Query Translation from XPath to SQL in the Presence of Recursive DTDs

Wenfei Fan[†] [*] **Jeffrey Xu Yu**[‡] **Hongjun Lu**[♮] **Jianhua Lu**[‡] **Rajeev Rastogi**[§]

[†]University of Edinburgh & Bell Laboratories, wenfei@inf.ed.ac.uk
[‡]The Chinese University of Hong Kong, Hong Kong, China, {yu,jhlu}@se.cuhk.edu.hk
[♮]The Hong Kong University of Science and Technology, Hong Kong, China, luhj@cs.ust.hk
[§]Bell Laboratories (India), rastogi@research.bell-labs.com

Abstract

The interaction between recursion in XPATH and recursion in DTDs makes it challenging to answer XPATH queries on XML data that is stored in an RDBMS via schema-based shredding. We present a new approach to translating XPATH queries into SQL queries with a simple least fixpoint (LFP) operator, which is already supported by most commercial RDBMS. The approach is based on our algorithm for rewriting XPATH queries into regular XPATH expressions, which are capable of capturing both DTD recursion and XPATH queries in a uniform framework. Furthermore, we provide an algorithm for translating regular XPATH queries to SQL queries with LFP, and optimization techniques for minimizing the use of the LFP operator. The novelty of our approach consists in its capability to answer a large class of XPATH queries by means of only low-end RDBMS features already available in most RDBMS. Our experimental results verify the effectiveness of our techniques.

1 Introduction

It is increasingly common to find XML data stored in a relational database system (RDBMS), typically based on DTD/schema-based shredding into relations [24] as found in many commercial products (e.g., [11, 19, 21]). With this comes the need for answering XML queries using RDBMS, by translating XML queries to SQL.

The query translation problem can be stated as follows. Consider a mapping τ_d, defined in terms of DTD-based shredding, from XML documents conforming to a DTD D to relations of a schema $\mathcal{R}$. Given an XML query Q, we want to find (a sequence of) *equivalent* SQL queries Q' such that for any XML document T conforming to D, Q on T can be answered by Q' on the database $\tau_d(T)$ of $\mathcal{R}$ that represents T; that is, the set of nodes (ids) selected by Q on T equals the set of (unary) tuples (encoding T nodes) selected by Q' on $\tau_d(T)$; to simplify the exposition we denote this by $Q(T) = Q'(\tau_d(T))$ in the sequel. We allow DTDs D to be recursive and consider queries Q in XPATH [6], which is essential for XML query languages XQuery and XSLT.

The query translation problem is, however, nontrivial: DTDs (or XML Schema) found in practice are often recursive [5] and complex. This is particularly evident in real-life applications (see, e.g., BIOML [4], which contains a number of nested and overlapping cycles when represented as a graph). The interaction between recursion in a DTD and recursion in an XML query complicates the translation. When the DTD has a tree or DAG structure, a natural approach [12] is based on enumerating all matching paths of the input XPATH query in a DTD, sharing common sub-paths, rewriting the paths into SQL queries, and taking a union of these queries. However, this approach no longer works on recursive DTDs since it may lead to infinitely many paths in the presence of '//' (descendants-or-self) in XPATH. Another approach is by means of a rich intermediate language and middleware as proposed in [23]: first express input XML queries in the intermediate language, and then evaluate the translated queries leveraging the computing power of the middleware and the underlying RDBMS. However, as pointed out by a recent survey [15], this approach requires implementation of the middleware on top of RDBMS, and introduces communication overhead between the middleware and the RDBMS, among other things. It is more convenient and possibly more efficient to translate XPATH queries to SQL extended with a recursion operator, and push the work (SQL queries) to the underlying RDBMS, capitalizing on the RDBMS to evaluate and optimize the queries. However, as observed by [15], although there has been a host of work on storing and querying XML using an RDBMS [7, 10, 12, 14, 23], the problem of translating recursive XML queries into SQL in the presence of recursive DTDs has not been well studied, and it was singled out as the most important open problem in [15].

Recently an elegant approach was proposed in [14] to translating path queries to SQL'99, which is capable of

[*] Supported in part by EPSRC GR/S63205/01, EPSRC GR/T27433/01 and NSFC 60228006.

Proceedings of the 31st VLDB Conference, Trondheim, Norway, 2005

translating queries with // and limited qualifiers to (a sequence of) SQL queries with the linear-recursion construct *with...recursive*. Unfortunately, this approach has several limitations. The first weakness is that it relies on the SQL'99 recursion functionality, which is not currently supported by many commercial products including Oracle and Microsoft SQL server. One wants an effective query translation approach that works with a wide variety of products supporting low-end recursion functionality, rather than requiring an advanced DBMS feature of only the most sophisticated systems. Second, the SQL queries with the SQL'99 recursion produced by the translation algorithm of [14] are typically large and complex. As a result, they may not be effectively optimized by all platforms supporting SQL'99 recursion for the same reasons that not all RDBMS platforms can effectively optimize mildly complex non-recursive queries [9]. A third problem is that path queries handled by the algorithm of [14] are too restricted to express XPATH queries commonly found in practice.

In light of this we propose a new approach to translating a class of XPATH queries to SQL, based on *regular* XPATH *expressions* introduced in [18] and a simple least fixpoint (LFP) operator. Regular XPATH expressions extend XPATH by supporting general Kleene closure E^* instead of //. The LFP operator $\Phi(R)$ takes a single input relation R instead of multiple relations as does the SQL'99 *with...recursion* operator. It is already supported by many commercial systems such as Oracle (*connectby*) and IBM DB2 (*with...recursion*), and will be supported by Microsoft SQL server 2005 (*common table* [20]). We show that regular XPATH queries are capable of expressing a large class of XPATH queries over a (recursive) DTD D. That is, regular XPATH expressions capture both DTD recursion and XPATH recursion in a uniform framework. Moreover, we show that each regular XPATH expression can be rewritten to a sequence of equivalent SQL queries with the LFP operator.

Taken together, our approach works as follows. Given an XPATH query Q, we first rewrite Q into a regular XPATH query E_Q, and then translate E_Q to an equivalent sequence Q' of SQL queries. Both E_Q and Q' are bounded by a low polynomial in the size of the input query Q and the DTD D. We provide an efficient algorithm for translating an XPATH query over a (recursive) DTD D to an equivalent regular XPATH query, and a novel algorithm for rewriting a regular XPATH query into a sequence of SQL queries with the LFP operator. Furthermore, we introduce optimization techniques to minimize the use of the LFP operator and to push selections into LFP in the rewritten SQL queries.

Contributions. Our main contributions are the following.

- The use of regular XPATH expressions to capture both DTD recursion and XPATH recursion.
- The use of the simple LFP operator found in most commercial RDBMS to answer XPATH queries.
- An efficient algorithm for translating XPATH queries into regular XPATH queries (Section 4.1).
- A novel algorithm for rewriting a regular XPATH query to SQL queries with the LFP operator (Section 5).
- Optimization techniques for minimizing the use of the LFP operator and pushing selections into LFP in the SQL translation (Sections 4.2 and 5.2).
- Experimental results verifying the effectiveness of our techniques, using real-life XML DTDs (Section 6).

Section 3 outlines our query translation approach as opposed to [14], and Section 7 discusses related work.

Our approach has several salient features. (1) It requires only low-end RDBMS features instead of the advanced SQL'99 recursion functionality. As a result it provides a variety of commercial RDBMS with an immediate capability to answer XPATH queries over recursive DTDs. (2) It produces SQL queries that are less complex than their counterparts generated with the SQL'99 recursion, and can be optimized by RDBMS platforms by existing techniques for, e.g., multi- and recursive SQL query optimization [1, 2, 3, 22]. (3) It is capable of handling a class of XPATH queries beyond those studied in [14]. (4) This work is also a concrete step toward answering XPATH queries over (virtual) recursive XML views of relational data.

2 DTD, XPath, Schema-Based Shredding

In this section, we review DTDs, XPATH queries, and DTD-based shredding of XML data into relations.

2.1 DTDs

We represent a DTD D as (Ele, Rg, r), where Ele is a set of element types; r is a root type; and Rg defines the types: for any A in Ele, $Rg(A)$ is a regular expression:

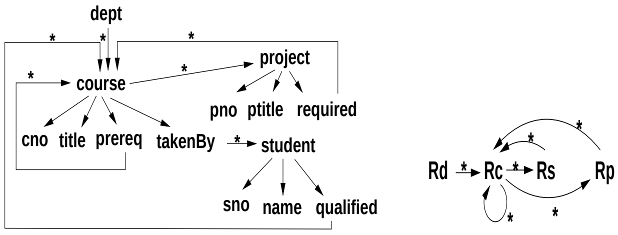
$$\alpha ::= \epsilon \ | \ B \ | \ \alpha,\alpha \ | \ (\alpha \,|\, \alpha) \ | \ \alpha^*,$$

where ϵ is the empty word, B is a type in Ele (referred to as a *subelement* or *child* type of A), and '|', ',' and '$*$' denote disjunction, concatenation and the Kleene star, respectively. We refer to $A \rightarrow Rg(A)$ as the *production* of A. To simplify the discussion we do not consider attributes, and we assume that an element v may possibly carry a text value (PCDATA) denoted by $v.val$. An XML document that conforms to a DTD is called an XML *tree of the* DTD.

As in [24], we represent DTD D as a graph, called the DTD *graph* of D and denoted by G_D. In G_D, each node represents a distinct element type A in D, called *the* A *node*, and an edge denotes the parent/child relationship. Specifically, for any production $A \rightarrow \alpha$, there is an edge from the A node to the B node for each subelement type B in α; the edge is labeled with '$*$' if B is enclosed in α_0^* for some sub-expression α_0 of α. This simple graph model for DTDs suffices since, as will be seen shortly, we do not consider ordering in XPATH. When it is clear from the context, we shall use DTD and its graph interchangeably.

A DTD is *recursive* if its DTD graph is *cyclic*. A DTD graph G_D is called a n-cycle graph if G_D contains n simple cycles in which no node appears more than once.

Example 2.1: A *dept* DTD is depicted Fig. 1 (a), which is a 3-cycle graph (Fig. 1 (b) will be described shortly). □



(a) (b)

Figure 1: A graph representation of the *dept* DTD.

F	T
-	d_1

(a) R_d

F	T
d_1	c_1
c_1	c_2
c_2	c_3
p_1	c_4
s_2	c_5

(b) R_c

F	T
c_1	s_1
c_1	s_2

(c) R_s

F	T
c_2	p_1
c_4	p_2

(d) R_p

Table 1: A database encoding an XML tree of the *dept* DTD

2.2 XPath Queries

We consider a fragment of XPATH [6] that supports recursion (descendants) and rich qualifiers, given as follows.

$$p ::= \epsilon \mid A \mid * \mid p/p \mid p//p \mid p \cup p \mid p[q]$$
$$q ::= p \mid text() = c \mid \neg q \mid q \wedge q \mid q \vee q$$

where ϵ, A and $*$ denote the *self*-axis, a label and a wildcard, respectively; '$\cup$', '$/$' and '$//$' are *union*, *child-axis* and *descendants-or-self-axis*, respectively; and q is called a *qualifier*, in which c is a constant, and p is defined above.

An XPATH query p, when evaluated at a *context node* v in an XML tree T, returns the set of nodes of T reachable via p from v, denoted by $v[\![p]\!]$. We also use $\emptyset$ to denote a special query, which returns the empty set over all XML trees, with $\emptyset \cup p$ equivalent to p and $p/\emptyset/p'$ equivalent to $\emptyset$. To simplify the discussion we assume that qualifiers $[text() = c]$ and $[\neg q]$ only appear in the form of $p[text() = c]$ and $p[\neg q]$ where p is an XPATH query that is not ϵ.

This class of XPATH queries properly contains branching path queries studied in [14] and tree patterns. In the sequel, we refer to this class of queries simply as XPATH queries.

Example 2.2: Consider two XPATH queries.

```
Q1 = dept//project
Q2 = dept/course[ε//prereq/course/cno="cs66"∧¬ε//project
       ∧ ¬takenBy/student/qualified//course/cno = "cs66"]
```

On an XML tree of the *dept* DTD of Fig.1, the first query is to find all projects, and the second one is to find courses that (1) have a prerequisite cs66, (2) have no project related to them or to their prerequisites, but (3) also have a student who registered for the course but did not take cs66. □

2.3 Mapping DTDs into a Database Schema

We focus on DTD-based shredding of XML data into relations, e.g., the shared-inlining technique of [24] as supported by most RDBMS [11, 19, 21] (see Section 7 for discussions about schema-oblivious XML storage methods). In a nutshell, a DTD-based shredding is a mapping $\tau_d : D \to \mathcal{R}$ from XML trees of DTD D to databases of relational schema $\mathcal{R}$.

To simplify the discussion we assume that τ_d maps each element of type A to a relation R_A in $\mathcal{R}$, which has three columns F (from, i.e., *parentId*), T (to, i.e., ID) and V (value of all other attributes). Intuitively, in a database $\tau_d(T_r)$ representing an XML tree T_r, each R_A tuple (f, t, v) represents an edge in T_r from a node f to an A-element t which may have a text value v, where t and f are denoted by the node IDs in T_r and are thus *unique* in the database, and v is '_' in the absence of text value at t. In particular, $f =$ '_' if f is the root of T_r. This assumption does not lose generality: our query translation techniques can be easily extended to cope with mappings without this restriction.

Example 2.3: With the shared-inlining technique, the DTD of Fig. 1 (a) is mapped to a schema with four relation schemas, R_d, R_c, R_p and R_s, representing *dept, course, project* and *student*, respectively (see Fig. 1 (b) for the simplified representation of Fig. 1 (a)). A sample database is given in Table 1, which only shows F and T columns. □

3 Overview: From XPath to SQL

The query translation problem from XPATH to SQL is stated as follows. For a mapping $\tau_d : D \to \mathcal{R}$ from XML trees of DTD D to databases of relational schema $\mathcal{R}$, it is to find an algorithm that, given an XPATH query Q, effectively computes an equivalent sequence of relational queries Q' such that for any XML tree T of the DTD D, $Q(T) = Q'(\tau_d(T))$.

In this section we first review the approach proposed by [14], the only solution published so far for the query translation problem in the presence of recursive DTDs. To overcome its limitations, we then outline a new approach. detailed algorithms are provided in the next two sections.

3.1 Linear Recursion of SQL'99

The algorithm of [14], referred to as SQLGen-R, handles recursive path queries over recursive DTDs based on SQL'99 recursion. In a nutshell, given an input path query, SQLGen-R first derives a *query graph*, G_Q, from the DTD graph to represent all matching paths of the query in the DTD graph. It then partitions G_Q into strongly-connected components $c_1, \ldots, c_n$, sorted in the top-down topological order. It generates an SQL query Q_i for each c_i, and associates Q_i with a temporary relation *TR*$_i$ such that *TR*$_i$ can be directly used in later queries Q_j for $j > i$. The sequence *TR*$_1 \leftarrow Q_1; \ldots;$ *TR*$_n \leftarrow Q_n$ is the output of the algorithm. If a component c_i is cyclic, Q_i is defined in terms of the *with...recursive* operator. More specifically, it generates an *initialization* part and a *recursive* part from c_i. The initialization part captures all "incoming edges" into c_i. The recursion part first creates an SQL query for each edge in c_i, and then encloses the union of all these (edge) queries in a *with...recursive* expression. Note that if c_i has k edges, Q_i actually calls for a fixpoint operator $\phi(R, R_1, R_2, \cdots R_k)$ with $k+1$ input relations, defined as follows:

$$R^0 \leftarrow R \qquad (1)$$
$$R^i \leftarrow R^{i-1} \cup (R^{i-1} \bowtie_{C_1} R_1) \cup \cdots \cup (R^{i-1} \bowtie_{C_k} R_k)$$

```
with
R (F, T, Rid) as (
   (select R.F, R_c.T, Rid('c')
         from R, R_c where R.T = R_c.F and Rid = 'c')
   union all    /* followed by 5 more similar select queries
                and 4 more union all operations */
```

Figure 2: The SQL statement generated by SQLGen-R

where R^0 corresponds to the initialization part, R_j corresponds to an SQL query coding an edge in c_i, and C_j is a Boolean expression on join, for each $j \in [1, k]$.

Example 3.1: Recall the mapping from the *dept* DTD to the relational schema $\mathcal{R}$ consisting of R_s, R_c, R_p, R_d given in Example 2.3, and the XPATH query Q_1 = *dept//project* given in Example 2.2, which, over the DTD graph of Fig. 1 (b), indicates $R_d//R_p$. Given Q_1 and the DTD graph of Fig. 1 (b), the algorithm SQLGen-R finds a strongly-connected component ($R_c//R_p$) having 3 nodes and 5 edges, and produces a single SQL query using a *with...recursive* expression, as shown in Fig. 2. □

Observe the following about the query of Fig. 2. First, it actually requires a fixpoint operator that takes 4 relations as input. As remarked in Section 1, the functionality of $\phi(R, R_1, R_2, \cdots R_k)$ is a high-end feature that few RDBMS support. Second, it is a complex query such that *each iteration* of the fixpoint needs to compute 5 joins and 5 unions. Third, all 5 relations join the result relation R in the center, which forms a *star* shape and is hard to optimize.

3.2 A New Approach

To this end we propose a new approach to translating XPATH queries to SQL, based on a notion of extended XPATH expressions [18] and the simple LFP operator $\Phi(R)$.

Regular XPATH expressions. A regular XPATH expression E over a DTD D is syntactically defined as follows:

$$E ::= \epsilon \mid A \mid E/E \mid E \cup E \mid E^* \mid E[q],$$
$$q ::= E \mid text() = c \mid \neg q \mid q \wedge q \mid q \vee q.$$

where A is an element type in D. The semantics of evaluating E over an XML tree is similar to its XPATH counterpart.

Regular XPATH differs from XPATH in that it supports general Kleene closure E^* as opposed to restricted recursion '//'. The motivation for using E^* instead of '//' is that with E^* one can define a finite representation of (possibly infinite) matching paths of an XPATH query over a recursive DTD. In a nutshell, E takes a union of all matching simple cycles of // and then E^* applies the Kleene closure to the union; each of these paths can then be mapped to a sequence of relations connected by joins.

The simple LFP operator. The LFP operator $\Phi(R)$ takes a single input relation R, as shown below.

$$\begin{array}{lcl} R^0 & \leftarrow & R \\ R^i & \leftarrow & R^{i-1} \cup (R^{i-1} \bowtie_C R^0) \end{array} \qquad (2)$$

where C is a Boolean expression on the join. This LFP operator is already supported by most commercial RDBMS products. For example, Fig. 3 shows an implementation of

```
LFP Φ(R) in Oracle
   select F, T from R connect by F = prior T

LFP Φ(R) in DB2
   with
   R_Φ(F, T) as (
   (select F, T from R)
   union all
   (select R_Φ.F, R.T from R_Φ, R where R_Φ.T = R.F)
```

Figure 3: An implementation of LFP in Oracle and DB2

$\Phi(R)$ in Oracle and IBM DB2 when C is simply $R_\Phi.T = R.F$, where R_Φ is the relation being computed by $\Phi(R)$.

To illustrate how the LFP operator handles Kleene closure, consider a regular XPATH query $(A_2/\cdots/A_n/A_1)^*$ representing a simple cycle $A_1 \rightarrow \cdots \rightarrow A_n \rightarrow A_1$. This query can be rewritten into $\Phi(R)$ (Eq. (2)) by letting

$$R \leftarrow \Pi_{R_2.F, R_1.T}(R_2 \bowtie R_3 \bowtie \cdots \bowtie R_n \bowtie R_1) \qquad (3)$$

Here, the projected attributes are taken from the attributes F (from) and T (to) in relations R_2 and R_1, respectively. The join between R_i/R_j is expressed as $R_i \bowtie_{R_i.T=R_j.F} R_j$, i.e., it returns R_i tuples that *connect* to R_j tuples. In general, we rewrite E^* to $\Phi(R)$, where R is a temporary relation associated with a query coding E.

In contrast to $\Phi(R)$ which takes a single input relation R, the linear-recursion operator ϕ (Eq. (1)) can take an unbounded number k of relations. One might be tempted to think that Eq. (1) can be coded with Eq. (2), as follows:

$$\begin{array}{lcl} R^0 & \leftarrow & R \\ R^i & \leftarrow & R^{i-1} \cup (R^{i-1} \bowtie R') \end{array}$$

where $R' = \cup_{j=1}^{k} R_j$. But this is incorrect since different conditions are associated with different joins in Eq. (1).

A new approach for query translation. Based on the LFP operator $\Phi(R)$ and regular XPATH, we propose a new framework for translating XPATH to SQL. As depicted in Fig. 4, the framework translates an input XPATH query Q to SQL in two steps. First, it rewrites Q over a (recursive) DTD D to an equivalent regular XPATH query E_Q over D. Second, it rewrites E_Q into an equivalent sequence Q' of SQL queries based on a mapping $\tau : D \rightarrow \mathcal{R}$, and using the LFP operator to handle Kleene closure. We provide these translation algorithms in Sections 4 and 5, which produce E_Q and Q' bounded by a low polynomial in the size $|Q|$ of the XPATH query Q and the size $|D|$ of the DTD D.

Example 3.2: Consider again evaluating the XPATH query Q_1 = *dept//project* over the *dept* DTD of Fig. 1, in the same setting as in Example 3.1. Our algorithms first translate Q_1 to a regular XPATH query $E_{Q_1} = R_d/R_c/E^*/R_p$, where $E = (R_c \cup R_s/R_c \cup R_p/R_c)$; and then rewrite E_{Q_1} to a sequence of SQL queries (written in relational algebra):

$$\begin{array}{rcl} R_{cc} & \leftarrow & R_c \\ R_{csc} & \leftarrow & \Pi_{R_s.F, R_c.T}(R_s \bowtie_{R_s.T=R_c.F} R_c) \\ R_{cpc} & \leftarrow & \Pi_{R_p.F, R_c.T}(R_p \bowtie_{R_p.T=R_c.F} R_c) \\ R & \leftarrow & R_{cc} \cup R_{csc} \cup R_{cpc} \\ R_\gamma & \leftarrow & \Phi(R) \cup \Pi_{T,T}(R_c) \\ R_f & \leftarrow & \Pi_{R_d.T, R_p.T}(R_d \bowtie_{R_d.T=R_c.F} R_c \bowtie_{R_c.T=R_\gamma.F} R_\gamma \\ & & \quad \bowtie_{R_\gamma.T=R_p.F} R_p) \end{array}$$

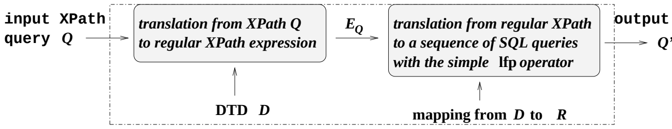


Figure 4: Translation from XPath to SQL

The above sequence is the output of our algorithms. □

Contrast Example 3.2 with the SQL query of Fig. 2. While our SQL queries use 3 unions and 5 joins in total, they are evaluated once only, instead of *once in each iteration* of the LFP computation of Fig. 2. We pull join/union out from the iteration and thus reduce the evaluation cost.

4 From XPath to Regular XPath

In this section, we first present an algorithm for rewriting an XPATH query Q over a (recursive) DTD D to an equivalent regular XPATH query E_Q over D such that for any XML tree T of D, $Q(T) = E_Q(T)$. We then introduce an optimization technique that can be incorporated into the algorithm to minimize the number of Kleene closures in E_Q.

4.1 Translation Algorithm

The algorithm, XPathToReg, is based on *dynamic programming*: for each sub-query p of the input query Q and each type A in D, it computes a local translation $E_p = \mathsf{x2r}(p, A)$ from XPATH p to a regular XPATH query E_p, such that p and E_p are equivalent when being evaluated at any A element. Composing the local translations one will get the rewriting $E_Q = \mathsf{x2r}(Q, r)$ from Q to E_Q, where r is the root type of D. For each $\mathsf{x2r}(p, A)$ the algorithm "evaluates" p over the sub-graph of the DTD graph G_D rooted at A, substituting regular expressions over element types for wildcard $*$ and descendants //, by incorporating the structure of the DTD into E_p. This also allows us to "optimize" the XPATH query by capitalizing on the DTD structure: certain qualifiers in p can be evaluated to their truth values and thus be eliminated during the translation.

To conduct the dynamic-programming computation, our algorithm uses the following variables. First, it works over a list L that is a postorder enumeration of the nodes in the parse tree of p, such that all sub-queries of p (i.e., its descendants in p's parse tree) precede p in L. Second, all the element types of the DTD D are put in a list N. Third, for each sub-query p in L and each node A in N, we use $\mathsf{x2r}(p, A)$ to denote the *local translation* of p at A, which is a regular XPATH expression. We also use $\mathsf{reach}(p, A)$ to denote the types in D that are *reachable* from A via p. Abusing this notation, we use $\mathsf{reach}([q], A)$ for a qualifier $[q]$ to denote whether or not $[q]$ can be evaluated to false at an A element, indicated by whether or not $\mathsf{reach}([q], A)$ is empty. Finally, for each A and its descendant B in the DTD graph G_D of D, we use $\mathsf{rec}(A, B)$ to denote the regular expression representing all the paths from A to B in G_D, such that $\mathsf{rec}(A, B)$ is equivalent to the XPATH query $\epsilon//B$ when being evaluated at an A element.

It is a bit tricky to compute $\mathsf{rec}(A, B)$ and $\mathsf{reach}(\epsilon//, A)$ over a recursive DTD. With the general Kleene closure,

Algorithm XPathToReg

Input: an XPATH query Q over a DTD D.
Output: an equivalent regular XPATH query E_Q over D.

1. compute the ascending list L of sub-queries in Q;
2. compute the list N of all the types in D;
3. for each p in L do
4. for each A in N do
5. if $p \neq \epsilon//$ /* $\mathsf{x2r}(\epsilon//, A)$, $\mathsf{reach}(\epsilon//, A)$ are precomputed */
6. then $\mathsf{x2r}(p, A) := \emptyset$; $\mathsf{reach}(p, A) := \emptyset$;
7. for each p in the order of L do
8. for each A in N do
9. case p of
10. (1) ϵ: $\mathsf{x2r}(p, A) := \epsilon$; $\mathsf{reach}(p, A) := \{A\}$;
11. (2) B: if B is a child type of A
12. then $\mathsf{x2r}(p, A) := B$; $\mathsf{reach}(p, A) := \{B\}$;
13. else $\mathsf{x2r}(p, A) := \emptyset$; $\mathsf{reach}(p, A) := \emptyset$;
14. (3) $*$: for each child type B of A in D do
15. $\mathsf{x2r}(p, A) := \mathsf{x2r}(p, A) \cup B$; /* $\cup$: XPATH operator */
16. $\mathsf{reach}(p, A) := \mathsf{reach}(p, A) \cup \{B\}$; /* $\cup$: set union */
17. (4) p_1/p_2: if $\mathsf{x2r}(p_1, A) = \emptyset$
18. then $\mathsf{x2r}(p, A) := \emptyset$; $\mathsf{reach}(p, A) := \emptyset$;
19. else *cons* := $\emptyset$;
20. for each B in $\mathsf{reach}(p_1, A)$ do
21. *cons* := *cons* $\cup\ \mathsf{x2r}(p_2, B)$;
22. $\mathsf{reach}(p, A) := \mathsf{reach}(p, A) \cup \mathsf{reach}(p_2, B)$;
23. if *cons* $\neq \emptyset$
24. then $\mathsf{x2r}(p, A) := \mathsf{x2r}(p_1, A)/$*cons*;
25. else $\mathsf{reach}(p, A) := \emptyset$; $\mathsf{x2r}(p, A) := \emptyset$;
26. (5) $\epsilon//p_1$: /* reach, rec are already precomputed */
27. for each child C of A do
28. if $p_1 = B/p'$ and $\mathsf{reach}(p', B) \neq \emptyset$
29. then $\mathsf{x2r}(p, A) := \mathsf{x2r}(p, A) \cup \mathsf{rec}(C, B)/\mathsf{x2r}(p', B)$;
$\mathsf{reach}(p, A) := \mathsf{reach}(p', B)$;
30. else for each B in $\mathsf{reach}(\epsilon//, C)$ do
31. if $\mathsf{x2r}(p_1, B) \neq \emptyset$
32. then $\mathsf{x2r}(p, A) := \mathsf{x2r}(p, A) \cup \mathsf{rec}(C, B)/\mathsf{x2r}(p_1, B)$
33. $\mathsf{reach}(p, A) := \mathsf{reach}(p, A) \cup \mathsf{reach}(B, p_1)$;
34. (6) $p_1 \cup p_2$: $\mathsf{x2r}(p, A) := \mathsf{x2r}(p_1, A) \cup \mathsf{x2r}(p_2, A)$;
35. $\mathsf{reach}(p, A) := \mathsf{reach}(p_1, A) \cup \mathsf{reach}(p_2, A)$;
36. (7) $p'[q]$:
37. for each B in $\mathsf{reach}(p', A)$ do
38. if $\mathsf{x2r}([q], B) = [\epsilon]$ /* $[q]$ holds at B */
39. then $\mathsf{x2r}(p, A) := \mathsf{x2r}(p, A) \cup \mathsf{x2r}(p', A)$;
40. $\mathsf{reach}(p, A) := \mathsf{reach}(p, A) \cup \{B\}$;
41. else if $\mathsf{reach}([q], B) \neq \emptyset$ /* $[q]$ is not false at B */
42. then $\mathsf{x2r}(p, A) := \mathsf{x2r}(p, A) \cup \mathsf{x2r}(p', A)[\mathsf{x2r}(q, B)]$;
43. $\mathsf{reach}(p, A) := \mathsf{reach}(p, A) \cup \{B\}$;
44. (8) $[p_1]$: $\mathsf{x2r}(p, A) := [\mathsf{x2r}(p_1, A)]$;
45. $\mathsf{reach}(p, A) := \mathsf{reach}(p_1, A)$;
46. (9) $p'[text() = c]$: $\mathsf{x2r}(p, A) := \mathsf{x2r}(p', A)[text() = c]$;
47. $\mathsf{reach}(p, A) := \mathsf{reach}(p', A)$;
48. (10) $[q_1 \wedge q_2]$: if $\mathsf{reach}(q_1, A) \neq \emptyset$ and $\mathsf{reach}(q_2, A) \neq \emptyset$
49. then $\mathsf{x2r}(p, A) := [\mathsf{x2r}([q_1], A) \wedge \mathsf{x2r}([q_2], A)]$;
50. $\mathsf{reach}(p, A) := \{true\}$;
51. else $\mathsf{x2r}(p, A) := \emptyset$; $\mathsf{reach}(p, A) := \emptyset$;
52. (11) $[q_1 \vee q_2]$: if $\mathsf{reach}(q_1, A) \neq \emptyset$ and $\mathsf{reach}(q_2, A) \neq \emptyset$
53. then $\mathsf{x2r}(p, A) := [\mathsf{x2r}([q_1], A) \vee \mathsf{x2r}([q_2], A)]$;
54. else if $\mathsf{reach}(q_1, A) \neq \emptyset$ and $\mathsf{reach}(q_2, A) = \emptyset$
55. then $\mathsf{x2r}(p, A) := [\mathsf{x2r}([p_1], A)]$;
56. else if $\mathsf{reach}(q_1, A) = \emptyset$ and $\mathsf{reach}(q_2, A) \neq \emptyset$
57. then $\mathsf{x2r}(p, A) := [\mathsf{x2r}([p_2], A)]$;
58. else $\mathsf{x2r}(p, A) := \emptyset$;
59. $\mathsf{reach}(p, A) := \mathsf{reach}(q_1, A) \cup \mathsf{reach}(q_2, A)$;
60. (12) $p'[\neg q]$: if $\mathsf{reach}(q, B) = \emptyset$ for all $B \in \mathsf{reach}(p', A)$
61. then $\mathsf{x2r}(p, A) := \mathsf{x2r}(p', A)$;
62. $\mathsf{reach}(p, A) := \{true\}$;
63. else $\mathsf{x2r}(p, A) := \mathsf{x2r}(p', A)[\neg\mathsf{x2r}([q], A)]$;
64. $\mathsf{reach}(p, A) := \mathsf{reach}(p', A)$;
65. optimize $\mathsf{x2r}(Q, r)$ by removing $\emptyset$ using $\emptyset \cup E = E$, $E_1/\emptyset/E_2 = \emptyset$
66. return $\mathsf{x2r}(Q, r)$; /* r is the root of D */

Figure 5: Rewriting algorithm from XPath to regular XPath

one can compute these by using, e.g., Tarjan's fast algorithm [25], which finds a regular expression representing all the paths between two nodes in a (cyclic) graph. Thus $\mathsf{rec}(A, B), \mathsf{reach}(\epsilon//, A)$ can be computed by:

for each A in N
  for each descendant B of A do
    $\mathsf{rec}(A, B)$:= the regular expression found by the algorithm of [25];
    $\mathsf{reach}(\epsilon//, A)$:= $\mathsf{reach}(\epsilon//, A) \cup \{B\}$;

The fast algorithm takes $O(|D| \log |D|)$ time, and thus so is the size of $\mathsf{rec}(A, B)$. In Section 4.2 we shall present another algorithm for computing $\mathsf{rec}(A, B)$. Note that $\mathsf{rec}(A, B)$ is determined by the DTD D regardless of the input query Q; thus it can be precomputed for each A, B, once and for all, and made available to XPathToReg. A second issue concerns the special query $\emptyset$, which returns an empty set over any XML tree, as described in Section 2. In our translation we use $\emptyset$ for optimization purposes.

Algorithm XPathToReg is given in Fig. 5. It computes $E_Q = \mathsf{x2r}(Q, r)$ as follows. It first enumerates the list L of sub-queries in Q and the list N of element types in D, as well as initializes $\mathsf{x2r}(p, A)$ to the special query $\emptyset$ and $\mathsf{reach}(p, A)$ to empty set for each $p \in Q$ and $A \in N$ (lines 1–6). Then, for each sub-query p in L in the topological order and each element type A in N, it computes the local translation $\mathsf{x2r}(p, A)$ (lines 7–63), bottom-up starting from the inner-most sub-query of Q. To do so, it first computes $\mathsf{x2r}(p_i, B_j)$ for each (immediate) sub-query p_i of p at each possible DTD node B_j under A (i.e., B_j in reach (p, A)); then, it combines these $\mathsf{x2r}(p_i, B_j)$'s to get $\mathsf{x2r}(p, A)$. The details of this combination are determined based on the formation of p from its immediate sub-queries p_i, if any (cases 1-12). In particular, in the case $p = \epsilon//p_1$ (case 5), it ranges over the children C of A to compute $\mathsf{rec}(C, \_)$ instead of $\mathsf{rec}(A, \_)$ since the context node A is already in the latter, where '_' denotes an arbitrary type. We also single out a special case, namely, when p_1 is of the form B/p', and handle it by using $\mathsf{rec}(C, B)/\mathsf{x2r}(p', B)$. Note that when p is a qualifier $[q]$ (cases 7–12), it may evaluate $[q]$ to a truth value (ϵ for *true* and $\emptyset$ for *false*) in certain cases based on the structure of the DTD D, and thus optimize the query evaluation. At the end of the iteration $E_Q = \mathsf{x2r}(Q, r)$ is obtained, optimized by removing $\emptyset$, and returned as the output of the algorithm (lines 64–65).

Example 4.1: Recall the XPATH query Q_2 from Example 2.2. Observe that the algorithm of [14] *cannot* handle this query over the *dept* DTD of Fig. 1 (a). In contrast, XPathToReg translates Q_2 to the regular XPATH query

$$E_{Q_2} = dept/course[E_{course\_course}/prereq/course/cno="cs66" \wedge \neg\, E_{course\_project} \wedge \neg\, takenBy/student/E_{qualified\_course}/cno="cs66"].$$

where the following is computed by Tarjan's algorithm:

$$E_{course\_course} = \mathsf{rec}\ (course, course) = course/E_1^* \cup E_2^+/E_1^*,$$
$$E_{course\_project} = \mathsf{rec}\ (course, project) = (course/E_1^* \cup E_2^+/course/E_1^*)/project,$$
$$E_{qualified\_course} = \mathsf{rec}\ (qualified, course) = qualified/course/E_1^* \cup (qualified/E_2)^+/course/E_1^*,$$
$$E_1 = prereq/course \cup takenBy/student/qualified/course$$
$$E_2 = course/E_1^*/project/required$$

The algorithm to be given in the next section can then translate E_{Q_2} to equivalent relational queries. $\Box$

Algorithm XPathToReg takes at most $O(|Q| * |D|^3)$ time, since each step in the iteration takes at most $O(|D|)$ time except that case 5 may take $O(|D|^2)$ time, the size of the list L is linear in the size of Q, and variables $\mathsf{rec}(A, B)$ are precomputed as soon as the DTD D is available. Furthermore, taken together with the complexity of Tarjan's algorithm [25] the size of the output E_Q is at most $O(|Q| * |D|^4 log|D|)$. One can verify the following.

Theorem 4.1: *Each* XPATH *query Q over a* DTD *D can be rewritten to an equivalent regular* XPATH *expression E_Q over D of size $O(|Q| * |D|^4 log|D|)$.* $\Box$

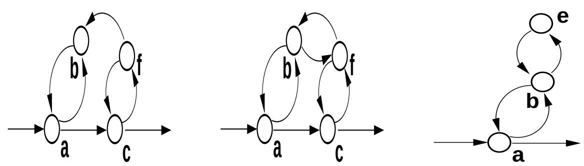
Observe the following. First, regular XPATH queries capture DTD *recursion and* XPATH *recursion* in a uniform framework by means of the general Kleene closure E^*. Second, during the translation, algorithm XPathToReg conducts *optimization leveraging the structure of the* DTD. Third, Kleene closure is only introduced when computing $\mathsf{rec}(A, B)$; thus there are no qualifiers *within* a Kleene closure E^* in the output regular query. Fourth, both $|Q|$ and $|D|$ are far smaller than the data (XML tree) size in practice.

4.2 Optimization via Cycle Contraction

A major criterion for computing a regular XPATH query E_Q is that the SQL query Q' translated from E_Q should be efficient. Among the relational operators in Q', LFP is perhaps the most costly. Thus, one wants E_Q to contain as few Kleene closures as possible. In other words, among possibly many regular expressions representing all the paths from A to B in a graph, we want to choose one for $\mathsf{rec}(A, B)$ with a minimal number of E^*'s. It is clear from Example 4.1 that the regular expressions $\mathsf{rec}(A, B)$ computed by the algorithm of [25] may contain excessively many E^*'s. Indeed, the focus of Tarjan's algorithm is the efficiency for finding *any* regular expression representing paths between two nodes, rather than the one with the least number of E^*'s. Furthermore, it is not realistic to expect an efficient algorithm to find $\mathsf{rec}(A, B)$ with the least number of E^*'s: this problem is PSPACE-hard (by reduction from the equivalence problem for regular expressions).

In response to this, we propose a new algorithm for computing $\mathsf{rec}(A, B)$, referred to as Cycle-C, which is a heuristic for minimizing the number of Kleene closures in a resulting regular XPATH query. As will be seen in Section 6, Cycle-C outperforms the algorithm of [25] in many cases.

Algorithm Cycle-C is based on the idea of *graph contraction*: given a DTD graph G_D, it repeatedly contracts simple cycles of G_D into nodes and thus reduces the interaction between these cycles in $\mathsf{rec}(A, B)$. In a nutshell, it first enumerates all distinct simple paths (i.e., paths without repeating labels) between A and B in G_D, referred to as *key label paths* and denoted by *AB-path*s. Assume that all the *AB-path*s are $L_1, \ldots, L_n$, where each L_i is of the form $A_1 \rightarrow \ldots \rightarrow A_k$, with $A = A_1$ and $B = A_k$. It encodes L_i with a regular expression E_i, which has an



(a) 3 cycles (b) 4 cycles (c) 2 cycles

Figure 6: $\text{rec}(a, c)$

initial value $A_1/\ldots/A_k$. Then, for each simple cycle C_j "connected" to A_i, the algorithm encodes C_j with a simple regular expression $E^*_{C_j}$, where E_{C_j} represents the simple path of C_j. It *contracts* C_j to the node A_i and replaces A_i in E_i with $A_i/E^*_{C_j}$; as a result of the contraction, cycles that were not directly connected to L_i may become directly connected to L_i. The algorithm repeats this process until all the cycles connected to L_i, directly or indirectly, have been incorporated into E_i. One can verify that $\text{rec}(A, B)$ is indeed $(E_1 \cup \ldots \cup E_n)$. Note that all simple cycles of a directed graph can be efficiently identified [26].

Below we discuss various cases dealt with by the Cycle-C algorithm, starting from simple ones.

Case-1: A DTD graph G_D has a single *AB-path* $L = A_1 \to \ldots \to A_k$ and a single simple cycle C connected to L.

First, assume that $A_i \in G_D$ is the only node shared by L and $C = A_i \to A'_1 \to \ldots \to A'_m \to A_i$. Then, the regular expression $E = E_a/E_\gamma/E_b$ suffices to capture all the paths between A and B, where $E_a = A_1/\ldots/A_i$, $E_b = A_{i+1}/\ldots/A_k$, and E_γ is E^*_C with $E_C = A'_1/\ldots/A'_m/A_i$.

Second, suppose that L and C share more than one node, say, A_i and A_j. It is obvious that we only need to incorporate C into E at one of those nodes, either at A_i or A_j, because E_γ has already covered the connections between A_i and A_j. Thus E is the same as the one given above. This property allows us to find E_γ using an arbitrary node A_i shared by multiple simple cycles.

Case-2. There exist a single *AB-path* L and multiple simple cycles $C_1, \cdots, C_n$, while all these cycles share a single node A_i on L. Here the regular expression E is a mild extension of case-1: E is $E_a/E_\gamma/E_b$ while $E_\gamma = (E_{C_1} \cup E_{C_2} \cup \cdots \cup E_{C_n})^*$, and E_{C_i} codes C_i as above.

Example 4.2: Such a case was given in Example 3.2. Consider $R_d//R_p$ over the DTD graph Fig. 1 (b). The graph has 3 simple cycles, $R_c \to R_c$, $R_c \to R_s \to R_c$ and $R_c \to R_p \to R_c$. The only *AB-path* is $L = R_d \to R_c \to R_p$ (i.e, *dept* → *course* → *project*). Here, R_c is the node shared by all the three cycles and L. The resulting regular XPATH query is then $R_d/R_c/((R_c \cup R_s/R_c \cup R_p/R_c)^*)/R_p$. □

Case-3. There exist a single *AB-path* L and multiple simple cycles $C_1, \cdots, C_n$, but not all the cycles share a node on L. For example, Fig. 6 (a) shows a DTD graph with 3 simple cycles $C_1 = a \to b \to a$, $C_2 = c \to f \to c$, and $C_3 = a \to c \to f \to b \to a$. Consider $\text{rec}(a, c)$, for which the only *AB-path* is $L = a \to c$. While C_1 and C_3 share a on L, and C_2 and C_3 share c, but not all the 3 cycles share a or c as a common node. Given these Cycle-C first generates $E = a/c$. Then, it contracts C_1, C_3 and replaces a with a regular expression a/E_{γ_1}, capturing paths from a to a via C_1 and C_3. It then contracts C_2 and C_3 by replacing c with c/E_{γ_2}, covering paths from c to c via C_2 and C_3. The final result is $E = a/E_{\gamma_1}/c/E_{\gamma_2}$. Observe the following. First, E_{γ_2} covers all possible paths that traverse E_{γ_1} since E_{γ_2} includes E_{γ_1} by replacing a with E_{γ_1}, and E covers all possible paths between a and c. Second, the processing order of the cycles is not sensitive. We can also first process C_2 and C_3 and obtain E_{γ_2}, and then let E_{γ_1} include E_{γ_2} by replacing c with E_{γ_2}.

```
Algorithm Cycle-C(G_D, A, B)
Input: a DTD graph G_D and two nodes A, B in G_D.
output: rec(A, B) in G_D.
find all distinctive AB-paths, L_1, L_2, ..., L_k, between A and B;
for each L_i do
    G_i := the subgraph including all simple cycles that
           are connected L_i directly and indirectly;
for each L_i = A_1 → ... → A_k do
  E_i := A_1/.../A_k;
  C_i := a list of all simple cycles in G_i found by the algorithm of [26]
         and sorted in topological order based on their distance to L_i
         from the farthest to those directly connected to L_i;
  for each cycle C in C_i in the order of C_i do
    if C does not directly connect to L_i
    then find node A_x on C with the shortest distance to L_i;
         G_x := the subgraph consisting of C;
         E_C := Cycle-C(G_x, A_x, A_x);   /* contract C to A_x */
         replace A_x and C with E_C^* in G_i;
  identify the nodes A'_1, ..., A'_m shared by simple cycles with L_i;
  for each A'_j shared by cycles C_1, ..., C_l
    E_{A_j} := a regular expression representing C_1, ..., C_l,
               computed based on cases 1–3 described earlier;
    replace A_j in E_i with A_j/E^*_{A'_j};
return E = E_1 ∪ ··· ∪ E_n;
```

Figure 7: Algorithm for computing $\text{rec}(A, B)$

Case-4. There are multiple *AB-paths*. Figure 6 (b) shows a DTD graph with 4 simple cycles $C_1 = a \to b \to a$, $C_2 = c \to f \to c$, $C_3 = a \to c \to f \to b \to a$, and $C_4 = b \to f \to b$. Consider $\text{rec}(a, c)$, which has two *AB-paths*: $L_1 = a \to c$ and $L_2 = a \to b \to f \to c$. On L_1 there are three simple cycles: C_1, C_2 and C_3, and on L_2 there are C_1, C_2 and C_4. Here the regular XPATH query is $E_{L_1} \cup E_{L_2}$, where each E_{L_i} is generated based on the single *AB-path* cases above.

Case-5. There are a single *AB-path* L and multiple simple cycles, but not all cycles are directly connected to L. For example, Fig. 6 (c) shows a DTD graph with 2 simple cycles $C_1 = a \to b \to a$ and $C_2 = b \to e \to b$. Consider $\text{rec}(a, a)$, for which the *AB-path* is a. Note that C_2 does not directly connect to a, but it is on C_1. It can be processed as follows. (1) We generate a regular expression $E = a$. (2) We contract C_2, generate E_{C_2} to capture C_2 and replace b in C_1 with b/E_{C_2}. (3) We contract C_1 and replace a with a/E_{C_1}, which includes E_{C_2}.

Putting these cases together, we present the Cycle-C algorithm in Fig. 7. It takes as input a DTD graph G_D and nodes A and B in G_D, and returns a regular expression $\text{rec}(A, B)$ as output. More specifically, it first identifies all the *AB-paths* $L_1, \ldots, L_n$ in G_D and for each L_i, finds the subgraph G_i that consists of L_i along with all the simple cycles that are connected to L_i directly or indirectly (lines 1–2). For each L_i, it finds all the simple cycles $\mathcal{C}_i$ using the algorithm of [26]. It then topologically sorts these cycles based on their shortest instance to any node on L_i (line 6). For each of these cycles starting from the one with the longest distance to L_i, it contracts the cycle based on case-5 (lines 4–12). It identifies all A_j nodes shared by some simple cycles (line 13) with L_i, and contracts those simple cycles to a single node based on cases 1–3 (line 14-16). Finally, it produces and returns the resulting regular expression based on case 4 (line 17). One can verify that $\mathsf{rec}(A, B)$ returned by Cycle-C captures all and only the paths between A and B in G_D.

Example 4.3: Recall the regular XPATH query E_{Q_2} from Example 4.1, which is generated from the XPATH query Q_2 by algorithm XPathToReg. Using Cycle-C, we get

E_{course_course} = *course/*E_{cc}*,*
$E_{course\_project}$ = *course/*E_{cc}*/project,*
$E_{qualified\_course}$ = *qualifi ed/course/*E_{cc}*,*
E_{cc} = *(*$E_1$$\cup$ *project/required/course)***,*
E_1 *is the same as the one given in Example 4.1.*

These are notably simpler than their counterparts in Example 4.1 computed by Tarjan's algorithm. □

5 From Regular XPath Expressions to SQL

In this section we present an algorithm for rewriting regular XPATH queries into SQL with the simple LFP operator, and an optimization technique for pushing selections into LFP.

5.1 Translation Algorithm

Consider a mapping $\tau_d : D \rightarrow \mathcal{R}$ from XML trees of a DTD D to relations of a schema $\mathcal{R}$. Given a regular XPATH query E_Q over D, we compute a sequence Q' of equivalent relational-algebra (RA) queries with the simple LFP operator Φ such that $E_Q(T) = Q'(\tau_d(T))$ for any XML tree T of D. The RA query Q' can be easily coded in SQL.

A subtle issue is that the LFP operator Φ supports $(E)^+$ but not $(E)^*$. Thus $(E)^*$ needs to be converted to $\epsilon \cup (E)^+$. To simplify the handling of ϵ, we assume a relation R_{id} consisting of tuples $(v, v, v.val)$ for all nodes (IDs) v in the input XML tree except the root r. Note that R_{id} is the identity relation for join operation: $R \bowtie R_{id} = R_{id} \bowtie R = R$ for any relation R. With this we translate $(E)^*$ to $\Phi(R) \cup R_{id}$, where R codes E and R_{id} tuples will be eliminated at a later stage. We rewrite ϵ into R_{id} just to simplify the presentation of our algorithm; a more efficient translation is adopted in our implementation.

We now give our translation algorithm, RegToSQL, in Fig. 8. The algorithm takes a regular XPATH query E_Q over the DTD D as input, and returns an equivalent sequence Q' of RA queries with the LFP operator Φ as output. The algorithm is based on dynamic programming: for each sub-expression e of E_Q, it computes $\mathsf{r2s}(e)$, which is the RA query translation of e; it then associates $\mathsf{r2s}(e)$ with a

```
Algorithm RegToSQL

Input: a regular XPATH expression E_Q over a DTD D.
Output: an equivalent list Q' of RA queries over R, where τ : D → R.
1.  compute the ascending list L of sub-expressions in E;
2.  Q' := empty list [ ];
3.  for each e in the order of L do
4.      case e of
5.      (1)  ε:  r2s(e) := R_id;
6.      (2)  A:  r2s(e) := R_A;
7.      (3)  e1/e2:  let R1 = r2s(e1), R2 = r2s(e2);
8.                r2s(e) := Π_{R1.F,R2.T,R2.V}(R1 ⋈_{R1.T=R2.F} R2);
9.      (4)  e1 ∪ e2:  let R1 = r2s(e1), R2 = r2s(e2);
10.                 r2s(e) := R1 ∪ R2;
11.     (5)  E*:  let R = r2s(e);
12.            r2s(e) := Φ(R) ∪ R_id;
13.     (6)  e1[q]:  let R1 = r2s(e1), Rq = r2s(q);
14.          r2s(e) := Π_{R1.F,R1.T,R2.V}(R1 ⋈_{R1.T=Rq.F} Rq);
                 /* returns R1 tuples that connect with R2 tuples */
15.     (7)  [e1]:  r2s(e) := r2s(e1);
16.     (8)  e1[text() = c]:  let R1 = r2s(e1);
17.                        r2s(e) := σ_{R1.V=c} R1;
                           /* select tuples t of R1 with t.V = c */
18.     (9)  [q1 ∧ q2]:  let R1 = r2s(q1);  R2 = r2s(q2);
19.                  r2s(e) := R1 ∪ R2 \ ((R1 \ R2) ∪ (R2 \ R1));
                       /* r2s(e) = R1 ∩ R2; */
20.     (10) [q1 ∨ q2]:  let R1 = r2s(q1);  R2 = r2s(q2);
21.                  r2s(e) := R1 ∪ R2;
22.     (11) e1[¬q]:  let Rq = r2s(q),   R1 = r2s(e1);
23.                  r2s(e) := R1 \ Π_{R1.F,R1.T,R1.V}
                                  (R1 ⋈_{R1.T=Rq.F} Rq);
                  /* only R1 tuples not connecting to any Rq tuple */
24.     Q' := (R_e ← r2s(e)) :: Q';   /* add r2s(e) to Q' */
25. r2s(E_Q) := σ_{F='_'} r2s(E_Q);  /* select nodes reachable from root */
26. Q' := r2s(E_Q) :: Q';
27. optimize Q' by extracting common sub-queries;
28. return Q';
```

Figure 8: Rewriting algorithm from regular XPath to SQL

temporary table R_e (which is used in later queries) and increments the list Q' with $R \leftarrow \mathsf{r2s}(e)$. More specifically, $\mathsf{r2s}(e)$ is computed from $\mathsf{r2s}(e_i)$ where e_i's are its immediate sub-queries. Thus upon the completion of the processing one will get the list Q' equivalent to E_Q. To do this, the algorithm first finds the list L of all sub-expressions of E_Q and topologically sorts them in ascending order (line 1). Then, for each sub-query e in L, it computes $\mathsf{r2s}(e)$ (lines 3–23), bottom-up starting from the inner-most sub-query of E_Q, and based on the structure of e (cases 1-11). In a nutshell, it encodes different cases of e as follows.

(1) A label A in terms of the relation R_A (case 2).

(2) Concatenation '/' with projection Π and join $\bowtie$ (case 3).

(3) Union and disjunction with union $\cup$ in RA (cases 4, 10).

(4) Kleene closure $(E)^*$ with the LFP operator Φ (case 5).

(5) $e_1[q]$ is converted to a RA query $\mathsf{r2s}(e)$ that returns only those $\mathsf{r2s}(e_1)$ tuples t_1 for which there exists a $\mathsf{r2s}(q)$ tuple t_2 with $t_1.T = t_2.F$, i.e., when the qualifier q is satisfied at the node represented by $t_1.T$ (case 6).

On the other hand, it rewrites $e_1[\neg q]$ to a RA query $\mathsf{r2s}(e)$ that returns only those $\mathsf{r2s}(e_1)$ tuples t_1 for which there exists no $\mathsf{r2s}(q)$ tuple t_2 such that $t_1.T = t_2.F$, i.e., when the qualifier q is not satisfied at the node $t_1.T$ (and hence $[\neg q]$ is satisfied at $t_1.T$; case 11); this captures the semantics of negation in XPATH (recall our assumptions about $[\neg q]$ and $[text() = c]$ from Section 2).

(6) $[e_1]$ is rewritten into r2s(e_1) (case 7).

(7) $e_1[text() = c]$ in terms of selection σ that returns all tuples of r2s(e_1) that have the text value c (case 8).

(8) Conjunction $q_1 \wedge q_2$ in terms of set intersection implemented with union $\cup$ and set difference $\setminus$ in RA (case 9).

In each of the cases above, the list Q' is incremented by adding $R_e \leftarrow$ r2s(e) to Q' as the head of Q' (line 24). Finally, after the iteration it yields $\pi_T\ \sigma_{F='_'}$r2s(E_Q) (line 25), which selects only those nodes reachable from the root of the XML tree, removing unreachable nodes including those introduced by R_{id}. It also optimizes the sequence Q' of RA queries by eliminating empty set and extracting common sub-queries (details omitted from Fig. 8), and returns the cleaned Q' (lines 27–28).

One can verify that Q', in its reverse order, is a sequence of RA queries equivalent to the regular XPATH query E_Q.

Example 5.1: Recall the XPATH query Q_2 from Example 2.2, and its regular XPATH translation E_{Q_2} from Example 4.1, which contains $E_{\text{course\_course}}$, $E_{\text{course\_project}}$ and $E_{\text{qualified\_course}}$ generated by Cycle-C and given at the end of Section 4. Given E_{Q_2}, the RegToSQL algorithm generates the RA translation below:

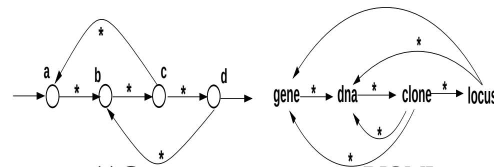
E_{cc} : R_γ *with* LFP*, the same as the one in Example 3.2.*
E_{course_course} : $R_{cc} \leftarrow R_c \bowtie R_\gamma$,
$E_{course\_project}$: $R_{cp} \leftarrow R_c \bowtie R_\gamma \bowtie R_p$,
$E_{qualified\_course}$: $R_{qc} \leftarrow R_{cc}$,
$E_{course\_course}/prereq/course/cno = "cs66"$
: $R_1 \leftarrow \sigma_{cno="cs66"}(R_{cc} \bowtie R_c)$
$takenBy/student/E_{qualified\_course}/cno = "cs66"$
: $R_2 \leftarrow \sigma_{cno="cs66"}(R_s \bowtie R_{qc})$

Note that Q_2 is of the form (with a complex qualifier) *dept/course*$[q_1 \wedge \neg q_2 \wedge \neg q_3]$, which is handled by our algorithms by treating it as Q_2^1 = *dept/course*$[q_1]$, $Q_2^2 = Q_2^1[\neg q_2]$ and $Q_2 = Q_2^2[\neg q_3]$. Thus $Q_2^1 \leftarrow R_d \bowtie R_c \bowtie R_1$, $Q_2^2 \leftarrow Q_2^1 \setminus (Q_2^1 \bowtie R_{cp})$, and E_{Q_2} becomes $Q_2^2 \setminus (Q_2^2 \bowtie R_2)$ where projections are omitted. In contrast, the algorithm of [14] cannot translate XPATH queries of this form. □

Algorithm RegToSQL takes at most $O(|E_Q|)$ time. Combined with the complexity of Algorithm XPathToReg (Theorem 4.1), one can verify the following:

Theorem 5.1: *Each* XPATH *query Q over a* DTD *D can be rewritten to an equivalent sequence of* SQL *queries (with the* LFP *operator) of total size $O(|Q| * |D|^4 log|D|)$.* □

Observe the following. First, algorithm RegToSQL shows that the simple LFP operator $\Phi(R)$ suffices to express XPATH queries over recursive DTDs; thus there is no need for the advanced SQL'99 recursion operator. Second, the total size of the produced SQL queries is bounded by a low polynomial of the sizes of the input XPATH query Q and the DTD D. Finally, the algorithms XPathToReg and RegToSQL can be easily combined into one; we present them separately to focus on their different functionality.



(a) Cross (b) BIOML

Figure 9: DTD Graphs

5.2 Pushing Selections into the LFP Operator

Algorithms XPathToReg and RegToSQL show that SQL with the simple LFP operator is powerful enough to answer XPATH queries over recursive DTDs. While certain optimizations are already conducted during the translation, other techniques, e.g., sophiscated methods for pushing selections/projections into the LFP operator [1, 2, 3]. can be incorporated into our translation algorithms to further optimize generated relational queries.

We next show how to push selections into LFP. Consider an XPATH query $Q_3 = R_d$*[id = a]*$/R_c//R_p$. To simplify the discussion, assume that our algorithms rewrite Q_3 into $R_1 \leftarrow Q_d$ and $R_2 \leftarrow$ LFP(R_0), where Q_d and LFP(R_0) compute R_d*[id = a]* and $R_c//R_p$, respectively. While $R_1 \bowtie R_2$ yields the right answer, we can improve the performance by pushing the selection into the LFP computation such that it only traverses "paths" starting from the R_c children of those R_d nodes with *id = a*. Recall from Eq. (2) that one can specify a predicate C on the join between R_Φ and R_0 in LFP, where R_0 is the input relation and R_Φ is the relation being computed by the LFP (Section 3; supported by *connectby* of Oracle and *with...recursion* of IBM DB2). Here C can be given as $R_\Phi.F \in \pi_T(R_1) \wedge R_\Phi.T = R_0.F$ ('$\in$' denotes *in* in SQL), i.e., besides the equijoin $R_\Phi.T = R_0.F$ we want the F (*from*) attribute of R_Φ to match a T (*to*) attribute of R_1. Then, each iteration of the LFP only adds tuples (f, t), where f is a child of a node in $\pi_T(R_1)$. Similarly, the selection in $R_d//R_c/R_p$*[id=c]* can be pushed into LFP(R_0) for rec(R_d, R_c). Indeed, let R_1 be the relation found for R_p*[id=c]*, and the LFP join condition be: $R_\Phi.F = R_0.T \wedge R_\Phi.T \in \pi_F(R_1)$. Then the LFP only returns tuples of the form (f, t), where t is the parent of a node in $\pi_F(R_1)$. As will be seen in Section 6, this optimization is effective.

6 A Performance Study

To verify the effectiveness of our rewriting and optimization algorithms, our performance study evaluated XPATH queries using an RDBMS with three approaches: (1) the SQLGen-R algorithm [14] using the *with...recursive* operator, (2) our rewriting algorithms by using Tarjan's method (referred to as Cycle-E as it is based on cycle expansion) to find rec(A, B), i.e., paths from node A to B in a DTD graph, and (3) our rewriting algorithms by using Cycle-C of Fig. 7 to compute rec(A, B), referred to as Cycle-C.

We experimented with these algorithms using (a) a simple yet representative DTD depicted in Fig. 9 (a) (2 cross cycles), and (b) a real-life DTD as shown in Fig. 9 (b), which is a 4-cycle DTD extracted from **BIOML** [4].

While testing several different types of XPATH queries, our performance study focused on the evaluation of // because // is the only operator in XPATH queries that, in the presence of recursive DTDs, leads to LFP in RDBMS, which is a dominant factor of performance.

Implementation. We have implemented a prototype system supporting SQLGen-R, Cycle-E and Cycle-C, using Visual C++, denoted by **R**, **E** and **C** in the figures, respectively. We executed rewritten SQL queries in a batch. We only implemented some basic optimizations, e.g., common sub-expressions were executed only once. We conducted experiments using IBM DB2 (UDB 7) on a single 2GHz CPU with 1GB main memory. We did not experiment with Oracle because Oracle does not support the SQL'99 recursion. The queries output ancestor-descendant pairs.

Testing Data: Testing data was generated using IBM XML Generator (*http://www.alphaworks.ibm.com*). The input to the Generator is a DTD file and a set of parameters. We mainly controlled two parameters, X_L and X_R, where X_L is the maximum number of levels in the resulting XML tree, and X_R is the maximum number of children of any node in the tree. Together X_L and X_R determine the shape of an XML tree: the larger the X_L value, the deeper the generated XML tree; and the larger the X_R value, the wider the tree. The default values used in our testing for X_L and X_R were 4 and 12, respectively. The default number of elements in a generated XML tree was 120,000. There is a need to control the sizes of XML trees to be the same in different settings for comparison purposes, and thus excessively large XML trees generated were trimmed. For the other parameters of the Generator, we used their default settings.

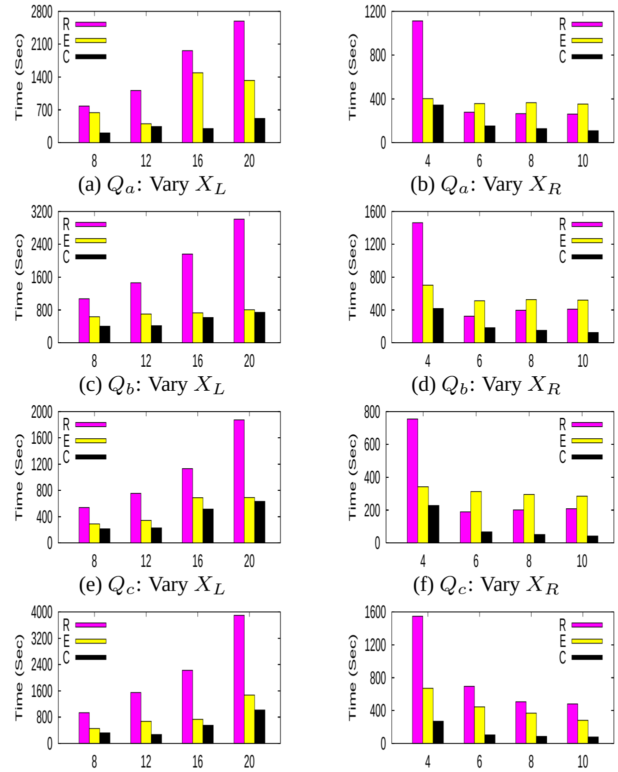
Relational Database. The XML data generated was mapped to a relational database using shared-inlining [24]. Indexes were generated for all possible joined attributes.

Query Evaluation. (1) We tested four XPATH queries using different databases (fixing the database size while varying the relations sizes). (2) We evaluated the optimization technique of Section 5.2 by comparing SQL queries translated from XPATH queries with and without pushing selections into the LFP operator. (3) We tested the scalability of our generated SQL queries w.r.t. different database sizes using a query containing //. These were conducted with the simple cross-cycle DTD graph. (4) We tested several XPATH queries with different DTDs, which are subgraphs of the real-life BIOML DTD, using the same database. The main difference between (1) and (4) is that the former tested the same queries with different databases, and the latter tested different queries with the same database.

6.1 Exp-1: Evaluation of Selective Queries

In this study, over the simple cross-cycle DTD (Fig. 9 (a)), we tested the following four XPATH queries:

- $Q_a = a/b//c/d$ (with //),
- $Q_b = a[\epsilon//c]//d$ (a twig join query),
- $Q_c = a[\neg\epsilon//c]$ (with $\neg$ and //), and
- $Q_d = a[\neg\epsilon//c \vee (b \wedge \epsilon//d)]$ (with $\neg$, $\vee$, $\wedge$ and //).



(a) Q_a: Vary X_L (b) Q_a: Vary X_R
(c) Q_b: Vary X_L (d) Q_b: Vary X_R
(e) Q_c: Vary X_L (f) Q_c: Vary X_R
(g) Q_d: Vary X_L (h) Q_d: Vary X_R

Figure 10: Processing time for cross cycles (Fig. 9 (a)).

The XPathToReg algorithm rewrites these queries into four XPATH regular queries, namely, $Q'_a = a/E_{\text{b\_c}}/d$, $Q'_b = a[E_{\text{a\_b}}/c]/E_{\text{a\_c}}/d$, $Q'_c = a[\neg E_{\text{a\_b}}/c]$, and $Q'_d = a[\neg E_{\text{a\_b}}/c \vee (b \wedge E_{\text{a\_c}}/d)]$, while Cycle-E generates:

$$
\begin{aligned}
E_{b\_c} &= \text{rec}(b,c) = (E_{bb} \cup (E_{bb}/c/a/(E_{bb}/c/a)^*/E_{bb}))/c \\
E_{a\_b} &= \text{rec}(a,b) = a/(E_{bb}/c/a)^*/E_{bb} \\
E_{a\_c} &= \text{rec}(a,c) = a/(E_{bb}/c/a)^*/E_{bb}/c \\
E_{bb} &= b/(c/d/b)^*
\end{aligned}
$$

In contrast, Cycle-C generates the following:

$$
\begin{aligned}
E_{b\_c} &= \text{rec}(b,c) = b/(c/a/b \cup c/d/b)^*/c, \\
E_{a\_b} &= \text{rec}(a,b) = a/b/(c/a/b \cup c/d/b)^*, \\
E_{a\_c} &= \text{rec}(a,c) = a/b/(c/a/b \cup c/d/b)^*/c.
\end{aligned}
$$

For each $\text{rec}(A,B)$, Cycle-C uses one LFP, but Cycle-E uses two LFP's. Since the last three XPATH queries cannot be handled by SQLGen-R, we tested SQLGen-R by generating a *with...recursive* query for each $\text{rec}(A,B)$ in our translation framework. The DTD has 4 nodes and 5 edges, and SQLGen-R produced a *with...recursive* query using 5 joins and 5 unions, which are computed in *each* iteration.

We used an XML tree with a fixed size of 120,000 elements. The same queries were evaluated over different shapes of XML trees controlled by the height of the tree (X_L) and the width of the tree (X_R). Since an XML tree with different heights and/or widths results in relations of different sizes in a database, even though the database size is fixed, the same SQL query generated may end up having different query-processing costs. We report elapsed time (seconds) for each query in Fig. 10: one figure shows the elapsed time while varying X_L from 8 to 20 with $X_R = 4$, and the other shows the time while varying X_R from 4 to

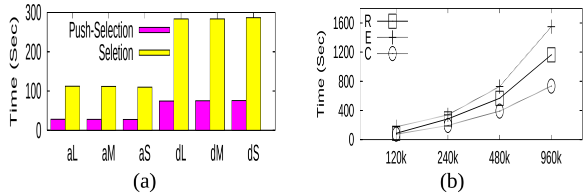


Figure 11: Pushing Selection (a) and Scalabity (b) (In (a), $X_R = 8$ and $X_L = 12$, and in (b) $X_R = 4$ and $X_L = 16$)

10 with $X_L = 12$. In all the cases, Cycle-C noticeably outperforms SQLGen-R and Cycle-E.

6.2 Exp-2: Pushing Selections into LFP

We tested two XPATH queries with selection conditions: $Q_e = a[id = A_i]/b//c/d$, $Q_f = a/b//c/d[id = D_i]$. For each query we generated two SQL queries, one with selections pushed into LFP and the other without. We evaluated these queries using datasets of the DTD of Fig. 9 (a), fixing the size of the datasets while varying the size of the set selected by the qualifiers of a_i and d_i. Figure 11 (a) shows the result, in which (1) aL, aM and aS indicate that an a_i element has large/medium/small number of d descendants; and (2) dL, dM and dS indicate that a d_i element has large/medium/small number of a ancestors, respectively. It shows that performance improvement by pushing selections into the LFP operator is significant.

6.3 Exp-3: Scalability Test

Figure 11 (b) demonstrates the scalability of our algorithms by increasing the dataset sizes, for an XPATH query *a//d* over the cross-cycle DTD (Fig. 9 (a)). The XML dataset size increases to 960,000 elements from 120,000. We set $X_L = 16$ because the default $X_L = 12$ is not large enough for the XML generator to produce such large datasets. We find that Cycle-C outperforms both SQLGen-R and Cycle-E noticeably, and SQLGen-R outperforms Cycle-E. When the dataset size is 960,000, the costs of Cycle-E and SQLGen-R are 2.1 times and 1.58 times of the cost of Cycle-C, respectively. This shows that when dataset is large, our optimization technique (Cycle-C) outperforms SQLGen-R by reducing the use of LFP operators and unnecessary joins and unions. Moreover, Cycle-C is linearly scalable.

6.4 Exp-4: Complex Cycles from Real-Life DTD

We next evaluate XPATH queries on the extracted 4-cycle BIOML DTD. We considered four subgraphs, as shown in Fig. 12, of the BIOML DTD of Fig. 9 (b) in order to demonstrate the impact of different DTDs on the translated SQL queries. Similar XPATH queries were tested on top of these extracted DTDs, and are summarized in Table 2.

All these XPATH queries were run on the same dataset which was generated using the largest 4-cycle DTD graph extracted from BIOML (Fig. 9 (b)) with $X_R = 6$ and $X_L = 16$. Unlike Exp-1, we did not trim the XML trees generated by the IBM XML Generator. The generated dataset consists of 1,990,858 elements, which is 16 times larger than the

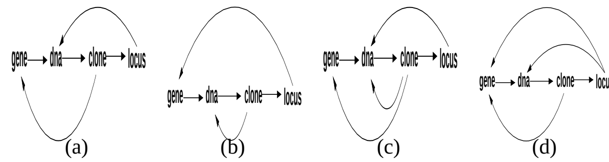


Figure 12: Different DTD graphs extracted from BIOML

Case	Query	n-Cycles	DTD Graph
2a	*gene//locus*	2	Fig. 12 (a)
2b	*gene//locus*	2	Fig. 12 (b)
2c	*gene//dna*	2	Fig. 12 (b)
3a	*gene//locus*	3	Fig. 12 (c)
3b	*gene//locus*	3	Fig. 12 (d)
4a	*gene//locus*	4	Fig. 9 (b)
4b	*gene//dna*	4	Fig. 9 (b)

Table 2: XPATH queries over different DTDs from BIOML

dataset (120,000 elements) used in Exp-1. The sizes of relations for *gene*, *dna*, *clone* and *locus* are 354,289, 703,249, 697,060 and 236,260, respectively.

As shown in Fig. 13, Cycle-C significantly outperforms SQLGen-R and Cycle-E in all the cases, and except case *2a*, Cycle-E outperforms SQLGen-R. In case *4a*, for example, SQLGen-R needs 7 joins and 7 unions in each iteration; Cycle-E needs to process 6 join, 2 LFP and 3 union operators; and Cycle-C uses 5 join, 1 LFP and 4 union operators. Note that because the Cycle-E execution sequence is determined by Tarjan's algorithm [25], it is inflexible to change the order of execution. Cycle-C outperforms SQLGen-R and Cycle-E because it uses less join and LFP.

7 Related Work

There has been a host of work on querying XML using an RDBMS, over XML data stored in an RDBMS or XML views published from relations (e.g., [7, 9, 12, 13, 14, 17, 23]; see [15] for an excellent recent survey). However, with the exception of the recent work of [14], as observed by [15], no algorithm has been published for translating recursive XML queries over recursive DTDs for schema-based XML storage or XML views of relations. Closest to our work is [14], which proposed the first technique to rewrite (recursive) path queries over recursive DTDs to SQL with the SQL'99 recursion. We have remarked the differences between their approach and ours in Sections 1 and 3.

At least two approaches have been proposed to querying XML data stored in relations via DTD-based shredding. One approach is based on middleware and XML views, e.g., XPERANTO [23] and SilkRoute [9]. In a nutshell, it provides clients with an XML view of the relations representing the XML data; upon receiving an XML query against the view, it composes the query with the view, rewrites the composed query to a query in a (rich) intermediate language, and answers the query by using both the middleware and the underlying RDBMS. However, this approach is tempered by the following observations. First, it is nontrivial to define a (recursive) XML view of the relational data without loss of the original information. Second, it requires middleware support and incurs communication overhead between the middleware and the RDBMS. Third, as observed

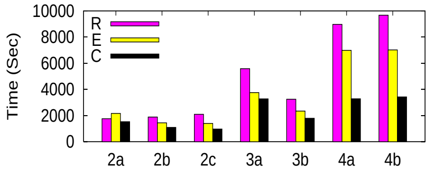


Figure 13: XPATH queries on the extracted BIOML DTDs

by [14], no algorithms have been developed for handling recursive queries over XML views with a recursive DTD.

Another approach is by rewriting XML queries into SQL (with recursion). Translations of XSLT [12], XQuery [7, 16] and (recursive) path queries [14] have been studied. While the algorithms of [12, 16] cannot handle query translation in the presence of recursive DTDs, their optimization techniques by leveraging, e.g., integrity constraints [16] and aggregation handling [12], are complementary to our work. These and techniques for query pruning [8], multi-query [22] and recursive-query optimization [1, 2, 3] can be used in our translation. Another interesting idea is to leverage XML region information: each node is stored with its position/offset [7, 17] (or path prefix [13]) in the document order such that the ancestor-descendant relation can be tested via range (or prefix) test. While recursion can be handled efficiently, this approach complicates update management due to the maintenance of regions, and may not apply to (virtual) XML views of relational data for the lack of region information in the source. Even for XML data stored in relations with usual compression for sharing common subtrees, the technique may not work well since the region of a node is unclear in a graph. In contrast, our rewriting technique does not have these limitations.

8 Conclusion

We have proposed a new approach to translating a practical class of XPATH queries over (recursive) DTDs to SQL queries with a simple LFP operator found in many commercial RDBMS. The novelty of the approach consists in efficient algorithms for rewriting an XPATH query over a recursive DTD into an equivalent regular XPATH query that captures both DTD recursion and XPATH recursion, and for translating a regular XPATH query to an equivalent sequence of SQL queries, as well as in new optimization techniques for minimizing the use of the LFP operator and for pushing selections into LFP. These provide the capability of answering important XPATH queries within the immediate reach of most commercial RDBMS.

We recognize that the LFP operator in produced SQL queries is not the only factor for efficiency, and we are developing a cost model to provide a better guidance for XML query rewriting. We are also exploring techniques for multi- and recursive-query optimization [1, 2, 3, 22]. Another topic for future work is to extend our algorithms to handle more complex XML queries, over (ordered) XML data stored in an RDBMS or XML views of relational data.

References

[1] R. Agrawal and P. Devanbu. Moving selections into linear least fixpoint queries. In *ICDE*, 1988.

[2] F. Bancilhon, D. Maier, Y. Sagiv, and J. Ullman. Magic sets and other strange ways to implement logic programs. In *PODS*, 1986.

[3] C. Beeri and R. Ramakrishnan. On the power of magic. *J. Log. Program*, 10, 1991.

[4] BIOML. BIOpolymer Markup Language. *http://xml.coverpages.org/BIOML-XML-DTD.txt.*

[5] B. Choi. What are real DTDs like. In *WebDB*, 2002.

[6] J. Clark and S. DeRose. XML path language (XPath). W3C Working Draft, Nov. 1999.

[7] D. DeHaan, D. Toman, M. Consens, and T. Ozsu. Comprehensive XQuery to SQL translation using dynamic interval encoding. In *SIGMOD*, 2003.

[8] M. Fernandez and D. Suciu. Optimizing regular path expression using graph schemas. In *ICDE*, 1998.

[9] M. F. Fernandez, A. Morishima, and D. Suciu. Efficient evaluation of XML middleware queries. In *SIGMOD*, 2001.

[10] D. Florescu and D. Kossmann. Storing and querying XML data using an RDMBS. *IEEE Data Eng. Bull*, 22(3), 1999.

[11] IBM. DB2 XML Extender. *http://www-3.ibm.com/software/data/db2/extended/xmlext/index.html.*

[12] S. Jain, R. Mahajan, and D. Suciu. Translating XSLT programs to efficient SQL querie. In *WWW*, 2002.

[13] H. Kaplan, T. Milo, and R. Shabo. A comparison of labeling schemes for ancestor queries. In *SODA*, 2002.

[14] R. Krishnamurthy, V. Chakaravarthy, R. Kaushik, and J. Naughton. Recursive XML schemas, recursive XML queries, and relational storage: XML-to-SQL query translation. In *ICDE*, 2004.

[15] R. Krishnamurthy, R. Kaushik, and J. Naughton. XML-SQL query translation literature: The state of the art and open problems. In *Xsym*, 2003.

[16] R. Krishnamurthy, R. Kaushik, and J. Naughton. Efficient XML-to-SQL query translation: Where to add the intelligence. In *VLDB*, 2004.

[17] Q. Li and B. Moon. Indexing and querying xml data for regular path expressions. In *VLDB*, 2001.

[18] M. Marx. XPath with conditional axis relations. In *EDBT*, 2004.

[19] Microsoft. SQLXML and XML mapping technologies. *http://msdn.microsoft.com/sqlxml/default.asp.*

[20] M. Nunn. An overview of SQL server 2005 for the database developer, 2004. *http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dnsql90/html/sql_ovyukondev.asp.*

[21] Oracle. Oracle9i XML Database Developer's Guide – Oracle XML DB Release 2. *http://otn.oracle.com/tech/xmldb/content.html.*

[22] P. Roy, S. Seshadri, S. Sudarshan, and S. Bhobe. Efficient algorithms for multi query optimization. In *SIGMOD*, 2000.

[23] J. Shanmugasundaram, J. Kiernan, E. J. Shekita, C. Fan, and J. Funderburk. Querying XML views of relational data. In *VLDB*, 2001.

[24] J. Shanmugasundaram, K. Tufte, G. He, C. Zhang, D. DeWitt, and J. Naughton. Relational databases for querying XML documents: Limitations and opportunities. In *VLDB*, 1999.

[25] R. E. Tarjan. Fast algorithms for solving path problems. *JACM*, 28(3):594–614, 1981.

[26] H. Weinblatt. A new search algorithm for finding the simple cycles of a finite directed graph. *JACM*, 19(1):43–56, 1972.