Selectively Storing XML Data in Relations

Selectively Storing XML Data in Relations

Wenfei Fan[1] and Lisha Ma[2]

[1] University of Edinburgh and Bell Laboratories
[2] Heriot-Watt University

Abstract. This paper presents a new framework for users to select relevant data from an XML document and store it in an existing relational database, as opposed to previous approaches that shred the entire XML document into a newly created database of a newly designed schema. The framework is based on a notion of XML2DB mappings. An XML2DB mapping extends a (*possibly recursive*) DTD by associating element types with semantic attributes and rules. It extracts either part or all of the data from an XML document, and generates SQL updates to increment an existing database using the XML data. We also provide an efficient technique to evaluate XML2DB mappings in parallel with SAX parsing. These yield a systematic method to store XML data selectively in an existing database.

1 Introduction

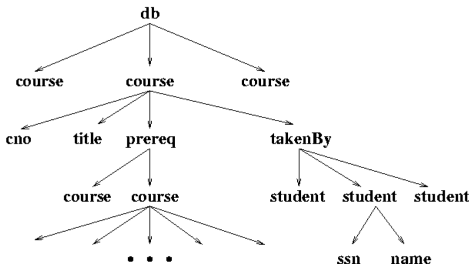
A number of approaches have been proposed for shredding XML data into relations [3,6,14,15], and some of these have found their way into commercial systems [10,7,13]. Most of these approaches map XML data to a newly created database of a "canonical" relational schema that is designed starting from scratch based on an XML DTD, rather than storing the data in an existing database. Furthermore, they often store the entire XML document in the database, rather than letting users select and store part of the XML data. While some commercial systems allow one to define schema-based mappings to store part of the XML data in relations, either their ability to handle recursive DTDs is limited [7,10] or they do not support storing the data in an existing database [13]. In practice, it is common that users want to specify what data they want in an XML document, and to increment an existing database with the selected data. Moreover, one often wants to define the mappings based on DTDs, which may be recursive as commonly found in practice (see [4] for a survey of real-life DTDs).

Example 1.1. Consider a *registrar* database specified by the relational schema R_0 shown in Fig. 1(a) (with keys underlined). The database maintains *student* data, *enroll*ment records, *course* data, and a relation *prereq*, which gives the prerequisite hierarchy of courses: a tuple *(c1, c2)* in *prereq* indicates that *c2* is a prerequisite of *c1*.

Now consider an XML DTD D_0 also shown in Fig. 1(a) (the definition of elements whose type is PCDATA is omitted). An XML document conforming to D_0 is depicted in Fig. 1(b). It consists of a list of *course* elements. Each *course* has a *cno* (course number), a course *title*, a *prereq*uisite hierarchy, and all the *student*s who have registered for the course. Note that the DTD is recursive: *course* is defined in terms of itself via *prereq*.

```
Relational schema R0:
course(cno, title),
student(ssn, name),
enroll(ssn, cno),
prereq(cno1, cno2).

DTD D0:
<!ELEMENT db      (course*)>
<!ELEMENT course  (cno, title,
                   prereq, takenBy)>
<!ELEMENT prereq  (course*)>
<!ELEMENT takenBy (student*)>
<!ELEMENT student (ssn, name)>
```



(a) Relational schema R_0 and DTD D_0 (b) An XML document of D_0

Fig. 1. Relational Schema R_0, DTD D_0 and an example XML document of D_0

We want to define a mapping σ_0 that, given an XML document T that conforms to D_0 and a relational database I of R_0, (a) extracts from T all the CS *course*s, along with their *prereq*uisites hierarchies and *student*s registered for these related courses, and (b) inserts the data into relations *course*, *student*, *enroll* and *prereq* of the relational database I, respectively. Observe the following. (a) We only want to store in relations a certain part of the data in T, instead of the entire T. (b) The selected XML data is to be stored in an existing database I of a predefined schema R_0 by means of SQL updates, rather than in a newly created database of a schema designed particularly for D_0. (c) The selected XML data may reside at arbitrary levels of T, whose depth cannot be determined at compile time due to the recursive nature of its DTD D_0. To our knowledge, no existing XML shredding systems are capable of supporting σ_0. □

Contributions. To overcome the limitations of existing XML shredding approaches, we propose a new framework for mapping XML to relations. The framework is based on (a) a notion of XML2DB *mappings* that extends (*possibly recursive*) DTDs and is capable of mapping either part of or an entire document to relations, and (b) a technique for efficiently evaluating XML2DB mappings.

XML2DB mappings are a novel extension of attribute grammars (see, *e.g.,* [5] for attribute grammars). In a nutshell, given a (*possibly recursive*) XML DTD D and a predefined relational schema R, one can define an XML2DB mapping $\sigma : D \rightarrow R$ to select data from an XML document of D, and generates SQL inserts to increment an existing relational database of R. More specifically, σ extends the DTD D by associating semantic attributes and rules with element types and their definitions in D. Given an XML document T of D, σ traverses T, selects data from T, and generates SQL inserts Δ by means of the semantic attributes and rules during the traversal. Upon the completion of the traversal, the SQL updates Δ are executed against an existing database I of R, such that the updated database $\Delta(I)$ includes the extracted XML data and is guaranteed to be an instance of the predefined schema R. For example, we shall express the mapping σ_0 described in Example 1.1 as an XML2DB mapping (Fig. 2(a)).

To evaluate efficiently an XML2DB mapping σ, we propose a technique that combines the evaluation with the parsing of XML data by leveraging existing SAX [9] parsers. This allows us to generate SQL updates Δ in a single traversal of the document without incurring extra cost. To verify the effectiveness and efficiency of our technique we provide a preliminary experimental study.

Taken together, the main contributions of the paper include the following:

- A notion of XML2DB mappings, which allow users to increment an existing relational database by using certain part or all of the data in an XML document, and are capable of dealing with (*possibly recursive*) XML DTDs in a uniform framework.
- An efficient technique that seamlessly integrates the evaluation of XML2DB mappings and SAX parsing, accomplishing both in a single pass of an XML document.
- An experimental study verifying the effectiveness of our techniques.

The novelty of our uniform framework consists in (a) the functionality to support mappings based on (*possibly recursive*) DTDs from XML to relations that, as opposed to previous XML shredding approaches, allows users to map *either part of or the entire* XML document to a relational database, rather than core-dumping the entire document; (b) the ability to extend an *existing* relational database of a *predefined* schema with XML data rather than creating a new database starting from scratch; (c) efficient evaluation techniques for XML2DB mappings via *a mild extension of* SAX *parsers* for XML.

Organization. Section 2 reviews DTDs and SAX. Section 3 defines XML2DB mappings. Section 4 presents the evaluation technique. A preliminary experimental study is presented in Section 5, followed by related work in Section 6 and conclusions in Section 7.

2 Background: DTDs and SAX

DTDs. Without loss of generality, we formalize a DTD D to be (E, P, r), where E is a finite set of *element types*; r is in E and is called the *root type*; P defines the element types: for each A in E, $P(A)$ is a regular expression α defined by PCDATA $\mid \epsilon \mid B_1, \ldots, B_n \mid B_1 + \ldots + B_n \mid B^*$, where ϵ is the empty word, B is a type in E (referred to as a *child type* of A), and '$+$', '$,$' and '$*$' denote disjunction, concatenation and the Kleene star, respectively (we use '$+$' instead of '$|$' to avoid confusion). We refer to $A \rightarrow P(A)$ as the *production* of A. A DTD is *recursive* it has an element type defined (directly or indirectly) in terms of itself.

Note that [2] all DTDs can be converted to this form in linear time by using new element types and performing a simple post-processing step to remove the introduced element types. To simplify the discussion we do not consider XML attributes, which can be easily incorporated. We also assume that the element types $B_1, \ldots, B_n$ in $B_1, \ldots, B_n$ (resp. $B_1 + \ldots + B_n$) are distinct, w.l.o.g. since we can always distinguish repeated occurrences of the same element type by referring to their positions in the production.

SAX Parsing. A SAX [9] parser reads an XML document T and generates a stream of SAX events of five types: startDocument(), startElement(A, eventNo), text(s), endElement(A), endDocument(), where A is an element type of T and s is a string (PCDATA). The semantics of these events is self-explanatory.

3 XML2DB Mappings: Syntax and Semantics

In this section we formally define XML2DB mappings.

Syntax. The idea of XML2DB mappings is to treat the XML DTD as a grammar and extend the grammar by associating semantic rules with its productions. This is in the same spirit of Oracle XML DB [13] and IBM DB2 XML Extender [7], which specify XML shredding by annotating schema for XML data. When the XML data is parsed *w.r.t.* the grammar, it recursively invokes semantic rules associated with the productions of the grammar to select relevant data and generate SQL updates.

We now define XML2DB mappings. Let $D = (E, P, r)$ be a DTD and R be a relational schema consisting of relation schemas $R_1, \ldots, R_n$. An XML2DB mapping $\sigma : D \to R$ takes as input an XML document T of D, and returns an SQL group of inserts Δ which, when executed on a database I of R, yields an incremented instance ΔI of schema R. The mapping extracts relevant data from T and uses the data to construct tuples to be inserted into I. More specifically, σ is specified as follows.

- For each relation schema R_i of R, σ defines a *relation variable* Δ_{R_i}, which is to hold the set of tuples to be inserted into an instance I_i of R_i. The set Δ_{R_i} is initially empty and is gradually incremented during the parsing of the XML document T.
- For each element type A in E, σ defines a semantic attribute $\$A$ whose value is either a relational tuple of a fixed arity and type, or a special value $\top$ (denoting $\$r$ at the root r) or $\perp$ (denoting undefined); intuitively, $\$A$ extracts and holds relevant data from the input XML document that is to be inserted into the relational database I of R. As will be seen shortly, $\$A$ is used to pass information top-down during the evaluation of σ.
- For each production $p = A \to \alpha$ in D, σ specifies a set of semantic rules, denoted by $rule(p)$. These rules specify two things: (a) how to compute the value of the semantic attribute $\$B$ of B children of an A element for each child type B in α, (b) how to increment the set in Δ_{R_i}; both $\$B$ and Δ_{R_i} are computed by using the semantic attribute $\$A$ and the PCDATA of text children of the A element (if any). More specifically, $rule(p)$ consists of a sequence of *assignment* and *conditional statements*:

$rule(p)$	:=	statements
statements	:=	ϵ | statement; statements
statement	:=	X := expression | if C then statements else statements

  where ϵ denotes the empty sequence (*i.e.,* no semantic actions); and X is either a relation variable Δ_{R_i} or a semantic attribute $\$B$. The assignment statement has one of two forms. (a) $\$B := (x_1, \ldots, x_k)$, *i.e.,* tuple construction where x_i is either of the form $\$A.a$ (projection on the a field of the tuple-valued attribute $\$A$ of the A element), or val (B'), where B' is an element type in α such that it precedes B in α (*i.e.,* we enforce sideways information passing *from left to right*), B's production is of the form $B' \to$ PCDATA, and val (B') denotes the PCDATA (string) data of B' child. (b) $\Delta_{R_i} := \Delta_{R_i} \cup \{(x_1, \ldots, x_k)\}$, where $(x_1, \ldots, x_k)$ is a tuple as constructed above and in addition, it is required to have the same arity and type as specified by the schema R_i. The condition C is defined in terms of equality or

```
Relational variables: Δcourse, Δprereq, Δstudent,
        Δenroll, with ∅ as their initial value.
Semantic rules:
db → course*
  $course := ⊤;
course → cno, title, prereq, takenBy
  if val (cno) contains 'CS' or ($course ≠ ⊥
                          and $course ≠ ⊤)
  then $prereq := val (cno);  $takenBy := val (cno);
       Δcourse := Δcourse ∪ {(val (cno), val (title))};
       if $course ≠ ⊤ and $course ≠ ⊥
       then Δprereq := Δprereq ∪ {($course, val (cno))};
  else $title := ⊥;  $prereq := ⊥;  $takenBy := ⊥;
prereq → course*
  $course := $prereq;
takenBy → student*
  $student := $takenBy;
student → ssn, name
  if $student ≠ ⊥
  then Δstudent := Δstudent ∪ {(val (ssn), val (name))};
       Δenroll  := Δenroll ∪ {(val (ssn), $student)};
```

(a) XML2DB mapping σ_0

```
Relational variables: Δcourse, Δprereq, Δstudent,
        Δenroll, with ∅ as their initial value.
Semantic rules:
db → course*
  $course := ⊤;
course → cno, title, prereq, takenBy
  $prereq := val (cno);  $takenBy := val (cno);
  Δcourse := Δcourse ∪ {(val (cno), val (title))};
  if $course ≠ ⊤
  then Δprereq := Δprereq ∪ {($course, val (cno))};
prereq → course*
  $course := $prereq;
takenBy → student*
  $student := $takenBy;
student → ssn, name
  Δstudent := Δstudent ∪ {(val (ssn), val (name))};
  Δenroll  := Δenroll ∪ {(val (ssn), $student)};
```

(b) XML2DB mapping σ_1

Fig. 2. Example XML2DB mappings: storing part of (σ_0) and the entire (σ_1) the document

string containment tests on atomic terms of the form val (B'), $\$A.a$, $\top$, $\bot$, and it is built by means of Boolean operators and, or and not, as in the standard definition of the selection conditions in relational algebra. The mapping σ is said to be *recursive* if the DTD D is recursive.

We assume that if p is of the form $A \to B^*$, $rule(p)$ includes a single rule $\$B := \$A$, while the rules for the B production select data in each B child. This does not lose generality as shown in the next example, in which a list of *student* data is selected.

Example 3.2. The mapping σ_0 described in Example 1.1 can be expressed as the XML2DB mapping $\sigma_0 : D_0 \to R_0$ in Fig. 2(a), which, given an XML document T of the DTD D_0 and a relational database I of the schema R_0, extracts all the CS courses, their prerequisites and their registered students from T, and inserts the data as tuples into I. That is, it generates Δ_{course}, Δ_{student}, Δ_{enroll} and Δ_{prereq}, from which SQL updates can be readily constructed. Note that a *course* element c is selected if either its *cno* contains 'CS' or an ancestor of c is selected; the latter is captured by the condition *(\$course* $\neq \bot$ and *\$course* $\neq \top$*)*. The special value $\bot$ indicates that the corresponding elements are not selected and do not need to be processed. Note that the rules for *takenBy* and *student* select the data of *all* student who registered for such courses. □

Semantics. We next give the operational semantics of an XML2DB mapping $\sigma : D \to R$ by presenting a conceptual evaluation strategy. This strategy aims just to illustrate the semantics; a more efficient evaluation algorithm will be given in the next section.

Given an input XML document (tree) T, $\sigma(T)$ is computed via a top-down depth-first traversal of T, starting from the root r of T. Initially, the semantic attribute $\$r$ of r is assigned the special value $\top$. For each element v encountered during the traversal, we do the following. (1) Identify the element type of v, say, A, and find the production $p = A \rightarrow P(A)$ from the DTD D and the associated semantic rules $rule(p)$ from the mapping σ. Suppose that the tuple value of the semantic attribute $\$A$ of v is t. (2) Execute the statements in $rule(p)$. This may involve extracting PCDATA value val (B') from some B' children, projecting on certain fields of the attribute t of v, and performing equality, string containment tests and Boolean operations, as well as constructing tuples and computing union of sets as specified in $rule(p)$. The execution of $rule(p)$ assigns a value to the semantic attribute $\$B$ of each B child of v if the assignment of $\$B$ is defined in $rule(p)$, and it may also increment the set Δ_{R_i}. In particular, if p is of the form $A \rightarrow B^*$, then each B child u of v is assigned the same value $\$B$. (3) We proceed to process each child u of v in the same way, by using the semantic attribute value of u. (4) The process continues until all the elements in T are processed. Upon the completion of the process, we return the values of relation variables $\Delta_{R_1}, \ldots, \Delta_{R_n}$ as output, each of which corresponds to an SQL insert. More specifically, for each Δ_i, we generate an SQL insert statement:

$$\text{insert into } R_i \quad \text{select } * \quad \text{from } \Delta_{R_i}$$

That is, at most n SQL inserts are generated in total.

Example 3.3. Given an XML tree T as shown in Fig 1(b), the XML2DB mapping σ_0 of Example 3.2 is evaluated top-down as follows. (a) All the *course* children of the root of T are given $\top$ as the value of their semantic attribute \$*course*. (b) For each *course* element v encountered during the traversal, if either \$*course* contains 'CS' or it is neither $\perp$ nor $\top$, *i.e.,* v is either a CS course or a prerequisite of a CS course, the PCDATA of *cno* of v is extracted and assigned as the value of \$*title*, \$*prereq* and \$*takenBy*; moreover, the set Δ_{course} is extended by including a new tuple describing the course v. Furthermore, if \$*course* is neither $\top$ nor $\perp$, then Δ_{prereq} is incremented by adding a tuple constructed from \$*course* and val (*cno*), where \$*course* is the *cno* of c *inherited* in the top-down process. Otherwise the data in v is not to be selected and thus all the semantic attributes of its children are given the special value $\perp$. (c) For each *prereq* element u encountered, the semantic attributes of all the *course* children of u *inherit* the \$*prereq* value of u, which is in turn the *cno* of the *course* parent of u; similarly for *takenBy* elements. (d) For each *student* element s encountered, if \$*student* is not $\perp$, *i.e.,* s registered for either a CS course c or a prerequisite c of a CS course, the sets Δ_{student} and Δ_{enroll} are incremented by adding a tuple constructed from the PCDATA val (*ssn*), val (*name*) of s and the semantic attribute \$*student* of s; note that \$*student* is the *cno* of the course c. (e) After all the elements in T are processed, the sets Δ_{course}, Δ_{student}, Δ_{enroll} and Δ_{prereq} are returned as the output of $\sigma_0(T)$. □

Handling Recursion in a DTD. As shown by Examples 3.2 and 3.3 XML2DB mappings are capable of handling recursive DTDs. In general, XML2DB mappings handle recursion in a DTD following a data-driven semantics: the evaluation is determined by the input XML tree T at run-time, and it always *terminates* since T is finite.

Storing Part of an XML Document in Relations. As demonstrated in Fig. 2(a), users can specify in an XML2DB mapping what data they want from an XML document and store only the selected data in a relational database.

Shredding the Entire Document. XML2DB mappings also allow users to shred the entire input XML document into a relational database, as shown in Fig. 2(b). Indeed, for any XML document T of the DTD D_0 given in Example 1.1, the mapping σ_1 shreds the entire T into a database of the schema R_0 of Example 1.1.

Taken together, XML2DB mappings have several salient features. (a) They can be evaluated in a *single* traversal of the input XML tree T and it visits each node *only once*, even if the embedded DTD is recursive. (b) When the computation terminates it generates sets of tuples to be inserted into the relational database, from which SQL updates Δ can be readily produced. This allows users to update an existing relational database of a predefined schema. (c) The semantic attributes of children nodes *inherit* the semantic attribute of their parent; in other words, semantic attributes pass the information and control top-down during the evaluation. (d) XML2DB mappings are able to store either part of or the entire XML document in a relational database, in a *uniform framework*.

4 Evaluation of XML2DB Mappings

We next outline an algorithm for evaluating XML2DB mappings $\sigma : R \rightarrow D$ in parallel with SAX parsing, referred to as an *extended* SAX *parser*. Given an XML document T of the DTD D, the computation of $\sigma(T)$ is combined with the SAX parsing process of T.

The algorithm uses the following variables: (a) a relation variable Δ_{R_i} for each table R_i in the relational schema R; (b) a stack S, which is to hold a semantic attribute $\$A$ during the evaluation (parsing); and (c) variables X_j of string type, which are to hold PCDATA of text children of each element being processed, in order to construct tuples to be added to Δ_{R_i}. The number of these variables is bounded by the longest production in the DTD D, and the same string variables are repeatedly used when processing different elements. Recall the SAX events described in Section 2. The extended SAX parser incorporates the evaluation of σ into the processing of each SAX event, as follows.

- startDocument(). We push the special symbol $\top$ onto the stack S, as the value of the semantic attribute $\$r$ of the root r of the input XML document T.
- startElement(A, eventNo). When an A element v is being parsed, the semantic attribute $\$A$ of v is already at the top of the stack S. For each child u of v to be processed, we compute the semantic attribute $\$B$ of u based on the semantic rules for $\$B$ in $rule(p)$ associated with the production $p = A \rightarrow P(A)$; we push the value onto S, and proceed to process the children of u along with the SAX parsing process. If the production of the type B of u is $B \rightarrow$ PCDATA, the PCDATA of u is stored in a string variable X_j. Note that by the definition of XML2DB mappings, the last step is only needed when p is of the form $A \rightarrow B_1, \ldots, B_n$ or $A \rightarrow B_1 + \ldots + B_n$.
- endElement(A). A straightforward induction can show that when this event is encountered, the semantic attribute $\$A$ of the A element being processed is at the top of the stack S. The processing at this event consists of two steps. We first increment the set Δ_{R_i} by executing the rules for Δ_{R_i} in $rule(p)$, using the value $\$A$ and the PCDATA values stored in string variables. We then pop $\$A$ off the stack.
- text(s). We store PCDATA s in a string variable if necessary, as described above.

- endDocument(). At this event we return the relation variables Δ_{R_i} as the output of $\sigma(T)$, and pop the top of the stack off S. This is the last step of the evaluation of $\sigma(T)$.

Upon the completion of the extended SAX parsing process, we eliminate duplicates from relation variables Δ_{R_i}s, and convert Δ_{R_i} to SQL insert command Δ_is.

Example 4.4. We now revisit the evaluation of $\sigma_0(T)$ described in Example 3.3 using the extended SAX parser given above. (a) Initially, $\top$ is pushed onto the stack S as the semantic attribute $db of the root *db* of the XML tree T; this is the action associated with the SAX event startDocument(). The extended SAX parser then processes the *course* children of *db*, pushing $\top$ onto S when each *course* child v is encountered, as the semantic attribute $course of v. (b) When the parser starts to process a *course* element v, the SAX event startElement(course, eNo) is generated, and the semantic attribute $course of v is at the top of the stack S. The parser next processes the *cno* child of v, extracting its PCDATA and storing it in a string variable X_j; similarly for *title*. It then processes the *prereq* child of u, computing $prereq by means of the corresponding rule in $rule(course)$; similarly for the *takenBy* child of v. After all these children are processed and their semantic attributes popped off the stack, endElement(course) is generated, and at this moment the relation variables Δ_{course} and Δ_{prereq} are updated, by means of the corresponding rules in $rule(course)$ and by using $course at the top of S as well as val (*cno*) and val (*title*) stored in string variables. After this step the semantic attribute $course of v is popped off the stack. Similarly the SAX events for *prereq* and *takenBy* are processed. (c) When endDocument() is encountered, the sets Δ_{course}, Δ_{student}, Δ_{enroll} and Δ_{prereq} are returned as the output of $\sigma_0(T)$. □

Theorem 4.1. *Given an* XML *document T and an* XML2DB *mapping $\sigma : D \rightarrow R$, the extended* SAX *parser computes $\sigma(T)$ via a single traversal of T and in $O(|T||\sigma|)$ time, where $|T|$ and $|\sigma|$ are the sizes of T and σ, respectively.* □

5 Experimental Study

Our experimental study focuses on the scalability of our extended SAX parser, denoted by ESAX, which incorporates the XML2DB mapping evaluation. We conducted two sets of experiments: we ran ESAX and the original SAX parser (denoted by SAX) (a) on XML documents T of increasing sizes, and (b) on documents T with a fixed size but different shapes (depths or widths). Our experimental results showed (a) that ESAX is linearly scalable and has the same behavior as SAX, and (b) the performance of ESAX is only determined by $|T|$ rather than the shape of T. The experiments were conducted on a PC with a 1.40 Ghz Pentium M CPU and 512MB RAM, running Windows XP. Each experiment was repeated 5 times and the average is reported here; we do not show confidence interval since the variance is within 5%.

We built XML documents of the DTD of Fig. 1(a), using the Toxgene XML generator (http://www.cs.toronto.edu/tox/toxgene). We used two parameters, X_L and X_R, where X_L is the depth of the generated XML tree T, and X_R is the maximum number of children of any node in T. Together X_L and X_R determine the shape of T: the larger the X_L value, the deeper the tree; and the larger the X_R value, the wider the tree.

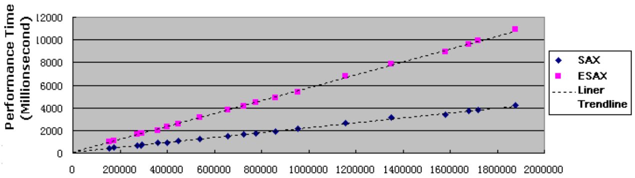


Fig. 3. Scalability with the size of XML document T: vary $|T|$

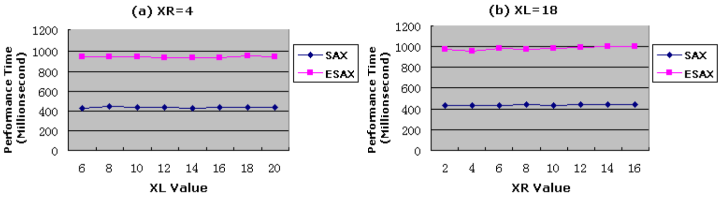


Fig. 4. Scalability with the shape of XML document T: vary X_L and X_R with a fixed $|T|$

Figure 3 shows the scalability of ESAX by increasing the XML dataset size from 153505 elements (3M) to 1875382 (39M). The time (in ms) reported for ESAX includes the parsing and evaluation time of XML2DB mapping. As shown in Fig. 3, ESAX is linearly scalable and behaves similarly to SAX, as expected. Furthermore, the evaluation of an XML2DB mapping does not incur a dramatic increase in processing time vs. SAX.

To demonstrate the impact of the shapes of XML documents on the performance of ESAX, we used XML documents T of a fixed size of 160,000 elements, while varying the height (X_L) and width (X_R) of T. Figure 4 (a) shows the elapsed time when varying X_L from 8 to 20 with $X_R = 4$, and Fig. 4(b) shows the processing time while varying X_R from 2 to 16 with X_L = 12. The results show that ESAX takes roughly the same amount of time on these documents. This verifies that the time-complexity of ESAX is solely determined by $|T|$ rather than the shape of T, as expected.

6 Related Work

Several approaches have been explored for using a relational database to store XML documents, either DTD-based [3,15,13,7] or schema-oblivious [6,14,10] (see [8] for a survey). As mentioned in section 1, except [10,7] these approaches map the entire XML document to a newly created database of a "canonical" relational schema, and are not capable of extending an existing database with part of the data from the document.

Microsoft SQL 2005 [10] supports four XML data-type methods QUERY(), VALUE(), EXIST() and NODES(), which take an XQuery expression as argument to retrieve parts of an XML instance. However, the same method is not able to shred the entire document into relations via a single pass of an XML document, in contrast to our uniform framework to store either the entire or part of an XML document. Furthermore, it does not support semantic-based tuple construction, *e.g.*, when constructing a tuple (a, b), it does not allow one to extract attribute b based on the extracted value of a, which is supported by XML2DB mappings via semantic-attribute passing. Both Oracle XML DB [13] and IBM DB2 XML Extender [7] use schema annotations to map either the entire or parts of XML instances to relations. While Oracle supports recursive DTDs, it cannot increment an existing database. Worse, when an element is selected, the entire element has to be stored. IBM employs user-defined *Document Access Definitions* (DADs) to map XML data to DB2 tables, but supports only fixed-length DTD recursion (see also [12] for upcoming XML support in DB2). Neither Oracle nor IBM supports semantic-based tuple construction, which is commonly needed in practice.

We now draw the analogy of XML2DB mappings to attribute grammars (see, *e.g.,* [5]). While the notion of XML2DB mappings was inspired by attribute grammars, it is quite different from attribute grammars and their previous database applications [11]. First, an attribute grammar uses semantic attributes and rules to constrain the parsing of strings, whereas an XML2DB mapping employs these to control the generation of database updates. Second, an attribute grammar outputs a parse tree of a string, whereas an XML2DB mapping produces SQL updates.

Closer to XML2DB mappings are the notion of AIGs [2] and that of structural schema [1], which are also nontrivial extensions of attribute grammars. AIGs are specifications for schema-directed XML integration. They differ from XML2DB mappings in that they generate XML trees by extracting data from relational sources. Furthermore, the evaluation of AIGs is far more involved than its XML2DB mapping counterpart. Structural schemas were developed for querying text files, by extending context-free grammars with semantic attributes. The evaluation of structural schemas is different from the SAX-parser extension of XML2DB mappings.

7 Conclusion

We have proposed a notion of XML2DB mappings that in a uniform framework, allows users to select either part of or an entire XML document and store it in an existing relational database of a predefined schema, as opposed to previous XML shredding approaches that typically shred the entire document into a newly created database of a new schema. Furthermore, XML2DB mappings are capable of supporting recursive DTDs and flexible tuple construction. We have also presented an efficient algorithm for evaluating XML2DB mappings based on a mild extension of SAX parsers. Our preliminary experimental results have verified the effectiveness and efficiency of our technique. This provides existing SAX parsers with immediate capability to support XML2DB mappings.

We are extending XML2DB mappings by incorporating (a) the support of SQL queries and (b) the checking of integrity constraints (*e.g.,* keys and foreign keys) on the underlying relational databases. We are also developing evaluation and optimization techniques to cope with and leverage SQL queries and constraints.

Acknowledgment. Wenfei Fan is supported in part by EPSRC GR/S63205/01, GR/T27433/01 and BBSRC BB/D006473/1. We thank Hamish Taylor for the helpful comments.

References

1. S. Abiteboul, S. Cluet, and T. Milo. Querying and updating the file. In *VLDB*, 1993.
2. M. Benedikt, C. Y. Chan, W. Fan, J. Freire, and R. Rastogi. Capturing both types and constraints in data integration. In *SIGMOD*, 2003.
3. P. Bohannon, J. Freire, J. Haritsa, M. Ramanath, P. Roy, and J. Siméon. Bridging the XML relational divide with LegoDB. In *ICDE*, 2003.
4. B. Choi. What are real DTDs like. In *WebDB*, 2002.
5. P. Deransart, M. Jourdan, and B. Lorho (eds). Attribute grammars. *LNCS*, 1988.
6. D. Florescu and D. Kossmann. Storing and querying XML data using an RDMBS. *IEEE Data Eng. Bull*, 1999.
7. IBM. DB2 XML Extender. *http://www-3.ibm.com/software/data/db2/extended/xmlext/.*
8. R. Krishnamurthy, R. Kaushik, and J. Naughton. XML-SQL query translation literature: The state of the art and open problems. In *Xsym*, 2003.
9. D. Megginson. SAX: A simple API for XML. *http://www.megginson.com/SAX/.*
10. Microsoft. XML support in Microsoft SQL server 2005, December 2005. *http://msdn.microsoft.com/library/en-us/dnsql90/html/sql2k5xml.asp/.*
11. F. Neven. Extensions of attribute grammars for structured document queries. In *DBPL*, 1999.
12. M. Nicola and B. Linden. Native XML support in DB2 universal database. In *VLDB*, 2005.
13. Oracle. Oracle database 10g release 2 XML DB technical whitepaper.
14. A. Schmidt, M. L. Kersten, M. Windhouwer, and F. Waas. Efficient relational storage and retrieval of XML documents. In *WebDB*, 2000.
15. J. Shanmugasundaram, K. Tufte, G. He, C. Zhang, D. DeWitt, and J. Naughton. Relational databases for querying XML documents: Limitations and opportunities. In *VLDB*, 1999.