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

Wenfei Fan†  Jeffrey Xu Yu‡  Hongjun Lu‡  Jianhua Lu‡  Rajeev Rastogi§

†University of Edinburgh & Bell Laboratories
wenfei@inf.ed.ac.uk

‡The Chinese University of Hong Kong, Hong Kong
{yu,jhlu}@se.cuhk.edu.hk

‡The Hong Kong University of Science and Technology
luhj@cs.ust.hk

§Bell Laboratories (India)
rastogi@research.bell-labs.com
Overview

- Introduction of the XML to SQL translation problem.
- An database example
- The class of XPATH queries we support
- A related work
- Our new approach
- Performance study
- Conclusion
The XML to SQL Translation (1)

- Consider a mapping \( \tau \) from DTD \( D \) to a relational schema \( \mathcal{R} \),
  \[ \tau : D \rightarrow \mathcal{R} \]
  defined in terms of DTD-based shredding.
- XML documents, \( T \), conforming to \( D \) can be mapped onto relations of \( \mathcal{R} \), based on a data mapping \( \tau_d \).
- Given an XML query \( Q \), find (a sequence of) equivalent SQL queries \( Q' \).
  \[ Q(T) = Q'(\tau_d(T)) \]
The XML to SQL Translation (2)

- DTDs can be **recursive**.
- XPATH queries may use **recursion** (//, descendants-or-self).

(a) BIOML

(b) GedML: Genealogy ML
The XML to SQL Translation (3)

- What class of XPath queries can be supported?
- What are the operators needed in SQL to support the class of XPath queries?
- How many RDBMS can support the required operators?
A DTD Example ($dept$)

- Let $D$ be a DTD as follows.

```xml
<!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*>  
```
**DTD-based shredding for dept**

- $\tau(D)$: $R_d(F, T)$, $R_c(F, T, \text{cno, title, prereq, takenBy})$, $R_s(F, T, \text{sno, name, qualified})$, $R_p(F, T, \text{pno, ptitle, required})$
An Example Database for \texttt{dept}
Two XPath Queries

- Find all course-related projects.
  \[ Q_1 = \text{dept//project} \]

- Find courses that
  - have a prerequisite cs66,
  - have no project related to them or to their prerequisites,
  - have a student who registered for the course but did not take cs66.

\[ Q_2 = \text{dept/course[//prereq/course/cno="cs66" \∧
  \¬//project \∧
  \¬takenBy/student/qualified//course/cno = "cs66"]} \]
**XPATH Queries**

- We support a fragment of XPATH that supports recursion (descendants) and rich qualifiers.

\[
p ::= \epsilon \mid A \mid \ast \mid p/p \mid p//p \mid p \cup p \mid p[q]
\]

\[
q ::= p \mid text() = c \mid \neg q \mid q \land q \mid q \lor q
\]
A Linear Recursion SQL’99 Approach (1)

- By R. Krishnamurthy et al in ICDE’04 to handle recursive path queries over recursive DTDs based on the SQL’99 recursion operator.

- 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 partitions $G_Q$ into strongly-connected components $c_1, \ldots, c_n$, sorted in the top-down topological order.
A Linear Recursion SQL’99 Approach (2)

- 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.

- The component $c_i$ is cyclic, and the SQL query $Q_i$ is defined in terms of the with...recursive operator.
A Linear Recursion SQL’99 Approach (3)

- 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:

\[
\begin{align*}
R^0 & \leftarrow R \\
R^i & \leftarrow R^{i-1} \cup (R^{i-1} \bowtie R_1) \cup \cdots \cup (R^{i-1} \bowtie R_k)
\end{align*}
\]

where $R^0$ corresponds to the initialization part, and $R_j$ corresponds to an SQL query coding an edge in $c_i$ for each $j \in [1, k]$. 
Linear Recursion of SQL’99 for $Q_1 (\text{dept} // \text{project})$

1. with
2. $R (F, T, \text{Rid})$ as
3.  (select $R_c.F$, $R_c.T$, $\text{Rid}(\text{c'})$ from $R_d$, $R_c$
4.  where $R_c.T = R_d.F$
5.  union all
6.  (select $R.F$, $R_c.T$, $\text{Rid}(\text{c'})$
7.  from $R$, $R_c$ where $R.T = R_c.F$ and $\text{Rid} = \text{c'}$
8.  union all
9.  (select $R.F$, $R_s.T$, $\text{Rid}(\text{s'})$
10. from $R$, $R_s$ where $R.T = R_s.F$ and $\text{Rid} = \text{c'}$
11. union all
12. (select $R.F$, $R_c.T$, $\text{Rid}(\text{c'})$
13. from $R$, $R_c$ where $R.T = R_c.F$ and $\text{Rid} = \text{s'}$
14. union all
15. (select $R.F$, $R_p.T$, $\text{Rid}(\text{p'})$
16. from $R$, $R_p$ where $R.T = R_p.F$ and $\text{Rid} = \text{c'}$
17. union all
18. (select $R.F$, $R_c.T$, $\text{Rid}(\text{c'})$
19. from $R$, $R_c$ where $R.T = R_c.F$ and $\text{Rid} = \text{p'}$)
The Limits of the SQL’99 Approach

- It relies on the SQL’99 recursion functionality, which is not currently supported by many commercial products including Oracle and Microsoft SQL server.

- The SQL queries generated are typically large and complex. It may not be effectively optimized by all platforms supporting SQL’99 recursion.
The Overview of Our Approach

- **Regular XPath expressions** which extend XPath by supporting general Kleene closure $E^*$ instead of $//$.
- **A simple least fixpoint** (LFP) operator, $\Phi(R)$, which takes a single input relation $R$.

**Diagram:**

```
input XPath query $Q$  →  translation from XPath $Q$ to regular XPath expression  →  $E_Q$  →  translation from regular XPath to a sequence of SQL queries with the simple LFP operator  →  output $Q'$
```

**Diagram Notes:**
- Input XPath query $Q$ is translated to a regular XPath expression $E_Q$.
- The regular XPath expression is then translated to a sequence of SQL queries using the simple LFP operator $\Phi(R)$.
- The output is the translated SQL query $Q'$. 

**Diagram Elements:**
- **input XPath query $Q$**
- **translation from XPath $Q$ to regular XPath expression**
- **$E_Q$**
- **translation from regular XPath to a sequence of SQL queries with the simple LFP operator**
- **output $Q'$**
- **DTD $D$**
- **mapping from $D$ to $R$**

---

**Query Translation from XPath to SQL in the Presence of Recursive DTDs**
The Simple LFP Operator (1)

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

$$
R^0 \leftarrow R
$$

$$
R^i \leftarrow R^{i-1} \cup (R^{i-1} \bowtie_C R^0)
$$

It is supported by DB2 and Oracle, and will be supported by Microsoft SQL Server 2005 using common table.
The Simple \textbf{LFP Operator} (2)

- The \textbf{LFP} operator handles \textbf{Kleene Closure} $E^*$.
- A regular \textbf{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 \textbf{XPATH} expression can be rewritten into $\Phi(R)$ by letting

$$R \leftarrow \Pi_{R_2.F,R_1.T}(R_2 \bowtie R_3 \bowtie \cdots \bowtie R_n \bowtie R_1)$$
Use LFP for \( Q_1 (\text{dept}/**/\text{project}) \)

- Translate \( Q_1 \) to a regular XPATH query

\[
E_{Q_1} = R_d/R_c/E^*/R_p
\]

where \( E = (R_c \cup R_s/R_c \cup R_p/R_c) \).

- Rewrite \( E_{Q_1} \) to a sequence of SQL queries (written in relational algebra).
Use LFP for $Q_1$ (dept//project)

\[
R_{cc} \leftarrow R_c \\
R_{csc} \leftarrow \Pi_{R_s.F,R_c.T}(R_s \bowtie_{R_s.T=R_c.F} R_c) \\
R_{cpc} \leftarrow \Pi_{R_p.F,R_c.T}(R_p \bowtie_{R_p.T=R_c.F} R_c) \\
R \leftarrow R_{cc} \cup R_{csc} \cup R_{cpc} \\
R_{\gamma} \leftarrow \Phi(R) \cup \Pi_{T,T}(R_c) \\
R_f \leftarrow \Pi_{R_d.T,R_p.T}(R_d \bowtie_{R_d.T=R_c.F} R_c \\
\bowtie_{R_c.T=R_{\gamma}.F} R_{\gamma} \\
\bowtie_{R_{\gamma}.T=R_p.F} R_p)
\]
The Sizes

- A \textsc{XPath} query \( Q \) over a \textsc{DTD} \( D \) will be rewritten to \textsc{SQL} queries (with the \textsc{LFP} operator), via an equivalent regular \textsc{XPath} expression \( E_Q \).

- \( E_Q \) is bounded by \( O(|Q| \cdot |D|^{4\log|D|}) \).

- \textsc{SQL} is bounded by \( O(|Q| \cdot |D|^{4\log|D|}) \).
All Data Paths from A to B

- The LFP operator is perhaps most costly.
- Tarjan’s fast algorithm finds a regular expression representing all the paths between two nodes in a (cyclic) graph.
- Can we generate an $E_Q$ that contains a few Kleene closures?
Optimization via Cycle 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 for $A//B$.
- It first enumerates all distinct simple paths between $A$ and $B$ in $G_D$, denoted by $AB$-paths.
- An $AB$-path 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 algorithm encodes $C_j$ with a simple regular expression $E^*_{C_j}$, where $E_{C_j}$ represents the simple path of $C_j$. 
Example-1 (a//c)

- A DTD with 3 simple cycles: $C_1 = a \rightarrow b \rightarrow a$, $C_2 = c \rightarrow f \rightarrow c$, and $C_3 = a \rightarrow c \rightarrow f \rightarrow b \rightarrow a$. **AB-path**: $L = a \rightarrow c$.
- $C_1$ and $C_3$ share $a$ on $L$, and $C_2$ and $C_3$ share $c$.
- Contracts $C_1, C_3$ and replace $a$ with a regular expression $a/E_{\gamma_1}$, which captures paths from $a$ to $a$ via $C_1$ and $C_3$.
- Then contracts $C_2$ and $C_3$ by replace $c$ with $c/E_{\gamma_2}$, which captures paths from $c$ to $c$ via $C_2$ and $C_3$.
- The final result is $E = a/E_{\gamma_1}/c/E_{\gamma_2}$.
Example-2 (a//c)

- A DTD 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 $C_4 = b \rightarrow f \rightarrow b$.
- Two AB-paths: $L_1 = a \rightarrow c$ and $L_2 = a \rightarrow b \rightarrow f \rightarrow c$.
- On $L_1$ there are three simple cycles: $C_1$, $C_2$ and $C_3$.
- On $L_2$ there are $C_1$, $C_2$ and $C_4$.
- The result regular XPath expression is $E_{L_1} \cup E_{L_2}$, where each $E_{L_i}$ is generated for $L_i$. 
A Performance Study

- Testing data was generated using IBM XML Generator. The input to the Generator is a DTD file and a set of parameters.

- We mainly controlled two parameters, $X_L$ and $X_R$, where $X_L$ is the maximum number of levels in the resulting XML tree, and $X_R$ is the maximum number of children of any node in the tree.

- The default values used in our testing for $X_L$ and $X_R$ were 4 and 12, respectively.

- The default number of elements in a generated XML tree was 120,000.
Selected Testing Results

- $Q_a = a/b//c/d$ (with //)
Push Selection

- $Q_e = a[id = A_i]/b//c/d$.
- $Q_f = a/b//c/d[id = D_i]$. 
Conclusion

• We 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.

• These provide the capability of answering important XPATH queries within the immediate reach of most commercial RDBMS.