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.

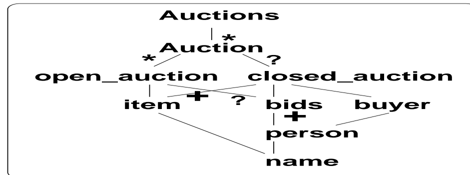
⟨PharmaLab⟩ **(1)**
⟨Trials @type="T1"⟩ **(2)**
⟨Trial⟩ **(3)** ⟨Patient⟩ **(4)** John Doe ⟨/Patient⟩ ...
⟨Status⟩ **(10)** Complete ⟨/Status⟩ ⟨/Trial⟩
⟨Trial⟩ **(11)** ⟨Patient⟩ **(12)** Jennifer Bloe ⟨/Patient⟩ ...⟨/Trial⟩
⟨/Trials⟩
⟨Trials @type = "T2" ⟩ **(13)**
⟨Trial⟩ **(14)** ⟨Patient⟩ **(15)** Mary Moore ⟨/Patient⟩ ...⟨/Trial⟩
⟨/Trials⟩
⟨/PharmaLab⟩

(a) A Sample XML Document D

⟨Trial⟩ **(3)** ⟨Patient⟩ **(4)** John Doe ⟨/Patient⟩...
⟨Status⟩ **(10)** Complete ⟨/Status⟩ ⟨/Trial⟩
⟨Trial⟩ **(11)** ⟨Patient⟩ **(12)** Jennifer Bloe ⟨/Patient⟩ ...⟨/Trial⟩
⟨Trial⟩ **(14)** ⟨Patient⟩ **(15)** Mary Moore ⟨/Patient⟩ ...⟨/Trial⟩

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

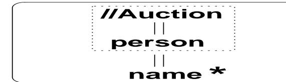
Figure 1: Example: Maximal Containment Rewriting Without Schema



(a) Schema S

(b) View V

(c) Query Q



(d) Maximal Contained Rewriting R

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 $h : Q \rightarrow D$ 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 $Q(D) = \{\hat{h}(x) \mid h : Q \rightarrow D \text{ is a matching}\}$. 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.

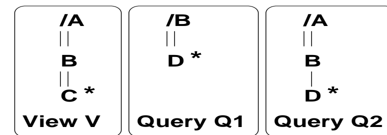


Figure 3: Examples of non-existence of containment mapping

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).

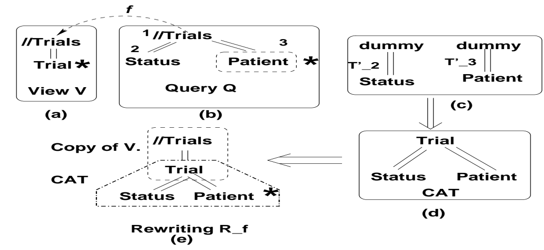


Figure 4: Useful Embeddings, Clip-away Tree, and Rewriting.

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-

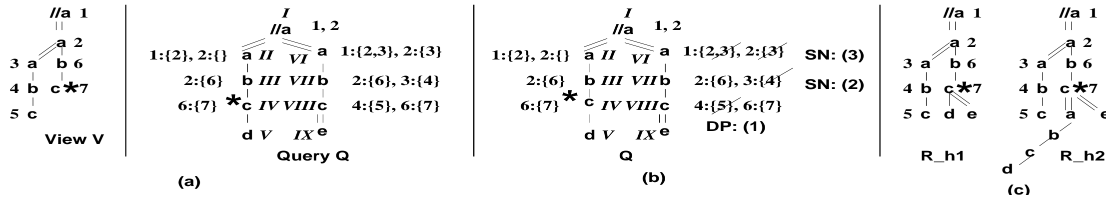
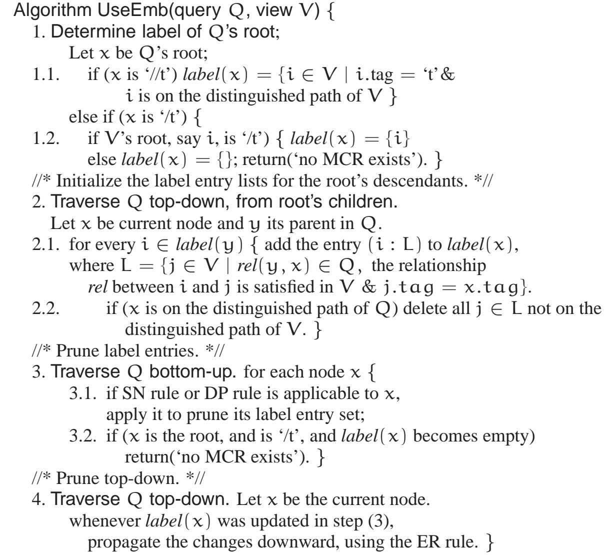


Figure 5: Illustrating the labeling and generation of useful & good embeddings. Nodes numbered partially to avoid clutter.



```
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 ∈ V | i.tag = '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 ∈ label(y) { add the entry (i : L) to label(x),
         where L = {j ∈ V | rel(y, x) ∈ 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 ∈ 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.

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 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

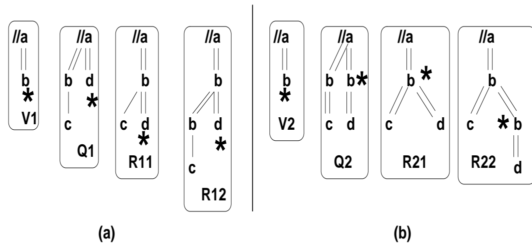


Figure 7: Irredundant CRs.

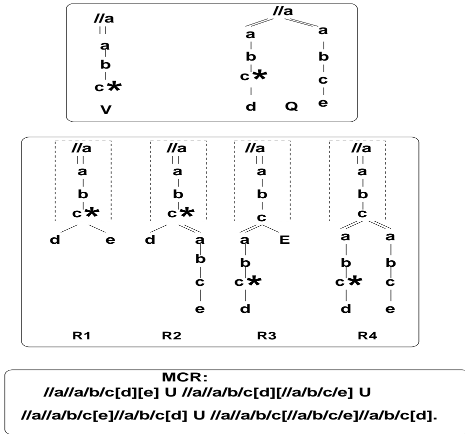


Figure 8: Size of MCR.

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, $1 \leq i \leq n$, 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 $(x_1, ..., x_k)$, $k \geq 1$, such that there is a pc-edge from x_i to x_{i+1}, $1 \leq i < n$. 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-

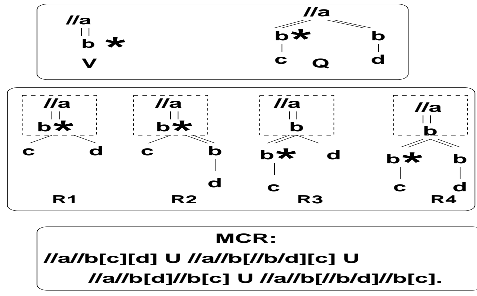


Figure 9: MCR and Illustrating Irredundant CRs.

```
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 : {} ↓ person holds in every legal instance of this schema. (2) buyer can only be the child of node