ABSTRACT

We consider the problem of minimizing tree pattern queries (TPQ) that arise in XML and in LDAP-style network directories. In [Minimization of Tree Pattern Queries, Proc. ACM SIGMOD Intl. Conf. Management of Data, 2001, pp. 497–508], Amer-Yahia, Cho, Lakshmanan and Srivastava presented an $O(n^3)$ algorithm for minimizing TPQs in the absence of integrity constraints (Case 1); $n$ is the number of nodes in the query. Then they considered the problem of minimizing TPQs in the presence of three kinds of integrity constraints: required-child, required-descendant and subtype (Case 2). They presented an $O(n^4)$ algorithm for minimizing TPQs in the presence of only required-child and required-descendant constraints (i.e., no subtypes allowed; Case 3). We present $O(n^2)$, $O(n^4)$ and $O(n^2)$ algorithms for minimizing TPQs in these three cases, respectively, based on the concept of graph simulation. We believe that our $O(n^2)$ algorithms for Cases 1 and 3 are runtime optimal.

Keywords

XML queries, LDAP queries, tree pattern queries, integrity constraints, query minimization, graph simulation

1. INTRODUCTION

In XML and in LDAP-style network directories, data is represented as a tree; associated with each node of the tree is an element type from a finite alphabet $\Sigma$. In XML (see [1]), each node corresponds to an XML element; the children of a node are ordered from left to right, and represent the content (i.e., list of subelements) of that element. In LDAP directories (see [13]), the children of a node are not ordered; the parent-child edges represent hierarchical information.

XML queries in languages such as XPath [24], XQuery [25], XML-QL [8] and Quilt [6] use tree patterns to extract relevant portions from the input database. LDAP directory queries [14] use tree patterns to specify certain structural relationships (child, descendant, etc.) between the desired entries. A tree pattern query that we consider in this paper, denoted by TPQ from now on, was defined in [2] as follows (see Figure 1 for examples). The nodes of a TPQ $Q$ are labeled by element types from $\Sigma$; let $\tau(u)$ denote the type of node $u$. One node of $Q$ is called the output node, and it corresponds to the output of $Q$; it is denoted by $op(Q)$ and is indicated by $\star$ in the figures. There are two kinds of edges: child edges ($c$-edges) and descendant edges ($d$-edges). A $c$-edge from node $u$ to node $v$ is denoted by $u \rightarrow v$ in the text, and by a single edge in the figure; $v$ is called a $c$-child of $u$. A $d$-edge is denoted by $u \Rightarrow v$ in the text, and by a double edge in the figure; $v$ is called a $d$-child of $u$. A $c$-child or $d$-child of $u$ is called a child of $u$.

Any directed acyclic graph (dag), a node $v$ is said to be a descendant of a node if there exists a path (sequence of edges) from $u$ to $v$. In the case of a TPQ, this path could consist of any sequence of $c$-edges and/or $d$-edges.

An embedding of a TPQ $Q$ into a tree database $db$ is a mapping $\beta : Q \rightarrow db$, from the nodes of $Q$ to the nodes of $db$, that satisfies the following conditions:

1. Preserve node types: For each node $u \in Q$, $\beta(u)$ are of the same type.

2. Preserve $c/d$-edge relationships: If $u \rightarrow v$ in $Q$, then $\beta(v)$ is a child of $\beta(u)$ in $db$; if $u \Rightarrow v$ in $Q$, then $\beta(v)$ is a descendant of $\beta(u)$ in $db$.

Note that an embedding could map several nodes of the query (of the same type) to the same node of the database. Answering $Q$ for a given tree database $db$ requires finding all possible embeddings of $Q$ in $db$. The answer to $Q$ is formed from the set of database nodes $\beta(op(Q))$, obtained over all possible embeddings. For LDAP applications, the output consists simply of this set of nodes; for XML applications, the output consists of the subtrees rooted at each of these nodes, placed under one new root labeled result.

Let us consider some examples of TPQs. The query shown in Figure 1b asks for those nodes (in the case of XML, subtrees rooted at those nodes) of type $b$ in $db$ that are children of a node of type $a$, and have a child of type $c$ which in turn has a descendant of type $d$. It corresponds to the XPath expression $a/b[c]/d$. The queries shown in Figure 1a and 1c correspond to the expressions $a[b][c]/[d]$ and $a[b][c]/[d]$, respectively.

In general, the efficiency of finding the result of a query on a given input database depends on the size of the query. So, it is important to minimize the query before attempting to compute the result of the query. [2] pointed out that a TPQ $Q$ may fail to be minimal for one of two reasons:

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

ACM SIGMOD 2002 June 4-6, Madison, Wisconsin, USA

Copyright 2002 ACM 1-58113-497-5/02/06 ...$5.00.
1. $Q$ might contain redundant branches that can be removed, independent of any integrity constraints.

2. Some integrity constraints might be known to hold on the input databases; these constraints might make some branches of $Q$ redundant.

They considered three kinds of integrity constraints (denoted by ICs from now on):

1. Required child: Every database node of type $\tau_1$ has a child of type $\tau_2$, denoted by $\tau_1 \rightarrow \tau_2$.

2. Required descendant: Every database node of type $\tau_1$ has a descendant of type $\tau_2$, denoted by $\tau_1 \Rightarrow \tau_2$.

3. Subtype: Every database node of type $\tau_1$ is also of type $\tau_2$, denoted by $\tau_1 \leq \tau_2$. We follow the convention that $\tau \leq \tau$, for all types $\tau$.

Let us consider some examples of TPQ minimization. In the TPQ shown in Figure 1a, the left branch is made redundant by the right branch; so the TPQ is equivalent to the one shown in Figure 1b. In the absence of ICs, the latter TPQ is minimal. In Figure 1a, if the output node is the left child (instead of the right child) of the root, then the TPQ is minimal in the absence of ICs.

The TPQ shown in Figure 1c is minimal in the absence of ICs. In the presence of the IC $b \rightarrow e$, the TPQ is equivalent to the one shown in Figure 1a; as discussed above, its minimal equivalent is shown in Figure 1b.

The TPQ shown in Figure 1d is minimal in the absence of ICs; it is also minimal in the presence of ICs $\{b \leq b', c \leq c\}$.

In the presence of ICs $\{b \leq b', c \leq c'\}$, its minimal equivalent is the one shown in Figure 1e.

The TPQ shown in Figure 1f is minimal in the absence of ICs. In the presence of the IC $c \Rightarrow d$, its minimal equivalent is the one shown in Figure 1e.

Query minimization is a well-studied area of database systems. One of the first results in this area was that of Chandra and Merlin [7] who showed that for a class of relational database queries, called conjunctive queries, the minimization problem is NP-Complete [11]. Amer-Yahia et al. [2] pointed out that TPQs are essentially a special kind of conjunctive queries on a tree-structured domain. Florescu et al. [10] showed that containment of conjunctive queries with regular path expressions, over semistructured data, is decidable; for some special cases, they showed that the problem is NP-Complete. Query minimization in the presence of constraints has also been studied by several authors. Calvanese et al. [5] studied the problem of conjunctive query containment in the presence of a special class of inclusion dependencies, and established some decidability/undecidability results.

Amer-Yahia et al. [2] presented an $O(n^4)$ algorithm for minimizing TPQs in the absence of ICs, and an $O(n^5)$ algorithm in the presence of ICs; $n$ is the number of nodes of the TPQ. We show (Section 5) that the latter algorithm is incorrect: It might not produce a minimal TPQ when subtype ICs are present. In Section 3, we present an efficient $O(n^2)$ algorithm for minimizing TPQs in the absence of ICs. In Section 4, we present an $O(n^4)$ algorithm in the presence of the three kinds of ICs discussed above. In Section 5, we present an $O(n^2)$ algorithm in the presence of only required-child and required-descendant ICs (i.e., no subtypes). We

Figure 1: Examples of TPQs and Minimization
believe that our $O(n^2)$ algorithms in Sections 3 and 5 are runtime optimal. The main idea behind our improvement in runtime is the following: Amer-Yahia et al. use a function (called an endomorphism) to identify and remove one redundant TPQ node at a time, bottom up; we use a relation (called a simulation) to identify all the redundant nodes in one shot. In Section 6, we present our conclusions and some extensions of our algorithms. In Section 2, we discuss some of the limitations of TPQs and ICs; we also discuss why these limitations are somewhat necessary in order to have an efficient (polynomial time) algorithm for query minimization.

Apart from the results of Amer-Yahia et al., the results that most closely relate to ours are those of Wood [20, 21, 22, 23]. He studied the minimization of a special class of XPath queries [24] that he called simple XPath queries. Simple XPath queries are TPQs without d-edges, but with the added flexibility that the label of a node could be $\epsilon$; $\epsilon$ stands for "any" type, and in any embedding of the query in a database, the image of a node labeled $\epsilon$ could be of any type in $\Sigma$. Wood showed that, in the absence of constraints, the minimal query equivalent to a simple XPath query can be found in polynomial time. In Section 6 (Conclusions), we show how our $O(n^2)$ algorithm in Section 3 can be extended to simple XPath queries (no d-edges). Miklau and Suciu [16] show that the problem of minimizing TPQs that contain c-edges, d-edges and nodes labeled $\epsilon$ is co-NP complete. In Section 2, we discuss required-sibling constraints studied by Wood.

2. LIMITATIONS OF TPQ’S AND IC’S

Papakonstantinou and Vianu [17] introduced a very general tree pattern query language for XML called loto-ql (also see [18]). Loto-ql queries contain regular expressions over $\Sigma$ on the edges; they allow for vertical and horizontal navigation in the input database, respectively. Compared to loto-ql queries, TPQs are limited as follows:

1. The horizontal navigation allowed in TPQs (by having multiple children at a node) is very limited compared to the ones in loto-ql queries. In particular, TPQs ignore the order of children of a node in the database; in an embedding, this allows for different nodes of the query (of the same type) to be mapped to the same node of the database.

2. The vertical navigation in loto-ql queries can be specified by arbitrary regular expressions. TPQs only allow vertical navigation using c-edges and d-edges. A c-edge corresponds to the regular expression $a$ for some $a \in \Sigma$; a d-edge corresponds to the regular expression $\Sigma^* a$, for some $a \in \Sigma$; here, $a$ is the element type of the destination node of the c/d-edge.

This simplicity of TPQs is necessary in order to have efficient algorithms for query minimization. If arbitrary regular expressions are allowed for vertical navigation, then query minimization becomes PSPACE-Hard: Testing if an edge with regular expression $r_1$ is subsumed by another edge with regular expression $r_2$ is equivalent to testing if $L(r_2) \subseteq L(r_1)$; this is known to be PSPACE-Complete [11].

The ICs defined in Section 1 are also somewhat limited. Document Type Definitions (DTD) are usually used to specify a schema for a class of XML documents. A DTD specifies, for each type $a \in \Sigma$, a regular language $R(a)$ over $\Sigma$ consisting of those strings $w$ that could form the sequence of types of the children of a node of type $a$. Even ignoring the order, this specifies which element types can appear together as children of a particular node type; this can not be specified using the ICs we consider. This simplicity of the ICs also seems to be an important factor in designing efficient minimization algorithms for TPQs. In fact, in the presence of a DTD, there need not be a unique minimal TPQ equivalent to a given TPQ, as seen from the following example: Let $\Sigma = \{a, b, c, d\}$, and let the DTD D be $R(a) = bc + d$, $R(b) = R(c) = R(d) = c$. Consider the 3-node TPQ $Q = a \cdot (bc)$, where the root of type $a$ is the output node, and it has two children of types $b$ and $c$. $Q$ has two minimal equivalent TPQs (in the presence of D): $a \cdot (b)$ and $a \cdot (c)$.

The above example contradicts a claim of Wood [23]. He considered modeling the effects of a DTD using required-sibling constraints, and studied the minimization of simple XPath queries in the presence of such constraints. A required-sibling constraint of the form $r_1 : T \rightarrow r_2$, where $r_1, r_2 \in \Sigma$ and $T \subseteq \Sigma$, means that every database node of type $r_1$ that has children of type $r_2$ in each child of type $r_2$. The DTD $D$ above implies the following set of required-sibling constraints: $S = \{a : \{b\} \rightarrow c, a : \{c\} \rightarrow b\}$. Wood [23] claimed that, in the presence of a DTD or required-sibling constraints, there is a unique minimal XPath query equivalent to a given simple XPath query without $\epsilon$ label. The above example serves as a counterexample to this claim.

While the TPQs and ICs we consider seem to be somewhat limited (as explained above), TPQs do capture a significant part of current XML query languages such as XPath and XQuery. Also, we feel that our algorithms form the basis of an important first step towards efficient algorithms for minimizing more complex XML and LDAP queries, in the presence of more complex integrity constraints.

3. $O(n^2)$ MINIMIZATION ALGORITHM IN THE ABSENCE OF IC’S

Let $Q(D)$ denote the result of a query $Q$ on a database $D$. Following [7, 19], we say that $Q_1 \subseteq Q_2$ for queries $Q_1$ and $Q_2$, if $Q_1(D) \subseteq Q_2(D)$ for all databases $D$; $Q_1$ and $Q_2$ are equivalent (denoted by $Q_1 = Q_2$), if $Q_1 \subseteq Q_2$ and $Q_2 \subseteq Q_1$. A query $Q$ is said to be minimal if no query of smaller size is equivalent to $Q$. For TPQs, $s\text{ize}$ is the number of nodes. $Q_1 \subseteq Q_2$ if there exists a query homomorphism (also called a containment mapping in the literature) from $Q_2$ to $Q_1$ (see [7, 19]). When specialized to TPQs [2], a homomorphism $h : Q_2 \rightarrow Q_1$ is a mapping from $Q_2$’s nodes to $Q_1$’s nodes that satisfies the following conditions:

1. Preserve node types: For each node $u \in Q_2$, $u$ and $h(u)$ are of the same type; also, $h(op(Q_2)) = op(Q_1)$.
2. Preserve c/d-edge relationships: If $u \rightarrow v$ in $Q_2$, then $h(u) \rightarrow h(v)$ in $Q_1$; if $u \Rightarrow v$ in $Q_2$, then $h(v)$ is a descendant of $h(u)$ in $Q_1$.

A homomorphism from a TPQ $Q_2$ into itself is called an endomorphism. A node $u \in Q$ is said to be redundant if the query obtained from $Q$ by deleting $u$ and all its descendants is equivalent to $Q$. Amer-Yahia et al. [2] stated the following.
PROPOSITION 3.1. [2] A node \( u \) of a TPQ \( Q \) is redundant iff there exists an endomorphism \( h \) on \( Q \) such that \( h(u) \neq u \).

Then they proved the following.

THEOREM 3.2. [2] For a given TPQ \( Q \), there exists a unique minimal equivalent TPQ \( Q' \). \( Q' \) can be obtained from \( Q \) by repeatedly removing a redundant leaf node, until no leaf is redundant (note that a leaf node at some intermediate step could be an internal node of \( Q \)).

They presented an \( O(n^3) \) algorithm (based on Proposition 3.1) to determine if a given leaf is redundant. This leads to an \( O(n^4) \) algorithm (based on Theorem 3.2) for minimizing \( Q \).

We present an efficient \( O(n^2) \) algorithm using the concept of simulation. Simulation is a binary relation on the set of nodes, as opposed to endomorphism which is a function. Simulation provides one possible notion of dominance between the nodes of a graph, and has been in process equivalence and in graph models for data. In particular, it is used in defining schema for semistructured data [1, 4].

Consider a directed graph \( G = (V, E) \) consisting of a set \( V \) of nodes and a set \( E \) of directed edges; each node \( u \in V \) has a type \( \tau(u) \) associated with it. For \( u \in V \), let \( \text{post}(u) \) denote the set of nodes to which there is an edge from \( u \). Simulation is the largest binary relation \( \preceq \) on \( V \) such that the following holds: If \( u \preceq v \), then \( \tau(u) = \tau(v) \) and for each \( u' \in \text{post}(u) \), there exists \( v' \in \text{post}(v) \) such that \( u' \preceq v' \). If \( u \preceq v \), we say that \( u \) is simulated by \( v \), \( v \) simulates \( u \), or \( v \) is a simulator of \( u \); let \( \text{sim}(u) \) denote the set of all simulators of \( u \). It is well-known that the simulation relation is reflexive and transitive, but it may not be symmetric. Vertices \( u \) and \( v \) are said to be similar, denoted by \( u \sim v \), if \( u \preceq v \) and \( v \preceq u \); clearly, similarity is an equivalence relation.

Consider the example tree shown in Figure 2a. We have \( \text{sim}(v_4) = \{v_4, v_5\} \), \( \text{sim}(v_5) = \{v_2, v_3\} \), and \( \text{sim}(v_i) = \{v_i\} \) for all other nodes \( v_i \). So, the only nontrivial relational pairs are \( v_4 \preceq v_5 \) and \( v_2 \preceq v_3 \).

Let \( |V| = n \) and \( |E| = m \). Bloom and Paige [3] and Henzinger et al. [12] presented \( O(mn) \) algorithms for computing the simulation relation of arbitrary graphs. If the graph is a tree, then \( m = n - 1 \); in this case, their algorithms run in \( O(n^2) \) time. Unfortunately, these algorithms will not work when there are \( d \)-edges, as in a TPQ. First, we need to define the concept of simulation to account for the presence of \( d \)-edges. We define the simulation relation for a TPQ \( Q \) as follows. It is the largest binary relation \( \preceq \) on the nodes of \( Q \) such that, whenever \( u \preceq v \), the following conditions hold:

1. Preserve node types: \( \tau(u) = \tau(v) \); also, if \( u = op(Q) \) then \( v = op(Q) \).
2. Preserve \( c \)-edge relationships: If \( u \to u' \), then \( v \) has a \( c \)-child \( v' \) such that \( u' \preceq v' \).
3. Preserve \( d \)-edge relationships: If \( u \Rightarrow u'' \), then \( v \) has a descendant \( v'' \) such that \( u'' \preceq v'' \).

Consider the example TPQ shown in Figure 2b. We have \( \text{sim}(v_4) = \{v_4, v_5\} \), \( \text{sim}(v_5) = \{v_2, v_3\} \), \( \text{sim}(v_2) = \{v_2, v_3\} \), \( \text{sim}(v_1) = \{v_1\} \) for all other nodes \( v_i \).

For the sake of completeness, let us consider the connection between simulation and endomorphism. For any node \( u \) in a TPQ \( Q \), let

\[
\text{endo}(u) = \{v \mid \text{there exists an endomorphism } f \text{ on } Q \text{ such that } f(u) = v\}.
\]

If \( v \in \text{endo}(u) \), then \( v \in \text{sim}(u) \); so \( \text{sim}(u) \supseteq \text{endo}(u) \). The containment could be strict because whether \( v \in \text{sim}(u) \) depends only on the descendants of \( u \) and \( v \), whereas whether \( v \in \text{endo}(u) \) depends also on the ancestors of \( u \) and \( v \), and their descendants. Also, by Proposition 3.1, \( u \) is redundant iff there exists \( v \in \text{endo}(u) \), \( v \neq u \). In contrast, the condition that there exist \( v \in \text{sim}(u) \), \( v \neq u \), is necessary, but not sufficient, for \( u \) to be redundant. For example, if \( u \) is a leaf and \( v \) is any other node with \( \tau(v) = \tau(u) \), then \( v \in \text{sim}(u) \); this certainly does not imply that \( u \) is redundant.

The simulation relation on TPQs is reflexive and transitive, but it may not be symmetric. Also note that, by condition 1 above, \( \text{sim}(op(Q)) = \{op(Q)\} \). The algorithms of Bloom and Paige [3] and Henzinger et al. [12] referred to above for computing the simulation relation on graphs will not work for TPQs because of condition 3) above. In fact, it is unlikely that there exists an \( O(mn) \) algorithm for computing the simulation relation of arbitrary graphs that contain \( d \)-edges. We will present an \( O(n^2) \) algorithm for TPQs; it can be easily extended to an \( O(mn) \) algorithm for acyclic graphs. Before we present our algorithm, let us consider the connection between simulation and minimization of TPQs.

We have the following result.
Algorithm TPQS\textit{Simulation}\n
$V \leftarrow \text{set of nodes of } Q \text{ in some bottom-up order}$

for each $u \in V$ in order do

\hspace{1em} if $u = \text{op}(Q)$ then $\text{sim}(u) = \{u\}$
\hspace{1em} else if $u$ is a leaf then $\text{sim}(u) = \{v \in V \mid \tau(v) = \tau(u)\}$
\hspace{1em} else $\text{sim}(u) = \{v \in V \mid \tau(v) = \tau(u), v \in \text{cpar}(\text{sim}(u')) \text{ for each c-child } u' \text{ of } u,$
\hspace{1em} \hspace{1em} and $v \in \text{anc}(\text{sim}(u'')) \text{ for each d-child } u'' \text{ of } u\}$

\hspace{1em} compute $\text{cpar}(\text{sim}(u))$ and $\text{anc}(\text{sim}(u))$

\vspace{1em}

\textbf{Figure 3: The Simulation Algorithm}

Algorithm TPQ\textit{Minimization}(u) \hspace{1em} /* $u$ is a nonredundant node of a TPQ\n
for each child $v$ of $u$ do

\hspace{1em} if $v$ is a c-child then
\hspace{1em} \hspace{1em} if $u$ has another c-child $w \in \text{sim}(v)$ that has not been deleted
\hspace{1em} \hspace{1em} then delete $v$ /* the entire subtree rooted at $v$ is deleted
\hspace{1em} \hspace{1em} else $\text{TPQMinimization}(v)$ /* node $v$ is nonredundant
\hspace{1em} if $v$ is a d-child then
\hspace{1em} \hspace{1em} if $u$ has another child $w \in \text{sim}(v) \cup \text{anc}(\text{sim}(v))$ that has not been deleted
\hspace{1em} \hspace{1em} then delete $v$ /* the entire subtree rooted at $v$ is deleted
\hspace{1em} \hspace{1em} else $\text{TPQMinimization}(v)$ /* node $v$ is nonredundant

\vspace{1em}

\textbf{Figure 4: The Minimization Algorithm}

\vspace{1em}

\textbf{LEMMA 3.3.} Let $u$ be a nonredundant node of a TPQ $Q$.

1. A c-child $v$ of $u$ is redundant in $Q$ iff $u$ has another c-child $w \in \text{sim}(v)$.

2. A d-child $v$ of $u$ is redundant in $Q$ iff $u$ has another descendant $w \in \text{sim}(v)$.

\textbf{Proof.} Let $u$ be a nonredundant node of a TPQ $Q$, and let $v$ be a child of $u$. Consider any embedding $\beta$ of $Q$ into a tree database $db$. The presence of $v$ in $Q$ imposes the following restriction on $\beta$. If $v$ is a c-child (resp. d-child) of $u$, then $\beta(v)$ should be a child (resp. descendant) of $\beta(u)$ in $db$. The descendants of $v$ in $Q$ translate to corresponding restrictions on $\beta(v)$ in $db$.

Now consider the "if" parts of conditions 1 and 2 in the lemma. If there exists a $w$ as specified, then the conditions imposed on $\beta$ (specifically, on $\beta(u)$ and its descendants) by $w$ and its descendants subsume those imposed by $v$ and its descendants. Hence $v$ is redundant.

Next, consider the "only if" part. Since $u$ is nonredundant, all its ancestors are nonredundant. A nonredundant node is unique in the sense that no other node can play its role. Since $u$ is unique, the restrictions imposed on $\beta$ (specifically, on $\beta(u)$ and its descendants) by a c-child (resp. d-child) $v$ of $u$ can be subsumed by the restrictions imposed by another c-child (resp. descendant) $w$ of $u$; also, the restrictions imposed on $\beta$ by $w$ and its descendants must subsume those imposed by $v$ and its descendants, i.e., $w \in \text{sim}(v)$.

Our algorithm for minimizing $Q$ consists of two parts. First, algorithm TPQS\textit{Simulation} (Figure 3) computes the simulation relation on $Q$, in bottom-up order. Then, algorithm TPQ\textit{Minimization} (Figure 4) finds the minimal TPQ equivalent to $Q$: it deletes maximal redundant subtrees, in top-down order, using Lemma 3.3 (In contrast, the algorithm in [2] eliminates redundant nodes one at a time, bottom-up, as specified in Theorem 3.2). The phrase "in $Q$" in items 1 and 2) of Lemma 3.3 requires us to check for the redundancy of the children of $u$, one child at a time. If a child $v$ is found to be redundant and deleted along with all its descendants (resulting in a new TPQ $Q'$), then checking for the redundancy of the next child $v'$ should be done in $Q'$.

For a set $S$ of some nodes of $Q$, let $\text{parents}(S)$ (denoted by $\text{cpar}(S)$) be the set of nodes of $Q$ that have a c-child in $S$; let $\text{ancestors}(S)$ (denoted by $\text{anc}(S)$) be the set of nodes that have a proper descendant in $S$. Clearly, $\text{cpar}(S)$ can be computed in $O(n)$ time; $\text{anc}(S)$ can also be computed in $O(n)$ time bottom-up, in the order of decreasing depth (distance from the root). Note that conditions 2) and 3) in the above definition of simulation can be restated as follows:

2. Preserve c-edge relationships: If $u \rightarrow u'$, then $v \in \text{cpar}(\text{sim}(u'))$.

3. Preserve d-edge relationships: If $u \Rightarrow u''$, then $v \in \text{anc}(\text{sim}(u''))$.

Using this restatement, algorithm TPQS\textit{Simulation} computes the simulation relation on $Q$. First, it orders the nodes of $Q$ bottom-up: all the children of a node $u$ must appear before node $u$. This can be done, for example, according to the post order traversal of $Q$, in linear time. For each node $u$, in order, the algorithm computes $\text{sim}(u)$, $\text{cpar}(\text{sim}(u))$ and $\text{anc}(\text{sim}(u))$; each of them will be represented as a boolean array of $n$ elements, indexed by the nodes $v \in V$. For each node $u$, the algorithm first computes $\text{sim}(u)$. For a leaf node $u$, computing $\text{sim}(u)$ takes $O(n)$ time. Now consider an internal node $u$. For each $v \in V$, determining if $v \in \text{sim}(u)$
Algorithm MinimizeChase

compute \(\text{chase}(Q)\)

compute \(\text{sim}(u)\) for the original nodes \(u\) of \(\text{chase}(Q)\) using \(\text{ChaseSimulation}\)

\(\text{ChaseMinimization}(\text{root}(\text{chase}(Q)))\)

drop the remaining chase nodes

Figure 5: The Overall Algorithm with Constraints

takes time proportional to the number of children of \(u\). Hence, the total time to find \(\text{sim}(u)\) is \(O(n \times \text{children}(u))\). As pointed out above, \(\text{par}(\text{sim}(u))\) and \(\text{anc}(\text{sim}(u))\) can be computed in \(O(n)\) time. So, a single pass thru the for loop takes time \(O(n \times (\text{children}(u) + 1))\); hence, the entire for loop takes \(O(n \times \sum_{u \in V} (\text{children}(u) + 1)) = O(n^2)\) time. Since the simulation relation could be of size \(\Theta(n^2)\), this algorithm has optimal runtime.

Now consider the recursive algorithm \(\text{TPQMinimization}\).

The input node \(u\) is a nonredundant node of a TPQ; the algorithm minimizes the subtree rooted at \(u\), using the simulation relation computed by \(\text{TPQSimulation}\). Recall that the simulation relation is transitive. So, the order in which the children \(v\) of node \(u\) are considered is irrelevant. Let \(v_1, v_2, v_3\) be three children of \(u\) such that \(v_1 \leq v_2\) and \(v_2 \leq v_3\); then \(v_1\) and \(v_3\) should be deleted in favor of \(v_2\). It doesn’t matter whether we first delete \(v_1\) and then delete \(v_2\), or we first delete \(v_2\) and then delete \(v_1\) (because, by transitivity, \(v_1 \leq v_3\)). The correctness of the algorithm follows from Lemma 3.3. For each child \(v\) of \(u\), it takes \(O(\text{children}(u)^2)\) time to check if \(v\) is redundant; so, the call \(\text{TPQMinimization}(\text{root}(Q))\) runs in \(O(\sum_{u \in V} \text{children}(u)^2)\) time.

For an example, let \(Q\) be the TPQ shown in Figure 2b.

For the leaves, we have \(\text{sim}(v_4) = \{v_4, v_6\}\), \(\text{sim}(v_5) = \text{sim}(v_8) = \{v_5, v_8\}\), \(\text{par}(\text{sim}(v_4)) = \text{par}(\{v_4, v_6\}) = \{v_1, v_2, v_3, v_7\}\), and \(\text{anc}(\text{sim}(v_5)) = \text{anc}(\{v_5, v_8\}) = \{v_1, v_2, v_3, v_7\}\). Then, since \(v_3 \in \text{par}(\text{sim}(v_4))\) and \(v_3 \in \text{anc}(\text{sim}(v_5))\), \(\text{TPQSimulation}\) concludes that \(v_2 \in \text{sim}(v_2)\). Finally, since \(v_3 \in \text{sim}(v_2)\), \(\text{TPQMinimization}\) deletes the subtree rooted at \(v_2\); the resulting TPQ is minimal.

In summary, we have the following.

THEOREM 3.4. Algorithms \(\text{TPQSimulation}\) and \(\text{TPQMinimization}\) together correctly compute the minimal TPQ equivalent to a given TPQ in \(O(n^2)\) time.

We believe that our algorithm has optimal runtime.

4. \(O(n^4)\) MINIMIZATION ALGORITHM IN THE PRESENCE OF IC’S

Let \(C\) be a set of ICs of the three kinds described in Section 1. Amer-Yahia et al. [2] presented an \(O(n^5)\) algorithm for minimizing TPQs, in the presence of \(C\). In Section 5, we will show that this algorithm is incorrect; it might not produce a minimal TPQ when subtype ICs are present. In this section, we present an \(O(n^4)\) algorithm.

Following [7, 19, 2], we say that \(Q_1 \subseteq C Q_2\) for queries \(Q_1\) and \(Q_2\), if \(Q_1(D) \subseteq Q_2(D)\) for all databases \(D\) that satisfy the constraints in \(C\); \(Q_1\) and \(Q_2\) are equivalent in the presence of \(C\) (denoted by \(\equiv_C Q_2\)), if \(Q_1 \subseteq C Q_2\) and \(Q_2 \subseteq C Q_1\). A query \(Q\) is said to be minimal in the presence of \(C\), if no query of smaller size is equivalent to \(Q\) in the presence of \(C\). From now on, we will usually drop the qualifier “in the presence of \(C\)”.

For relational database queries, the classical \(\text{chase}\) technique [15, 19] is used to rewrite a query to incorporate the effects of given integrity constraints. Then, a minimal query equivalent to the original query, in the presence of the integrity constraints, can be obtained by minimizing the rewritten query (without further regard to the integrity constraints). In this section, we use the same approach to minimize a TPQ \(Q\) in the presence of \(C\).

Amer-Yahia et al. [2] specified a procedure to add nodes to \(Q\), one at a time, to incorporate the effects of \(C\), but cautioned:

“a blind application of \(\text{chase}\) [their procedure to add nodes] can make the result of the chase arbitrarily bigger than the original query; in particular, its depth can increase arbitrarily under \(\text{chase}\).”

They did not show how to add nodes to \(Q\), in a systematic manner, to get a finite query that incorporates the effects of \(C\) (as we will show in Section 5, their procedure, called “augmentation”, might not work when subtype ICs are present). We show how to construct a finite query \(\text{chase}(Q)\) from \(Q\), that incorporates the effects of \(C\), while adding only the required nodes. In general, our \(\text{chase}(Q)\) is not a tree, but a directed acyclic graph (dag).

The closure of \(C\), denoted by \(\text{closure}(C)\), can be obtained by first initializing it to \(C\), and then repeatedly doing the following until no more changes occur:

1. If \(r_1 \to r_2\), then add \(r_1 \Rightarrow r_2\)
2. If \(r_1 \Rightarrow r_2\) and \(r_2 \Rightarrow r_3\), then add \(r_1 \Rightarrow r_3\)
3. If \(r_1 \leq r_2\) and \(r_2 \leq r_3\), then add \(r_1 \leq r_3\)
4. If \(r_1 \leq r_2\) and \(r_2 \to r_3\), then add \(r_1 \to r_3\)
5. If \(r_1 \leq r_2\) and \(r_2 \Rightarrow r_3\), then add \(r_1 \Rightarrow r_3\)
6. If \(r_1 \to r_2\) and \(r_2 \leq r_3\), then add \(r_1 \to r_3\)
7. If \(r_1 \Rightarrow r_2\) and \(r_2 \leq r_3\), then add \(r_1 \Rightarrow r_3\)

Note that \(|\text{closure}(C)| = O(|C|^2)\). Also, \(\text{closure}(C)\) (as well as \(C\)) must be acyclic: If \(r_1 \Rightarrow r_2\) and \(r_2 \Rightarrow r_1\), then there can be no finite database containing an element of type \(r_1\) or \(r_2\). If \(C\) does not contain any subtype constraints (as in Section 5), then steps 3 thru 7 above can be dropped.

The constraint graph \(G_C = (V_C, E_C)\) is the directed graph defined as follows. The set \(V_C\) of nodes is \(\Sigma\) (recall that \(\Sigma\) is the set of all element types under consideration). The set \(E_C\) consists of e-edges and d-edges:

1. If \(r_1 \to r_2\) is in \(\text{closure}(C)\), then there is an e-edge from \(r_1\) to \(r_2\).
Algorithm ChaseSimulation

\[ V \leftarrow \text{set of nodes of } \text{chase}(Q) \]
\[ V_0 \leftarrow \text{set of original nodes of } \text{chase}(Q) \text{ in some bottom-up order} \]
for each \( u \in V_0 \) in order do
  if \( u = \text{op}(Q) \) then \( \text{sim}(u) = \{u\} \)
  else if \( u \) is originally a leaf then \( \text{sim}(u) = \{v \in V \mid \tau(v) \leq \tau(u)\} \)
  else \( \text{sim}(u) = \{v \in V \mid \tau(v) \leq \tau(u), v \in \text{cpar}(\text{sim}(u')) \text{ for each original } \text{c-child } u' \text{ of } u, \)
  and \( v \in \text{anc}(\text{sim}(u')) \text{ for each original } \text{d-child } u' \text{ of } u \} \)
  compute \( \text{cpar}(\text{sim}(u)) \) and \( \text{anc}(\text{sim}(u)) \)

Figure 6: The Simulation Algorithm with Constraints

Algorithm ChaseMinimization(u) /* u is a non-redundant original node of \text{chase}(Q) */

for each original child \( v \) of \( u \) do
  if \( v \) is a c-child then
    if \( v \) has another c-child \( w \in \text{sim}(v) \) that has not been deleted then delete \( v \) /* the entire subgraph rooted at \( v \) is deleted */
    else \( \text{ChaseMinimization}(v) */ node \( v \) is nonredundant */
  if \( v \) is a d-child then
    if \( v \) has another child \( w \in \text{sim}(v) \cup \text{anc}(\text{sim}(v)) \) that has not been deleted then delete \( v \) /* the entire subgraph rooted at \( v \) is deleted */
    else \( \text{ChaseMinimization}(v) */ node \( v \) is nonredundant */

Figure 7: The Minimization Algorithm with Constraints

2. If \( r_1 \Rightarrow r_2 \) is in \text{closure}(C), but \( r_1 \rightarrow r_2 \) is not in \text{closure}(C), and there is no type \( r_3 \) such that both \( r_1 \Rightarrow r_3 \) and \( r_3 \Rightarrow r_2 \) are in \text{closure}(C), then there is a \text{d-edge} from \( r_1 \) to \( r_2 \).

\( GC \) represents all the required-child and required-descendant constraints in \text{closure}(C), in a concise manner; the effects of the subtype constraints on the required-child and required-descendant constraints are already reflected in \text{closure}(C) (and hence in \( GC \), due to steps 4) thru 7) above in the computation of \text{closure}(C). Since \text{closure}(C) is acyclic, \( GC \) is a dag. \( GC \) contains \( |\Sigma| \) nodes and \( \mathcal{O}(|\Sigma|^2) \) edges; we let the size \( |GC| \) denote \( |V_G| + |E_G| \). For a given \( \Sigma' \subseteq \Sigma \), let \( G'_C = (V'_C, E'_C) \) be the constraint graph pertaining to the element types in \( \Sigma' \). \( G'_C \) can be obtained from \( G_C \) by \( \mathcal{O}(|G_C| + |G'_C|) \) time. For any type \( \tau \in \Sigma' \), let \( G_C(\tau) \) denote the subgraph of \( G_C \) that is rooted at \( \tau \). \( G_C(\tau) \) represents all constraints in \text{closure}(C) of the form \( \tau \rightarrow \tau' \) and \( \tau' \Rightarrow \tau' \), for \( \tau' \in \Sigma' \).

We define \text{chase}(Q) to be the dag obtained from \( Q \) as follows: Let \( \Sigma' \) consist of the element types common to \( C \) and \( Q \) (so \( |\Sigma'| \leq n \)); from \( G_C \), construct the constraint graph \( G_C' = (V'_C, E'_C) \) for \( \Sigma' \) (so \( |V'_C| = |\Sigma'| \) and \( |E'_C| = \mathcal{O}(|\Sigma'|^2) \)); at each node of \( Q \) of type \( \tau \), attach a copy of \( G_C(\tau) \) rooted at that node. The new nodes added in this process will be called \text{chase nodes}; other nodes (i.e., those from \( Q \) will be called \text{original nodes}. \text{Chase}(Q) is a dag with \( \mathcal{O}(n|\Sigma'|) \) nodes and \( \mathcal{O}(n|E_C|) \) edges; its height is at most \( \text{height}(Q) + |\Sigma'| \).

\text{Chase}(Q) incorporates the effects of all the required-child and required-descendant constraints in \text{closure}(C). So, as in the case of relational database queries, the minimal TPQ equivalent to \( Q \) can be obtained from \text{chase}(Q), using algorithm \text{MinimizeChase} (Figure 5). \text{MinimizeChase} calls \text{ChaseSimulation} to compute \( \text{sim}(u) \) for all the original nodes \( u \) in \text{chase}(Q). It then calls \text{ChaseMinimization} to delete the redundant original nodes (and their descendants) in \text{chase}(Q). Finally, it deletes all the remaining chase nodes, which are clearly redundant.

The working of algorithms \text{ChaseSimulation} and \text{ChaseMinimization} (Figures 6 and 7) parallel those of algorithms \text{TPQSimulation} and \text{TPQMinimization} (Figures 3 and 4, Section 3), respectively. \text{ChaseSimulation} is identical to \text{TPQSimulation}, except that it computes \( \text{sim}(u) \) only for the original nodes \( u \) in \text{chase}(Q), and also incorporates the effects of the subtype constraints in \text{closure}(C). \text{ChaseMinimization} is identical to \text{TPQMinimization}, except that it only deletes redundant original nodes in \text{chase}(Q).

The runtime analyses are also partly similar. First consider \text{ChaseSimulation}. Let \( V_0 \) be the set of nodes and the set of original nodes of \text{chase}(Q), respectively; \( |V_0| = |Q| = n \), and \( |V| = \mathcal{O}(n \cdot |V_0|) \). In \( V_0 \), the nodes are in bottom-up order. For each node \( u \in V_0 \), the algorithm computes \( \text{sim}(u) \), \( \text{cpar}(\text{sim}(u)) \) and \( \text{anc}(\text{sim}(u)) \); each of them will be represented as a boolean array of \( |V| \) elements, indexed by the nodes \( v \in V \). For each \( u \), the algorithm first computes \( \text{sim}(u) \). For an original leaf node \( u \), computing \( \text{sim}(u) \) takes \( \mathcal{O}(|V|) \) time. Now consider an original internal node \( u \). For each \( v \in V \), determining if \( v \in \text{sim}(u) \) takes time proportional to the number of children of \( u \) in \( Q \). Hence, the total time to find \( \text{sim}(u) \) is \( \mathcal{O}(|V| \cdot |\text{children}_Q(u)|) \). So, the total time to compute \( \text{sim}(u) \) for all the original nodes is \( \mathcal{O}(|V| \cdot \sum_{u \in Q} |\text{children}_Q(u)|) = \mathcal{O}(n \cdot |V|) \). Now consider the computation of \( \text{cpar}(\text{sim}(u)) \) and \( \text{anc}(\text{sim}(u)) \). Note that each chase node in \( \text{sim}(u) \) could have \( \mathcal{O}(|V_0|) \) c-parents and/or d-parents, since \( G_C \) is a dag (not necessarily a tree). In any case, \( \text{cpar}(\text{sim}(u)) \) can be
computed in $|\text{chase}(Q)| = O(n \cdot |G_C'|)$ time. Anc(sim(u)) can also be computed in $O(|\text{chase}(Q)|)$ time bottom-up, in the order of decreasing depth (distance from the root). So, $\text{cpp}(\text{sim}(u))$ and $\text{anc(\text{sim}(u))}$, for all the original nodes $u$, can be computed in $O(n \cdot |\text{chase}(Q)|)$ time. So, the overall runtime for $\text{ChaseSimulation}$ is $O(n \cdot |\text{chase}(Q)|) = O(n^2 \cdot |G_C'|) = O(n^3)$.

Now consider the recursive algorithm $\text{ChaseMinimization}$. The input node $u$ is a nonredundant original node of $\text{chase}(Q)$. For each original child $v$ of $u$, it takes $O(\text{children}_{\text{chase}(Q)}(u)) = O(n)$ time to check if $v$ is redundant; so, the call $\text{ChaseMinimization}(\text{root}(\text{chase}(Q)))$ runs in $O\left(\sum_{v \in V} |\text{children}(u)| \cdot |\text{children}_{\text{chase}(Q)}(u)|\right) = O(n \cdot \sum_{u \in V} |\text{children}(u)|) = O(n^2)$ time. In summary, we have the following.

**THEOREM 4.1.** Algorithm $\text{MinimizeChase}$ correctly computes the minimal TPQ equivalent to a given TPQ, in the presence of ICs, in $O(n^3)$ time.

**PROOF.** First, let us consider the correctness. $\text{chase}(Q)$ correctly incorporates the effects of all the required-child and required-descendant constraints in $\text{closure}(C)$. Then, $\text{ChaseSimulation}$ and $\text{ChaseMinimization}$ correctly identify and remove all the redundant original nodes in $\text{chase}(Q)$, while taking into account all the subtype constraints in $\text{closure}(C)$. The remaining chase nodes are clearly redundant, and can be dropped. So, $\text{MinimizeChase}$ correctly computes the minimal TPQ.

Now, let us consider the runtime of $\text{MinimizeChase}$. $\text{Chase}(Q)$ has $O(n^2)$ nodes and $O(n^3)$ edges; it can be constructed in $O(n^3)$ time. As discussed above, $\text{ChaseSimulation}$ and $\text{ChaseMinimization}$ take $O(n^3)$ and $O(n^2)$ time, respectively. Dropping the remaining chase nodes takes $O(n^3)$ time. So, $\text{MinimizeChase}$ runs in $O(n^3)$ time. \qed

5. $O(n^2)$ MINIMIZATION ALGORITHM IN THE ABSENCE OF SUBTYPES

In this section, let $C$ be a closed set of ICs; i.e., $\text{closure}(C) = C$. Initially, let $C$ consist of ICs of the three kinds described in Section 1. Amer-Yahia et al. [2] gave the following procedure to obtain an augmented TPQ $A(Q)$ from a given TPQ $Q$. For each node $u \in Q$ of type $\tau_1$:

1. If $\tau_1 \rightarrow \tau_2$ is in $C$, add a new leaf node $u'$ of type $\tau_2$, and a $c$-edge $u \rightarrow u'$.
2. If $\tau_1 \Rightarrow \tau_2$ is in $C$, but $\tau_1 \rightarrow \tau_2$ is not in $C$, add a new leaf node $u''$ of type $\tau_2$, and a $d$-edge $u \Rightarrow u''$.
3. If $\tau_1 \leq \tau_2$ is in $C$, also associate type $\tau_2$ with node $u$.

The leaf nodes added during augmentation are called augmentation leaves; there could be $\Theta(n)$ such leaves for each node in $Q$, for a total of $\Theta(n^2)$ leaves. So, $A(Q)$ could have $\Theta(n^2)$ nodes, and can be constructed in $O(n^2)$ time.

Then, Amer-Yahia et al. [2, Lemma 5.4] gave the following procedure to find the minimal equivalent TPQ $Q'$ in the presence of $C$.

First construct the augmented TPQ $A(Q)$. Then minimize $A(Q)$ (without further regard to $C$) with the exception that the augmentation leaves should not be tested for redundancy. Then finally, remove all the remaining augmentation leaves.

If subtypes are present, this procedure might not produce the minimal TPQ. This is seen from the following example. Let

$Q = (\text{univ(college(dept(lab)), scicollege*)})$ (see Figure 8a) and let $C$ be the closure of

$\{\text{scicollege} \leq \text{college}, \text{scidept} \leq \text{dept}, \text{scicollege} \rightarrow \text{scidept}, \text{scidept} \rightarrow \text{lab}\}$. $A(Q)$ is shown in Figure 8b; one $c$-edge and one $d$-edge added from the $\text{scicollege}$ node correspond to the ICs $\text{scicollege} \rightarrow \text{scidept}$ and $\text{scicollege} \Rightarrow \text{lab}$ in $C$, respectively; these edges result from steps 1) and 2) above, respectively. The above procedure of Amer-Yahia et al. would output $Q$ itself, whereas the minimal equivalent query is $(\text{univ(scicollege)})$ (Figure 8c). The only way to obtain the minimal query requires first appending the chain $(\text{scidept(lab)})$ under the $\text{scicollege}$ node in $Q$, as done in $\text{chase}(Q)$ described in the previous section. When subtypes are present, the only algorithm known is the one given in the previous section.
Algorithm MinimizeCTPQ
compute R(Q)
compute the simulation relation on R(Q) using CTPQSimulation
CTPQMinimization(root(R(Q)))

Figure 9: The Overall Algorithm in the Absence of Subtypes

Algorithm CTPQSimulation
V ← set of nodes of R(Q) in some bottom-up order
for each u ∈ V in order do
  if u = op(Q) then
    sim(u) = \{ u \}
    compute cpar(sim(u))
    auganc(sim(u)) ← anc(sim(u))
  continue
  /* Go to the next pass of the for loop */
  if u is a leaf then
    sim(u) = \{ v ∈ V | \tau(v) = \tau(u) \}
    compute cpar(sim(u))
    auganc(sim(u)) = anc(sim(u)) ∪ \{ v ∈ V | \tau(v) ⇒ \tau(u) \text{ is in } C \} ∪ anc(\{ v ∈ V | \tau(v) ⇒ \tau(u) \text{ is in } C \})
  else
    sim(u) = \{ v ∈ V | \tau(v) = \tau(u), v ∈ cpar(sim(u')) for each c-child u' of u, v ∈ auganc(sim(u'')) for each d-child u'' of u \}
    compute cpar(sim(u))
    auganc(sim(u)) ← anc(sim(u))

Figure 10: The Simulation Algorithm in the Absence of Subtypes

From now onwards, we assume that C consists only of required-child and required-descendant ICs. Then, Amer-Yahia et al.’s algorithm outlined above correctly finds the minimal TPQ equivalent to Q, in the presence of C. Applying their original O(n^2) minimization algorithm to A(Q) takes O(n^2) time so, their overall minimization algorithm runs in O(n^2) time. We present an efficient O(n^2) algorithm.

Let v be a leaf node of a TPQ, and let u be its parent. We say that v is redundant due to C if either of the following holds:

1. u → v and \tau(u) → \tau(v) is in C, or
2. u ⇒ v and \tau(u) ⇒ \tau(v) is in C.

Let R(Q) be the reduced query obtained from Q as follows: Repeatedly remove a leaf node that is redundant due to C, until no leaf is redundant due to C (note that a leaf node at some intermediate step could be an internal node of Q). R(Q) can be computed in O(n^2) time and has at most n nodes.

The minimal TPQ equivalent to Q, in the presence of C, can be obtained using algorithm MinimizeCTPQ (Figure 9). It computes the simulation relation on R(Q) using CTPQSimulation, and then calls CTPQMinimization to remove the redundant nodes. The working of algorithms CTPQSimulation and CTPQMinimization (Figures 10 and 11) parallel those of algorithms TPQSimulation and TPQMinimization (Figures 3 and 4, Section 3), respectively. The runtime analyses are also similar: CTPQSimulation and CTPQMinimization run in O(n^2) time. So, MinimizeCTPQ runs in O(n^2) time.

For an example, let Q be the TPQ shown in Figure 2b, and let C = \{ e ⇒ d \}. v_8 is the only node that is redundant due to C; the reduced query R(Q) is obtained by dropping this node from Q. In R(Q), we have sim(v_4) = sim(v_6) = \{ v_4, v_6 \}, sim(v_5) = \{ v_5 \}, cpar(sim(v_4)) = cpar(\{ v_4, v_6 \}) = \{ v_2, v_3 \}, and anc(sim(v_5)) = anc(\{ v_5 \}) = \{ v_1, v_2 \}. Since e ⇒ d is in C, auganc(sim(v_5)) contains, in addition to anc(sim(v_5)), v_7 and its ancestors; so auganc(sim(v_5)) = \{ v_1, v_2, v_3, v_7 \}. Then, since v_3 ∈ cpar(sim(v_4)) and v_3 ∈ auganc(sim(v_5)), CTPQSimulation concludes that v_3 ∈ sim(v_2). Finally, since v_3 ∈ sim(v_2), CTPQMinimization deletes the subtree rooted at v_2; the resulting TPQ is minimal.

Now consider the correctness of MinimizeCTPQ. The first step, namely the reduction of Q, is certainly harmless; as we will show below, it is absolutely essential for the correctness of the whole algorithm. Algorithm CTPQSimulation computes the simulation relation on R(Q) in the presence of C. It differs from TPQSimulation only due to auganc; the prefix aug in auganc stands for augmented. For an internal node u in R(Q), auganc(sim(u)) is same as anc(sim(u)). For a leaf node v, auganc(sim(u)) contains, in addition to anc(sim(u)), the following: Those nodes v in R(Q) such that \tau(v) ⇒ \tau(u) is in C, and their ancestors. On input R(Q), the algorithm essentially mimics the computation of algorithm TPQSimulation on the augmented input A(R(Q)). We will show that CTPQSimulation and ChaseSimulation (Figure 6, Section 4) compute essentially the same simulation relation, on inputs R(Q) and chase(R(Q)), respectively. In what follows, let sim(u), cpar(sim(u)), anc(sim(u)) and auganc(sim(u)) be the sets computed by CTPQSimulation
for nodes $u \in R(Q)$; let $sim_c(u)$, $cpar_c(sim_c(u))$ and $anc_c(sim_c(u))$ be the sets computed by $ChaseSimulation$ for original nodes $u \in chase(R(Q))$. We have the following results.

LEMMA 5.1. Let $u$ be a leaf node of $R(Q)$.

1. $sim(u) = sim_c(u) \cap R(Q)$
   (i.e., the original nodes in $sim_c(u)$)
2. $cpar(sim(u)) \subseteq cpar_c(sim_c(u)) \cap R(Q)$
3. $anc(sim(u)) \subseteq aux\_anc(sim(u))$

   PROOF. Clearly, $sim(u)$ consists of the original nodes in $sim_c(u)$; so 1) holds. Also, $cpar(sim(u))$ consists of those (original) nodes in $R(Q)$ that have a c-child in $sim_c(u)$; so 2) holds. For 3), the first part $anc(sim(u)) \subseteq aux\_anc(sim(u))$ is obvious; the second part $aux\_anc(sim(u)) = anc_c(sim_c(u)) \cap R(Q)$ follows from the way $aux\_anc$ is computed.

LEMMA 5.2. Let $u$ be an internal node of $R(Q)$.

1. $sim(u) = sim_c(u)$
2. $cpar(sim(u)) = cpar_c(sim_c(u))$
3. $aux\_anc(sim(u)) = aux\_anc(sim(u)) = anc_c(sim_c(u))$

   PROOF. The proof is by induction on the height of node $u \in R(Q)$. First note that, by 1), $sim(u)$ consists only of original nodes; also, in $chase(R(Q))$, the parent or an ancestor of an original node must be an original node. So, 2) and 3) follow from 1). The proof of 1) depends crucially on the fact that we are dealing with the reduced query $R(Q)$, and that subtrees are not allowed.

   Clearly, $sim(u) \subseteq sim_c(u)$; we show that $sim(u) \supseteq sim_c(u)$.

   Let $v \in sim_c(u)$; since subtrees are not allowed, $\tau(v) = \tau(u)$. We show that $v \in sim(u)$ by proving that $v \in cpar(sim(u))$ for each c-child $u'$ of $u$ in $R(Q)$, and $v \in aux\_anc(sim(u'))$ for each d-child $u''$ of $u$ in $R(Q)$ (refer to algorithm $CTPQSimulation$ in Figure 10).

   Let $u'$ be a c-child of $u$ in $R(Q)$; since $R(Q)$ is reduced, in $chase(R(Q))$, $u'$ must have an original c-child $v' \in sim_c(u')$.

   Now, let $u''$ be a d-child of $u$ in $R(Q)$; since $R(Q)$ is reduced, in $chase(R(Q))$, $u''$ must have an original c-child $v'' \in sim_c(u'') \cup aux\_anc(sim_c(u''))$. By Lemma 5.1 and the inductive hypothesis, $v'' \in sim(u'') \cap aux\_anc(sim(u''))$; so, $v'' \in aux\_anc(sim(u''))$.

Algorithm $CTPQMinimization$ is almost identical to $TPQMinimization$; the only difference is that $aux\_anc(sim(u))$ has been replaced by $aux\_anc(sim_c(u))$. Lemmas 5.1 and 5.2 lead to the following result.

THEOREM 5.3. Algorithm $Minimize\_CTPQ$ correctly computes the minimal $TPQ$ equivalent to a given $TPQ$, in the presence of required-child and required-descendant ICs, in $O(n^3)$ time.

6. CONCLUSIONS

In this paper, we considered the problem of minimizing tree pattern queries (TPQs), in the absence and presence of three kinds of integrity constraints (ICs): required-child, required-descendant and subtype). In Section 3, we presented an efficient $O(n^3)$ algorithm for minimizing TPQs in the absence of ICs. In Section 4, we presented an $O(n^5)$ algorithm in the presence of the three kinds of ICs. In Section 5, we presented an $O(n^5)$ algorithm in the presence of only required-child and required-descendant ICs (i.e., no subtypes). These three algorithms represent substantial improvement over the previously known algorithms of Amer-Yahia et al. [2]. Also, we believe that our $O(n^3)$ algorithms in Sections 3 and 5 are runtime optimal; it might be possible to improve upon the $O(n^3)$ algorithm in Section 4.

Now, let us consider extending the applicability of our algorithms. First, consider allowing TPQs to contain multiple output nodes; i.e., $op(Q)$ is a sequence (an ordered set) of nodes, instead of being a single node. For example, in Figure 1c), consider making the rightmost leaf (of type $d$) as the second output node. This corresponds to the following XQuery query:

   for $x$ in $a[.\beta[/e and ./.d]]$ return $b$

   for $y$ in $x/c/d$ return $output(x, y)$

In general, let $op(Q) = (u_1, u_2, \ldots, u_k)$; we will assume that the user’s intent is that none of the output nodes should be deleted as being redundant. The answer to $Q$ is formed from the set of sequences $(\beta(u_1), \beta(u_2), \ldots, \beta(u_k))$ of database nodes, obtained over all possible inclusions $\beta$. Our algorithms in Sections 3, 4 and 5 can be easily modified to handle this extension, without changing the runtime. The only change needed is to replace the expression “if $u = op(Q)$” by the expression “if $u \subseteq op(Q)$”, in $TPQ\_Simulation$, $Chase\_Simulation$ and $CTPQ\_Simulation$ (Figures 3, 6 and 10, respectively).
Next, consider allowing TPQs to contain nodes labeled — ("any" type), as in [20, 21, 22, 23]. Miklau and Suciu [16] show that the problem of minimizing TPQs that contain c-edges, d-edges and nodes labeled — is co-NP complete. Our O(n^3) algorithm in Section 3 can be extended to TPQs with nodes labeled — (but no d-edges). In TPQSimulation (Figure 3), we allow a node labeled — to be simulated by a node of any type \( \tau \in \Sigma \cup \{-\}; \) a node of type \( \tau \in \Sigma \) can be simulated only by a node of type \( \tau \).

The minimization algorithm in Section 5 can not be extended to allow the — label (even in the absence of d-edges), because the — label induces subtyping: \( \tau \leq \tau \), for all \( \tau \in \Sigma \). The TPQ obtained from Figure 3a, by replacing the college and dept labels with —, along with \( C = \{ \text{school}\rightarrow\text{scidept}\rightarrow\text{lab}\} \), proves this point.

Now, let us consider some other kinds of integrity constraints. Wood [20, 22] studied required-parent constraints. A required-parent constraint of the form \( \tau_1 \leftarrow \tau_2 \) means that every database node of type \( \tau_2 \) has a parent of type \( \tau_1 \). Amer-Yahia et al. [2] mentioned required-ancestor constraints that are analogous to required-parent constraints. Fan and Simeon [9] studied key, foreign key and inverse constraints. Further research is needed to study the minimization of TPQs in the presence of such constraints.

7. ACKNOWLEDGEMENTS

The author would like to thank the IEEE ICDE reviewers (who rejected an earlier, half-sized version of this paper) and the reviewers of this conference for helpful suggestions that led to improved presentation in this final version.

8. REFERENCES