2004-01-29

Semantic Caching for XML Queries

Li Chen

Worcester Polytechnic Institute

Follow this and additional works at: https://digitalcommons.wpi.edu/etd-dissertations

Repository Citation

This dissertation is brought to you for free and open access by Digital WPI. It has been accepted for inclusion in Doctoral Dissertations (All Dissertations, All Years) by an authorized administrator of Digital WPI. For more information, please contact wpi-etd@wpi.edu.
Semantic Caching for XML Queries

by

Li Chen

A Dissertation
Submitted to the Faculty
of the
WORCESTER POLYTECHNIC INSTITUTE
in Partial Fulfillment of the Requirements for the
Degree of Doctor of Philosophy
in
Computer Science
by

January 12, 2004

APPROVED:

Prof. Elke A. Rundensteiner
Advisor

Prof. Carolina Ruiz
Committee Member

Prof. Dan Dougherty
Committee Member

Dr. Ashok Malhotra
Microsoft Research
External Committee Member

Prof. Wang-Chien Lee
Penn State University
External Committee Member

Prof. Michael Gennert
Head of Department
Abstract

With the advent of XML, great challenges arise from the demand for efficiently retrieving information from remote XML sources across the Internet. The semantic caching technology can help to improve the efficiency of XML query processing in the Web environment. Different from the traditional tuple or page-based caching systems, semantic caching systems exploit the idea of reusing cached query results to answer new queries based on the query containment and rewriting techniques. Fundamental results on the containment of relational queries have been established. In the XML setting, the containment problem remains unexplored for comprehensive XML query languages such as XQuery, and little has been studied with respect to the cache management issue such as replacement. Hence, this dissertation addresses two issues fundamental to building an XQuery-based semantic caching system: XQuery containment and rewriting, and an effective replacement strategy.

We first define a restricted XQuery fragment for which the containment problem is tackled. For two given queries $Q_1$ and $Q_2$, a preprocessing step including variable minimization and query normalization is taken to trans-
form them into a normal form. Then two tree structures are constructed for respectively representing the pattern matching and result construction components of the query semantics. Based on the tree structures, query containment is reduced to tree homomorphism, with some specific mapping conditions. Important notations and theorems are also presented to support our XQuery containment and rewriting approaches.

For the cache replacement, we propose a fine-grained replacement strategy based on the detailed user access statistics recorded on the internal XML view structure. As a result, less frequently used XML view fragments are replaced to achieve a better utilization of the cache space.

Finally, we have implemented a semantic caching system called ACE-XQ to realize the proposed techniques. Case studies are conducted to confirm the correctness of our XQuery containment and rewriting approaches by comparing the query results produced by utilizing ACE-XQ against those by the remote XQuery engine. Experimental studies show that the query performance is significantly improved by adopting ACE-XQ, and that our partial replacement helps to enhance the cache hits and utilization comparing to the traditional total replacement.
Acknowledgments

I would like to express my gratitude to my advisor, Prof. Elke A. Rundensteiner, for her support, advice, patience, and encouragement throughout my graduate studies. It is not often that one finds an advisor and colleague that always finds the time for listening to the little problems and huddles that unavoidably crop up in the course of performing research. Her technical and editorial advice was essential to the completion of this dissertation and has taught me innumerable lessons and insights on the workings of academic research in general.

My thanks also go to the members of my Ph.D. committee, Prof. Carolina Ruiz, Prof. Dan Dougherty, Prof. Nabil Hachem, Dr. Ashok Malhotra and Prof. Wang-Chien Lee, who provided valuable feedback and suggestions to my previous talks and dissertation drafts that helped to improve the presentation and contents of this dissertation.

The friendship of Andreas Koeller, Kajal Claypool, Xin Zhang, Amy Lee, Hong Su, and all the other previous and current DSRG members is much appreciated. They have contributed to many interesting and good-spirited discussions related to this research. They also provided tremen-
dous mental support to me when I got frustrated at times. I am also grateful
to my colleagues Luping Quan, Gleb G. Ralka, Shirish K. Nilekar and Song
Wang for helping considerably with realizing the system implementation
and experiments. In particular, Gleb G. Ralka implemented the module for
extracting variable dependencies from the input queries and he, with great
ingenuity, convinced that this is the data structure efficient way we wanted
it. My thanks go to Song Wang who helped to conduct the experimental
studies for the replacement strategy used.

Last, but not least, I would like to thank my husband Hao for his un-
derstanding and love during the past few years. His support and encour-
agement was in the end what made this dissertation possible. My parents
receive my deepest gratitude and love for their dedication and the many
years of support during my studies.
Contents

I Semantic Caching

1 Introduction
  1.1 Motivation ................................................. 2
  1.2 Some of the Issues Involved in Semantic Caching .......... 4
    1.2.1 Query Containment and Rewriting ................. 5
    1.2.2 Semantic Cache Management ..................... 7
    1.2.3 Data-shipping Architecture ........................ 10
    1.2.4 Recapitalization of Involved Tasks ............. 11
  1.3 New Challenges in the Web Context ..................... 12
    1.3.1 Emergence of New Data Model and Query Language 13
    1.3.2 Challenges in Query Containment and Rewriting ...... 16
    1.3.3 Challenges in Semantic Cache Management .......... 19
    1.3.4 Challenges in Maintaining Cache Coherence ........ 22
  1.4 Problem Definition ..................................... 23
    1.4.1 Containment and Rewriting for XQuery .......... 23
    1.4.2 Replacement for XQuery-based Semantic Cache .... 26
  1.5 Approach Overview .................................... 27
    1.5.1 XQuery Containment and Rewriting Approach .... 27
    1.5.2 Approach for XQuery Cache Replacement .......... 31
  1.6 The Contributions ..................................... 32
  1.7 Organization of this Dissertation ...................... 36

II Containment and Rewriting for XQuery

2 Background on Query Containment and Rewriting ............ 39
  2.1 Preliminaries .................................. 39
  2.2 Challenges from XML and its Query Languages ........ 43
    2.2.1 Background on XML .................... 43
<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.2.2</td>
<td>Background on XML Query Languages</td>
<td>45</td>
</tr>
<tr>
<td>2.3</td>
<td>Background on XPath Containment</td>
<td>47</td>
</tr>
<tr>
<td>2.3.1</td>
<td>Complexity Results for XPath Containment</td>
<td>50</td>
</tr>
<tr>
<td>2.3.2</td>
<td>XPath Containment Algorithm</td>
<td>52</td>
</tr>
<tr>
<td>3</td>
<td>XQuery Containment</td>
<td>54</td>
</tr>
<tr>
<td>3.1</td>
<td>Towards XQuery Containment</td>
<td>54</td>
</tr>
<tr>
<td>3.1.1</td>
<td>Motivating Example of XQuery Containment</td>
<td>55</td>
</tr>
<tr>
<td>3.1.2</td>
<td>Problem Requirements in XQuery Containment</td>
<td>58</td>
</tr>
<tr>
<td>3.1.3</td>
<td>XQuery Containment Approach Overview</td>
<td>59</td>
</tr>
<tr>
<td>3.1.4</td>
<td>Outline of This Chapter</td>
<td>62</td>
</tr>
<tr>
<td>3.2</td>
<td>An XQuery Fragment Beyond XPath</td>
<td>62</td>
</tr>
<tr>
<td>3.2.1</td>
<td>The Formal Semantics of the XQuery Fragment</td>
<td>63</td>
</tr>
<tr>
<td>3.3</td>
<td>Variables in XQuery</td>
<td>67</td>
</tr>
<tr>
<td>3.3.1</td>
<td>Variable Binding Dependency</td>
<td>67</td>
</tr>
<tr>
<td>3.3.2</td>
<td>Variable Essentiality</td>
<td>71</td>
</tr>
<tr>
<td>3.4</td>
<td>XQuery Normalization</td>
<td>80</td>
</tr>
<tr>
<td>3.4.1</td>
<td>Different Goals in XQuery Normalization</td>
<td>80</td>
</tr>
<tr>
<td>3.4.2</td>
<td>Our Normalization Steps</td>
<td>83</td>
</tr>
<tr>
<td>3.5</td>
<td>XQuery Decomposition</td>
<td>86</td>
</tr>
<tr>
<td>3.5.1</td>
<td>VarTree and TagTree</td>
<td>86</td>
</tr>
<tr>
<td>3.5.2</td>
<td>Tree Representations Connected to Formal Semantics</td>
<td>90</td>
</tr>
<tr>
<td>3.6</td>
<td>VarTree-based Containment Mapping</td>
<td>91</td>
</tr>
<tr>
<td>3.6.1</td>
<td>VarTree Homomorphism</td>
<td>91</td>
</tr>
<tr>
<td>3.6.2</td>
<td>The MAC Mapping Procedure</td>
<td>94</td>
</tr>
<tr>
<td>3.6.3</td>
<td>VarTree Homomorphism in the Logic View</td>
<td>96</td>
</tr>
<tr>
<td>3.7</td>
<td>Extended MAC Mapping Procedure</td>
<td>101</td>
</tr>
<tr>
<td>3.7.1</td>
<td>Extended VarTree Structure</td>
<td>101</td>
</tr>
<tr>
<td>3.7.2</td>
<td>Additional Conditions for VarTree Homomorphism</td>
<td>104</td>
</tr>
<tr>
<td>3.7.3</td>
<td>Put All Together: Extended VarTree Homomorphism</td>
<td>106</td>
</tr>
<tr>
<td>4</td>
<td>XQuery Rewriting</td>
<td>110</td>
</tr>
<tr>
<td>4.1</td>
<td>Issues in XQuery Rewriting</td>
<td>110</td>
</tr>
<tr>
<td>4.1.1</td>
<td>Intuitive Idea for Rewriting</td>
<td>110</td>
</tr>
<tr>
<td>4.1.2</td>
<td>TagTree Revisit</td>
<td>112</td>
</tr>
<tr>
<td>4.1.3</td>
<td>Intuitive Idea Invalidated</td>
<td>113</td>
</tr>
<tr>
<td>4.1.4</td>
<td>Interpretation of TagTree Semantics</td>
<td>114</td>
</tr>
<tr>
<td>4.2</td>
<td>The XQuery Rewriting Approach</td>
<td>117</td>
</tr>
<tr>
<td>4.2.1</td>
<td>The Key Idea: Translating Restructuring Logic</td>
<td>118</td>
</tr>
<tr>
<td>4.2.2</td>
<td>Translation of Skolem Functions</td>
<td>118</td>
</tr>
<tr>
<td>Section</td>
<td>Title</td>
<td>Page</td>
</tr>
<tr>
<td>---------</td>
<td>-------</td>
<td>------</td>
</tr>
<tr>
<td>4.2.3</td>
<td>Translation of Deep Returning Functions</td>
<td>121</td>
</tr>
<tr>
<td>4.2.4</td>
<td>Overall XQuery Rewriting Procedure</td>
<td>122</td>
</tr>
<tr>
<td>III</td>
<td>Cache Replacement</td>
<td>125</td>
</tr>
<tr>
<td>5</td>
<td>Background on Replacement Strategies</td>
<td>126</td>
</tr>
<tr>
<td>5.1</td>
<td>Basics in Cache Management</td>
<td>126</td>
</tr>
<tr>
<td>5.1.1</td>
<td>The Importance of Replacement and Its Metrics</td>
<td>126</td>
</tr>
<tr>
<td>5.1.2</td>
<td>Traditional Replacement Policies</td>
<td>127</td>
</tr>
<tr>
<td>5.2</td>
<td>Comparisons of Semantic and Page/Tuple Caches</td>
<td>129</td>
</tr>
<tr>
<td>6</td>
<td>Replacement for XQuery-based Semantic Cache</td>
<td>131</td>
</tr>
<tr>
<td>6.1</td>
<td>The Total Replacement Strategy and Its Drawbacks</td>
<td>131</td>
</tr>
<tr>
<td>6.1.1</td>
<td>Alternative Cache Region Management Schemes</td>
<td>131</td>
</tr>
<tr>
<td>6.1.2</td>
<td>Total Replacement at the Query Level</td>
<td>132</td>
</tr>
<tr>
<td>6.2</td>
<td>The Partial Replacement Strategy</td>
<td>135</td>
</tr>
<tr>
<td>6.2.1</td>
<td>Overview of Our Partial Replacement Approach</td>
<td>135</td>
</tr>
<tr>
<td>6.2.2</td>
<td>Query Descriptor Hierarchy</td>
<td>137</td>
</tr>
<tr>
<td>6.2.3</td>
<td>Utility Value and Replacement Function</td>
<td>139</td>
</tr>
<tr>
<td>6.2.4</td>
<td>The Partial Replacement Algorithm</td>
<td>142</td>
</tr>
<tr>
<td>6.3</td>
<td>The Analysis of Cache Performance</td>
<td>145</td>
</tr>
<tr>
<td>6.3.1</td>
<td>Query Trace Model</td>
<td>146</td>
</tr>
<tr>
<td>6.3.2</td>
<td>Cache Hit Probability</td>
<td>147</td>
</tr>
<tr>
<td>6.3.3</td>
<td>Hit Ratio and Different Replacement Strategies</td>
<td>151</td>
</tr>
<tr>
<td>IV</td>
<td>System Implementation and Evaluation</td>
<td>154</td>
</tr>
<tr>
<td>7</td>
<td>ACE-XQ: An Cache-aware XQuery Processing System</td>
<td>155</td>
</tr>
<tr>
<td>7.1</td>
<td>The Objectives of the ACE-XQ System</td>
<td>155</td>
</tr>
<tr>
<td>7.1.1</td>
<td>The Purposes of System Developing</td>
<td>156</td>
</tr>
<tr>
<td>7.1.2</td>
<td>The Two Phases of System Implementation</td>
<td>156</td>
</tr>
<tr>
<td>7.2</td>
<td>The ACE-XQ System Overview</td>
<td>158</td>
</tr>
<tr>
<td>7.2.1</td>
<td>The ACE-XQ Architecture</td>
<td>158</td>
</tr>
<tr>
<td>7.2.2</td>
<td>The ACE-XQ System Interface</td>
<td>161</td>
</tr>
<tr>
<td>7.2.3</td>
<td>The Main Internal Modules</td>
<td>163</td>
</tr>
</tbody>
</table>
8 Experimental Studies

8.1 Experiment Setup ............................................. 169
  8.1.1 Objectives of Experimental Study ..................... 169
  8.1.2 System Deployment .................................... 170
  8.1.3 Experiment Design ..................................... 173

8.2 Experimental Evaluation .................................. 176
  8.2.1 Use Case Studies for Approach Validation .......... 176
  8.2.2 Query Performance With and Without Cache ...... 179
  8.2.3 Experiments on Replacement Strategies ........... 188

V Conclusions

9 Conclusions and Future Work

9.1 Conclusions .................................................. 200
  9.2 Results and Contributions of this Dissertation ....... 202
  9.3 Ideas for Future Work .................................... 206
# List of Figures

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.1</td>
<td>Query Performance Improvements from Semantic Caching</td>
<td>5</td>
</tr>
<tr>
<td>1.2</td>
<td>Research Dimensions in Query Containment and Rewriting</td>
<td>7</td>
</tr>
<tr>
<td>1.3</td>
<td>Query Semantic Region</td>
<td>8</td>
</tr>
<tr>
<td>1.4</td>
<td>Probe and Reminder Queries</td>
<td>9</td>
</tr>
<tr>
<td>1.5</td>
<td>The Framework of Semantic Caching for XQuery</td>
<td>33</td>
</tr>
<tr>
<td>2.1</td>
<td>The Example XML Document: bib.xml</td>
<td>44</td>
</tr>
<tr>
<td>2.2</td>
<td>The Example DTD: bib.dtd</td>
<td>45</td>
</tr>
<tr>
<td>2.3</td>
<td>The Framework of Semantic Caching for XQuery</td>
<td>49</td>
</tr>
<tr>
<td>2.4</td>
<td>Illustration of XPath Evaluation and Containment</td>
<td>50</td>
</tr>
<tr>
<td>2.5</td>
<td>Tree Pattern Homomorphism Algorithm for XPath Containment</td>
<td>53</td>
</tr>
<tr>
<td>3.1</td>
<td>Example Queries Q1 and Q2</td>
<td>55</td>
</tr>
<tr>
<td>3.2</td>
<td>First Trial of Representing XQuery by Tree Patterns</td>
<td>56</td>
</tr>
<tr>
<td>3.3</td>
<td>Second Trial of Representing XQuery by Tree Patterns</td>
<td>57</td>
</tr>
<tr>
<td>3.4</td>
<td>Flow of Containment Checking</td>
<td>61</td>
</tr>
<tr>
<td>3.5</td>
<td>BNF Rules for XQuery Fragment</td>
<td>62</td>
</tr>
<tr>
<td>3.6</td>
<td>Variable Essentiality Matrix</td>
<td>75</td>
</tr>
<tr>
<td>3.7</td>
<td>Essentiality Identifying Model</td>
<td>76</td>
</tr>
<tr>
<td>3.8</td>
<td>The Variable Minimization Procedure</td>
<td>79</td>
</tr>
<tr>
<td>3.9</td>
<td>BNF Rules for Normalized XQuery</td>
<td>85</td>
</tr>
<tr>
<td>3.10</td>
<td>Example Query Q1: Before and After Normalization</td>
<td>86</td>
</tr>
<tr>
<td>3.11</td>
<td>Q1’s VarTree and TagTree</td>
<td>88</td>
</tr>
<tr>
<td>3.12</td>
<td>Q2’s VarTree and TagTree</td>
<td>89</td>
</tr>
<tr>
<td>3.13</td>
<td>The MAC Mapping Procedure</td>
<td>96</td>
</tr>
<tr>
<td>3.14</td>
<td>MAC Mapping between VarTrees of Q1 and Q2</td>
<td>97</td>
</tr>
<tr>
<td>3.15</td>
<td>Extended VarTrees and Implied Variable Region Dependencies</td>
<td>104</td>
</tr>
<tr>
<td>3.16</td>
<td>The Extended MAC Mapping Procedure</td>
<td>109</td>
</tr>
</tbody>
</table>
## LIST OF FIGURES

4.1 The XQuery Rewriting Procedure .......................... 123
4.2 First Rewriting Example ............................... 124

6.1 Pictorial Illustration of the First Region-splitting Technique . 134
6.2 Pictorial Illustration of the Second Region-splitting Technique 134
6.3 Path Table with Initial Statistics for $Q_1$ ............... 138
6.4 $Q_2$’s Path Table and $Q_1$’s Path Table with Updated Statistics 139
6.5 The Replacement Control Flow ........................... 142
6.6 An Example Filter Query ............................... 143
6.7 The Partial Replacement Algorithm ....................... 144

7.1 The ACE-XQ System Architecture ......................... 159

8.1 Passing/Failing Use Cases .............................. 177
8.2 Query Response Time for Different Document Sizes .... 180
8.3 Query Response Time for Different Query Selectivities .. 182
8.4 Query Response Time for Different Query Depths .... 184
8.5 Time Decomposition for Computational Overhead .... 185
8.6 Contained Query Workload ............................ 186
8.7 Mixed Query Workload ................................. 187
8.8 Totally Random Query Workload ....................... 187
8.9 Refining trace: HCR for Two Replacement Strategies With Varying Cache Sizes .......................... 191
8.10 Random trace: HCR for Two Replacement Strategies With Varying Cache Sizes .......................... 192
8.11 Refining trace: HBR for Two Replacement Strategies With Varying Cache Sizes .......................... 194
8.12 Random trace: HBR for Two Replacement Strategies With Varying Cache Sizes .......................... 195
8.13 Refining trace: RT for Two Replacement Strategies With Varying Cache Sizes .......................... 196
8.14 Random trace: RT for Two Replacement Strategies With Varying Cache Sizes .......................... 197
Part I

Semantic Caching
Chapter 1

Introduction

1.1 Motivation

Accessing information sources to retrieve data requested by a user can be expensive, especially when dealing with distributed information sources. Because performance is a crucial issue in database systems, data caching techniques have been studied in the database research field, especially in client-server databases [N. 91, A. 92] and distributed databases [OV99, BG84, BFH+90]. Clients with substantial CPU and memory to be capable of performing intensive computations can help to offload the servers and improve system performance and scalability by caching and maintaining data locally. Even for the modern multi-tier web applications, caching pre-computed results at the client side for future reuses is often beneficial.

Recently, the idea of semantic caching [DFJ96] has been proposed. The approach uses semantic information to describe cached data items with the goal of exploiting semantic locality to improve query response time. That
is, semantic caching suggests to cache users’ queries and correspondingly to organize the answers (instead of pages or tuples) by their query descriptions. Based on these query descriptions, a semantic caching system determines whether a given input query \( Q \) is logically contained or overlapping with a cached query \( V \).

This logical containment relationship at the query description level implies the subset relationship at the query result level. If the current query \( Q \) is answerable from the cache, namely, there exists a query \( V \) whose answer set subsumes that of \( Q \), then no communication with the data server is necessary. If \( Q \) is only partially answerable from the cache, \( Q \) is then split into a *probe query* that can be rewritten w.r.t. \( V \) to retrieve the portion of the result available in the cache and a *remainder query* that retrieves any missing tuples in the answer from the remote data site. The amount of data needed from the servers may be substantially reduced this way. In addition, redundant network bandwidth and connection overhead can be saved. Users will benefit from the lowered response latency and improved connection reliability.

Semantic caching is a query-based caching strategy. It provides an alternative approach to page-caching [CFZ94] and tuple-caching architectures [DFMV90], in which the unit of transfer between servers and clients is a page or a tuple, respectively. Semantic caching is advantageous compared to the tuple-based and page-based caching schemes in terms of its flexibility: it can be used to answer new queries which are semantically related to previous queries that were cached. This is a quite common scenario during query sessions, in which a user asks a number of related and follow-up
queries. For example, the user may refine the previously asked query by tightening the specified conditions to become more restricted. Tuple-based and page-based caches are relatively inflexible. The new query must ask the same sources for the same information which is usually the physical tuple or page IDs. Therefore, tuple-based and page-based caches cannot satisfy those applications that require associative accesses to data via query constraints.

With the advance of the World Wide Web, exchange, utilization and integration of information across heterogeneous sources has become a ubiquitous need. A mechanism capable of caching previously computed view results for answering future queries would be beneficial for improving the query performance. This can be achieved by the savings of not sending queries and their results over a distance. Such a scenario is depicted in Figure 1.1.

1.2 Some of the Issues Involved in Semantic Caching

A semantic caching system needs to provide support for the integration of three essential parts: 1) a fundamental technique for tackling the query containment and rewriting problem; 2) a cache management system supporting associative accesses, region replacement and cache coherence; 3) a distributed architecture based on data-shipping.
1.2 SOME OF THE ISSUES INVOLVED IN SEMANTIC CACHING

The problem of query containment and rewriting is fundamental to semantic caching, in addition to many other applications such as query evaluation and optimization in database systems. This problem was first studied by Chandra and Merlin [CM77] for conjunctive queries, whose expressive power is equivalent to that of the Select-Project-Join (SPJ) queries in relational algebra. Since then a flurry of extensive research efforts have followed to investigate all the relevant aspects ranging from the complexity theory to its practical applications in optimizing queries, answering queries using views, detecting update relevancy, etc.

Basically, the research of the query containment problem has been car-
1.2. SOME OF THE ISSUES INVOLVED IN SEMANTIC CACHING

ried out from the following different angles (see the details in the survey [Lev00]). One category is the theoretical foundation regarding the solvability and computational complexity. The theoretical research first studied the class of relational conjunctive queries. The containment complexity for this class of queries is NP-complete in general. However, efficient algorithms have been found for the queries satisfying particular constraints. For example, it has been proven in [CR97] that the evaluation of conjunctive queries of bounded tree width is in polynomial time.

The second research direction in this field targets to find algorithms that help to establish containment mappings and consequently rewrite queries using views. Four well-known algorithms, i.e., the bucket algorithm [LRO96], inverse-rules algorithm [X. 96, DG97], MiniCon algorithm [PL00], and shared-variable-bucket algorithm [Mit01], have been proposed to rewrite queries using views before executing the queries.

The third branch of research studies the containment problem in the context of different data models and query languages. Beyond the study of containment for relational conjunctive queries, dozens of extensions that include query features such as aggregations [SDJL96], disjunctions [AGK99], recursive queries and non-recursive views [CV92, DG97], and build-in predicates [P. 98] have been extensively studied. There is also work extending the relational query containment algorithms for the complex object model [LS97a]. Recently, XML has emerged as the principal medium for data exchange over the Web. With the ever-increasing popularity of XML data and the growing maturity of its standard query language – XQuery [W3C03c], studying containment for queries over semistructured data such
as XML has renewed immense attention [FLS98, LS97b, G. 02b, A. 01a, Woo03, FT03].

Figure 1.2 summarizes the previous research work in three directions.

Figure 1.2: Research Dimensions in Query Containment and Rewriting

1.2.2 Semantic Cache Management

Cache Content Organization. In a semantic cache, cached contents are organized by queries. That is, a hash-table like data structure is managed by a semantic cache to enable the associative access of cached data via the lookup by queries. For example, Figure 1.3 depicts an SQL query and its corresponding answer set represented by a spatial region scoped in terms of query predicates.
1.2. SOME OF THE ISSUES INVOLVED IN SEMANTIC CACHING

Hence, each cached entry consists of a (key, value, function) triple for representing a semantic region. The key is a high-level query description extracted from the original syntactical format of a query. The value refers to the corresponding query answers accessible through the key. When a new incoming query is found to be semantically contained or overlapping with a cached query, a probe query is computed for retrieving the available portion of the query answers within the cache, and a remainder query is sent to the remote data server to fetch the remaining part of the query answers. Such a re-use of cached contents based on the semantic query containment relationship is depicted in Figure 1.4.

The function in the entry triple refers to a replacement function which is used to help with the cache replacement decision. Based on such a replacement function, a particular replacement policy (e.g., LRU, LFU) is carried out to determine a victim query semantic region to be removed from the cache in case there is no room left in the cache for a new query.

Figure 1.3: Query Semantic Region
1.2. SOME OF THE ISSUES INVOLVED IN SEMANTIC CACHING

Cache Replacement. An important issue needs to be dealt with in a semantic caching system or a cache system in general is the policy for admission into the cache and eviction from the cache. When a query is executed, we must decide what part, if any, of the query result to add to the cache. If the cache is full, we must also decide which of the currently cached results to evict from the cache. The objective of the replacement policy is to minimize the cost of executing the current and future queries and therefore to maximize the benefit of the cache. User access statistics can be gathered to be used by a variety of replacement approaches.

In the setting of a semantic caching system, the replacement strategy would be influenced by the characteristics specific to semantic caches. For example, the user traces recording query sessions are different from the physical page accesses in a file system. Also, since the cache content is organized in terms of the answers for different queries, the cache region sizes

Q2: select * from Employee where age $\geq$ 35 and age $\leq$ 50 and salary $\geq$ 2500 and salary $\leq$ 4000

Figure 1.4: Probe and Reminder Queries
likely vary from query to query. Therefore, the analysis of these characteristics and the evaluation of the performance results of a replacement strategy are indispensable to establish a valuable replacement strategy maximizing the hit rate or other performance criteria of a semantic cache.

**Cache Coherence.** Caches cause multiple copies of data which introduces a coherence problem. The data copies cached at the client side are subject to become stale since updates may occur to the original data sources. A survey article by Stenstrom [P. 90] provides a nice overview of the snoopy and directory-based cache coherence protocols for maintaining cache consistency in a file system, and of various cache consistency algorithms used in transactional client/server DBMS architecture.

In a data integration environment such as a data warehouse, the issue of cache coherence can also be equated to be similar to materialized view maintenance, a common topic in the database field [GM95, ZM98, S. 98, Run92, QCR00].

### 1.2.3 Data-shipping Architecture

Many client/server database systems are based on data-shipping. In a data-shipping architecture, query processing is performed largely at the clients, and copies of data are brought on-demand from servers to be processed at the clients. This way, the computational and storage resources of the client machines are effectively utilized, which is the key to achieve high performance and scalability in client/server database systems.

The same reasons that caused the emergence of the client-server database
systems to become the predominant distributed database architecture in local area networks (LANs), combined with ever faster and cheaper computers, high-speed networks and network-aware applications, are now causing unprecedented growth in distributed database systems over wide area networks (WANs). The World Wide Web (WWW) is a prime example of a WAN data server system which is primarily read-only for the time being. In the future, it is expected that data shipping systems in which clients perform much of their query and transaction processing locally will be largely deployed over WWW. In this case, the WAN clients empowered with the semantic cache mechanism can evolve and function as data servers. We envision that this can help to realize cooperative process transactions across multiple WAN servers, hence enhancing the query processing performance, minimizing user response latency, creating an even more powerful distributed database environment and collaborating data servers over WWW.

1.2.4 Recapitalization of Involved Tasks

To recap, a semantic caching system needs to provide the solutions to the following problems.

1). Query Containment and Rewriting

   a. How to determine query containment for two given queries;

   b. How to rewrite a query with respect to the result structure of the containing or overlapping query;

2). Semantic Cache Space Management
1.3. NEW CHALLENGES IN THE WEB CONTEXT

1.3.1. How to organize query answers in a semantic cache;

1.3.2. How to record user access statistics information and maintain a better utilization of the cache space;

1.3.3. Cache Coherence

3). How to decide the relevancy of data updates to the cached content;

b. How to efficiently maintain the cached content to be consistent with the up-to-date source data.

1.3. New Challenges in the Web Context

In the era of the Internet, the traditional distributed information systems have evolved into the World Wide Web, which has the significant impact on how organizations manage data and build software systems. Traditional tightly coupled proprietary networks (DCOM, CORBA) are gradually giving way to the dynamic, loosely-coupled, data-driven Web. Recently, the emergence of a new data model for integrating heterogenous Web information has opened up new possibilities for server to server interaction and offered more potential for interoperability, which is the key to the paradigm shift.

The idea of applying the semantic caching approach to Web applications appears very promising, since the savings of data transmissions across the Internet are likely to be more substantial than those achievable in a local network environment. On the other hand, the new data model suitable for
representing the Web data and the compatible query languages utilized by a variety of Web applications have imposed challenges on developing an appropriate semantic caching solution in the Web context.

1.3.1 Emergence of New Data Model and Query Language

Traditionally, the most popular database model is the relational model proposed by Ted Codd in 1970. He also found out that there is a connection between relations and first order logic (FOL for short), which lends formal methods to the analysis of relational queries over finite relational structures.

The emergence of the Web has provided tremendous opportunities for database researchers to investigate issues of Web information management. Web sites are increasingly powered by databases. Collections of linked Web pages distributed across the Internet are themselves tempting targets for a database. A semistructured data model [CGMH\textsuperscript{+}94] has been first introduced for modelling information available on the Web. Being able to represent data with loosely defined or irregular structure, the semistructured data model allows a “schema-less” description format in which the data is less constrained than usually in databases.

Recently, the advance of eXtensible Markup Language (XML) [W3C98] as the lingua franca of the Web meets the needs for an interoperable data exchange format on the Web. Unlike HTML, XML imposes a logical structure on a document by explicitly tagging semantic information, while ignoring details about possible presentation layout options on the screen. This way, structured queries can be processed against such semantics-exposing XML
data instead of using only keywords to search within the “word sea” of unstructured HTML pages. In addition, many Web applications such as Web Services now use XML for both transient messages, like SOAP [W3C03a] or XML-RPC messages [Win99], and as persistent storage, like in XML databases or content management systems [Bou99]. In this sense, XML is the enabling technology for a wide array of Web applications.

An XML document can be modeled as an ordered tree composed of nodes that represent the data objects (i.e., elements and attributes) and edges which are labelled by the (tag) names of the elements and attributes. The XML Document Type Definition (DTD) can be seen as the schema definition of a set of XML document instances. It is used to impose structural constraints on XML documents, i.e., the elements and attributes allowed and their allowable content and nesting structure.

The formal concept underlying the relational model is the set-theoretic relation. Common relational queries are expressible in relational calculus, relational algebra, and non-recursive Datalog, i.e., non-recursive Datalog = relational algebra = SPJ (select-project-join) SQL queries = relational calculus. In particular, the core of the relational database query language has the same expressive power as first-order logic. This has been intensively exploited in the theory of relational databases.

The fundamental role of logic for database query languages continues to hold within the XML context. A flurry of research work has focused on investigating the connection between XML query languages and the decidable logics as well as automata. Suppose $L$ denotes a finite alphabet of labels, and $D$ represents an infinite alphabet of data values. XML trees can
be represented by $T_{L,D}$, where $L$ models XML tags and attributes and $D$ models XML leaf values. Such XML trees correspond to regular tree languages, which are defined based on the corresponding tree automaton as the acceptable languages. Furthermore, there is a connection between regular tree languages and monadic second order logic, i.e., the regular tree languages are exactly the tree languages definable in monadic second order (MSO) logic [H. 02]. MSO logic is an extension of the first order logic with second order variables interpreted as sets of nodes along with quantification over these variables.

To exploit the tree structure of semi-structured or XML data for extracting the desired information, many query languages such as Lore [AQM+97], YAT [SC98], UnQL [P. 00], XPath [W3C03b], XSLT [Gro], XML-QL [DFF+99a], XQL [RLS], Quilt [D. 00] and XQuery [W3C03c] have been proposed in recent years. Among these query languages, Lore, YAT and UnQL have been designed originally for querying semi-structured data (not XML in particular). The others [W3C03b, Gro, DFF+99a, RLS, W3C03c], though with different expressive powers and a variety of syntax, share the important commonality of being capable of addressing parts of an XML document using path expressions [W3C03b].

Since a general requirement of an XML query language is to locate desired element nodes within XML documents, the XPath expressions capable of specifying location paths are therefore one of the most useful constructs in any XML query language. It is discovered that regular expressions have the same expressive power as the MSO logic [B. 90].

However, a powerful and elegant query language is needed to reach
the full potential of querying and transforming XML documents. XPath expressions do not go far enough. Therefore, XQuery has recently been proposed by W3C as the standard XML query language [W3C03c]. Besides XPath expressions, the language elements of XQuery also include different types of query expressions, such as the “element constructor” mechanism for constructing arbitrary XML result documents, the FLWR expressions for collecting variable bindings, filtering retrieved nodes, and returning the result, the user-defined functions, conditional expressions, operators, and quantifiers.

1.3.2 Challenges in Query Containment and Rewriting

Due to the fundamental role played by query containment [CM77] in many database applications such as query optimization and information integration [Lev00], it has received considerable attention over the past few decades in the context of relational queries.

Some major query containment results in the relational query context are summarized below.

- Query containment is undecidable for arbitrary first order queries [O. 93].

- Query containment is decidable (NP-complete) for conjunctive queries without build-in predicates [CM77].

- Query containment is decidable ($\Pi_2^p$-complete) for conjunctive queries with build-in predicates [P. 98].
With the prevalence of XML data and its query languages such as XQuery, the study of containment for queries over semistructured data such as XML has received increased attention. Extensive research [FLS98, LS97b, G. 02b] has focused on path-expression-oriented queries, i.e., regular path expressions and a variety of XPath fragments. Within this limited navigational query mechanism, significant theoretical results in terms of containment decidability and complexity have been established in the presence and absence of DTDs and other path constraints [A. 01a, Woo03, FT03].

However, the containment checking technique has so far been left unexplored when it concerns other query constructs, beyond the pure navigational path expression mechanism, as the essential compositional components for a rather comprehensive XML query language such as XQuery. XQuery has rapidly gained a growing recognition due primarily to its adoption of the FLWR expressions. With the full compositionality, FLWR expressions can specify variable bindings, perform conditional selections and result restructuring in a flexible manner. These features enhance the query expressiveness. On the other hand, they impose significant challenges to the containment reasoning problem.

The problem of query rewriting is closely related to the problem of query containment. Rewriting queries using views is a powerful technique that has applications in data integration, data warehousing, query optimization and modern applications such as mobile computing. Query rewriting in relational databases is by now rather well investigated. However, the problem of rewriting has received much less attention for semistructured data.
1.3. NEW CHALLENGES IN THE WEB CONTEXT

Similar to query containment, the techniques for query rewriting are also language specific. Given a new query \( Q \) and a set of views \( V \), query rewriting often involves the rewriting of \( Q \) using a set of views \( V \), rather than an individual view \( V_i \) (\( V_i \in V \)). The notations of \textit{minimal rewriting} and \textit{complete rewriting} are often mentioned \[LMSS95\]. They refer to the rewriting with the minimal total number of literals and that uses only the views in the rewriting respectively.

In the relational query context, the results on the complexity of query containment entail the following complexity results on the problem of query rewriting \[Lev00\]:

- \( Q \) is a conjunctive query with build-in predicates and \( V \) are conjunctive views without build-in predicates, then the problem of determining whether there exists a rewriting of \( Q \) that uses \( V \) is NP-complete.

- If both \( Q \) and \( V \) are conjunctive and have build-in predicates, then the problem of deciding whether there exists a rewriting of \( Q \) that uses \( V \) is \( \Pi_2^{P} \)-complete.

Answering queries using views, which consists of query containment and rewriting, has recently been studied for semistructured data models in \[CGLV00, PV99b\]. \[CGLV00\] focuses on the rewriting of regular path queries and \[PV99b\] addresses the problem of query rewriting in a specific semistructured data model and query language.

However, query rewriting for comprehensive XML query languages such as XQuery has not yet been studied due to the lack of a query containment strategy for it thus far. Even with the placement of a query con-
tainment solution, the arbitrarily nested view structures could impose new challenges to the rewriting of a new query with respect to them.

1.3.3 Challenges in Semantic Cache Management

The support of a semantic cache implies several issues concerning the management of the semantic cache space. The first issue is about how the cached query answers are organized. As we have mentioned before, the hash-table like data structure for mapping from logical query descriptors to their corresponding data content is the enabling mechanism for the management of a semantic cache. The accesses and manipulations of the cached data for answering new queries, recording statistics, region merging, splitting and evicting are managed at the query level. Compared to the tuple [DFMV90] or page-based [CFZ94] caching mechanisms which organize the cached data by their physical tuple identifications or page numbers, the semantic caching systems bring the flexibility in answering queries using cached views based on their logical containment relationships.

In a traditional semantic caching system [DFJ96] for relational queries, each query is encoded to serve as the logic descriptor of the associated query answer. The encoding scheme needs to capture the essential ingredients of relational queries, i.e., select-project-join. For example, the datalog rule representation [CM77] can be used as the encoded form. The query descriptors serve two functionalities. One, they can facilitate the containment mapping process. Two, they are also associated with user access statistics information for the replacement algorithms to decide which victim queries to purge.
1.3. NEW CHALLENGES IN THE WEB CONTEXT

In the case when a semantic caching system deals with XML queries, we need to re-think of what may be an appropriate representation of the query descriptor. Due to the fundamental difference in data models, XML queries are naturally different from relational queries in both syntax and expressiveness in specifying the tree-structure-based selection, projection and restructuring. A concise form that can capture the essence of XML queries while serving as the basis for containment reasoning is one of the sought-after objectives.

In the traditional CPU cache system, various replacement schemes have been studied [RD90, NNW93, JS94, CFL94, PG95, LCN+99]. Among them, the Least Recently Used (LRU) and the Least Frequently Used (LFU) replacement schemes are the most well-known replacement policies. The LRU policy considers that the recently referenced objects are likely to be re-referenced in the near future [CD73], while the Least LFU policy [RD90] assumes that the more often a query is being used to answer future queries, the more likely it is to be used to answer a future query.

In the Web context, exploiting the cache mechanism for maintaining a cache of web documents can dramatically reduce demand on the network as well as latency seen by the user. Therefore, web caching has recently been extensively studied with the goal of reducing response latency and network traffic. However, conventional wisdom gained in the context of page replacement for CPU caches does not necessarily transfer to Web caches. In particular, the LRU policy does not take into account the size of the documents, which is an important impact on web caching performance since Web documents usually vary dramatically in size depending largely
on the type of information they contain (video, audio, text, etc). Another
important difference between CPU caches and web caches is that the cost
to bring in data to the cache can vary dramatically in the context of web
caches depending on where the data resides on the Internet. The cost to
fetch a block from disk is much more uniform.

Furthermore, semantic caching in the web context is not equivalent to
web caching in general. The cache content in a semantic cache is the query
result, rather than the static web pages or fragments of pages. In a dis-
tributed web application environment, the cache mechanism is usually de-
ployed at the web proxy side. If plugged with a “query applet”, current
web proxies could deal with simple queries and answer queries that exactly
match to cached queries. In other words, they do not consider database
query containment nor the evaluation of subsumed queries at the proxy.

In terms of replacement strategies, the key factors of concern in web
proxies are the characteristics of Web object accesses including the object
size and frequency of reference. A lot of research efforts have focused
on analyzing the importance of different Web proxy workload character-
istics in making good cache replacement decisions. In the semantic caching
context, we also aim at reducing network traffic, server load, and user-
perceived retrieval delays in the web environment. Since the semantic
cache utilizes query containment for answering queries that are subsumed
within or overlapped with cached queries, query access statistics can be
recorded at a finer granularity level than the whole view document. More-
over, the semantic cache can possibly incorporate the consideration of such
detailed access statistics information in the replacement algorithms to ef-
The main idea of a semantic caching system is to keep the local copies of data in the form of query answers in order to achieve a fast response time for processing possibly recurring or contained subsequent queries. However, this can also cause the problem of stale data. That is, the remote data is often subject to updates (this is especially true for web data which is in continuous flux) and hence renders the locally cached query answers obsolete and inconsistent with the source data. This then requires the development of efficient techniques for maintaining the cached views to be consistent with the remote dynamic source data.

In the conventional database systems, the materialized view mechanisms and their maintenance issues have for long been a well-studied topic [AMR98, GGMS97, GM95, GMS93, KLMR97, RKRC96, SLT91]. However, developing the maintenance techniques for the cached XML query answers over the dynamic XML data is an issue with new challenges due to the complex tree-oriented XML data model and the expressive XML query language.

Zhuge et. al. [ZM98] first address the maintenance issue of the graph-structured views over semi-structured data. They simplify views by considering only the select-project view specifications. Those each produce a flat collection of objects without being organized in any structure. Abiteboul et. al. [AMR+98] generalize the previous studies to cover arbitrary graph-structured databases. Their approach can handle queries with joins...
and structured results. Moreover, it can improve the maintenance efficiency over the re-computation approach by utilizing a local auxiliary structure for accommodating the objects relevant to the web view computation.

1.4 Problem Definition

1.4.1 Containment and Rewriting for XQuery

We first investigate the challenges imposed by the problem of XQuery containment and present an approach for tackling it. The containment for a complete XML query language such as XQuery remains unexplored in the literature. Compared to XPath, XQuery is more powerful in the sense that it can specify sophisticated queries by utilizing variable bindings, element constructions, and result restructuring. XPath expressions serve as a basic query construct of XQuery for selecting objects to be associated with variables or to be returned in the result. Clearly, any result on the XPath containment topic provides the foundation for solving XQuery containment. In particular, it can resolve the containment relationship between two selected node sets if they are derived from the same starting node via different XPath expressions. However, bridging the gap between the containment for XPath and for XQuery is not trivial. We are facing the following difficulties.

First, the research on XPath containment has been primarily focussed on the complexity of the containment problem for various fragments of XPath [G. 02b, Woo03, FT03] rather than on the full XPath language [W3C03b]. Among other results, the containment of the XPath fragment $X P^{(/,//|\star)}$ is
coNP-complete [G. 02b]. If disjunction “|” is added into this fragment and XML documents are restricted to a finite alphabet, than the containment complexity jumps to PSPACE-complete [FT03]. Therefore, restricting the XPath features being considered for the containment problem to be within a certain fragment is a valid research methodology towards problem solving. In our problem domain, many other features such as variable bindings, nested FLWR expressions, and element constructors coexist with XPath to compose XQuery. The question followed is that which subset of XQuery features should be given higher priority to be considered for the containment problem.

Second, the existing XPath containment work often exploits certain pattern trees [G. 02b] for representing XPath expressions, based on which the XPath containment problem can be reduced to tree homomorphism. Hence the questions is if XQuery can be represented in a similar pattern tree form as that for XPath fragment, or what is an appropriate representation of XQuery such that it precisely captures the semantics of the considered features and can serve as a mechanism facilitating the containment checking and rewriting.

Third, since XQuery is has more sophisticated features than XPath, the procedure for determining the containment of XQuery is likely more complicated than that for XPath. The questions hence are what are the specific conditions required for containment checking, and how are they utilized in the containment algorithm for determining the containment.

Fourth, the result of an XPath expression query is a node set of a single element type, while that of an XQuery is a tree composed of data bind-
nings derived from the original XML document. Such a result restructuring capability of XQuery imposes difficulties to the rewriting problem. If a containment relationship $Q_2 \subseteq Q_1$ is determined, we need to answer the question of how to match the data pieces in $Q_1$’s result tree to their origins in the input XML data tree so that $Q_2$ can be rewritten to be redirected to locate the desired data bindings in $Q_1$’s result tree instead of the original XML data tree.

**Objectives.** With the overall goal of solving the XQuery containment and rewriting problem, we list the followings as our tasks.

1. Define an XQuery fragment containing an appropriate subset of XQuery features to be considered for the containment problem$^1$;

2. Devise a precise representation for XQuery which can serve as a mechanism facilitating the containment checking and rewriting;

3. Propose the containment checking conditions and the algorithm that utilizes them for determining containment;

4. Prove the soundness of our containment checking approach;

5. Find a mechanism for establishing the mapping between data in the restructured result tree and their origins in the input tree, and design the rewriting technique exploiting this mapping.

$^1$By “appropriate”, we mean that the core features such as nested FLWR expressions, variable bindings and result constructions which are distinct from XPath features but do not necessarily induce a jump of containment complexity are included in the fragment. Features that are either too trivial to be considered or would cause high complication for the containment problem will however be left out for this work.
1.4.2 Replacement for XQuery-based Semantic Cache

In the traditional semantic caching systems, a query region is the minimal granularity managed in the cache. Upon the incoming of a new query overlapping with a cached one, the cached query region is either split into two or preserved as a whole. The former cache region management scheme will cause the cache space to be severely fragmented over time, while the latter scheme does not allow for a precise recording of the XML fragment popularities due to the coarse granularity of total query regions.

The replacement strategy based on either of these two schemes suffers drawbacks. If the cache space is over-fragmented, caching a new query may require to purge many tiny query regions. If imprecise user access statistics are recorded on query regions, the replacement would become rather random. Also, the replacement unit may be too coarse-grained to maintain an efficient cache space utilization.

The question is if we can record user access statistics at a fine-grained level rather than on query regions while still avoiding the physical region splitting. This way, the replacement function can calculate the utility value based on the precise statistics but does not suffer the over-fragmentation problem. Our objectives are hence to find such a way for recording fine-grained statistics, to propose a replacement function utilizing such information, and to maintain the cache regions.
1.5 Approach Overview

1.5.1 XQuery Containment and Rewriting Approach

We first define an XQuery fragment as the scope for tackling the XQuery containment and rewriting problem. We then provide an XQuery normalization technique for pre-processing XQuery to derive a canonical form that can facilitate the separation of pattern matching and result construction. Based on their connection via variables, we then reduce the problem of XQuery containment to the problem of variable-based containment mapping. The key is to utilize the variable dependencies inferred from their definitions. We capture those by a tree-like structure. Then the variable-based containment mapping can be performed in a tree embedding fashion. That is, to determine the XQuery containment, e.g., $Q_2 \subseteq Q_1$, we check whether an embedding of the tree representing the variable dependencies of $Q_2$ to that of $Q_1$ exists. We show that the variables constituting both trees need to be “essential” for their respective result constructions. We call such an overall mapping at the tree level MAC (macroscopic) mapping. It is established based on the decision of individual variable node mappings called MIC (microscopic) mappings. To set up a MIC mapping between a variable node pair, we apply XPath containment on the derivation path expressions of the corresponding variables.

If a view XQuery $Q_1$ is determined to be useful for answering the new XQuery $Q_2$, we further rewrite $Q_2$ with respect to $Q_1$ in order to re-use its cached view result. The rewriting of $Q_2$ involves the rewriting of the derivation paths of the variables specified in $Q_1$ with respect to the result
structure of $Q_1$. For this, the maximal variable mapping pairs produced by
the XQuery containment procedure can be utilized. We hence deal with the
issues such as “data correspondence” in order to merge the results derived
from the same source by different queries using a combining query.

Figure 3.4 depicts the flow of our containment checking and rewriting
process. Below we highlight the main steps involved in this process.

1). XQuery Pre-processing: Minimization and Normalization
In this step, we minimize the variables specified in the input query to elim-
inate those that can be substituted for their corresponding bound expres-
sions without affecting the query semantics. Based on the minimized vari-
able set and their dependencies, we can then utilize containment mapping
of variables for determining XQuery containment.

Next, a normalization technique is taken to simplify the input query
into a canonical form for the following two purposes. One, the flexibility
in composing an XQuery using nested FLWR expressions imposes difficul-
ties for reasoning the containment relationship between two given queries.
By sorting out the query constructs and rewriting the input query in ac-
cordance to the pre-defined normalization rules, it may facilitate the con-
tainment reasoning. Two, the goal is for the canonical form to reveal the
pattern matching and result construction semantics in a more decoupled
way, so that we can conduct the containment reasoning in two separate
stages with the focus on the respective parts.

2). XQuery Decomposition: Pattern Matching and Restructuring
After the normalized form of a given query is obtained, pattern matching is
represented by the definitions of variables by XPath expressions in the for clauses. The result construction part corresponds to the nested FWR structure, the new element constructor and the return expressions at each level in the return clauses. Therefore, pattern matching and result construction can be clearly separated and captured from the normalized query form.

Furthermore, the two parts, i.e., pattern matching and result construction, are connected via variables, which accommodate the intermediate data binding result. These data bindings can then be used for invoking new element constructions, or for providing handles for their descendants to be accessed and returned in the result. This way, both pattern matching and result construction semantics can be captured by two tree structures respectively, one of which is constructed based on the variable binding dependencies and the other reflects the result construction template by utilizing the specified variables. We call the former a VarTree and the latter a TagTree.

3). XQuery Containment via VarTree Homomorphism
A common approach for the containment of conjunctive queries is to utilize graph homomorphism [CR97] that finds the most general unifier for mapping the predicates and variables that are encoded as components of a graph structure. In the same spirit, we propose a containment mapping technique, called MAC mapping, which is fundamentally a tree-embedding process [Kil92] with extensions to accommodate the VarTree characteristics.

We further extend the MAC mapping to incorporate a validation of the established containment mapping with the consideration of the control-
flow induced variable dependencies. Together with the variable minimization step, this additional validation can ensure that our extended MAC mapping is a sufficient condition for the containment of the XQuery fragment we are concerned with.

4). XQuery Rewriting

Based on the maximal variable mapping pairs produced in the XQuery containment phase, we rewrite the bound expressions for the essential variables in $Q_2$ using the navigation paths to their corresponding variables, if there is any, in the result structure of $Q_1$. Such a path expression rewriting needs to utilize the tagging template of the containing query, which reveals how the result structure is constructed based on variables. Hence, we can utilize the established variable mapping pairs and the tagging template of $Q_1$ for rewriting $Q_2$.

Since we deal not only with the rewriting of a new query when it is totally contained within a cached query, but also with the rewriting when the new query is partially overlapping with a cached one. We need to consider the issues of formulating the remainder query as well as combining its results with that of the probe query. Unlike other XML merging work [Man01] which either assume or impose object identifiers throughout the source documents to make it possible merging two pieces, we utilize primarily the DTD knowledge and user designated key constraints for identifying the node equivalency. The concept of relative key path proposed by [aDWFH01] is exploited to allow the source subscriber to specify such key constraints.
Overall, our solution is the first step towards reasoning XQuery containment beyond XPath containment and rewriting the new query with respect to the possibly restructured view schema.

1.5.2 Approach for XQuery Cache Replacement

We first investigate the existing alternative schemes and compare them in the presence of replacement activities. The common characteristic among these existing replacement schemes is that the replacement is always tightly coupled with query regions. Namely, regardless of whether query regions are split or preserved upon the caching of new queries, the existing replacement strategies decide on a complete query region at each iteration. Such a replacement strategy indicates that the data granularity being deleted each time is at the level of a whole query region, which may be too coarse for "large" query results. These replacement schemes at the granularity of complete XML views suffer apparent drawbacks such as query regions are over-split or non-precise statistics are recorded on redundantly cached regions.

We hence design a fine-grained replacement strategy which records user access statistics at a finer granularity than the complete XML query regions. Namely, user access statistics information is recorded at the internal structure level for each cached view, rather than a uniform value being recorded for each whole query region. For example, when a cached query contains or partially overlaps with a new query, the utility statistics of the requested portion in the cached query result by the probe query are updated, however without splitting the cached query. If the cache is full, the replacement
manager does not select complete regions but only specific portions with the lowest utility value within such query regions for replacement. A filter query can be composed to remove the less useful portions. The relevant query descriptors are then modified accordingly to be consistent with the changed view document.

Such a decoupling of recording user access statistics and the query region management allows for the popularity tracing of query answers at a fine-grained level while avoiding the explicit region splitting upon every incoming query. Therefore, this so-called partial replacement strategy helps to maintain in the cache the “hot” query regions and hence a better utilization of the cache space.

1.6 The Contributions

The overall contribution of this dissertation work is that it provides a framework for the XQuery-based semantic caching systems. In other words, it gives the general solutions to the main issues involved in building a semantic caching system for XQuery, namely, answering XQuery queries using cached ones, and designing an appropriate replacement strategy that improves the cache space utilization. We have built a prototype system to realize the proposed techniques. This prototype system helps to validate the approach and provides the basis for conducting experimental studies.

This framework for XQuery-based semantic caching systems is depicted in Figure 1.5. It integrates the related techniques that deal with the containment and rewriting for XQuery, and an effective cache replacement strat-
In principle, the idea in a previous research work [CR00, QCR00] conducted by the author of this dissertation that employs an auxiliary index structure for facilitating the incremental maintenance of a simple XPath-based cache system can be incorporated into this XQuery-based semantic caching framework as well.

Figure 1.5: The Framework of Semantic Caching for XQuery

The dissertation work contributes to the two main aspects of an XQuery-based semantic caching system. The first aspect is about XQuery containment and rewriting, which is the fundamental issue underlying a semantic caching system. A variable-based containment mapping approach is proposed. Then the rewriting technique is presented based on the established variable mapping pairs. The second part is about semantic cache management, which deals with the limited cache space resource and dynamically maintains the cache utilization by adapting cache content to users query pattern. A fine-grained partial replacement strategy is presented. The ex-
perimental studies have shown that it helps to optimize the cache space utilization by improving the hit ratios and the average user response time over time. The key contributions of this work are given below.

1). A fragment of XQuery that allows nested FLWR expressions beyond XPath is defined for which the containment and rewriting problem is dealt with. This fragment contains the core XQuery features that are distinct from the XPath features but do not induce a prohibitive containment complexity.

2). The notion of variable essentiality is introduced and a procedure that identifies the essential variables in a query is proposed. Although the reduction of let-variables has been discussed in [M. 01, MFK01], our minimization of non-essential for-variables is novel.

3). An XQuery normalization approach is employed to transform an input query into a canonical form which facilitates the separation of pattern matching from result construction. This is a fundamental step in support of containment reasoning. Our normalization strategy hence targets a different goal than those in [M. 01, MFK01, W3C03d].

4). Two tree structures based on variables are designed for capturing the pattern matching and result construction semantics of a query respectively. These variable-based tree structures are crucial for exploiting the XPath containment technique in the process of solving the larger XQuery containment problem.

5). A containment checking technique is proposed for the defined XQuery
1.6. THE CONTRIBUTIONS

fragment. It consists of a series of steps, i.e., XQuery normalization and minimization, variable-tree based containment mapping, and an extended validation step tackling the control-flow-induced variable dependencies.

6). An equivalent logic rule-based containment mapping is also given. Formal notations are introduced to explain the important concepts and theorems are supported with proofs.

7). A rewriting technique is proposed for the defined XQuery fragment. It rewrites the variable bound expressions in the new query based on the generated maximal variable mapping pairs and the result structure of the cached query.

8). A semantic caching system called ACE-XQ is designed and implemented to realize the proposed XQuery containment and rewriting techniques.

9). Experimental studies are conducted to evaluate the query performance improvement by utilizing ACE-XQ.

10). A table structure is proposed to allow for user access statistics information to be recorded at the fine granularity of the XPath level.

11). A replacement strategy that utilizes the user access statistics recorded at the XPath level is proposed to perform a fine-grained region purging, as opposed to the strategy of completely replacing a cached query and its associated XML view content.
12). This partial replacement strategy is realized in the prototype system. Experimental studies are conducted and show that the performance of the cache system deployed with this partial replacement can be improved over that deployed with the total replacement.

1.7 Organization of this Dissertation

This dissertation is organized into five parts. Part I includes this introduction (Chapter 1). It reviews the related issues and solutions under the umbrella of semantic caching, and the challenges faced with the new data model and query languages in the web environment which provides the motivation for the work in this dissertation.

Part II describes the techniques for XQuery containment and rewriting. Chapter 2 first introduces this topic and then reviews the state-of-the-art research in this field. Chapter 3 defines the containment problem targeting a restricted XQuery fragment beyond XPath fragments, and then presents an approach for it. Chapter 4 describes the rewriting technique for XQuery based on the established containment mapping.

Part III addresses the issue of semantic cache replacement. Chapter 5 reviews the background on a variety of existing replacement strategies and discusses their advantages and disadvantages. Chapter 6 presents a fine-grained replacement strategy particularly tailored for the XQuery-based semantic caching system.

Part IV describes the implementation and evaluation of an XQuery-based semantic caching system called ACE-XQ. Chapter 7 delineates the
1.7. ORGANIZATION OF THIS DISSERTATION

overall architecture of the ACE-XQ system. Chapter 8 summarizes the experimental studies focusing on the approach validation and the replacement strategy evaluation.

Finally, Part V concludes the dissertation. In Chapter 9, we summarize our results and give the directions for future work.
Part II

Containment and Rewriting for XQuery
Chapter 2

Background on Query Containment and Rewriting

2.1 Preliminaries

In this section, we begin by introducing the terminologies used in the context of query containment and rewriting.

Definition 2.1 (Query Containment) The set of answers of query $Q$ on database $D$ are denoted $Q(D)$. A query $Q_2$ is said to be contained in a query $Q_1$, denoted $Q_2 \subseteq Q_1$, if $Q_2$ produces a subset of the answers of $Q_1$ for any given database $D$, i.e., $\forall D: Q_2(D) \subseteq Q_1(D)$.

The complexity of query containment has first been studied for the fundamental class of conjunctive queries which are also known as subsets of datalog. A datalog program is a set of datalog rules. A datalog rule has the form:
2.1. PRELIMINARIES

\[ q(\bar{X}) \leftarrow r_1(\bar{X}_1), \ldots, r_n(\bar{X}_n), \]

where \( q \) and \( r_1, \ldots, r_n \) are predicate (also called relation) names. The atom \( q(\bar{X}) \) is called the head of the rule, and the atoms \( r_i(\bar{X}_i) \), \( i = 1, \ldots, n \) are the subgoals in the body of the rule. \( \bar{X}_1, \ldots, \bar{X}_n \) denote tuple variables including constants. \( \bar{X} \) denotes the set of head variables that are projected out from the source relations in the rule body to compose the view relation. Multiple occurrences of the same variable in different subgoals imply a join predicate of the query. It is required that every rule is safe, i.e., every variable that appears in the head must also appear in the body.

A conjunctive query is a datalog program consisting of a single rule. A non-recursive datalog program is a set of datalog rules such that there exists an ordering \( R_1, \ldots, R_m \) of the rules so that the predicate name in the head of \( R_i \) does not occur in the body of rule \( R_j \) whenever \( j \leq i \). Such datalog programs can always be unfolded into a finite union of conjunctive queries. When comparison predicates \( \neq, <, >, \leq, \geq \) are considered in queries and views, it then requires that for each datalog rule, if a variable \( X \) appears in a subgoal with a comparison predicate, then \( X \) must also appear in a relational subgoal in the body of the rule.

An important theorem about the query containment for conjunctive queries given in [CM77] is listed next.

**Theorem 2.1 (Containment Mapping)** A conjunctive query \( Q_2 \) is contained in another query \( Q_1 \), denoted by \( Q_2 \subseteq Q_1 \), if and only if there is a containment mapping from \( Q_1 \) to \( Q_2 \). The containment mapping maps variables of \( Q_1 \) to those of \( Q_2 \) such that every subgoal in \( Q_1 \) is mapped to a subgoal in \( Q_2 \).
2.1. PRELIMINARIES

An important implication of this one-to-one containment mapping is that a join variable appearing twice in different subgoals of \( Q_1 \) must correspondingly have coherent mappings to one join variable in \( Q_2 \). An intuitive explanation of the containment mapping is that an isomorphism between the atomic predicates of \( Q_1 \) and a subset of the atomic predicates of \( Q_2 \) can guarantee the answer set \( Q_2 \) is subsumed by that of \( Q_1 \).

**Containment Mapping Example.** For example, given two conjunctive queries \( Q_1 \) and \( Q_2 \) as below, the containment mapping from \( Q_1 \) to \( Q_2 \) is that \( \{ X \rightarrow X', Y \rightarrow Y', W \rightarrow W', Z \rightarrow W' \} \).

\[
\begin{align*}
Q1: & \quad p(X,Y) \rightarrow r(X,W), b(W,Z), r(Z,Y) \\
Q2: & \quad p(X',Y') \rightarrow r(X',W'), b(W',W'), r(W',Y')
\end{align*}
\]

However, there is no containment mapping from \( Q_2 \) to \( Q_1 \) because for \( b(W',W') \) in \( Q_2 \), its only possible target in \( Q_1 \) is \( b(W,Z) \). But we cannot have a mapping \( W' \rightarrow W \) and \( W' \rightarrow Z \), since one variable cannot be mapped to two different variables.

Containment of conjunctive queries is NP-complete [CM77]. Containment of a conjunctive query in a non-recursive query is also NP-complete, while containment of two non-recursive queries is \( \Pi_2^p \)-complete [CV92]. Containment is decidable when at least one of the two queries is non-recursive [CV92], while containment of arbitrary datalog programs is undecidable [O. 93]. Furthermore, containment of conjunctive queries with arithmetic comparisons in the form of \( A_1 \theta A_2 \) (operator \( \theta \) is \( \leq, \leq, =, \geq, \geq \)) is \( \Pi_2^p \)-complete [F. 02].
Graph Homomorphism and Simulation. Besides the logic-based formalism, graph theory is also applied as a tool to study query containment.

Definition 2.2 (Graph Homomorphism) Suppose $G_1$ and $G_2$ are two directed labelled graphs. A simulation is a relation $R$ between nodes in $G_1$ and $G_2$, if $R(x_1, x_2)$ and $(x_1, a, y_1) \in G_1$, i.e., $y_1$ is a child of $x_1$ and its label is $a$, then exists $(x_2, a, y_2) \in G_2$ such that $R(y_1, y_2)$. Such a relation $R$ is also referred to as a graph homomorphism function.

It is shown in [CR97] that conjunctive query containment, although a NP-complete problem in general, can be solved in polynomial time, if the query is acyclic. Their idea generalizes the notion of acyclicity using a term called query width that is derived from graph theory. [LS97a] gives a containment algorithm for complex objects by relating the problem with graph simulation. The problem of query containment for complex objects is shown to be decidable by the approach of graph simulation. Also, it is shown that checking the equivalence of conjunctive queries for complex objects with grouping and aggregates is NP-complete.

The connection between query containment and rewriting is also studied in [CR97]. It is shown the guess of a rewriting can be extended to a guess for containment mappings and for showing the equivalence of the rewriting and the query, the latter of which has the complexity of NP-complete. However, there are many polynomial-time cases of the rewriting problems in practice, analogous to those for query containment.

Definition 2.3 (Maximal Rewriting) Given a view query $Q_1$ and a new query $Q_2$, a query $Q_2'$ is a contained rewriting of query $Q_2$ using $Q_1$ if $Q_2'$ uses only
2.2. CHALLENGES FROM XML AND ITS QUERY LANGUAGES

$Q_1$ and $Q_2' \subseteq Q_2$. $Q_2'$ becomes a maximally-contained rewriting if (1) $Q_2'$ is a contained rewriting of $Q_2$, and (2) there is no contained rewriting $Q_2''$ of $Q_2$ such that $Q_2' \subset Q_2''$. If a contained rewriting $Q_2'$ of $Q_2$ satisfies $Q_2' \equiv Q_2$, then it is an equivalent rewriting of $Q_2$.

Researchers have considered the algorithms for answering queries using views in the context of several query languages. There has been extensive work on finding the equivalent rewritings for SQL queries (e.g., [YL87, SJGP90, TS94, CR94, CKPS95, LMSS95]). Yang and Larson [LY85, YL87] considered the problem of finding rewritings for SPJ queries using SPJ views. There is another work considering queries and views with binding patterns [RSU95], with grouping and aggregation [GHQ95, SDJL96, GBLP96], and with multi-block queries [ZCL00].

2.2 Challenges from XML and its Query Languages

With the advent of semi-structured data on the Web, interest was resurrected in the containment and rewriting problem for queries that target such schema-less data. We first introduce this new Web data model, XML, and its query languages. Then we give a survey on the complexity results for the containment of XPath queries, a simple but common sub-language of many existing XML query languages.

2.2.1 Background on XML

Extensible Markup Language (XML) [W3C98] is a hierarchical data format for information representation and exchange on the Web. An XML docu-
ment presents a nested element structure with each element containing an
ordered list of attributes and/or sub-elements. The example XML docu-
ment in Figure 2.1 shows that it contains information about a book. In this
example, there is a book element that has the three sub-elements: title, author
and price. This book element also has a year attribute with value 1990. The
author element further contains a last element and a first element, which
encapsulate the values of the last and first names respectively.

```xml
<bib>
  <book year = "1990">
    <title>Data Warehousing Technologies</title>
    <author>
      <last>Daya</last>
      <first>M.</first>
    </author>
    <price>59.99</price>
  </book>
  <book year = "1992">
    <title>TCP/IP Illustrated</title>
    <author>
      <last>Stevens</last>
      <first>W.</first>
    </author>
    <publisher>Addison – Wesley</publisher>
    <price>39.99</price>
  </book>
</bib>
```

Figure 2.1: The Example XML Document: bib.xml

DTDs (Document Type Definitions) [W3Ca] for XML, which are inher-
ited from the schema mechanism for SGML (Standard Generalized Markup
Language) [W3Cb], can be used to define content models (the valid order
and nesting of elements) and to a limited extent the data types of attributes.

Figure 2.2 shows the DTD for the example XML document in Figure 2.1.
The example DTD describes the possible arrangement of tags in a valid
XML document.

```xml
<?xml version = "1.0" ?>
<!DOCTYPE bib [
<!ELEMENT bib (books*)>
<!ELEMENT book (title, (author + |editor+), publisher?, price)>
<!ATTLIST book year CDATA #REQUIRED>
<!ELEMENT author (last, first)>
<!ELEMENT editor (last, first, affiliation)>
<!ELEMENT title (#PCDATA)>
<!ELEMENT last (#PCDATA)>
<!ELEMENT first (#PCDATA)>
<!ELEMENT affiliation (#PCDATA)>
<!ELEMENT publisher (#PCDATA)>
<!ELEMENT price (#PCDATA)>
]
```

Figure 2.2: The Example DTD: bib.dtd

### 2.2.2 Background on XML Query Languages

A crucial enabling technique of many XML applications is the XML query
languages. In the last few years, there has been a great deal of research
[AQM+97, CJS99, P.00, FFK+98] into semi-structured query languages to
enable the execution of database-style queries over these XML files. How-
ever, some of these languages have not initially been designed to query
XML documents but rather each has its own system proprietary data model.
2.2. CHALLENGES FROM XML AND ITS QUERY LANGUAGES

For example, Lore bears the flavor of OQL and extends it using path-expression to query data modelled in OEM [PMW95]. StruQL [FFK+98] models the data as a graph and applies a pattern-matching query paradigm to achieve some graph transformation. YATL (YAT Language) [CJS99] is also based on a pattern-matching paradigm adopting operators from object-oriented algebras as well as Bind and Tree operators. Even though all these language proposals are claimed to be extensible to query XML documents, there still exist mismatches between them and the XML query data model [W3C00]. Hence these languages are not that suitable for querying the Internet populated with XML data.

More recently, several XML-oriented query languages including XML-QL [DFF+99b] and XQL [RLS] (from Microsoft) have been proposed. Similar to Lore [QWG+96] and UnQL [P.00], they all have a notion of path expressions for navigating the nested structure of XML. For example, XML-QL uses a nested XML-like structure to specify the parts of a document to be selected and specifies the structure of the result XML document using a result template. Introduced first at the W3C’s conference on XML query languages in 1998, XQL has since been implemented by several large IT vendors. It now can be seen as the precursor and the core part of XPath [W3C03b]. However, neither XQL nor XPath is a full-blown query language. They lack features like variable binding, joins across documents and restructuring.

Introduced in 2000, Quilt [CRF00] quickly gained intensive attention as an expressive XML query language featuring nested FLWR expressions, a modernized SQL-ish construct, on top of XPath. Shortly after, W3C pro-
posed a language called XQuery [W3C03c], whose syntax bears a great similarity with Quilt, as the standard query language for XML documents. XQuery borrows its features from many languages including XPath, XQL, XML-QL and even SQL. For example, it exploits a powerful path expression syntax from XPath and XQL which enables navigations within a hierarchical document, selecting a set of nodes that satisfy a tree pattern. From XML-QL it draws the notion of bound variables and a versatile syntax that can generate an output document of arbitrary structure. Therefore, XQuery has been designed to provide necessary functionalities that are flexible enough to query a broad spectrum of XML information sources including both documents and databases. In this sense, XQuery can help to realize the potential of XML as a universal web data exchange medium.

A FLWR expression is one important query construct in XQuery. The for clause specifies bindings (i.e., the objects selected by the bound XPath expression) to be associated with a variable, the let clause binds a variable to an aggregated sequence of objects, the where clause specifies predicates, and the return clause defines how the result document is constructed with the obtained variable bindings.

2.3 Background on XPath Containment

For the topic of query containment and rewriting, the focus of many research papers is on the complexity of the containment problem for regular-path-expression based queries and various fragments of XPath.

In [CGL99], the authors consider the problem of rewriting a regular
path query using a set of regular path views. They show that the problem is in doubly EXPTIME and that the problem of checking whether the rewriting is an equivalent one is in 2EXPSPACE. Later the results are extended in [CGL99, CGLV00] to path expressions with inverse operators, allowing both forward and backward traversals in a graph.

More specifically, XPath is a common sublanguage of many XML query languages for which it enables navigations within a document and provides the basic node selecting mechanism. Containment of XPath queries can be used to show equivalence of queries for optimization, to reuse stored queries or materialized views. Furthermore, containment of XPath queries is the basis for solving the containment for a various XML query languages that employ XPath. Therefore, many research efforts have been recently attracted to the topic of XPath containment [AYCLS01, PV99a, Woo01, G. 02b, FT03]. Most of these papers have studied different fragments of XPath, by including different XPath constructs. Below we introduce different fragments of XPath and survey the containment complexity results for each of them.

The syntax of an XPath query $P$ in the fragment $XP\{[\cdot,\cdot,\cdot,\cdot]\}$ is given by the grammar: $P ::= P/P \mid P//P \mid P[P] \mid P|P | * | n$, where $n$ denotes an element name. In this fragment, node test, composition of location steps, predicate “$]$”, wildcard “$*$”, descendant axis “$//$”, and disjunctive symbol “$|$” are all included. The commonly studied XPath fragments include $XP\{[\cdot,\cdot,\cdot,\cdot]\}$ and those composed of a variety of subsets of the features allowed in $XP\{[\cdot,\cdot,\cdot,\cdot]\}$.

In [G. 02b], an XPath expression $P$ is represented as an unordered tree
pattern over the alphabet of $P$. As depicted in Figure 2.3, navigation steps specified by the labelled nodes in XPath are marked as nodes in the tree pattern, child axis “/” are marked as edges, and descendant “//” are marked as edges with double lines. For a tree pattern $P$, the *arity* of the result tuple, i.e., the number of returned element types, is called the arity of $P$.

![Figure 2.3: The Framework of Semantic Caching for XQuery](image)

With XPath expressions represented in the form of tree patterns, graph-theoretic algorithms can be used to reformulate and solve XPath query evaluation and containment problems. For example, given an XML document tree $T$, the query evaluation of XPath $P$ can be seen as a tree embedding process from $P$ to $T$ with the considerations of the navigational semantics of “/”, “//”, “*” and “[ ]”. The definition of *containment* between two XPaths $P$ and $P'$ (i.e., $P \subseteq P'$) is that, for every XML document, the result of applying XPath $P$ will be contained in the result of applying XPath $P'$.\(^1\)

\(^1\)The result of either $P$ or $P'$ would be a set of nodes without considering the order.
For instance, suppose two XPath queries $P$ and $P'$ are $/\text{bib/book}[\text{price}]/\text{title}$ and $// * /\text{title}$ respectively. They both query the same source XML document tree $T$ as shown in Figure 2.4. The evaluation process of each XPath query can be seen as an embedding process from the query to $T$. The result set $P(T)$ derived from embedding $P$ into $T$ is $\{6\}$ (the node marked by the id 6 in $T$) while $P'(T)$ is $\{6, 8\}$. According to the meaning of XPath containment, $P(T) \subseteq P'(T)$ and for any possible XML tree $T$, hence $P \subseteq P'$, i.e., $/\text{bib/book}[\text{price}]/\text{title} \subseteq // * /\text{title}$.

### 2.3.1 Complexity Results for XPath Containment

The start-of-the-art theoretical research on the containment complexity for different XPath fragments is summarized in Table 2.1. The complexity results are for the cases when DTD constraints are not considered.

With the presence of DTD, certain constraints such as child constraints,
2.3. BACKGROUND ON XPATH CONTAINMENT

<table>
<thead>
<tr>
<th>/</th>
<th>//</th>
<th>[ ]</th>
<th>*</th>
<th>Complexity</th>
<th>Reference</th>
</tr>
</thead>
<tbody>
<tr>
<td>+</td>
<td>+</td>
<td>+</td>
<td></td>
<td>PTIME</td>
<td>Amer-Yahia et al, 2001</td>
</tr>
<tr>
<td>+</td>
<td>+</td>
<td>+</td>
<td></td>
<td>PTIME</td>
<td>Wood, 2001</td>
</tr>
<tr>
<td>+</td>
<td>+</td>
<td>+</td>
<td></td>
<td>PTIME</td>
<td>Neven &amp; Schwentick, 2003</td>
</tr>
<tr>
<td>+</td>
<td>+</td>
<td>+</td>
<td>+</td>
<td>coNP-complete</td>
<td>Miklau &amp; Suciu, 2002</td>
</tr>
<tr>
<td>+</td>
<td>+</td>
<td>+</td>
<td></td>
<td>coNP-complete</td>
<td>Miklau &amp; Suciu, 2002</td>
</tr>
<tr>
<td>+</td>
<td>+</td>
<td>+</td>
<td></td>
<td>coNP-complete</td>
<td>Miklau &amp; Suciu, 2002</td>
</tr>
<tr>
<td>+</td>
<td>+</td>
<td>+</td>
<td>+</td>
<td>coNP-complete</td>
<td>Neven &amp; Schwentick, 2003</td>
</tr>
</tbody>
</table>

Table 2.1: Containment Complexity for Different XPath Fragments Without DTD Constraints

Sibling constraints, and functional constraints may be implied. For example, the declaration of the element type `author` in Figure 2.2 is `<ELEMENT author (last, first)>`. It implies both the child constraints, e.g., “every author must have a last name”, and the sibling constraints like “every last name of an author must have a first name also”. Under the DTD constraints, the complexity result for the containment of different XPath fragments is given in Table 2.2.

<table>
<thead>
<tr>
<th>/</th>
<th>//</th>
<th>[ ]</th>
<th>*</th>
<th>Complexity</th>
<th>Reference</th>
</tr>
</thead>
<tbody>
<tr>
<td>+</td>
<td>+</td>
<td>+</td>
<td></td>
<td>PTIME</td>
<td>Neven &amp; Schwentick, 2003</td>
</tr>
<tr>
<td>+</td>
<td>+</td>
<td>+</td>
<td></td>
<td>coNP-complete</td>
<td>Wood, 2001</td>
</tr>
<tr>
<td></td>
<td>+</td>
<td>+</td>
<td></td>
<td>coNP-hard</td>
<td>Neven &amp; Schwentick, 2003</td>
</tr>
<tr>
<td>+</td>
<td>+</td>
<td>+</td>
<td></td>
<td>EXPTIME-complete</td>
<td>Neven &amp; Schwentick, 2003</td>
</tr>
<tr>
<td>+</td>
<td>+</td>
<td>+</td>
<td></td>
<td>EXPTIME-complete</td>
<td>Neven &amp; Schwentick, 2003</td>
</tr>
<tr>
<td>+</td>
<td>+</td>
<td>+</td>
<td></td>
<td>EXPTIME-complete</td>
<td>Neven &amp; Schwentick, 2003</td>
</tr>
</tbody>
</table>

Table 2.2: Containment Complexity for Different XPath Fragments With DTD Constraints
2.3. BACKGROUND ON XPATH CONTAINMENT

From Table 2.2, we see that deciding containment under DTDs is coNP-complete even for the queries in $XP^{[[1]]}$. Containment is undecidable when the XPath fragment includes $XP^{[[1],[*]/1]}$ along with any variable binding and equality testing. In [G. 02b], it is proven that the containment is decidable, actually in coNP-complete for $XP^{[[1],[*,/]/]}$. Also, subclasses of $XP^{[[1]]}$ queries and subclasses of DTDs for which containment is in PTIME are further studied in [Woo03]. It is found that when a DTD $D$ is duplicate-free, namely each element name $n$ appears at most once in each content model, the containment for queries in $XP^{[[1]]}$ can be decided in PTIME.

2.3.2 XPath Containment Algorithm

As shown in Table 2.1, the containment complexity for $XP^{[[1]/1]}$, $XP^{[*]/1}$ and $XP^{[[1],[*/]}$ are all in PTIME, while the complexity for $XP^{[[1],[*,/]}$ is coNP-complete. [G. 02b] shows that finding a containment checking algorithm that will be both efficient and complete for this XPath fragment is difficult. However, an efficient algorithm which is sound but not complete is given in [G. 02b] for determining if $P \subseteq P'$ ($P$ and $P'$ are two queries within the $XP^{[[1],[*,/]}$ fragment) by utilizing the tree homomorphism between their tree pattern representations, as shown in Figure 2.5.

In this tree homomorphism algorithm, a mapping $h$ from the nodes in $P'$ to those in $P$ is established if the following conditions are satisfied:

1). Root preserving, i.e., the entry points of $P$ and $P'$ are the same. In other words, if $P$ and $P'$ specify navigations starting from the root elements of different XML documents, then the mapping function $h$
2.3. BACKGROUND ON XPATH CONTAINMENT

Homomorphism for XPath tree pattern:
- node labeled with n maps to a node labeled n
- node labeled with * maps to any node
- || requires node mapped to a descendant node
- | requires node mapped to a child node

Figure 2.5: Tree Pattern Homomorphism Algorithm for XPath Containment

returns an empty set;

2). For each node $x \in \text{Nodes}(P')$, $\text{Label}(x) = *$ or $\text{Label}(x) = \text{Label}(h(x))$;

3). Child and descendant relations preserving. That is, for each $x, y \in \text{Nodes}(P')$, if $(x, y)$ is a child edge “/” in $P'$ then $(h(x), h(y))$ is a child edge in $P$ as well, and if $(x, y)$ is a descendant edge “//” in $P'$ then $h(y)$ is a descendant of $h(x)$.

It is shown in [G. 02b] that the existence of such a tree homomorphism is a sufficient condition for determining the containment of $XP[[\cdot,\cdot]/]$ patterns. The complexity of finding such a homomorphism is $O(|P|^2 \cdot |P'|)$. In the cases of $XP[[\cdot,\cdot]/]$ and $XP[[\cdot,\cdot//]]$, the existence of a homomorphism is also a necessary condition for containment, whereas it is not necessary for the containment of $XP[[\cdot,\cdot//]]$. 
Chapter 3

XQuery Containment

3.1 Towards XQuery Containment

In the XML setting, although XPath containment has recently received much attention, the containment problem so far remains unexplored for more comprehensive XML query languages such as XQuery. Compared to XPath, XQuery is more powerful and it can handle sophisticated query scenarios in which users may want to select and join XML data across diverse sources, extract elements from documents while preserving their original hierarchy, and re-construct the output.

XQuery has rapidly gained a growing recognition. It adopts the FLWR expressions for specifying variable bindings, performing conditional selections and result restructuring in a flexible manner. These features help to enhance the query expressiveness. On the other hand, they impose significant challenges to the containment problem, even if the XQuery language in consideration is already restricted to utilize only the well-behaved XPath
fragments for which the containment has been determined to be decidable or even in PTIME.

3.1.1 Motivating Example of XQuery Containment

In Figure 3.1, we show two example queries Q1 and Q2. Both queries employ nested FLWR expressions that extract their respective data bindings from the XML document books.xml\(^1\) and utilize them to construct their respective results.

\begin{verbatim}
<Q1res>
  {let $d := doc("books.xml")
   for $b in (for $b1 in $d return $b1/book)
     where $b/price < 50
     return <entry> {for $t in $b/title
       return {$t }
     }
     {for $a in $b/author
       return {$a/last }
     }
   </entry>}
</Q1res>

<Q2res>
  {for $b in doc("books.xml")/bib/book
   where $b/price < 40
   return <item> {for $t in $b/title, $a in $b/author
     return <pair> {$t } {$a/last }
   }
   </pair> }
</Q2res>
\end{verbatim}

Figure 3.1: Example Queries Q1 and Q2

As shown in the example queries, XPath expressions serve as a basic query construct of XQuery for selecting nodes to be associated with variables or to be returned in the result. Clearly, the result on XPath contain-

\(^1\)For brevity, the long URL for the source XML document referred to by both example queries is omitted.
3.1. TOWARDS XQUERY CONTAINMENT

ment provides the foundation for solving XQuery containment since it can resolve the containment relationship between two node sets which are derived from the same starting node via different XPath expressions. But the question is if we can represent XQuery using a similar tree pattern as that for XPath, and then reduce XQuery containment to the problem of tree homomorphism? Can we simply ignore the XQuery features beyond XPath such as variables, element constructors, differently nested structures, etc.?

**Trials and Errors.** To answer the above questions, we begin with some attempts to represent the example XQuery \( Q_1 \) using tree patterns.

In our first trial, all the return expressions are collected and their prefixing variables are substituted with their bound expressions. This way, we obtain a list of absolute XPath expressions each indicating the complete navigation steps required to be taken to derive the corresponding return node bindings from the document root element. Since each XPath expression can be represented as a unary tree pattern, \( Q_1 \) is represented by a list of unary pattern trees, as shown in Figure 3.2.

Figure 3.2: First Trial of Representing XQuery by Tree Patterns
However, this trial 1 representation cannot capture the implied conjunctive pattern constraints on the bindings of $b/title$ and $b/author/last$ which share the common variable prefix $b$. That is, the semantics of returning title and author’s last elements while preserving their dependencies on the same context book element is lost in this unary pattern tree list. Instead, it represents returning title and author’s last elements document-wise regardless of their hierarchical context.

In the second trial, we build a large tree pattern by appending the small tree pattern of an XPath expression $P_2$ to that of $P_1$ if $P_1$ defines the variable which is exactly the prefix variable of $P_2$. Such a concatenated tree pattern is shown in Figure 3.3, where the returning nodes are again all marked by the double circle.

Figure 3.3: Second Trial of Representing XQuery by Tree Patterns

In contrast to the trial 1 representation, this $k$-ary pattern tree representation of $Q1$ is capable of capturing conjunctive pattern constraints. It however goes to the other extreme of forcing a full tree pattern matching for all the nodes involved. For instance, a book element can be bounded to the
3.1. TOWARDS XQUERY CONTAINMENT

book node in this large pattern tree only if it has all three types of children, i.e., price, title, and author. In addition, one of its author children elements needs to have a last child. This means that the variable bound expression for $b$ becomes “//book[price][title][author/last]”. This is unnecessarily more restricted than the original query.

Therefore, both tree pattern representations fail to represent XQuery semantics in a precise fashion. The reason for the trial 1 tree pattern is the lack of representing necessary variable dependencies, while the trial 2 tree pattern fails due to no distinction of contexts within which different constraints take effect. We thus set out to find some alternative suitable representation of XQuery semantics.

3.1.2 Problem Requirements in XQuery Containment.

We have shown in Figures 3.2 and 3.3 that some trials of representing XQuery using tree patterns that are straightforwardly extended from the XPath tree pattern fail to capture the XQuery specific semantics. The difficulties in bridging the gap between the containment for XPath and for XQuery are analyzed below.

First, XPath expressions used in XQuery are not always absolute path expressions, but may be relative path expressions prefixed by variables. In such cases, their evaluations are not started from the root element of the input document, but rather are based on the derived data bindings of their prefixing variables. Therefore, XPath containment cannot be applied on stand-alone XPath expressions. Consideration about the order of their evaluation implied by the prefixing variables needs to be incorporated.
Second, XPath expressions are often used together with other query constructs such as for or return clauses. The reasoning of XPath containment thus must distinguish XPath expressions used in different contexts.

Third, query containment is essentially result inclusion. The result construction in XQuery is however a stepwise process composed of possibly several levels of node selections and corresponding element constructions. Thus, containment reasoning for XQuery likely may involve multiple steps of containment checking between the mapping of data pieces.

Fourth, variables are flexibly used in XQuery. The data bindings of a particular variable may or may not be utilized for constructing part of the result. Also, a variable can be imposed with different where conditions within its different scopes of life (nested contexts and clauses). A precise reasoning of XQuery containment needs to analyze all these uses of variables.

3.1.3 XQuery Containment Approach Overview

In this dissertation, we first define a fragment of XQuery capturing nested FLWR expressions beyond XPath for which we address the containment problem. Then we propose a solution for XQuery containment based on the following three observations.

1). Pattern matching and result construction are the two semantic components of XQuery connected by the use of variables. In other words, a variable is specified by a for clause to be associated with some data nodes, while its occurrences in a return clause indicate that the corre-
sponding data bindings are used for constructing the result.

2). Dependencies exist among the specified variables based on their bound expressions. Such variable dependencies imply that the hierarchical relationships among their respective data bindings are to be preserved during the query evaluation.

3). For determining query containment, it is necessary to reason about the containment of the preserved data bindings which compose the query result.

The key idea of the approach is to reduce the problem of XQuery containment to the classic tree inclusion problem. We first define a fragment of XQuery that allows nested FLWR expressions beyond XPath to be considered for the containment problem. To determine the XQuery containment, e.g., \( Q_2 \subseteq Q_1 \), we conduct a containment checking process as depicted in Figure 3.4. The main steps involved in this process are highlighted next.

**Step 1).** Based on the notion of variable essentiality to be introduced, utilize a minimization procedure to obtain the essential variables in a query.

**Step 2).** Employ an XQuery normalization approach to transform an input query into a canonical form which facilitates the separation of pattern matching from result construction. This canonical form contains only essential variables. This is an important step in support of XQuery containment reasoning. Our normalization strategy hence targets a different goal than those in [M. 01, MFK01, W3C03d].
Step 3). Decompose the query and separately represent the pattern matching and result construction semantics in two proposed tree structures called VarTree and TagTree respectively.

Step 4). Conduct a tree-embedding-like mapping for the VarTree representations of two given queries with the considerations of particular VarTree characteristics. XPath containment is utilized in this mapping process for determining each mapping between two VarTree nodes.

Step 5). Extend the VarTree-based containment mapping to incorporate a validation of the established containment mapping with the consideration of the control-flow induced variable dependencies.

To validate our XQuery containment approach, we provide definitions, theorems and proofs for the important concepts being utilized. In Chap-
3.2. AN XQUERY FRAGMENT BEYOND XPATH

In this section, we provide a concise definition, i.e., the BNF rules shown in Figure 3.5, for the core fragment of the XQuery language handled in this dissertation.

\[
\begin{align*}
    e & ::= v \mid (v)?xp \mid flwr \mid re \\
    flwr & ::= \text{for-let where-return} \\
    \text{for-let} & ::= \text{for } [v] \text{ in } [e] \mid \text{let } [v]:= [e] \\
    \text{where-return} & ::= (\text{where } c)? \text{ return } e \\
    re & ::= \{e_1\}..\{e_k\} \mid <\text{tag}> e <\text{/tag}> \\
\end{align*}
\]

Figure 3.5: BNF Rules for XQuery Fragment

In this set of BNF rules, \(xp\) denotes a path expression, \(\{e_i\}(i = 1..k)\)
3.2. AN XQUERY FRAGMENT BEYOND XPATH

denotes a sequence of enclosed expressions, \( v \) is a variable, and \( c \) is a where condition. We use \( [v] \) to represent a sequence of variables defined in the given for-clause, and \( [e] \) to denote their corresponding bound expressions.

The XQuery fragment defined above is similar to the minimal XQuery defined in [DTCO03]. It is very close to the XQuery core used in the XQuery formal semantics proposal [W3C03d], being slightly more tolerant in allowing multiple variables to be defined in a single for-clause. This fragment allows us to focus on the core features of XQuery such as the existential qualifier “some”, the conjunction connective “and”, and the conditions involving only semi-interval arithmetic comparisons (i.e., only \( =, \geq \) or \( \leq \)) with constants.

In addition, our XQuery fragment contains only the well-behaved XPath fragment (i.e., the core XPath in [GK02]) that has been identified to have desirable features such as PTIME containment complexity [G.02b]. To avoid recursive computations, we allow only the XPath expressions specifying downwards navigations. This XPath fragment is denoted as \( XP_{//\text{.}.*[\]} \), which is free of disjunction, negation and recursion.

However, we ignore other features such as user-defined functions, aggregations, arithmetic, string, namespace and position related operations. We do not deal with type switch or casting or the if-else constructs.

3.2.1 The Formal Semantics of the XQuery Fragment

In [W3C03d], formal notations such as expressions, XML values and environments are used to reflect the meaning of the XQuery language with mathematical rigor. In particular, the dynamic semantics relate XQuery
expressions to the XML values to which they evaluate. This provides a reference for XQuery implementations.

As a functional language, XQuery utilizes the important concept of dynamic environment, denoted dynEnv, as a dictionary structure that maps variables to data bindings [W3C03d]. That is, \( \text{dynEnv} = \mathcal{M}(\text{VarRef} \Rightarrow \text{Value}) \). The symbol \( \Rightarrow \) means a judgment and is read as "yields". VarRef denotes a set of variable references and Value represents the sets of data bindings associated with VarRef.

In an XQuery, each expression \( e \) is evaluated in a specific context where the information available is the mappings in dynEnv at that time. This is captured by the rule \( \text{dynEnv} \vdash e \Rightarrow \text{Value} \). The symbol \( \vdash \) indicates that the right hand side evaluation of expression \( e \) yielding Value is performed in the dynamic environment \( \text{dynEnv} \).

For any \( v \) occurring in \( e \), its evaluation is a "looking up" of \( v \)'s bindings in \( \text{dynEnv} \), denoted \( \text{dynEnv}.\text{varValue}(v) \). If \( e \) is a for clause, its evaluation may result in a list of new mappings, which will be added into dynEnv to provide the context for evaluating the next expression in order. Along with the evaluation of a query, the mappings held in dynEnv evolve. For example, new mappings are added into dynEnv when new variables in a nested block are evaluated to be associated with their corresponding data bindings. The mapping Value of a \( v \) in dynEnv can be overwritten if \( v \) is asserted with new conditions in a nested block which results in different bindings. When the query evaluation exits a nested query block, the mappings established in this block are removed from dynEnv. This explains why dynEnv is called "dynamic" and why variables have their life scopes.
The formal semantics of our XQuery fragment, a restricted subset of
the full XQuery [W3C03c], corresponds to the formal semantics for some of
the primary expressions (i.e., literals, variable references, parenthesized ex-
pressions, and context item expressions, referred to in Sections 4.1.1~4.1.4
in [W3C03d]), path expressions (see Section 4.2 in [W3C03d]), sequence
expressions (see Section 4.3 in [W3C03d]), partial comparison expressions
(i.e., value comparisons and general comparisons, referred to in Sections
4.5.1~4.5.2 in [W3C03d]), “and” logical expressions (see Section 4.6 in [W3C03d]),
direct element constructors (see Section 4.7.1 in [W3C03d]), partial FLWR
expressions (i.e., for, let, return expressions referred to in Sections 4.8.2~4.8.4
in [W3C03d]), and quantitative qualifier expressions (see Section 4.11 in
[W3C03d]).

In brief, the core part of the formal semantics of our XQuery fragment
can be captured by the following rules extracted from [W3C03d]:

(1) $\text{dynEnv} \vdash \text{VarRef} \Rightarrow \text{dynEnv}.\text{varValue}(\text{VarRef}) = \text{Value}$
(2) $\text{dynEnv} \vdash \text{e} \Rightarrow \text{Value}$
(3) $\text{dynEnv} \vdash \text{for VarRef in e} \Rightarrow \text{dynEnv + M(VarRef \Rightarrow Value)}$
(4) $\text{dynEnv} \vdash \text{e1, e2} \Rightarrow \text{Value}\text{e1, Value}\text{e2}$

Rule (1) captures the semantics of “looking up” variable bindings in
$\text{dynEnv}$, denoted by $\text{dynEnv}.\text{varValue(VarRef)}$. That is, $\text{dynEnv}$ maps a
variable name (i.e., $\text{VarRef}$) onto the variable’s current value (i.e., $\text{Value}$).
Rule (2) captures the semantics of an XPath or a return expression. In Rule
(3), $\text{dynEnv}$ is updated by adding the mappings $\text{M(VarRef \Rightarrow Value)}$
yielded from the for clause. Rule (4) indicates that each expression in the
sequence is evaluated consecutively and the resulting values are concatenated into one sequence. We can see the evaluation of a query by applying these rules as an incremental dynEnv building process.

The XQuery fragment we focus on allows great flexibility in employing a variety of query constructs. For instance, XPath expressions help to select nodes from the source document, and nested FLWR expressions make it possible to specify variable bindings, impose conditions and construct the nested result structure. The features considered in this XQuery fragment all have some affordable computational complexities with respect to the containment problem.

The arbitrary nesting of FLWR expressions is more flexible than that of select-from-where clauses in SQL. For example, the inner SQL sub-queries produce the value domain that can serve as the condition constraints for the outer sub-query, while only the outermost select-clause in an SQL query can project attributes as output. In contrast, the construction of new elements and the return of data bindings can occur at any nesting level of the FLWR expressions. Moreover, the nested SFW-clauses in an SQL query can be de-correlated to be represented as a regular join operation in a straightforward way. It would be difficult to apply such a de-correlation approach directly to the nested FLWR expressions since this may distort the return semantics and cause an undesired unnesting of the variable bindings.
3.3 Variables in XQuery

Many expression-based languages contain the notion of a variable, which can be used to save multiple evaluations of the same expression occurring at many places. Variables in XQuery are specified by the for and let clauses to accommodate the intermediate data bindings that are derived by the respective bound expressions. By the use of variables, their bindings can serve as the context for deriving other bindings, for invoking new element constructions, or for providing handles for their descendants to be accessed and returned in the result. In short, variables serve as an important medium that stores the intermediate data bindings for future reuse.

3.3.1 Variable Binding Dependency

Given a query, if more than one variable is defined and they are dependent on each other based on their definitions, then the original hierarchies among their corresponding data bindings are preserved. Below we define this notion of variable dependencies.

**Definition 3.1 (Variable Binding Dependency)** Suppose a variable $v_j$ in a query is defined based on a previously defined variable $v_i$, e.g., “for $v_j$ in $v_i p_j$”, where $p_j$ is the relative XPath expression used for deriving the bindings of $v_j$ from those of $v_i$. We call this dependency of $v_j$ on $v_i$ a **variable binding dependency** and denote it as $v_i \xrightarrow{p_j} v_j$. $v_i$ is referred to as the parent variable in this dependency and $v_j$ as the child variable.
A variable dependency $v_i \xrightarrow{p_j} v_j$ means that the variable bound expression for $v_j$ can be evaluated only in the context when the mapping $\mathcal{M}(v_i \Rightarrow Value_i)$ is already available in $dynEnv$ (see Rule (3) in Section 3.2.1). This is because the bindings $Value_i$ of $v_i$ need to be looked up to derive the new mapping $\mathcal{M}'(v_j \Rightarrow Value_j)$. The latter is then added into $dynEnv$. Given a data binding $o$ in the binding set $Value_i$, the set of data objects that are rooted at $o$ and navigated to via $p_j$ can be identified and added in $Value_j$. No two data objects in $Value_i$ have an overlapping descendant object set in $Value_j$.

For example, in the example query $Q1$ shown in Figure 3.1, variable $\$b$ is defined by the XPath expression navigating from the root to the $book$ elements. Later the variable $\$t$ is defined based on $\$b$. Then the hierarchies between the $book$ elements and their respective children $title$ elements are preserved by the bindings of $\$b$ and those of $\$t$, similarly the hierarchies between the bindings of $\$b$ and those of $\$a$ are preserved. This way, the sibling $title$-$author$ association derived from their common ascendant $book$ element is also preserved. Such associations are reflected by the $title$ and author’s $last$ elements contained within each new $entry$ element being constructed correspondingly for each $book$ element binding.

For a given query $Q$, we call every direct parent-child variable binding dependency of $Q$ a base variable binding dependency. From the set of such base dependencies, the ancestor-descendant dependencies can be derived inductively by applying the rule: $v_i \xrightarrow{p_j} v_j$ and $v_j \xrightarrow{p_k} v_k \implies v_i \xrightarrow{p_{jk}} v_k$. In this rule, $p_{jk}$ is the path expression obtained by concatenating $p_j$ and $p_k$. 
and $v_i \xrightarrow{k_l \circ k_k} v_k$ denotes a derived variable binding dependency.

Since we restrict XPath expressions in the defined XQuery fragment to be non-recursive and downwards navigations only, all the base variable dependencies form a tree structure if a single source XML document is involved. In this variable dependency tree, the hierarchies among the data bindings associated with the variable nodes are all preserved. In other words, the hierarchical relationships among the intermediate data bindings are implied by the base variable dependency set of a query. Hence, we may infer the subsumption of the preserved data bindings and their hierarchical relationships in different queries by reasoning about the containment of their respective variable dependencies.

In the example query $Q_1$ shown in Figure 3.1, variable $b$ is defined by a nested FLWR expression returning the book elements derived by navigating from the binding of $b_1$ via the path expression //book. Suppose we assign a default variable $r$ to be bound to the root element of the source document. Thus $r_{Q_1} \xrightarrow{}$ $d_{Q_1}$, $d_{Q_1} \xrightarrow{}$ $b_{Q_1}$ and $b_{Q_1} \xrightarrow{}$ $l_{Q_1}$. $t$ and $a$ are both defined based on $b$. Hence we have $b_{Q_1} \xrightarrow{/title} t_{Q_1}$ and $b_{Q_1} \xrightarrow{/author} a_{Q_1}$. $Q_2$ defines its variable $b$ directly from the root element via the path expression //book, and the other two variables $t$ and $a$ by the path expressions /title and /author based on $b$ respectively. Hence the base variable dependencies obtained from $Q_2$ are $r_{Q_2} \xrightarrow{}$ $b_{Q_2}$, $b_{Q_2} \xrightarrow{/title} t_{Q_2}$, and $b_{Q_2} \xrightarrow{/author} a_{Q_2}$. $r_{Q_1}$ and $r_{Q_2}$ are the same since they both are bound to the root element of the same source document.

**Definition 3.2 (Joinable Variable Binding Dependencies)** Let $Q$ be a query
3.3. VARIABLES IN XQUERY

and \( v_i \xrightarrow{p_j} v_j \), \( v_i \xrightarrow{p_k} v_k \) are two base variable dependencies in it. For each binding of
\( v_i \), the respective binding sets of its children variables \( v_j \) and \( v_k \) can be determined.
We hence call such variable dependencies **joinable** based on their common parent variable.

For example, in both queries \( Q_1 \) and \( Q_2 \), variables \( t \) and \( a \) are defined based on \( b \). For each binding of \( b \), the respective bindings of \( t \) and \( a \) can be determined. In this sense, the association relationship between the sibling \textit{title} and \textit{author} elements that are derived from the same \textit{book} parent element is established.

Suppose another query \( Q_3 \) defines its two variables \( s_t^{Q_3} \) and \( s_a^{Q_3} \) respectively from the root element of the same source document via the path expressions of \( \text{//book/title} \) and \( \text{//book/author} \). Although \( s_t^{Q_3} \) and \( s_a^{Q_3} \) are bound to the \textit{title} and \textit{author} elements, which are the same as for \( t \) and \( a \) defined in \( Q_1 \) or in \( Q_2 \), the information about the sibling \textit{title-author} element associations grouped by their parent \textit{book} elements is lost. That is, the intermediate binding results for \( s_t^{Q_3} \) and \( s_a^{Q_3} \) are grouped document-wise, whereas those for \( t \) and \( a \) in \( Q_1 \) or \( Q_2 \) are grouped by their respective parent \textit{book} elements. For example, by merging all the \textit{title} data bindings of \( s_t^{Q_2} \) and those of \( s_a^{Q_2} \) regardless of their different \textit{book} parent elements, we can obtain the same data bindings as those of \( s_t^{Q_3} \) and those of \( s_a^{Q_3} \) respectively. This is implied by the fact that from the base dependency set \( \{ s_r^{Q_2} \xrightarrow{\text{//book}} b_t^{Q_2}, s_b^{Q_2} \xrightarrow{\text{//title}} t_t^{Q_2}, s_b^{Q_2} \xrightarrow{\text{//author}} a_t^{Q_2} \} \), we can derive \( \{ s_r^{Q_2} \xrightarrow{\text{//book/title}} t_t^{Q_2}, s_r^{Q_2} \xrightarrow{\text{//book/author}} a_t^{Q_2} \} \). Since \( s_r^{Q_2} \) is exactly the same as \( s_r^{Q_3} \), the derived variable dependencies are the same as
the base variable dependencies defined in $Q_3$. However, the base variable dependencies in $Q_2$ cannot be derived from those in $Q_3$.

### 3.3.2 Variable Essentiality

We have shown that variables can help to preserve the hierarchies among the intermediate data bindings. Given two queries $Q_1$ and $Q_2$, intuitively we can reason about the containment of their corresponding data bindings by mapping variable pairs with the consideration of their dependencies (the details will be described in Section 3.6).

However, such a containment between the intermediate data bindings of two queries is not equivalent to query containment. That is, the latter is by definition about result inclusion. The meaning is twofold. One, to determine $Q_2 \subseteq Q_1$, the information required for constructing the result of $Q_2$ needs to be contained in the result of $Q_1$. Two, not all the intermediate data bindings of $Q_2$ are needed by the result construction. It may not be necessary to find mappings for all data bindings in $Q_1$ in order to decide $Q_2 \subseteq Q_1$.

To utilize the containment of intermediate data bindings of two queries for the purpose of query containment, we propose to identify the variables that have their bindings utilized in the result, i.e., **essential variables**.

**Definition 3.3 (Essential Variable)** Let $Q$ be a FLWR query.

- If a variable $\nu$ defined in $Q$ is essential, then the substitution of a $\nu$’s occurrence in $Q$ with $\nu$’s definition bound expression will give rise to a different query result from the original $Q$’s result for some database $D$. 
3.3. VARIABLES IN XQUERY

- If \( v \) is not essential then for all databases \( D \), if one substitutes any \( v \)'s occurrence with its definition bound expression then the result of \( Q \) will not change.

Below we give the criteria for distinguishing essential variables from those non-essential ones.

Explicit vs. Implicit Uses. After a for-variable \( v \) is defined, it may occur as the prefix of an XPath expression in either a for or a return clause. We refer to such a case as explicit use of \( v \). More specifically, we call it a Var use of \( v \) when a for clause defines another variable based on \( v \), whereas a Ret use when \( v \) occurs in a return clause.

In some cases, a for-variable may be influential in a certain way even without being explicitly used. For example, suppose the FLWR block where \( v \) is defined, called home FLWR of \( v \), encloses other nested FLWR blocks. Even if \( v \) has no Var or Ret use, it is still indispensable since the number of its bindings is the “loop counter” for invoking the inner FLWR blocks. This is also true when the return clause of the home FLWR of \( v \) contains any element constructor, since the construction of new elements is based on the iteration of \( v \)'s data bindings\(^2\). In this case, the newly constructed elements have a one-to-one correspondence relationship with the data bindings of \( v \). We refer to these cases as Encl uses of \( v \) (Encl is short for “Enclosing”).

Another implicit use of \( v \) is when variable(s) other than \( v \) occur in the return clause of the home FLWR of \( v \). We call this a Ret\(^*\) use of \( v \), in which

\(^2\)We assume that a variable is usually evaluated to have a multiplicity of data bindings, unless the schema knowledge implies otherwise.
case the number of data bindings of \( v \) influences the return times of the data bindings of the variable(s) other than \( v \).

Hence, *implicit uses* include Encl and Ret* uses. They serve certain contextual purposes that affect the query semantics. We have so far deliberately left out the cases when \( v \) occurs in the *where* clauses since we can resolve them as follows. If \( v \) occurs in the local *where* clause of the home FLWR, we push the *where* condition into the filter [ ] to be associated with \( v \)'s bound expression. Otherwise, i.e., \( v \) occurs in the *where* clauses of the FLWR blocks enclosed by the home FLWR of \( v \), then such cases are already covered by the Encl uses.

**One vs. Multiple Uses.** A defined variable is useless if there exists neither an explicit nor an implicit use of it in the query. We can hence remove its definition without affecting the query semantics. As we discussed earlier, a variable with implicit uses usually serves the contextual purpose such as the “loop counter” and hence the removing of it can result in different return semantics. Therefore, we identify a variable as a non-essential one if it has no use, or as essential if it has an implicit use. However, there is an exception for the latter case. Suppose a *for* clause defines both \( v_1 \) and \( v_2 \) and \( v_1 \overset{xp}{\supseteq} v_2 \). If the *return* clause contains an element constructor, this implies that both \( v_1 \) and \( v_2 \) have an Encl use. In the case when \( v_1 \) does not occur anywhere, we can replace \( v_1 \) in the definition of \( v_2 \) with its bound expression and then remove the definition of \( v_1 \). We thus identify \( v_1 \) as non-essential.

Identifying the essentiality of a variable used in only explicit cases is
more complicated since the number of uses matters. Suppose $v_1$ is only used once in a Var case for defining $v_2$. The data bindings of $v_1$ are then utilized only for deriving the bindings of $v_2$. In some sense, they are transient data results obtained in the middle of navigation for deriving the bindings of $v_2$. In this case, the single occurrence of $v_1$ can be replaced with its bound expression and then its definition can be removed. This is the same case for a variable being used only once in a Ret case. For example, in the example query $Q_1$ in Figure 3.1, both $t$ and $a$ are used only once in a Ret case. We can hence remove them after having rewritten the original return expressions $t$ and $a/last$ in the inner blocks as ${b/title}$ and ${b/author/last}$ respectively by substituting $t$ with $b/title$ and $a$ with $b/author$. Since $b_1$ in $Q_1$ is also used once in a Ret case, it can be substituted by its bound expression. Hence the original definition of $b$, i.e., \( \text{for } b \text{ in } d \text{ return } b_1//book \), is changed to \( \text{for } b \text{ in } d//book \) consequently.

A variable that has two or more explicit uses is however indispensable. For example, if two or more return expressions are prefixed with $v_1$, then the data being returned by them are grouped by each data binding of $v_1$. As explained in Section 3.3.1, such a sibling association between two types of data bindings is induced by their dependencies on a common variable. Hence, such a variable cannot be replaced. We thus first identify the essential variables that have more than one Ret uses, then those with one Ret use plus at least one Var use in which the dependent variable is already identified as an essential one, and lastly those that are used for defining at least two essential variables.
### Variable Essentiality Matrix

The criteria used for identifying essential variables are summarized by the matrix shown in Figure 3.6. Each cell in this matrix contains either the symbol “√” or “X”, indicating that the variable fitting in it is essential or non-essential respectively.

<table>
<thead>
<tr>
<th>No Use</th>
<th>Explicit Use</th>
<th>Implicit Use</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Var</td>
<td>Ret</td>
</tr>
<tr>
<td>Explicit Use</td>
<td>Var</td>
<td>√</td>
</tr>
<tr>
<td></td>
<td>Ret</td>
<td>X</td>
</tr>
<tr>
<td>Implicit Use</td>
<td>Ret*</td>
<td>√</td>
</tr>
<tr>
<td></td>
<td>Encl</td>
<td>√</td>
</tr>
</tbody>
</table>

**Figure 3.6: Variable Essentiality Matrix**

The matrix is used as follows. After a variable $v$ is defined, we seek for its first use and decide a row that matches it for $v^3$. We then look for the other uses of $v$ in the precedence order “explicit use first, implicit use second”. Namely, if any of the other uses is a *Var* or *Ret* use, we place $v$ in the *Var* or *Ret* column. Otherwise, we find the column with the matching use for $v^4$.

This process for identifying variable essentiality may be better sketched by the transition model shown in Figure 3.7. A variable $v$ enters the *Def* state at the time when it is defined. Then the transition from one state to another occurs when $v$ is used in the case as indicated on the corresponding arc. If $v$ enters any state denoted by a double circle, then it is identified as essential. Otherwise, it is identified as non-essential.

---

3If $v$ has no use at all, then it cannot be placed in any row of the matrix. We recognize $v$ as non-essential.

4If $v$ has only a single use, then it matches with the “No Use” column.
Theorem 3.1 (Soundness of Variable Essentiality Identifying Model) Let $Q$ be a query satisfying the BNF rules for the defined XQuery fragment in Section 3.2. For a variable $v$ defined in $Q$, it starts from the Def state of the essentiality identification model in Figure 3.7 and transits between states if $v$ occurs in the use case marked on the corresponding transition arc. If $v$ ends at a double circled state, then $v$ is an essential variable. Otherwise, $v$ is not essential. □

Proof Sketch: Based on the BNF rules shown in Figure 3.5 for the defined XQuery fragment, a defined variable $v$ may appear in other variables’ bound expressions, return expressions or where clauses.

First, the number of Var and Ret uses of $v$ is relevant to its essentiality. That is, if $v$ has more than one occurrence, i.e., $v$ has either at least two Var uses, or one Var use and one Ret use, or two Ret uses, then replacing $v$’s occurrences with its bound expression would cause the loss of joinable dependencies (see Definition 3.2) among those data bindings that have ascendant-descendant relationships with $v$’s bindings. The query re-
result may be different due to such an information loss and \( v \) is thus essential based on Definition 3.3. Such cases are identified by the state transitions from the single circled \( \text{Var} \) or \( \text{Ret} \) state to the double circled \( \text{Var} \) or \( \text{Ret} \) state.

Second, \( v \) is essential if it is not used for defining another variable in the same \textit{for} clause and the \textit{return} clause contains an element constructor \(< \text{tag} > e < /\text{tag} >\). This is because that if \( v \) is removed, the construction of the new \(< \text{tag} >\) elements would not be invoked and thus the query result will be different. This case is captured by the state transitions ending at \( \text{Encl} \). Note that if \( v \) enters the single circled \( \text{Var} \) state before it encounters an \( \text{Encl} \) use, i.e., \( v \) is used for defining another variable \( v' \), then \( v \) can be eliminated while substituting its bound expression for its occurrence in the definition of \( v' \). This will not affect the construction of new \(< \text{tag} >\) elements which are still invoked by the bindings of \( v' \).

Another scenario for identifying \( v \) to be essential is when \( v \) has an implicit \( \text{Ret}^* \) use. In this case, the existence of \( v \) is required since the number of \( v' \)'s bindings serves as the “counter” for invoking the return expressions. The state transitions ending at \( \text{Ret}^* \) capture this case. The reason for why there is no transition from the single circled \( \text{Var} \) state to the \( \text{Ret}^* \) state is the same as the reason for there not being any transition from \( \text{Var} \) to \( \text{Encl} \).

In all other cases, i.e., \( v \) has no use or it is stuck in the single circled \( \text{Var} \) or \( \text{Ret} \) state after its first use, the query result will not change if \( v \) is eliminated while its occurrences, if any, are substituted with its bound expression. Even if \( v \) occurs in a \textit{where} clause which asserts a condition \( c \) on \( v' \)’s binding set, it is irrelevant with respect to the essentiality identification process. This is because that 1) if \( v \) occurs in the local \textit{where} clause, then this
where clause can be removed by pushing \( v \)'s bound expression as a filter subcomponent; 2) if \( v \) occurs in the nested \( \text{where} \) clauses, substituting \( v \)'s occurrences in them with \( v \)'s bound expression would not cause the loss of information and hence the query result since no joinable dependencies based on \( v \) exist. □

For example, variable \( b^{Q1} \) ends up in the \( Encl \) state and has hence been identified as essential. Both \( t^{Q1} \) and \( a^{Q1} \) start from the \( Def \) state and then enter the single circled \( Ret \) state. Hence, \( t^{Q1} \) and \( a^{Q1} \) are identified as non-essential.

Based on this variable essentiality identifying model depicted in Figure 3.7, we present a variable minimization algorithm in Figure 3.8. This algorithm carries out the following steps and outputs an essential variable set \( EVS \) which collects all the essential variables.

In Step 3 of the Variable Minimization Algorithm shown in Figure 3.8, the top-level \( if \) and the second-level \( Else if \) statements check if the uses of a variable \( v \) conform to either the state transitions ending at \( Encl \) or \( Ret^* \). The innermost \( Else if \) statement checks if \( v \) follows the state transition from \( Def \) to \( Var \) or \( Ret \) and then keeps track of the number of its uses. Then Step 4 of the algorithm covers the checking of two other state transitions to see if \( v \) has multiple \( Var \) or \( Ret \) uses to be identified as an essential variable.
ALGORITHM Variable_Minimization

INPUT: Query $Q$ in the defined XQuery fragment.

OUTPUT: A set named $EVS$ containing all the essential variables in $Q$.

Step 1. Initialize a stack $S$ and two sets $RVS$ and $EVS$; // $S$ is used for keeping track of the nested block structure of $Q$ being entered and processed. $RVS$ stores the variables whose essentialities are temporarily undecided yet, while $EVS$ stores those that have been identified as essential.

Step 2. Navigate the parsed syntax tree of $Q$;
   If a new FLWR block $flwr_i$ is encountered Then
   $S = push(S, flwr_i)$;
   Foreach $v$ defined in $flwr_i$ Do
   $v.counts=0$;
   $RVS = RVS \cup \{v\}$;

Step 3. If the return clause in $flwr_i$ contains an element constructor (corresponding to the state transition from Def to Encl in Figure 3.7) Then
   // identify all variables in the current RVS as essential and hence move them into EVS
   $EVS = EVS \cup RVS$;
   $RVS = \emptyset$;

   Else If more than one variable occurs in the return expressions (i.e., from Def to Ret*) Then
   $EVS = EVS \cup RVS$;
   $RVS = \emptyset$;

   Else If a variable $v$ in RVS is used in a for or return expression (i.e., from Def to Var or Ret) Then
   $v.counts++$;

Step 4. If $flwr_i = top(S)$ and all the children FLWR blocks of $flwr_i$ have been processed Then
   $flwr_i = pop(S)$; mark $flwr_i$ as being processed;

Foreach $v \in RVS$ Do
   // decide the essentiality for the variables remaining in RVS
   If $v.counts > 1$ Then
   $EVS = EVS \cup \{v\}$;

   Else
   $v$ is non-essential;

Figure 3.8: The Variable Minimization Procedure
3.4 XQuery Normalization

3.4.1 Different Goals in XQuery Normalization

In traditional database query engines, the most often used query normalization technique is to normalize SQL-like statements into either a canonical disjunction of conjunctions (DOC) or a canonical conjunction of disjunctions (COD). Such a normalization changes the structure of a query tree so that it can be exploited for issues such as logic mapping.

The flexibility in composing an XQuery using nested FLWR expressions imposes difficulties for reasoning about the containment relationship between two given queries. We hence propose a normalization technique that translates each input query into a canonical form with certain characteristics which can facilitate the containment reasoning. XQuery normalization must preserve the query equivalency.

One of the desired characteristics for the canonical form is a clear separation between pattern matching and result construction. As the two constituent parts of the XQuery semantics, the former captures which information is extracted from the source document, and the latter represents the template for transforming the intermediate data bindings and constructing the result. In this sense, the former can be seen as a mapping from portions of the source data tree to the intermediate result bindings, while the latter is a mapping from the intermediate result to the components constructed in the result document. By separating these two parts, containment reasoning can be related to the two-step mappings.

Therefore, our normalization strategy is different from the other XQuery
normalization work [MFK01, W3C03d, M. 01] in their goals. Below we first introduce the existing three major efforts in normalizing XQuery.

The middleware systems SilkRoute [FTS00] by AT&T and Xperanto [CKS+00] by IBM target at efficiently publishing relational data as XML documents. They extract data by means of SQL which utilizes the mature query processing capabilities available in modern relational database engines and then structure and tag the result tuples. Besides data extraction and transformation from relational data to XML, they provide the user with a view of the relational database which can be queried using XML query languages such as XQuery. The Agora project [MFK01] is also such a data integration system which provides a global XML view over multiple relational data sources. All three systems take the approach of splitting an input query into a navigation part and a result construction part, the former of which can then be translated into SQL over a virtual generic relational schema.

**Agora’s Normalization Rules.** Agora [MFK01] takes an extra step before mapping XQuery to SQL. It applies a syntax-based normalization approach to simplify XQuery first to facilitate the translation from XQuery to SQL. Such a normalization is more than just the traditional SQL/OQL query optimization rewriting, which mainly apply the rules of “predicate push-down”. Agora suggests a set of simplification/unnesting rules for normalizing the input XQuery into a form which can be directly translated to SQL, if possible. Although these rules are designed for transforming XQuery into a form from which the translation to SQL is more direct, they are also useful for general-purpose optimizations.
3.4. XQUERY NORMALIZATION

**W3C’s XQuery Normalization.** W3C has also proposed a set of normalization rules for syntactically reducing XQuery into XQuery core [W3C03d], which is a relatively small but fully expressive sub-language of XQuery. For instance, a complex for expression might be rewritten as a composition of several simple for expressions. Also, an element or scalar projection expression is rewritten as a verbose for iteration of node and kind tests. A where expression is rewritten as a conditional if-then-else expression. The purpose of such a normalization is to derive a declarative query core, which can be straightforwardly given the full semantics needed for both the static type analysis phase and the dynamic evaluation phase.

**AT&T’s Semi-monad Law Based Normalization.** In [M. 01], some additional normalization rules based on the monad laws from the functional language theory are supplied. Analogous to the algebraic laws used for relational query optimization, these rules are used to simplify the XQuery expressions by eliminating unnecessary for expressions or by reordering or distributing the computations.

**Discussion.** These three normalization techniques differ in their goals. The W3C’s approach targets to make the query as declarative as possible, to be close to the XQuery operational semantics. It requires the for or let clauses be transformed into the elementary form where return clauses are added when necessary to separate multiple variable definitions, so that only one variable is defined in each for clause. Agora’s motivation is to utilize normalization rules to obtain an appropriate XQuery form for which
the XQuery-SQL translation can be initiated. A complex for clause where a list of variables are defined does not have to be transformed into the elementary nested FLWR expressions. AT&T’s normalization approach aims to syntactically optimize XQuery or its core expression by applying some equivalence semi-monad rewriting rules founded on the functional language theory. [MFK01] and [M. 01] share many similarities in their normalization rules although different approaches are taken.

### 3.4.2 Our Normalization Steps

We first adopt existing normalization techniques to derive a form which satisfies the following conditions: (1) let-clause free; (2) no for-variable bound to an empty sequence "()" or a unit expression; and (3) the bound expression of a for-variable is free of nested FWR expressions.

To meet condition (1), we adopt a commonly used normalization rule which eliminates all the let-variables by substituting their occurrences with their bound expressions. We can henceforth assume that only the FWR expressions are to be dealt with.

For condition (2), we apply the empty and unit rules from [M. 01]. Namely, \[ \text{for } v \text{ in } () \text{ return } e = \{ \} \] and \[ \text{for } v \text{ in } e \text{ return } v = e \] respectively.

To satisfy condition (3), the associative law in [M. 01] or the cancel-out rule in [MFK01] is employed to simplify the for-clause with nested FWR expressions by unnesting them into the return-clause instead. For example, \[ \text{for } v_2 \text{ in } (\text{for } v_1 \text{ in } e_1 \text{ return } e_2) \text{ return } e_3 = \]

\[ \text{for } v_1 \text{ in } e_1 \text{ return } \{ \text{for } v_2 \text{ in } e_2 \text{ return } e_3 \} \]

As a result, a for clause contains only XPath expressions, and nested
3.4. XQUERY NORMALIZATION

FWR expressions only occur in the return clauses. Therefore, the BNF rule of for-let in Figure 3.5 can be revised as \( \text{for-let} ::= \text{for} [v] \in ([v]?xp) \) for the normalized query. It indicates that no more let-clauses exist, and the for clauses contain only XPath expressions but no more unit variables nor nested FLWR expressions.

**Our Normalization Rules.** In addition, we apply a normalization technique which makes use of the variable essentiality. That is, for an identified non-essential variable, we substitute its occurrences with its bound expression and then remove the variable definition. Below are the three concrete rules to apply.

**Single-Ret-Use Rule:**
\[
\text{for } v \text{ in } e1 \text{ return } e2 = \{v := e1\}
\]

It means that if \( v \) occurs just once in \( e2 \), which is the only return expression in the home FWR of \( v \), then we directly return \( e2 \), however with the single occurrence of \( v \) in it replaced with \( e1 \), denoted as \( \{v := e1\} \). This rule can be seen as an extension of the right unit law in [M. 01], which requires \( e2 \) to be exactly \( v \).

**Single-Var-Use Rule:**
\[
\text{for } v1 \text{ in } e1, v2 \text{ in } e2 \text{ return } e3 = \text{for } v2 \text{ in } e2\{v1 := e1\} \text{ return } e3
\]

In the left hand side, \( v1 \) has only one occurrence in \( e2 \) for defining \( v2 \). When \( v1 \) still has only one \( \text{Var} \) use but \( v2 \) is defined in an enclosed FWR block, the rule we apply is:
\[
\text{for } v1 \text{ in } e1 \text{ return } \{\text{for } v2 \text{ in } e2 \text{ return } \{e3\}\{e4\}\} = \text{for } v2 \text{ in } e2\{v1 := e1\} \text{ return } \{e3\}\{e4\}
\]
Local-Where Rule: \[\text{for } v \text{ in } e_1 \text{ where } c \text{ return } e_2 = \text{for } v \text{ in } e_1[c] \text{ return } e_2\]

where \(c\) is a condition imposed on \(v\). That is, the local where-conditions are inlined into the corresponding for-variable bound expressions as their attached filter conditions. This rule however does not apply to the where-conditions imposed on the variables defined in the outer blocks.

Lemma 3.1 (XQuery Normalization) If \(Q'\) is a normal form of query \(Q\) obtained by applying the given normalization rules, then \(Q'\) satisfies:

- \(Q'\) is equivalent with \(Q\);
- every variable occurring in \(Q'\) is essential;
- \(Q'\) is consistent with the BNF shown in Figure 3.9.

\[e ::= v \mid (v)\times p \mid \text{fwr} \mid r e\]
\[
fwr ::= \text{for } \text{where-return}\]
\[
\text{for} ::= \text{for } [v] \text{ in } [(v)\times p]\]
\[
\text{where-return} ::= (\text{where } c^*)? \text{ return } c\]
\[
r e ::= \{e_1\} \ldots \{e_k\} \mid \langle \text{tag} \rangle e \langle \text{/tag} \rangle\]

\(c^*\) are the conditions imposed on the non-local variables.

Figure 3.9: BNF Rules for Normalized XQuery

In comparison to the BNF for the considered XQuery fragment defined in Figure 3.5, the normalized query has (1) no let clause; (2) no for-variable bound to an empty sequence “()” or a unit expression; (3) no for-variable defined by a unit expression, a return expression or a nested FWR expression; (4) no where-condition imposed on local variables.
Below we apply the normalization rules to simplify the example query \( Q_1 \) into its normalized form, as depicted in Figure 3.10.

![Diagram of query transformation](image)

**Figure 3.10: Example Query \( Q_1 \): Before and After Normalization**

### 3.5 XQuery Decomposition

#### 3.5.1 VarTree and TagTree

After the normalized form of a given query is obtained, pattern matching is represented by the definitions of variables by XPath expressions in the `for` clauses. The result construction part corresponds to the nested FWR structure, the new element constructor and the return expressions at each level in the `return` clauses.

We now define two tree structures for capturing the preserved data binding hierarchies in the pattern matching process and the output view structure built in the result construction process respectively. We refer to the former tree as a *variable dependency tree* and the latter as a *tagging template tree*. For brevity, they are called VarTree and TagTree respectively.
Definition 3.4 (Variable Dependency Tree) Given a normalized query $Q$, a variable dependency tree $\text{VarTree} = (V_{vt}, E_{vt}, L_{vt})$ is constructed as follows:

- each internal node $v \in V_{vt}$ represents a defined variable $v$;
- each edge $(v, w) \in E_{vt}$ labelled $p_i$ ($p_i \in L_{vt}$) represents the dependency $v \overset{p_i}{\rightarrow} w$, i.e., $w$ is defined as “for $w$ in $v$ $p_i$”;
- each unlabelled leaf node represents a return expression “return $v$ $p_j$”, with its incoming edge labelled $p_j$.

Our VarTree is different from the tree pattern structure in [AYCLS01] for representing XPath queries with no nesting semantics. In their tree pattern, each edge corresponds to a navigation step which is a component of an XPath expression, whereas every edge in our VarTree represents an XPath expression which may contain arbitrarily long navigation steps. In addition, each node of our VarTree is built for a specified variable and it captures the semantics of which information is preserved as the intermediate results in the variable. However, only the leaf nodes in the tree pattern in [AYCLS01] represent the information to be returned, while the internal nodes correspond to the transient data after the evaluation of each navigation step.

Definition 3.5 (Tagging Template Tree) For a normalized query $Q$, a tagging dependency tree $\text{TagTree} = (V_{tt}, E_{tt}, L_{tt})$ is constructed as follows:

- each internal node $s \in V_{tt}$ represents an element constructor producing new $\langle s \rangle$ elements;
3.5. XQUERY DECOMPOSITION

- each edge \((s, t) \in E_H\) labelled by one or more variables \(v_i, \ldots, v_m\) \((v_i, \ldots, v_m \in L_H)\) indicates that a new \(\langle t \rangle\) element is constructed for each variable binding (if there is a single variable) or for each resulting tuple of the binding product (if there is more than one variable). Also, the \(\langle t \rangle\) elements are enclosed within their corresponding \(\langle s \rangle\) elements constructed in the nearest outer block;

- each unlabelled leaf node represents a return expression which corresponds to a leaf node in VarTree.

A TagTree can be seen as the view structure of the query result. Each internal node can be seen as a skolem function that generates new elements with their implicit node ids based on the input variable bindings. While each leaf node corresponds to a function capable of deeply returning the associated data bindings and their subtrees.

For example, \(Q1\)’s VarTree and TagTree are depicted in Figure 3.11 and those of \(Q2\) are in Figure 3.12.

![Figure 3.11: Q1’s VarTree and TagTree](image)

For a normalized \(Q\), the nodes in its TagTree is connected via variables to the nodes in its VarTree, which represent the definitions of those respective variables. In addition, there is a one-to-one correspondence between
the variables captured as nodes in VarTree and those represented as labels in TagTree.

**Proposition 3.1 (Variable Correspondences between VarTree and TagTree)**

Given a normalized $Q$, its $VarTree = \langle V_{vt}, E_{vt}, L_{vt} \rangle$ and its $TagTree = \langle V_{tt}, E_{tt}, L_{tt} \rangle$, then $v_i \in L_{tt} \iff v_i \in V_{vt}$.

This means that VarTree contains only essential variables which are those exactly required for constructing the result as indicated by TagTree. Such an exact correspondence between the variables in VarTree and those in TagTree provides the logical base for us to utilize VarTree-based containment mapping as the means for reasoning about the overall query containment.
3.5.2 Tree Representations Connected to Formal Semantics

We have introduced the VarTree and TagTree representations for a normalized XQuery and informally explained their capabilities of capturing the pattern matching and result construction semantics respectively. We now connect these two tree structures to the formal semantics of our XQuery fragment introduced in Section 3.2.1. Based on such a connection, we can reduce XQuery containment to VarTree-based tree matching and use the TagTree to facilitate XQuery rewriting.

The query formal semantics as explained in Section 3.2.1 can be precisely captured by the VarTree and TagTree structures. Let’s take VarTree for example first. The construction of a VarTree node \( v \) corresponds to the adding of a mapping for variable \( v \) into \( \text{dynEnv} \), which is captured by the formal semantics Rule (3) in Section 3.2.1. The parent node of \( v \) corresponds to the variable being “looked up” (i.e., Rule (1)) for evaluating “\( \text{for } v \text{ in } e \)”, and the incoming edge is labelled by the XPath expression \( e \) (i.e., Rule (2)). This way, we capture the incremental \( \text{dynEnv} \) building process using the VarTree structure.

In the TagTree, the parent-child node relationship reflects the nested return structure, while the sibling node relationship indicates the sequence of result constructing (i.e., Rule (4)). The variables labelled on the TagTree edges are those being “looked up” for their bindings based on which the new elements are constructed. We hence have the proposition below.

**Proposition 3.2 (XQuery Evaluation Tree Model)** Let \( VT \) and \( TT \) be the VarTree and TagTree of a normalized \( Q \) respectively. Each node in \( VT \) denotes an essential
3.6 VarTree-based Containment Mapping

3.6.1 VarTree Homomorphism

A common approach for the containment of conjunctive queries is to utilize graph homomorphism [CR97] or tree pattern homomorphism [CR97] that finds the mappings between the query constructs encoded as a graph or a tree structure. In the same spirit, we now propose an XQuery containment checking technique that utilizes VarTree homomorphism.

The classic tree homomorphism is defined by a mapping function \( f \) between two trees [F. 84]. Given a pattern tree \( P = (V_P, E_P) \) and a target tree \( T = (V_T, E_T) \), the mapping from nodes in \( V_P \) to those in \( V_T \) requires to preserve labels and ancestorships. That is, for all nodes \( u \) and \( v \) in \( V_P \), it requires that:

1. \( \text{label}(v) = \text{label}(f(v)) \), and
2. \( u \) is an ancestor of \( v \) in \( P \) iff \( f(u) \) is an ancestor of \( f(v) \) in \( T \).

We now give modifications to the classic tree homomorphism requirements and use them to define VarTree homomorphism.
Definition 3.6 (VarTree homomorphism) Suppose $VT_1$ and $VT_2$ are VarTrees of $Q_1$ and $Q_2$ respectively. A VarTree homomorphism from $VT_2$ to $VT_1$, denoted $VT_2 \subseteq VT_1$, is a total mapping function $\Phi$ from the nodes of $VT_2$ to those of $VT_1$ such that for all nodes $u$ and $v$ of $VT_2$:

1). $u$ is an ancestor of $v$ in $VT_2$ iff $\Phi(u)$ is an ancestor of $\Phi(v)$ in $VT_1$;

2). $\rho_2(v) \subseteq \rho_1(\Phi(v))$;

3). if $T(v) = VAR$, then $T(\Phi(v)) = VAR$ or $RET$; if $T(v) = RET$, then $T(\Phi(v)) = RET$.

Below we explain each of the required conditions in Definition 3.6.

**Condition of Ancestorship Preserving.** Condition 1) required by VarTree homomorphism requires the preserving of ancestorships. If $u$ is an ancestor of $v$ in $VT_2$, then $\Phi(u)$ is an ancestor of $\Phi(v)$ in $VT_1$ and vice versa.

**Condition of XPath Containment.** In the VarTree homomorphism, the condition of label preserving required by the classic tree homomorphism is changed to $\rho_2(v) \subseteq \rho_1(\Phi(v))$. We first explain the functions $\rho_2$ and $\rho_1$, and then the meaning of $\subseteq$ in this context.

Both $\rho_2$ and $\rho_1$ are the functions that retrieve labels (i.e., XPath expressions) from the VarTree edges. $\rho_2$ applies to nodes in $VT_2$, while $\rho_1$ applies to nodes in $VT_1$. Another difference is that $\rho_2(v)$ ($v \in V_{VT_2}$) obtains the XPath expression labelled on the edge directly incident to $v$, whereas $\rho_1(\Phi(v))$ ($\Phi(v) \in V_{VT_1}$) may involve the chaining of XPath expressions. That is, depending on which node $\Phi(w)$ in $VT_1$ that the parent node $w$ of
3.6. VARTREE-BASED CONTAINMENT MAPPING

$v$ is mapped to, $\rho_1(\Phi(v))$ obtains an XPath expression by concatenating the XPath expressions along the path from $\Phi(u)$ to $\Phi(v)$. For example, if $u$ is the parent node of $v$ in $VT_2$, then $\rho_1(\Phi(v))$ returns the concatenated XPath expressions from $\Phi(u)$ to $\Phi(v)$.

Suppose $\rho_2(v) = xp_2$ and $\rho_1(\Phi(v)) = xp_1$, checking the condition $xp_2 \subseteq xp_1$ applies the technique of XPath containment from [G. 02b]. That is, $\subseteq$ denotes XPath containment as in [G. 02b]. This condition hence utilizes XPath containment for checking the containment between the data bindings associated with a pair of VarTree nodes.

Note that for the root node of a VarTree, its incoming edge is labelled with the URL of the source XML document being queried against. In this case, $\rho_1$ and $\rho_2$ retrieve the respective URLs of the source documents involved in the two queries, and $\rho_2(v) \subseteq \rho_1(\Phi(v))$ ($v$ and $\Phi(v)$ are the root nodes of $VT_2$ and $VT_1$ respectively) only if the two URLs are the same.

**Condition of Node Type Mapping.** The tree nodes involved in the classic tree homomorphism are uniformly treated as being of the general node type. A VarTree however has nodes of the variable type $VAR$ and of the return type $RET$. The nodes of the $VAR$ type are those corresponding to essential variables. The nodes of the $RET$ type are those leaf nodes generated for the return expressions. We use $T(v)$ to denote the type of $v$.

The reason for condition 3) in VarTree homomorphism is the following. If $T(v) = RET$, it indicates that $v$’s bindings are to be deeply returned to constitute the result. Thus $\Phi(v)$ must also be of the $RET$ type to make its bindings available in the view result so that they can later be used to
3.6. VAR TREE-BASED CONTAINMENT MAPPING

Substitute $v$’s bindings. That is, if $T(v) = RET$, then $T(\Phi(v)) = RET$.

On the other hand, if $T(v) = VAR$, then $v$’s bindings are not to be deeply returned but correspond to new elements. In a sense, $v$ serves as a place holder in the result tagging template for preserving the hierarchy. Hence, $v$ can be mapped to a node of either the $VAR$ type or the $RET$ type, i.e., $T(\Phi(v)) = VAR$ or $RET$.

If the mapping node of $v$ is of the $RET$ type, then we extend the XPath containment symbol $\subseteq$ to also accommodate the domain covering cases. For instance, assume $v$ is defined as $b/author/last$, the node $w$ mapped to is of the $RET$ type and defined by $b/author$. Since the type of $w$ inferred from the schema knowledge is above the type of $v$ in the type hierarchy, the information required by $v$ can be derived from the deeply returned subtree rooted at $w$. We allow such a case in $\rho_2(v) \subseteq \rho_1(w)$.

3.6.2 The MAC Mapping Procedure

Here we describe an algorithm which procedurally conducts the condition checking required by the VarTree homomorphism for establishing variable mappings.

**Definition 3.7 (MAC Mapping)** Given $Q_1$ and $Q_2$ and their respective VarTrees $VT_1$ and $VT_2$, MAC mapping is a **top-down tree inclusion** process from $VT_2$ to $VT_1$. Along with this process, conditions required in Definition 3.6 for VarTree homomorphism are checked to for establishing variable mappings.

From the requirement of condition 2) in Definition 3.6 for VarTree homomorphism, we can infer that the MAC mapping procedure needs to be
carried out in a top-down fashion. That is, we traverse VT\textsubscript{2} in preorder and establish the mapping for each node encountered in that order. This is because to find a mapping node for \( v \) in VT\textsubscript{2}, condition 2) requires to know the mapping node for the \( v \)'s parent to which \( \rho 1 \) can be applied to obtain the concatenated XPath expression. In addition, the other conditions required by VarTree homomorphism need to checked along the top-down VarTree traversal.

We called this a MAC mapping since this tree-scale containment mapping procedure is relatively macroscopic in contrast to the microscopic XPath containment mapping performed at the tree node level. In contrast, the low-level node mapping via XPath containment is called an MIC mapping. Their names are obtained by extracting the first three letters of macroscopic and microscopic respectively. This algorithm for realizing MAC mapping is sketched in Figure 3.13.

We now illustrate such a MAC mapping process between the VarTrees of the two example queries \( Q_1 \) and \( Q_2 \) in Figure 3.14. For example, since both \( Q_1 \) and \( Q_2 \) query against the same XML document, node \( \$r \) in VT\textsubscript{2} is mapped to node \( \$r \) in VT\textsubscript{1}. The rationale of mapping \( \$b \) in VT\textsubscript{2} to \( \$b \) in VT\textsubscript{1} is the existence of a homomorphism from the tree pattern for “//book[price<50]” to that for “/bib/book[price<40]” by utilizing the XPath containment approach in [G. 02b]. The MAC mapping process continues until there is no mapping node for \( \$a \) in VT\textsubscript{2}. Therefore, VT\textsubscript{2} \notin VT\textsubscript{1} since it is not a total mapping from the nodes in VT\textsubscript{2} to those in VT\textsubscript{1}.
ALGORITHM VarTree_MAC_Mapping

INPUT: Two VarTrees $VT_1$ and $VT_2$ for $Q_1$ and $Q_2$ respectively.
OUTPUT: variable mapping pair set $M = \{ (v, \Phi(v)) \}$.

 Traverse $VT_2$ in preorder, and meanwhile

 Foreach encountered variable $v \in VT_2$ Do

 Check if $\text{parent}(v)$ has a mapping node, continue only if one exists;
 Search for $v$’s mapping node $w$ within the scope of the subtree rooted at
 $\Phi(\text{parent}(v))$ such that $w$ satisfies:
 1). $\rho_2(v) \subseteq \rho_1(w)$
 2). if $\tau(v) = \text{VAR}$, then $\tau(w) = \text{VAR}$ or $\text{RET}$;
 if $\tau(v) = \text{RET}$, then $\tau(w) = \text{RET}$.

 If such a $w$ exists Then

 establish a mapping pair $(v, w)$ and add it into the mapping set $M$

 End If

 End Foreach

 If there exists a total mapping from variables in $VT_2$ to those in $VT_1$ Then

 output $VT_2 \subseteq VT_1$ and the mapping set $M$

 End If

 Figure 3.13: The MAC Mapping Procedure

3.6.3 VarTree Homomorphism in the Logic View

We now take an alternative approach to the VarTree homomorphism approach for reasoning about the inclusion of information preserved by the two queries. In this approach, we first study the logic formalism underlying the VarTree structure and then interpret our MAC mapping in terms of the logic rule-based containment mapping.
Related Work for Logics for XML. Recently, there has been a renewed interest in employing monadic second order logic (MSO) as the yardstick for the expressiveness of XPath-like query languages [Nev02, LF03, FT00]. Gottlob and Koch have discovered that a monadic datalog program, defined as a set of conjunctive monadic queries with unary head predicates [GK02], exactly captures monadic second-order logic (MSO) over trees [G.02a]. An interesting application of monadic query languages is the extraction of information from Web document trees such as the Web wrapping language ELLog~ [BFG01].

In [G.02a], ELLog~ is represented by a set of extended monadic datalog rules in the form \( p(X) := p_0(X_0), subelem_{path}(X_0, X), C, R \). \( p \) is a pattern predicate, \( p_0 \) is the “parent pattern” being either a pattern predicate or root, \( subelem_{path} \) can be seen as defined through a conjunction of child and label atoms, \( R \) denotes the “pattern references”, and \( C \) is a set of “condition atoms”. Since VarTree has the similar information extraction capability as ELLog~, we adopt a similar monadic datalog representation for VarTree.
Translating VarTree to Monadic Datalog Program. Given a VarTree, we formulate for each edge that represents a parent-child variable dependency pair \( v_i \rightarrow v_j \) a rule: 

\[
q_j(X_i, X_j) := q_i(X_i), \text{subsub}(X_i, X_j), C, R.
\]

In this rule, the two head variables, i.e., \( X_i \) and \( X_j \), are the set variables representing the data bindings associated with \( v_i \) and \( v_j \) respectively. In the rule body, the subscript \( xp \) of the subgoal predicate \( \text{subsub} \) is the XPath expression for deriving the data bindings represented by \( X_j \) from those by \( X_i \). \( C \) refers to a set of conditions imposed on the set variables other than \( X_j \), \( R \) represents a set of references to the relevant predicates. We adopt the binary head predicates instead of unary predicates since they can explicitly illustrate the extraction of variable dependency hierarchies. By mapping each edge of the VarTree to such a rule with the binary head predicate representing the preserved variable hierarchy, we obtain a set of rules which compose a non-recursive monadic datalog program \( P \).

The starting rule in \( P \) is however a rule with a unary head predicate. It is represented as 

\[
\text{root}(X_0) := \text{URL}(X_0),
\]

where \( X_0 \) is a singleton set variable bound to the root element of the input document, and \( \text{URL} \) is a constant predicate.

All the other rules possess the following properties:

- each rule body contains a binary predicate \( q_i(X_i) \), whose second variable \( X_i \) must also appear in the second position in the head predicate of another rule;

- no two rules have the same second head variable;

\( P \) is non-recursive due to the restriction of non-recursive XPath expressions and non-recursive variable dependencies.
3.6. VARTREE-BASED CONTAINMENT MAPPING

- the first head variable of each rule must appear either in the starting rule or as the second head variable of some binary rule;

- the predicate arity for all the rules is bounded (≤ 2);

- both two head variables of a rule must also occur in the subgoal predicate \textit{subelem}_{x,p}.

Containment Mapping between Monadic Datalog Programs. Based on the monadic datalog representation of a VarTree, we build the connection between our MAC mapping and an extended containment mapping between two rule sets.\footnote{The containment mapping technique utilized here is applied to the second-order set variables. It is hence an extension of the basic containment mapping approach used for the first-order queries.}

Suppose \( \mathcal{P}_1 \) and \( \mathcal{P}_2 \) are the two monadic datalog programs obtained from the translation of \( VT1 \) and \( VT2 \) respectively. Hence, there are implicit dependencies among the derived rules of each program due to the variable dependencies captured by each VarTree. That is, if the first head variable \( X_i \) of a rule \( q_j \) is the second head variable of a rule \( q_i \), then \( q_j \) is dependent on \( q_i \). The structure of rule dependencies for each program is consistent with the the structure of the corresponding VarTree. The conditions for establishing the containment mapping \( \Psi \) from the rule set in \( \mathcal{P}_2 \) to the rule set in \( \mathcal{P}_1 \) are given below:

1). the starting rule of \( \mathcal{P}_2 \) maps to that of \( \mathcal{P}_1 \), which requires \( URL_1 = URL_2 \) (the two constant predicates are the same) and \( \psi(X_{10}) = X_{20} \) (one root variable is mapped to another via \( \psi \)).
2. \( q^2_u \) is an ancestor of \( q^2_v \) by the rule dependencies in \( \mathcal{P}2 \) iff \( \Psi(q^2_u) \) is an ancestor of \( \Psi(q^2_v) \) in \( \mathcal{P}1 \);

3. each rule \( q^2_j(X_2i, X_2j) := q^2_i(\_, X_2i), subelem_{xp2}(X_2i, X_2j), C2, R2 \) in \( \mathcal{P}2 \) is mapped to a rule\(^7\) \( q^1_j(X_1i, X_1j) := q^1_i(\_, X_1i), subelem_{xp1}(X_1i, X_1j), C1, R1 \) in \( \mathcal{P}1 \), denoted as \( \Psi(q^2_j) = q^1_j \), iff:

   a. \( \Psi(q^2_i) = q^1_i, \psi(X_1i) = X_2i, \psi(X_1j) = X_2j, xp2 \subseteq xp1; \)

   b. every variable involved in \( C1 \) is mapped to a variable in those involved in \( C2 \);

   c. every referred predicate in \( R1 \) is mapped to a predicate in those referred by \( R2 \).

Here we emphasize the two inverse directions of mapping. The mapping \( \Psi \) is between rules of two programs while the mapping \( \psi \) is between variables of two rules. \( \Psi \), which maps from the rules of \( \mathcal{P}2 \) to the rules of \( \mathcal{P}1 \), corresponds to the MAC mapping at the tree level. The mapping direction of \( \psi \) is however from each variable in a rule of \( \mathcal{P}1 \) to a variable in a rule of \( \mathcal{P}2 \). To establish the \( \psi \) mapping, XPath containment is utilized. In this sense, it is like the MIC mapping at the tree node level.

Assume we ignore the mapping of \( Cs \) and \( Rs \) for now, then the mapping between two rule programs is equivalent to the MAC mapping between two VarTrees from which the rule programs are translated.

\(^7\)This rule may be obtained from chaining a sequence of rules, each dependent on the one before. In the resulting rule, the first head variable is that of the first rule and the second head variable is that of the last rule in the sequence. The subscript of \( subelem_{xp1} \) is obtained by concatenating all the XPath expressions. Such “rule chaining” is similar to the “path concatenation” introduced in Section 3.6.1 for the VarTree homomorphism.
3.7 Extended MAC Mapping Procedure

If \( VT_2 \subseteq VT_1 \), then for any node of \( VT_2 \), say \( v \), its bindings are contained within the bindings associated with \( \Phi(v) \). Therefore, our VarTree homomorphism seems to be a sufficient condition for determining XQuery containment. However, this is under the assumption that there is no information loss during the result construction from the preserved variable bindings.

In fact, this is not always true since the return of \( v \)'s bindings may also depend on the bindings of some variable that is not an ancestor node of \( v \) in \( VT_2 \). In this section, we first investigate how such a dependency is induced. We then extend the MAC mapping to incorporate a validation step that deals with it.

3.7.1 Extended VarTree Structure

**Definition 3.8 (Control Flow Tree)** A normalized \( Q \) presents a nested FWR block hierarchy, which is representable by a control flow tree \( CFTree = (V_{ct}, E_{ct}, C_{ct}) \) with the following properties:

- each node \( n \in V_{ct} \) corresponds to a FWR block;
- each directed edge \((n, m)\) indicates the control flow from an enclosing FWR block to an enclosed one;
- each node \( n \in V_{ct} \) is associated with the set of where conditions \( c_1, \ldots, c_k \) \((c_1, \ldots, c_k \in C_{ct})\) that are imposed on the not-locally-defined variables\(^8\).

\(^8\)The conditions on the local variables are already incorporated into the corresponding variable definitions as the filter expressions.
For a given query, its CFTree structure is consistent with its TagTree but not exactly with its VarTree. The main differences between VarTree and CFTree are: 1) each node \( n \in V_{ct} \) represents a FWR block which may contain several variables, while each node \( n \in V_{vt} \) corresponds to exactly one variable; 2) each edge \( (v, w) \in E_{vt} \) represents a variable binding dependency \( v \triangleright w \). Variables \( v \) and \( w \) reside in the same FWR node or different FWR nodes which are ascendent and descendent but not necessarily parent and child; 3) \textit{where} conditions are more naturally associated with CFTree rather than with VarTree since each \textit{where} clause is hosted by a FWR block and the specified conditions affect all the variables defined in it.

Below we define another kind of variable dependencies induced by the CFTree structure.

**Definition 3.9 (Variable Region Dependency)** The operational semantics of a hierarchical structure indicates that a return expression is evaluated only if no context variable in \( \text{dynEnv} \) is bound to an empty binding set. We interpret this as implicit variable dependencies induced by the control flow structure to differentiate them from the explicit variable binding dependencies. Such dependencies are called variable region dependencies denoted by \( \leftarrow \). They exist in the following two cases:

- for a FWR node \( n \in V_{ct} \), if \( v_1, \ldots, v_k \) are defined in the \textit{f or} clause, then there are mutual dependencies amongst them. That is, if any variable is bound to empty binding set, then the down-flowing control that invokes the \textit{return} clause is stopped;

- for a variable \( v_i \) defined in a node \( n \in V_{ct} \), it is dependent on those defined in its ascendent nodes (i.e., enclosing FWR blocks).
3.7. EXTENDED MAC MAPPING PROCEDURE

We now extend VarTree to a hypertree [B. 98] which reflects the region dependencies in addition to the binding dependencies among the variables.

**Definition 3.10 (Extended VarTree)** For the VarTree of a normalized Q as defined in Definition 3.4, its structure is extended to VarTree = (\(V_{et}, E_{et}, L_{et}, HV_{et}, HE_{et}, HC_{et}\)) which possesses the following additional properties:

- each hypernode \(r_i \in HV_{et}\) contains one or more variable nodes in \(V_{et}\);
- each hypernode \(r_i \in HV_{et}\) is associated with the set of where conditions \(c_i, \ldots, c_k (c_i, \ldots, c_k \in HC_{et})\) on the not-locally-defined variables;
- each directed hyperedge \((r_i, r_j) \in HE_{et}\) indicates the control flow;
- for any two variables \(v, w \in V_{et}\), if they are contained in the same hypernode \(r_i \in HV_{et}\), then \(v \leftarrow w\) and \(w \leftarrow v\); if \(v\) and \(w\) respectively reside in different hypernodes \(r_i\) and \(r_j\) linked by a hyperedge, then \(v \leftarrow w\) assuming the hyperedge is directed from \(r_i\) to \(r_j\).

We can view such an extended VarTree as the result of combining the original VarTree and the CFTree. Each hypernode corresponds to a node in CFTree and each hyperedge corresponds to an edge in CFTree. The extended VarTree for the example query \(Q1\) is depicted in Figure 3.15. The bigger nodes marked with dashed circles denote the hypernodes, while the bold arcs represent the hyperedges. The additional variable region dependencies are indicated by the bold lines with arrow heads.

An intuitive observation from Figure 3.15 is that if there is a binding dependency between two variables \(u\) and \(v\), i.e., \(u \triangleright v\), then \(v\) can only be
3.7. EXTENDED MAC MAPPING PROCEDURE

Figure 3.15: Extended VarTrees and Implied Variable Region Dependencies

defined either in the same FWR block as $u$ or an enclosed FWR block of $u$’s, but not a sibling FWR block with $u$’s. Thus $u \leftarrow v$. However, if $u \leftarrow v$, then $u \triangleright v$ is not necessarily true since the definition locations of $v$ and $u$ do not indicate a binding dependency between them. We hence have the following proposition.

**Proposition 3.3 (Inference of Variable Dependencies)** Given two variables $u$ and $v$, if $u \triangleright v$, then $u \leftarrow v$. The other direction of inference is not true.

### 3.7.2 Additional Conditions for VarTree Homomorphism

With the presence of variable region dependencies implied by the extended VarTree, our VarTree homomorphism needs to consider two additional conditions which further validate each variable mapping pair $(v, w)$ with $w = \Phi(v)$ established based on Definition 3.6.

**Condition of Variable Region Dependency Mapping.** Suppose $VT1$ and $VT2$ are the extended VarTrees of $Q1$ and $Q2$ respectively. For a parent-child variable pair $u, v \in V_{vt2}$, if $\Phi(u) = x$ and $\Phi(v) = w$ ($x, w \in V_{vt1}$)
are established based on the conditions in Definition 3.6, and \( x \mapsto w \) (not necessarily \( x \triangleright w \) though), then \( \Phi(v) = w \) is valid only if \( u \mapsto v \) as well.

This is because if variable \( x \) is evaluated to be an empty binding set, then even if it may not be the case for \( w \), \( w \)'s bindings will not be returned due to \( x \mapsto w \). Another fact is that since \( x \) is the mapping node for \( u \) (i.e., \( \Phi(u) = x \)), the binding set of the former subsumes that of the latter. This implies that \( u \) is evaluated to be an empty binding set too. Suppose \( u \not\mapsto v \), then a non-empty binding set of \( v \) will be returned regardless of the empty binding set of \( u \). This means that \( \Phi(v) = w \) is not valid.

Below we add condition 4) into the requirements for VarTree homomorphism (see Definition 3.6).

4). if \( \Phi(u) \mapsto \Phi(v) \), then \( u \mapsto v \).

**Condition of Where-Condition Mapping.** For each established variable pair mapping, we also need to check its validity with the presence of where-conditions. That is, for each variable \( v \in V_{vt2} \), if it resides in a hypernode \( r_i \in HV_{vt2} \) which is associated with a set of *where* conditions denoted as \( C2 \) (\( C2 \subseteq HC_{vt2} \)), then \( \Phi(v) \in V_{vt1} \) resides in a hypernode \( r_j \in HV_{vt1} \) which is associated with a condition set \( \Phi(C2) \) (\( \Phi(C2) \subseteq HC_{vt1} \)) such that

5). \( \Phi(C2) \subseteq C2 \).

\( \subseteq \) here means that there is mapping from the predicates and variables in \( \Phi(C2) \) to those of \( C2 \). That is, for each semi-interval arithmetic comparison involved in \( \Phi(C2) \), there is a mapping comparison in \( C2 \) which asserts a more strict condition.
3.7. EXTENDED MAC MAPPING PROCEDURE

3.7.3 Put All Together: Extended VarTree Homomorphism

To sum it up, we give the definition for the extended VarTree homomorphism below.

**Definition 3.11 (Extended VarTree homomorphism)** Let $VT_1$ and $VT_2$ be the extended VarTrees of Q1 and Q2 respectively. An extended VarTree homomorphism from $VT_2$ to $VT_1$, denoted $VT_2 \subseteq VT_1$, if a total mapping function $\Phi$ from the nodes of $VT_2$ to those of $VT_1$ exists such that for all nodes $u$ and $v$ of $VT_2$:

1). $u$ is an ancestor of $v$ in $VT_2$ iff $\Phi(u)$ is an ancestor of $\Phi(v)$ in $VT_1$;

2). $\rho_2(v) \subseteq \rho_1(\Phi(v))$;

3). if $T(v) = VAR$, then $T(\Phi(v)) = VAR$ or $RET$; if $T(v) = RET$, then $T(\Phi(v)) = RET$;

4). if $\Phi(u) \leftrightarrow \Phi(v)$, then $u \leftrightarrow v$;

5). if $C_2$ and $\Phi(C_2)$ are the condition sets associated respectively with two hypernodes $r_i \in HV_{vt2}$ and $r_j \in HV_{vt1}$ and every variable node enclosed within $r_i$ is mapped to a variable node enclosed by $r_j$, then $\Phi(C_2) \subseteq C_2$.

We now give the main result of this chapter, i.e., the technique of containment checking for the XQuery fragment defined in Section 3.2 which guarantees the soundness of the containment conclusion.

**Theorem 3.2 (Extended VarTree Homomorphism for XQuery Containment)**

Given two normalized Q1 and Q2, $Q_2 \subseteq Q_1$ if an extended VarTree homomor-
phism \( VT_2 \subseteq VT_1 \) exists with \( VT_1 \) and \( VT_2 \) being extended VarTrees as defined in Definition 3.10. □

Proof. Suppose \( VT_2 \subseteq VT_1 \). This implies that the conditions 1)~3) required in Definition 3.6 and the two additional conditions 4)~5) described in Section 3.7.2 are all satisfied. This is a contained query case, i.e., \( Q_2 \subseteq Q_1 \), due to the reasons below. First, based on Proposition 3.2, we can derive that for each node \( v \) in \( VT_2 \), \( v \)'s binding set is subsumed by the binding set of \( \Phi(v) \). Second, since \( VT_1 \) and \( VT_2 \) both contain only essential variables, there is a one-to-one correspondence mapping between variables in \( VT_1 \) and \( TT_1 \) and between variables in \( VT_2 \) and \( TT_2 \) based on Proposition 3.1. This means that the query result containment can be reduced to variable binding set containment. Third, condition 4) guarantees that if there is a loss of information during the construction of \( Q_1 \)'s result from \( \Phi(v) \)'s bindings due to variable region dependencies, then the return of \( v \)'s bindings in \( Q_2 \)'s result is also blocked. Fourth, condition 5) ensures that all the conditions in \( Q_1 \) have mapping conditions and they are less strict than their corresponding ones. Therefore, it can be inferred that each new element constructed in the result of \( Q_2 \) has a mapping element in the result of \( Q_1 \), and that each variable binding returned in the result of \( Q_2 \) also has a mapping binding in the result of \( Q_1 \). Thus we derive \( Q_2 \subseteq Q_1 \). □

Similarly, we can also prove that a containment mapping between the variables and predicates of the two monadic datalog programs \( P_1, P_2 \) translated from \( VT_1 \) and \( VT_2 \) respectively can be used for determining query containment. The basic mapping translation is introduced in Sec-
tion 3.6.3, while condition 4) is translated to the mapping of Rs, i.e., the references to other head predicates, and condition 5) is translated to the mapping of Cs, i.e., the condition-related predicates.

The Extended MAC Mapping Procedure. To incorporate conditions 4)~5) into the previous MAC mapping procedure (given in Figure 3.13) and thus to realize the extended VarTree homomorphism, we modify the VarTree traversal process to take care of the additional properties of an extended VarTree. That is, for a given extended VarTree, the navigation from a hypernode to another is via the directed hyperedge and the navigation between variable nodes within a hypernode is via the variable binding dependency edge. This corresponds to a top-down traversal of the nested FWR blocks and check for variable mappings.

Such a traversal asserts a stricter order than the previous traversal order as described in Figure 3.13. However, it does not conflict with the previous one due to Proposition 3.3.

In addition, we add the checking steps for conditions 4)~5) to validate the established mapping \( \Phi \). The main idea exploited to achieve this is to aggregate the mapping nodes in \( VT_1 \) for the variables residing in each hypernode of \( VT_2 \) and then check if more variable region dependencies or more strict conditions can be induced from this aggregated set than from the original variable set.

Below the extended MAC mapping procedure is sketched in Figure 3.16.
3.7. EXTENDED MAC MAPPING PROCEDURE

ALGORITHM Extended_VarTree_MAC_Mapping

INPUT: Extended VarTrees VT1 and VT2.
OUTPUT: Variable mapping pair set \( MS = \{ \langle v, \Phi(v) \rangle \} \).

Traverse VT2 in preorder

Foreach encountered hypernode \( r_i \in HV_{vt2} \) Do

  Initialize \( V_{ri} = \{ v \mid v \text{ resides in } r_i \} \), \( S_{ri} = \emptyset \), \( R_j = \emptyset \).

  Foreach variable node \( v \) enclosed in \( r_i \) Do

    Check if \( parent(v) \) has a mapping, continue only if one exists;
    Search for \( v \)'s mapping node \( w \) within the scope of the subtree rooted at \( \Phi(parent(v)) \) such that \( w \) satisfies:
    1). \( \rho_2(v) \subseteq \rho_1(w) \)
    2). if \( T(v) = \text{VAR} \), then \( T(w) = \text{VAR} \) or \( \text{RET} \);
        if \( T(v) = \text{RET} \), then \( T(w) = \text{RET} \).

    If such a \( w \) exists Then
    
      establish a mapping pair \( \langle v, w \rangle \);
      Suppose \( w \) resides in a hypernode \( r_j \in HV_{vt1} \), then
      \( S_{ri} = S_{ri} = \{ u \mid u \in r_j \} \), \( R_j = \{ r_j \} \);
    
    End If
  
End Foreach

If \( V_{ri} = S_{ri} \) Then

  \( \langle v, w \rangle \) is valid and hence added into the mapping set \( MS \);

End If

End Foreach

If there exists a total mapping from variables in VT2 to those in VT1 Then

  output VT2 \( \subseteq \) VT1 and the mapping set \( MS \);

End If

Figure 3.16: The Extended MAC Mapping Procedure
Chapter 4

XQuery Rewriting

4.1 Issues in XQuery Rewriting

If a new XQuery $Q_2$ is determined to be contained in or overlap with a view XQuery $Q_1$, we then need to formulate the probe query of $Q_2$ with respect to the result view structure of $Q_1$ so that the (total or partial) answer to $Q_2$ can be directly retrieved from the cached view result of $Q_1$. Intuitively, this would involve rewriting the bound expressions of $Q_2$’s variables with respect to the view structure of $Q_1$.

4.1.1 Intuitive Idea for Rewriting

As shown in Figure 3.16, the extended MAC mapping procedure takes two queries $Q_1$ and $Q_2$ as input and outputs the maximal set of variable mapping pairs. If each variable in $Q_2$ has a mapping variable in $Q_1$, then $Q_2$ is totally contained in $Q_1$ (i.e., $Q_2 \subseteq Q_1$). Otherwise, if it is not a total mapping from variables in $Q_2$ to those in $Q_1$ and the mapping set is not empty,
then $Q_2$ is partially overlapping with $Q_1$. In either case, if the extended MAC mapping procedure has established a mapping for variable $v$ in $Q_2$, the bindings of $v$ can be obtained from the result of $Q_1$. We therefore propose a technique for rewriting $Q_2$ with respect to the view structure of $Q_1$ by utilizing these variable mapping pairs. The result of such a rewriting is the probe query of $Q_2$.

For the overlapping case, we also need to consider the issues of formulating the remainder query as well as combining its results with that of the probe query. Unlike other XML merging work [Man01] which either assume or impose object identifiers throughout the source documents to make it possible merging two pieces, we utilize primarily the DTD knowledge and user designated key constraints for identifying the node equivalency. The concept of relative key path proposed by [aDWFH01] is exploited to allow the source subscriber to specify such key constraints. In this section, with the focus on the query rewriting technique with respect to the view structure of the cached query, we refer the readers to [LCR03] for the details of the approach for generating the remainder query and merging its result with that of the probe query.

For the query rewriting problem, the question is if we can rewrite the original bound expressions of variables in $Q_2$ in terms of the paths navigated to where their corresponding mapping variables are located in the view structure of $Q_1$. Since all the mapped variables in $Q_1$ are the essential ones (see the definition of essential variables in Section 3.3), they are definitely utilized by the result construction of $Q_1$ and hence should appear in $Q_1$’s TagTree $TT_1$. 
Suppose for a variable $v$ in $Q^2$’s VarTree $VT^2$, the MAC mapping procedure generates a mapping pair $\langle v, w \rangle$, namely $\Phi(v) = w$. We can locate $w$ on an edge of $VT1$ and use the complete navigation path to it as the rewritten form of the XPath expression in the bound expression of $v$.

The above rationale forms the rough idea of our rewriting technique. This idea utilizes the variable mapping pairs produced by the MAC mapping procedure and the view query $Q1$’s TagTree to derive the rewritten query for $Q^2$.

However, this intuitive idea for rewriting does not always work. We will show these failing cases and then explain the reasons in Section 4.1.3. We will give an XQuery rewriting procedure in Section 4.2.

4.1.2 TagTree Revisit

Let’s first revisit TagTree and its implied semantics. Given a TagTree $TT$, the result view structure can be inferred from the information carried by the nodes and edges. This has a two-fold meaning.

First, each node represents an element constructor. Each edge annotated with variable(s) indicates the correspondence between new element constructions and variable bindings. For example, if a node in $TT$ contains the tag name $\langle t1 \rangle$ and the edge incident to it is annotated with variable $v$, then for each data binding of $v$, a new element with its tag name $\langle t1 \rangle$ is created. Thus, a node and the edge incident to it represent the relationship between an element constructor and its corresponding variable.

Second, the nested structure presented by the nodes of $TT$ implies the enclosing-enclosed relationships among their corresponding new elements.
That is, a new element constructed at the node containing \langle t1 \rangle encloses the elements constructed at its descendant nodes.

**Constraints on TagTree Nodes and Edges.** Some constraints on \( TT \) are worth mentioning. One constraint is that each node can contain none, one, or a list of tag names in a form like \langle t1 \rangle \langle t2 \rangle \ldots \langle tn \rangle. If a node contains a single tag name, then an element with that name will be constructed upon the invocation of each variable binding. When a node contains no tag name, then no explicit element is created for the corresponding variable binding. However, the new elements constructed at the descendant level nodes are still grouped, although implicitly, by each variable binding. In the case when a node contains a list of tag names such as \langle t1 \rangle \langle t2 \rangle \ldots \langle tn \rangle, an aggregated element with its tag name being \langle t1 \rangle \langle t2 \rangle \ldots \langle tn \rangle is created upon each binding invocation.

Another constraint is that each edge in \( TT \) is annotated with one or more variables. If an edge is annotated with a single variable, then a new element is created corresponding to each variable binding. When an edge is labelled with more than one variable, then the cartesian product of all these variable bindings results in a set which is iterated for each resulting tuple to invoke the construction of a new element.

### 4.1.3 Intuitive Idea Invalidated

When the intuitive idea is directly applied for query rewriting, it may result in a wrong rewriting due to the following reasons.

First, variable binding dependencies presented in \( Q2 \) are not reflected
in the rewritten $Q_2$. The query rewriting technique needs to respect the original variable binding dependencies and preserve them in the rewritten form. Thus, the bound expression of a variable should not be rewritten as an absolute path but a relative one prefixed by its parent variable.

Second, path rewriting for each variable’s bound expression in $Q_2$ may cause the problem of "product". That is, in the case when more than one variable is part of the label on an edge $e$ in $TT1$, the construction of a corresponding new element is for each tuple in the cartesian product of these variable bindings. Suppose variables $v$ and $w$ are defined in the same for clause in $Q_2$ and they both are mapped to variables on $e$, then the rewritten paths for re-defining them will be the same, and the new elements constructed for their binding product will hence be the product-of-the-product of $v$’s and $w$’s bindings. This is semantically wrong.

This raises the question whether the query rewriting is composed of the path rewritings of individual variables or of some variable groups. More concretely in this example, the question can be rephrased as: should we rewrite the bound expressions for $v$ and $w$ separately, or together? Our observation is the rewriting of a query is not simply composed of the rewritings for its individual variables, but may need to aggregate variables to form groups for which to rewrite.

4.1.4 Interpretation of TagTree Semantics

To tackle these issues in order to derive a correct rewriting of $Q_2$, below we interpret the implied semantics of the internal nodes and leaf nodes of a TagTree.
TagTree Internal Node Semantics. For a given TagTree, each internal node represents an element constructor which can be interpreted by a *skolem function* that composes an implicit identifier for each new element constructed corresponding to a variable binding.

We first introduce the semantics of skolem functions. Skolem functions were first introduced in the context of object-oriented databases by Maier [D. 86]. They were later refined in [S. 89] for F-logic [M. 93b] and in [R. 90] for ILOG. It is observed in [S. 89] that the skolem function based set formation can replace explicit grouping operators.

For a skolem function, the input is a list of arguments such as variables and the output are the generated new object identifiers. Given a skolem function such as newOid(x1, ..., xn), the functional dependency relationship between the input variable arguments and the output element is implied. If provided with no argument, the function newOid() creates a new object associated with a new identifier upon each invocation. With more than one argument x1, ..., xn, the function creates a new object for every distinct cartesian product of the values of all variables. Hence each new object and its identifier depend on the values of x1, ..., xn.

In XML-QL, skolem functions are explicitly employed for automatically generating the id attribute value for each element. In XQuery, although no skolem function is explicitly used, we can infer one for each element constructor. The skolem functions we use are in the form of \( \langle \text{tag}_i \rangle(x_1, \ldots, x_n) \), where \( \text{tag}_i \) is the tag name of the new elements and \( x_1, \ldots, x_n \) are the input variables that the element construction is based on.

For example, the skolem functions corresponding to the two internal
nodes in $TT1$ (see Figure 3.11) are $\langle Q1res \rangle ()$ and $\langle entry \rangle ($$b$) respectively. For the three internal nodes in $TT2$ (see Figure 3.12), their skolem functions are $\langle Q2res \rangle ()$, $\langle item \rangle ($$b$), and $\langle pair \rangle ($$t$, $a$). Below we explain the semantics conveyed by the skolem function attached to each internal node in $TT2$.

- The function $\langle Q2res \rangle ()$ is associated with the root node in $TT2$, whose binding set is a singleton – the root element of the source document. Hence, $\langle Q2res \rangle ()$ is invoked once and outputs a new $\langle Q2res \rangle$ element as the root element of the result tree.

- The internal node right below the root node in $TT2$ is interpreted by the skolem function $\langle item \rangle ($$b$). This means that a $\langle entry \rangle$ element is constructed for each binding of $b$.

- The internal node on the third level of $TT2$ is interpreted by the skolem function $\langle pair \rangle ($$t$, $a$). Two variables $t$ and $a$ are the input arguments. This expresses a cartesian product of the bindings of $t$ and $a$. Corresponding to each result tuple of this product, a $\langle pair \rangle$ element is constructed.

- The element constructions are conducted in a nested fashion, i.e., new elements constructed corresponding to an internal node in $TT2$ are nested within those constructed at the parent node.

Hence, we can view the result construction semantics represented by a TagTree as a result tree creation process by utilizing the variable bindings generated during the pattern matching phase.
**TagTree Leaf Node Semantics.** Each leaf node in a TagTree has an incoming edge labelled by a return expression. The semantics can be interpreted by a “deep-returning” function that returns the obtained bindings together with the subtrees rooted at them.

In terms of implementation, this “deep-returning” function can be implemented conforming to either the reference semantics or the copy semantics. The former means that the references to the original elements constitute the result. This way, the original binding ids, if exist, are not changed. These operational semantics thus allow the support for the definition and management of views over the source XML. In contrast, the copy semantics mean that deep recursive copies of subtrees from the input XML tree instead of the references to them are returned to constitute the result. These copied elements are actually new nodes with fresh ids, hence losing the ties with their originating XML data. In our context, we can ignore the implementation details and assume a general “return” semantics for the TagTree leaf nodes.

### 4.2 The XQuery Rewriting Approach

As explained in Section 4.1.2, the result construction semantics of a query, i.e., the restructuring logic that maps the variable bindings extracted from the source document to the query result composed of new elements and content, is captured by the TagTree.
4.2.1 The Key Idea: Translating Restructuring Logic

Given a cached query $Q_1$ and a new query $Q_2$, and their variable mapping pairs generated from the containment checking stage, the key of rewriting $Q_2$ with respect to $TT1$ lies in a correct translation of $Q_2$'s restructuring logic such that it accepts the input of the restructured bindings from $Q_1$’s result. In this section, we show that this translation requires a mapping between the edges in the TagTrees of the two queries. We also give the rewriting procedure which exploits such edge mappings and the variable mappings derived in the containment checking phase.

Since skolem functions and “deep returning” functions are respectively implied by the internal and leaf nodes of a TagTree, we tackle the query rewriting problem by rewriting these functions involved. Rewriting a skolem function corresponds to the rewriting of a for clause, while rewriting a “deep returning” function corresponds to rewriting a return expression. The hierarchical skeleton of element constructors remain the same.

4.2.2 Translation of Skolem Functions

As explained before, a skolem function may have none, one, or multiple input variable arguments, for which the respective element construction semantics are a single element, one element for each variable binding, and one element for each binding resulted from the cartesian product of variable bindings. For simplicity, we may consider the third case as a general case and the other two cases as its special case. That is, we consider a skolem function attached to the TagTree as a relational cartesian product.
In this sense, a stand-alone skolem function can be seen as a relational view specification that produces a table of new elements by conducting the product operation with the input single-column relation(s), each containing the bindings of an input variable. This way, we can equate the problem of skolem function rewriting with the problem of relational query rewriting. That is, we replace each skolem function \( f \) associated with an internal node of \( TT_2 \) with a set of skolem functions \( g_1, \ldots, g_k \) involved in \( TT_1 \), such that there exists a mapping \( \Phi \) (established by the containment checking technique introduced in Chapter 3) from each input variable argument of \( f \) to one of the input variables of \( g_1, \ldots, g_k \).

For example, the skolem function \( \langle \text{pair}\rangle(\$t, \$a) \) implied by \( Q_2 \) has two input variable arguments \( \$t \) and \( \$a \). Since there is no total containment mapping from \( VT_2 \) to \( VT_1 \), we do not consider rewriting \( Q_2 \) with respect to \( Q_1 \). However, suppose there is another query \( Q_3 \), for which we find out \( VT_2 \subseteq VT_3 \) (\( VT_3 \) is the extended VarTree of \( Q_3 \)) and \( \Phi(\$t^{Q_2}) = \$t^{Q_3}, \Phi(\$a^{Q_2}) = \$a^{Q_3} \). In addition, two skolem functions \( \langle \text{name}\rangle(\$t) \) and \( \langle \text{writer}\rangle(\$a) \) are associated with these two internal nodes \( \$t \) and \( \$a \) in \( TT_3 \). Then we can rewrite \( \langle \text{pair}\rangle(\$t, \$a) \) as \( \langle \text{name}\rangle(\$t), \langle \text{writer}\rangle(\$a) \).

This means each new \( \langle \text{pair} \rangle \) element corresponds to the cartesian product of \( \langle \text{name} \rangle \) and \( \langle \text{writer} \rangle \) elements in \( Q_3 \)'s result. The number of skolem functions used in the rewriting indicates the number of variables needed to be defined in the rewritten for clause.

Next, we will first give the approach called TagTree edge mapping for deriving rewritings of skolem functions. Then we explain how to translate the skolem function rewriting as for clause rewriting.
4.2. THE XQUERY REWRITING APPROACH

Definition 4.1 (TagTree Edge Mapping) Suppose for two queries $Q_1$ and $Q_2$, $VT_2 \subseteq VT_1$, then there exists an edge mapping $\Omega$ from the edges in $TT_2$ to those in $TT_1$ such that it satisfies:

- for each edge $e \in E_{TT_2}$, $\Omega(e) = \{e_i, \ldots, e_k\}$ with $e_i, \ldots, e_k \in E_{TT_1}$;
- $\Phi(V) = V_i \cup \ldots \cup V_k$ with $V$ the set of variables annotated on $e$, and $V_i, \ldots, V_k$ the sets of variables annotated on $e_i, \ldots, e_k$ respectively). That is, there exists a containment mapping $\Phi$ from variables annotated on $e$ to those annotated on the mapped edges.

From the definition, we can see that this edge rewriting is achieved by exploiting both variable mapping pairs and the structure of $TT_1$. Since each edge corresponds to a skolem function, the rewriting of skolem function utilizes this edge mapping for finding the right replacement skolem functions in $Q_1$.

Also note that $\Omega$ is a one-to-many mapping. Namely, each edge $e$ in $TT_2$ can be mapped to more than one edge in $TT_1$. The number does not depend on the number of variables annotated on $e$, but on how they are distributed on the edges in $TT_1$. This implies that the rewriting of $Q_2$ is not simply the rewriting of each of its variables, but of certain variable groups based on this edge mapping.

Rewriting For Clauses. Based on the established edge mappings, the rewriting of the for clauses in $Q_2$ takes the following two steps. First, since the local variables defined in each for clause in $Q_2$ are all annotated on an edge $e$ of $TT_2$ (according to Definition 3.5), we divide them into subgroups each
containing the variables annotated on one of the mapping edges of \( e \). Second, we define a new variable corresponding to each variable subgroup. For a variable subgroup resides on the edge \( e \) in \( TT1 \), we use the navigation path conforming to \( e \) as the bound expression of the new variable which replaces the original variables within this subgroup. This bound expression is prefixed by some new variable that replaces the variable subgroup corresponding to \( e \)'s parent edge (i.e., the edge whose ending node is where \( e \) originated from).

### 4.2.3 Translation of Deep Returning Functions

To rewrite return expressions in \( Q2 \) which correspond to the leaf nodes of \( TT2 \), we need to map the leaf nodes in \( TT2 \) to those in \( TT1 \) first. Since each return expression corresponds to a leaf node in the TagTree (according to Definition 3.5), the leaf node mapping must be one-to-one.

Since the edge incident to each leaf node is not annotated with variable(s), but the relative path component of the corresponding return expression, we identify a mapping leaf node pair by checking if the following two conditions are satisfied by their corresponding incoming edges \( e2, e1 \):

1) \( e2, e1 \) are annotated with the same relative path expression; 2) there exists an edge mapping \( \Omega(parentEdge(e2)) = parentEdge(e1) \).

If both conditions are satisfied, then the two leaf nodes are considered as a mapping pair. For the return expression corresponding to the leaf node \( n \) in \( TT2 \), we now rewrite it by simply replacing its prefixing variable with the new variable created to replace the variable subgroup annotated on the parent edge of \( n \).
4.2.4 Overall XQuery Rewriting Procedure

Given a new query \( Q_2 \) and a cached query \( Q_1 \), suppose the MAC mapping procedure determines \( VT_2 \subseteq VT_1 \) and produces a containment mapping \( \Phi \) via which each variable defined in \( Q_2 \) is mapped to a variable in \( Q_1 \). We now are ready to give the procedure for rewriting \( Q_2 \) with respect to the result structure of \( Q_1 \) (see Figure 4.1) by exploiting \( TT_1 \) and \( TT_2 \), as well as the set of variable mapping pairs.

Next we show a query rewriting example by applying the proposed rewriting procedure. In this example as shown in Figure 4.2, \( Q_2 \) is the same example query \( Q_2 \) in Figure 3.1 but \( Q_1' \) is not the same as the example \( Q_1 \) in Figure 3.1. In fact, \( Q_1' \) extends \( Q_1 \) with a return expression $b/price$ and changes the return clause \( return \: $a/last \) into \( return \: ⟨name⟩$a/last, $a/first⟩⟨/name⟩ \). The established variable mappings from nodes in \( VT_2 \) to \( VT_1' \) are indicated by the dashed lines with arrow heads, while the correspondence mapping between variables in \( VT_1 \) to \( TT_1' \) are denoted by solid lines. The XPath component in the bound expression for variable $b$ in \( Q_2 \) is rewritten as \( /Q1res/entry \), since \( \Phi(⟨b⟩Q_2) = ⟨b⟩Q_1' \) and \( ⟨b⟩Q_1' \) can be located in \( TT_1' \) via this navigation path. Similarly, the path expression for defining variable $a$ in \( Q_2 \) is rewritten as \( $b/name \). These two rewritten bound expressions are underlined in Figure 4.2.
4.2. THE XQUERY REWRITING APPROACH

PROCEDURE XQuery Rewriting

INPUT: The original $Q_2$, its TagTree $TT_2$ and that of $Q_1$ $TT_2$; and a set $S$ containing the variable mapping pairs.

OUTPUT: Rewritten $Q_2$ with respect to the view structure of $Q_1$.

 Traverse $TT_2$ in preorder, and

\textbf{Foreach} encountered edge $e$ in $TT_2$ \textbf{Do}

Locate the corresponding FWR block $fr$ in $Q_2$;

// Step1: rewrite the $for$ clause of $fr$

Extract the variable set $VS$ annotated on $e$;

Find a group of edges $G$ in $TT_1$, such that

$VS = \Phi(VS), VS$ collects all the variables annotated on the edges in $G$;

\textbf{Foreach} edge $e_i$ in $G$ \textbf{Do}

Extract the variable set $VS_{1i}$ annotated on $e_i$;

Form a variable subgroup $VS_i$ of $VS$, such that $VS_i = \Phi^{-1}(VS_{1i})$;

Generate a new variable $v_i$ to replace the definitions of variables $VS_i$ in $fr$;

Rewrite the bound expression for $v_i$ as described in Section 4.2.2;

End Foreach

// Step2: retain the original element constructor in the $return$ clause of $fr$;

// Step3: rewrite the return expressions enclosed within the $return$ clause of $fr$;

\textbf{If} $e$ is an edge incident to a leaf node \textbf{Then}

Retrieve the relative path component $p$ annotated on $e$;

Locate the corresponding return expression $re$ in $fr$;

Find a leaf edge $e_1$ in $TT_1$ which is annotated with $p$;

Rewrite $re$ using the technique introduced in Section 4.2.3;

End If

End Foreach

Figure 4.1: The XQuery Rewriting Procedure
Figure 4.2: First Rewriting Example

```xml
<Q2res>
{for $b in doc("Q1view.xml")/Q1res/entry[price<40]
return <item>
  {for $t in $b/title, $a in $b/name
    return <pair> {$t} {$a/last }
    </pair> }
</item>} </Q2res>
```
Part III

Cache Replacement
Chapter 5

Background on Replacement Strategies

5.1 Basics in Cache Management

5.1.1 The Importance of Replacement and Its Metrics

Caching is one of the most fundamental and popular metaphor in modern computing. It is widely used in storage systems, databases, web servers, middleware, processors, file systems, disk drives, operating systems, and numerous other applications.

The cache is assumed to be significantly faster than the secondary device, but is also significantly more expensive. Hence, the cache space is usually considered a limited resource and designed to be only a fraction of the size of the secondary storage. Upon the incoming of a stream of requests, the available space of a cache will be run out at some point. For
a full cache, before a new page (or answers to a request if it is a semantic cache) can be brought into the cache, some old pages (or answers to the old requests) must be purged out. The victim pages (or request answers) are selected using a cache replacement policy.

The most important metric for a cache replacement policy is its hit rate: the fraction of pages (or answers to requests) that can be served from the cache instead of from the remote storage system. Another important metric for a cache replacement policy is its overhead. It is desirable that the overhead is low. The problem of cache management is to design a replacement policy that maximizes the hit rate over a request trace while satisfying the practical constraints of the computational and space overhead involved in implementing the replacement strategy.

### 5.1.2 Traditional Replacement Policies

Various replacement schemes have been studied [RD90, NNW93, JS94, CFL94, PG95, LCN+99] during the past few decades. Yet, the Least Recently Used (LRU) replacement scheme is still widely used due to its simplicity. While simple, it adapts well to the changes of the workload, and has been shown to be effective when recently referenced objects are likely to be re-referenced in the near future [CD73]. A main drawback of the LRU scheme, however, is that it cannot exploit regularities in region accesses such as sequential and looping references and thus yields degraded performance [JS94] in such cases. Instead of using recency as the parameter for replacement, the Least Frequently Used (LFU) replacement scheme [RD90] uses reference frequency. LFU assumes that the more often a query is being used to an-
swer sequential queries, the more likely it is to be used to answer a future query. However, a potential hole of LFU is that some data may accumulate its use to a high number and then is never used again. When applying the LFU scheme, such data tend to be more difficult to purge to free space for other useful data.

The varieties of the LRU and LFU schemes include the LRU-K scheme [NNW93], the IRG scheme [PG95] and the LRFU scheme [LCN⁺99]. The LRU-K scheme bases its replacement decision on the regions’ kth-to-last reference. The IRG scheme [PG95] considers the inter-reference gap factor and the LRFU scheme [LCN⁺99] considers both the recency and frequency factors. These schemes, however, show limited performance improvements because they do not consider regular references such as sequential and looping references. Some researchers have proposed other replacement schemes oriented to the reference regularities. For example, the 2Q scheme [JS94] can quickly remove from the cache sequentially-referenced blocks and looping-referenced blocks with long loop periods. The SEQ scheme [PG95] detects long sequences of page faults and applies the Most Recently Used (MRU) scheme to those pages.

In the Web context, other replacement functions have been proposed to address the size and latency concerns. Among them, GreedyDual [Yon91] is a simple yet popular algorithm which handles variable-cost cache replacement. One of its extended versions GreedyDual-Size [CI97] combines locality, size and latency cost concerns to achieve a better performance in terms of hit ratio and latency reduction.
5.2 Comparisons of Semantic and Page/Tuple Caches

One major difference between semantic caching systems [DFJ96, HKU99, CFZ94] and the traditional tuple [DFMV90] or page-based [CFZ94] caching systems is that the data cached at the client side of the former is logically organized by queries instead of physical tuple identifications or page numbers. To achieve effective cache management, the access and management of the cached data in a semantic caching system is thus typically at the level of query descriptions. For example, the decision of whether the answers of a new query can be retrieved from the local cache is based on the query containment analysis of the new query and the cached query descriptors themselves, rather than by looking up each and every tuple or page identification of objects that could possibly answer a current user request.

The semantic caching idea has been extensively studied in the relational context [DFJ96]. However, query evaluation and containment dealing with XML data differ in their nature and difficulty from those in the relational setting. New challenges are being imposed by the tree-oriented nature of XML and the XQuery language on the tasks of query containment and rewriting, as we will point out in this dissertation.

In a query-based caching system, the data granularity for replacement is the query and its associated query result. The cache manager in a semantic cache system maintains a collection of query regions, each composed of a query descriptor and the corresponding XML view document, i.e., query region = query descriptor + result XML view. Query descriptors can be utilized for reasoning about the containment relationships between the cached queries.
and the new query. Also, user access statistics information may be attached to the query descriptors by the deployed replacement strategy to calculate the region utility values. The replacement manager usually picks the cached query with the lowest utility value and purges it to make room for the new query.

The differences between the semantic caches and page/tuple caches are summarized in Table 5.1.

<table>
<thead>
<tr>
<th></th>
<th>Data Granularity</th>
<th>Data Miss</th>
<th>Cache Replacement</th>
</tr>
</thead>
<tbody>
<tr>
<td>Page Cache</td>
<td>page</td>
<td>faulting</td>
<td>temporal locality (LRU,LFU,MRU)</td>
</tr>
<tr>
<td>Tuple Cache</td>
<td>tuple</td>
<td>faulting</td>
<td>spatial locality (clustering)</td>
</tr>
<tr>
<td>Semantic Cache</td>
<td>query descriptor</td>
<td>remainder queries</td>
<td>semantic locality (replacement value)</td>
</tr>
</tbody>
</table>

Table 5.1: Comparisons between Semantic and Page/Tuple Caches
Chapter 6

Replacement for XQuery-based Semantic Cache

6.1 The Total Replacement Strategy and Its Drawbacks

6.1.1 Alternative Cache Region Management Schemes

Since a new query is often conceptually subsumed by or overlapping with previously cached queries, the query region of the latter can be seen logically segmented into two pieces. One corresponds to the overlapping part which is to be retrieved by the probe query for answering the new query. The left-over piece does not contribute to answering the new query. The replacement manager of a traditional query-based caching system may split the containing query region into two regions corresponding to their respective usefulness in this latest query answering process. After the splitting, a uniform utility value is then maintained for each query region. When-
ever the cache is full, a complete query region would be the unit for replacement. However, such a region-splitting scheme entails a large decomposition overhead each time when a new query overlaps with the cached queries. Also, it would result in more and more smaller XML view documents over time which are possibly less useful in answering future queries due to their fragmentation.

An alternative solution is to tolerate some redundancy in the cached queries. That is, even if newly incoming queries partially overlap with existing queries, we would opt to not split existing queries in order to avoid fragmentation. Then a straightforward application of replacement would be to replace a complete query region at each iteration. However, the data granularity of a whole query region being deleted each time in such a replacement strategy may be too coarse for “large” XML views. This would impact the cache space utilization. Also, such a replacement strategy doesn’t reflect the contribution of different fragments in a cached XML view which may participate in answering different subsequent queries. Replacement at the granularity of complete XML views hence suffers apparent drawbacks.

6.1.2 Total Replacement at the Query Level

In traditional query-based caching systems, a query region is the minimal granularity managed in the cache. A query region consists of an encoded query descriptor and a pointer to access the associated result view. Here, we take a look at existing alternative schemes and compare them in their ways of managing the query regions for replacement.
When a new query arrives, the containment mapper will first determine if it is contained or partially overlapping with a cached query. If yes, a probe query $PQ$ is formulated to access the cached data which satisfies the new query and thus will contribute to the answer. If not all the desired data requested by the new query is available in the cache, a remainder query $RQ$ will also be sent to the remote servers to fetch the rest of the answer. In this sense, query regions may logically be segmented by probe queries upon the arrival of new queries. Below we describe several possible schemes proposed by [DFJ96, KB96] for maintaining such query regions.

One region management scheme is to allow redundancy between query regions. In such a scheme, query regions are never adjusted once they have been formed. They are not split even if the subsequently cached queries overlap with them. For each such cached query, one uniform utility value is maintained that assesses the perceived usefulness of the query region to the users of the system. We refer to this as the *region-preserving* scheme.

Another way of managing the query regions is to split a cached query $Q1$ into two regions upon the arrival of a new query $Q2$. One corresponds to the part utilized by the probe query $PQ2$ for answering the new query, and the other, represented by $Q1 - PQ2$, corresponds to the part not usable for answering the new query. The region $Q1 - PQ2$ inherits its utility value from its parent region from which it was split off. The region $PQ2$ is marked with an increased utility value compared to the original cached query to indicate its contribution in answering this current query. This process is shown in Figure 6.1.

Alternatively, the query regions for the earlier cached queries may be
6.1. THE TOTAL REPLACEMENT STRATEGY AND ITS DRAWBACKS

1. Q1

2. Q1, Q2

3. Q1, Q2, Q3

Figure 6.1: Pictorial Illustration of the First Region-splitting Technique

preserved as a whole. In this scheme, a new region is allocated to capture only the remainder query $RQ2$ since $PQ2$ is already contained in the existing query region $Q1$. The XML view content of this new region $RQ2$ represents the “net” increase of information obtained by the new query. This can equivalently be seen as a process of splitting the newly incoming query region first and then caching only the non-redundant portion of the region as a new region. This process is shown in Figure 6.2. The utility values of each region will be increased every time a hit occurs.

1. Q1

2. Q1, Q2

3. Q1, Q2, Q3

Figure 6.2: Pictorial Illustration of the Second Region-splitting Technique

In the latter two scenarios, in effect the region-splitting scheme is applied. This helps to reduce the cache redundancy. However, the first region-splitting scheme of managing regions tends to result in too many small re-
6.2. THE PARTIAL REPLACEMENT STRATEGY

Region fragments over time which tend to be less useful in answering future queries. Also, such a scheme entails the overhead of query region splitting each time when a new query is launched. Hence, it may have to resort to frequent coalescing to make up for the fact that the cache space has been severely fragmented over time.

The second region-splitting scheme avoids the coalescing computation overhead compared to the first splitting schema. However, it has its drawbacks as well. First, the uniform utility value assigned for the whole region does not precisely indicate the various contributions made by different region fragments in answering subsequent queries. Second, a straightforward application of a replacement strategy would replace a complete region at a time. Such a replacement unit may likely be too coarse grained, requiring us to remove potentially huge sets of XML elements even when only a small space is needed in the cache. This would result in less efficient cache space utilization.

6.2 The Partial Replacement Strategy

6.2.1 Overview of Our Partial Replacement Approach

We now propose a refined replacement strategy, namely, to record utility values for finer regions of existing cached views in terms of their internal structure rather than assigning a uniform value for the whole cached query region [CWR02]. To be precise, we attach to each query descriptor a detailed path table listing all paths returned in the query. When a cached query contains or partially overlaps with a new query, the utility statistics
of those paths requested by the probe query are updated, however without splitting the cached query. When the cache is full, the replacement manager does not select complete regions but only specific paths with the lowest utility value within such query regions for replacement. It then composes a *filter query* to remove the fragments corresponding to those paths from the cached XML view. The relevant query descriptors are then modified accordingly to be consistent with the changed XML view.

This proposed *partial replacement* strategy utilizes the view structure to maintain utility values at a finer granularity than complete query regions. This way, the replacement helps to maintain in the cache the most likely “hot” query regions. This is because the original cached queries may be refined by future filter queries that remove the less useful fragments within them. It hence forgoes the explicit region splitting upon every new incoming query, avoiding the generation of too many small region fragments with little use for answering future queries.

We have also implemented both the proposed *partial replacement* strategy as well as the complete region replacement strategy (which we now call *total replacement*) within our ACE-XQ caching system. In this dissertation, we now report upon the extensive experimental study we have conducted to compare the performance of our partial replacement and the alternative total replacement strategies in a variety of scenarios. The results show that in most cases especially when the cache size is medium, the partial replacement strategy outperforms the total replacement strategy in terms of hit count ratio, hit byte ratio and query response delay.
6.2. THE PARTIAL REPLACEMENT STRATEGY

6.2.2 Query Descriptor Hierarchy

To overcome the drawbacks identified above of naive region-splitting replacement strategies, we instead suggest here that different utility values may be maintained for finer parts within a given region to account for different levels of accesses by users. This then should be done independently from the final decision of splitting the region. When a new query overlaps with a cached query region, the overlapped portion in the cached region is accessed by the return expressions of the probe query. They correspond to the data objects associated with certain paths in the region. To record such accesses of certain parts of a cached region, we extend each query descriptor with a utility tracking table containing all complete path expressions in the corresponding XML view document. Each path expression corresponds to a row in this path table, referred to as XPathRow.

The XPathRows of such a path table can be easily constructed based on the return expressions in a query. We simply enumerate all the complete paths from the root of the view document to the leaf element types in the view schema and use them as XPathRows. For example, the type inferred for a return path expression $/b/author$ in Q1 is Author which contains two leaf element types Last and First. Therefore, two XPathRows /bklist/entry/author/last and /bklist/entry/author/first corresponding to these types are listed in our XPathRow table. All the other XPathRows in the path table are complete paths appearing in the view schema of Q1. The statistics related to the user access information are now no longer associated with the complete region, but more precisely with the specific
6.2. THE PARTIAL REPLACEMENT STRATEGY

XPathRows. The different types of statistics such as hit frequency, last access timestamp and etc., are all associated with each XPathRow. With the utilization of the path table, we maintain the user access statistics at the granularity level of XPathRows for each cached query. Figure 6.3 displays a snapshot of the extended query descriptor of Q1.

<table>
<thead>
<tr>
<th>Q1</th>
<th>XPathRow</th>
<th>hits</th>
<th>last_access_time</th>
<th>obj_bytes</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>/bklist/entry/@year</td>
<td>1</td>
<td>12:33pm May 30</td>
<td>1600</td>
</tr>
<tr>
<td></td>
<td>/bklist/entry/title</td>
<td>1</td>
<td>12:33pm May 30</td>
<td>2100</td>
</tr>
<tr>
<td></td>
<td>/bklist/entry/author/last</td>
<td>1</td>
<td>12:33pm May 30</td>
<td>2860</td>
</tr>
<tr>
<td></td>
<td>/bklist/entry/author/first</td>
<td>1</td>
<td>12:33pm May 30</td>
<td>2620</td>
</tr>
<tr>
<td></td>
<td>/bklist/entry/publisher</td>
<td>1</td>
<td>12:33pm May 30</td>
<td>1840</td>
</tr>
</tbody>
</table>

Figure 6.3: Path Table with Initial Statistics for Q1

When a new query overlaps with a cached query, the probe query PQ is formulated to retrieve the relevant data in the XML view via the path expressions specified in the return clauses. These path expressions correspond to the XPathRows in the path table. We can hence correspondingly update the statistics for these XPathRows that are involved in the probe query. For example, Figure 6.4 shows the path table constructed for Q2’s query region and the updated statistics in the cached query region Q1. As we explained before, the answer of Q2 has utilized the cached data from region Q1. The replacement manager hence correspondingly modifies the statistics of the highlighted XPathRows in Q1’s path table that are involved in the probe query for answering Q2. The fragments along two paths /bklist/entry/@year, /bklist/entry/title and /bklist/entry/author/first in Q1’s XML view contribute to answering Q2.
6.2. THE PARTIAL REPLACEMENT STRATEGY

6.2.3 Utility Value and Replacement Function

The utility value is considered to be the indicator for the replacement likelihood of cached objects. Based on the collected statistics, a caching system may adopt a particular replacement policy in favor of purging some cached objects with certain characteristics over other ones. A replacement function is used to reflect the replacement preference of a caching system. It calculates the utility values of cached objects, based on which the replacement manager chooses the victim to be purged to make room for new objects.

Cache replacement policies have been extensively studied in different scenarios, such as page-based [CFZ94] and tuple-based [DFMV90] caches. Various replacement schemes [RD90, NNW93, CFL94, PG95, LCN+99] have been investigated. Among them, the well-known replacement schemes are the Least Recently Used (LRU), the Least Frequently Used (LFU) schemes

---

<table>
<thead>
<tr>
<th>XPathRow</th>
<th>hits</th>
<th>last_access_time</th>
<th>obj_bytes</th>
</tr>
</thead>
<tbody>
<tr>
<td>/goodbook/title</td>
<td>1</td>
<td>12:47pm May 30</td>
<td>220</td>
</tr>
<tr>
<td>/goodbook/price</td>
<td>1</td>
<td>12:47pm May 30</td>
<td>150</td>
</tr>
<tr>
<td>/goodbook/author/first</td>
<td>1</td>
<td>12:47pm May 30</td>
<td>380</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>XPathRow</th>
<th>hits</th>
<th>last_access_time</th>
<th>obj_bytes</th>
</tr>
</thead>
<tbody>
<tr>
<td>/bklist/entry/@year</td>
<td>1</td>
<td>12:33pm May 30</td>
<td>1600</td>
</tr>
<tr>
<td>/bklist/entry/title</td>
<td>2</td>
<td>12:47pm May 30</td>
<td>2100</td>
</tr>
<tr>
<td>/bklist/entry/author/last</td>
<td>1</td>
<td>12:33pm May 30</td>
<td>2860</td>
</tr>
<tr>
<td>/bklist/entry/author/first</td>
<td>2</td>
<td>12:47pm May 30</td>
<td>2620</td>
</tr>
<tr>
<td>/bklist/entry/publisher</td>
<td>1</td>
<td>12:33pm May 30</td>
<td>1840</td>
</tr>
</tbody>
</table>

Figure 6.4: Q2’s Path Table and Q1’s Path Table with Updated Statistics
and their varieties. The LRU scheme is widely used due to its simplicity while still being effective when recently referenced objects are likely to be re-referenced in the near future. The LFU policy [RD90] uses reference frequency instead of recency as the parameter for the replacement function.

In this dissertation, we propose to utilize the detailed path tables\(^1\) to perform a finer granularity replacement than replacing a complete query region at a time. That is, the input to the replacement function are not the statistics recorded at the whole query level, but those at the level of the internal path structure of view documents. For example, \(\text{last\_access\_time}\) is a timestamp recorded when an XPathRow is used in the latest probe query. \(\text{hits}\) is the number of times an XPathRow has been used for answering subsequent queries. \(\text{obj\_bytes}\) is a size estimation of the data collected along a particular path. If a path were to be selected as the next victim, this number gives a hint about how large a fragment would be purged from the XML view. We also keep track of more global statistics at the query level such as \(\text{xml\_doc\_size}\), which is the overall document size in bytes, and \(\text{fetching\_delay\_cost}\), the original query evaluation time.

The considered statistics can be classified into several categories. One category concerns the user access pattern, such as the recency value \(\text{last\_access\_time}\) and frequency value \(\text{hits}\). The \(\text{hits}\) measure on an XPathRow is increased by one each time when it is requested by a probe query. Its \(\text{last\_access\_time}\) is updated to the current time upon such an update. The second category is related to the data size that would be freed upon a

\(^1\text{Concerning the enlarged descriptor size caused by this path table structure, we consider to minimize the space overhead by adopting some indexing or compression techniques.}\)
6.2. THE PARTIAL REPLACEMENT STRATEGY

purge, i.e., the \textit{obj} \textit{bytes} on a particular XPathRow and the \textit{xml} \textit{doc} \textit{size}. If two groups of XPathRows have a tie in their frequency values, the one associated with a larger \textit{obj} \textit{bytes} is replaced. This is because our fine-grained partial replacement strategy may need to perform path-related-region subtraction several times to free enough space for a new region. Hence our replacement function is in favor of purging a larger piece at a time for efficiency.

The distance involved in the data transmission and network delay fall into the third category. We consider the benefits brought by preserving a region by measuring the loss caused by not caching it, which can be represented by the initial response delay for answering a query, denoted as \textit{fetching delay cost}. By retaining regions with longer initial fetching delay, large fetching cost for such regions in the future could be avoided. Furthermore, we roughly measure the \textit{byte fetching delay cost} for a particular query by dividing \textit{fetching delay cost} by \textit{xml doc size} (in bytes). Therefore, the benefits of retaining a particular XPathRow in the query region can be measured by \textit{byte fetching delay cost} \times \textit{obj bytes}. Below, we propose a replacement function to calculate the comprehensive utility value of each XPathRow based on the collected statistics.

\[
r_{p \text{ fun}} = \frac{\text{hits} \times \text{fetching delay cost} \times \text{obj bytes}}{\text{xml doc size}} \quad (6.1)
\]

Other functions in favor of different scenarios can be easily plugged into our system. We have indeed experimentally studied several such functions when designing this formula.
6.2.4 The Partial Replacement Algorithm

Figure 6.5: The Replacement Control Flow

Figure 6.5 shows the control flow of the replacement manager in ACE-XQ. After the statistics information has been updated for those XPathRows involved in a probe query, the pre-defined replacement function re-calculates their utility values. Only when there is a need for replacement due to exhausting of the cache capacity, the replacement manager chooses those XPathRows with the lowest utility value as the victim XPathRows. It then composes a filter query to remove the fragments corresponding to these paths from the relevant XML view(s). The query descriptors of those affected cached queries are also modified accordingly to be consistent with their changed XML views.

Filter Query. Suppose ten queries $Q_1$ to $Q_{10}$ are in the cache after the cache has been in use for a while. Different utility values are recorded
in these queries’ path tables. Now a query Q11 arrives and there is not enough space in the cache for it. Based on the lowest utility value, the replacement manager decides that \textit{victimQ} and \textit{victimRows[ ]} are Q1 and \\
\texttt{[/bklist/entry/@year, /bklist/entry/author/last, /bklist/entry/publisher]} \textit{respectively (the statistics of Q1 and Q2 may not be the same as illustrated in Figure 6.4 any more).}

To remove XML fragments corresponding to \textit{victimRows[ ]} from \textit{Q1Res.xml}, XML update statements can be adopted [TIHW01]. An alternative method is to extract the parts that are of interest and discard the remainder from the view content with the help of some filtering mechanism such as the filter query shown in Figure 6.6.\footnote{Note that the filter query returns the complement path set of \textit{victimRows[ ]}. The construction of such a filter query utilizes the view DTD knowledge for preserving the structural hierarchy in the updated view content.}

\begin{verbatim}
<bklist>
  {for $b in doc("Q1Res.xml")/bklist/entry
   return <entry>
     <$b/title> <$b/author/first> 
   <//entry>
  }
<bklist>
\end{verbatim}

Figure 6.6: An Example Filter Query

The detailed replacement algorithm is described in Figure 6.7.

**Comparison With a Related Work.** In XCacher [HP02], all cached queries are integrated to be represented by one \textit{modified incomplete tree (MIT)}. Corresponding to the different value domains specified in consecutive queries, a set of disjunctive conditional tree types are specified in MIT. The concept of
ALGORITHM Partial_Replacement

INPUT: cached queries \( CQ_s \), new query \( Q \), and replacement function \( rp_{\text{fun}} \).

OUTPUT: \( \text{victim}Q \), its filter query \( f_q \) and new query descriptor.

If a cached query \( CQ \) is used for answering \( Q \) Then

update statistics of overlapped XPathRows[] in \( CQ \)'s region.

End If

Calculate the needed space \( S \) for caching \( Q \);

While \( (S \geq 0) \) Do // not enough space for holding \( Q \)

Foreach \( CQ_i \in CQ_s \) Do

Foreach XPathRow in \( CQ_i \) Do

Invoke \( rp_{\text{fun}} \) to calculate its utility value;

End Foreach

End Foreach

victimRows[] \( \leftarrow \) victimRows with lowest utility value;

\( \text{victim}Q \leftarrow \) cached query containing victimRows[];

Compose a filter query \( f_q \) to purge fragments related to victimRows[];

Formulate a new query descriptor for \( \text{victim}Q \);

Calculate the freed space \( \text{freed}S \) by fragment purging;

\( S = S - \text{freed}S \);

End While

Figure 6.7: The Partial Replacement Algorithm

specialized tree type in XCacher is similar to that of query region in ACE-XQ. Hence their replacement achieved by utilizing the update statement for purging data off the cached data tree is analogous to our partial replacement strategy in some sense. However, when a new query enters XCacher, only the incremental value domain differences caused by it with respect to
MIT are specialized into new disjunctive types to be added into MIT. This means that they do not identify the overlapped region. Hence there is no natural way to record the relative high usage of the overlapped fragment in XCacher. In this sense, their replacement strategy is not as fine-grained as our replacement, which may incorporate a variety of replacement policies.

In addition, the specialization mechanism for refining the MIT upon new queries may cause an exponential blow-up of specialized types. Even if the structural complexity of MIT may be bound to be \( \text{PTIME} \) with respect to the number of incoming queries, it is still not comparable to our approach. Our cache management module controls the region splitting/merging and causes only a linear increase of query regions with respect to the number of incoming queries.

6.3 The Analysis of Cache Performance

We have discussed earlier the methodologies adopted by our partial replacement approach versus the alternative approaches. Here, we attempt to give an analytical model for better understanding how the caching system interacts with various factors such as cache size and query access pattern. Based on this model, we analyze how the cache would behave in the face of different replacement strategies. This may help us to gain insights into the reason why the cache equipped with our partial replacement strategy can achieve a higher cache hit ratio than the alternative one.
6.3. THE ANALYSIS OF CACHE PERFORMANCE

6.3.1 Query Trace Model

We describe the semantic nature of a query trace in terms of query selectivity, locality and skewness, etc. Suppose there is a sequence of query access requests for one of \(N \gg 1\) documents \(D_1, \ldots, D_N\). Visually, we may imagine the whole semantic space is set by the source document DTDs. A query can then be seen as an object with certain spatial expansion like a rectangle region. Query selectivity, denoted as \(S\), is the size ratio of the query result over the source documents. Assume the semantic space is normalized, the selectivity of \(Q_i\), namely \(S(Q_i)\), can then be used for representing its query region size. \(S(Q_i)\) is closely related to the query condition strictness and the projection scope, which can be viewed as the length and the width of \(Q_i\). The query locality is measured by the distance \(D\) between a given query pair. Suppose \(Q_i, Q_j\) are two queries with \(Q_i\) preceding \(Q_j\) in a given query trace. Below we show three scenarios where the computing of \(D(Q_i, Q_j)\) (i.e., the distance between \(Q_i\) and \(Q_j\)) takes different approaches.

\[
D(Q_i, Q_j) = \begin{cases} 
\infty & \text{when } Q_i \text{ and } Q_j \text{ are semantically disjoint;} \\
\frac{S(Q_j)}{S(Q_j \cap Q_i)} & \text{when } Q_j \text{ overlaps with } Q_i; \\
1 & \text{when } Q_j \text{ is totally contained within } Q_i.
\end{cases}
\]

The query skewness \(K\) is an overall characteristic of a query trace indicating how the query regions are distributed in the semantic space. Corresponding to our two-tier query descriptor scheme, we group queries into a two-tier cluster hierarchy. Each top-tier cluster (called TTC) corresponds
6.3. THE ANALYSIS OF CACHE PERFORMANCE

6.3.2 Cache Hit Probability

Numerous studies in the web caching field have concluded that web access follows a Zipf-like distribution [L. 99, L. 98]. That is, the relative probability of a request for a document is inversely proportional to its popularity rank $i$ ($i = 1..N$). The probability $P_d(i)$ of a request for the $i$'th popular document is proportional to $1/i^\alpha$ ($0 < \alpha \leq 1$). In our context, we think it is appropriate to model the XML document access pattern using this Zipf-like distribution. Since $\alpha$ in the distribution model implies document access skewness, it is closely related to $Kt$.

Independent of the document access pattern, we also observe that the average distance of query pairs within a given document likely indicates the skewness of query region access distribution. That is, the smaller the average distance, the more intensive is the concentration of query accesses to hot regions. Hence, the indicator of query region access skewness $Kl$ is $Kl = \frac{\sum_{i=1}^{m}(1/P_d(Q_i)Q_j))}{m(m-1)/2}$. In this formula, the numerator is the sum of the in-

---

$^3$When $\alpha$ is close to 1, the top 1 popular document gets twice as many query accesses than the next most popular one. If $\alpha$ is close to 0, likely every document is evenly accessed.
verse of the distances (of all query pairs within a certain document cluster) adjusted by a parameter \( \tau \). The denominator is the number of all possible combinations of query pairs, assuming \( m \) is the number of queries accessing the given document. Suppose the region access distribution follows a similar Zipf-like model, we then compute the probability \( Pr(j) \) of a request for the \( k' \)th popular region is proportional to \( 1/j^\beta \) (0 < \( \beta \) ≤ 1 and \( \beta \) closely related to \( KJ \)).

For Zipf-like distributions, the cumulative probability that one of the top \( k \) documents (among the total \( N \) documents) is accessed is given asymptotically by:

\[
\phi(k) = \sum_{i=1}^{k} \frac{\Omega}{i^\alpha}, \text{ where } \Omega = (\sum_{i=1}^{N} 1/i^\alpha)^{-1} \approx (1 - \alpha)/N^{1-\alpha}.
\]

Thus \( \phi(k) \approx (k/N)^{1-\alpha} \) (when \( \alpha = 1, \phi(k) \approx \ln(k/N) \)). Because \( k/N < 1 \), a larger \( \alpha \) increases \( \phi(k) \), meaning more queries focus on a few hot documents. The probability \( Pd(i) \) of an access to the top \( i' \)th popular document is \( Pd(i) = \frac{\Omega}{i^\alpha} \approx \frac{1-\alpha}{N}(\frac{N}{i})^\alpha \). Similarly, if considering the probability \( Pr(j) \) of a query request for the top \( j' \)th popular region within a particular document, we have \( Pr(j) \approx \frac{1-\beta}{M}(\frac{M}{j})^\beta \), where \( M \) is the number of query regions in a document, and \( \beta \) is the parameter suits the region access distribution in a particular document.

In a query-based caching environment, we are concerned about the popularity ranking of query regions across documents. First, we look at the overall probability \( P(i, j) \) of a query request for the \( j' \)th popular region within the \( i' \)th popular document. Suppose \( Pd(i) \) and \( Pr(j) \) are independent of each other, we obtain \( P(i, j) = Pd(i) \times Pr(j) \approx \frac{(1-\alpha)(1-\beta)}{M \times N} (\frac{N}{i})^\alpha (\frac{M}{j})^\beta \).
6.3. THE ANALYSIS OF CACHE PERFORMANCE

If the situation is simpler and a uniform $\alpha$ suits both the document and all the query region access distributions, $P(i, j) \approx \frac{(1-\alpha)^2}{M \times N^2} (\frac{M \times N}{i \times j})^\alpha$, which implies a multivariate Zipf-like distribution. We infer from this equation that if two query regions have the same $i \times j$ production value (i.e., the document popularity rank times the local query region popularity rank), then they have the same overall popularity. For such a multivariate distribution, we have the cumulative probability

$$\phi(k) = \sum_{i=1}^{t} \sum_{j=1}^{u} P(i, j),$$

where $t \times u \leq k$.

This model assumes that the query requests are independent and both the document and query region access patterns follow the Zipf-like distribution with the same parameter. It may be not very realistic, but the model is tractable and it is sufficient to help us understand how the hit ratio can be influenced by various factors.

**Correlation between Hit Ratio and Cache Size.** Studies of web caching have found that when the cache size is infinitely large, the correlation between the access frequency and document size, if any, is weak in general and can be ignored [L. 99]. We believe this finding is valid in our context as well. However, if the cache source is limited, the Zipf-like distribution will be “cut-off” and eventually the top $c$ most popular query region groups\(^4\) will fill the cache to its size limit ideally. However, it is hard to derive $c$ from the cache size $C$ due to the factoring problem. If we assume that the query region sizes are the same and the factoring can be continuous (not a realistic assumption though), $c$ is approximated as $\sqrt{2C}$ due to $\sum_{i=1}^{c} \sum_{j=1}^{c/i} = C$.

\(^4\)If multiple query regions have the same overall popularity, e.g., $P(1, 6) = P(2, 3) = P(3, 2) = P(6, 1)$, we consider them as one query region group.
Thus the cumulative probability

$$\phi(C) \approx \sum_{i=1}^{\sqrt{2C}} \sum_{j=1}^{\sqrt{2C}/i} P(i, j) \approx \frac{(1-\alpha)^2}{(MN)^{1-\alpha}} \sum_{i=1}^{\sqrt{2C}} \sum_{j=1}^{\sqrt{2C}/i} (\frac{1}{i})^\alpha.$$

The asymptotic hit ratio $H(C)$ is closely related to $\phi(C)$. If $\alpha$ is very close to 1, $H(C)$ grows with the cache size $C$ logarithmically, i.e., $H(C) \approx ln \frac{C}{MN}$. Otherwise, $H(C)$ cannot easily be approximated by a particular function. However, it is bounded by some polynomial function with a small power, e.g, $H(C) = (\frac{C}{MN})^{1-\alpha}$.

**Correlation between Hit Ratio and Query Pattern.** From the hit ratio function, we can see that the parameter $\alpha$ plays a role in controlling the slope steepness of the curve. Since $\frac{C}{MN} < 1$, the closer $\alpha$ is to 1, the smaller $1 - \alpha$ is and consequently the larger $H(C)$ gets. As we discussed before, $\alpha$ is related to the document access skewness $\mathcal{K}t$ and the query region access skewness $\mathcal{K}l$. Therefore, the more query requests are concentrated on a few hot spots, the higher hit ratios can be achieved, ideally. We also observe that, if the overall document size is fixed, $N$ and $M$ will increase when the average individual document size and query region sizes decrease. With the same cache size and query selectivity $S$ as we have discussed earlier, a larger $S$ implies a larger region and thus a smaller $H(C)$. 
6.3.3 Hit Ratio and Different Replacement Strategies

We have analyzed the cache hit ratio with varied cache sizes, document sizes, query selectivities and skewnesses. The assumption is that the cache replacement strategy is nearly ideal and it replaces query regions by strictly following the popularity order. It is hence unlikely for a real cache to achieve this high expectation. Also, the concept of query regions used in the model is oversimplified. For example, all queries would have to have the same selectivity to result in the uniform region size. Furthermore, a query region is the smallest unit, i.e., one either exact matches or is disjoint with another. There is no notion of partial overlap between query regions. In this sense, the model cannot be directly applied to our query-based caching context. However, we could make some adjustment and still use this model to analyze how the cache would behave differently in the face of different replacement strategies, in particular for recurring partial overlapping query cases.

Suppose a new query arrives at the cache and it partially overlaps with a cached one. We now consider a query region consists of only one return path expression. That is, we associate the concept of query region with our concept of XPathRow. For an incoming query $Q_n$, it is then viewed as being broken down into a series of $n$ smaller query regions. Among them, $p$ regions that belong to the overlapping part with a cached query $Q_c$ can be seen as cache hits while the rest as misses.

Imagine $Q_n$ enters a cache where the region-preserving strategy is deployed. As a result, all $n$ query regions that $Q_n$ is composed of together
with those \( m \) composing \( Q_n \) are marked with one more hit, even though only \( p \) regions really hit the cache. This is because this region-preserving does not split queries to indicate different uses of different query regions. This means a unfaithful recording of the utility values (i.e., the popularity ranks in the model) and would likely impair the cache performance due to its failing to separate hot-pots from cold-pots.

If the region-splitting strategy is applied, \( Q_n \) would be physically divided into \( PQ \) (containing \( p \) regions) and \( Q-PQ \) (containing \( n - p \) regions). Having overcome the aforementioned shortcomings of the region-preserving strategy, the region-splitting however raises a new issue that too many small queries are likely produced from splitting. Over time, queries in the cache may have been split to be as small as just a query region unit. In this scenario, the size difference between a new query (which has not experienced splitting) and a cached query is expected to be big. This is not desirable when query containment and rewriting is considered. Due to the problem of query fragmentation, we need to find possibly up to \( p \) cached queries for matching them with \( Q_n 's \) \( p \) regions. This exhaustive search within the cache space is usually costly, let alone the cost for combining results from \( p \) different views. Therefore, we prefer the cached queries not being overly split as this may possibly induce extraneous efforts for containment mapping and rewriting\(^5\).

In our partial replacement strategy, we set up for a query its descriptor in two tiers, namely corresponding to the query itself and its component

\(^5\)Even if we can adopt the “first-found” policy to randomly picks one candidate cached query for query rewriting, the cache utilization would be very inefficient since the small cache query size further restricts the possibly overlapping region size.
query regions. Our replacement approach overcomes both disadvantages of the region-preserving and the region-splitting strategy due to its strategy of separating the notion of physical splitting from that of statistics value recording. This way, the utility values can be computed at the finer query region granularity without necessarily splitting the query physically. The coarser-level query splitting only occurs when replacement is inevitable. When it happens, the statistics recorded at the finer-level regions help to replace fairly the less popular regions and preserve the more popular ones. A by-product of this replacement process is an “optimized” query in the sense that the “fluff” (i.e., less useful portions of query regions) is removed from the query to result in an overall more popular query. In summary, the cache utility is improved due to the fine-grained optimization of each individual query in the cache.
Part IV

System Implementation and Evaluation
Chapter 7

ACE-XQ: An Cache-aware XQuery Processing System

7.1 The Objectives of the ACE-XQ System

In Part II and Part III, the solutions for the problem of query containment and rewriting in the XQuery context and a partial replacement strategy tailored for improving the XQuery-based cache performance are provided respectively. These proposed techniques lay the foundation for the XQuery-based semantic caching framework as depicted in Figure 1.5. In this chapter, we describe an XQuery-based semantic caching system called ACE-XQ\(^1\), which realizes the proposed ideas and shows that it can help to reduce the perceivable response time for requests formulated in the contemporary XML query language – XQuery over a distributed environment.

\(^1\)It was previously called XCache, but has been renamed due to the conflicting name registration issue with a company’s product.
7.1. THE OBJECTIVES OF THE ACE-XQ SYSTEM

7.1.1 The Purposes of System Developing

The main objective of the ACE-XQ system is to realize the proposed techniques including the query pre-processing, the containment checking, the query rewriting, and the partial replacement strategy targeting XQuery. The purposes of developing this ACE-XQ System are summarized below.

- To implement the first XQuery-based semantic caching system based on the proposed techniques.
- To utilize some existing XQuery engine to provide the query parsing and evaluation functionalities needed for developing and testing the semantic caching system.
- To validate the correctness of the proposed ideas with the developed system.
- To experiment with the system for evaluating the query performance with the aid of caching against a baseline system without caching.
- To provide a basic XQuery-based semantic caching system where more cache-related techniques such as the cached view maintenance and region coalescing can be further explored and incorporated.

7.1.2 The Two Phases of System Implementation

Our ACE-XQ system is designed to extend the state-of-the-art XML Query engine with a semantic caching system which can help to improve the query performance. Due to historical reasons, we have implemented two
7.1. THE OBJECTIVES OF THE ACE-XQ SYSTEM

versions of ACE-XQ systems, one of which is built on top of the Kweelt query engine [A. 00] (the source code is available at http://db.cis.upenn.edu/Kweelt) and the other on IPSI-XQ [Fra]. When we first started out with the semantic caching idea, Quilt [CRF00] was the state-of-the-art XML query language conforming to W3C’s requirements and it is actually the inspiration and precursor of XQuery, which was proposed by W3C later as the standard XML query language. Quilt and XQuery bear a great resemblance in terms of their syntactical skeletons and semantics implications.

During our first phase implementation, Kweelt query engine was the only available query engine of the kind. The fact that it is an open source project makes it possible for us to quickly learn the detailed internal implementation of the query engine and in turn to design our own caching solution based on it. The prototype semantic caching system implemented in first phase validated our preliminary caching ideas and it helped us to further refine the semantic caching ideas and the system design. Due to the emergence of XQuery and a variety of implementations of this standard XML query language proposed by W3C, we re-implement our ACE-XQ system to support XQuery during the second phase.

In the second phase implementation, we build the ACE-XQ system on top of the GMD IPSI-XQ engine, which more closely follows the formal semantics of XQuery and realizes a fully conformant, static type-checking incorporated XQuery implementation. In this newer version of ACE-XQ system, the query interface accepts queries written in the precise XQuery syntax. Also, some refined ideas in terms of both caching and system design are implemented. For example, we implement the query decomposer
module based on the visitor design pattern which performs one pass of the XQueryTree structure yield from the parsing phase of the IPSI-XQ engine and produces the VarTree and TagTree structures which together serve as the internal query region descriptor. Another example is that more containment cases are handled by the query containment mapper module by exploiting the type-inference functionality implemented within IPSI-XQ.

The XQuery containment checking and rewriting techniques described in this dissertation are what the second-phase implementation of ACE-XQ system is based on. In the next section, we will give an overview of the system and describe the functionalities of the main modules involved.

7.2 The ACE-XQ System Overview

7.2.1 The ACE-XQ Architecture

The ACE-XQ system architecture is depicted in Figure 7.1. It consists of two subsystems, a Query Matcher which implements the query containment and rewriting techniques and a Cache Manager which manages the cache space and deploys the replacement strategies.

When a new user query comes in, it is first parsed by the Query Parser provided by the native query engine, which in our case is IPSI-XQ. Then the internal XQueryTree structure is further pre-processed by our Query Decomposer module (shown in the Query Matcher subsystem on the left hand side of Figure 7.1) which performs the following two steps. First, the query minimization and normalization techniques are applied to derive a uniformly nested query format containing only essential variables. Second, the
VarTree and TagTree structures are constructed based on the extracted variable dependency hierarchies and the result construction template. These two tree structures are then registered via the Query Pattern Register as the internal query descriptor to be associated with the corresponding semantic region whose space is assigned by the Cache Manager subsystem.

For a given input query, the Query Containment Mapper is called to check whether it is contained within some cached query, or whether it overlaps with a cached query to some degree, if there is not a totally contained case. The query containment decision is made according to the proposed VarTree-based one-to-one variable containment mapping technique as in-
troduced in Chapter 3. Based on the established variable mapping pairs, the \textit{Query Rewriter} rewrites the new query with respect to the view structure of the containing cached query. Thus the user’s new XQuery is divided into a \textit{probe query} to retrieve answers from the cached local views, and a \textit{remainder query} to obtain the remaining answers from remote sources, and a \textit{combining query} to make one complete answer.

As shown on the right hand side of Figure 7.1, the \textit{Cache Manager} subsystem of ACE-XQ manages a collection of query regions, each composed of a semantic \textit{query descriptor} of a cached query and the result XML view. The former part is used for reasoning about query containment while the latter can be queried by the probe query to provide any answer available in the cache to the new query as quickly as possible.

Due to the limited resource of cache space, the proposed partial replacement strategy is implemented in the \textit{Replacement Manager} with the \textit{Cache Manager} subsystem of ACE-XQ. When a new query arrives and not enough cache space is left, victim (partial) queries are chosen and their associated view contents are evicted from the cache to make room for the new query. The user access statistics are recorded at the fine-grained path level of the view structure, based on which the utility values are calculated and the replacement decision is made accordingly. Besides the proposed partial replacement strategy, the complete region replacement (so-called total replacement) strategy is also implemented in our ACE-XQ system.

Next, we describe the web interface and the internal modules implemented in the ACE-XQ system in more details.
7.2. THE ACE-XQ SYSTEM OVERVIEW

7.2.2 The ACE-XQ System Interface

The interface for the ACE-XQ system has been designed to provide the users with the full functionalities of the system via an easy to use graphical interface. Intended to serve a web-based cache-aware query system, the technologies of JSP and Java Servlet are used in the implementation of ACE-XQ. That is, the ACE-XQ system is accessed via a JSP-based web page, where the users can formulate their requests in the syntax of XQuery. By clicking on the “submit” button on the interface web page, the query request is sent to a local web server enable with the Tomcat Servlet engine. The whole ACE-XQ system is implemented in Java 1.4 and hence can be deployed as Java Servlet programs on that web server.

On the interface web page, users are also provided with the option of loading the sample queries and changing the to-be-queried XML document amongst a set of default ones. In addition, the user can toggle on or off the “cache-aware” mode to indicate her preference of executing the specified query with the aid of the ACE-XQ semantic caching system or without. Furthermore, the user can specify the amount of disk space allocated for the cache, and which replacement strategy to be deployed, etc.

Besides serving as the interface for accepting the user’s input queries and adjustment of various parameters, this interface page also provides text display windows or it can spawn other pages that contain certain information such as the current system status, the rewritten probe and rewritten queries, the links to the cached view documents, and the utilization of the cache space, etc.
The functionalities made available through the web interface of the ACE-XQ system are sketched below.

1). It allows users to visually check the correctness of the rewritten queries by:
   a. viewing the generated probe and remainder queries;
   b. accessing and comparing the XML document generated from the semantic cache system by rewriting the input queries with respect to the result view structure of the containing query and the one produced from the baseline query execution engine with no awareness of the semantic cache;

2). It reveals the system processing status by:
   a. indicating the up-to-date status while the input query passes the phases of query decomposition, containment mapping, query rewriting, and query execution;
   b. displaying the decision on the query containment relationship;
   c. showing the cache utilization and the number of query regions over time;

3). It helps with the query performance analysis by:
   a. reporting the relevant statistics on the size of the source XML documents, on the query workload characteristics such as query selectivity, hit ratios, etc;
b. displaying the query response time spent by utilizing the cache system versus bypassing it.

7.2.3 The Main Internal Modules

Now we look more closely at some main modules implemented or incorporated in the ACE-XQ system. We will describe how they interact with each other and which internal data structures are exploited.

**XQuery Engine.** The ACE-XQ system is integrated with the IPSI-XQ query engine version 1.0.1, which is implemented faithfully along the W3C Working Drafts on XQuery 1.0 as of the date 2002-01-08 (i.e., the working drafts on the language syntax, data model, functions and operators, and formal semantics of XQuery 1.0).

In the ACE-XQ system architecture, three components of the IPSI-XQ engine, namely, the query parser, the static type inference, and the query executor, are utilized. Below we explain how each of these three components are interfaced with the other modules in the ACE-XQ system.

- Upon the submission of a query request, the input query is first parsed by the query parser provided by the IPSI-XQ engine. The parsed query tree structure is then intercepted by our query decomposer to perform the proposed query minimization, normalization and decomposition techniques.

- The type inference and subtyping functionalities are provided by the static type inference component of the IPSI-XQ engine. Such func-
tionalities are exploited by the query containment mapper module to realize the proposed containment checking technique.

- If a query is determined to be contained or overlapping with a cached query, then the rewritten probe query is executed by the query executor of the IPSI-XQ engine installed at the local cache site for retrieving the answer available in the cache, while the remainder query sent to the remote data server where an IPSI-XQ engine is installed for query execution. In the case when no cached query can be used for answering the new query, the whole query is transmitted to the remote data server to be executed. The ACE-XQ system can treat the query executor as a black box which processes the query and returns its results in the form of an XML document to the ACE-XQ system.

**Query Decomposer.** Our query decomposer is interfaced with the query parser module of the IPSI-XQ engine from which it intercepts the parsed query tree structure and performs three passes on it.

The purpose of the first pass is to apply the variable minimization procedure as described in Figure 3.8. After this pass, essential variables are distinguished from those non-essential ones and the former are collected into a set named EVS. Based on the identified variable essentiality, the appropriate normalization rules are then applied which result in a uniformly nested query form containing only the essential variables.

In the second pass, the variable dependencies are extracted from the parsed query tree and the VarTree structure is constructed to reflect the pattern matching semantics of the input query.
7.2. THE ACE-XQ SYSTEM OVERVIEW

The data structure implemented for VarTree is a recursive tree structure composed of \textit{VarNodes}, each of which is a data structure accommodating the information about an essential variable, such as the variable name, the bound path expression, and a list of children VarNodes. A \textit{Region} data structure is also implemented and it has a field containing a list of VarNode references. This is used for indicating the associations between variables and their corresponding home FLWR blocks. The control-flow-induced variable dependencies can be inferred from such associations. The Region data structure also contains a field for a list of where-conditions which are specified in the current FLWR block but with the conditions imposed on the variables defined in the enclosing FLWR blocks.

The third pass focused on the hierarchy of the return constructs in the parsed query tree and the TagTree structure is built to reflect the result construction semantics of the input query.

Similar to VarTree, TagTree is also implemented as a recursive tree structure which is however composed of \textit{RetNodes}, the data structure for result constructors each containing the tag name, a list of locally defined variables, and a list of children RetNodes. The query rewriter module of the ACE-XQ system would need to utilize this TagTree structure of the containing cached query for rewriting a new query.

Finally, the constructed VarTree and TagTree structures are registered as the descriptor of the corresponding query region assigned in the cache system. These two tree structures help to explicitly reveal the query semantics and they provide the basis on which the proposed containment checking and query rewriting techniques are founded. Hence, this query
decomposer is necessary since otherwise the query containment is hard to be reasoned about based on the input query strings.

**Query Containment Mapper.** After the query decomposer module finishes the query pre-processing step, the query containment mapper conducts the containment checking between the new query and each of the cached query based on their corresponding VarTree structures. As explained in more details in Chapter 3, our proposed containment checking technique is primarily a tree embedding process that “maps” each VarTree node of the new query to a correspondence node of a cached query.

We implement such a tree embedding algorithm based on the data structure designed for VarTree. At the macroscopic level, this tree-embedding algorithm guides the top-down mapping between the VarNodes of two VarTrees; At the microscopic level, a mapping is established between a pair of VarNodes only if there exists a containment relationship between their respective bound path expressions. In the implementation, the type inference function provided by the IPSI-XQ engine is called along the traversal of each VarTree for inferring the type of each variable represented by a VarNode based on the type inferred for the parent variable and the relative XPath expression.

In addition, the containment checking is also conducted between the conditions contained in the respective Region nodes that the candidate variable pairs are associated with. All the other necessary checking steps indicated in the extended MAC mapping algorithm as shown in Figure 3.16 are also conducted.
Given a new query $Q_2$ and a cached query $Q_1$, our containment mapping procedure finally produces the maximum mapping $h$ from variables of $Q_2$ to those of $Q_1$. Depending on different mapping results obtained, we can classify the containment cases into the categories as follows:

- **Totally Contained**, if the mapping $h$ is a total injective mapping, namely, each variable in $Q_2$ has a mapping variable in $Q_1$. This means that the answers to $Q_2$ is a subset of the answers to $Q_1$. By rewriting $Q_2$ with respect to the view structure of $Q_1$, the answer to $Q_2$ can be obtained directly from the local cache.

- **Disjoint**, if there is no such a mapping $h$ that maps any variable in $Q_2$ to a variable in $Q_1$. For example, when the two queries involve different XML documents. This indicates that $Q_1$ and $Q_2$ share no answers in common. In this case, the cached answers are not useful for answering $Q_2$, which is hence sent to the remote data server to be executed.

- **Overlapping**, if the mapping $h$ maps some of the variables in $Q_2$ to the variables in $Q_1$. In this case, only partial answers to $Q_2$ can be retrieved from the cache and the rest has to be obtained by sending a remainder query to the remote data server.

**Query Rewriter.** If $Q_1$ is decided to be totally contained within $Q_2$ or overlapping with it, the query rewriter is then called for rewriting $Q_1$ based on the view structure of $Q_1$. Unlike the relational queries which produce flat output schema, an XQuery results in an output tree. Therefore, we use
TagTree for representing the view tree structure of a query. To rewrite $Q_2$ with respect to $Q_1$, the query rewriter rewrites the variable bound expressions and return expressions by utilizing the TagTree structure registered for $Q_1$ and the containment mapping pairs produced by the query containment mapper.

In one nutshell, our contribution lies in a complete semantic caching framework called ACE-XQ that we provide for targeting at XQuery. The core of our ACE-XQ system consists of three tightly related procedures: XQuery decomposition including the pre-processing steps, containment mapping and query rewriting, which are developed based on the most current theoretical results from the literature but own their unique features different from the alternative techniques.
Chapter 8

Experimental Studies

8.1 Experiment Setup

8.1.1 Objectives of Experimental Study

Our experimental studies serve three main purposes. The first purpose is to validate our containment checking and rewriting ideas for XQuery. For this, we pick example queries from the W3C working draft “XML Query Use Case” [W3Cc] and run them through the XQuery engine with and without the caching mechanism provided by the ACE-XQ system. We check the correctness of the rewritten queries by comparing their answers with those produced by directly executing the corresponding original queries. Furthermore, we examine the features of XQuery that can be handled by the query containment and rewriting techniques utilized in the ACE-XQ system.

The second purpose is to determine whether or not the ACE-XQ system
can help to improve query performance. If yes, how much is the performance gain achieved, which are the cache preferable cases and which are not, what is the computational overhead of the ACE-XQ system when the space limitation is ignored temporarily.

The third purpose of conducting the experimental studies is to examine the effectiveness of the alternative replacement strategies with different query workloads streaming into the ACE-XQ system and different sizes of the cache space is assumed. In particular, we will compare the performance of our partial replacement and the total replacement strategies in a variety of scenarios.

An XQuery engine equipped with the ACE-XQ system can serve as a testbed for further conducting more in-depth experimental studies in the future or as the base semantic caching system extensible with other auxiliary modules.

8.1.2 System Deployment

The assumption of this work web information retrieval in a distributed environment. The information available on the web is represented in the XML data format and an appropriate retrieval means is the standard XML query language – XQuery.

We install our ACE-XQ system together with the IPSI-XQ query engine on a local Linux machine with 800MHz and 514M memory. On a remote web server we upload a set of XML documents with different sizes and also install an IPSI-XQ query engine which is responsible for the query execution at the remote data server site.
Without the aid of the cache, a distributed querying process can be broken down into three distinct stages. We now analyze the cost distribution in these three phases and aim to identify the factors that most strongly influence the user perceivable response time.

1). The first stage is sending the query from the client to the server. There is a query shipping cost associated with this stage, although it should not fluctuate between queries as the size of queries themselves are very small compared to that of an XML document.

2). The second stage is the execution of the query on the remote server against the originally specified XML document. This time component should greatly depend on both the query and the size of the XML document being queried against.

3). The last stage of the process is sending the query results back to the client. The result data shipping time associated with this stage is dependent on the size of the results and therefore also dependent on the query and the size of the source XML document.

Roughly, we can use the formula below to indicate how the user perceivable respond time is summed up.

\[ T_{total} = T_{query\ shipping} + T_{query\ execution} + T_{result\ shipping} \]

We can see from the above that the cost spent in the first and third stages involves the query shipping and the result data shipping across the network respectively. Hence, the network delay is a major factor besides the
local query processing time. The former is related to the network distance, the available bandwidth, the quality of network transmission, as well as the result data size. The latter is dependent on the efficiency of the query engine being used, the source document size and the query size.

The time saving can be achieved in possibly all three stages by deploying the ACE-XQ system at the client side to assist the local query engine. That is, if the new query is totally contained within a cached query, then neither $T_{\text{query, shipping}}$ nor $T_{\text{result, shipping}}$ is necessary any more, even $T_{\text{query, execution}}$ will be scaled down since the XML document being queried against is not the original XML document but only the view document of the containing query, which can be reasonably assumed to be a small percentage of the original one.

In the partially contained (i.e. overlapping) case, it is a little difficult to predict the time saving since the remainder query still needs to be transmitted to the remote data server to be executed and the results need to be transmitted back across the network. Although there might be some time saving since the remainder query result being shipped across the network is likely in a smaller size than that for the original query, it may be counteracted by the overhead cost on splitting the query into a probe query and a remainder query, and by the cost on combining their respective results into a complete view document.

In the disjoint case, the whole original query is sent to the remote data server to be executed. Thus there is no time saving in this case. On the contrary, some additional computational overhead is attached due to the query preprocessing and containment checking steps involved.
Therefore, the overall query performance in practice depends on how these three cases are mixed. The cache replacement manager hence comes into play and its role is to enhance the hit ratio, i.e., to maximize the cases when new incoming queries can be answered by utilizing the cached query answers, especially the totally contained cases. The criteria for a good replacement strategy is that it can better adapt to the user query access pattern.

8.1.3 Experiment Design

Existing Benchmarks for XML Engines. Currently, there are primarily five benchmarks [T.01, S.01, A.01b, B.B02, K.03] proposed for evaluating the performance of XML data management systems. XMach-1 [T.01] and XMark [A.01b] generate XML data that models data from particular Internet applications. In XMach-1 [T.01], the data exploited by the web applications are the directory documents and text documents consisting of chapters, sections and paragraphs. In XMark [A.01b], the XML data is oriented to the Internet-based auction applications. XOO7 [S.01] is an XML version of the OO7 Benchmark [M.93a] for OODBMSs, XBench and MBench are the two newly emerged XML benchmarks used for identifying characteristics of data and queries that may effect the performance of XML query processing engines.

While each of these benchmarks has its own targeted domain and XML application, they are all designed primarily for testing different performance aspects of XML databases or XML query engines. As explained in Section 8.1.1, our experimental studies have the goals of studying cases that
can be handled by our containment checking and rewriting techniques and those cannot, validating the correctness of rewritten queries, evaluating the performance gains achieved by caching, and the effectiveness of replacement strategies. That is, the data and queries needed in our experiments are mainly for testing the functionalities and performance of the caching-related modules in the ACE-XQ system, not the XQuery engine itself.

**Synthetic Data Generator.** Therefore, we choose to use the exemplar queries from the W3C working draft “XML Query Use Case” [W3Cc] which showcase the important features and possible styles allowed by the full syntax of XQuery. We are then able to examine the capabilities of our ACE-XQ system by performing the case studies with these queries with different features. To this goal, we implement our own XML data generator which can produce XML documents conforming to three example DTDs, i.e., bib.dtd, reviews.dtd, and prices.dtd in [W3Cc] going with those query cases.

Despite the simple schema restricted by these DTDs, the XML documents produced by our synthetic XML data generator can illustrate a variety of structures based on the parameterized input. For example, the input parameters can be used for specifying the tree fanout, data value range, ratio of optional, and random number generators for a number of attributes, which all contribute to the rich structure of XML data.

Our synthetic XML data generator hence enables us with a full control of the data being posed against by tuning the input parameters. In addition, we can easily formulate the queries with desirable characteristics such as query tree fanout, depth and selectivities, with the knowledge of
the structures and sizes of the underlying source data.

**Plans of Experimental Queries.** We plan three types of experiments each targeting a different objective as described in Section 8.1.1.

1). With a fixed data set, we perform case studies with the example queries directly extracted from the W3C working draft “XML Query Use Case” for the purpose of checking the capabilities and limitations with respect to our query containment and rewriting techniques.

For queries that can be rewritten with respect to the cached queries, we check the correctness of rewritten queries by comparing their results against those of the original ones.

For some of the queries that fail the containment checking, we identify the features ignored by our techniques.

2). With the synthetic data generator, we generate a set of XML documents with different characteristics and overall different sizes, and also adjust the example queries used for the first set of experiments to have conditions on different paths with different selectivities. We evaluate the query response time with the help of the cache system against that evaluated by the remote baseline query engine.

3). Again, we change the characteristics of the source data and queries as we did for the second set of experiments, but to test the effectiveness of different replacement strategies, namely, the total and partial replacement policies. We especially choose a refining query trace and
8.2 Experimental Evaluation

8.2.1 Use Case Studies for Approach Validation

To illustrate important applications for an XML query language, the XML Query Working Group has created a set of use cases in the categories named “R”, “XMP”, “TREE”, “PARTS”, “NS”, “STRING”, “SEQ”, “SGML”, and “STRONG”. Among them, we pick the twelve queries in the use case category “XMP”, which are exemplars representing the most commonly used cases from the database and document communities.

Now we try to pass these twelve queries through the ACE-XQ system. Table 8.1 below collects the passing/failing status for each of these use cases indicating whether they are handleable by the proposed containment checking and rewriting technique. None of the queries below can be solely handled by XPath containment. As shown in Table 8.1, half of the queries can pass through the system, indicating that these queries fall into the defined XQuery fragment and that their descriptors represented by the VarTree and TagTree structures can be registered into the cache based on which containment checking and rewriting are possible to occur.

For each pair of the handleable use cases, we further conduct the query
8.2. EXPERIMENTAL EVALUATION

containment checking and rewriting test. If there is a contained or overlapping case, we check the correctness of the rewritten queries by comparing their results with those of the corresponding original queries.

<table>
<thead>
<tr>
<th>Use cases</th>
<th>Handleable</th>
<th>Correctness / Failing Features</th>
</tr>
</thead>
<tbody>
<tr>
<td>“XMP” Q1</td>
<td>Yes</td>
<td>Correct</td>
</tr>
<tr>
<td>“XMP” Q2</td>
<td>Yes</td>
<td>Correct</td>
</tr>
<tr>
<td>“XMP” Q3</td>
<td>Yes</td>
<td>Correct</td>
</tr>
<tr>
<td>“XMP” Q4</td>
<td>Yes</td>
<td>Correct</td>
</tr>
<tr>
<td>“XMP” Q5</td>
<td>No</td>
<td>Value-based joins</td>
</tr>
<tr>
<td>“XMP” Q6</td>
<td>No</td>
<td>count(), Functions (position), If-then-else</td>
</tr>
<tr>
<td>“XMP” Q7</td>
<td>Yes</td>
<td>Correct</td>
</tr>
<tr>
<td>“XMP” Q8</td>
<td>No</td>
<td>Functions (contains, ends-with)</td>
</tr>
<tr>
<td>“XMP” Q9</td>
<td>No</td>
<td>Function (contains)</td>
</tr>
<tr>
<td>“XMP” Q10</td>
<td>No</td>
<td>Value-based joins, max()</td>
</tr>
<tr>
<td>“XMP” Q11</td>
<td>Yes</td>
<td>Correct</td>
</tr>
<tr>
<td>“XMP” Q12</td>
<td>No</td>
<td>Negation, Function (deep-equal)</td>
</tr>
</tbody>
</table>

Figure 8.1: Passing/Failing Use Cases

On the other hand, the limitation of our containment and rewriting technique is that it focuses on only a core fragment of XQuery features as defined in Section 3.2. Thus the use cases that contain negations, order-by clauses, if-then-else clauses, build-in or user-defined functions, aggregations, value-based joins, etc. simply are not handled by our current technique.

However, extensions of our technique for tackling these more comprehensive query cases are not difficult to be incorporated. Query containment in the relational context has encountered some similar problems such as
value-based joins and aggregations. The techniques used there can be borrowed and integrated into our solution. Furthermore, for each build-in or user-defined function such as contains, ends-with, or position used for condition specification, we analyze the semantics implication of the function and may derive an appropriate criteria for reasoning the condition containment. For example, if variable $v$ in $Q2$ is mapped to variable $w$ in $Q1$ in the containment mapping step, and they are respectively imposed by the conditions \( \text{where contains}(v, \text{"XML"]) \) and \( \text{where contains}(w, \text{"XML"}) \), then it is easy to tell that the former condition is stricter than the latter and hence the containment mapping from $v$ to $w$ still holds.

Query containment incorporating other functions such as deep-equal or negative where-conditions is more about the complexity issue. For example, in the relational context, the containment problem for the class of conjunctive queries was first considered. Techniques for testing the containment of conjunctive queries with other features such as safe negated subgoals were developed based on it. The containment of conjunctive queries with arbitrarily negated subgoals would jump to exponential. In this sense, the defined XQuery fragment allows us to focus on the most important features of XQuery, like the counterpart of the relational conjunctive queries, for the containment checking and rewriting problem. Extensions of the proposed technique for increasingly incorporated other advanced XQuery features remain as the future work.
8.2. EXPERIMENTAL EVALUATION

8.2.2 Query Performance With and Without Cache

In this set of experiments, we set the cache size to unlimited, which means no replacement would occur. We first conduct a series of experiments analyzing the contributive factors to the query response time. By varying the source document sizes, query selectivities, and query depths, we compare the query response time for answering a query that would be totally contained in a cached query in the case when the ACE-XQ system is used versus when it is not. Below we show the result for each of these experiments and give our brief analysis.

The Factor of XML Document Size. In the first experiment, we let the selectivity of the cached query $Q_1$ kept as 2% and that of the new contained query $Q_2$ as 0.2%. At the same time, we vary the size of the source document being posed against by both $Q_1$ and $Q_2$ from as small as 1M bytes to as large as 20M bytes, as shown on the x axis of Figure 8.2. The y axis shows the query response time in logarithmic scale for the case that uses the cache (shown by the solid-lined boxes) versus the case bypassing the cache (shown by the dotted-lined boxes) with the increasing of the source document sizes.

From Figure 8.2, we observe that the query response time in the bypassing-cache case increases very quickly with a nearly linear growth of the document sizes. Especially, the response time is in the scale of minutes which is almost intolerant when querying against a relatively large XML document ($\text{size} \geq 10M$ bytes). In contrast, the query response time in the using-cache case increases in a much slower rate. For example, when the source XML
Figure 8.2: Query Response Time for Different Document Sizes

document size is 1M bytes, the response time for the using-cache case and that for the bypassing-cache case are about 5400 ms and 12350 ms respectively. The time saved by using cache is about a half of the time spent by evaluating the query using the remote query engine in this case. When the source XML document increases to 20M bytes, the time spent in the using-cache case is about only 1/200 of that used in the bypassing-cache case.

The time saving achieved by the use of cache over the baseline remote query engine comes from two sources. One, the query as well as the result shipping time is totally saved for the using-cache case when \( Q_2 \) is contained within \( Q_1 \). Two, the cached result for \( Q_1 \), which is a small percentage (i.e., 2%) of the source XML document, is the source document being queried against by the rewritten \( Q_2 \). Since the efficiency of query evalu-
ation is largely dependent on the size of the input document, the smaller cached view document contributes to the response time saving.

The Factor of Query Selectivity. In this experiment, we fix the source XML document size to be 10M bytes and test two scenarios when the cached query $Q_1$ has different selectivities, i.e., 2% and 20% respectively. We then vary the selectivities of $Q_2$ from 0.2% to 20%. When the selectivity of $Q_1$ is 20%, the series of $Q_2$s are all contained within $Q_1$. Whereas when the selectivity of $Q_1$ is 2%, the first four $Q_2$s whose selectivities are 0.2%, 0.5%, 1% and 2% are contained within $Q_1$ while the later three $Q_2$s each contains $Q_1$. That is, $Q_1$’s result in this case is only part of the answers to these three $Q_2$s. Remainder queries need to be sent to the remote query engine to be evaluated and the returned answers need to be combined with $Q_1$’s result.

Figure 8.3 shows that the response time for $Q_2$ with respect to the cached $Q_1$ when they have different selectivities. We can see that when $Q_2$ is contained within $Q_1$, whose selectivity is either 2% or 20%, the response time for $Q_2$ by using the cache is much smaller than that by directly sending $Q_2$ to the remote query engine. Furthermore, the response time for $Q_2$ with respect to the $Q_1$ with the smaller selectivity, i.e., 2%, is even more reduced compared to when $Q_1$ has the selectivity of 20%. This is because that $Q_2$ is totally contained with $Q_1$ in both cases and its rewritten query can be evaluated faster against a smaller view document for $Q_1$ than against a relatively larger one.

However, when $Q_1$ has the selectivity of 2%, it cannot contain $Q_2$ with the selectivity larger than 2%. The last three groups of boxes in Figure 8.3
show that the response time for \( Q_2 \) when using the cache is comparable to that when the cache is skipped, because the former consists of the time spent on shipping the remainder queries and their results as well as on the result combination.

The experimental results shown here suggest that in the case when \( Q_2 \) is overlapping with \( Q_1 \) instead of totally contained within \( Q_1 \), the benefits in terms of response time gained by using the cache may be low or even negative. However, if the user cares more about the initial response time than about the complete delivery of the final result, then using cache for answering the new query even if it is not a totally contained case would be favorable. In fact, a research trend has emerged recently addressing such user preferences of delivering a subset of results fast in the area of “web
search” as well as “continuous query answering” and “fast approximate query answering”.

Furthermore, probe query and remainder query can be evaluated in parallel. This may off-load some of the burden on the remote data servers especially when only a small portion of the query answers are missing in the cache. It also helps to improve data availability in the case of network failures and unreliable data servers. In such cases, the query answers contained in the cache can be made available immediately, even if they are still incomplete. Hence, we make the option of using semantic cache for the overlapping case open and leave the decision to the user.

The Factor of Query Depth. In this experiment, we fix the source XML document size to be 10M bytes and the selectivity of the cached query \( Q_1 \) to be 5%. We vary the nested level of \( Q_2 \) from 1 to 4 while keeping its selectivity to be 2% always. All these tested \( Q_2 \) are contained with \( Q_1 \). Figure 8.4 shows that the response time for \( Q_2 \) does not change significantly with different depths for the nested query structure.

The Time Decomposition of Computational Overhead. From the above three sets of experiments, we have observed that the time spent for evaluating a totally contained query by using the cache is usually faster than that for evaluating the query remotely. As explained earlier, this can be attributed to the saving of transmission time and the small size of cached view document.

However, using cache for answering new queries comes with the com-
8.2. EXPERIMENTAL EVALUATION

Figure 8.4: Query Response Time for Different Query Depths

computational overhead for decomposing the incoming query and constructing its VarTree and TagTree representations, for containment checking and query rewriting, for evaluating the rewritten query with respect to the cached query result. For a contained case where the remote XML document size is 10M, the selectivity of the cached query is 2% and that for the new query is 0.5%, the breakdown of computational overhead is shown in the pie chart in Figure 8.5.

We can see from this pie chart that the computational overhead in terms of query decomposition, containment checking and query rewriting is very slim comparing to the rewritten query execution time. This is because that the first three time components are related to the query tree size (i.e., VarTree and TagTree sizes), which is usually much smaller than the document size, the main contributive factor for the total evaluation time.
Query Workloads With Different Characteristics. Now, we conduct an experiment comparing the query response time using cache versus without cache over a query workload with certain characteristics. First, we “warm up” the cache with several queries and then use several types of query workloads to drive the system. We use the average response time per query in a workload to compare the performance when using the cache system versus when directly fetching the result from the remote server.

In this experiment, we work with three distinct query workloads, each composed of 40 XQueries. In the first query workload, the later incoming queries are designed to be totally contained in one of the previous queries in the trace. So every time the new query can be answered by a cached query. The second query trace has a mixture of totally contained queries and partially overlapping queries. Some of them differ by conditions, while others have different return expressions. For the third query workload, we randomly generate a set of queries. We vary the source document size and
run these traces on 8 different source documents.

![Figure 8.6: Contained Query Workload](image)

We show in from Figure 8.6 to Figure 8.6 that the query response time for these three different query workloads in two scenarios, i.e., when using the cache versus when directly fetching the result from the remote server without cache. Consistent with our expectation, the results shows that the response time decreasing utilizing the cache system is obvious for all three workloads. In particular for the query workload with only contained queries in Figure 8.6, the reduction in response time delay is rather significant. We can also see from the result that both the response time using and not using the cache will increase with the growth of the source XML document size.

For the mixed query workload, subsequent queries are either totally contained in or overlapping with the cached queries. From Figure 8.7 we observe that the overage query response time when using the cache is
smaller than when bypassing the cache. However, the degree of response time reduction for the mixed query workload is not as high as that for the totally contained query workload.

If the query workload consists of randomly generated queries which
may fall in either the totally contained case, the overlapping case, or the disjoint case, then there is still some performance gains by using the cache over bypassing it, although the gap between the response time delay in two scenarios is even more reduced compared to for the mixed query workload.

8.2.3 Experiments on Replacement Strategies

We now compare the performance of the two different replacement strategies, i.e., partial replacement versus total replacement, that are employed in ACE-XQ when the cache space is limited. We generate two workloads for this purpose. Each workload has one query trace composed of 40 queries that query against 10 different XML documents located on remote web servers. The average document size is around 175K bytes, and totally is about 1.75M bytes.

Queries in the random trace are randomly selected from all possible user select-project user queries against source XML documents. The second query trace contains only refining queries on different XML source files, i.e. most subsequent queries are contained in some previous queries requesting the same documents. We refer to the first query trace as a random trace and the second one as a refining trace. Practical scenarios for both query traces can easily be found. For example, in a web search, a user may issue a query with conditions expressing her main concerns before she has enough knowledge about the queried web source. She then may refine the query conditions over time based on the information gained from previous query

\(^1\)Actually in the case when the partial replacement strategy is applied, we cannot guarantee that the subsequent queries are totally contained in previous ones, since part of a cached query result may be already have been replaced.
results. So this would form a trace of type refining query with more and more refined conditions and thus smaller returned query result.

In both query traces, we control the query selectivities to range from 15% to 70%. That is, for a query imposed against a source document of 180K bytes, the returning result size varies from 30K bytes to 130K bytes. Initially, we set the cache space to be 150K bytes and then increase it each time by 50K bytes until it reaches 1.2M bytes. That is, for each designed query trace, our smallest cache size is still large enough to hold at least one query result, while the largest cache may approximately hold the most frequently queried fragments of all the source XML documents in the most ideal scenario. In the following experiments, we employ three metrics as shown in Table 8.1 to measure the query performance.

<table>
<thead>
<tr>
<th>Metric (acronym)</th>
<th>Meaning</th>
<th>Formula</th>
</tr>
</thead>
<tbody>
<tr>
<td>Hit Count Ratio (HCR)</td>
<td>Number of queries that re-use cached query results compared to total number of asked queries</td>
<td>$\frac{#\text{Cache Hits}}{#\text{Total Queries}}$</td>
</tr>
<tr>
<td>Hit Byte Ratio (HBR)</td>
<td>Average bytes of cached query results that are re-used compared to cache size</td>
<td>$\frac{\sum\text{Reused Bytes}}{#\text{Total Queries} \times \text{Cache Size}}$</td>
</tr>
<tr>
<td>Response Time (RT)</td>
<td>Average response time per query in a query trace</td>
<td>$\frac{\sum\text{Response Time}}{#\text{Total Queries}}$</td>
</tr>
</tbody>
</table>

Table 8.1: Query Performance Metrics

As shown in the replacement function $rp_{fun}$ in Section 6.1, the usage information hits (hit counts) recorded in the path table of a cached query is one of our main statistics in determining the victim XPathRows in our partial replacement strategy. In other words, the LFU policy is in-
corporated into our cache replacement strategy for the selection of the victim. However, if more than one XPathRow is selected as victim having the same hit counts, then we will pick one of the victims using the LRU policy. The replacement will then be in favor of the XPathRow with the earliest last_access_time. Using LFU or LRU as the main criteria may affect the comparison of the performance between caching and non-caching. But for both partial and total replacement strategies we select the same replacement policy from LFU or LRU.

**Hit Count Ratio Comparison of the Two Replacement Strategies.** In this set of experiments, we compare the query performance of the two replacement strategies in terms of the Hit Count Ratio (HCR). Each time when a new query is contained or overlapping with one of the queries in the cache and hence a probe query is generated, we consider this as a cache hit. HCR refers to the percentage of such hits over the total number of queries in the given workload (see Figure 8.1).

Figures 8.9 and 8.10 show the HCR for the refining and random query traces respectively. Two different replacement strategies are partial and total replacement. For both strategies and for both query traces, HCR increases with the increase of the cache size. Overall, we see that HCR of partial replacement is always higher than that of total replacement. From both Figures 8.9 and 8.10, we see that the relative HCR improvement of the partial replacement compared to the total replacement is declining when enlarging the cache size. The “zigzag” for the small cache mainly is due to the “thrashing” activity of cache when it is rather small and not stable. This
8.2. EXPERIMENTAL EVALUATION

Figure 8.9: Refining trace: HCR for Two Replacement Strategies With Varying Cache Sizes

also explains the unstable curves for the relative HCR improvement of our partial replacement over the total replacement during these stages. However, the relative HCR improvement is in general positive, indicating that our partial replacement strategy always wins over the total replacement strategy.

For Figures 8.9 and 8.10, our partial replacement outperforms the total replacement with relative improvement of HCR at 40%-50% for both query traces when the cache works with HCRs for both strategies from 30% to 50%. This would happen for a medium size cache, when replacements may not occur that frequently for both strategies. In this case the cache may hold more valuable queries when the partial replacement is applied than when the total replacement is used. The reason is that the partial replacement
can adjust the cached queries and preserve only the more useful partial queries. This way, a given size cache holds likely more but smaller queries for the partial replacement than for the alternative total replacement. While for a very large cache, the cache space resource is not precious any more. The query region refinement pursued by partial replacement becomes less critical.

As we have expected, both replacement strategies are especially beneficial for the refining query trace. The sequence of queries in this trace are designed to have more contained query cases than the other, hence more cache hits are expected. We can see that the relative improvement of the partial replacement over the total replacement for the refining query trace is better than that for the random query trace. The more total containments cases occur between the new queries and the cached ones, the more parts
8.2. EXPERIMENTAL EVALUATION

in cache are being reused. So partial replacement can then work more precisely to keep these “hot parts” that are likely to be used later.

**Hit Byte Ratio Comparison of the Two Replacement Strategies**  
We now use the metric of *Hit Byte Ratio (HBR)* to compare the performance of the two different replacement strategies. The two curves in Figure 8.11 represent the HBR for the partial and total replacement strategies respectively. When the cache size is small, the HBR is shown to be about 10% and 8%, with the partial strategy winning over the total one by 2%. With an increase of the cache size, both HBRs slowly decline to about 4% when the cache size reaches 1.2M bytes. The second chart in Figure 8.11 illustrates the relative HBR improvement of the partial replacement over the total replacement for the refining query trace. This is shown to be above 30% when the cache is roughly of medium size.

We then repeat this experiment with the random query trace. The HBR results are shown in Figure 8.12. The HBRs for both replacement strategies follow the same trend as for the refining trace, decreasing when the cache size becomes gradually larger. Although the partial replacement still outperforms the total replacement in most cases, the relative HBR improvement is shown to be less significant than that for the refining trace.

We can also observe that in some cases when the cache size is very large, the partial replacement may not always work better than the total replacement. We interpret this phenomenon as below. When applying the partial replacement strategy for a large cache, the cached queries tend to become smaller due to the strategy of partial replacement even though this is not
necessarily be needed given the availability of space. Therefore, even if the hit count ratio (HCR) of the partial replacement may be slightly higher than that of the total replacement, the average bytes used by each hit in a partial replacement scenario are likely smaller than those used in the total replacement.

Figure 8.11: Refining trace: HBR for Two Replacement Strategies With Varying Cache Sizes
8.2. EXPERIMENTAL EVALUATION

Figure 8.12: Random trace: HBR for Two Replacement Strategies
With Varying Cache Sizes

Response Time Comparison of the Two Replacement Strategies. In the last set of experiments, we study the effects of the two replacement strategies on the query performance in terms of response time. This intuitively has a close relationship with the HCR and HBR measures. Since the response time for a local probe query is usually much smaller than that for a query fetching results across the Internet, the higher the HCR or the HBR is
the larger a reduction in the average query response time is likely achieved.

![Figure 8.13: Refining trace: RT for Two Replacement Strategies With Varying Cache Sizes](image)

In both Figures 8.13 and 8.14, we compare the response times under three scenarios: 1) retrieving the query answers by *bypassing* our caching system and directly from the remote data server; 2) utilizing our caching system supported by the total replacement strategy; 3) utilizing our caching system...
system supported by the partial replacement strategy. Figure 8.13 shows that the response time for the bypassing scenario (without cache) is roughly the same, i.e., about 5 seconds. The partial replacement strategy helps to improve the query performance by about 25% to 35% in terms of the average reduction in response time, while the total replacement achieves about 15% to 25% improvement on average.

![Graph showing response time vs cache size for different strategies](image1)

![Graph showing relative decreasing of response time vs cache size for different strategies](image2)

Figure 8.14: Random trace: RT for Two Replacement Strategies With Varying Cache Sizes
These experiments show that our partial replacement strategy outperforms the total replacement in most of the tested scenarios. In particular the partial replacement strategy will perform better than the total replacement when the cache is running at a medium size. This means the cache hit ratio is about 40% to 50%. Then the cache size is neither too small to hold the result for a single query, nor too huge to make a replacement of necessity since all the query results can be held in the cache. We can have over 40% relative improvement on hit count ratio and 15% to 30% on relative improvement on hit byte ratio under this scenario. It is also clear that the partial replacement will be more useful when the user query trace is in the favor of refining queries. So when there are some queries totally contained in previous ones in the user access history, the partial replacement will win.
Part V

Conclusions
Chapter 9

Conclusions and Future Work

9.1 Conclusions

Due to the growing demand by web applications for retrieving information from multiple remote XML sources, it has become increasingly critical to improve the efficiency of XML query evaluation. One key step towards achieving such an optimization is to exploit caching technology to reduce the response latency caused by data transmission over the Internet.

Different from the traditional tuple or page-based caching systems, semantic caching systems exploit the idea of reusing the cached queries and their results to answer subsequent queries by reasoning about their containment relationships. With respect to the cache space management, the data cached in the semantic cache is logically organized by queries instead of physical tuple identifications or page numbers. The access and management of the cached data in a semantic caching system is thus typically at the level of query descriptions.
9.1. CONCLUSIONS

Since query containment plays a fundamental role in many database applications including query optimization, information integration, and semantic caching, it has received considerable attention over the past few decades with the main focus on relational queries. With the emergence of XML as the principal medium for data exchange over the Web and its increasingly popular query languages, there is a resurrected theme in studying containment for queries over semistructured data like XML. This may significantly enhance the efficiency of XML query processing in the Web environment. However, the state-of-the-art research in this field is on XPath containment while containment for more comprehensive XML query languages such as XQuery has so far been ignored. The challenges arise for bridging the gap between the containment for XPath and for XQuery due to the many rich and powerful features added on the former to compose the latter.

In addition, the data granularity being managed in an XQuery-based semantic cache is the result XML view document of a query. There hence induces some cache management issues cannot handled well by the traditional techniques. One problem is the replacement. Purging a victim query means purging the whole XML document associated with it, which can be huge. Such a coarse-grained replacement may result in low cache space utilization, high region splitting overhead, or imprecise statistics on the user query pattern. Therefore, an effective replacement strategy that can better adapt to the user query access pattern and improve the cache space utilization is needed. Also, research work on XML-query-oriented semantic cache region coalescing or coherence is largely left unexplored yet.
Hence, the goal of this dissertation is to provide a practical framework for the XQuery-based semantic caching systems. We have concentrated on two key issues fundamental to such a semantic caching system: the containment of XQuery, and the replacement strategy better adaptable to user query pattern.

9.2 Results and Contributions of this Dissertation

XQuery Containment and Rewriting

XPath is a popular sub-language of most XML query languages and hence its containment has received much attention recently. Compared to XPath, XQuery is more powerful in expressing variable bindings, condition filtering, and result construction in an arbitrarily composed nested form. Pattern matching and result construction are the two semantic components of XQuery connected by the use of variables. They are however specified in an interleaving manner for progressively constructing the output tree by utilizing the intermediate variable bindings. The obstacles for reasoning about XQuery containment are hence the arbitrary query syntax, the flexible uses of variables, and the obscure containment conditions.

In Part II of this dissertation, we have shown a solution for determining the containment of a defined XQuery fragment and for rewriting a query with respect to the containing query. As described in Chapter 3, the XQuery containment approach is based on the variable dependencies induced from both the variable definitions and the query program control flow. First, a technique is proposed for identifying essential variables, which can then
be utilized in the normalization process to derive the minimal query form. A given query in its normalized form can then be broken into two trees which are connected by essential variables. Namely, a VarTree contains the dependencies among the essential variables and a TagTree captures the nested result structure of the query.

The VarTree-based MAC mapping approach is then introduced for building the containment mapping between the essential variables of two queries. In this mapping process, the top-down pattern tree matching provides the guidance for finding the candidate variable mapping pairs, while XPath containment helps to resolve the containment of the two bound expressions for a candidate variable mapping pair. The control-flow induced variable dependencies are also dealt with by incorporating an additional checking step in the MAC mapping process. We prove the correctness of our XQuery containment technique by showing the connection of the VarTree-based MAC mapping procedure with the containment mapping between two monadic datalog programs each representing a given query.

New challenges are also posed on the problem of XQuery rewriting due to the arbitrarily nested view structure that the new query is rewritten against. The intuitive idea is to rewrite the bound expression for each of the variables in the new query based on the location of its mapped variable in the TagTree structure. The knowledge of variable mapping pairs obtained from the containment checking phase and the TagTree structure of the cached query need to be utilized. However, the rewritten query produced in this way may not exactly preserve the restructuring semantics of the original query.
In Chapter 4, we have given a comprehensive query rewriting technique based on a careful equivalent translation of the result construction semantics captured by the TagTree structure of the new query, besides the knowledge of variable mapping pairs and the TagTree structure of the cached query. Such a translation involves the edge mapping between two TagTrees, which corresponds to the mapping between the variable groups for the input of the corresponding skolem functions. The precise preserving of the result construction semantics by the query rewriting can be guaranteed by the equivalent skolem function translations.

In Part IV of this dissertation, we have described a semantic caching system called ACE-XQ that is designed and implemented based on the proposed XQuery containment and rewriting techniques. As described in Chapter 7, the ACE-XQ system is built over the state-of-the-art XQuery engine and extends its query evaluation functionality with the capabilities of caching queries and reusing their results for answering new queries.

In Chapter 8, we have shown by case studies the correctness of our containment checking and rewriting techniques. Among the use cases that fall in our defined XQuery fragment, if there is a contained or an overlapping case, the result produced for the rewritten query is compared to that generated by the direct query evaluation. The correctness our approach is confirmed by these case studies. We have further conducted experiments to evaluate the query performance affected by the adoption of the ACE-XQ system. Experimental results have shown that the ACE-XQ system can improve the query performance for all the three types of designed query workloads. The most significant performance improvement is for the query
trace composed of continuously refining queries, the less is for the query trace containing both contained and overlapping cases, and the least is for the one with random containment cases.

We have also studied the effectiveness of different factors such as the source document sizes, query selectivities, and query depths towards contributing to the query response time. The source document size seems to be the main factor. This suggests that we may use the query selectivity to determine whether to cache a query or not.

**XQuery Cache Replacement Strategies**

A semantic cache manages the access and manipulation of the cached data typically at the query level by using some concise form of query descriptions. In the relational context, a query is composed of relational operations such as selections by conditions imposed on a set of attributes and projections of a group of attributes. In the relational query-based caching systems, a query region is usually the minimal granularity managed in the cache, and in particular used as the replacement unit when caching the new query needs more room. Although different replacement strategies may favor region splitting before a replacement or no splitting but accommodating the potential data redundancy, they all suffer the drawbacks from associating the replacement unit tightly with the total query region.

In Part III of this dissertation, we have proposed a refined replacement strategy which utilizes the utility values recorded at a finer granularity, i.e., on the XPath level of the view structure, than the whole query region. That is, each query descriptor is attached with a detailed path table list-
9.3. IDEAS FOR FUTURE WORK

Incorporating all paths returned in the query. These path-based utility statistics is updated upon the query containment decision which indicates which path sets are requested by the new query. At the same time, the whole query region is not split. When the cache is full, the replacement manager does not select complete regions but only specific paths with the lowest utility value within such query regions for replacement. A filter query is then composed to remove the fragments corresponding to those paths from the cached XML view. The relevant query descriptors are then modified accordingly to be consistent with the changed XML view.

The experiments comparing the performance of our partial replacement and the total replacement strategies are illustrated in Chapter 8 for different query workloads. It shows that both replacement strategies are especially beneficial for the refining query trace, and that the relative improvement of the partial replacement over the total replacement for the refining query trace is somewhat better than that for the random query trace. The more total containments cases occur between the new queries and the cached ones, the more parts in cache are being reused. The experimental results also show that in some cases when the cache size is very large, the partial replacement may not always work better than the total replacement.

9.3 Ideas for Future Work

Query Containment Incorporating Extended XQuery Fragment. The XQuery fragment defined in this dissertation provides a good scope for us to focus on a set of important XQuery features with respect to the containment prob-
The future work is to extend the proposed containment mapping approach to accommodate a broader fragment of XQuery that includes value-based joins, aggregations, and disjunction among others. It also would be interesting to study XQuery containment with the presence of DTD or path constraints.

In addition, we expect some heuristics can be applied for “pre-selecting” containing query candidates in the cache. Our current solution indistinguishably requires a pre-processing step which contains variable minimization, normalization, and decomposition for each incoming query, and then performs the MAC mapping based on the obtained VarTree structures. The overhead of this pre-processing step can be saved or reduced if appropriate heuristics are applied. For example, we may utilize a very simple parser to extract information such as the source document from an incoming query, based on which some quick prune process can be performed to narrow down the search space within the cached queries.

**Cache Region Coalescing.** Another direction of the future work is to investigate an effective cache region organization technique. When the cache size is large and can hold many cached queries, the time spent on iterating through them in search of a potential matching (containing or overlapping) query can be substantial. As a result, the performance of the query containment decision making may be greatly impacted due to the large search space of candidate queries in the cache.

Hence, how to organize the cached queries in a hierarchical structure to explicitly reveal their semantic relationships becomes an interesting topic.
9.3. IDEAS FOR FUTURE WORK

An immediate thought to this is to utilize an appropriate hierarchy to index query semantic descriptors instead of the cached data. This way, the search for a candidate matching query among many cached queries in order to make query containment decision can be facilitated. This should consequently improve the efficiency of query processing.

Another concern is that the redundancy of the cached query results due to overlapping query semantics may cause the space utilization to deteriorate drastically. When a large proportion of the cache space is wasted on caching redundant data, especially when the needed space exceeds the size of the remote XML documents being queried against, we could potentially do better by simply caching the complete documents instead of caching their pieces several times inadvertently due to being part of several overlapping queries.

These issues of effective query organization and data redundancy control of the cache have been ignored in the current design of the ACE-XQ cache manager. A useful extension of the current ACE-XQ system could be adopting a dynamically adjustable index structure for organizing the cached queries in a hierarchy, which may also help to maintain the local knowledge reservoir accumulated over time in terms of query results. In addition, such a hierarchy can be used to prune a large search space for candidate matching queries. It may provide a framework that facilitates making intelligent replacement decisions.

Cache Coherence. Future work could also include the investigation of different cache maintenance policies such as invalidation of infrequently
used query results and update of frequently used ones using advanced techniques to determine update relevancy. For example, I have done previously a work designing an incremental maintenance mechanism based on a novel index structure for the XQL-based web view system. The key idea is to first check the relevancy of a specific update with respect to the view by analyzing the conformance of the updated structure or value to the view definition, and then to propagate the relevant updates to the view. It will be interesting to extend this work and apply it to the XQuery-based semantic caching system. Also, investigating the approach for generating update notifications is another aspect of future work.
Bibliography


BIBLIOGRAPHY


