國立臺灣海洋大學

資訊工程系

碩士學位論文

指導教授：張雅惠　博士

以資料定義為基礎轉換 XQuery 查詢句至 SQL 查詢句

Schema-Based Query Translation From XQuery to SQL

研究生：呂信賢　撰

中華民國九十三年七月

摘要

使用關聯式資料庫儲存 XML 文件，所衍生出來的查詢語言轉換是個重要的課題。W3C 制訂的 XQuery 查詢語言可針對 XML 文件做查詢處理，在關聯式資料庫中則使用 SQL 查詢語言做查詢處理，兩種不同查詢語言所依循的資料定義亦不相同。所以本論文研究的重點，是以資料定義為基礎，轉換巢狀、有聚合函式和複雜路徑的 XQuery 成為一份等價的 SQL。

在資料定義方面，DTD 規範了 XML 文件中元素的結構、Relational Schema 則是定義關聯式資料的表格和欄位，對此兩種不同資料定義，本論文提出對應表格和連結對應表格，來表示 DTD 中的每一條可能路徑對應的 Relational Schema 表格和欄位。以對應表格和連結對應表格為依據，我們提出查詢語言轉換系統「FORET」，將使用者輸入的 XQuery 剖析後，對 XQuery 中不同的子句建立 For Tree 和 Return Tree，並將子句中的路徑轉換成查詢樹節點。為了有系統地合併與建構每一棵查詢樹上節點的 SQL 片斷，本論文會根據每棵查詢樹的 Level Number 所組成的 Level Sequence，做有順序性地收集節點上的 SQL 片斷，並且利用 Level Sequence，來解決巢狀結構的 XQuery。

為了評估轉換後的 SQL 查詢語言和輸入的 XQuery 等價，設計了正確性評估實驗。並且對不同的資料定義，如一份 Relational Schema 對應到多份 DTD、一份 DTD 對應到多份 Relational Schema，分析查詢句轉換效率。最後是針對巢狀結構和扁平結構的 XQuery 做轉換效率的比較。

Abstract

Query translation is an important issue when using an RDBMS to store XML documents. The XQuery language proposed by W3C is used for querying XML data, and the SQL language is used for RDBMS. These two different query languages are posed against different schemas. Therefore, the focus of this thesis is to based on schema definition to translate a nested and complex XQuery with aggregation functions to an equivalent SQL.

As to schemas, DTD defines the structures of elements that can appear in XML documents, and relational schema defines the tables and fields of relational data. To represent the correspondence between two different schemas, we proposed the Mapping Table and the Join Mapping Table to represent the “equivalent” table or field of a relational schema for each possible path in DTD. Based on the two mapping tables, we proposed a query translation system「FORET」. After parsing XQuery that users input, we constructed the For Tree and Return Tree for different clauses in the XQuery and represented paths in the clause as nodes of the query tree. In order to systemically combine the SQL fragment associated with each node in the query tree, we used Level Sequence, which consists of Level Number for each query tree. This is used to resolve nested XQuery.

Several experiments are designed and performed in this thesis. The first one is designed to estimate the equivalence of the input XQuery and the output SQL. We also analyze the efficiency of query translation based on different mappings of schemas, for example, a relational schema corresponds to many DTDs or a DTD corresponds to many relational schemas. Finally, we compare the efficiency of translating a nested XQuery with translating a flat XQuery.

誌謝

研究所的生涯近尾聲，二年的學習時光匆匆即逝，在這期間感謝恩師　張雅惠博士的耐心指導，不但讓我在專業領域上有所精進外，處事想法和做事態度上有更大的改變。

論文口試承系上教授　梁德容博士和國立台灣大學資訊工程系趙坤茂博士於百忙中抽空參與論文審查工作，對本文提出寶貴意見及指正之處。

感謝我的父親呂義誠先生和母親江綢女士的支持，不但給予我許多鼓勵，精神上的支持是讓我能堅持下去的最大原動力。此外，求學期間，感謝謝璨隆學長、邱豐傑學長、李政達同學、蘇智宏學弟、劉邦鑑學弟和羅介璋學弟的幫忙，讓我在困惑中能釐清問題、解決難處。

目錄

圖表目錄

第一章　緒論

1.1 研究動機與目標

資訊科技發展迅速，網際網路（Internet）的興起加速全球化的趨勢，伴隨著全球資訊網（World Wide Web），資訊分享、交換的速度遠大於以往使用電話、傳真聯絡，企業間商務行為日漸倚重網際網路做為重要的媒介。目前全球資訊網以超文字標記語言（Hypertext Markup Language；HTML）格式描述資料，但 HTML 為版面配置導向的語言，主要做為資料顯示、設計之用，缺乏資料內容和結構性的描述，不利資料傳遞與交換。

W3C 組織制訂的可擴充式標註語言（eXtensible Markup Language；XML）是一種描述資料的標註語言，可自由定義標籤，且可將定義的資料結構化，其結構可由 DTD（Document Type Definition）或 XML Schema 的定義所限制，加上 XML 為純文字格式，非常有利做為資料傳遞和交換的格式。目前許多企業、學術單位和機關團體都已將 XML 做為資料交換的統一格式，許多資料模組也以此呈現，由於 XML 文件日益增多，資料儲存量大，因此查詢 XML 文件資料內容已是目前很重要的研究課題，W3C 組織亦制定 XML 文件的查詢標準 XPath 和 XQuery 查詢語言，讓使用者能正確地從 XML 中查詢到所需要資料。

然而，大多資料仍儲存在非 XML 資料庫系統中，如具有高穩定性和成熟度特性的關聯式資料庫系統（Relational Database Management System；RDBMS），目前典型的作法是當由目的地接收 XML 資料，將 XML 資料重新對應至應用系統資料結構或 RDBMS 中，但兩種異質性資料造成資料管理上的困難。首先，資料定義的不同，XML 文件使用 DTD 或 XML Schema 的定義做驗證，而關聯式資料則是使用關聯式綱要（Relational Schema）驗證，要將 XML 資料重新對應 RDBMS 中，[HHL+02、ML02、SKZ+02、SYU99、YAS+01、AS02]已提出相關的研究及解決方法。

另一個困難的課題，即是查詢語言的轉換，如何透過查詢語言的轉換，來達

成 XML 資料與關聯式型態資料能彼此交換與整合。在查詢 XML 資料方面，W3C 制定了 XQuery 查詢語言，依據 DTD 定義下達合法的路徑表示法（XPath Expression）；關聯式資料則是使用結構化查詢語言（Structure Query Language；SQL），根據關聯式綱要下達正確的查詢句。

可是 XQuery 是以路徑導向為基礎，對 XML 文件做資料的查詢，其中 XQuery 包含 12 種豐富的語意，查詢語言亦有巢狀結構的特性；SQL 則是以資料列導向，結構上是扁平式的，語意無 XQuery 複雜，可是有 GROUP BY（分群）的語法，對聚合函式（Aggregation Function）的限制和使用較為嚴謹，此點在 XQuery 中並無特殊的表示式呈現。

假設 XML 存在 RDB 中，所以需要將 XQuery 轉換至 SQL，為了轉換 XQuery 查詢句為一份等價可執行的 SQL 查詢句，本論文設計了一套以資料定義為基礎（Schema-Based）的查詢語言的轉換系統－ FORET (FOr tree and REturn Tree)。在轉換之前，系統管理者先建立一套整合 DTD 和 RDB 定義的對應表格（Mapping Table）和連結對應表格（Join Mapping Table），此表格清楚表示 DTD 路徑和元素（屬性）對應到 RDB 表格或欄位，以此兩表格為根據，進行複雜的 XQuery 查詢語言轉換成等價 SQL 查詢語言的流程，其中 XQuery 可以是具巢狀結構的查詢句，並且探討有聚合函式的 XQuery，根據查詢句下達語意，使其轉成具有 GROUP BY 子句的 SQL 查詢句。

1.2 相關研究

以下列舉相關研究。從關聯式資料庫中將資料取出成為 XML 文件方面的研究（稱為 XML Publishing），請參見 [BCF+02, FKS+02]；將 XML 文件資料轉入關聯式資料庫方面（稱為 XML Storage），其中使用 DTD 資訊的，請參見 [HHL+02、ML02、SKZ+02]；不使用 DTD 資訊的，請參見 [SYU99、YAS+01]；另一方面，[AS02]提出綜合的作法，而 [DTC+03] 則是目前唯一提出轉換複雜

的 XQuery 為 SQL 的研究。[CR03、YLL03、 ME03、LS03、XE03]則是提出專門針對 schema 對應與轉換的技術。

至於每篇論文大致的作法再仔細說明如下。在 XML Publishing 方面，[BCF+02]使用 Attribute Translation Grammar（簡稱 ATG）將關聯式資料庫的資料依循 DTD 定義取出，再將資料組成一份 XML 文件。首先分析定義好要輸出的 DTD 的語意，由 DTD 語意產生 ATG Graph，將 ATG Graph 和關聯式資料庫的資料輸入到系統中，配合 XML 結構分析函式產生出 XML 文件。此研究保證 XML 文件符合 DTD 定義外，對於非決定論的 DTD 先做正規化的處理，以及有遞迴定義時，會套入統計運算，算出適當的深度後，再將資料依循遞迴定義聚合起來。

另一篇作法則為 [FKS+02]。該論文討論將關連式資料以 XML view 的方式 publish 之後，使用者會再下達 XQuery 的查詢句，取出所欲之資料。但是由於 XQuery 和 SQL 的表現能力不同，可能必須重複的下達同一個 SQL 查詢句，所以該篇論文提出一個 middleware 的架構，以達到此功能。本論文觀點則是著重在分析 XQuery 語意方面，不但考慮到輸出，也考慮資料連結，完整地記錄下查詢句的結構和語意，而[FKS+02]則是以輸出結構為 SQL 查詢句的轉換依據。

而在 XML Storage 方面，[HHL+02、ML02、SKZ+02]討論到的是以 Schema 為基礎的儲存方式，[HHL+02] 使用 Content Management System（簡稱 CMS）儲存及檢索 XML 文件，並整合非 XML 文件，如傳統文字檔、多媒體等。首先整合 XML 文件方面，先對 XML Schema 做簡化的動作，再依簡化後的 Schema 在 Library Server 中建立階層式表格；非 XML 文件，在 Library Server 建立表格儲存檔案資訊，將檔案實體存入 Object Server，使用者只需要藉由 Link 表格就可藉由檔案資訊抓取到檔案實體。

[ML02] 為了讓關聯式 Schema 能清楚明確地在 XML Schema 上表示其限制式，使用規則樹文法（Regular Tree Grammar）轉換 XML Schema 成自定義的 XSchema，從 XSchema 對應到關聯式 Schema 之間，會先做 Schema 的簡化和套用 Hybird Inlining 方法，而 IDREF 和 IDREFs 的限制式，則使用和 inlining 類似

的做法，將 IDREFs 屬性用 IDREFSinling 函式處理。對於遞迴結構則使用 Foreign key 關係來判斷，最後對於每個表格做主鍵值包含性關係處理，合併或減少不必要的表格和欄位。

[SKZ+02]根據 DTD 建立 DTD Graph，以簡化後的 DTD Graph 結構來定義資料庫的表格定義。提出 Basic Inlining、Shared Inlining 和 Hybrid Inlining 三種方法，Basic 對每個節點產生一個對應表格，並將子孫節點全部包含進來變成自己的欄位，除了節點是*型態和有遞迴性質的；Shared 依循 Basic 的方法，但是有多個父節點的元素（in-degree > 1），也需自成一個表格；Hybrid 是和 Shared 相似，但是有多個父節點的元素，只要不具有遞迴關係亦可被包含進來變成自己的欄位。Hybrid Inlining 做法中多餘節點（dummy node）亦會產生一表格，但表格中僅產生鍵值欄位，考慮此不必要的表格，減少連結關係，所以本論中若使用 DTD 要轉成對應的 Relational Schema，會改良 Hybrid Inlining 做法。

忽略 Schema 做 XML Storage 中，[SYU99]將 XML 文件所有節點分成元素、屬性和文字節點，並取出所有路徑。首先分析元素節點，取出順序關係、在 XML 檔案中的的絕對位置和對應的路徑；屬性節點記錄文字值和絕對位置；文字節點記錄文字值、在檔中的範圍和對應路徑；最後是所有可能路徑，將此四類資料匯入資料庫中成為各別的表格中。[YAS+01]延續[SYU99]的做法，主要著重在 XPath 查詢句轉換成 SQL 查詢句，由於記錄 XML 文件的所有不重複路徑，在處理 AD（Ancestor-Descendant）表示式，將‘//’轉成‘%/’，利用 SQL 的 LIKE 限制式，將原本需要多條結合限制的查詢句，變成簡單的字串比對。

至今提出了許多 XML Storage 的方法，但大多程式不易撰寫且資訊難以分享，研究 [AS02] 歸類了六種 XML 對應到關聯式資料庫中的方法，此方法是彈性可擴充的，當有新的對應方法，只需要在對應的 Schema 中記錄此方法的名稱、程式演算法、查詢句轉換方式和工具等，存入對應倉儲中，即可從倉儲中得到新的對應方法，除此之外，也設計了許多應用程式介面，讓使用者可以設過這些介面抓取對應 Schema 中的資訊。

不論 XML Publishing 或 XML Storage，皆有查詢句轉換的問題，要將表示式豐富的 XQuery 轉成 SQL 語法，或 XPath 轉成 SQL，仍有很大的發展空間。[SYU+99] 提出將 XQL 轉換成 SQL 的方法，[HHL+02] 則是提出 XQuery 轉換成 SQL 的演算法，將 XQuery 或 XPath 的每條路徑和節點分節出來，再根據這此路徑和節點產生 SQL，最後再合併這些查詢句。不過上述都沒有很詳細的轉換步驟，且僅能轉換非巢狀結構的查詢句。

而 [DTC+03, KCK+04] 是提出轉換 XQuery 為 SQL 的研究。[DTC+03]它使用動態區間編碼來輔助轉換和資料查詢的動作，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，本論文對 DTD 使用此編碼，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。此研究可轉換巢狀結構、含複雜路徑的 FLWR 表示式，首先定義查詢句中可能使到的運算子，再來對每個元素計算出其區間且在查詢句中再動態地建立查詢句自訂元素或 XML 文件元素的區間，配合定義好的運算子，將數值和運算子套入 SQL 中，轉換成有效的查詢句。

[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。雖然可以處理 Recursive 的 XML Schema，可是對於巢狀結構的 XQuery 並沒有提出解決方法，[JLS+04]則是對 XML 的架構，利用顏色來區分 XML Tree 中不同的屬性及層次關係，如此一來可以清楚的劃分及查詢資料，並可減輕扁平結構 DTD 查詢句的 join 次數。

取之上述三篇複雜查詢句轉換的優點，著重在以資料定義為基礎，XQuery 查詢句的範圍為可具巢狀、聚合函式和複雜路徑，且特別針對上述所沒有處理的聚合函式做深入的討論，不過尚未處理到遞迴架構的 DTD。而本論文定義的 For Tree 和 Return Tree 顧慮到階層的問題，因此參考[CJL+03]，設計出 Level Number

來判斷查詢句的階層，並將 Level Number 組成有意義的順序形成 Level Sequence，根據此 Level Sequence，除了可清楚看出查詢樹個數和類型，還可有系統地合併查詢樹節點上的 SQL 片斷。

另外一個和 XML publishing 和 XML storage 皆有相關的問題，則是彼此 schema 的對應和整合。[CR03] 提出一種具彈性、有延伸性且可使用性的轉換模組化平台——Sangam。使用最基本的運算子去模組複雜的轉換。使用者可以輸入 XML DTD 或關聯式資料庫 Schema，接著依據 Sangam 的定義轉成 Sangam Graph，使用以圖形化方式定義的基礎運算子，以及用基礎運算子合併衍生而出的 Cross Algebra 運算子整合 Sangam 圖形，之後再將整合好的 Sangam 圖形套入模組轉換的執行策略中，最後可以輸出成 XML DTD 或關聯式資料庫 Schema，在輸入時也可以將 XML 文件或資料庫的資料輸入至系統中，最後亦會將資料整合成同質性資料。

針對 Schema 對應的問題，[LS03] 不需要每個區域資料端間的對應，或區域資料端共同的對應，而對每個資料端用標準的詞彙宣告（ontology 宣告），增加區域資料來源的述語表示。標準的 ontology 是由述語名稱和用法組成，例如 book-author，book 會對應到 book 的鍵值；author 會對應到 author 的鍵值，且兩種都會加上名稱資訊和型態，如 book 會是 ISBN 編號，型態是字串。此研究適用於分散式或點對點的的架構上，當下達查詢句時，系統協調機制會將查詢句對應各個區域資料的述語表示進而得到部份的答案，通過系統協調機制統一整合將答案回覆給使用者。

[ME03] 則是針對分別當 A 對應到 B，B 對應 C 的時，如何產生一份 A 對應 C 的直接對應且等價於原來的對應關係。在 Schema 的對應上，對結構編碼，編碼出來的定式建立與資料端的關聯性，再進行限定大小的合併。而在查詢句對應上，查詢句是限制在沒有比較式的 SPJ 範圍內，和 Schema 對應相仿，最後會建立 Query-Rewrite Graph，並保證重新撰寫後合併產生出來的查詢句最佳化。

[XE03]挑戰自動對應，除了從一份來源 Schema 直接對應目標 Schema 的元

素外，亦可間接對應元素。在間接對應上會遇到一般化、特殊化、合併數值、分割數值和元素名稱為數值的問題，之後在對應技術上，分別有(1) 分析來源端和目標端的元素名稱字詞語意分；(2) 比較名稱長度和文數字比率；(3) 用輕量的 ontology domain 分析間接對應出現的問題，對一般化和特殊化用 Union 和 Selection 運算子解決、合併數值和分割數值用 Composition 和 Decomposition 解決、元素名稱為數值用 Boolean 來表示，前三種做法中都會給予積分，並計算出期預值；(4) 比較 Schema 的結構。最後由實驗結果發現，如果使用以上四種方法，對應的正確率會在 90%以上。

[YLL03] 則是將 XML Schema 中的語意轉換成 ORA-SS（Object-Relationship-Attribute Model for Semi-Structured Data），如主鍵值、兩個物件之間的關係和作用依賴性等，配合此研究提出的演算法，解決整合時所會遇到的衝突，主要四個步驟為：(1) 解決屬性－物件衝突、一般化和特殊化物件包含關係；(2) 計算每一條關係上的權重，並先忽略物件上的屬性建立整合的圖形；(3) 轉換圖形，移除權重較高和多餘的關係、移除循環關係和多餘物件，以及移除多重父節點的物件；(4) 將每個物件的屬性增加到整合的圖形上。

1.3 論文架構

本論文的架構介紹如下：第二章，將先對 XML 資料、關聯式資料表示方法，以及 XQuery 和 SQL 查詢語言作一簡介，並且定義查詢語言轉換的問題癥結和規範處理的查詢句範圍。第三章將先介紹本論文建構的系統，之後提出整合 XML 資料定義（DTD）和關聯式資料定義（Schema）的對應表格以及連結對應表格。第四章，以對應表格和連結對應表格為根據，提出執行查詢語言轉換的演算法。查詢語言轉換成本論文定義的查詢樹後，會將查詢樹內的 SQL 片斷合併並且合理化，此會在第五章提出。經轉換所得之查詢語言的正確性評估，以及轉換效率比較，在第六章探討。最後，將在第七章作綜合討論，以及描述未來的研究方向。

第二章　相關定義

在本章中，首先提出 XML 和關聯式資料範例，說明 XML 文件格式、DTD 定義方式和關聯式綱要（Schema）定義方式。接著說明 SQL 查詢語言和適用於本論文轉換方法的 XQuery 範圍。

2.1 DTD 與關聯式綱要

此章節以範例說明 DTD 和關聯式綱要兩者資料格式、資料結構的差異，並詳細說明範例內容。

2.1.1 XML 和 DTD

XML 和 HTML 一樣同屬於 SGML(Standard Generalized Markup Language)，為控制輸出和排版之用，而 XML 則允許使用者自行定義標籤，以說明內容資料之意涵。XML 文件是一種簡單、彈性很大的純文字格式檔案，元素之間可以用樹狀（巢狀）結構來呈現，而檢查 XML 文件的標籤是否正確，結構是否合乎使用者需求，會使用 DTD 來驗證（Validity）。

圖 2.1 裏的 DTD 是本論文對 XMark 所提出的關於拍賣記錄的 DTD 稍加修改而來，由根元素 site 的定義可知，主要記錄了四個子元素：items（物品）、people（使用者）、open_auctions（起標記錄）和 closed_auctions（結標記錄）。這四個子元素皆有可重複的子元素，如 L3，代表 items 元素裏可包含多個 item 子元素。而此 DTD 中的屬性大多為 ID 或 IDREF 型態表示（Key Constraint），ID 屬性代表不可重複設定相同(重複)的屬性值，例如物品編號、使用者身份證字號；IDREF 為 Identifier Reference 的縮寫，顧名思義，它的使用時機為當屬性值是參考到另一個屬性值型態是 ID 的屬性值適用，例如記錄起標記錄的 open_auctions 元素，其下並沒有任何元素記錄有關於起標物品的資料，利用 open_auction/itemref 元素下為 IDREF 的屬性@item，從@item 的值可以在 site/items/item/@id 中找到一樣

的值，即可從此屬性參考中得到競標的物品詳細資料。

2.1.2 關聯式資料與關聯式綱要

關聯式資料具有固定的結構，同一表格的資料列都具有相同的欄位，資料合法性可藉由定義綱要（Schema）驗證，從圖 2.2 的 Schema 裏可看到五個表格：Item、Person、Open_Auction、Closed_Auction 和 Bidder。此份 Schema 是使用[SKZ+02]的 Hybrid Inlining 方法轉出來的對應 Schema，除此之外，機關單位或組織制訂好的 DTD 和 Schema（例如：mondial、XQuery Use Case“R”）也可使用本論文提出的方法。本論文使用的轉換方法，有幾個地方改良了 Hybrid Inlining 的做法：

- 若 DTD 路徑中存在著 e_1/e_2*宣告方式，e_2*元素自成一表格，而 e_1 亦對應到 e_2 的表格。例如 site/items/item*，item 元素自成一表格 Item，而 items 為多餘節點（Dummy Node），因此遇到此情形，多餘節點對應的表格為型態*子元素所產生的表格。
- Hybrid Inlining 轉出來的表格都會有一個“ID”欄位做為 Primary Key，在本論文的做法中，會先判斷轉成表格的元素中是否含有 ID 型態的屬性，有的話，此屬性轉出來的欄位做為主鍵值（Primary Key）欄位；反之，產生”ID”欄位做為主鍵值欄位。
- IDREF 型態的屬性在轉成表格的欄位時，必須考慮到兩種情形：(1)參考到 ID 型態屬性依附的元素若自成一表格，此 IDREF 屬性轉成欄位時會宣告成外鍵值（Foreign Key）；(2)反之，即轉成一般的表格欄位。

在轉出來的所有表格，首先 Item 表格表示所有物品資料，每一個物品有自己的唯一編號（ID），還有物品名稱（Name）、物品描述（Description）。Person 表格表示所有註冊使用者的資料，每位使用者的編號（PID）、名字（Name）和收入(Income)。在 Open_Auction（起標記錄）表格中，有每一筆起標記錄的編

```
L1  <?xml version="1.0" encoding="UTF-8"?>
L2  <!ELEMENT site (items, people, open_auctions, closed_auctions)>
L3  <!ELEMENT items (item*)>
L4  <!ELEMENT item (name, description)>
L5  <!ATTLIST item id ID #REQUIRED>
L6  <!ELEMENT name (#PCDATA)>
L7  <!ELEMENT description (#PCDATA)>
L8  <!ELEMENT people (person*)>
L9  <!ATTLIST person id ID #REQUIRED>
L10 <!ELEMENT person (name, profile?)>
L11 <!ELEMENT profile EMPTY>
L12 <!ATTLIST profile income CDATA #IMPLIED>
L13 <!ELEMENT open_auctions (open_auction*)>
L14 <!ELEMENT open_auction (itemref, seller, bidder*)>
L15 <!ATTLIST open_auction id ID #REQUIRED>
L16 <!ELEMENT itemref EMPTY>
L17 <!ATTLIST itemref item IDREF #REQUIRED>
L18 <!ELEMENT seller EMPTY>
L19 <!ATTLIST seller person IDREF #REQUIRED>
L20 <!ELEMENT bidder (datetime, personref, increase)>
L21 <!ELEMENT datetime (#PCDATA)>
L22 <!ELEMENT personref EMPTY>
L23 <!ATTLIST personref person IDREF #REQUIRED>
L24 <!ELEMENT increase (#PCDATA)>
```

圖2.1：DTD文件範例 (1/2)

```
L25 <!ELEMENT closed_auctions (closed_auction*)>
L26 <!ELEMENT closed_auction (price, itemref, buyer, seller, datetime)>
L27 <!ELEMENT price (#PCDATA)>
L28 <!ELEMENT buyer EMPTY>
L29 <!ATTLIST buyer person IDREF #REQUIRED>
```

圖2.1：DTD文件範例 (2/2)

```
CREATE TABLE Open_Auction(
    OID CHAR(10) PRIMARY KEY,
    Itemref CHAR(10) FOREIGN KEY
            Reference Item(ID),
    Seller CHAR(10) FOREIGN KEY
            Reference Person(PID))

CREATE TABLE Closed_Auction(
    CID CHAR(10) PRIMARY KEY,
    Price NUMBER,
    Datetime DATE,
    Itemref CHAR(10) FOREIGN KEY
            Reference Item(ID),
    Buyer CHAR(10) FOREIGN KEY
            Reference Person(PID),
    Seller CHAR(10) FOREIGN KEY
            Reference Person(PID))

CREATE TABLE Item(
    ID CHAR(10) PRIMARY KEY,
    Name CHAR(25),
    Description TEXT)

CREATE TABLE Person(
    PID CHAR(10) PRIMARY KEY,
    Name CHAR(25),
    Income NUMBER)

CREATE TABLE Bidder(
    BID CHAR(10) PRIMARY KEY,
    OID CHAR(10) FOREIGN KEY
            Reference Open_Auction(OID),
    Personref CHAR(10) FOREIGN KEY
            Reference Person(PID),
    Increase NUMBER,
    Datetime DATE)
```

圖 2.2：關聯式綱要

號（OID）外，還有 Itemref 欄位 Foreign Key 到 Item 表格的主鍵值 ID 和 Seller 欄位 Foreign Key 到 Person 表格的主鍵值 PID。和 Open_Auctino 表格相似的 Closed_Auction（結標記錄）表格，主鍵值 CID，記錄結標價錢（Price）、結標日期時間（Datetime）、還有三個 Foreign Key 欄位，Itemref、Buyer 和 Seller。最後一個競價（Bidder）表格，每個物品的所有出價記錄都會在這個表格中表示，BID

欄位 Foreign Key 到 Open_Auction 的主鍵 OID，Personref 欄位 Foreign Key 到 Person 表格的主鍵 PID，還有每次加價（Increase）的記錄以及加價的時間（Datetime）。

2.2 XQuery 的相關定義

XML 資料其文件內的元素之間有著階層關係，而可以形成一樹狀結構，若搜尋 (Traverse) 該樹而將經過的元素集合起來，則可構成一條的路徑，稱為路徑表示法(Path Expression)。以此路徑表示法為基礎，W3C 定義了相關技術來處理 XML 資料，譬如 XPath 及 XQuery。其中 XQuery 是 W3C 為了提供更便利的查詢方式，針對各類 XML 資料所設計，不論其儲存為文件或在關聯式資料庫中。

【範例 2.1】

以下是 XMark 所提出的查詢句，“*List the names of persons that income is greater than 5000 and the number of items they bought is greater than 3.*”（Q8）。

```
FOR $p IN document("auction.xml")//person
LET $a := FOR $t IN document("auction.xml")//closed_auction
          WHERE $t/buyer/@person = $p/@id
          RETURN $t
WHERE $p//@income > 5000 AND count($a) > 3
ORDER BY $p/name
RETURN { <items>
           <item person=$p/name/text()> COUNT($a) </item>
</items>}
```

上述範例 2.1 中可看出 XQuery 是以路徑表示法為基礎，由 FOR 子句開始分析，XML 文件中允許同一元素重複出現，表示相同結構但不同的元素值或屬性

值集合，利用 FOR 子句以遞迴方式取得一個路徑表示法的結果，如$p 變數對應到每一個 person 元素，FOR 的文法涵義相當於 SQL 查詢語言中的 FROM 子句，皆是取得資料集合。

LET 子句在 XQuery 中是較為特殊的子句，它是讓變數和 XPath 或 FLOWR 結合在一起，如$a 變數，是結合一份 FLOWR 表示法，此種表示法稱之為巢狀結果（Nested）。LET 和 FOR 子句相似，除了可連結到資料集合外，更特殊的是可以連結到資料的結果，例如 LET $k := 5，表示$k 變數連結到數字“5”，在 FOR 子句中是無法下達此查詢句。LET 連結到的資料集合，從另一角度看可視為是連結到“許多資料結果”，因此本論文的觀點是，LET 子句不論連結到 XPath 或 FLOWR 子句，所代表的皆是路徑表示式代表的結果，之後再將這些結果包裝起來成為一份新的資料集合，因此對應到 SQL 查詢語言中的 Derived Relation。

RETURN 子句建構好新的資料回傳給使用者，相當於 SQL 查詢語言中的 SELECT 子句，在 RETURN 中不僅可回傳一般的 XPath 路徑，也可包含新的 FLOWR 子句，成為巢狀結構的查詢句。WHERE 子句很直觀地就是對應到 SQL 查詢語言中的 WHERE 子句，根據下達的限制句內容，濾除掉不必要資料。ORDER BY 子句即是表示以那個元素或屬性做為排列依序，對應到 SQL 查詢語言的 ORDER BY 子句。

【範例 2.2】

轉換範例 2.1 的 XQuery 成為等價的 SQL。

```
SELECT p.name as person, count(*) as item
FROM Person p, (SELECT *
                FROM Person p, Closed_Auction b
                WHERE b.buyer = p.id )  a
WHERE p.pid = a.pid and p.income > 5000
GROUP BY p.name
 HAVING count(*) > 3
```

```
ORDER BY p.name
```

從範例 2.2 可知，SQL 查詢語言是以資料列導向，且關聯式資料是扁平、正規化的；而 XQuery 是路徑導向，且 XML 資料可具巢狀結構、非正規化的，雖然上述中介紹了 FLOWR 表示式和 SQL 子句各部份的對應，可是基本資料結構的差異性，造成轉換 XQuery 成等價的 SQL 時，必須要先確定雙邊查詢語言的資料連結、資料限制和資料輸出是相同的。

資料連結部份，資料連結正確的對應是查詢句轉換的核心，因為資料限制和資料輸出必須依循資料連結的部份做處理，若轉換資料連結的語意錯誤，轉換的查詢句並不會等價。XQuery 中的 FOR、LET 子句是資料連結的語意，相同於 SQL 中的 FROM 子句，在此處理的重點是：FOR、LET 子句變數連結到的路徑對應的表格和欄位。資料連結的路徑，對應到關聯式綱要時，有可能剛好對應到一份表格，亦有可能對應到多份表格，當對應到多份表格時，需要使用連結（Join）關係確保資料連結的正確性。而 LET 如前面所敘，會視為一份 Derived Relation 來確保資料連結的正確性。

資料限制部份，除了一條路徑對應到多份表格所產生的連結關係外，對於 XQuery 中的 WHERE 子句，會先將限制句視為兩種類型：Selection Condition 和 Join Condition。Selection Condition 即是做文數值的限制式，只要找出對應的欄位，並加入限制即可；Join Condition 是做兩個表格之間的連結限制式，若下達的 XQuery 是扁平式的寫法，將限制式轉成等價的 SQL 限制即可；倘若下達的 XQuery 具巢狀寫法，其中資料連結可以引用到父層或祖先層，則限制式語意必須清楚正確、合理地轉換成等價 SQL。

例如範例 2.1 的變數$p，可以被變數$a 的巢狀查詢句裡取用，並且最外層有對變數$p 路徑下的節點做限制式，先轉換變數$p 連結的路徑，會得到範例 2.2 的 FROM 子句中的 Person 表格 p；變數$a 連結的巢狀查詢句視為 Derived Relation a，因為巢狀查詢句取用了變數$p 的路徑，當轉換至 SQL 時，必須加入 Person

表格，因此 Derived Relation a 中的 FROM 子句有 Person 表格 p，但是根據 XQuery 的原意，僅只有一條路徑對應到 Person 表格，但轉換成 SQL 時得到兩份不同的 Person 表格，因此必須要加入正確的連結關係，以確保資料連結的正確性。

資料輸出部份，範例 2.1 中 RETURN 子句有聚合函式 COUNT($a)的輸出，可是除此之外亦有一條一般路徑$p/name 的輸出，當轉成 SQL 時，需要有分群（Group By）的語意。但在 XQuery 查詢句並沒有對分群有特別的子句去處理，若只將 FLOWR 轉成 SQL 的 SELECT、FROM、WHERE 和 ORDER BY 子句的話，轉換出來的 SQL 查詢句會是錯誤的。而目前查詢語言轉換課題中也沒有相關的討論，在 RETURN 子句中有聚合函式和一般路徑做為回傳值，且 ORDER BY 子句中也有以一般路徑做為排序依據，轉換至 SQL 時，就必須考慮 GROUP BY 子句的產生。

除了資料輸出部份有聚合函式外，另一個聚合函式的轉換即是在 XQuery 的 WHERE 子句中出現聚合函式的限制，範例 2.1 的 WHERE 子句中，有一個限制是總共買的物品要大於 3 件以上，因為 XQuery 中沒有分群相關的子句表示，此聚合函式的限制放在 WHERE 子句中，但在 SQL 中，聚合函式的限制必須寫在 HAVING 子句中，且根據 SQL 語法的規定，必須先有 GROUP BY 子句，才能有 HAVING 子句，因此聚合函式的限制要轉換成 HAVING 子句時，代表最後的 SQL 查詢句一定有 GROUP BY 子句。

聚合函式除了在 XQuery 的 WHERE 和 RETURN 子句中出現外，在 LET 子句中亦可使用，可是代表的意思又不相同，對於在 LET 子句中的聚合函式，本論文不予討論。因此從資料連結、資料限制和資料輸出三個方向分析 XQuery 轉換成 SQL，範例 2.1 的 XQuery 轉成的等價 SQL 即為範例 2.2 的結果。

以下介紹 W3C 制定的 XQuery，XQuery 由 12 種表示法組成，分別為 Primary Expressions、Path Expressions、Sequence Expressions、Arithmetic Expressions、Comparison Expressions、Logical Expressions、Constructors、FLOWR Expressions、Unordered Expressions、Conditional Expressions、Quantified Expressions 和

Expressions on Sequence Types，是一種語意非常豐富的查詢語言。總括來說，XQuery 敘述句可視為一組 FLOWR（FOR-LET-ORDERBY-WHERE-RETURN）表示法，在 FLOWR 裡包含了其它 11 種表示法的語意，而且每一個子句有自己的功能。

範例 2.1 為一份 FLOWR 子句，其中的變數$p、數字和括號等表示，為 XQuery 查詢語中最基本的基本表示法（Primary Expression）；而第一行變數$p 所連結到的路徑“*//person*”為路徑表示法（Path Expression）；順序表示法（Sequence Expression）是數個 *item* 的排序集合，而 item 可以是節點或最小不可分割數值，而每個順序可以使用 union、intersect 或 except 這些 W3C 提供的運算子來進行合併；算術表示法（Arithmetic Expression）、比較表示法（Comparison Expression）和邏輯表示法（Logical Expression）則是一般常見的表示法；結構子（Constructor）的意思是允許使用者在 XQuery 語言中建構 XML 元素、屬性和文字，例如 RETURN 中<item>和<person>即為使用者自己建構的，且輸出時，使用“*fn:unordered*”函式，可以不按照 XML 原先的排序，此為非排序表示法（Unordered Expression）。

至於條件表示法（Conditional Expression），即是在查詢句中加入”*if … then … else …*”的條件判斷表示法，量化表示法（Quantified Expression）用兩個運算子“*some*”、”*every*“來做限制，例如”*every*“，它的意思代表每一個元素（屬性）必須要滿足某些條件，這兩種表示法和 Expressions on Sequence Types 此三種表示法由於語意上較複雜，且轉換到 SQL 上沒有適合的語意，因此這三種表示法不在本論文的處理範圍內，表 2.1 將欲處理的 XQuery 表示法列出解決方法和目標。

可以處理的 XQuery 文法範圍，圖 2.3 定義之，在本論文中所處理的複雜 XQuery 查詢句是 FLOWR 表示法，且可具巢狀結構、Wildcard 表示法、聚合函式和一些特殊函式。可處理的 XQuery 文法上，除了會轉換基本的 PC 路徑

基本表示法	文數值資料會直接轉換，至於連結變數“$Var”，轉換核心會以此連結變數為主設計本論文自訂的查詢樹。
路徑表示法	設計對應表格（Mapping Table）和連結對應表格（Join Mapping Table）整合 DTD 和 Schema，並提出演算法轉換路徑成具有 SELECT、FROM 和 WHERE 子句的等價 SQL 片斷。
順序表示法	轉換成 SQL 後的輸出順序必須和 XQuery 同義，之後再直接轉換運算子。
算術表示法	直接轉換成同義的運算子。
比較表示法	
邏輯表示法	
結構子	SQL 輸出欄位，以自我定義標籤名稱重新命名。
非排序表示法	當有 fn:unordered 函式，轉換後的輸出可不考慮順序性。
FLOWR 表示法	提出演算法處理 FLOWR 五種字句，並考慮到查詢句有巢狀結構時的處理。

表 2.1：XQuery 表示法的解決方法和目標

（Parent-Child），對於 AD 路徑（Ancestor-Descendant）和 Wildcard 中的“*”也會處理，因為 XQuery 是以路徑表示為導向，所以找出路徑對應到等價的 SQL 為最基本的要件，對應方法在第四章會詳細說明。

巢狀結構的表示，如範例 2.1，LET 子句變數$a 後接一份 FLOWR 表示句，或是 RETURN 子句裡再包含 FLOWR 表示句，必須清楚地表示每個子句間的階層關係，本論文會使用 Level Number 編碼表示查詢句每個子句的階層關係，Level Number 定義和轉換 FLOWR 敘述句在第四章會詳細說明。

FLOWR ::= ForClause + LetClause + WhereClause? + OrderByClause? +
ReturnClause
ForClause ::= FOR $fv1 IN E1, ... ,$fvn IN En
LetClause ::= LET $lv1 ::= E1, ... ,$lvn ::= En
WhereClause ::= WHERE φ(E1, ... ,En)
OrderByClause ::= OrderBy φ (E1, ... , En) (“ascending” | “descending”)?
ReturnClause ::= Return {E1}, ... ,{En}
Ei ::= FLOWR | Expr
Expr ::= Var
| Literal
| PathExpr
| Var “/” PathExpr
| UnaryOp Expr
| Expr BinOp Expr
Var ::= $fvn | $lvn
Literal ::= String | Integer | Float | …
PathExpr ::= RegularExpr | "/" RegularExpr | "//" RegularExpr
RegularExpr ::= Step | RegularExpr "/" Step | RegularExpr "//" Step
Step ::= NameTest | "@" NameTest
NameTest ::= QName | Wildcard
Wildcard ::= "*"
UnaryOp ::= “+” | “-“ | “not”
BinOp ::= CompOp | ArithOp | LogicalOp
CompOp ::= “=” | “!=” | “<” | “<=” | “>” | “>=”
ArithOp ::= “+” | “-“ | “*” | “div”
LogicalOp ::= “and” | “or”
AggFun ::= “count” | “avg” | “min” | “max” | “sum”
另外在 RETURN 子句中支援 count、avg、min、max、sum 等聚合函式。
在 WHERE 子句中支援 contains 和聚合函式限制。
在 FOR 子句中支援 distinct-values 函式。

圖 2.3：XQuery 文法範圍

第三章　對應表格

在本章中，首先介紹本論文的查詢語言轉換系統 — FORET（FOr tree and REturn Tree）的架構，和說明系統中的每個元件以及流程。在系統中有二個很重要的參考資料，整合 XQuery 查詢句和 SQL 查詢句之間的結構差異性的對應表格（Mapping Table）和連結對應表格（Join Mapping Table），會對此二份對應表格做詳細說明和建立步驟，最後藉由對應表格和連結對應表格，求得路徑查詢句的等價 SQL。

3.1 系統架構

查詢語言轉換系統（簡稱 FORET）的架構如圖 3.1 所示，使用者下達一份 XQuery 查詢句後，經過 Parser 產生內部自訂的表示式，對於 FLOWR 中的每個子句分別去建立 For Tree 和 Return Tree，而 For Tree 和 Return Tree 中每個節點的對應 SQL 從對應表格（Mapping Table）和連結對應表格（Join Mapping Table）中得到，而產生的 For Tree 和 Return Tree 不止一棵，將每棵 For Tree 收集起來即形成 For Forest，Return Tree 亦然，會形成 Return Forest，接著將 Forest 中每棵樹的每個節點 SQL 做合併、合理化的處理（SQL Constructor），最後將 SQL 做簡化或檢查語意缺漏處（SQL Validater）。

直覺上來說，會將 XQuery 轉換成 For Tree 和 Return Tree，而轉換後與關聯式資料綱要最重要的連繫點即在對應表格和連結對應表格上，DTD 中每條路徑表示式所對應的 SQL，都藉由對應表格元件得知，合併之後的 SQL 也仍要參考對應表格的資訊做更合理化的查詢句修飾。以下就對 FORET 系統中的每個元件做簡單的描述，詳細做法內容後面章節會一一詳述：

3.1.1 Parser：

剖析使用者輸入的 XQuery 查詢句，如果查詢句在文法範圍內會將 XQuery

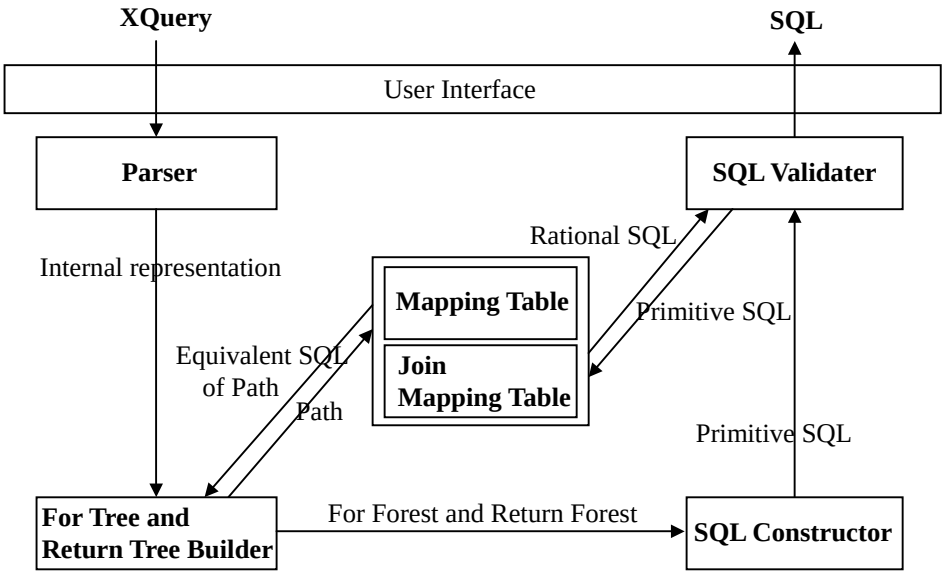


圖 3.1：FORET 系統架構

轉成自己定義的內部表示式。

3.1.2 For Tree and Return Tree Builder：

對每一個在 FOR、LET 子句中的連結變數（Binding Variable），產生一棵以其變數為根節點的 For Tree，再將 WHERE 子句中連結變數的限制句加入至 For Tree 中；對每一個在 RETURN 子句中的連結變數，亦產生一棵以其變數為根節點的 Return Tree，ORDER BY 子句則連結變數加入至 Return Tree 中。在建立這兩種查詢樹時，每一棵樹都有唯一的階層碼（Level Number）可做分辨，且每一個節點都會有對應的 SQL 片斷，最後將所有 For Tree 形成一集合，稱之為 For Forest；Return Tree 形成一集合，稱之為 Return Forest。

3.1.3 SQL Constructor

傳入的 For Forest 和 Return Forest，對每一棵樹，將所有節點的 SQL 片斷合併起來，接著根據階層碼之間的關係再合併每棵樹的 SQL 片斷成為一份原始的

SQL（Primitive SQL）。

3.1.4 SQL Validater

原始的 SQL 傳入後，(1)會先檢查表格間的連結關係是否缺少；(2)輸出資料有聚合函式時，檢查是否需要分群依據欄位。

3.1.5 Mapping Table and Join Mapping Table

對應表格記錄了 DTD 路徑和關聯式資料綱要的對應，剖析過後的路徑表示式，可經由 Mapping Table 和 Join Mapping Table 這個元件得到此路徑等價的 SQL 片斷，且之後的 SQL Valider 亦需要參考對應表格和連結對應表格讓 SQL 查詢句更合理化。

3.2 對應表格與連結對應表格

在定義對應表格和連結對應表格之前，對 DTD 會先做編碼的處理，編碼是為了解決路徑查詢句中的多個父元素的元素（屬性）、AD（Ancestor -Descendant）路徑和 Wildcard 問題，當編碼完成後才會建立對應表格，並從 RDB 的限制再建立連結對應表格。

3.2.1 DTD Tree 與區間編碼

將 DTD 以樹狀結構展開，如圖 3.1，在 DTD 定義中有多個父元素的元素或屬性，若由根節點向下搜尋，路徑會不相同，如 item 下的 name 和 person 下的 name，其路徑分別為 site/items/item/name 和 site/people/person/name，因此在其父元素下分別展開出一個獨立的 name 元素，接下來將 DTD Tree 中的每一個元素或屬性以 Preorder（前序）的方式開始編碼，每個 DTD 中的節點包含兩個數字：左區間碼和右區間碼。由 0 開始，遞增加 1，當遇到葉節點路徑時返回，返回時

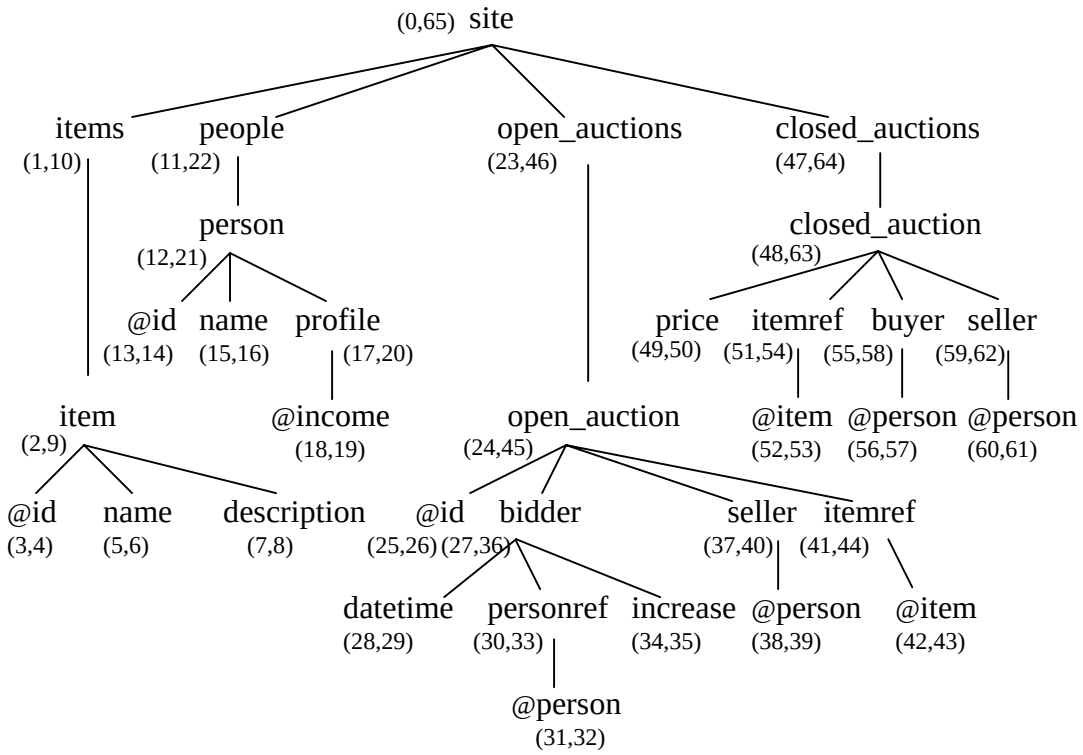


圖 3.2：區間編碼後的 DTD Tree

數字依然遞增加 1，此編碼稱之為區間編碼（Interval Encoding）。

【定義 3.1】：區間編碼（Interval Encoding）

令 DTD Tree 為 d，d 中的每一個節點 s，s 包含的左區間碼 L 和右區間碼 R 有下列的限制：

- L＜R，對每個（s，L，R）∈ d，
- 假如 s_1 是 s_2 的祖先，（s_1，L_1，R_1）∈ d 且（s_2，L_2，R_2）∈ d，必滿足 L_1 ＜L_2 且 R_1＞R_2。

編碼完成後，如圖 3.2，得到每個節點皆有一組左右區間碼，利用此區間碼可以判斷出節點的祖孫關係，例如 name（15，16）元素，只要所有節點的 L 小於 15 且 R 大於 16，即為 name（27，36）的祖先，符合條件的有 site（0，61）、people（11，22）和 person（12，21）。再從 DTD Tree 可知，在 DTD 中每一條

根節點到葉節點的所有可能路徑都被展開，路徑中的每一個元素或屬性都有自己的左右區間碼，換言之，若列出此 DTD 的每一條可能路徑，且以路徑最後一個元素（屬性）的區間碼做為此條路徑的區間碼，即 DTD Tree 中的每一條可能路徑都有一組唯一的區間碼。

【範例 3.1】

site/people/person/name 這條路徑，根據 name 的區間碼（15，16），則可以得到 site/people/person/name 此路徑的區間碼是（15，16），利用區間碼的特性，可得到所有祖先節點，而這些祖先節點所代表的路徑又分別為 site（0，61）、site/people（11，22）、site/people/person（12，21）。

3.2.2 對應表格與連結對應表格

對應表格主要記錄 DTD Tree 中的每條可能路徑對應到 RDB 的表格或欄位，因為根據 XQuery 的路徑表示導向特性，所以在轉換 XQuery 成 SQL 時，即是在處理每條路徑所對應到的 SQL 查詢句，以下定義 3.2 是對應表格的定義：

【定義 3.2】：對應表格（Mapping Table）

對應表格 M={DTDPath, Tag, L, R, RTable, RField} 由六個欄位組成，其中：

- DTDPath 為 DTD 中的每一條路徑。
- Tag 為 DTDPath 中最後一個元素或屬性的標籤名稱（TagName）。
- L 為 DTD 路徑用區間編碼後的左區域數值，$L \in N$。
- R 為 DTD 路徑用區間編碼後的右區域數值，$R \in N$。
- RTable 為 DTD 路徑對應到關聯式資料庫表格名稱。
- RField 為 DTD 路徑對應到關聯式資料庫欄位名稱。

XQuery 以路徑表示法為主，所以下達的路徑查詢句需要知道對應的關聯式表格和欄位資訊，以 DTD 中的每一條路徑為主，記錄其對應的關聯式表格和欄

DTDPath	Tag	L	R	RTable	RField
Site	site	0	61		
site/items	items	1	10	Item	
site/items/item	item	2	9	Item	
site/items/item/@id	@id	3	4	Item	ID
site/items/item/name	name	5	6	Item	Name
site/items/item/description	description	7	8	Item	Description

表 3.1：對應到圖 2.2 的關聯式綱要的部份對應表格

位，當使用者下一份 XQuery 查詢句時，部析出許多路徑查詢句，系統會根據此份對應表格，找出路徑查詢句對應的表格或欄位。而決定路徑對應到的表格和欄位，當使用改良的 Hybrid Inlining 做法，由於是從 DTD 路徑產生表格和欄位，因此在產生過程中即可以知道路徑對應的表格和欄位；若是要建立制訂好的 DTD 和 RDB 之間的對應表格，參考[YLL03、XE03]提出的方法，將 DTD 中元素（屬性）對應到表格或欄位分數最高視為相對應。

表 3.1 是對應表格中根節點 site 和 site/items 路徑下所有的欄位記錄，使用者在 XQuery 中的 FOR 下達 *"FOR $i IN doc("auction.xml")/site/items/item"*，剖析之後得到變數$i 的路徑查詢句為 site/items/item，拿此路徑到對應表格中查詢，可得到對應到 RDB 中的 Item 表格。

從表 3.1 可看到，根節點 site 並沒有對應到任何表格，主要原因是因為 Inlining 過程中並沒有任何有的元素或屬性可以併入成為 site 的欄位，而且在一般的 DTD 和 Schema 情況下，根節點大多對應資料庫名稱，因此不會有對應的表格或欄位，且在圖 2.1 的 DTD 中，根節點僅做為整合不同份 DTD 的標籤，items、people、open_auctions 和 closed_auctions 可視為是四份不同的 DTD，利用 site 做為根節點，將此四份整合成一份 DTD。

記錄左右區間碼和標籤名稱，是為了解決多個父元素的元素（屬性）、AD 路徑"//"和 Wildcard 表示"*"的問題，因為使用者在下路徑查詢句時不一定會輸入完整的 PC 路徑，利用此左右區間碼加標籤名稱可以順利地找出正確的路

徑，主要想法是以 Bottom-Up 的方式尋找出在對應表格中正確的路徑。

首先對於 AD 路徑，AD 路徑最難以解決的是無法得知中間經過的節點，如“site//item”，缺乏了中間需要經過的節點資訊，在 DTD 中完整的路徑會是“site/items/item”，而求得正確路徑後，才可轉換成對應的等價 SQL，為了解決 AD 路徑，可以從 M 中得到正確路徑，再產生等價 SQL。

路徑表示法中亦會遇到多個父元素（屬性）的路徑，因為多個父元素的元素（屬性）最大困難即是要找出到底是由那個父元素所指下來，例如在 site//item//name 路徑中，由標籤 name 至 M 中尋找，可以得到“site/items/item/name”、“site/people/person/name”兩條路徑，但是後者路徑並非正確答案，再根據 name 前面的 item 標籤做區間碼限制，才得以知曉正確路徑是“site/items/item/name”。至於 Wildcard 路徑，若使用者下的路徑查詢句中最後一個標籤是*，則必須採用 top-down 方式處理。

當剖析完 XQuery 後對每條路徑查詢句，先將路徑中所有的屬性（元素）標籤名稱 t 取出，得到 t_1. t_2..... t_n，先讓最後一個標籤 t_n 和對應表格 M 中 Tag 欄位所有標籤名稱做比對，取出所有 Tag 的名稱和 t_n 相同的資料列，並且得到所有根節點到 t_n 節點的路徑 $p_1.p_2...p_k$。得到所有可能路徑後，由 p_1 開始，去比對 t_{n-1} 對應到的所有路徑中是否有左區間碼 L 小於 p_1 的 L 且右區碼 R 大於 p_1 的 R，一直做到 t_1，只要有一個節點的所有路徑中都無法滿足區間碼限制關係，即代表 p_1 這條路徑不是正確路徑，然後再換下一條路徑 p_2 繼續重覆比對的工作，直到 p_k。

當 DTD 路徑對應至 Schema 表格或欄位時，某些路徑會對應到主鍵值或外鍵值型態的欄位，在關聯式資料庫中，主鍵值（Primary Key）和外鍵值（Foreign Key）是很重要的特性，主鍵值是判斷資料唯一性、不可重複性的依據，而外鍵值則是表格間連結的重要橋樑，例如 Item 表格的 ID 欄位是主鍵值，為每一個物品的身份辨認，Open_Auction 表格中的 Itemref 欄位是外鍵值，參考到 Item 表格的 ID 欄位，當要從 Open_Auction 表格中得到起標物品的詳細資料，必須藉由

Itemref 欄位參考到 Item 表格得到物品資料。

因為主鍵值和外鍵值的特性，在轉換 XQuery 查詢語言成 SQL 查詢語言時，必須考慮到外鍵值和主鍵值的連結關係，對應表格中僅記錄每條路徑對應的表格和欄位，若路徑對應到的是外鍵值欄位，從對應表格中是無法得知參考到的主鍵值表格和主鍵值欄位，因此還需要另一份連結對應表格（Join Mapping Table），將對應到外鍵值欄位的路徑，額外記錄在連結對應表格中，並從 Schema 資訊得到參考到的主鍵值表格和欄位，以下為連結對應表格的定義：

【定義 3.3】：連結對應表格（Join Mapping Table）

連結對應表格 JM={DTDPath, PTable, PK, FTable, FK} 由八個欄位組成，其中：

- DTDPath 為 DTD 中的每一條路徑。
- PTable 為外鍵值欄位參考到的關聯式資料庫表格名稱。
- PK 為外鍵值欄位參考到的關聯式資料庫主鍵值欄位名稱。
- FTable 為對應到的外鍵值表格名稱。
- FK 為對應到的外鍵值欄位名稱。

建立 JM 時是以 Relational Schema 為主，從 Relational Schema 中取出所有外鍵值和主鍵值的參考關係，因為在 M 中已經將所有路徑對應到的表格或欄位資訊記錄起來，所以若表格中有外鍵值欄位，至 M 中比對 RTable 和 RField 是否相同後，取出資料列，DTDPath 即為 DTD 的路徑中對應到外鍵值欄位的路徑，接著從 Schema 中取出參考的主鍵值表格和欄位，加入到 PTable 和 PK 欄位，最後將外鍵值表格和欄位加入到 FTable 和 FK 中。

此外，亦有可能外鍵值欄位在 M 中找不到對應的資料列，例如 Bidder 表格中的 OID 欄位，它參考到 Open_Auction 表格的 OID 欄位，可是 DTD 中並沒有任何一條路徑對應到 Bidder 表格的 OID 欄位。遇此狀況，會以「表格」的對應的資料列為主，從 M 中取出表格對應的 DTDPath，再將其它資訊加入至 JM 中

DTDPath	PTable	PK	FTable	FK
site/open_auctions/ open_auction/seller/@person	Person	PID	Open_Auction	Itemref
site/open_auctions/ open_auction/itemref/@item	Item	ID	Open_Auction	Seller
site/closed_auctions/ closed_auction/itemref/@item	Item	ID	Closed_Auction	Itemref
site/closed_auctions/ closed_auction/buyer/@person	Person	PID	Closed_Auction	Buyer
site/closed_auctions/ closed_auction/seller/@person	Person	PID	Closed_Auction	Seller
site/open_auctions/ open_auction/bidder	Open_Auction	OID	Bidder	OID
site/open_auctions/open_auction/ bidder/personref/@person	Item	ID	Bidder	Personref

表 3.2：對應到圖 2.2 的關聯式綱要的連接對應表格

若以 Bidder 表格的 OID 欄位為例，會找出 Bidder 表格對應的路徑“site/open_auctions/open_auction/bidder”為 DTDPath，再加入參考的主鍵值 Open_Auction 表格和 OID 欄位至 PTable 和 PK 中、加入本身表格和欄位至 FTable 和 FK 中，表 3.2 為圖 2.2 關聯式綱要的連接對應表格。

3.3 路徑的等價 SQL

建立好對應表格和連結對應表格後，接著 FORET 系統以此為基礎，開始進行查詢句的轉換，其中會將 FLOWR 子句內容建立 For Tree 和 Return Tree，在定義 For Tree 和 Return Tree 前，必須先了解路徑的等價 SQL 如何產生。將 FLOWR 子句剖析過後，得到許多條路徑查詢句，其中會先對 *PathExpr* 範圍的路徑產生等價的 SQL 片斷，此 SQL 片斷僅包含 SELECT、FROM 和 WHERE 三個子句，當路徑得到等價的 SQL 片斷後，再根據是在 FLOWR 中的那一個子句去做處理。

路徑查詢句中除了一般的 PC 路徑，亦可以下達 AD 路徑和包括 Wildcard 的

路徑查詢句，可是在關聯式資料庫中並沒有類似 AD 路徑和 Wildcard 的觀念，必須先經過處理找出合理的完整路徑，再從對應表格和連結對應表格中找到對應的關聯式資料表格和欄位，除了這兩個問題之外，也有可能下達路徑查詢句中最後一個元素（屬性）在 DTD 中是多個父元素的元素（屬性），如果此路徑中包含 AD 路徑，更難判斷正確的合理路徑。

為了解決以上問題，對應表格中記錄了左右區間碼和標籤名稱，從對應表格中找對應路徑的想法在 3.2.2 節提到，考慮到以上三種情況和一般 PC 路徑的路徑查詢句下達方式，圖 3.3 和圖 3.4 為路徑查詢句找出對應 SQL 片斷的演算法。圖 3.3 的 Path2SQL 演算法中，L05 和 L06 先將路徑中的最後一個標籤名稱至 M 中比對和 Tag 名稱一樣的，接下來取出來所有符合的資料列（Tuple），倘若查詢路徑的最後一個元素（屬性）有多個父元素，L06 取出的 *tp* 就會有多筆資料列，但並非每一條資料列中的 DTDPath 都是正確的。

對每一筆資料列的路徑，必須去判斷是否為正確的路徑，L01 和 L02 做變數宣告，*tp-set*、*c-set* 和 *correct_set* 集合為資料列集合，所包含的資料列型態和 M 定義相同。*q* 為 SQL 片斷，可以由許多 SQL 片斷聯集而成，*tag_valid* 和 *err_path* 為條件判斷的布林值。

L07 迴圈是要對 *tp-set* 中的每筆資料列 *tp* 去做判斷，將查詢路徑的剩餘標籤，由後向前去 M 中尋找符合的資料列集合 *c-set*（L08～L09），令 *c-set* 中的每一個資料列 *ctp*，去和 *tp* 做區間碼判斷，每筆資料列 *ctp* 的區間碼是否包含 *tp* 資料列的區間碼，並將符合的 *ctp* 加入到 *correct_set* 集合中。而當資料列 *ctp* 中沒有符合條件的，L15 會中斷迴圈並讓 *err_path* 此布林值設為 true，代表現在這個資料列 *tp* 的資料不是正確的路徑，換下一筆資料列 *tp* 繼續做判斷。

對一筆 *tp* 找出所有符合區間碼包含性關係的 *ctp*，並且加入至 *correct_set* 集合中，圖 3.4 演算法 GenSQL 會產生 *correct_set* 集合對應的 SQL 片斷。L03 對 *correct_set* 集合中的所有資料列 *cp*，開始將資料列中的 RTable 資料和 RField 資料加入至 SQL 片斷中，L04 先檢查 *sql.FROM* 中有沒有重複的表格，沒有的話即

Algorithm Path2SQL(m, jm, p)	
輸入	對應表格 *m*，連結對應表格 *jm*，路徑查詢句 *p*
輸出	聯集後的 SQL 片斷 *q*
L01	*tp-set* = {}; *c-set* = {}; *correct_set* = {}; *q* = {};
L02	*tag_valid* = false; *err_path* = false;
L03	
L04	t_n = Get last tag name from *p*;
L05	*tp-set* = Get all tuple that *m.Tag* = t_n from *m*;
L06	
L07	**for**(each tuple *tp* from *tp-set*) {
L08	**for**(each tag t_k from t_{n-1} to t_1 in *p*) {
L09	*c-set* = Get all tuple that *m.Tag* = t_k from *m*;
L10	**for**(each tuple *ctp* in *c-set*)
L11	**if**(*ctp.L* < *tp.L* and *ctp.R* > *tp.R*) {
L12	add *ctp* to *correct_set*;
L13	*tag_valid* = true;
L14	}
L15	**if**(*tag_valid* != true) {
L16	*err_path* = true;
L17	**break**;
L18	}
L19	**else** *tag_valid* = false;
L20	}
L21	**if**(*err_path* != true)
L22	*q* = *q* ∪ GenSQL(*m*, *jm*, *correct_set*);
L23	**else** *err_path* = 0;
L24	}
L25	**return** *q*;

圖 3.3：演算法 Path2SQL

Algorithm GenSQL(m, jm, correct_set)	
輸入	對應表格 *m*，連結對應表格 *jm*，對應表格資料列集合 *correct_set*
輸出	單條路徑的 SQL 片斷 *sql*
L01	*sql* = {};
L02	
L03	**for**(each tuple *cp* from *correct_set*) {
L04	**if**(*cp.RTable* is not in *sql.FROM*) {
L05	add *cp.RTable* to *sql.FROM*;
L06	**if**(find join condition *jc* between *cp.RTable* and any a table of *sql.FROM* in *jm*)
L07	add *jc* to *sql.WHERE*;
L08	**else** {
L09	*sql* = {};
L10	**break**;
L11	}
L12	}
L13	**if**(*cp.RField* isn't empty)
L14	add *cp.RField* to *sql.SELECT*;

L15	}
L16	**return** *sql*;

圖 3.4：演算法 GenSQL

加入 RTable 至 *sql.FROM* 中（L05）。接著檢查 *sql.FROM* 裡的表格和加入的表格是否有 Join Condition，可以從 JM 中去找有無外鍵值參考到主鍵值的關係，若有的話，L06 加入 Join Condition 至 *sql.WHERE* 中；反之，將 sql 清除，並且離開迴圈，因為根據 Hybrid Inlining 轉出來的 Relational Schema 或制訂好的 Relational Schema，一條 DTD 路徑中，若對應到二個或二個以上表格，必存在表格間的主鍵值和外鍵值參考關係，因為必須保持資料關聯性。最後 L13 加入 RField 資訊到 *sql.SELECT* 中。

當每條路徑的片斷各自產生完畢後，最後在 Path2SQL 的 L22 行做聯集的動作，把對應到的 SQL 片斷聯集後輸出。

【範例 3.2】

從範例 2.1 中取出 FOR 子句中的“*document(“auction.xml”)//person*”路徑、WHERE 子句中的“*$p//@income > 5000*”路徑、“*site//name*”此條較特殊路徑，分別轉換出等價的 SQL 片斷。

- *document（”auction.xml”）//person*

由最後一個標籤 *person* 去 M 中找，找到 DTDPath 為“site/people/person”的欄位，取出所有區間碼包含“site/people/person”的區間碼的資料列後，再加上自己的資料列，產生的 SQL 片斷為“FROM *Person p* “，可以看到僅得到 FROM 子句的資訊，而 SELECT 和 WHERE 子句沒有任何資料。

- *$p//@income > 5000*

系統中的 Parser 會自動把此路徑剖析成“//person//@income >5000”，在步驟中只針對 PathExpr 範圍的路徑做產生等價 SQL 的動作，因此僅“//person//@income”路徑會動作，產生的等價 SQL 片斷為“SELECT *p.income* FROM *Person p*”。

- *site//name*

此條路徑根據演算法會找到“site/items/item/name”和“site/people/person/name”，這兩條路徑都是合理的答案，可是分別對應到不同的 SQL 片斷，首先“site/items/item/name”對應到的是“SELECT *i.name* FROM *Item i*”，而“site/people/person/name”對應到的是“SELECT *p.name* FROM *Person p*”，最後聯集這兩個 SQL 片斷，得到“(SELECT *i.name* FROM *Item i*) union (SELECT *p.name* FROM *Person p*)”這個正確的結果。

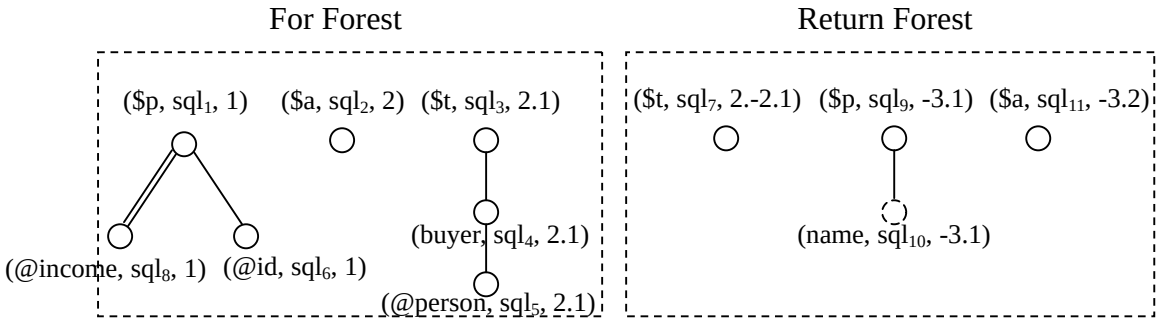
Path2SQL 演算法僅得到包含 SELECT、FROM 和 WHERE 子句的 SQL 片斷，例如範例 2.1 中的“*$p//@income > 5000*”限制句，經由 Path2SQL 之後僅得到“SELECT *p.income* FROM *Person p*”此 SQL 片斷，可是根據限制句的完整意思應該要轉成“FROM *Person p* WHERE *p.income > 5000*”，至於如何將由 Path2SQL 產生的 SQL 片斷變成更合理的 SQL 片斷，此一階段是在建立 For Tree 和 Return Tree 時動作，當 FLOWR 每一個子句建立成 For Tree 或 Return Tree 時，每個節點都會有其對應的 SQL，而此 SQL 會先從 Path2SQL 演算法中得到，再根據限制句的語意做正確的改寫。

第四章　For Tree and Return Tree Builder

剖析過後的 XQuery 在 FORET 系統中會轉成 For Forest（For Tree 集合）和 Return Forest（Return Tree 集合），這兩種查詢樹配合上 Level Number（階層碼），可表示 XQuery 的結構和巢狀關係。本章節會先定義 Level Number 和 Level Sequence，最後介紹 For Tree 和 Return Tree 的定義和建立演算法。

以下介紹範例 4.1，將範例 2.1 查詢句轉成以 For Tree 和 Return Tree 表示：

【範例 4.1】



sql_1　：FROM Person p

sql_2　：FROM (//SQL result of level 2.n) a GROUP BY ‘G1’ HAVING count(*) > 3

sql_3　：FROM Closed_Auction t

sql_4　：FROM Closed_Auction t

sql_5　：FROM Closed_Auction t, Person p WHERE p.pid = t.buyer

sql_6　：( )

sql_7　：SELECT * FROM Closed_Auction t

sql_8　：FROM Person p WHERE p.income > 5000

sql_9　：SELECT * FROM Person p

sql_{10}　：SELECT p.name as person FROM Person p ORDER BY p.name

sql_{11}　：SELECT COUNT(*) as item FROM (//SQL result of level 2.n) a

Level Sequence：[1, 2[2.1, 2.-2[2.-2.1]], -3[-3.1, -3.2]]

範例 4.1 即為 XQuery 轉成 For Tree 和 Return Tree 後的樣子，產生 For Tree 的最基本規則是對 FOR 和 LET 子句中每一個連結變數產生一棵根節點以其連結變數為名稱的 For Tree，因此產生了名稱為$p、$a 和$t 三棵 For Tree；相同的，建構 Return Tree 是對 RETURN 子句中每一個連結變數產生一棵根節點以其連結變數

為名稱的 For Tree，所以產生了$t、$p 和$a 三棵 Return Tree。For Tree 和 Return Tree 的節點都由三個部份組成，第一部份為名稱，第二部份為 SQL 片斷，最後是 Level Number，詳細定義於後介紹。

4.1 Level Number

建立 For Tree 和 Return Tree 基本準則是為每一個連結變數（Binding Variable）建立一棵以其連結變數為根節點的查詢樹，FOR、LET、WHERE 子句內容建立成 For Tree，RETURN、ORDER BY 子句內容建立成 Return Tree。將 FOR 和 LET 子句的每個連結變數建立一棵 For Tree，再將 WHERE 子句中的限制式，根據對應的連結變數，建立 For Tree 的節點；Return Tree 是 RETURN 子句中每一個連結變數產生的，而 ORDER BY 子句的限制式，根據對應的連結變數，建立 Return Tree 的節點。然而，以此種做法，在轉換複雜的 XQuery 查詢句成為這兩種查詢樹時遇到了三個很重要的問題：

1. For Tree 記錄的是資料連結、資料限制的部份，而 Return Tree 記錄的是資料回覆、資料排序，這兩種不同的查詢樹，要如何用統一的規則去分辨？
2. 具有巢狀結構的 XQuery 查詢句，要如何知道整個 XQuery 查詢句的巢狀階層關係？
3. 要如何將所有查詢樹對應的 SQL 查詢句整合起來？

For Tree 和 Return Tree 的根節點都是以連結變數為名稱，範例 2.1 中的$p 變數，在 FOR 子句中是連接到 person 元素，而在 RETURN 子句中是要將符合條件的 person 元素的 name 元素內容輸出，兩種的意義是不同的，因為 For Tree 主要記錄資料連結和資料限制；而 Return Tree 是記錄資料連結和資料輸出。

具有巢狀結構的複雜 XQuery 查詢句，要記錄巢狀階層的關係性，才可確保在轉換成 SQL 查詢句時，對應的 SQL 查詢句結果正確，例如範例 2.1 中的變數$a，連接一份 FLOWR 形成巢狀結構，而此 FLOWR 中 FOR 子句的$t，根據本論文提出的作法，會在 For Tree 集合中建立一棵 For Tree，可是此棵 For Tree 卻

是$a 的子樹。

建立好查詢樹，但最重要的是如何合併這些查詢樹所擁有的 SQL 片斷，而且還必須根據 XQuery 查詢句的結構性做合併，像變數$a，會產生一份 derived relation，但這份 derived relation 的內容卻是變數$a 後接的 FLOWR 子句的 SQL 查詢句，也就是必須要先將變數$a 裡的巢狀 FLOWR 子句產生的 For Tree 和 Return Tree 的 SQL 合併完成後，才能繼續外層的 SQL 合併工作。為了清楚分辨每棵查詢樹、XQuery 的結構關係和合併 SQL 片斷，定義了 Level Number（階層碼），以下為 Level Number 的定義：

【定義 4.1】：階層碼（Level Number）

Level Number *l*，是由一個或一個以上的 Level Component *lc* 組成，$\forall\ lc \in Z$ 且由 *1* 開始，每個 *lc* 之間以“.”隔開，因此 $l = lc_1.lc_2.....lc_n$，$n \in N$。並且具下列性質：

- *LevelCount(l)*，回傳一個正整數，計算有幾個 Level Component，表示此 *l* 的層數。
- *LastComponenet(l)*，取出 l 的最後一個 Level Component。
- 任兩個 Level Number l_m 和 l_n，
  - 若 $LevelCount(l_m) - LevelCount(l_n) = 0$，則 l_n 和 l_m 同層。
  - 若 $LevelCount(l_m) - LevelCount(l_n) = 1$，且 $l_m \supseteq l_n$，則稱 l_n 為 l_m 的父階層（Parent Level）。
  - 若 $LevelCount(l_m) - LevelCount(l_n) > 1$，且 $l_m \supseteq l_n$，則稱 l_n 為 l_m 的祖先階層（Ancestor Level）。

開始建立 For Tree 和 Return Tree 時，遇到 FOR 和 LET 子句的每個連結變數都給予一個唯一的 Level Number，一開始的 Level Number 僅只有一個 Level Component，由 1 開始遞增，WHERE 子句的限制式就依據連結變數，到連結變數的 For Tree 下新增節點。比較特殊的是，遇到巢狀結構的查詢句時，為了分辨

結構上的不同，因此會增加 Level Component 來表示，而在 FLOWR 表示法中定義下列三種情況為巢狀結構：

1. LET 子句的連結變數後接的是一份新的 FLOWR 子句。
2. RETURN 子句裡包含了新的 FLOWR 子句。
3. 遇到“return”關鍵字時。

第一種情況 LET 子句後接的是一份新的 FLOWR 子句，如範例 2.1 的連結變數$a 後又接一份 FLOWR 子句；第二種情況則是在 RETURN 子句中又包含了一份 FLOWR 子句；第三種情況是較為特殊的狀況，當遇到“return”關鍵字時，表示預備建立 Return Tree，依然給予 Level Number，因為一個“return“關鍵字可能接著很多變數要出輸出，所以將其視為巢狀結構，而為每一個輸出變成用 Level Component 表示。

可是卻無法從 Level Number 中清楚地分辨這棵查詢樹是屬於 For Tree 還是 Return Tree，因此當看到“return”關鍵字，會為“return”關鍵字這個部份增加一個“虛擬的 Level Number”，首先 *LastComponent(l)*依然遞增加 1，但是會變成負數，其目的是為之後的 SQL 合併，所有 RETURN 子句內的查詢樹合併好之後，會用此虛擬的 Level Number 包裝起來，再和 For Tree 的結果合併。如範例 4.2 的 Return Forest 中，所有的 Return Tree 的 Level Number 中都可以看到有負號的 Level Component 出現，表示此棵查詢樹是屬於 Return Tree，而且“return”關鍵字的 Level Number（如範例 4.3 中的-3、2.-2），也是往後在收採所有 SQL 片斷時很重要的資訊，收集 SQL 片斷部份於第五章詳述。

【範例 4.2】

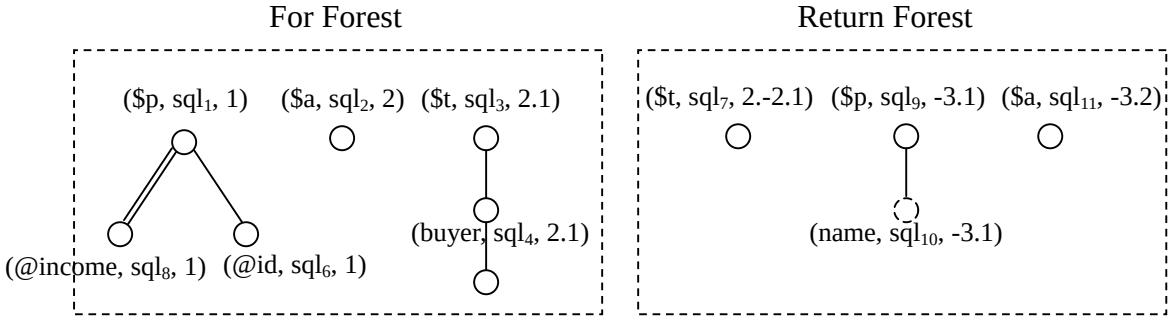
```
FOR $p IN document(“auction.xml”)//person
 LET $a := FOR $t IN document(“auction.xml”)//closed_auction
           WHERE $t/buyer/@person = $p/@id
           RETURN $t
WHERE $p//@income > 5000 AND count($a) > 3
```

ORDERY BY $p/name

RETURN { <items>

<item person=$p/name/text()> COUNT($a) </item>

</items>}



以範例 2.1 的 Q8 為例（如上），會得到 For Forest 中有三棵 For Tree；Return Forest 中有三棵 Return Tree，分別如下：

範　　圍	連結變數	Level Number
For Tree	$p	1
	$a	2
	$t	2.1
Return Tree	return 關鍵字	2.-2
	$t	2.-2.1
	return 關鍵字	-3
	$p	-3.1
	$a	-3.2

Level Number 初設只有一個 Level Component，由 1 開始，遇到每個 FOR 和 LET 子句的連結變數都給予一個唯一的 Level Number，因此 FOR 子句的變數$p 為 1，LET 子句的變數$a 為 2。而$a 接了新的 FLOWR 表示式形成巢狀結構，所以巢狀結構裡的 Level Number 必須加入一個 Level Component 以表示巢狀結構，

所以變數$t 為 2.1，代表是$a 的子階層。

當遇到“return”關鍵字時，代表要傳回值，Level Number 的最後一個 Level Component 加 1 外，還要變成負數代表進入 RETURN 子句的範圍，將所有傳回的部份看成一個巢狀結構，所以在 RETURN 子句裡的$t 的 Level Number 為 2.-2.1。以此類推，最外層 RETURN 子句裡的$p、$a 的 Level Number 分別為-3.1、-3.2。

定義完 Level Number 和說明在 FLOWR 子句中的改變情況，僅解決了清楚分辨每棵查詢樹和 XQuery 的結構關係，至於 For Tree 和 Return Tree 中每個節點的 SQL 片斷合併，仍然是沒有一個有效的方法。譬如前段所述敘過的範例 2.1 中的$a 變數，會產生一份 derived relation，但這份 derived relation 的內容卻是變數$a 後接的 FLOWR 子句的 SQL 查詢句，為此必須有儲存 Level Number 資訊的序列，藉由此序列可以有系統地合併 SQL，因此定義了階層序列（Level Sequence）：

【定義 4.2】：階層序列（Level Sequence）

Level Sequence *ls*，*ls* = {*l*}，任兩個 Level Number l_n 和 l_m，若 l_n 和 l_m 同層，加入 *ls* 中時，以“l_n, l_m”隔開表示；若 l_n 為 l_m 的父階層，將兩者加入至 *ls* 中時，會以“l_n [l_m]”表示。

【範例 4.3】

根據範例 4.2 得到的查詢樹 Level Number，會將 Level Number 依其關係加入 Level Sequence 中，因此得到： [1, 2[2.1, 2.-2[2.-2.1]], -3[-3.1, -3.2]]。

4.2 For Tree 與 Return Tree

一份良好的查詢語言最基本的包括資料連結、資料限制、資料輸出，在

FLOWR 子句中，資料連結的部份包括 FOR、LET 子句；資料限制的部份是 WHERE 子句；資料輸出則是 RETURN 子句，資料輸出又可以使用 ORDER BY 子句做資料排序的動作。在此，本論文的觀點是將資料連結和資料限制試為一個群組，將其內容轉成 For Tree；而資料輸出則是轉成 Return Tree，但 Return Tree 裡仍然包含資料連結的資訊，確保來源和輸出的值是一致的。

為了有效地轉換 FLOWR 子句成 For Tree 和 Return Tree，在此定義 For Tree 和 Return Tree 的節點、查詢樹結構，並提出建立的演算法。

4.2.1 基本定義

在介紹 For Tree 和 Return Tree 前，起初必須先定義兩種查詢樹的節點內容，此節點稱之為 FORET Node，適合在 For Tree 和 Return Tree 中，只是會因為在不同的查詢樹中而有儲存資訊上的差異。並且對 WHERE 限制式中的類型做一個分類，因為不同的限制式類型，在轉換成 For Tree 後，會有不同的表示方式，而且也必須考慮到路徑的結構限制是否能順利對應到表格欄位的結構限制。

【定義 4.3】：FORET Node

對每個 FORET Node *fn*，由三個欄位組成，*fn* = {Vname , SQLFrag , LN}，其中：

- Vname = binding variable | element | attribute
- SQLFrag = {SELECT , FROM, WHERE, ORDER BY, GROUP BY, HAVING}六個 SQL 子句的片斷。
- LN ∈ *l*。

每個節點都會有名稱，其名稱可能是連結變數、元素名稱或屬性名稱，其中僅只有查詢樹的根節點的名稱才為連結變數，例如以$p 為節點名稱，節點中還有記錄著 SQL 片斷，SQL 片斷是路徑查詢句經由 Path2SQL 演算法得來，在節

點中的 SQL 片斷是調整過的，例如在 WHERE 子句產生 For Tree 的節點後，要將 SELECT 的欄位移至 WHERE 中寫成正確的限制式，而且在 For Tree 和 Return Tree 中的節點，所包含的 SQL 片斷範圍不一樣，如 For Tree 中的 SQL 片斷包含 FROM、WHERE、GROUP BY、HAVING 子句；Return Tree 中包含 SELECT、FROM、ORDER BY 子句，因為節點中包含的 SQL 範圍不同，所以查詢樹會分開建立。最後的 LN 即是 Level Number，每個根節點的 Level Number 都是唯一的，根節點其下的節點，Level Number 都會和根節點一樣。

在定義 For Tree 之前，必須對於 WHERE 子句做更明確的定義，由於 Path2SQL 演算法所產生的 SQL 片斷僅是對路徑查詢句做轉換，並未對路徑語意做分析，例如$p//@income > 5000，只得到$p//@income 的等價 SQL，此路徑的語意限制並未考慮進去，而且這只是針對某個元素（屬性）做數值的限制，倘若是兩條路徑的連結限制，需要考慮的層面更複雜。為此要為限制式的類型做分別：

【定義 4.4】：Selection Condition 和 Join Condition

在 FLOWR 子句的 WHERE 子句中，若限制式為和 Literal 範圍比較的（包含函式，如 contains、聚合函式限制），稱之為 *Selection Condition*；若限制式為兩條路徑做比較，稱之為 *Join Condition*。

【範例 4.4】

範例 2.1 中的 WHERE 限制式，其中：

- $p//@income > 5000 和 count($a) > 3 為 Selection Condition
- $t/buyer/@person = $p/@id 為 Join Condition

Selection Condition 除了一般路徑與 Literal 比較外，WHERE 子句中支援的 contains 函式和聚合函式，也屬於 Selection Condition 的範圍，因為 contains 和 SQL 中的“LIKE”比對限制式意義相符，做字串的比對；而聚合函式的限制都為數字的比較，比較特別的是，在 SQL 中聚合函式的限制是在 HAVING 子句中

下達的，可是在 XQuery 中沒有任何相關的子句，反而是將聚合函式的限制句放在 WHERE 子句中，當轉成等價的 SQL 時，必須將限制式加入至 HAVING 子句中，可是 SQL 的語法中，必須要先有 GROUP BY 子句的產生，才能有 HAVING 子句，因此聚合函式的限制句除了轉換成 HAVING 子句外，又必須考慮到要依那些欄位做分群。

至於 Join Condition，因為限制的是兩條路徑，可能是來源不同的路徑，亦有可能是來源相同，不管如何，都必須產生兩份資料來源去做比對，對應至 SQL 的話，就是表格間做連結關係。在 FLOWR 子句中，Join Condition 的路徑限制式若再配合上巢狀結構，子階層內可以使用父階層的資料連結來源，甚至可以用祖階層的資料連結來源，至於用到那一階層的資料來源，利用到前面定義的 Level Number 可以解決此問題。

而轉換 Join Condition 成等價 SQL 後，若涉及到來源不同的 For Tree，表示兩棵 For Tree 之間會做連結關係，如果 For Tree 之間非同層關係，而是有父子或祖孫階層關係，必須考慮到轉換後的 SQL 片斷置放位置，以呼應查詢句的巢狀結構關係。

4.2.2 For Tree

FLOWR 子句中的 FOR、LET、WHERE 子句的內容會產生成 For Tree，對每個連結變數會產生一棵 For Tree，而 WHERE 子句的內容則會根據對應的連結變數產生在其 For Tree 下，在此會定義 For Tree 的架構，包括根節點、非葉節點和葉節點。

【定義 4.5】：For Tree

對每個 For Tree *ftree* =（FN, E），其中 FN 為 *fn* 節點集合，E 為 PC 路徑和 AD 路徑集合。*ftree* 中的 *fn* 節點可分為三種類型，根節點（root node）、非葉節點（non-leaf node）、葉節點（leaf node），每種節點具有下列性質：

- 根節點：

- 根節點的 Vname 為 FOR 子句、LET 子句中的連結變數名稱。
- 根節點的 SQLFrag，連結變數對應的路徑查詢句，由 Path2SQL 演算法產生，再判斷是在 FOR 或 LET 子句中的那種型態，修改 SQLFrag。
- 每個根節點的 LN 都是唯一的。

- 非葉節點：
  - 非葉節點的 Vname，為 WHERE 子句中限制式路徑中非最後一個元素（屬性）的元素名稱。對每個元素，建立一個節點在對應的 *ftree* 下。
  - 非葉節點的 SQLFrag，為對應根節點的路徑查詢句加上到本身的路徑，由 Path2SQL 演算法產生。
  - 非葉節點的 LN，和根節點的 LN 相同。
- 葉節點：
  - 葉節點的 Vname，為 WHERE 子句中限制式路徑中最後一個元素（屬性）的名稱，建立一個節點在對應的 *ftree* 下。
  - 葉節點的 SQLFrag，為對應根節點的路徑查詢句加上到本身的路徑，由 Path2SQL 演算法產生後，再判斷限制式類型、語意修改 SQLFrag。若限制式為 Selection Condition，將限制式轉為對應 SQL 字句加入至葉節點的 SQLFrag 中；若限制式為 Join Condition，讓兩邊路徑查詢句對應到的 For Tree 為 $ftree_1$ 和 $ftree_2$，(1)若 $ftree_1$ 的 LN 和 $ftree_2$ 的 LN 的同層，將限制式的對應 SQL 任意擇選加入一邊的葉節點；(2)若 $ftree_1$ 的 LN 為 $ftree_2$ 的 LN 的父階層或祖先階層，將限制式的對應 SQL 加入 $ftree_2$ 的葉點節中，$ftree_1$ 葉節點的 SQLFrag 僅包含 FROM 子句。
  - 葉節點的 LN，和根節點的 LN 相同。

每個節點的 SQLFrag 會反應查詢句的語意，可是在產生正確的 SQLFrag 之前，路徑經由 Path2SQL 產生後並不一定是正確的，根據每個子句所代表的意義，產生出來 SQL 片斷後，再根據語意做適當的修改，例如 WHERE 子句中，限制

式中的路徑由 Path2SQL 產生出來的 SQL 片斷會將欄位放在 SELECT 子句中，但是根據整個限制式意義，應該要將此欄位移到 WHERE 中，再加上限制式的語意內容。因此對於 FOR、LET 和 WHERE 三個子句，除了對路徑產生等價的 SQL 片斷外，還必須依照語意內容做 SQL 片斷的適度修改，每個子句詳細的 SQLFrag 產生演算法於 4.2.4 節說明。

Join Condition 限制式方面，涉及到兩棵 For Tree，就必須考慮到兩邊的階層關係，若兩邊 For Tree 同層，任意選擇一 For Tree，將 SQL 片斷置入葉點節中，因為同層，所以合併時會有重複的資訊出現，因此只需要找一邊加入；不同階層時，只需要將 SQL 片斷置入子（子孫）階層 For Tree 的葉點節中，理由是有巢狀結構的 FLOWR 子句，子（子孫）階層可以引用到父（祖先）階層的資料連結來源，根據本論文的作法，會將巢狀結構的內容轉為 Derived Relation，因此 Derived Relation 內會有父（祖先）階層的對應資料連結來源，若父（祖先）階層 For Tree 的葉點節中也加入 SQL 片斷，查詢句的語意上不合理，因為巢狀結構中，子（子孫）階層可以看到父（祖先）階層，但父（祖先）階層看不到子（子孫）階層的資訊，若父（祖先）階層也加入 SQL 片斷，在合併時會變成兩份不同的表格。

4.2.3 Return Tree

FLOWR 子句中的 RETURN、ORDER BY 子句內容會建立成 Return Tree，對 RETURN 裡的每個連結變數都會產生一棵 Return Tree，由於看到“return”關鍵字時，即將“return”所包含的部份視為一個巢狀結構，而裡面可再包含 FLOWR 子句形成巢狀結構，根據 Level Number 編碼規則，遇到“return”關鍵字時，會增加 Level Component，讓“return”裡包含的連結變數或 FLOWR 子句做有階層關係的 Level Number 編碼。

和 For Tree 不同的是，Return Tree 的節點所包含的 SQLFrag 不同，For Tree 節點包含的 SQLFrag 範圍為 FROM、WHERE、HAVING，而 Return Tree 的範圍

是 SELECT、FROM、ORDERY BY、GROUP BY，所表現的是查詢句中的資料連結、資料輸出和資料排序的部份，因此節點的定義上會和 For Tree 略有不同。

【定義 4.6】：Return Tree

對每個 Return Tree *rtree* =（FN, E），其中 FN 為 *fn* 節點集合，E 為 PC 路徑和 AD 路徑集合。*rtree* 中的 *fn* 節點可分為三種類型，根節點（root node）、非葉節點（non-leaf node）、葉節點（leaf node），每種節點具有下列性質：

- 根節點：
  - 根節點的 Vname 為 RETURN 子句中的連結變數名稱。
  - 根節點的 SQLFrag，連結變數對應的路徑查詢句，由 Path2SQL 演算法產生，其中若 SELECT 子句中為空，則以“SELECT *”表示。
  - 每個根節點的 LN 都是唯一的。
- 非葉節點：
  - 非葉節點的 Vname，為 RETURN 子句中傳回值路徑中非最後一個元素（屬性）的元素名稱。對每個元素，建立一個節點在對應的 *rtree* 下。
  - 非葉節點的 SQLFrag，為對應根節點的路徑查詢句加上到本身的路徑，由 Path2SQL 演算法產生，其中若 SELECT 子句中為空，則以“SELECT *”表示。
  - 非葉節點的 LN，和根節點的 LN 相同。
- 葉節點：
  - 葉節點的 Vname，為 RETURN 子句中傳回值路徑中最後一個元素（屬性）的名稱，建立一個節點在對應的 *rtree* 下。
  - 葉節點的 SQLFrag，路徑查詢句先由 Path2SQL 演算法產生。
  - 葉節點的 LN，和根節點的 LN 相同。

Path2SQL 演算法產生出來的 SQL 片斷僅包含 SELECT、FROM、WHERE 三個子句，而 Return Tree 中的節點僅需要 SELECT 和 FROM 的資訊，WHERE

的表格連結限制，在往後做 SQL 合併時即可以得到。而當路徑查詢句只得到表格沒有欄位時，表示 SELECT 子句中為空，會用“*”選擇所有欄位資料。

在建立 Return Tree 過程中，有兩個比較特殊的地方，第一、ORDER BY 子句中欄位的建立；第二、遇到聚合函式和一般路徑輸出時，GROUP BY 子句中欄位的建立。當 FLOWR 子句中出現 ORDERY BY 子句時，表示要依 RETURN 子句中的元素（屬性）做排序，對應到 SQL 即是依 SQLFrag.SELECT 中的欄位做排序，因此 ORDERY BY 子句的排序路徑在 RETURN 子句中是一定會出現的，因此遇到 ORDER BY 子句時，根據連結變數，從 Return Tree 集中找到對應的 Return Tree，並在對應的葉節點中 SQLFrag.ORDER BY 欄位，因此嚴格說起來，ORDER BY 子句中的內容並沒有獨立建立 Return Tree，而是從建好的 Return Tree 集合中找到符合的路徑，然後將對應欄位加入 SQLFrag.ORDER BY。而有 ORDER BY 子句的葉點節，會以虛線表示之，如範例 4.1 的 Return Tree $p，其葉節點有 ORDERY BY 資訊，因此以虛線表示之。

另一個特別的地方就是聚合函式，當 RETURN 子中出現傳回路徑由聚合函式所聚合時，SQLFrag.SELECT 中會有對應的聚合欄位，而在一般的 XQuery 查詢句寫法中，若 RETURN 中除了有聚合函式外，又包含了其它一般路徑，表示需要做分群的語意，對應到 SQL 會產生 SQLFrag.GROUP BY 子句。此部份在第五章的 SQL Valider 中會根據自訂的規則，分析查詢句語意，做 GROUP BY 子句欄位的新增或修改。

4.2.4 For Tree 與 Return Tree 建立演算法

輸入剖析過後的 FLOWR 子句，會建立出 For Forest 和 Return Forest，最後必須將這兩個查詢樹集合和 Level Sequence 至入一個新集合中，稱之為 FRS（For Forest and Return Forest Set），以下為 FRS 的定義。

【定義 4.7】：FRS（For Forest and Return Forest Set）

輸入剖析過後的 FLOWR 子句，會建立出一份 FRS *frs* = (ForForest, ReturnFroest, *ls*)，其中 ForForest 為 *ftree* 的集合，ReturnFroest 為 *rtree* 的集合，*ls* 為 Level Sequence。

為了有系統地建立出 FRS 中的 For Forest、Return Forest 和 Level Sequence，本論文提出 FRS 演算法，針對 FLOWR 子句的每個子句和結構，建立出 For Tree 和 Return Tree，因為所處理的查詢句範圍包括巢狀結構的查詢句，因此演算法的架構上是遞迴的型式做處理，而且每個子句中的 SQL 片斷處理盡不相同，必須考慮到可能出現的文法語意，並且限制可處理的範圍，例如“/site/items/item”路徑，在 FOR 子句出現的 SQL 片斷和在 RETURN 子句出現的不會相同，但經由 Path2SQL 演算法後得到的 SQL 片斷是一樣，因此對每個子句內的路徑需要另外的 SQLFrag 處理函式。

圖 4.1 為建立 For Tree 和 Return Tree 的 FRS 演算法，對 FLOWR 子句的五個部份分別做建立動作，在下查詢句時必須依照 FOR、LET、WHERE、RETURN、的順序下，而 LET 和 REUTURN 中可包含新的 FLOWR 子句，也必須依照此順序。演算法開始時傳入值，*ln* 初設為 1，演算法裡宣告 *nest_ln* 此暫存的 Level Number，因為是採遞迴的演算法，所以遇到巢狀結構時，必須傳入一份巢狀結構的初設 Level Number，因此宣告 *nest_ln*。

首先 L03 是處理 FOR 子句部份，為 FOR 子句中的每一個連結變數產生一棵以其為名稱的 For Tree，並且加入唯一的 Level Number。L08 是處理 LET 子句部份，LET 子句的連結變數可能連結到一般的路徑，也有可能是連結到一份 FLOWR 子句，因此必須考慮到這兩種情形，但是 SQLFrag 形成的觀念是一樣的，將路徑或 FLOWR 子句傳回的內容形成一份新的 derived relation，此步驟會在 LetSQLFrag 函式中產生，L09 是當連結到一般路徑時，也產生一棵以連結變數為名稱的 For Tree，L13 是連結到一份 FLOWR 子句時，建立 For Tree，其中 For

Tree 根節點的 SQLFrag 會產生一份 derived relation，而 derived relation 的內容必須經 SQL Constructor 步驟後才能得到，因此傳入的 FLOWR 只是暫存變數，意在說明 derived relation 的內容需要從後面過程才能得到。接下來將 FLOWR 子句再傳入 FRS 函式中做遞迴的處理，並且 Level Number 必須增加一個 Level Component 後傳入。

Algorithm FRS(*E, m, jm, ln, f*)

輸入 FLOWR 敘述句 *E*，對應表格 *m*，連結對應表格 *jm*，Level Number ln，FRS 集合 *f*

輸出 FRS 集合 *f*

```
L01  sqlfrag = {}; nest_ln = ""; orderby_tmp = {};
L02
L03  for(each "FOR $fv IN E") {    //processing FOR
L04     sqlfrag = ForSQLFrag(fv, E, m, jm);
L05     create a For Tree and root node that includes $fv, sqlfrag and ln;
L06     add ln to f.ls, then LastComponent(ln)++;
L07  }
L08  for(each "LET $lv := E") {    //processing LET
L09     if(E != FLOWR) {
L10        sqlfrag = LetSQLFrag(lv, E, m, jm);
L11        create a For Tree and root node that includes $lv, sqlfrag and ln;
L12     }
L13     else {
L14        nest_ln = ln + ".1";
L15        sqlfrag = LetSQLFrag(lv, FLOWR, m, jm);
L16        create a For Tree and root node that includes $lv, sqlfrag and ln;
L17        f = FRS(FLOWR, m, jm, nest_ln, f);
L18     }
L19     add ln to f.ls, then LastComponent(ln) ++;
L20  }
L21  for(each E in "WHERE φ( E )") {    //processing WHERE
L22     if(E is Selection Condition "Expr CompOp Literal") {
L23        sqlfrag = WhereSQLFrag(Expr CompOp Literal, m, jm);
L24        append child node that includes element(attribute) name of Expr, sqlfrag
                 and ln to corresponding For Tree;
L25     }
L26     if(E is Join Condition, "Expr1 CompOp Expr2") {
L27        sqlfrag = WhereSQLFrag(Expr1 CompOp Expr2, m, jm);
L28        get corresponding For Tree of Expr1 and Expr2;
L29        if(ftree1.ln and ftree2.ln are the same level) {
L30              append child node that includes element(attribute) name of Expr1,
                       sqlfrag and ln to corresponding For Tree1;
L31        }
L32        else if(ftree1.ln is parent of ftree2.ln) {
L33              append child node that includes element(attribute) name of Expr2,
```

```
                    sqlfrag and ln to corresponding For Tree2;
L34       }
L35       else {
L36           append child node that includes element(attribute) name of Expr1,
                    sqlfrag and ln to corresponding For Tree1;
L37       }
L38   }
L39   for(each E in "ORDER BY φ( E )") {      //processing ORDERY BY
L40     if(E != FLOWR) {
L41       sqlfrag = OrderbySQLFrag(Expr, m, jm);
L42       record sqlfrag and Expr to orderby_tmp;
L43     }
L44   }
L45   if("RETURN") {    //processing RETURN
L46     add -ln to f.ls;
L47     ln = ln + ".1";
L48     for(each E in "RETURN {$lv($fv)/PathExpr}") {
L49       if(E != FLOWR) {
L50         sqlfrag = ReturnSQLFrag($lv($fv)/PathExpr, m, jm);
L51         create a Return Tree and root node that includes $lv($fv), sqlfrag and ln;
L52       }
L53       else {
L54         nest_ln = ln + ".1";
L55         sqlfrag = ReturnSQLFrag(FLOWR, m, jm);
L56         create a Return Tree and root node that includes $lv($fv), sqlfrag and ln;
L57         f = FRS(FLOWR, m, jm, nest_ln, f);
L58       }
L59       add ln to f.ls, then LastComponent(ln) ++;
L60     }
L61   }
L62   while(orderby_tmp is not empty) {
L63     append sqlfrag to corresponding child node of Return Tree;
L64     mark child node;
L65   }
L66   return f;
```

圖 4.1：演算法 FRS

L21～L38 是 WHERE 子句的建立，在此的做法是將限制式看成 Selection Condition 和 Join Condition 兩種型態。L22，Selection Condition 為單條路徑的限制，僅會對應到一棵 For Tree，會在 For Tree 中建立出子節點，接著將限制式對應的 SQLFrag 加入到此葉節點中；L26，Join Condition 則是牽涉到兩條路徑以上的限制，先將各別路徑在其對應的 For Tree 下建立好節點，節點名稱是路徑的最後一個元素（屬性）的名稱。接著判斷 For Tree 的階層關係，若是同階層，將

SQLFrag 加入到 Expr1 下的節點中，若有父子階層（或祖孫階層）關係，一律將對應的 SQLFrag 加入至低階層的 For Tree 中。

L39 是 ORDER BY 子句的處理，先將 ORDERY BY 的路徑對應的 SQLFrag 產生出來，之後將 SQLFrag 和 PathExpr 配對記錄至 orderby_tmp 集合中，直到 Return Tree 建立完成後，L62 再到 Return Tree 中找是否有一樣的傳回路徑，因為必須要先有傳回路徑，才可以指定依照那些傳回路徑做排序，當在對應的 Return Tree 中找到一樣的傳回路徑，將排序路徑對應的 SQLFrag 加入至 Return Tree 葉節點中的 SQLFrag 中，並且 L64 標註此葉節點為虛線，代表此葉節點有 ORDER BY 語意。

L45 是遇到“return”關鍵字時，表示要處理 RETURN 子句，在 RETURN 子句中，可以用一般路徑表示傳回值，亦可以再包含 FLOWR 子句形成巢狀關係。先將 Level Number 加入新的 Level Component 表示階層後，L48 是處理一般路徑，對每個連結變數建立一棵以其為名稱的 Return Tree，接著建立子節點，將傳回路徑對應的 SQLFrag 加入此葉節點中；L53 遇到 FLOWR 子句時的做法和 LET 子句中的一樣，一樣是利用遞迴的做法處理 FLOWR 子句。

Algorithm ForSQLFrag(*fv, E, m, jm*)	
輸入	連結變數名稱 *fv*，敘述句 *E*，對應表格 *m*，連結對應表格 *jm*
輸出	SQLFrag 集合 *sqlfrag*
L01	*sqlfrag* = {}; *q* = {};
L02	
L03	**switch**(*E*) {
L04	**case** *PathExpr* **:**
L05	*q* = Path2SQL(*m, jm, PathExpr*);
L06	*sqlfrag = q*;
L07	**break**;
L08	**case** “DISTINCT-VALUES(“ *PathExpr* “)” **:**
L09	*q* = Path2SQL(*m, jm, PathExpr*);
L10	**if**(*q.SELECT* = “”)
L11	*q.SELECT* = “DISTINCT *”;
L12	**else**
L13	*q.SELECT* = “DISTINCT” + *q.SELECT*;
L14	*sqlfrag.FROM* = “(“ + *q* + “) as “ + *fv*;
L15	**break**;
L16	**case default :**
L17	**break**;

L18	}
L19	**return** *sqlfrag*;

圖 4.2：演算法 ForSQLFrag

FRS 演算法中，每個子句的皆有自己 SQLFrag 產生演算法，因為每個子句中語意方式皆不同，例如聚合函式不會在 FOR 子句中出現。對於每個子句的 SQLFrag 產生演算法如下，圖 4.2 是針對 FOR 子句裡的 SQLFrag 語意分析演算法。FOR 主要為資料連接的子句，所連結的路徑查詢句為一般路徑，L04 即是當連結的路徑查詢句為一般路徑時所做的 SQL 片斷處理。

而 L08 則是針對特殊狀況，當 FOR 子句裡包含 DISTINCT-VALUES 函式時，其意義為取出不具重覆的欄位，因此必須先將表格的欄位做 DISTINCT 限制，L10 行，如果對應的 SQL 片斷 SELECT 子句是空的，那就以"SELECT *"取代；L12 行，反之，就 DISTINCT 欄位名稱。取出沒有重覆的欄位後，再用這些欄位做成一份新的表格，成為 Derived Relation（L14），而 Derived Relation 的表格名稱則是用連接變數的名稱來命名。

Algorithm LetSQLFrag(*lv, E, m, jm*)	
輸入	連結變數名稱 *lv*，敘述句 *E*，對應表格 *m*，連結對應表格 *jm*
輸出	SQLFrag 集合 *sqlfrag*
L01	*sqlfrag* = {}; *q* = {};
L02	
L03	**switch**(*E*) {
L04	**case** *PathExpr* **:**
L05	*q* = Path2SQL(*m, jm, PathExpr*);
L06	**if**(q.SELECT = "")
L07	q.SELECT = "*";
L08	*sqlfrag.FROM* = "(" + *q* + ") as " + *lv*;
L09	**break**;
L10	**case** *FLOWR* **:**
L11	*sqlfrag.FROM* = "(" + *temp FLOWR* ") as " + *fv;*
L12	**break**;
L13	**case default :**
L14	**break**;
L15	}
L16	**return** *sqlfrag*;

圖 4.3：演算法 LetSQLFrag

圖 4.3 是針對 LET 子句裡的 SQLFrag 語意分析演算法。LET 子句是屬於比較特殊的子句，除了可連結一般路徑查詢句，亦可以連接一份 FLOWR 子句形成巢狀結構，而 LET 子句建出來的 For Tree 節點的 SQLFrag，不論是連結一般路徑查詢句或 FLOWR 子句，SQLFrag 都會有一份 derived relation，因為是將結果包裝起來成為一份新的資料集合，所以不論那一種都會形成新的表格，表格名稱用連接變數的名稱來命名。

不過當連接到 FLOWR 子句，形成巢狀結構，derived relation 的 SQL 必須是巢狀結構內的 FLOWR 所等價轉換出來的，因此建立完所有的 For Tree 和 Return Tree 之後，需要 Level Sequence 的協助，將 LET 連結變數對應的子層 FLOWR 子句 SQL 正確地對應成 derived relation 的內容。

Algorithm WhereSQLFrag(*E, m, jm*)	
輸入 敘述句 *E*，對應表格 *m*，連結對應表格 *jm* **輸出** SQLFrag 集合 *sqlfrag*	
L01	*sqlfrag* = {}; *q* = {}; *q'* = {};
L02	
L03	**switch**(*E*) {
L04	**case** *PathExpr* **:**
L05	*q* = Path2SQL(*m, jm, PathExpr*);
L06	*sqlfrag = q*;
L07	**break**;
L08	**case** *Expr1* (*ArithOp* | *CompOp*) *Literal* **:**
L09	*q* = WhereSQLFrag(*Expr1*);
L10	*sqlfrag.FROM = q.FROM*;
L11	*sqlfrag.WHERE = q.SELECT* + (*ArithOp* | *CompOp*) + *Literal*;
L12	**break**;
L13	**case** *Expr1* (*ArithOp* | *CompOp*) *Expr2* **:**
L14	*q* = WhereSQLFrag(*Expr1*);
L15	*q'* = WhereSQLFrag(*Expr2*);
L16	*sqlfrag.FROM* = combine(*q.FROM, q'.FROM*);
L17	*where_expr = q.SELECT* + (*ArithOp* | *CompOp*) + *q'.SELECT*;
L18	*sqlfrag.WHERE* = combine(*where_expr, q.WHERE, q'.WHERE)*;
L19	**break**;
L20	**case** "CONTAINS(" *PathExpr* "," *Literal* ")" **:**
L21	*q* = Path2SQL(*m, jm, PathExpr*);
L22	*sqlfrag.FROM = q.FROM*;
L23	*sqlfrag.WHERE = q.SELECT* + "LIKE " + "%" + *Literal* + "%";
L24	**break**;
L25	**case** *AggFun* "(" *PathExpr* ")" *CompOp Literal* **:**
L26	*q* = Path2SQL(*m, jm, PathExpr*);
L27	*sqlfrag.FROM = q.FROM*;

```
L28       sqlfrag.GROUPBY = "G1";
L29       sqlfrag.HAVING = AggFun + "(" + q.SELECT + ")" + CompOp Literal;
L30       break;
L31    case default :
L32       break;
L33 }
L34 return sqlfrag;
```

圖 4.4：演算法 WhereSQLFrag

圖 4.4 是針對 WHERE 子句裡的語意做 SQLFrag 產生的演算法。限制式分成兩個部份，Selection Condition 和 Join Condition，利用遞迴演算法，將限制式的完整 SQLFrag 產生出來。L04 對路徑查詢句產生對應的 SQLFrag；L08 是 Selection Condition 的等價 SQL 轉換，“+”運算子代表連接的意思，將字串連接起來；”combine”函式代表合併兩個 SQL 成為一份，重複表格和欄位的會去除。L13 則是利用遞迴方式將兩邊的敘述句分開產生出 SQLFrag 之後，再把兩邊的結果做合併的動作；L20 是對字串比對函式 CONTAINS 做處理，所以 CONTAINS 對應到 SQL 的“LIKE“字串比較運算子；L25 是聚合函式的限制式，必須要將轉換出來的限制式放入到 HAVING 子句中，可是產生 HAVING 之前，GROUP BY 子句中必須有分群依據欄位，因此 L28 先設一個虛擬的欄位”G1“，保持 SQL 的文法合理，而此欄位”G1“亦可幫助當合併完查詢樹的所有 SQL 片斷後，可快速地檢查出 SQL 中需要 GROUP BY 的語意。

Algorithm ReturnSQLFrag(*E, m, jm*)

輸入　敘述句 *E*，對應表格 *m*，連結對應表格 *jm*
輸出　SQLFrag 集合 *sqlfrag*

```
L01 sqlfrag = {}; q = {};
L02
L03 switch(E) {
L04    case PathExpr :
L05       q = Path2SQL(m, jm, PathExpr);
L06       if(q.SELECT = "")
L07          q.SELECT = "*";
L08       sqlfrag = q;
L09       break;
L10    case AggFun "(" PathExpr ")" :
L11       q = Path2SQL(m, jm, PathExpr);
L12       if(q.SELECT = "")
L13          if(AggFun = "Count")
```

L14	*q.SELECT = AggFun* + “(*)”;
L15	**else**
L16	output “XQuery Semantic Error!”;
L17	**else**
L18	*q.SELECT = AggFun + q.SELECT*;
L19	*sqlfrag = q*;
L20	**break**;
L21	**case default :**
L22	**break**;
L23	}
L24	**return** *sqlfrag*;

圖 4.5：演算法 ReturnSQLFrag

圖 4.5 是 RETURN 子句中傳回路徑的 SQLFrag 產生演算法。路徑經由 Path2SQL 演算法產生 SQL 片斷後，如果路徑對應的是表格，SELECT 中不會有任何資料，因此在 ReturnSQLFrag 演算法中，如果 SELECT 是空的，就將傳回預設為“*”，若傳回路徑有聚合函式所聚合時，L10 需要將聚合函式加入到 SQLFrag 中，在此會注意到，如果 *q.SELECT* 子句為空且聚合函式為“Count”，L14 才會加入聚合函式至 *q.SELECT* 中，因為只有“Count”可以作用在“*”上。圖 4.6，是 ORDER BY 子句中排序路徑的 SQLFrag 產生演算法，會產生 SQLFrag.ORDERY BY 的欄位。

當建立好 FRS 集合後，接下來即是將查詢樹的 SQLFrag 建構合併，在建構合併的過程中需要 Level Sequence 的資訊，從中可以得知道那些 For Tree 和 Return Tree 是屬於同一階層，知道階層關係進行合併時才不會出錯。在合併完之後還需要簡化以及合理化的步驟，因為在合併的過程中可能會有兩個或兩個表格以上的合併，此刻必須加入表格間的連結關係，以得到正確的資料，最後再確認聚合函式的分群依據欄位。

Algorithm OrderbySQLFrag(*E, m, jm*)	
輸入 敘述句 *E*，對應表格 *m*，連結對應表格 *jm* **輸出** SQLFrag 集合 *sqlfrag*	
L01	*sqlfrag* = {}; *q* = {};
L02	
L03	**switch**(*E*) {
L04	**case** *PathExpr* (“ascending” | “descending”)? **:**
L05	*q* = Path2SQL(*m, jm, PathExpr*);

```
L06         sqlfrag.ORDER BY = q.SELECT + (“ascending” | “descending”)?;
L07         break;
L08       case default :
L09         break;
L10   }
L11   return sqlfrag;
```

圖 4.6：演算法 OrderbySQLFrag

第五章　SQL 合併建構與 SQL 合理化

建好 FRS 後，得到 For Forest、Return Forest 和 Level Sequence，每一棵查詢樹中的節點都包含 SQLFrag，接下來必須將 SQLFrag 依照階層關係合併，並且做合理化的動作。本章中會先說明如何利用 Level Sequence 建構出每一棵 For Tree 和 Return Tree 的 SQLFrag，建構完成後會將 SQLFrag 合併成一份 SQL，最後再檢查合併完成的 SQL，是否存在分群（GROUP BY）的語意。

5.1 SQL Constructor

建立好 FRS 集合後，裡面的 For Forest、Return Forest 中，每一棵查詢樹的節點都有 SQLFrag，由於一棵查詢樹，可能由許多節點構成，要從 FLOWR 子句轉換成對應 SQL，必須將這些節點的 SQLFrag 收集起來，再合併成合理的 SQL 查詢句。為了有系統地收集每一棵樹所有節點的 SQLFrag，將一棵查詢樹的 SQLFrag 收集起來建構之後，會用 Level Number 去分辨這些建構好的 SQLFrag，接著根據 Level Sequence 的順序和階層關係合併 SQLFrag，最後檢查在 SELECT 子句中若有聚合函式的欄位和一般欄位傳回值，且 ORDER BY 子句中也有以一般欄位做為排序欄位，須增加分群（GROUP BY）的語意至 SQL 中；此外，亦有可沒有 ORDER BY 子句的出現，但仍然必須考慮分群的語意。

為了有系統地收集各查詢樹的 SQLFrag，提出了 SQLConstructor 演算法。根據 Level Sequence 的順序，依序將 Level Number 和對應的查詢樹的所有 SQLFrag 加入到堆疊（Stack）中，而在收集節點的 SQLFrag 時，有兩個規則：第一、同一棵樹中，節點的 SQLFrag 有重複的部份不加入；第二、Reuturn Tree 中，節點的 SQLFrag.SELECT 為“*”時，若葉點節的 SQLFrag.SELECT 中有表格欄位時，移除“*”，將表格欄位加入。

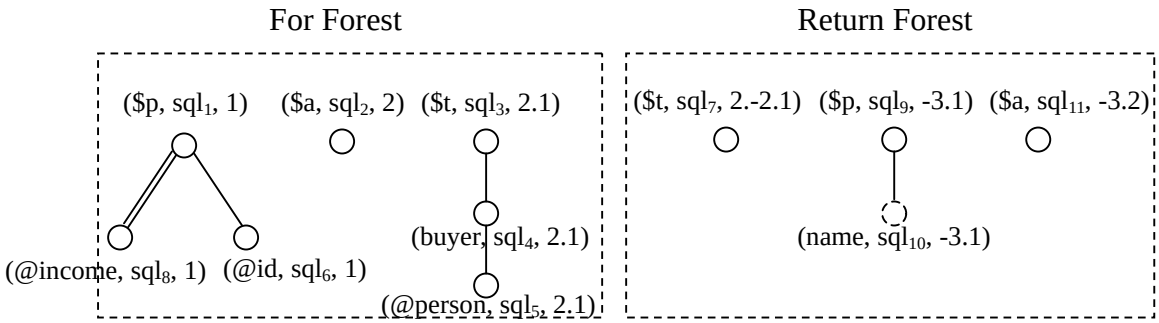
因為同一棵樹對應到同一個變數，而每個節點的 SQLFrag 是依照路徑產生，所以當收集所有 SQLFrag 時，可能會有重複的片斷，因此不需要將每個重複的

片斷視為不同重新命名，例如範例 5.1 中，$p 變數連結到的路徑“//person”，將會是根節點的路徑，而$p 的 For Tree 下有兩個葉節點，其 SQLFrag 產生的路徑為“//person//@income”和”//person/@id”，均從根節點的路徑下延伸而來，因此 SQLFrag 中，部份會有重複的片斷。

至於 SELECT 子句的部份，Return Tree 根節點的 SQLFrag.SELECT 是取出所有欄位，但葉節點的 SQLFrag.SELECT 更清楚地表示所要回傳的欄位，過濾掉更多不必要的欄位輸出，所以當有欄位要加入時，即可將“*”移除。

【範例 5.1】重複範例 2.1 和範例 4.1

```
FOR $p IN document(“auction.xml”)//person
  LET $a := FOR $t IN document(“auction.xml”)//closed_auction
            WHERE $t/buyer/@person = $p/@id
            RETURN $t
WHERE $p//@income > 5000 AND COUNT ($a) > 3
ORDERY BY $p/name
RETURN { <items>
           <item person=$p/name/text()> COUNT($a) </item>
         </items>}
```



sql_1 ：FROM Person p

sql_2 ：FROM (//SQL result of level 2.n) a GROUP BY ‘G1’ HAVING count(*) > 3

sql_3 ：FROM Closed_Auction t

sql_4 ：FROM Closed_Auction t

sql_5 ：FROM Closed_Auction t, Person p WHERE p.pid = t.buyer
sql_6 ：( )
sql_7 ：SELECT * FROM Closed_Auction t
sql_8 ：FROM Person p WHERE p.income > 5000
sql_9 ：SELECT * FROM Person p
sql_{10} ：SELECT p.name FROM Person p ORDER BY p.name
sql_{11} ：SELECT COUNT(*) FROM (//SQL result of level 2.n) a

Level Sequence：[1, 2[2.1, 2.-2[2.-2.1]], -3[-3.1, -3.2]]

範例 5.1 中，（@id，sql_6，1）節點中的 SQL 片斷內容為空，是因為此節點和（@person，sql_5，2.1）有 Join Condition 關係，根據演算法，會將 SQL 片斷置入 Level Number 階層較低的一方，因此（@id，sql_6，1）節點中的 SQL 片斷內容為空。而（）節點中的 SQL 片斷中，Derived Relation 的資訊必須從子階層得知，因此標註起來，根據 Level Sequence 的順序來取得 Derived Relation 內的資訊。

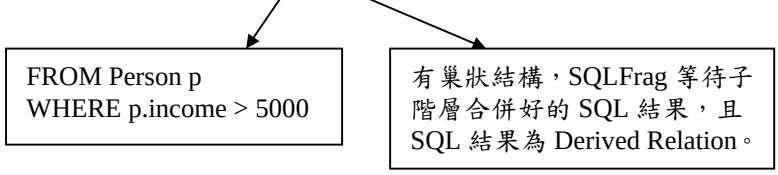
收集的過程是依照 Level Sequence 從左至右的順序做動作，在收集中，可能會遇到有子階層的 Level Number，當有子階層時，會有兩種情況需判斷：一、有子階層的 Level Number 為 LET 子句，即為 LET 連結變數後接一份新的 FLOWR 子句，該 Level Number 所對應的 SQLFrag 會是子階層合併好的 SQL 後成為一份 Derived Relation；二、Level Number 為“return”關鍵字的虛擬 Level Number，所對應的 SQLFrag 是子階層合併好的 SQL。

當然，在收集子階層的所有查詢樹的 SQLFrag，會做合併、合理化的動作，最後再把結果傳回，因此在設計演算法上時是使用遞迴的方式做處理，一邊收集一邊合併。在介紹合併的演算法之前，先合併範例 5.1 的 FRS 集合。

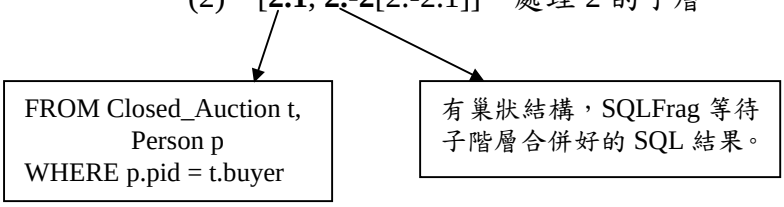
【範例 5.2】

合併範例 5.1 的 SQLFrag，根據 Level Sequence，先從第一層開始合併。

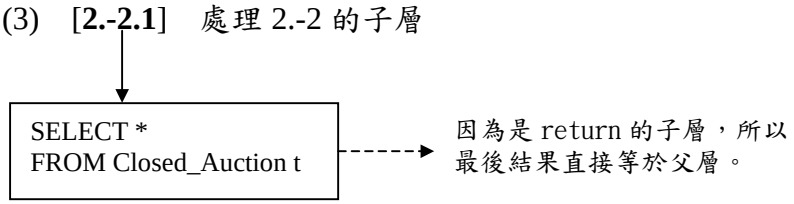
(1) [**1**, **2**[2.1, 2.-2[2.-2.1]], -3[-3.1, -3.2]]



(2) [**2.1**, **2.-2**[2.-2.1]] 處理 2 的子層

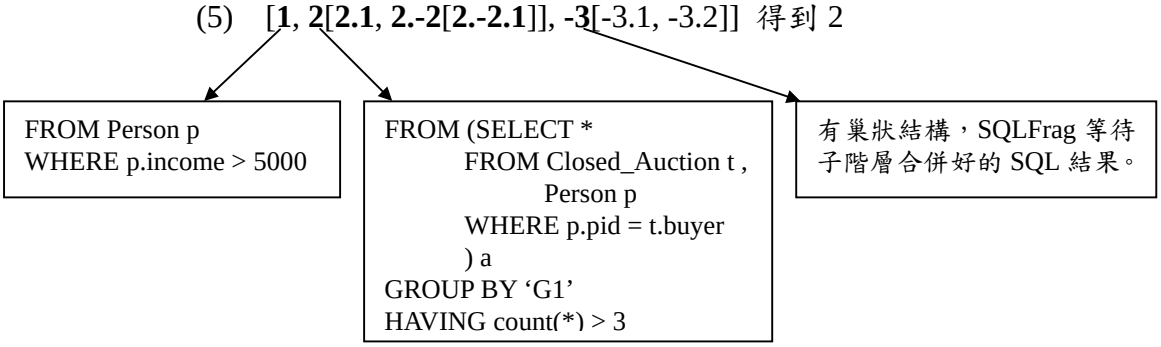


(3) [**2.-2.1**] 處理 2.-2 的子層

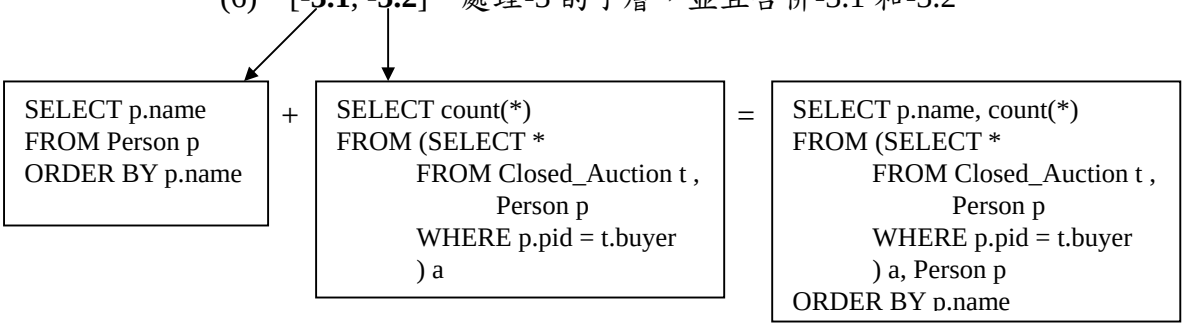


(4) [**2.1**, **2.-2**[**2.-2.1**]] 如此得到 2.-2，再將 2.1 和 2.-2 合併

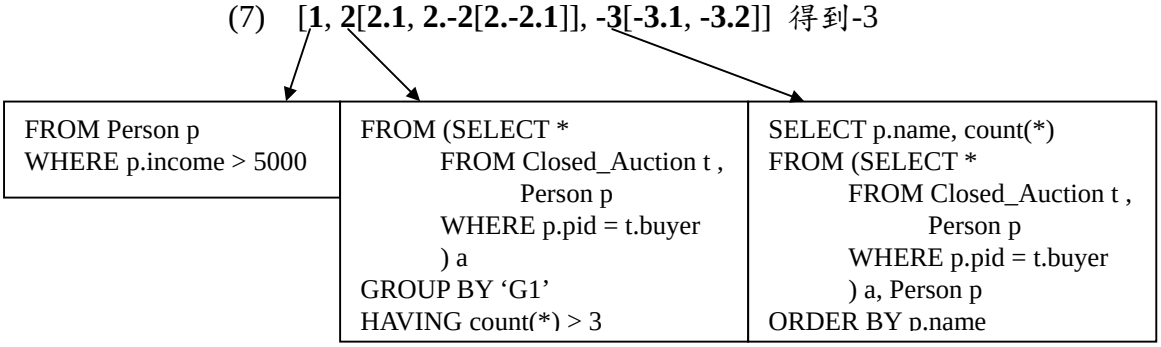
(5) [**1**, **2**[**2.1**, **2.-2**[**2.-2.1**]], **-3**[-3.1, -3.2]] 得到 2



(6) [**-3.1**, **-3.2**] 處理-3 的子層，並且合併-3.1 和-3.2



(7) [**1**, **2**[**2.1**, **2.-2**[**2.-2.1**]], **-3**[**-3.1**, **-3.2**]] 得到-3



(8) [**1**, **2**[**2.1**, **2.-2**[**2.-2.1**]], **-3**[**-3.1**, **-3.2**]] 合併 1, 2, -3，得到

```
SELECT p.name, count(*)
FROM (SELECT *
        FROM Closed_Auction t ,
             Person p
        WHERE p.pid = t.buyer
        ) a, Person p
WHERE p.income > 5000
GROUP BY 'G1'
HAVING count(*) > 3
ORDER BY p.name
```

Algorithm SQLConstructor(*f, ls, m, jm*)

輸入 FRS 集合 *f*，Level Sequence *ls*，對應表格 *m*，連結對應表格 *jm*

輸出 Primitive SQL *sql*

```
L01  Stack sf;
L02  sql = {}; tq = {};
L03
L04  for(each level number ln in ls) {
L05     if(LastComponent(ln) > 0)
L06        for(each node n in the tree where f.ForForest.ln = ln)
L07           tq = combine(tq, n.SQLFrag);
L08     if(ln has not sub-level)
L09        push(sf, (tq, ln));
L10     else {
L11        sub_ls = sub-level;
L12        tq' = SQLConstructor(f, sub_ls, m, jm);
L13        if(LastComponent(ln) > 0)    //LET Clause
L14           append tq' to tq.FROM, and let tq' be a derived relation;
L15        else                         //RETURN Clause
L16           tq = tq';
L17        push(sf, (tq, ln));
L18     }
L19  }
L20
L21  sql = pop(sf);
L22  while(sf != empty)
L23     sql = combine(sql , pop(sf));
L24
L25  return sql;
```

圖 5.1：演算法 SQLConstructor

範例 5.2 的 SQLFrag 合併流程是系統裡 SQL Constructor 的動作過程，圖 5.1 為 SQLConstructor 的演算法。L05 對每一個 Level Sequence 的 Level Number，從左至右有順序地開始處理，如果 Level Number 的最後一個 component 大於 0，表示此 Level Number 是由 FOR、LET 或 RETURN 子句產生的，而不是“RETURN”關鍵字，因為“RETURN” 關鍵字產生的 Level Number 並沒有對應的 For Tree 或 Return Tree，接著 L06 是收集某一棵查詢樹的所有節點的 SQLFrag，如第(1)中的 1。

L08 判斷是否有子層（巢狀結構），有子層的話，Level Number 後會接“[”此括號，並找到配對的“]”，其間的 Level Sequence 就為 Level Number 的子層，如第(1)中的 2，其後接的“[2.1, 2.-2[2.-2.1]]“，即為 2 的子層。L12 將子層的 Level Sequence 傳入 SQLConstructor 中先做子層的 SQLFrag 合併，L13 則是當子層的 SQLFrag 合併完成後，判斷父層是 LET 子句的巢狀結構或 RETURN 子句的巢狀結構，若為 LET 子句的話，那麼子層的合併的 SQL 結果會成為 Derived Relation 的內容，如第(5)中 2 的內容；若為 RETURN 子句，就直接傳回合併的 SQL 結果，如第(7)中的-3。因為各 Level Number 合併的 SQL 結果會置入堆疊中，所以最後必須要做堆疊中所有元素的合併動作，L21~L23 就是將堆疊中的所有元素做合併。

由範例 5.1 可知，雖然 SQL Constructor 產生了一份 SQL（稱之為 Primitive SQL），仔細檢查，此份 SQL 仍包含著錯誤：(1) Derived relation a 中的 Person p 表格和表格 Person p 之間並沒有連結關係；(2) SELECT 中有 p.name 欄位和聚合函式 count，卻沒有 GROUP BY 分群欄位的產生。因此此份 Primitive SQL 必須經過系統中的 SQL Validater，簡化以及合理化 Primitive SQL，如此才會是一份等價可執行的 SQL。

5.2 SQL Validater

當 XQuery 查詢句中沒有聚合路徑傳回值或語法沒有很複雜時，SQL

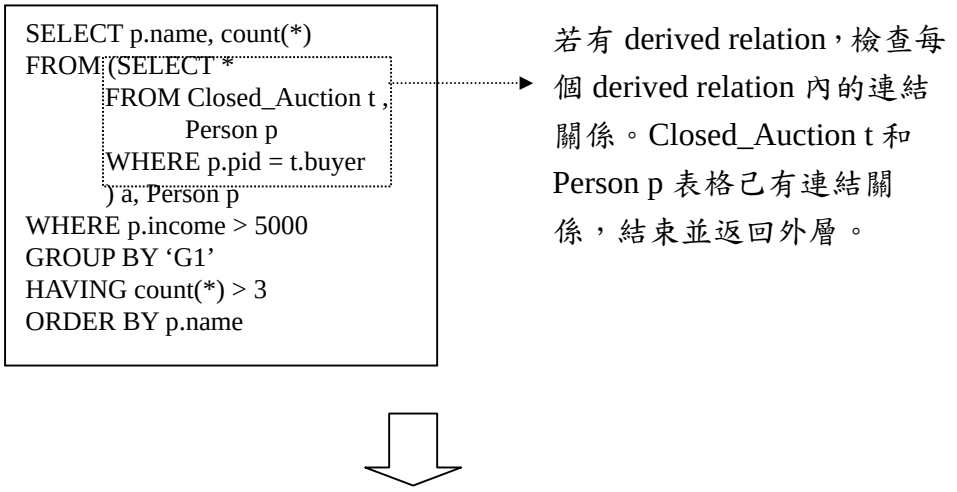
Constructor 產生的 Primitive SQL 即有可能就是等價可執行的完整 SQL，不過仍有可能是不完整的 SQL（如範例 5.1 的最後結果所示）。在此步驟中，會對 SQL Constructor 傳入的 Primitive SQL 做合理化的處理，所考慮的項目包括：

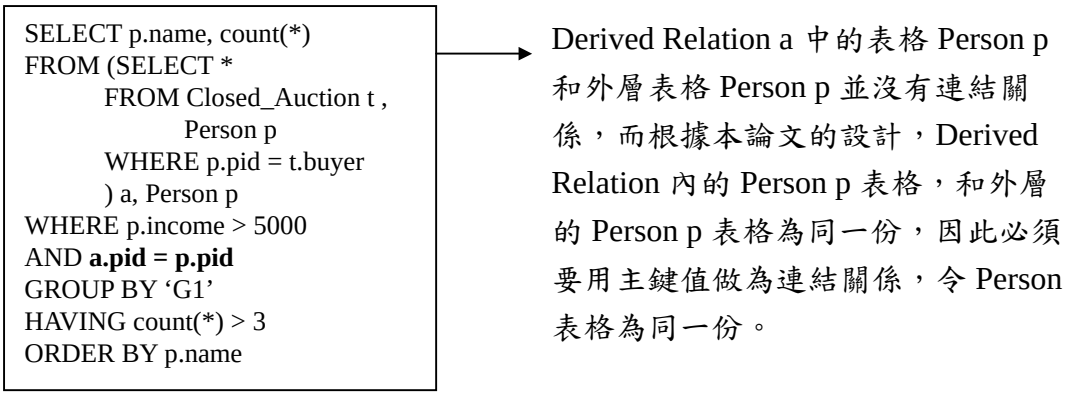
- 表格間的連結關係是否缺少
- 輸出資料有聚合函式時，檢查是否有分群依據欄位

5.2.1 表格間的連結關係

當一份 Primitive SQL 傳入後，從最外層開始，檢查 FROM 子句中的表格是否有 Derived Relation，若有，就將 Derived Relation 內 FROM 子中的表格和外層的 FROM 子句中的表格做比對，找出和外層名稱相同且 rename 後的名稱也相同的表格，並用表格的主鍵值做連結關係。因此範例 5.2 的 Primitive SQL 中，有一個 Derived Relation "*a*"，先檢查其內容，裡層 FROM 子句有兩個表格 Closed_Auction t 和 Person p，而外部 FROM 子句中有一個表格 Person p，而外部的 WHERE 中發現 Derived Relation a 中的表格 Person p 和外部的表格 Person p 並沒有連結關係，因此從 a 裡找到 Person p 表格，且用 Person p 表格的主鍵值 pid 欄位做連結，加入連結限制句至 WHERE 子句中。

【範例 5.2】





至於為何能從 Derived Relation 中找到和外層一樣的表格，理由是當 XQuery 查詢句具巢狀結構時，若內層的查詢句沒有引用到外層的資料來源（FOR、LET），必定在外層的 WHERE 子句中會做連接關係限制，因此當產生 Primitive SQL 時，就會有 Derived Relation 和表格的連結關係在 SQL.WHERE 子句中；若引用到外層的資料來源，如範例 2.1 的 LET $a 中的巢狀查詢句，根據 FRS 演算法，“$t/buyer/@person = $p/@id”此限制式中，會將 SQLFrag 加入到比較低階層，也就是變數$t 的 For Tree 中，可是此 SQLFrag 包含了變數$p 的表格來源（Person），當 SQL Constructor 合併所有查詢樹時，Derived Relation 內會有 Person 表格，且外層也會有 Person 表格，但若沒有連結關係，只會是兩份獨立的 Person 表格，因此需要用彼此的主鍵值做連結關係，以符合 XQuery 查詢句語意，因為產生 Person 表格的只有變數$p 的路徑。

圖 5.2，JoinCheck 演算法檢查有 Derived Relation 出現時，表格間是否缺少連結關係，並加入正確的限制式到 WHERE 子句中，在此考慮到 Derived Relation 內又可能包含 Derived Relation 的情況，所以有 Derived Relation 時，會先進入 Derived Relation 中檢查是否又有包含 Derived Relation，所以演算法會採用遞迴方式的處理。

起初設 FROM 子句中有 n 個表格，令兩個表格一組（ti, tj），i,j = 1~n 且 $i \neq j$，L02 判斷其中是否有一個表格為 Derived Relation 且另一個不是，滿足條件後，L03 再比對 Derived Relation 內是否有和外部一樣的表格，如果有且表格重新命名的名稱相同，就以表格的主鍵值做為連結關係，加入此限制式到 WHERE 子句

中。L04 則是利用遞迴處理 Derived Relation 內的 SQL 查詢句，因為可能 Derived Relation 內再包含 Derived Relation，之後再將處理好的 Derived Relation 代入原本的 Derived Relation 內容（L06）。

Algorithm JoinCheck(*q*, *jm*)	
輸入	Primitive SQL *q*，連結對應表格 *jm*
輸出	Join check primitive SQL *q*
L01	**for**(each table pair (t_i, t_j) in *q.FROM*) {
L02	**if**(t_i(t_j) is a derived relation, and another not) {
L03	**if**(find the same table in t_i(t_j) with t_j(t_i)) {
L04	append join condition using primary key to *q.WHERE*;
L05	*q'* = JoinCheck(t_i(t_j), *jm*);
L06	content of t_i(t_j) replaced by *q'*;
L07	}
L08	**else**
L09	**continue**;
L10	}
L11	**else**
L12	**continue**;
L13	}
L14	
L15	**return** *q*;

圖 5.2：演算法 JoinCheck

5.2.2 分群欄位

本論文所處理的 XQuery 範圍可允許 WHERE 和 RETURN 子句中出現聚合函式，而在 SQL 查詢語言中，GROUP BY 子句伴隨著聚合函式出現，但在 XQuery 中並沒有分群相關的子句定義，因為 XQuery 中的路徑表示式允許集合，加入結構包含關係，因此可以依照使用者需求聚集結果在某一個元素之下，形成分群的功能。

當 Primitive SQL 經過連結關係的檢查後，接下來就是檢查是否有 GROUP BY 子句的產生。若有 GROUP BY 子句的產生，其內容會是一個虛擬欄位 'G1'，這是為了符合 HAVING 子句是伴隨 GROUP BY 子句出現的語意，因為要有分群依據，才有做聚合函式限制的可能。此刻會檢查 SELECT 子句中的輸出結果，若有聚合函式的欄位且保含一筆或一筆以上的一般欄位，則那些一般欄位會做為

分群依據，但並非每一個一般欄位都適合做為分群依據，因此會檢查是否有 ORDERY BY 子句，若有 ORDER BY 子句，且排序的欄位在 SELECT 子句亦有出現，就以 ORDER BY 子句內的排序欄位做為分群依據，增加至 GROUP BY 子句中；反之，若沒有 ORDER BY 子句，就將所有一般欄位視為分群依據，增加至 GROUP BY 子句中。

以上是 Primitive SQL 中已經有 GROUP BY 子句時，若沒有 GROUP BY 子句，且 SELECT 中有聚合函式的欄位且保含一筆或一筆以上的一般欄位，GROUP BY 子句中的分群欄位產生規則和上述一樣，先檢查 ORDER BY 子句內的排序欄位是否在 SELECT 有出現，有的話就以 ORDER BY 內的排序欄位做為分群依據；分之，將所有一般欄位視為分群依據，增加至 GROUP BY 子句中。

【範例 5.3】

```
SELECT p.name, count(*)
FROM (SELECT *
        FROM Closed_Auction t ,
                Person p
        WHERE p.pid = t.buyer
        ) a, Person p
WHERE p.income > 5000
AND a.pid = p.pid
GROUP BY 'G1'
HAVING count(*) > 3
ORDER BY p.name
```

→ 經過 JoinCheck 之後的 primitive SQL，有 GROUP BY 子句，表示需要做分群。

```
SELECT p.name, count(*)
FROM (SELECT *
        FROM Closed_Auction t ,
                Person p
        WHERE p.pid = t.buyer
        ) a, Person p
WHERE p.income > 5000
AND a.pid = p.pid
GROUP BY p.name
HAVING count(*) > 3
ORDER BY p.name
```

→ SELECT 子句中，除了聚合函式 count(*)外，還有一筆一般欄位做為傳回值，並且 ORDER BY 子句中的排序欄位出現在 SELECT 子句中，以此欄位做為 GROUP BY 子句的分群依據。

圖 5.3 GroupbyCheck 演算法處理分群欄位，輸入一份檢查過表格間連結關係的 primitive SQL，接著 L01，先判斷是否有 GROUP BY 子句，表示需要有 HAVING，有的話代表此 primitive SQL 需要做分群語意，檢查是否有 ORDER BY

子句，L03 是當 ORDER BY 的排序欄位在 SELECT 中也有時，就依排序欄做為分群欄位加入到 GROUP BY 中；若 ORDER BY 裡沒有，就將 SELECT 的所有一般欄位當成分群欄位加入到 GROUP BY 中。而 L07 則是沒有 ORDER BY 子句時，用 SELECT 的所有一般欄位當成分群欄位加入到 GROUP BY 中。

若 primitive SQL 中沒有 GROUP BY 子句，可是在 SELECT 中有聚合函式的欄位和一筆或一筆以上的一般普通欄位，仍然是需要 GROPU BY 語意的產生。L11 即是判斷此點，和上述一樣，檢查是否有 ORDER BY 子句，並且排序欄位是否出現在 SELECT 子句中，沒有 ORDER BY 子句，就用 SELECT 的一般欄位做為分群欄位。

	Algorithm GroupbyCheck(*q*)
	輸入 Join check primitive SQL *q* **輸出** Final SQL *q*
L01	**if**(*q.GROUP BY*) {
L02	**if**(*q.ORDER BY*)
L03	**if**(field(s) of *q.ORDER BY* are found in *q.SELECT*)
L04	append field(s) of *q.ORDER BY* to *q.GROUP BY*;
L05	**else**
L06	append normal field(s) of *q.SELECT* to *q.GROUP BY*;
L07	**else**
L08	append normal field(s) of *q.SELECT* to *q.GROUP BY*;
L09	}
L10	**else**
L11	**if**(*q.SELECT* has aggregation function and normal field(s) >= 1)
L12	**if**(*q.ORDER BY*)
L13	**if**(field(s) of *q.ORDER BY* are found in *q.SELECT*)
L14	append field(s) of *q.ORDER BY* to *q.GROUP BY*;
L15	**else**
L16	append normal field(s) of *q.SELECT* to *q.GROUP BY*;
L17	**else**
L18	append normal field(s) of *q.SELECT* to *q.GROUP BY*;
L19	**else**
L20	**nothing**;
L21	
L22	**return** *q*;

圖 5.3：演算法 GroupbyCheck

第六章　正確性與轉換效率評估、分析

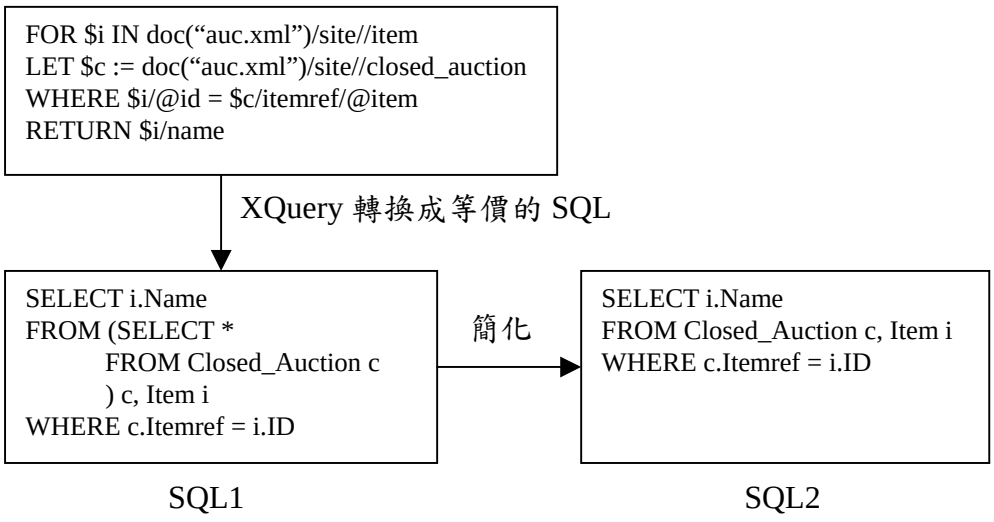
在本章節中，藉由 SQL 查詢句和 XQuery 查詢句的雙邊輸出結果，評估輸出的 SQL 查詢句是否與傳入的 XQuery 查詢句具有相同語意。接著探討資料定義對查詢句的影響，首先是一份 DTD 對應到不同的 RDB，同樣的 XQuery 查詢句對應到不同的 RDB，轉換效率是否會有影響；接著是階層式 DTD 和扁平式 DTD 對查詢句的影響，最後是 XQuery 查詢句的類型是否影響轉換效率。

6.1 正確性評估

根據本論文的建立方法，建立出來的 SQL 查詢句會保留部份 XQuery 查詢句結構，如 XQuery 的 LET 子句，若有巢狀結構，在 SQL 的 FROM 子句中會以 Derived Relation 呈現。且以每一個連結資料為主建立查詢樹，做資料限制、資料排序、資料輸出和資料分群時，易於辨識是那個資料來源，而當各查詢樹的部份結果合併時，亦會做最後的檢查，讓整個 SQL 查詢句合理化。

最後轉換出的 SQL 查詢句，在架構可以使其更趨於扁平，例如範例 6.1 中 LET 子句，轉換成 SQL1 時會以 Derived Relation c 表示，可是整個 Derived Relation 僅只是決定表格來源，並沒有要特定欄位的輸出或限制，因此對 SQL1 做簡化可以得到等價的 SQL2，但本論文所討論是轉換 XQuery 查詢語言成等價的 SQL 查詢語言，對於 SQL 查詢的最佳化處理於未來研究中討論。

【範例 6.1】



為了廣泛地驗證本論文的轉換機制所得之查詢語言正確性，蒐集了 W3C XQuery Use Case “R”、XMark Benchmark 和 Mondial 三種不同架構的 XML 文件，並為了便利實驗，適度修改了 DTD 架構。XQuery 查詢句則是使用其制訂的 Use Case，其中 W3C 和 XMark，去除掉不可處理範圍的查詢句後，再根據修改後 DTD 架構做合理的修改，而 Mondial 的查詢句則是由自己所設計（DTD、Schema 和 XQuery 的測試查詢句，請參考附錄）。

除了做正確性的評估外，用此三份不同架構的 XML 文件做實驗的原因如下：

- W3C ‘R’：由 W3C 制訂的 XQuery Use Case，主要探討 XML 資料儲存於關聯式資料內的使用，對此部份，提供 DTD、對應的 Relational Schema、XQuery 和查詢結果，使用此範例，假設本論文提出的轉換方法可轉換出等價的 SQL 查詢句，關聯式資料查詢出來的結果會和 W3C 所提供的 XQuery 查詢結果一致。
- XMark：使用 XMark 可探討同一份 DTD 對應到三種不同的 Schema 時的查詢句轉換效率，第一種是從 XMark 提出的資料定義 ER Model 中建立的表格，並做過 BCNF（Boyce-Codd Normal Form）處理；第二種是使用 [SKZ+02]提出的 Hybrid Inlining 所建立的對應 Schema；第三種則是本論文改良 Hybrid Inlining 後得到的對應 Schema。
- Mondial：Mondial 則是制訂了兩種不同的 DTD（階層式、扁平式）對應到同一份 Schema，因此會有兩種 XQuery，一種是針對階層式 DTD、另一種則是扁平式 DTD，但是 XML 的輸出結果是一樣的，只是在 XQuery 的結構上和限制式上有稍微不同。比較兩種不同結構的 XML 文件，在轉換成 SQL 上是否有影響。

在實驗實際測試部份，為了驗證轉換出來的查詢句是等價的，採用比較真實資料的傳回結果是否一致。先將 XML 和關聯式資料各別儲存到資料庫中，下達 XQuery 和使用本論文方法轉換出來的等價 SQL，比較兩邊的輸出結果是否相符。實驗測試環境以個人電腦來進行，該個人電腦基礎軟、硬體以中央處理器為

Pentium III 500、記憶體為 512MB，作業系統是微軟視窗 2000 進階伺服器版，儲存 XML 的是 XML 原生式資料庫 X-Hive 6[1]、儲存關聯式資料是 Oracle 9i[2]，轉換系統是以 Visual Studio C++ .NET 開發。

接著在雙邊的輸出結果相等比較上，以人工方式判斷，將 XQuery 輸出的結果包裝成符合 SQL 資料列結構比較。其中 XQuery 的結果中，若為父元素包含子元素，會將巢狀展開，包裝成符合 SQL 資料列結構，如範例 6.2。

【範例 6.2】

W3C—XQuery Q2

```
for $i in doc("auction.xml")//item_tuple
let $b := doc("auction.xml")//bid_tuple
where contains($i/description, "Bicycle")
and $b/@itemno = $i/@itemno
order by $i/@itemno
return
    <itemtuple>
      <maxbid>
        { <itemno>$i/@itemno </itemno>}
        { $i/description }
        { <highbid>max($b/bid)</highbid> }
      </maxbid>
    </itemtuple>
```

⇩

```
<itemtuple>
  <maxbid>
    <itemno>1001</itemno>
    <description>Red Bicycle</description>
    <highbid>55.0</highbid>
  </maxbid>
  <maxbid>
    <itemno>1003</itemno>
    <description>Old Bicycle</description>
    <highbid>22.0</highbid>
  </maxbid>
```

⋮ 略

對應的 SQL

```
select i.itemno, i.description, max(b.bid)
from items i,
       (select *
        from bids) b
where i.description LIKE '%Bicycle%'
and i.itemno = b.itemno
group by i.itemno, i.description
order by i.itemno
```

⇩

ItemNO	Description	highbid
1001	Red Bicycle	55.0
1003	Old Bicycle	22.0

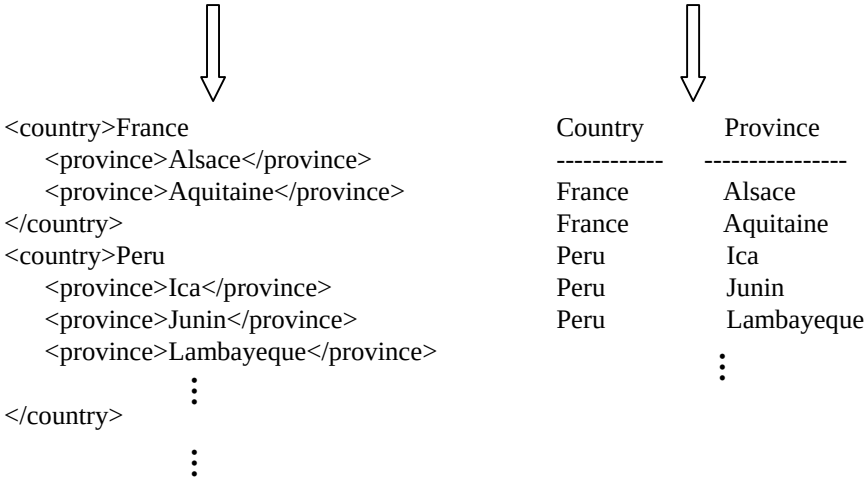
⋮ 略

Mondial—XQuery Q7

```
for $c in doc("mondial.xml")//country
let $pro := $c/province
return <country> {$c/name}
            <province> {$pro/name}
                  </province>
       </country>
```

對應的 SQL

```
select c.Name, pro.Name
from Country c,
      (select pro_2.*, pro_1.Code
       from Country pro_1, Province pro_2
       where pro_1.Code = pro_2.Country ) pro
where c.Code = pro.Code
```



範例 6.2 的 W3C-XQuery Q2 中，可以很直觀地將輸出結果包裝成 SQL 的資料列結構，每一條路徑對應到一筆欄位；Mondial-XQuery Q7 中，輸出結果為父元素包含一筆以上的子元素，在 SQL 中無法以此結構方式輸出資料，因此會將父元素內容和所有子元素配對，以符合 SQL 的輸出結構。

每一份 XQuery 會先在 X-Hive 中求出結果，接著將 XQuery 轉換成的 SQL，在 Oracle 中執行，比較兩份的輸出結果是否相符。X-Hive 系統可支援本論文所提出的 XQuery 範圍，且提供了聚合函式可供使用。在查詢句的轉換過程中，查詢樹的 SQL 片斷會先經過 SQL Constructor 合併建構得到 Primitive SQL，會先將此份 Primitive SQL 丟入 Oracle 9i 執行，實驗是否已為等價可執行的 SQL，接著再測試經過 SQL Validater 的 SQL。

在實驗的查詢句類型方面，分為四種：

1. 扁平結構（F）
2. 扁平結構＋聚合函式（F+A）
3. 巢狀結構（N）
4. 巢狀結構＋聚合函式（N+A）

首先 W3C XQuery Use Case ‘R’的查詢句檢測兩個階段，第一個階段是 SQL Constructor 後的 Primitive SQL，檢查是否可以執行，且得到的結果和 XQuery 下的結果相同；第二個階段則是 SQL Validater 後的 SQL。表 6.1 為檢測的結果，（查

	N Q1	F+A Q2	F Q3	N+A Q4	N+A Q5	F+A Q6	F+A Q7	N+A Q8	N+A Q9	F+A Q10	F+A Q11	F Q12
SC	○	×	○	×	×	○	○	×	×	×	×	×
SV	○	○	○	○	○	○	○	○	○	○	○	○
SC：SQL Constructor　SV：SQL Validater　×：不可執行或答案錯誤　○：正確												

表 6.1：W3C XQuery Use Case 'R' 查詢句轉換正確性評估表

	F Q1	F+A Q2	N+A Q3	N Q4	N+A Q5	F Q6	F Q7
SC	○	○	×	×	×	○	○
SV	○	○	○	○	○	○	○
SC：SQL Constructor　SV：SQL Validater　×：不可執行或答案錯誤　○：正確							

表 6.2：XMark 查詢句轉換正確性評估表

詢句請見附錄 1)，從表中得知，經過 SQL Construcotr 後得到的 SQL 即為正確結果的僅有 Q1、Q3、Q6 和 Q7，而其它的仍需 SQL Validater 後才會是正確的結果，因為 W3C 的查詢句範例中大部份皆有使用聚合函式，所以轉換成 SQL 時就必須考到分群的語意，Q6 和 Q7 雖然輸出值有使用聚合函式，但是查詢句的意思是對輸出的全部值做聚合的動作，因此不需要考慮到要依那個欄位為分群依據。

表 6.2 為 XMark 查詢句的正確評估表（查詢句請見附錄 2)，XMark DTD 對應到三份不同的 RDB，M 和 JM 亦不相同，但三份查詢句的反應結果都一樣，因此合併成一表說明。大部份的查詢句在經過 SQL Constructor 後就為正確的結果，但 Q3、Q4 和 Q5 為巢狀結構的查詢句，Q3 和 Q5 還包含著聚合函式的輸出，其中 Q4 並沒有聚合函式的輸出，可是根據本論文的轉換法則，巢狀結構會成為 Derived Relation，並且會引用到外部的表格，因此需要增加連結關係，如果只經過 SQL Constructor 的 SQL 語句，可能會不可執行或答案錯誤，當加入主鍵值

	F Q1	N Q2	F Q3	N+A Q4	F+A Q5	F Q6	F Q7
SC	×	×	○	×	×	×	×
SR	○	○	○	○	○	○	○
SC：SQL Constructor　SR：SQL Validater　×：錯誤　○：正確							

表 6.3(a)：Mondial 查詢句轉換正確性評估表（階層式 DTD）

	F Q1	F Q2	F Q3	F+A Q4	F+A Q5	F Q6	F Q7
SC	×	○	○	×	×	○	○
SR	○	○	○	○	○	○	○
SC：SQL Constructor　SR：SQL Validater　×：錯誤　○：正確							

表 6.3(b)：Mondial 查詢句轉換正確性評估表（扁平式 DTD）

連結關係後，不必要的答案會過濾掉，得到正確的結果。

表 6.3(a)和表 6.3(b)為 Mondial 查詢句正確性評估（查詢句請見附錄 3），比較兩個表可得到，表 6.3(a)是階層式 DTD 的 XQuery 查詢句轉換成等價 SQL 正確性估評，而表 6.3(b)是扁平式 DTD，因為 DTD 的結構不同，會影響到 XQuery 查詢句的結構和語意。階層式 DTD 的查詢句在經過 SQL Constructor 後大多為不正確或不可執行的 SQL，主要是階層式 DTD 在下查詢句時，必須對每一個可重複元素做連結變數，因此會在 FOR 或 LET 子句中宣告，若宣告在 LET 中，轉換成 SQL 時會成為 Derived Relation，此刻就會考慮到表格間的連結關係，因此大部份都必須經過 SQL Validater 檢查；而扁平式 DTD 的結構和 RDB 結構相似，除非有聚合函式或巢狀結構語句，大部份在經過 SQL Constructor 後即可以得到正確的結果。

階層式 DTD 的 XQuery 中，因為 province 元素為 country 的子元素，且兩者

都為可重複元素，必須對每一個可重複元素做連結，而又為確保 province 元素為 country 元素的子元素此一關係，所以使用 LET 子句。其中 country 元素對應到 Country 表格、province 元素對應到 Province 表格，而 Province 有一外鍵值參考到 Country，當使用 LET 子句時，會產生 Derived Relation，Derived Relation 內的會有 Country 和 Province 表格，此刻為了確保 Country 表格唯一，必須用主鍵值做連結關係，因此需要經過 SQL Validater 合理化 SQL。扁平式 DTD 的 XQuery，因為結構和關聯式 Schema 相似，所以各部份建立好對應的 SQL 後，經過 SQL Constructor 後即可得到正確結果。

從以上三個 SQL 正確性評估結果可知，經由本論文提出的轉換方法所轉換出來的SQL查詢句中，大多數必須經過SQL Validater後才成為正確可執行的SQL查詢句，主要的原因為考慮到 XQuery 查詢句中的聚合函式和巢狀結構，從 Mondial 的正確性評估中更可得到，階層式 DTD 的 XQuery 查詢句轉換成 SQL 時大多數需要經過 SQL Validater。而需要在 SQL Validater 裡動作的 SQL 是否轉換效率會受到影響，本論文於下一節會對各查詢句的轉換時間做分析，從中了解影響轉換效率的因素，以及不同的資料定義（如一份 DTD 對應到三份 RDB）是否對查詢句轉換有影響。

6.2 不同資料定義對查詢句轉換影響

為了解轉換查詢句的過程中，影響轉換效率的因素，對每一個 XQuery 詳細地分析各階段的轉換時間，共分為三個階段：一、建立完成 FRS；二、經過 SQL Constructor；三、經過 SQL Validater。從這個三個階段觀察彼此間是否有影響，以及不同的資料定義（如 XMark 一份 DTD 對應到三份 RDB、Mondial 二份 DTD 對應到一份 RDB）是否對轉換效率有影響。

圖 6.1 是 W3C XQuery Use Case ‘R’查詢句的詳細轉換時間，W3C 的是一份 DTD 對應到一份 RDB，且 DTD 結構為扁平式，和 RDB 結構相似。時間顯示，

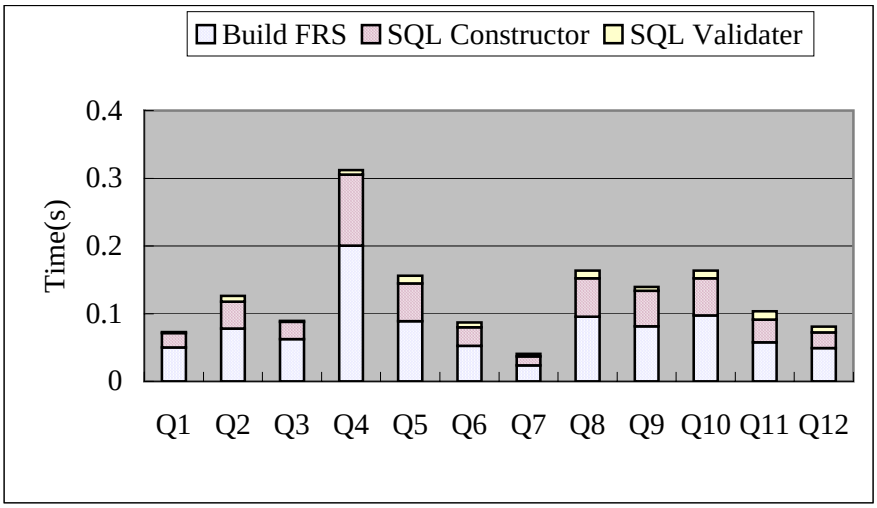


圖 6.1：W3C XQuery Use Case ‘R’查詢句轉換時間

每一份查詢句 FRS 的建立佔了一半以上的時間，表示建立查詢樹的時間是影響轉換時間比較大的因素，如 Q4，共建立了六棵 For Tree、四棵 Return Tree，且共有七條限制句，除了建立查詢樹的時間是影響轉換時間的因素外，限制句的多寡也是影響轉時間的因素之一。尤其是 Join Condition，尋找到對應的查詢樹，轉換成等價 SQL 後，還必須判斷查詢樹階層關係後加入節點至查詢樹中，此部份會花費較多時間，而且在 SQL Constructor 中動作時，較多的查詢樹，且查詢樹的節點愈多時，收集的時間自然會花費比較多。

另外 SQL Validater 部份，佔的時間比例很少，甚至有一些查詢句經過 SQL Validater 時沒有花費太多時間，主要是因為有些 SQL 查詢句在 SQL Constructor 後即成為正確可執行的 SQL 查詢句。且需要在 SQL Validater 檢查的 SQL 查詢句，主要是最後的檢查，加入欄位或限制句是加到 Primitive SQL 中，不需要再找對應的查詢樹、產生節點增加資訊、最後插入查詢樹，因此佔的時間較少。

圖 6.2、圖 6.3 和圖 6.4 是 XMark 查詢句轉換的時間，此部份是同一份 DTD 對應到不同份 RDB 的轉換時間比較。圖 6.2 是對應到 BCNF 的 RDB 的轉換時間，結果顯示 Q3、Q4 和 Q5 都花秒超過 0.1 秒的時間，和其它查詢句比較起來差異較大，主要是這三個 XQuery 查詢句包含巢狀結構，其中 Q4 具有雙

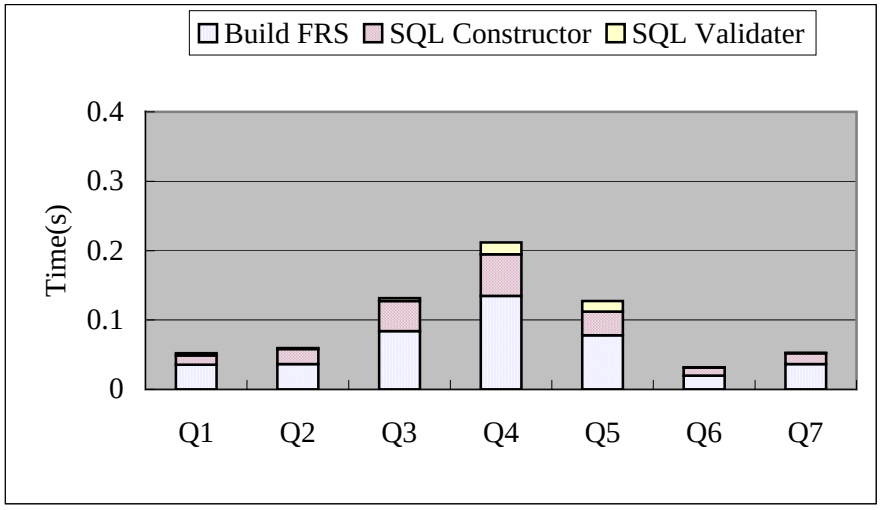


圖 6.2：XMark 查詢句轉換時間（BCNF）

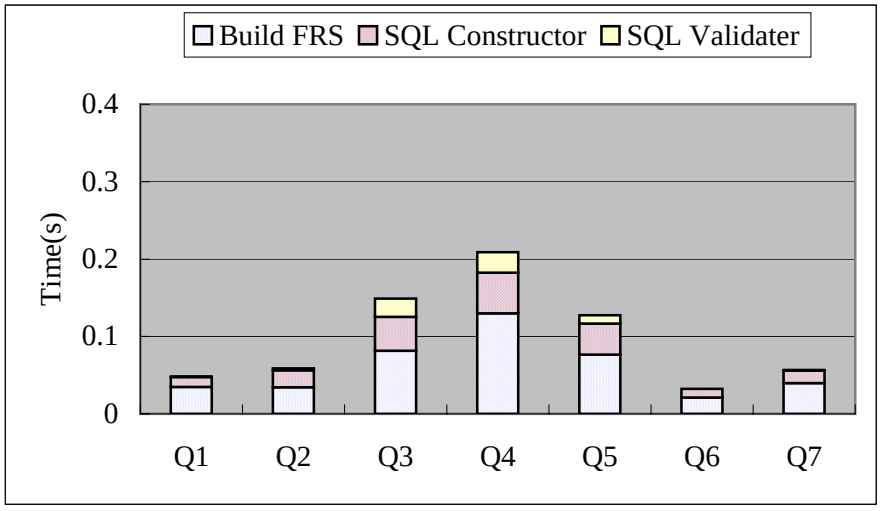


圖 6.3：XMark 查詢句轉換時間（改良式 Hybrid Inlining）

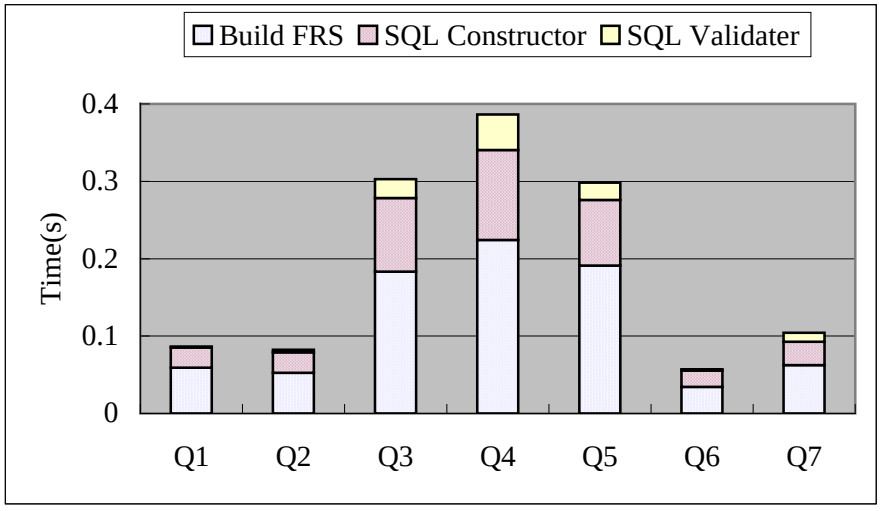


圖 6.4：XMark 查詢句轉換時間（Hybrid Inlining）

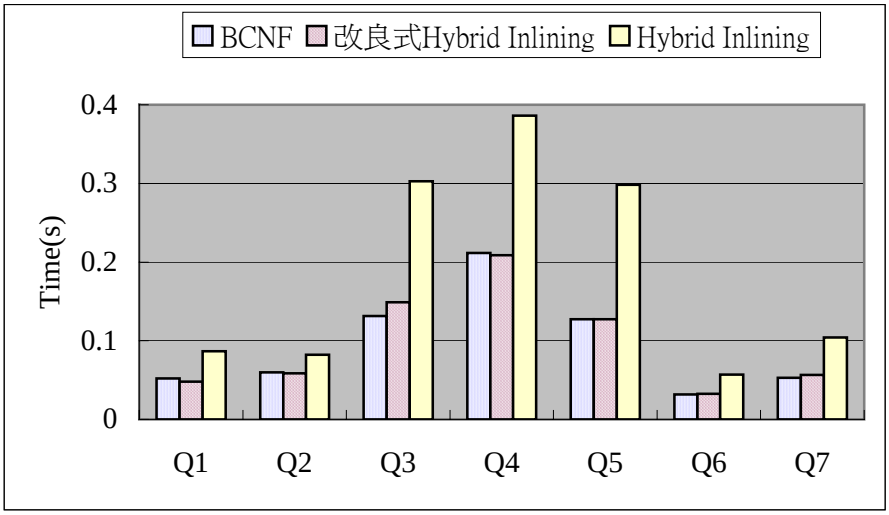


圖 6.5：XMark 對應三種 RDB 的轉換時間

層的巢狀結構，Q3、Q5 除了巢狀結構外，亦需要判斷聚合函式的語意。

圖 6.3 是對應到改良式 Hybrid Inlining 的轉換時間，和圖 6.2 比較，兩者的時間幾乎相同，表示改良式的 Hybrid Inlining 可以得到和 BCNF 一樣的結果。圖 6.4 則是對應到 Hybrid Inlining 的結果，從數據發現，時間比前兩者還高，甚至巢狀結構較複雜的查詢句，會差距到一倍以上的時間。主要原因 Hybrid Inlining 對於多餘的節點亦會建立表格，會造成較多的連結關係，當在產生路徑對應的等價 SQL 時，就必須從 JM（連結對應表格）中得到表格間的連結關係，之後加入至 SQL 的 WHERE 子句，因為一條路徑可能會有兩個或兩個以上的表格需要到 JM 中取得資訊建立連結關，因此在建立 FRS 時就會花費較多的時間，自然影響到 SQL Constructor。

而最後的 SQL Validater 部份則是因為表格太多，檢查的時候自然會花費較多時間，但是整體來看 SQL Validater 的影響並不大。圖 6.5 是將對應三種 RDB 的轉換時間做比較，Q3、Q4 和 Q5 的時間差異最大，主要是因為這三種查詢句是巢狀結構較複雜的查詢句，轉換成等價 SQL 時，在 Derivation Relation 內的表格會和外層重覆，因此會建立許多連結關係。

圖 6.6 和圖 6.7 是 Mondial 查詢句轉換時間，此部份針對不同的 DTD 對應到

同一份 RDB 的正確性和轉換效率時間比較。階層式的 DTD 在下查詢句時，可以省去 WHERE 子句中的連結關係限制句，但必須對每一個可重複元素做連結變數，因此會在 FOR 或 LET 子句中宣告，因為會宣告連結變數，代表建立 FRS 時會為其連結變數建立查詢樹，而查詢樹的建立則會花費較多的時間；扁平式的 DTD，除了每一個可重複元素做連結變數宣告外，還需要在 WHERE 子句中做連結關係，而連結關係也是影響查詢樹建立的重要因素之一。

從圖 6.8 階層式和扁平式 DTD 查詢句轉換時間比較表可知，其實兩者的時時相差不大，主要原因為：一、不論階層式或扁平的 DTD，在下查詢句時，都必須為可重複元素建立連結變數，而本論文的作法是根據連結變數建立查詢樹，因此查詢樹的數目相差不大；二、階層式 DTD 雖然可以省去 WHERE 子句中的連結限制句，但是對應到 Schema 時仍然需要有連結關係，因此會從 JM 中得到資訊，加入對應的 SQL 連結關係。

從範例 6.1 的 XQuery 中可發現，階層式 DTD 的 XQuery，country 和 province 都是可重複元素，雖然 province 是 country 的子元素，除了連結每個 country 元素，還必須連結每一個 province，因此會形成二棵 For Tree 和二棵 Return Tree。而 country 元素對應到 Country 表格、province 元素對應到 Province 表格，雖然在 XQuery 中不需要連結限制式，但轉換成 SQL 時必須對這兩個做連結關係，因此會從 JM 中找出連結欄位並且加入。

扁平式 DTD 的 XQuery，country 和 province 之間並無階層關係，取而代之是用 ID 和 IDREF 型態的屬性來表示元素之間的連結關係，因此會在 WHERE 子句中加入連結關係限制句，並且也是產生二棵 For Tree 和二棵 Return Tree。雖然轉換時間相同，可是階層式 DTD 的 XQuery 轉換成的等價 SQL 會比扁平式 DTD 轉換出來的複雜，縱使轉換出來的 SQL 皆等價，查詢的結果亦相同，但若考慮到 SQL 查詢句最佳化的問題，扁平式 DTD 轉換的結果會較階層式 DTD 的理想，至於轉換的 SQL Plan 選擇，是未來討論的課題之一。

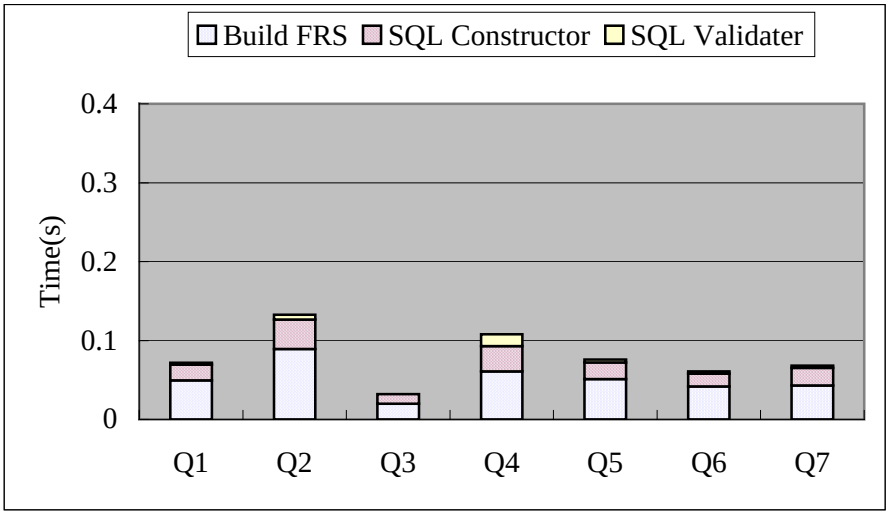


圖 6.6：階層式 Mondial 查詢句轉換時間（Hierarchical）

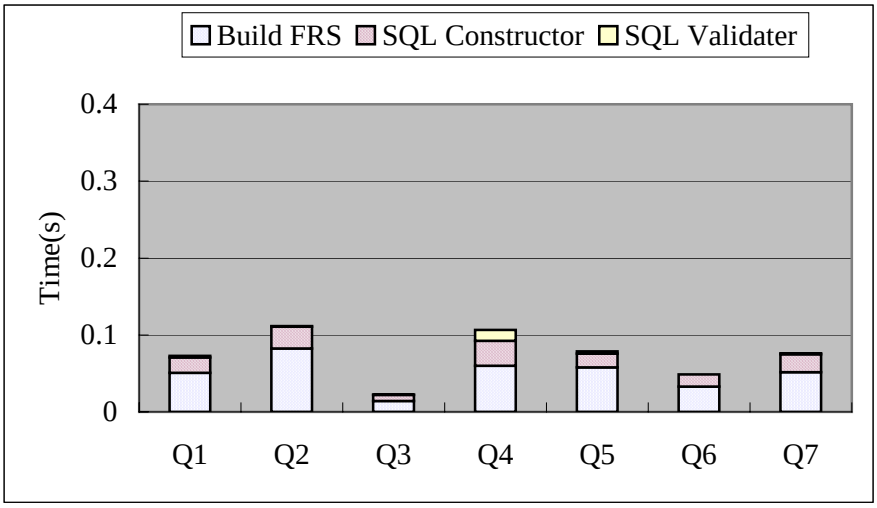


圖 6.7：扁平式 Mondial 查詢句轉換時間（Flat）

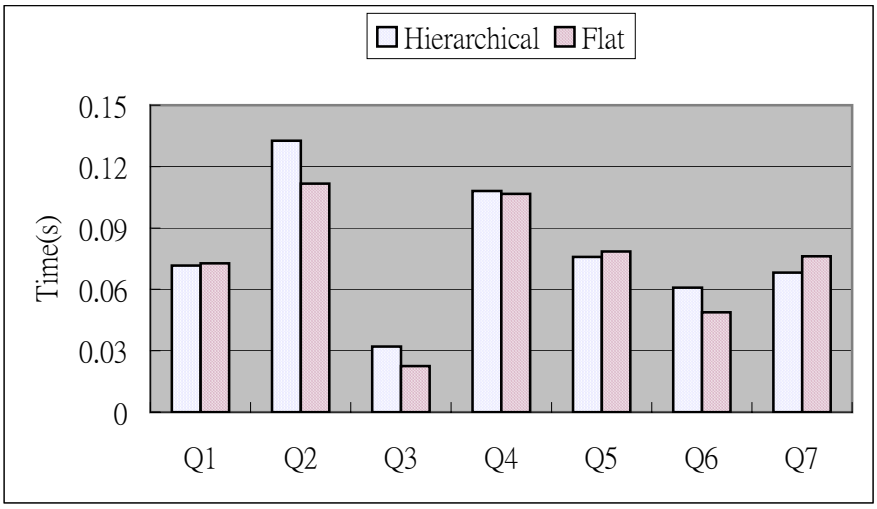


圖 6.8：扁平式和階層式 Mondial 查詢句轉換時間比較

從實驗的結果得到結論，將 XQuery 轉換成 SQL 時，影響查詢句轉換效率率中，RDB 的定義會比 XQuery 的撰寫方式重要，表格較少、Join 較少的 Schema 可以讓轉換達到更好的效率。且在選擇 DTD 上，階層式和扁平式的 DTD 運用本論文提出的方法，不但可得到正確可執行的 SQL 查詢句，轉換的效率也相差無幾，只是轉換出來的 SQL 查詢句，扁平式 DTD 會比階層式 DTD 更容易閱讀。

在上述的實驗結果中發現到一個有趣的現象，具有巢狀結構的 XQuery 查詢句，轉換的時間都會比其它沒有巢狀結構的 XQuery 查詢句來得耗時，為了了解巢狀結構的 XQuery 在轉換時是否會因為巢狀層數的增加，時間會呈現何種的增長，接下來比較不同結構的 XQuery 查詢句在轉換效率上的比較。

6.3 巢狀結構 XQuery 與扁平結構 XQuery 轉換效率比較

本論文的查詢句轉換重點之一即是解決具有巢狀結構的 XQuery，因為 XML 文件的特性之一就是具有巢狀結構，因此下達 XQuery 時，具有巢狀結構，可以清楚地表示出語意和 XML 中資料的結構關係。可是將巢狀結構 XQuery 轉換成 SQL 時，因為 XML 資料可具巢狀關係、XQuery 是路徑導向，由 12 種表示法組成；關聯式資料是扁平的、正規化的，SQL 是資料列導向，由 6 種子句組成。除了上述的差異性外，巢狀結構的 XQuery 還必須考慮到查詢句結構，解決後轉換成等價的 SQL 查詢句。但是若將巢狀結構的 XQuery 寫成同義的扁平結構的 XQuery，因為結構相似於關聯式資料庫查詢語言，所以在轉換上會較為直觀。

從圖 6.9 中，雖然扁平結構的 XQuery 要轉換成 SQL 時，可以很清楚直觀地做轉換，但是巢狀結構的 XQuery 表達查詢句的意思比扁平結構的清楚，可是考慮到轉換效率上，扁平結構的理論上會比巢狀結構的來得快，轉出來的 SQL 查詢句也會比巢狀結構的更容易閱讀，為此設計一實驗，比較巢狀結構 XQuery 和扁平結構 XQuery 的轉換效率。

巢狀結構

```
FOR $p IN document("auction.xml")//person
LET    $a := FOR $t IN document("auction.xml")//closed_auction
                WHERE $t/buyer/@person = $p/@id
                RETURN $t
WHERE    $p//@income > 5000
    AND    COUNT($a) > 3
RETURN <item person=$p/name/text()> COUNT ($a) </item>
```

扁平結構

```
FOR $p IN document("auction.xml")//person,
      $t IN document("auction.xml")//closed_auction
WHERE    $t/buyer/@person = $p/@id
   AND    $p//@income > 5000
   AND    COUNT($a) > 3
RETURN <item person=$p/name/text()> COUNT ($t) </item>
```

圖 6.9：範例 2.1 的巢狀結構 XQuery 查詢句轉為扁平結構 XQuery 查詢句

巢狀結構的部份為在 LET 子句後接了一份新的 FLOWR 子句，若是在 RETURN 子句加入一份新的 FLOWR 子句，根據 SQL Constructor 的演算法，雖然 Level Sequence 中的資訊呈現巢狀結構，可是在合併 SQL 片斷時，會將 RETURN 內的結果合併好之後，等於虛擬的 Level Number，而不是形成 Derived Relation，如此的撰寫方式，在本論文的處理中，會當成扁平結構處理，所以設計巢狀結構 XQuery 上，都以 LET 後接一份新的 FLOWR 子句為主。

圖 6.10 為巢狀結構 XQuery 的轉換時間，首先 Q1 為雙層結構，例如範例 6.2 的巢狀結構 XQuery，代表雙層、Q2 為三層結構、Q3 為四層結構，以此類推。圖 6.11 為扁平結構 XQuery 的轉換時間，每一個查詢句都和巢狀結構的同義，例如範例 6.2 中的扁平結構 XQuery，就和巢狀結構的同義，只是查詢句的撰寫結構上不同。

首先觀察圖 6.10，整體轉換時間隨著層數增加時間呈線性增加，其中 SQL Validater 的部份，並沒有因為層數的增加而有太大的起伏。圖 6.11 扁平結構 XQuery 的轉換時間，很明顯的，因為沒有巢狀結構，所以不會產生 Derived Relation，經過 SQL Validater 檢查時，不需要動作，因此幾乎無花費時間。

最後的圖 6.12 是巢狀結構 XQuery 和扁平結構 XQuery 的轉換時間比較，從結果可以看出扁平結構轉換成 SQL 的時間比巢狀結構的還快，主要是扁平結構

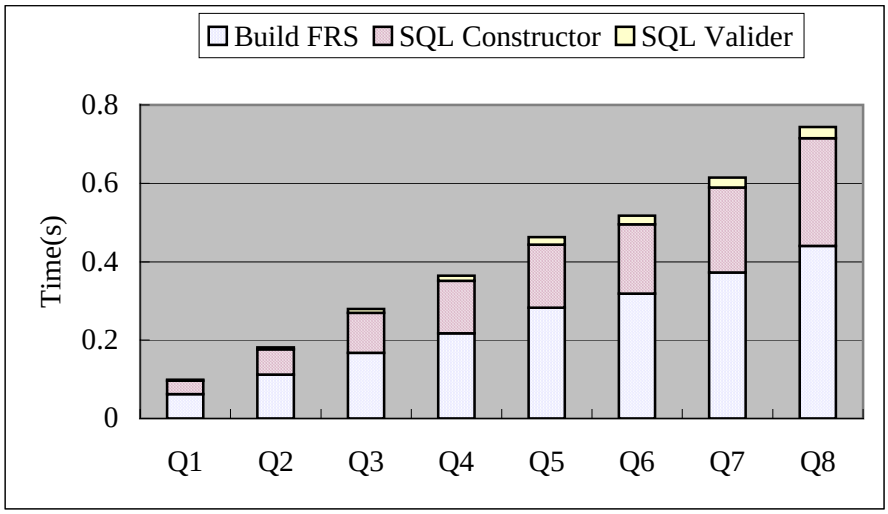


圖 6.10：巢狀結構 XQuery 轉換時間

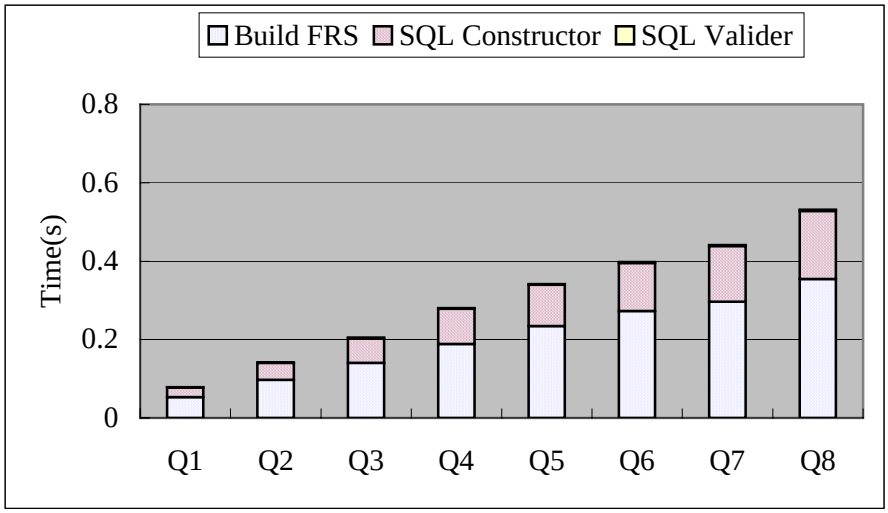


圖 6.11：扁平結構 XQuery 轉換時間

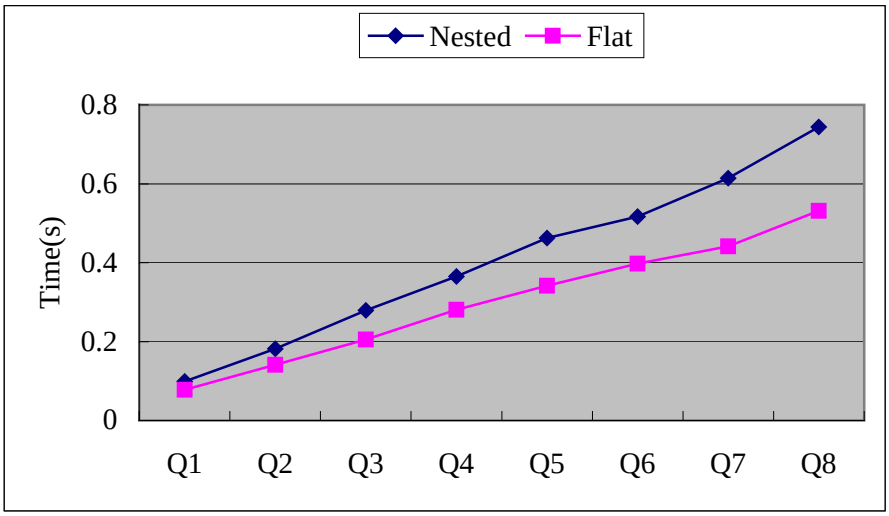


圖 6.12：巢狀結構 XQuery 與扁平結構 XQuery 轉換時間比較

XQuery 在 FRS 演算法時不需要進入遞迴程式中，並且在 SQL Validater 的部份不需要動作，因此整體時間扁平結構 XQuery 會較巢狀結構 XQuery 快。

比較兩者的時間，皆呈線性增長，彼此的差距則是隨著連結變數和 Join Condition 增加而有等比例的差距，平均每增加一層，會多出 0.03 秒的處理時間，至 Q8（九層）時，差距已有 0.2 秒。觀看雙邊時間，發覺在 SQL Constructor 上，巢狀結構會比扁平結構耗時，因為巢狀結構的 Level Sequence 會呈現許多 Sub Level Sequence，尋找 Sub Level Sequence 再用遞迴方式合併結果，會較線性掃描後合併耗時。

從全部的實驗可歸納出，使用本論文的方法轉換 XQuery 至等價 SQL，不論 DTD 是階層式或扁平式的，RDB 中 Schema 定義以及 XQuery 結構是影響轉換時間比較大的因素，其中使用正規化的 Schema 和使用扁平結構撰寫的 XQuery 會得到較好的轉換效率，並且得到的 SQL 較容易閱讀。雖然巢狀結構 XQuery 會花費較多時間，且轉換出來的 SQL 較複雜，但是查詢句的語意保持的較完整。

第七章　結論與未來研究

XQuery 與 SQL 分別是 XML 資料端與關聯式資料端，標準且統一的查詢語言，若能正確地轉換兩者，則可達成 XML 資料與關聯式資料的交換。本論文提出一套轉換系統來進行查詢語言的轉換，其中包含了整合 DTD 和 Schema 的對應表格和連結對應表格，對應表格記錄 DTD 路徑對應到關聯式資料庫的表格或欄位資訊，並記錄區間碼解決特殊路徑（如 AD 路徑、Wildcard）；連結對應表格則是當 DTD 路徑對應到的表格或欄位有參考到另一表格時，必須將主鍵值表格欄位和外鍵值表格欄位記錄下來。

以對應表格和連結對應表格為 XML 資料與關聯式資料對應關係的提供者，接著將 XQuery 轉換成本論文定義的 For Tree 和 Return Tree，使用 Level Number 區別每一棵查詢樹，並用 Level Sequence 記錄查詢樹之間的階層結構。當查詢樹建立完成後，根據 Level Sequence 的資訊，進行 SQL Constructor，將所有查詢樹的 SQL 片斷合併，再經過 SQL Valider，檢查表格間是否缺少連結關係、是否缺少分群（GROUP BY）的語意，最後合理化成一份等價的 SQL。之後設計實驗觀察轉換後的查詢句正確性、不同資料定義對查詢句的影響以及不同結構查詢句的轉換效率影響。

未來研究方面，規劃了二大方向，第一是解決更複雜的 DTD 轉換，如遞迴（Recursive）的 DTD，運用對應表格為整合基礎，考慮遞迴 DTD 和關聯式資料的差異性後，做對應表格的修改，進而可以做遞迴 DTD 的 XQuery 查詢句轉換。第二是解決 XQuery 更多的表示法和函式轉換，如條件表格法（Condition Expression）、量化表示法（Quantified Expression）和 Expressions on Sequence Types，對於這三種表示法，必須考慮更嚴謹、等價的轉換法則，除此之外，特殊函式和 XQuery 更豐富的語意的轉換也是未來研究課題，例如 EMPTY 函式、在 LET 子句中又出現聚合函式等，令系統能支援更多不同類型的 XQuery 查詢句轉換，且可處理更豐富的資料定義。

參考文獻

[AS02] Sihem Amer-Yahia, Divesh Srivastava. "A Mapping Schema and Interface for XML Stores", Fourth ACM CIKM Nternational Workshop on Web Information and Data Management (WIDM 2002), SAIC Headquaters, LcLean, Virginia, USA, Pages 23-30, November 8, 2002.

[BCF+02] Michael Benedikt, Chee Yong Chan, Wenfei Fan, Rajeev Rastogi,Shihui Zheng, Aoying Zhou. "DTD-Directed Publishing with Attribute Translation Grammars", Proceedings of the 28th VLDB Conference, page(s): 814-825, Hong Kong, China, 2002.

[CH02] Ya-Hui Chang and Tsan-Lung Hsieh, “A Document-based Approach to Indexing XML Data”, Proceedings of the 4th International Conference on Information Integration and Web-based Applications & Services, 2002.

[CJL+03] Zhimin Chen, H. V. Jagadish, Laks V. S. Lakshmanan, Stelios Paparizos. “From Tree Patterns to Generalized Tree Patterns: On Efficient Evaluation of XQuery”, In Proc. VLDB Conf., Berlin, Germany, Pages 237-248, Sep. 2003.

[CL03] Ya-Hui Chang and Cheng-Ta Lee, “Supporting Selection-Projection XQuery Processing Based on Encoding Paths”, Proceedings of DASFAA, 2003.

[CR03] Kajal T. Claypool and Elke A. Rundensteiner. "Sangam: A Transformation Modeling Framework", Eighth International Conference on Database Systems for Advanced Applications, Kyoto, Japan, March 26-28, 2003.

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

[FKS+02] Mary F. Fernandez, Yana Kadiyska, Dan Suciu, Atsuyuki Morishima, Wang Chiew Tan, “SilkRoute: A Framework For Publishing Relational Data in XML”, TODS 27(4): 438-493, 2002.

[GMD+02] Georges Gardarin, Antoine Mensch, Tuyet-Tram Dang-Ngoc, L. Smit. “Integrating Heterogeneous Data Sources with XML and XQuery", DEXA Workshops 2002: 839-846.

[HHL+02] Hui-I Hsiao, Joshua Hui, Ning Li, and Parag Tijare. "Integrated XML Document Management", Proc. VLDB Workshop on Efficiency and Effectiveness of XML Tools, and Techniques (EEXTT), Hong Kong, China, August 2002.

[JLS+04] H. V. Jagadish, Laks V. S. Lakshmanan, Monica Scannapieco, Divesh Srivastava, Nuwee Wiwatwattana. “Colorful XML: One Hierarchy Isn’t Enough”, ACM SIGMOD Conference, Paris, France, pages: 251-262, 2004.

[KCK+04] Rajasekar Krishnamurthy, Venkatesan T. Chakaravarthy, Raghav Kaushik, Jeffrey F. Naughton. “Recursive XML Schemas, Recursive XML Queries, and Relational Storage: XML-to-SQL Query Translation”, 20th International Conference on Data Engineering (ICDE 2004), 30 March – 2 April 2004, Boston, MA, USA.

[KKN+03] Rajasekar Krishnamurthy, Raghav Kaushik, Jeffrey F. Naughton. "XML-to-SQL Query Translation Literature: The State of the Art and Open Problems", XML Symposium (XSym) In Conjunction with VLDB , September 2003, Berlin, Germany.

[LS03] Laks V. S. Lakshmanan, and Fereidoon Sadri. "XML Interoperability", Sixth International Workshop on the Web and Databases (WebDB'2003).

[ME03] Jayant Madhavan and Alon Halevy. "Composing Mappings among Data Sources", the 29th International Conference on Very Large Databases (VLDB'2003), Berlin, Germany.

[ML02] Murali Mani, Dongwon Lee. "XML to Relational Conversion using Theory of Regular Tree Grammars", Proc. VLDB Workshop on Efficiency and Effectiveness of XML Tools, and Techniques (EEXTT), Hong Kong, China, August 2002.

[SKZ+99] Jayavel Shanmugasundaram, Kristin Tufte, Gang He, Chun Zhang, David DeWitt and Jeffrey Naughton."Relational Databases for Querying XML Documents: Limitations and Opportunities" Proc. of the 25th International Conference on Very Large Databases, Edinburgh, Scotland, Pages 302-314, September 1999.

[SYU99] T. Shimura, M. Yoshikawa and S. Uemura. "Storage and Retrieval of XML Documents Using Object-Relational Databases", in Proceedings of the 10th International Conference on Database and Expert Systems Applications(DEXA), Florence, Italy, Pages 206-217, August-September 1999.

[XE03] Li Xu and D.W. Embley. "Discovering Direct and Indirect Matches for Schema Elements", The 8th International Conference on Database Systems for Advanced Applications (DASFAA'03).

[YAS+01] M. Yoshikawa and T. Amagasa, T. Shimura and S. Uemura. "XRel: A Path-Based Approach to Storage and Retrieval of XML Documents using Relational Databases", in ACM Transactions on Internet Technology, Vol.

1, No. 1, Page 110-141, August 2001.

[YLL03] Xia Yang, Mong Li Lee, Tok Wang Ling. "Resolving Structural Conflicts in the Integration of XML Schemas: A Semantic Approach", in 22nd International Conference on Conceptual Modeling (ER), Chicago, Illinois, 2003.

[邱03] 邱豐傑, “轉換 SQL 為 XQuery 之研究與實作”, 碩士論文，國立台灣海洋大學資訊科學系, 2003。

[1] http://www.x-hive.com/

[2] http://www.oracle.com/

[3] http://www.w3c.org/

附錄 1：W3C XQuery Use Case ‘R’

```
<!ELEMENT auction (users, items, bids)>
<!ELEMENT users (user_tuple*)>
<!ELEMENT user_tuple (name, rating?)>
<!ELEMENT name (#PCDATA)>
<!ELEMENT rating (#PCDATA)>
<!ATTLIST user_tuple userid ID #REQUIRED>
<!ELEMENT items (item_tuple*)>
<!ELEMENT item_tuple (description, offered_by, start_date?, end_date?,
                      reserve_price? )>
<!ELEMENT description (#PCDATA)>
<!ELEMENT offered_by (#PCDATA)>
<!ELEMENT start_date (#PCDATA)>
<!ELEMENT end_date (#PCDATA)>
<!ELEMENT reserve_price (#PCDATA)>
<!ATTLIST item_tuple itemno ID #REQUIRED>
<!ELEMENT bids (bid_tuple*)>
<!ELEMENT bid_tuple (itemno, bid, bid_date)>
<!ATTLIST bid_tuple userid IDREF #REQUIRED
                    itemno IDREF #REQUIRED>
<!ELEMENT bid (#PCDATA)>
<!ELEMENT bid_date (#PCDATA)>
```

附錄 4.1：W3C XQuery Use Case ‘R’ DTD

Users 表格

UserID	Name	Rating

Items 表格

ItemNO	Description	Offered_By	Start_Date	End_Date	Reserve_Price

Bids 表格

UserID	ItemNO	Bid	Bid_Date

附錄 4.2：W3C XQuery Use Case ‘R’之關聯式資料架構

測試的 XQUery：

```
--Q1
for $i in doc("auction.xml")//item_tuple
where $i/start_date <= '1999-01-31'
   and $i/end_date >= '1999-01-31'
   and contains($i/description, "Bicycle")
order by $i/@itemno
return
      <item_tuple>
            { $i/@itemno }
            { $i/description }
      </item_tuple>
```

```
--Q2
for $i in doc("auction.xml")//item_tuple
let $b := doc("auction.xml")//bid_tuple
where contains($i/description, "Bicycle")
and $b/@itemno = $i/@itemno
order by $i/@itemno
return
     <itemtuple>
       <maxbid>
          { $i/@itemno } { $i/description }
          { max($b/bid) }
       </maxbid>
     </itemtuple>
```

```
--Q3
for $u in doc("auction.xml")//user_tuple,
      $i in doc("auction.xml")//item_tuple
where $u/rating > "C"
     and $i/reserve_price > 1000
     and $i/offered_by = $u/@userid
return
   <warning>
      <person>
       { $u/name } { $u/rating }
       { $i/description } { $i/reserve_price }
      <person>
   </warning>
```

```
--Q5
for $item in doc("items.xml")//item_tuple
let $b := doc("bids.xml")//bid_tuple
let $z := for $t in doc("bids.xml")//bid_tuple
          return { $t/@itemno }
                 <max_bid> { max($t/bid) }
                       </max_bid>
where $item/reserve_price * 2 < $z
and $b/@itemno = $item/@itemno
return
      <successful_item>
            { $item/itemno }
            { $item/description }
            { $item/reserve_price }
            <high_bid>{$z }</high_bid>
       </successful  item>
```

轉換出來的 SQL：

```
--Q1
select i.ItemNO, i.Description
from Items i
where i.Start_date <= '1999-01-31'
   and i.End_date >= '1999-01-31'
   and i.Description LIKE '%Bicycle%'
order by i.ItemNO
```

```
--Q2
select i.ItemNO, i.Description, max(b.Bid)
from items i,
        (select *
         from Bids) b
where i.Description LIKE '%Bicycle%'
   and i.ItemNO = b.ItemNO
group by i.ItemNO
order by i.ItemNO
```

```
--Q3
select u.Name, u.Rating, i.Description,
         i.Reserve_price
from Items i, Users u
where i.Reserve_price > 1000
   and i.Offered_by = u.UserID
   and u.Rating > 'C'
```

```
--Q5
select distinct item.ItemNO, item.Description,
         item.Reserve_Price, z.max_bid as high_bid
from Items item,
       (select * from Bids b) b,
       (select t.ItemNO , max(t.Bid) as max_bid
         from Bids t
         group by t.ItemNO) z
where item.Reserve_Price * 2 < z.max_bid
   and b.ItemNO = item.ItemNO
   and z.ItemNO = item.ItemNO
```

```
--Q4
for $seller in doc("users.xml")//user_tuple,
      $buyer in    doc("users.xml")//user_tuple,
      $item in    doc("items.xml")//item_tuple,
      $highbid in    doc("bids.xml")//bid_tuple
let $maxbid := for $t in doc("bids.xml")//bid_tuple
                  where $t/@itemno = $item/@itemno
                  return <max_bid>
                           <item_no> $t/@itemno </item_no>
                           <item_max>max($t/bid)</item_max>
                         </max_bid>
where $seller/name = "Tom Jones"
   and $seller/@userid    = $item/offered_by
   and contains($item/description , "Bicycle")
   and $item/@itemno    = $highbid/@itemno
   and $highbid/@userid    = $buyer/@userid
   and $highbid/bid = $maxbid/item_max
order by ($item/@itemno)
return
      <jones_bike>
            { $item/@itemno }
            { $item/description }
            <high_bid>{ $highbid/bid }</high_bid>
            <high_bidder>{ $buyer/name }</high_bidder>
```

```
--Q4
select item.ItemNO, item.Description,
         highbid.Bid as high_bid,
         buyer.Name as high_bidder
from Items item, Users seller, Users buyer,
      Bids highbid,
      (select t.ItemNO as item_no,
              max(t.Bid) as item_max
       from Bids t, Items item
       where t.ItemNO = item.ItemNO
       group by t.ItemNO) t2
where item.Offered_By = seller.UserID
   and seller.Name = 'Tom Jones'
   and buyer.UserID = highbid.UserID
   and highbid.ItemNO = item.ItemNO
   and item.Description LIKE '%Bicycle%'
   and highbid.Bid = t2.item_max
   and item.ItemNO = t2.item_no
order by item.ItemNO
```

```
--Q6
let $allbikes := doc("auction.xml")//item_tuple
let $bikebids := doc("auction.xml")//bid_tuple
where contains($allbikes/description, "Bicycle")
or contains($allbikes/description, "Tricycle")
and $bikebids/@itemno = $allbikes/@itemno
return
      <HighBid>
        {
          max($bikebids/bid)
        }
      </HighBid>
```

```
--Q6
select max(bikebids.bid) as HighBid
from (select *
        from items) allbikes,
      (select *
        from bids) bikebids
where (allbikes.description LIKE '%Bicycle%'
      or allbikes.description LIKE '%Tricycle%')
   and allbikes.itemno = bikebids.itemno
```

```
--Q7
let $item := doc("items.xml")//item_tuple
where $item/end_date >= '1999-03-01'
   and $item/end_date <= '1999-03-31'
return
      <item_count>
        {
          count($item)
        }
      </item_count>
```

```
--Q7
select count(*)
from (select * from items item) item
where item.end_date >= '1999-03-01'
   and item.end_date <= '1999-03-31'
```

```
--Q8
for $highbid in doc("bids.xml")//bid_tuple,
    $user in doc("users.xml")//user_tuple
let $maxbid := for $t in doc("bids.xml")//bid_tuple
                return { $t/@itemno }
                        <max_bid> { max($t/bid) }
                                </max_bid>
where $user/@userid = $highbid/@userid
  and $highbid/bid = $maxbid/max_bid
order by $highbid/@itemno
return
      <high_bid>
          { $highbid/@itemno }
          { $highbid/bid }
          <bidder>{ $user/name/text() }</bidder>
      </high_bid>
```

```
--Q8
select highbid.ItemNO, highbid.Bid,
         user1.Name as bidder
from (select t.ItemNO, max(t.Bid) as max_bid
        from Bids t
        group by t.ItemNo) maxbid,
       Bids highbid, Users user1
where highbid.Bid = maxbid.max_bid
  and highbid.UserID = user1.UserID
  and highbid.ItemNO = maxbid.ItemNo
order by highbid.ItemNO
```

```
--Q9
let $highbid := for $t in doc("items.xml")//bid_tuple
                return { $t/@itemno }
                 <max_bid> { max($t/bid) }
                                </max_bid>
return
      <result>
       {
          for $item in doc("items.xml")//item_tuple,
              $b in doc("bids.xml")//bid_tuple
          where $b/bid = $highbid
            and $b/@itemno = $item/@itemno
          return
                <expensive_item>
                  { $item/@itemno }
                  { $item/description }
                  <high_bid>{ $highbid }</high_bid>
                </expensive_item>
       }
      </result>
```

```
--Q9
select item.ItemNO, item.Description,
         highbid.max_bid as high_bid
from Items item,
        (select * from Bids b) b,
        (select t.ItemNO, max(t.Bid) as max_bid
          from Bids t
          group by t.ItemNO) highbid
where b.bid = highbid.max_bid
  and b.ItemNO = item.ItemNO
  and highbid.ItemNO = item.ItemNO
```

```
--Q10
for $uid in distinct-values(
              doc("auction.xml")//bid_tuple/@userid),
      $u in doc("auction.xml")//user_tuple
let $b := doc("auction.xml")//bid_tuple
where $u/@userid = $uid
and $b/@userid = $uid
order by $u/@userid
return
      <bidder>
        <avgbid>
          { $u/userid }
          { $u/name }
          { count($b) }
          { avg($b/bid) }
        </avgbid>
      </bidder>
```

```
--Q10
select u2.userid, u2.name, count(*), avg(b.bid)
from (select distinct userid from bids) u,
        (select * from bids) b,
          users u2
where u.userid = u2.userid
  and b.userid = u2.userid
group by u2.userid
order by u2.userid
```

```
--Q11
for $i in distinct-values(
            doc("auction.xml")//bid_tuple/@itemno)
let $b := doc("auction.xml")//bid_tuple
where count($b) >= 3
and $b/@itemno = $i
order by avg($b/bid) descending
return
    <PopularItem>
       <item>
          { $i }
          { avg($b/bid) }
       </item>
    </PopularItem>
```

```
--Q11
select i.itemno, count(*), avg(b.bid)
from (select distinct itemno
        from bids) i,
       (select *
        from bids) b
where b.itemno = i.itemno
group by i.itemno
having count(*) >= 3
order by avg(b.bid) desc
```

```
--Q12
for $u in doc("auction.xml")//user_tuple
let $b := doc("auction.xml")//bid_tuple
where count($b) > 1
and $b/@userid = $u/@userid
and $b/bid >= 100
return
     <BigSpender>
          <name> { $u/name } </name>
     </BigSpender>
```

```
--Q12
select u.name as name, count(*)
from users u,
       (select *
        from bids) b
where b.userid = u.userid
   and b.bid >= 100
group by u.name
having count(*) > 1
```

附錄 2：XMark

XMark 的 DTD 以和改良 Hybrid Inlining 的關聯式資料架構請參閱第二章的圖 2.1 和圖 2.2。

Person 表格

PID	Name	Income

Item 表格

ID	Name	Description	Seller	Datetime

Open_Auction 表格

OID	Bidder	Increase	Itemref	Datetime

Closed_Auction 表格

CID	Itemref	Price	Buyer

附錄 4.3(a)：XMark 之關聯式資料架構（BCNF）

Site 表格

ID

Items 表格

ID	ParentID

People 表格

ID	ParentID

Open_Auctions 表格

ID	ParentID

Closed_Auctions 表格

ID	ParentID

Item 表格

ID	ParentID	Name	Description

Person 表格

ID	ParentID	Name	Income

Open_Auction 表格

ID	ParentID	Seller	Itemref

Bidder 表格

ID	ParentID	Datetime	Personref	Increase

Closed_Auction 表格

ID	ParentID	Itemref	Seller	Buyer	Price

附錄 4.3(b)：XMark 之關聯式資料架構（Hybrid Inlining）

測試的 XQUery：

```
--Q1
for $b in doc("auction.xml")//person
where $b/@id = 'person0'
return $b/name/text()
```

```
--Q1（改良式 Hybrid Inlining）
select b.Name
from Person b
where b.PID = 'person0'
```

```
--Q2
let $a := for $i in doc("auction.xml")/ /closed_auction
          where $i/price/text() >= 40
          return $i/price
return COUNT($a)
```

```
--Q2（改良式 Hybrid Inlining）
select a.COUNT(*)
from (select i.Price from Closed_Auction i
      where i.Price >= 40 ) a
```

```
--Q3
for $p in doc("auction.xml")//person
let $a := for $t in doc("auction.xml")/ /closed_auction
          where $t/buyer/@person = $p/@id
          return $t
where   $p//@income > 5000
  and   COUNT($a) > 3
return <item person=$p/name/text()>
            COUNT ($a) </item>
```

```
--Q3（改良式 Hybrid Inlining）
select p.Name as person, count(*) as item
from Person p,
     (select t.*, p.PID
      from Person p, Closed_Auction t
      where t.Buyer = p.PID ) a
where p.Income > 5000
  and p.PID = a.PID
group by p.Name
having count(*) > 3
```

轉換出來的 SQL：

```
--Q1（BCNF）
select b.Name
from Person b
where b.PID = 'person0'
```

```
--Q1（Hybrid Inlining）
select b_3.Name
from Site b_1, People b_2, Person b_3
where b_1.SID = b_2.SID
  and b_2.People_ID = b_3.People_ID
  and b_3.Name = 'person0'
```

```
--Q2（BCNF）
select a.COUNT(*)
from (select i.Price from Closed_Auction i
      where i.Price >= 40 ) a
```

```
--Q2（Hybrid Inlining）
select a.COUNT(*)
from (select i_3.Price
      from Site i_1, Closed_Auctions i_2,
           Closed_Auction i_3
      where i_1.SID = i_2.SID
        and i_2.Closed_ID = i_3.Closed_ID
        and i_3.Price >= 40) a
```

```
--Q3（BCNF）
select p.Name as person, count(*) as item
from Person p,
     (select t.*, p.PID
      from Person p, Closed_Auction t
      where t.PID = p.PID ) a
where p.Income > 5000
  and p.PID = a.PID
group by p.Name
having count(*) > 3
```

```
--Q3（Hybrid Inlining）
select p_3.Name as person, count(*) as item
from Site p_1, People p_2, Person p_3,
     (select t_2.*, p_3.PID
      from Site p_1, People p_2, Person p_3,
           Closed_Auctions t_1, Closed_Auction t_2
      where p_1.SID = p_2.SID
        and p_2.People_ID = p_3.People_ID
        and p_1.SID = t_1.SID
        and t_1.Closed_ID = t_2.Closed_ID
        and p_3.PID = t_2.Buyer) a
where p_1.SID = p_2.SID
  and p_2.People_ID = p_3.People_ID
  and p_3.Income > 5000
  and p_3.PID = a.PID
group by p_3.Name
having count(*) > 3
```

```
--Q4
for $p in doc("auction.xml")//person
let $a := for $t in doc("auction.xml")/ /closed_auction
          let $n := for $t2 in doc("auction.xml")/ /item
                    where $t/itemref/@item = $t2/@id
                    return $t2
          where $p/@id = $t/buyer/@person
          return <item> $n/name/text() </item>
return <person name=$p/name/text()> $a </person>
```

```
--Q4（BCNF）
select p.Name , a.*
from Person p,
     (select n.name, p.PID
      from Person p, Closed_Auction t,
           (select t2.*, t.ID
            from Item t2, Closed_Auction t
            where t2.ID = t.ID ) n
      where t.PID = p.PID and t.ID = n.ID) a
where p.PID = a.PID
```

```
--Q4（改良式 Hybrid Inlining）
select p.Name , a.*
from Person p,
     (select n.name, p.PID
      from Person p, Closed_Auction t,
           (select t2.*, t.ID
            from Item t2, Closed_Auction t
            where t2.ID = t.Itemref ) n
      where t.Buyer = p.PID and t.CID = n.CID) a
where p.PID = a.PID
```

```
--Q4（Hybrid Inlining）
select p_3.Name, a.*
from Site p_1, People p_2, Person p_3,
     (select n.name, p_3.PID
      from Site p_1, People p_2, Person p_3,
           Closed_Auctions t_1, Closed_Auction t_2
           (select t2.*, t_2.CID
            from Site t2_1, Items t2_2, Item t2_3,
                 Closed_Auctions t_1, Closed_Auction t_2
            where t2_1.SID = t2_2.SID
              and t2_2.Items_ID = t2_3.Items_ID
              and t2_1.SID = t_1.SID
              and t_1.Closed_ID = t_2.Closed_ID
              and t2_3.ID = t_2.Itemref) n
      where p_1.SID = p_2.SID
       and p_2.People_ID = p_3.People_ID
       and p_1.SID = t_1.SID
       and t_1.Closed_ID = t_2.Closed_ID
       and t_2.Buyer = p_3.PID
       and t_2.CID = n.CID) a
where p_1.SID = p_2.SID
  and p_2.People_ID = p_3.People_ID
  and p_3.PID = a.PID
```

```
--Q5
for $p in doc("auction.xml")/ /person
let $l := for $i in doc("auction.xml")/ /open_auction
          where $p/profile/@income > (5000 * $i/initial/text())
          return $i
where $p/profile/@income > 50000
return <items>
          <name> $p/name/text() </name>
          <number> COUNT ($l) </number>
       </items>
```

```
--Q5（BCNF）
select p.Name as name, count(l.*) as number
from Person p,
     (select i.*, p.PID
     from Open_Auction i, Person p
     where p.Income > 5000*i.Initial ) l
where p.Income > 50000 and p.PID = l.PID
group by p.Name
```

```
--Q5（改良式 Hybrid Inlining）
select p.Name as name, count(l.*) as number
from Person p,
     (select i.*, p.PID
     from Open_Auction i, Person p
     where p.Income > 5000*i.Initial ) l
where p.Income > 50000 and p.PID = l.PID
group by p.Name
```

```
--Q5（Hybrid Inlining）
select p_3.Name as name, count(a.*) as number
from Site p_1, People p_2, Person p_3,
     (select i.*, p_3.PID
      from Site i_1, Open_Auctions i_2, Open_Auction i_3,
           People p_2, Prson p_3
      where i_1.SID = i_2.SID
       and i_2.Open_ID = i_3.Open_ID
       and i_1.SID = p_2.SID
       and p_2.People_ID = p_3.People_ID
       and p_3.Income > 5000*i_3.Initial) l
where p_1.SID = p_2.SID
  and p_2.Peope_ID = p_3.People_ID
  and p_3.Income > 50000
  and p_3.PID = l.PID
group by p_3.Name
```

```
--Q6
for $i in doc("auction.xml")/site//item
where CONTAINS ($i/description,"gold")
return $i/name/text()
```

```
--Q6（BCNF）
select i.Name
from Item i
where i.Description LIKE '%gold%'
```

```
--Q6（改良式 Hybrid Inlining）
select i.Name
from Item i
where i.Description LIKE '%gold%'
```

```
--Q6（Hybrid Inlining）
select i_3.Name
from Site i_1, Items i_2, Item i_3
where i_1.SID = i_2.SID
   and i_2.Items_ID = i_3.Items_ID
   and i_3.Description LIKE '%gold%'
```

```
--Q7
for $b in doc("auction.xml")/site//item
where CONTAINS($b/description,"bicycle")
return <item>
          <name> $b/name/text() </name>
          <description> $b/description/text()
                    </description>
       </item>
order by $b/name
```

```
--Q7（BCNF）
select b.Name as name,
         b.Description as description
from Item b
where b.Description LIKE '%bicycle%'
order by b.Name
```

```
--Q7（改良式 Hybrid Inlining）
select b.Name as name,
         b.Description as description
from Item b
where b.Description LIKE '%bicycle%'
order by b.Name
```

```
--Q7（Hybrid Inlining）
select b_3.Name as name,
         b_3.Description as description
from Site b_1, Items b_2, Item b_3
where b_1.SID = b_2.SID
   and b_2.Items_ID = b_3.Items_ID
   and b_3.Description LIKE '%bicycle%'
order by b_3.Name
```

附錄 3：Mondial

```
<!ELEMENT mondial (country*, organization*)>
<!ELEMENT country (name, population, religions*, province*, city*)>
<!ATTLIST country car_code ID #REQUIRED
                  captial IDREF #REQUIRED>
<!ELEMENT name (#PCDATA)>
<!ELEMENT population (#PCDATA)>
<!ATTLIST religions percentage CDATA #REQUIRED>
<!ELEMENT province (name, population)>
<!ATTLIST province id ID #REQUIRED
                  country IDREF #REQUIRED
                  captial IDREF #REQUIRED >
<!ELEMENT city (name, population)>
<!ATTLIST city id ID #REQUIRED
                  country IDREF #REQUIRED
                  province IDREF #REQUIRED >
<!ELEMENT organization (members*)>
<!ATTLIST organization id ID #REQUIRED
                        name CDATA #REQUIRED
                        abbrev CDATA #REQUIRED
                        established CDATA #REQUIRED >
<!ELEMENT members EMPTY>
<!ATTLIST member type CDATD #REQUIRED
                  country IDREF #REQUIRED>
```

附錄 4.4(a)：Mondial Hierarchical DTD

```
<!ELEMENT mondial (country*, province*, city*, organization*)>
<!ELEMENT country (religions*)>
<!ATTLIST country id ID #REQUIRED
                  population CDATA #REQUIRED
                  captial IDREF #REQUIRED
                  car_code CDATA #REQUIRED>
<!ATTLIST religions name CDATA #REQUIRED>
<!ELEMENT province EMPTY>
<!ATTLIST province id ID #REQUIRED
                  name CDATA #REQUIRED
```

```
                    country IDREF #REQUIRED
                    population CDATA #REQUIRED
                    captial IDREF #REQUIRED>
<!ELEMENT city (population)>
<!ELEMENT population (#PCDATA)>
<!ATTLIST city id ID #REQUIRED
                name CDATA #REQUIRED
                country IDREF #REQUIRED
                province IDREF #REQUIRED >
<!ELEMENT organization (members*)>
<!ATTLIST organization id ID #REQUIRED
                        name CDATA #REQUIRED
                        abbrev CDATA #REQUIRED
                        established CDATA #REQUIRED >
<!ELEMENT members EMPTY>
<!ATTLIST member type CDATD #REQUIRED
                    country IDREF #REQUIRED>
```

附錄 4.4(b)：Mondial Flat DTD

Country 表格

Code	Name	Capital	Province	Population

Organization 表格

Abbreviation	Name	Established

Is_member 表格

Country	**Organization**	Type

Religion 表格

Country	**Name**	Percentage

City 表格

Name	**Country**	**Province**	Population

Province 表格

Name	**Country**	Population	Capital

附錄 4.5：Mondial 之關聯式資料架構

測試的 XQUery：（階層式）

```
--Q1
for $c in doc("mondial.xml")//country,
      $o in doc("mondial.xml")//organization
let $m := $o/members
where $c/@car_code = "D"
   and $c/@car_code = $m/@country
return $o/@name


--Q2
for $c in doc("mondial.xml")//country
let $pro := $c/province
where $c/name = "Spain"
return <province> {$pro/name}
                  { for $cit in $c/city
                    where $cit/@province = @pro/@id
                    return <city> {$cit/name} </city> }
           </province>

--Q3
for $cit in doc("mondial.xml")//country/city
where contains($cit/name,"tre")
return $cit


--Q4
for $c in doc("mondial.xml")//country
let $a := for $o in doc("mondial.xml")//organization
           let $m := $o/members
           where $c/@car_code = $m/@country
           return $o
where count($o) > 60
return {<country> $c/name </country>}


--Q5
for $c in doc("mondial.xml")//country
let $r := $c/religions
where $r = "Roman Catholic"
   and $r/@percentag > 30
return count($c)


--Q6
for $c in doc("mondial.xml")//country
let $cit := $c/city
where $cit/population > 1000000
return
   <bigcountry> distinct-values($c/name)
   </bigcountry>
```

轉換出來的 SQL：

```
--Q1
select o.Name
from (select m_2.Country, m_1.Abbreviation
        from Organization m_1, Is_member m_2
        where m_1.Abbreviation = m_2.Organization ) m,
      Country c, Organization o
where c.Code = m.Country and c.Code = 'D'
   and m.Abbreviation = o.Abbreviation

--Q2
select pro.Name as province, cit_2.Name as city
from Country c,
        (select c_2.*, c_1.Code
          from Country c_1, Province c_2
          where c_1.Code = c_2.Country ) pro, City cit_2
where c.Name = 'Spain'
   and cit_2.Province = pro.Name
   and pro.Code = c.Code

--Q3
select cit_2.*
from Country cit_1, City cit_2
where cit_1.Code = cit_2.Country
   and cit_2.Name LIKE '%tre$'

--Q4
select c.Name as country
from Country c,
        (select o.*, c.Code
          from Organization o, Country c
                (select m_1.Abbreviation, m_2.*
                  from Organization m_1, Is_member m_2
                  where m_1.Abbreviation = m_2.Organization ) m,
          where c.Code = m.Country
             and o.Abbreviation = m.Abbreviation) a
where c.Code = a.Code
group by c.Name
having count(*) > 60

--Q5
select count(*)
from Country c,
       (select r_2.*, r_1.Code
         from Country r_1, Religion r_2
         where r_1.Code = r_2.Country ) r
where r.Code = c.Code
   and r.Name = 'Roman Catholic'
   and r.Percentage > 30

--Q6
select distinct c.Name as bigcountry
from Country c,
        (select cit_2.*, cit_1.Code
          from Country cit_1, City cit_2
          where cit_1.Code = cit_2.Country ) cit
where cit.Population > 1000000
   and cit.Code = c.Code
```

```
--Q7
for $c in doc("mondial.xml")//country,
let $pro := $c/province
return <country> {$c/name}
                <province> {$pro/name}
                        </province>
          </country>
```

```
--Q7
select c.Name as country, pro.Name as province
from Country c,
        (select pro_2.*, pro_1.Code
          from Country pro_1, Province pro_2
          where pro_1.Code = pro_2.Country ) pro
where c.Code = pro.Code
```

測試的 XQUery：(扁平式)

```
--Q1
for $c in doc("mondial-flat.xml")//country,
      $o in doc("mondial-flat.xml")//organization
let $m := $o/members
where $c/@car_code = "D"
   and $c/@car_code = $m/@country
return $o/@name
```

```
--Q2
for $c in doc("mondial-flat.xml")//country,
      $pro in doc("mondial-flat.xml")//province,
      $cit in doc("mondial-flat.xml")//city
where $c/name = "Spain"
   and $c/@id = $pro/@country
   and $pro/@id = $cit/@province
return <province> {$pro/@name}
                <city> {$cit/@name} </city>
          </province>
```

```
--Q3
for $cit in doc("mondial-flat.xml")//city
where contains($cit/@name,"tre")
return $cit
```

```
--Q4
for $c in do("mondial-flat.xml")//country
let $a := for $o doc("mondial-flat.xml")//organization
          let $m := $o/members
          where $c/@id = $m/@country
          return $o
where sum($o) > 60
return {<country> $c/@name </country>}
```

```
--Q5
for $c in doc("mondial-flat.xml")//country
let $r := $c/religions
where $r/@name = "Roman Catholic"
   and $r > 30
return count($c)
```

轉換出來的 SQL：

```
--Q1
select o.Name
from (select m_2.Country, m_1.Abbreviation
        from Organization m_1, Is_member m_2
        where m_1.Abbreviation = m_2.Organization ) m,
      Country c, Organization o
where c.Code = m.Country and c.Code = 'D'
   and m.Abbreviation = o.Abbreviation
```

```
--Q2
select pro.Name as province, cit.Name as city
from Province pro, Country c, City cit
where c.Name = 'Spain'
   and pro.Country = c.Code
   and cit.Province = pro.Name
```

```
--Q3
select *
from City cit
where cit.Name LIKE '%tre%'
```

```
--Q4
select c.Name as country
from Country c,
      (select o.*, c.Code
        from Organization o, Country c
              (select m_1.Abbreviation, m_2.*
               from Organization m_1, Is_member m_2
               where m_1.Abbreviation = m_2.Organization ) m,
        where c.Code = m.Country
          and o.Abbreviation = m.Abbreviation ) a
group by c.Name
having count(*) > 60
```

```
--Q5
select count(*)
from Country c,
      (select r_2.*, r_1.Code
        from Country r_1, Religion r_2
        where r_1.Code = r_2.Country ) r
where r.Code = c.Code
   and r.Name = 'Roman Catholic'
   and r.Percentage > 30
```

```
--Q6
for $c in doc("mondial-flat.xml")//country,
      $cit in doc("mondial-flat.xml")//city
where $c/@id = $cit/@country
   and $cit/population > 1000000
return <bigcountry>
   distinct-values($c/name) </bigcountry>
```

```
--Q6
select distinct c.Name
from City cit, Country c
where cit.Country = c.Code
   and cit.Population > 1000000
```

```
--Q7
for $c in doc("mondial-flat.xml")//country,
      $pro in doc("mondial-flat.xml")//province
where $c/@id = $pro/@country
return <country> {$c/@name}
            <province> {$pro/@name} </province>
         </country>
```

```
--Q7
select c.Name as country, pro.Name as province
from Province pro, Country c
where pro.Country = c.Code
```