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

Jeffrey Xu Yu

Chinese University of Hong Kong, China yu@se.cuhk.edu.hk

A joint work by Wenfei Fan, Jeffrey Xu Yu, Hongjun Lu and Jianhua Lu.

The XML to SQL Translation Problem

- Consider a mapping τ_d, defined in terms of DTD-based shredding, from XML documents conforming to a DTD D to relations of a schema $\mathcal{R}$.
- Given an XML query Q, find (a sequence of) *equivalent* SQL queries Q' such that for any XML document T conforming to D, Q on T can be answered by Q' on the database $\tau_d(T)$ of $\mathcal{R}$ that represents T, i.e.,

$$Q(T) = Q'(\tau_d(T)).$$

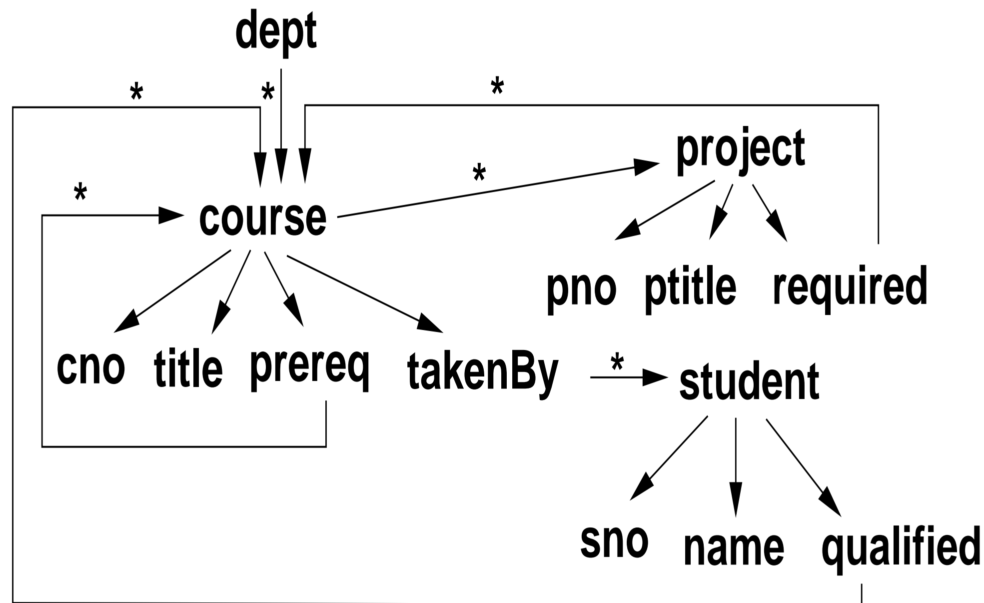
- We allow DTDs D to be recursive and consider queries Q in XPATH. which is essential for XML query languages XQuery.

A DTD Example (dept)

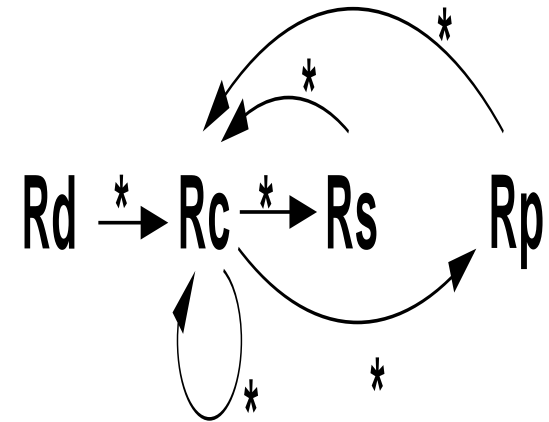
- Let D be a DTD as follows.

```
<!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



(a)



(b)

- Then, $\tau(D)$ becomes

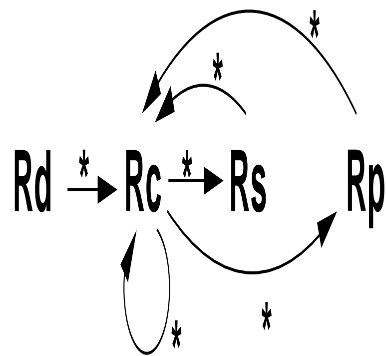
  R_d(F, T)

  R_c(F, T, cno, title, prereq, takenBy)

  R_s(F, T, sno, name, qualified)

  R_p(F, T, pno, ptitle, required)

An Example Database for `dept`



F	T
-	d_1

R_d

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

R_c

F	T
c_1	s_1
c_1	s_2

R_s

F	T
c_2	p_1
c_4	p_2

R_p

Two XPATH Queries

- The first is to find all course-related projects.

  Q_1 = `dept//project`

- The second is to 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 = `dept/course[//prereq/course/cno="cs66"` $\wedge$ $\neg$`//project`
  $\wedge$ $\neg$`takenBy/student/qualified//course/cno = "cs66"]`

SQLGen-R: A Linear Recursion of SQL'99 Approach (1)

- SQLGen-R is proposed by R. Krishnamurthy et al in ICDE'04 to handles 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.
- 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.

SQLGen-R: A Linear Recursion of SQL'99 Approach (2)

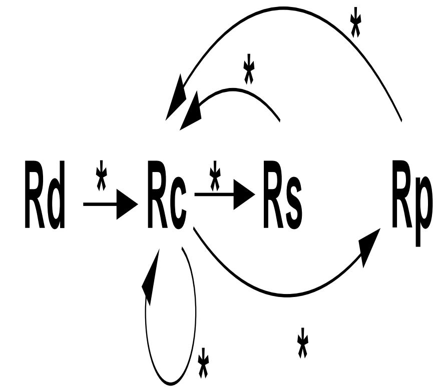
- If a component c_i is cyclic, the SQL query Q_i is defined in terms of the *with...recursive* operator.
- It generates two parts from c_i: an *initialization* part and a *recursive* part.
- The initialization part captures all "incoming edges" into c_i.
- The recursion part first creates an SQL query for each edge in c_i, and then encloses the union of all these (edge) queries in a *with...recursive* expression.
- 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:

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

  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 (`dept//project`)

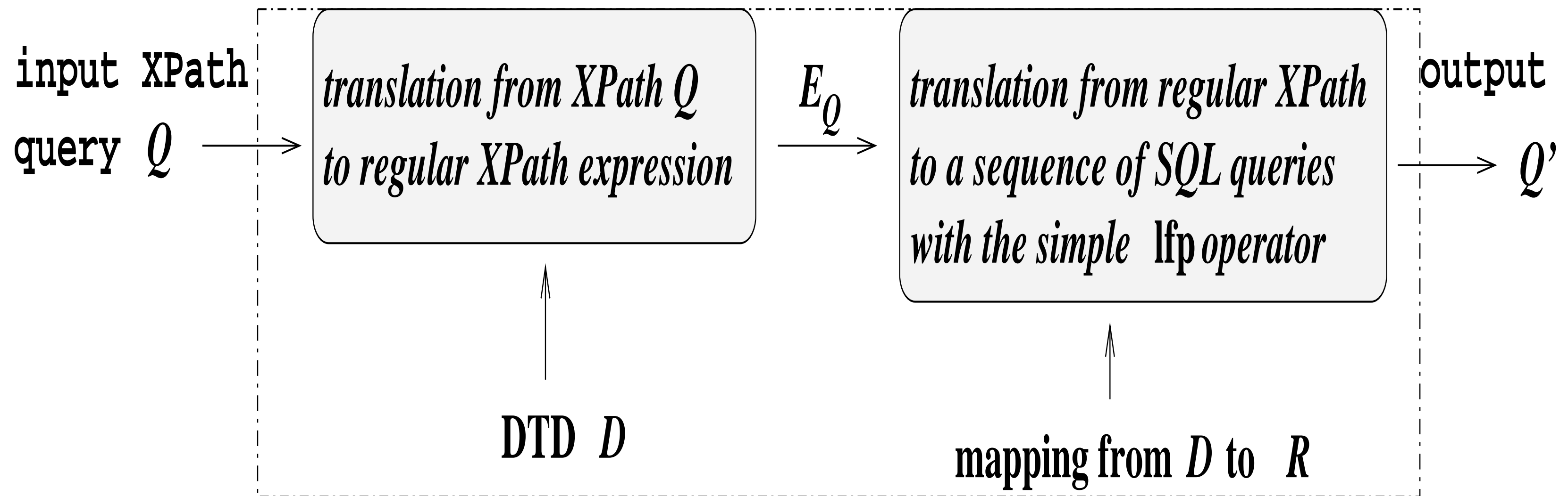
1. **with**
2. $R\ (F, T, Rid)$ **as** (
3. (**select** $R_c.F$, $R_c.T$, Rid('c') **from** R_d, R_c)
4. **where** $R_c.T = R_d.F$
5. **union all**
6. (**select** $R.F$, $R_c.T$, Rid('c')
7. **from** R, R_c **where** $R.T = R_c.F$ **and** Rid = 'c')
8. **union all**
9. (**select** $R.F$, $R_s.T$, Rid('s')
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'))



The Issues Related to SQLGen-R

- It is an elegant approach to translating path queries to SQL'99.
  - translate queries with // and limited qualifiers to (a sequence of) SQL queries with the linear-recursion construct *with...recursive*.
- It has several limitations.
  - 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
  - Path queries handled are too restricted to express XPATH queries commonly found in practice.

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 instead of multiple relations as does the SQL'99 *with...recursion* operator.

The Simple LFP Operator

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

$$\begin{aligned} R^0 &\leftarrow R \\ R^i &\leftarrow R^{i-1} \cup (R^{i-1} \bowtie_C R^0) \end{aligned}$$

It is supported by DB2 and Oracle, and will be supported by Microsoft SQL Server 2005 using *common table*.

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_Φ, R **where** $R_\Phi.T = R.F$)

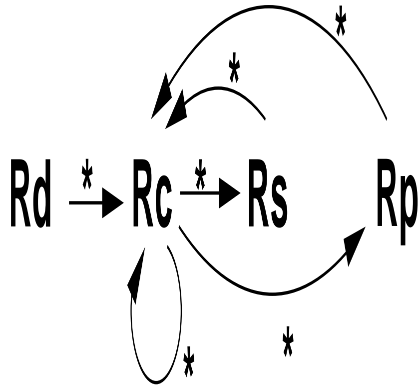
The Kleene Closure and The Simple LFP Operator

- The LFP Operator handles Kleene Closure E^*.
- 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 $\Phi(R)$ by letting

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

  Here, the projected attributes are taken from the attributes F (from) and T (to) in relations R_2 and R_1, respectively. The join between R_i/R_j is expressed as $R_i \bowtie_{R_i.T=R_j.F} R_j$.

Use LFP for Q_1 (`dept//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).

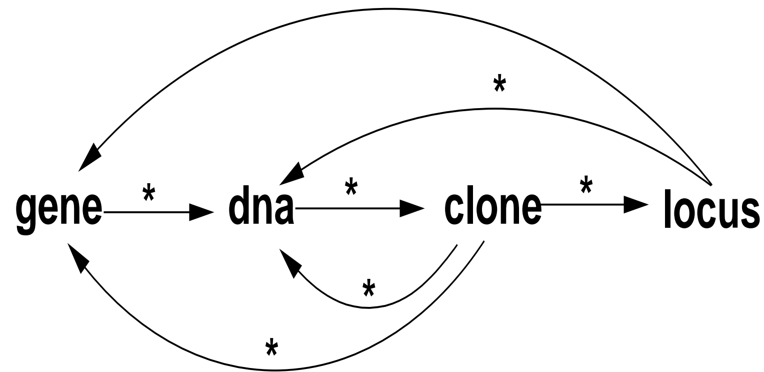
$$
\begin{aligned}
R_{cc} &\leftarrow R_c \\
R_{csc} &\leftarrow \Pi_{R_s.F, R_c.T}(R_s \bowtie_{R_s.T=R_c.F} R_c) \\
R_{cpc} &\leftarrow \Pi_{R_p.F, R_c.T}(R_p \bowtie_{R_p.T=R_c.F} R_c) \\
R &\leftarrow R_{cc} \cup R_{csc} \cup R_{cpc} \\
R_\gamma &\leftarrow \Phi(R) \cup \Pi_{T,T}(R_c) \\
R_f &\leftarrow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

From XPath to Regular XPath

- Rewrite an XPATH query Q over a (recursive) DTD D to an equivalent regular XPATH expression E_Q over D such that for any XML tree T of the DTD D, $Q(T) = E_Q(T)$.
- Propose A translation algorithm, XPathToReg, based on *dynamic programming*.
- For each sub-query p of the input query Q and type A in D, XPathToReg computes the local translation $E_p = \text{x2r}(p, A)$ from XPATH p to a regular XPATH expression E_p, such that p and E_p are equivalent when being evaluated at an A element.
- Composing the local translations one will get the rewriting $E_Q = \text{x2r}(Q, r)$ from Q to E_Q, where r is the root type of the DTD.

All The Paths from A to B

- Let rec(A, B) to denote the regular expression representing all the paths from A to its descendant B in the DTD graph G_D of D.
- Compute rec(A, B) to generate a regular XPATH expression, E^*, using Tarjan's fast algorithm which finds a regular expression representing all the paths between two nodes in a (cyclic) graph.
- Among the relational operators in Q', the LFP operator is perhaps most costly. Can we generate an E_Q to contain as few Kleene closures as possible?



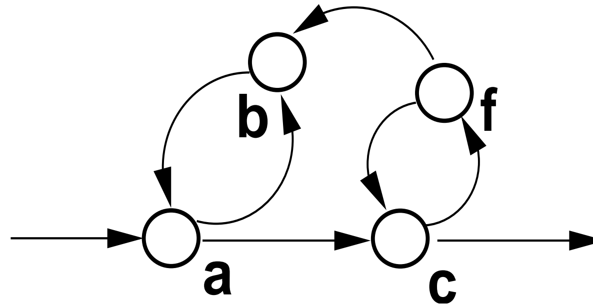
Optimization via Cycle Contraction (Cycle-C) (1)

- 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 rec(A, B).
- It first enumerates all distinct simple paths (i.e., paths without repeating labels) between A and B in G_D, 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 algorithm encodes C_j with a simple regular expression $E^*_{C_j}$, where E_{C_j} represents the simple path of C_j.

Optimization via Cycle Contraction (Cycle-C) (2)

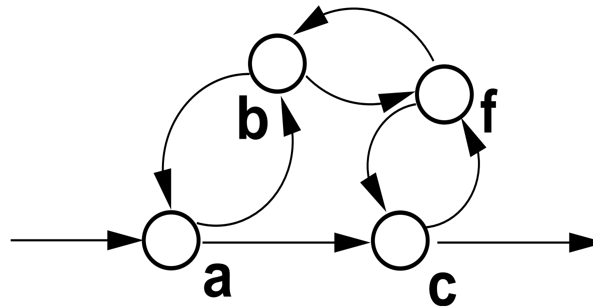
- Recall an *AB-path* is $A_1 \rightarrow \ldots \rightarrow A_k$.
- It *contracts* C_j to the node A_i and replaces A_i in E_i with $A_i/E^*_{C_j}$; as a result of the contraction, cycles that were not directly connected to L_i may become directly connected to L_i. The algorithm repeats this process until all the cycles connected to L_i, directly or indirectly, have been incorporated into E_i.
- One can verify that rec(A, B) is indeed $(E_1 \cup \ldots \cup E_n)$, for all the *AB-path*s, $L_1, \ldots, L_n$.
- Note that all simple cycles of a directed graph can be efficiently identified.

Example-1



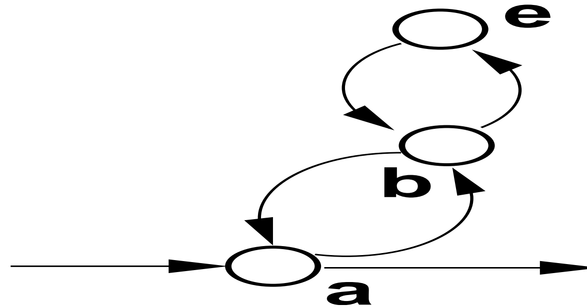
- A DTD with 3 simple cycles: $C_1 = a \to b \to a$, $C_2 = c \to f \to c$, and $C_3 = a \to c \to f \to b \to a$.
- Consider rec(a, c), the only *AB-path* is $L = a \to c$.
- C_1 and C_3 share a on L, and C_2 and C_3 share c.
- Contracts C_1, C_3 and replace a with a regular expression a/E_{γ_1}, which 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_{γ_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 DTD with 4 simple cycles $C_1 = a \to b \to a$, $C_2 = c \to f \to c$, $C_3 = a \to c \to f \to b \to a$, and $C_4 = b \to f \to b$.
- Consider $\text{rec}(a, c)$, which has two *AB-path*s: $L_1 = a \to c$ and $L_2 = a \to b \to f \to c$.
- On L_1 there are three simple cycles: C_1, C_2 and C_3.
- 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 based on the single *AB-path* cases above.

Example-3



- A DTD with 2 simple cycles $C_1 = a \to b \to a$ and $C_2 = b \to e \to b$.
- Consider $\text{rec}(a, a)$, for which the *AB-path* is a.
- C_2 does not directly connect to a, but it is on C_1.
- First, generate a regular expression $E = a$.
- Second, contract C_2, generate E_{C_2} to capture C_2 and replace b in C_1 with b/E_{C_2}.
- Finally, we contract C_1 and replace a with a/E_{C_1}, which includes E_{C_2}.

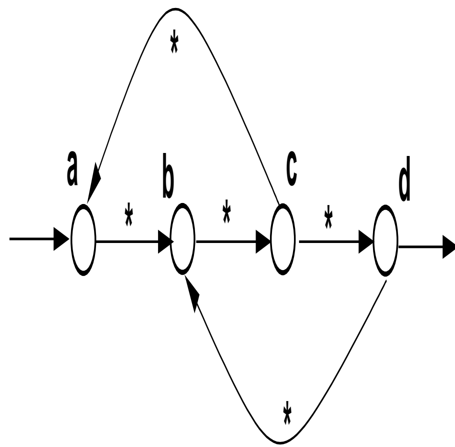
From Regular XPath Expressions to SQL

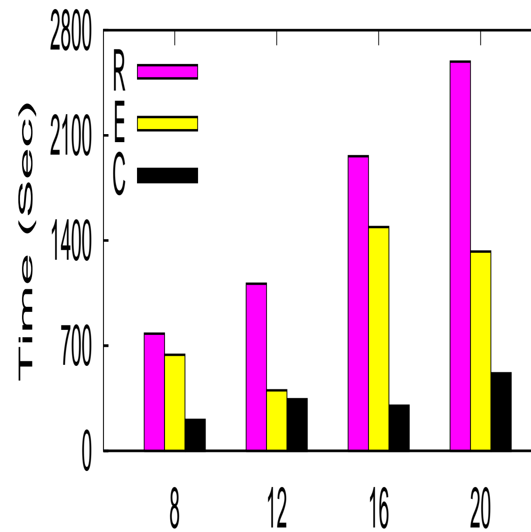
- Consider a mapping $\tau : D \rightarrow \mathcal{R}$, where D is a DTD and $\mathcal{R}$ is a relational schema, such that its associated data mapping τ_d shreds XML trees of D into databases of $\mathcal{R}$.
- Given a regular XPATH expression E_Q over D, compute a sequence Q' of equivalent relational queries with the simple LFP operator Φ such that for any XML tree T of D, $E_Q(T) = Q'(\tau_d(T))$.

A Performance Study

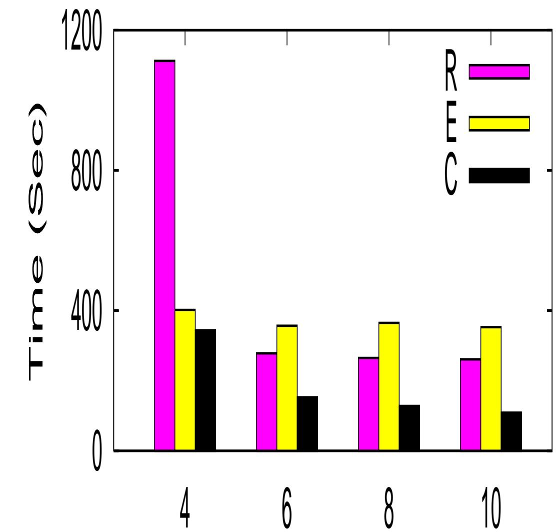
- Testing data was generated using IBM XML Generator. The input to the Generator is a DTD file and a set of parameters.
- We mainly controlled two parameters, X_L and X_R, where X_L is the maximum number of levels in the resulting XML tree, and X_R is the maximum number of children of any node in the 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 (1)





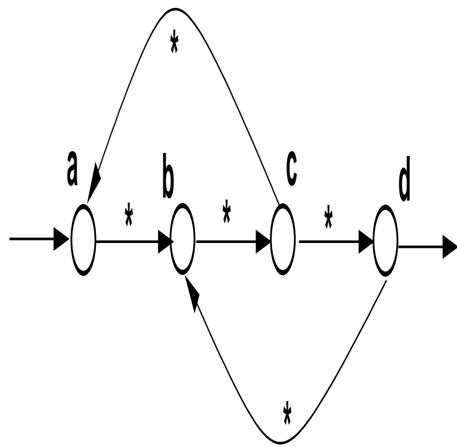
(a) Q_a: Vary X_L

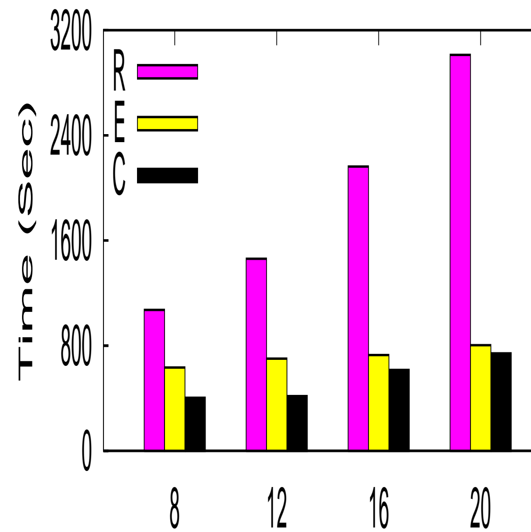


(b) Q_a: Vary X_R

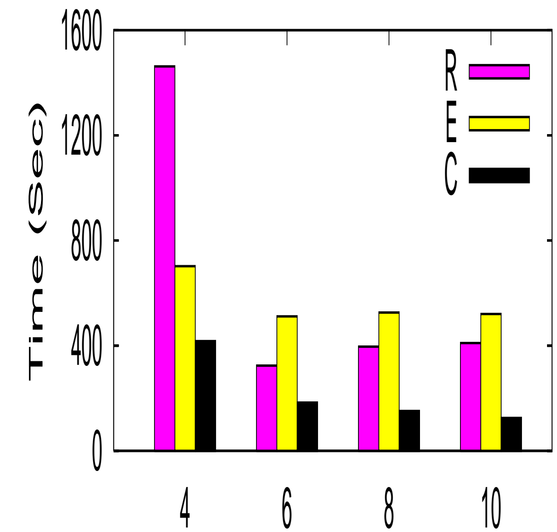
- $Q_a = a/b//c/d$ (with //)

Selected Testing Results (2)





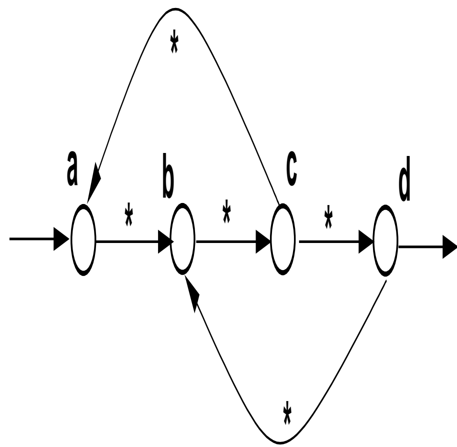
(a) Q_b: Vary X_L

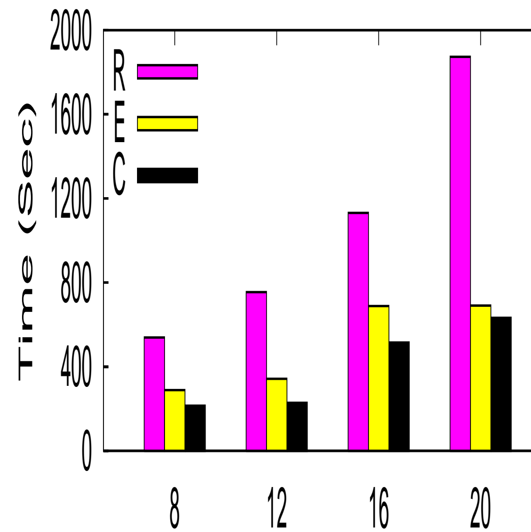


(b) Q_b: Vary X_R

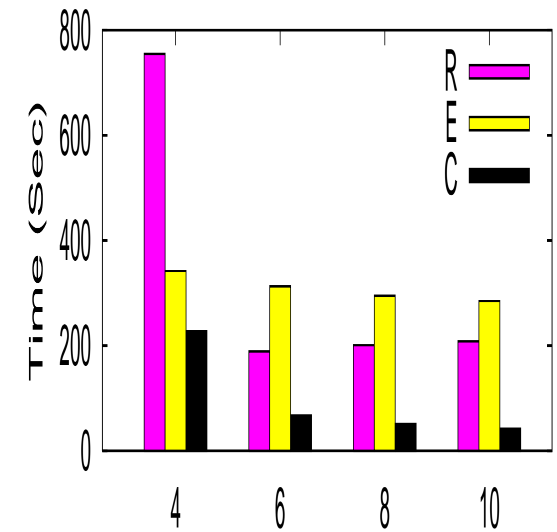
- $Q_b = a[//c]//d$ (a twig join query).

Selected Testing Results (3)





(a) Q_c: Vary X_L



(b) Q_c: Vary X_R

- $Q_c = a[\neg//c]$ (with ¬ and //).

Conclusion

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