A Framework of Inlining Algorithms for Mapping DTDs to Relational Schemas.

Kwok-Bun Yue
Department of Computer Science
University of Houston-Clear Lake
yue@cl.uh.edu

Sreenivasan Alakappan
Department of Computer Science
University of Houston-Clear Lake
sreeni@digitalwitness.net

William K. Cheung
Department of Computer Science
Hong Kong Baptist University
william@comp.hkbu.edu.hk

ABSTRACT

As XML becomes ubiquitous quickly, there is a strong need for efficiently storing and querying XML documents. A popular approach is to store them in relational databases to take advantage of their maturity. To reduce the number of joins for XML querying, several inlining algorithms have been proposed to map XML Document Type Definitions (DTD) to relational schemas. These algorithms generally include three steps: (1) simplify the DTD; (2) use the simplified DTD to create a DTD graph; and (3) use the DTD graph to generate the relational schemas. To provide a framework for understanding them, this paper discusses issues that inlining algorithms must resolve. It makes significant contributions to all of the three steps. We have developed and implemented: (1) a more optimal and complete DTD simplification algorithm, (2) an algorithm for creating a generic DTD graph that can be used as an universal basis for inlining algorithms, and (3) a new inlining algorithm that inlines more aggressively than existing algorithms, thus providing better potential in efficiently handling XML queries. The new inlining algorithm also handles issues not clearly considered by existing algorithms.

Keywords

XML, DTD, Relational Schemas, Inlining Algorithms

1. INTRODUCTION

As XML becomes ubiquitous quickly, there is a strong need for efficiently storing and querying XML documents. XML is essentially based on an ordered tree model, which is unlike other popular persistent data models, such as the relational model or the object-oriented model. As a result, new language standards, such as XPath 2.0 [19] and XQuery 1.0 [20], are being developed for querying XML documents. Effective storage of XML documents must thus take into consideration of the efficient implementations of these queries.

There are two major approaches in storing XML documents [1]. Native Databases (NDB), such as the open source Apache Xindice [17], store XML documents natively as fundamental *logical* units. The second approach, XML Enabled Database (XEDB), builds an XML layer on top of the underlying data model. The layer maps the storage and querying of XML documents into the underlying database. In case of relational databases, the XML documents are stored in relations and XML queries are mapped to SQL. One advantage of this approach is that data can now be queried through either XML technologies, such as XQuery, XPath and DOM, or the underlying database technologies, such as SQL.

Research on storing XML documents into relational databases (called relational schema generation in [9]) can be classified by whether the Document Type Definitions (DTDs) or XML Schemas are known (Schema-based or Schema-oblivious storage [9]), and whether user input are available [3, 5]. For example, [5] and [21] deal mostly with Schema-oblivious storage. [7] handles Schema-based mapping and [2] handles both.

Inlining algorithms are a Schema-based approach for automatically storing XML documents in relational databases. Shanmugasundaram initiated this approach by proposing three inlining algorithms for mapping DTDs to relational schemas [14]. These inlining algorithms use different criteria to inline child XML elements into relations storing the parent elements. Their performances have been evaluated favorably when compared with other strategies on various datasets [6, 16]. Based on [14], Lu proposed another new inlining algorithm [12]. Inlining algorithms have also been extended to handle constraints in DTD [10, 11] and preservation of XML element order [15]. Examples of other related works include using statistics to select the lowest

cost relational mapping [4], using a generic query processor to support general relational schema generation techniques [13], and support of XQuery processing speedup [8].

Inlining algorithms usually involve three steps to map a DTD to relational schemas: (1) Simplify the DTD. (2) Use the simplified DTD to create a DTD graph. (3) Use the DTD graph to generate the relational schemas. This paper makes significant contributions to all of the three steps. However, it does not discuss extended issues such as XML constraints and element ordering.

1.1 Organization

Section 2 presents an optimal and complete DTD simplification procedure. Section 3 discusses the subsequent generation of generic DTD graphs. Section 4 describes the issues that inlining algorithms must resolve, providing a framework to understand these algorithms. It also discusses how existing algorithms resolve them. Section 5 presents a new hybrid inlining algorithm that is more aggressive and complete than the existing ones. Section 6 presents a full example to illustrate the algorithms. We draw our conclusions in Section 7.

2. DTD SIMPLIFICATION

Element type declarations in DTD can be constructed using parentheses and five operators: *sequence* (,), *choice* (|), *zero or one* (?), *one or more* (+) and *zero or more* (*). These declarations can be nested and complicated, for example, `<!ELEMENT a (b,((b+,c)|(d,b*,c?)),(e*,f)?)>`. From the relational schema's point of view, it is only important to store parent-child element relationship and order information to faithfully implement XML queries. For example, for `<!ELEMENT a (b?)>`, it is important for the relational schemas to provide the capability to store a `<b>` child element of `<a>` somewhere. Thus, the mapping algorithm can treat the declaration as `<!ELEMENT a (b)>`. If `<b>` is actually absent in `<a>`, it can still faithfully be represented in the relation, such as by a suitable null value. Thus, the purpose of DTD simplification is to reduce the complexity by removing all nested parentheses and the operators +, |, and ?. For example, `<!ELEMENT a (b,((b+,c)|(d,b*,c?)),(e*,f)?)>` can be simplified to `<!ELEMENT a (b*,c,d,e*,f)>`.

DTD simplification procedures have been described in [5, 10, 12, 14]. However, details of these procedures are not always entirely clear in the papers. For examples, even the more recent procedure by [12] does not clearly specify when nested parentheses can be removed. More importantly, existing procedures do not always give the most optimal result as they do not optimally handle the | operator. For example, for `<!ELEMENT a ((b,c)|(c,d))>`, using the best existing procedure, the following simplification occurs:

```
((b,c)|(c,d))
-> ((b,c),(c,d))
-> (b,c,c,d) [We add this step to remove the parentheses]
-> (b,c*,d)
```

In the simplified result, `<a>` may contain zero or more `<c>` child elements. However, in the original declaration, `<a>` contains exactly one `<c>` child no matter which choice is selected. Thus, the optimal result should be `(b,c,d)`. Simplification to `(b,c*,d)` introduces unnecessary complexity, resulting in a lower degree of inlining.

To handle this situation, we cannot simply convert all occurrences of the choice operator (|) to the sequence operator (,), as in existing algorithms. Instead, operands of the | operator should be compared. We define several terms before presenting a new algorithm for DTD simplification.

Definition 1. An **atom** is either e or e*, where e is an XML element.

Definition 2. A **comma-separated clause** is a sequence of atoms, i.e., (a_1, a_2,...,a_n), where every a_i is an atom. A comma-separated clause may contain only one atom, i.e., n=1.

Definition 3. A **minimal comma-separated clause** is a comma-separated clause where no two atoms are formed from the same element.

Definition 4. An element `<e>` **appears** in a comma-separated clause c if e or e* is an atom in c. An element `<e>` **appears as e*** in a comma-separated clause c if e* is an atom in c.

Example 1. If `<a>` is an element in the DTD, a and $a*$ are atoms. a? and $a+$ are not atoms. (a, b, $c*$, $a*$) is a comma-separated clause. However, it is not a minimal comma-separated clause since `<a>` appears in two atoms in the clauses. ($a*$, b, $c*$) is a minimal comma-separated clause. (a, b, $c+$) is not a comma-separated clause as $c+$ is not an atom. a, b and c appear in the clause ($a*$, b, $c*$). Furthermore, `<a>` appears as $a*$ in the clause ($a*$, b, $c*$), but `<b>` does not appear as $b*$ in the clause ($a*$, b, $c*$).

The following DTD simplification algorithm is based on [12]. It produces a simplified DTD such that each element declaration is a minimal comma-separated clause. Simplification rules used by the algorithm are shown in Figure 1 below. For the rules in Figure 1, *expr* and $expr_i$ are DTD expressions; `<e>` is an element; a and a_i are atoms; and c and c_i are comma-separated clauses.

Algorithm 1 Simplification of DTD

1: *Input* : *A DTD D.*
2: *Output* : *The simplified DTD of D.*
3: Treat #PCDATA as a special child element.
4: **for all** *DTD expressions* of element declarations in D **do**
5: Removal of +: Apply rule R1 recursively until the expression contains no +;
6: Removal of ?: Apply rule R2 recursively until the expression contains no ?;
7: Production of *comma-separated clauses*: Apply rules R3 to R6 recursively until the expression is a comma-separated clause;
8: Removal of *redundant atoms*: Apply rule R7 recursively until the expression is a minimal comma-separated clause;
9: **end for**

$(R1)\ expr+ \rightarrow expr*$
$(R2)\ expr? \rightarrow expr$
$(R3)$ (a) $(c_1|c_2|\ldots|c_n) \rightarrow (c)$ such that (i) an element `<e>` appears in c if and only if `<e>` appears in at least one c_i, and (ii) an element `<e>` appears as $e*$ in c if and only if `<e>` appears in at least one c_i as $e*$.
(b) $\ldots|expr_1|(a_1|a_2|\ldots|a_n)|expr_2|\ldots \rightarrow \ldots|expr_1|a_1|a_2|\ldots|a_n|expr_2|\ldots$
$(R4)$ (a) $(a_1, a_2, \ldots, a_n)* \rightarrow (a_1*, a_2*, \ldots, a_n*)$
(b) $\ldots, expr_1, (a_1, a_2, \ldots, a_n), expr_2, \ldots \rightarrow \ldots, expr_1, a_1, a_2, \ldots, a_n, expr_2, \ldots$
$(R5)\ a** \rightarrow a*$
$(R6)\ ((expr)) \rightarrow (expr)$
$(R7)$ (a) $\ldots, e, \ldots, e, \ldots \rightarrow \ldots, e*, \ldots$
(b) $\ldots, e, \ldots, e*, \ldots \rightarrow \ldots, e*, \ldots$
(c) $\ldots, e*, \ldots, e, \ldots \rightarrow \ldots, e*, \ldots$
(d) $\ldots, e*, \ldots, e*, \ldots \rightarrow \ldots, e*, \ldots$

Figure 1:DTD Simplification Rules.

Note 1. Rule R3(a) is the new rule for comparing the operands of an `|` operator to ensure that the optimal atom is used. Furthermore, there is no rule for ($expr_1$`|`$expr_2$`|`$\ldots$`|` $expr_n$)`*`, where every $expr_i$ is a general DTD expression. Rules R3 to R6 must be applied recursively to simplify the DTD expressions to comma-separated clauses before rule R4(a) can be applied.

Note 2. #PCDATA can be regarded as a regular child element in Algorithm 1. This may create invalid DTD syntax for elements with mixed contents. For example, `<!ELEMENT a (#PCDATA|b)*>` is simplified to `<!ELEMENT a (#PCDATA*, b*)>`. However, this does not create any problem since the simplified DTDs are not used to validate XML documents. Instead, they are only used as an intermediate conceptual step for creating the corresponding relational schemas.

The following example illustrates the simplification action of Algorithm 1.

Example 2.

```
   (b?,((b+,c,d)|(c?,(d|(e*,f))+)))
-> (b?,((b*,c,d)|(c?,(d|(e*,f))*))) [Repeated uses of Rule R1]
-> (b,((b*,c,d)|(c,(d|(e*,f))*)))   [Repeated uses of Rule R2]
-> (b,((b*,c,d)|(c,((d,e*,f))*)))   [Rule R3(a)]
-> (b,((b*,c,d)|(c,(d,e*,f)*)))     [Rule R6]
-> (b,((b*,c,d)|(c,(d*,e**,f*))))   [Rule R4(a)]
-> (b,((b*,c,d)|(c,(d*,e*,f*))))    [Rule R5]
-> (b,((b*,c,d)|(c,d*,e*,f*)))      [Rule R4(b)]
-> (b,(b*,c,d*,e*,f*))              [Rule R3(a)]
-> (b,b*,c,d*,e*,f*)                [Rule R4(b)]
-> (b*,c,d*,e*,f*)                  [Rule R7(b)]
```

We have the following theorems on the properties and complexity of Algorithm 1.

THEOREM 1. *Algorithm 1 is complete in the sense that it handles all DTD cases and produces minimal comma-separated clauses for all element declarations.*

PROOF. The theorem is obvious since there are applicable rules for removing the operators `+`, `?` and `|` until the DTD is simplified into comma-separated clauses. The recursive applications of Rule (7) guarrantees that the final clauses are minimal. □

THEOREM 2. *Algorithm 1 is optimal in the sense that it does not include any atom $e*$ in any resulting minimal comma-separated clauses if e is sufficient.*

PROOF. It is only necessary to consider Rule (7a), since of all rules with `*` in the right hand side, only rule (7a) has no + or `*` in the left hand side. Since Rule (7a) has at least two occurrences of the element `<e>` in the left hand side, it is not optimal only if a superfluous occurrence of `<e>` can be generated by other rules. Careful examinations of all rules indicate that no rule generates a superfluous element. This includes Rule (3a), which is carefully constructed to avoid generation of superfluous element for the `|` operator. □

THEOREM 3. *The worst case time complexity of Algorithm 1 is $O(N_{\text{op}}^2)$, where N_{op} is the total number of operators, including parentheses, in the DTD.*

PROOF. Every occurrrence of `?` or `|` can be removed by the application of Rule (1) or (2) once. Except for Rule (4a), each application of all other rules reduce the number of operators by at least one. Each application of Rule (4a) can generate at most N_{op}-1 `*` and the rule can only be applied for less than N_{op} times since it removes a pair of parentheses (counted as an operator). □

3. DTD GRAPHS

Before generating the relational schemas, most existing inlining algorithms include a further step of generating DTD graphs from the simplified DTDs. These DTD graphs do not only assist in constructing the eventual mapping by providing a familiar data structure to represent the simplified DTD. They may also incorporate inlining criteria to determine when inlining should occur, such as [12]. However, the dual purpose of representing the DTD and incorporating inlining criteria may result in added complexity and decreased universality (one DTD graph cannot be used by another inlining algorithm). We propose to decouple the representation from the inlining criteria. Algorithm 2 generates *generic* DTD graphs that can serve as the universal basis of inlining algorithms. This generic DTD graph is simply a graphical representation of the simplified DTDs without considering any inlining criteria.

Note 3. o-edges are called ,-edges in [12]. Since "," may be confused with the sequence operator, we select to use the term o-edges to indicate that the child element can appear within the parent element at most *one* time, as opposed to many times for `*`-edges.

Example 3. `Consider the following DTD declarations:`

```
<!ELEMENT a (b,c*)>
<!ELEMENT c (#PCDATA)>
```

Algorithm 2 Generation of DTD Graphs

```
1: Input : A simplified DTD D.
2: Output : G, the DTD Graph of D.
3: Create a node in G for each element in D;
4: for all element <p> of D that does not contain only #PCDATA do
5:    for all child element <e> (including #PCDATA) of p do
6:       if <e> appears as e* in the declaration of <p> then
7:          add to G an *-edge from <p> to <e>;
8:       else
9:          add to G an o-edge from <p> to <e>;
10:      end if
11:   end for
12: end for
```

```
<!ELEMENT b (f)>
<!ELEMENT h (c)>
<!ELEMENT f (#PCDATA)>
```

Figure 2 shows the corresponding DTD graph generated by Algorithm 2.

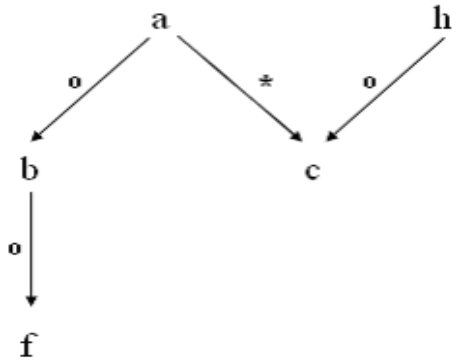

Figure 2:DTD Graph generated by Algorithm 2

Definition 5. The **in-degree** of a node in a DTD graph is the number of incoming edges to the node.

Note 4. The in-degree of a node is thus the number of times the element appears in the declarations of other elements, or the number of parent element types.

THEOREM 4. *The time complexity of Algorithm 2 is $O(N_e + N_{ec})$ where N_e and N_{ec} are the total number of all element declarations and the total number of parent child relationship in these element declarations (edges) respectively.*

PROOF. The theorem is obvious since each element and each child element in its declaration (i.e. parent child relationship) is visited exactly once. □

4. INLINING CONSIDERATIONS

Different inlining algorithms have different criteria for inlining descendant elements. However, there are common issues concerning all inlining algorithms and they will be discussed here. We will also discuss how three existing inlining algorithms (SHARED [14], HYBRID [14] and LU [12]) resolve these issues. There also exist other inlining algorithms which are however either not complete enough (such as BASIC [14]) or very similar to one of these three algorithms (such as [10, 11]).

A naive approach for mapping DTDs to relational schemas is to convert every XML element to a unique relation. However, as [14] has pointed out, this naive approach will likely lead to excessive fragmentation of the XML document.

Example 4. Consider the following simple partial DTD declarations:

```
<!ELEMENT a (b)>
<!ELEMENT b (c)>
<!ELEMENT c (d)>
<!ELEMENT d (#PCDATA)>
```

The naive approach creates four relations: *a*, *b*, *c* and *d*. An XML query for the entire content of the element `<a>` thus requires the joining of all four relations.

The basic idea of inlining algorithms is to store some descendent elements into the relation of an element. In the example above, inlining the elements `<b>`, `<c>` and `<d>` into the relation *a* will thus result in only one relation. The contents of the elements `<b>`, `<c>` and `<d>` are stored in the relation *a*. No join will then be necessary to query the content of `<a>`.

There are common features of existing algorithms.

1. An element with in-degree=1 (exactly one incoming o-edge) is inlined into its parent element, which may in turn be inlined.
2. An element with in-degree=0 has its own relation.
3. A surrogate primary key is created for every relation.

The following subsections discuss four common issues in detail: XML root elements, multiple occurrences, multiple parent element types and recursions. Shanmugasundaram *et. al.* have briefly discussed multiple occurrences (as set-value attributes) and recursions in [14].

4.1 XML Root Element

DTD does not specify the root elements of the XML documents it validates.

Example 5. Consider the DTD:

```
<!ELEMENT a (b)>
<!ELEMENT b (#PCDATA)>
```

The DTD can be used to validate XML documents with root elements `<a>` or `<b>`. If inlining is used, a naive relational schema may contain only one relation *a(a_id, a.b)*, where the data content is stored in *a.b*. It is then necessary to have a mechanism to specify whether `<a>` or `<b>` is the root for a given XML document. This is accomplished by using a Boolean column *isroot* in SHARED and HYBRID, and an integer column *nodetype* in LU. Note that we use the term column instead of the more formal term attribute in relations to avoid confusion with XML attributes.

4.2 Multiple Occurrences

This corresponds to elements with incoming `*`-edges in our formalism. If an element has multiple child elements of the same type, it should not be inlined since relations do not accept set-value columns. To circumvent this mismatch between the XML and the relational data models, it is necessary for inlining algorithms to store the child elements in a separate relation.

Inlining algorithms may differ in how the parent child relationship is stored. Both SHARED and HYBRID store the parent element id as a foreign key in the child relation. In LU, a single relation *edge(parentID,childID, parentType, childType)* stores all relationships between any two XML elements [12].

Example 6. Consider the DTD:

```
<!ELEMENT a (b*)>
<!ELEMENT b (c*)>
<!ELEMENT c (#PCDATA)>
```

Both SHARED and HYBRID generate relations similar to *a(aID)*, *b(bID, b.parentID)* and *c(cID,c,c.parentID)*. On the other hand, LU generates relations similar to *a(ID),b(ID)* (a and b will actually be coalesced into one relation), *c(ID,PCDATA)* and *edge(parentID,childID,parentType,childType)*.

In [12], the design decision of using the separate relation *edge* in LU was explained as a way to avoid redundancy in storing the child elements of the same contents, which "bears the same spirit as the rule of mapping many-to-many relationships into separate relations." However, since the parent-to-child relationship in XML documents are one-to-many, we do not see any advantage for doing so.

In order to take advantage of LU's algorithm to minimize redundancy, additional processing is needed to determine that two elements have the same content before they can be represented by the same tuple. Even so, it may still not be possible to do so in some uncommon situation, as illustrated by the following example.

Example 7. Consider the XML document:

```
<?xml version="1.0" ?>
<b>
<c>hello</c>
<c>hello</c>
</b>
```

Even though the two `<c>` elements have the same content, it may not be represented by a single tuple since there is no way in LU to specify that the `<b>` element has two occurrences of the tuple.

4.3 Multiple Parent Element Types

This issue is whether a separate relation should be created for an element node in a DTD graph that has an in-degree > 1. If the element has an incoming *-edge, a separate relation should have already been created for it (see the previous subsection). Thus, it is only necessary to consider the case when all incoming edges to the element are o-edges. In this case, either a separate relation or inlining can be used to store the element.

SHARED generates a relation for every such element `<e>` with in-degree > 1. The column *e.parentID* is added to specify its parent. Since `<e>` may have parents of different types, an additional column *e.parentCode* is used to specify the parent's element type. LU also generates a relation for `<e>`. However, since there is a separate relation, *edge*, to store parent and child relationships, no additional column is needed. HYBRID inlines `<e>` into its parents if there is no recursion. The following example illustrates their differences.

Example 8. Consider the DTD:

```
<!ELEMENT a (c)>
<!ELEMENT b (c)>
<!ELEMENT c (#PCDATA)>
```

SHARED generates the relations *a(aID)*, *b(bID)* and *c(cID,c,c.parentID,c.parentCode)*. LU generates relations similar to *a(ID)*, *b(ID)*, *c(ID,PCDATA)* and *edge(parentID,childID, parentType,childType)*. HYBRID generates the relations *a(aID, c,c.isroot)* and *b(bID,c,c.isroot)*.

As pointed out in [14], the performance of HYBRID in query processing is generally better than that of SHARED. However, there remains an unanswered problem in HYBRID regarding where the element `<c>` should be stored if it happens to be the root element of an XML document (in relation *a* or *b*?). For our new inlining algorithm to be discussed in the next section, we propose to create a separate relation, *c*, for `<c>` just for documents with `<c>` as the root elements.

4.4 Recursion

Recursions occur in DTDs when there exist elements that may be descendants of each other.

Example 9. Consider the DTD:

```
<!ELEMENT a (b?)>
<!ELEMENT b (a)>
<!ATTLIST b x CDATA #REQUIRED>
```

It validates:

```
<?xml version="1.0" ?>
<a><b x="1"><a /></b></a>
```

Recursions are not very common in XML documents. However, there are proper uses of them. For example, the XHTML specification contains many recursion definitions [18]. More importantly, in order to be complete, an inlining algorithm needs to be able to handle all forms of recursions. That is, in a set of mutually recursive elements, it is necessary to ensure the creation of at least one relation for storing the elements.

Recursion has not been handled entirely satisfactorily in existing work. LU assumes that in order for a DTD to be consistent (validated XML documents have finite sizes), a cycle in a DTD graph could not be composed of o-edges entirely and should contain at least one `*`-edge. If this were true, a relation will be created for the `*`-edge and LU does not consider handling cycles with all elements of in-degree=1. However, a closer look indicates that this is inappropriate. An o-edge to an element `<e>` may be e or $e?$. The case $e?$ allows the validated XML documents to avoid infinite expansion of elements. For example, the consistent DTD in the previous example with only `<!ELEMENT a (b?)>` and `<!ELEMENT b (a)>` contains a cycle of o-edges only. Thus, it is necessary to consider cycles in DTD graphs that contain only o-edges.

SHARED and HYBRID consider cycles with only 0-edges but check only those cycles containing mutually recursive elements all having in-degree=1. In this case, the algorithms create a separate relation for one element in the cycle. However, this is still not sufficient since recursive elements may all have in-degree `>` 1 and recursion may also interfere with the issue of multiple parent element types.

Example 10. Consider the DTD graph shown in Figure 3

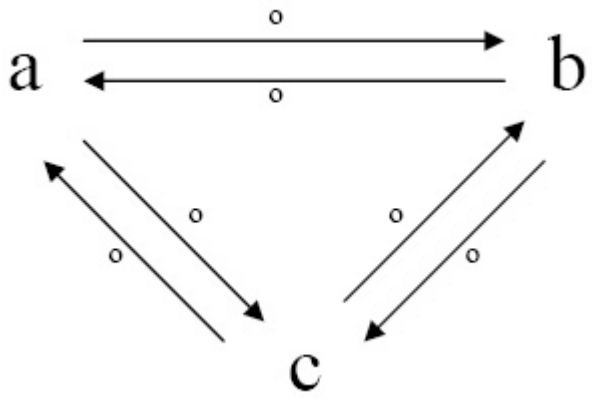

Figure 3: DTD graph with mutaully recursive elements.

SHARED generates three relations for a, b and c. For HYBRID, the action is not clear on how recursion interacts with multiple parent element types. In this particular case, since every element in the cycle has in-degree=2, the stated recursion rule for in-degree=1 does not apply. It is thus necessary to have a clearer and more accurate set of rules for recursion.

5. A NEW INLINING ALGORITHM

In this section, we present a new inlining algorithm (Algorithm 3) that completely addresses all the issues we have mentioned in the previous sections. Algorithm 3 adopts some of the best practices from existing algorithms, but also proposes new solutions and clarifies ambiguity in other aspects. In particulars, it inlines more aggressively than any existing algorithms, thus providing better potential in efficiently handling XML queries. The basic inlining criteria of Algorithm 3 as compared to others are listed below:

1. SHARED and LU: An element with in-degree of 1 through an o-edge is inlined to its sole parent.
2. HYBRID: An element with in-degree >= 1 and with no incoming *-edge is inlined to all its parents. An element with an incoming *-edge is not inlined.
3. Algorithm 3: An element with in-degree >= 1 is inlined to all its parents that have an o-edge to the element.

Example 11. Consider the DTD:

```
<!ELEMENT a (c)>
<!ELEMENT b (c,d)>
<!ELEMENT c (#PCDATA)>
<!ELEMENT e (d*)>
<!ELEMENT d (#PCDATA)>
```

SHARED and LU do not inline any element. HYBRID inlines only <c> to both <a> and <b>. Algorithm 3 inlines <c> to <a> and <b>, but also <d> to <b>.

Algorithm 3 *CreateRelationalSchemas(D)*: create the relational schemas for a DTD D

```
1: Input : A DTD D and its associated DTD graph G.
2: Output : The relational schemas for D.
3: if the node #PCDATA is in G then
4:    Create the relation $PCDATA($ID, $data, $parentID, $parentType) with the primary key $ID;
5: end if
6: for all <e> that is a child element with an incoming *-edge in G do
7:    CreateRelation(e);
8:    Add the columns e + ".$parentID" and e + ".$parentType" to the relation e;
9:    // + is the string concatenation operator.
10: end for
11: for all element <e> that has in-degree=0 in G do
12:    CreateRelation(e);
13: end for
14: for all element <e> that has more than one incoming o-edges in G do
15:    if no relation has already been created for <e> then
16:       CreateRelation(e);
17:    end if
18: end for
19: for all cycle that composes only of o-edges in G do
20:    if no relation has already been created for any element in the cycle then
21:       CreateRelation(one of the elements in the cycle);
22:    end if
23: end for
```

In line 4 of Algorithm 3, a generic relation, *$PCDATA*, is created to store PCDATA contents of elements with **mixed contents**. The columns *$parentID* and *$parentType* store the ID and types of the elements respectively. Since $ is not allowed in XML names, the prefix $ creates a 'namespace' for special columns to avoid potential ambiguity.

Lines 6 to 10 handle multiple occurrences by creating relations for such elements and adding columns to point to their parents. Lines 11 to 13 create relations for elements with in-degree=0. Lines 14 to 18 resolve the problem of multiple parent element types in HYBRID, as discussed in Section 4.4. When an element has more than one incoming o-edges, if necessary, a relation is created to store the element if and only if it is the root element of the XML document. It increases the number of relations but not the storage size or query performance. Elements with exactly one incoming o-edge are inlined and no relations are created for them.

Lines 19 to 23 handle recursive elements completely since all cycles are considered, not limited to those of in-degree=1. Recursive elements are left unmarked by the *CreateRelation* procedure, and finding them takes linear time (see Theorem 6).

Algorithm 4 basically uses a breadth-first traversal of an element and all of its descendants to create a relation and add their contents as columns. In case a back edge is found, columns pointing to the parent node from the back node are ensured to be added. The variable *visitedNodes* is used to remember all the visited nodes with full paths. These nodes can then be used for detecting back edges and as prefixes of pointers to parents. When a new node is visited, the full path is added to the node before it is enqueued to *nodesToTraverse*.

Algorithm 5 handles the cases when the element content is #PCDATA, ANY or EMPTY, which are not completely considered by existing algorithms. For EMPTY, it is necessary to add the Boolean column $exists to indicate whether such a child element exists in a given XML document.

Algorithm 4 *CreateRelation(e)*: Create the relation e for the element `<e>`

Input : *An element* `<e>`.
Output : *The relational schema e for the element.*
Create a relation e with the primary key $e.\$ID$;
$nodesToTraverse \leftarrow \{e\}$;
$visitedNodes \leftarrow \emptyset$;
if there is at least one outgoing o-edge from `<e>` **then**
  Add the column e + ".$*nodeType*";
end if
while $nodesToTraverse \neq \emptyset$ **do**
  $currentNode \leftarrow$ dequeue($nodesToTraverse$);
  $visitedNodes \leftarrow visitedNodes \cup \{currentNode\}$;
  $AddColumns(e, currentNode)$;
  for all outgoing o-edge from $currentNode = (currentNode, c)$ **do**
    if c does not appear in $visitedNodes$ **then**
      // Add the child node with the full path.
      enqueue($nodesToTraverse, currentNode + "." + c$);
    else
      // A back edge to a previously visited node.
      Let $fullPathC$ in $visitedNodes$ be the same node as c but with *full path*;
      Add the columns $fullPathC$ + ".$*parentID*" and $fullPathC$+ ".$*parentType*" to the relation e if they have not been added;
    end if
  end for
end while

Algorithm 5 *AddColumns(e,currentNode)*: Add the content of *currentNode*, not including its descendants, to the relation e

Input : *The relation e and an element node currentNode*
Output : *The content of currentNode (#PCDATA and attributes) is added to e.*
for all attribute *attr* of currentNode **do**
  add the column $currentNode$ + "." + $attr$ to the relation e;
end for
if *currentNode* is of the type #PCDATA or ANY **then**
  add the column *currentNode* to the relation e;
end if
if *currentNode* is of the type EMPTY **then**
  add the column $currentNode$ + ".$*exists*" to the relation e;
end if

The following theorems specify some properties and the time complexity of Algorithm 3.

THEOREM 5. *The total number of relational schemas created is less than or equal to N_e+1, where N_e is the total number of element declarations in the DTD.*

PROOF. Examination of Algorithm 3 indicates that for a given element `<e>`, exactly one relation will be created for `<e>` by calling the procedure *CreateRelation(e)* if `<e>` has no incoming edge or `<e>` has an incoming `*`-edge. A relation may or may not be created for `<e>` if it has only o-edges. We add one to the

bound for the creation of the relation *$PCDATA*. A corollary of this theorem is that *CreateRelation* will be called at most N_e times. □

THEOREM 6. *The worst case time complexity of Algorithm 3 is bounded by* $O(N_e * (N_e + N_a))$ *where* N_a *is the total number of attributes.*

PROOF. The procedure *CreateRelation* (Algorithm 4) is called at most N_e times, The only non-constant part of *CreateRelation* is the while loop in line 9, which will execute at most N_e times. Within the for loop, lines 10 and 11 take constant time. Time for line 12 will depend on the number of the attributes. Since each edge is visited at most once, the cumulative number of iterations of the for loop starting in line 13 is limited by the number of edges in the graph. Hence, the while loop has a time complexity of $O(N_e + N_a)$. □

To summarize, Algorithm 3 contributes in several areas:

1. *Aggressiveness*: It is more aggressive in inlining than existing algorithms.
2. *Completeness*: It handles recursion, #PCDATA, ANY and EMPTY completely and explicitly.
3. *Clarity*: It provides more low level implementation details, reducing ambiguity.

6. A FULL EXAMPLE

A full example that covers all major features in the algorithms is presented here. Consider the following DTD:

```
<!ELEMENT a ((b,c*)|(c,b))>
<!ELEMENT b (f?)>
<!ELEMENT c (#PCDATA|g)*>
<!ELEMENT d (e)>
<!ELEMENT e (b,d?)>
<!ELEMENT f (EMPTY)>
<!ELEMENT g (#PCDATA)>
<!ELEMENT h (c)>
<!ATTLIST b p CDATA #REQUIRED>
<!ATTLIST c q CDATA #REQUIRED>
<!ATTLIST e r CDATA #REQUIRED>
```

Algorithm 1 simplifies only elements `<a>`, `<b>`, `<c>` and `<e>` to:

```
<!ELEMENT a (b,c*)>
<!ELEMENT b (f)>
<!ELEMENT c (#PCDATA*,g*)>
<!ELEMENT e (b,d)>
```

Note that the element `<a>` is simplified to contain b, not $b*$. Algorithm 2 generates the DTD graph shown in Figure 4.

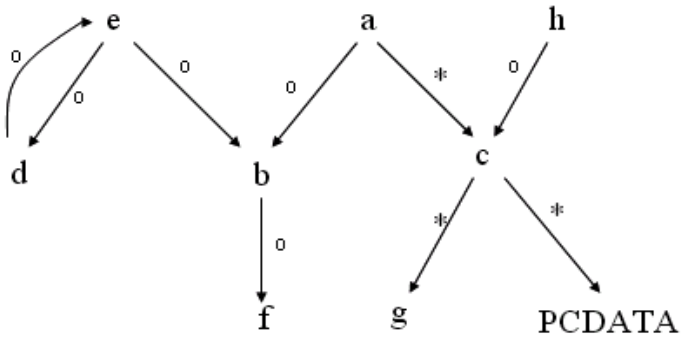

Figure 4:The DTD graph generated by Algorithm 2

Algorithm 3 creates the following relational schemas.

```
$PCDATA($ID, $data, $parentID, $parentType)
a(a.$ID, a.b.p, a.b.f.$exists, a.$nodeType)
b(b.$ID, b.p, b.f.$exists, b.$nodeType)
c(c.$ID, c.q, c.$parentID, c.$parentType)
d(d.$ID, d.e.r, d.e.b.p, d.e.b.f.$exists, d.$nodeType, d.$parentID, d.$parentType)
g(g.$ID, g, g.$parentID, g.$parentType)
h(h.$ID, h.c.q, h.$nodeType)
```

Note that the element <c> is inlined in the relation h, which is not inlined in other existing algorithms.

The following examples show how XML documents are stored in these relations.

Example 12. Consider the following XML document:

```
<?xml version="1.0" ?>
<a>
  <b p="p1"><f /></b>
  <c q="q1">c1<g>g1</g></c>
  <c q="q2">c2<g>g2</g></c>
</a>
```

It can be stored by the following relational tuples:

```
a(a.$ID:1, a.b.p:'p1', a.b.f.$exists:true, a.$nodeType:'a')
c(c.$ID:1, c.q:'q1', c.$parentID:1, c.$parentType:'a')
c(c.$ID:2, c.q:'q2', c.$parentID:1, c.$parentType:'a')
g(g.$ID:1, g:'g1', g.$parentID:1, g.$parentType:'c')
g(g.$ID:2, g:'g2', g.$parentID:2, g.$parentType:'c')
$PCDATA($ID:1, $data:'c1', $parentID:1, $parentType:'c')
$PCDATA($ID:2, $data:'c2', $parentID:2, $parentType:'c')
```

Note that the value of a.$nodeType is 'a'. The content of the elment <b> is inlined into the tuple a. Furthermore, order information is not stored as it is not considered in this paper. They can be stored by adding a proper column in the relations.

Example 13. Consider the following XML document, which highlights the recursive nature between elements <d> and <e>:

```
<?xml version="1.0" ?>
<d>
  <e r="r1">
    <b p="p1">
    <d>
      <e r="r2">
        <b p="p2"><f /></b>
      </e>
    </d>
  </e>
</d>
```

It can be stored by the following two relational tuples:

```
d(d.$ID:1, d.e.r:'r1', d.e.b.p:'p1', d.e.b.f.$exists:false, d.$nodeType:'d',
   d.$parentID:null, d.$parentType:null)
d(d.$ID:2, d.e.r:'r2', d.e.b.p:'p2', d.e.b.f.$exists:true, d.$nodeType:'d',
   d.$parentID:1, d.$parentType:'d')
```

Note that only two tuples are required to store six elements since <b> and <e> are inlined in the relation b.

Example 14. Consider the following XML document:

```
<?xml version="1.0" ?>
<h>
  <c q="q1">c1<g>g1</g>c2<g>g2</g></c>
</h>
```

It can be stored by the following relational tuples:

```
h(h.$ID:1, h.c.q:'q1', h.$nodeType:'h')
g(g.$ID:1, g:'g1', g.$parentID:1, g.$parentType:'h')
g(g.$ID:2, g:'g2', g.$parentID:1, g.$parentType:'h')
$PCDATA($ID:1, $data:'c1', $parentID:1, $parentType:'h')
$PCDATA($ID:2, $data:'c2', $parentID:1, $parentType:'h')
```

This example shows that the element <c> is inlined in the relation h, which is not done in other existing algorithms. Note that the values of $parentType is 'h' to refer to the relation h. Meta information is needed to determine that the contents of ¡g¿ and$PCDATA are child nodes of the element <c>.

Example 15. Consider the following XML document:

```
<?xml version="1.0" ?>
<c q="q1">c1<g>g1</g>c2<g>g2</g></c>
```

It can be stored by the following relational tuples:

```
c(c.$ID:1, c.q:'q1', c.$parentID:null, c.$parentType:null)
g(g.$ID:1, g:'g1', g.$parentID:1, g.$parentType:'c')
g(g.$ID:2, g:'g2', g.$parentID:1, g.$parentType:'c')
$PCDATA($ID:1, $data:'c1', $parentID:1, $parentType:'c')
$PCDATA($ID:2, $data:'c2', $parentID:1, $parentType:'c')
```

Note that since <c> is the root element, it is stored in the relation c, but not h.

7. FUTURE WORK AND CONCLUSIONS

This paper contributes to the development of inlining algorithms by presenting a complete and optimal algorithm for simplifying DTDs, an algorithm for creating a generic DTD graph, a framework of issues for constructing inlining algorithms, and a newly proposed inlining algorithm that is more complete and aggressive. These algorithms have been implemented using Java's JDK 1.4 and tested with about 30 DTD test cases with expected results. In the future, we will perform a comparative performance evaluation of query processing of the new algorithm. We will also explore to extend the algorithm to map XML Schema and handle such issues as element ordering and XML constraints.

8. REFERENCES

[1] S. W. Ambler. *Agile Database Techniques*. John Wiley & Sons, New Jersey, 2003.

[2] S. Amer-yahia, D. Srivastava A mapping schema and interface for XML stores. In *Preecedings of the fourth international workshop on web information and data management*, pages 23-30, 2002

[3] E. Bertino and B. Catania. Integrating xml and databases. *IEEE Internet Computing*, 5(4):84–88, 2001.

[4] P. Bohannon, J. Freire, P. Roy, and J. Simon. From xml schema to relations: A cost-based approach to xml storage. In *18th International Conference on Data Engineering (ICDE'02)*, page 64. IEEE Computer Society, February 2002.

[5] A. Deutsch, M. Fernandez, and D. Suciu. Storing semistructured data with stored. In *SIGMOD '99: Proceedings of the 1999 ACM SIGMOD international conference on Management of data*, pages 431–442. ACM Press, 1999.

[6] D. Florescu and D. Kossmann. A performance evaluation of alternative mapping schemes for storing xml data in a relational database. *INRIA Technical Report*, 1(3), May 1999.

[7] L. Khan, Y. Rao. A Performance evaluation of storing XML data in relational database Management. *Proceedings of the third international workshop on web information and data management*, pages 31-38, 2001.

[8] J. Kim, S. Park. XQuery speedup using replication in mapping XML into relations. *Proceedings of the 2003 ACM symposium on Applied Computing*, pages 536=543, 2003.

[9] R. Krishnamurthy, R. Kaushik, and J. F. Naughton. Xml-to-sql query translation literature: The state of the art and open problems. In *Lecture Notes in Computer Science, Vol. 2824*, pages 1–18. Springer-Verlag, September 2003.

[10] D. Lee and W. W. Chu. Cpi. constraint-preserving inlining algorithm for mapping xml dtd to relational schema. *Data and Knowledge Engineering*, 39:3–25, 2001.

[11] D. Lee, M. Mani, and W. W. Chu. Schema conversion methods between xml and relational models. *Knowledge Transformation for the Semantic Web*, pages 11–17, 2003.

[12] S. Lu, Y. Sun, M. Atay, and F. Fotouhi. A new inlining algorithm for mapping xml dtds to relational schemas. In *ER (Workshops)*, pages 366–377, 2003.

[13] J. Shanmugasundaram, E. Shekita, J. Kiernan, R. Krishnamurthy, E. Viglas, J. Naughton, and I. Tatarinov. A general technique for querying xml documents using a relational database system. *SIGMOD Rec.*, 30(3):20–26, 2001.

[14] J. Shanmugasundaram, K. Tufte, C. Zhang, G. He, D. J. DeWitt, and J. F. Naughton. Relational databases for querying xml documents: Limitations and opportunities. In *VLDB'99*, pages 302–314. Morgan Kaufmann, 1999.

[15] I. Tatarinov, S. D. Viglas, K. Beyer, J. Shanmugasundaram, E. Shekita, and C. Zhang. Storing and querying ordered xml using a relational database system. In *SIGMOD '02*, pages 204–215. ACM Press, 2002.

[16] F. Tian, D. J. DeWitt, J. Chen, and C. Zhang. The design and performance evaluation of alternative xml storage strategies. *SIGMOD Rec.*, 31(1):5–10, 2002.

[17] A. Xindice. Apache xindice. `http://xml.apache.org/xindice`.

[18] XHTML: The Extensible HyperText Markup Language (Second Edition), `http://www.w3.org/TR/xhtml`

[19] Xml path language (xpath) 2.0 w3c working draft (2003). `http://www.w3.org/TR/xpath20`.

[20] Xquery 1.0: An xml query language w3c working draft (2003). `http://www.w3.org/TR/xquery`.

[21] M. Yoshikawa, T. Amagasa, T. Shimura, and S. Uemura. Xrel: a path-based approach to storage and retrieval of xml documents using relational databases. *ACM Trans. Inter. Tech.*, 1(1):110–141, 2001.