Self Maintenance of Materialized XQuery Views via Query Containment and Re-writing

by

Shirish K. Nilekar

A Thesis Report

Submitted to the Faculty

of the

WORCESTER POLYTECHNIC INSTITUTE

in Partial Fulfillment of the Requirements for the

Degree of Master of Science

in

Computer Science

by

April 20, 2006

APPROVED:

Prof. Elke A. Rundensteiner
Advisor

Prof. George T. Heineman
Reader

Prof. Michael A. Gennert
Head of Department

Abstract

In recent years XML[29], the eXtensible Markup Language has become the de-facto standard for publishing and exchanging information on the web and in enterprise data integration systems. Materialized views are often used in information integration systems to present a unified schema for efficient querying of distributed and possibly heterogenous data sources. On similar lines, ACE-XQ [4], an XQuery [32] based semantic caching system shows the significant performance gains achieved by caching query results (as materialized views) and using these materialized views along with query containment techniques for answering future queries over distributed XML data sources. To keep data in these materialized views of ACE-XQ up-to-date, the view must be *maintained* i.e. whenever the base data changes, the corresponding cached data in the materialized view must also be updated. This thesis builds on the query containment ideas of ACE-XQ and proposes an efficient approach for self-maintenance of materialized views. Our experimental results illustrate the significant performance improvement achieved by this strategy over view re-computation for a variety of situations.

Acknowledgements

Many people have contributed directly and indirectly to this thesis. First and foremost, I would like to express my sincere thanks to Dr. Li Chen whose excellent work on the ACE-XQ research project provided the basis for this thesis work. Her suggestions, explanations, and co-operation from the beginning to the very end have been instrumental in me completing this thesis. This thesis would not have been possible without you, Lily.

My thesis advisor, Prof. Elke A. Rundensteiner provided superb guidance by asking all the right questions, including the difficult ones and used her extensive research experience to make some important decisions during the course of this thesis. Her patience, understanding and an infectious enthusiasm for work have gone a long way in completing this thesis. I could not have asked for a better advisor. Thank you, Elke.

I would also like to thank my thesis reader, Prof. George T. Heineman for reviewing my work and providing me with valuable suggestions for improving the quality of this thesis. My special thanks also go to Prof. Emmanuel Agu for providing me with financial assistance at a crucial time during my graduate studies. And, it was also great fun to work on his interesting research project, MADGRAF.

Finally, and equally importantly, it was my family, friends, and a few very good professors who made my grad school experience so enjoyable. I cherish the good times I had while at WPI. And with you folks around, I can only expect even better things in the future!

Contents

1 Introduction **1**
- 1.1 Motivation . . . 2
- 1.2 Terminology . . . 5
- 1.3 Summary of ACE-XQ . . . 7
- 1.4 Thesis Overview . . . 10
  - 1.4.1 Problem Definition . . . 10
  - 1.4.2 From XQuery Containment to View Maintenance: Our Approach to View Self Maintenance . . . 11
- 1.5 Contributions . . . 19

2 Preliminaries **21**
- 2.1 XML Query and Update Languages . . . 22
- 2.2 AceXQ . . . 26
- 2.3 XPath Containment . . . 26
- 2.4 An Example . . . 29

3 Our Approach To View Self Maintenance **34**
- 3.1 Overview . . . 35
  - 3.1.1 A Query Subset Suitable for View Self Maintenance . 36
- 3.2 Query Pre-processing . . . 37
  - 3.2.1 Variable Minimization . . . 37
  - 3.2.2 Query Normalization . . . 38
  - 3.2.3 Query Decomposition . . . 41
- 3.3 Update Query Analysis . . . 45
  - 3.3.1 Using XPath Containment for View Self Maintenance 46
  - 3.3.2 Handling Insert Operations . . . 49
  - 3.3.3 VarTree Node Mappings for Query Containment . . 52
  - 3.3.4 VarTree Node Mappings for View Self Maintenance . 55
  - 3.3.5 The Analysis Algorithm . . . 57

3.4 Update Query Rewriting 63
3.5 Complexity Analysis of Analysis Algorithm 66
3.5.1 Cost of MAC Mapping 66
3.5.2 Cost of Analyzing Update Nodes 67
3.6 Examples 68
3.6.1 Case 1: Relevant Delete Operation 69
3.6.2 Case 2: Non Relevant Delete Operation 71
3.6.3 Case 3: Non-determinable Delete Operation 73
3.6.4 Case 4: Relevant Insert Operation 75
3.6.5 Case 5 Non Relevant Insert Operation 80
3.6.6 Case 6: Non determinable Insert Operation 80
3.6.7 Case 7: Other Undeterminable Updates 81
3.6.8 Update Query Rewriting Example 81

4 System Implementation and Experiments 85
4.1 System Implementation 86
4.2 Experimental Setup and Results 90
4.2.1 Cost of single update operation 91
4.2.2 Effect of changing the document size 92
4.2.3 Effect of changing the view query selectivity 94
4.2.4 Effect of changing the update size 94
4.2.5 Cost of Query pre-processing and Query Analysis . . 96

5 Related Work 99
5.1 Related Research 100

6 Conclusion 103
6.1 Results and Contributions of this Thesis 104
6.2 Ideas for Future Work 105

List of Figures

1.1 ACE-XQ Overview 9
1.2 Thesis Framework Overview 14

2.1 Syntax Of XML Update Queries 25
2.2 The XMark Schema 30
2.3 An example XML document 31
2.4 View Query 32
2.5 Result of evaluating query in Figure 2.4 on XML document . 32
2.6 Non Relevant Update Query 32
2.7 Relevant Update Queries 33

3.1 XQuery Before Normalization 38
3.2 XQuery After Normalization 39
3.3 Update Query Before Normalization 40
3.4 Update Query After Normalization 40
3.5 VarTree Representation of Query in Figure 3.2 42
3.6 TagTree Representation of Query in Figure 3.2 43
3.7 VarTree Representation of Update Query in Figure 3.4 45
3.8 Update Query Analysis Algorithm 59
3.9 Update Node Analysis Algorithm 60
3.10 Procedure RewriteUpdateQuery 64
3.11 Algorithm to get mapping paths from a TagTree 65
3.12 Normalized View Query used in examples 68
3.13 Relevant Delete Query 69
3.14 *VarTree* representation of query in Figure 3.13 70
3.15 *VarTree* mappings for Case 1 71
3.16 Non Relevant Delete Query 71
3.17 *VarTree* representation of query in figure 3.16 72
3.18 *VarTree* mappings for Case 2 73

3.19 Non-Determinable Delete Query 73
3.20 *VarTree* representation of query in figure 3.19 74
3.21 *VarTree* mappings for Case 3 75
3.22 An Insert Query relevant to View Query in Figure 3.2 77
3.23 VarTree representation of query in Figure 3.22 77
3.24 VarTree mappings for Case 4 78
3.25 Generated Document For analyzing insert query in Figure 3.22 79
3.26 Result of evaluation view query on the generated document of Figure 3.25 79
3.27 Non determinable insert query 81
3.28 TagTree Representation of View Query in Figure 3.2 82
3.29 VarTree of relevant Delete operation 82
3.30 Rewritten Update Query 84

4.1 The ACE-XQ System Architecture 87
4.2 The ACE-XQ Architecture with View Self Maintenance . . . 88
4.3 Cost of single Update Operation 93
4.4 Test for changing document size 94
4.5 Test for changing view query selectivity 95
4.6 Test for changing size of delete operation 96
4.7 Test for changing size of insert operation 97
4.8 Cost of Query Preprocessing and Analysis for 5 successive trials of experiment in Figure 4.4 98

Chapter 1

Introduction

Introduction

1.1 Motivation

XML has definitely moved from a hype to a mainstream technology in recent years. It is commonly used for information modeling [10] and information interchange [31]. It is also supported as a data-type in popular commercial relational databases like Oracle, IBM DB2, Microsoft SQL Server. With the aim of providing an XML interface to non-XML data sources, the research community has investigated publishing relational and object relational data sources [3, 13] as XML views. Commercial products like BEA WebLogic's Liquid Data, and the Xyleme Content Management Solution also essentially aggregate data from heterogenous sources like Web services, databases, flat files, XML files, applications, and Web sites and publish it in XML format. Due to its wide usage in information modeling and interchange, XML is an ideal technology for providing a unified schema over heterogenous and distributed data sources (i.e. information integration systems). Information integration is an interesting research as well as a business problem and practical applications of effective information integration abound in various real world domains like finance, health care etc. The ACE-XQ [4] research project at WPI is aimed at exploring the effectiveness of using semantic caching techniques for efficient querying of distributed XML data sources using XQuery. ACE-XQ caches user's queries and organizes the answers to these queries by their query descriptions. These locally cached query results are *materialized views* over the data sources. Based on these descriptions, the semantic caching system determines if a new input query can be answered either partially [1] or completely using data cached in the materialized views. The overall performance of ACE-XQ is significantly enhanced by keeping frequently required [2] query results (*i.e. materialized views*) in the local cache thereby improving the chances of finding answers for future input queries in the local cache instead of having to query the remote data sources.

In addition to the central issue of determining the semantic containment of input and cached queries, an important problem in the ACE-XQ system is that of view maintenance, i.e. the materialized views may need to be refreshed when the base data changes. In the absence of view maintenance, the materialized views may contain stale data and input queries answered using these cached views may yield incorrect results. These materialized views can easily be maintained by invalidating and recomputing the views for each change in the base data sources. Such a brute-force approach is

[1]In such a case, the input query us decomposed into two queries which query the remote data and the local data and the results from two queries are coalesced together to form the input query result.

[2]The ACE-XQ system maintains a dynamic *utility* metric for cached query results based on their usage in answering user queries. This metric is used to isolate the more frequently used query results from those less frequently used.

often wasteful in many cases. A batch-oriented approach commonly used in practice is to periodically refresh all materialized views, say, every six hours, and/or also provide the ability for users to request a refresh of all materialized views before submitting their queries. This batch-oriented approach can improve the overall performance of the caching system, but it can and does result in input queries returning stale data in many instances. An ideal approach would be one which efficiently detects relevant updates and modifies only the affected parts of materialized views without much delay [3]. As an example, consider a base database containing flight information from various airlines at an airport. A caching system which has access to this database may contain a materialized view of all direct flights from San Francisco to Hawaii. In this scenario, suppose an airline changes the flight number of its flight from San Francisco to Hawaii. This change can be achieved by a single update query on the base database, and the materialized view can also be maintained via a single update query. Updating just the affected data in a materialized view via a single update query is much more efficient than re-computing it. Such intelligent algorithms greatly improve the overall performance of caching systems like ACE-XQ. Our thesis proposes an *algorithmic* [4] solution based on query containment techniques for intelligently maintaining materialized XQuery views in ACE-XQ. We focus on a subset of XQuery for which deciding containment between queries is possible. The exact details of are provided later in this

[3]In this thesis, we do not assume that updates to the base data and the materialized view happen in the context of a transaction.

[4]*Algebraic* approaches [9, 11] to incremental xml view maintenance also exist. The strength of a view maintenance approach (in terms of cases it can handle) depends on the amount of information available for view maintenance [16], regardless of the approach.

thesis.

This thesis report is organized as follows: The remainder of this chapter gives an overview of the problem, and our approach to solve it. Chapter 2 introduces the background concepts necessary to understand our approach, the details of which are presented in Chapter 3. In Chapter 4 we discuss our system implementation and experimentation details. Chapter 5 explains the related research from the literature and then we conclude this thesis in Chapter 6 with a recap of the essential ideas from this thesis and suggestions for future work in this area.

1.2 Terminology

We first explain the meaning of some terms used throughout this thesis.

XML Database Any repository of XML documents which can be queried via an XML query language like XQuery [32]. XML documents in such a database are also called *base document(s)* or *base data*.

XML Query Language A language used to query XML documents or databases. XQuery [32] is now an accepted industry standard as an XML Query Language.

XML Update Language A language used to update XML documents or databases. Currently, there is no accepted standard for an XML Update Language. We present a discussion of some existing XML update languages in section 2.1.

XML View A view is a derived XML document defined in terms of base XML document(s). A view is defined via an XML query language like XQuery. The view definition query is simply called a *view query*.

Materialized XML View A materialized view (MV) is similar to a view but the view query is actually computed and its results are persisted, usually on disk. Evaluating an XQuery on an XML database results in XML data. All materialized views in this thesis are, in fact, XML documents. Materialized views are often used for improving query performance in databases, integrating data from remote databases, or in some cases to make a snapshot of a database available on a remote system.

Relevant and Non Relevant Update Queries Data in a materialized view is derived from that in the base database. An update query on the base database [5] which requires the materialized view to be refreshed to keep it in sync with its base data is called a *relevant* update. *Non-relevant* updates do not affect data in materialized views and hence do not result in refreshing the materialized view. Note that relevant and non-relevant updates are queries on the *base document* and not updates to the materialized views.

Incremental Maintenance of Materialized Views Materialized views must be refreshed when the data in the underlying database is changed. This process of refreshing the data in a materialized view is called *view maintenance* [16]. This can be achieved by refreshing (i.e. re-computing) the

[5]We assume that updates to XML data sources are expressed in an XML Update Language as described in Section 2.1.

entire materialized view for every update to the base database. However, heuristically it has been found that an update to the base data usually affects only a small portion of the view. The process of intelligently detecting and updating only the affected portions of a materialized view is called *Incremental View Maintenance*. Incremental view maintenance is often more efficient than view re-computation.

Self Maintainable Materialized Views Views which are maintained using information that is local to the materialized view, i.e. the view definition, the update definition, and the materialized view contents are called *self maintainable* views. Self maintainable views do not access the base data for view maintenance. Thus, self-maintenance of materialized view is a type of incremental maintenance strategy which relies on information local to the materialized view only.

1.3 Summary of ACE-XQ

An overview of the main concepts of ACE-XQ is essential to the understanding of our work. ACE-XQ provides a practical framework for 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, designing an appropriate replacement strategy that improves the cache space utilization, and efficient cached view maintenance upon data updates. The heart of the ACE-XQ system lies in determining containment relationships between nested conjunctive XQueries. To reason about such containment relationships, the ACE-XQ system uses *hierarchical multivalued dependencies (HMVD)* [6] between the elements of an XML document and the *variable binding dependencies*[6] in an XQuery on that document. For an XML document D, the HMVDs represent the dependencies among its elements which have a multiple cardinality relationship with their respective parents. *Variable binding dependencies* in an XQuery arise when one variable is defined in terms of another. The main idea of this XQuery containment approach is to incorporate the checking of the containment of the utilized HMVDs in addition to the checking of the pattern tree homomorphism (i.e., the embedding of the containing query pattern tree into that of the contained query). The main steps of this approach depicted in Figure 1.1 are:

XQuery decomposition. This decompositions separate the variable definition part from the result construction part and represents each using a tree structure. The former tree (i.e., VarTree) captures all the preserved HMVDs. It is different from the navigation pattern tree used in [35], as will be explained later in this thesis. The latter tree (i.e., TagTree) is used to represent the result construction template.

Variable minimization. This step identifies the variables that are neither directly nor indirectly utilized in the result construction and degrades them to navigation steps. This way, it is possible to derive a minimal set of variable binding dependencies on which the containment checking is conducted. This is a critical step for ensuring the correctness of the containment result.

Containment mapping. This involves conducting three types of containment mappings. The first is a minimal VarTree embedding to check the containment of the utilized HMVDs. Second, is to check the tree embedding relationship between the navigation patterns. And lastly, apply a mapping that deals with the effects of block-structure-induced variable dependencies on the containment of XQuery.

XQuery rewriting. If the new query Q_1 is contained within a cached query Q_2, then the mapping $\mathcal{M}_c$ established in the containment mapping phase can be used for rewriting Q_1 against the query result structure of Q_2. The basic idea is to substitute each path expression p in Q_1 for its corresponding path expression p in the TagTree of Q_2 based on $p = \mathcal{M}_c(p) \quad \mathcal{M}_t$, where $\mathcal{M}_t$ represents the mapping of path expressions from the VarTree of Q_2 to its TagTree. Namely, p is computed by the composition of $\mathcal{M}_c$ and $\mathcal{M}_t$.

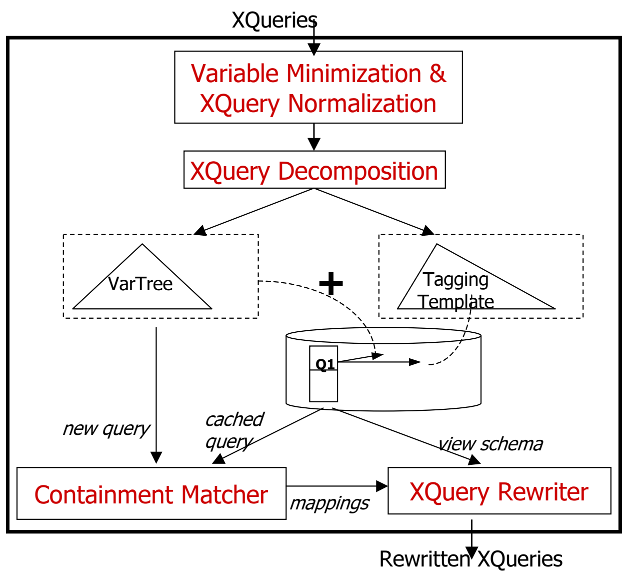


Figure 1.1: ACE-XQ Overview

1.4 Thesis Overview

1.4.1 Problem Definition

An XML view is a derived document defined in terms of an existing XML document. XML views are expressed in a query language like XQuery. When the view query is actually computed and the results are persisted, usually on disk, it is a *materialized view*. To maintain data consistency between the base document and its materialized views, the views must be maintained, i.e. when a change is made to the base document, an equivalent change might also be needed for the materialized view. This process of updating a materialized view whenever a base document changes is called *view maintenance*. The brute-force approach of recomputing the materialized view for every change made to the base document is often wasteful in many cases for the following reasons:

1). If an update to the base document does not affect the materialized view at all, view re-computation is completely unnecessary.

2). If the update operation on the base document affects only a small part of the materialized view, it is usually more efficient to update only the affected part of the materialized view (i.e. incremental updates) instead of re-computing the entire view.

This problem of computing and applying only the incremental updates to materialized views is called *Incremental View Maintenance*. In ACE-XQ query performance over distributed data sources is improved by storing query results locally and re-using these query results to answer future queries, thus avoiding access to the remote data sources. Hence, to maintain these materialized views in ACE-XQ, it is also very desirable that the view maintenance process of ACE-XQ avoid accessing the remote data sources. Such views which can be maintained without accessing the remote data source and using only information local to the materialized views are called *self maintainable views.* This thesis proposes a solution for the requirement of self maintainable materialized views in ACE-XQ.

1.4.2 From XQuery Containment to View Maintenance: Our Approach to View Self Maintenance

For a given XML database D with two queries Q_1 and Q_2, the query containment problem determines if the results of one query are contained within those of the other by analyzing the two queries. In other words, query containment tries to reason about containment relationship between the result data sets of the two queries based on their query definitions. Similarly, given a materialized view MV (defined by view query V_1), and an update query U_1 for a database D, let the state of D and MV before and after executing the update query U_1 on D be represented by D_1, D_2, MV_1, MV_2 respectively. Let the differential data sets of D (diff between D_1 and D_2) and MV (diff between MV_1, MV_2) be represented by D and MV respectively. Then, by analyzing the two queries V_1 and U_1, for any containment relationship between D and MV, we can determine if the update query U_1 is relevant to the materialized view defined by V_1. In the absence of any containment relationship, the update query U_1 is not relevant to materialized view MV. This is our primary motivation for exploring query containment techniques for view self-maintenance.

We propose an algorithmic approach to View Self Maintenance wherein we rely solely on the view and update query definitions, and the materialized view itself. We do not assume access to the base data for performing view maintenance. We decompose the view self maintenance problem into two main sub-problems, viz.

1). Analysis of Update Query for Relevance w.r.t View Query, and

2). Update Query Rewriting

Analysis of Update Query for Relevance w.r.t View Query Henceforth, we will refer to this relevance analysis as simply *Update Query Analysis*. This analysis tries to determine if an update query executed on the base data is relevant to a materialized view. For update query analysis, we use the pattern tree (i.e. VarTree) homomorphism techniques developed for deciding *Query Containment* between two view queries in the ACE-XQ system. Additionally, Update Query Analysis deals with one view query and one update query which may have *insert* and *delete* operations [6]. These update operations require additional reasoning to determine their relevance to a view query. The specifics of the update query analysis, including handling of update operations are provided later in this section.

[6]We consider *replace* operations as *insert* followed by *delete* and hence do not handle it separately.

Update Query Re-writing XML Query languages like XQuery allow output restructuring wherein the structure (i.e. the schema) of the materialized view may be different from that of its corresponding base document. Thus, even after Update Query Analysis determines that an update query is relevant to a view, applying it as-is on the materialized view may lead to incorrect results due to differing schemas of the base document and the materialized view. A relevant update query on the base document must hence be re-written in terms of the materialized view's schema. The ACE-XQ system stores mappings between paths of elements returned in the view query and their corresponding paths in the materialized view. We use this mapping information to do correct re-writings of relevant update queries.

Figure 1.2 depicts the flow of our analysis and rewriting process. Below, we highlight the main steps involved in this process:

1). **XQuery Pre-processing: Minimization and Normalization**

This step is a precursor to the analysis phase to facilitate query analysis. In this step, we minimize the variables in the input queries to obtain an essential minimal set that do not alter query results. The reason is that, before we can utilize containment mapping of variables defined in two queries based on their XPath expressions and dependencies, we need to make sure that the variable dependency tree inclusion is both *sufficient* and *necessary* [4] for determining query containment. Since our analysis builds on the XPath containment based algorithm for view queries in ACE-XQ, the pre-processing of both view and update queries is identical to that required in ACE-XQ.

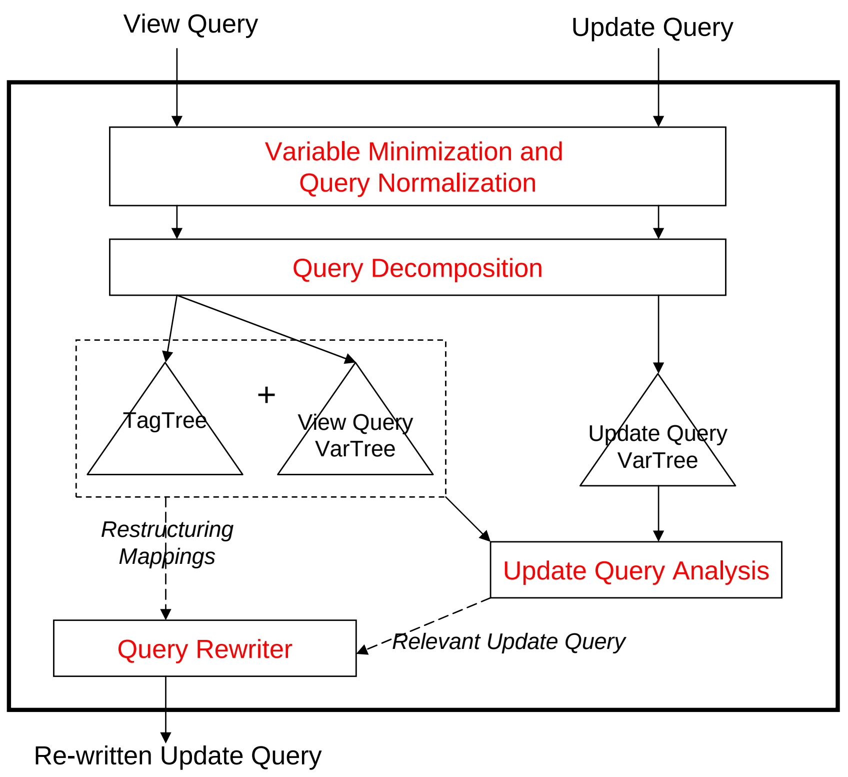


Figure 1.2: Thesis Framework Overview

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 (or FLWU expressions in case of update queries) imposes difficulties for reasoning the containment relationship between two given queries. By sorting out the query constructs and rewriting the input query in accordance to the pre-defined normalization rules, it may facilitate the containment reasoning. Two, the canonical form reveals the pattern matching and result construction semantics of the view query in a more decoupled way, which allows us to easily decompose and address the view maintenance problem as two sub problems, viz Update Query Analysis and Update Query Rewriting.

2). **Query Decomposition: Pattern Matching and Restructuring**

After the normalized form of a given view query is obtained, pattern matching is represented by the variable bindings and their dependencies in the query's *for* clauses. For a view query, 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. For an update query, the update operations correspond to the *update* clauses at each level in the nested FWU structure. Thus, in the normalized query form, the pattern matching and result construction (or update operations) can be clearly separated and captured.

In a view query, the two parts, i.e., pattern matching and result construction, are connected via the essential variables. That is, 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. Their 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**.

A normalized update query, unlike a view query, has no result restructuring functionality. It only contains pattern matching represented by variable bindings and their dependencies in the *for* clauses and the update operations represented in *update* clauses. This simplicity allows us to capture both the variable binding dependencies and update operation semantics of an update query in one single tree structure. This tree structure is similar to the VarTree structure of a view query, except that instead of *return* nodes, it has *update* leaf nodes. The *update* nodes contain details of the update operation being performed. We call this structure an *Update* VarTree, or simply VarTree when there is no chance of confusion with a view query VarTree.

3). **Update Query Analysis**

As mentioned in Section 1.3, determining query containment involves establishing three kinds of mappings, viz., a minimal VarTree embedding to check the containment of the utilized HMVDs, a mapping to check the tree embedding relationship between the navigation patterns (a.k.a. MAC mapping), and a mapping that deals with the effects of block-structure-induced variable dependencies on the containment of XQuery. We use a MAC mapping like VarTree inclusion approach to determine containment relationships between the variable bindings (i.e. their XPath expressions) of a view query and an update query. Similar to the MAC mapping for Query Containment, our MAC mapping for update query analysis also relies on containment of XPath expressions in the two queries. Moreover, the direction of mapping is also from the contained XPath expression to the container XPath expression. There are, however, some subtle differences due the fact that query containment MAC mapping is between two view query VarTrees while update query analysis MAC mapping is between a view query and an update query VarTree. Sections 3.3.3 and 3.3.4 define and compare these two MAC mapping procedures in detail. Unlike view queries, and update query does not have block-structure-induced variable dependencies and the end result of an update query evaluation on a document is independent of the HMVDs in the document. It is for these reasons that the containment mappings based on these concepts does not apply in the Update Query Analysis context.

Once a MAC mapping is established between a view query and an update query, we reason about the update operations in the update query as follows: The location of ***Delete*** operations in our XML Update Language is specified via an XPath expression. Hence, checking for containment between the XPath expression of a delete operation and those of variable bindings in a view query is sufficient to determine its relevance to a materialized view [7]. Section 3.3 discusses this analysis in much more details. For ***insert*** operations, the relevance of the update depends not only on the location of the insert operation,

[7]However, there are cases where the containment of XPath expressions itself is undecidable, in which case determining relevance of delete operation is also undecidable. We explain this in more detail in Chapter 3.

but also on the content being inserted. In addition to checking for XPath containment between view query XPaths and the insert location XPath, we use a procedure based on the following distributive property [8] of view queries:

$$f(d \uplus \quad d) = f(d) \uplus f(\quad d) \tag{1.1}$$

where

- f is a view query,
- d is an XML document,
- $\quad d$ is the new content being inserted into d,
- $f(d)$ is result of evaluating query f on document d, i.e. $f(d)$ creates a materialized view, and
- $\biguplus$ is a special union operation (also called deep union in [22]) on XML Trees which has the same end result as an insert operation.

Looking at the RHS of the above property, intuitively it can be reasoned that evaluating the view query on the new content being inserted and then inserting the result of this evaluation into the materialized view will synchronize it with the update to the base document. We provide the details of achieving this in ACE-XQ in Chapter 3.

4). **Update Query Rewriting**

[8]This property is valid only for unordered XML documents.

A relevant update query may need re-writing in order to be applied to the corresponding materialized view. This rewriting is essentially re-writing of XPath expressions in the update query. Such a path expression rewriting utilizes the *tagging template* of the view query, which reveals how the result structure is constructed based on variable bindings in the view query. We explain the tagging template and its uses in query re-writing in detail in Chapter 3.

1.5 Contributions

The primary contributions of this thesis are:

- We have proposed a unique approach to the self maintenance of materialized XML views using a query containment approach. To the best of our knowledge, we are the first to investigate and use query containment for XQuery view maintenance.

- We have proposed a view maintenance approach that relies solely on the view and update query definitions, and data in the materialized view itself. Other view maintenance strategies either require knowledge of schema of the base document, assume the presence of unique object identifiers [1, 37], or assume access to the base data [9]. While the amount of information available during view maintenance has an impact on the cases that can be incrementally maintained, the simplicity of our approach does not restrict its usability. In practice, a caching system can easily get around some of these restrictions by having two layers of views where-in one layer consists of simpler, self-maintainable materialized views and the second layer uses the first layer to define complex non-materialized views.

- We believe that the concept of pattern matching used in our approach can be effectively used for more complicated update and view queries. This pattern matching can help detect non-relevant queries sooner in the query analysis process and reduce the number of update queries for further analysis. This filtering of non-relevant update queries based entirely on pattern matching of queries can have very significant benefits for the average case performance of a caching system like ACE-XQ.

Chapter 2

Preliminaries

Preliminaries

In this chapter we explain several background concepts necessary for understanding the details of our solution to the view self maintenance problem. We also introduce an example which will be re-used throughout this thesis in examples and explanations.

2.1 XML Query and Update Languages

XPath [30], a regular expression based path definition language is the basis for several XML query languages. The earliest XML Query languages like XQL, XML-QL, XSL added features like iteration over simple XPath expressions and XSL allowed nested evaluations too. XQuery, the current W3C standard for XML Queries (and its predecessor Quilt) have a FOR-LET-WHERE-RETURN (FLWR) based syntax which allows binding variables to XPath expressions in the FOR and LET clauses, specifying complex predicates using the WHERE clause, and output restructuring within the RETURN clause. The RETURN clause in turn can contain nested queries, and also supports sorting, aggregation.

There is no standard yet for an XML Update Language. We are aware of three efforts to define an update language for XML documents.

XUpdate is a pure descriptive XML update language which is designed with references to the definition of XSL Transformations (XSLT). Similar to XSLT, the update query itself is an XML document and uses the expression language defined by XPath. Those XPath expressions are used in XUpdate for selecting nodes for processing afterwards. Features of XUpdate include:

- Elements like *xupdate:insert-before/after*, *xupdate:update*, *xupdate:remove* for XML document modification
- *constructor* elements like *xupdate:element*, *xupdate:attribute* etc. for creating new elements, attributes, processing instructions, or comments to be used in update instructions.
- *xupdate:variable* and *xupdate:value-of* elements which allow binding a variable name to a value which can be used later in updates.
- *xupdate:if* element which allows conditional processing.

A simple XUpdate query which deletes the first *closed_auction* element in Figure 2.3 would be as follows:

```
<xupdate:remove select = \
"document("auction.xml")/site/closed_auctions/closed_auction[1]"/>
```

Microsoft's SQL Server 2005 database treats XML documents and content as a new data type for attributes/columns of a relation. For updating the content of XML columns, Microsoft has extended standard SQL to include a *modify* clause which can specify an XML Update Query. An example SQL update query on a table having an XML column would look like:

```
UPDATE <TABLENAME> SET
<XML_COLUMN_NAME>.modify('<XML_UPDATE_QUERY>') WHERE
<UPDATE_CONDITION>
```

The XML Update Query language used in the *modify* clause allows a single *insert*, *delete* or *update* operation. The location of this update is specified via an XPath expression. Subtrees can be inserted before or after a specified node, or as the leftmost or rightmost child. Furthermore, a subtree can be inserted into a parent node, in which case it becomes the rightmost child of the parent. Attribute, element, and text node insertions are supported. Deletion of subtrees is supported. In this case, the entire subtree is removed from the XML instance. Scalar values can be replaced with new scalar values.

I. Tatarinov and A. Halevy [28] have defined an XML Update Language which is very similar in syntax to XQuery. The general form of an update query as defined in [28] is shown in Figure 2.1

From figure 2.1 we can see that:

- This update query language allows binding of elements to variables via the *for-let* clauses similar to XQuery and also allows filtering in the *where* clause.

```
FOR $binding1 IN Xpath-expr, …
LET $binding := Xpath-expr, …
WHERE predicate1, …
updateOp, …

where updateOp is defined in EBNF as:
          UPDATE $binding1 {subOp {, subOp}* }
and subOp is:
          DELETE $child |
          RENAME $child TO name |
          INSERT content [BEFORE | AFTER $child] |
          REPLACE $child WITH content |
          FOR $binding2 IN Xpath-subexpr, …
          WHERE predicate1, … updateOp
```

Figure 2.1: Syntax Of XML Update Queries

- Updates can be deletes, inserts, renaming or replacement of XML elements, attributes, comments etc.

  1). The variable binding of an **update** clause is the common ancestor of XML elements being updated. In other words, all updates always occur at child elements of the update clause's variable binding.

  2). Child elements to be *Delete*d are specified via an XPath expression.

  3). The content to be *Insert*ed can either be explicit XML content or can be specified via variable bindings. Content can optionally be inserted **before** or **after** a specified child variable binding. In cases where the order of insert is not specified, the default behavior is to *appended* new content to the existing content.

4). *Replace*s and *Rename*s also occur on child elements of the update clause's variable binding.

- The *update* clause itself can further contain nested *for-let-where-update* queries, similar to the nested structure of XQuery. The variable bindings in the nested *for-let* clauses must be sub-expressions of (i.e. derived from) those of the enclosing *for-let* block.

This update language is designed to be a language in which queries are concise and easily understood. It is also flexible enough to update a broad spectrum of XML information sources, including both databases and documents. Due to its similarity with XQuery, it is easy to understand, learn, and is more likely to be adopted by users of XML. Also, the SafeXUpdate[18] project at WPI implemented this update query language, and this system was freely available to us. For all these reasons, we have chosen the XML Update Language of [28] for this thesis.

2.2 AceXQ

Section 1.3 provides a summary of the AceXQ system. Since our work is developed in the context of AceXQ, the interested reader should read [4] for a deeper and clearer understanding of XML Query Containment.

2.3 XPath Containment

The XQuery containment approach of [4] is based on containment of XPath expressions. The basic constructs of any XPath expression are: child, descendant, filter, wildcard, disjunction, and variables [/, //, [], *, , $]. It is conventional to represent the fragment we use by listing the allowed operators. For example, XP(/, //) denotes an XPath fragment where only child and descendant operators are allowed. Since the result of an XPath query is a set of nodes that satisfies the specified pattern, the containment of two XPath expressions E1 and E2, denoted $E1 \quad E2$ means that the result node set of E1 is a subset of the result node set of E2 assuming the two expressions are evaluated on the same XML document. For example,

$/site/closedAuctions/closedAuction \quad //closedAuction$, and

$/site/(africa \; asia)/item \quad /site/ \quad /item$, regardless of the presence of a DTD.

The complexity of determining XPath expression containment depends on the structure of the XPath expressions involved. The XPath containment algorithm has attracted significant attention in the recent past. Miklau and Suciu obtained that containment of XP(//, [], *) is CoNP-Complete. In their research, they also cite previous works which shows that the problem can be solved in PTIME if we consider any two of the three operators. More recently, [14] extended this XPath fragment to include disjunction, variables, and the presence of DTDs and obtained a more complete classification of the containment problem w.r.t to these fragments. Tables 2.1 and 2.2 summarize important research results about XPath containment with and without DTD constraints.

In this thesis we use the XPath fragment XP(/, //, [], *) i.e. XPath expressions containing child, descendant, filter, and wildcard operators. Moreover, to determine containment in PTIME, we restrict the XPath ex-

DTD	/	//	[]		*	Complexity
+	+	+			+	in P
+	+		+			CoNP-complete
+		+	+			CoNP-hard
+	+	+	+	+	+	EXPTIME-complete
+	+	+		+		EXPTIME-complete
+	+	+	+		+	EXPTIME-complete
+	+	+	+		+	undecidable with nodeset comparisons

Table 2.1: Containment Complexity for Different XPath Fragments With DTD

/	//	[ ]	*		Complexity	Reference
+	+	+			PTIME	Amer-Yahia et al, 2001
+		+	+		PTIME	Wood, 2001
+	+		+		PTIME	Neven & Schwentick, 2003
+	+	+	+		coNP-complete	Miklau & Suciu, 2002
+				+	coNP-complete	Miklau & Suciu, 2002
	+			+	coNP-complete	Miklau & Suciu, 2002
+	+	+	+	+	coNP-complete	Neven & Schwentick, 2003

Table 2.2: Containment Complexity for Different XPath Fragments Without DTD

pressions in queries to contain any two of XP(//, [], *) operators, in addition to the child ('/') operator. This is a fairly sound subset of XPath as most XPath expressions in real life exclusively use these operators and seldom use other operators like disjunction etc. Our algorithms will work efficiently for any XPath fragment which is amenable to containment in PTIME. Also, in general containment of XPath expressions involving node-set comparisons is undecidable. To keep containment decidable, we restrict

XPath expressions involving predicate expressions as follows:

- Predicate expressions involve comparison of a nodeset with string or numeric literals.
- The predicate expression may be a simple expression involving only one comparison operation or a compound expression composed from simple expressions. Compound predicate expressions must be expressed in CNF (Conjunctive Normal Form).
- Predicate expressions where the expression is an arbitrary XPath expression which can contain any two of XP(//, [], *) operators.
- We require that the Xpaths in predicate expression not contain any nodes returned by the XQuery. This restriction makes query analysis and re-writing easier.

2.4 An Example

Consider the XML document in Figure 2.3 which conforms to the XMark[27] benchmark schema shown graphically in Figure 2.2.

The XMark schema contains information about auctions. Suppose a user is only interested in buyer and price information of sold items costing more than $1000. Figure 2.4 shows an XQuery to extract this information from a database conforming to the XMark [27] benchmark.

The result evaluating this query on the XML document in Figure 2.3 is shown in Figure 2.5

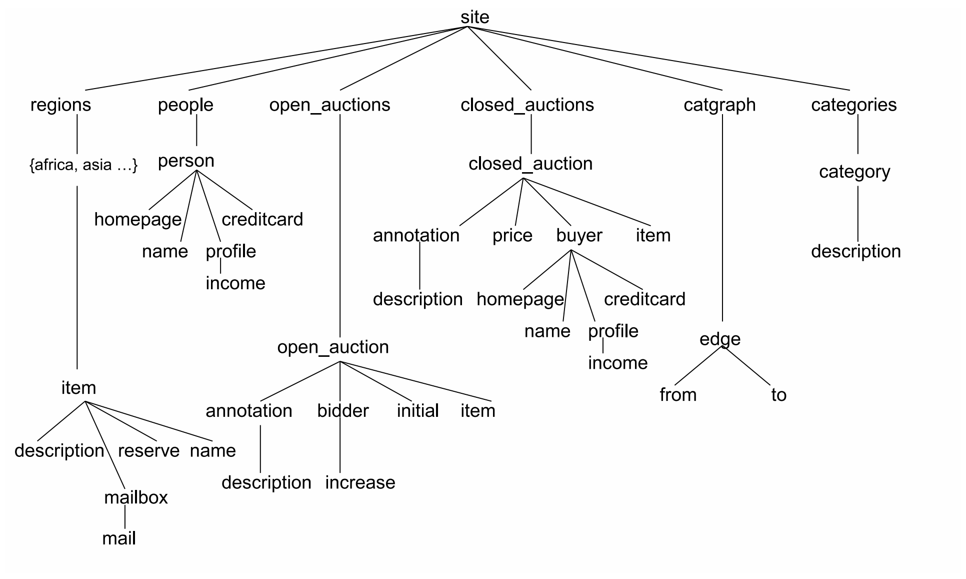


Figure 2.2: The XMark Schema

ACE-XQ stores the results of this query locally as a materialized view to provide faster answers to similar future queries. As the remote database is updated with new information, this materialized view may need updating. Figures 2.7 and 2.6 show two such update queries with operate on the sample document of Figure 2.3. Since the materialized view of Figure 2.5 stores only a portion of the entire document, not all updates to the base document are relevant to the view. In example 2.6, the update query deletes the home-page information of a person and does not affect the view in any way. Such an update query will be detected as non-relevant by our view maintenance approach.

However, the query in Figure 2.7 which deletes a closed auction element is relevant to the materialized view of Figure 2.5. We will use these and

```
<?xml version="1.0" standalone="yes"?>
<site>
  <closed_auctions>
    <closed_auction>
      <seller>
         <name>Sinisa Farrel</name>
         <emailaddress>mailto:Farrel@duke.edu</emailaddress>
      </seller>
      <buyer>
        <name>Lee Tzitzikas</name>
        <emailaddress>mailto:Tzitzikas@whizbang.com</emailaddress>
        <homepage>http://www.whizbang.com/~Tzitzikas</homepage>
        <creditcard>6491 3985 6149 1938</creditcard>
      </buyer>
      <item>
        <name>The Girl From Malabar</name>
        <description>Oil painting by Raja Ravi Varma depicting a beautiful native Malabar girl</description>
      </item>
      <price>2830.20</price>
      <date>04/16/2005</date>
      <quantity>1</quantity>
      <annotation>
        <author name="Hiroko Schhwartz"/>
        <description><text> Good buy! </text></description>
      </annotation>
    </closed_auction>

    <closed_auction>
      <seller><name>Roman Keustermans</name></seller>
      <buyer><name>Shavinder Giger</name></buyer>
      <item>
          <name>Reverse The Curse</name>
          <description>Collage of Babe Ruth's contract and 2004 Red Sox World Series celebration photo</description>
      </item>
      <price>456.90</price>
      <date>04/16/2005</date>
      <quantity>1</quantity>
    </closed_auction>
  </closed_auctions>
</site>
```

Figure 2.3: An example XML document

similar examples later in this report to explain our approach to view self maintenance.

```
<result>
FOR $item IN doc("auctions.xml")/site/close_auctions/closed_auction
WHERE $item/price > 1000
RETURN
            <entry>
            {
                        $item/buyer, $item/price, $item/happiness
            }
            </entry>
</result>
```

Figure 2.4: View Query

```
<result>
   <entry>
      <buyer>
        <name>Lee Tzitzikas</name>
        <emailaddress>mailto:Tzitzikas@whizbang.com</emailaddress>
        <homepage>http://www.whizbang.com/~Tzitzikas</homepage>
        <creditcard>6491 3985 6149 1938</creditcard>
       </buyer>
       <price>2830.20</price>
  </entry>
</result>
```

Figure 2.5: Result of evaluating query in Figure 2.4 on XML document

```
FOR $p IN doc("auctions.xml")/site/people
WHERE $p/person[@id=97]
UPDATE $p
{
            DELETE $p/homepage
}
```

Figure 2.6: Non Relevant Update Query

```
FOR $item IN doc(“auctions.xml”)/site/closed_auctions
WHERE $item/closed_auction/price = 1321
UPDATE $item
{
            DELETE $item/closed_auction
}
```

```
FOR $item IN doc(“auctions.xml”)/site/closed_auctions
UPDATE $item
{
            INSERT
                        <closed_auction>
                                    <buyer>
                                                <name>JP</name>
                                    </buyer>
                                    <item>Water Lillies</item>
                                    <price>2345</price>
                                    <date>Jan 12, 2004</date>
                        </closed_auction>
}
```

Figure 2.7: Relevant Update Queries

Chapter 3

Our Approach To View Self Maintenance

Our Approach to View Self Maintenance

3.1 Overview

As mentioned previously in Chapter 1, we decompose the view self maintenance problem into two sub-problems:

- **Update Query Analysis:** In this phase we determine if an update query on a base XML document is relevant to an XML view defined on it. To facilitate analysis, the query is first pre-processed as explained in Section 3.2.

- **Update Query Rewriting:** If the Query Analysis procedure determines that an update query is relevant to a view, we rewrite the update query on the base document into an update query on the materialized view. This re-written query when executed on a materialized view synchronizes it with the base document.

The remainder of this chapter explains the details of our approach along with examples. We first start with an explanation of query pre-processing in section 3.2, then explain the details of update query analysis and update query re-writing in sections 3.3 and 3.4. Section 3.5 discusses the complexity analysis of our update query analysis procedure, while section 3.6 concludes this chapter explaining our approach with several examples.

3.1.1 A Query Subset Suitable for View Self Maintenance

We first define the subset of view and update query languages for which our view self maintenance approach works. Since our view self maintenance approach builds on the XPath containment based query containment of ACE-XQ, we restrict ourselves to view and update queries containing XPath expressions for which containment is decidable in PTIME [1]. We also require an unordered XML model and that queries not contain any relative or absolute order related constructs.

Our approach works for view queries which allow nested blocks, conjunctive equality based conditions, set and bag semantics and which

- Do not use negation, disjunction, aggregation, universal quantifiers, tag variables, and
- Do not use pre-defined or user-defined functions.

and for update queries which

- Contain only *delete* and *insert* operations, and
- Have insert content specified explicitly in the query, and not via variable bindings.

[1] See Section 2.3 for additional details on XPath containment and the XPath subset we use.

3.2 Query Pre-processing

The flexibility in composing an XQuery using nested FLWR expressions (or FLWU in case of update queries) imposes difficulties for reasoning the containment relationship between two given queries. We hence pre-process both the view and update queries and translate them into a canonical form [4] with certain characteristics which can facilitate our analysis. Our analysis algorithm builds on the query containment algorithm presented in [4], and hence the pre-processing of queries is similar to that used in ACE-XQ. To summarize, there are three pre-processing steps, viz.

1). Variable Minimization

2). Query Normalization

3). Query Decomposition

All these pre-processing steps are required for both view and update queries. Since the view and update queries are syntactically similar to each other, the preprocessing steps for an update query resemble those of a view query, but are not identical. We discuss the similarities and differences in pre-processing of view and update queries in the following sections.

3.2.1 Variable Minimization

In this step, we delete non-essential variables from the query definition. Non essential variables are those which can be safely deleted without causing a change in the query result. For the query in Figure 3.1, the variables **$buyer** and **$price** are non-essential as their usages in the *return* clause can be replaced by variable bindings derived from **$item** (as **$buyer = $item/buyer** and **$price = $item/price**). Since both view and update queries use a *For-Let-Where* syntax for specifying variable bindings, this variable minimization procedure is identical for both types of queries. For more details on the algorithm, soundness and completeness of variable minimization see [4].

```
<result>
FOR $item IN doc(“auctions.xml”)/site/closed_auctions/closed_auction
WHERE $item/price > 1000
RETURN
<entry>
{
        FOR $buyer IN $item/buyer, $price IN $item/price
        RETURN $buyer, $price
}
</entry>
</result>
```

Figure 3.1: XQuery Before Normalization

3.2.2 Query Normalization

View Query Normalization

Note: The details view query normalization in this subsection have been adapted from [4].

Our goal is that the normalized query can facilitate the separation of the path expressions that are to be output in the result from those that are used for specifying variable bindings, such that the later query decomposition step is made easy. There are a number of XQuery normalization techniques

[33, 24, 8] available. They overlap in some commonly used normalization rules. For example, unnesting the FLWR expression within a *for* clause (as illustrated before) is a standard rule shared by many techniques.

We adopt a set of query normalization rules including rules (R2)   (R5), (R7)   (R10), and (RG1) from [8]. We also apply rules (R1), (R6), (R11), and (R12), but in their reverse directions. Rule (R13) does not apply in our context since we exclude disjunctions from our XQuery fragment. Since we consider the XQuery fragment with no aggregations, we can also apply the rule that substitutes each *let-variable* with its definition. After applying these rules, the query is free of *let* clauses, empty sequence expressions and unit expressions. Also, only *return* clauses may contain nested FWR[2] expressions.

```
<result>
FOR $item IN doc("auctions.xml")/site/closed_auctions/closed_auction[price>1000]
RETURN
<entry>
{
        $item/buyer, $item/price
}
</entry>
</result>
```

Figure 3.2: XQuery After Normalization

If we apply the normalization rules to an example query shown in Figure 3.1, the normalized form is shown in Figure 3.2. Notice that in Figure 3.2, the non-essential variables are deleted as well as the predicate from the WHERE clause is pushed up into the XPath variable binding of the FOR clause.

[2]Letter *L* for representing *let* is removed since the normalized query is let-clause free.

Update Query Normalization

Since update queries have a *for-let-where-update (FLWU)* syntax very similar to XQuery's FLWR syntax, the normalization rules are also very similar, in fact even simpler. Using normalization rules very similar to those used for the *For*, *Let*, and *Where* clauses in a view query, we 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) only the update-clauses may contain nested FWU expressions; (4) *Local-Where Rule* wherein the predicates in a *Where* clause are pushed up as filters in the XPath expression of the corresponding variable binding in the *For* clause.

If we apply the normalization rules to an example query shown in 3.3, the normalized form is shown in Figure 3.4.

```
FOR $p IN doc("auctions.xml")/site/people
WHERE $p/person[@id=97]
UPDATE $p
{
          DELETE $p/homepage
}
```

Figure 3.3: Update Query Before Normalization

```
FOR $p IN doc("auctions.xml")/site/people/person[@id=97]
UPDATE $p
{
          DELETE $p/homepage
}
```

Figure 3.4: Update Query After Normalization

3.2.3 Query Decomposition

View Query Decomposition

Definition 3.1. *Given a normalized XQuery Q, a tree structure named* ***VarTree=(V, E, L)*** *can be constructed based on the extracted variable binding dependencies. Each defined variable is denoted by a* ***var node*** $v \quad V$*. Each dependency* $v_i \overset{p_j}{\rhd} v_j$ *corresponds to an edge* $e = (v_i, v_j) \quad E$ *labeled* $p_j \quad L$*. We refer to* e *as the* ***derivation edge*** *of* v_i*.*

The VarTree is different from the pattern tree concept referred to in other research [35]. An edge in the pattern tree corresponds to an axis step (/ or //) and the associated element type test. In contrast, a derivation edge in VarTree denotes the navigation pattern used for deriving a child variable from its parent. Actually this is indicated by the label on a derivation edge which is an XPath expression composed of possibly multiple steps and branches. In this sense, the VarTree can be considered as a nested tree with each edge encapsulating the navigation pattern corresponding to the label on it. Figure 3.5 shows the VarTree representation of query in Figure 3.2.

Definition 3.2. *For a normalized XQuery Q, a tree structure conforming to its nested block structure can be constructed to represent the result construction semantics. It is called* ***TagTree=(N,A)****. Each* ***block node*** $n \quad N$ *is a quadruple* $[\bar{V}, \bar{C}, \bar{R}, \bar{T}]$ *and each edge* $a = (n_i, n_j) \quad A$ *denotes that block* n_j *is nested within block* n_i*. Furthermore,*

- $\bar{V}$*,* $\bar{C}$*,* $\bar{R}$*, and* $\bar{T}$ *respectively represent the variables, where-conditions, re-*

turn expressions, and to-be-constructed new elements specified in the corresponding block;

- $\bar{C}$ *is denoted by a forest of constraint pattern trees each rooted at a variable defined in the local or an ancestor block. Equality conditions are associated with the corresponding node(s);*

- *If unnesting of the bindings of variables in* $\bar{V}$ *results in a non-empty set and conditions* $\bar{C}$ *are satisfied, then the construction of a new element denoted by* $\bar{T}$ *will be invoked for each tuple in that unnested binding set;*

- $\bar{T}$ *may have either none, one, or a sequence of tag names in the form* $t1$ $t2$ $\ldots$ tn *. This means that the returns of* $\bar{R}$ *will be enclosed by an empty tag,* $t1$ *and* $/t1$ *, or* $t1$ $t2$ $\ldots$ tn *and* $/tn$ $\ldots$ $/t2$ $/t1$ *.*

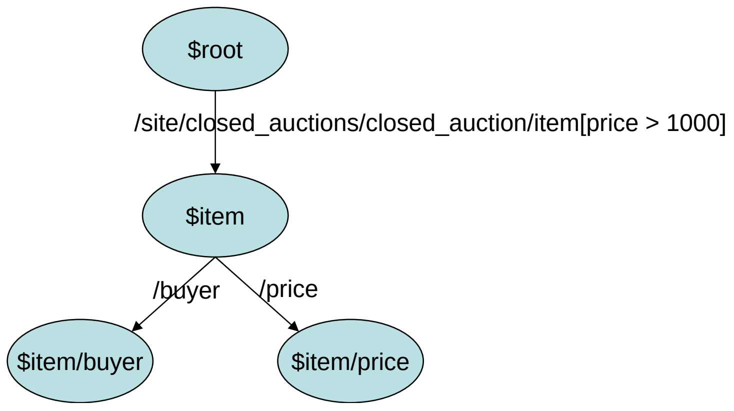


Figure 3.5: VarTree Representation of Query in Figure 3.2

In this thesis, we consider a simplified version of TagTree, wherein each block node in the TagTree is a tuple, $[\bar{V}, \bar{T}]$ instead of the quadruple

$[\bar{V}, \bar{C}, \bar{R}, \bar{T}]$. In this simplified version, instead of having $\bar{R}$ (i.e. the *return* nodes) information in the TagTree block nodes, we show them as leaf nodes in the TagTree itself. The edges represent nesting of elements to be constructed and are labeled with the XPaths for deriving the child nodes variable binding from it's parent node. This simplified version of the TagTree closely resembles a VarTree in structure but has additional information in its block nodes. It also allows a simpler algorithm for finding mapping paths of XML elements, which is used during update query rewriting. The TagTree structure for the view query in Figure 3.2 is shown in Figure 3.6.

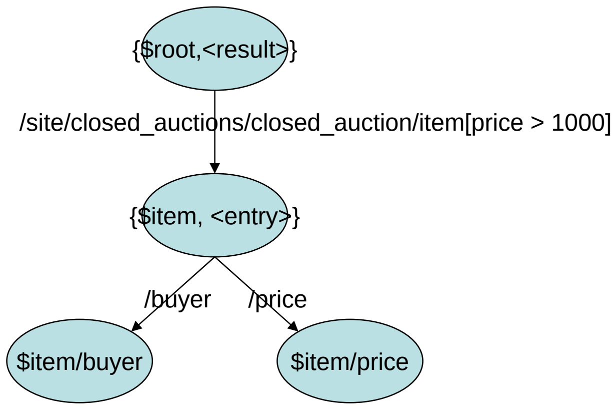


Figure 3.6: TagTree Representation of Query in Figure 3.2

Update Query Decomposition

Definition 3.3. *Given a normalized XML Update Query Q, a tree structure named* ***VarTree=(V, E, L, U)*** *can be constructed based on the extracted variable binding dependencies and update operations. Each defined variable is denoted by a* ***var node*** *$v \quad V$. Each dependency $v_i \overset{p_j}{\triangleright} v_j$ corresponds to an edge $e = (v_i, v_j) \quad E$*

*labeled p_j L. We refer to e the **derivation edge** of v_i. Each update operation (insert, or delete) attached to an **update** clause in the update query is denoted by a child node u U of the* var *node corresponding to the update clause. The derivation edge of a **delete** node is the XPath expression of its **delete** clause in the query. The incident edge of an **insert** node has no variable dependency associated with it, but the update node itself contains the content to be inserted.*

Figure 3.7 shows the VarTree representation of update query in Figure 3.4. An update query can be completely represented using a single tree structure, also called a **VarTree** [3]. Like a view query's VarTree, an update query VarTree's internal nodes also represent the variable bindings and the edges between variable nodes are labeled with the XPath expressions used to derive the child variable node from the parent. In addition to these variable nodes, an update query's VarTree also has *update* nodes which represent the actual update operations. From the update query syntax in Figure 2.1, it is clear that updates are attached to the variable binding of an **update** clause, or to its child nodes. The variable binding of an **update** clause is represented by a variable node in the VarTree. Thus, updates attached to such a variable node will be its child nodes in the VarTree representation. It is also easy to see that the *update* nodes in a VarTree are *leaf* nodes of the VarTree.

Update nodes in the VarTree can represent both delete and insert operations. The **delete** clause of an update query also specifies the XPath expression (derived from the update clause's variable binding) of the ele-

[3]We will explicitly qualify a VarTree as an Update Query VarTree or a View Query VarTree if there is any chance of confusion.

ments to delete. Hence the derivation edge from the parent node to a delete node is labeled with this derived XPath expression. The edge between an insert operation and its parent is not labeled as there is no variable dependency involved. However, the insert node itself contains the content to be inserted.

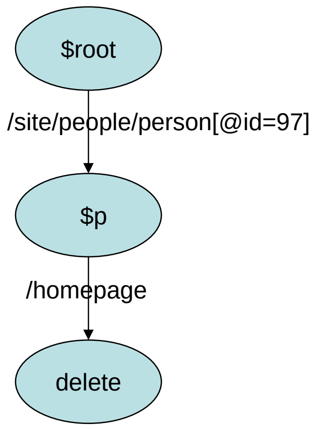


Figure 3.7: VarTree Representation of Update Query in Figure 3.4

3.3 Update Query Analysis

Given *VarTree* representations of a view query and an update query, it is possible to determine if the update query is relevant to the view. We achieve this by introducing containment based mappings between nodes of the two VarTrees. Recall that the path to each node in a *VarTree* also defines the XPath expression of the variable binding corresponding to that node. We use the containment relationship between these XPath expressions of view and update queries to establish mappings between nodes of the two

*VarTree*s. Section 3.3.5 describes an efficient algorithm to achieve this mapping between the nodes of a view query and an update query. To begin with, sections 3.3.1 and 3.3.2 introduces the usage of XPath containment for view self maintenance.

3.3.1 Using XPath Containment for View Self Maintenance

Consider the following observations:

- Each *delete* operation in an update query is essentially specified via an XPath expression for the XML elements to be deleted. If this delete operation is relevant to a view query then there must exist a containment relationship between the XPath specifying this delete operation and some XPath expression in the view query. In the absence of any containment relationship, the delete operation is not relevant to the materialized view.

- Each *insert* operation in an update query involves adding content at a specific location in a document. The location of this insert is also specified by an XPath expression. If this insert is to be relevant to a view query, then there must exist a containment relationship between the XPath specifying the insert location and some XPath expression in the view query. In the absence of any containment relationship, the insert operation is also not relevant to the materialized view. The relevance of an insert operation is also additionally dependent on the content being inserted. In Section 3.3.2 we explain how to reason about this insert content when deciding relevance of update operations.

From the above observations, we can elaborate further about XPath containment between a view query and an update query as follows:

1). For any two variable nodes u and v where u belongs to an update query VarTree and v belongs to a view query VarTree, and a containment relationship exists between XPath(u) and XPath(v), then Update Query Analysis is possible only if $XPath(v) \quad XPath(u)$. In case of the contrary containment relationship wherein $XPath(u) \quad XPath(v)$, Update Query Analysis is not determinable. To understand the reason for this non determinability, consider the contrary case where the update query XPath is contained in the view query variable node's XPath (i.e. $XPath(u) \quad XPath(v)$). In this case, the update query would have to update a subset of elements in the materialized view. This exact subset of elements from the materialized view can only be determined by their parent elements which are available in the base document and not in the materialized view. Our view maintenance approach does not assume access to the base document and hence the contrary case (i.e. $XPath(u) \quad XPath(v)$) becomes non determinable for view self maintenance.

2). If the XPath expression for a *delete* node subsumes, is equivalent to or is contained in the XPath expression of some *return* node in the view query, then the delete operation is relevant to the view. The reason for this is obvious.

3). If the XPath expression of a *delete* node subsumes or is equivalent to the XPath expression of a variable node, then the delete operation is relevant to the view.

4). If the XPath expression of the delete node is completely contained in (but not equivalent to) some non-return (i.e. variable) XPath expression from the view query, then determining relevance of this delete operation to the view query is not determinable.

Similar to delete operations, the following observations hold for insert operations

1). If the XPath expression of an *insert* node is contained in the XPath expression of some *return* node in the view query, then the insert operation is relevant to the view.

2). If the XPath expression of an *insert* node subsumes or is equivalent to any XPath expression in the View Query, then the insert operation may be relevant to the view. We need further analysis of the content being inserted to determine relevance. This additional processing is described in section 3.3.2

3). If the XPath expression of the insert node is completely contained in (but not equivalent to) some non-return (i.e. variable) XPath expression from the view query then determining relevance of this insert operation is not determinable.

The above observations may be summarized as follows:

Given the essential VarTree representations VT_1 and VT_2 of Update Query and View Query respectively, then

1). VT_1 is relevant to VT_2 and view self maintenance is possible if and only if

   a. XPath(v)   XPath(u) holds for some u, v AND

   b. XPath(u)   XPath(v) does not hold for any u, v

   where u and v are *variable* nodes belonging to VT_1 and VT_2 respectively.

2). For a delete node d in VT_1, d is relevant to the view query only if

   - XPath(d)   XPath(r) OR XPath(r)   XPath(d), or
   - XPath (v)   XPath(d)

   where r is a *return* node in VT_2 and v is any *variable* node in VT_2

3). For an insert node i in VT_1,

   - if XPath(i)   XPath(r) where r is any $return$ node in VT_2, then the insert operation is relevant to the view query.
   - if XPath(v)   XPath(i) where v is any $variable$ node in VT_2, then the insert operation may be relevant to the view query.

3.3.2 Handling Insert Operations

The relevance of an insert operation depends not only on the location of insert, but also on the content being inserted. We solve this problem as follows:

Assume that the base XML document has already been updated by this insert operation [4]. If this insert content were relevant to the materialized view, then the result of evaluating the view query on the updated base XML document will contain some or all of the content being inserted by the insert query. If it does not, then the insert operation is not relevant to the view. The above approach will always be able to determine if an insert operation is relevant or not. However, it requires accessing the remotely located base XML document and recomputing the entire view query to determine relevance of insert operations, which defeats the purpose of view self maintenance. To overcome this problem, we use the distributive property [5] of view queries mentioned in Equation 1.1 and repeated here for easy reference:

$f(d \lceil \quad d) = f(d) \lceil f(\quad d)$

where

- f is a view query,
- d is an XML document,
- $\quad d$ is the new content being inserted into d,
- $f(d)$ is result of evaluating query f on document d, i.e. $f(d)$ creates a materialized view, and
- $\lceil$ is a special union operation (also called deep union in [22]) on XML Trees which has the same end result as an insert operation.

[4]Note that our approach only handles insert operations where the content is explicitly specified via insert content, and not via variable bindings.

[5]This property is valid only for unordered XML documents

Thus, for an insert of new content d into the base document, a materialized view defined on it can be maintained as follows:

1). Evaluate the view query on the new content d.

2). If this evaluation does not return any result, the insert is not relevant to the materialized view.

3). Else, insert the result of view query evaluation into the the materialized view.

The XPath expressions in a view query conform to the document structure of the base document and hence applying it as-is on the new content being inserted will yield incorrect results in most cases. We hence use the document structure information available in XPath expressions of the *view* query, and the insert content available in the update node to *generate* a new temporary XML document on which the view query can execute. For example, the XML document for XPath expression $a//b/c$ will have element $'a'$ as the root node, with element $'b'$ as its child node, and element $'c'$ as its grand child node. The location of the insert content in this *generated* document is determined by the XPath of the view query's VarTree node which maps to the insert node's parent node [6]. This *generated* XML document has structural *resemblance* to the base XML document. The view query is then evaluated on this newly generated document. If this query evaluation does not return any results, the insert operation is not relevant to the materialized view. This outcome is the same as view re-computation not resulting

[6]Mapping of View Query and Update Query VarTree nodes is explained in Section 3.3.1

in any change in the materialized view. If however, the query evaluation returns some results, the update query is relevant to the view query and the query evaluation results can be used to update the materialized view.

The size of this *generated* document is likely to be much smaller than that of the base XML document. This smaller sized document, along with the fact that this query execution happens locally within the ACE-XQ cache does not add any significant overhead to the query analysis procedure as we show in our experiments later.

3.3.3 VarTree Node Mappings for Query Containment

Note: The contents of this subsection have been adapted from [4].

We extend the traditional tree homomorphism (namely based on root, label, and ancestor-descendant relationship preserving) to define the MAC mapping.

Definition 3.4. *Suppose VT_1 and VT_2 are the minimal VarTrees of Q_1 and Q_2 respectively. For determining $Q_1 \quad Q_2$, there must be a **MAC mapping** from VT_1 to VT_2, denoted by $\Phi(VT_1) = VT_2$, such that the following conditions are satisfied:*

1). *$C1 : roots(VT_1) \quad roots(VT_2)$,*

2). *$C2$: for any node $u \quad VT_1$, there is a match $\Phi(u) \quad VT_2$ such that $\mathcal{T}(u)= \mathcal{T}(\Phi(u))$ if $\Phi(u)$ is a var node, and $\mathcal{T}(u)<:\mathcal{T}(\Phi(u))$ if $\Phi(u)$ is a ret node ($\mathcal{T}$ returns the type of the element, and $<:$ denotes the subtype-supertype relationship),* 3). *$C3$: u is an ancestor of v for all u, v VT_1 if and only if $\Phi(u)$ is an ancestor of $\Phi(v)$ in VT_2, and*

4). *$C4$: if u is a var node in VT_1, then $\Phi(u)$ is either a var or a ret node; if u is a ret node, then $\Phi(u)$ must be a ret node.*

Below we explain each of these required conditions.

C1: Root inclusion[7]. This condition requires that each source XML document referred to in Q_1 must also be referred to in Q_2. Correspondingly in the VarTrees, $roots(VT_1)$ returns the URLs of the source XML documents involved in Q_1, which should be a subset of those returned by $roots(VT_2)$.

C2: Mapping of element types. This condition requires a total but not necessarily injective mapping from nodes in VT_1 to those of VT_2. In addition, a node u in VT_1 must be mapped to a node in VT_2 that has either the same type or a supertype[8] of u's depending on if the matched node is a var node or a ret node. The element type of a node can be inferred from the XPath expression on its incoming derivation edge. u can be mapped to a super-type ret node $\Phi(u)$ because the associated bindings of $\Phi(u)$ are all deeply returned (due to the semantics of a return expression) to enable the retrieval of u's bindings from subtrees of $\Phi(u)$'s bindings in Q_2's result.

C3: Preservation of ancestorships. In a minimal VarTree, nodes represent essential variables and the HMVDs among them are captured by their ancestor-descendant relationships. Therefore, if all the ancestor-descendant

[7]Our technique allows a query to involve more than one XML document. In this case, the corresponding VarTree is actually a forest of trees, which may be connected by equality conditions on variables across trees.

[8]Here the concept of subtype-supertype is not the same as those in the object-oriented modeling domain. Instead, it corresponds to the element inclusion hierarchy in the DTD.

relationships in VT_1 have correspondence mappings in VT_1, then it means that the to-be-utilized HMVDs required by Q_1 are all preserved by Q_2 and also present in Q_2's result.

C4: Correspondence of construct types. This condition checks the correspondence between query construct types. A var node represents a *for* expression while a ret node denotes a *return* expression. The bindings of a ret node are definitely returned whereas those of a var node may be used for constructing new elements correspondingly. Therefore, a var node can be mapped to a ret node and still get the correct bindings, while a ret node cannot be mapped to a var node since the new elements in Q_2's result rather than the original bindings would be returned in doing so.

We can see from the above conditions that the MAC mapping ensures that all the essential variable bindings, the HMVDs among them, and their attached returns required by Q_1 are preserved in the result of Q_2.

MIC Mapping

In addition to the MAC mapping, we need to check if the binding set of each node in VT_1 is indeed a subset of that of its match in VT_2. This is guaranteed by the MIC mapping, which essentially checks XPath containment.

Definition 3.5. *Let VT_1 and VT_2 be the minimal VarTrees of Q_1 and Q_2 respectively. Suppose $\Phi(VT_1)=VT_2$ according to the MAC mapping. In* ***MIC mapping****, tree homomorphism is checked between the encapsulated navigation patterns for each pair of matched nodes. Two steps are carried out for each node u in VT_1:*

1). *If u $roots(VT_1)$, concatenate the XPath expressions along the path from $\Phi(parent(u))$ to $\Phi(u)$;*

2). *Assume that the XPath expression on the derivation edge of u is p_1 and the one obtained from step (1) is p_2. p_1 p_2 is checked with denoting XPath containment (i.e., there is a tree homomorphism from the pattern tree representation of p_2 to that of p_1[9]).*

3.3.4 VarTree Node Mappings for View Self Maintenance

Similar to VarTree node mappings for query containment in section 3.3.3, we define VarTree node mappings for view self maintenance as follows:

Definition 3.6. *Suppose VT_1 and VT_2 are the minimal VarTrees of update query U_1 and view query Q_1 respectively. For determining relevance of U_1 to Q_1, there must be a **MAC mapping** from VT_2 to VT_1, denoted by $\Phi(VT_2) = VT_1$, such that the following conditions are satisfied:*

1). *$C1 : roots(VT_2)$ $roots(VT_1)$,*

2). *$C2$: $\rho(v)$ $\rho(\Phi(v))$ where v VT_2, the ρ function returns the XPath binding of a VarTree node, and the ' ' operator denotes XPath containment.*

3). *$C3$: v can be either a var node or a ret node in VT_2, but $\Phi(v)$ is always a var node in VT_1,*

[9]Our XQuery fragment allows XPath(//,*,[]), for which the complexity of containment is CoNP-complete. However, the XPath containment complexity is reduced to PTIME if only two out of the three features are included. We refer the readers to [15] for the details of XPath containment.

4). *$C4$: u is an ancestor of v for all u, v VT_2 if $\Phi(u)$ is an ancestor of $\Phi(v)$ in VT_1 or if $\Phi(u) = \Phi(v)$, and*

5). *$C5$: If $\Phi(v)$ exists, then for any descendant node ω of $\Phi(v)$, $\rho(v) \nsubseteq \rho(\omega)$*

C1: Root inclusion[10]. This condition requires that each source XML document referred to in Q_1 must also be referred to in U_1. Correspondingly in the VarTrees, $roots(VT_2)$ returns the URLs of the source XML documents involved in Q_1, which should be a subset of those returned by $roots(VT_1)$.

C2: MIC Mapping. This condition is identical to the condition of **MIC mapping** used for query containment. It states that for a MAC mapping (Φ) to exist between the VarTree nodes of VT_2 and VT_1, there must also exist containment between their XPath expressions. Note that the direction of MAC mapping for view self-maintenance is from the contained XPath expression to the container XPath expression.

C3: Mapping of element types. In VarTree mappings for view self maintenance we map the *var* and *ret* nodes of the view query VarTree only to *var* nodes in the update query VarTree. The update nodes in the update query VarTree are analyzed separately after this MAC mapping process is completed.

C4: Preservation of ancestorships. This condition ensures that each variable node in the view query VarTree is mapped to a node in the subtree of its parent's mapping node in the update query VarTree. This is useful in restricting the scope of the tree to search for a mapping node during

[10]Our technique allows a query to involve more than one XML document. In this case, the corresponding VarTree is actually a forest of trees, which may be connected by equality conditions on variables across trees.

the analysis algorithm and results in improved performance of the analysis algorithm from Section 3.3.5.

C3: Depth of mapped node. The MAC mapping process in Figure 3.8 traverses the view query VarTree and tries to map nodes which satisfy the MIC mapping condition (C2). But because of the variable dependencies in VarTrees, $\rho(v) \quad \rho(\Phi(parent(v)))$ where v is any node in view query VarTree. In other words, a node can always be mapped to it's parent's (or any ancestor's) mapping node. This can result in multiple nodes in view query VarTree mapping to a single node in the update query VarTree. To prevent such mappings, we require that a view query VarTree node be mapped to the deepest possible update query VarTree node for which the MIC mapping condition (C2) holds true.

We would like to note again that the direction of MAC mapping for view maintenance is from the view query VarTree to the update query VarTree, i.e. from the contained nodes to the container nodes. This direction may seem contrary to the direction of MAC mapping used in query containment in section 3.3.3, but it makes the query analysis algorithm simpler.

3.3.5 The Analysis Algorithm

The analysis algorithm uses the following *pre-defined* functions:

- $\Phi(node)$**:** Returns the *VarTree* node to which the specified *VarTree* node is mapped. This is the MAC mapping function explained in the previous section.

- $\rho(node)$: Returns the XPath expression of the specified *VarTree* node.
- $\mathcal{T}(node)$: Returns type of the specified *VarTree* node. Possible return values are ret, var , del, or ins corresponding to *return*, *variable*, *delete*, or *insert* nodes respectively in a view or update query's *VarTree*.

```
Procedure: UpdateQueryAnalysis
Input: Essential VarTrees $VT_1$ and $VT_2$ of update query and view query respectively.
Output: A modified update query VarTree, VT, with only relevant updates, or an empty VarTree for
        a non-relevant input update query, or null when relevance analysis is non-determinable.
Begin Procedure:
01: boolean goDeeper = false
02: $VT$ = Copy of $VT_1$ /* Now VT = Update Query VarTree */
03: Map root node of $VT_2$ to root node of $VT_1$
04: foreach node $v$ in preorder traversal of $VT_2$ do
05:     if $\Phi(parent(v))$ does not exist then /* Parent of u is not mapped */
06:         continue
07:     endif
08:     foreach var node $\omega$ in preorder traversal of subtree of $\Phi(parent(v))$ do
09:         if XPath containment between $v$ and $\omega$ is undecidable then
10:              return null
11:         elseif $\rho(v) \nsubseteq \rho(\omega)$ then
12:             if $\rho(\omega) \nsubseteq \rho(v)$ then
13:                 continue /* Completely separate XPaths, so skip */
14:             elseif $\mathcal{T}(v) == var$ then
15:                 return null /*Non determinable case */
16:             endif
17:         endif
18:         foreach child $\psi$ of $\omega$ do
19:             if $\rho(v)$      $\rho(\psi)$ then
20:                 goDeeper = true
21:                 break
22:             end if
23:         end foreach
24:         if goDeeper == true then
25:             continue
26:         else
27:             $\Phi(v) = \omega$ /*Mapping found, map $u$ to $\omega$ */
28:             goDeeper = false
29:         break
30:     end foreach
31: end foreach
32: Delete all update nodes in VT with parent nodes not mapped
33: foreach update node $u$ in $VT$ do
34:     AnalyzeUpdateNode($u$)
35:     if $u.status ==$ non-determinable then
36:         return null
37:     endif
38: end foreach
39: Delete all non-relevant delete and insert nodes in VT.
40: return $VT$
End Procedure
```

Figure 3.8: Update Query Analysis Algorithm

```
Procedure: AnalyzeUpdateNode
Input: Update node $u$ in VarTree of update query $U_1$, VarTree representations
      $VT_1$ and $VT_2$ of update query $U_1$ and of view query $Q_2$, respectively;
Output: Sets the status of update node as either relevant, notrelevant or
      non-determinable
Begin Procedure:
01   if $parent(u)$ is mapping of a ret node in $VT_1$ then
02      u.status = relevant
03      return
04   end if
05   if $\mathcal{T}(u) == del$ then
06      foreach node $\omega$ in postorder traversal of subtree of $\Phi^{\ 1}(parent(u))$ do
07         if XPath containment between $u$ and $\omega$ is undecidable then
08            u.status = non-determinable
09         return
10         elseif $\rho(\omega) \not\subseteq \rho(u)$ then
11            if $\rho(u) \not\subseteq \rho(\omega)$ then
12                  continue /* Completely separate XPaths, so skip */
13            elseif $\mathcal{T}(\omega) == var$ then
14               u.status = non-determinable
15               return
16            endif
17         endif
18         u.status = relevant
19         return
20      end foreach
21      if u.status == null then
22         u.status = notrelevant
23         return
24      end if
25   else if $\mathcal{T}(u) == ins$ then
26a      Generate XML document using XPath expression of
            $\Phi^{\ 1}(u)$ and the insert content of $u$
26b      Evaluate XML View query on this generated document.
26c      If query evaluation does not return anything, this
            insert is not relevant to the view. Else the result of
            query evaluation is the content which is relevant to the view.
27   end if
End Procedure
```

Figure 3.9: Update Node Analysis Algorithm

The general idea of the algorithm in Figure 3.8 is as follows: For each node (v) in the view query VarTree find a *var* node (u) in update query VarTree such that v u. If such a node exits, then $\Phi(v) = u$ i.e. a MAC mapping is established from v to u. After such MAC mappings have been established, the algorithm analyzes each update node in the update query VarTree individually. The details of the query analysis algorithm in Figure 3.8 are as follows:

- This algorithm takes the VarTree representations of an update query (VT_1) and view query (VT_2) as inputs. The output is a modified update query VarTree which is relevant to the view query. If the output is an empty VarTree it means that the update is not relevant to the view query. If the output is **null**, it means that the analysis algorithm cannot determine if the query is relevant or not.
- Line 02 creates a copy of the original update query VarTree. This leaves the original VarTree undisturbed.
- Line 03 maps the root nodes of the two VarTrees. Whenever two queries operate on the same XML document this mapping is valid.
- Lines 04 through 31, iterate and process each variable node (v) in the view query VarTree as follows:
  - Each node in a view query VarTree is **derived** from its parent node due to the variable binding dependencies. If the parent of a variable node has not been mapped, then the child node also cannot be mapped. We skip processing of such a node and

continue with the analysis of the next variable node. This is done in lines 05 through 07.

  - We then try to find a mapping node for view query node (v) by searching each node (ω) in the subtree of the node to which the parent of v has been mapped. If XPath containment is undecidable for any of these nodes, then view self maintenance is not possible either and we return a **null** value (line 10). The two **if** conditions on lines 11 and 12 check if the Xpaths of ω and v are completely separate, i.e. there is absolutely no containment relationship between them. If they are, skip the node and continue with the analysis of next node in the update query subtree. The **if** condition on line 14 checks for non-determinable cases where the XPath of ω (update query variable node) is contained in the XPath of v (view query variable node) and returns a **null** value for such a case.
  - Lines 18 through 30 ensure that we find the deepest possible mapping node for v from the update query subtree. This satisfies Condition 5 of the MAC mapping described in section 3.3.4.

- Line 32 deletes all unmapped variable nodes and their children from the update query VarTree. This leaves only the possibly relevant update nodes in the update query VarTree.

- Lines 33 through 38 analyze all update nodes in the update query VarTree as explained in procedure in Figure 3.9. If analysis for any of the nodes is non-determinable, a **null** value is returned.

- Finally, we delete all update nodes marked as non-relevant in the update query VarTree (VT) and return the modified update query VarTree.

The update node analysis algorithm of Figure 3.9 works as follows:

- The input to the algorithm is the update node (u) to analyze, an update query VarTree (VT_1) and a view query VarTree (VT_2). This algorithm will set the status of this node as either **relevant**, **notrelevant**, or **non-determinable**.
- If the parent node of update node u is mapped to a *return* node in the view query VarTree, then the update is relevant.
- Determining relevance of delete nodes is very similar to the mapping of variable nodes in the analysis procedure of Figure 3.8.
- Insert nodes are processed as explained previously in section 3.3.2

3.4 Update Query Rewriting

Query rewriting is generally classified into two types, viz. Syntactic Query Rewriting, and Semantic Query Rewriting. Syntactic rewriting primarily involves rewriting the structure of the query, while semantic rewriting involves query rewriting using additional information like schema, indexes, materialized views, integrity constraints etc.

We define a syntactic query rewriting procedure to convert a relevant update query on the base document into an update query on a materialized view. By replacing path expressions in the original update query with

```
Procedure: RewriteUpdateQuery
Input: VarTree Representation of Update Query and TagTree
    representation of view query
Output: VarTree Representation of rewritten update query, or null
    if re-writing is not possible
Begin Procedure:
foreach node $n$ in VarTree do
    String m = GetMappingPath(tagtree.rootNode(), $n.xpath()$)
    if m is null then
        return null
    else
        String p = GetMappingPath(tagtree, (parent($n$)).xpath())
        if p.length $>$ 0 then
            Replace xpath expression of edge incident on $n$ with
            m.substring(p.length(), m.length())
        else
            Replace xpath expression of edge incident on $n$ with
            empty string
        end if
    end if
end foreach
Delete all nodes with empty xpath labels on edges incident to them
End Procedure
```

Figure 3.10: Procedure RewriteUpdateQuery

their equivalent paths in the materialized view, we get a new update query which when executed on a materialized view will synchronize it with the changes in base document. For performing this rewriting, we make use of the **TagTree** data structure which stores the output restructuring information of a view query. Figure 3.11 shows an algorithm for calculating mapping paths from a TagTree. The rewriting algorithm of Figure 3.10 uses this algorithm to rewrite update query VarTrees.

Procedure: GetMappingPath
Input: *node* ω, root node of a TagTree, and *xpath* the Xpath whose mapping path is to be found
Output: The mapping XPath for input xpath, or an empty string if none found, or **null** if finding mapping path is not possible
Begin Procedure:

```
String newpath = ""
if Xpath containment between xpath and $\rho(\omega)$ is undecidable then
   return null /*Undecidable*/
else if $\rho(\omega) \not\subseteq xpath$ and $xpath \not\subseteq \rho(\omega)$ then
   return newpath /*No mapping exists*/
else if $\omega$ is a leaf node then
   if $xpath$     $\rho(\omega)$ then
      newpath = newpath + "/" + deepCopy(xpath, $\rho(\omega)$)
   else
      newpath = newpath + "/" + $\rho(\omega)$
   end if
else
   newpath = node.tags /*Add the node's tags to the xpath*/
   if $\rho(\omega) \not\subseteq xpath$ then
      foreach child in $\omega.children()$
         String childpath = GetMappingPath(child, xpath)
         if childpath is null then
            return null
         else if childpath.length $> 0$ then
            newpath = newpath + "/" + childpath
         end if
      end foreach
   end if
end if
return newpath
```

End Procedure

Figure 3.11: Algorithm to get mapping paths from a TagTree

3.5 Complexity Analysis of Analysis Algorithm

We now discuss the time complexity of our Update Query Analysis algorithm. The two primary costs involved in this analysis are

- Cost of MAC Mapping, and
- Cost of analyzing update nodes.

3.5.1 Cost of MAC Mapping

MAC mapping tries to map nodes in the view query VarTree to nodes in the update query VarTree, based on their XPath containment relationship. MAC mapping thus involves VarTree traversals and checking for XPath containment.

VarTree Traversals Due to variable binding dependencies [11] in the VarTree representations of queries, the traversals during MAC mapping are downward only. This downward-only direction of traversal and mapping progressively limits the scope of update query VarTree to search for possible mapping nodes. On an average, each node in the view query and update query VarTree is traversed only once.

XPath Containment Checking Chapter 2 presents a summary of results from XPath containment. In this thesis, we use a subset of XPath which is amenable to PTIME containment checking [12].

[11] Recall that each variable node in a VarTree is derived from it's parent node.

[12] XPath expressions using any two of the the three operators from $XP^{[}/.[], //]$ are amenable to PTIME containment checking

From the above two costs, we can easily see that the overall complexity of the analysis algorithm is polynomial in the number of nodes of the view query and update query VarTree, i.e. polynomial in the number of essential variables in the two queries.

3.5.2 Cost of Analyzing Update Nodes

Delete Nodes can be analyzed solely based on XPath containment techniques and hence the only cost involved is that of XPath containment checking of the delete node with view query nodes.

Insert Nodes require additional handling as explained in Section 3.3.2. In addition to XPath containment, our analysis of insert nodes requires

- Generation of a temporary XML document based on XPath expression in the view query.
- Evaluating the view query on this temporary XML document.

Due to the small sizes of the XPath used for generating the temporary XML document and also the size of the document, the cost of this process is negligible when compared even to the cost of XPath containment. The results of our experiments in Chapter 4 confirm this analysis. Thus, even for insert nodes, the cost of analysis is the same as that of XPath containment which is polynomial in the size of the XPath expressions.

Thus, the complete cost of the analysis algorithm is polynomial in the size of the essential variables and update operations in the view and update queries.

3.6 Examples

We now explain our solution with several examples. In all these examples, we will use a document conforming to the XMark Schema and the view query example from Chapter 2. Recall that the view query outputs the buyer and price information of all sold items costing more than $1000. The view query definition is repeated in Figure 3.12 for easy reference. We focus our examples on update operations for elements. Attributes, comments, processing instructions etc. are handled similar to elements and do not need special explanation. Also, in these examples we explain the three possible outcomes of query analysis (**relevant**, **notrelevant**, and **non-determinable**) for both delete and insert operations, i.e. we consider six possible cases in these examples.

```
<result>
FOR $item IN doc("auctions.xml")/site/closed_auctions/closed_auction[price>1000]
RETURN
<entry>
{
        $item/buyer, $item/price
}
</entry>
</result>
```

Figure 3.12: Normalized View Query used in examples

3.6.1 Case 1: Relevant Delete Operation

If a user wants to delete home-page information of all buyers of sold items, the update query for that is shown in Figure 3.13. This update query is clearly relevant to the view defined in 3.12 because it updates information returned by the view query.

```
FOR $solditem IN doc(“auction.xml”)/site/closed_auctions/closed_auction
UPDATE $solditem
{
          DELETE $solditem/buyer/homepage
}
```

Figure 3.13: Relevant Delete Query

The *VarTree* representation of this update query is as shown in Figure 3.14. In this *VarTree* the leaf node represents the delete operation, the variable bindings are represented by non-leaf nodes and edges are labeled the XPath expressions corresponding to the variable nodes. As a convention, the root node of the VarTree is always bound to the document's root node.

To determine relevance of this query to the view query, we perform the MAC mapping procedure outlined in Figure 3.8. This results in three mappings between the two queries as shown in Figure 3.15.

1). Mapping 1 exists between the two root nodes because the two queries operate on the same base document

2). Mapping 2 exists because the XPath for **$solditem** node in the view query's VarTree is a subset of the XPath of **$solditem** node in the update query's VarTree.

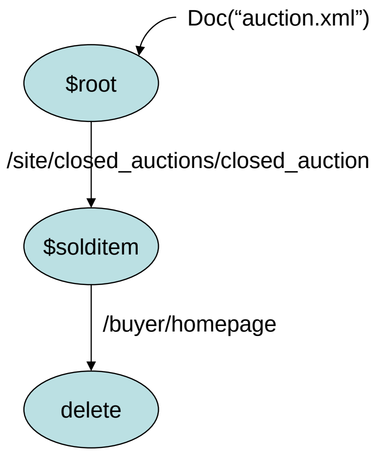


Figure 3.14: *VarTree* representation of query in Figure 3.13
Figure

3). To analyze the delete node, we search the subtree of view query rooted in **$solditem** for XPath containment relationship between the subtree nodes and the delete node. In this case, the XPath expression of delete node is contained in the XPath expression of return node **$solditem/buyer**. This containment relationship means that the delete operation deletes some nodes returned by the view query. Hence this delete node will be marked as **relevant** by the analysis procedure.

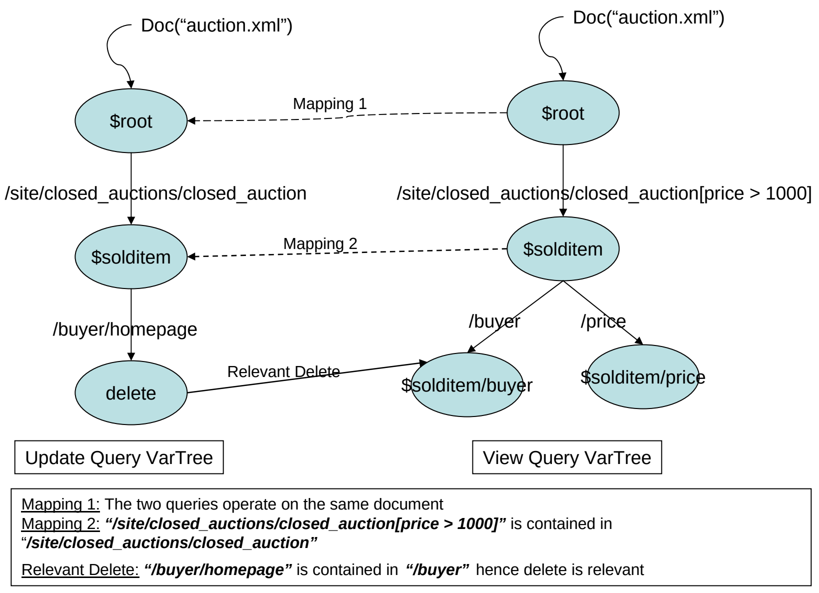

Doc("auction.xml")
Doc("auction.xml")
$root
Mapping 1
$root
/site/closed_auctions/closed_auction
/site/closed_auctions/closed_auction[price > 1000]
$solditem
Mapping 2
$solditem
/buyer/homepage
/buyer
/price
delete
Relevant Delete
$solditem/buyer
$solditem/price
Update Query VarTree
View Query VarTree
Mapping 1: The two queries operate on the same document
Mapping 2: "/site/closed_auctions/closed_auction[price > 1000]" is contained in "/site/closed_auctions/closed_auction"
Relevant Delete: "/buyer/homepage" is contained in "/buyer" hence delete is relevant

Figure 3.15: *VarTree* mappings for Case 1

3.6.2 Case 2: Non Relevant Delete Operation

Consider an update query which deletes the annotation description for a sold item. The update query for such a delete is shown in Figure 3.16. Clearly, this query does not affect any data in the materialized view and is hence is not relevant.

```
FOR $solditem IN doc("auction.xml")/site/closed_auctions/closed_auction
UPDATE $solditem
{
        DELETE $solditem/annotation/description
}
```

Figure 3.16: Non Relevant Delete Query

This query can be decomposed into its *VarTree* representation as shown in Figure 3.17.

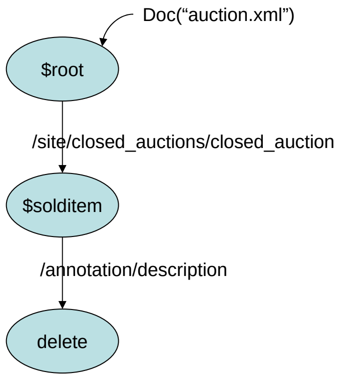


Figure 3.17: *VarTree* representation of query in figure 3.16

To determine relevance of this update query, we again perform an MAC mapping procedure between the nodes of the view query VarTree and the update query VarTree. This mapping is shown in Figure 3.18. The variable node **$solditem** gets mapped similar to the previous case. Notice that this time, for the delete node in the update query VarTree:

1). The XPath of the delete node is not contained in the XPath of any return node path.

2). No XPath in the view query is equivalent to or completely contained in the XPath of the delete node.

This means that the delete operation does not affect any elements returned by the view query in the materialized view. Hence the analysis procedure marks this query as **notrelevant**.

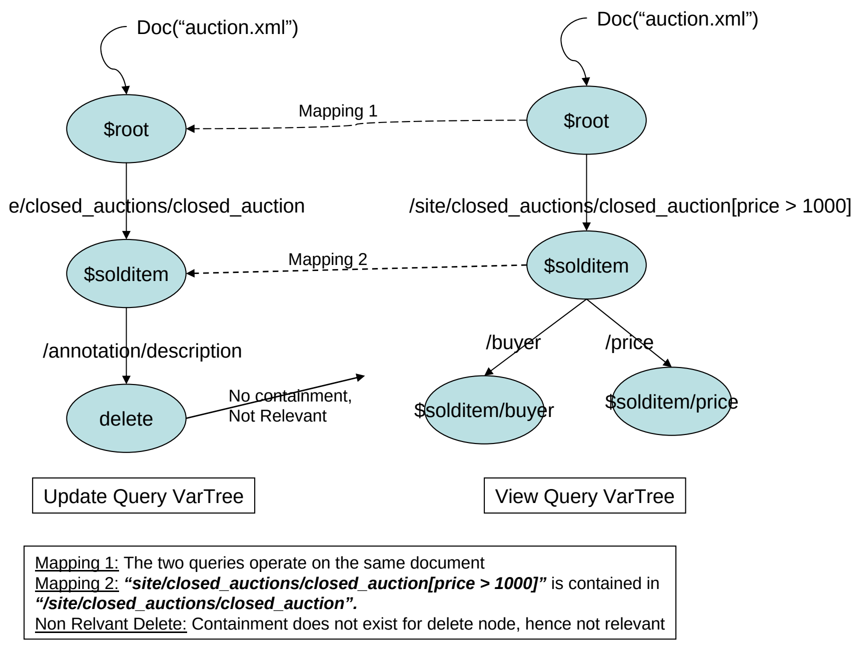

Doc("auction.xml")
Doc("auction.xml")
$root
Mapping 1
$root
e/closed_auctions/closed_auction
/site/closed_auctions/closed_auction[price > 1000]
$solditem
Mapping 2
$solditem
/annotation/description
/buyer
/price
delete
No containment, Not Relevant
$solditem/buyer
$solditem/price
Update Query VarTree
View Query VarTree
Mapping 1: The two queries operate on the same document
Mapping 2: "site/closed_auctions/closed_auction[price > 1000]" is contained in "/site/closed_auctions/closed_auction".
Non Relvant Delete: Containment does not exist for delete node, hence not relevant

Figure 3.18: *VarTree* mappings for Case 2

3.6.3 Case 3: Non-determinable Delete Operation

Consider an update query similar to that in case 1, except that here we delete home-page information of all buyers of sold items costing more than $5000. The update query for such a delete is shown in 3.19.

```
FOR $solditem IN doc("auction.xml")/site/closed_auctions/closed_auction[price > 5000]
UPDATE $solditem
{
        DELETE $solditem/buyer/homepage
}
```

Figure 3.19: Non-Determinable Delete Query

This query can be decomposed into its *VarTree* representation as shown in 3.20

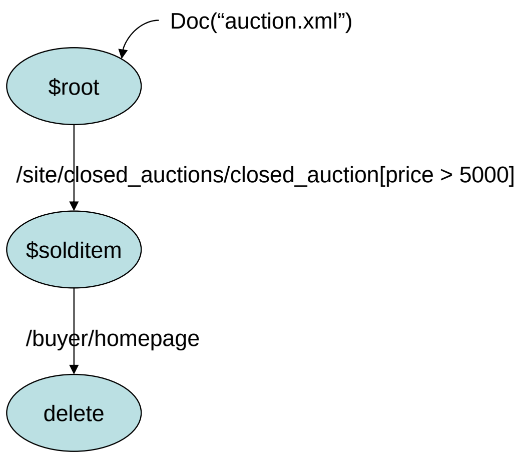


Figure 3.20: *VarTree* representation of query in figure 3.19

Also consider a slightly modified view query which returns the buyer information and the **quantity** of items bought instead of **price** of the item. To determine relevance of this update query, we again perform a MAC mapping procedure between the nodes of the view query VarTree and the update query VarTree. This mapping is shown in 3.21. Notice that due to the predicates on the *price* element, the **$solditem** variable node in the update query is contained within the **$solditem** variable node of the view query. Hence as explained in section 3.3.5 the analysis procedure will not be able determine the exact elements to delete from the materialized view. Thus, this specific case becomes non-determinable for the analysis algorithm. The materialized view will have to be recomputed for such an update query.

The original view query used in all other examples contains a filter on **price** element in the variable binding of **$solditem** and **price** element is re-

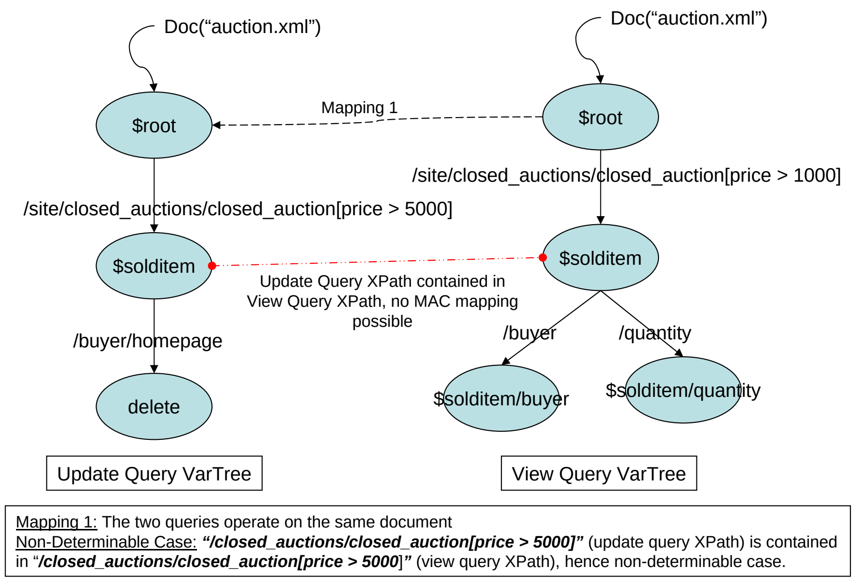

Doc("auction.xml")
Doc("auction.xml")
$root
Mapping 1
$root
/site/closed_auctions/closed_auction[price > 1000]
/site/closed_auctions/closed_auction[price > 5000]
$solditem
$solditem
Update Query XPath contained in View Query XPath, no MAC mapping possible
/buyer/homepage
/buyer
/quantity
delete
$solditem/buyer
$solditem/quantity
Update Query VarTree
View Query VarTree
Mapping 1: The two queries operate on the same document
Non-Determinable Case: "/closed_auctions/closed_auction[price > 5000]" (update query XPath) is contained in "/closed_auctions/closed_auction[price > 5000]" (view query XPath), hence non-determinable case.

Figure 3.21: *VarTree* mappings for Case 3

turned by the view query. In such a case, even if the update query variable binding is contained in the view query, the materialized view has enough information for analysis to determine the exact elements to delete. Our approach can certainly be extended to handle such special cases.

3.6.4 Case 4: Relevant Insert Operation

Figure 3.22 shows an update query which adds a newly closed auction into the auctions database. The VarTree decomposition of this query is shown in Figure 3.22. The first step is to establish MAC mappings between the nodes of the view query and those of the update query VarTree. The queries operate on the same XML document, the root nodes map. The **closeditem** variable node in view query maps to the **ca** node in the update query since there is containment relationship between the XPaths of these nodes.

```
FOR $ca IN doc(“auction.xml”)/closed_auctions
UPDATE $ca
{
            INSERT
                        <closed_auction>
                        <item><name>Ship Starlight</name></item>
                        <price>1050</price>
                        <buyer>
                                    <name>Jack Gambini</name>
                                    <creditcard>7657 7675 9786 4592</creditcard
                        </buyer>
                        <seller><name>Assef Muniz</name></seller>
                        <date>08/29/2005</date>
                        </closed_auction>
}
```

Figure 3.22: An Insert Query relevant to View Query in Figure 3.2

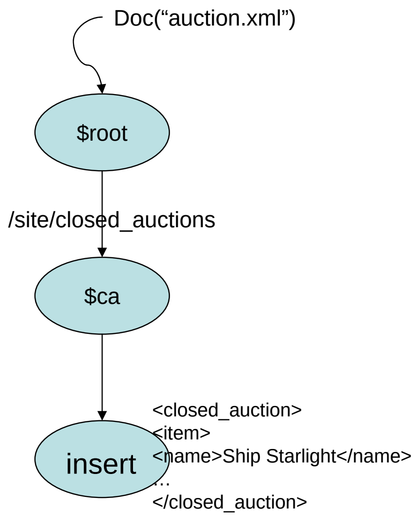


Figure 3.23: VarTree representation of query in Figure 3.22

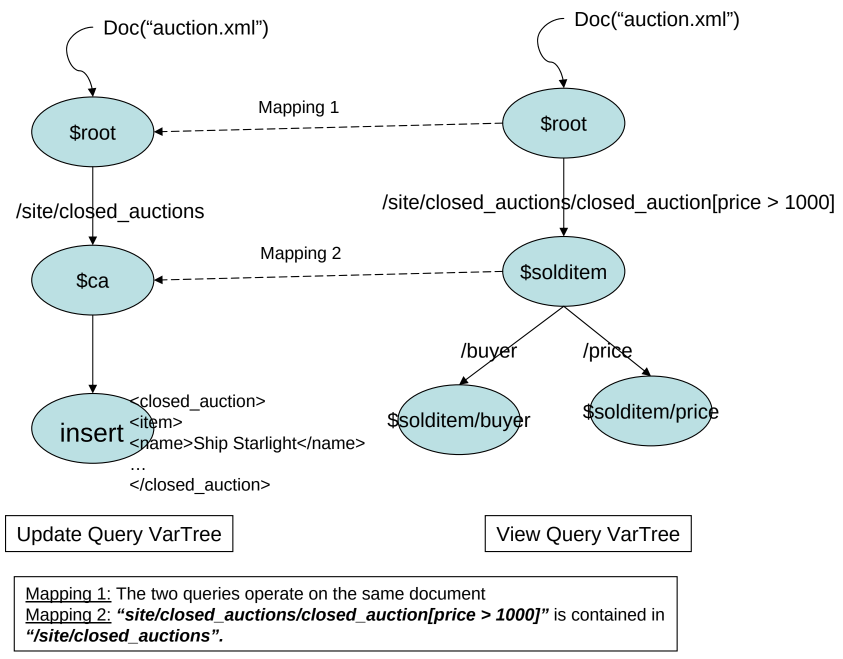

Doc("auction.xml")
$root
/site/closed_auctions
$ca
insert
<closed_auction>
<item>
<name>Ship Starlight</name>
...
</closed_auction>
Update Query VarTree
Mapping 1
Mapping 2
Doc("auction.xml")
$root
/site/closed_auctions/closed_auction[price > 1000]
$solditem
/buyer
/price
$solditem/buyer
$solditem/price
View Query VarTree
Mapping 1: The two queries operate on the same document
Mapping 2: "site/closed_auctions/closed_auction[price > 1000]" is contained in "/site/closed_auctions".

Figure 3.24: VarTree mappings for Case 4

These mappings are shown in Figure 3.24. The next step is to determine the relevance of the insert operation. For this, notice that the content is being inserted at location

$/site/closed\_auctions$

in the document. This XPath location corresponds to the node **$solditem** in the view query. We can use this XPath location information from the view query and the insert content from the update query to generate a new XML document as shown in Figure 3.25. In this figure, the insert content from the update query is in the gray box, while the rest of the XML elements have been generated using the XPath expression of **$solditem** i.e. ($/site/closed_auctions$) variable binding in the view query.

```
<site>
 <closed_auctions>
 <closed_auction>
        <item><name>Ship Starlight</name></item>
        <price>1050</price>
        <buyer>
                <name>Jack Gambini</name>
                <creditcard>7657 7675 9786 4592</creditcard>
        </buyer>
        <seller><name>Assef Muniz</name></seller>
        <date>08/29/2005</date>
 </closed_auction>
 </closed_auctions>
</site>
```

Figure 3.25: Generated Document For analyzing insert query in Figure 3.22

Next, we evaluate the view query on this generated document shown in Figure 3.25. The result of this evaluation is shown in Figure 3.26. Since the result is not empty, the insert query is relevant to the view and Figure 3.26 shows the actual insert content to update the materialized view

```
<result>
        <entry>
        <buyer>
                <name>Jack Gambini</name>
                <creditcard>7657 7675 9786 4592</creditcard>
        </buyer>
        <price>1050</price>
        </entry>
</result>
```

Figure 3.26: Result of evaluation view query on the generated document of Figure 3.25

3.6.5 Case 5 Non Relevant Insert Operation

An update query could be non-relevant to a materialized view in either of the following two cases:

- Absence of XPath containment between view query and update query VarTree nodes
- Insertion of not-relevant content

An example for absence of XPath containment is already considered in Case 2 for non-relevant delete operation. To understand how we eliminate queries where the insert content is not-relevant, you can re-work the example in Case 4 with price 900 instead of 1050. You will notice that the result of evaluating the view query on the generated document in this case does not contain any elements from the original elements being inserted. Hence we can conclude that such a query is not relevant to the materialized view.

3.6.6 Case 6: Non determinable Insert Operation

Consider an update query which adds a new annotation for a closed auction where the buyer bought more than 5 items named "Ship Starlight". Such an update query is shown in 3.27. If you work out the MAC mapping procedure for this example, you will notice that the containment relationship between the XPath expressions of **$ca** node of the update query and **$item** node of the view query is undecidable. Hence it is also not possible to determine if this update query is relevant to the view or not. The materialized view will have to be recomputed in this case.

```
FOR $ca IN doc(“auction.xml”)/site/closed_auctions
WHERE $ca/item/name=“Ship Starlight” and $ca/quantity>5
UPDATE $ca
{
   INSERT
     <annotation>
       <description>This buyer really likes paintings by Fitz Hugh Lane!</descriptio
     </annotation>
}
```

Figure 3.27: Non determinable insert query

3.6.7 Case 7: Other Undeterminable Updates

Update queries which modify the parent elements of returned elements elements thereby causing their child elements to be relevant or not relevant to the view query cannot be handled by our approach.

3.6.8 Update Query Rewriting Example

In this final example, we explain the update query re-writing procedure. Figure 3.28 shows the TagTree representation of the View Query in Figure **??**. Each internal node in the TagTree represents a new element construction in the view query, and each leaf node corresponds to a return expression. Using this TagTree structure, the algorithm in Figure 3.11 can be used to find paths of elements in the materialized view from their original XPaths in the base document.

Once an update query is determined relevant to a materialized view, the query-rewriting algorithm rewrites it into an update query on the materialized view. The rewriting algorithm replaces each XPath in the original update query with its corresponding XPath in the materialized view. For

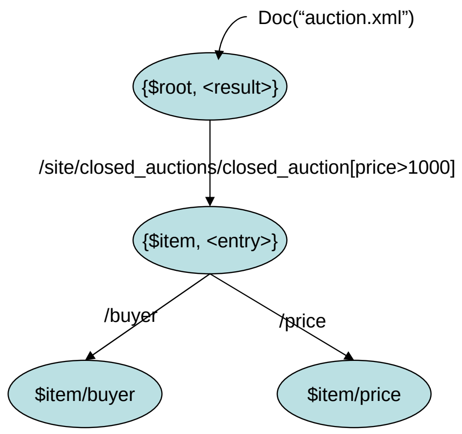


Figure 3.28: TagTree Representation of View Query in Figure 3.2

example, we repeat the VarTree representation of a relevant delete query in Figure 3.29

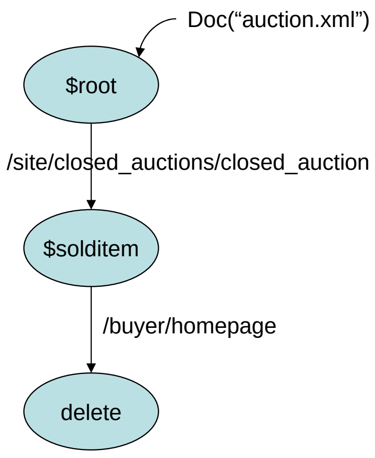


Figure 3.29: VarTree of relevant Delete operation

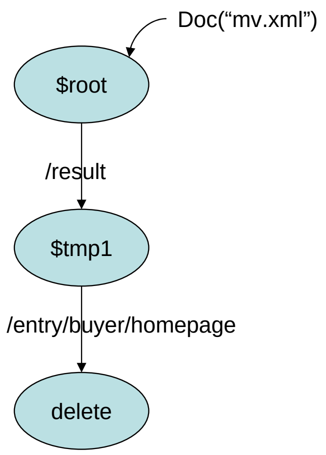
The path mappings for this update query are as shown in Table 3.1

Using these path mappings, the re-written update query VarTree is shown in Figure 3.30. This update query when executed on the materialized view

Original Path	New Path
$/site/closed\_auctions/closed\_auction(= \$a)$	$/result$
$\$a/buyer/homepage$	$/result/entry/buyer/homepage$

Table 3.1: Path Mappings for Update Query in Figure 3.29

will synchronize it with the base document.



```
FOR $root in doc(“mv.xml”)
Update $root {
        FOR $tmp1 in $root/result
        Update $tmp1 {
                Delete $tmp1/entry/buyer/homepage
        }
}
```

Figure 3.30: Rewritten Update Query

Chapter 4

System Implementation and Experiments

System Implementation and Experiments

In this chapter we explain the system we built and the experiments we conducted to validate our ideas on view self maintenance.

4.1 System Implementation

Our View Self Maintenance system has been built and integrated into the ACE-XQ semantic caching system. Figure 4.1 shows the ACE-XQ System Architecture. The various components of this system are:

The **Query Parser** is responsible for parsing user input X-Queries. ACE-XQ has been prototyped using two query engines, viz. Kweelt [26] and IPSI-XQ [17]. The output of the query parser is then intercepted by the **Query Decomposer** which does the three steps involved in query pre-precessing, viz., *variable minimization*, *query normalization*, and *query decomposition*. The query decomposition process outputs the VarTree and the TagTree representations of an input XQuery. The **Query Pattern Register** associates input queries with the cached query results. Various cache operations like cache invalidation, cache region coalescing and splitting etc. are handled by the **Replacement Manager**. The **Query Containment Mapper** performs the important function of deciding containment between two queries. Depending on the result of the query containment mapper, the **Query Rewriter** uses the query's TagTree structure to decompose the input query into a *probe query* (to retrieve answers from the cached local views), a *remainder query* (to retrieve answers from the remote data sources), and a *combining query* to make one complete answer.

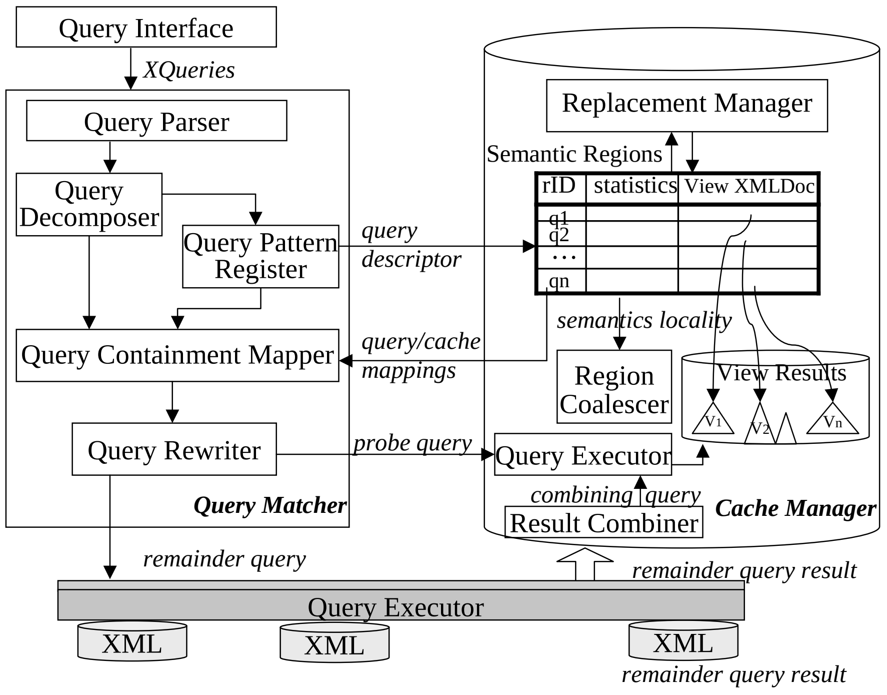


Figure 4.1: The ACE-XQ System Architecture

The ACE-XQ system has been developed using two query engines viz., Kweelt and IPSI-XQ. ACE-XQ uses the query engines' query parsing capabilities, intercepts the output of the query parser and then does further processing like query normalization, decomposition etc. For integrating our view self-maintenance approach into ACE-XQ, we also need XML Update functionality. The SafeXUpdate [18] project at WPI has implemented the update query language of [28] into the Kweelt Query engine. We used this modified version of the Kweelt query engine for our development and testing.

Figure 4.2 shows the combined ACE-XQ system with both the view self maintenance and the semantic caching components.

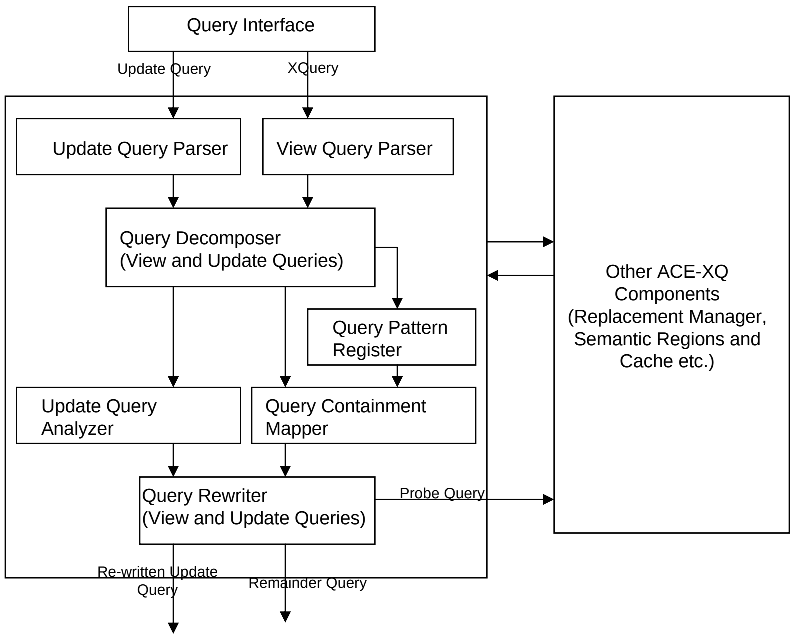


Figure 4.2: The ACE-XQ Architecture with View Self Maintenance

The primary software components that we developed for view self maintenance are:

1). **Query Interface:** We improved the Query Interface of ACE-XQ to accept both view and update queries. Depending on the the user input, the query interface inputs the user query either to the view query parser or the update query parser.

2). **Update Query Normalizer and Decomposer:** Similar to the view query processing, we intercept the output of the update query parser to normalize it and decompose it into an update query VarTree. The normalization and decomposition routines are very similar to those of the view query.

3). **Update Query Analyzer:** We developed query analysis engine according to the algorithm explained in Section 3.3. The analysis algorithm examines nodes in both the update query and view query VarTrees for determining containment between their XPath expressions. We re-used the XPath containment routines developed in ACE-XQ. Also, during analysis of INSERT nodes we construct an XML document and evaluate the view query on this XML document. We used the Kweelt Query engine itself for the query evaluation during this analysis step.

4). **Update Query Rewriter:** The Query rewriting step utilizes the TagTree structure of the view query. ACE-XQ Query Decomposer has the capability to decompose a view query into its equivalent TagTree representation. From the TagTree component itself, we can find mapping paths in the materialized view and perform query rewriting using the algorithms described in Section 3.4.

4.2 Experimental Setup and Results

The main objective of our experiments is to measure the performance of view self maintenance as compared to recomputing the view for every update. We measure these performance enhancements for both insert and delete operations. The parameters to control and measure in our experiments include the base document size, the size of the update, the view query selectivity and also the location of update. In our experiments, we vary one parameter at a time, keeping the others constant. Another possible factor to consider is the complexity of the view or the update query (measured in terms of the number of variable binding dependencies and the number of update operations). Intuitively, query analysis and re-writing will cost more for complex queries than for simpler ones. However, in our experimental setup we focus more on simpler view queries which do not involve more than one level of variable binding dependencies and on update queries which contain only one update operation. View Query preprocessing is a one-time operation and update query analysis occurs in-memory. For these reasons, we believe that the complexity of view and update queries will unlikely affect the overall performance of our view maintenance strategy in a big way. Hence we do not consider the complexity of queries as a parameter in these experiments.

In all our tests, we use data generated by the XMark benchmark data generator. The base document consists of several auctions related elements, including ”person” elements which constitute about 10% of the entire document size. The view query in all our experiments returns a subset of ”person” elements from the base document. The result of this view query execution is stored as a materialized view on a local disk, while the base documents used for querying are located on a remote host.

We conducted our experiments on a host running the Linux OS (RedHat Linux 9.0, kernel version 2.4) with 256MB RAM and a 1.2 GHz Intel Pentium 4 processor. The Linux OS was booted in run level 3 (multi-user, non graphical mode) with only the essential OS services running. This setup allowed a more controlled environment with little chance of interference from other software services running on the system. In all our experiments, we ran the tests seven times, discarded the minimum and maximum values and averaged the remaining five readings.

4.2.1 Cost of single update operation

Figure 4.3 shows the cost of materialized view maintenance for a single relevant insert or delete operation on the base document is much lesser than the cost of recomputing the materialized view. In this setup, the base document is about 1M in size, with 254 person elements.

It is important to understand the major parameters affecting these costs. The view query engine evaluates queries by reading in the entire base document, examining all elements and then selecting only the desired subset of elements to form the query result. The update query engine also has a similar evaluation strategy. Thus, the size of the XML document on which these queries operate is a major determining factor in the view or update query performance. Moreover, the I/O cost of reading the XML document from a local disk or over a network usually outweighs the in-memory filtering cost during query evaluation. For both disk based I/O and network I/O, the total time for I/O directly depends on the amount of data transferred. However, this is not a linear dependency due to one-time setup costs at the beginning of a network I/O operation, or due to strategies like block-level reading, buffering, and read-ahead pre-fetching commonly used by disk drive controllers [25]. For example, doubling the amount of data transferred over an I/O channel will typically require less than twice the amount of time, provided other parameters like disk/network queues lengths during the I/O operation are similar. These factors explain why view self-maintenance costs in Figure 4.3 are less than view re-computation. Also, for view self maintenance, the cost of update query analysis is a small fraction of the total cost as can been seen in Figure 4.8.

4.2.2 Effect of changing the document size

In these experiments, we vary the size of the base document (from 100KB to 10MB) while keeping view query selectivity at 5% and performing only a single update operation. Figure 4.4 show that view self maintenance again outperforms view re-computation. In this figure we see that both the cost of recomputation as well as view maintenance increase with an increase in the base document size. However, the cost of view self maintenance is much less than the cost of re-computation and the increase is also more gradual. Another aspect to note is that a five-fold increase in the base document size

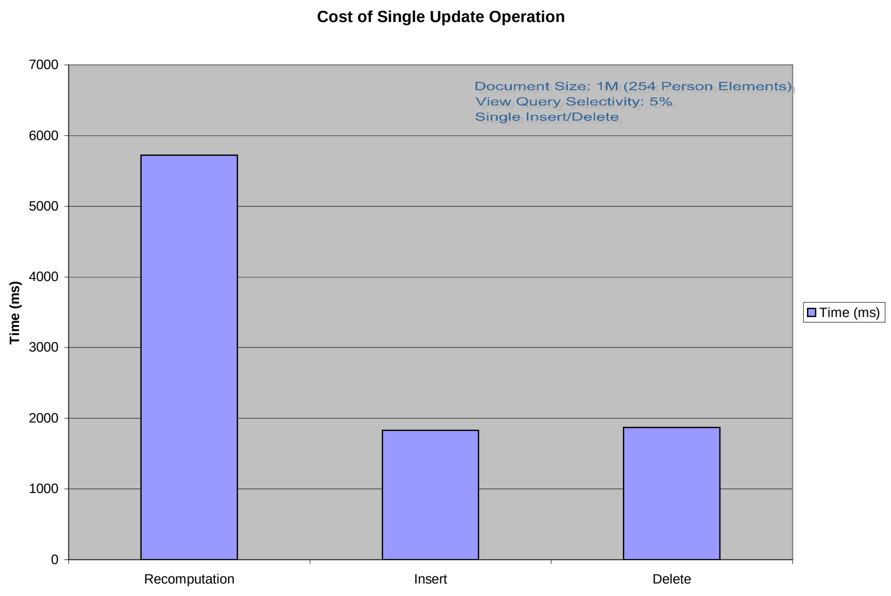


Figure 4.3: Cost of single Update Operation

(from 1M to 5M) does not translate into a five-fold increase in query evaluation time due to the non-linear dependency between document sizes and query evaluation performance explained in the previous experiment. Beyond a certain base document size (around 8MB on our system), the cost of recomputation shoots up dramatically as the OS cannot allocate any more main memory to the query engine process and it starts using disk based virtual memory. We expect similar behavior even for view maintenance when the materialized view size crosses the threshold size of about 8MB on our system.

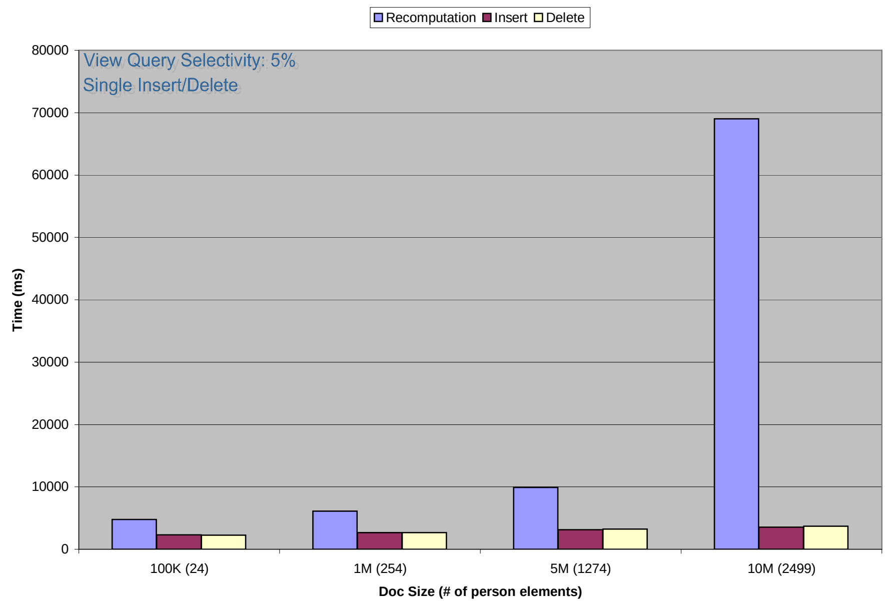


Figure 4.4: Test for changing document size

4.2.3 Effect of changing the view query selectivity

In these experiments, we measure the effect of changing view query selectivity while keeping the base document size (5M) and update size (single update) constant. Figure 4.5 shows the results for this test. In both cases, as the view query selectivity increases, the cost of recomputation and view maintenance also increases. Again, the cost of view maintenance is significantly less than the cost of recomputation for both kinds of update.

4.2.4 Effect of changing the update size

In these experiments, we vary the number of elements affected by the update operation from 5% to 30% for insert queries and from 2% to 8% for delete queries while keeping other factors , i.e. base document size (5M)

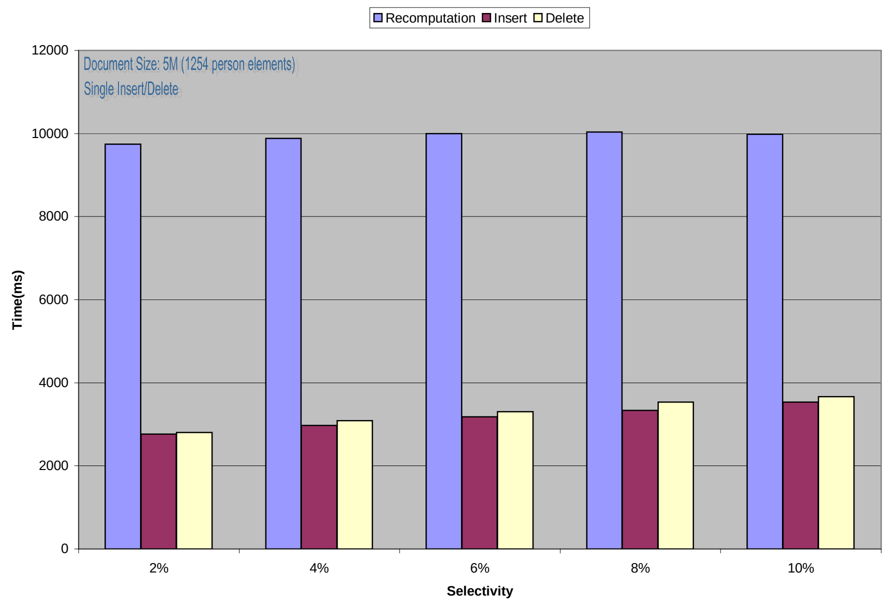


Figure 4.5: Test for changing view query selectivity

and view query selectivity (5%) constant. Figures 4.6 and 4.7 show the results of these experiments. In case of insert operations, the cost of recomputation as well as the cost of view maintenance increases as more elements are added to the base document. However, the recomputation costs are much higher than the view maintenance costs. For delete operations, the cost of recomputation decreases as more person elements are deleted from the base document. View maintenance continues to outperform recomputation even as a larger amount of the base document is deleted. The reason for this is that base document contains some other elements other than ”person” elements. Even when all the person elements are deleted from the base document, the query engine we use, parses all these other elements when recomputing the view. In a smarter query engine with indexing and query optimization features, we expect that the cost of examining other elements will be negligible. In such a case, we expect the cost of view maintenance to be higher than the cost of recomputation when a large portion of the base document is being deleted.

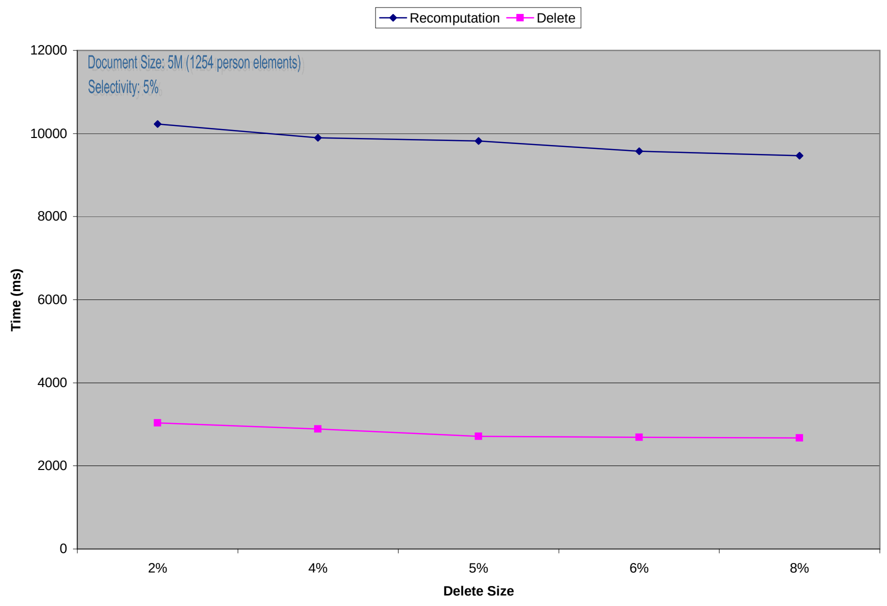


Figure 4.6: Test for changing size of delete operation

4.2.5 Cost of Query pre-processing and Query Analysis

Figure 4.8 shows the cost of view query pre-processing and query analysis (for a delete query) for the experiment in Figure 4.4 for five successive trials. Query pre-processing cost involves reading and parsing the view query as well as variable minimization, normalization, and decomposition. This is a one-time cost for each view query in the ACE-XQ system. (We do not show the query pre-processing cost for update queries as it is very similar). The

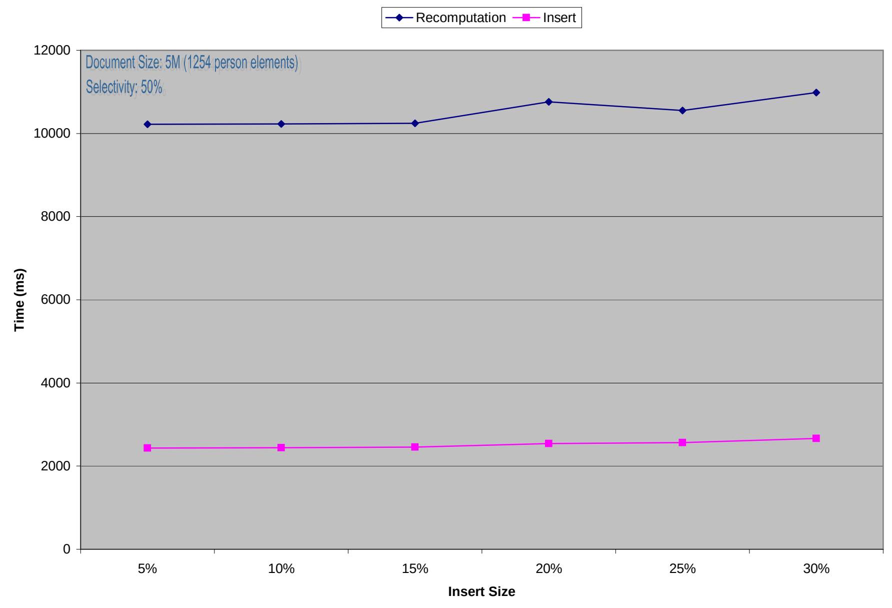


Figure 4.7: Test for changing size of insert operation

cost of update query analysis is very small fraction of the overall processing cost. For most practical queries, we expect the cost of query analysis will remain a tiny fraction of the overall query processing cost.

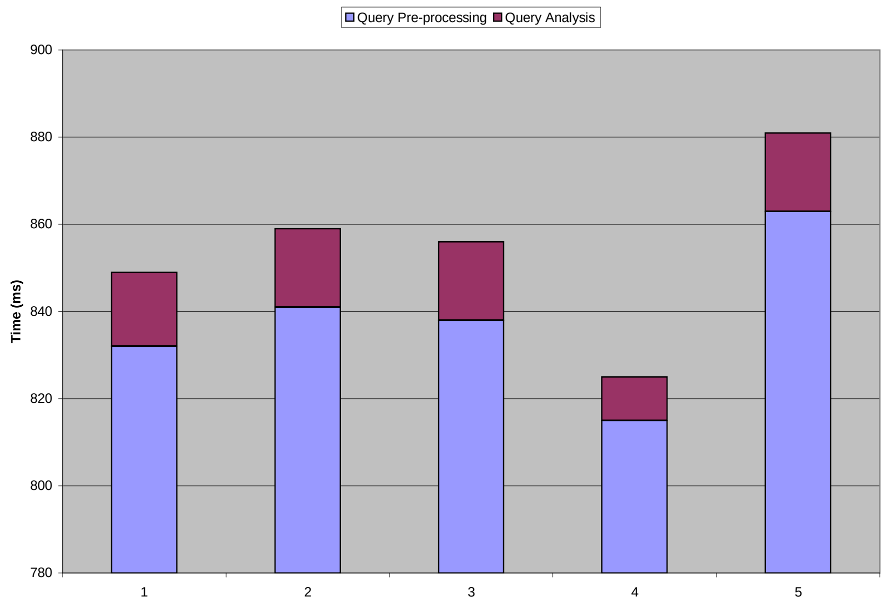


Figure 4.8: Cost of Query Preprocessing and Analysis for 5 successive trials of experiment in Figure 4.4

Chapter 5

Related Work

Related Work

5.1 Related Research

Incremental maintenance of materialized views has been studied in significant detail in the context of relational databases [7, 19]. [16] presents a taxonomy of this problem based upon the class of views considered (Language Dimension), upon the resources used to maintain the view (Information Dimension), upon the types of modifications to the base data that are considered during maintenance(Modification Dimension), and whether the technique works for all instances of databases and modifications (Instance Dimension). They also discuss several techniques in solving this problem including the counting algorithms, algebraic differencing algorithms and also algorithms like the DRed (Derivation and Reduction) algorithm, the PF (Propagation/Filtration algorithm) and the Kuchenhoff algorithm for maintaining recursive views.

The MultiView project examines views and their maintenance in object oriented databases [20]. They use object oriented concepts like virtual classes, class types, generalizations etc. to perform view maintenance.

For XML Views there are two possible approaches to incremental maintenance of materialized XML views viz., *algebraic*[9, 12] and *algorithmic* [2, 36], The algebraic approach requires the existence of a formal query algebra. In [12] an XML query is decomposed into an XML Algebra Tree (XAT) consisting of elementary algebra operators like SELECT, TAG, NEST, UNNEST etc. Operations like query execution, query optimization, as well as incremental maintenance of materialized views is performed at this algebra tree level.

The algorithmic approaches, by contrast rely on procedures to detect relevance of updates to data sources and applies a corresponding change on the materialized view. The approach in [2] is based on the XQL query language and it maintains an index structure of OID's affected during view computation. This index structure is then used for view maintenance. APIX [5] improves on this index structure to perform more efficient view maintenance. However, both these approaches are based on a simpler query language (XQL) without much expressive power and output restructuring capabilities. Moreover, they also assume unique Object Identifiers (OIDs) for each XML element which may not be true for most XML databases. [36] explain view maintenance of select-project graph structure views defined as a collection of objects from a graph structured database. Their approach also assumes unique OIDs and does not allow restructuring of elements in the view. [21] developed WHAX (Warehouse Architecture for XML) which uses a variation of XML-QL for view definition. Their approach to view maintenance is based on the concept of multi-linearity of views which is conceptually similar to our approach to handling insert operations. However, the WHAX approach also uses local keys which can provide unique object identifiers. More recently, other research projects also have explored XML caching environments which are likely to require incremental view maintenance. Both [23] and [34] primarily rely on caching of XPath expressions rather than XQuery results as in ACE-XQ. The approach in [23] is extended to handle a very limited subset of XQuery. Their caching system has no support for incremental maintenance of materialized views. Also, in [34] the term ”incremental maintenance” is used in the context of expiring unused cached data rather than maintaining the materialized views in the presence of updates to the base data.

The algebraic and algorithmic approaches require and assume availability of different amount of information during view maintenance. Hence the size of the problems addressed by these approaches are different. The algorithmic approaches tend to be view self maintenance approaches wherein view maintenance is possible without access to the base documents. The algebraic approach requires the existence of a formal algebra, and access to the base documents when performing incremental view maintenance.

Chapter 6

Conclusion

Conclusion

We have presented an efficient and novel method for self maintenance of materialized XML views using a Query Containment approach. We investigated this problem in the context of a semantic caching system called ACE-XQ developed at the DSRG labs in WPI. We have exploited and extended the query containment and re-writing ideas developed for ACE-XQ to solve this problem. Effective cache coherence can lead to major performance improvements in a caching system. Our thesis work has demonstrated that it is easy to develop and incorporate an effective cache coherence strategy in a semantic caching system.

6.1 Results and Contributions of this Thesis

The primary contributions of this thesis are:

- We have proposed a unique approach to the self maintenance of materialized XML views using a query containment approach. To the best of our knowledge, we are the first to investigate and use a query containment approach for XML view maintenance.

- We have proposed a view maintenance approach that relies solely on the view and update query definitions, and data in the materialized view itself. Other view maintenance strategies either require knowledge of schema of the base document, assume the presence of unique object identifiers [1, 37], or assume access to the base data [9]. While the amount of information available during view maintenance has an impact on the cases that can be incrementally maintained, the simplicity of our approach does not restrict its usability. In practice, a caching system can easily get around some of these restrictions by having two layers of views where-in one layer consists of simpler, self-maintainable materialized views and the other layer, on top of the first one has complex non-materialized views.

- We believe that the concept of pattern matching used in our approach can be effectively used for more complicated update and view queries. This pattern matching can help detect non-relevant queries sooner in the query analysis process and reduce the number of update queries for further analysis. This filtering of non-relevant update queries based entirely on pattern matching of queries can have very significant benefits for the average case performance of a caching system like ACE-XQ.

6.2 Ideas for Future Work

Future work using our approach for view maintenance can be in two different directions: One is to improve the view maintenance approach itself by handling more cases. The other is to make it more useful by considering the practical problems involved like update notifications, transactional updates, batched updates etc.

Currently we support the XPath fragment containing any two of ([], //, *) operators. Our work eventually relies of the XPath containment and recent work in this area [14] suggests that a wider fragment of XPath is amenable for containment. We can use these results from the literature to allow more types of queries. Moreover, we only consider queries on single documents without joins, aggregates, or user defined functions. Extending our approach to handle these queries is an important future step. Another unique feature of XML is that it is order sensitive. In our approach we have neglected the order of elements in results. We can investigate approaches like LexKeys in the context of ACE-XQ for order sensitive view maintenance.

Since each variable node in a VarTree is derived from its parent, it is possible to re-use the XPath containment results of the parent node and to do only incremental XPath containment checking for child nodes. Such an incremental algorithm can greatly improve the performance of the analysis procedure.

Our current implementation performs analysis of one update at a time. If several updates occur simultaneously, some of the updates might invalidate some later updates. If the system can batch the updates, it may detect such invalid updates which will lead to improved average performance of the system. Another assumption in this implementation is that the updates to the base document and to the materialized view are not within the context of a transaction. Although transactional XML systems are not yet popular, they will be required in applications where high data integrity is essential. Exploring view maintenance in the context of transactional XML systems is also another avenue for future work in this area.

Bibliography

[1] S. Abiteboul. On views and xml. In *PODS, Philadelphia, Pennsylvania*, pages 1–9, May 1999.

[2] S. Abiteboul, J. McHugh, M. Rys, V. Vassalos, and J. Wiener. Incremental Maintenance for Materialized Views over Semistructured Data. In *Int. Conference on VLDB*, pages 38–49, August 1998.

[3] M. J. Carey, J. Kiernan, J. Shanmugasundaram, E. J. Shekita, and S. N. Subramanian. XPERANTO: Middleware for Publishing Object-Relational Data as XML Documents. In *The VLDB Journal*, pages 646–648, 2000.

[4] L. Chen. A Semantic Caching System for XML Queries. Dissertation, WPI, 2003.

[5] L. Chen and E. A. Rundensteiner. Aggregate Path Index for Incremental Web View Maintenance. In *The 2nd Int. Workshop on Advanced Issues of E-Commerce and Web-based Information Systems, San Jose*, pages 231–238, 2000.

[6] L. Chen and E. A. Rundensteiner. Xquery containment in presence of variable binding dependencies. In *WWW '05: Proceedings of the 14th international conference on World Wide Web*, pages 288–297, New York, NY, USA, 2005. ACM Press.

[7] L. S. Colby, T. Griffin, L. Libkin, I. S. Mumick, and H. Trickey. Algorithms for Deferred View Maintenance. In *SIGMOD, Montreal, Canada*, pages 469–480, June 1996.

[8] A. Deutsch, Y. Papakonstantinou, and Y. Xu. The NEXT Logical Framework for XQuery. In *VLDB, Toronto, Canada*, pages 168–179, 2004.

[9] K. Dimitrova, M. El-Sayed, and E. A. Rundensteiner. Order-sensitive view maintenance of materialized xquery views. In *ER*, 2003.

[10] I. Distributed Management Task Force. *Common Information Model (CIM) Infrastructure Specification*, 2005.

[11] M. EL-Sayed, L. Wang, L. Ding, and E. A. Rundensteiner. An Algebraic Approach for Incremental Maintenance of Materialized XQuery. Technical report, Computer Science Department, WPI, 2002. in progress.

[12] M. EL-Sayed, L. Wang, L. Ding, and E. A. Rundensteiner. An Algebraic Approach for Incremental Maintenance of Materialized XQuery Views. In *WIDM*, 2002.

[13] M. Fernandez, W. Tan, and D. Suciu. SilkRoute: Trading between Relations and XML. http://www.www9.org/w9cdrom/202/202.html, May 2000.

[14] F.Neven and T.Schwentick. XPath Containment in The Presence of Disjunction, DTDs, and Variables. In *ICDT, Siena, Italy*, pages 315–329, 2003.

[15] G. Miklau and D. Suciu. Containment and Equivalence for an XPath Fragment. In *PODS, Madison, Wisconsin*, pages 65–76, June 2002.

[16] A. Gupta and I. S. Mumick. Maintenance of Materialized Views: Problems, Techniques, and Applications. In *Bulletin of the Technical Committee on Data Engineering, 18(2)*, pages 3–18, June 1995.

[17] G. Huck and I. Macherius. GMD-IPSI XQL Engine. http://xml.darmstadt.gmd.de/xql/, 1999.

[18] B. Kane, H. Su, and E. R. Rundensteiner. Consistently Updating XML Documents Using Incremental Constraint Check Queries. In *WIDM, McLean, Virginia*, pages 1–8, November 2002.

[19] A. Kawaguchi, D. F. Lieuwen, I. S. Mumick, and K. A. Ross. Implementing Incremental View Maintenance in Nested Data Models. In *Workshop on Database Programming Languages*, pages 202–221, August 1997.

[20] H. A. Kuno and E. A. Rundensteiner. Incremental Maintenance of Materialized Object-Oriented Views in MultiView: Strategies and Performance Evaluation. *IEEE Transaction on Data and Knowledge Engineering*, 10(5):768–792, Sep./Oct. 1998.

[21] H. Liefke and S. B. Davidson. View maintenance for hierarchical semistructured data. In *Data Warehousing and Knowledge Discovery*, pages 114–125, 2000.

[22] M. M. Maged El-Sayed, Elke A. Rundensteiner. Incremental fusion of xml fragments through semantic identifiers. In *9th International Database Engineering and Application Symposium (IDEAS'05)*, pages 369–378, 2005.

[23] B. Mandhani and D. Suciu. Query caching and view selection for xml databases. In *VLDB*, pages 469–480, 2005.

[24] I. Manolescu, D. Florescu, and D. Kossmann. Answering XML Queries on Heterogeneous Data Sources. In *VLDB, Roma, Italy*, pages 241–250, 2001.

[25] C. Ruemmler and J. Wilkes. An introduction to disk drive modeling. *computer*, 27(3):17–28, March 1994.

[26] A. Sahuguet. KWEELT, the Making-of: Mistakes Made and Lessons Learned. Technical report, University of Pennsylvania, Nov. 2000.

[27] A. Schmidt, F. Waas, R. Busse, M. Carey, D. Florescu, M. Kersten, and I. Manolescu. Xmark – The XML-Benchmark Project. http://www.xml-benchmark.org, April 2001.

[28] I. Tatarinov, Z. G. Ives, A. Y. Halevy, and D. S. Weld. Updating xml. In *SIGMOD*, pages 413–424, 2001.

[29] W3C. XML^{TM} . http://www.w3.org/XML, 1998.

[30] W3C. XML Path Language (XPath)Version 1.0. W3C Recommendation. http://www.w3.org/TR/xpath.html, March 2000.

[31] W3C. *SOAP Version 1.2 Part 1: Messaging Framework*, 2003.

[32] W3C. XQuery 1.0: An XML Query Language. http://www.w3.org/TR/xquery/, May 2003.

[33] W3C. XQuery 1.0 and XPath 2.0 Formal Semantics. http://www.w3.org/TR/query-semantics/, May 2003.

[34] W. Xu. The framework of an xml semantic caching system. In *WebDB*, pages 127–132, 2005.

[35] S. A. Yahia, S. Cho, L. S. Lakshmanan, and D. Srivastava. Minimization of Tree Pattern Queries. In *SIGMOD, Santa Barbara, CA*, pages 315–331, 2001.

[36] Y. Zhuge, H. García-Molina, J. Hammer, and J. Widom. View maintenance in a warehousing environment. In *Proceedings of the ACM SIGMOD, San Jose*, pages 316–327, May 1995.

[37] Y. Zhuge and H. G. Molina. Graph Structured Views and Their Incremental Maintenance. In *Proceedings of the 14th International Conference on Data Engineering, Orlando, Florida*, pages 116–125, February 1998.