,
       /takenBy/student = SK0.0
```

SK0 是由 Mapping 的 exist 資訊中的變數路徑宣告，$T0 in /dept/course，所產生，因無父變數，因此不需繼承別的 SK Function 所產生的 FOR 變數與 WHERE 條件限制。由 Mapping 中得知，$T0 於 with 中所對應葉元素有兩筆資訊，分別為：$T0/cno = $S0/cno 與 $T0/title = $S0/title，因此只需要宣告 Mapping 中 Foreach 所定義的 $S0 in /dept/course 即可，並由此資訊得知，於 Skolem Function 的 return 中需加入兩個葉元素的對應，Target DTD中的 /cno 與 /title 葉元素，分別對應到 $S0/cno 與 $S0/title。由於 Mapping 中 Foreach 所定義的內部限制（where）並無關於 $S0 的限制，因此在此 SK 中沒有

WHERE 限制。再由 Mapping 中 Exist 所定義的變數中得知，有兩個變數，$T1 與 $T2，其上層變數皆為 $T0，因此，這兩個變數所產生的 SK Function 資訊亦要加入 RETURN 資訊中，並分別對應到變數所生成的 SK Function。由於 $T1 於 RR($T1, $T0, $S0, $S4)中被定義，因此給予編碼 RRSK0.0。而 $T2 沒有 RR 的相關定義，且於 Mapping 中，屬於宣告以 $T0 為父變數的第一個非遞迴定義之變數，因此依照 Dewey 編碼方式，給予編碼為 SK0.0。

由變數 $T1 in $T0//prereq/course 所產生的 SK Function，由於該變數符合 RR 的定義，因此產生的 SK Function 為 RRSK Function，RRSK0.0，內容如下：

```
for   $S4 in //prereq/course
return /cno = $S4/cno, /title = $S4/title, /prereq/course = RRSK0.0,
            /takenBy/student = RRSK0.0.0
```

因 $T1 符合 RR($T1, $T0, $S0, $S4)所設定的條件，代表其為 recursive to recursive mapping。由於符合 RR 條件，表示 $T1 在 DTD 中遞回到 $T0 所指向的元素上，因此 RRSK0.0 先繼承 SK0 的 RETURN 資訊，但若有變數為 $S0 者，皆需取代為 $S4，且 $S4 所宣告的父變數必須被移除，以符合遞迴 Skolem Function 的精神。但由於 $T0 下有 $T1 與 $T2 兩個變數所產生的 SK Function，在 RRSK Function 產生的狀況下，需另外產生適合的 RRSK Function 給予此兩變數，因 $T1 已被標記為 RRSK0.0，因此 RRSK 直接延用，不需再重新產生；因此，RETURN 資訊中只有 /takenBy/student = SK0.0 修改為 RRSK0.0.0。

其它 SK Function 則依照於 Mapping 中的次序產生。分別如下：

變數 $T2 in $T0/takenBy/student 產生的 SK Function，SK0.0，內容如下：

```
for   $S0 in /dept/course,
      $S1 in $S0/takenBy,
      $S2 in /dept/student
where $S1/sno = $S2/sno
return /sno = $S2/sno, /name = $S2/name,
            /qualified/course = SK0.0.0
```

變數 $T3 in $T2/qualified/course 產生的 SK Function，SK0.0.0，內容如下：

```
for   $S0 in /dept/course,
      $S1 in $S0/takenBy,
      $S2 in /dept/student,
      $S3 in $S2/qualifieds/cno
where $S1/sno = $S2/sno
return /cno = $S3
```

以下為 RRSK0.0 針對 SK0 的 CSK、DSK 重新產生符合的新 Skolem Function，對路徑為 //prereq/course/takenBy/student 產生的RRSK Function，RRSK0.0.0，內容如下：

```
for   $S4 in //prereq/course,
      $S1 in $S4/takenBy,
      $S2 in /dept/student
where $S1/sno = $S2/sno
return /sno = $S2/sno, /name = $S2/name,
            /qualified/course = RRSK0.0.0.0
```

對路徑為 //prereq/course/takenBy/student/qualified/course 產生的 RRSK Function，RRSK0.0.0.0，內容如下：

```
for   $S4 in //prereq/course,
      $S1 in $S4/takenBy,
      $S2 in /dept/student,
      $S4 in $S2/qualifieds/cno
where $S1/sno = $S2/sno
return /cno = $S4
```

由以上的 SK Function 的資料，將運用於 Target Query 轉換為 Source Query 時的便利工具。

第四章、 轉換模組與演算法

在本章中，將介紹轉換的演算法。在轉換的過程中，將利用第三章所設計的 Mapping，產生轉換時所需要的 Skolem Function，並由給予的 XML 文件深度資訊，取得每 Skolem Function所能到達的 Skolem function 資訊(Tracing Information），並建立對照表格，再進一步進行查詢句轉換的處理。透過查詢句中變數所定義的路徑，以及 Tracing Information，取得每一條路徑的解。最後再產生路徑的解集合，並轉換為 Source Query。

4.1 轉換模組架構

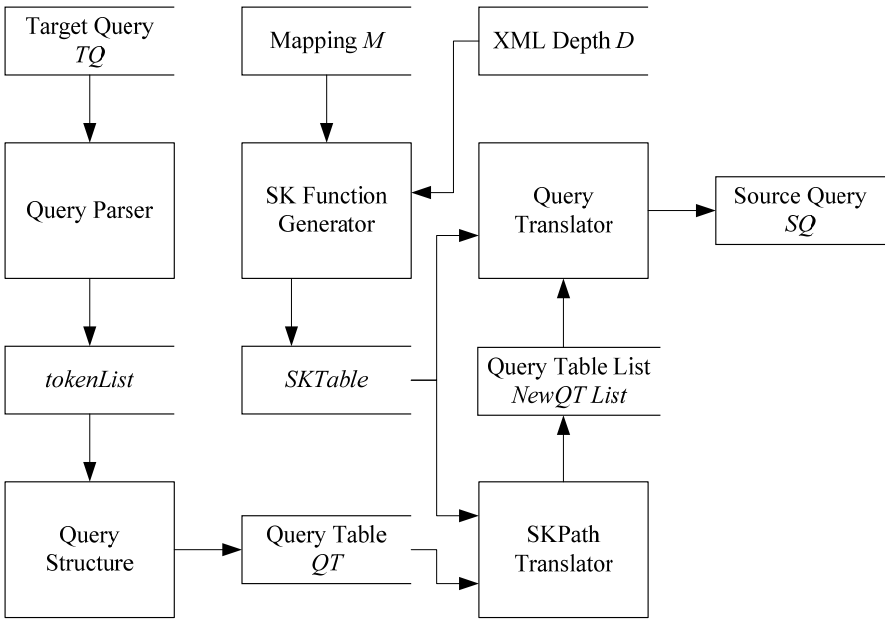


圖 4.1 Target Query 至 Source Query 轉換模組架構

圖 4.1 為整體轉換架構，此架構的中間所示，首先在轉換查詢句之前，必須先讀入 Mapping，利用 Mapping 資訊與查詢之 XML 文件深度 D 產生 SKTable，接著讀入 Target Query，使用 Query Structure 分析 Target Query 資訊並存於 Query Table 中。透過 SKTable 查表，將 Target Query 中的變數所宣告的路徑轉換為由 Step 與 SK Function 所組成的 SKPath，在此處因為路徑可能有跳層，在有 Recursive Element 的狀況下，符合的 SK Function也就可能有多個，因此我們透過記錄這些可能的解，在 Query Translator 進行展開後與篩選合

理的路徑後，透過查表將 Target Query 轉換成 Source Query。

4.2 演算法及範例說明

Algorithm Main(*TQ*, *M*)	
輸入: *TQ*，*M*，*D* 輸出: *SQTList* 變數說明: *M*, 由 DBA 所定義的 Schema Mapping *D*, 要查詢的 XML 文件深度 *SKTable*, *Path* 或 *Dewey* 為 key，Skolem Function List 為 data *TQ*, 經過一般化後的 Target Query *TokenList*, 一串 String 所組成的 List *QT*, 節構化的 Query Table，QNo 為key，Query 為 data *SQTList*, 記錄轉換成 Source Query 的表格 *PathSolutionTable*, 將路徑轉換為SK路徑後，儲存的表格	
L01	*SKTable* = SKTableGenerator(*M*, *D*);
L02	*TokenList* = QueryParser(*TQ*);
L03	*QT* = QueryStructure(*TokenList*, 0);
L04	for each *q* in *QT* do
L05	*PathSolutionTable* = TranslatePath2SKPath(*q*, *SKTable*, *QT*);
L06	end for
L07	*SQTList* = GenerateSourceQuery(*QT*, *SKTable*, *PathSolutionTable*);

圖 4.2 主程式

在本節我們說明查詢句轉換的演算法。圖 4.2 列出主程式。首先藉由 Mapping *M*、XML Depth *D*，產生 Skolem Table，*SKTable*。再讀入 Target Query *TQ*，處理字串後，儲存於 *TokenList*，於 QueryStructure 再遞迴分析處理產生 Query Table，*QT*（如圖 4.3）。該表格對 *TQ* 中的 Nested Query 出現順序分別給與流水號（註一），並以此流水號作為 Query Table *QT* 查詢之 key，而 Target Query 則為 *QT* 中儲存之 Data（圖 4.4 之 Query）。SKPathGenerator 會將 *QT* 中每一個 Query 的 for 所宣告的變數路徑，分段比照 *SKTable* ，將 *Path* 轉換為由「/」或「//」與 SK 所連接而成的 SKPathList，並更新該變數的資訊。最後，利用 *QT* 與 *SKTable* 轉換 Target Query，並輸出的 Source Query。

註一、 最外層的 Target Query 流水號為 0，其次依 Nested Query 出現順序遞增。

【範例 4.1】

```
for   $t1 in /dept/course,
      $t2 in $t1/takenBy/student
let   $t3 :=     for   $t4 in $t2/prereq/course
                 where     $t4/title = "資料庫"
                 return    $t4
for   $t5 in $t3/takenBy/student
where     $t1/title = "進階資料庫" and $t2/sno = $t5/sno
return    <ans> { $s/name } </ans>
```

以範例 4.1 的查詢句為例，會將 Query 分成 Q0 與 Q1，並儲存於 *QT*中，結果如圖 4.3 和圖 4.4 所示。

key	Query
0	Q0
1	Q1

圖 4.3 Query Table *QT*

Q0	Q1
for $c in /dept/course, $s in $c/takenBy/student let $pc := Q1 for $pcs in $pc/takenBy/student where $c/title = "進階資料庫" and $s/sno = $pcs/sno return <ans> { $s/name } </ans>	for $c1 in $c/prereq/course where $c1/title = "資料庫" return $c1

圖 4.4 Query of *Q0* and *Q1*

Algorithm SKTableGenerator(*M*, *D*)

輸入: *M*，*D*

輸出: *SKTable*

變數說明: *M*, 由DBA所定義的Schema Mapping

D, 要查詢的XML深度

SKTable, *Path*或*Dewey*為key，Skolem Function List為data

PathExp, 為*PVar* and *Path*或 *Path* only

PVar, 父變數

Path, 除了變數之外的路徑

pd, parent skolem function's dewey

圖 4.5 SKTableGenerator 演算法 (1/2)

L01	for each *Var in PathExp* in Exist Definition in *M* do
L02	if *PathExp* contain *PVar* and *Path* then
L03	get SK's dewey *pd* created from *PVar's* SK function
L04	generate *dewey* which is generated by *pd*
L05	else //path from root
L06	generate *dewey*
L07	end if
L08	generate new *SK* which is made by *Var*
L09	store *dewey* and *SK* into *SKTable*, store *Path* and SK into *SKTable*
L10	store *dewey* in *SKIDList;*
L11	end for
L12	*SKTable* = SkolemTracing(*SKTable, D, SKIDList*)

圖 4.5 SKTableGenerator 演算法 (2/2)

接下來我們說明主程式所呼叫的幾個重要的副程式。圖 4.5 所列為演算法 SKTableGenerator。該演算法依照 *M* 中 Exist的 設定，產生出所有的 Skolem Function，並依照其所對應的路徑，如 $*T1* in $*T0*/*Path* 或 $*T1* in */Path*，以 *Path* 為 *key*，*SK* 為資料，存於 *SKTable* 中。*SK* 之 *dewey* 產生方式需判斷是否有 *PVar* 存在，若有，則依 *PVar* 所產生的 *SK* 之 *dewey* 繼續延伸，若無，則產生一個新的 *dewey*，若 *key* 已有資料存在，則附加於其後（SKList）。再以該 Skolem Function 的 *dewey* 編碼為 *key*，再將 Skolem Function 存入 *SKTable* 中。另外，將每一個 Skolem Function 的 ID（Dewey編碼），存於*SKIDList*中。

對 *SKIDList* 中的每一個 Skolem Function，依照 XML 文件深度 *D* 的限制，取得該 Skolem Function 所能到達的任何 Skolem Function，並記錄所有路徑於該Skolem Function 的 *TraceTable* 中。

以圖 3.5 之 Mapping 為例，3.3 節的 Skolem Function 為資料，產生的 SKTable 如圖 4.6 所示，以路徑及 SK + dewey 或 RRSK + dewey 為 key，Skolem Functions 組成的 SKList 為資料所產生的表格。以路徑 /takenBy/student 為例，於圖 4.6 的 SKTable 查詢後，有兩個函數符合此路徑，SK0.0 與 RRSK0.0.0。

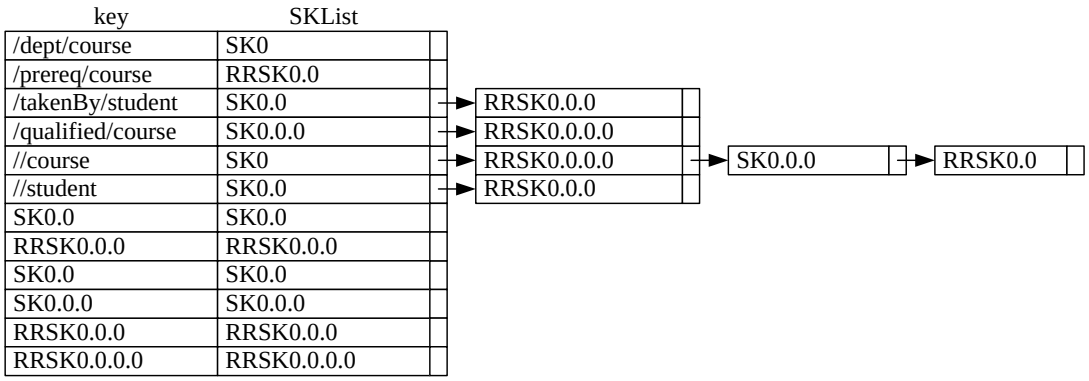


圖 4.6 SKTable

Algorithm QueryStructure(*TokenList*, *CurrentLevel*)	
輸入: *TokenList* 輸出: *QT* 變數說明: *TokenList*, 一串String所組成的List *QT*, 節構化的Query Table，QNO為key，Query為data *t*, 單一字串 *type*, 記錄該變數為for或let宣告 *PathExp*, 為*PVar* and *Path*或 *Path* only *Var*, 變數 *PVar*, 父變數 *Path*, 除了變數之外的路徑 *VarTable*, 變數表格 *ForList*, 記錄*Var*出現的順序 *CurrentLevel*, 現在Query的流水號 *Comparison*, "=" | "!=" | "<" | "<=" | ">" | ">=" *WhereList*,以 Var1, Path1, Comparison, Var2, Path2 的格式儲存 *Tag*, Tag name	
L01	while *TokenList* not empty do
L02	*t* = TokenList.popfirst( );
L03	switch(t) {
L04	case "for":
L05	*type* = "for";
L06	pop (*Var in PathExp)* from *TokenList*;
L07	(*PVar*, *Path)* = splitVar(*PathExp*);
L08	store *Var*, *PVar*, *Path*, *type* into *VarTable*;
L09	store *Var* into *ForList*;

圖 4.7 QueryStructure 演算法 (1/3)

```
L10     case "let":
L11         type = "let";
L12         pop Var := PathExp from TokenList;
L13         if PathExp eq "for" then
L14             store Var, "NQ", CurrentLevel, type in VarTable;
L15             QueryStructure(TokenList, CurrentLevel+1);
L16         end if
L17         store Var in ForList;
L18     case "where":
L19         pop (PathExp1 Comparison PathExp2) from TokenList;
L21         (Var1, Path)1 = split(PathExp1);
L22         (Var2, Path2) = split(PathExp2);
L23         store Var1, Path1, Comparison, Var2, Path2 into WhereList ;
L24     case "return":
L25         t = TokenList.popfirst( );
L26         if t is <Tag> then
L27             create ReturnTree R, name by Tag;
L28             current node = R;
L29             while </Tag> not appear do
L30                 if <Tagi> then
L31                     create Return Tree node C as child of current node;
L32                     current node = C;
L33                 else if </Tagi> then
L34                     current node = C's parent node
L35                 else if {Var1, Var2, …} then
L36                     store Variables as children to current node
L38                 else {Nested Query}
L39                     QueryStructure(TokenList, CurrentLevel+1);
L41                     store ("NQ", CurrentLevel+1) as children into
L42                         current node
L43                 end if
L44             end while
L45         else if t is "{" then
L46             create dummy node of Return Tree R
L47             while "}" not appear do
L48                 store Variable as child into R
L49             end while
```

圖 4.7 QueryStructure 演算法 (2/3)

L50	else
L51	create dummy node of Return Tree *R*
L52	pop PathExp from *TokenList*;
L53	store (*Var*, *Path*) or *Path* only into R
L54	end if
L55	}
L56	end while

圖 4.7 QueryStructure 演算法 (3/3)

圖 4.7 列舉的演算法 QueryStructure，會分析整個查詢句的結構，基本上它會分析輸入之 *TokenList* 中的文字，並將資訊分為三大類，for、let 分為第一類，where 分為第二類，return 分為第三類，以便分類處理。整體來講，此演算法會將 for 與 let 之後的資訊，分析變數、父變數、路徑、型態（for 或 let）儲存於當前 Query 的 *VarTable*，並於 *ForList* 記錄變數出現的順序；將 where 所給的條件限制式或連結限制式，存於當前 Query 的 Where List，若為條件限制式時，記錄於 Where 資訊中的 *Var* 為空字串；將 return 中的 tag name 與 { … } 所定義之變數路徑、巢狀表示式等資訊，存於當前 Query 的 Return Tree，若為 tag，則 <type, var, path> 之儲存資訊為 <"tag", "", tag_name>，若為 { … }，當內容非 Nested Query，則 <type, var, path> 之儲存資訊為 <var, var_name, path>；若內容為 Query 時，則 <type, var, path> 之儲存資訊為 <"NQ", "", *QNo*>，並將該 Query 與 *QNo* 存入 *QT*，並遞迴建構 Nested Query，直至 Query 的結尾。

以範例 4.1 的查詢句為例，透過 QueryStructure 演算法後，該所查詢句所分析產生的 *Q0* 與 *Q1* 如圖 4.8 與圖 4.9 所示：

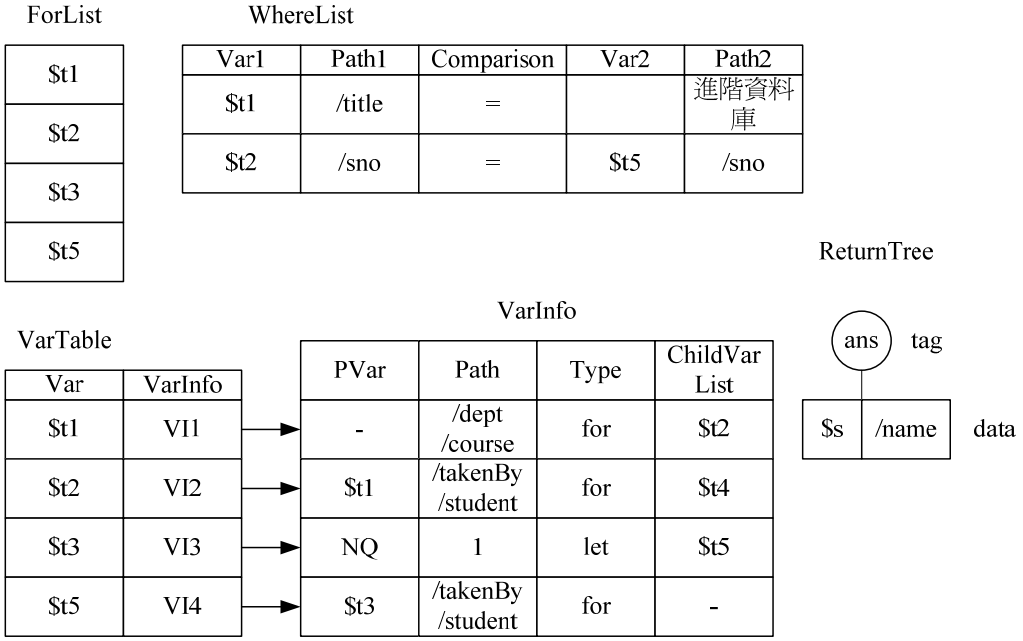


圖 4.8 *Q0* Data Structure

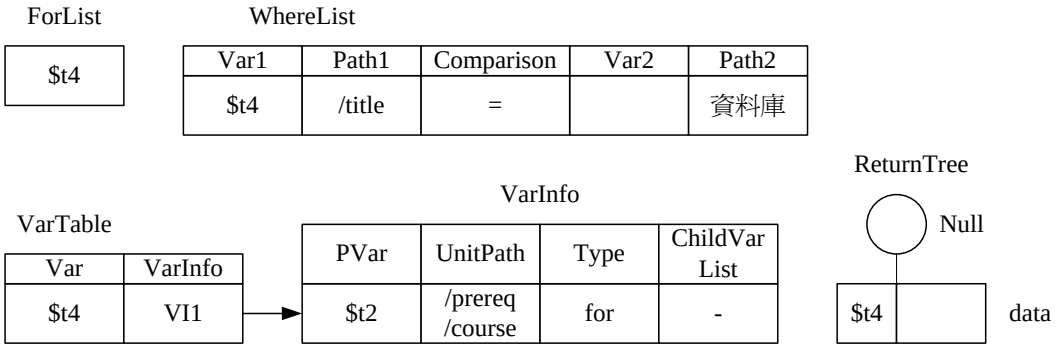


圖 4.9 *Q1* Data Structure

將查詢句的結構分解儲存之後，接著主程式會呼叫 TranslatePath2SKPath，該演算法列於圖 4.10 。此演算法根據取得 Query *q* 中的 Variable Table *VT*，取得 *VT* 中的每一筆資料（*var, VarInfo*），如果是Nested Query，則不處理此變數。否則，利用 GeneratePathSolution 分析 *VarInfo* 中的*UnitPathList*，並查詢SKTable，取得所有可能的解存於 *SKPathList*，並更新 *VarInfo* 的資訊，再將所得到的 *SKPathList* 存入 *PathSolutionTable*。

此時只把所有可能符合的 Skolem Function 加入，產生 *SKPathList (*Skolem Path List)，正確性則在將 *PathSolutionTable* 展開成 *SolutionTable* 時，才會進行

判斷。

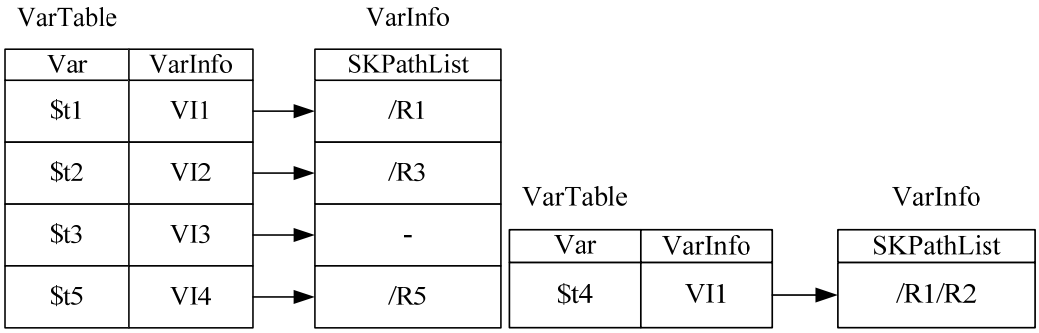
Algorithm TranslatePath2SKPath(*QT, SKTable*)

輸入: *QT*，*SKTable*
輸出: *PathSolutionTable*
變數說明: *QT*, 結構化的 Query Table，以 QNO 為 key，Query 為 data 組成的表格
SKTable, 以 *path* 或 *Dewey* 為 key，Skolem Function 為 data 組成的表格
var, 變數名稱，在Var Table中，是以 *var*為 key，*VarInfo* 為 data
VarInfo, struct VARINFO 所定義的結構，用以儲存變數資訊
SKPathList, 儲存變數之路徑所對應的 SKs List
PathSolutionTable, 儲存變數與 SKPathList 的對應表格

```
L01  PathSolutionTable = [ ];
L02  foreach (q in QT) do
L03      VT = q.GetVarTable( );
L04      foreach (var, VarInfo) in VT do
L05          if VarInfo.PVar is "NQ" then  continue;
L06          else
L07              SKPathList =  GeneratePathSolution(VarInfo.Path, SKTable);
L08              update VarInfo, store SKPathList to PathSolutionTable
L09          end if
L10      end do
L11  end do
L12  return PathSolutionTable;
```

圖 4.10 TranslatePath2SKPath 演算法

以圖 4.8 的 *Q0* 與圖 4.9 的 *Q1* 為例，其 *VarTable* 裡每一個 *VarInfo* 在透過 TranslatePath2SKPath 產生 *SKPathList* 之後，更新後的結果如圖 4.11。如下，在此只列出 VarInfo 新增的部份：



（a）Q0 之 VarTable　　（b）Q1 之 VarTable

圖 4.11 VarTable

Algorithm GeneratePathSolution(*Path, SKTable*)	
輸入	*Path*，*SKTable*
輸出	*SKPathList*
變數說明:	*SKTable*, 以 *path* 或 *Dewey* 為 key，Skolem Function 為 data 組成的表格
	Path, 等待處理的 Target Path
	SKPathList, 由（*step, skl*）所組成的列表，記錄 *Path* 轉換成 *SK* 後的資訊
	step, 跳層 "//" 或非跳層 "/"

```
L01  while Path not empty do
L02      (step, element) = PopFirst(Path);
L03      if step eq "//" and path not empty then
L04          Path Error or Normalize Error, exist some path can't match SK;
L05      else
L06          append step + element to p;
L07          if p match skl in SKTable then
L08              addd (step, skl) into SKPathList;
L09              clear path;
L10          else continue;
L11          end if
L12      end if
L13  end while
L14  return SKPathList
```

圖 4.12 GeneratePathSolution 演算法

於圖 4.10 的程式中，會呼叫圖 4.12 的 GeneratePathSolution 演算法，此演算法會自 *Path* 取出第一筆 (*step, element*)，並在 *SKTable* 中尋找是否有符合的 Skolem Function List (*skl)*，有找到則將（*step, skl*）加入 *SKPathList* 中，否則

，表示 Query 的路徑有誤。若不為跳層路徑，因路徑為一般路徑，在尚未找到任何符合的 *skl* 時，重覆將 *Path* 中的路徑取出，並加入 *p* 中，直到找到符合的*skl*，再將（*step, skl*）加入 *SKPathList* 中；若為跳層路徑，則在找到符合的 *skl* 後，將資訊加入 *SKPathList* 中。最後結果將如圖 4.11 中所示之 SKPathList。

Algorithm GenerateSourceQuery(*QT, SKTable*)	
輸入　*QT*，*SKTable* 輸出　*SQTList* 變數說明　*QT*, 節構化的 Query Table，以 QNO 為 key，Query 為 data 組成的表格 *SKTable*, 以 *path* 或 *Dewey* 為 key，Skolem Function 為 data 組成的表格 *SolutionList*, 以 *QNo* 做為分層的依據, *Var*, 與 *SKPath* 為資料所組合而成的表格的列表 *VarHeadList*, 儲存最上層的待處理變數 *STList*, Solution Table List，由多個 Solution Table 組成，每個 Solution Table 皆可以轉換出一個 Source Query *TS*, 記錄 Target Var 所產生的 Source Variable List *SQT*, 記錄轉換成 Source Query 的表格	
L01	*VarHeadList* = GenerateFirstVarList(*QT*);
L02	*SolutionList* = CreateVarAndSKSolution(*VarHeadList, QT, SKTable*);
L03	*STList* = GenerateAllSKPathSolution(*SolutionList, QT*);
L04	foreach (*st* in *STList*) do
L05	SQueryTable *SQT* = TranslateSKPath2SourcePath(*QT, st, SKTable, TS*);
L06	*SQT* = TranslateTargetQuery2SourceQuery(*QT, SKTable, TS, st*);
L07	store *SQT* in *SQTList*;
L08	end do
L09	return *SQTList*;

圖 4.13 GenerateSourceQuery 演算法

最後，主程式會呼叫 GenerateSourceQuery 產生最後的查詢句結果，如圖 4.13 所示，轉換成 Source Query 時，需先自 *QT* 中取出最上層的變數，在此指的是沒有 *PVar* 的變數，再利用此資訊連同 *QT* 中所儲存的變數資訊，取出先前圖 4.10 的演算法中所產生之 PathSolutionTable，與 *SKTable* 中的函數資訊，遞迴產生每一個 Target 變數的可能之解集合 SolutionList，再將此集合展開。將每一種解存入 *STList* 中，最後再依照 *STList* 中所記錄的解，將 Target Variable

與 *SKPath* 轉換成可用之 Source Variable 與 Source Path，存於 *TS* 中，並將 *SKPath* 轉換中所出現的連結限制式先存入 SQueryTable *SQT* 中，最後利用查表的方式，將 *QT* 中的每一筆資料透過查詢 *TS* 表格，將 *QT* 中的資訊都轉換成 Source Variable 與 Source Path，並存在相對應的 *SQT* 中，再將 *SQT* 存入 *SQTList*，於所有的解都轉換完成後輸出結果。

Algorithm GenerateFirstVarList(*QT*)	
輸入　*QT* 輸出　*VarList* 變數說明　*QT*, 節構化的 Query Table，以 QNO 為 key，Query 為 data 組成的表格 *VarList*, 儲存最上層的待處理變數	
L01	*VarList* = [];
L02	foreach (q in QT) do
L03	foreach (var in q.GetForList( )) do
L04	if(var has no parent var) then
L05	add var to *VarList*;
L06	end do
L07	end do
L08	return *VarList*;

圖 4.14 GenerateFirstVarList 演算法

圖 4.13 的程式中，會呼叫圖 4.14 的 GenerateFirstVarList 演算法，此演算法會判斷 Query Table 中的每一個變數，將沒有父變數的變數記錄於 *VarList* 中，最後傳回 *VarList*。

在取得 VarList 後，圖 4.13 的演算法會接著呼叫 圖 4.15 的演算法，將 *VarList* 中所列之變數進行所有符合之 *SKPath* 的轉換，產生每個變數路徑所對應的 Skolem Function 之所有解，並將結果存在 *SolutionTable* 中。並將此 *SolutionTable* 中的所有變數與符合的 *SKPath* 結果展開，存入 *SolutionList*。接著取得新的 *VarList* 產生新的 *SolutionTable*，依照先前所產生的 *SolutionList* 中的每一個解，判斷下層變數所展開的 *SKPath* 的正確性，並將正確的結果存入 *tempSolutionList*，並將此結果與 *Solution* 進行結合後存於 *NewSolutionList*。依據每一筆 *SolutionList* 中的 Solution 產生結果後，取得新的結果

NewSolutionList。若此 VarList 中的變數仍有其它下層變數，則將 *NewSolutionList* 當作新的 *SolutionList* 繼續重覆處理，直到不再有下層變數，則回傳此 New*SolutionList*。

Algorithm CreateVarAndSKSolution(*VarHeadList*, *QT*, *SKTable*)	
輸入	*VarHeadList*, *QT*, *SKTable*
輸出	*SolutionList*
變數說明	*VarList*, 儲存最上層的待處理變數 *QT*, 節構化的 Query Table，以 QNO 為 key，Query 為 data 組成的表格 *SKTable*, 以 *path* 或 *Dewey* 為 key，Skolem Function 為 data 組成的表格 *SolutionList*, 以 *QNo* 做為分層的依據, *Var*, 與 *SKPath* 為資料所組合而成的表格的列表 *SolutionTable*, 以 *QNo* 做為分層的依據, *Var*, 與 *SKPath List* 為資料組組成的表格 *tempSolutionList*, 同 *SolutionList*, 用於暫存展開之結果
L01	*emptySolutionList* = [ ];
L02	*SolutionTable* = GenerateSolutionTable(*VarList*, *QT*);
L03	*SolutionList* = UnfoldSolutionTable(*SolutionTable*, *QT*);
L04	*VarList* = GenerateNextVarList(*VarList*, *QT*);
L05	while(*VarList* not empty) do
L06	*SolutionTable* = GenerateSolutionTable(*VarList*, *QT*);
L07	if (*SolutionTable* not empty) then
L08	foreach (*s* in *SolutionList*) do
L09	*tempSolutionList* =
L10	UnfoldSolutionTable(*SolutionTable*, *s*, *QT*);
L11	CombineSolution(*s*, *tempSolutionList*, *NewSolutionList*);
L12	end do
L13	*SolutionList* = *NewSolutionList*;
L14	end if
L15	*VarList* = GenerateNextVarList(*VarList*, *QT*);
L16	end do
L17	return *SolutionList*;

圖 4.15 CreateVarAndSKSolution 演算法

在取得 *SolutionList* 之後，圖 4.13 的演算法會接著呼叫圖 4.16 的演算法，將資料轉存入 STList中。資料將以 Pair(*QNo*, *Var*）為 key，*SKPath* 為 Value

存入 *STList* 中。

Algorithm GenerateAllSKPathSolution(*SolutionList*, *QT*)	
輸入	*SolutionList*, *QT*
輸出	*STList*
變數說明	*SolutionList*, 以 *QNo* 做為分層的依據, *Var*, 與 *SKPath* 為資料所組合而成的表格的列表 *QT*, 節構化的 Query Table，以 QNO 為 key，Query 為 data 組成的表格 *STList*, 儲存展開後的所有變數與 SKPath 資訊的表格的列表 *ST*, 儲存展開後的所有變數與 SKPath 資訊的表格 *var*, Target Var *level*, *var* 所在的 Target Query No *skpath*, 由(step, sk)組合成的資訊
L01	*STList* = [ ];
L02	foreach (s in *SolutionList*) do
L03	foreach ((*level*, *var*, *skpath*) in *s*) do
L04	add ((*level*, *var*), *skpath*) to *ST*;
L05	end do
L06	add *ST* to *STList*;
L07	clear *ST*;
L08	end do
L09	return *STList*

圖 4.16 GenerateAllSKPathSolution 演算法

取得轉換成所需要的資料結構的 *STList* 之後，圖 4.13 的演算法會接著呼叫圖 4.17 的演算法，依照 *STList* 中的每一筆 Solution，將 Target Query 中的 Target 變數轉換為 Source 變數，並存入 Source Query中，再將每一筆 Solution 所產生的 Source Query，此時僅有將 Skolem Function 中的 where condition，存入 *SQT* 中，Source 變數的加入與 Target where 與 return 中的 Target Path 的轉換將於圖 4.18 的演算法中處理。在將 Target 變數轉換為 Source 變數的同時，會利用 *TS* 記錄 Target 變數透過 Skolem Function 產生了哪些 Source 變數，使其子孫變數在進行變數轉換時，能透過 *TS* 取得父變數已經產生了哪些 Skolem Function 中的變數，並得知該變數於該 Skolem Function 中仍須產生的變數有哪些。

Algorithm TranslateSKPath2SourcePath(*QT, st, SKTable, TS*)	
輸入 *QT, st, SKTable, TS* 輸出 *SQT* 變數說明 *st,* 以 *QNo* 做為分層的依據, *Var,* 與 *SKPath* 為資料所組合而成的表格 *QT,* 節構化的 Query Table，以 QNO 為 key，Query 為 data 組成的表格 *SKTable,* 以 *path* 或 *Dewey* 為 key，Skolem Function 為 data 組成的表格 *TS,* 記錄 Target Var 所產生的 Source Variable List *SQT,* 記錄轉換成 Source Query 的表格 *SQ,* Source Query *var,* 當前要處理的變數 *skp, var* 所對應的 SKPath *PVar, var* 的父變數 *preSK,* 父變數的 SKPath 解的最後一筆 SK	
L01	*SQ* = [ ];
L02	foreach ((*QNo, q*) in *QT*) do
L03	foreach (*var* in *q*.GetForList( )) do
L04	if (*var* 有父變數，且父變數或其祖先變數，是由 let 所宣告的
L05	，且其 Path Expression 為一 Query) then
L06	continue;
L07	*skp* = *st*.GetSKPath(*var*);
L08	if (*var* has *PVar*) then
L09	*PQNo* = *q*.FindPVarQNo(*PVar*)
L10	*preSK* = *st*.GetVarLastSK(*PVar*);
L11	(*TS, SQ*) = GenerateNewVar(*var, PVar, preSK, skp, QNo, PQNo*)
L12	end do
L13	*SQT*.add(*QNo, SQ*);
L14	end do
	return *SQT*;

圖 4.17 TranslateSKPath2SourcePath 演算法

圖 4.17 的演算法會呼叫圖 4.18 的演算法來產生 Source Var 列表，並回傳 Target 變數與其產生的 Source 變數與路徑對照表 *TS*，並將每個變數所對應的 Skolem Function 中的 where condition 先加入 SQ 中。在 L03 ~ L18 處理無 *preSK* 的變數路徑所產生的 *skp*，除了將每個 *sk* 中的變數更名後加入 *sv* 中以外，還需判斷 *sk* 中是否有 where condition存在，因該 *sk* 並無 *preSK*，因此取

得更名後的變數後，即可直接將 where condition 加入 SQ 的 where 中。

Algorithm GenerateNewVar(*var, PVar, preSK, skp, QNo, PQNo*)

輸入　*var, PVar, preSK, skp, QNo, PQNo*
輸出　*TS, SQ*
變數說明
var, 當前要處理的變數
PVar, *var* 的父變數
preSK, 父變數的 SKPath 解的最後一筆 SK
skp, *var* 所對應的 SKPath
QNo, 變數所在的 Query 之 ID
PQNo, 父變數所在的 Query 之 ID
sv, Source Var Table
counter, 計算 *skp* 中有多少個sk
step, "/" or "//"
sk, Skolem Function
VarDef, 變數與變數宣告的路徑，包含 PVar 與 Path
newSVar, 更名後的 Source Var
SPVar, Source Parent Var
NewPath, 依照 *step* 與 *sk* 產生的路徑
wl, where list
newWL, 修正變數之後的 where condition list
TS, 記錄 Target Var 所產生的 Source Variables，
以 (QNo, TVar) 為 key, Source Var Table 為 value
SQ, Source Query

```
L01  sv = [ ]; counter = 0;
L02  foreach ((step, sk) in skp) do
L03      if(preSK is empty) then
L04          foreach (VarDef in sk.GetForList( )) do
L05              newSVar = VarDef.Var + "_" + var + "_" + counter;
L06              if(VarDef.PVar not empty) then
L07                  SPVar = TS.FindSPVar(TS, QNo, VarDef.PVar);
L08              else SPVar = [];
L09              if(step eq "//") then
L10                  NewPath = "//" + _GetLastStep(VarDef.Path);
L11              else NewPath = VarDef.Path;
L12              sv.add(newSVar, SPVar, NewPath);
L13              TS.add(QNo, Var, sv);
L14          end do
```

圖 4.18 GenerateNewVar演算法 (1/3)

```
L15             if( (wl = sk.GetWhereList( )) not empty) then
L16                 newWL = Change2NewVar(wl, sv);
L17                 SQ.add2WhereList(newWL);
L18             end if
L19         else if(preSK.GetType( ) eq sk.GetType( )) then
L21             if(sk.GetType( ) eq "RRSK") then
L22                 if(step eq "/"  且 preSK 為 sk 的 Parent SK)
L23                     SameSKType_1Step_VarGenerator(var, PVar, sk, sv, TS,
L24                         QNo, PQNo)
L25                     AddWhereCondition(SQ, QNo, sk, preSK, 0);
L26                 else if(step eq "/" 且 sk 為遞迴至上層之 RRSK )
L27                     RRSK2RRSK_Recursive_VarGenerator(var, PVar, sk,
L28                         sv, TS, QNo, PQNo)
L29                     AddWhereCondition(SQ, QNo, sk, preSK, 1);
L30                 else if(step eq "//" )
L31                     if(sk 不為遞迴至上層之 RRSK)
L32                         SameSKType_JumpStep_VarGenerator(var, PVar,
L33                             sk, sv, TS, QNo, PQNo);
L34                         AddWhereCondition(SQ, QNo, sk, tempSK, 0);
L35                     else if(preSK.GetID( ) eq sk.GetID( ))
L36                         RRSK2RRSK_Recursive_VarGenerator(var, PVar,
L38                             sk, sv, TS, QNo, PQNo)
L39                         AddWhereCondition(SQ, QNo, sk, preSK, 1);
L41                     else   // sk 為遞迴至上層之 RRSK
L42                         tempSK = 取得 preSK 至遞迴 SK 之前一 SK
L43                         SameSKType_1Step_VarGenerator(var, PVar,
L44                             tempSK, sv, TS, QNo, PQNo);
L45                         AddWhereCondition(SQ, QNo, tempSK, preSK, 0);
L46                         SameSKType_JumpStep_VarGenerator(var, PVar,
L47                             sk, sv, TS, QNo, PQNo);
L48                         AddWhereCondition(SQ, QNo, sk, tempSK, 1);
L49                     end if
L50                 end id
```

圖 4.18 GenerateNewVar演算法 (2/3)

在 L19 ~ L59 處理有相同 SK Type 的 *sk* 與 *preSK*，即同為 SK 或 RRSK。在 L22 ~ L50 處理同為 RRSK 的狀況，分為三類，第一類（L22 ~ L25）：*preSK* 為 *sk* 的 Parent RRSK。第二類（L26 ~ L29）：*sk* 為遞迴至上層的 RRSK。前二

類的兩個 SK 距離為 1，而第三類（L30 ~ L50）與前者不同，是自 *preSK* 跳層至 *sk*。

因此第三類再細分為：1. *preSK* 至 *sk* 不為遞迴至上層的 RRSK；2. *preSK* 與 *sk* 為同一個 SK，即遞迴至自己本身；3. 遞迴至上層的 RRSK。在不同的狀況下，運用不同的方式產生 Source 變數路徑，以及將變數的後存入 *sv* 中。並判斷是否需要加入 SK 中所限制的 where 聯結限制式，因該限制式可能已被 *preSK* 加入，須避免重覆加入。在 L51 ~ L59 處理 *preSK* 與 *sk* 皆為非遞迴 SK 的狀況，對於是否為跳層路徑，採取不同的處理方式。

此演算法中，所有處理方式主要分為：I. SameSKType，前後 SK 的型態相同；II. RRSK2RRSK，兩者皆為 RRSK；III. SK2RRSK，*preSK* 為 SK，而 *sk* 則為 RRSK。

在 I. SameSKType 的狀況下，會判斷 SK 中的變數是否已被前者所使用，來決定是否加入新變數，或繼續處理 SK 中的下一個變數。

在 II. RRSK2RRSK 的狀況下，如 L30 ~ L50 的程式所述，若不為遞迴至上層的 RRSK，則同 I. 的方式，判斷是否需要加入變數；若遞迴至自己本身，則需將 *preSK* 所產生的變數當作自己的 PVar，使前後遞迴 SK 能相互串聯；若遞迴至上層的 RRSK，則需先取得遞迴至上層 RRSK 的前一筆 RRSK（*tempSK*），並先產生 *preSK* 至 *tempSK* 的變數轉換結果，再產生由 *tempSK* 至 *sk* 的變數轉換結果。

在 III. SK2RRSK 的狀況下，如 L60 ~ L76 的程式所述，若 step 為非跳層，則 SK 與 RRSK 為 PC 關係，只須將 SK 的變數當做 RRSK 中的變數之 PVar 即可；若為跳層，則亦須先取得 SK 至 RRSK 的 SK 路徑中，的最後一筆 SK，產生轉換結果後，再加入 RRSK 的轉換結果。

而前後 SK 之間透過 1Step（PC關係的 step：“/”），或是 JumpStep（AD關係的 step：“//”），影響的是 Source 的變數路徑在產生時，採用 //Element 或是 /Path。

```
L51             else if(step eq "/")  // SK 1 step to SK
L52                     SameSKType_1Step_VarGenerator(var, PVar,
L53                         sk, sv, TS, QNo, PQNo);
L54                 else                // SK jump to SK
L55                     SameSKType_JumpStep_VarGenerator(var, PVar,
L56                             sk, sv, TS, QNo, PQNo);
L57                 end if
L58                 AddWhereCondition(SQ, QNo, sk, preSK, 0);
L59             end if
L60         else // preSK與sk的type不同，但只有preSK為SK, sk為RRSK的狀況
L61             if(step eq "/") then
L62                 SK2RRSK_1Step_VarGenerator (var, PVar,
L63                         sk, sv, TS, QNo, PQNo);
L64                 AddWhereCondition(SQ, QNo, sk, preSK, 1);
L65             else
L66                 tempSK = 取得 preSK 至遞迴 SK 之前一 SK
L67                 if(tempSK not empty) then
L68                     SameSKType_1Step_VarGenerator(var, PVar,
L69                         tempSK, sv, TS, QNo, PQNo);
L70                     AddWhereCondition(SQ, QNo, tempSK, preSK, 0);
L71                 end if
L72                 SK2RRSK_JumpStep_VarGenerator(var, PVar,
L73                         sk, sv, TS, QNo, PQNo);
L74                 AddWhereCondition(SQ, QNo, sk, preSK, 1);
L75             end if
L76         end if
L77         preSK = sk;
L78         counter++;
L79 end do
L80 TS.add(QNo, var, sv);
L81 return (TS, SQ)
```

圖 4.18 GenerateNewVar演算法 (3/3)

最後，圖 4.13 的演算法會呼叫圖 4.19 的演算法，利用圖 4.17 所產生的 Source 變數，以及 *TS*，將 *TQ* 中的 where 與 return tree 中的變數轉換為 Source Query 使用之 Source Path，並存入 *SQT* 中。

Algorithm TranslateTargetQuery2SourceQuery(*QT, SKTable, TS, st*)

輸入　*QT, SKTable, TS, st*

輸出　*SQT*

變數說明　*st*, 以 *QNo* 做為分層的依據, *Var*, 與 *SKPath* 為資料所組合而成的表格

QT, 節構化的 Query Table，以 QNO 為 key，Query 為 data 組成的表格

SKTable, 以 *path* 或 *Dewey* 為 key，Skolem Function 為 data 組成的表格

TS, 記錄 Target Var 所產生的 Source Variable List

QNo, Query ID

var, Target Var

SVList, *var* 所使用的 Source Var List

where, where condition

rt, Return Tree

newRt, Return Node

```
L01  SQT = [ ];
L02  foreach ((QNo, q) in QT) do
L03      foreach (var in q.GetForList( )) do
L04          SVList = TS.GetSourceVarDef(var);
L05          SQT.add2For(QNo, SVList);
L06      end do
L07      foreach (where in q.GetWhereList( )) do
L08          newWhere = Translate2SourceWhere(TS, where)
L09          SQT.add2Where(QNo, newWhere);
L10      end do
L11      rt = q.GetReturnTree( );
L12      newRt = TracingTreeAndTranslateTPath2SPath(rt, TS);
L13      SQT.add2Return(QNo, newRt);
L14  end do
```

圖 4.19 TranslateTargetQuery2SourceQuery 演算法

第五章、 正確性分析與轉換結果

在本章中，我們將以不同結構的查詢句，跳層及巢狀表示式的混合，來驗證查詢句轉換系統的正確性，評估輸入的查詢句與轉換出的查詢句是否具有相同的結構關係。我們並進行一系列的實驗來評估本系統的執行效率。

5.1 實驗結果，正確性討論

以下的查詢句以圖 2.1 Target DTD 做為輸入之 Schema，以圖 2.3 Source DTD 做為輸出之 Schema，其 Mapping 如圖 3.5 所示。實驗輸入之查詢句是以圖 2.1 之 DTD 做為 Schema 之查詢句，輸出之查詢句則是以圖 2.3 之 DTD 做為 Schema 之查詢句。查詢的 XML 文件深度設定為 11。

Target Query M1_Q1

```
for   $a in /dept/course,
      $b in $a/takenBy/student
let   $c :=      for   $d in $b/qualified/course,
                       $e in /dept/course
                 where     $d/cno = $e/cno
                 return    <Title> { $e/title } </Title>
where $a/title = "DataBase"
return     <result>
                 <CourseNo> { $a/cno } </CourseNo>
                 <StudentName> { $b/name } </StudentName>
                 <Qualifieds> { $c } </Qualifieds>
           </result>
```

中間結果 Nested TQuery Level 0

```
for   $a in GetSKs("/dept/course"),
      $b in GetSKs( $a, "/takenBy/student")
let   $c := Nested TQuery Level 1
where $a/title = "DataBase"
return     <result>
                 <CourseNo> { $a/cno } </CourseNo>
                 <StudentName> { $b/name } </StudentName>
                 <Qualifieds> { $c } </Qualifieds>
           </result>
```

圖 5.1 Target Query M1_Q1 (1/2)

中間結果 Nested TQuery Level 1
for $d in GetSKs($b, "/qualified/course") for $e in GetSKs("/dept/course") where $d/cno = $e/cno return <Title> { $e/title } </Title>
輸出結果
for $S0_a_0 in /dept/course, $S1_b_0 in $S0_a_0/takenBy/sno, $S2_b_0 in /dept/student let $c := for $S3_d_0 in $S2_b_0/qualifieds/cno, $S0_e_0 in /dept/course where $S3_d_0 = $S0_e_0/cno return <Title> { $S0_e_0/title } </Title> where $S1_b_0 = $S2_b_0/sno and $S0_a_0/title = "DataBase" return <result> <CourseNo> { $S0_a_0/cno } </CourseNo> <StudentName> { $S2_b_0/name } </StudentName> <Qualifieds> { $c } </Qualifieds> </result>

圖 5.1 Target Query M1_Q1 (2/2)

結果	變數	SKPath	變數	SKPath
1	$a	/SK0	$b	$a/SK0.0
	$c	-	$d	$b/SK0.0.0
	$e	/SK0		

表格 5.1 Target Query M1_Q1 轉換輸出結果之變數與 SKPath 對應

圖 5.1 的查詢句是由無跳層的路徑與巢狀表示式所組合而成，由輸出的結果可以看到，$S2_b_0 藉由 WHERE 的連結限制式 $S1_b_0/sno = $S2_b_0/sno，保存了與變數 $S0_a_0 之間的結構關係。而 LET 所宣告的巢狀表示式，其內之變數也因為外部的變數轉換而跟著轉換，得到正確的連結，並且依照 SK Function 正確的轉換成 Source Path，也同時修正了 WHERE 連結限制式中的路徑。RETURN 中的變數也得到了正確的替換。而由 Let 所宣告的變數 $c，因為所指向的是一巢狀表示式所回傳的資料，因此，相關於變數 $c 的任何路徑，皆不會因為查詢句轉換而有改變。

表格 5.1 的結果是由圖 5.1 的中間結果所產生的，目的在顯示出每一個輸

出結果的變數與函數之間的關係。變數 $a 是自Root 開始的路徑，因此只有唯一解 /SK0，而變數 $b 之父變數為 $a，因此所能對應的函數受到 $a 的限制，亦只有對應到 /SK0.0，變數 $c 因其所指向之路徑為一巢狀表示式，因此不做對應到任何函數的置換，變數 $d 之父變數為 $b，因此也受到 $b 所對應之函數影響，只有對應到 /SK0.0.0，而變數 $e 與變數 $a 的狀況相同，皆對應到 /SK0。之後的SKPath Solution 表格皆是如此。

Target Query M1_Q2

```
for   $a in //course,
      $b in $a/takenBy/student
let   $c :=      for   $d in $b/qualified/course,
                       $e in /dept/course
                 where     $d/cno = $e/cno
                 return    <Title> { $e/title } </Title>
where $a/title = "DataBase"
return     <result>
                 <CourseNo> { $a/cno } </CourseNo>
                 <StudentName> { $b/name } </StudentName>
                 <Qualifieds> { $c } </Qualifieds>
           </result>
```

中間結果 Nested TQuery Level 0

```
for $a in GetSKs("//course")
for $b in GetSKs( $a, "/takenBy/student")
let $c := Nested TQuery Level 1
where $a/title = "DataBase"
return     <result>
                 <CourseNo> { $a/cno } </CourseNo>
                 <StudentName> { $b/name } </StudentName>
                 <Qualifieds> { $c } </Qualifieds>
           </result>
```

中間結果 Nested TQuery Level 1

```
for $d in GetSKs( $b, "/qualified/course")
for $e in GetSKs("/dept/course")
where $d/cno = $e/cno
return   <Title>   { $e/title }   </Title>
```

圖 5.2 Target Query M1_Q2 (1/2)

輸出結果 1

```
for   $S0_a_0 in //course,
      $S1_b_0 in $S0_a_0/takenBy/sno,
      $S2_b_0 in /dept/student
let $c :=  for   $S3_d_0 in $S2_b_0/qualifieds/cno,
                 $S0_e_0 in /dept/course
           where $S3_d_0 = $S0_e_0/cno
           return <Title> { $S0_e_0/title } </Title>
where $S1_b_0 = $S2_b_0/sno and $S0_a_0/title = "DataBase"
return     <result>
                 <CourseNo> { $S0_a_0/cno } </CourseNo>
                 <StudentName> { $S2_b_0/name } </StudentName>
                 <Qualifieds> { $c } </Qualifieds>
           </result>
```

輸出結果 2

```
for   $S4_a_0 in //course,
      $S1_b_0 in $S4_a_0/takenBy/sno,
      $S2_b_0 in /dept/student
let $c :=  for   $S3_d_0 in $S2_b_0/qualifieds/cno,
                 $S0_e_0 in /dept/course
           where $S3_d_0 = $S0_e_0/cno
           return <Title> { $S0_e_0/title } </Title>
where $S1_b_0 = $S2_b_0/sno and $S4_a_0/title = "DataBase"
return     <result>
                 <CourseNo> { $S4_a_0/cno } </CourseNo>
                 <StudentName> { $S2_b_0/name } </StudentName>
                 <Qualifieds> { $c } </Qualifieds>
           </result>
```

圖 5.2 Target Query M1_Q2 (2/2)

結果	變數	SKPath	變數	SKPath
1	$a	/SK0	$b	$a/SK0.0
	$c	-	$d	$b/SK0.0.0
	$e	/SK0		
2	$a	/RRSK0.0	$b	$a/RRSK0.0.0
	$c	-	$d	$b/RSK0.0.0.0
	$e	/SK0		

表格 5.2 Target Query M1_Q2 轉換輸出結果之變數與 SKPath 對應

圖 5.2 的查詢句是由跳層路徑與巢狀表示式組合而成，因為查詢句中有跳

層，所以有兩個符合的解，第一個是一般路徑下得到的解，第二個解是透過 RRSK 得到的解，但由於是跳層路徑，所以第二個解因為結構問題，無法明顯看到透過RRSK轉換的結果。下面的範例將可以明顯看到透過RRSK轉換後得到的結果。

Target Query M1_Q3

```
for   $a in /dept/course,
      $b in $a/prereq/course,
      $c in $a/takenBy/student,
      $d in $b/takenBy/student
where     $c/sno = $d/sno
return    <result>
              <course_title> { $a/title } </course_title>
              <prereq_course_title> { $b/title } </prereq_course_title>
              <student_name> { $c/name } </student_name>
          </result>
```

中間結果 1

```
Nested TQuery Level 0
for $a in GetSKs("/dept/course")
for $b in GetSKs( $a, "/prereq/course")
for $c in GetSKs( $a, "/takenBy/student")
for $d in GetSKs( $b, "/takenBy/student")
where $c/sno = $d/sno
return    <result>
              <course_title> { $a/title } </course_title>
              <prereq_course_title> { $b/title } </prereq_course_title>
              <student_name> { $c/name } </student_name>
          </result>
```

圖 5.3 Target Query M1_Q3 (1/2)

輸出結果 1
for $S0_a_0 in /dept/course, $S4_b_0 in $S0_a_0/prereq/course, $S1_c_0 in $S0_a_0/takenBy/sno, $S2_c_0 in /dept/student, $S1_d_0 in $S4_b_0/takenBy/sno, $S2_d_0 in /dept/student where $S1_c_0 = $S2_c_0/sno and $S1_d_0 = $S2_d_0/sno and $S2_c_0/sno = $S2_d_0/sno return <result> <course_title> { $S0_a_0/title } </course_title> <prereq_course_title> { $S4_b_0/title } </prereq_course_title> <student_name> { $S2_c_0/name } </student_name> </result>

圖 5.3 Target Query M1_Q3 (2/2)

結果	變數	SKPath	變數	SKPath
1	$a	/SK0	$b	$a/RRSK0.0
	$c	$a/SK0.0	$d	$b/RRSK0.0.0

表格 5.3 Target Query M1_Q3 轉換輸出結果之變數與 SKPath 對應

圖 5.3 的查詢句範例，由於宣告了變數 $b 是指向 recursive element 的變數，可以看到即使是透過迴圈，依然能保持 Target Query 中，變數 $b 與 $d 之間的結構關係，並不會因為迴圈而消失。

接下來我們要測試跳層會產生的結果，利用兩個跳層，透過 //course 與 //course 經過迴圈而產生複雜的 SK Function 與 RRSK Function 的混合結果。

Target Query M1_Q4
for $a in //course return <result> { for $c in $a//course return { $c/cno } } </result>
中間結果 Nested TQuery Level 0
for $a in GetSKs("//course") return <result> { NQ1 } </result>

圖 5.4 Target Query M1_Q4 (1/3)

中間結果 Nested TQuery Level 1
for $c in GetSKs($a, "//course") return $c/cno
輸出結果 1
for $S0_a_0 in //course return <result> { for $S1_c_0 in $S0_a_0/takenBy/sno, $S2_c_0 in /dept/student, $S3_c_0 in $S2_c_0//cno where $S1_c_0 = $S2_c_0/sno return { $S3_c_0 } } </result>
輸出結果 2
for $S0_a_0 in //course return <result> {for $S4_c_0 in $S0_a_0//course return { $S4_c_0/cno } } </result>
輸出結果 3
for $S0_a_0 in //course return <result> { for $S4_c_0 in $S0_a_0//course, $S1_c_0 in $S4_c_0/takenBy/sno, $S2_c_0 in /dept/student, $S3_c_0 in $S2_c_0//cno where $S1_c_0 = $S2_c_0/sno return { $S3_c_0 } } </result>
輸出結果 4
for $S4_a_0 in //course return <result> { for $S4_c_0 in $S4_a_0//course return { $S4_c_0/cno } } </result>

圖 5.4 Target Query M1_Q4 (2/3)

輸出結果 5
for $S4_a_0 in //course return <result> { for$S1_c_0 in $S4_a_0/takenBy/sno, $S2_c_0 in /dept/student, $S3_c_0 in $S2_c_0//cno where $S1_c_0 = $S2_c_0/sno return { $S3_c_0 } } </result>

圖 5.4 Target Query M1_Q4 (3/3)

圖 5.4 輸出的結果有五項，但再經過刪減相同的結果後，會有三項。會有相同結果的原因，以輸出結果 2 與 4 為例，是因為 $a in //course 符合 SK0.0 與 RRSK0.0.0 兩個函數，而 $c in $a//course 在此兩例中，其路徑的解符合的是 RRSK0.0.0，但因 //SK0.0 與 //RRSK0.0.0 在轉換後，皆為 //course，造成結果相同，但其轉換時所使用的 SK Function 是不同的。而輸出結果 1 與 5 也是相同的狀況 $a in //course 分別符合 SK0.0 與 RRSK0.0.0，而 $c in $a//course 符合的是 RRSK0.0.0.0。與其他兩解不同的是輸出結果 3，其 $a in //course 符合的是 SK0.0，而 $c in $a//course 所符合的函數，是透過了兩個函數而產生的結果，分別是 RRSK0.0.0 與 RRSK0.0.0.0。

結果	變數	SKPath	變數	SKPath
1	$a	//SK0	$c	$a//RRSK0.0.0.0
2	$a	//SK0	$c	$a//RRSK0.0
3	$a	//SK0	$c	$a/RRSK0.0//RRSK0.0.0.0
4	$a	//RRSK0.0	$c	$a//RRSK0.0
5	$a	//RRSK0.0	$c	$a//RRSK0.0.0.0

表格 5.4 Target Query M1_Q4 轉換輸出結果之變數與 SKPath 對應

5.2 效能測試

在本節中，我們針對查詢句轉換中的主要演算法進行效能的測試，以不同的變量進行實驗。圖 4.1 之查詢句轉換模組中，SK Function Generator（圖 4.5 SKTableGenerator 演算法）產生 SkolemTracing 資訊時，會因查詢之 XML 文件深度而影響效能（見圖 5.5）；模組 Query Translator （圖 4.13

GenerateSourceQuery 演算法）為主要處理查詢句轉換的部份；其它的模組所花費的時間皆為極小值（約 1～10ms），因此不列入效能測試中。我們先對圖 4.13 GenerateSourceQuery 的演算法進行效能測試，再對其內部之演算法（圖 4.15、圖 4.16、圖 4.17、圖 4.19）進行效能測試。在 5.2.1 節中，我們測試 SkolemTracing 函數的效能，在 5.2.2 節中，我們測試查詢句轉換的效能。

5.2.1 SkolemTracing 函數的效能測試

在本節中，我們測試圖 4.5 的演算法中，位於 L12 的 SkolemTracing 函數產生時間與查詢之 XML 文件深度的關係，實驗結果會與設定的 Mapping 迴圈長度成正相關。以圖 3.5 之 Mapping 產生之 Skolem Function 做為 SkolemTracing 的輸入資料。測試時，每次以執行 1000 次為單位，測試 10 次後，再取平均值。

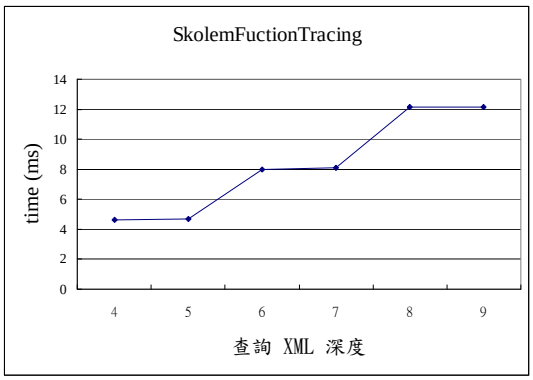


圖 5.5 SkolemTracing 演算法效率測試

圖 3.5 的 Mapping 中，遞迴的迴圈長度為 2，因此可以看到圖 5.5 中，查詢之 XML 文件深度每增加 2，所花費的時間就有顯著增加，可以得知在產生由任意 Skolem function 到達其它任意 Skolem Function 的路徑資料時，所花費的時間是隨著迴圈長度與搜索深度而增加。

5.2.2 產生 Source Query 效能測試

首先我們列出在下列實驗所使用的主要查詢句範例，我們藉由調整（1）變數個數、（2）跳層個數、（3）變數間有上下層關係，來測試以上三者對於轉換效率之影響。以下四個實驗分別使用四個範例查詢句作為基礎，並加以延伸。

【範例 5.2-1】此例為直接路徑之查詢句，所有變數路徑皆不含有跳層，變數之間無上下層關係。

```
for   $a1 in /dept/course/takenBy/student,
      $a2 in /dept/course/takenBy/student,
      $a3 in /dept/course/takenBy/student,
      $a4 in /dept/course/takenBy/student,
      $a5 in /dept/course/takenBy/student,
      $a6 in /dept/course/takenBy/student,
      $a7 in /dept/course/takenBy/student,
      $a8 in /dept/course/takenBy/student,
      $a9 in /dept/course/takenBy/student,
      $a10 in /dept/course/takenBy/student
return      { $a1, $a2, $a3, $a4, $a5, $a6, $a7, $a8, $a9, $a10 }
```

【範例 5.2-2】此例為單一跳層路徑之查詢句，所有變數路徑皆為一跳層路徑，變數之間無上下層關係。

```
for   $a1 in //course
return { $a1 }
```

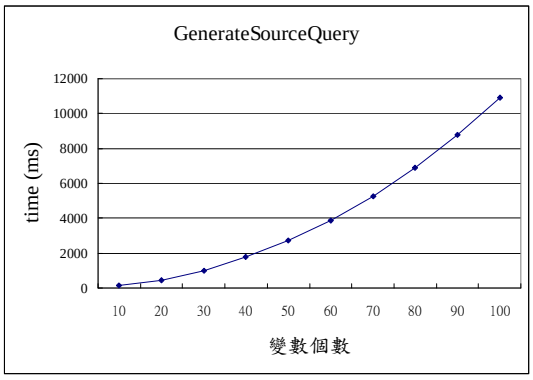
【範例 5.2-3】此例為雙跳層路徑之查詢句，所有變數路徑皆為兩層的跳層路徑，變數之間無上下層關係。

```
for   $a1 in //course//course
return { $a1 }
```

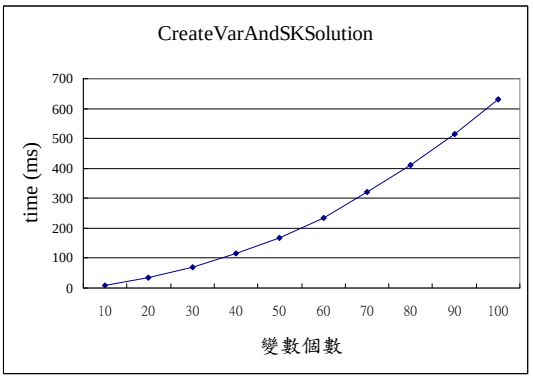
【範例 5.2-4】此例為單一跳層路徑，變數兩兩成對且有上下層關係之查詢句，下層變數的轉換結果會受到上層變數轉換之結果的影響。

```
for   $a1 in //course,
      $a11 in $a1//course
return { $a1, $a11 }
```

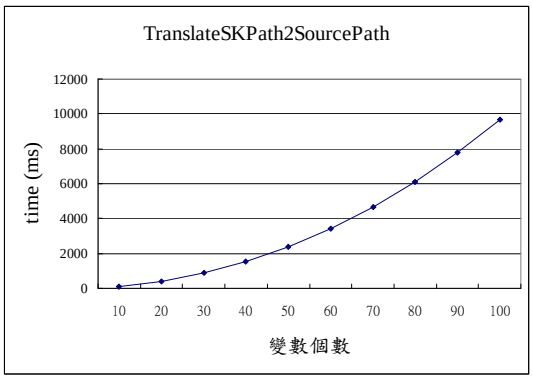
第一個實驗，我們用範例 5.2-1 的查詢句作為基礎查詢句，改變變數個數從 10 ~ 100，在此實驗中，因為直接路徑，因此查詢之 XML 文件深度不會影響查詢結果，預設查詢之 XML 文件深度為 10。由圖 5.6（a）得知，隨著變數的增加，查詢句轉換所花費的時間也會增長。因時間呈線性稍微成長，便進行更細部的時間測試，針對圖 4.13 的演算法中的主要部份進行效能的測試。主要分為圖圖 4.15、圖 4.17 及圖 4.19 的四個演算法。結果如圖 5.6（b）、（c）、（d）。



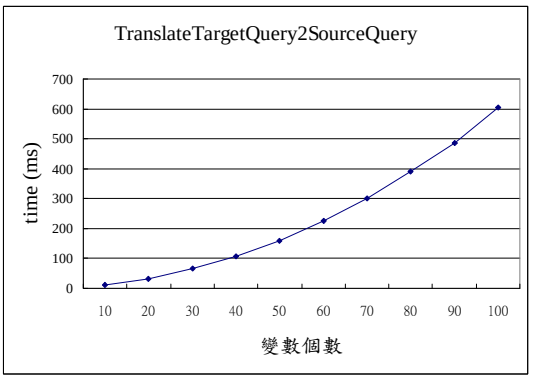
（a）圖 4.13 演算法效率



（b）圖 4.15 演算法效率



（c）圖 4.17 演算法效率

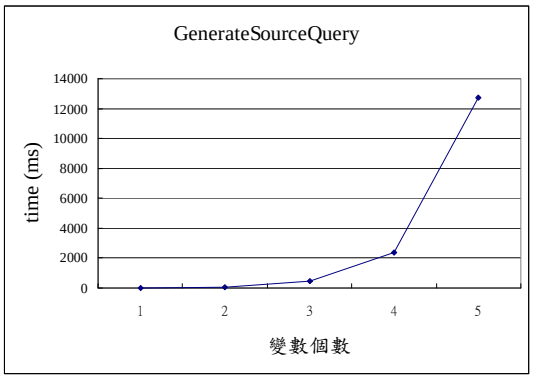


（d）圖 4.19 演算法效率

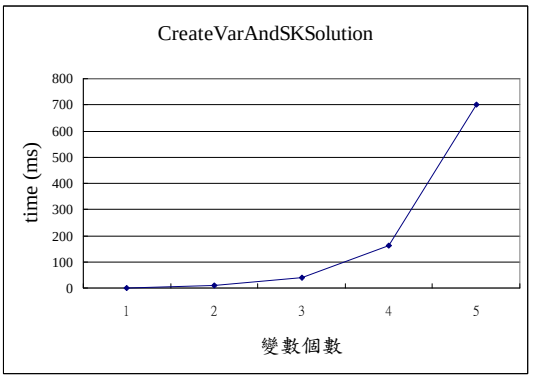
圖 5.6 直接路徑之變數個數與效率

由圖 5.6（b）、（c）、（d）三個實驗結果，可以得知轉換的效能會隨著變數的增加，而逐漸降低，但仍然是偏向線性稍微成長的結果。

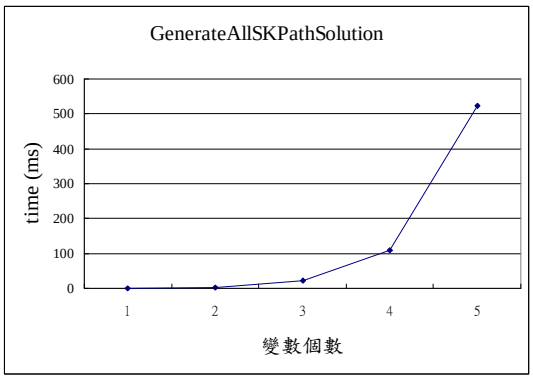
第二個實驗，我們以範例 5.2-2 的查詢句作為基礎查詢句，改變變數個數自 1 ~ 5 來測試效能，查詢之 XML 文件深度為 10。除了第一個實驗中測試的演算法效率之外，我們再加入圖 4.16 產生所有解的演算法來進行效率測試，實驗結果如下：



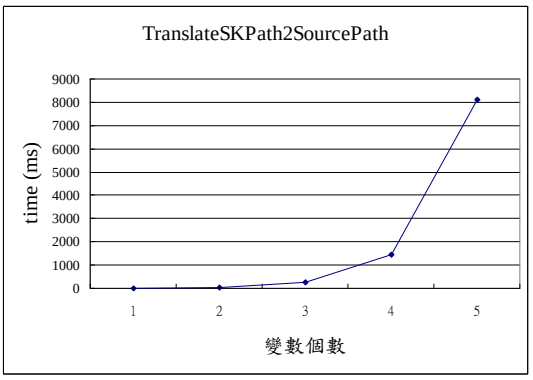
（a）圖 4.13 演算法效率



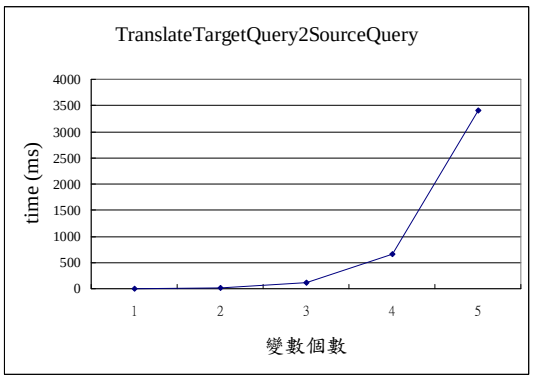
（b）圖 4.15 演算法效率



（c）圖 4.16 演算法效率



（d）圖 4.17 演算法效率



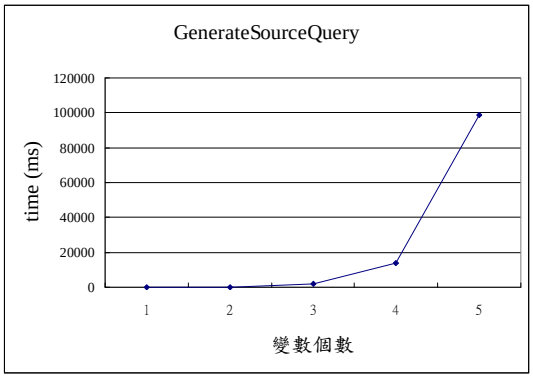
（e）圖 4.19 演算法效率

圖 5.7 單一跳層路徑之變數個數與效率

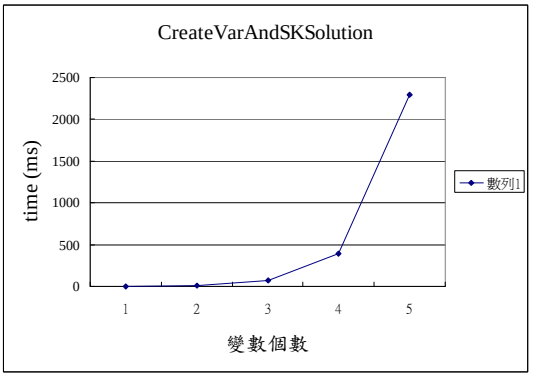
由圖 5.7 所示，實驗結果會隨著變數個數呈指數成長，原因在於變數間是互相獨立，且皆為跳層路徑，因此每個變數符合的結果相乘之後，展開所有解時，結果會是以指數成長，轉換查詢句的時間花費也就呈指數成長。

接著我們進行第三個實驗，以範例 5.2-3 的查詢句作為基礎，增加變數個數

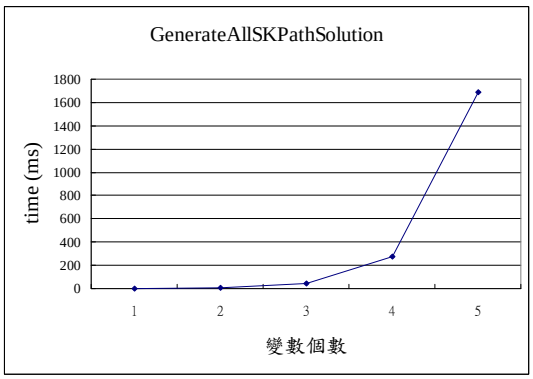
自 1 ~ 5，查詢之 XML 文件深度為 10，測試在兩個跳層路徑的狀況下，變數個數對查詢句轉換的效能之影響。測試之演算法同第二個實驗。結果如下：



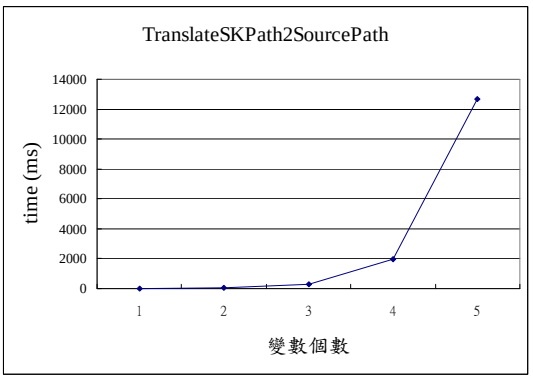
（a）圖 4.13 演算法效率



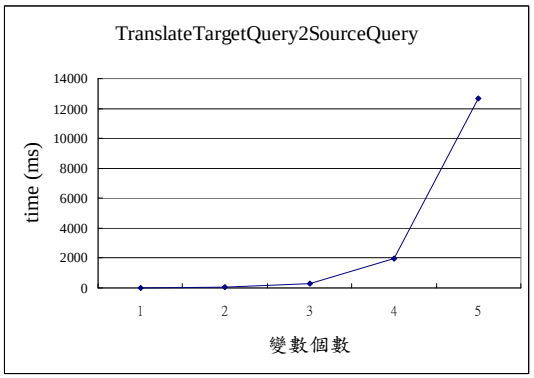
（b）圖 4.15 演算法效率



（c）圖 4.16 演算法效率



（d）圖 4.17 演算法效率

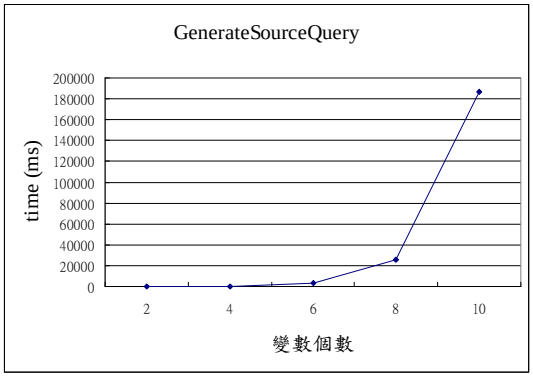


（e）圖 4.19 演算法效率

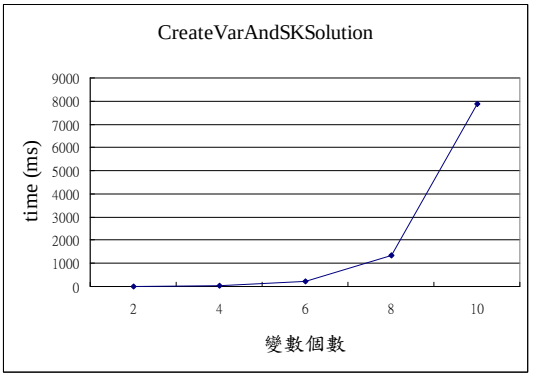
圖 5.8 雙跳層路徑之變數個數與效率

與第二個實驗不同之處在於每個變數路徑是兩個跳層，在圖 5.8 中我們可以得知，跳層路徑的增加亦會影響查詢句轉換的效能。

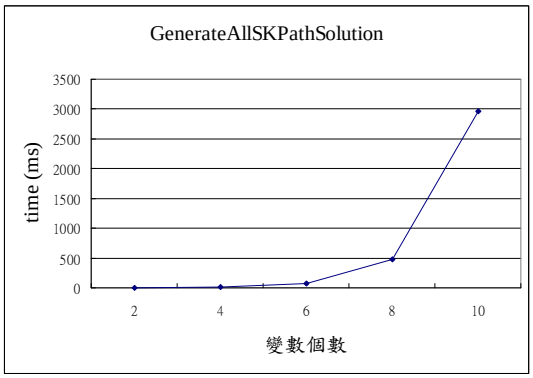
最後，第四個實驗，我們測試變數之間的關係對查詢句轉換的影響。以範例 5.2-3 的查詢句作為基礎，將雙跳層路徑分開在兩個變數下，使每個變數路徑中只有一個跳層路徑，並使此兩變數為上下層關係，增加變數個數自 2～10，查詢 XML 文件深度為 10，實驗結果如下：



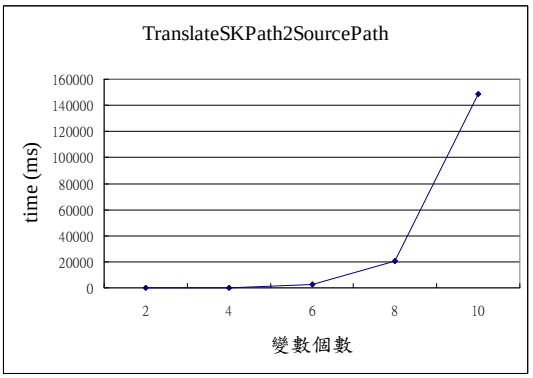
（a）圖 4.13 演算法效率



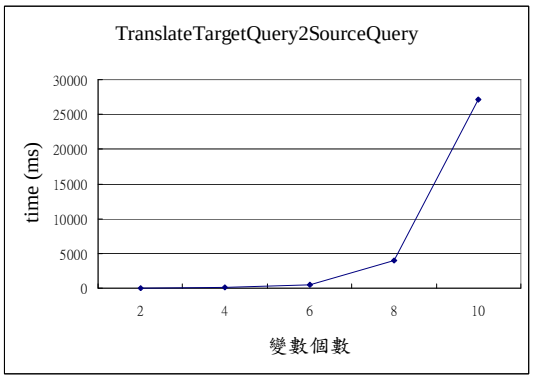
（b）圖 4.15 演算法效率



（c）圖 4.16 演算法效率



（d）圖 4.17 演算法效率



（e）圖 4.19 演算法效率

圖 5.9 單一跳層路徑，變數兩兩成對且有上下層關係之變數個數與效率

在最後的實驗中，可以得知在變數兩兩成對，變數路徑為跳層，且變數有上下層關係，實驗結果仍會呈指數成長。圖 4.19 演算法的效能會隨著產生的所有解之數量而成長，因此當圖 4.16 之演算法轉換出大量的結果時，其效能也隨之降低。

最後我們比較第三個實驗與第四個實驗的差異，如圖 5.10，我們可以看到，單一跳層且兩兩成對的變數之查詢句轉換，所花費的時間明顯高於雙跳層的變數之查詢句轉換結果。原因在於變數之間有上下層關係時，每個下層變數必須查表取得上層變數的轉換資訊，才能依據上層變數的轉換結果進行轉換，因此花費的時間較高。

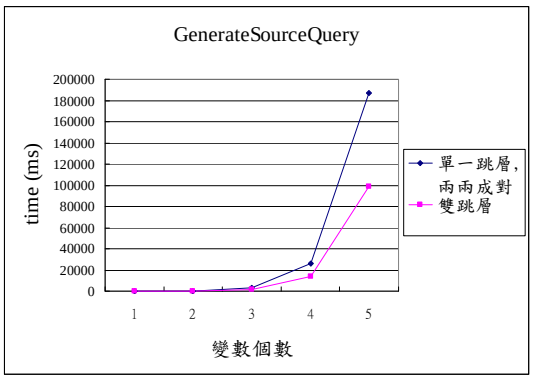


圖 5.10 變數間關係獨立與互相關聯之效能測試

綜合全部的實驗結果可以得知，在單一跳層路徑下，變數個數在 5 以內仍能有效率的轉換查詢句。在雙跳層路徑下，變數個數在 4 以內亦能有效率的轉換查詢句。在單一跳層路徑，但變數兩兩成對，且有上下層關係的狀況下，變數個數在 8 以內仍為有效率的查詢句轉換。

第六章、 結論與未來展望

本論文所提出支援遞迴資料定義之 XQuery 查詢句轉換系統，透過正確的 Mapping 定義及藉由 Mapping 產生的 Skolem Function 進行查詢句轉換後，仍能保持原有的結構關係，使得查詢句能透過此系統，在經由轉換後，至另一個資料庫取得所需的資料。我們加入了遞迴資料的對應關係，並找出合理的產生 Skolem Function 方式，在透過建立變數之間的關係表格，與路徑及符合路徑之 Function 表格，將查詢句轉換成由 SKPath 所組合成的中間結果，再透過展開中間結果，並刪去不符合的，將每一筆適合的結果透過查表轉成查詢句後輸出。雖然在查詢句與 Mapping 的使用上有諸多限制，但對於能加入遞迴路徑的對應並處理跳層所帶來的複雜結果，是一種新的挑戰。

在第五章的實驗測試出在查詢句轉換時，主要花費的時間皆在於圖 4.15、圖 4.16、圖 4.17，這三個演算法中，若欲使整個查詢句轉換更為快速，則必需找出更有效率的變數路徑的轉換方式，例如在圖 4.16 的演算法中，若能先行判斷出哪些轉換的結果是可以合併的，則能減少後續的演算法的執行次數，相對的執行時間也會跟著下降。

在未來研究規劃了兩個方向，第一個是設計更一般化的 Mapping 結構，使得除了遞迴路徑對應到遞迴路徑以外的對應資訊都能夠寫入 Mapping 中，並轉換成適合的函數來使用，例如：Target DTD 之遞迴路徑與 Source DTD 之非遞迴路徑的對應、Target DTD 之非遞迴路徑與 Source DTD 之遞迴路徑的對應。第二個是針對查詢句中，LET 能夠定義巢狀表示式以外的資訊，能夠將 LET 宣告之變數所指向之一般路徑或跳層路徑，轉換成新的巢狀表示式後，回傳給 LET 宣告的變數，並測試是否需要轉換 LET 宣告之變數所定義的其他相關路徑。

參考文獻：

[DTC+03] David DeHaan, David Toman, Mariano P. Consens, and M. Tamer Ozsu, “A Comprehensive XQuery to SQL Translation using Dynamic Interval Encoding”, In Proceedings of the ACM SIGMOD International Conference on Management of Data, pages 623-634, 2003.

[FYL+05] Wemfei Fan, Jeffery Xu Yu, Hongjun Lu, Jianhua Lu, Rajeev Rastogi, “Query Translation from XPath to SQL in the Presence of Recursive DTDs”, VLDB 2005, page 337 - 348, Trondheim, Norway.

[KCK+04] Rajasekar Krishnamurthy, Venkatesan T. Chakaravarthy, Raghav Kaushik, Jeffery F. Naughton, “Recursive XML Schemas, Recursive XML Querys, and Relational Storage: XML-to-SQL Query Translation”, ICDE 2004, page 42 - 53, Paris, France.

[ODP+06] Nicola Onose, Alin Deutsch, Yannis Papakonstantinou, Emiran Curtmola, “Rewriting Nested XML Queries Using Nested Views”, In Proceedings of the SIGMOD, Chicago, Illinois, USA, pages 443 - 454, June 27-29, 2006

[VMM+05] Yannis Velegrakis, Renacuteee J. Miller, John Mylopoulos, “Representing and Querying Data Transformations”, In Proceedings of the *ICDE*, Tokyo, Japan,, Pages 81 - 92, 5-8 April 2005.

[W3C] http://www.w3.org/TR/2005/WD-xquery-xpath-parsing-20050404/

[YP04] Cong Yu, Lucian Popa, “Constraint-Based XML Query Rewriting for Data Integration”, SIGMOD 2004, page 371 - 382, Paris, France.