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

**Abstract**

We study the problem of evaluating XPATH queries over XML data that is stored in an RDBMS via schema-based shredding. The interaction between recursion in XPATH queries and recursion in DTDs makes it challenging to answer XPATH queries using RDBMS. We present a new approach to translating XPATH queries into SQL queries based on a notion of *regular* XPATH *expressions* and a simple least fixpoint (LFP) operator. Regular XPATH expressions are a mild extension of XPATH, and the LFP operator takes a single input relation and is already supported by most commercial RDBMS. We show that regular XPATH expressions are capable of capturing both DTD recursion and XPATH queries in a uniform framework. Furthermore, they can be translated into an equivalent sequence of SQL queries with the LFP operator. We present algorithms for rewriting XPATH queries into regular XPATH expressions and for translating regular XPATH expressions to SQL queries. We also provide 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, as well as its flexibility to accommodate existing relational query optimization techniques. 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 [34] as found in many commercial products (e.g., [17, 26, 29]). 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 $\tau_d$, 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 $D$, $Q$ over $T$ can be answered by $Q'$ over the instance $\tau_d(T)$ of $\mathcal{R}$ that represents $T$, i.e., $Q(T) = Q'(\tau_d(T))$. Here we allow DTDs $D$ to be recursive and consider queries $Q$ in XPATH [9], which is essential for other XML query languages such as XQuery and XSLT.

The query translation problem is, however, nontrivial: DTDs (or XML Schema) found in practice are often recursive [7] and complex. This is particularly evident in real-life applications (see, e.g., BIOML [6] and GedML [15], which, when represented as graphs, contains a number of nested and overlapping cycles). 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 [18] is based on enumerating all matching paths of the input XPATH query in a DTD, sharing a single representation of common sub-paths, rewriting these 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 when dealing with descendants '//' in XPATH. Another approach is by means of a rich intermediate language and middleware as proposed in [32]: 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 [22], 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 and push the work (SQL queries) to the underlying RDBMS, capitalizing on the RDBMS to evaluate and optimize the queries. This, however, calls for an extension of SQL to support certain recursive operator. As observed by [22], although there has been a host of work on storing and querying XML using an RDBMS [10, 14, 18, 21, 25, 32, 33], 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 [22].

Recently an elegant approach was proposed in [21] to translating path queries to SQL with the linear-recursion construct *with...recursive* of SQL'99. The algorithm of [21] is capable of translating path queries with // and limited qualifiers to (a sequence of) SQL queries with the SQL'99 recursion operator. 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 [21] 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 [13]. Worse still, as the *with...recursive* operator is treated as a blackbox, the user can do little to optimize it. A third problem is that the class of path query handled by the algorithm of [21] is 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 a notion of *regular* XPATH *expressions* and a simple least fixpoint (LFP) operator. Our 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 required by 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* [28]). We show that regular XPATH expressions are capable of expressing a large class of XPATH queries over a (recursive) DTD $D$, by substituting the general Kleene closure $E^*$ for //, and by giving a finite representation of possibly infinite matching paths of an XPATH query in terms of $E^*$. 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. That is, low-end RDBMS features (SQL with $\Phi(R)$) suffice to support complex XPATH queries.

Taken together, our approach works as follows. Given an XPATH query $Q$, we first rewrite $Q$ into a regular XPATH expression $E_Q$, 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 $D$. To this end we provide an efficient algorithm for translating an XPATH query over a (recursive) DTD $D$ to an equivalent regular XPATH expression, and a novel algorithm for rewriting a regular XPATH expression into a sequence of SQL queries with the LFP operator. Furthermore, we introduce optimization techniques to minimize the use of the LFP operator in the rewritten SQL queries.

**Contributions.** The main contributions include the following.

- A notion of regular XPATH expressions that captures DTD recursion and XPATH recursion in a uniform framework.
- The use of the simple LFP operator commonly found in commercial products to express a large class of XPATH queries.
- An efficient algorithm for translating XPATH queries into regular XPATH expressions, based on dynamic programming.
- A novel algorithm for rewriting a regular XPATH expression to a sequence of SQL queries with the LFP operator.
- New optimization techniques for minimizing the use of the LFP operator in SQL translation of XPATH queries.
- Experimental results verifying the effectiveness of our approach and techniques, using real-life XML DTDs.

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 most RDBMS platforms. Furthermore, it can easily accommodate optimization techniques developed for SQL queries, e.g., multi-query [30] and recursive SQL query optimization [31]. (3) It is capable of handling a class of XPATH queries supporting child, descendants and union as well as rich qualifiers with data values, conjunction, disjunction and negation, which are beyond those studied in earlier proposals. These thus yield an effective and efficient method that works with most RDBMS products, to answer a large class of XPATH queries found in practice. This work is also a concrete step toward answering XPATH queries over (virtual) recursive XML views of relational data.

**Organization.** Section 2 reviews DTDs, XPATH and schema-based mapping from XML to relations. Section 3 outlines our query translation approach as opposed to the one given in [21]. Section 4 provides an algorithm for translating XPATH queries to regular XPATH expressions, followed by an algorithm for rewriting regular XPATH expressions into SQL with a simple LFP operator in Section 5. Experimental results are presented in Section 6, followed by related work in Section 7. Finally, Section 8 concludes the paper.

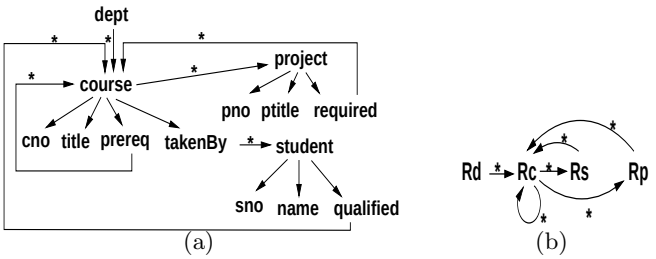


Figure 1: A graph representation of the `dept` DTD.

## 2 DTD, XPath, and Schema-Based Shredding

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

### 2.1 DTDs

Without loss of generality, we represent a DTD $D$ as an extended context-free grammar of the form $(Ele, Rg, r)$, where $Ele$ is a finite set of element types; $r$ is a distinguished type, called the root type; and $Rg$ defines the element types: for any $A$ in $Ele$, $Rg(A)$ is a regular expression of the following form:

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

where $\epsilon$ is the empty word, $B$ is a type in $Ele$ (referred to as a *subelement* 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.

Along the same lines as [34], 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 represents the parent/child relationship. More 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 $\alpha$; the edge is labeled with '$*$' if $B$ is enclosed in $\alpha_0^*$ for some sub-expression $\alpha_0$ of $\alpha$. This simple graph representation of 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 $D$ is *recursive* if it has an element type that is defined (directly or indirectly) in terms of itself. Note that the DTD graph $G_D$ of $D$ is *cyclic* if $D$ is recursive. A DTD graph $G_D$ is called a $n$-cycle graph if $G_D$ consists of $n$ simple cycles. Here, a simple cycle refers to a cycle in which no node appears more than once.

**Example 2.1:** We consider a `dept` DTD as our running example.

```
<!ELEMENT dept course*>
<!ELEMENT course (cno, title, prereq, takenBy, project)>
<!ELEMENT prereq course*>
<!ELEMENT student (sno, name, qualified)>
<!ELEMENT qualified course*>
<!ELEMENT project (pno, ptitle, required)>
```

<!ELEMENT required course*>

A dept has a list of course elements. A course consists of a cno (course code), a title, a prerequisite hierarchy (via prereq), and all the students who have registered for the course (via takenBy). A student has a sno (student number), a name and a list of qualified courses. A course may have several projects. Each project has a pno (project number), a ptitle (title) and required knowledge of other courses (required). Its DTD graph, a 3-cycle graph, is shown in Fig. 1 (a). □

## 2.2 XPath Queries

We consider a class of XPATH queries [9] that supports recursion (descendants) and rich qualifiers, given as follows.

$$p \quad ::= \quad \epsilon \quad | \quad A \quad | \quad * \quad | \quad p/p \quad | \quad //p \quad | \quad p \cup p \quad | \quad p[q]$$

where $\epsilon$, $A$ and $*$ denote the empty path, a label and a wildcard, respectively; '$\cup$', '/' and '//' are *union*, *child-axis* and *descendant-or-self-axis*, respectively; and $q$ is called a *qualifier*, defined as

$$q \quad ::= \quad p \quad | \quad text() = c \quad | \quad \neg q \quad | \quad q \wedge q \quad | \quad q \vee q$$

where $c$ 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 $v[\![p]\!]$. Qualifiers are interpreted as follows: at a context node $v$, the atomic predicate $[p]$ holds iff $v[\![p]\!]$ is nonempty, i.e., there exists a node reachable via $p$ from $v$; and $[text() = c]$ is true iff $v.val$ equals the constant $c$. The boolean operations are self-explanatory. 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 $\epsilon$.

Note that this class of XPATH queries properly contains branching path queries studied in [21] and tree pattern queries. In the sequel, we refer to this class of queries simply as XPATH queries.

**Example 2.2:** Consider two XPATH queries.

$Q_1$ = dept//project
$Q_2$ = dept/course[//prereq/course/cno="cs66" ∧ ¬//project
∧ ¬takenBy/student/qualified//course/cno = "cs66"]

Over an XML tree of the dept DTD of Fig.1, the first XPATH query is to find all course-related 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 next review shredding of XML data into relations. We focus on a DTD-based approach since it is supported by most RDBMS [17, 26, 29], rather than schema-oblivious XML storage methods.

| 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

We adopt the shared-inlining technique of [34]. In a nutshell, the inlining algorithm partitions a DTD graph $G_D$ into subgraphs, $G_1, G_2, \cdots$ such that any A-node is represented in exactly one subgraph and there is no edge labeled '$*$' in any subgraph. Each subgraph $G_i$ is mapped to a relation schema $R_i$. Each relation schema has a key attribute ID. The edges from a subgraph $G_i$ to a subgraph $G_j$ are specified using *parentId* in the corresponding relation schema $R_j$. If a subgraph $G_j$ has more than one incoming edges, say from $G_i$ and $G_k$, a *parentCode* attribute is introduced into the relation schema $R_j$ indicating the parent code of the $R_j$ tuples.

We use $\tau : D \rightarrow \mathcal{R}$ to denote a mapping from DTD $D$ to a relational database schema $\mathcal{R}$, which consists of a set of relation schemas. Observe that from $\tau$ one can easily derive a *data mapping*, denoted by $\tau_d$, from XML trees of $D$ to instances of $\mathcal{R}$.

To simplify the discussion we assume that the mapping $\tau$ maps each element 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 $\tau_d(\mathcal{T})$ representing an XML tree $\mathcal{T}$, each $R_A$ tuple $(f, t, v)$ represents an edge in $\mathcal{T}$ 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 $\mathcal{T}$ and are thus *unique* in the database, and $v$ is '_' in the absence of text value at $t$. In particular, $f =$ '_' if and only if $f$ is the root of $\mathcal{T}$. 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 graph $G_D$ of Fig. 1 (a) is partitioned into four subgraphs rooted at *dept*, *course*, *project*, and *student*, respectively (see Fig. 1 (b)). It is mapped to a relational database schema $\tau(D)$ consisting of four corresponding relation schemas, $R_d$, $R_c$, $R_p$ and $R_s$:

$R_d$`(F, T)`
$R_c$`(F, T, cno, title, prereq, takenBy, parentCode)`
$R_s$`(F, T, sno, name, qualified)`
$R_p$`(F, T, pno, ptitle, required)`

A sample database is shown in Table 1, which only shows $F$ and $T$ attributes. From Table 1 one can find paths in the XML tree of the `dept` DTD, e.g., $d_1.c_1.c_2.c_3$ and $d_1.c_1.c_2.p_1.c_4.p_2$. □

## 3 Overview: From XPath to SQL

The query translation problem from XPATH to SQL is stated as follows. Let $\tau : D \rightarrow \mathcal{R}$ be a mapping from a DTD $D$ to a relational schema $\mathcal{R}$, and $\tau_d$ be the corresponding data mapping from XML trees of $D$ to the relational instance of $\mathcal{R}$. The problem 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 [21], the only solution published so far for the query translation problem in the presence of recursive DTDs. To overcome its limitations, we then propose a new approach and outline it in this section; detailed algorithms are provided in the next two sections.

### 3.1 Linear Recursion of SQL'99

The algorithm of [21], referred to as SQLGen-R, handles recursive path queries over recursive DTDs based on the SQL'99 recursion operator. 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 $c_i$ in the topological order, 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, the SQL query $Q_i$ is defined in terms of the *with...recursive* operator. More specifically, it generates two parts from $c_i$: an *initialization* part and a *recursive* part. 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. It should be noted that if $c_i$ has $k$ edges, the query $Q_i$ actually calls for a fixpoint operator $\phi(R, R_1, R_2, \cdots R_k)$ with $k+1$ input relations, defined as follows:

$$\begin{aligned} R^0 &\leftarrow R \\ R^i &\leftarrow R^{i-1} \cup (R^{i-1} \bowtie R_1) \cup \cdots \cup (R^{i-1} \bowtie R_k) \end{aligned} \tag{1}$$

where $R^0$ corresponds to the initialization part, and $R_j$ corresponds to an SQL query coding an edge in $c_i$ 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. More specifically, the initial part of the recursion is given in lines 3-4, while the recursion part is lines 6-19. Each edge in the graph Fig.1 (b) is translated into a select statement. Observe that in the select statement, it uses *Rid* to keep track of where the tuples in the result relation $R$ come from. For example, the select statement for the edge $R_c \rightarrow R_c$ (lines 6-7) inserts a tuple into the result relation $R$ with its $F$ and $T$ values in addition to a *Rid* value '$c$' indicating that it is from relation $R_c$. The usage of *Rid* is to join right parent/child tuples. As line 10 shows, in the select statement for the edge $R_c \rightarrow R_s$, it needs to join with tuples in $R$ that is originally from $R_c$ ($Rid$ = 'c'). Similarly for $R_s \rightarrow R_c$, $R_c \rightarrow R_p$, and $R_p \rightarrow R_c$ (lines 12-13, 15-16 and 18-19, respectively). When evaluated over the relational database of Table 1, the query of Fig. 2 returns the result shown in Table 2. Using a selection on $Rid$ = '$p$' on Table 2, one can find that $p_1$ and $p_2$ are the descendants of $p$. □

Observe the following about the query of Fig. 2. First, it actually requires a fixpoint operator that takes 4 relations as input. As we have remarked in Section 1, while most commercial RDBMS

```
1.  with
2.  R (F, T, Rid) as (
3.     (select R_c.F, R_c.T, Rid('c') from R_d, R_c)
4.     where R_c.T = R_d.F
5.     union all
6.     (select R.F, R_c.T, Rid('c')
7.           from R, R_c where R.T = R_c.F and Rid = 'c')
8.     union all
9.     (select R.F, R_s.T, Rid('s')
10.          from R, R_s where R.T = R_s.F and Rid = 'c')
11.    union all
12.    (select R.F, R_c.T, Rid('c')
13.          from R, R_c where R.T = R_c.F and Rid = 's')
14.    union all
15.    (select R.F, R_p.T, Rid('p')
16.          from R, R_p where R.T = R_p.F and Rid = 'c')
17.    union all
18.    (select R.F, R_c.T, Rid('c')
19.          from R, R_c where R.T = R_c.F and Rid = 'p'))
```

Figure 2: The SQL statement generated by SQLGen-R

| iteration | $F$ | $T$ | $Rid$ |
|---|---|---|---|
| 0 | $d_1$ | $c_1$ | 'c' |
| 1 | $c_1$ | $c_2$ | 'c' |
| | $c_1$ | $s_1$ | 's' |
| | $c_1$ | $s_2$ | 's' |
| 2 | $c_2$ | $c_3$ | 'c' |
| | $c_2$ | $p_1$ | 'p' |
| | $s_2$ | $c_5$ | 'c' |
| 3 | $p_1$ | $c_4$ | 'c' |
| 4 | $c_4$ | $p_2$ | 'p' |

Table 2: An output of SQLGen-R at each iteration.

support a LFP $\Phi(R)$ that takes a single input relation, 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 consisting of 5 joins and 5 unions. That is, each iteration of the fixpoint computation needs to compute 5 joins and 5 unions. Third, *with...recursive* is treated as a black box. In this example, all 5 relations join the result relation $R$ in the center, which forms a *star* shape. The relation in the center keeps growing, but one can do little to optimize the operations inside the *with...recursion* expression.

### 3.2 A New Approach

To overcome the limitations of the previous approach, we propose a new approach to translating XPATH queries to SQL, based on a notion of extended XPATH expressions and the simple LFP operator $\Phi(R)$. Below we first define regular XPATH expressions and review the simple LFP operator. We then outline our approach.

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

$$\begin{array}{lcl} E & ::= & \epsilon \mid A \mid E/E \mid E \cup E \mid E^* \mid E[q], \\ q & ::= & E \mid text() = c \mid \neg q \mid q \wedge q \mid q \vee q. \end{array}$$

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

Regular XPATH expressions are the "least upper bound" of XPATH and regular expressions. They differ from XPATH queries in that, first, they support general Kleene closure $E^*$ as opposed to restricted recursion '//', and second, they do not allow wildcard $*$ and descendant '//'. They extend regular expressions by supporting qualifiers. The motivation for using $E^*$ instead of '//' is twofold. First, the expressive power of $E^*$ is required for encoding both DTD recursion and XPATH recursion. As will be seen shortly, with $E^*$ one can define a finite representation of (possibly infinite) matching paths of an XPATH query over a recursive DTD. Second, $E^*$ "instantiates" // with paths in the DTD. In a nutshell, $E$ takes a union of all matching simple cycles of // and $E^*$ then applies the Kleene closure to the union; each of these paths can then be mapped to a sequence of relations connected with joins. Furthermore, $E^*$ introduces more opportunities for optimizing a query than //.

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

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

This LFP operator is already supported by most commercial products. For example, the implementations of $\Phi(R)$ in Oracle and IBM DB2 are shown in Fig. 3.

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

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

Here, the projected attributes are taken from the attributes $F$ (from) and $T$ (to) in relations $R_1$ and $R_n$, 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 least fixpoint operator $\phi$ (Eq. (1)) can take an unbounded number $k$ of input relations. One might be tempted to think that Eq. (1) can be coded with Eq. (2), as follows:

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

LFP $\Phi(R)$ **in Oracle**
**select** $F$, $T$ **from** $R$ **connect by** $F =$ **prior** $T$

LFP $\Phi(R)$ **in DB2**
1. **with**
2. $R_\Phi(F,T)$ **as** (
3. (**select** $F$, $T$ **from** $R$)
4. **union all**
5. (**select** $R_\Phi.F$, $R.T$ **from** $R_\Phi, R$ **where** $R_\Phi.T = R.F$))

Figure 3: Implementation of LFP in Oracle and DB2

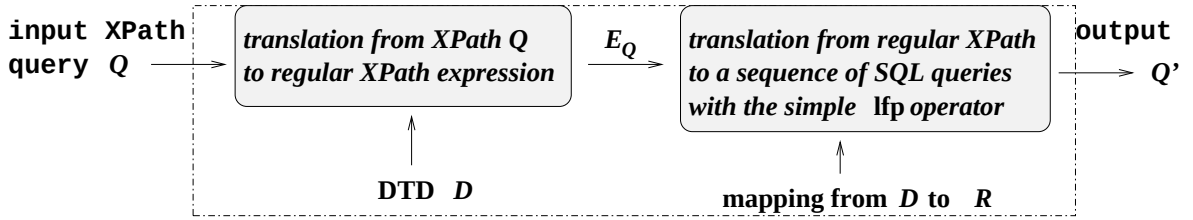



Figure 4: Translation from XPath to SQL

where $R' = \cup_{j=1}^{k} R_j$. This is unfortunately incorrect since different conditions are associated with different joins in Eq. (1). Indeed, observe that different *select* statements (joins) in the query of Fig. 2 have different conditions. As a result, taking a big union of the relations in the *from* clauses of these *select* statements may either lead to incorrect answer, or yield a horrendously large query and relation (when outer union is used instead of union).

**A new approach for query translation.** Based on the LFP operator $\Phi(R)$ and regular XPATH expressions, 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 expression $E_Q$ over $D$. Second, it rewrites $E_Q$ into an equivalent sequence $Q'$ of SQL queries based on a mapping $\tau : D \to \mathcal{R}$, and using the LFP operator to handle Kleene closure. Indeed, in Section 4 we present a translation algorithm to show that every XPATH query $Q$ over a (recursive) DTD $D$ can be rewritten to an equivalent regular XPATH expression $E_Q$ over $D$. Then, we provide another algorithm in Section 5 to show that the simple LFP operator $\Phi(R)$ suffices to handle general Kleene closure in a regular XPATH expression $E_Q$. Furthermore, the regular XPATH expression $E_Q$ and SQL queries $Q'$ are both 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:** Let us 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 translation algorithms first translate $Q_1$ to a regular XPATH query $E_{Q_1} = R_d/R_c/E^*/R_p$, where $E = (R_c \cup R_s/R_c \cup R_p/R_c)$. It then rewrites $E_{Q_1}$ to a sequence of SQL queries (written in relational algebra):

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

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. In other words, we pull join/union out from the black box of *with...recursive*. This not only gives us more opportunities to optimize join/union, but also allows us to push selection conditions into the fixpoint operator, along the same lines as the LFP optimization by distribution of selections suggested by [2].

## 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

**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) $\epsilon$: $\mathsf{x2r}(p, A) := \epsilon$; $\mathsf{reach}(p, A) := \{A\}$;
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) $p_1 \cup p_2$: $\mathsf{x2r}(p, A) := \mathsf{x2r}(p_1, A) \cup \mathsf{x2r}(p_2, A)$; $\mathsf{reach}(p, A) := \mathsf{reach}(p_1, A) \cup \mathsf{reach}(p_2, A)$;
35. (7) $p'[q]$:
36. for each $B$ in $\mathsf{reach}(p', A)$ do
37. if $\mathsf{x2r}([q], B) = [\epsilon]$ /* $[q]$ holds at $B$ */
38. then $\mathsf{x2r}(p, A) := \mathsf{x2r}(p, A) \cup \mathsf{x2r}(p', A)$; $\mathsf{reach}(p, A) := \mathsf{reach}(p, A) \cup \{B\}$;
39. else if $\mathsf{reach}([q], B) \neq \emptyset$ /* $[q]$ is not false at $B$ */
40. then $\mathsf{x2r}(p, A) := \mathsf{x2r}(p, A) \cup \mathsf{x2r}(p', A)[\mathsf{x2r}(q, B)]$; $\mathsf{reach}(p, A) := \mathsf{reach}(p, A) \cup \{B\}$;
41. (8) $[p_1]$: $\mathsf{x2r}(p, A) := [\mathsf{x2r}(p_1, A)]$; $\mathsf{reach}(p, A) := \mathsf{reach}(p_1, A)$;
42. (9) $p'[text() = c]$: $\mathsf{x2r}(p, A) := \mathsf{x2r}(p', A)[text() = c]$; $\mathsf{reach}(p, A) := \mathsf{reach}(p', A)$;
43. (10) $[q_1 \wedge q_2]$: if $\mathsf{reach}(q_1, A) \neq \emptyset$ and $\mathsf{reach}(q_2, A) \neq \emptyset$
44. then $\mathsf{x2r}(p, A) := [\mathsf{x2r}([q_1], A) \wedge \mathsf{x2r}([q_2], A)]$; $\mathsf{reach}(p, A) := \{true\}$;
45. else $\mathsf{x2r}(p, A) := \emptyset$; $\mathsf{reach}(p, A) := \emptyset$;
46. (11) $[q_1 \vee q_2]$: if $\mathsf{reach}(q_1, A) \neq \emptyset$ and $\mathsf{reach}(q_2, A) \neq \emptyset$
47. then $\mathsf{x2r}(p, A) := [\mathsf{x2r}([q_1], A) \vee \mathsf{x2r}([q_2], A)]$;
48. else if $\mathsf{reach}(q_1, A) \neq \emptyset$ and $\mathsf{reach}(q_2, A) = \emptyset$
49. then $\mathsf{x2r}(p, A) := [\mathsf{x2r}([p_1], A)]$;
50. else if $\mathsf{reach}(q_1, A) = \emptyset$ and $\mathsf{reach}(q_2, A) \neq \emptyset$
51. then $\mathsf{x2r}(p, A) := [\mathsf{x2r}([p_2], A)]$;
52. else $\mathsf{x2r}(p, A) := \emptyset$;
53. $\mathsf{reach}(p, A) := \mathsf{reach}(q_1, A) \cup \mathsf{reach}(q_2, A)$;
54. (12) $p'[\neg q]$: if $\mathsf{reach}(q, B) = \emptyset$ for all $B \in \mathsf{reach}(p', A)$
55. then $\mathsf{x2r}(p, A) := \mathsf{x2r}(p', A)$; $\mathsf{reach}(p, A) := \{true\}$;
56. else $\mathsf{x2r}(p, A) := \mathsf{x2r}(p', A)[\neg \mathsf{x2r}([q], A)]$; $\mathsf{reach}(p, A) := \mathsf{reach}(p', A)$;
57. optimize $\mathsf{x2r}(Q, r)$ by removing $\emptyset$ using $\emptyset \cup E = E$, $E_1/\emptyset/E_2 = \emptyset$
58. return $\mathsf{x2r}(Q, r)$; /* $r$ is the root of $D$ */

Figure 5: Rewriting algorithm from XPath to regular XPath

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, one can compute these by using, e.g., Tarjan's fast algorithm [35], 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:

1. for each $A$ in $N$
2. for each descendant $B$ of $A$ do
3. $\mathsf{rec}(A, B)$ := the regular expression found by the algorithm of [35];
4. $\mathsf{reach}(\epsilon//, A) := \mathsf{reach}(\epsilon//, A) \cup \{B\}$;

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 7–56), 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 $\mathsf{reach}\ (p, A)$); 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 ($\epsilon$ 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 57–58).

**Example 4.1:** Recall the XPATH query $Q_2$ from Example 2.2. Observe that the algorithm of [21] *cannot* handle this query over the *dept* DTD of Fig. 1 (a). In contrast, XPathToReg translates $Q_2$ to the regular XPATH query

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

where the following is computed by Tarjan's algorithm:

$$\begin{aligned} E_{\mathit{course\_course}} &= \mathsf{rec}\ (\mathit{course}, \mathit{course}) = \mathit{course}/E_1^* \cup E_2^+/E_1^*, \\ E_{\mathit{course\_project}} &= \mathsf{rec}\ (\mathit{course}, \mathit{project}) \\ &= (\mathit{course}/E_1^* \cup E_2^+/\mathit{course}/E_1^*)/\mathit{project}, \end{aligned}$$

$$E_{qualified\_course} = \mathsf{rec}\ (qualified,\ course)$$
$$= qualified/course/E_1^* \cup (qualified/E_2)^+/course/E_1^*,$$

$$E_1 = prereq/course \cup takenBy/student/qualified/course$$
$$E_2 = course/E_1^*/project/required$$

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

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 [35] 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 $O(|Q| * |D|^4 log|D|)$.* □

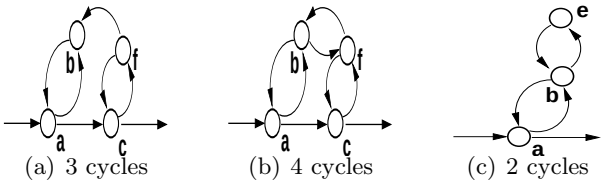
Observe the following. First, regular XPATH queries capture DTD *recursion and* XPATH *recursion* in a uniform framework by means of the general Kleene closure $E^*$. Second, during the translation, algorithm XPathToReg conducts *optimization leveraging the structure of the* DTD. Third, Kleene closure 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 [35] 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 [35] 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 initial value $A_1/\ldots/A_k$. Then, for each simple cycle $C_j$ "connected" to $A_i$, the



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

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

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 $\mathsf{rec}(A, B)$ is indeed $(E_1 \cup \ldots \cup E_n)$. Note that all simple cycles of a directed graph can be efficiently identified [36].

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* $L = A_1 \rightarrow \ldots \rightarrow A_k$ 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 $C = A_i \rightarrow A'_1 \rightarrow \ldots \rightarrow A'_m \rightarrow A_i$. 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 $E_\gamma$ 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 $E_\gamma$ 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 $E_\gamma$ 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, $R_c \rightarrow R_c$, $R_c \rightarrow R_s \rightarrow R_c$ and $R_c \rightarrow R_p \rightarrow R_c$. The only *AB-path* is $L = R_d \rightarrow R_c \rightarrow R_p$ (i.e, *dept* $\rightarrow$ *course* $\rightarrow$ *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 $C_1 = a \rightarrow b \rightarrow a$, $C_2 = c \rightarrow f \rightarrow c$, and $C_3 = a \rightarrow c \rightarrow f \rightarrow b \rightarrow a$. Consider $\mathsf{rec}(a, c)$, for which the only *AB-path* is $L = a \rightarrow c$. 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 $a/E_{\gamma_1}$, capturing paths from $a$ to $a$ via $C_1$ and $C_3$. It then contracts $C_2$ and $C_3$ by replacing $c$ with $c/E_{\gamma_2}$, 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, $E_{\gamma_2}$ covers all possible paths that traverse $E_{\gamma_1}$ since $E_{\gamma_2}$ includes $E_{\gamma_1}$ by replacing $a$ with $E_{\gamma_1}$, and $E$ covers all possible paths between $a$ and $c$. Second, the processing order of the cycles is

**Algorithm Cycle-C($G_D$, $A$, $B$)**
*Input*: a DTD graph $G_D$ and two nodes $A$, $B$ in $G_D$.
*output*: $\mathsf{rec}(A, B)$ in $G_D$.

1. find all distinctive *AB-path*s, $L_1, L_2, \cdots, L_k$, between $A$ and $B$;
2. for each $L_i$ do
3. $G_i$ := the subgraph including all simple cycles that are connected $L_i$ directly and indirectly;
4. for each $L_i = A_1 \rightarrow \ldots \rightarrow A_k$ do
5. $E_i := A_1/\ldots/A_k$;
6. $\mathcal{C}_i$ := a list of all simple cycles in $G_i$ found by the algorithm of [36] and sorted in topological order based on their distance to $L_i$ from the farthest to those directly connected to $L_i$;
7. for each cycle $C$ in $\mathcal{C}_i$ in the order of $\mathcal{C}_i$ do
8. if $C$ does not directly connect to $L_i$
9. then find node $A_x$ on $C$ with the shortest distance to $L_i$;
10. $G_x$ := the subgraph consisting of $C$;
11. $E_C$ := Cycle-C($G_x$, $A_x$, $A_x$); /* contract $C$ to $A_x$ */
12. replace $A_x$ and $C$ with $E_C^*$ in $G_i$;
13. identify the nodes $A'_1, \cdots, A'_m$ shared by simple cycles with $L_i$;
14. for each $A'_i$ shared by cycles $C_1, \ldots, C_l$
15. $E_{A_j}$ := a regular expression representing $C_1, \ldots, C_l$, computed based on cases 1–3 described earlier;
16. replace $A_j$ in $E_i$ with $A_j/E^*_{A'_j}$;
17. return $E = E_1 \cup \cdots \cup E_n$;

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

not sensitive. We can also first process $C_2$ and $C_3$ and obtain $E_{\gamma_2}$, and then let $E_{\gamma_1}$ include $E_{\gamma_2}$ by replacing $c$ with $E_{\gamma_2}$.

**Case-4.** There are multiple *AB-path*s. Figure 6 (b) shows a DTD graph with 4 simple cycles $C_1 = a \rightarrow b \rightarrow a$, $C_2 = c \rightarrow f \rightarrow c$, $C_3 = a \rightarrow c \rightarrow f \rightarrow b \rightarrow a$, and $C_4 = b \rightarrow f \rightarrow b$. Consider $\mathsf{rec}(a, c)$, which has two *AB-path*s: $L_1 = a \rightarrow c$ and $L_2 = a \rightarrow b \rightarrow f \rightarrow c$. 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 $C_1 = a \rightarrow b \rightarrow a$ and $C_2 = b \rightarrow e \rightarrow b$. Consider $\mathsf{rec}(a, a)$, 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 $\mathsf{rec}(A, B)$ as output. More specifically, it first identifies all the *AB-path*s $L_1, \ldots, L_n$ 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 $L_i$

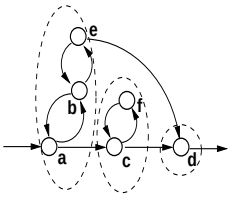


Figure 8: A divide-and-conquer example

directly or indirectly (lines 1–2). For each $L_i$, it finds all the simple cycles $\mathcal{C}_i$ using the algorithm of [36]. 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 the following. **Theorem 4.2:** *Given a* DTD *graph of $G_D$ and nodes $A$ and $B$ in $G_D$,* Cycle-C *correctly computes a regular expression* $\mathsf{rec}(A, B)$ *that 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

$E_{course\_course} = course/E_{cc},$
$E_{course\_project} = course/E_{cc}/project,$
$E_{qualified\_course} = qualified/course/E_{cc},$

$E_{cc} = (E_1 \cup project/required/course)^*,$
$E_1$ *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. □

It is worth mentioning that while the number of the LFP operators in the produced SQL query $Q'$ is not the only factor for the performance of $Q'$, it is among the most important ones. The Cycle-C algorithm above shows only the basic ideas. It can be improved to contract a group of simple cycles that do not share a node with a *AB-path*, instead one by one. Furthermore, in order to reduce the computing cost of Cycle-C when dealing with large DTD graphs, a divide-and-conquer approach can be adopted. For example, consider the DTD graph shown in Fig. 8, which includes three subgraphs (strongly connected components), as indicated by dotted circles. Let $G_1$, $G_2$ and $G_3$ denote the left, middle and the right subgraphs, respectively. Consider $\mathsf{rec}(a, d)$, which yields two *AB-path* $a \to c \to d$ and $a \to b \to e \to d$. We divide the computation into computing $E_{\mathrm{a\_a}} = \mathsf{rec}(a, a)$ in $G_1$, $E_{\mathrm{a\_e}} = \mathsf{rec}(a, e)$ in $G_1$, $E_{\mathrm{c\_c}} = \mathsf{rec}(c, c)$ in $G_2$, and $E_{\mathrm{d\_d}} = \mathsf{rec}(d, d)$ in $G_3$, each is computed with a separate smaller subgraph. Then, $\mathsf{rec}(a, d)$ can be obtained by putting them together: $E_{\mathrm{a\_a}}/E_{\mathrm{c\_c}}/E_{\mathrm{d\_d}} \cup E_{\mathrm{a\_e}}/E_{\mathrm{d\_d}}$. Hence, instead of computing $\mathsf{rec}(a, d)$ with a large DTD graph, we can achieve the same by calling rec() three times, each dealing with a much smaller sub-graph.

## 5 From Regular XPath Expressions to SQL

In this section, we present an algorithm for rewriting regular XPATH expressions into SQL queries with the simple LFP operator.

### 5.1 Translation Algorithm

Consider a mapping $\tau : D \to \mathcal{R}$, where $D$ is a DTD and $\mathcal{R}$ is a relational schema, such that its associated data mapping $\tau_d$ shreds XML trees of $D$ into databases of $\mathcal{R}$. Given a regular XPATH expression $E_Q$ over $D$, we compute a sequence $Q'$ of equivalent relational queries with the simple LFP operator $\Phi$ such that for any XML tree $T$ of $D$, $E_Q(T) = Q'(\tau_d(T))$. We write $Q'$ in the relational algebra ($RA$), which can be easily coded in SQL.

A subtle issue is that the LFP operator $\Phi$ supports $(E)^+$ but not $(E)^*$ (where $(E)^*$ means repeating $E$ zero or more times, while $(E)^+$ indicates repeating $E$ at least once). Thus $(E)^*$ needs to be converted to $\epsilon \cup (E)^+$. To simplify the handling of $\epsilon$, 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 in a later stage. We rewrite $\epsilon$ 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. 9. The algorithm takes a regular XPATH expression $E_Q$ over the DTD $D$ as input, and returns an equivalent sequence $Q'$ of $RA$ queries with the LFP operator $\Phi$ as output. The algorithm is based on dynamic programming: for each sub-expression $e$ of $E_Q$, it computes r2s($e$), which is the $RA$ query translation of $e$; it then associates r2s($e$) with a temporary table $R_e$ (which is used in later queries) and increments the list $Q'$ with $R \leftarrow$ r2s($e$). More specifically, r2s($e$) is computed from 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 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 handles different cases of $e$ as follows.

(1) It rewrites a label $A$ to the corresponding relation $R_A$ (case 2).

(2) Concatenation is coded with projection $\Pi$ and join $\bowtie$ (case 3).

(3) Union and disjunction are encoded with union $\cup$ in relational algebra (cases 4 and 10).

(4) Kleene closure $(E)^*$ is converted to the LFP operator $\Phi$ (case 5).

(5) Conjunction is coded with set intersection implemented with union $\cup$ and set difference $\setminus$ in relational algebra (case 9).

(6) An expression with qualifier $e = e_1[q]$ is converted to a $RA$ query r2s($e$) that returns only those r2s($e_1$) tuples $t_1$ for which there exists a 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).

(7) On the other hand, it rewrites $e_1[\neg q]$ to a $RA$ query r2s($e$) that returns only those r2s($e_1$) tuples $t_1$ for which there exists no 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

**Algorithm** RegToSQL

*Input:* a regular XPATH expression $E_Q$ over a DTD $D$.
*Output:* an equivalent list $Q'$ of $RA$ queries over $\mathcal{R}$, where $\tau : D \to \mathcal{R}$.

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) $\epsilon$: $\mathsf{r2s}(e) := R_{id}$;
6. (2) $A$: $\mathsf{r2s}(e) := R_A$;
7. (3) $e_1/e_2$: let $R_1 = \mathsf{r2s}(e_1)$, $R_2 = \mathsf{r2s}(e_2)$;
8. $\mathsf{r2s}(e) := \Pi_{R_1.F,R_2.T,R_2.V}(R_1 \bowtie_{R_1.T=R_2.F} R_2)$;
9. (4) $e_1 \cup e_2$: let $R_1 = \mathsf{r2s}(e_1)$, $R_2 = \mathsf{r2s}(e_2)$;
10. $\mathsf{r2s}(e) := R_1 \cup R_2$;
11. (5) $E^*$: let $R = \mathsf{r2s}(e)$;
12. $\mathsf{r2s}(e) := \Phi(R) \cup R_{id}$;
13. (6) $e_1[q]$: let $R_1 = \mathsf{r2s}(e_1)$, $R_q = \mathsf{r2s}(q)$;
14. $\mathsf{r2s}(e) := \Pi_{R_1.F,R_1.T,R_2.V}(R_1 \bowtie_{R_1.T=R_q.F} R_q)$;
/* returns $R_1$ tuples that connect with $R_2$ tuples */
15. (7) $[e_1]$: $\mathsf{r2s}(e) := \mathsf{r2s}(e_1)$;
16. (8) $e_1[text() = c]$: let $R_1 = \mathsf{r2s}(e_1)$;
17. $\mathsf{r2s}(e) := \sigma_{R_1.V=c}R_1$;
/* select tuples $t$ of $R_1$ with $t.V = c$ */
18. (9) $[q_1 \wedge q_2]$: let $R_1 = \mathsf{r2s}(q_1)$; $R_2 = \mathsf{r2s}(q_2)$;
19. $\mathsf{r2s}(e) := R_1 \cup R_2 \setminus ((R_1 \setminus R_2) \cup (R_2 \setminus R_1))$;
/* $\mathsf{r2s}(e) = R_1 \cap R_2$; */
20. (10) $[q_1 \vee q_2]$: let $R_1 = \mathsf{r2s}(q_1)$; $R_2 = \mathsf{r2s}(q_2)$;
21. $\mathsf{r2s}(e) := R_1 \cup R_2$;
22. (11) $e_1[\neg q]$: let $R_q = \mathsf{r2s}(q)$, $R_1 = \mathsf{r2s}(e_1)$;
23. $\mathsf{r2s}(e) := R_1 \setminus \Pi_{R_1.F,R_1.T,R_1.V}$
$(R_1 \bowtie_{R_1.T=R_q.F} R_q)$;
/* only $R_1$ tuples not connecting to any $R_q$ tuple */
24. $Q' := (R_e \leftarrow \mathsf{r2s}(e)) :: Q'$; /* add $\mathsf{r2s}(e)$ to $Q'$ */
25. $\mathsf{r2s}(E_Q) := \sigma_{F='\_'}\mathsf{r2s}(E_Q)$; /* select nodes reachable from root */
26. $Q' := \mathsf{r2s}(E_Q) :: Q'$;
27. optimize $Q'$ by extracting common sub-queries;
28. return $Q'$;

Figure 9: Rewriting algorithm from regular XPath to SQL

precisely the semantics of negation in XPATH (recall our assumptions about $[\neg q]$ and $[text() = c]$ from Section 2).

(8) It converts $[e_1]$ to $\mathsf{r2s}(e_1)$ when $e_1$ is a regular XPATH expression (case 7).

(9) It rewrites $e = e_1[text() = c]$ in terms of selection $\sigma$ that returns all tuples of $\mathsf{r2s}(e_1)$ that have the text value $c$.

| F | T |
|---|---|
| $d_1$ | $c_1$ |
| $c_1$ | $c_2$ |
| $c_2$ | $c_3$ |
| $p_1$ | $c_4$ |
| $s_2$ | $c_5$ |
| $c_1$ | $c_5$ |
| $c_2$ | $c_4$ |

(a) $R$

| F | T |
|---|---|
| $c_1$ | $c_2$ |
| $c_1$ | $c_3$ |
| $c_1$ | $c_4$ |
| $c_1$ | $c_5$ |
| ... | ... |

(b) $R_\gamma$

| F | T |
|---|---|
| $d_1$ | $p_1$ |
| $d_1$ | $p_2$ |

(c) $R_f$

Table 3: Intermediate and final results of `dept//project`.

In each of the cases above, the list $Q'$ is incremented by adding $R_e \leftarrow \mathsf{r2s}(e)$ to $Q'$ as the head of $Q'$ (line 24). Finally, after the iteration it yields $\sigma_{F='\_'}\mathsf{r2s}(E_Q)$ (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. 9), 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 input regular XPATH expression $E_Q$.

**Example 5.1:** Consider the XPATH query $Q_1 =$ `dept//project` over the `dept` DTD of Fig. 1 (a). Over the simplified DTD is Fig. 1 (b), $Q_1$ becomes $R_d//R_p$. Its equivalent $RA$ translation $Q_1'$ has been given in Example 3.2, which includes a single LFP operation $R_\gamma = \Phi(R) \cup \Pi_{T,T}(R_c)$, where $R = R_{cc} \cup R_{csc} \cup R_{cpc}$. When evaluated over the relational database of Fig. 1 (which encodes an XML tree of the `dept` DTD), $Q_1'$ produces $R$, $R_\gamma$, and the final result as shown in Table 3 (a), (b) and (c), respectively.

As another example, 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:

$E_{cc}$ : $R_\gamma$ *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 $Q_2 = Q_2^2[\neg q_3]$. Therefore, $Q_2^1 \leftarrow R_d \bowtie R_c \bowtie R_1$, $Q_2^2 \leftarrow Q_2^1 \setminus (Q_2^1 \bowtie R_{cp})$, and $E_{Q_2}$ becomes $Q_2^2 \setminus (Q_2^2 \bowtie R_2)$ where projections are omitted. In contrast, the algorithm of [21] cannot translate XPATH queries of this form to relational queries. □

Algorithm RegToSQL takes at most $O(|E_Q|)$ time. Taken together with the complexity of algorithm XPathToReg given in Section 4, 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.

### 5.2 Pushing Selections into the LFP Operator

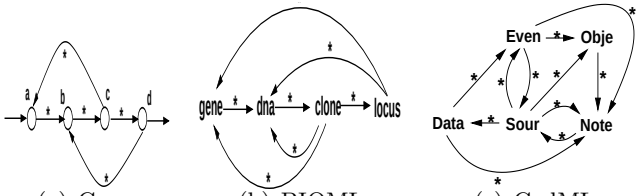
Algorithms XPathToReg and RegToSQL show that SQL with the simple LFP operator is powerful enough to answer XPATH queries over recursive DTDs. While certain optimizations are already conducted during the translation, other techniques, e.g., sophiscated methods for pushing selections/projections into the LFP operator [1, 3, 5]. 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 $R_2 \leftarrow \text{LFP}(R_0)$, where $Q_d$ and $\text{LFP}(R_0)$ 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 $R_\Phi$ and $R_0$ in LFP, where $R_0$ is the input relation and $R_\Phi$ 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 $R_\Phi$ 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 $\text{LFP}(R_0)$ for $\mathsf{rec}(R_d, R_c)$. 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.

### 5.3 Discussions

**Query Optimization**: Observe that in our generated relational queries, all joins and unions are outside of the LFP operator, as opposed to embedding joins/unions in the blackbox of the operator *with...recursive*. As a result, one can capitalize on RDBMS to optimize those joins/unions. Indeed, making use of relational optimizers is one of the reasons for one to want to push the work to RDBMS before XML query optimizers become as sophisticated as their RDBMS counterparts. Furthermore, our translation framework makes it easy to accommodate all existing techniques in commercial RDBMS [27, 16]; in particular, multi-query optimization techniques (e.g., [30]) can be easily incorporated into our framework to optimize a sequence of SQL queries produced by our algorithms.

**XML Reconstruction**: It is worth mentioning that our rewriting algorithms can be easily extended such that they not only find ancestor/descendant pairs, but also preserve the path information between each pair. A simple way to do so is to use an additional attribute $P$ in LFP




(a) Cross (b) BIOML (c) GedML

Figure 10: DTD Graphs

$\Phi()$ such that the $P$ attribute keeps track of the path information by concatenating edges when tuples are joined. Both DB2 and Oracle support such a string concatenation operator.

## 6 A Performance Study

To verify the effectiveness of our rewriting and optimization algorithms, we have conducted a performance study on evaluating XPATH queries using an RDBMS with three approaches:

- the SQLGen-R algorithm proposed in [21],
- 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,
- 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 simple yet representative DTD and two complex DTDs from real world. The simple DTD is depicted in Fig. 10 (a) (2 cross cycles). The two real-life DTDs are (1) a 4-cycle DTD extracted from **BIOML** (BIOpolymer Markup Language [6]), as shown in Fig. 10 (b); and (2) a 9-cycle DTD extracted from **GedML** (Genealogy Markup Language [15]), given in Fig. 10 (c).

While testing several different types of XPATH queries, our performance study focuses on the evaluation of // because // is the only operator in XPATH queries that, in the presence of recursive DTDs, leads to Kleene closures and therefore LFP in RDBMS, and is a dominant factor of XPATH query evaluation. Two considerations on query evaluation are given below. First, as shown in our rewriting algorithms, // is translated into a sequence of projection, join and union, along with LFP. The evaluation of this sequence should be isolated from other operators that do not contribute to the evaluation of //. Second, the non-recursive operators in XPATH queries are translated into selection, projection, join and union that the existing relational query processing techniques can support, and is beyond the scope of this evaluation.

Our experimental results demonstrate that our rewriting algorithms with Cycle-C outperform the other two in most cases.

**Implementation.** We have implemented a prototype system supporting SQLGen-R, Cycle-E and Cycle-C, using Visual C++, denoted by **R**, **E** and **C**, respectively, in all the figures. SQLGen-R rewrites a query with the *with...recursive* operator, while Cycle-E and Cycle-C translate a query to a sequence of SQL queries. We run a batch to execute these rewritten SQL queries. In this study, 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 compare SQLGen-R with ours on Oracle, because Oracle does not support the SQL'99 recursion. The queries output ancestor, descendant pairs.

**Testing Data**: Testing data were generated using IBM XML Generator (`http://www.alphaworks.ibm.com`). The input to the IBM XML Generator is a DTD file and a set of parameters. We mainly control two parameters, $X_L$ and $X_R$, in order to study the impacts of the shape of XML trees. Here $X_L$ is the maximum number of levels in the resulting XML tree. If a tree goes beyond $X_L$ levels, it will add none of the optional elements (denoted by * or ? in the DTD) and only one of each of the required elements (denoted by + or with no option); $X_R$ controls the maximum number of occurrences of child elements in the presence of the $*$ or + option. In other words, the number of children of each element of a type defined with this option is a random number between 0 and $X_R$. 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 XML tree. The default values used in our testing for $X_L$ and $X_R$ are 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 generated XML data was mapped to a relational database using the shared-inlining technique [34]. Indexes were generated for all possible joined attributes.

**Query Evaluation.** (1) We tested four XPATH queries: a query with //, a twig join query, a query with ¬ and //, and a query with ¬, ∨, ∧ and //. The testing was done using different databases (fixing the database size while varying the relation sizes). (2) We tested the scalability of our generated SQL queries w.r.t. different database sizes using a query containing //. (1) and (2) were conducted with the simple cross-cycle DTD graph. (3) 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 (3) is that the former tested the same queries with different databases, and the latter tested different queries with the same database. (4) We tested a simple // query on the complex real-life GedML DTD on several large databases.

### 6.1 Exp-1: Evaluation of Selective Queries

In this study, over the simple cross-cycle DTD (Fig. 10 (a)), we tested the following four XPATH queries:

- $Q_a$ = `a/b//c/d` (with //),
- $Q_b$ = `a[//c]//d` (a twig join query),
- $Q_c$ = `a[¬ //c]` (with ¬ and //), and
- $Q_d$ = `a[¬ //c ∨ (b ∧ //d)]` (with ¬, ∨, ∧ and //).

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

$$E_{b\_c} = \mathsf{rec}(b,c) = (E_{bb} \cup (E_{bb}/c/a/(E_{bb}/c/a)^*/E_{bb}))/c$$
$$E_{a\_b} = \mathsf{rec}(a,b) = a/(E_{bb}/c/a)^*/E_{bb}$$
$$E_{a\_c} = \mathsf{rec}(a,c) = a/(E_{bb}/c/a)^*/E_{bb}/c$$
$$E_{bb} = b/(c/d/b)^*$$

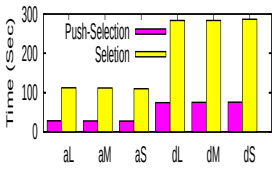



Figure 11: Pushing Selection ($X_R = 8$ and $X_L = 12$)

In contrast, Cycle-C generates the following:

$$E_{b\_c} = \mathsf{rec}(b, c) = b/(c/a/b \cup c/d/b)^*/c,$$
$$E_{a\_b} = \mathsf{rec}(a, b) = a/b/(c/a/b \cup c/d/b)^*,$$
$$E_{a\_c} = \mathsf{rec}(a, c) = a/b/(c/a/b \cup c/d/b)^*/c.$$

For each $\mathsf{rec}(A, B)$, 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 $\mathsf{rec}(A, B)$ in our translation framework. Since the cross-cycle DTD graph consists of 4 nodes 5 edges, SQLGen-R produced a *with...recursive* expression 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 tree ($X_R$). Since an XML tree with different heights and/or widths results in different sizes of relations in a database, even though the database size is the same, the same translated SQL query may end up having different query-processing costs. We report elapsed time (seconds) for each query in Fig. 12. For a single query, one figure shows the elapsed time while varying $X_L$ from 8 to 20 with $X_4 = 4$, whereas the other figure shows the elapsed time while varying $X_R$ from 4 to 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. 10 (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-2: Scalability Test

Figure 13 demonstrates the scalability of our algorithms by increasing the dataset sizes, for an XPATH query `a//d` over the cross-cycle DTD (Fig. 10 (a)). The XML dataset size increases to 960,000 elements from 120,000 elements. We set $X_L = 16$, because the default $X_L = 12$ is not large enough for the XML generator to generate 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) is effective enough to outperform *with...recursive*, because it can reduce the number of LFP operators and unnecessary joins and unions. Furthermore, Cycle-C is linearly scalable.

### 6.4 Exp-3: Complex Cycles (Extracted from Real-Life DTDs)

We next show the results of testing XPATH queries on the extracted 4-cycle BIOML DTD and the 9-cycle GedML DTD.

First, we tested XPATH queries over the extracted DTD graphs from BIOML. We considered four subgraphs of the BIOML DTD of Fig. 10 (b) in order to demonstrate the impact of different DTDs on the translated SQL queries. These subgraphs are shown in Fig. 14. Similar XPATH queries were tested in the presence of these extracted DTD graphs, and are summarized in Table 4.

All these XPATH queries were run on the same dataset which was generated using the largest 4-cycle DTD graph extracted from BIOML (Fig. 10 (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 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.

We show the query processing results in Fig. 15. Except case 2a, Cycle-E outperforms SQLGen-R. We find that Cycle-C outperforms SQLGen-R and Cycle-E in all the cases. In case 4a, for example, SQLGen-R needs to use 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 [35], it is inflexible to change the order of execution. Cycle-C significantly outperforms SQLGen-R and Cycle-E because less number of join and LFP are used, while it uses more union operators than others. The cost of union is comparatively small, if one relation involved in the union operator is indexed.

Second, we tested an XPATH query, `Even//Data`, over the 9-cycle DTD graph extracted from GedML (Fig. 10 (c)). Here SQLGen-R uses 11 joins and 11 unions in each iteration, because this DTD consists of 11 edges. Cycle-E generates a sequence of 23 join, 4 LFP and 16 union operators. Cycle-C uses 10 join, 1 LFP and 10 union operators, because there are three key label paths: *Even* → *Sour* → *Data*, *Even* → *Note* → *Sour* → *Data*, and *Even* → *Obje* → *Note* → *Sour* → *Data*. Although all the simple cycles on every key label path shares a common node *Sour* on which cycle contraction is processed, the number of key label paths and the lengths of key label paths all contribute to the processing cost.

For this test, we generated large datasets using the IBM XML Generator without trimming. Figure 16 (a) shows the results while varying $X_L$ with $X_R = 6$. The dataset sizes are 286,845 ($X_L = 13$), 84,5045 ($X_L = 14$), 1,019,798 ($X_L = 15$), and 5,320,417 ($X_L = 16$). Figure 16 (b) shows the results while varying $X_R$ with $X_L = 16$. The dataset sizes are 13,992 ($X_R = 5$), 226,663 ($X_R = 6$), 119,999 ($X_R = 7$), and 5,041,437 ($X_R = 8$). The largest datasets are 2 times larger than that used for the BIOML test. All the three algorithms performed in a similar way. Cycle-C marginally outperforms Cycle-E and SQLGen-R for different $X_L$ values in Fig. 16 (a). When the width parameter $X_R$ is small, SQLGen-R performs better than Cycle-E and Cycle-

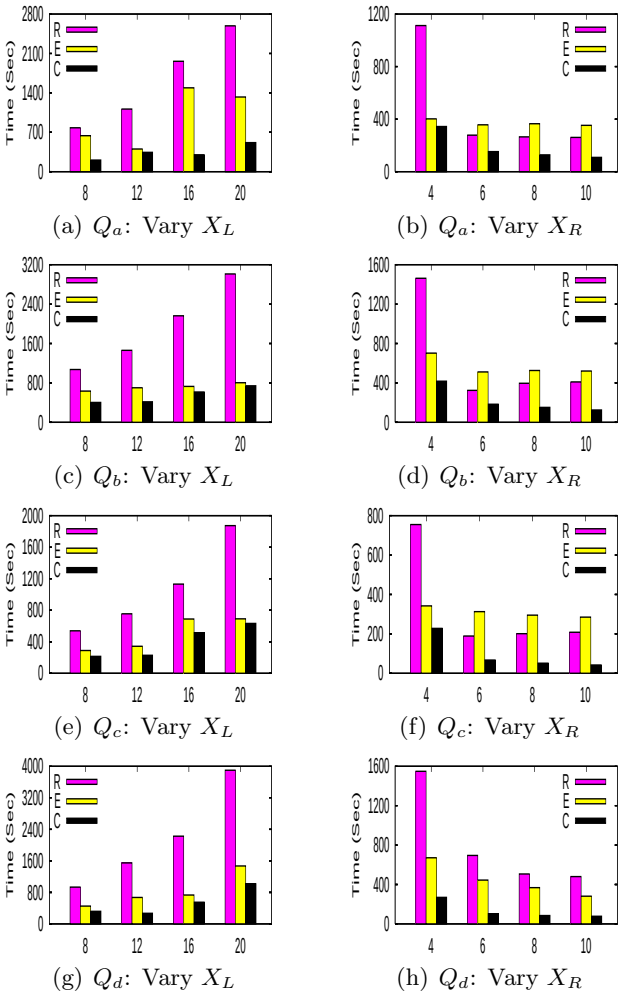



Figure 12: Processing time for cross cycles (Fig. 10 (a)).

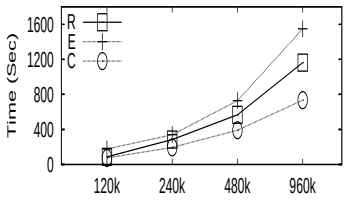



Figure 13: Scalability Test ($X_R = 4$ and $X_L = 16$)

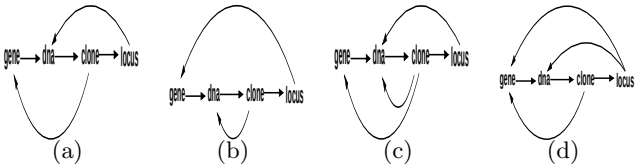



Figure 14: Different DTD graphs extracted from BIOML

C because Cycle-C needs to join all relations on key label paths. When $X_R$ becomes larger, e.g., $X_R = 8$, Cycle-C performs better (Fig. 16 (b)). In this case, Cycle-C, Cycle-E and SQLGen-R used 3,867, 26,353, and 12,507 seconds.

| Case | Query | $n$-Cycles | DTD Graph |
|---|---|---|---|
| 2a | gene//locus | 2 | Fig. 14 (a) |
| 2b | gene//locus | 2 | Fig. 14 (b) |
| 2c | gene//dna | 2 | Fig. 14 (b) |
| 3a | gene//locus | 3 | Fig. 14 (c) |
| 3b | gene//locus | 3 | Fig. 14 (d) |
| 4a | gene//locus | 4 | Fig. 10 (b) |
| 4b | gene//dna | 4 | Fig. 10 (b) |

Table 4: XPATH queries over different DTD graphs extracted from BIOML

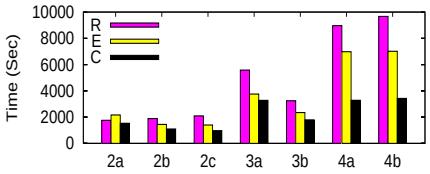
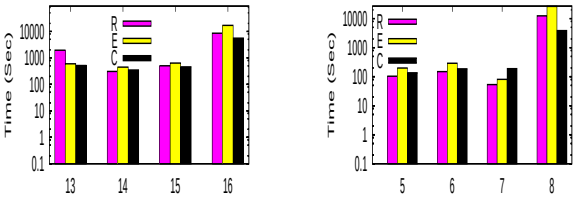



Figure 15: XPATH queries on the extracted BIOML DTDs




(a) Vary $X_L$ ($X_R = 6$) (b) Vary $X_R$ ($X_L = 12$)

Figure 16: `Even//Data` on the extracted 9-cycle GEDML DTD

## 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., [10, 11, 25, 13, 21, 18, 33, 32]; see [22] for an excellent recent survey). However, with the exception of the recent work of [21], as observed by [22], no algorithm has been published for handling recursive XML queries over recursive DTDs for schema-based XML storage or in the context of XML publishing. Closest to our work is [21], which proposed the first technique to rewrite (recursive) path queries over recursive DTDs to SQL with the SQL'99 recursion operator. 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 [33, 32] and SilkRoute [13]. 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 supported by middleware, and answers the query by using the computing power of 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 by [21], no algorithms have been developed for handling recursive queries over XML views with a recursive DTD.

Another approach is by providing an algorithm for rewriting XML queries into SQL (extended with a recursion operator), which is the approach adopted by this work. To this end, translation and optimization techniques have been proposed for translating XSLT

queries [18], XQuery [10, 11, 23, 25] and (recursive) path queries [21]. While the algorithms of [18, 10, 11, 23, 25] cannot handle query translation in the presence of recursive DTDs, their optimization techniques by leveraging, e.g., integrity constraints [11, 23], virtual generic schema and query normalization [25], dynamic interval encoding [10] and aggregation handling [18] are complementary to our work. Some of these, along with techniques for query pruning and rewriting [12], minimizing the use of joins [24], multi-query [30] and recursive-query optimization [31], can be incorporated into our translation framework.

Surveys on recursive and cyclic query processing strategies can be found in [4, 19]. For OODBs, [20] introduced techniques for processing cyclic queries, which are restricted to 1-cycle queries. [8] proposed optimization techniques for generalized path expressions based on OO algebraic transformation rules. These techniques are not directly applicable to query translations from XML to SQL.

## 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 (1) a notion of regular XPATH expressions capable of capturing DTD recursion and XPATH recursion in a uniform framework; (2) an efficient algorithm for translating an XPATH query over a recursive DTD to an equivalent regular XPATH expression; (3) an efficient algorithm for rewriting a regular XPATH expression into an equivalent sequence of SQL queries; and (4) new optimization techniques for minimizing the use of the LFP operator. These provide the capability of answering important XPATH queries within the immediate reach of most commercial RDBMS.

Several extensions to the optimization techniques are targeted for future work. First, we recognize that the use of the LFP operator in produced SQL queries is not the only factor for efficiency, and we are currently developing a cost model in order to provide better guidance for XPATH query rewriting. We are also exploring techniques for multi-query and recursive-query optimization [30, 31] to simplify SQL queries produced by our translations algorithms. We intend to incorporate optimization by means of semantic information such as integrity constraints [11] and satisfiability analysis of XPATH queries in the presence of DTDs. We also plan to extend our algorithms to handle more complex XML queries, over XML data stored in an RDBMS or (virtual) XML views of relational data.

## References


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

[2] A. Aho and J. Ullman. Universality of data retrieval languages. In *POPL*, 1979.

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

[4] F. Bancilhon and R. Ramakrishnan. An amateur's introduction to recursive query processing strategies. In *SIGMOD*, 1986.

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

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

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

[8] V. Christophides, S. Cluet, and G. Moerkotte. Evaluating queries with generalized path expressions. In *SIGMOD*, 1996.

[9] J. Clark and S. DeRose. XML path language (XPath). In *W3C Recommendation, Nov. 1999*.

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

[11] A. Deutsch and V. Tannen. MARS: A system for publishing XML from mixed and redundant storage. In *VLDB*, 2003.

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

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

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

[15] GedML. Genealogy Markup Language. *http://xml.coverpages.org/gedml-dtd9808.txt*.

[16] G. Graefe. Query evaluation techniques for large databases. *ACM Computing Surveys*, 25(2), 1993.

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

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

[19] Kambayashi. *Query Processing in Database Systems*, chapter Processing Cyclic Queries, pages 63–78. Springer, 1985.

[20] Y.-C. Kim, W. Kim, and A. Dale. Cyclic query processing in object-oriented databases. In *ICDE*, 1989.

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

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

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

[24] I. K. Kunen and D. Suciu. A scalable algorithm for query minimization. Technical report, University of Washington, 2004.

[25] I. Manolescu, D. Florescu, and D. Kossmann. Answering XML queries on heterogeneous data sources. In *VLDB*, 2001.

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

[27] P. Mishra and M. H. Eich. Join processing in relational databases. *ACM Computing Surveys*, 24(1), 1992.

[28] 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*.

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

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

[31] M.-C. Shan and M.-A. Neimat. Optimization of relational algebra expressions containing recursion operators. In *ACM Annual Computer Science Conference*, 1999.

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

[33] J. Shanmugasundaram, E. Shekita, R. Barr, M. Carey, B. Lindsay, H. Pirahesh, and B. Reinwald. A general techniques for querying XML documents using a relational database system. *SIGMOD Record*, 30(3), 2001.

[34] 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.

[35] R. E. Tarjan. Fast algorithms for solving path problems. *Journal of the ACM*, 28(3):594–614, 1981.

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