Answering Tree Pattern Queries Using Views

Laks V.S. Lakshmanan
Department of Computer Science, University of British Columbia
2366 Main Mall
Vancouver, Canada
laks@cs.ubc.ca

Hui (Wendy) Wang
Department of Computer Science, University of British Columbia
2366 Main Mall
Vancouver, Canada
hwang@cs.ubc.ca

Zheng (Jessica) Zhao
Amazon Corp.
Seattle, USA
zzhao@cs.ubc.ca

ABSTRACT

We study the query answering using views (QAV) problem for tree pattern queries. Given a query and a view, the QAV problem is traditionally formulated in two ways: (i) find an equivalent rewriting of the query using only the view, or (ii) find a maximal contained rewriting using only the view. The former is appropriate for classical query optimization and was recently studied by Xu and Ozsoyoglu for tree pattern queries (TP). However, for information integration, we cannot rely on equivalent rewriting and must instead use maximal contained rewriting as shown by Halevy. Motivated by this, we study maximal contained rewriting for TP, a core subset of XPath, both in the absence and presence of a schema. In the absence of a schema, we show there are queries whose maximal contained rewriting (MCR) can only be expressed as the union of exponentially many TPs. We characterize the existence of a maximal contained rewriting and give a polynomial time algorithm for testing the existence of an MCR. We also give an algorithm for generating the MCR when one exists. We then consider QAV in the presence of a schema. We characterize the existence of a maximal contained rewriting when the schema contains no recursion or union types, and show that it consists of at most one TP. We give an efficient polynomial time algorithm for generating the maximal contained rewriting whenever it exists. Finally, we discuss QAV in the presence of recursive schemas.

1. INTRODUCTION

With the popularity of XML for data exchange as well as for representing and manipulating semistructured data, there has been substantial work on optimizing XML queries. XPath [29] is the language recommended by W3C for navigation of XML documents and for information extraction. It is a core sublanguage of other major XML query languages like XQuery [30] and XSLT [28]. There has been much work on efficient XPath evaluation [11], indexing techniques [21, 24], structural join algorithms [1, 5], studies on expressive power [4], and containment and equivalence of XPath queries [2, 8, 17, 18, 10, 25].

One of the touted applications of XML is the integration of information from multiple sources. The sources are regarded as views and queries need to be answered using the (materialized) views. This is the well-known query answering using views (QAV) problem. For relational databases, this problem has been studied extensively (e.g., see [16, 13]). In particular, [22] discusses an efficient algorithm for finding maximal contained rewriting for conjunctive relational queries using views. The QAV problem for XML is receiving increasing attention.

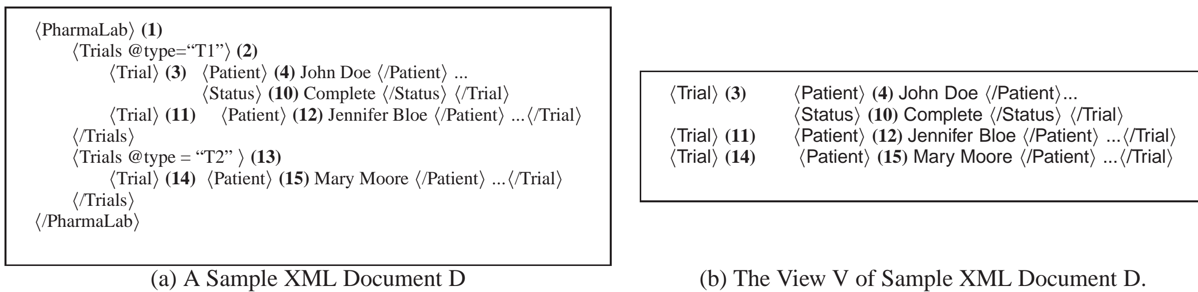
The QAV problem is traditionally formulated in two different ways. The *equivalent rewriting* formulation, motivated by classical query optimization, is given a query Q and view V,find if there is a rewriting of Q using V that is equivalent to Q. Using materialized views for speeding up query processing has been studied in the context of semistructured data for regular path queries [12, 6]. Deutsch and Tannen [9] have studied and characterized the query reformulation problem for XQuery in the context of XML publishing. Chen and Rudensteiner [7] and Yang et al. [27] as well as Balmin et al. [3] use heuristic approaches for using materialized views for speeding up XPath query evaluation. Tang and Zhou [23] and Xu and Ozsoyoglu [26] conduct a theoretical study QAV for XPath fragments corresponding to tree patterns. All these works focus on equivalent rewriting (or its restriction).

However, there are several situations where we cannot find a rewriting that is equivalant to the query because of the data sources' limited coverage, which is very common in information integration scenario [22]. Instead, we search for a *maximally-contained rewriting*, which provides the best possible answer, given the available sources. The problem definition is that given Q and V, find if there is a rewriting of Q using V that is contained in Q (over all possible databases) and is maximal. That is, the rewriting produces sound answers (contained) and no other contained rewriting produces more answers (maximal). It is well-known that contained rewriting is more appropriate for *information integration* [13, 15, 22].

Our focus in this paper is the contained rewriting problem for tree pattern (TP) queries. Tree patterns capture a fragment of XPath, specifically $XP^{/,//,[]}$, consisting of child, descendant, and branching. We illustrate the problem next.

Rewriting without schema: Figure 1(b) shows a materialized view computed by the expression V, "//Trials//Trial" on some database containing clinical trials and patient data. Figure 1(a) shows one possible database D that V's result might have come from, in which we have numbered nodes for easy reference. We consider this D in the rest of this example. The materialized view contains all Trial elements from D, i.e., 3, 11, 14. Consider the query Q, //Trials[//Status]//Trial. Of the two Trials elements (2, 13) in D, only 2 has a Status descendant. So, by applying Q on

Permission to copy without fee all or part of this material is granted provided that the copies are not made or distributed for direct commercial advantage, the VLDB copyright notice and the title of the publication and its date appear, and notice is given that copying is by permission of the Very Large Data Base Endowment. To copy otherwise, or to republish, to post on servers or to redistribute to lists, requires a fee and/or special permission from the publisher, ACM.
VLDB '06, September 12-15, 2006, Seoul, Korea.
Copyright 2006 VLDB Endowment, ACM 1-59593-385-9/06/09.



(a) A Sample XML Document D

(b) The View V of Sample XML Document D.

Figure 1: Example: Maximal Containment Rewriting Without Schema

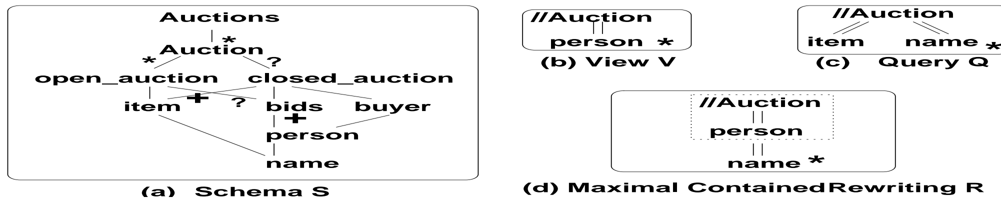


Figure 2: **Example: Maximal Containment rewriting with schema. Rewriting XPath expression for (e) is** //name**.**

D, the two Trial subelements (3, 11) of 2 will be returned. Now suppose only the materialized result of V (Figure 1(b)) is available (as a data source). To answer Q using (only) V, we can apply some *compensation query* E to the result of V. The query is thus rewritten as E ∘ V, where ∘ denotes composition. In our example, a compensation query is ".[//Status]". The composition ".[//Status]"∘"//Trials//Trial" is actually the query R, "//Trials//Trial[//Status]". We call R a *rewriting* (query). It is a *contained rewriting* since R is contained in Q, i.e., on every database, the result of applying R is a subset of that of applying Q. The reason is descendants of Trial are also descendants of Trials. For our example database D, R returns the first Trial element (3) , but not the second (11). Thus, using R we get sound answers to Q but not all of them. As our techniques will show, among all contained rewritings, R is the *maximal contained rewriting* (MCR) in that it is not possible to get more (sound) answers to Q using V. These notions are made precise in Section 2. In this example, we have no knowledge of the schema.

Rewriting with schema: Consider Figure 2(a). It shows a schema for auctions in schematic form. An Auction consists of zero or more (edge label '*') open_auctions and an optional ('?') closed_auction. An open_auction has a mandatory (no label) item and an optional ('?') bids and so on. Consider the view V, //Auction//person and the query Q, //Auction[//item]//name. To obtain the MCR, notice that from person elements returned by V, we can easily extract their descendant names. However, we need to ensure the ancestor Auction of the person has a descendant item. According to the schema (Figure 2(a)), item cannot appear below person, so we have no apparent way of ensuring the ancestor Auction of person has a descendant item. However, the schema has the following constraints: there are three paths from Auction to person, one passing through open_auction and two through closed_auction. Both open_auction and closed_auction have a mandatory child item. *So every* Auction *that has a descendant* person *must have a descendant* item. Thus, we can safely extract the names of the persons returned by V. Thus, we can use the compensation query "//name" and obtain the contained rewriting R, "//name ∘ //Auction//person", i.e., "//Auction//person//name". The details of inferring constraints from the schema will be explained in Section 4. We will show R is the MCR, given the schema of Figure 2(a). R is not equivalent to Q, however, since given a database instance of the schema of Figure 2(a), Q will also find item names but not R.

In general, given a view V in $XP^{/,//,[]}$ which is materialized, and a query Q in $XP^{/,//,[]}$ to be answered, we consider the problem finding the maximal query rewriting both in the absence and presence of a schema. In this paper, we make the following contributions:

- We characterize the existence of contained rewriting in the absence of schema and show that testing existence of MCR can be done in polynomial time (Section 3.1). We also show that in the worst case, the maximal contained rewriting, when it exists, can only be expressed as a union of exponentially many queries in $XP^{/,//,[]}$. This shows that the size of the MCR can be exponential in the size of the query (Section 3.2). We develop an algorithm for generating the MCR, when it exists, in Section 3.3. The algorithm has an exponential worst case complexity, which is also the worst case the MCR size.
- We consider QAV in the presence of a schema without recursion and union types. To obtain the MCR, we extract the essence of a schema using five types of constraints (Sections 4.1 and 4.2). These include the well-known *sibling constraints* [25] as well as new constraints such as *cousin constraints* (e.g., as shown in Figure 2(a), "every Auction having descendant person must also have descendant item"). We provide a chase procedure w.r.t. these constraints and show it preserves equivalence w.r.t. the schema (Section 4.3).
- We characterize the existence of MCR in the presence of schema with no recursion or union types (Section 4.4). Finally, we use the chased view to develop an efficient algorithm for obtaining the MCR of a query w.r.t. a schema. Based on our algorithm, we are able to show that in the presence of a schema, the MCR, when it exists, can be expressed by exactly a single $XP^{/,//,[]}$ query (Section 4.4).
- We discuss issues arising in solving the contained rewriting problem in the presence of recursive schemas (Section 5).

Some background appears in Section 2. Related work appears in Section 6. We summarize the paper and discuss future work in Section 7.

2. PRELIMINARIES

XML Databases & Tree Patterns: An XML *database* is a finite rooted ordered tree $D = (\mathcal{N}, \mathcal{E}, r, \lambda)$, where $\mathcal{N}$ represents element nodes, $\mathcal{E}$ represents parent-child relationship, λ, the labeling function, assigns a tag to each node, and r is the root. In this paper, we do not consider order. Elements may have associated attributes. Attributes and leaf elements have associated values. Figure 1 shows a sample XML database.

A *tree pattern query* (TPQ) [2] is a pair $Q = (N, E)$, where (N, E) is a rooted tree, with nodes in N labeled by tags, and with $E = E_c \cup E_d$ consisting of two kinds of edges, called *pc*-edges (E_c) and *ad*-edges (E_d), corresponding to the child (/) and descendant (//) axes of XPath. A distinguished node in N corresponds to the answer element. Figure 2(b)-(d) are examples of TPQs. In each figure, we identify the distinguished node by placing an asterisk ('*') next to it. E.g., the query Q in Figure 2(c) represents the XPath expression //Auction[//item]//name. TPQs capture the XPath fragment $XP^{/,//,[]}$. Notationally, we write $rel(x, y) \in Q$ to mean that Q contains a *rel*-edge from x to y, where *rel* is one of pc or ad.

Answers for TPQs are captured using matchings. A *matching* of a TPQ Q to a database D is a function $h : Q \to D$ that maps nodes of Q to nodes of D such that: (i) structural relationships are preserved – whenever $pc(x, y) \in Q$, $h(y)$ is a child of $h(x)$ in D and whenever $ad(x, y) \in Q$, there is a path from $h(x)$ to $h(y)$ in D; and (ii) for each node $x \in N$, its tag matches the tag of $h(x)$ in D. We use $\hat{h}(x)$ to denote the element of D rooted at the node $h(x)$, A TPQ may have multiple matchings to a database. The answer to a TPQ Q with distinguished node x on database D is $Q(D) = \{\hat{h}(x) \mid h : Q \to D \text{ is a matching}\}$. Notice that $Q(D)$ is a set of elements.

QAV: Let Q, Q' be any queries. Then Q is contained in Q', $Q \subseteq Q'$, provided for every database D, $Q(D)$ is a subset of $Q'(D)$. For the class of queries considered in this paper, the existence of a homomorphism from Q' to Q is a necessary and sufficient condition for $Q \subseteq Q'$ [2, 17]. Q is *equivalent* to Q', $Q \equiv Q'$, when $Q \subseteq Q'$ and $Q' \subseteq Q$. We write $Q \subset Q'$ to indicate $Q \subseteq Q'$, but $Q \not\equiv Q'$. Let Q be a query and V a view, both in $XP^{/,//,[]}$. Then Q is said to be *answerable using* V provided there is a *compensation query* E such that the *rewriting query* $R \equiv E \circ V$ is contained in Q, and for some database D, $R(D) \neq \emptyset$ [13, 22]. We call this rewriting R a *contained rewriting* (CR) of Q using V.[1] We require CRs to be tree pattern queries, i.e., expressible in $XP^{/,//,[]}$. A *maximal contained rewriting* (MCR) R is a contained rewriting that is maximal, i.e., there is no other contained rewriting R' such that $R \subset R'$. We allow MCRs to be unions of one or more CRs, i.e., unions of expressions in $XP^{/,//,[]}$.

Schema: We study QAV in the presence of a schema. We model schema of XML databases using schema graphs (see Figure 2(a) for an example). A schema graph is a directed edge labeled and node labeled graph $S = (N, E)$. S has a node corresponding to each element of the schema it models. This node is labeled with the element tag. S has an edge (u, v) whenever v is a subelement of u.[2] Edges are labeled by one of the quantifiers '1' (one), '+' (one or more), '?' (zero or one), '*' (zero or more). The default label is '1' and is usually omitted. Additionally, S may have sequence nodes and union nodes. These nodes are unlabeled. Sequence nodes are used to model groups of subelements occurring with a common cardinality. Union nodes are used to model union types. Schema graphs can model DTDs as well as a core fragment of the structural aspects of XML schema. Unless otherwise specified, we consider schemas without union types and recursion. We assume the reader is familiar with the notion of a database conforming to a schema, as defined, e.g., in [18].

Query containment can be relativized to a schema. E.g., let S be a schema and Q, Q' queries. Then Q is S-contained in Q', written $Q \subseteq_S Q'$, provided on all databases D that conform to S, $Q(D) \subseteq Q'(D)$. Other notions follow easily. Let S be a schema, and Q and V be as above. Then Q is *answerable using* V *w.r.t.* S provided there is a *compensation query* E such that the *rewriting query* $R \equiv E \circ V$ is contained in Q w.r.t. S, i.e., $R \subseteq_S Q$, and for at least one database D, $R(D) \neq \emptyset$. Contained and maximal contained rewritings are defined as for the schemaless case, except we use $\subseteq_S$ instead of $\subseteq$.

Problem Statement: We would like to characterize whether a tree pattern query can be answered using a tree pattern view and develop algorithms for testing this as well as for generating the MCR if it exists. We wish to do this both in the absence of a schema and in the presence of a schema. Unless otherwise specified, we assume the schema contains no union types or recursion. We discuss recursive schemas in Section 5.

3. MAXIMAL CONTAINED REWRITING WITHOUT SCHEMA

When no schema is available, we do not have any constraints on which elements may (not) appear as (transitive) subelements of which other elements. This raises the questions: (i) how do we detect if a query is answerable using a view?, (ii) can the MCR be expressed as a TPQ?, (iii) how do we generate the MCR? We address these questions below.

3.1 Testing Existence of Rewriting

Our approach for testing the existence of MCR makes use of the notion of embedding. Let Q and V be tree pattern queries. An *embedding* is a partial matching $f : Q \rightsquigarrow V$ that preserves node tags and structural relationships, and additionally is upward closed: if f is defined on a node $x \in Q$, it is also defined on all ancestors of x in Q.

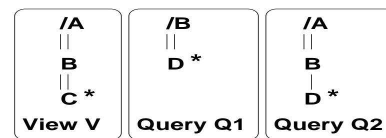


Figure 3: Examples of non-existence of containment mapping

Intuitively, the basic idea of maximal containment rewriting is to find the set of query nodes that don't have an embedding in the view, and appropriately put those nodes under the distinguished node of the view, such that the structural relationships in the query are preserved. Such set of un-embedded nodes can be any (possibly empty) subset of the set of all query nodes. Then the question is, since this set always exists, then does a containment rewriting always exist? Our answer is "no". As an example, Figure 3 shows two queries Q_1 and Q_2 that cannot have any MCR against the view V in the same figure. Q_1 fails because it asks for d elements in an XML document that is rooted at b, while V materializes the c elements in the document rooted at a. Due to mismatching document roots expected, the result of V is useless for Q_1, i.e., Q_1 is not answerable using V. Now consider Q_2. It is straightforward that

[1]In practice, what we really need to answer Q using V, is for the compensation query E to be applied to the materialized results of V. But our analysis requires working with the rewriting query R.

[2]We blur the distinction between elements and attributes.

the d node in Q_2 has no embedding in V. We could try attaching d under the c node in V, which is the distinguished node of V. But the resulting rewriting is not a containment rewriting, since the parent-child relation between b and d node in Q_2 is not preserved in the rewriting.

The distinguished nodes of Q and V are denoted d_Q and d_V respectively. The path from the root of a query Q (resp. view V) to d_Q (resp., d_V) is called the *distinguished path*, denoted as P_Q (resp. P_V). To characterize the existence of contained rewritings, we define *useful embedding*. We first introduce a few notions. Let $f : Q \rightsquigarrow D$ be an embedding. Let $P = (x_1, ..., x_n)$ be a path in Q. We call x_2 a *successor* of x_1 in P and so on. Suppose x_i, $i < n$, is the last node in P such that $f(x_i)$ is defined and $f(x_i) \in P_V$. We call x_i the *anchor* of P w.r.t. f.

DEFINITION 1 (USEFUL EMBEDDINGS). An embedding $f : Q \rightsquigarrow V$ is *useful* provided: (i) f is the empty embedding and the root of Q is qualified with a '//'; OR (ii) (a) $\forall x \in P_Q$, if $f(x)$ is defined, then $f(x) \in P_V$; AND (b) $\forall$ path P in Q, one of the following holds: (I) f is defined on every node in P; OR (II) $\exists x \in P$ such that x is the anchor of P and y its successor in P, and *either* $f(x) = d_V$, the distinguished node of V, *or* $f(x)$ is a descendant of d_V, *or* $ad(x, y) \in Q$. ∎

We illustrate this next. Condition (i) says an empty embedding is useful as long as the query root is qualified with a '//'. If not, the root node *must* be embedded to the root of V, for obtaining a CR, making the embedding non-empty. Condition (ii)(a) says that if every node on P_Q has an embedding in the view V, the target node must be on P_V. Thus the context of d_Q is captured by the context of d_V. Condition (ii)(b) says the anchor node (if any) w.r.t. any query path P must be either mapped to d_V or its descendant, so that its successors that don't have any embedding can be attached below d_V without violation of any query predicates, or it cannot have a query child connected with a parent-child edge (e.g., see b node in query Q_2 in Figure 3 as a counterexample). By studying the Q_2 and V shown in Figure 3 again, if d_V in V is changed to be b instead of c, then we can obtain an MCR of Q_2 by attaching the d node under the b node in V with a parent-child edge, i.e., .[/d]"∘"/a//b[//c].

A useful embedding intuitively captures which query obligations are already fulfilled by the view and which ones are left over. We capture the left-over obligations via the notion of clip-away trees, defined next. Let $f : Q \rightsquigarrow V$ be a useful embedding. Call a node $x \in Q$ a *terminal node* if $f(x)$ is defined and x has at least one (pc- or ad-) child $y \in Q$ such that $f(y)$ is not defined. Figure 4(a)-(b) shows a useful embedding. The embedding is defined only on the Trials node (id 1) in Q. Thus, it is a terminal node: it has two children Status (id 2) and Patient (id 3) which are not embedded into V. For each child y_i of a terminal node x such that $f(y_i)$ is undefined, let T_{y_i} be the subtree of Q rooted at y_i and let T'_{y_i} be the tree obtained by adding a dummy root as the parent of the root of T_{y_i}, the type of the edge connecting the dummy root to the root of T_{y_i} being the same as the edge type of (x, y_i) in Q. Figure 4(c) illustrates T'_2 and T'_3 for the two children 2 and 3 of the terminal node Trials (1) there. Finally, the *clip-away tree* (CAT) induced by the useful embedding f is obtained by merging the dummy roots of all the trees T'_{y_i} identified above and changing the tag of the dummy root to match the tag of the distinguished node of V. E.g., doing so for T'_2 and T'_3 results in the CAT shown in Figure 4(d). Finally, the rewriting R_f induced by f can be obtained by merging the root of the CAT with the distinguished node of V, and marking the distinguished node in R_f based on the distinguished node of Q, as shown in Figure 4(e) (Patient in our example).

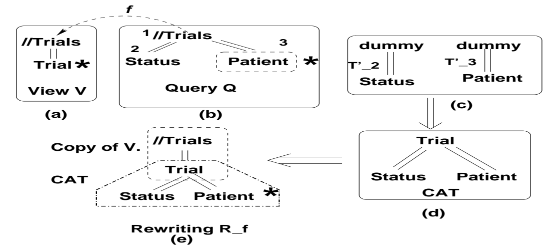


Figure 4: Useful Embeddings, Clip-away Tree, and Rewriting.

Our first result shows that useful embeddings completely characterize the existence of an MCR in the absence of a schema.

THEOREM 1. **[Existence of MCR] :** Let Q and V be tree patterns. Then Q is answerable using V, i.e., there exists an MCR of Q using V, iff there is a useful embedding $f : Q \rightsquigarrow V$. ∎

The proof will appear in the full paper. *In the sequel, we only consider useful embeddings unless otherwise stated.*

Figure 6 presents our algorithm for generating useful embeddings, which is the basis for the test for the existence of MCR. We use the query Q and view V in Figure 5 to illustrate the algorithm. Nodes are numbered (Arabic for V and Roman for Q) in Figure 5. The algorithm consists of 4 key steps.

(1) Assign a *label entry set* for the query root. The root's label entry set is of the form L, where L is a set of node id's from V. If the query root is '//t' the label is the set of the nodes with tag t on the distinguished path of V. E.g., the query root *I* is assigned the label $\{1, 2\}$, i.e., $label(I) = \{1, 2\}$ (Figure 5(a)). If the query root is '/t' then if we can't find a matching root ('/t') in V we exit with failure (line 1.2).

(2) We assign label entries for other nodes x, by making a top-down pass on Q. The latter label entries are of the form $i : L$, where i is a node id in V and L is a set of node id's in V. It says if an embedding maps the parent of x to i, then it can map x to one of the nodes in L. We overload notation and use $label(u)$ to denote the label entry set of any query node u and write $i \in label(u)$ to mean $i \in L$, for some label entry $j : L$ in $label(u)$. If there is no such $j : L$ in $label(u)$, we write $i \notin label(u)$. Line 2.1 adds a label entry $i : L$ to $label(x)$ provided $i \in label(y)$, where y is the parent of x and L is the set of nodes in V to which x can be consistently embedded. Line 2.2 prunes those nodes from L not on P_V whenever x is on P_Q. (Recall P_V is the distinguished path of V.) E.g., in Figure 5(a), we obtain $label(II) = 1 : \{2\},\ 2 : \{\}$. The first label entry says if the root (node *I*) is mapped to the view node 1, then *II* can be mapped to one of 2. Note that we cannot map *II* to 3 since 3 is not on P_V (See Definition 1). The same reasoning explains why the second label entry of *II* in Q is $2 : \{\}$. Other label entry sets are obtained similarly.

Steps (3) and (4) of the algorithm make a bottom-up and a top-down pass respectively, pruning label entries. If the root is '/t' and its label entry set becomes empty, we exit with failure (line 3.2). Otherwise, when the algorithm terminates, we are left with a compact encoding of a set of useful embeddings that can be used to generate the MCR. In the following, for a view node i, whenever $(i : L) \notin label(x)$, for any $L \neq \{\}$, we write $(i : \{\}) \in label(x)$, even when the entry $(i : \{\})$ does not explicitly appear in $label(x)$. This convention is consistent with the meaning of label entries. The pruning rules, used to prune label entries, follow. We next define and explain the pruning rules used in Steps (3) and (4).

Distinguished Path (DP) [used in Step (3) of the algorithm]: Fol-

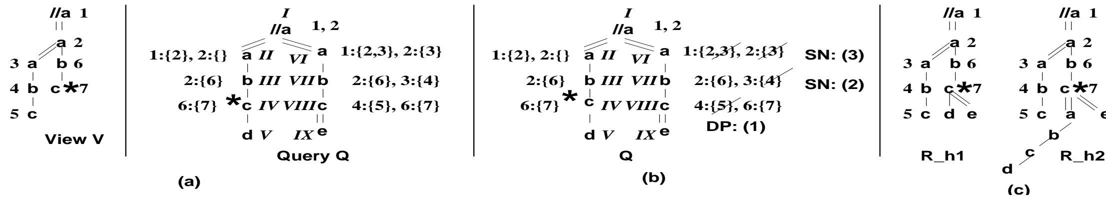


Figure 5: Illustrating the labeling and generation of useful & good embeddings. Nodes numbered partially to avoid clutter.

lowing condition (ii)(a) in Definition 1, the DP rule is designed for the nodes on the distinguished paths (P_Q, P_V) of the query and the view. Suppose $rel(x, y) \in Q$, and $(j : \{\}) \in label(y)$. Then for every entry $i : L \in label(x)$, such that $j \in L$, whenever $j \notin P_V$, then delete j from L. In Figure 5(a), let x and y be nodes *VIII* and *IX*. Then we can remove 5 from the second label entry $4 : \{5\} \in label(VIII)$ using this rule, since $(5 : \{\}) \in label(IX)$ and 5 is not on the distinguished path of V (illustrated in Figure 5(b)). *Thus, this label entry henceforth becomes* $(4 : \{\})$.

Special Nodes (SN) [used in Step (3)]: SN rule is designed for the condition (ii)(b) in Definition 1, focusing on a parent-child edge from an anchor node to its successor. Suppose $pc(x, y) \in Q$ and $(j : \{\}) \in label(y)$. Then for every $(i : L) \in label(x)$ such that $j \in L$, if j is not the distinguished node of V or its descendant, then delete j from L. In Figure 5(b), let x and y be nodes *VII* and *VIII* in Q. Note: $pc(VII, VIII) \in Q$ and $(4 : \{\}) \in label(VIII)$. Using the SN rule, we can delete 4 from the label entry $(3 : \{4\})$ of *VII* since 4 is not the distinguished node of V nor its descendant. This renders the above label entry $(3 : \{\})$. The figure shows the propagation of this up the tree Q in successive applications of the SN rule.

Embedding Rule (ER) [used in Step (4)]: ER rule is applied to construct the general embeddings based on the node tags and structural relationships. Suppose $rel(x, y) \in Q$, $(i : L) \in label(y)$, and $i \notin label(x)$. Here *rel* is pc or ad. Then remove $(i : L)$ from $label(y)$. In Figure 5(a), let x and y be the nodes *II* and *III* in Q. Suppose $label(III)$ also contained the entry $3 : \{4\}$. Then since 3 does not appear in the label entry list of the parent *II*, we can apply this rule to delete the entry $3 : \{4\}$ from $label(III)$. Notice that this rule is always applied from the parent to the child, not vice versa.

Steps (3) and (4) of the algorithm: In step (3), we apply the rules DP and SN bottom-up. Initially, SN is not applicable to any node, while DP is applicable to $x = VIII$ and $y = IX$. Then we can remove 5 from the second label entry $4 : \{5\} \in label(VIII)$, since $(5 : \{\}) \in label(IX)$ and 5 is not on the distinguished path of V (line 3.1). Then successive applications of SN eliminate the entries knocked off in Figure 5(b) and as explained before. No other entries in Figure 5(b) are affected. Step (4) does not change anything so, when it terminates, the algorithm leaves behind the labeled query tree shown in Figure 5(b). Notice that the entry $2 \in label(I)$ cannot be eliminated. Since we did not encounter a case where the root is '$/t$' and has an empty label entry set, we conclude the query has an MCR.

When the algorithm terminates, if it delivers at least one useful embedding, then MCR must exist, else not. The embeddings encoded in the labeling are obtained by following the chain of labels from the query root down. We have the following result, where $|Q|$ denotes the number of nodes in Q.

THEOREM 2. **[Useful Embeddings] :** `Algorithm UseEmb` terminates in time $O(|Q| \times |V|^2)$. It correctly concludes whether a query has an MCR using a view. ∎

The correctness follows from Theorems 1. The complexity is established as follows. Line 1 takes $O(|V|)$ time for initializing

Algorithm UseEmb(query Q, view V) {
1. Determine label of Q's root;
   Let x be Q's root;
1.1. if (x is '//t') $label(x) = \{i \in V \mid i.\text{tag} = \text{'t'}\ \&$ i is on the distinguished path of V }
   else if (x is '/t') {
1.2. if V's root, say i, is '/t') { $label(x) = \{i\}$
   else $label(x) = \{\}$; return('no MCR exists'). }
//* Initialize the label entry lists for the root's descendants. *//
2. Traverse Q top-down, from root's children.
   Let x be current node and y its parent in Q.
2.1. for every $i \in label(y)$ { add the entry $(i : L)$ to $label(x)$, where $L = \{j \in V \mid rel(y, x) \in Q,$ the relationship *rel* between i and j is satisfied in $V\ \&\ j.tag = x.tag\}$.
2.2. if (x is on the distinguished path of Q) delete all $j \in L$ not on the distinguished path of V. }
//* Prune label entries. *//
3. Traverse Q bottom-up. for each node x {
3.1. if SN rule or DP rule is applicable to x, apply it to prune its label entry set;
3.2. if (x is the root, and is '/t', and $label(x)$ becomes empty) return('no MCR exists'). }
//* Prune top-down. *//
4. Traverse Q top-down. Let x be the current node.
   whenever $label(x)$ was updated in step (3), propagate the changes downward, using the ER rule. }

Figure 6: Testing existence of MCR.

the query root label. The loop in line 2 takes time $O(|Q| \times |V|^2)$, since for each edge (y, x) we can compute the initial label for y from that of x in time proportional to the product of the size of y's label and $|V|$, a factor that is upper bounded by $|V|^2$. Lines 3 and 4 can both be completed in $O(|Q| \times |V|)$ time. Thus, the overall time complexity is $O(|Q| \times |V|^2)$.

3.2 MCR Size

In this section, we determine the size of an MCR. Clearly, if we take the union of all possible CRs, the result is guaranteed to be the MCR. However, this is both inefficient and may contain *redundant* CRs, i.e., CRs that are contained in other CRs. A CR generated by an embedding is *irredundant* if it is not contained in a CR generated by any other embedding. We could obtain all CRs, then test for redundancy and then take the union of irredundant CRs. It is easy to see the union of all irredundant CRs is equivalent to the MCR, by definition. We develop a more efficient procedure for constructing the MCR in this section.

When `Algorithm UseEmb` terminates (Figure 6) with a non-empty set of label entries, it leaves a compact encoding of a set of useful embeddings. However, some of them lead to redundant CRs. Thus, first we want to eliminate such embeddings. E.g., in Figure 5(b), there are three embeddings: $h_1 : I \mapsto 1,\ II, VI \mapsto 2,\ III, VII \mapsto 6,\ IV, VIII \mapsto 7;\ \ h_2 : I \mapsto 1, VI \mapsto 2, VII \mapsto 6, VIII \mapsto 7$; and $h_3 : I \mapsto 2$. Each embedding leads to a CR that can be expressed as a $XP^{/,//,[]}$ query. Of these, it can be shown that the CR generated from h_1 contains the CR generated from h_3, so the latter CR is redundant. The CR generated from h_1 is irredundant, but surprisingly, is not the only one. It turns out we

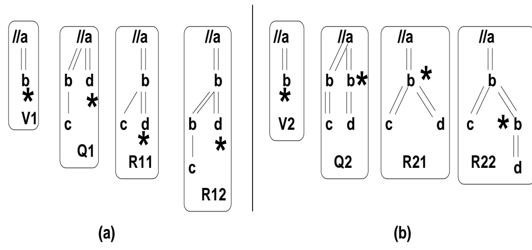


Figure 7: Irredundant CRs.

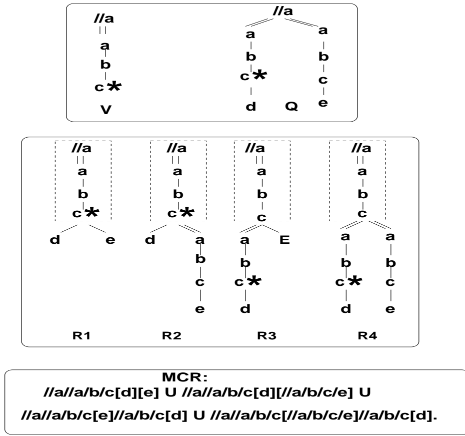


Figure 8: Size of MCR.

can obtain additional irredundant CRs from h_1 itself by choosing not to embed certain nodes. We will later return to this point as well as to the CR generated by h_2.

In this paper, we call an embedding g an *extension* of an embedding f (f a *restriction* of g) provided g is defined on every node that f is defined on (and maybe more). Notice that we do not require that $f(x) = g(x)$ whenever both are defined on x. This nonstandard notion of extension turns out to be the appropriate one for our purposes. We show two more examples of (ir)redundancy in Figure 7(a) and (b), making an important point. Figure 7(a) shows a query Q_1 and a view V_1. There are two interesting embeddings from Q_1 to V_1: (i) f_1: embed both a and b or (ii) f_2: embed only a. The CRs induced by the embeddings – R_{11} and R_{12} – are also shown. Even though f_1 is an extension of f_2, *neither of the CRs is contained in the other*. This example shows the MCR cannot always be expressed as a single TPQ and may need to be expressed as the union of CRs, each being a TPQ.

In Figure 7(b), there are several embeddings from Q_2 to V_2. The embeddings (i) f_1: embed a and both b's, and (ii) f_2: embed a and the left b (but *not* the right b), yield the two CRs R_{21} and R_{22} shown in the figure and they do not contain each other. However, consider the embedding (iii) f_3: embed a and the right b (but not the left b). It is easy to check the resulting CR, $//a//b[//b//c]//d$, is contained in R_{21} and so is redundant. So, sometimes, restrictions of embeddings yield redundant CRs, sometimes not. *This makes obtaining irredundant CRs challenging*. We will address this challenge in the next subsection.

The last question for this subsection is how many irredundant CRs are there in general? This has a bearing on the size of the MCR and thus on the complexity of generating MCRs. We give an example below to show that this number can be exponential in the size of the query.

EXAMPLE 1. **[Size of MCR] :** Figure 8 shows a query Q and a view V, both in $XP^{/,//,[]}$. The MCR of Q using V involves the union of four irredundant CRs. If the root $//a$ in Q has n branches $//a//a/b/c/d_i$, where d_i's are distinct tags, $1 \le i \le n$, then the MCR will be the union of 2^n irredundant CRs. ∎

It is easy to check that the MCR above is not expressible as a single TPQ. Indeed, the XPath standard also includes a self-or-descendant axis, permitting a limited form of disjunction. It is important to note that even with the addition of this axis, we cannot express the above MCR as a single TPQ (enriched with this predicate).

3.3 MCR Generation

We next address the generation problem of the MCR. The main challenge is that there are exponentially many possible embeddings, not all of which yield irredundant CRs. Indeed, some embeddings that are restrictions of others yield irredundant CRs and some don't (Figure 7)! We need to characterize exactly when embedding a node is *not* mandatory for obtaining an irredundant CR so that we can identify the right embeddings that will yield irredundant CRs.

As a motivating example, consider Figure 9. For the query root and its two ad-children b, we have the choice of embedding them into the view V or not. If we do not embed the query root, the resulting CAT will be the query Q itself. The resulting CR $Q \circ V$ is //a//b//a[//b/c]//b[d] is redundant, since it is contained in the CR R_4 in Figure 9. However, when we embed $//a$ but choose not to embed one or more of the b children, the resulting CR is R_2 (don't embed left b), R_3 (don't embed right b), or R_4 (don't embed either b), which are all irredundant.

We have the following result on when *not* embedding a node would still yield an irredundant CR. By a pc-path we mean a sequence of nodes $(x_1, ..., x_k)$, $k \ge 1$, such that there is a pc-edge from x_i to x_{i+1}, $1 \le i < n$. For an embedding f, we say a node $v \in Q$ is *special*, if f maps v to the distinguished node of V, v has a pc-child u in Q, and f is undefined on u. We say that two nodes in Q are *incomparable* provided neither of them is an ancestor of the other. The following technical lemma serves two purposes. First, it lets us eliminate those useful embeddings produced by `Algorithm UseEmb` that will give redundant CRs. Second, it guides us in obtaining all irredundant CRs from the remaining embeddings, by choosing to embed or not, certain query nodes.

LEMMA 1. **[Irredundant CRs] :** Let Q and V be queries in $XP^{/,//,[]}$. Suppose $f : Q \rightsquigarrow V$ is a useful embedding and T is the CAT induced by f. Then the CR $T \circ V$ is irredundant iff: for every node $x \in Q$ for which $f(x)$ is undefined, one of the following holds:

1. there is no extension g of f such that g is defined on x, or

2. $\exists$ a node $z \in Q$: Q contains a pc-path from x to z, and z is special for every extension h of f that is defined on z, or

3. x is the distinguished node of Q, and every extension h of f that is defined on x, maps x to d_V, the distinguished node of V, and there is a node $y \in Q$, incomparable with x, such that $h(y)$ is undefined. ∎

Not all useful embeddings lead to irredundant CRs. In order to generate a compact expression for the MCR, we need to identify embeddings leading to CRs which are not contained in other CRs. We call a (useful) embedding *good* provided it yields an irredundant CR. As an example, in Figure 9, consider the embedding, say f, that is defined only on the root $//a$. There are em-

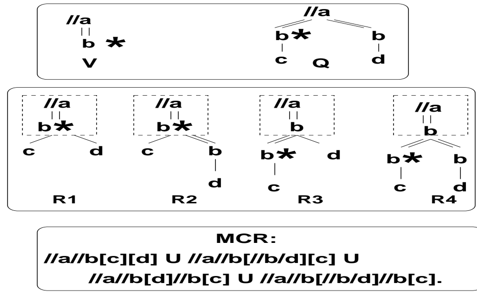


Figure 9: MCR and Illustrating Irredundant CRs.

```
Algorithm MCRGen(query Q, view V) {
  1. Run Algorithm UseEmb (Figure 6);
  2. Generate all embeddings from the final labeled tree Q;
  3. If (there are embeddings f, g s.t. g is an extension of f) {
       if (∃x ∈ Q : x ∈ dom(g) − dom(f)) {
         if (x does not satisfy the conditions in Lemma 1) {
           discard f. } } }
  4. for each remaining embedding h {
       for each (x ∈ Q : h(x) is defined &x doesn't have pc-parent {
         let f be identical to h except it's undefined
           on x and its descendants;
         if x satisfies Condition 2 or 3 in Lemma 1 w.r.t. h and f) {
           mark node x w.r.t. embedding h. } }
         generate all additional embeddings by making
         embedding of marked nodes optional. }
  5. for each embedding generated f {
       produce the CAT corresponding to f and create
         the CR R_f induced by f.}
  return(the union of all CRs generated above). }
```

Figure 10: Generating the MCR.

beddings which extend this by mapping one or both the b's. However, no extension is defined on the pc-child c or d as the case may be. Thus, each of the b nodes acts as both x and z in the theorem. The CAT induced by f is b[//b/d]//b[c] and the resulting CR //a//b[//b/d]//b[c] (R_4 in the figure) is indeed irredundant. Similarly, each of the embeddings used in Figure 9 and Figure 8 can be shown to yield irredundant CRs. The proof of this lemma will appear in the full paper.

Algorithm MCRGen (in Figure 10) makes use of Lemma 1 to produce the MCR. It first obtains all useful embeddings using Algorithm UseEmb (Lines 1-2). It then eliminates those embeddings that will give redundant CRs, by a straight application of Lemma 1 (Step 3). Step 4 is interesting since it uses the same lemma but in a different direction. Finally, it produces the CAT for each embedding it generates and takes the union of the resulting CRs (Step 5).

We illustrate Algorithm MCRGen next. Continuing with the query/view in Figure 5(a), line (1) yields the final labeled query tree in Figure 5(b). We obtain the three embeddings h_1, h_2, h_3 shown in Section 3.2 (line 2). It is easy to see that h_1 is an extension of h_3 and node $VI \in \mathrm{dom}(h_1) - \mathrm{dom}(h_3)$ does not satisfy conditions 2-3 in Lemma 1. So, we drop h_3 (line 3). It turns out h_2 cannot be dropped since there is no $II \in \mathrm{dom}(h_1) - \mathrm{dom}(h_2)$ satisfies Lemma 1. For h_1, we will then mark node *II* (and nothing else) [Line 4]. It turns out that for h_2 no nodes can be marked. The embedding resulting from marking *II* for h_1 is identical to h_1 except it's undefined on *II* and its descendants. This latter embedding happens to coincide with h_2. So, the only two good embeddings are h_1 and h_2. Finally, we generate the two CRs corresponding to h_1 and h_2 (shown in Figure 5(c)) and take their union (line 5). We have the following:

THEOREM 3. **[MCR Generation] :** For a given tree pattern query Q and view V such that Q is answerable using V, Algorithm MCRGen correctly produces the MCR of Q using V. ∎

While Algorithm MCRGen has an exponential worst case time complexity in the size of the query (we know the MCR may be the union of exponentially many tree pattern CRs in the worst case), it tries to minimize the generation of embeddings that are not good, i.e., those that will yield redundant CRs. We now move to QAV in the presence of schema.

4. MAXIMAL CONTAINED REWRITING WITH SCHEMA

We first make precise what it means to rewrite a query using a view in the presence of schema. Let $inst(\mathcal{S})$ denote the set of legal database instances that conform to a given schema $\mathcal{S}$. We say query Q is rewritable using view V (both from $XP^{/,//,[]}$) in the presence of schema $\mathcal{S}$, provided there is an expression E such that on every legal instance $T \in inst(\mathcal{S})$, $E(V(T)) \subseteq Q(T)$, and additionally, there exists $T \in inst(\mathcal{S})$ such that $E(V(T)) \neq \emptyset$. We require E to be expressible in $XP^{/,//,[]}$.

As explained in the introduction, generating MCR of a query using a view in the presence of a schema involves reasoning about the structure of the schema. In this section, we identify the types of constraints that affect QAV for $XP^{/,//,[]}$ (Section 4.1), and develop algorithms: (i) for inferring them from the schema (Section 4.2), (ii) for applying the constraints to the view (Section 4.3), and finally (iii) for generating an MCR if one exists (Section 4.4). Throughout, we assume the schema contains no recursion or union types. We discuss recursive schemas in Section 5.

4.1 Constraints from Schema

As we will show, the essence of a schema can be captured by using five types of constraints on legal instances of the schema. These are defined and explained next. We call a node with tag a, an a node. We have:

Sibling constraint (SC): A *sibling constraint* (SC) [25] is of the form $a : b \downarrow c$, and denotes that whenever an a node has a b child node, then the a node must also have a c child node.

Functional constraint (FC): A *functional constraint (FC)* [25] is of the form $a \rightarrow b$, and says that no a node has more than one b child node.

Cousin constraint (CC): A *cousin constraint* (CC) is of the form $a : b \Downarrow c$, and says that every a node that has a b descendant node must also have a c descendant node.

Parent-child constraint (PC): A *parent-child constraint* (PC) is of the form $a \Downarrow 1\ b$, and says that whenever a b node is a descendant of an a node, it is necessarily a child.

Intermediate node constraint (IC): An *intermediate node constraint* (IC) is of the form $a \xrightarrow{c} b$, and says whenever there is a path from an a node to a b node, there is a c node on the path.

Satisfaction of constraints by an instance is straightforward and is omitted. The notion of legal instance of a schema is similarly omitted here for brevity. The reader is referred to [18] for more details. Of these, SC, FC have been previously studied by Wood [25], whereas the remaining constraints are new. Note that a special case of SC is the constraint $a : \{\} \downarrow c$, which says every a node necessarily has a c child. Similarly, a special case of CC is $a : \{\} \Downarrow c$, which says every a node necessarily has a c descendant. We illustrate the constraints next.

Consider the schema of Figure 2(a). According to the schema, we can observe the following: (1) Every bids must have at least one person node, i.e., bids : $\{\} \downarrow$ person holds in every legal instance of this schema. (2) buyer can only be the child of node

closed_auction, i,e., closed_auction $\Downarrow$1 buyer holds. (3) Every Auction node has at most one closed_auction child, i.e., Auction $\rightarrow$ closed_auction holds. (4) Observe that any path from Auction to person in Figure 2(a) must pass through either a closed_auction node or through an open_auction node. Each of the latter node types is guaranteed to have a descendant of type item. Thus, Auction : person $\Downarrow$ item holds. (5) Suppose the edge from item to name was absent. Then every path from closed_auction to name would pass through person, i.e., closed_auction $\stackrel{\text{person}}{\longrightarrow}$ name would hold.

4.1.1 Properties of Constraints

We say that a constraint σ is implied by a schema $\mathcal{S}$, $\mathcal{S} \models \sigma$, provided every legal instance T of $\mathcal{S}$ satisfies σ. Sibling constraints, as introduced by Wood [25], are more general than those considered here, in that they are of the form $a : S \downarrow c$, where a, b are node tags and S is a *set of tags*. It says if an a node has children corresponding to each tag in S, then it must have a c child. Correspondingly, in general, a schema can imply a cousin constraint of the form $a : S \Downarrow c$, with corresponding meaning. A key result is that for the class of schemas we consider, the cardinality of the set can never be more than one.

LEMMA 2 (SCs & CCs ARE UNARY). Let $\mathcal{S}$ be a schema without union types or recursion and $a : S \downarrow c$ and $a : S \Downarrow c$ any sibling and cousin constraints. Then $\mathcal{S} \models a : S \downarrow c$ iff $\mathcal{S} \models a : b \downarrow c$, for some $b \in S$. Similarly, $\mathcal{S} \models a : S \Downarrow c$ iff $\mathcal{S} \models a : b \Downarrow c$, for some $b \in S$. ∎

We note that Wood [25] proved a similar result for SCs for a different class of schemas called "duplicate free" schemas.

One of the byproducts of the proof of Lemma 2 is the following lemma, which yields an efficient algorithm for inferring CCs from schemas. Define a *guaranteed path* in $\mathcal{S}$ to be a path such that all the edge labels are either '1' or '+'. E.g., in Figure 2(a), there are three paths from closed_auction to name all of which are guaranteed. Basically, then $\mathcal{S} \models a : b \Downarrow c$ iff on every path from a to b there is a node x such that there is a guaranteed path from x to c.

LEMMA 3 (CC CHARACTERIZATION). Let $\mathcal{S}$ be a schema and $a : b \Downarrow c$ a cousin constraint. Then $\mathcal{S} \models a : b \Downarrow c$ iff every path in $\mathcal{S}$ from a to b passes through some node x such that there is a path from x to c all of whose edge labels are not equal to '*', '?'. ∎

In the sequel, unless otherwise specified, by a *constraint*, we mean one of SC, FC, PC, IC, CC. We use Σ to denote the set of constraints implied by a schema $\mathcal{S}$. We will discuss how to derive them from $\mathcal{S}$ in the next section. As a last desirable property of constraints, we have the following result, where $\subseteq_\Sigma$ denotes containment w.r.t. databases satisfying the constraints Σ.

THEOREM 4. **[Containment with Schema] :** Let $\mathcal{S}$ be a schema without recursion and union types and let Σ be the set of constraints implied by $\mathcal{S}$. Let Q, Q' be any two queries in $XP^{/,//,[]}$. Then $Q \subseteq_{\mathcal{S}} Q'$ iff $Q \subseteq_\Sigma Q'$. ∎

This result is important for the QAV problem, specifically for generating the MCR.

4.2 Inference

How do we infer the aforementioned constraints given a schema? Wood [25] gives efficient algorithms for inferring both SCs and FCs. We address the inference of the rest in this section.

For a PC $a \Downarrow 1\ b$, we just need to make sure no path from a to b in the schema $\mathcal{S}$ passes through an element node with a tag $\neq a, b$.

Algorithm extractConstraints($\mathcal{S}$) {
1. $\textit{guarantPath}(x, y) \leftarrow arc(x, y, \ell), (\ell = \text{`}1' \vee \ell = \text{`}+')$.
   $\textit{guarantPath}(x, y) \leftarrow arc(x, z, \ell), (\ell = \text{`}1' \vee \ell = \text{`}+')$,
   $\textit{guarantPath}(z, y)$.
2. $\textit{cousin}(x, y, z) \leftarrow \textit{path}(x, y), \neg\textit{avoid}(x, y, z)$.
   $\textit{avoid}(x, y, z) \leftarrow \textit{arc}(x, y, \ell), \neg\textit{guarantPath}(x, z)$,
   $\neg\textit{guarantPath}(y, z)$.
   $\textit{avoid}(x, y, z) \leftarrow \textit{arc}(x, w, \ell), \neg\textit{guarantPath}(x, z)$,
   $\neg\textit{guarantPath}(w, z), \textit{avoid}(w, y, z)$.
   $\textit{path}(x, y) \leftarrow \textit{arc}(x, y, \ell)$.
   $\textit{path}(x, y) \leftarrow \textit{arc}(x, u, \ell), \textit{path}(u, y)$.
3. $\textit{inter}(x, y, z) \leftarrow \textit{path}(x, y), \neg\textit{bypass}(x, y, z)$.
   $\textit{bypass}(x, y, z) \leftarrow \textit{arc}(x, y, \ell), x \neq z, y \neq z$.
   $\textit{bypass}(x, y, z) \leftarrow \textit{arc}(x, u, \ell), x \neq z, u \neq z, \textit{bypass}(u, y, z)$.
}

Figure 11: Algorithms for Inferring CCs and ICs expressed as Datalog programs.

This can be done easily in polynomial time in the size of $\mathcal{S}$. We omit the details.

Figure 11 gives an algorithm for inferring CCs and ICs. For clarity, we present the algorithms as simple datalog programs, although more efficient implementation is possible. The base predicate $\textit{arc}(x, y, \ell)$ says there is an edge from x to y whose label is ℓ. The program for $\textit{cousin}(x, y, z)$, i.e., for $x : y \Downarrow z$, says this constraint is implied by $\mathcal{S}$ provided there is a path from x to y, but none of these paths avoids those nodes from which there is a guaranteed path to z. The program for $\textit{inter}(x, y, z)$, i.e., $x \stackrel{z}{\rightarrow} y$ says it is implied by $\mathcal{S}$ provided there is a path from x to y and none of the $x \rightarrow y$ paths bypasses z. While more efficient implementations are possible, it is trivial to see that both CCs and ICs can be inferred from a schema $\mathcal{S}$ in time $O(|\mathcal{S}|^3)$, where $|\mathcal{S}|$ denotes the number of nodes in $\mathcal{S}$. Indeed, we have the following result, where for inferring SCs and FCs, we use Wood's algorithms [25]. The complexity bound follows easily from the arity of the Datalog program used to express Algorithm extractConstraints.

THEOREM 5. **[Constraint Inference] :** Given a schema $\mathcal{S}$, all SCs, FCs, CCs, PCs, and ICs implied by $\mathcal{S}$ can be inferred in time $O(|\mathcal{S}|^3)$. ∎

4.3 Chasing the View

Before we can check the existence of MCR, we need to apply the constraints inferred from the schema to the view. We formalize this next by adapting the well-known chase procedure [8]. Given a tree pattern view (query) V and a set Σ of constraints, the *chase* of V w.r.t. Σ, written $\text{Chase}_\Sigma(V)$, is obtained by a repeated application of the following *chase rules* until there is no change to V.

- PC: whenever $ad(a, b) \in V$ and $a \Downarrow 1\ b \in \Sigma$, replace $ad(a, b)$ by $pc(a, b)$ in Q.
- SC: whenever $pc(a, b) \in Q$, $a : b \downarrow c \in \Sigma$, add $pc(a, c)$ to Q if it is not already present.
- FC: whenever Q contains an a node with two pc-children both tagged b and $a \rightarrow b$, merge those two b nodes.
- IC: whenever $ad(a, b) \in Q$ and $a \stackrel{c}{\rightarrow} b \in \Sigma$, then replace $ad(a, b)$ by $ad(a, c)$ and $ad(c, b)$ in Q. That is, insert c as an intermediate node in between a and b using ad-edges.
- CC: whenever $ad(a, b) \in Q$ and $a : b \Downarrow c \in \Sigma$, add $ad(a, c)$ to Q if not already present.

Notice that only when rules SC, IC, CC are applied, new nodes are introduced. The shape of V is always preserved as a tree during the chase. It is easy to see the chase always terminates since there are no cycles in the schema graph. The following result brings out the value of chasing.

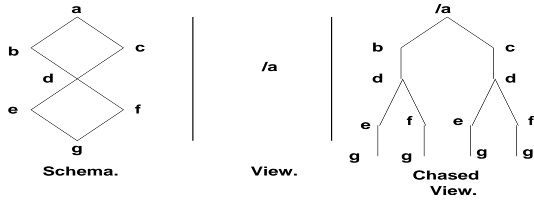


Figure 12: Illustrating explosion of chase with DAG schemas.

THEOREM 6. **[Properties of Chase] :** Let Q and Q' be any TPQs, $\mathcal{S}$ a given schema, and Σ the set of constraints implied by $\mathcal{S}$. Then $Q \subseteq_{\mathcal{S}} Q'$ iff $Q \subseteq_{\Sigma} Q'$ iff $\text{Chase}_{\Sigma}(Q) \subseteq \text{Chase}_{\Sigma}(Q')$. ∎

This theorem suggests a method for testing the existence of MCR for a query using a view, in the presence of a schema. We could chase both the query and the view and then look for useful embeddings. *However, it is important to note that the chase may take time exponential in the schema size in the worst case.* This is because the schema graph is a DAG. An application of chase using all constraints Σ inferred from the schema will result in an explosion in the size of $\text{Chase}_{\Sigma}(V)$. Figure 12 shows such a schema and view, which explodes when chased. Note that in the schema, every node of type a necessarily has a child of type b and a child of type c, and similarly for these latter node types, etc. The view is simply $/a$, whereas the chased view contains 13 nodes! In fact, the figure for "chased view" does not even show all possible nodes that would be added by chasing with redundant constraints implied by $\mathcal{S}$. By stacking the diamonds several times, we could make the size of the chased view exponential in the number of diamonds, and hence in the schema size. Thus, an approach based on exhaustive chase will take exponential time to compute in the worst case. In the next section, we show fortunately we do not have to chase exhaustively.

4.4 Generating MCR

A direct application of "useful embeddings" from the schemaless case will not work when there is schema. The reason is it is possible that the CAT induced by an embedding, when composed with the view, leads to a rewriting query that is unsatisfiable w.r.t. the schema. As an example, consider the query $Q \equiv //a//b$ and the view $V \equiv //b$. Suppose according to the schema $\mathcal{S}$, no a node can be a descendant of any b node.[3] Then the rewriting query $//b//a//b$, while correct from a schemaless perspective, leads to an unsatisfiable query w.r.t. $\mathcal{S}$. This motivates the following.

DEFINITION 2 (USEFUL EMBEDDINGS WITH SCHEMA). Let $Q, V, \mathcal{S}$ be a query, view, and schema, and Σ the set of constraints implied by $\mathcal{S}$. Then an embedding $f : Q \rightsquigarrow \text{Chase}_{\Sigma}(V)$ is a useful embedding provided it satisfies the conditions in Definition 1, and additionally $\forall x \in Q$ such that $f(x)$ is undefined, there is a path from $tag(d_V)$ to $tag(x)$ in the schema graph, where $tag(u)$ denotes the tag of a node u, and d_V is the distinguished node of V. ∎

It is now easy to establish the counterpart of Theorem 1 in the presence of a schema. Embedding from a query to schema graph is defined in the same way as between queries.

THEOREM 7. **[Existence of MCR] :** Let Q, V, and $\mathcal{S}$ be a query, view, and schema respectively. Then Q is answerable using V in the presence of $\mathcal{S}$ iff: (i) there is a useful embedding from $f : Q \rightsquigarrow \text{Chase}_{\Sigma}(V)$, where Σ is the set of constraints implied by $\mathcal{S}$ and (ii) there is a total embedding from $T \circ V$ to the schema graph of $\mathcal{S}$, where T is the CAT induced by f. ∎

[3] This can be inferred by examining the schema graph. However, we don't need to infer such constraints explicitly.

As hinted at in the last section, this theorem does not immediately yield an efficient algorithm for testing the existence of MCR.

We show that by conducting chase in a "goal-directed" fashion, *we can indeed check existence of MCR as well as construct one if it exists, in time polynomial in the sum of sizes of the query, the view, and the schema.* The first observation is that to test whether $Q \subseteq_{\Sigma} Q'$, it suffices to test $\text{Chase}_{\Sigma}(Q) \subseteq Q'$. The reason is that on databases D satisfying Σ, $Q(D) = \text{Chase}_{\Sigma}(Q)$. Thus, we do not need to chase Q'. In the context of QAV, this means it suffices to chase the view alone.

The following lemma is key to showing that we can indeed conduct the chase in a goal-directed way. Intuitively, what we want to do is for each node a in Q but not in V, force it to appear in V by chase if its presence is guaranteed by $\mathcal{S}$. We do not care about nodes that are *not* present in Q. By avoiding those nodes, we can make sure that we only need to conduct the chase at most $|Q|$ times, where $|Q|$ is the number of nodes in Q.

Recall that during the chase either a node type is added to the view, or two nodes are merged, or an ad-edge is converted to a pc-edge. Of these node type addition alone is responsible for the high complexity so we will focus on that. The node type may be added as a leaf (for SC and CC) or as an intermediate node (for IC). In the lemma below, we use $Q - V$ to denote the set of nodes (tags) in Q but not V.

LEMMA 4. **[Intelligent Chase] :** Suppose $\alpha \in Q - V$ be a node type that appears in Q but not in V and let Σ be the set of constraints implied by a schema $\mathcal{S}$. Suppose $\alpha \in \text{Chase}_{\Sigma}(V)$. Then there is a constraint $\sigma \in \Sigma$ such that chasing V w.r.t. σ would add α to V, i.e., $\alpha \in \text{Chase}_{\{\sigma\}}(V)$. ∎

The lemma says there is no need to add a node type ever to the chase unless it also appears in Q. Furthermore, such a node can be added to the chase in one application of a chase rule using some constraint $\sigma \in \Sigma$. This means we can stop the chase when for every node type a originally in $Q - V$, either a has been added to the chase of V, or it can never be added. The latter condition can be easily checked by testing whether any rule that has node type a as a "consequent" (conclusion) can be fired. Thus, this lemma guarantees there is no need to apply the chase rules any more than $|Q - V|$ times.

The above lemma can be proved based on the following facts: (1) each chase rule has at most 2 antecedents and one consequent: e.g., $a : b \Downarrow c$ has antecedents a and b and consequent c; (2) whenever we have a pair of chase steps of the form $\alpha : \beta\, R_1\, \gamma$ and *either* $\alpha : \gamma\, R_2\, \delta$ *or* $\gamma : \beta\, R_2\, \delta$, where R_1, R_2 are any of the chase rules SC, IC, CC, α, β occur in the chase so far, $\delta \in Q - V$ but $\gamma \notin Q$, then we can always add δ to the chase using $\alpha : \beta\, R_3\, \delta$, for some $R_3 \in \{\text{SC, IC, CC}\}$. The lemma is proved from this.

In view of the lemma, if there is a way to add to the chase a node type occurring in Q (but not in V) it is possible to do so directly without having to add node types not occurring in Q. E.g., consider a view $//a//b$. Suppose we use $a \xrightarrow{c} b$ to chase it to $//a//c//b$ and $c : b \Downarrow d$ to derive $//a//c[//d]//b$. Then it is possible for us to directly add the last added node d as a descendant of a, to yield $//a[//d]//b$. The reason is $a \xrightarrow{c} b$ and $c : b \Downarrow d$ together imply $a : b \Downarrow d$. Thus we could completely bypass adding c if we wanted to.

Notice that the complexity of the intelligent chase is $O(|Q - V| \,.\, |V|^2)$: in each step, we search and locate a constraint

```
Algorithm MCRGenSchema(query Q, view V, schema S) {
1.   Use Algorithm extractConstraints to derive constraints Σ implied by S;
2.   for each node type x ∈ Q − V {
2.1.     if (∃ a constraint σ ∈ Σ: chasing V with σ gives x)
             add x to V at the appropriate place; }
3.   Let the result be V′;
//* V′ is actually the result of "intelligent" chase. *//
4.   Use Algorithm UseEmb (Figure 6) to label the nodes of Q using
     label entries and prune them as appropriate;
5.   If (the label entry of the root becomes empty && root is '/t')
        return("no MCR exists").
6.   Pick any embedding encoded by he label entries left on Q's nodes,
     by following the chain of the entries from the root down.
7.   Form the CAT and the CR R using the chosen embedding.
8.   if (there is no total embedding from R to the schema S)
        return("no MCR exists").
9.   return(R). }
```

Figure 13: Testing Existence of and Generating the MCR with Schema.

that might yield the node we wish to add and then test if the chase rule is applicable, a step that costs at most $O(|V|^2)$ operations, since the constraints have at most two antecedents (which have to be located in V) and one conclusion (which is known). The number of iterations is $O(|Q-V|)$. Thus, the intelligent chase plays a key role in our overall polynomial time algorithm for testing existence of MCR and eventually generating MCR in the presence of schema.

We finally give the algorithm for testing the existence of MCR in the presence of schema.

EXAMPLE 2. **[Size of MCR] :** Consider the schema, query, and view shown in Figure 2(a),(b), and (d). Using `Algorithm extractConstraints`, we can infer several constraints from the schema (line 1), including 1.person : {} ↓ name, 2.item : {} ↓ name, 3.closed_auction : {} ⇓ name, 4.open_auction : {} ⇓ name, 5.Auction : person ⇓ item, etc. Note that item is the only node in $Q - V$ and it can be added in one step using constraint 5 above (line 2). This leads to the view in Figure 2(c) (called V' in line 3). In line 4, we will determine that nodes Auction and item in Q can be mapped to the corresponding nodes in V'. More precisely, the label of the query root will be 1 and that of item will be $(1 : \{2\})$, where 1 (2) is the node id of Auction (item) node in V'. Line 5 does not apply and line 6 generates the only embedding encoded in the labeling, as described above. In line 7, the corresponding CAT is the tree person//name. This leads to the CR Auction[//item]//person//name. In line 8, we check that there is indeed a total embedding from this CR to the schema $\mathcal{S}$ of Figure 2(a). So, in line 9, we output R, namely Auction[//item]//person//name. Notice that we have written the rewritten query R using V'. Dropping nodes from R that were not present in V gives the rewriting query R', namely Auction//person//name, which is equivalent to R in the presence of schema $\mathcal{S}$. ∎

The next example illustrates some nuances of the above algorithm.

EXAMPLE 3. **[Size of MCR] :** Consider Figure 14, showing the schema (a), view (b), query (d), view chased using intelligent chase (c), and MCR (e). Note that during the chase, we cannot distinguish between the two bids nodes. Thus, we chase them uniformly, thus creating the same substructure below each bids node. The query is labeled using `Algorithm UseEmb`. Note that the label of the distinguished node bids only includes $(1 : \{5\})$ and not $(1 : \{4\})$, just as for the schemaless case. This is because this node must be mapped on to the distinguished path by any useful embedding. Figure 14(d) shows the result after Algorithm UseEmb terminates, i.e., after labels are pruned. At this point, the label entries in Q encode two embeddings from Q to the chased view V'. We can pick either of them. No matter which one we pick, notice that it embeds away all the nodes of Q. Thus, the CAT tree is just the trivial tree with one node bids, i.e., CAT corresponds to the identity compensation query. We need to compose it with the *original* view V, to get the rewriting query shown in Figure 14(e). The reader is invited to check it is indeed the maximal contained rewriting. ∎

The following theorem is key to the correctness and efficiency of the above algorithm.

THEOREM 8. **[Uniqueness of embedding] :** Let Q, V, and $\mathcal{S}$ be the input query, view, and schema to Algorithm MCRGenSchema. Then after line 5 of the algorithm, every embedding that is encoded by the label entries at the nodes of Q will embed the same set of nodes of Q. ∎

The theorem says that if at all more than one embedding is encoded by the label entries that are left after pruning is completed, then all of them will be defined on the same set of nodes of Q, although they may not agree on where the nodes are mapped. This is significant, since from the point of view of CAT formation, what matters is which nodes are embedded (or not), not where they are embedded. This means all the embeddings that are left over essentially induce the exact same CAT, indicating they all induce the same CR. Since the embeddings that were pruned by the algorithm generate redundant CRs, this means in the presence of a schema (without recursion or union types), *the MCR of a query using a view consists of at most one tree pattern CR*, a result that is not obvious.

Consider any of the examples in Figures 5, 7, 8, and 9 (from the schemaless section), where the MCR was the union of two or more CRs. In all those cases, either the query, or the view, or the rewriting involved repeating tags on the same path. This will be inconsistent with a non-recursive schema. Thus, the MCR remains a single TPQ.

Theorem 8 is proved using the following steps. (1) If a path of Q is completely embedded into V, then choosing not to embed some of its nodes will lead to redundant CRs. Also, it does not matter where nodes on the path are mapped as long as they are al completely embedded. (2) If a path has a terminal node x (i.e., descendants of x on the path are not embedded), then by definition x and all its ancestors must be mapped to P_V, the distinguished path of V. Since the schema graph contains no cycles (as there is no recursion), each of these nodes must map to a unique node on P_V.

Finally, we can show the following:

THEOREM 9. **[QAV with Schema] :** Algorithm MCRGenSchema correctly decides whether a query Q is answerable using a view V in the presence of schema $\mathcal{S}$. Furthermore, the rewriting query R returned by it is the MCR. The algorithm takes time $O(max\{|\mathcal{S}|^3, |Q|.(|Q|+|V|)^2, |Q-V|.|V|^2\})$. ∎

The first two claims follow from the preceding theorem. The complexity is established as follows. Line 1 (constraint extraction) takes $O(|\mathcal{S}|^3)$ time as already discussed. Line 2 is essentially intelligent chase and so takes $O(|Q-V|.|V|^2)$ time, as discussed earlier. Line 4 takes time $O(|Q|.(\text{size of view})^2)$ and since we are applying this step (i.e., Algorithm UseEmb) to chased view, in the worst case, the size of the view might be $|Q|+|V|$. Line 5 is negligible. Line 8 checks whether the rewriting query R is embeddable into the schema graph $\mathcal{S}$. This can be done by simply checking for every edge in R whether there is a path/edge between corresponding tags in $\mathcal{S}$, as the case may be.

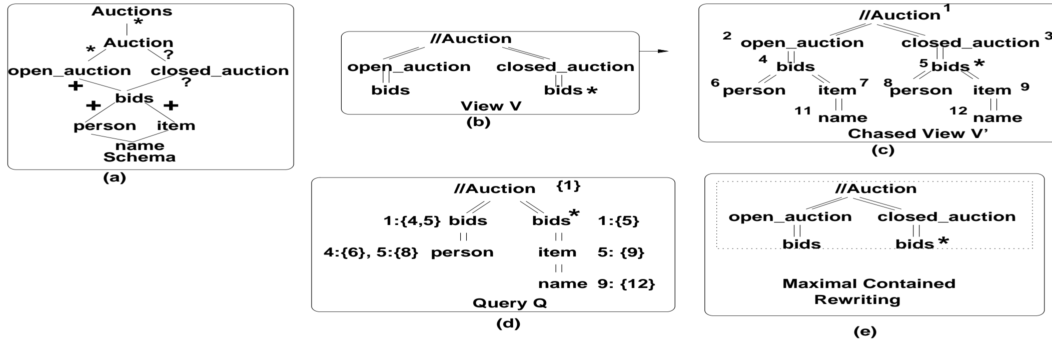


Figure 14: Illustrating Algorithm MCRGenSchema.

In sum, when schema is available, we have shown the MCR is a single TPQ and its existence and generation can both be done in polynomial time.

5. DISCUSSION: RECURSIVE SCHEMAS

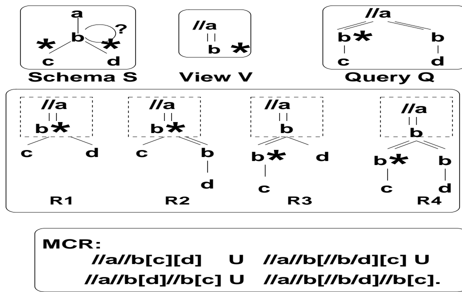


Figure 15: Example: MCR for Schema with Recursions.

QAV in the presence of recursive schemas shares certain common traits with QAV without schemas and also with QAV with non-recursive schemas (without union).

Compared with schema without recursion, the difference for QAV schemas with recursion is that, instead of the MCR consisting of a single TPQ, it may be the union of contain multiple TPQs. E.g. Figure 15 shows a recursive schema and a query Q and view V. The MCR is the union of four TPQs R_1-R_4. The query, view, and CRs are all satisfiable w.r.t. the schema. This example shows that the size of MCR may be exponential when the schema contains recursion. What is the source of this exponentiality? We offer an insight below.

Recall that the exponential size of the MCR with no schema stems from the optionality of embedding, i.e., for certain query nodes that are embeddable in the view, we have the choice of embedding them or not: if we don't use the embedding, the nodes will be put into the clip-away tree (CAT) and be attached below the distinguished node of view. One observation is that such attachment will result in the repetition of node tags on the same path. Thus if such repetition is allowed by the cycles in a schema, MCRs that are unions of exponentially many CRs can also arise for schema with recursion, with a corresponding implication for the worst-case complexity of generating the MCR.

Note that the presence of a recursive schema, just like a non-recursive schema, does impose some constraints on legal database instances. We need to establish whether additional types of constraints are needed for capturing the effect of a recursive schema, on top of the five types needed for the non-recursive case. Secondly, we need to ascertain whether the chase terminates and in how many steps.

For a recursive schema, constraint inference procedure for the five constraint types should be modified as follows. It turns out inference of all constraint types but the PC constraint, remains unchanged. Cycles have the effect of permitting arbitrarily long paths in the instance, with node tag repetitions. This only impacts constraints that are related to path length. Among the five constraint types, PC is the only such type. The modified inference for PC is as follows: for any two nodes p and q, if there is no node between p and q in the schema, and p, q are not in any cycle, i.e., there is no path from p (q) to itself in the schema, then $p \Downarrow_1 q$ holds.

Compared with the MCRGen algorithm for the schemaless case shown in Figure 10, the only modification is in deciding which nodes can have choice of embedding. To be more specific, for every query node p on the distinguished path of query Q, assume its corresponding node in the schema is p', also assume the corresponding node of the distinguished node of view V, d_V, is d_V'. If there exists a schema path from d_V' to p', then we have the choice of embedding the p node or not, i.e., p node can either be embedded into the view or be put into the clip-away tree and be attached under d_V. More work is needed for solving QAV completely for recursive schemas. However, the insights generated from QAV for recursion-free (and union-free) schemas is promising.

6. RELATED WORK

Query answering using views has been studied extensively for the relational model [13]. This problem has been rigorously studied for the class of ' regular path queries [12, 6] and in semistructured graph databases [20]. Deutsch and Tannen have studied the problem of query reformulation in the context of relational-to-XML publishing. They reduce the QAV problem for XQuery to relational query reformulation under constraints. They show that their extended chase and back chase procedure is complete for this problem for a subclass of XQueries. Tang and Zhou [23] conducted a formal analysis of QAV for XPath, but they adopt a tuples-of-nodes semantics, which is at variance with the standard. Xu and Ozsoyoglu [26] specifically focused on rewriting TP queries using materialized views and is the closest to our work by far. They characterized existence of rewrites for various subclasses of $XP^{/,//,[],*}$, which includes wildcards, and analyzed the complexity of finding minimal equivalent rewrites. The major difference between all these works and the problem we consider is that we focus on contained rewriting (which is more suited for information integration) as opposed to equivalent rewriting (more suited for classical query optimization). Besides, neither of [23] or [26] consider the problem in

the presence of schema.

It turns out there are dramatic differences between these problems as far as XPath goes. For instance, [26] show that for $Q, V \in XP^{/,//,[\,],*}$, with roots as distinguished nodes, there exists a rewriting of Q using V iff $Q \subseteq V$. This doesn't hold in our setting. E.g., let Q be $//a$ and V be $//b$. In the absence of any schema knowledge, Q is answerable using V: $Q \circ V$, i.e., $//b//a$ is a contained rewriting. But Q and V are incomparable. In their setting, even in the absence of schema, the rewrite query is always expressible in the same language as query, i.e., there is no need for adding union. In sharp contrast, for us, the rewriting query is in general expressible only as a union of tree pattern queries, which may be exponentially many in the worst case. It is remarkable even without adding wildcard (which together with branching is known to be a source of high complexity even for containment [17]), the complexity of the QAV problem for this class can be high. Analyzing the problem in the presence of a schema, to the best of our knowledge, is novel.

There has been much related work on semantic caching [7], heuristic uses of materialized views for speeding up XPath query processing [3], query evaluation against a set of materialized views in a schema evolution scenario [19], and determining XPath views to materialize based on mining [27]. While the goals are similar to ours at a high level, the methodologies and the results are substantially different from us.

Last but not the least, there has been much work XPath containment and equivalence in the absence and presence of schema (e.g., see [8, 2, 17, 18]). The connection between QAV and containment is well understood. Our embeddings are essentially partial homomorphisms, but they raise challenges that are unique to them: e.g., some embeddings that are restrictions of other embeddings sometimes generate redundant CRs and sometimes irredundant CRs. This needs careful handling. The notion of chase is well known in the database literature. While numerous papers have used it before, it had to be tailored to the particular class of constraints we have, to make the procedure intelligent and efficient.

7. SUMMARY AND DISCUSSION

Motivated by information integration applications, we studied the problem of answering tree pattern queries using tree pattern views by (maximal) contained rewriting. We studied this first in the absence of schema and showed that in the worst case, the size of the maximal contained rewriting can be exponential in the size of the query. However, we are able to test the existence of MCR in polynomial times. We also developed an algorithm for generating the MCR when one exists, based on our characterization of MCR.

When a schema (without recursion and union types) is given, the MCR consists of a single CR which is a TPQ. Besides showing this, we developed algorithms for testing the existence of MCR as well as for generating it. Both of them take polynomial time in the sum of sizes of the query, the view, and the schema. Our approach is based on capturing the essence of semantic information in a schema using five classes of constraints. We also gave a simple algorithm for inferring all constraints implied by a given schema, which is then exploited in the chase.

There are three orthogonal directions for future work: (i) larger query classes (e.g., inclusion of wildcard, limited forms of disjunction, and order), (ii) larger classes of schema (e.g., with union types, recursion, etc.), and (iii) other data integration models such as GLAV [15]. Contained rewriting for XQuery is also an important problem with significant impact.

We conducted a series of experiments to measure the performance gains and the overhead (for testing query answerability) and report the results in [14]. They confirm exactly what one would expect from use of materialized views for query answering: substantial savings and minor overhead.

8. REFERENCES

[1] S. Al-Khalifa et al. “Structural Joins: A Primitive for Efficient XML Query Pattern Matching”, ICDE 2002.

[2] S. Amer-Yahia et al. ”Minimization of Tree Pattern Queries”, SIGMOD 2001.

[3] A. Balmin et al. ”A Framework for Using Materialized XPath Views in XML Query Processing”, VLDB 2004.

[4] M. Benedikt et al. Structural properties of XPath fragments. ICDT 2003. pp. 79-95.

[5] N. Bruno et al. Holistic twig joins: optimal XML pattern matching. SIGMOD 2002. pp. 310-321.

[6] D. Calvanese et al. ”Answering Regular Path Queries Using Views”, ICDE 2000.

[7] L. Chen et al., ”ACE-XQ: A Cache-aware XQuery Answering System”, WebDB 2002.

[8] A. Deutsch and V. Tannen, ”Containment and Integrity Constraints for XPath”, KRDB 2001.

[9] A. Deutsch and V. Tannen., “Reformulation of XML Queries and Constraints”, ICDT 2003, pp. 225-241.

[10] S. Flesca et al., “On the Minimization of XPath Queries.” VLDB 2003, pp. 153-164.

[11] G. Gottlob et al., “XPath Query Evaluation: Improving Time and Space Efficiency”, ICDE 2003. pp. 379-390.

[12] G. Grahne and A. Thomo., “Query Containment and Rewriting Using Views for Regular Path Queries under Constraints”, PODS 2003.

[13] A. Y. Halevy, ”Answering Queries Using Views: A Survey”, VLDB Journal 2001.

[14] L. Lakshmanan et al., “Answering Tree Pattern Queries Using Views: Experiments”, ftp://ftp.cs.ubc.ca/local/laks/papers/qav-experiment.pdf

[15] M. Lenzerini, “Data Integration: A Theoretical Perspective”, PODS 2002.

[16] A. Y. Levy et al. “Answering Queries Using Views”, PODS 1995: 95-104.

[17] G. Miklau and D. Suciu, ”Containment and Equivalence for an XPath Fragment”, PODS 2002.

[18] F. Neven and T. Schwentick, ”XPath Containment in the Presence of Disjunction, DTDs, and Variables”, ICDT 2003.

[19] S.Pal et al., “Managing Collections of XML Schemas in Microsoft SQL Server 2005”, EDBT 2006, pp. 1102-1105.

[20] Y. Papakonstantinou and V. Vassalos., “Query Rewriting Using Semistructured Views”, SIGMOD 1999. pp. 455-466.

[21] S. Paparizos et al., “TIMBER: A Native XML Database”, VLDB J. 11(4): 274-291 (2002).

[22] R. Pottinger and A. Levy, “A Scalable Algorithm for Answering Queries Using Views”, VLDB 2000, pp. 484-495.

[23] J. Tang and S. Zhou, “A Theoretic Framework for Answering XPath Queries Using Views”, XSym 2005: 18-33.

[24] H. Wang et al. “ViST: A Dynamic Index Method for Querying XML Data by Tree Structures”, SIGMOD 2003. pp. 110-121.

[25] P. T. Wood, ”Containment for XPath Fragments under DTD Constraints”, ICDT 2003.

[26] W Xu, Z. M. Ozsoyoglu, ”Rewriting XPath Queries Using Mat erialized Views”, VLDB 2005.

[27] L. H. Yang, M. Lee and W. Hsu. ”Efficient Mining of XML Query Patterns for Caching”, VLDB 2003.

[28] XSL Transformations (XSLT) 1.0, http://www.w3.org/TR/xslt.

[29] XML Path Language (XPath) 2.0, http://www.w3.org/TR/xpath20/.

[30] XQuery 1.0: An XML Query Language, http://www.w3.org/TR/xquery/.