Query Answering Using Views for XML

by

Zheng Zhao

BSc. Hon., The University of Western Ontario, Canada, 2002

A THESIS SUBMITTED IN PARTIAL FULFILLMENT OF

THE REQUIREMENTS FOR THE DEGREE OF

Master of Science

in

THE FACULTY OF GRADUATE STUDIES

(Department of Computer Science)

we accept this thesis as conforming
to the required standard

The University of British Columbia

March 2005

Abstract

The problem of answering query using views is to find efficient methods of answering a query using a set of previously materialized views over the database, rather than accessing the database. As XML becomes the standard of data representation and exchange over the internet, the problem has recently drawn more attentions because of its relevance to a wide varieties of XML data management problems, there is a pressing needs to develop more techniques to solve it for XML data effectively and efficiently.

We study a class of XPath queries and materialized views which may contain child, descendant axis and predicates. We first describe an algorithm to find the maximally-contained rewritings in the absence of database schema. We then present an efficient algorithm to search the maximally-contained rewriting under choice-free acyclic schema and prove the uniqueness of the maximally-contained rewriting. Finally we show its performance experimentally by extending our algorithm to answer queries in XQuery expression.

Contents

List of Figures

Acknowledgments

I am deeply appreciate my supervisor Dr. Laks V.S. Lakshmanan who provided me invaluable guidance and support. Without him, this would never have been completed.

I would like to thank all the members in database lab, especially Dr.George Tsiknis and Wendy Hui Wang for their help in my work.

ZHENG ZHAO

The University of British Columbia

March 2005

To my parents and husband, for their endless love and continuous encouragement.

Chapter 1

Introduction

The problem of answering query using views is to find efficient methods of answering a query using a set of previously materialized views over the database, rather than accessing the database [7]. This problem is relevant to many data management problems. One of the major context where the problem of answering queries using views is considered is data integration and data warehouse design where the efforts focus on searching a maximally-contained rewriting, the best results possible.

Data integration systems combine data residing at a multitude of autonomous data sources, and provide a uniform query interface, called global schema, which can be queried by the user. In the design of a data integration system, we need to make a basic decision which is related to the problem of how to specify the relation between the sources and the global schema. There are basically two approaches for this problem. The first approach, called global-as view (GAV), requires that the global schema is expressed in terms of the data sources. This means that every concept of the global schema is associated with a view over the data sources, so that its meaning is specified in terms of the data residing at the sources. In the second

approach, called local-as-view (LAV), the global schema is specified independently from the sources, and the relationships between the global schema and the sources are established by defining every source as a view over the global schema. In the area of data warehouse design we need to choose a set of materialized views in the warehouse to improve the query performance. In this case, the most important step is to select a set of views to materialize that answers all the queries of interest while minimizing the total query evaluation and view maintenance cost. When a query is posed, it is evaluated locally, using the materialized views. Accessing the original data sources are avoided mainly because either the original sources are not accessible any more or it costs too much. Both problems are translated into the problem of query rewriting using views in which we often need to settle for a contained result which is a subset of the original query result rather than an equivalent one because the given materialized views may not cover the entire database.

The problem of rewriting queries using materialized views has been extensively studied in the relational world. Many algorithms were developed for a specific area of applications [3, 6, 13, 20, 7] such as the bucket algorithm, the inverse-rules algorithm, the MiniCon algorithm, etc. In contrast, this problem for XML data management has not been fully explored. Some of the existing work is outlined in the Chapter 6 Related Work. XML has become the standard for data representation and exchange over Internet. With W3C's recommendation, XQuery[17] emerges as the standard query language for XML and XPath[16] is a language for navigating XML documents which is embedded in XQuery. Both these languages are based on a basic paradigm of finding bindings of variables by matching tree patterns against a database. Similar to relational databases, the problem of finding a rewriting of XQuery/XPath queries using a set of XPath views is relevant to a wide varieties of

XML data management problems. Besides those two major applications we mentioned above, this problem is also related to semantic web applications as illustrated in [8] when the query is posed over the schema of source S and we wish to reformulate it over the schema of target T which is the schema neighbor of S. The problem of refomulating Q is known as answering queries using views. Therefore there is a pressing need to develop better techniques to solve the problem of rewriting queries using materialized views effectively and efficiently.

In this thesis, we consider this problem for XPath expressions with/without a database schema. Informally, we define the problem as following. Suppose we are given a query Q, and a set of previously materialized view definitions $V_1, \ldots, V_n$ all expressed in XPath. Is it possible to answer query Q using only the answers to the views $V_1, \ldots, V_n$ without accessing accessing the database? If so, how? When the database schema is given, the query and views are over the same schema. Currently we concentrate on XPath expressions containing child, descendant axis and predicates. The specific contributions of this thesis are the following:

- We propose an algorithm to check when a view is usable to answer a XPath query and find maximally-contained rewritings in the absence of schema. We show the lower bound of the time complexity is EXPTIME.
- We show that containment for $XP^{\{/,//,[\,]\}}$ can be decided in PTIME under acyclic choice-free DTDs.
- We describe a PTIME algorithm to find the maximally-contained query rewriting using views under acyclic choice-free DTDs.
- We introduce an approach to answer an XQuery query using XPath views by extending our algorithm and present detailed experimental results to show the

performance.

The rest of the thesis is organized as follows. In Chapter 2 we describe the class of XPath fragments and database schema we studied. We present our algorithm in the absence of schema in Chapter 3. In Chapter 4, we prove that for tree pattern queries in $XP^{\{/,//,[\ ]\}}$, five types of necessary and sufficient constraints implied by choice-free, acyclic DTD can be used to decide query containment problem and extend to solve the problem of a query rewriting using views using a PTIME algorithm. We provide experimental results in Chapter 5 where we illustrate how to use our algorithm to answer XQuery query using XPath views. Finally, we discuss related work in Chapter 6 and conclude in Chapter 7.

Chapter 2

Background and Problem Studied

2.1 XPath and Tree Pattern Queries

An XML database is a finite rooted ordered tree $T = (\mathcal{N}, \mathcal{E}, r, \lambda)$, where $\mathcal{N}$ represents element nodes, $\mathcal{E}$ represents parent-child relationship, λ denotes the labelling function to assign a tag with each node, and r is the root. Associated with each node is a set of attribute-value pairs. In our work, we do not consider order any further.

Tree pattern queries, introduced in [1], capture a useful fragment of XPath. A *tree pattern query* (TPQ) is a triple $Q = (N, E, F)$, where (N, E) is a rooted tree, with nodes N labelled by variables, and with $E = E_c \cup E_d$ consisting of two kinds of edges, called pc- (E_c) and ad-edges (E_d), corresponding to the child and descendant axes of XPath. A distinguished node in N (shown boxed in Figure 3.3) corresponds to the answer element. The path from root node to the distinguished

node is the distinguished path. F is a conjunction of tag constraints (TCs), value-based constraints (VBCs), and node identity constraints (NICs). TCs are of the form $\$x.tag = t$, where t is a tag name. VBCs include selection constraints $\$x.val$ relop c, $\$x.attr$ relop c, and join constraints $\$x.attr$ relop $\$y.attr'$, and $\$x.val$ relop $\$y.val$, where relop $\in \{=, \neq, >, \leq, \geq, <\}$, *attr, attr'* are attributes, *val* represents content, and c is a constant. NICs are $\$x$ *idop* $\$y$ where *idop* $\in \{=, \neq\}$. Q is *join-free* if it contains no join constraints and no NICs. We assume no disjunctions appear in VBCs and queries are join-free throughout the thesis with a few clearly identified exceptions.

We denote the nodes of a query Q by $N(Q)$ and the nodes of a view V by $N(V)$. The root nodes of Q and V will be denoted by $R(Q)$ and $R(V)$ respectively. We use d_Q and d_V to denote the distinguished nodes of query Q and view V. The distinguished paths in Q and V are denoted by D_Q and D_V (i.e. the paths in Q and V from $R(Q)$ to d_Q and $R(V)$ to d_V respectively). For any node x in Q or V, the tag name associated with that node will be denoted by $tag(x)$ and the value based constraints associated with that node will be denoted by $VBC(x)$.

Answers for TPQs are formalized using homomorphism. A *homomorphism* is a function h : query $Q \rightarrow$ a tree T with the following properties:

1. $h(R(Q)) = h(R(T))$;
2. $\forall x \in Q, tag(x) = tag(h(x))$;
3. $\forall x, y \in Q$, if (x, y) is a pc edge in Q then $(h(x), h(y))$ must be a pc edge in T;
4. $\forall x, y \in Q$, if (x, y) is an ad edge in Q then $(h(x), h(y))$ must be a path in T.

2.2 Materialized XPath Views

We consider XPath views are in the class of copy semantics which implies that views store copies of answer elements. This implies that XPath views can be used to answer XPath queries with subsequence operations on the results of the view without navigating to the parent or ancestors. Since we consider join-free XPath query in our work, only a single view would be involved in rewriting if it is usable. For the ease of readability, we denote an XPath query and a view by Q and V respectively.

2.3 Query Containment and Query Rewriting

Query containment is a necessary condition for rewriting query using views. As proven in [14], for any wildcard-free XPath queries Q and Q', $Q' \subseteq Q$ iff there is a containment mapping from $Q \rightarrow Q'$. A *containment mapping* is a function $h : Q \rightarrow Q'$ with the following properties: (1) $h(R(Q)) = h(R(Q'))$; (2) $\forall x \in Q, tag(x) = tag(h(x))$; (3) $\forall x, y \in Q$, if (x, y) is a pc a edge in Q then $(h(x), h(y))$ must be a pc edge in Q'; (4) $\forall x, y \in Q$, if (x, y) is a ad edge in Q then $(h(x), h(y))$ must be a path in Q',which may include pc edges and/or ad edges.

In our context, the correctness of the rewriting is verified by using query containment. We say that Q is *rewritable* using V if there exists an XPath expression E such that for every XML database D, $E \circ V(D) \subseteq Q(D)$ then E is said to be a *sound rewrite* of Q using V. In addition, our goal is to find maximal sound rewriting(s). A sound rewriting E of Q is said to be *maximal* if there has no E' such that for every XML database D, $E \circ V(D) \subset E' \circ V(D)$.

2.4 Schema and DTDs

We are especially interested in studying the problem of rewriting query using views in the presence of schema. We abstract the schema of a database (in our work, we only consider DTDs) as a graph with nodes corresponding to tags and edges labelled by one of the quantifiers '?, 1, *, +' with their standard meaning of 'optional', 'one', 'zero or more', and 'one or more' respectively. All tags in D denotes set σ. The set of trees satisfying DTD D is denoted SAT(D). A XPath query Q is *satisfiable* if there is a tree $T \in SAT(D)$ such that $Q(T) \neq \varnothing$. Otherwise, Q is unsatisfiable. The satisfiability of TPQs with/without schema is recently studied in [12]. Without losing the generality, we assume that both query Q and view V are satisfiable with regard to DTD.

DTDs provides constraints on the structure of XML documents. Hence, while Q_1 may not rewritable using Q_2 in general, it may be the case that given a DTD D, Q is rewritable using V when both satisfy D, by applying a compensation expression E on V. For ease of exposition, we initially focus on acyclic choice-free DTDs. If C is a set of constraints inferred by DTD, then $SAT(C)$ denotes the set of trees in T_Σ which satisfy each constraint in C.

Problem Statement: We formally define the problem of query answering using views (QAV) for XPath fragment, denoted $XP^{\{/,//,[\ ]\}}$ in our context, as follows: Given a query Q and a view V both expressed in XPath, check whether V is usable for answering Q. If so, find all maximally-contained rewriting(s) of Q using V, with/without choice-free acyclic DTD.

Chapter 3

Query Answering Using Views without Schema

In this chapter, we illustrate an algorithm for computing maximally-contained rewriting(s) in the absence of schema and prove the soundness of our algorithm. We firstly give some useful definitions, provide a detailed proof and then present the algorithms. An example follows to show that time complexity can not be better than EXPTIME.

3.1 Sound Rewritings and Maximal Rewritings

Definition 3.1 (Embedding) *An* embedding $f : Q \rightsquigarrow V$ *is a partial function from* $N(Q)$ *to* $N(V)$ *satisfying the following properties.*

1. *If the first character in the XPath expressions* Q *and* V *are* / *then,* $f(R(Q)) = R(V)$.

2. $\forall x \in Q$, f *is defined on* x *implies* $(tag(x) = tag(f(x)) \wedge (VBC(f(x)) \rightarrow$

$VBC(x)$).

3. $\forall x, y \in Q$, f *is defined on* x *and* y, (x, y) *is a pc edge in* Q *implies that* $(f(x), f(y))$ *is a pc edge in* V.

4. $\forall x, y \in Q$, f *is defined on* x *and* y, (x, y) *is an ad edge in* Q *implies that there exists a path from* $f(x)$ *to* $f(y)$ *in* V *which may include pc or ad edges.*

5. $\forall x \in Q$, *then* f *is defined on every ancestor of* x *(upward closed).*

Definition 3.2 (Useful Embedding) $f : Q \rightsquigarrow V$ *is a* useful embedding, *provided:*

1. f *is an embedding;*

2. $\forall x \in D_Q$, *if* $f(x)$ *is defined,* $f(x) \in D_V$;

3. *Let* $P = \{v_0, v_1, \ldots, v_k\}$ *be any path in* Q.

   (a) *either* f *is defined on* $v_0, v_1, \ldots, v_k$; *or*

   (b) $\forall i : f(v_i)$ *is defined,* $f(v_i) \in D_V$ *and suppose* $v_l = max\{i | f(v_i)$ *is defined* $\}$, *then either* $f(v_l) = d_V$ *or* (v_l, v_{l+1}) *is an ad edge in* Q.

Definition 3.3 (CAT: Clipped Away Tree) *Let the distinguished path in* Q, $D_V = \{v_0, v_1, \ldots, v_k\}$. *A Clipped Away Tree (CAT) is a subtree of* Q *rooted at* v_i, *s.t.* f *is not defined on* v_i *but is defined on* v_{i-1}.

Definition 3.4 (Extension of Useful Embedding) *A useful embedding* g *is an* extension *of another useful embedding* f *if* $dom(f) \subset dom(g)$.

In this chapter, we denote different rewritings $(E_g \circ V)$ and $(E_f \circ V)$ as R_g and R_f. Both are the sound rewritings derived from the useful embeddings g and f respectively.

Theorem 3.1 *Let $Q, V \in XP^{\{/,//,[\ ]\}}$ and Q is join-free. Q is rewritable using V iff there exists a useful embedding $f : N(Q) \rightsquigarrow N(V)$.*

Proof (Only if) Let $E \circ V$ be a sound rewrite of Q using V, i.e. $\forall databaseT$: $E \circ V(T) \subseteq Q(T)$. Let $h : N(Q) \rightarrow N(E \circ V)$ be containment mapping, s.t. $\forall x \in N(Q), h(x) \in N(V)$ or $h(x) \in N(E)$. We construct a useful embedding $f : N(Q) \rightsquigarrow N(V)$ as follows:
$\forall x \in Q : h(x) \in N(V), f(x) = h(x)$. By the definition of containment mapping, f is a valid embedding from $N(Q) \rightsquigarrow N(V)$. f also satisfies all path constraints defined in useful embedding because:

1. $\forall x \in N(D_Q)$ and x is defined in f: $f(x) \subseteq N(D_V)$ since $f(R(Q)) = f(R(V))$ and $f(D_Q) = D_{E \circ V}$.

2. Mark nodes of Q top down as follows. If $x \in Q$ and $h(x) \in V$, mark the node as V, else mark it as E. The marks on all paths from $R(Q)$ to any leaf node are of the form V^*E^*. Let x be the last node in any path marked V and y be the first node in the path marked E. Since E is a valid rewrite,

   (a) either x is mapped to d_v, OR

   (b) (x, y) is an ad edge and x is mapped to a node in D_v

(If) Let $f : N(Q) \rightsquigarrow N(V)$ be a useful embedding. We construct E and extend f as follows:
$\forall x \in N(Q)$ s.t. $x \in dom(f)$ and $\exists y$ s.t. $edge(x, y) \in Q$ and $y \notin dom(f)$. Let T_y denote the subtree rooted at y. Do the following: add a copy T'_y of T_y as a child subtree of d_v and define for every node $z \in T_y, f(z) = z'$ where z' is the corresponding node in T'_y. If (x, y) is a pc(ad) edge, then (d_v, y') is a pc(ad) edge. E

contains all such T'_ys. The extended f is the required containment mapping because $\forall x$ defined in the useful embedding is defined in f and $\forall x$ NOT defined in the useful embedding, $f(x)$ is its image in the corresponding T'_y. f is a valid containment mapping from $Q \rightarrow E \circ V$. Therefore, $E \circ V$ is a sound rewriting of Q using V. □

For efficiency concerns, we aim to generate only maximal rewrites. The following lemma makes this goal possible to achieve. We will describe the algorithm in the next section.

Lemma 3.1 *Let a useful embedding g is an extension of f. $R_f \subseteq R_g$ iff $dom(f) \subset dom(g)$ and $\forall x \in dom(g) - dom(f) \mid \exists y \notin dom(g)$ and $edge(x, y) \in Q$: (x, y) is an ad edge.*

Proof (Only if) We know that g is an extension of f. Mark every node x of Q top down as follows: if $x \in dom(f)$, mark the node as F; if $x \in dom(g) - dom(f)$, mark the node as G; else mark it as E. The marks on all paths from $R(Q)$ to any leaf node are of the form $F^*G^*E^*$. Let u be the last node in any path marked F, x be the first node in the path marked G, y be the last node in the path marked G and z be the first node in the path marked E. From the way we construct R_f and R_g, we know all subtree T_x will be copied as T'_x to attached to d_v using pc(ad) edge in R_f if (u, x) is a pc(ad) edge; all subtree T_z will be copied as T'_z to attached to d_v in R_g using pc(ad) edge if (y, z) is a pc(ad) edge. Since $R_f \subseteq R_g$, there is a containment mapping $h : N(R_g) \rightarrow N(R_f)$. z', the image of z in R_g is mapped to z'', the image of z in R_f which is a node in T'_x. Since path(d_v, z'') in R_f through node x' must contain at least two edges. Therefore, edge (d_v, z') in R_g must be ad edge. Thus the pre-image of u in Q, (y, z) must be an ad edge in Q.

(If) We show $R_f \subseteq R_g$ by constructing a containment mapping $h : N(R_g) \rightarrow N(R_f)$ as follows:

1. $\forall x \in N(V)$ of R_g, h is defined as its image in $N(V)$ of R_f since $R_f = E_f \circ V$ and $R_g = E_g \circ V$,

2. $\forall u \in T_y$, a child subtree of D_V rooted at y in R_g, s.t. the pre-image of y in Q has an edge to node x and $x \in dom(f)$: $\exists T_y$', a child subtree of D_V in E_f, T_y is isomorphic to T_y', $h(u)$ is defined as its image in T'_y. Since we derive R_g and R_f from g and f, we know $\forall y$ whose pre-image $\notin dom(g) : \exists$ edge (t, y)in Q and $t \in dom(g) \cap dom(f) : T_y$ is duplicated in E_g and E_f,

3. $\forall T_y$, a child subtree of D_V in R_g: the pre-image of y in Q has an edge to node x s.t. $x \in dom(g) - dom(f) : \exists T'_y$, a subtree of D_V in R_f s.t. T_y is isomorphic to T'_y. y can be mapped to y' since (D_V, y) is an ad edge and (D_V, y') is a path in R_f. $\forall u \in T_y, h(u)$ is defined as its image in T_v.

□

3.2 Algorithm and Time Complexity

3.2.1 Algorithms

We first introduce three help functions which are used to simplify the problem solving. The first and second function are quite straight forward so we just give a brief description rather than details. The third one is the most complicated so we would show it step by step.

We next introduce the main procedure to find all useful embedings from Q to V.

We show the execution of the above algorithm using the example of Figure 3.3. For readability, whenever the tag constraint \$x.tag = t appear in Q and V,

1. **Function: map-DPath**

   Input: The distinguished path of Q and V, D_V and D_Q. Let $N(D_V)$ =$\{v_0, v_1, \ldots, v_k\}$ and $N(D_Q)$ =$\{q_0, q_1, \ldots, q_m\}$.
   Output: A set of valid (partial) path mappings H from $N(D_Q) \rightsquigarrow N(D_V)$.

   Each $h \in H$ will preserve path and tag obligations and for every unmapped node q_i such that $h(q_{i-1})$ is defined in h, if pc-edge(q_{i-1},q_i) then h(q_{i-1})= v_k(the distinguished node of V).It will also generate a candidate list Cq_l for node q_l s.t. $v_j \in Cq_l$ if there exist a h' such that $h(q_j) = v_j$.

2. **Function: map-Subtree**

   Input: One node q_i in Q and the other node v_j in V.
   Output: Return the total mapping if the tree rooted at q_i in Q has a containment mapping to the tree rooted at v_j in V. Otherwise, it will return NULL.

   We implemented the PTIME containment mapping algorithm introduced in [1].

3. **Function: map-To-Dv**

   Input: One node q_i' in Q and the other node $v_j \in D_V$.
   Output: Return a set of all valid (partial) tree mappings T from the tree rooted at q_i' in Q to the fragment of D_v starting from v_j. If no mapping exists, return NULL.

   As a valid tree mapping $t \in T$, it will preserve path and tag obligations and for every unmapped node q_j' such that q_{j-1}' is mapped, if pc-edge(q_{j-1}',q_j') then t(q_{j-1}')= v_k(the distinguished node of V).It will also generate a candidate list Cq_i' (root of the tree) for q_i' s.t. $v_j \in Cq_i'$ if there exist a t such that $t(q_i') = v_j$.

Figure 3.1: Help Functions

Procedure: get-UsefulEmbeddings

Let the distinguished path D_V $=\{v_0, v_1, \ldots, v_k\}$ and D_Q $=\{q_0, q_1, \ldots, q_l\}$. Node not lying on the D_Q and D_V denote q' and v' respectively.

Input: Q and V.
Output: All useful embeddings in set F.

Assign unique id to each node in Q and V;
H = map-Dpath(D_Q, D_V);
If H is empty, return NULL;
For each node $q_i \in D(Q)$ s.t. $Cq_i \neq \varnothing$

1. For each child node q' of q_i which is not on D_Q

   (a) For each node v' in Cq_i

   i. If pc(q', q_i),get all pc-child nodes v_j of v' s.t. tag(v_j)= tag(q')
   ii. If ad(q', q_i),get all descendant node v_j of v' s.t. tag(v_j)= tag(q')
   iii. Save all v_js in set V
   iv. For each v_j in V
   A. For v_j is not in D_V, map-Subtree(q_i, v_j){
   B. If success, record the mapping and add v_j to Cq_i. Break;}
   C. If v_j is in D_V, map-To-Dv(q_i,v_j){
   D. If success, record all mappings, and add v_j to Cq_i.Break;
   E. If fail and pc(q', q_i), prune v' from Cq_i.
   F. If fail and ad(q', q_i), add $\varnothing$ to Cq_i}

Use all pre-stored candidate list of query node's mapping, output all useful embeddings.

Figure 3.2: Algorithm to find rewriting of Q using V

we write t right next to $x in the figure.

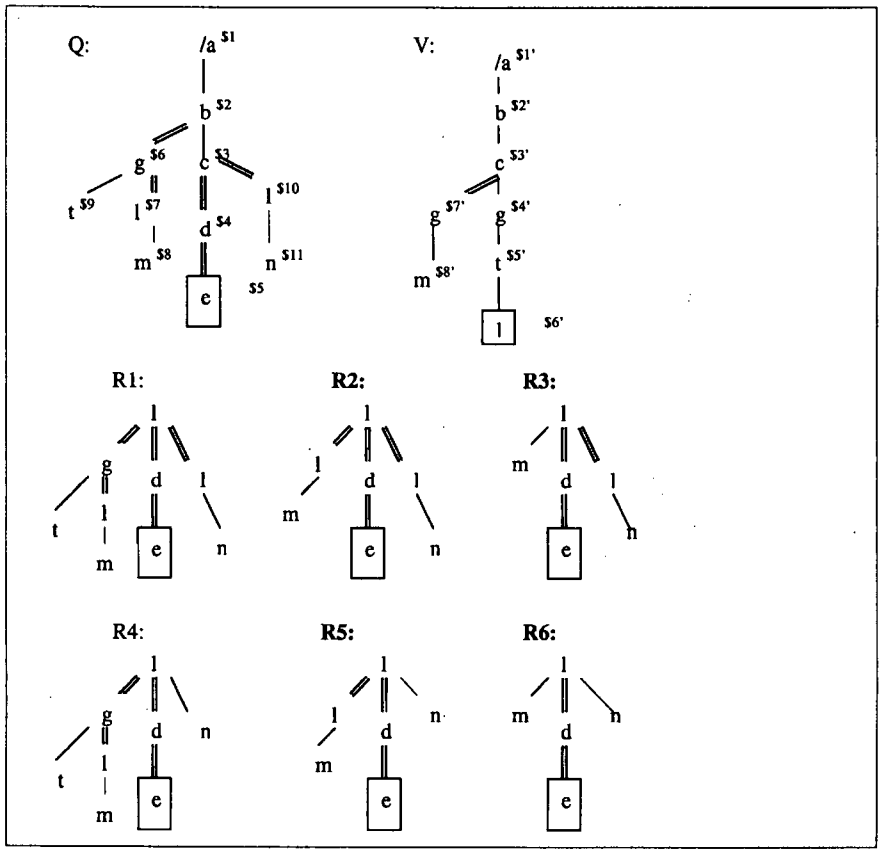

Figure 3.3: Schemaless Case Example

1. Call function map-Dpath(D_Q, D_V). H contains only one mapping which is h $= \{1 \rightarrow 1', 2 \rightarrow 2', 3 \rightarrow 3'\}$. The candidate lists are $C_1 = 1', C_2 = 2', C_3 = 3'$. The CAT is the subtree rooted at node 4.

2. Node 1 in Q has no other children besides node 2, so 1 will be skipped.

3. Node 2 has one child 6 which is not on D_Q. h(2)=2' and ad(2,6) so 6 may

map to two nodes in V, 4' and 7' which are descendants of node 2'. We always try to map the candidate which is not on D_V. By doing that we may get the total mapping of the subtree rooted at 6. So we test map-Subtree(6, 7'), it fails. Then test map-To-Dv(6, 4'), we got three mappings: $h2_1 = \{6 \rightarrow 4', 9 \rightarrow 5', 7 \rightarrow 6'\}$; $h2_2 = \{6 \rightarrow 4', 9 \rightarrow 5'\}$; $h2_3 = \{\varnothing\}$. $h2_3$ implies the whole subtree rooted at 6 will be attached to d_v as part of rewrite.

4. Node 3 has one child 10 which is not on D_Q. h(3)=3' and ad(3,10). Same as the operation done on Node 2, we will obtain two mappings $h3_1 = \{10 \rightarrow 6'\}$; $h3_2 = \{\varnothing\}$.

5. We generate all the embedding using combinations of CAT, h_2 and h_3 which gives 6 different embeddings:

   - $f_1 = \{1 \rightarrow 1', 2 \rightarrow 2', 3 \rightarrow 3', 6 \rightarrow 4', 9 \rightarrow 5', 7 \rightarrow 6', 10 \rightarrow 6'\}$.
   - $f_2 = \{1 \rightarrow 1', 2 \rightarrow 2', 3 \rightarrow 3', 6 \rightarrow 4', 9 \rightarrow 5', 10 \rightarrow 6'\}$.
   - $f_3 = \{1 \rightarrow 1', 2 \rightarrow 2', 3 \rightarrow 3', 10 \rightarrow 6'\}$.
   - $f_4 = \{1 \rightarrow 1', 2 \rightarrow 2', 3 \rightarrow 3', 6 \rightarrow 4', 9 \rightarrow 5', 7 \rightarrow 6'\}$.
   - $f_5 = \{1 \rightarrow 1', 2 \rightarrow 2', 3 \rightarrow 3', 6 \rightarrow 4', 9 \rightarrow 5'\}$.
   - $f_6 = \{1 \rightarrow 1', 2 \rightarrow 2', 3 \rightarrow 3', 10 \rightarrow 6'\}$.

6. Finally, we generate six rewritings R1, R2,..., R6 corresponding to six distinct useful embeddings $f_1, f_2, ..., f_6$ in the following way: for each f_i, mark those nodes which is not defined in $f_i$in Q, copy the branches and subtrees connected by those nodes and attach them to the distinguished node of V. Please refer to the figure for final results where the root node of each rewriting R_i, is the distinguished node of V.

As we mentioned before, our goal is to generate maximal rewrites for efficiency. To achieve this, we improve the above algorithm to be as "greedy" as possible in searching the mappings based on the result of Lemma 3.1. In the function **Map-To-Dv(q, v')**, we add prune procedure in the end: t_i will be pruned from T if there exist t_j in T such that $dom(t_j) \supset dom(t_i)$ and every unmapped node m which has an edge connected with some node $n \in dom(t_j) - dom(t_i)$, (n, m) is an ad edge. In the example, we will prune $h2_3$ from the mappings of node 2 in Q because $dom(h2_3) \subset dom(h2_2)$ and there is only one node 7 which has parent node 6 in $dom(h2_2) - dom(h2_3)$ and $ad(6, 7)$. However, we can prune neither $h2_1$ nor $h2_2$. Although $dom(h2_2) \subset dom(h2_1)$, 7 is in $dom(h2_1) - dom(h2_2)$ which has a pc child which in unmapped. Similarly, we can not prune both mapping for node 10. In the end, we now have four useful embeddings remaining: f_2, f_3, f_5, f_6 which corresponds to four maximal rewritings R_2, R_3, R_5, R_6 in the figure.

3.2.2 Time Complexity

We discuss the time complexity of QAV problem in the absence of schema using the example in the Figure 3.3.

The example shows that the mappings of the two subtree rooted at node 6 and 10 have two choices each even after the pruning procedure is applied, which results four distinct maximal rewrites. Therefore, the number of optimal output in the worst case would be exponential which implies it is impossible to have an algorithm to solve this problem better than EXPTIME.

Chapter 4

Query Answering Using Views in the Presence of Schema

In this chapter, we study the problem of answering queries using views under schema for the same class of XPath fragments as in the schemaless case and currently consider only acyclic choice-free DTD as database schema. Since DTD provides constraints, we need to consider those with the given query and views in the problem.

As we show in previous chapter, our algorithm in schemaless case is based on containment mapping. When the schema is available, we first solve the containment mapping problem under DTD and then extend our algorithm to find the maximal rewriting.

Without losing the generality, we assume that both Q and V are satisfying with regard to DTD $\triangle$.

4.1 Constraints from Acyclic choice-free DTD

At the beginning, we formally define five types of constraints implied by DTDs as follows.

Definition 4.1 (Sibling Constraints) *Let t be a document tree satisfying DTD. If whenever a node labelled a in t has children labelled with each $b \in B$, it has a child node labelled with c, t satisfies the* Sibling Constraints(SC) $a : B \downarrow c$. *When B is $\varnothing$, the SC is called* child constraint*[18].*

Definition 4.2 (Functional Constraints) *Let t be a document tree satisfying DTD. If no node labelled a in t has two distinct children labelled with b, t satisfies the* Functional Constraints(FC) $a \downarrow b$*[18].*

Definition 4.3 (Cousin Constraints) *Let t be a document tree satisfying DTD. If whenever a node labelled with a in t has descendant labelled with each $b \in B$, it has a descendant node labelled with c, then t satisfies the* Cousin Constraints(CC) $a : B \Downarrow c$.

Definition 4.4 (PC Constraints) *Let t be a document tree satisfying DTD. If whenever there is a path from node labelled with a to a node labelled with b, the path length from a to b is always 1, then t satisfies the* PC Constraints(PC) $a \searrow b$.

Definition 4.5 (Intermediate Node Constraints) *Let t be a document tree satisfying DTD. If whenever there is a path from node labelled a in t to a descendant labelled with c, b must present on this path between a and c. t satisfies the* Intermediate Node Constraints(IC) $a, c :\updownarrow b$.

Sibling Constraints and Functional Constraints were first introduced in [18] where set B contains multiple elements. We prove in Lemma 4.1 and 4.2 that SC and CC are both unary when DTD is choice-free.

Lemma 4.1 *SCs are unary when DTD is choice-free.*

Proof Let DTD be represented in grammar notation. Because SC only involves parent and child relationship, each SC associates with one production. We presents a production using a graph G_P such that the root node is the context node a, dummy nodes D_i are used to factor out nested occurrences if any, leaf nodes are child nodes of a, and quantifies('*', '+', '1', '?') are labels on the edges. We call it *production graph.* Assume that a SC inferred by DTD be $a : B \downarrow c$, B is a set of child nodes of a.

Cases (a) DTD is *duplicate-free*: Clearly, the resulting production graph is a tree because each child element only appear once in a production graph because child node of a appear at most once in the right side of the production. Also c must be connect to G_P with edge labelled with '1' or '+'. Otherwise c can not guarantee to present in any case. Let node φ be the highest ancestor of c such that all edges on the path from φ to c are labelled with '1' or '+'. There are two possibilities:

- (a.1) $\varphi = a$: c is a guaranteed child node of a, therefore $B = \varnothing$ in SC;
- (a.2) $\varphi = D_i$ (some dummy node): This means that if $\varphi \neq \varnothing$ then c must present as a's child. Hence any leaf node b_i except c itself reachable from φ can ensure it. So $B = b_i$. SC $a : b_i \downarrow c$ is unary.

Cases (b) DTD allows *duplicates*: since a child node may appear multiple times in the right side of the production, the production graph may contain a DAG

such that leaf node may have in-degree greater than 1. Same as in Case (a), c must be connect to G_P with edge labelled with '1' or '+' to make SC $a : B \downarrow c$ hold. Let node φ be the highest ancestor of c such that all edges on the path from φ to c are labelled with '1' or '+'. There are also two possibilities:

- (b.1)$\varphi = a$: same as in Case (a.1)$B = \varnothing$ in SC;
- (b.2)$\varphi = D_i$: Since DTD is not duplicate free, a leaf node may be reachable from multiple paths. Hence for every leaf node b_i reachable from D_i: SC $a : b_i \downarrow c$ may not be true if there is a node D_l, unreachable from D_i, but reachable to b_i. So b_i's presence can be independent from c's presence. But Still b_i itself is sufficient to guarantee c's presence without requiring that such D_l exists. SCs are unary.

□

The following is an example of choice-free DTD with duplications. In example (1), there are multiple paths to b and c. SC $a : b \downarrow c$ is not true.

Example: One production of a choice-free DTD with duplications and its production graph: $a \rightarrow ((b?, c)+, (b*, e, c?)?)*$.

In the above example, we find that SC $a : b \downarrow e$ is not true. Although there is one path from a to b through D1 and D3 where the presence of b is related to the presence of e, there is another path from node a to node b through D2 where the presence of b is not relevant to e.

We denote DTD implies a specific constraint c as $\triangle \vDash c$.

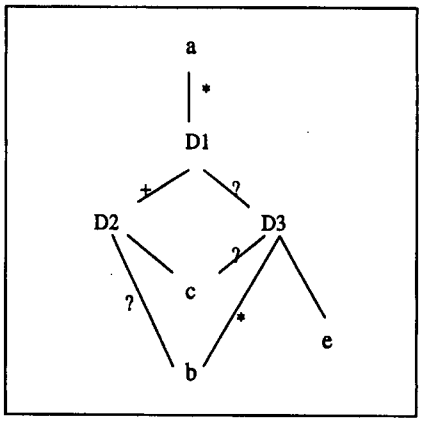

Figure 4.1: Duplicate DTD Example

Claim 4.1 *DTD* $\triangle \models a : b_i \Downarrow c$ *iff* $\forall$ *path* P_j *from* a *to* b_i *in* $\triangle$*:* $\exists$ *a node* d_i *on* P_j *such that there exists a guaranteed path from* d_i *to* c*.*

Proof (If) Assume that every path from a to b_i there is a node d_i such that the path from d_i to c are all labelled with '+' or '1'. Obviously, for any valid instance t of $\triangle$, in any path from a to c in t, c must present as a's descendant. Therefore, $\triangle \models a : b_i \Downarrow c$.

(Only if) Assume that $\triangle \models a : b_i \Downarrow c$ and there exists one path P_i in $\triangle$ from a to c such that there is a node d_j on P_i that has an optional path to c. Then we can create a valid instance t of $\triangle$ in which P_i is selected from a to b_i and c is not present in the path starting at d_j. This is a contradiction. □

Claim 4.2 *If DTD* $\triangle \not\models a : b_i \Downarrow c$ *and* $\triangle \not\models a : b_j \Downarrow c$*, then* $\triangle \not\models a : \{b_i, b_j\} \Downarrow c$*.*

Proof From Claim 1, we know if $\triangle \not\models a : b_i \Downarrow c$ then there must exist a path P_i from a to b_i such that none of node on P_i has a guaranteed path to c. Similarly, if $\triangle \not\models a : b_j \Downarrow c$ then there must exist a path P_j from a to b_j such that none of node

on P_j has a guaranteed path to c. Assume that $\triangle \not\models a : \emptyset \Downarrow c$ and $\nexists b_k$, descendant of a and $\triangle \models a : b_k \Downarrow c$. We can create a tree t by choosing P_i and P_j and extend other paths and nodes as $\triangle$ required to make t valid to $\triangle$. In t, both b_i and b_j are a's descendants, but c is not present as a's descendant under our assumption of $\triangle$. Therefore, $\triangle \not\models a : \{b_i, b_j\} \Downarrow c$. □

Lemma 4.2 *CCs are unary when DTD $\triangle$ is choice-free and acyclic.*

Proof Assume that $\triangle \models a : b_i, b_j \Downarrow c$. From Claim 4.2, we know if $\triangle \not\models a : b_i \Downarrow c$ and $\triangle \not\models a : b_j \Downarrow c$, then $\triangle \not\models a : \{b_i, b_j\} \Downarrow c$. Therefore, CCs are unary. □

4.2 Decidability of Containment Under Acyclic Schema

The correctness of rewritings need to be verified via containment mapping. Therefore, it is necessary to study the containment mapping problem first before solving the problem of answering query using views.

In order to test query containment under a set of constraints C of ICs, PCs, SCs, FCs and CCs for $Q \in XP^{\{/,//,[\ ]\}}$, we introduce a variation of the chase, a procedure for applying constraints in C to V:

1. Change ad edge to pc edge using PC: Let $p \in PC$ of the form $a \searrow b$. For all ad(a,b) in V, change it to pc(a, b) in V.

2. Add guaranteed pc children using SC: Let $s \in SC$ of the form $a : b \downarrow c$, where $B = b_1, ..., b_n$. Let a be a node in V with pc children $b_1, ..., b_n$, and a does not have a pc child labelled c. Then add pc edge(a,c) in V where c is a new node.

3. Merge pc children using FC: Let $f \in FC$ of the form $a \downarrow c$. Let a be a node in V with distinct children c_1 and c_2 labelled as c. Then merge c_1 and c_2 in V. (Note: we will never need to merge ad children. If ad(a,b) is retained in chase V, this means there exist multiple paths from a to b according to D.

4. Add guaranteed intermediate nodes for ad-edges using IC: Let $i \in IC$ of the form $a, c :\updownarrow b$. For all $ad(a, c)$ in (chased) V, insert b between (a, c) using ad edges.

5. Add guaranteed ad children using CC: Let $c \in CC$ of the form $a : b \Downarrow c$. Let a be a node in V with all ad children $b \in B$ and if a has no ad children labelled with c present in V, add c as a's ad child in V where c is a new node.

We denote by $Chase_C(Q)$ the result of applying the set of constraints C to Q.

The set of trees satisfying DTD $\triangle$ is denoted SAT($\triangle$). Let C be a set of ICs, PCs, SCs, FCs and CCs implied by $\triangle$, SAT(C) denotes the set of trees satisfying each constraint in C. The following sequence of results present that C is sufficient and necessary to show $\triangle - containment$ of queries in $XP^{\{/,//,[\ ]\}}$ when $\triangle$ is choice-free and acyclic.

Lemma 4.3 *Let C be a set of ICs, PCs, SCs, FCs and CCs implied by* $\triangle$. $Q \equiv_{SAT(C)} Chase_C(Q)$.

Proof Q $\equiv_{SAT(C)}$ $Chase_C(Q)$ if for any document tree t satisfying C, $Q(t) \equiv Chase_C(Q(t))$.

First we prove that a single application of each chase rule to an XPath query in $XP^{\{/,//,[\ ]\}}$ maintains equivalence w.r.t. C. The result then follows by an induction on the length of a chasing procedure.

1. Chase rule one only applies to ad edges in (chased) Q. Let p be the PC $a \downarrow b$ and Q' be the result applying p to Q. Q' is same as Q except one $ad(x, y)$ will be changed to $pc(x, y)$ in Q'. Obviously $Q' \subseteq Q$ because there is a containment mapping from Q to Q'. So $Q' \subseteq_C Q$. Let $T \in SAT(C)$ and (x, y) in $Q(T)$. Since Q satisfies C, there exists a homomorphism h from Q to T. Since C implies p, T also satisfies p. If a node z labelled a has in T must have a descendant w labelled b, then w must be z's pc child. Hence,h can be extended to a homomorphism from Q' to T without any change. So $Q \subseteq_C Q'$.

2. Chase rule two is applied only to pc edges in (chased) Q. Let s be the SC $a : b \downarrow l$ and Q' be the result of applying s to Q. Q' is Q with one extra pc child u labelled l for some node v labelled a in Q. Clearly $Q' \subseteq Q$ because $\exists$ a containment mapping g from Q to Q'. So $Q' \subseteq_C Q$. Let $T \in SAT(C)$ and (x, y) in Q(T). Since Q satisfies C, there exists a homomorphism h from Q to T. Since C implies s, T also satisfies s. Every node z labelled a in T must have a child w labelled l if z has a child labelled b. Hence, h can be extended to a homomorphism from Q' to T by mapping u to w. So $Q \subseteq_C Q'$.

3. Chase rule three only applies to pc edges in (chased) Q. Let f be the FC $a :\downarrow l$ and Q' be the result of applying f to Q. Q' is Q with pc children $b_1, ..., b_i$ labelled l merged to one node b labelled l for some node v labelled a in Q. Clearly $Q' \subseteq Q$ because $\exists$ a containment mapping g from Q to Q'. So $Q' \subseteq_C Q$. Let $T \in SAT(C)$ and (x, y) in $Q(T)$. Since Q satisfies C, there exists a homomorphism h from Q to T. Since C implies f, T also satisfies f. Every node z labelled a in T have only have a unique child w labelled l. Hence, h can be extended to a homomorphism from Q' to T by replacing

$h(b_1) = l, ..., h(b_i) = l$ with $h(b) = l$. So $Q \subseteq_C Q'$.

4. Chase rule four only applies to ad edges in (chased) Q. Let i be the IC $a, c :\updownarrow b$ and Q' be the result of applying i to Q. Q' is Q with one extra node u labelled b inserted between an ad edge (a, c). Clearly $Q' \subseteq Q$ because there is a containment mapping g from Q to Q'. So $Q' \subseteq_C Q$. Let $T \in SAT(C)$ and (x,y) in $Q(T)$. Since Q satisfies C, there exists a homomorphism h from Q to T. Since C implies i, T also satisfies i. If every node z labelled a in T has a path to w labelled c with length greater than 1, then one node l labelled b must be present in this path. Hence, h can be extended to a homomorphism from Q' to T by mapping u to l. So $Q \subseteq_C Q'$.

5. Chase rule five is applied only to ad edges in (chased) Q. Let c be the CC $a : b \Downarrow l$ and Q' be the result of applying c to Q. Q' is Q with one extra ad child u labelled l for some node v labelled a in Q. Clearly $Q' \subseteq Q$ because $\exists$ a containment mapping g from Q to Q'. So $Q' \subseteq_C Q$. Let $T \in SAT(C)$ and (x,y) in $Q(T)$. Since Q satisfies C, there exists a homomorphism h from Q to T. Since C implies c, T also satisfies c. Every node z labelled a in T must have a descendant w labelled l if z has a descendant labelled with b. Hence, h can be extended to a homomorphism from Q' to T by mapping u to w. So $Q \subseteq_C Q'$.

□

Lemma 4.4 *Let $\triangle$ be a choiceless DTD. $Q \equiv_{SAT(\triangle)} Chase_C(Q)$.*

Proof Since SAT(C) contains SAT($\triangle$), by Lemma 4.3 Lemma 4.4 holds. So we prove the soundness of the chase. □

Lemma 4.5 *Let $\triangle$ be a choiceless acyclic DTD, C be the set of ICs, PCs, SCs, FCs and CCs implied by $\triangle$, and Q be $XP^{\{/,//,[\ ]\}}$ query satisfied with $\triangle$. $Chase_C(Q)$ is 1-1 homomorphic to a subtree of a tree in SAT($\triangle$).*

Proof Because Q is satisfiable with $\triangle$ and the chase is sound, $Chase_C(Q)$ is also satisfiable with $\triangle$; hence there is a non-empty set of trees $S \in SAT(\triangle)$ such that there is a homomorphism from $Chase_C(Q)$ to each tree in S. Assume that $Chase_C(Q)$ 1-1 homomorphic to no subtree of a tree in S. This can only be the case if there is always a pair of child nodes in $Chase_C(Q)$ which are mapped to a single node of a tree in S. Let the child nodes be labelled b and have parent node labelled a. There are three cases in $Chase_C(Q)$:

1. Node labelled a has two pc children labelled b: the FC constraint $a \downarrow b$ must not be implied by $\triangle$. Otherwise, $Chase_C(Q)$ would have merged two b pc children of a node. Then there must exist trees in S with unbounded number of pc children labelled b of node a. Therefore there would be a subtree to which $Chase_C(Q)$is 1-1 homomorphic, a contradiction;

2. Node labelled a has one pc child labelled b and one ad child labelled b: the PC $a \searrow b$ must not be implied by $\triangle$, otherwise ad(a,b) will be chased to pc(a,b) in $Chase_C(Q)$. This means there are trees in S with an path from a-node to b-node with path length greater than 1 and there would be a subtree to which $Chase_C(Q)$ is 1-1 homomorphic, a contradiction;

3. Node labelled a has two ad children labelled b: the only possible failure of homomorphism is that in all trees in S, a node has an unique b node as its descendant, which means there is a only one dtd path p from a to b and each node in p has a unique child node in p, then chains of ICs and FCs would be

implied by this duplicate-free $\triangle$. Thus by the end of chase procedure, the two b nodes would have been merged by using ICs and FCs. $Chase_C(Q)$ is 1-1 homomorphic to some subtree of trees in S, a contradiction.

□

Definition 4.6 (Core Node) *Let Q be $\triangle$-satisfiable, $R \subseteq SAT(\triangle)$, be the set of trees with a subtree that $Chase_C(Q)$ is 1-1 homomorphic to. We call R the satisfying set for Q. Each tree in R has a core subtree to which $Chase_C(Q)$ is 1-1 homomorphic, and each node in the core subtree is called a core node. Each node which is not a core node is called a non − core node.*

From the definition of Core Node, node in $Chase_C(Q)$ may be mapped to a set of nodes $X = x_1, ..., x_i$ in $t \in R$. All $x_i \in X$ are core nodes. In addition, every node lying on the path from one core node x to another core node y is core node.

Lemma 4.6 *Let $\triangle$ be a choiceless acyclic DTD, C be the set of ICs, PCs, SCs, FCs and CCs implied by $\triangle$, and P and Q be $XP^{\{/,//,[\ ]\}}$ queries satisfied with $\triangle$ and $R \subseteq SAT(\triangle)$ in which $Chase_C(Q)$ has 1-1 homomorphism to a subtree in each tree in R. If $P \supseteq_{SAT(\triangle)} Q$, for each node w in P, either w can be mapped to a core node in every tree in R or w can be mapped to a non-core node in every tree in R.*

Proof Since Q is satisfiable with $\triangle$ and chase is sound, $Chase_C(Q)$ is satisfiable with $\triangle$. Hence $R \neq \varnothing$. Assume that $P \supseteq_{SAT(\triangle)} Q$ but there are trees $t_1, t_2 \in R$ such that node w in P can be mapped to only a core node in t_1 and only to a non-core node in t_2. Let $V = v_1, ..., v_n$ be the set of core nodes to which w can be mapped to in t_1. By the definition of core node and the property of R, each node in V also appear in t_2. According to our assumption, w can not map to any

subtree rooted at a node of V in t_2. Because $\triangle$ is context-free, we can replace each v_i tree in t_1 with the corresponding v_i tree in t_2 and obtain tree t_1' which still in set R. However, w can not be mapped to any node in t_1'. Therefore $P(t_1') = \varnothing$ while $Q(t_1') \neq \varnothing$; then $P \supsetneq_{SAT(\triangle)} Q$, is a contradiction. □

Lemma 4.7 *Let C be the set of ICs, PCs, SCs, FCs and CCs implied by $\triangle$, and P and Q be $XP^{\{/,//,[\ ]\}}$ queries. $P \supseteq_{SAT(C)} Q$ iff $P \supseteq Chase_C(Q)$.*

Proof (If) Assume that $P \supseteq Chase_C(Q)$, then $P \supseteq_{SAT(C)} Chase_C(Q)$. From Lemma 4.3, $Q \equiv_{SAT(C)} Chase_C(Q)$, hence $P \supseteq_{SAT(C)} Q$.

(Only If) Assume that $P \supseteq_{SAT(C)} Q$, then for all tree $t \in SAT(C)$, $P(t) \supseteq Q(t)$. $Chase_C(Q)$ is a quasi-instance which satisfies C and may contain ad edges. We can extend $Chase_C(Q)$ to a tree instance t' as following: for each ad edge(x, y), insert a node z in between, (label(z) never appear in $\triangle$); and connect z with x and y using pc edge. Obviously $t' \in SAT(C)$ because z does not involve in C and the extended path x-z-y with length 2 satisfies ad(x,y) obligation. So $P(t') \supseteq Q(t')$. From the chasing procedure defined in previous paragraph, there is a mapping from Q to $Chase_C(Q)$. Hence there is a mapping from Q to t' and $result(t') \in Q(t')$. Then $result(t') \in P(t')$ follows and there exists a mapping c from P to t'. Since z nodes we added in t' never appear in $Chase_C(Q)$ and P, we can easily convert t' back to $Chase_C(Q)$ by replacing every two pc edges connected by z node with an ad edge. c is a mapping from P to node in $Chase_C(Q)$. Hence c is a containment mapping from P to $Chase_C(Q)$, and therefore $P \supseteq Chase_C(Q)$. □

Theorem 4.1 *Let $\triangle$ be choiceless acyclic DTD and C be the set of ICs, PCs, SCs, FCs and CCs implied by $\triangle$. For $XP^{\{/,//,[\ ]\}}$ queries P and Q, $P \supseteq_{SAT(\triangle)} Q$ iff*

$P \supseteq_{SAT(C)} Q$.

Proof (If)Assume $P \supseteq_{SAT(C)} Q$, then $P \supseteq_{SAT(\triangle)} Q$ because $SAT(C) \supseteq SAT(\triangle)$.

(Only if) Assume $P \supseteq_{SAT(\triangle)} Q$ but P $\not\supseteq_{SAT(C)} Q$. We will derive a contradiction. By Lemma 4.5, $Chase_C(Q)$ is 1-1 homomorphic to a subtree of a tree in SAT($\triangle$). Let $R \subseteq SAT(\triangle)$ be the satisfying set for Q. Since $P \supseteq_{SAT(\triangle)} Q$ and there is a homomorphism from Q to each $T \in R$, there must be a homomorphism from P to each $T \in R$.

If P $\not\supseteq_{SAT(C)} Q$, by Lemma 4.7, there is no containment mapping from P to $Chase_C(Q)$. It must be the following two cases (See Figure 4.2) (a) single path in P fail to map to any path in $Chase_C(Q)$ (b) each path in P can map but for some node w which is common ancestor of node u and v, two mapped paths for u and v in $Chase_C(Q)$ can not be joint on w.

Case(a) There is a node x in P with parent y such that y is mapping to a node in $Chase_C(Q)$ but no mapping from x to any node in $Chase_C(Q)$. So in any $T \in R$, x can never be mapped to a core node while y can always be mapped to a core node u. Because $P \supseteq_{SAT(\triangle)} Q$, by Lemma 4.6, x can always be mapped to a non-core node v in every $T \in R$. Since both P and $Chase_C(Q)$ may contain ad edges, there are two possibilities: (1) y is a pc child of x in P. Since v is non-core node and x cannot map to a core node, $\triangle$ cannot imply the SC $label(u) : b_i \downarrow label(v)$, where b_i is the labels of a core child of u. So there must be a tree $U \in SAT(\triangle)$ which has node w with label(u) and child labelled with b_i but no child with label(v). But we can replace the non-core child subtrees of u by the non-core child subtrees of w and still have a tree $T' \in R$. Node x cannot map to any non-core node in T', a contradiction. (2) y is an ad child of x in P. Similar as case (1), v is non-core node and x cannot

map to a core node, $\triangle$ cannot imply neither the SC $label(u) : b_j \downarrow label(v)$ where b_j is the label of core child of u, nor the CC $label(u) : b_k \Downarrow label(v)$ where b_j is the label of core descendant of u. So there must be a tree $U \in SAT(\triangle)$ which has node w with label(u) and descendant labelled with b_j but no descendant with label(v). But we can replace the non-core child subtrees of u by the non-core child subtrees of w and still have a tree $T' \in R$. Node x cannot map to any non-core node in T', a contradiction.

Case(b) Let w_n be the common ancestor of u and v that is closest to the root

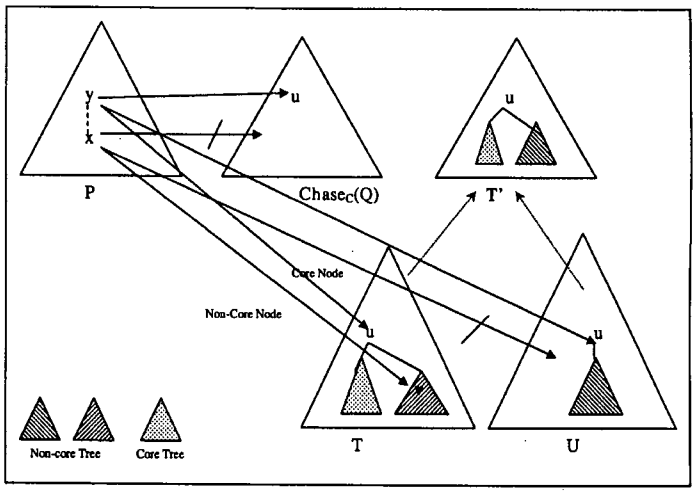

Figure 4.2: Theorem 4.1 Proof - Case a

in P such that the path from root to u via w_n maps to a path in $Chase_C(Q)$ using embedding function f_1 and the path from root to v via w_n maps to a path in $Chase_C(Q)$ using embedding function f_2 but $f_1(w) \neq f_2(w)$(See Figure 4.3). Since $Chase_C(Q)$ has a single root, node w_n can not be root itself. This means the parent of w_n is mapped to the same node w_0 in $Chase_C(Q)$ by f_1 and f_2 and w_0 has a pair

of children with the same label as w_n. There are two possibilities:

1. w_n is a pc child of w_0. Hence $\triangle$ cannot imply the $FC : label(w_0) \downarrow label(w_n)$, this means w_0 can have unbounded number of pc children with $label(w_n)$. Let T be the smallest tree in set R such that the node in T corresponding to w_0 in $Chase_C(Q)$ has only two child nodes with $label(w_n)$. So there is no homomorphism from P to T, a contradiction.

2. w_n is an ad child of w_0. Let the subpath P' from w_0 to w_n in P be $w_0, ..., w_i, w_n$. Hence $\triangle$ cannot imply y has an unique w descendant by the chain of ICs and FCs from w_0 to w_n, i.e. $IC : label(w_i), label(w_{i+2}) :\updownarrow label(w_{i+1})$ $(0 \leqslant i \leqslant n-2)$ and $FC : label(w_i) \downarrow label(w_{i+1})$ $(0 \leqslant i \leqslant n-1)$. Also $\triangle$ cannot imply the $PC : label(w_0) \searrow label(w_n)$ and the $FC : label(w_0) \downarrow label(w_n)$. These means w_0 may have unbounded number of descendants with label w_n. Let T be the smallest tree in R such the node in T corresponding to w_0 in $Chase_C(Q)$ has two distinct paths to two nodes with $label(w_n)$.So there is no homomorphism from P to T, a contradiction.

□

By Lemma 4.7 and Theorem 4.1, we drive the following:

Let $\triangle$ be a choiceless acyclic DTD and C be the set of ICs, PCs, SCs, FCs and CCs constraints implied by $\triangle$. For $XPE^{\{/,//,[\ ]\}}$ queries P and Q, $P \supseteq_{SAT(\triangle)} Q$ iff $P \supseteq Chase_C(Q)$.

4.3 Query Answering Using Views Under DTD

We first introduce several useful definitions.

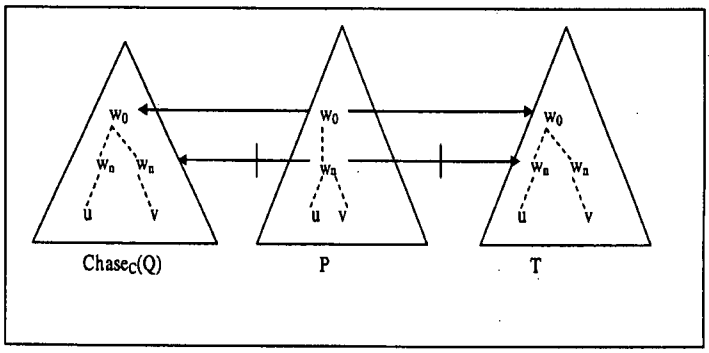

Figure 4.3: Theorem 4.1 Proof - Case b

Definition 4.7 (Sound Rewrite w.r.t. DTD) *We say that Q is* rewritable *using V w.r.t. DTD $\triangle$ if $\exists E \mid \forall t \in SAT(\triangle) : E \circ V(t) \subseteq Q(t)$ and E is said to be a* sound rewrite *of Q using V w.r.t. $\triangle$. (Note E can not be a null expression which means $\exists t \in SAT(\triangle)$ s.t. $E \circ V(t) \neq \varnothing$).*

Definition 4.8 (Useful Embedding w.r.t. DTD) *An embedding $f : N(Q) \rightsquigarrow N(V)$ is said to be a* useful embedding w.r.t. DTD $\triangle$ *if f is a* Useful Embedding *and in addition f satisfies the following constraints: $\forall u \notin dom(f)$: $\exists$ a path from $label(dv)$ to $label(u)$ in $\triangle$.*

Lemma 4.8 *Q is rewritable using V w.r.t. DTD $\triangle$ iff Q is rewritable using $Chase_C(V)$.*

Proof

It follows from Lemma 4.4, $V \equiv_{SAT(\triangle)} Chase_C(V)$. Hence, $E \circ V \equiv_{SAT(\triangle)} E \circ Chase_C(V)$. $Q \supseteq_{SAT(\triangle)} E \circ V$ iff $Q \supseteq_{SAT(\triangle)} E \circ Chase_C(V)$. E is a sound rewrite of Q using V w.r.t. $\triangle$. □

Theorem 4.2 *Q is rewritable using V w.r.t. DTD $\triangle$ iff there exists a useful embedding $f : Q \rightsquigarrow Chase_C(V)$ w.r.t. $\triangle$.*

Proof

(Only if) By Lemma 4.8, Q is rewritable using V w.r.t. $\triangle$ iff Q is rewritable using $Chase_C(V)$. Theorem 3.1 still holds here. Hence, there must exist a useful embedding from Q to $Chase_C(V)$.

(If): Let $f : Q \rightsquigarrow Chase_C(V)$ be a useful embedding w.r.t. $\triangle$. We construct E and extend f as follows:
$\forall x \in Q$ s.t. $x \in dom(f)$ and $\exists y$ s.t. $edge(x, y) \in Q$ and $y \notin dom(f)$. Let T_y denote the subtree rooted at y. Do the following: add a copy T'_y of T_y as a child subtree of d_v and define for every node $z \in T_y, f(z) = z'$ where z' is the corresponding node in T'_y. If (x, y) is a pc(ad) edge, then (d_v, y') is a pc(ad) edge. E contains all such T'_ys. The extended f is the required homomorphism because $\forall x$ defined in the useful embedding is defined in f and $\forall x$ NOT defined in the useful embedding, $f(x)$ is its image in the corresponding T'_y. f is a valid homomorphism function from $Q \rightarrow E \circ Chase_C(V)$. $E \circ Chase_C(V)$ is satisfiable with $\triangle$. Therefore it is a sound rewrite of Q using V. □

When acyclic dtd is present, there is an important property that there are no two nodes on any path with a duplicated tag. Lemma 4.9 is based on this intuition.

Lemma 4.9 *There exists at most one maximal sound rewrite of Q using V w.r.t. acyclic DTD* $\triangle$.

Proof Assume that there are two distinct maximal rewrite E_1 and E_2 which derived from two useful embedding f_1 and f_2. Then there must exist at least one node $q_i \in dom(f_1) - dom(f_2)$. Since in acyclic DTD there is no repeated tags on any root to leaf path in Q and V. It is obvious that CAT is unique. Hence, q_i lying on the branching above CAT in Q. Let $f(q_i)$ is defined in f_1 but not defined in f_2.

Function: get-IC(src,des)

Input: DTD nodes src, des
Output: return a set of DTD nodes A such that $\forall a \in A$ (IC) $src, des :\updownarrow a$.

Check whether IC between src and des has been computed already. If so, return pre-saved result;
For each node n {

1. block all paths connected A;
2. check whether there is a path from src to des;
3. If it is not, add n to the list A; }

Store the result and return A;

Figure 4.4: Finding IC from DTD

Then there are two cases: 1)$f(q_i) \in D_v$: f_2 will not be a sound embedding because $tag(q_i)$ can not appear twice on D_v, a contradiction; 2) $f(q_i) \notin D_v$: according to the definition of useful embedding, the whole branching q_i lying on must be defined in f_1, then f_2 can not be a maximal rewrite, a contradiction. Therefore, neither of these two cases can occurs. Hence, it is impossible to exist two distinct maximal rewrites. □

4.4 Algorithms and Time Complexity

In order to chase on view, we need to find those five types of constraints from DTD. Obviously inferring PC, FC, SC can be done in PTIME because they only involves parent-child relationship corresponding to one production in DTD. However, the cost of inferring ICs and CCs are not trivial. To find these two constraints, we use the PTIME algorithms shown in Figure 4.4 and 4.5, which use depth-first search in DTD graph.

Therefore, the time complexity to get each of five types of dtd constraints

Function: get-CC(src, des)

Input: DTD node src, des, and all the other DTD node
Output: return a set of DTD nodes A such that $\forall a \in A$: (CC)$src, des \Downarrow a$.

Check whether CC between src and des has been computed already. If so, return pre-saved result;
For each node n, which src is its ancestor and des is its cousin{

1. Find all of n's guaranteed ancestors g and block all paths connected g;
2. Check whether there is a path from src to des;
3. If it is not, add n to the list; }

Store the result and return A.

Figure 4.5: Finding CC from DTD

are at most $O(N^2)$ where N is the number of dtd elements in $\triangle$ because we use graph depth first search. To improve the efficiency, we save the computation result of each type of constraints so that we will not repeat the computation of same constraint in the chase procedure when the premise is the same. We next briefly introduce an efficient implementation of the chase in Figure 4.6, which only scans the (Chased) query tree three times. In each scan, we always start from the edges on the distinguished path of the tree.

The Chase Procedure would take time $O((V + N)^2 * N^2)$, where V is the number of nodes in query and N is the number of elements in $\triangle$. This is because finding each constraints from $\triangle$ takes $O(N^2)$ at most; updating query tree each time takes linear time and the number of query node would increase to (V+2N) in the worst case when IC, CC, SC are involved.

Finally we briefly introduce an algorithm to compute useful embedding. The whole algorithm is very similar to the one we present in previous chapter when schema is not present. However, we can further simplify it based on the result of the uniqueness of the rewriting shown in Lemma 4.9. Here are several major modi-

Procedure: FastChase(Query Q, DTD $\triangle$)

Input: Query Q and DTD $\triangle$
Output: $Chase_C(Q)$.

For each ad edge e in Q connected two nodes a and b{

1. If get-PC(a,b, $\triangle$) is true, update e as pc edge;
2. Else If list L = get-IC(a,b, $\triangle$) is not empty, insert each node c in L between a and b in order and preserve pc, ad obligation as implied by $\triangle$;
3. Update Q; }

For each edge e in Q connected two nodes a and b{

1. If e is an ad edge and list B = get-CC(a,b, $\triangle$) is not empty, add each node in B as a ad child of a;
2. If e is a pc edge and list C= get-PC(a,b, $\triangle$), get all other children of a which has same tag name as a in list C {
   (a) If C is not empty, merge all nodes in C with a;
   (b) Update Q; }

   }

For each pc edge e in Q connected two nodes a and b{

1. List D =get-SC(a,b, $\triangle$);
2. If D is not empty {
   (a) For each node c in D, add c as pc child of a if c is not already present in Q and also keep SC chasing on c until saturation;
   (b) Update Q; }

}
Return Q;

Figure 4.6: Apply Chase on Q

fications in the algorithm:

- V would be replaced by $Chase_C(V)$, we use D'_V to present the distinguished path of $Chase_C(V)$;
- In the function **map-Dpath** and **map-To-Dv**, we don't need to keep candidate list anymore. The mapping would be unique as shown in Lemma 4.9 which implies we intend to map some node n in Q, and if there is any node m on D'_v with the same tag, then m is the only candidate to be mapped; otherwise the mapping fails.
- One extra step need to be done: for any node l in Q which is not mapped to V, we need to check whether there is a path from tag(l) to tag(d_V) in $\triangle$. If it is not, the mapping fails.

The time complexity of compute the useful embedding and rewrite in this case would be PTIME. We know map-Subtree takes PTIME, and both map-Dpath and map-To-Dv also take O(V) where V is the number of query nodes. Since the useful embedding is unique, we only need one iteration to go through the query tree.

Chapter 5

Experimental Results

To study the effectiveness of our work, we systematically ran a range of experiments to measure the impact of various parameters. We focus on testing the schema aware case. In addition to measure savings and overhead, we also measure the scalability when executed over large collections of views and test the performance when the query size varies.

We ran our experiments on the XMark benchmark dataset[19]. We constructed the document of size 100MB using the IBM XMLGenerator[9]. We used Wutka DTDparser[10] to parse the DTD, which is needed for static analysis of schema. For query evaluation, we use an XQuery engine XQEngine[11] for convenience and flexibility. Both tools are open sources developed in Java. We implemented our tests in Java as well.

Setup: We ran our experiments on a sparc workstation sunning SunOS version 5.9 with 8 processors each have a speed of 900MHz and 32GB of RAM. All values reported are the average of 5 trials after dropping the maximum and minimum, observed during different workloads.

- **Simple Selection Query**
  Q1: for $a in doc("auction.xml")/site[//person][//quantity]//itemref
  where $a/@item >= "item20"
  return <result> {$a} </result>

  V1: <view>{doc("auction.xml")/site[//profile]//open_auction[privacy]//itemref}</view>

  R1: <result> {for $a in doc("view.xml")/view/itemref[@item>= "item20"] return $a} <result>

  V1': <view>{doc("auction.xml")/site[//person/@id][//privacy]//itemref/@item}</view>

- **Complex Selection Query**
  Q2: for $p in doc("auction.xml")//people/person[//profile[gender/text()="female"][interest]]//address
  where $p/country/text()= "United States" and $p/province/text() = "Maryland"
  return <result> {$p/city}</result>
  V2:<view>{doc("auction.xml")//person[[//profile/gender/text()= "female"][profile/interest][//country/text() = "United States"]] /address }</view>

  R2: for $p' in doc("view.xml")/view/address where $p'/province/text() = "Maryland"
  return <result>$p'/city</result>

  V2': doc("auction.xml")//person[[//profile/gender/text()='female'][//country/text()= "United Sates"]]/address

Figure 5.1: Selection Queries and Views on auction.dtd

5.1 Query Set

We run the tests over the queries and views listed in Figure 5.1 and Figure 5.2. XQueries are labelled with initial "Q", useful Xpath views are labelled with initial "V". These views are used to test saving ratio while their primed variants are useless views which are used to test overhead ratio. All rewritings using given views are equivalent to the original query result. We give the formal definition of saving and overhead ratio in the next section.

Before we show the experimental results, we explain how to set up our experiment to apply our technique of answering XPath queries using XPath views to solve the problem of rewriting XQueries using XPath views by using Q_3 and given

- **Simple Join Query**
  Q3: for $t in doc("auction.xml")/site//closed_auctions/closed_auction[annotation] ,
  $p in doc("auction.xml")//regions/europe[//description//text/bold]/item
  where $t/itemref/@item=$p/@id and $t/price/text( ) >= "100"
  return <result> $t/itemref</result>
  V3a: <view>{doc("auction.xml")//closed_auction[price/text() >= "100"]//itemref }</view>
  V3b: <view> {doc("auction.xml")//europe[//description//bold]//item} </view>
  R3: for $t' in doc("view3a.xml")/view/itemref,
  $p' in doc("view3b.xml")/view/item
  where $t'/@item = $p'/@id
  return <result>$t'</result>
  V3': doc("auction.xml")/europe[//text]//item

- **Complex Join Query**
  Q4: for $p in doc("auction.xml")/site/people/person,
  $t in doc("auction.xml")/site/closed_auctions[//annotation],
  $t2 in doc("auction.xml")/site/regions/europe/item
  where $p/@id = $t/closed_auction/buyer/@person and $t/closed_auction/happiness/text() >="0.6"
  and $t/closed_auction/itemref/@item = $t2/@id
  return <result> {$t2/name/text()} {$p/name/text()}
  </result>
  V4a: <view>{doc("auction.xml") //person[profile/gender/text() = "female"]}</view>
  V4b: <view>{doc("auction.xml")//closed_auction}</view>
  V4c: <view>{doc("auction.xml")/europe/item }</item>
  R4: for $p in doc("view4a.xml")/view/person,
  $t in doc("view4b.xml")/view/closed_auction,
  $t2 in doc("view4c.xml")/view/item
  where $p/@id = $t/buyer/@person and $t/itemref/@item = $t2/@id and $t/happiness/text()
  >="0.6"
  return <result> {$t2/name/text()} {$p/name/text()} </result>
  V4c': doc("auction.xml")/regions[//text]/item

- **Group By Query**
  Q5: for $p in doc("auction.xml")/site/people/person[//age/text()>="40"]
  let $l :=
  for $i in doc("auction.xml")/site/open_auctions[//privacy]/open_auction where
  $p/profile/@income > 5000 * $i/initial/text() return $i return <items>{$l//itemref}</items>
  <person>{$p/name}</person>
  V5a: <view>{doc("auction.xml") //person[//age/text() >= "40"]}</view>
  V5b: <view>{doc("auction.xml")/site[//privacy]//open_auction}</view>
  R5: for $p in doc("view5a.xml")//person
  let $l :=
  for $i in doc("view5b.xml")//open_auction
  where $p/profile/@income > 5000 * $i/initial/text()
  return $i
  return <items>{$l//itemref}</items>
  <person>{$p/name}</person>
  V5b': doc("auction.xml")/site//person[//age/text() <= "35"]

Figure 5.2: Join/Group By Queries and Views on auction.dtd

views in Figure 5.2.

Step 1: Given a query in XQuery expression and a set of views in XPath expression, we build a generalized tree pattern(GTP) [4] for each *independent* variables declared in FOR or LET clauses in Q. A variable is "independent" when its declaration is directly related with document. In Q_3, both $t and $p are independent variables. So we build two separate trees, one for $t and the other for $p.

Step 2: Mark the interest nodes and return nodes in each tree. We capture all nodes appearing in WHERE and RETURN clauses associated with each independent variable in its GTP. Return nodes and those involved in join predicate must be reachable from the distinguished node of the useful view. Figure 5.3 shows two trees Q_t and Q_p constructed for Q3.

Step 3: Each GTP can be represented as a TPQ which is equivalent to a XPath expression Q'. We test Q' against a single view each time. If the view is usable, we search the useful embedding and compute the rewriting. In the example, V3a is usable for Q_t and V3b is usable for Q_p. The rewritings are the following:

- R_t: doc("view3a.xml")/view/itemref
- R_q: doc("view3b.xml")/view/item

Step 4: After we obtain rewriting, we will replace the declarations for each variable with the corresponding rewriting and reassemble the query based on the structure in original query to give the final rewriting.

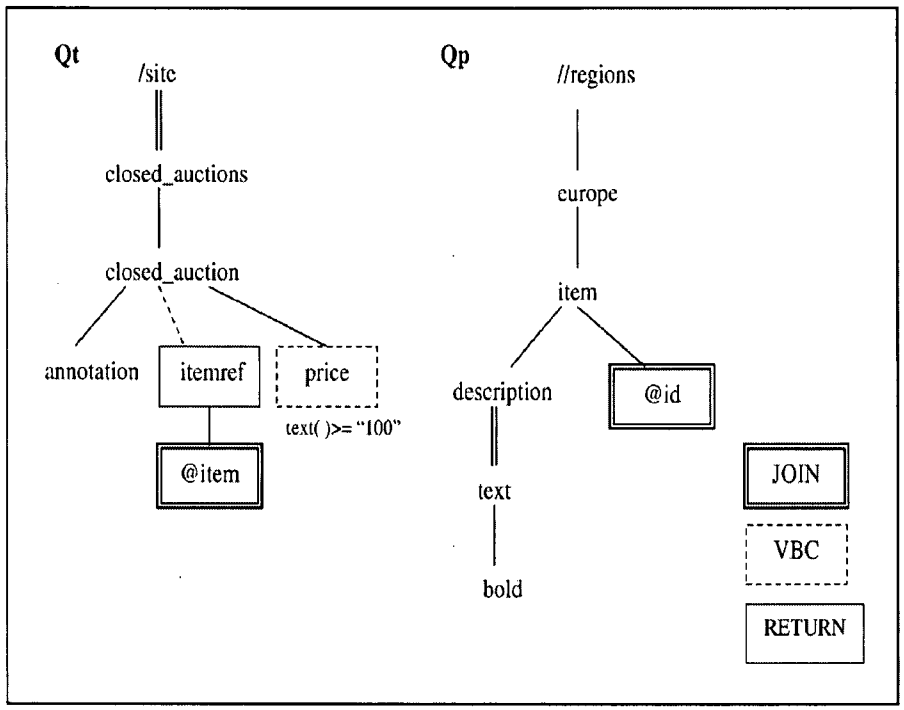

Figure 5.3: GTPs built for Q3

Here is the final rewriting for Q3 using V3a and V3b:

for $t' in doc("view3a.xml")/view/itemref,

$p' in doc("view3b.xml")/view/item

where $t'/@item = $p'/@id

return <result>$t'</result>

5.2 Savings and Overhead on Queries Answering using Views

Let e_q be the time taken to evaluate the original query over the document. Let c_c be the time taken to determine whether a given view is useful for rewriting the query and let c_r be the time to compute the rewrite using the useful views and let e_r be the time it takes to evaluate the rewrites over the materialized view documents. The *saving ratio* S_Q obtained by using the usability check procedure on useful views is defined as $S_Q = \frac{c_c+c_r+e_r}{e_q}$. The *overhead ratio* O_Q obtained by using the usability check procedure on useless views is defined as $O_Q = \frac{c_c+e_q}{e_q}$. Intuitively, the closer to 0 the saving ratio is the better and the closer to 1 the overhead ratio is the better.

5.2.1 Savings on useful views

Figure 5.4 shows the saving ratio with the same document size for the five queries Q1-Q5 using their corresponding useful views. We expect the saving ratio to be close to 0 because the computation time of rewriting is very small. If the document size of each view is less than 1/3 of the size of the original database then the evaluation of the query rewriting is much faster than original query evaluation. This is exactly what happened in the experiments.

5.2.2 Overheads on useless views

Figure 5.5 shows the overhead ratio with same document size for the five queries Q1-Q5 using their corresponding useless views. We expect the overhead ratio to be very close to 1 because the computation time of checking embedding is very small. The result shows the overhead is a negligible fraction compare to the query

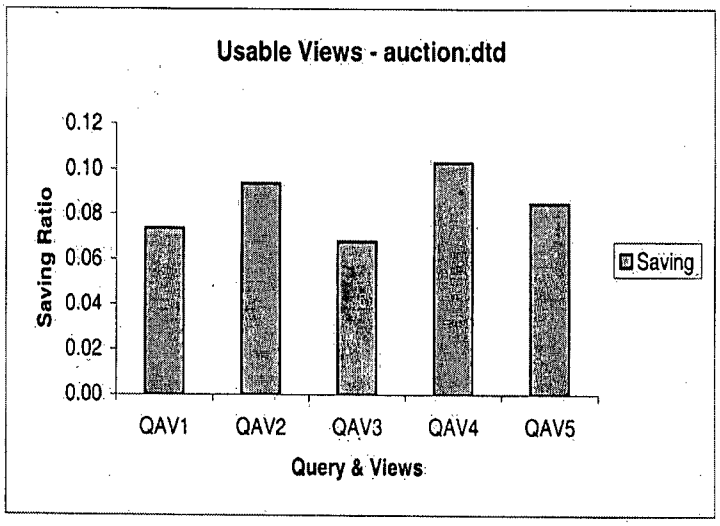

Figure 5.4: Saving Ratio - Useful Views

evaluation time.

5.2.3 Various number of views

Now we test how the performance is when the number of views varies from 1 to 100 and none of view is useful. Figure 5.6 shows the overhead ratio of Q1-Q3 when the number of useless views varies from 1 to 100.

Figure 5.7 shows the saving ratio of Q3-Q5 when the number of views varies from 5 to 100. Each of Q3-Q5 needs multiple views to rewrite the query. We design this test in such a way that there is only one useful view and the others are all useless views. In the rewriting we access original database if no view can be used to extract the required information.

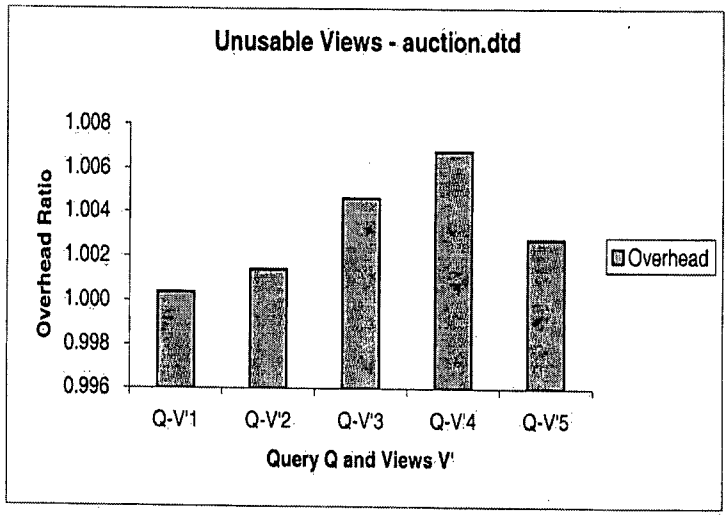

Figure 5.5: Overhead Ratio - Useless Views

5.2.4 Varying query size

Figure 5.8 and Figure 5.9 show the saving ratio and the overhead ratio of a simple join query Q when the query size varies from 5 to 50. We increase the query size by adding more query nodes and value based constraints. When we test saving, the exact number of useful views are provided. When we test overhead, the exact number of useless views are given. We found both ratio did not vary much as the query sizes changed. This may result from the fact that the more complex the query is, the more query evaluation time is required in general although the rewrite computation and evaluation time or the embedding checking time would take longer, the ratio would remain at the same level.

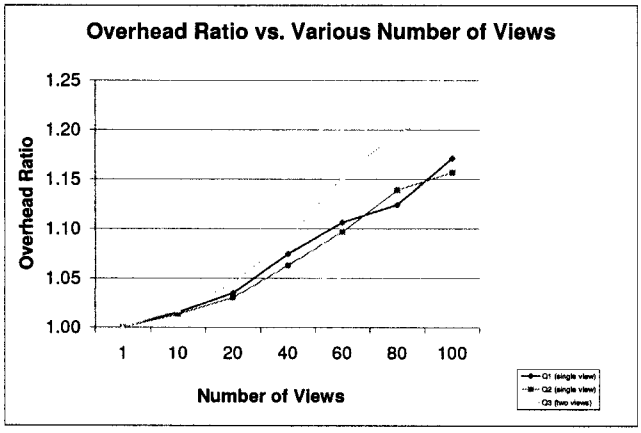

Figure 5.6: Overhead Ratio - Various Number of Useless Views

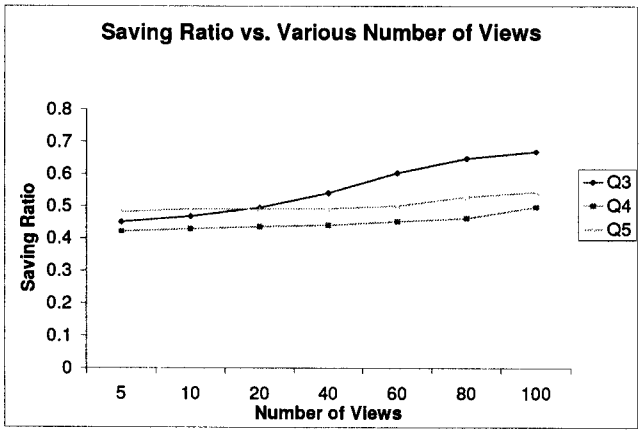

Figure 5.7: Saving Ratio - Various Number of Useless Views

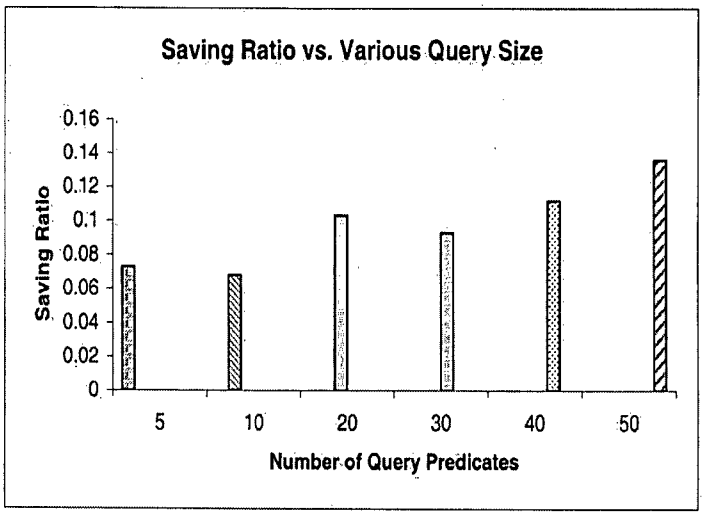

Figure 5.8: Saving Ratio - Various Query Size

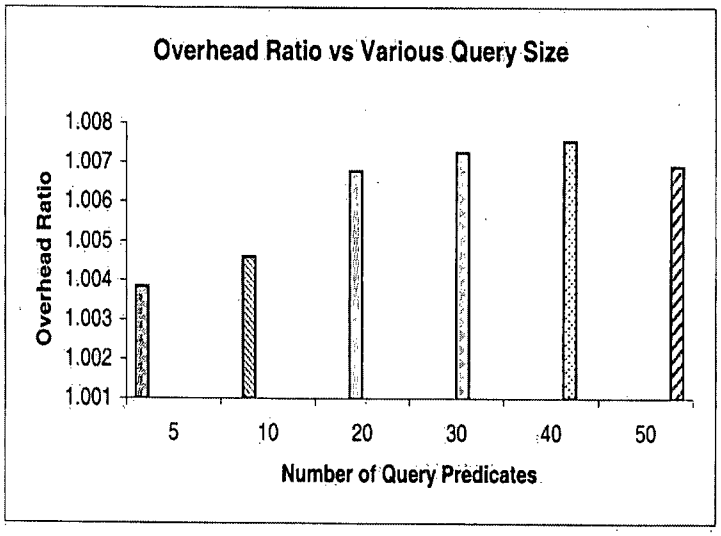

Figure 5.9: Overhead Ratio - Various Query Size

Chapter 6

Related Work

XPath query containment is close related to the use of materialized views in answering query. This relation provides a necessary condition for designing and testing sound algorithm for query rewriting using views. There has been much work on query containment and minimization of various XPath fragments [1, 5, 14, 15]. Containment checking of XPath queries, in the absence of constraints, containment is in PTIME for $XP^{\{/,//,[\ ]\}}$ as shown in [1], while it is proven to be CONP-complete for $XP^{\{//,[\ ],*\}}$ in [14]. Containment under constraints is shown to be undecidable in [5], when $XP^{\{//,[\ ],*\}}$ is allowed along with disjunction, variable binding and equality testing, and the bounded/unbounded simple XPath integrity constraints(SXICs) implied by DTD. A comprehensive study of the complexity of containment of XPath fragments under DTD constraints are given in [15]. The most relevant work is [18], which shows that containment is decidable for $XP^{\{//,[\ ],*\}}$ when the constraints are DTDs. The same paper also identifies $XP^{\{[\ ]\}}$ for which containment under duplicate-free DTDs can be decided in PTIME. In our work, we consider a richer subset of XPath queries, including descendant edges under choice-free acyclic DTDs

and provide PTIME algorithm to decide the containment problem.

Query answering using materialized views for XML is recently studied in [2], where they propose a framework for using XPath views in XML query processing in the absence of schema. However, there are important differences in the contribution of the two papers, as we explain in detail below.

The major contribution of [2] was presenting an XPath matching algorithm to determine certain class of views which can be used to answer query containing XPath expression and construct compensation expressions to be applied on views to produce the query result without schema knowledge. They explored a class of materialized XPath views, which may contain a combination of XML fragments, typed data values, full paths and node references. This means that the users may access to the original database when necessary and the goal is to obtain equivalent results between evaluating query and applying a compensation expression on views. By contrast, we target different applications where the original database is no longer available to the users and it is replaced by as a set of materialized XPath views. Therefore our effort is to produce maximally contained results instead of equivalent results, depending on the given views. More importantly, we classify a class of XPath fragment and DTDs for which we provide an efficient algorithm to decide whether a view is useful for query rewriting and compute the rewrite when it is possible under DTD constraints. In the experiment, we also illustrate the possible use of our work to answer XQueries. To the best of our knowledge, the problem we study here is not addressed in the literature.

Chapter 7

Conclusion

While there has been considerable work on query answering using views in relational world, the same problem has not been extensively studied for XML. We developed a method for testing the usability of XPath view for answering XPath/Xquery queries. We study this problem both with and without a schema and identify cases in which it is EXPTIME and when it is PTIME. In the latter case, we developed efficient algorithms based on a chase procedure and containment mapping. We complemented our analytical results with an extensive set of experiments.

Our study in the presence of database schema is confined to schema without cycles and choices. In the presence of either of them, the reasoning becomes considerably more complex. It would be interesting to determine whether the techniques proposed here can be extended to solve this problem efficiently when there are cycles and/or choices in the database schema. The other direction is to consider more complex XPath queries invloving join, wildcards, etc.

Bibliography

[1] Sihem Amer-Yahia, SungRan Cho, Laks V. S. Lakshmanan, and Divesh Srivastava. Minimization of tree pattern queries. In *ACM SIGMOD Conference*, pages 497–508, 2001.

[2] Andrey Balmin, Fatma Ozcan, Kevin S. Beyer, Roberta Cochrane, and Hamid Pirahesh. A framework for using materialized xpath views in xml query processing. In *VLDB*, pages 64–71, 2004.

[3] Surajit Chaudhuri, Ravi Krishnamurthy, Spyros Potamianos, and Kyuseok Shim. Optimizing queries with materialized views. In *ICDE*, 1995.

[4] Zhimin Chen, H. V. Jagadish, Laks V. S. Lakshmanan, and Stelios Paparizos. From tree patterns to generalized tree patterns: On efficient evaluation of xquery. In *VLDB*, 2003.

[5] Alin Deutsch and Val Tannen. Containment and integrity constraints for xpath. In *KRDB*, 2001.

[6] Jonathan Goldstein and Per ke Larson. Optimizing queries using materialized views: A practical, scalable solution. In *SIGMOD Conference*, 2001.

[7] Alon Y. Halvy. Answering queries using views: A survey. In *Journal VLDB J. Volume 10 Number 4*, pages 270–294, 2001.

[8] Alon Y. Halvy, Zachary Ives, Peter Mork, and Ignor Tatarinov. Piazza: Data management infrastructure for sematic web applications. In *WWW*, pages 556–567, 2003.

[9] IBM. http://www.alphaworks.ibm.com/tech/xmlgenertor.

[10] Wutka Consulting Inc. http://www.wutka.com/dtdparser.html.

[11] Howard Katz. http://xqengine.sourceforge.net.

[12] Laks V. S. Lakshmanan, Ganesh Ramesh, Hui Wang, and Zheng Zhao. On testing satisfiability of tree pattern queries. In *VLDB*, pages 120–131, 2004.

[13] Alon Y. Levy, Alberto O. Mendelzon, Yehoshua Sagiv, and Divesh Srivastava. Answering queries using views. In *PODS*, 1995.

[14] Gerome Miklau and Dan Suciu. Containment and equivalence for an xpath fragment. In *PODS*, pages 65–76, 2002.

[15] Frank Neven and Thomas Schwentick. Xpath containment in the presence of disjuction, dtds, and variables. In *ICDT*, pages 315–329, 2003.

[16] W3C. XML Path Language: XPath Version 1.0. http://www.w3.org/TR/xpath.

[17] W3C. Xquery 1.0: An xml query language. http://www.w3.org/TR/xquery.

[18] Peter Wood. Containment for xpath fragments under dtd constraints. In *ICDT*, pages 300–314, 2003.

[19] XMark. Xmark an xml benchmark project. http://monetdb.cwi.nl/xml.

[20] Markos Zaharioudakis, Roberta Cochrane, George Lapis, Hamid Pirahesh, and Monica Urata. Answering complex sql queries using automatic summary tables. In *SIGMOD Conference*, 2000.