XQuery Containment in Presence of Variable Binding Dependencies

Li Chen
San Diego Supercomputer Center
9500 Gilman Drive, La Jolla, CA 92093
lichen@sdsc.edu

Elke A. Rundensteiner
CS Department, Worcester Polytechnic Institute
Worcester, MA 01609
rundenst@cs.wpi.edu

ABSTRACT

Semantic caching is an important technology for improving the response time of future user queries specified over remote servers. This paper deals with the fundamental query containment problem in an XQuery-based semantic caching system. To our best knowledge, the impact of subtle differences in XQuery semantics caused by different ways of specifying variables on query containment has not yet been studied. We introduce the concept of *variable binding dependencies* for representing the hierarchical element dependencies preserved by an XQuery. We analyze the problem of XQuery containment in the presence of such dependencies. We propose a containment mapping technique for nested XQuery in presence of variable binding dependencies. The implication of the nested block structure on XQuery containment is also considered. We mention the performance gains achieved by a semantic caching system we build based on the proposed technique.

Categories and Subject Descriptors

F.3 [**Theory of Computation**]: Logics and Meanings of Programs; I.1.1 [**Computing Methodologies**]: Expressions and Their Representation; H.2.8 [**Information Systems**]: Database Applications

General Terms

XQuery containment mapping algorithm and theory

Keywords

XQuery Containment, variable binding dependency

1. INTRODUCTION

Due to its fundamental role in many database applications such as query optimization and information integration [16], the problem of query containment has received considerable attention over the past few decades. With the initial focus on relational queries, researchers have recently begun to study the containment problem for various fragments of XPath [13, 1, 22, 12] and XQuery [9, 18, 10]. The key technique of containment mapping for relational queries [4] has been extended in the new contexts to derive mappings between navigation pattern trees and nested XQuery constructs. It has been commonly recognized that extended containment mapping is central for minimizing XML queries [23, 9], and for reformulating queries in a mediator system [9] or a peer-to-peer environment [18].

WWW 2005, May 10-14, 2005, Chiba, Japan.
ACM 1-59593-046-9/05/0005.

1.1 Motivation

This work is motivated by the promising application of semantic caching for answering XML queries using cached XML views [6, 8]. The idea of semantic caching is that the (mobile) client maintains both the semantic descriptions and associated answers of previous queries in its cache, in the hope of being able to reuse them to speed up the processing of subsequent queries.

An XQuery-based semantic caching system named **ACE-XQ** has been proposed [6, 7] for facilitating XQuery processing in the Web environment. The main techniques exploited by ACE-XQ include the containment mapping approach for nested XQuery, XQuery rewriting, and a multi-granularity replacement strategy. With [8] focussed on the proposed replacement strategy and the cache performance evaluation, we introduce, in this paper, the fundamental query containment technique underlining ACE-XQ which is the first comprehensive practical semantic cache solution for handling nested conjunctive XQuery.

1.2 The Related Work

In the XML setting, extensive research has focussed on the query containment problem for regular path expressions on general cyclic graph databases [3], tree pattern queries and XPath queries over XML data [13, 1, 22, 12]. Especially the containment problem for XPath and tree pattern queries has attracted a lot of attention recently due to the fundamental role they play in many XML query languages.

Different fragments of XPath have been targeted by different works. A well recognized core XPath fragment includes child axis '/', descendant axis "//", branching "[ ]", and wildcard '*'. It is shown in [13] that query containment for this fragment, denoted $XP^{\{*,//,[\,]\}}$, is coNP-complete. If any of the three constructs "//", "[ ]", and '*' is dropped, query containment is PTIME. The essence of their containment mapping technique is the polynomial-time *tree homomorphism* algorithm[1], which serves as a sufficient but not necessary condition for containment of $XP^{\{*,//,[\,]\}}$ in general. On the other hand, if tag variables and equality testing are allowed, query containment is NP-complete. The complexity increases to Π_2^p with disjunctions added. We refer

[1] *Tree homomorphism* and *tree embedding* are exchangeable.

the readers to [1, 22, 12] for discussions of the containment complexity results under different XPath fragments.

However, research on the containment problem for XQuery is still in its infancy. Besides using XPath expressions as the navigation mechanism, XQuery also employs other query constructs such as FLWR expressions and the nesting of query blocks. These features make XQuery more expressive than XPath. On the other hand, they also impose new difficulties on the containment problem. Specifically, difficulties arise since an XQuery cannot simply be represented by a navigation tree pattern. Hence containment mapping based on tree homomorphism alone is no longer sufficient for determining XQuery containment.

To our best knowledge, the containment of nested XQuery has so far been studied only in [9], [18], and [10]. [9] exploits XQuery containment for query optimization. It utilizes containment mapping for identifying redundant navigation patterns in a query and later for collapsing them to minimize the query. In [18], the containment of nested XQuery is researched for the purpose of rewriting queries posted on one peer to be answered by another peer. [10] studies the complexity of the problem regarding completeness.

Targeting different goals, these three works exploit different approaches. The containment mapping technique proposed in [9] essentially extends tree homomorphism between navigation patterns with additional requirements for mapping the equality-based *where-conditions*, *groupby_id* and *groupby_value* variables. In [18], two types of mappings, i.e., a *query-head embedding* $E_{head}(Q_1, Q_2)$ and a *query-body embedding* $E_{body}(Q_2, Q_1)$, are employed as the sufficient conditions for deriving $Q_1 \sqsubseteq Q_2$ (assuming Q_1 and Q_2 are two nested XQueries). E_{head} embeds the block structure of Q_1 into that of Q_2 while E_{body} embeds the navigation pattern of Q_2 into that of Q_1. In [10], containment of nested XQuery is defined based on *XML instance containment*. The theoretical complexity result for methods that ensure completeness is established.

Among these three works, [10] presents an approach that guarantees completeness (i.e., no false negative answers). In answering-queries-using-views scenarios, it is commonly considered more crucial to guarantee the soundness while the completeness is often ignored to avoid the high complexity. For example, the containment of nested XQuery in general is coNEXPTIME when ensuring completeness [10].

In contrast, [9] and [18] attempt to provide more practical containment mapping techniques by extending tree homomorphism with additional mapping conditions. In [9], a technique is proposed for identifying redundant navigation within *one* query. It considers the mapping of equality-based where-conditions and that of variables distinguished by the set or bag semantics they each represent. However, all return expressions are considered as black-box functions and ignored in the containment mapping process. This containment mapping technique is hence a not suitable foundation for determining the containment relationship between *two* queries. This is obvious considering the fact that whether the direct bindings of variable v or subelements obtained from further navigation of v's bindings are returned does make a major difference in the query result.

Furthermore, neither of the two techniques considers the effect of dependencies among variable bindings on the query result and consequently on the containment result. In Section 1.3, we give examples of subtle differences in XQuery semantics caused by different dependencies among the specified variables. Since these two techniques have failed to address the critical effect of such differences on the query containment result, we propose our containment mapping approach which provides sufficient mapping conditions for correctly deriving the containment decision.

1.3 Problem Analysis

In this work, we target the containment problem for nested XQuery. We consider a core fragment of XQuery that allows nested blocks, conjunctive equality-based conditions, set and bag semantics. Disjunctions, negations, universal quantifier and tag variables are not considered. This XQuery fragment is the same as that being studied in [9]. [18] and [10] study a subset of this fragment as they exclude the bag semantics.

```
for $t in document(``bib.xml")//book/title,
   $a in document(``bib.xml")//book/author
return <pairQ1> $t, $a
         </pairQ1>
                                          Q1
```

```
for $b in document(..)//book,
    $t in $b/title, $a in $b/author
where some $p in $b/price
          satisfies $p=30
return <pairQ2> $t,$a/last
          </pairQ2>                  Q2
```

```
for $b in document(..)//book
return <pairQ3>
     {for $t in $b/title, $a in $b/author
      return $t, $a}
         </pairQ3>                      Q3
```

```
for $b in document(..)//book
return <pairQ4>
    {for $t  in $b/title return $t},
    {for $a in $b/author return $a}
          </pairQ4>                 Q4
```

Figure 1: Example Queries

Now let us consider the example queries in Figure 1. All four queries Q_i (i=1..4) specify $\$t$ and $\$a$ and return their bindings in the results. Suppose the input document *bib.xml* is shown at the left top corner in Figure 2, we can see that their results R_{Qi} (i=1..4) (also shown in Figure 2) are all different due to the subtle differences in their variable specifications and nested block structures. Suppose that the DTD for *bib.xml* specifies $<!Element\, book(title, author*, publisher?, price?)>$. R_{Q1} contains six *title* and *author* pairs derived from all combinations of the \$t and \$a bindings document-wide regardless of whether the paired *title* and *author* elements belong to the same book. In contrast, the \$t and \$a bindings in Q_2 are specified based on \$b. Therefore, the *title* and *author* elements corresponding to different *book* parents do not appear in the same pair in R_{Q2}. For example, $t2$ is paired with $a1$ and $a2$ but not with $a3$ in R_{Q2}.

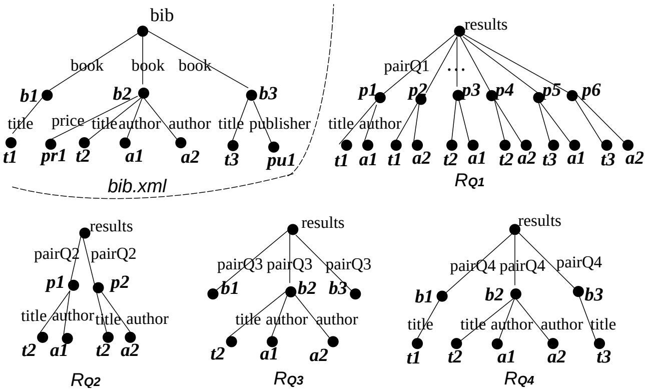

Figure 2: bib.xml and Example Query Results

The differences in the structure of R_{Q1} versus that of R_{Q2} can be intuitively explained by the differences in specifying variable dependencies in Q_1 and Q_2. That is, the variables $\$t$ in Q_2 (denoted by $\$t_{Q2}$) and $\$a_{Q2}$ are defined based on $\$b_{Q2}$, while $\$t_{Q1}$ and $\$a_{Q1}$ are based on $\$r$ (i.e., the default root variable bound to the root element of *document("bib.xml")*).

We first explain the effect of variable dependencies on the resulting query result for Q_2. When constructing the result R_{Q2}, since $\$t_{Q2}$ and $\$a_{Q2}$ are defined in the same query block, the corresponding new element $\langle pairQ2 \rangle$ is produced for each tuple in the cartesian product of the bindings of $\$t_{Q2}$ and $\$a_{Q2}$. Due to the way how $\$t_{Q2}$ and $\$a_{Q2}$ are specified, the bindings of $\$t_{Q2}$ and $\$a_{Q2}$ derived from the same binding of $\$b_{Q2}$ preserve the sibling $\langle title \rangle$–$\langle author \rangle$ element associations under the same parent *book* element. Such hierarchical data dependencies in the source XML are preserved in the intermediate variable bindings based on which the query result is constructed. In this case, each $pairQ2$ element in R_{Q2} combines bindings of $\$t_{Q2}$ and bindings of $\$a_{Q2}$ only if they share the same parent binding of $\$b_{Q2}$. In contrast, the sibling $\langle title \rangle - \langle author \rangle$ associations are not kept in the bindings of $\$t_{Q1}$ and $\$a_{Q1}$. Q_1 hence produces $\langle pairQ1 \rangle$ elements based on the cartesian product of all the bindings of $\$t$ and $\$a$ regardless of their respective parent *book* elements. Q_2 hence preserves a finer hierarchy of element dependencies among its intermediate variable bindings than Q_1 does.

We now analyze the effect of such preserved dependencies on the containment result. Suppose that the containment mapping technique proposed in [18] is applied to Q_1 and Q_2 in Figure 2. $Q_2 \sqsubseteq Q_1$ would be derived since both $E_{head}(Q_2, Q_1)$ and $E_{body}(Q_1, Q_2)$ can be established as illustrated in Figure 3[2]. To derive $Q_2 \sqsubseteq Q_1$, this approach utilizes not only the navigation pattern based mapping represented by E_{body}, but also E_{head} for checking if the variables returned by Q_2 are a subset of those returned by Q_1. However, whether such dependencies among variable bindings influence the query containment result has not been studied in either [18] or [9].

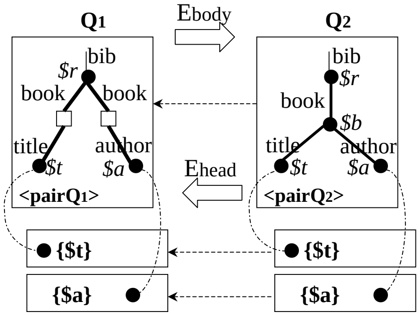

Figure 3: Illustration of Containment Mapping via E_{head} and E_{body}

Assume Q_2 is answered using R_{Q1} based on the containment result $Q_2 \sqsubseteq Q_1$. Then there is no way to re-group the returned bindings of $\$t$ and $\$a$ in R_{Q1} by their respective *book* parent elements as required by Q_2. Ignoring this requirement, the produced result of Q_2 would contain superfluous pairs, namely, $t1-a1$, $t1-a2$, $t3-a1$ and $t3-a2$.

1.4 Our Contributions

First, we address the problem of producing superfluous answers based on the query containment result when ignoring the effect of variable binding dependencies in the containment mapping process. Correspondingly we identify some important concepts and their connections, as illustrated in Figure 4, to this problem.

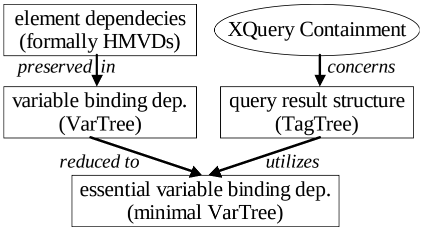

Figure 4: Connection between Preserving of Element Dependencies and XQuery Containment

The left hand side flow illustrates the preservation of data dependencies in the source XML in the intermediate bindings via the specification of variables. The right hand side flow represents the fact that XQuery containment needs to take the query result structure constructed based on the binding dependencies into consideration. Terms enclosed in the parentheses in Figure 4 will be introduced in Section 2.

Also, we realize that not all the intermediate binding dependencies preserved by a query are necessarily utilized in constructing the final result. Thus we call a subset of variable binding dependencies being utilized the essential ones via which both flows are connected.

Second, based on our problem analysis, we propose a containment mapping technique that considers the containment of the utilized binding dependencies in the query result. For this, we first decompose the input query and represent the two parts of its semantics, i.e., variable binding and result construction, by respective tree structures. Then we identify the binding dependencies that are preserved by the former and utilized by the latter. We call it variable minimization. Next we propose to employ three types of containment mappings for deriving the containment decision.

In sum, we will show that our containment mapping approach is more comprehensive than the prior works [9, 18, 10] in that it deals with the effect of variable binding dependencies on the query containment result. In other words, it avoids deriving the query containment decision which may lead to producing superfluous answers for the contained query by using the result of the containing query. Like [9] and [18], our approach does not necessarily ensure completeness.

1.5 Paper Outline

The rest of the paper is organized as follows. In Section 2, we define the problem of XQuery containment in the presence of variable binding dependencies. Section 3 gives the overview of our approach. We describe the pre-step of query decomposition and minimization in Section 4. This is followed by our containment mapping technique in Section 5. We show the query performance gains achieved by apply-

[2] As explained more in detail in [18], $E_{head}(Q_2, Q_1)$ embeds the nested block structure of Q_2 into that of Q_2. The dashed arrows denote the mappings between blocks within which the corresponding returned variables match. $E_{body}(Q_1, Q_2)$ embeds the navigation patterns (denoted by the bold tree edges) specified in Q_1 into those in Q_2. $E_{body}(Q_2, Q_1)$ but not $E_{head}(Q_1, Q_2)$ can be established. Hence $Q_1 \not\sqsubseteq Q_2$.

ing the proposed technique in a semantic caching system in Section 6 and conclude in Section 7.

2. PROBLEM DEFINITION

In this section, we first introduce the notion of *hierarchical multivalued dependencies (HMVDs)* which represent a typical type of data dependencies in the source XML. Also, we define *variable binding dependencies* as the HMVDs being preserved by the query in the intermediate variable bindings. We then define the problem of nested XQuery containment in the presence of variable binding dependencies.

2.1 Hierarchical Multivalued Dependencies

It has been recently recognized that studying the extension of the traditional integrity constraints in the XML setting is both theoretically and practically meaningful. Several classes of integrity constraints including key constraints, path constraints, functional constraints, and inclusion constraints have been defined for XML [11]. The more advanced constraints such as the multivalued dependencies (aka tuple generating dependencies) have also been studied in [19, 2] with the goal to develop a normalization theory for XML and in [15] for mapping XML DTDs to relational schemas.

XPath containment in the presence of DTD constraints such as sibling constraints and functional constraints has been investigated in [11]. The semantics of an XPath query can be captured by a unary pattern tree in which only one node has its bindings returned as the result while others are matched but not returned. However in XQuery, even a single *for* clause may specify multiple variables which correspond to an n-ary ($n \geq 1$) pattern tree. This is where the challenges arise for XQuery containment.

Let us first analyze the semantics of a single-block XQuery for the sake of simplicity. In a single-block XQuery that utilizes a FLWR expression, the *return* clause is invoked for all the cartesian product combinations of the variable bindings produced by the *for* clause. These combinations are determined based on how variables are defined based on others. As far as we know, no research has studied the implication of such dependencies on XQuery containment. This is the task of our work.

Definition 2.1. *Given a DTD, suppose ε is a set of binary edge relations between element type e and its children element types, each labeled with the corresponding cardinality relationship 1, ?, * or +* [3]*. For any two descendant element types x and y of e, if either x or y has a multiple cardinality relationship (i.e., * or +) with e, then we call the dependency among their corresponding elements in a conforming XML a **hierarchical multivalued dependency (HMVD)**, denoted $e \twoheadrightarrow x|y$.*

Recall that the notion of multivalued dependency (MVD) in relational databases defines that if a relation has two or more multivalued independent attributes (e.g., x and y), every value of one attribute (e.g., x) must be repeated with every value of the other attribute (e.g., y). HMVD extends MVD in the sense that e, x and y are not attributes in a relation but element types in a DTD. If e, x and y are mapped to a 3-column relation and their bindings are unnested, then in each partition with an e binding, every x binding needs to be repeated with every y binding.

2.2 Variable Binding Dependency

For an XML document D, the dependencies among its elements which have multiple cardinality relationships with their respective parents can be represented by HMVDs. A query imposed against D specifies a subset of HMVDs (direct or derived) to be preserved by its variable bindings.

Definition 2.2. *Suppose a given query defines variable v_j based on v_i, e.g., for v_j in $v_i(/|//)p_j$, where p_j is the relative XPath expression used for deriving v_j's bindings from each binding of v_i. We call this dependency of v_j's bindings on their respective v_i bindings a **variable binding dependency**, denoted $v_i \overset{p_j}{\rhd} v_j$.*

For example, $\$b \overset{/title}{\rhd} \$t$ and $\$b \overset{/author}{\rhd} \$a$ hold for $Q_{2\sim 4}$ in Figure 2. They all specify their corresponding $\$t$ and $\$a$ based on $\$b$.

The variable binding dependency relationship satisfies:

$$v_i \overset{p_j}{\rhd} v_j,\ \ v_j \overset{p_k}{\rhd} v_k \ \ \Rightarrow \ \ v_i \overset{p_{jk}}{\rhd *} v_k \ (\textbf{\textit{transitivity rule}}),$$

where p_{jk} is the path expression obtained by concatenating p_j and p_k, and $\rhd *$ denotes an induced variable binding dependency. Given an XQuery, the direct variable binding dependencies extracted from it compose a base dependency set from which the non-direct dependencies can be derived inductively.

For example, Q_2, Q_3 and Q_4 in Figure 2 define $\$b$ via an absolute path expression *//book* from the root of the source XML *"bib.xml"*. Suppose variable $\$r$ is used as a default *root variable* to be bound with the root element, then $\$r \overset{//book}{\rhd} \$b$. Also since $\$b \overset{/title}{\rhd} \$t$ and $\$b \overset{/author}{\rhd} \$a$, we can derive $\$r \overset{//book/title}{\rhd *} \$t$ and $\$r \overset{//book/author}{\rhd *} \$a$. In contrast, Q_1 in Figure 2 directly defines $\$t$ and $\$a$ based on the root variable $\$r$. Hence it has $\$r \overset{//book/title}{\rhd} \$t$ and $\$r \overset{//book/author}{\rhd} \$a$.

2.3 HMVD and XQuery Containment

To tackle XQuery containment in the presence of variable binding dependencies, we cannot solely utilize tree homomorphism between the two respective navigation pattern trees. Additional conditions need to be asserted in the containment mapping process to deal with the effect of variable binding dependencies on the query semantics.

Let us consider the containment relationship between Q_1 and Q_2 in Figure 2 again. A tree embedding of the pattern tree of Q_1 into that of Q_2 exists, as illustrated by $E_{body}(Q_1, Q_2)$ in Figure 3. As described before, if we were to use R_{Q1} (see Figure 2) to answer Q_2 according to $Q_2 \sqsubseteq Q_1$, then it will result in the superfluous answer pairs $t1$–$a1$, $t1$–$a2$, $t3$–$a1$ and $t3$–$a2$. With the new concepts introduced in this section, we can see that this is because the HMVD $\$b \twoheadrightarrow \$t|\$a$ is required by Q_2 but not preserved by Q_1.

Suppose that $Vars(Q)$ and $Rets(Q)$ are the defined variables and the returned expressions in a query Q respectively. All the variables occurring in $Rets(Q)$ must be defined in $Vars(Q)$ for Q to be safe. On the other hand, variables occurring in $Rets(Q)$ may be a subset of $Vars(Q)$. That is, not all the variable binding dependencies are utilized in the query result. To determine query containment, we need to reason about not only the containment of the returned

[3] 1, ?, * and + respectively represent the 1-1, 1-(0,1), 1-(0,m) and 1-(1,m) ($m \geq 1$) mappings.

bindings due to $Rets(Q)$, but also the containment of the utilized variable binding dependencies due to both $Vars(Q)$ and $Rets(Q)$. Correspondingly, we now define XQuery containment in the presence of variable binding dependencies.

Definition 2.3. *Let Q_1, Q_2 be two XQueries. $Q_1 \sqsubseteq Q_2$ if and only if: **1)** there exists a containment mapping from Ret(Q_1) to Ret(Q_2), and **2)** the HMVDs preserved in $Vars(Q_1)$ and utilized by Ret(Q_1) are subsumed by those preserved in $Vars(Q_2)$ and utilized by Ret(Q_2).*

For example, the HMVD $\$b \twoheadrightarrow \$t|\$a$ is reflected in R_{Q2} in Figure 2 but not in R_{Q1}. That is, the bindings of $\$t_{Q1}$ and $\$a_{Q1}$ are paired document-wise in R_{Q1}, whereas those of $\$t_{Q2}$ and $\$a_{Q2}$ are grouped by their common *book* elements in R_{Q2}. The former pairs can be derived from the latter by pairing all the $\$t_{Q1}$ bindings with $\$a_{Q1}$ bindings regardless of if they came from the same *book* parents. However, there is no way to recover the dependences of $\$t_{Q1}$ and $\$a_{Q1}$ bindings on their common *book* elements as required by Q_2.

3. OVERVIEW OF OUR APPROACH

The main idea of our 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 our approach are depicted in Figure 5.

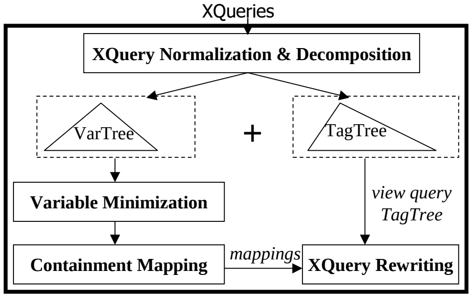

Figure 5: Containment Checking Flow

- *XQuery decomposition.* We separate the variable definition part from the result construction part and represent 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 [23], as will be explained later. The latter tree (i.e., TagTree) is used to represent the result construction template. The TagTree also indicates how the preserved HMVDs are utilized in the result construction.
- *Variable minimization.* We identify the variables that are neither directly nor indirectly utilized in the result construction and degrade them to navigation steps. This way, we derive a minimal set of variable binding dependencies for which we conduct the containment checking. This is a critical step for ensuring the correctness of the containment result.
- *Containment mapping.* We conduct three types of containment mappings. First, we perform the minimal VarTree embedding to check the containment of the utilized HMVDs. Second, we check the tree embedding relationship between the navigation patterns. Lastly, we 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 $p' = \mathcal{M}_c(p) \circ \mathcal{M}_t$, 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 $\mathcal{M}_t$. We skip the details of query rewriting in this paper.

4. DECOMPOSITION AND MINIMIZATION

4.1 XQuery Decomposition

The purpose of query decomposition is to separate the semantics of variable bindings from that of result construction. However, the semantic distinction is sometimes not very easily extracted from the surface syntax. For example, not necessarily all expressions in return clauses represent the return construction semantics. Due to the flexibility in composing a nested XQuery, FLWR expressions may be nested within a for clause, e.g., *for* v_2 *in (for* v_1 *in* e_1 *return* e_2*) return* e_3. In this case, e_2 in the nested $return$ clause does not result in returning its bindings in the ultimate query result but only serves for specifying v_2's binding.

Therefore, we need to first normalize the query to derive a form based on which this semantic distinction is made easy. Then we represent the two semantics respectively using two tree structures, which are connected via variable bindings.

4.1.1 The Normalization Rules in Use

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 [21, 17, 9] 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 [9]. 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[4] expressions.

4.1.2 Decomposition into VarTree and TagTree

Definition 4.1. *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** $v \in V$. Each*

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

dependency $v_i \overset{p_j}{\rhd} v_j$ *corresponds to an edge* $e = (v_i, v_j) \in E$ *labeled* $p_j \in L$. *We refer to e the* ***derivation edge*** *of* v_i.

The VarTree is different from the pattern tree concept referred to in other research [23]. 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.

DEFINITION 4.2. *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*** $n \in N$ *is a quadruple* $[\bar{V}, \bar{C}, \bar{R}, \bar{T}]$ *and each edge* $a = (n_i, n_j) \in A$ *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, return 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* $\langle t1\rangle\langle t2\rangle \ldots \langle tn\rangle$. *This means that the returns of* $\bar{R}$ *will be enclosed by an empty tag,* $\langle t1\rangle$ *and* $\langle /t1\rangle$, *or* $\langle t1\rangle\langle t2\rangle \ldots \langle tn\rangle$ *and* $\langle /tn\rangle \ldots \langle /t2\rangle\langle /t1\rangle$.

We now extend the VarTree structure with a few more features. Given the TagTree TT_Q of a query Q, we get each return expression v/p_m in a $\bar{R}$ of TT_Q and correspondingly attach to the *var* node for v in the VarTree VT_Q a leaf node (also referred to as **ret** node). Each *ret* node represents the corresponding return expression. To distinguish *var* nodes from *ret* nodes, we use solid circles to denote the former and use hollow circles for the latter.

The second extension is to shift the constraints and conditions in the $\bar{C}$'s of TT_Q to be represented in VT_Q. Specifically, if the constraint pattern represented by the XPath expressions (with or without variables defined in their respective where clauses[5]) is derived from v, or the equality-based conditions are affiliated to where-variables that are dependent of v, then we move them in the filter "[ ]" of the XPath expression p which labels the derivation edge of v in VT_Q. Intuitively, this can be done because these constraints and conditions are, in a sense, analogous to the relational *selection* operations. They hence can be pushed to be executed in the navigation pattern matching stage for deriving variable bindings.

For example, the extended VarTrees and TagTrees of example Q_1 and Q_2 are depicted in Figure 6 respectively. Note that the where-condition *"some \$p in \$b/price satisfies \$p=30"* in the $\bar{C}$ of the bottom block in TT_{Q2} is serialized into *"price=30"* and then moved in "[ ]" as the filter expression for defining \$$b$ in VT_{Q2}.

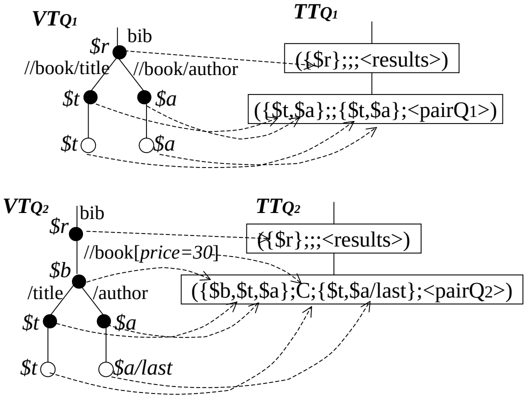

Figure 6: VarTrees and TagTrees of Q_1 and Q_2

However, we must carry out this VarTree extending process with caution. That is, the shifting of return expressions in $\bar{R}$ and where-conditions in $\bar{C}$ would not change the query semantics only if the to-be-attached *var* node v is defined in the same block where $\bar{R}$ and $\bar{C}$ are specified. Some return expressions in $\bar{R}$ and where-conditions in $\bar{C}$ refer to variables that are defined in ancestor blocks. By moving them up along the nested block hierarchy to be attached to their referring variables, more or fewer bindings than desired may be returned. For example, suppose the example query Q_3 in Figure 1 also specifies the where-condition *"some \$p in \$b/price satisfies \$p=30"*, however in the inner block. Then attaching *"[price=30]"* to the definition expression of \$$b$ in the outer block may cause generating fewer <$pairQ$3> elements due to the push-up condition. We hence leave such return expressions and where-conditions in their original block nodes in TT_Q.

The VarTree with these extensions is comprehensive enough to also represent the to-be-returned bindings and the effect of where-conditions on variable bindings. It is also noteworthy that the VarTree and TagTree of a query are connected via variables. In particular, all variables in $\bar{V}$'s and those referred to in $\bar{R}$'s in the TagTree must be present as *var* nodes in the VarTree for the query to be safe.

4.2 Use-based Variable Minimization

We explained earlier that the VarTree of a query is a nested tree with navigation patterns encapsulated in its derivation edges. On one hand, the query semantics stays the same if we fully expand the VarTree by unnesting all the encapsulated navigation patterns and by naming each node in them with a variable. On the other hand, it is also possible not to affect the query semantics by degrading some variables into navigation pattern nodes to be encapsulated in derivation edges. We call the latter a *variable minimization* process since the number of *var* nodes is reduced (however with more complex navigation patterns encapsulated) and the VarTree structure seems minimized. A variable can be minimized without affecting the query semantics only if it does not participate in preserving any HMVD that is uti-

[5] *Where-variables* refer to variables defined in *where* clauses, while *for-variables* are those defined in *for* clauses. Unless indicated otherwise, "variable" means a for-variable. A where-variable can be removed since its life scope is refrained within the local where clause.

lized in the result construction, nor serve in any way as a constraint context (will be explained later) for the return expressions.

Our goal here is to explore the opportunities for variable minimization to obtain the minimal VarTree (i.e., no further minimization is possible). This is critical since the later containment checking of utilized HMVDs can be based on the derived minimal VarTrees of two given queries.

Definition 4.3. *Given an XQuery Q, suppose D is the source XML and v is a variable defined in Q. If by substituting all occurrences of v with v's definition, Q's result will not change for any XML data instance that conforms to the same DTD as D, then we say v is* ***nonessential****. Otherwise v is* ***essential****.*

Now we provide practical criteria for distinguishing essential variables from non-essential ones based on their uses.

Explicit vs. Implicit Uses. A variable v may either be used for defining another variable or in a return expression. We call the former case a *Var* use of v and the latter a *Ret* use of v. Both are referred to as *explicit* uses of v in general, regardless of where it is used (i.e., either in the local block where v is defined or in descendant blocks).

Besides explicit uses, v may also be *implicitly* used as a "loop counter" for invoking returns. For example, when the block where v is defined encloses return expressions referring to other variables than v, then the cardinality of v's bindings is used to determine the number of times that the returns are to be invoked. In the extreme case when the binding set is empty (i.e., cardinality is 0), no return will be invoked. In this sense, v serves as the constraint context for the returns.

If a variable v has neither explicit nor implicit uses, we call it has **no-use**. Such variables are definitely nonessential and can be minimized. Otherwise, the essentiality of v depends on the combination of different uses and the number of variable use occurrences.

One vs. Multiple Uses. Basically, v is essential if it has at least two explicit uses, being either *Var* or *Ret* uses, or a *Ret* use and an implicit use. The detailed case studies and rationale are depicted in Figure 7.

```
Essential Variable Identification Procedure

if v has no explicit use
    if v has no implicit use either                              case 1: no-use
    then v is nonessential
    else v is essential   # since removing v would cause the lost of ``loop counter''.
else if v has more than one explicit use (Var or Ret)     case 2: multiple uses
    then v is essential   # since it is necessary for preserving the HMVDs among
          its bindings and those of its dependent variablesor return expressions.
else (i.e., exactly one explicit use)
    if v has no implicit use                                     case 3: one use
    then v is nonessential   # since no two variables or return expressions have co-
          dependencies with v, minimizing v causes no lost of HMVDs or condition changes.
    else
        if v has one Ret use                     case 4: one ret with implicit uses
        then v is essential   # since removing v would cause lost of ``loop counter''.
        else (i.e., v has one Var use)           case 5: one var with implicit uses
            v is nonessential   # since no return expression will be affected by only v
              but not u due to their common life scope, and v occurs only in u's definition.
```

Figure 7: Identifying Variable Essentiality

Lemma 4.1. *All essential variables can be correctly identified by our analysis in Figure 7 based on variable uses.* □

Example 4.1. *We use Q_4 in Figure 2 to illustrate the minimization process. Before minimization (as shown in Figure 8), the var node for $\$t_{Q4}$ (denoted by the solid circles) in VT_{Q4} and that for $\$a_{Q4}$ each have one dependent ret node (denoted by the hollow circles). Hence $\$t_{Q4}$ and $\$a_{Q4}$ each have one Ret use. Also, the original TT_{Q4} reveals that $\$t_{Q4}$ and $\$a_{Q4}$ each are specified alone in a bottom block. Thus they have no chance to affect any other return. This means that $\$t_{Q4}$ and $\$a_{Q4}$ each have no implicit use. Therefore, the var nodes for $\$t_{Q4}$ and $\$a_{Q4}$ in VT_{Q4} can be minimized according to the analysis of case 3 in Figure 7. Correspondingly, the XPath expressions on the ret nodes are changed to $\$b/title$ and $\$b/author$ respectively by substituting the variable occurrences by their definitions.*

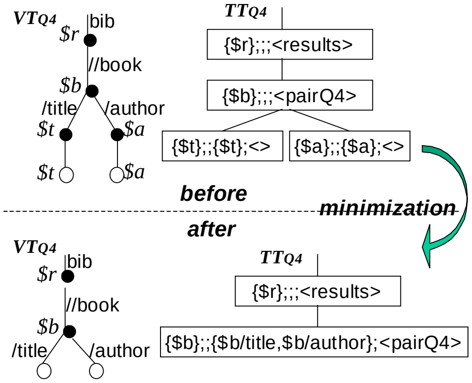

Figure 8: Minimization Example

5. XQUERY CONTAINMENT MAPPING

In this section, we present our containment mapping technique which is composed of three types of mappings. The first two mappings are based on the obtained minimal VarTrees, while the last one is based on the TagTrees.

5.1 VarTree-based Containment Mapping

Given two queries Q_1 and Q_2, the first mapping is to check the containment of the utilized HMVDs in two queries by conducting tree homomorphism (i.e., tree embedding) between their VarTrees. Suppose the embedding is from VT_{Q1} to VT_{Q2}. Then the second mapping is to make sure that the navigation pattern used for deriving each *var* node in VT_{Q1} implies a more restricted constraint than that for the matched *var* node in VT_{Q2}. These two mappings are called **MAC** mapping and **MIC** mapping respectively, indicating that the former is conducted at the macroscopic level of the VarTree (i.e., mapping of *var* nodes) while the latter is performed at the microscopic level of the VarTree (i.e., mapping of navigation patterns encapsulated in derivation edges).

5.1.1 MAC Mapping

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

Definition 5.1. *Suppose VT_1 and VT_2 are the minimal VarTrees of Q_1 and Q_2 respectively. For determining $Q_1 \sqsubseteq Q_2$, 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:*

- $C1)$ *$roots(VT_1) \subseteq roots(VT_2)$,*
- $C2)$ *for any node $u \in VT_1$, there is a match $\Phi(u) \in VT_2$ 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),*
- $C3)$ *u is an ancestor of v for all $u, v \in VT_1$ if and only if $\Phi(u)$ is an ancestor of $\Phi(v)$ in VT_2, and*
- $C4)$ *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[6]**.** 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[7] 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 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.

5.1.2 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 5.2. *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 $u \notin roots(VT_1)$, 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. $p_1 \subseteq p_2$ is checked with $\subseteq$ denoting XPath containment (i.e., there is a tree homomorphism from the pattern tree representation of p_2 to that of p_1[8]).*

Note that if a pair of parent-child nodes (p, c) in VT_1 maps to a pair of ancestor-descendant nodes (a, d) in VT_2 by the MAC mapping, then p_2 is the *concatenated* XPath expressions originated from a to d. This implies that, to make $Q_1 \sqsubseteq Q_2$ hold, more essential variables may be specified in Q_2 than Q_1 to preserve more HMVDs in Q_2's result. The MIC mapping makes sure that p_2, the XPath expression used for deriving d's bindings from a's, imposes a less restricted pattern constraint than p_1, the XPath expression used for deriving c's bindings from p's.

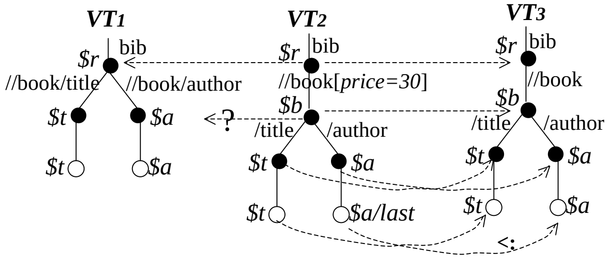

Figure 9: MAC Mapping between Minimal VarTrees

Example 5.1. *Figure 9 illustrates two MAC mappings. One is between the two VarTrees of Q_1 and Q_2 in Figure 1. $\Phi(VT_2) \neq VT_1$ as shown on the left hand side. For one reason, the var node \$b in VT_2 has no match in VT_1 that satisfies $C2$. We can hence derive $Q_2 \not\sqsubseteq Q_1$.*

The second mapping is between the two VarTrees of Q_2 and Q_3 in Figure 1. The right hand side of Figure 1 shows $\Phi(VT_2)=VT_3$, i.e., $\Phi(\$r_{Q2})=\$r_{Q3}$, $\Phi(\$b_{Q2})=\$b_{Q3}$, $\Phi(\$t_{Q2})=\$t_{Q3}$, $\Phi(\$a_{Q2})=\$a_{Q3}$, $\Phi(\$t'_{Q2})=\$t'_{Q3}$, and $\Phi(\$a/last)=\$a'_{Q3}$ ($\$t'_{Q2}$, $\$t'_{Q3}$, and $\$a'_{Q3}$ are the ret nodes). The mapping $\Phi(\$a/last)=\$a'_{Q3}$ holds due to $\mathcal{T}(\$a/last)=last$, $\mathcal{T}(\$a_{Q3})=author$, and $last \leq: author$.

The MIC mapping between the navigation pattern trees encapsulated in the derivation edges of VT_2 and those of VT_3 is also successful. For example, the pattern tree for the XPath expression "//book" in deriving $\$b_{Q3}$ can be embedded

[6] 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.

[7] 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.

[8] 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 [13] for the details of XPath containment.

into that for "//book[price=30]" in deriving $\$b_{Q2}$. Note that the tree embedding direction for XPath containment $p_1 \subseteq p_2$ is from p_2 to p_1.

5.2 TagTree-based Containment Mapping

We now address the implications of nested block structure on the containment of XQuery.

One intuitive example of such implications is the reliance of the return semantics on the emptiness of variable binding set(s). For example, note that since Q_2 in Figure 1 specifies both $\$t_{Q2}$ and $\$a_{Q2}$ in the outer block, the construction of a new *<pairQ2>* element occurs only when a *book* element has both *title* and *author* subelements. In other words, if the binding set of $\$a_{Q2}$ is empty for a specific $\$b_{Q2}$ binding as for example for $b1$ and $b3$ in the source XML in Figure 2, then there will be no invocation of the return to construct the new elements.

Contrary to this example, the construction of *<pairQ3>* elements for Q_3 in Figure 2 is solely based on the bindings of $\$b_{Q3}$, irrelevant of the bindings $\$t_{Q3}$ and $\$a_{Q3}$. The reason lies in the nested block structure of Q_3 (i.e., Q_3 has two query blocks versus that Q_2 has just one). While variables specified in Q_3 are the same as those in Q_2, they however are placed in different blocks (i.e., $\$t_{Q3}$ and $\$a_{Q3}$ are specified and returned in the inner block while $\$b_{Q3}$ is defined in the outer one) as oppose to being put in the same block as those in Q_2. Consequently, $Q_2 \sqsubseteq Q_3$. Similarly, we have $Q_3 \sqsubseteq Q_4$.

Recall that the TagTree structure of a query conforms to the nested block (see its definition in Definition 4.2). The $\bar{V}$ and $\bar{C}$ in an outer block together compose the evaluation context for those in its descendant blocks. Also, variables in the same $\bar{V}$ affect each other in the sense that their cartesian product would produce no tuple if any variable member in $\bar{V}$ has an empty binding set.

DEFINITION 5.3. *Let TT be the TagTree of query Q and $n{=}[\bar{V}, \bar{C}, \bar{R}, \bar{T}]$ be a block node n in it. Variables in $\bar{V}$ mutually depend on each other. Also, they all depend on those variables defined in n's ancestor block nodes. We call such dependencies* ***region dependencies*** *and denote them by $\hookrightarrow$.*

Intuitively, if there is a variable binding $u \rhd v$, then v can only be defined either in the same block or a descendant block of where u is defined, i.e., $u \hookrightarrow v$. However, we cannot imply $u \rhd v$ from $u \hookrightarrow v$. This is formally stated as below.

LEMMA 5.1. *For any two variables u and v of Q, if $u \rhd v$, then $u \hookrightarrow v$.* □

5.2.1 Block Mapping

We now define a third mapping that complements the previously defined MAC and MIC mappings.

DEFINITION 5.4. *Let TT_1 and TT_2 be the TagTrees of Q_1 and Q_2 respectively. The* ***block mapping*** *is a one-to-many mapping function θ from each block node n of TT_1 to nodes of TT_2, denoted by $\theta(TT_1){=}TT_2$, such that $n{=}(\bar{V}, \bar{C}, \bar{R}, \bar{T})$ in TT_1 and its* ***image set*** *$S{=}\theta(n)$ in TT_2 satisfy:*

- $C1'$) *for every variable $u \in \bar{V}$, $\Phi(u) \in \bigcup_{n_i} \bar{V}_i$ ($n_i {\in} S$),*
- $C2'$) *for any two variables $w, x {\in} \bigcup_{n_i} \bar{V}_i$ ($n_i {\in} S$), if $w \hookrightarrow x$, then there must be u and v in TT_1, such that $\Phi(u){=}w$, $\Phi(v){=}x$, $u \hookrightarrow v$, and*
- $C3'$) *any $c_i \in \bigcup_{n_i} \bar{C}_i$ ($n_i {\in} S$) can be implied by a condition $c \in (\bar{C} \cup \bigcup_{m_i} \bar{C}')$ (m_i is an ancestor block node of n).*

C1': Containment of variables. This condition is actually used for establishing the θ mappings (i.e., finding for each block node n in TT_1 its image node set S) based on the VarTree node matches. Intuitively, a block node n_i in TT_2 is included in S if any variable in it is the match of any variable $u \in \bar{V}$ in n. $\bar{V}$ and $\bigcup_{n_i} \bar{V}_i$ denote variables in n and the union of those defined in n's images $n_i {\in} S$ respectively.

C2': Implication of region dependencies. If a variable w in an image node n_i of TT_2 is involved in a region dependency (e.g., $w \hookrightarrow x$), then C2' ensures that there must be a region dependency between the corresponding variables in block nodes of TT_1. In other words, the region dependencies with matched variables in TT_2 involved must be a subset of those among the corresponding variables in TT_1.

C3': Implication of where-conditions. Suppose n_i in TT_2 is an image node of the node block n in TT_1. C3' checks if every where-condition c_i left in n_i can be implied from a where-condition c either in n or an ancestor block of n (i.e., $c \in \bigcup_{m_i} \bar{C}'$, with $m_i {\in} ancestors(n)$).

In a nutshell, $C1' \sim C3'$ required by the block mapping make sure that Q_1 must assert more restricted constraints, i.e., region dependencies and where-conditions, than Q_2 does.

EXAMPLE 5.2. *Suppose that two adjacently nested blocks n_1 and n_2 in TT_1 define variables u and v respectively. There is no other variable in n_1 or in n_2. We also suppose that $\Phi(u) = x$ and $\Phi(v) = w$, and that x and w are defined in the same block n in TT_2. By condition $C1'$, we have $\theta(n_1) = \{n\}$ and $\theta(n_2) = \{n\}$. We derive $x \hookrightarrow w$ and $w \hookrightarrow x$ due to the mutual region dependencies asserted by a block node. Also, from the enclosing relationship between n_1 and n_2, we know that $u \hookrightarrow v$ but $v \not\hookrightarrow u$. C2' is not satisfied based on these facts. Consequently, the block mapping fails and we derive $Q_1 {\not\sqsubseteq} Q_2$.*

Putting all three types of containment mappings together, we now have a sound (not generally complete) solution for XQuery containment in the presence of variable binding dependencies.

THEOREM 1. *Given two XQueries Q_1 and Q_2, $Q_1 \sqsubseteq Q_2$ if there exist a MAC mapping $\Phi(VT_1){=}VT_2$, a MIC mapping $p1 {\subseteq} p2$ (i.e., the encapsulated XPath containment) for every matched node pair, and a block mapping $\theta(TT_1){=}TT_2$.* □

6. SYSTEM AND EVALUATION

Based on our proposed containment mapping approach for XQuery, we have designed and implemented a semantic caching system called ACE-XQ [6, 7]. The ACE-XQ system is developed using Java 1.3. It utilizes the IPSI-XQ engine [14] installed at both the cache and the remote server sites to execute the rewritten query and the original query respectively. Source XML documents are hosted at the server.

The first set of experiments is for validating our containment mapping and rewriting methods for XQuery. For this, we designed some query workload[9] that includes queries that are similar to those W3C use cases [20] and are within the scope of our XQuery fragment. The experiment shows that the results produced by running a query with and without

[9] We mainly focus on the "refining" case. Namely, the hit ratio of a new query being contained in a cached one is high.

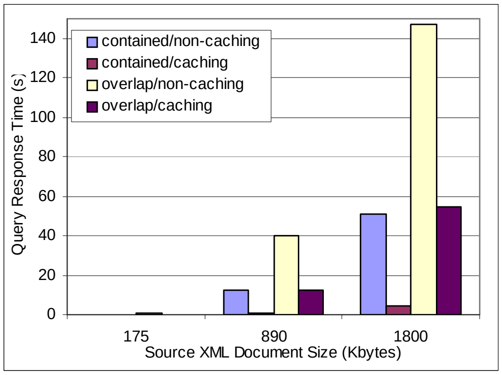

Figure 10: Query Response Times for Different Document Sizes w/o Caching

the attempt of conducting containment mapping and rewriting it against a containing view result are the same.

The second set of experiments is to evaluate the query performance with and without the semantic cache. As expected, Figure 10 shows the improved query performance by up to 10 folds for the totally contained cases in our setting.

Table 1 shows the break down of the query response time for a contained case into the computation overhead (i.e., query decomposition and minimization, containment mapping, and rewriting) and the query evaluation time. We see that the overhead is considerably small compared to the query evaluation time. This implies that although the complexity of our XQuery containment approach is NP-complete in general (since all three mappings are tree homomorphism extensions with additional checking of equivalence conditions, of the inclusive relationships between element types, etc.), it is efficient and practical in many real scenarios.

XML	Decomp. &	Cont.	Query	Query
Size	Minimization	Mapping	Rewriting	Execution
175KB	0.8ms	8.8ms	5.2ms	173.6ms
890KB	0.8ms	9.2ms	5.4ms	1068.8ms
1800KB	0.8ms	9.1ms	5.2ms	4525.4ms

Table 1: Processing Time Decomposition

Extensive experimental studies can be found in [8, 5].

7. CONCLUSION

In this paper, we proposed a containment mapping approach that handles the effects of variable binding dependencies and the nested block structure on XQuery containment. Our approach provides sufficient conditions for solving nested XQuery containment.

An intermediate future work would be to incorporate the XQuery logical optimization technique in [9] in our normalization step to reduce the possible navigation redundancies in the VarTree representation. This helps to prune the space for conducting containment mapping. However, the lack of this optimization step as of now does not impact the soundness of the approach.

The XQuery fragment defined in this paper provides a good scope for us to focus on a set of important XQuery features with respect to the containment problem. We plan to extend the proposed containment mapping approach to accommodate a broader fragment of XQuery that includes disjunctions, aggregations, and other features as well as to consider more general constraints in XML and XQuery.

8. REFERENCES

[1] A. Deutsch and V. Tannen. Containment of Regular Path Expressions under Integrity Constraints. In *KRDB, Rome, Italy*, pages 1–11, June 2001.

[2] M. Arenas and L. Libkin. An Information-Theoretic Approach to Normal Forms for Relational and XML Data. In *PODS, San Diego, CA*, pages 15–26, 2003.

[3] D. Calvanese, D. Giacomo, and M. Lenzerini. Rewriting of Regular Expressions and Regular Path Queries. In *PODS, Philadephia, PA*, pages 194–204, 1999.

[4] A. K. Chandra and P. M. Merlin. Optimal Implementations of Conjunctive Queries in Relational Data Bases. In *STOC*, pages 77–90, 1977.

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

[6] L. Chen and E. A. Rundensteiner. ACE-XQ: A CachE-aware XQuery Answering System. In *WebDB*, pages 31–36, June 2002.

[7] L. Chen, E. A. Rundensteiner, and S. Wang. XCache - A Semantic Caching System for XML Queries. In *SIGMOD demonstration paper, Madison, Wisconsin*, page 618, 2002.

[8] L. Chen, S. Wang, and E. A. Rundensteiner. Evaluation of Replacement Strategies for XML Query Cache. *Data and Knowledge Engineering Journal (DKE)*, 49(2):145–175, 2004.

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

[10] X. Dong, A. Halevy, and I. Tatarinov. Containment of nested xml queries. In *VLDB, Toronto, Canada*, pages 132–143, 2004.

[11] W. Fan and J. Simeon. Integrity Constraints for XML. In *PODS, Dallas, TX*, pages 23–34, 2000.

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

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

[14] IPSI-XQ. http://ipsi.fhg.de/oasys/projects/ipsi-xq/index_e.html.

[15] D. Lee and W. W. Chu. Constraints-Preserving Transformation from XML Document Type Definition to Relational Schema. In *ER, Salt Lake City, Utah*, pages 323–338, 2000.

[16] A. Levy. Answering Queries Using Views: A Survey. *VLDB Journal*, pages 270–294, 2001.

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

[18] I. Tatarinov and A. Halevy. Efficient Query Reformulation in Peer Data Management Systems. In *SIGMOD, Paris, France*, pages 539–550, 2004.

[19] M. Vincent and J. Liu. Multivalued Dependencies in XML. In *BNCOD, Coventry, UK*, pages 4–18, 2003.

[20] W3C. XML Query Use Cases, W3C Working Draft 02, May, 2003. http://www.w3.org/TR/xquery-use-cases.

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

[22] P. Wood. Containment for XPath Fragments under DTD Constraints. In *ICDT, Siena, Italy*, pages 300–314, 2003.

[23] 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.