Query translation from XPath to SQL in the presence of recursive DTDs

The VLDB Journal (2009) 18:857–883
DOI 10.1007/s00778-008-0131-0

REGULAR PAPER

Query translation from XPath to SQL in the presence of recursive DTDs

Wenfei Fan · Jeffrey Xu Yu · Jianzhong Li · Bolin Ding · Lu Qin

Received: 23 September 2007 / Revised: 17 November 2008 / Accepted: 17 November 2008 / Published online: 4 February 2009

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 (descendants-axis) 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 *extended XPath expressions* and a simple least fixpoint (LFP) operator. Extended 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 extended 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 over a (possibly recursive) DTD into extended XPath expressions and for translating extended XPath expressions to SQL queries, as well as optimization techniques. 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. In addition, these translation algorithms provide a solution to query answering for certain (possibly recursive) XML views of XML data. Our experimental results verify the effectiveness of our techniques.

Keywords XML database · XPath · SQL · Recursive DTD · Query translation

An extended abstract was presented at the 31st international conference on Very Large Data Bases (VLDB), 2005.

W. Fan (✉)
University of Edinburgh, Edinburgh, UK
e-mail: wenfei@inf.ed.ac.uk

W. Fan
Bell Laboratories, Madison, WT, USA
e-mail: wenfei@research.bell-labs.com

J. X. Yu · B. Ding · L. Qin
The Chinese University of Hong Kong, Hong Kong, China
e-mail: yu@se.cuhk.edu.hk

B. Ding
e-mail: blding@se.cuhk.edu.hk

L. Qin
e-mail: lqin@se.cuhk.edu.hk

J. Li
Harbin Institute of Technology, Harbin, China
e-mail: lijzh@hit.edu.cn

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 [59] as found in many commercial products (e.g., [33,49,52]). With this comes the need for answering XML queries using RDBMS, by translating XML queries to SQL.

The query translation problem can be stated as follows. Consider a mapping τ_d, defined in terms of DTD-based shredding, from XML documents conforming to a DTD D to relations of a schema $\mathcal{R}$. Given an XML query Q, we want to find (a sequence of) *equivalent* SQL queries Q' such that for any XML document T conforming to D, Q over T can be answered by Q' over the relation instance $\tau_d(T)$ of $\mathcal{R}$, i.e., $Q(T) = Q'(\tau_d(T))$. Here we allow DTDs D to be recursive and consider queries Q in XPath [15], which is the core of XML query languages XQuery and XSLT.

A closely related issue concerns query answering for XML views of XML data. Consider an XML view V of an XML document T. For practical reasons, e.g., XML access control [21] and data integration [44], the view V may necessarily be virtual and specified by a recursive DTD. To answer XPath queries Q posed on V without materializing V, one needs to rewrite Q into an equivalent XML query Q' on the underlying source T such that $Q(V) = Q'(T)$.

The query translation problem is, however, nontrivial: DTDs (or XML Schema) found in practice are often recursive [12] and complex. This is particularly evident in real-life applications (see, e.g., BIOML [10] and GedML [27], 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 [34] 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 the descendant-or-self axis '//' in XPath. Another approach is by means of a rich intermediate language and middleware as proposed in [57]: 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 [40], 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 [40], although there has been a host of work on storing and querying XML using an RDBMS [11,16,26,28,29,31,36,39,41,42,45,57,58,62,64], the problem of translating recursive XML queries into SQL has not been well studied in the presence of recursive DTDs, and it was singled out as the most important open problem for schema-based XML storage in [40].

Recently an elegant approach was proposed in [39] to translating path queries to SQL with the linear-recursion construct *with...recursive* of SQL'99. The algorithm of [39] is capable of translating path queries with // and limited qualifiers to (a sequence of) SQL queries with the SQL'99 recursion operator, handling XPath recursion and DTD recursion uniformly by means of product automata. Constraint-based techniques were also developed to optimize query translation [41,42]. 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 [39] are typically large and complex, and cannot be effectively optimized by all platforms supporting SQL'99 recursion for the same reasons that not all RDBMS can effectively optimize mildly complex non-recursive queries [26]. 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 [39] is too restricted to express XPath queries commonly found in practice. Finally, this approach does not help XPath query answering for XML views despite its analogy with XPath query translation to relational views.

There has also been a host of work on translating XML queries to SQL, for schema-oblivious XML storage, e.g., path-based methods [36,45,64], and region/Dewey encoding [11,16,31,62]. Combining path index and Dewey encoding, optimization techniques to minimize structural joins, e.g., Primitive Path Fragment (PPF) [28,29], have also been developed. Following this approach efficient XML query processors, such as MonetDB [11] and Saxon [55], have been developed on top of RDBMS, capable of processing XPath recursion without requiring the support of recursive operators by SQL. However, these methods typically store XML data in relations of a fixed schema, regardless of the schema of the XML data. This makes it difficult for, among other things, data exchange [38], XML access control (e.g., [21]) and XML view updates (e.g., [13]). Furthermore, the encoding and path index incurs additional overhead. Worse still, when the data is updated frequently, the cost of maintaining the encoding and path index could become prohibitively expensive. Moreover, in many applications one would prefer a lightweight tool that provides the capability of answering XPath queries within the immediate reach of commercial RDBMS, instead of using a heavy-duty system. In addition, the encoding and indexing approaches do not help when it comes to query answering over XML views.

In light of these, for schema-based XML storage, we propose a new approach to translating a class of XPath queries to SQL, which also provides a solution to query answering for certain XML views of XML data. The approach is based on a notion of *extended* XPath expressions and a simple least fixpoint (LFP) operator. Extended XPath expressions generalize XPath and regular XPath [48] by supporting variables and 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. Although theoretically the *with. . .recursive* operator can be encoded in terms of the LFP operator, the coding introduces additional overhead. The LFP operator is already supported by many commercial systems such as Oracle (*connectby*) and IBM DB2 (*with. . .recursion*), and is supported by Microsoft SQL server (*common table* [51]). We show that extended 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 variables and E^*, in polynomial time. That is, extended XPath expressions capture both DTD recursion and XPath recursion in a uniform and compact framework. Moreover, we show that each extended 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 on a (possibly recursive) DTD, we first rewrite Q into an extended XPath query E_Q that characterizes all matching paths, and then translate E_Q to an equivalent sequence Q' of SQL queries. To this end we provide an efficient algorithm for translating an XPath query over a (recursive) DTD D to an equivalent extended XPath query, and a novel algorithm for rewriting an extended XPath query into a sequence of SQL queries with the LFP operator. We show that the SQL queries are bounded by *a low polynomial* in the size of the input query Q and the DTD D. Furthermore, the translation algorithms effectively remove structural joins in the SQL queries by filtering out paths that Q does not match, based on the structural properties of the DTD. We also provide optimization techniques to speed up the processing of LFP computation.

Contributions. The main contributions of this paper include the following.

- The notion of extended 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 rewriting XPath queries over a (possibly recursive) DTD into extended XPath queries that characterize matching paths, based on dynamic programming.
- A novel algorithm for rewriting an extended XPath expression to a sequence of SQL queries with the LFP operator.
- Optimization techniques for speeding up the performance of LFP computation, and for eliminating unnecessary structural joins based on the properties of the input DTD.
- Experimental results verifying the effectiveness of our approach and techniques, using real-life XML DTDs.

Our approach has several salient features. (1) As will be seen in Sect. 3, the notion of extended XPath expressions yields a *low polynomial-time performance guarantee* on query translation from XPath to SQL; in contrast, direct use of XPath or regular XPath [48] in the translation may incur exponential blowup. Furthermore, this notion is also useful in developing native XML query engines [1,19]. (2) As opposed to prior work [11,39,62], our approach leads to a lightweight tool that provides a variety of commercial RDBMS with an immediate capability to answer XPath queries over recursive DTDs, requiring only low-end RDBMS features instead of the advanced SQL'99 recursion functionality. (3) It is capable of handling a class of XPath queries supporting child, self-or-descendants and union as well as rich qualifiers with data values, conjunction, disjunction and negation, which are beyond those studied in earlier proposals for schema-based XML storage at the SQL level. 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. (4) It produces SQL queries that are often 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 [54], recursive SQL query optimization [56] as well as integrity constraints [41,42]. (5) In contrast to [11,28,29,39,62], our approach provides also an effective solution to XPath query answering for certain XML views. As recently observed in [22], the query answering problem is nontrivial because XPath is not closed under query rewriting, i.e., for an XPath query Q posed on target XML data V, there may not exist an equivalent XPath query Q' on the underlying source such that $Q(V) = Q'(T)$; worse still, even if an equivalent XPath query Q' exists and when V is specified by a nonrecursive DTD, it takes exponential time to compute Q' in the size of Q. By leveraging extended XPath, our first translation algorithm, namely, the one from XPath to extended XPath, provides an effective solution to the query answering problem for a class of XML views.

Organization. The remainder of the paper is organized as follows. 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 [39], introduces extended XPath, and discusses its applications for answering XPath queries by using either RDBMS or native XML query engines. Section 4 provides an algorithm for translating XPath queries to extended XPath expressions, followed by an algorithm for rewriting extended XPath expressions into SQL with a simple LFP operator in Sect. 5. Experimental results are presented in Sect. 6, followed by related work in Sect. 7. Finally, Sect. 8 concludes the paper.

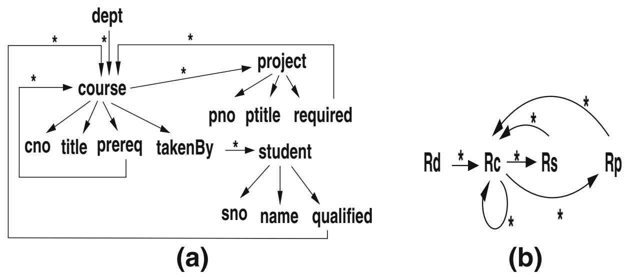


Fig. 1 A graph representation of the `dept` DTD

2 DTD, XPath, and schema-based shredding

We first 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 α:

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

where ϵ 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 $A \to Rg(A)$ 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 [59], 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 $A \to \alpha$, there is an edge from the A node to the B node for each subelement type B in α; the edge is labeled with '$*$' if B is enclosed in α_0^* for some sub-expression α_0 of α. This simple graph 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, where a simple cycle refers to a cycle in which no node appears more than once.

A DTD D is *contained* in another DTD D' if the DTD graph of D is a sub-graph of D', i.e., there is a homomorphism mapping from D to D' such that the root of D is mapped to the root of D'.

Example 2.1 We consider a `dept` DTD $(E, \texttt{dept}, Rg)$ as our running example, where E = {`course, cno, title, prereq, takenBy, project, student, sno, name, qualified, pno, ptitle, required`}, and Rg is defined as follows:

```
dept → course*           course → cno,
  title, prereq, takenBy, project*
prereq → course*         student → sno,
  name, qualified
takenBy → student*       project → pno,
  ptitle, required
qualified → course*      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 course may have several `projects`. A `student` has a `sno` (student number), a `name` and a list of `qualified` courses. Each `project` has a `pno` (project number), a `ptitle` (title) and required courses (`required`). Its DTD graph, a 3-cycle graph, is shown in Fig. 1a. □

2.2 XPath

We consider a class of XPath queries [15] that supports recursion (descendant-or-self), union and rich qualifiers, given as follows.

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

where ϵ, 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 a *qualifier*, defined as

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

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 $v[\![p]\!]$. In particular, $v[\![p_1[q]]\!]$ consists of nodes reachable via p_1 from v that satisfy the qualifier $[q]$. More specifically, a node v' satisfies the qualifier $[q]$ as follows: the atomic predicate $[p]$ holds at v' 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 $\emptyset$.

This class of XPath queries properly contains branching path queries studied in [39] and tree pattern queries (see, e.g., [4]). In the sequel, we refer to this class of queries simply as XPath.

Example 2.2 Consider two XPath queries.

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

Over an XML tree of the dept DTD of Fig.1, query Q_1 is to find all course-related projects, and Q_2 is to find courses that (1) have a prerequisite cs66, (2) have no project related to them or to their prerequisites, but (3) have no student who registered for the course and took cs66.

2.3 Mapping DTDs to a database schema

We next review shredding of XML data into relations. We consider a DTD-based approach since it is supported by most RDBMS [33,49,52]. To simplify the discussion, we focus on the shared-inlining technique of [59] although our query translation technique can be readily extended to work on most XML shredding methods for storing and querying XML data (see [40] for a comprehensive survey on XML shredding techniques). Extensions of our techniques to handle XML Schema instead of DTDs will be discussed in Sect. 8.

In a nutshell, the inlining algorithm partitions a DTD graph G_D into subgraphs, $G_1, G_2, \ldots$ 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 edge, 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 τ one can easily derive a *data mapping*, denoted by τ_d, from XML trees of D to instances of $\mathcal{R}$.

To simplify the discussion we assume that the mapping τ 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 (the value of all other attributes). Intuitively, in a database $\tau_d(T)$ representing an XML tree T, each R_A tuple (f, t, v) represents an edge in T from a node f to an A-element t which may have a text value v, where t and f are denoted by

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

F	T
R_d	
–	d_1
R_c	
d_1	c_1
c_1	c_2
c_2	c_3
p_1	c_4
s_2	c_5
R_s	
c_1	s_1
c_1	s_2
R_p	
c_2	p_1
c_4	p_2

the node IDs in 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 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. 1a is partitioned into four subgraphs rooted at *dept*, *course*, *project*, and *student*, respectively (see Fig. 1b). 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:

```
Rd(F, T)
Rc(F, T, cno, title, prereq, takenBy,
  parentCode)
Rs(F, T, sno, name, qualified)
Rp(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 τ_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 schema-based XML shredding, the relational schema $\mathcal{R}$ and the mapping τ_d are typically derived from the

DTD D, as opposed to being fixed as found in schema-oblivious XML storage.

In this section we first review the approach proposed by [39], the only solution published so far for the query translation problem in the presence of recursive DTDs. To overcome its limitations, we propose a new approach, which is based on the notion of extended XPath to handle the interaction between XPath recursion and DTD recursion in a uniform way. We introduce extended XPath and outline our query translation approach in this section; detailed translation algorithms will be presented in the next two sections. We also show that our algorithms also provides a solution to query answering for certain XML views of XML data.

3.1 Linear recursion of SQL'99

The algorithm of [39], 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 $\phi(R, R_1, R_2, \ldots R_k)$ with $k+1$ input relations, defined as follows:

$$\begin{aligned} &R^0 \leftarrow R \\ &R^i \leftarrow R^{i-1} \cup (R^{i-1} \bowtie_{C_1} R_1) \cup \cdots \cup (R^{i-1} \bowtie_{C_k} R_k) \end{aligned} \tag{1}$$

where R^0 corresponds to the initialization part, R_j corresponds to an SQL query coding an edge in c_i, and C_j indicates additional conditions associated with the join, for each $j \in [1, k]$.

Example 3.1 Recall the mapping given in Example 2.3 from the `dept` DTD to the relational schema $\mathcal{R}$ consisting of R_s, R_c, R_p, R_d, and the XPath query Q_1 = `dept//project` given in Example 2.2, which, over the DTD graph of Fig. 1b, indicates $R_d//R_p$. Given Q_1 and the DTD graph of Fig. 1b, 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.1b 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.

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

Fig. 2 The SQL statement generated by SQLGen-R

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 Sect. 1, $\phi(R, R_1, R_2, \ldots, R_k)$ is a high-end feature that few RDBMS support. Although theoretically one can encode this in terms of an LFP $\Phi(R)$ that takes a single input relation and is supported by most commercial RDBMS, the coding introduces space overhead. 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.

Table 2 An output of SQLGen-R at each iteration

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’

3.2 Extended XPath expressions

To overcome the limitations of the previous approach, we propose a new approach to translating XPath queries to SQL. In a nutshell, given an XPath query Q and a DTD D, we first rewrite Q to an expression Q' that captures all matching paths of Q in D. We then translate the expression into an equivalence sequence of SQL queries.

We want Q' to specify all and only those matching paths of Q in D, for the following reasons. First, to avoid unnecessary structural joins in the ultimate SQL queries, Q' should eliminate those paths that do not match Q in D. Second, Q' should be generic enough to work with various XML shredding mappings τ_d. More specifically, suppose that an XML tree T that conforms to D is stored in relations $\tau_d(T)$. Here T can be considered an XML view of relations $\tau_d(T)$, and depending on how τ_d is defined, the “source” $\tau_d(T)$ may not adhere to the same DTD D (note that relations are a special case of XML data). As will be seen in Sect. 3.4, we often need to deal with cases when view DTDs are contained in the corresponding source DTDs. To answer Q by using Q' and $\tau_d(T)$, Q' should faithfully capture the matching paths of Q in D. The need for this is more evident when it comes to query answering using XML views.

One might be tempted to assume that one could also express Q' in XPath. Unfortunately, there may not exist any XPath query that precisely enumerates all matching paths of Q in D when D is recursive [22]. To illustrate this, let us consider the following example.

Example 3.2 Consider recursive DTDs D, D' depicted in Fig. 3a, b, respectively. Note that D is contained in D', where D' has an additional edge (B, C). One can easily define a mapping σ from instances of D' to instances of D such that for any document T that conforms to D', from σ an XML document V can be derived such that V conforms to D. Now consider a query $Q = //\epsilon$ posed on V that is to find all nodes in V. Suppose that we want to compute an equivalent query Q' that, when posed on T, returns the same result as Q on V. Clearly, Q' should not return any C nodes that are children of some B nodes. Then one can verify that Q' is not expressible in the XPath fragment given above. Indeed, as D is recursively defined, Q' necessarily contains ‘//’, which returns all descendants of a context node. Within ‘//’, however, the XPath fragment is not expressive enough to specify any pattern such that any C children of B nodes are excluded from the answer to the query.

Worse still, even when we allow Q' to be in regular XPath proposed by [48] and D is nonrecursive, the rewriting may still be beyond reach in practice since it may incur exponential blowup. 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. Although regular XPath is more expressive than XPath, it is still not rich enough for query rewriting. The next example is taken from [18], which shows that representing a nondeterministic finite state automaton with a regular expression takes at least exponential time, even if the automaton is non-recursive.

Example 3.3 Consider a DTD D_1 for which the DTD graph consists of (a) nodes A_i for $i \in [1, n]$, where the root is A_1, (b) edges (A_i, A_j) for all $i, j \in [1, n]$ and $i < j$. Figure 3c shows such a DTD graph for $n = 4$. This DTD is contained in another DTD D_2, shown in Fig. 3d, which, in addition, has a node B and moreover, edges (B, A_n) and (A_i, B) for $i < n$. Note that these DTD graphs are acyclic, i.e., they are non-recursive. There is a natural mapping σ_0 from instances of D_1 to instances of D_2 such that for any document T that conforms to D_2, from σ_0 an XML document V can be derived such that V conforms to D_1 and moreover, (a) the root r_v of V maps to the root r_t of T, and (b) for any element u in V that is reached from r_v via a path ρ, it is mapped to an element $\sigma_0(u)$ that is reachable from r_t via the same path ρ.

Now consider a query $Q = //A_n$ posed on V that is to find all A_n nodes in V. Suppose that we want to find an equivalent query Q' that, when posed on T, returns the same result as Q on V. The query Q' is then to find all A_n nodes in T that are reachable from the root without going through any B node in T. One can verify that although query Q' is expressible in regular XPath, it takes necessarily $O(2^n)$ space. Indeed, a regular XPath expression of Q' will be given in Example 3.4, and a straightforward combinatorial analysis suffices to tell us that the size of the expression is of $O(2^n)$, from which the $O(2^n)$ space bound follows immediately. The example is borrowed from [18].

These suggest that we further extend XPath and regular XPath to translate XPath queries over a (possibly recursive) DTD to SQL. Below we introduce such an extension, referred

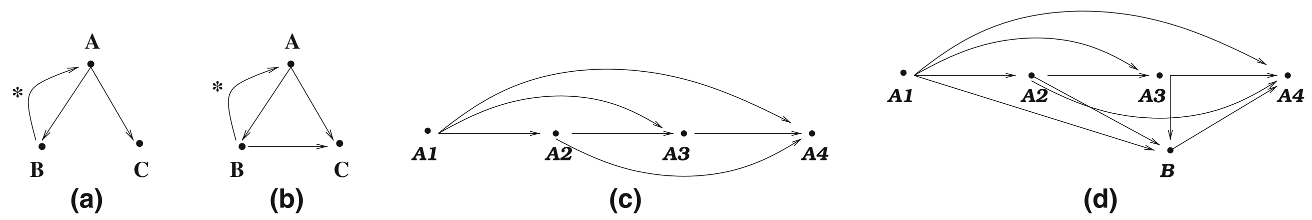


Fig. 3 Example DTD graphs

to as *extended* XPath *expressions* and syntactically defined as follows:

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

where X is a variable, and E^* denotes the Kleene closure of E.

Observe that an expression E without any variable is a query in regular XPath. 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. The semantics of evaluating E over an XML tree is similar to its XPath counterpart.

An *extended* XPath *query* Q is a sequence of equations of the form $X_i = E_i$, where for $i \in [1, k]$, X_i is a variable, E_i is an extended XPath expression, and X_i does not appear in E_j if $i < j$. Intuitively, the equations specify bindings of variables and sub-queries. It can be easily verified that Q is equivalent to a sequence of equations of the form $X_i = E'_i$, where E'_i is a regular XPath query, i.e., an extended XPath expression without variables. The semantics of evaluating Q over an XML tree is therefore straightforward: for i from k downward to 1, evaluate E_i and substitute the result of E_i for X_{i-1} in E_{i-1}.

The use of variables in extended XPath allows us to represent (possibly infinite) matching paths in polynomial that would otherwise take exponential time in regular XPath and would not be expressible in XPath.

Example 3.4 Recall the query $//\epsilon$ from Example 3.2, posed on XML trees of DTD D. Over DTD D', it can be readily translated to an equivalent query $(A/B)^*(\epsilon \cup A \cup A/C)$, in extended XPath. This is also in regular XPath.

The query Q of Example 3.3 posed over D_1 can be rewritten to query Q' over D_2 as follows.

$$X_{(1,4)} = A_1/(A_4 \cup A_2)/X_{(2,4)}$$
$$X_{(2,4)} = A_1/(A_4 \cup A_3)/X_{(3,4)}$$
$$X_{(3,4)} = A_1/A_4$$

This is in extended XPath, but in neither XPath nor regular XPath. Obviously Q' can be computed in polynomial time (and polynomial space), without paying the price of exponential space as required by any equivalent regular XPath expressions. This motivates us to use extended XPath instead of regular XPath and XPath.

3.3 A new approach

Based on a notion of extended XPath expressions and the simple LFP operator $\Phi(R)$, we propose a new approach to translating XPath queries to SQL. Below we first review the simple LFP operator. We then outline our approach.

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

$$R^0 \leftarrow R$$
$$R^i \leftarrow R^{i-1} \cup (R^{i-1} \bowtie_C R^0) \tag{2}$$

where C is a condition associated with the join. This LFP operator is already supported by most commercial products, e.g., by Oracle and IBM DB2 are shown in Fig. 4.

To illustrate how the LFP operator handles Kleene closure, consider an extended XPath expression $(A_1/\cdots/A_n)^*$ representing a simple cycle $A_1 \to \cdots \to A_n \to A_1$. This simple extended XPath expression can be rewritten into $\Phi(R)$ (Eq. (2)) by letting $R \leftarrow \Pi_{R_1.F, R_n.T}(R_1 \bowtie R_2 \bowtie \cdots \bowtie R_n)$, where 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 that encodes E.

A new approach for query translation. We propose a new framework for translating XPath to SQL that, as depicted in Fig. 5, translates an input XPath query Q to SQL in two steps. First, it rewrites Q over a (possibly recursive) DTD D to

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

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

Fig. 4 Implementation of LFP in Oracle and DB2

an equivalent extended XPath query E_Q over any DTD D' that contains D, i.e., the DTD graph of D is a subgraph of the DTD graph of D'. The query E_Q has the form $(X_1 = E_1, \ldots, X_k = E_k)$ as mentioned above. 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. The choice of extended XPath in the first step is motivated by the following reasons. As remarked earlier, the Kleene closure of extended XPath allows us to instantiate '//' of XPath, and capture recursion in XPath and DTD recursion in a uniform framework. Furthermore, as illustrated in Example 3.4, the use of variables allows us to extract common sub-queries and thus avoid the exponential lower bound of translation to regular XPath.

In contrast to the approach of [39], this framework introduces more opportunities for optimization, as illustrated by the example below.

Example 3.5 Let us consider again evaluating the XPath query $Q_1 = \texttt{dept//project}$ over the dept DTD of Fig. 1, in the same setting as in Example 3.1. Our translation algorithms first translate Q_1 to an extended XPath query $E_{Q_1} = (X_{Q_1} = R_d/R_c/X^*/R_p,\ X = R_c \cup R_s/R_c \cup R_p/R_c)$. It then rewrites E_{Q_1} to a sequence of SQL queries (written in relational algebra):

$$R_{cc} \leftarrow R_c$$

$$R_{csc} \leftarrow \Pi_{R_s.F, R_c.T}(R_s \bowtie_{R_s.T = R_c.F} R_c)$$

$$R_{cpc} \leftarrow \Pi_{R_p.F, R_c.T}(R_p \bowtie_{R_p.T = R_c.F} R_c)$$

$$R \leftarrow R_{cc} \cup R_{csc} \cup R_{cpc}$$

$$R_\gamma \leftarrow \Phi(R) \cup \Pi_{T,T}(R_c)$$

$$R_f \leftarrow \Pi_{R_d.T, R_p.T}(R_d \bowtie_{R_d.T = R_c.F} R_c \bowtie_{R_c.T = R_\gamma.F}$$
$$R_\gamma \bowtie_{R_\gamma.T = R_p.F} R_p)$$

The above SQL sequence is the output of our algorithms. Contrast this with Example 3.5. 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 LFP operator, along the same lines as the LFP optimization by distribution of selections suggested by [3], as will be illustrated in Sect. 5.

In Sect. 4 we present a translation algorithm to show that every XPath query Q over a (recursive) DTD D can be rewritten to an extended XPath query E_Q that is equivalent to Q over all DTDs containing D. Then, we provide another algorithm in Sect. 5 to show that the simple LFP operator $\Phi(R)$ suffices to handle general Kleene closure in an extended XPath query E_Q.

3.4 More on extended XPath: query answering using XML views

The notion of extended XPath is useful not only in translating XPath to SQL, but also in developing native XML engines for evaluating XML queries. Indeed, regular XPath, a special form of extended XPath, is being rapidly introduced into XML engines [1]. EXSLT [19], for example, supports a transitive closure operator `dyn:closure` that implements essentially regular XPath. Furthermore, at least some versions of Saxon [55] also support this operator. Extended XPath provides a more succinct form of regular XPath and is naturally expected to help in this line of work as well.

What we have seen so far concerns answering XPath queries posed on XML data that is stored in relations. This can be considered as answering XPath queries over certain XML views of relational data. One step further, it is natural to consider answering XPath queries posed on XML views of XML data. In this context, extended XPath also finds applications. Indeed, in addition to optimization opportunities, the first step of our translation framework given earlier in fact provides a solution to query answering for certain XML views of XML data.

Consider a class of GAV mappings $\sigma : D_1 \to D_2$, where D_1, D_2 are target and source DTDs, respectively (see, e.g., [32, 44] for GAV mappings), such that for any instance T of D_2, σ defines a view V such that for any XML element $v \in V$, $\sigma(v)$ is contained in an XML element u in T, and V is a sub-structure of T. For instance, the mapping σ_0 given in Example 3.3 is such a mapping from the DTD of Fig. 3c to the DTD of Fig. 3c. Such views are found in many applications, e.g., access control for XML [21] where one only wants to reveal parts of T to authorized users, or data integration [44] where part of the source is migrated to the target. In these applications V is often virtual.

Now consider an XPath query posed on V. We want to answer the query without materializing V. This highlights the need for a *query answering* algorithm that, given an XPath query Q posed on D_1, effectively computes a query Q' on D_2 such that $Q(V) = Q'(T)$.

This query answering problem is, however, nontrivial. Indeed, consider query Q given in Example 3.3, which is

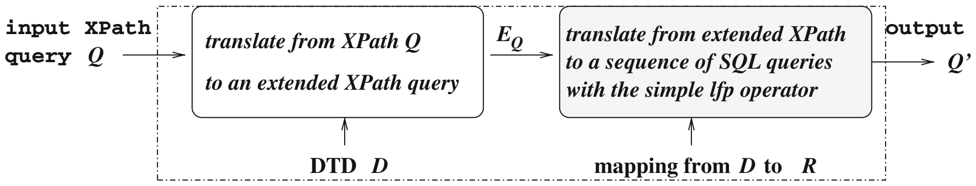

input XPath query Q
translate from XPath Q to an extended XPath query
E_Q
translate from extended XPath to a sequence of SQL queries with the simple lfp operator
output Q'
DTD D
mapping from D to R

Fig. 5 Translation from XPath to SQL

posed on the view V. The equivalent query Q' on the source T is to find all A_n nodes reachable from A_1 without going through any B nodes. As shown in Example 3.3, although this query is definable in regular XPath, it is necessarily of exponential size. As recently observed in [22], XPath is not closed under query rewriting and although regular XPath is closed, it incurs an exponential lower bound for rewritten queries.

In contrast, we show that the translation algorithm of the step 1 of our framework provides a solution to the query answering problem for the class of GAV XML views described above. Indeed, given any XPath query Q posed on D_1, the algorithm effectively computes Q' in extended XPath in *polynomial time*. Furthermore, the query Q' has the following property. For any D_2 that contains D_1, define a mapping σ from instances of D_1 to instances of D_2 that, given any instance T of D_2, derives an instance V of D_1 such that V is a subtree of T: σ maps the root r_v of V to the root r_t of T, each A element v of V to an A element $\sigma(v)$ of T such that if v is reached from r_v via a path ρ, then $\sigma(v)$ can reached from r_t via the same path ρ. Then $Q(V) = Q'(T)$.

Finally, we give some notations, which will be used in Sect. 4. We say that Q' is *equivalent to* Q over all DTDs D_2 that contain D_1. Furthermore, we say that Q' is *equivalent to Q w.r.t. two element types A, B* over all DTDs D_2 that contains D_1 if when evaluated at any A element v in V, the set of B elements returned by Q is the same as the set of B elements returned by Q' when evaluated at $\sigma(v)$ in T.

4 From XPath to extended XPath

In this section, we present an algorithm for rewriting an XPath query Q over a (recursive) DTD D to an extended XPath query E_Q that is equivalent to Q' over all DTDs containing D. We first develop an algorithm for handling the descendant-axis ('//') of XPath, and then give the translation algorithm for the XPath fragment defined in Sect. 2.

4.1 Translation of the descendant axis

Consider an XPath query $Q = A//B$ over a DTD D. The query, when evaluated at an A-element v in an instance V of D, is to find all B descendants of v. We want to find an extended XPath query Q', denoted by $\mathsf{rec}(A, B)$, that is equivalent to Q over all DTDs that contain D.

An algorithm is given by Tarjan in [61] that, given a graph G_D and two nodes $A, B \in G_D$, finds a regular expression which represents the set of all paths in G_D from A to B. We sketch Tarjan's algorithm [61] in Fig. 6, and denote it by CycleE as it is based on cycle expansion. Let $G_D = (N, E)$, where N is the set of nodes and E is the set of edges of G_D. Following the notations of [61], we associate the nodes in G_D with numbers from 1 to $n = |N|$, and use a regular expression $M[i, j, k]$ to maintain all possible paths from node i to node j via nodes whose numbers are less than or equal to k, where k can be zero indicating a "path" via no nodes in G_D. Algorithm CycleE first initializes all $M[i, j, 0]$ (line 1-7). It then expands $M[i, j, k]$ for all i, j by incrementing k, i.e., by inspecting $M[i, k-1, k]$ and $M[k, j, k]$ while including all possible cycles, i.e., $M[k, k, k-1]^*$, at node k (lines 8–13).

The regular expression $M[1, n, n]$ returned by Algorithm CycleE precisely represents all paths from node A to node B, where A and B are numbered 1 and n, respectively. Unfortunately, the algorithm takes exponential time and exponential space.

Lemma 4.1 *Given a graph G_D with n nodes and nodes A, B in G_D, CycleE finds a regular expression capturing all paths in G_D from node A to B in $\Theta(n^3 2^n)$-time and $\Theta(n^2 2^n)$-space.*

Proof The correctness of algorithm was verified in [61]. The upper bound is due to line 12 in CycleE (Fig. 6), which copies sub-expressions and concatenates them into one. In fact, it is already shown in [18] that the bounds given above are also the lower bounds for converting nondeterministic finite state (NFA) to regular expressions; as a result, when the graph D_G is treated as a NFA with A as the start state and B the final state, any regular expressions characterizing the NFA have the lower bounds given above. □

Recall the definition of extended XPath expression and extended XPath query. We show that one can find a query $\mathsf{rec}(A, B)$ representing all paths from A to B in *low polynomial time*. Indeed, we present a mildly modified CycleE, denoted by $\mathsf{CycleE_X}$, that computes $\mathsf{rec}(A, B)$. The algorithm uses the following variables. (a) $M[i, j, k]$ is an extended XPath expression representing all possible paths from node i to node j via nodes no larger than k. (b) Variable $X[i, j, k]$ indicates equation $X[i, j, k] = M[i, j, k]$ in the output. (c) $X[k, k, k-1]^*$ indicates equation $X[k, k, k-1] = cycle(M[k, k, k-1])$, which represents cycles at node k. The

Fig. 6 CycleE (Tarjan's Algorithm for finding regular expressions)

```
Algorithm CycleE($G_D, A, B$)
Input: a graph $G_D$ and two nodes $A$ and $B$ in $G_D$.
output: a regular expression representing all paths from $A$ to $B$ in $G_D$.
1.  for $i = 1$ to $n$ do
2.      for $j = 1$ to $n$ do
3.          if $i = j$
4.          then $M[i,j,0] := \emptyset$;
5.          else if $i \neq j$ and $(i,j) \in E(G_D)$
6.              then $M[i,j,0] := i/j$;
7.              else $M[i,j,0] := \emptyset$;
8.  for $k = 1$ to $n$ do
9.      for $i = 1$ to $n$ do
10.         for $j = 1$ to $n$ do
11.             if $M[i,k,k-1] \neq \emptyset$ and $M[k,j,k-1] \neq \emptyset$
12.             then $M[i,j,k] := (M[i,j,k-1]) \cup (M[i,k,k-1]/M[k,k,k-1]^*/M[k,j,k-1])$;
13.             else $M[i,j,k] := M[i,j,k-1]$;
14. return $M[A,B,n]$;
```

Fig. 7 CycleE_X for extended XPath expressions

```
Algorithm CycleE_X($G_D, A, B$)
Input: a DTD graph $G_D$ and two nodes $A$ and $B$ in $G_D$.
output: an extended XPath query rec($A, B$) representing all paths from $A$ to $B$ in $G_D$.
1.  for $i = 1$ to $n$ do
2.      for $j = 1$ to $n$ do
3.          if $i = j$
4.          then $M[i,j,0] := \emptyset$;
5.          else if $i \neq j$ and $(i,j) \in E(G_D)$
6.              then $M[i,j,0] := i/j$;
7.              else $M[i,j,0] := \emptyset$;
8.  for $k = 1$ to $n$ do
9.      for $i = 1$ to $n$ do
10.         for $j = 1$ to $n$ do
11.             if $M[i,k,k-1] \neq \emptyset$ and $M[k,j,k-1] \neq \emptyset$
12.             then $M[i,j,k] :=$ '$X[i,j,k-1] \cup X[i,k,k-1]/X[k,k,k-1]^*/X[k,j,k-1]$';
13.             else $M[i,j,k] :=$ '$X[i,j,k-1]$';
14. rec($A,B$) := $\{(X[i,j,k] = M[i,j,k],$ where $M[i,j,k] \neq \emptyset \mid i,j,k \in [0,n]\}$;
15. optimize rec($A,B$) by removing redundant equations and return rec($A,B$).
```

algorithm is shown in Fig. 7. The initialization part (lines 1–7) is the same as its counterpart in CycleE. In contrast to CycleE, $M[i, j, k]$ is represented as an expression (string) with only at most four operators and four variables rather than concatenating four expressions, by capitalizing on variables (lines 8–13). The length of each $M[i, j, k]$ is thus constant. Finally, we construct extended XPath query rec(A, B) by listing equations (i.e., $X_i = E_i$, where X_i is a variable and E is an extended XPath expression) $X[i, j, k] = M[i, j, k]$ in the order of k, and return the whole ordered set rec(A, B) as the output, where variable $X[A, B, n]$ represents the final result (lines 14–15). In line 15, the following redundant equations are pruned from rec(A, B): 1) $X[i, j, k] = \emptyset$; 2) $X[i, j, k] =$ '$X[i', j', k']$'; and 3) the variables that do not contribute to processing the variable $X[A, B, n]$.

Example 4.1 Consider again the query $//A_n$ on the DTD graph D_1 of Fig. 3c. Starting from A_1 and ending with A_4 ($A_1//A_n$), for $n = 4$, CycleE_X returns an extended XPath query as follows (suppose nodes A_1, A_2, A_3, and A_4 in D_1 are associated with numbers 1, 2, 3, and 4, respectively):

$$X[1, 3, 2] = \text{'}X[1, 3, 1] \cup X[1, 2, 1]/X[2, 3, 1]\text{'}, \quad (3)$$

$$X[1, 4, 2] = \text{'}X[1, 4, 1] \cup X[1, 2, 1]/X[2, 4, 1]\text{'}, \quad (4)$$

$$X[1, 4, 4] = \text{'}X[1, 4, 2] \cup X[1, 3, 2]/X[3, 4, 2]\text{'}, \quad (5)$$

where $X[1, 2, 1] = 1/2$, $X[1, 3, 1] = 1/3$, $X[1, 4, 1] = 1/4$, $X[2, 3, 1] = 2/3$, $X[2, 4, 1] = 2/4$, and $X[3, 4, 2] = 3/4$.[1] The output of CycleE_X produces an extended XPath query that contains 3 "$\cup$"-operators and 6 "/"-operators. Note: the "/" appearing in Eqs. (3), (4), and (5) is used to concatenate two variables, and is not a "/"-operator in the extended XPath expression. For example, consider '$X[1, 2, 1]/X[2, 3, 1]$' [Eq. (3)]. Here, $X[1, 2, 1] = 1/2$ indicates a path from 1

[1] Based on the pruning rule of (2), the $X[1, 2, 1] =$ '$X[1, 2, 0]$' is pruned and $X[1, 2, 1] = 1/2$ produced by Algorithm CycleE_X is used. The equation of $X[2, 4, 3] =$ '$X[2, 4, 2] \cup X[2, 3, 2]/X[3, 4, 2]$' will be pruned following the pruning rule of (3), because it does not contribute to the processing of $X[1, 4, 4]$.

to 2, and $X[2, 3, 1] = 2/3$ indicates a path from 2 to 3. Node 2 appears in both $X[1, 2, 1]$ and $X[2, 3, 1]$, and concatenates the two variables. Similarly, $X[1, 2, 1]/X[2, 3, 1]$ indicates an extended XPath expression $1/2/3$.

In contrast, CycleE gives an extended XPath expression of $X[1, 4, 4] = 1/4 \cup 1/2/4 \cup (1/3 \cup 1/2/3)/4$ with 3 "$\cup$"-operators and 7 "/"-operators.

Example 4.1 illustrates how $\mathsf{CycleE_X}$ works. Next, we also show $\mathsf{CycleE_X}$ (polynomial) outperforms CycleE (exponential), in terms of the number the '/'-operators.

Example 4.2 Consider again the query $//A_n$ on the DTD graph D_1 of Fig. 3c. Starting from A_1 and ending with A_n ($A_1//A_n$), $\mathsf{CycleE_X}$ returns an extended XPath query (a list of equations) with $\Theta(n^2)$ '/'-operators, while CycleE gives an extended XPath expression with $\Omega(2^n)$ "/"-operators. Indeed, when $\mathsf{CycleE_X}$ is used, only one "/"-operator, appearing on the right side of the equation for $X[1, i, j]$, where $1 \le j < i \le n$, will be executed. In total, there are $n \cdot (n-1)/2 \in \Theta(n^2)$ "/"-operators. Consider CycleE. Let $f(n)$ be the number of "/"-operators in the output of CycleE on the input DTD graph D_n. We can establish the following recursive relationship from line 12 in Algorithm CycleE:

$$\begin{aligned} f(2) &= 1;\\ f(3) &= f(2) + 2;\\ f(4) &= f(2) + f(3) + 3;\\ &\cdots\ \cdots\ \cdots\\ f(n) &= f(2) + f(3) + \cdots + f(n-1) + (n-1); \end{aligned}$$

Thus $f(n) = n + (n-1) + 2\cdot(n-2) + 2^2\cdot(n-3) + \cdots + 2^{i-1}\cdot(n-i) + \cdots + 2^{n-4}\cdot 3 \in \Omega(2^n)$.

Moreover, in contrast to CycleE, Algorithm $\mathsf{CycleE_X}$ has the following nice properties.

Theorem 4.1 *Given a DTD D with n element types and element types A, B in D, $\mathsf{CycleE_X}$ finds a query* rec(A, B) *in extended XPath in $O(n^3 log\ n)$ time, and moreover, when evaluated at any A-element,* rec(A, B) *is equivalent to $//B$ for all DTDs that contain D.*

Proof For the complexity, $\mathsf{CycleE_X}$ computes at most $n^3 + n$ equations, and each equation is of $O(log\ n)$ size (for encoding the four variables, and there are $n^3 + n$ variables in total). Each step of the inner-most loop takes at most $O(log\ n)$ time. From this the complexity bound follows.

We next show that rec(A, B) is equivalent to $//B$ for all DTDs that contain D. More specifically, let D' be an arbitrary DTD that contains D, and σ be the mapping that, given any instance T of D', extracts a subtree V of T that is an instance of D, as specified in Sect. 3. Let v be an A element in V, where σ maps v to an A element $\sigma(v)$ in T. We need to show that a node u' can be reached from $\sigma(v)$ via rec(A, B) in T iff there exists a node u in V such that $u' = \sigma(u)$ and u can be reached from v via rec(A, B) in V.

To show this, first observe that a regular expression $\mathcal{X}(A, B)$ can be derived from rec(A, B) by removing variables as described in Sect. 2. We claim the following: ρ is a path from A to B in D iff ρ is a word in $\mathcal{X}(A, B)$. This can be easily verified by showing ρ is a path from A to B without going through any node larger than k iff $\rho \in M[A, B, k]$, by induction on k.

This claim suffices. Indeed, for any XML tree T and any A element $\sigma(v)$ in T, a node u' can be reached from $\sigma(v)$ in T via rec(A, B) iff the path ρ from $\sigma(v)$ to u' in T is a word in $\mathcal{X}(A, B)$. By the claim and the definition of σ, this happens iff ρ is in D and there exists a node u in V such that $u' = \sigma(u)$ and u can be reached from v via the same path ρ in V. □

4.2 Translation algorithm

We next present an algorithm for translating XPath queries of the fragment of Sect. 2 over a DTD D to extended XPath queries that are equivalent over all DTDs that contain D.

The algorithm, referred to as XPathToEXp, is based on *dynamic programming*: for each sub-query p of the input query Q and each pair of element types A, B in D, it computes a local translation from XPath p to an equation $X_p(A, B) =$ x2e(p, A, B), where $X_p(A, B)$ is a variable and x2e(p, A, B) is an extended XPath expression, such that x2e(p, A, B) is equivalent to p w.r.t. A and B over any DTD D' that contains D (recall the notion from Sect. 3). Composing the local translations one will get the rewriting $E_Q = \cup_{B \in D} X_Q(r, B)$ from Q, where r is the root type of D. For each local translation x2e(p, A, B) the algorithm "evaluates" p over the sub-graph of the DTD graph G_D rooted at A, substituting extended XPath expressions over element types for wildcard $*$ and descendants //, by incorporating the structure of the DTD into x2e(p, A, B). 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, just by checking the structure of D.

To conduct the dynamic-programming computation, Algorithm XPathToEXp uses the following variables. First, it works over a list L that is a postorder enumeration of the nodes in the parse tree of Q, such that all sub-queries of a sub-query p (i.e., its descendants of p in Q'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 pair of types A, B in N, we use x2e(p, A, B) to denote the *local translation* of p at A, which is an extended XPath expression. Furthermore, we use a variable $X_p(A, B)$ which will be used in the equation $X_p(A, B) =$ x2e(p, A, B), such that $X_p(A, B)$ can be used instead of x2e(p, A, B) whenever the latter is needed. Finally, 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 $\mathsf{x2e}([q], A, A)$ is $[\epsilon]$.

Algorithm XPathToEXp is given in Fig. 8. It computes E_Q 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 $\mathsf{x2e}(p, A)$ to the special query $\emptyset$ and $\mathsf{reach}(p, A)$ to empty set for each $p \in Q$ and $A \in N$ (lines 1–5). Then, for each sub-query p in L in the topological order and each element type A in N, it computes the local translation $\mathsf{x2e}(p, A, B)$ (lines 6–28), bottom-up starting from the inner-most sub-query of Q. To do so, it first computes $\mathsf{x2e}(p_i, B_j, B)$ 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 $\mathsf{x2e}(p_i, B_j, B)$'s to get $\mathsf{x2e}(p, A, B)$. The details of this combination are determined based on the formation of p from its immediate sub-queries p_i, if any (cases 1–7). These cases are illustrated as follows.

First, when p is empty path ϵ, element type C, wildcard $*$ or descendants-or-self //, namely, cases (1)–(3) and (5), $\mathsf{x2e}(p, A, B)$ and $\mathsf{reach}(p, A)$ are determined by the DTD D alone regardless of the input query Q; thus it can be pre-computed for each A, B, once and for all, and made available to XPathToEXp. We include these cases [cases (1)–(3)] in Fig. 8 for ease of reference (lines 9–14).

When $p = p_1/p_2$ [case (4)], for each C reached via p_1 from A, XPathToEXp assembles $\mathsf{x2e}(p_1, A, C)$ and $\mathsf{x2e}(p_2, C, B)$ for each B reached via p_2 from C to precisely represent paths from A to B in D. Furthermore, variables $X_{p_1}(A, C)$ and $X_{p_2}(C, B)$ are used instead of $\mathsf{x2e}(p_1, A, C)$ and $\mathsf{x2e}(p_2, C, B)$ to avoid that $\mathsf{x2e}(p, A, B)$ has an exponential size.

Similarly, in the case $p = \epsilon//p_1$ [case (5)], XPathToEXp assembles $\mathsf{x2e}(p_1, A, C)$ and $\mathsf{x2e}(p_2, C, B)$ for each C reached via $\epsilon//$ and each B reached via p_1 from C. Here $\mathsf{x2e}(\epsilon, A, C)$ is essentially $X[A, C, n]$, the variable in $\mathsf{rec}(A, C)$ representing the final result. Note that $\mathsf{rec}(A, C)$ is precomputed by Algorithm $\mathsf{CycleE_X}$, and is also added into E_Q in line 31.

When $p = p_1 \cup p_2$ [case (6)], for each B in D, XPathTo EXp simply computes $\mathsf{x2e}(p_1, A, B) \cup \mathsf{x2e}(p_2, A, B)$. However, for the same reason given for case (4), $X_{p_1}(A, B)$ and $X_{p_2}(A, B)$ are used instead of $\mathsf{x2e}(p_1, A, B)$ and $\mathsf{x2e}(p_2, A, B)$

When p comes with a qualifier, i.e., when $p = p'[q]$ [case (7)], it invokes a procedure RewQual to translate $[q]$. Procedure RewQual may evaluate $[q]$ to a truth value (ϵ for *true* and $\emptyset$ for *false*) in certain cases based on the structure of the DTD D alone. If so, XPathToEXp simply drops $[q]$ in $\mathsf{x2e}(p, A, B)$, or leaves $\mathsf{x2e}(p, A, B)$ unchanged (i.e., $\emptyset$) if RewQual returns *false*.

At the end of the iteration, each $\mathsf{x2e}(p, A, B)$ is optimized by removing $\emptyset$, which returns an empty set over any XML tree, as described in Sect. 2. Finally, X_Q is defined to be the union of $\mathsf{x2e}(Q, r, B)$ for all $B \in \mathsf{reach}(Q, r)$, and the equations of the extended XPath query are put together into E_Q as the output of the algorithm (lines 29–32).

Procedure RewQual is shown in Fig. 9. It translates qualifiers $[q]$ into an extended XPath query, based on the structure of q. Furthermore, it "evaluates" q over the DTD and gets the truth value of $[q]$ if it can be determined based on the DTD structure alone. For example, when q is a path p, it concludes that $[q]$ is *false* if no node can be reached from A via p, and *true* if ϵ is contained in p, i.e., the current node is in the "result" of the query p. For Boolean operations, it invokes procedure optimize (not shown) that determines whether $q_1 \wedge q_2$, $q_1 \vee q_2$ and $\neg q_1$ can be evaluated to *true* or *false*. For $q_1 \wedge q_2$, for example, optimize evaluates it to *true* if both q_1 and q_2 are ϵ, and to *false* if one of q_1 and q_2 is $\emptyset$; similarly for $q_1 \vee q_2$ and $\neg q_1$.

Example 4.3 Recall the XPath query Q_2 from Example 2.2. Observe that the algorithm of [39] *cannot* handle this query over the *dept* DTD of Fig. 1a. In contrast, XPathToEXp translates Q_2 to the extended XPath query E_{Q_2} below:

$$\begin{aligned} X_{Q_2} = {} & dept/course[X_{course\_course}/prereq/course[cno = \\ & \text{“}cs66\text{''}] \wedge \neg X_{course\_project} \wedge \neg takenBy/student/ \\ & X_{qualified\_course}[cno = \text{“}cs66\text{''}]] \end{aligned}$$

Let i_c, j_p, and k_q be the number assigned to *course*, *project*, and *quantified*, respectively, and let n be the number of nodes in the DTD of Example 2.2. Here, $X_{\text{course\_course}} = X[i_c, i_c, n]$, obtained by computing $\mathsf{rec}(course, course)$; $X_{\text{course\_project}} = X[i_c, j_p, n]$, obtained by computing $\mathsf{rec}(course, project)$; and $X_{\text{qualified\_course}} = X[k_q, i_c, n]$, obtained by computing $\mathsf{rec}(qualified, course)$. The algorithm of Sect. 5 can then translate E_{Q_2} to equivalent relational queries.

The result below tells us that Algorithm XPathToEXp computes extended XPath queries in low polynomial time, as desired.

Theorem 4.2 *Each* XPath *query Q over a* DTD *D can be rewritten to an extended* XPath *expression E_Q in $O(|D|^3 * log|D| * |Q| * log|Q|)$ time, such that E_Q is equivalent to Q over all* DTDs *that contain D, and that the size of E_Q is bounded by $O(|D|^3 * |Q| * log|D|)$.*

Proof For the complexity, observe the following. (a) Algorithm XPathToEXp produces an extended XPath query with $O(|D|^2 * |Q|)$ many variables and $O(|D|^2 * |Q|)$ many equations, each consisting of at most two variables and thus takes $O(log(|D||Q|)$ space (to encode the variables). (b) In addition, $\mathsf{rec}(A, B)$ for '//' computed by Algorithm contains

```
Algorithm XPathToEXp
Input: an XPath query $Q$ over a DTD $D$.
Output: an extended XPath query $E_Q$ that is equivalent to $Q$ over all DTDs that contain $D$.
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 $A, B$ in $N$ do
5.          $\mathsf{x2e}(p, A, B) := \emptyset$;   $\mathsf{reach}(p, A) := \emptyset$;
6.  for each $p$ in the order of $L$ do
7.    for each $A$ in $N$ do
8.      case $p$ of
9.      (1)  $\epsilon$:  $\mathsf{x2e}(p, A, B) := \epsilon$ for all $B \in N$;   $\mathsf{reach}(p, A) := \{A\}$;
10.     (2)  $B$:  if $B$ is a child type of $A$
11.                then $\mathsf{x2e}(p, A, B) := B$;  $\mathsf{reach}(p, A) := \{B\}$;
12.                else $\mathsf{x2e}(p, A, B) := \emptyset$;  $\mathsf{reach}(p, A) := \emptyset$;
13.     (3)  $*$:  for each child type $B$ of $A$ in $D$ do
14.                   $\mathsf{x2e}(p, A, B) := B$;   $\mathsf{reach}(p, A) := \mathsf{reach}(p, A) \cup \{B\}$;
15.     (4)  $p_1/p_2$:  for each $C$ in $\mathsf{reach}(p_1, A)$ and each $B$ in $\mathsf{reach}(p_2, C)$ do
16.                   $\mathsf{x2e}(p, A, B) := X_{p_1}(A, C)/X_{p_2}(C, B)$;   $\mathsf{reach}(p, A) := \mathsf{reach}(p, A) \cup \{B\}$;
17.     (5)  $\epsilon//p_1$:   /* $\mathsf{reach}(A, \epsilon//)$, $\mathsf{rec}(A, B)$ are precomputed, with $X_r(A, B) = \mathsf{rec}(A, B)$ */
18.          for each $C$ in $\mathsf{reach}(A, \epsilon//)$ and each $B$ in $\mathsf{reach}(C, p_1)$ do
19.            $\mathsf{x2e}(p, A, B) := X_r(A, C)/X_{p_1}(C, B)$;   $\mathsf{reach}(p, A) := \mathsf{reach}(p, A) \cup \{B\}$;
20.     (6)  $p_1 \cup p_2$:  $\mathsf{x2e}(p, A, B) := X_{p_1}(A, B) \cup X_{p_2}(A, B)$ for all $B \in D$;
21.                $\mathsf{reach}(p, A) := \mathsf{reach}(p_1, A) \cup \mathsf{reach}(p_2, A)$;
22.     (7)  $p'[q]$:
23.          for each $B$ in $\mathsf{reach}(p', A)$ do
24.              RewQual $([q], B)$;     /* RewQual $([q], B)$ returns $\mathsf{x2e}([q], B, B)$ and $X_{[q]}(B, B)$ */
25.              if $\mathsf{x2e}([q], B, B) = [\epsilon]$     /* $[q]$ holds at $B$ */
26.              then $\mathsf{x2e}(p, A, B) := X_{p'}(A, B)$; $\mathsf{reach}(p, A) := \mathsf{reach}(p, A) \cup \{B\}$;
27.              else if $\mathsf{reach}([q], B) \neq [\emptyset]$     /* $[q]$ is not false at $B$ */
28.              then $\mathsf{x2e}(p, A, B) := X_{p'}(A, B)[X_{[q]}(B, B)]$; $\mathsf{reach}(p, A) := \mathsf{reach}(p, A) \cup \{B\}$;
29.     optimize $\mathsf{x2e}(p, A, B)$ by removing $\emptyset$ using $\emptyset \cup E = E$, $E_1/\emptyset/E_2 = \emptyset$;
30. $E_Q := \{X_Q = \bigcup_{B \in \mathsf{reach}(Q, r)} X_Q(A.B)\}$;     /* $r$ is the root of $D$ */
31. $E_Q := E_Q \cup \{X_p(A, B) = \mathsf{x2e}(p, A, B) \mid p \in L,\ A \in N, B \in N\} \cup \{\mathsf{rec}(A, B) \mid A \in N, B \in N\}$;
32. return $E_Q$;
```

Fig. 8 Rewriting algorithm from XPath to extended XPath

$O(|D|^3)$ many equations, which takes $O(|D|^3 * log|D|)$ space. Putting these together, the extended XPath query produced takes no more than $O(|D|^3 * log|D| * |Q|)$ space. (c) Each step of the inner-most loop of Algorithm XPathToEXp takes at most $O(|D|^2 * log|Q| * log|D|)$ time, where $log|Q| * log|D|$ is for writing the variables involved. Thus the algorithm takes no more than $O(|D|^3 * log|D| * |Q| * log|D|)$ time.

We show that E_Q is equivalent to Q over all DTDs that contain D, by induction on the structure of Q. For the base cases, i.e., when Q is ϵ, A, $*$ and //, the statement trivially holds. In particular, the argument for // is given in the proof of Theorem 4.1.

For the inductive step, assume that the statement holds for sub-queries p_1, p_2 of Q, i.e., $\mathsf{x2e}(p_i, A, B)$ is equivalent to p_i w.r.t. A and B for all $A, B \in D$ and $i = 1, 2$. We show that the statement holds for p_1/p_2. Proofs for the other cases are similar.

Let D' be an arbitrary DTD that contains D, and σ be the mapping that, given any instance T of D', extracts a subtree V of T that is an instance of D, as specified in Sect. 3. Let v be an A element in V, where σ maps v to an A element $\sigma(v)$ in T. We need to show that a node u' can be reached from $\sigma(v)$ via $\mathsf{x2e}(p_1/p_2, A, B)$ in T iff there exists a node u in V such that $u' = \sigma(u)$ and u can be reached from v via p_1/p_2 in V.

First, for any node u in V reached from v via p_1/p_2, there must be a node w in V such that w is reached from v via p_1 and u is reached from w via p_2. Assume w is labeled C. Then obviously $C \in \mathsf{reach}(p_1, A)$. By the induction

```
Procedure RewQual ([q], A)

Input: an XPath qualifier $[q]$ and an element type $A$ in a DTD $D$.
Output: an extended XPath query $\mathsf{x2e}[q], A, A)$ equivalent to $[q]$ at $A$ elements over DTDs that contain $D$.
1.  case $[q]$ of
2.  (a) $[p_1]$:  XPathToEXp $(p_1)$;     /* XPathToEXp computes $\mathsf{x2e}(p_1, A, B)$ and $\mathsf{reach}(p_1, A)$ */
3.           if $\mathsf{reach}(p_1, A) = \emptyset$
4.           then $\mathsf{x2e}([q], A, A) := \emptyset$;
5.           else if $\epsilon$ is in $\mathsf{x2e}(p_1, A, B)$
6.           then $\mathsf{x2e}([q], A, A) := \epsilon$;
7.           else for each $B \in \mathsf{reach}(p_1, A)$ do
8.                $\mathsf{x2e}([q], A, A) := \mathsf{x2e}([q], A, A) \cup X_{p_1}(A, B)$;
9.  (b) $[text() = c]$:  $\mathsf{x2e}([q], A, A) := text() = c$;
10. (c) $[q_1 \wedge q_2]$:  RewQual $(q_1, A)$;  RewQual $(q_2, A)$;
11.           $\mathsf{x2e}([q], A, A) := \mathsf{optimize}(X_{[q_1]}(A, A) \wedge X_{[q_2]}(A, A))$;
12. (c)  $[q_1 \vee q_2]$:  RewQual $(q_1, A)$;  RewQual $(q_2, A)$;
13.           $\mathsf{x2e}([q], A, A) := \mathsf{optimize}(X_{[q_1]}(A, A) \vee X_{[q_2]}(A, A))$;
14. (e)  $[\neg q_1]$:  RewQual $(q_1, A)$;
15.           $\mathsf{x2e}([q], A, A) := \mathsf{optimize}(\neg X_{q_1}(A, A))$;
16. $E_{[q]} := \{X_{q_1}(A, A) = \mathsf{x2e}([q_1], A, A)$    $q_1$ is a sub-query of $q\}$;
17. prune and return $E_{[q]}$;
```

Fig. 9 Rewriting algorithm from qualifiers

hypothesis and the definition of σ, there exist w', u' in T such that $w' = \sigma(w)$, $u' = \sigma(u)$; moreover, w' can be reached from $\sigma(v)$ via $\mathsf{x2e}(p_1, A, C)$ and u' can be reached from w' via $\mathsf{x2e}(p_2, C, B)$. Thus from the definition of $\mathsf{x2e}(p_1/p_2, A, B)$ and the semantics of XPath, it follows that u' can be reached from $\sigma(v)$ via $\mathsf{x2e}(p_1/p_2, A, B)$ in T.

Conversely, for any node u' in T reached from $\sigma(v)$ via $\mathsf{x2e}(p_1/p_2, A, B)$, by the definition of $\mathsf{x2e}(p_1/p_2, A, B)$ and the semantics of XPath, there must be a node w' in T such that w is reached from $\sigma(v)$ via $\mathsf{x2e}(p_1, A, C)$ and u' is reached from w' via $\mathsf{x2e}(p_2, C, A)$, where w' is labeled C. By the induction hypothesis for p_1, there exists w in V such that $w' = \sigma(w)$, and w can be reached from v via p_1. Moreover, by the definition of σ, $C \in \mathsf{reach}(p_1, A)$. Then by the induction hypothesis for p_2, there exists u in V such that $u' = \sigma(u)$ and u can be reached from w via p_2. By the semantics of XPath, it follows that u can be reached from v via p_1/p_2 in V. □

Observe the following. First, extended XPath queries capture DTD *recursion and* XP*ath recursion* in a uniform framework by means of the general Kleene closure E^*. The use of variables makes it possible to translate XPath queries in polynomial time, in contrast to the exponential blowup of query translation into regular XPath [22]. Second, during the translation, algorithm XPathToEXp 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 extended query. Fourth, both $|Q|$ and $|D|$ are far smaller than the data (XML tree) size in practice. Finally, as remarked earlier, Algorithm XPathToEXp also provides query answering ability for XPath queries over certain virtual XML views.

5 From extended XPath expressions to SQL

In this section we present an algorithm, Algorithm EXpToSQL, for rewriting extended XPath expressions into equivalent SQL queries with the simple LFP operator. Together with Algorithm XPathToEXp given in the last section, these provide a solution for answering XPath queries on XML data stored in relations. We also discuss optimization of the produced SQL queries.

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 τ_d shreds XML trees of D into databases of $\mathcal{R}$. Given an extended XPath expression E_Q over D, Algorithm EXpToSQL computes a sequence Q' of equivalent relational queries with the simple LFP operator Φ such that for any XML tree T of D, $E_Q(T) = Q'(\tau_d(T))$. We write Q' in the relational algebra (RA), which can be easily coded in SQL.

More specifically, Q' is a list of the form $R_e \leftarrow \mathsf{e2s}(e)$, where e is an sub-expression of the extended XPath expression E_Q, R_e is a temporary table which is used in later queries, and $\mathsf{e2s}(e)$ is the RA query equivalent to e. Conceptually, the list Q' can be properly ordered such that if

e is a sub-expression of e', then $R_e \leftarrow \mathsf{e2s}(e)$ precedes $R_{e'} \leftarrow \mathsf{e2s}(e')$ in Q', i.e., when $\mathsf{e2s}(e)$ is needed, the query has already been evaluated and its result is available in R_e, which can be directly used in $\mathsf{e2s}(e')$. As will be seen in Sect. 5.2, the SQL queries can be evaluated "top–down" following a lazy evaluation strategy: $\mathsf{e2s}(e)$ is not evaluated unless it is needed.

Algorithm EXpToSQL suffices to translate the query produced by Algorithm XPathToEXp into equivalent SQL queries. To see this, observe the following. First, the equations in the extended XPath query returned by Algorithm XPathToEXp can be sorted as $(X_{p_1} = E_{p_1}, \ldots, X_{p_m} = E_{p_m})$, where p_i's are sub-queries of the input query Q such that all sub-queries of a sub-query p precede p, with $p_m = Q$. In other words, X_{p_i} only appears in E_{p_j} if $i < j$. We can apply Algorithm EXpToSQL to $E_{p_1}, \ldots, E_{p_m}$ one by one in this order, creating a temporary table R_{p_i} for each $\mathsf{e2s}(E_{p_i})$. For each occurrence of X_{p_i} in E_{p_j}, we simply use R_{p_i} in $\mathsf{e2s}(E_{p_j})$. Thus it suffices for EXpToSQL to translate extended XPath expressions E_{p_i}. Second, the query returned by XPathToEXp is equivalent to the XPath query Q over all DTDs that contain D. Thus it is equivalent to Q over D since D contains itself. Putting these together, we have that EXpToSQL and XPathToEXp translate the XPath Q over D to equivalent SQL queries over $\mathcal{R}$.

We show Algorithm EXpToSQL in Fig. 10. The algorithm takes an extended XPath expression E_Q over the DTD D as input, and returns an equivalent sequence Q' of RA queries with the LFP operator Φ as output. The algorithm is based on dynamic programming: for each sub-expression e of E_Q, it computes $\mathsf{e2s}(e)$, which is the RA query translation of e; it then associates $\mathsf{e2s}(e)$ with a temporary table R_e and increments the list Q' with $R_e \leftarrow \mathsf{e2s}(e)$. More specifically, $\mathsf{e2s}(e)$ is computed by assembling $\mathsf{e2s}(e_i)$ where e_i's are its immediate sub-queries. Thus upon the completion of the processing one will get the list Q' equivalent to E_Q. To do this, the algorithm first finds the list L of all sub-expressions of E_Q and topologically sorts them in ascending order (line 1). Then, for each sub-query e in L, it computes $\mathsf{e2s}(e)$ (lines 3–24), bottom-up starting from the inner-most sub-query of E_Q, and based on the structure of e [cases (1)–(12)].

A subtle issue is that the LFP operator Φ 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 ϵ, we assume a relation R_{id} consisting of tuples $(v, v, v.val)$ for all nodes (IDs) v in the input XML tree except the root r. Note that R_{id} is the identity relation for join operation: $R \bowtie R_{id} = R_{id} \bowtie R = R$ for any relation R. With this we translate $(E)^*$ to $\Phi(R) \cup R_{id}$, where R codes E, and R_{id} tuples will be eliminated in a later stage. We rewrite ϵ into R_{id} just to simplify the presentation of our algorithm; a more efficient translation will be described in Sect. 5.2.

More specifically, EXpToSQL handles different cases of e as follows.
(Case 2) It rewrites a label A to the corresponding relation R_A.
(Case 3) It replaces each occurrence of variable X with its corresponding temporary table R_X. From the discussion above one can see that for each X appearing in an expression E_{p_j}, $X = E_{p_i}$ has already been processed and a table R_X has been associated with $\mathsf{e2s}(E_{p_i})$.
(Case 4) Concatenation is coded with projection Π and join $\bowtie$.
(Cases 5, 11) Union and disjunction are encoded with union $\cup$ in relational algebra.
(Case 6) Kleene closure $(E)^*$ is converted to the LFP operator Φ, as remarked above.
(Case 10) Conjunction is coded with set intersection implemented with union $\cup$ and set difference $\setminus$ in relational algebra.
(Qualifiers) An expression with qualifier $e = e_1[q]$ is converted to an RA query $\mathsf{e2s}(e)$ that returns only those $\mathsf{e2s}(e_1)$ tuples t_1 for which there exists an $\mathsf{e2s}(q)$ 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). For example, it converts $[e_1]$ to $\mathsf{e2s}(e_1)$ when e_1 is an extended XPath expression (case 7). There are, however, two special cases. First, it rewrites $e_1[\neg q]$ to a RA query $\mathsf{e2s}(e)$ that returns only those $\mathsf{e2s}(e_1)$ tuples t_1 for which there exists no $\mathsf{e2s}(q)$ 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 precisely the semantics of negation in XPath. Second, it rewrites $e = e_1[\mathit{text}() = c]$ in terms of selection σ that returns all tuples of $\mathsf{e2s}(e_1)$ that have the text value c.

In each of the cases above, the list Q' is incremented by adding $R_e \leftarrow \mathsf{e2s}(e)$ as the head of Q' (line 25). After the iteration it yields $\sigma_{F='\_'}\mathsf{e2s}(E_Q)$ (line 26), 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. 10), and returns the cleaned Q' (lines 28–29).

Example 5.1 Consider the XPath query $Q_1 =$ `dept//project` over the `dept` DTD of Fig. 1a. Over the simplified DTD is Fig. 1b, Q_1 becomes $R_d//R_p$. A possible equivalent RA translations Q_1' has been given in Example 3.5, 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_γ, and the final result as shown in Table 3.

As another example, recall the XPath query Q_2 from Example 2.2, and its extended XPath query translation X_{Q_2} from Example 4.3, which contains $X_{\text{course\_course}}$ $(= X[i_c, i_c, n])$, $X_{\text{course\_project}}$ $(= X[i_c, j_p, n])$, and $X_{\text{qualified\_course}}$ $(= X[k_q,$

```
Algorithm EXpToSQL

Input: an extended 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)  $\epsilon$:  e2s($e$) := $R_{id}$;
6.      (2)  $A$:  e2s($e$) := $R_A$;
7.      (3)  $X$:  e2s($e$) := $R_X$;   /* $R_X$ is a temporary table for storing $E$ where $X = E$ */
8.      (4)  $e_1/e_2$:  let $R_1$ = e2s($e_1$), $R_2$ = e2s($e_2$);
9.                e2s($e$) := $\Pi_{R_1.F, R_2.T, R_2.V}(R_1 \bowtie_{R_1.T=R_2.F} R_2)$;
10.     (5)  $e_1 \cup e_2$:  let $R_1$ = e2s($e_1$), $R_2$ = e2s($e_2$);
11.               e2s($e$) := $R_1 \cup R_2$;
12.     (6)  $E^*$:  let $R$ = e2s($e$);
13.               e2s($e$) := $\Phi(R) \cup R_{id}$;
14.     (7)  $e_1[q]$:  let $R_1$ = e2s($e_1$), $R_q$ = e2s($q$);
15.          e2s($e$) := $\Pi_{R_1.F, R_1.T, R_2.V}(R_1 \bowtie_{R_1.T=R_q.F} R_q)$;
                  /* returns $R_1$ tuples that connect with $R_2$ tuples */
16.     (8)  $[e_1]$:  e2s($e$) := e2s($e_1$);
17.     (9)  $e_1[text() = c]$:  let $R_1$ = e2s($e_1$);
18.               e2s($e$) := $\sigma_{R_1.V=c} R_1$;   /* select tuples $t$ of $R_1$ with $t.V = c$ */
19.     (10) $[q_1 \wedge q_2]$:  let $R_1$ = e2s($q_1$);  $R_2$ = e2s($q_2$);
20.               e2s($e$) := $R_1 \cup R_2 \setminus ((R_1 \setminus R_2) \cup (R_2 \setminus R_1))$;   /* e2s($e$) = $R_1 \cap R_2$; */
21.     (11) $[q_1 \vee q_2]$:  let $R_1$ = e2s($q_1$);  $R_2$ = e2s($q_2$);
22.               e2s($e$) := $R_1 \cup R_2$;
23.     (12) $e_1[\neg q]$:  let $R_q$ = e2s($q$),  $R_1$ = e2s($e_1$);
24.               e2s($e$) := $R_1 \setminus \Pi_{R_1.F, R_1.T, R_1.V}(R_1 \bowtie_{R_1.T=R_q.F} R_q)$;
                  /* only $R_1$ tuples not connecting to any $R_q$ tuple */
25.     $Q'$ := ($R_e \leftarrow$ e2s($e$)) :: $Q'$;   /* add e2s($e$) to $Q'$ */
26. e2s($E_Q$) := $\sigma_{F='\_'}$e2s($E_Q$);  /* select nodes reachable from root */
27. $Q'$ := e2s($E_Q$) :: $Q'$;
28. optimize $Q'$ by extracting common sub-queries;
29. return $Q'$;
```

Fig. 10 Rewriting algorithm from extended XPath to SQL

i_c, n]) computed by Algorithm CycleE$_X$. Given X_{Q_2}, the EXpToSQL algorithm generates the RA translation below:

$X_{\text{course\_course}}/prereq/course[cno = \text{“}cs66''] :$

$\sigma_{cno=\text{“}cs66''}(R_{cc} \bowtie R_c)$

$takenBy/student/X_{\text{qualified\_course}}[cno = \text{“}cs66''] :$

$\sigma_{cno=\text{“}cs66''}(R_s \bowtie R_{qc})$

Suppose the results of $X_{\text{course\_course}}$, $X_{\text{course\_project}}$, and $X_{\text{qualified\_course}}$ are stored in R_{cc}, R_{cp}, and R_{qc}, respectively.

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 X_{Q_2} becomes $Q_2^2 \setminus (Q_2^2 \bowtie R_2)$ where projections are omitted. In contrast, the algorithm of [39] cannot translate XPath queries of this form to relational queries.

The corollary below confirms that our translation algorithms provide an effective solution for answering XPath queries over (possibly recursive) DTDs by means of traditional RDBMS.

Corollary 5.1 *Each* XPath *query Q over a* DTD *D can be rewritten to an equivalent sequence of* SQL *queries (with the* LFP *operator) in $O(|D|^3 * log|D| * |Q| * log|Q|)$ time.*

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

F	T
R	
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
R_γ	
c_1	c_2
c_1	c_3
c_1	c_4
c_1	c_5
...	...
R_f	
d_1	p_1
d_1	p_2

Proof It is easy to verify the following. (a) Given an input extended XPath expression E_Q, Algorithm EXpToSQL takes $O(|E_Q|)$ time to compute a sequence Q' of SQL queries (with the LFP operator). The size of Q' is also $O(|E_Q|)$. (b) The SQL queries Q' are equivalent to E_Q, i.e., for any XML tree T of D, $E_Q(T) = Q'(\tau_d(T))$. This can be verified by induction on the structure of E_Q. Recall from Theorem 4.2 that given an input XPath query Q over D, Algorithm XPathToEXp computes an extended XPath query E_Q equivalent to Q over all DTDs that contain D. The query E_Q can be computed in $O(|D|^3 * log|D| * |Q| * log|Q|)$ time, and its size is in $O(|D|^3 * |Q| * log|D|)$. Putting these together, we have that the SQL queries Q' can be computed from Q by using XPathToEXp followed by EXpToSQL in $O(|D|^3 * log|D| * |Q| * log|Q|)$ time. Furthermore, the size of Q' is in $O(|D|^3 * |Q| * log|D|)$. □

Observe the following. First, algorithm EXpToSQL 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 XPathToEXp and EXpToSQL can be easily combined into one; we present them separately to focus on their different functionality.

5.2 Optimization: pushing selections into the LFP operator

Algorithms XPathToEXp and EXpToSQL 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., sophisticated methods for pushing selections/projections into the LFP operator [2,3,5,6,8] can be incorporated into our translation algorithms to further optimize generated relational queries. Below we present some optimization strategies.

Top–down evaluation. To simplify the discussion we presented in Sect. 5.1 a conceptual evaluation strategy for the list Q' of SQL queries generated: if e is a sub-expression of e', then e2s(e) is evaluated before e2s(e'). In practice, this can be computed following a lazy evaluation strategy: e2s(e) is not evaluated unless it is explicitly needed and invoked by e2s(e). Consider, e.g., e_1/e_2 (case 4 of Algorithm EXpToSQL). Instead of computing e_1/e_2 in the order of e2s(e_1), e2s(e_2) and e2s(e_1/e_2), this can be conducted lazily as follows: first evaluate e2s(e_1); if the result is empty, then there is no need to evaluate e2s(e_2) and e2s(e_1/e_2). That is, e2s(e_2) is evaluated only when necessary; similarly for other cases in Algorithm EXpToSQL. Alternatively, one can treat e2s(e_1) and e2s(e_2) as sub-queries of e2s(e_1/e_2) such that the queries in Q' are evaluated top–down following the inverse order of the list.

Pushing selections into LFP. We next show how to push selections into LFP. Consider an XPath query $Q_3 = R_d[id = a]/R_c//R_p$. To simplify the discussion, assume that our algorithms rewrite Q_3 into $R_1 \leftarrow Q_d$ and $R_2 \leftarrow$ LFP(R_0), where Q_d and LFP(R_0) compute $R_d[id = a]$ and $R_c//R_p$, (i.e., rec(R_c, R_p)), respectively. While $R_1 \bowtie R_2$ yields the right answer, we can improve the performance by pushing the selection into the LFP computation such that it only traverses "paths" starting from the R_c children of those R_d nodes with $id = a$. Recall from Eq. (2) that one can specify a predicate C on the join between R_Φ and R_0 in LFP, where R_0 is the input relation and R_Φ is the relation being computed by the LFP (Sect. 3; supported by *connectby* of Oracle and *with...recursion* of IBM DB2). That is, $R_\Phi^0 \leftarrow R_0$, $R_\Phi^i \leftarrow R_\Phi^{i-1} \cup (R_\Phi^{i-1} \bowtie_C R_0)$, and finally, $R_\Phi \leftarrow R_\Phi^m$, where $R_\Phi^m = R_\Phi^{m+1}$, i.e., the fixpoint. Here $C = R_\Phi.F \in \pi_T(R_1) \wedge R_\Phi^{i-1}.T = R_0.F$. ('$\in$' denotes *in* in SQL), i.e., besides the equijoin $R_\Phi.T = R_0.F$ we want the F (*from*) attribute of R_Φ to match the T (*to*) attribute of R_1. Then, each iteration of the LFP only adds tuples (f, t), where f is a child of a node in $\pi_T(R_1)$.

Similarly the selection in $R_d//R_c/R_p[id=c]$ can be pushed into LFP(R_0) for rec(R_d, R_c). Let R_1 be the relation found and the LFP join condition be: $R_\Phi^{i-1}.F = R_0.T \wedge R_\Phi.T \in \pi_F(R_1)$. Then the LFP only returns tuples of the form (f, t), where t is the parent of a node in $\pi_F(R_1)$.

In general, given an extended XPath expression $R_1 \bowtie$ LFP(R_0) (or LFP(R_0) $\bowtie R_1$), where R_1 is associated with a selection condition, we can push the selection into LFP along the same lines as above. Let us denote the query resulted from this as $\mathsf{push}(R_1, R_0)$.

Below we identify general cases where the push operation can be applied. We may decompose a list $\bar{R}$ of RA queries and employ the push operation as follows:

(i) by union: $\bar{R} = R_1 \bowtie \mathrm{LFP}(R_0) \cup R'_1 \bowtie \mathrm{LFP}(R_0)$, and we can rewrite $\bar{R}$ to equivalent yet more efficient $\bar{R} = \mathsf{push}(R_1, R_0) \cup \mathsf{push}(R'_1, R'_0)$;
(ii) by conjunction: $\bar{R} = R_1 \bowtie \mathrm{LFP}(R_0) \bowtie R'_1 \bowtie \mathrm{LFP}(R_0)$; in this case we can rewrite $\bar{R}$ to $\bar{R} = \mathsf{push}(\mathsf{push}(R_1, R_0) \bowtie R'_1, R'_0)$;
(iii) by nest: $\bar{R} = R_2 \bowtie \mathrm{LFP}(R_1 \bowtie \mathrm{LFP}(R_0))$, and we can rewrite it to $\bar{R} = \mathsf{push}(R_2, \mathsf{push}(R_1, R_0))$.

In particular, $\bar{R} = R_2 \bowtie R_1 \bowtie \mathrm{LFP}(R_0)$ and $\bar{R} = R_2 \bowtie \mathrm{LFP}(R_1 \bowtie R_0)$ can be optimized following cases (ii) and (iii) above. Here we assume that there may exist arbitrary selection condition on R_1 or R'_1. In fact, we can push selections into LFP even when there are no explicit user-given selection conditions. Consider, for example, $R_{1_1} \bowtie R_{1_2} \bowtie \cdots \bowtie R_{1_n} \bowtie \mathrm{LFP}(R_0)$. By first computing joins $R_{1_1} \bowtie R_{1_2} \bowtie \cdots \bowtie R_{1_n}$ followed by projection on R_{1_n}, namely, $\pi_{R_{1_n}}(R_{1_1} \bowtie R_{1_2} \bowtie \cdots \bowtie R_{1_n})$, we can also push this query into $\mathrm{LFP}(R_0)$ and thus speed up the computation of the fixpoint. As will be seen in Sect. 6, this optimization is effective.

Handling $(E)^*$. To simplify the discussion, in Sect. 5.1 we rewrite ϵ into R_{id}. In our implementation typically a much smaller relation is used instead of R_{id}. Consider e_1/e^* for instance, with a temporary table $R_1 \leftarrow \mathsf{e2s}(e_1)$. Then instead of using R_{id} in $\mathsf{e2s}(e^*)$, we use R_1 instead. Similarly we handle combinations of e^* with other sub-queries.

Other optimization: Besides the push operation, several other optimization techniques can be used to improve the rewritten SQL queries. 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 black-box 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 [30,50]; in particular, multi-query optimization techniques (e.g., [54]) can be easily incorporated into our framework to optimize a sequence of SQL queries produced by our algorithms. In addition, as remarked earlier, optimization techniques by leveraging integrity constraints [41,42] developed for [39] can also be incorporated into translations from extended XPath to SQL in our approach.

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 $\Phi()$ 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 $\mathsf{SQLGen\text{-}R}$ algorithm proposed in [39],
- our rewriting algorithms by using Tarjan's method (referred to as CycleE of Fig. 6) to find $\mathsf{rec}(A, B)$, i.e., paths from node A to B in a DTD graph, and
- our rewriting algorithms by using $\mathsf{CycleE_X}$ of Fig. 7 to compute $\mathsf{rec}(A, B)$.

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. 11a (2 cross cycles). The two real-life DTDs are (1) a 4-cycle DTD extracted from **BIOML** (BIOpolymer Markup Language [10]), as shown in Fig. 11b; and (2) a 9-cycle DTD extracted from **GedML** (Genealogy Markup Language [27]), given in Fig. 11c.

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 $\mathsf{CycleE_X}$ outperform the other two in most cases.

Implementation. We have implemented a prototype system supporting $\mathsf{SQLGen\text{-}R}$, CycleE and $\mathsf{CycleE_X}$, using Visual C++, denoted by **R**, **E** and **X**, respectively, in all the figures. $\mathsf{SQLGen\text{-}R}$ rewrites a query with the *with...recursive* operator, while CycleE and $\mathsf{CycleE_X}$ translate a query to

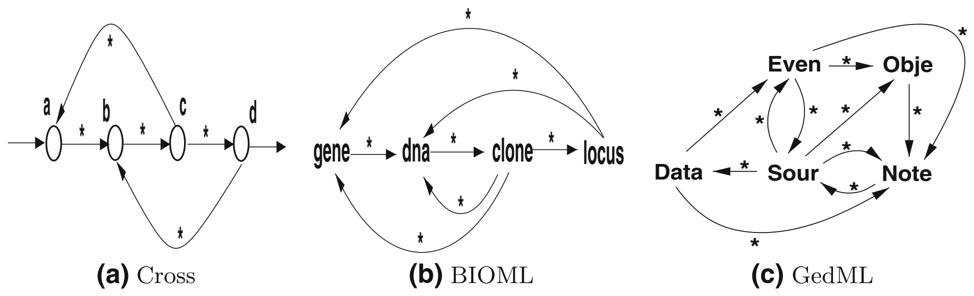


Fig. 11 DTD Graphs. **a** Cross, **b** BIOML, **c** GedML

a sequence of SQL queries. We run a batch to execute these rewritten SQL queries. We conducted experiments using IBM DB2 Enterprise 9 on a single 2.8 GHz 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 is 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 [59]. Indexes were generated for all possible joined attributes.

Experimental study. We conducted five sets of experiments. (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) In the second set of experiments we evaluated the effectiveness of our optimization method by pushing selections into the LFP operator. (3) We tested the scalability of our generated SQL queries w.r.t. different database sizes using a query containing //. Experiments (1)–(3) were conducted with the simple cross-cycle DTD graph. (4) We tested several XPath queries with different DTDs, which are subgraphs of the real-life DTDs BIOML using the same database. The main difference between (1) and (4) is that the former tested the same queries with different databases, and the latter tested different queries with the same database. (5) Finally, we examined the numbers of operators (LFP, etc) in the SQL queries generated by CycleE and CycleE$_X$, respectively.

6.1 Exp-1: evaluation of selective queries

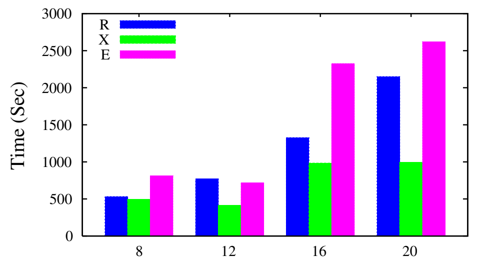
In this study, over the simple cross-cycle DTD (Fig. 11a), we tested the following four XPath queries:

- 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 //).

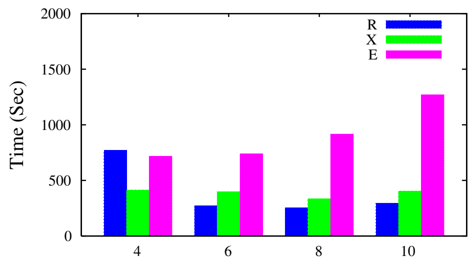
The XPathToEXp algorithm translates these XPath queries into four extended XPath expressions, namely, $Q'_a = a/X_{\text{b\_c}}/d$, $Q'_b = a[X_{\text{a\_b}}/c]/X_{\text{a\_c}}/d$, $Q'_c = a[\neg X_{\text{a\_b}}/c]$, and $Q'_d = a[\neg X_{\text{a\_b}}/c \vee (b \wedge X_{\text{a\_c}}/d)]$, respectively. Here, $X_{\text{b\_c}}$, $X_{\text{a\_b}}$, and $X_{\text{a\_c}}$ will be computed by rec(b, c), rec(a, b), and rec(a, c) using CycleE and CycleE$_X$ to test CycleE and CycleE$_X$, respectively. We tested SQLGen-R by generating a *with...recursive* query for each rec(A, B) in our translation framework.

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 $X_R = 4$,

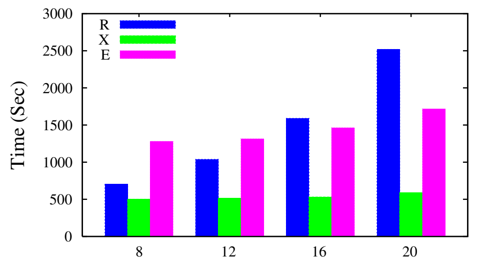
Fig. 12 Processing time for cross cycles (Fig. 11a). **a** Q_a: vary X_L, **b** Q_a: vary X_R, **c** Q_b: vary X_L, **d** Q_b: vary X_R, **e** Q_c: vary X_L, **f** Q_c: vary X_R, **g** Q_d: vary X_L, **h** Q_d: vary X_R



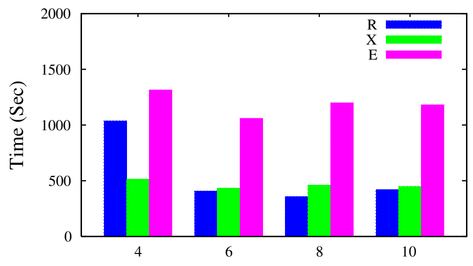
(a) Q_a: Vary X_L



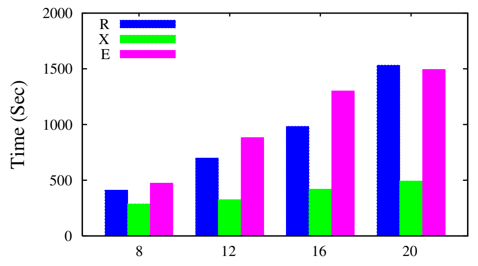
(b) Q_a: Vary X_R



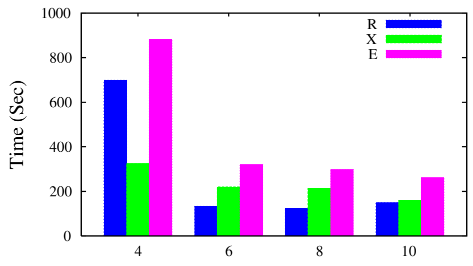
(c) Q_b: Vary X_L



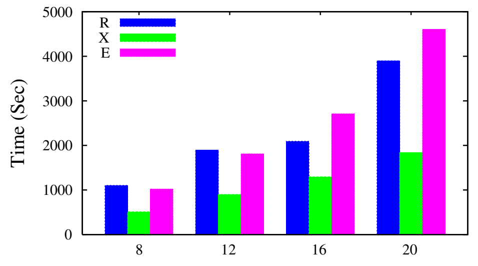
(d) Q_b: Vary X_R



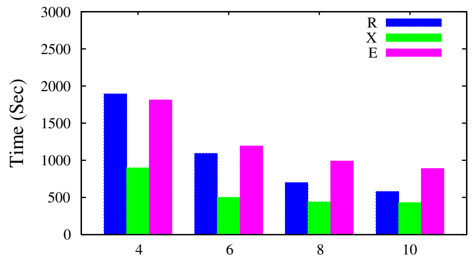
(e) Q_c: Vary X_L



(f) Q_c: Vary X_R

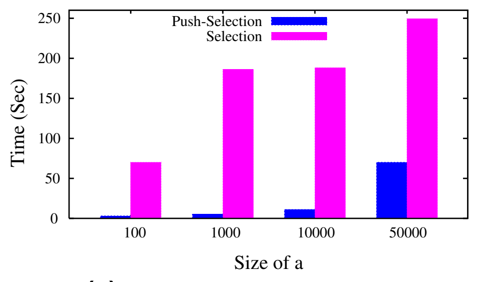


(g) Q_d: Vary X_L

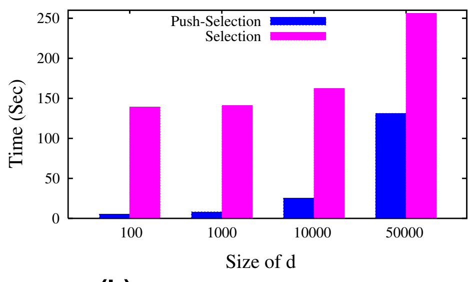


(h) Q_d: Vary X_R

Fig. 13 Pushing selection ($X_R = 8$ and $X_L = 12$). **a** Vary the number of a elements, **b** vary the number of d elements



(a) Vary the Number of a Elements



(b) Vary the Number of d Elements

whereas the other figure shows the elapsed time while varying X_R from 4 to 10 with $X_L = 12$.

Figure 12a, c, e, g, show the elapsed time while varying X_L, when X_R is fixed. The XML trees become higher, but the

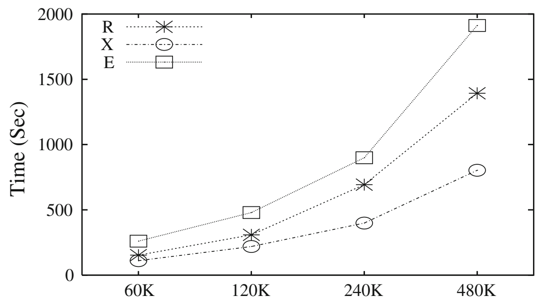


Fig. 14 Scalability test ($X_R = 4$ and $X_L = 16$)

distribution of widths in the XML trees remains unchanged, while X_L increases. The elapsed time for all the three approaches increases. As can be seen from the figures, the performance of SQLGen-R and CycleE is significantly affected while X_L increases. However, the performance CycleE$_X$ is marginally affected. CycleE$_X$ noticeably outperforms SQLGen-R and CycleE.

Figure 12b, d, f, h, show the elapsed time while varying X_R, when X_L is fixed. In other words, the average number of children per element in an XML increases, and the height of the XML tree remains unchanged, while X_R increases. More precisely, the XML generator generates an XML tree with more elements at the leaf level for a larger X_R value. The percentages of the leaf nodes in the XML trees are 50, 67, 74, 80%, when $X_R = 4$, $X_R = 6$, $X_R = 8$, and $X_R = 10$, respectively. With such distributions, SQLGen-R performs better, while X_R increases. It is difficult to analyze the *with...recursive*, but it can be because the most results are computed in a few iterations. CycleE$_X$ is marginally affected by the changes of X_R values, it shows similar performance while X_R increases. CycleE performs worst due to the large number of operations it needs to perform.

6.2 Exp-2: pushing selections into LFP

We tested two XPath queries with selection conditions: $Q_e = a[id = A_i]/b//c/d$ and $Q_f = a/b//c/d[id = D_i]$. 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. 11a, fixing the size of the datasets while varying the size of the set selected by the qualifiers of A_i and D_i. Figure 13a, b show the results. In Fig. 13a, we vary the number of qualified a elements from 100 to 50,000, while in Fig. 13b, we vary the number of qualified d elements from 100 to 50,000. It is shown that as expected, performance improvement by pushing selections into the LFP operator is significant.

6.3 Exp-3: scalability test

Figure 14 demonstrates the scalability of our algorithms by increasing the dataset sizes, for an XPath query `a//d` over the cross-cycle DTD (Fig. 11a). We set $X_L = 16$, because the default $X_L = 12$ is not large enough for the XML generator to generate such large datasets. When the parameters are fixed, the XML generator can generate different sizes of XML databases but with the similar distributions in terms of heights/widths. The XML dataset size increases to 480,000 elements from 60,000 elements. We find that CycleE$_X$ outperforms both SQLGen-R and CycleE noticeably, and SQLGen-R outperforms CycleE. When the dataset size is 480,000, the costs of CycleE and SQLGen-R are 2.4 times and 1.7 times of the cost of CycleE$_X$, respectively. This shows that when dataset is large, our optimization technique is effective enough to outperform *with...recursive*, because it can reduce the number of LFP operators and unnecessary joins and unions.

6.4 Exp-4: complex cycles (extracted from real-life DTDs)

We next show the results of testing XPath queries on the extracted 4-cycle BIOML DTD.

First, We tested XPath queries over the extracted DTD graphs from BIOML. We considered four subgraphs of the BIOML DTD of Fig. 11b in order to demonstrate the impact of different DTDs on the translated SQL queries. These subgraphs are shown in Fig. 15. The XPath queries tested on these extracted DTD graphs 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. 11b) 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. 16. We find that CycleE$_X$ outperforms SQLGen-R and CycleE in all the cases, except case `2b`. In case `4a`, for example, SQLGen-R needs to use 7 joins and 7 unions in each iteration; CycleE needs to process 6 join, 2 LFP and 3 union operators; and CycleE$_X$ uses 5 join, 1 LFP and 4 union operators. CycleE$_X$ significantly outperforms SQLGen-R and CycleE 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. 11c). Here SQLGen-R uses 11 joins and 11 unions in each iteration, because this DTD consists of 11 edges. CycleE generates a

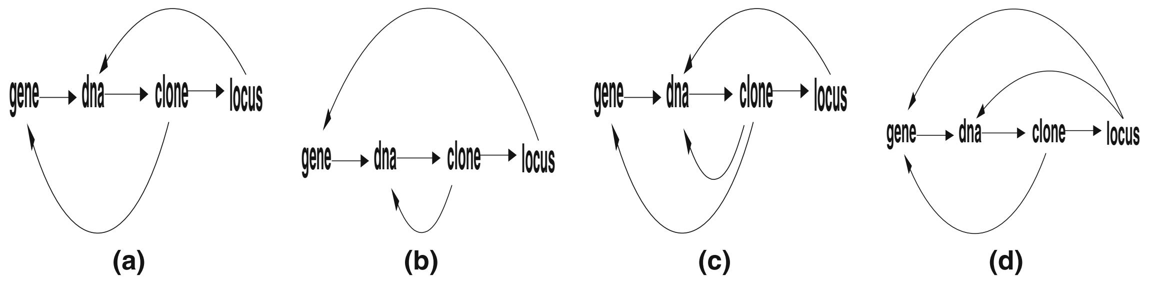


Fig. 15 Different DTD graphs extracted from BIOML

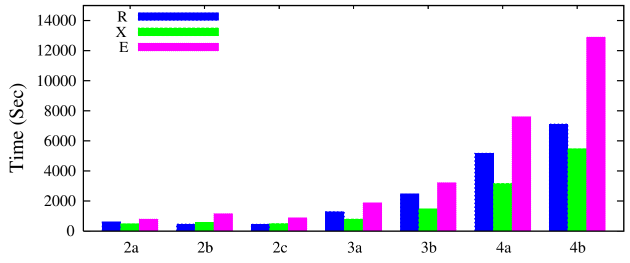


Fig. 16 XPath queries on the extracted BIOML DTDs

sequence of 77 join, 6 LFP and 79 union operators. CycleE$_X$ uses 12 join, 3 LFP and 9 union operators.

For this test, we generated large datasets using the IBM XML Generator without trimming. Figure 17a shows the results while varying X_L with $X_R = 6$, where the dataset sizes are 286,845 ($X_L = 13$), 845,045 ($X_L = 14$), and 1,019,798 ($X_L = 15$). Figure 17b shows the results while varying X_R with $X_L = 16$, where the dataset sizes are 226,663 ($X_R = 6$), 119,999 ($X_R = 7$), and 5,041,437 ($X_R = 8$). The largest dataset is 2 times larger than that used for the BIOML test. CycleE$_X$ outperforms CycleE and SQLGen-R for different X_L values in Fig. 17a. As shown in Fig. 17b, CycleE$_X$ performs noticeably better than CycleE. But CycleE$_X$ performs in a similar way as SQLGen-R when varying X_R, because X_R has impacts on the join selectivities but not on the number of iterations in the *with...recursive*.

Table 4 XPath queries over different DTD graphs extracted from BIOML

Case	Query	n-Cycles	DTD Graph
2a	`gene//locus`	2	Fig. 15a
2b	`gene//locus`	2	Fig. 15b
2c	`gene//dna`	2	Fig. 15b
3a	`gene//locus`	3	Fig. 15c
3b	`gene//locus`	3	Fig. 15d
4a	`gene//locus`	4	Fig. 11b
4b	`gene//dna`	4	Fig. 11b

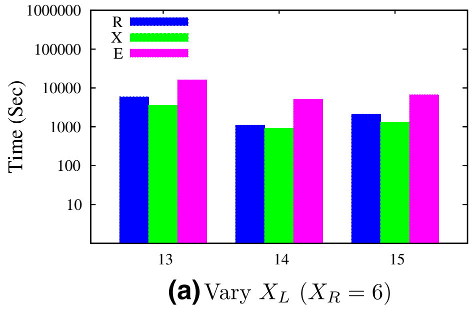
Table 5 Number of operations (min/max/average)

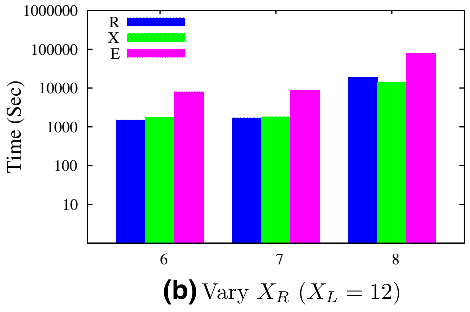
DTD	n	m	c	CycleE		CycleE$_X$	
				LFP	ALL	LFP	ALL
Cross (Fig. 11a)	4	5	2	5/9/6	38/78/51	2/2/2	7/11/8
BIOMLa (Fig. 15a)	4	5	2	8/14/12	80/124/104	3/5/4	12/22/16
BIOMLb (Fig. 15b)	4	6	3	6/14/11	50/94/75	2/5/3	9/20/14
BIOMLc (Fig. 15c)	4	6	3	8/14/12	80/124/104	3/5/4	12/22/16
BIOMLd (Fig. 15d)	4	7	4	8/14/12	88/134/112	3/5/4	13/23/17
GedML (Fig. 11c)	5	11	9	6/22/16	154/222/188	2/8/4	12/27/19

6.5 Exp-5: number of operations

We show the numbers of operations both in the resulting extended XPath expressions obtained from CycleE$_X$ and CycleE and in the resulting relational algebra (RA) in Table 5. Empirically, the lengths of the resulting extended XPath expressions and SQL are polynomial, even though in theory, the sizes of resulting extended XPath expressions are

Fig. 17 `Even//Data` on the extracted 9-cycle GEDML DTD. **a** Vary X_L ($X_R = 6$), **b** vary X_R ($X_L = 12$)





exponential, in the worst case, in terms of the size of $|G_D|$, based on [18].

In Table 5, the first column lists six DTD used in the testing. The second, third, and fourth columns indicate the numbers of nodes (n), edges (m), and simple cycles (c), respectively, in the DTD graphs. For each DTD, we enumerate all possible pairs of two nodes in the DTD, and select one as a start node (A) and the other as an end node (B). For each pair of A and B, we use CycleE and CycleE$_X$ to compute the extended XPath expression representing all paths from A to B, and then determine the number of operations in the resulting relational algebra (RA). They are shown in two groups in Table 5. The LFP and ALL show the numbers of LFP's and all operations used in extended XPath expressions in the format of (min/max/average). CycleE$_X$ outperforms CycleE in terms of the numbers of LFP and all operations used in all the cases.

7 Related work

This is an extension of the earlier work [24] by including (a) the notion of extended XPath (Sect. 2) and its application in query translation and query answering (Sect. 3), (b) revised translation algorithms (Sects. 4, 5), in particular a new algorithm CycleE$_X$ for handling the descendant axis of XPath; and (c) an extensive experimental study. As remarked earlier, extended XPath is useful not only in query translation from XPath to SQL, but also in developing native XML query engines [1,19] and answering XML queries over XML views. In particular, the use of variables in extended XPath allows us to represent each sub-query q in an extended XPath expression only once, no matter where and how often q appears in the query. This yields the low polynomial bound of CycleE$_X$; in contrast, [24] simply adopted Tarjan's algorithm for finding a regular-expression representation of all matching paths, which, in the worst case, may be of an exponential size.

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., [11,16,26,28,29,31,36,39,41,42, 45,57,58,62,64]; see [40] for a comprehensive survey). At least two approaches have been proposed to querying XML data stored in relations. One approach is based on middleware and XML views, and the other is by translating XPath queries to SQL.

The middleware-based approach, e.g., XPERANTO [57,58] and SilkRoute [26], 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. First, it is nontrivial to define a (recursive) XML view of the relational data without loss of the original information (see, e.g., [7,20] for detailed discussion). Second, it requires middleware support and incurs communication overhead between the middleware and the RDBMS. Third, as observed by [39,40], 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 (possibly extended with a recursion operator). This has been studied in two settings: for schema-based XML storage that chooses relational schema by making use of XML schema, and for schema-oblivious XML storage that stores XML data in relations of a fixed schema regardless of XML schema. The schema-based approach allows one to derive efficient relational storage for XML data, retaining the semantic and structural information of the XML data. This is important for, among other things, query optimization, data exchange (see, e.g., [38] for a recent survey), XML access control (e.g., [21]) and XML view updates (e.g., [13]). However, as observed by [40], with the exception of [28,29,39,41,42] and this work, we are aware of no algorithm published for translating recursive XML queries over recursive DTDs to SQL for *schema-based* XML *storage*.

Closest to our work is [39], which proposed the first technique to rewrite recursive path queries over recursive DTDs to SQL for schema-based XML storage. The translation consists of two phases. First, by representing the input DTD D and input XPath query Q as finite state automata, it constructs the product automaton of the two that captures XPath recursion and DTD recursion in a uniform framework. Second, it translates the product automata into a sequence of SQL queries with the SQL'99 recursion operator. This approach has a low polynomial bound $O(|D|^2 * |Q|^4)$ on the product automata generated, which is comparable to our bound on extended XPath queries given in Theorem 4.2. Furthermore, several optimization techniques have also been developed, to eliminate duplicate paths by making part of the automata deterministic, and to optimize SQL queries by leveraging integrity constraints during the translation [41,42]. As remarked earlier, this work differs from [39] in that we use the simple LFP operator, a low-end recursion functionality already supported by many RDBMS, instead of the SQL'99 recursion operator. In addition, we use extended XPath instead of automata. This allows us to handle rich qualifiers and selection paths in an XPath query uniformly rather than treating them separately. Furthermore, the approach presented here can also be used to answer XML queries over certain XML views. On the other hand, as mentioned earlier, the optimization techniques developed for [39], e.g., [41,42], are also applicable to our approach.

For schema-oblivious XML storage, a number of translation and optimization techniques have been proposed. These include path-based techniques that leverage index structures to store root-to-node paths [36,45,64], and interval-based approach, e.g., region encoding [31] and Dewey encoding [62], that maintains structural relationships among elements and their ordering [11,16,28,29,31,62]. MonetDB [11], for example, stores XML data in a "node" relation and associates each node with a pair of preorder traversal and postorder traversal ranks. Leveraging these, XPath recursion ('//') can be efficiently processed in terms of range comparisons, without requiring the support of recursive operators by SQL. This approach is hampered by the following problems. First, most of these techniques are developed for schema-oblivious XML storage and adopt relations of a fixed schema independent of the XML schema. As mentioned earlier, this makes it difficult for, among other things, data exchange, secure XML queries, and update XML views. Second, the indexes and structural coding introduce additional overhead when storing and querying the data. Worse still, the cost of the maintenance of the indexes and coding may become prohibitive expensive when the data is frequently changed (see, e.g., [60], for lower bounds on the maintenance cost). In contrast, our approach does not incur extra cost in the dynamic context. Third, in many applications one would prefer a lightweight tool that provides the capability of answering XPath queries within the immediate reach of commercial RDBMS, instead of using a heavy-duty system. Finally, one cannot use the encoding and indexing approaches to answer XML queries over XML views.

Recently an approach was proposed [28,29] that combines path-based techniques and Dewey encoding, and is applicable to both schema-oblivious XML storage and schema-based XML storage. Leveraging path index and regular expression matching, it introduces a notion of Primitive Path Fragment (PPF), to split XPath expressions and reduce the need for structural joins. Experimental results of [28,29] demonstrated that PPF is effective: an implementation based on PPF outperformed MonetDB/XQuery [11]. We expect that the use of PPF could be also beneficial for translating extended XPath to SQL. It remains to be explored, however, the overhead of maintaining the path indexing structure and Dewey encoding, when the data is updated frequently.

There has also been work on translating XSLT queries [34], XQuery [16,17,47] to SQL. While the algorithms of [16,17, 34,47] cannot handle query translation in the presence of recursive DTDs, their optimization techniques by leveraging, e.g., integrity constraints [17,41], virtual generic schema and query normalization [47], dynamic interval encoding [16] and aggregation handling [34] are complementary to our work. Some of these, along with techniques for query pruning and rewriting [25], minimizing the use of joins [43], multi-query [54] and recursive-query optimization [56], can be incorporated into our translation framework.

There has also been recent work on query answering for virtual XML views in the native XML setting [21,22]. This issue was studied in [21] for nonrecursive XML views, and it was revisited for recursive XML views in [22]. As remarked earlier, it was shown in [22] that for recursive XML views, query rewriting is not closed for XPath, but it is closed for regular XPath; however, the rewriting incurs an exponential-time lower bound even for nonrecursive XML views. To avoid the exponential blowup, [22] proposed a notion of automata to represent the rewritten regular XPath queries, and developed algorithms for evaluating these automata on XML data. Unfortunately, those automata cannot be directly translated into SQL with LFP. In contrast, this work introduces extended XPath and shows that extended XPath expressions can be translated into equivalent SQL queries. Regular XPath was introduced in [48]. Extended XPath proposed by this work is an extension of regular XPath by allowing bindings of variables and sub-queries.

Surveys on recursive and cyclic query processing strategies include [6,35]. For OODBs, [37] introduced techniques for processing cyclic queries restricted to 1-cycle queries. [14] 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 (possibly 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 extended 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 extended XPath expression that characterizes all matching paths, without incurring exponential blowup and better still, optimizing the query by filtering unnecessary computation based on the structural properties of the DTD during the translation; and (3) an efficient algorithm for rewriting an extended XPath expression into an equivalent sequence of SQL queries. These provide not only the capability of answering important XPath queries *within the immediate reach* of most commercial RDBMS, but also the query answering ability for certain XML views.

Several extensions are targeted for future work. First, we recognize that several factors affect the efficiency of the SQL queries produced by our translation algorithms, and we are currently developing a cost model in order to provide better guidance for XPath query rewriting. Second, we are also exploring techniques for multi-query and recursive-query optimization [54,56] to simplify the SQL queries produced. Moreover, we intend to incorporate optimization by means of semantic information such as integrity constraints [17,41,42] and satisfiability analysis of XPath queries in the presence of DTDs [9]. Third, we 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. Finally, a topic for future work is to deal with XML Schema [63] instead of DTDs. As observed by, e.g., [46], a schema in XML Schema is essentially a specialized DTD [53], an extension of DTDs by allowing several productions to be associated with the same element type A and "specializing" the choice of its productions based on the context in which A appears. More specifically, a specialized DTD D over element types Ele is a triple (Ele', D', g), where $Ele \subseteq Ele'$, g is a mapping $Ele' \mapsto Ele$, and D' is a DTD over Ele'. An XML tree T conforms to D if there exists an XML tree T' that satisfies D' and moreover, $T = g(T')$. Note that the mapping g can be encoded in terms of disjunctive production rules (see, e.g., [23] for the encoding), which our translation algorithms can already handle. The connection between specialized DTDs and DTDs allows us to adapt our techniques to translate XPath queries over XML Schema to SQL queries without significant degradation in performance.

Acknowledgments The work described in this paper was supported by grants of EPSRC GR/S63205/01, GR/T27433/01, and EP/E029 213/1, and the grants of Research Grants Council of the Hong Kong SAR, China (No. 418205, No. 418206).

References

1. Afanasiev, L., Grust, T., Marx, M., Rittinger, J., Teubner, J.: An inflationary fixed point in XQuery. In: Proc. of ICDE (2008)
2. Agrawal, R., Devanbu, P.: Moving selections into linear least fixpoint queries. In: Proc. of ICDE (1988)
3. Aho, A., Ullman, J.: Universality of data retrieval languages. In: Proc. of POPL (1979)
4. Amer-Yahia, S., Cho, S., Lakshmanan, L., Srivistava, D.: Minimization of tree pattern queries. In: Proc. of SIGMOD (2001)
5. Bancilhon, F., Maier, D., Sagiv, Y., Ullman, J.: Magic sets and other strange ways to implement logic programs. In: Proc. of PODS (1986)
6. Bancilhon, F., Ramakrishnan, R.: An amateur's introduction to recursive query processing strategies. In: Proc. of SIGMOD (1986)
7. Barbosa, D., Freire, J., Mendelzon, A.: Designing information-preserving mapping schemes for XML. In: Prof. of VLDB (2005)
8. Beeri, C., Ramakrishnan, R.: On the power of magic. J. Log. Program **10** (1991)
9. Benedikt, M., Fan, W., Geerts, F.: XPath satisfiability in the presence of DTDs. J. ACM **55**(2) (2008)
10. BIOML. BIOpolymer Markup Language. http://xml.coverpages.org/BIOML-XML-DTD.txt
11. Boncz, P.A., Grust, T., van Keulen, M., Manegold, S., Rittinger, J., Teubner, J.: MonetDB/XQuery: a fast XQuery processor powered by a relational engine. In: Proc. of SIGMOD (2004)
12. Choi, B.: What are real DTDs like. In: Proc. of WebDB (2002)
13. Choi, B., Cong, G., Fan, W., Viglas, S.: Updating recursive XML views of relations. In: Prof. of ICDE (2007)
14. Christophides, V., Cluet, S., Moerkotte, G.: Evaluating queries with generalized path expressions. In: Proc. of SIGMOD (1996)
15. Clark, J., DeRose, S.: XML path language (XPath). W3C Recommendation, Nov 1999
16. DeHaan, D., Toman, D., Consens, M., Ozsu, T.: Comprehensive XQuery to SQL translation using dynamic interval encoding. In: Proc. of SIGMOD (2003)
17. Deutsch, A., Tannen, V.: MARS: A system for publishing XML from mixed and redundant storage. In: Proc. of VLDB (2003)
18. Ehrenfeucht, A., Zeiger, P.: Complexity measures for regular expressions. In: Proc. of STC'74 (1974)
19. EXSLT.: http://www.exslt.org/dyn/functions/closure/index.html
20. Fan, W., Bohannon, P.: Information preserving XML schema embedding. TODS **33**(1) (2008)
21. Fan, W., Chan, C.-Y., Garofalakis, M.: Secure XML querying with security views. In: Proc. of SIGMOD (2004)
22. Fan, W., Geerts, F., Jia, X., Kementsietsidis, A.: Rewriting regular XPath queries on XML views. In: Proc. of ICDE (2007)
23. Fan, W., Geerts, F., Neven, F.: Expressiveness and complexity of XML publishing transducers. TODS (2008)
24. Fan, W., Yu, J.X., Lu, H., Lu, J., Rastogi, R.: Query translation from XPath to SQL in the presence of recursive DTDs. In: Proc. of VLDB (2005)
25. Fernandez, M., Suciu, D.: Optimizing regular path expression using graph schemas. In: Proc. of ICDE (1998)
26. Fernandez, M.F., Morishima, A., Suciu, D.: Efficient evaluation of XML middleware queries. In: Proc. of SIGMOD (2001)
27. GedML.: Genealogy Markup Language. http://xml.coverpages.org/gedml-dtd9808.txt
28. Georgiadis, H., Vassalos, V.: Improving the efficiency of XPath execution on relational systems. In: Proc. of EBDT (2006)
29. Georgiadis, H., Vassalos, V.: XPath on steroids: exploiting relational engines for XPath performance. In: Proc. of SIGMOD (2007)
30. Graefe, G.: Query evaluation techniques for large databases. ACM Comput. Surv. **25**(2) (1993)
31. Grust, T., van Keulen, M., Teubner, J.: Accelerating XPath evaluation in any RDBMS. TODS **29**, 91–131 (2004)
32. Halevy, A.Y.: Theory of answering queries using views. SIGMOD Record **29**(4) (2001)
33. IBM.: DB2 XML Extender. http://www-3.ibm.com/software/data/db2/extended/xmlext/index.html
34. Jain, S., Mahajan, R., Suciu, D.: Translating XSLT programs to efficient SQL querie. In: Proc. of WWW (2002)
35. Kambayashi, Y.: Processing cyclic queries. In: Query processing in database systems, pp. 63–78. Springer, Heidelberg (1985)
36. Kha, D.D., Yoshikawa, M., Uemura, S.: An XML indexing structure with relative region coordinate. In: Proc. of ICDE (2001)
37. Kim, Y.-C., Kim, W., Dale, A.: Cyclic query processing in object-oriented databases. In: Proc. of ICDE (1989)
38. Kolaitis, P.G.: Schema mappings, data exchange, and metadata management. In: Prof. of PODS (2006)
39. Krishnamurthy, R., Chakaravarthy, V.T., Kaushik, R., Naughton, J.: Recursive XML schemas, recursive XML queries, and relational storage: XML-to-SQL query translation. In: Proc. of ICDE (2004)
40. Krishnamurthy, R., Kaushik, R., Naughton, J.: XML-SQL query translation literature: The state of the art and open problems. In: Proc. of Xsym (2003)
41. Krishnamurthy, R., Kaushik, R., Naughton, J.: Efficient XML-to-SQL query translation: Where to add the intelligence. In: Proc. of VLDB (2004)
42. Krishnamurthy, R., Kaushik, R., Naughton, J.: XML views as integrity constraints and their use in query translation. In: Proc. of ICDE (2005)
43. Kunen, I.K., Suciu, D.: A scalable algorithm for query minimization. Technical Report, University of Washington (2004)
44. Lenzerini, M.: Data integration: A theoretical perspective. In: Proc. of PODS (2002)

45. Li, Q., Moon, B.: Indexing and querying XML data for regular path expressions. In: Proc. of VLDB (2001)
46. Likin, L.: Logics for unranked trees: An overview. Log. Meth. Comput. Sci. **2**(3) (2006)
47. Manolescu, I., Florescu, D., Kossmann, D.: Answering XML queries on heterogeneous data sources. In: Proc. of VLDB (2001)
48. Marx, M.: XPath with conditional axis relations. In: Proc. of EDBT (2004)
49. Microsoft.: SQLXML and XML Mapping Technologies. http://msdn.microsoft.com/sqlxml/default.asp
50. Mishra, P., Eich, M.H.: Join processing in relational databases. ACM Comput. Surv. **24**(1) (1992)
51. Nunn, M.: 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
52. Oracle.: Oracle9i XML Database Developer's Guide—Oracle XML DB Release 2. http://otn.oracle.com/tech/xmldb/content.html
53. Papakonstantinou, Y., Vianu, V.: Type inference for views of semi-structured data. In: Proc. of PODS (2000)
54. Roy, P., Seshadri, S., Sudarshan, S., Bhobe, S.: Efficient algorithms for multi query optimization. In: Proc. of SIGMOD (2000)
55. Saxon.: http://www.saxonica.com/
56. Shan, M.-C., Neimat, M.-A.: Optimization of relational algebra expressions containing recursion operators. In: Proc. of ACM Annual Computer Science Conference (1999)
57. Shanmugasundaram, J., Kiernan, J., Shekita, E.J., Fan, C., Funderburk, J.: Querying XML views of relational data. In: Proc. of VLDB (2001)
58. Shanmugasundaram, J., Shekita, E., Barr, R., Carey, M., Lindsay, B., Pirahesh, H., Reinwald, B.: A general techniques for querying XML documents using a relational database system. SIGMOD Record **30**(3) (2001)
59. Shanmugasundaram, J., Tufte, K., He, G., Zhang, C., DeWitt, D., Naughton, J.: Relational databases for querying XML documents: Limitations and opportunities. In: Proc. of VLDB (1999)
60. Silberstein, A., He, H., Yi, K., Yang, J.: BOXes: Efficient maintenance of order-based labeling for dynamic XML data. In: Proc. of ICDE (2005)
61. Tarjan, R.E.: Fast algorithms for solving path problems. J. ACM **28**(3), 594–614 (1981)
62. Tatarinov, I., Viglas, S., Beyer, K.S., Shanmugasundaram, J., Shekita, E.J., Zhang, C.: Storing and querying ordered XML using a relational database system. In: Proc. of SIGMOD (2002)
63. Thompson, H. et al.: XML Schema. W3C Working Draft, May 2001. http://www.w3.org/XML/Schema
64. Zhang, C., Naughton, J., DeWitt, D.J., Luo, Q., Lohman, G.M.: On supporting containment queries in relational database management systems. In: Proc. of SIGMOD'01 (2001)