Constraints-preserving Transformation from XML Document Type Definition to Relational Schema

Dongwon Lee **Wesley W. Chu**

Department of Computer Science
University of California, Los Angeles
Los Angeles, CA 90095, USA

Email: {dongwon,wwc}@cs.ucla.edu

Initial Version	:	January 12, 2000
Last Revised	:	May 16, 2000

Abstract

As Extensible Markup Language (XML) [5] is emerging as the *data format of the internet era, more needs to efficiently store and query XML data arise. One way towards this goal is using relational database by transforming XML data into relational format. In this paper, we argue that existing transformation algorithms are not complete in the sense that they focus only on* structural *aspects, while ignoring* semantic *aspects. We show the kinds of semantic knowledge that needs to be captured during the transformation in order to ensure correct relational schema at the end. Further, we show a simple algorithm that can 1) derive such semantic knowledge from the given XML Document Type Definition (DTD) and 2) preserve the knowledge by representing them in terms of* semantic constraints *in relational database terms. By combining the existing transformation algorithms and our* constraints-preserving *algorithm, one can transform XML DTD to relational schema where correct semantics and behaviors are guaranteed by the preserved constraints. Our implementation and complete experimental results are available from [12].*

UCLA-CS-TR-200001

Contents

1 Introduction **3**

2 Background **4**

2.1 Relational Schema 4

2.2 XML and DTD 4

2.3 Assumptions 7

3 Transforming DTD to Relational Schema **7**

3.1 Hybrid Inlining Algorithm 7

4 Semantic Constraints in DTD **10**

4.1 Domain Constraints 10

4.2 Cardinality Constraints 11

4.3 Inclusion Dependencies (IDs) 11

4.4 Equality-Generating Dependencies (EGDs) 12

4.5 Tuple-Generating Dependencies (TGDs) 12

5 Discovering and Preserving Semantic Constraints **12**

5.1 Data Structures 12

5.2 Discovering Semantic Constraints 13

5.3 Preserving Semantic Constraints 13

5.4 CPI: Constraints-preserving Inlining Algorithm 15

6 Experimental Results **17**

7 Application of the Semantic Constraints **18**

7.1 Semantic Query Optimization 18

7.2 Semantic Caching 19

8 Related Work **20**

9 Conclusion **20**

1 Introduction

As the World-Wide Web becomes a major mean of disseminating and sharing information, Extensible Markup Language (XML) [5] is emerging as a possible candidate data format due to its relative simplicity as compared to SGML and its relative powerfulness as compared to HTML. To query XML data, one way is to reuse the established relational database techniques by converting and storing XML data in relational storage. Since the hierarchical XML and the flat relational data models are not fully compliant, the transformation is not a straightforward task.

To this goal, several XML-to-relational transformation algorithms have been studied. For instance, [18] presents 3 algorithms that focus on the table level of the schema while [11] studies different performance issues among 8 algorithms that focus on the attribute and value level of the schema. They all transform the given XML Document Type Definition (DTD) to relational schema. Similarly, [10] presents a data mining-based algorithm that instead uses XML documents directly without DTD.

Although all these algorithms work well for the given applications, to a greater or lesser extent, they miss one important point. That is, the algorithms are specifically designed for applications where users are given only XML views and resulting relational schema are hidden from them. Thus, there is no need to worry about direct querying towards the relational schema. However, in applications where XML and relational data must co-exist or be merged together, queries against both XML and relational views are expected and certain anomalies can occur due to the result of incomplete transformation. Consider the following motivating example.

Example 1. A DTD regarding conference publications is given:

```
<!ELEMENT conf   (title,year,society,date,paper+)>
<!ELEMENT paper  (pid,title,...)>
```

Using the hybrid inlining algorithm (will be explained in detail in Section 3) in [18], the given DTD would be transformed to the following relational schema:

```
conf  (title,year,society,date)
paper (pid,title,conf_title,conf_year,...)
```

While the relational schema correctly captures the structural aspect of the DTD, it does not force correct semantics. For instance, it cannot prevent a tuple t_1: `paper(100,'DTD...','ER',2001,...)` from being inserted. However, tuple t_1 is inconsistent with semantics of the given DTD since the DTD implies that the paper cannot exist without being associated with a conference and there is apparently no conference "ER-2001" yet. In database terms, this kind of violation can be easily prevented by *inclusion dependency* saying "`paper[conf_title,conf_year]` $\subseteq$ `conf[title,year]`". ■

The reason of this inconsistency between the DTD and the transformed relational schema is that transformation algorithms only capture the *structure* of the DTD and ignore the *semantic* constraints hidden in it. In this paper, via our *constraints-preserving inlining (CPI)* algorithm, we show the kinds of semantic constraints that can be derived from DTD during transformation, and how to preserve them by re-writing them in resulting schema notation. Since our algorithm to capture and preserve semantic constraints from DTD is orthogonal to transformation algorithms, ours can be applied to various transformation algorithms with little change.

Figure 1 presents an overview of our approach. First, given a DTD, we transform it to a corresponding relational scheme using an existing algorithm. Second, during the transformation, we discover various

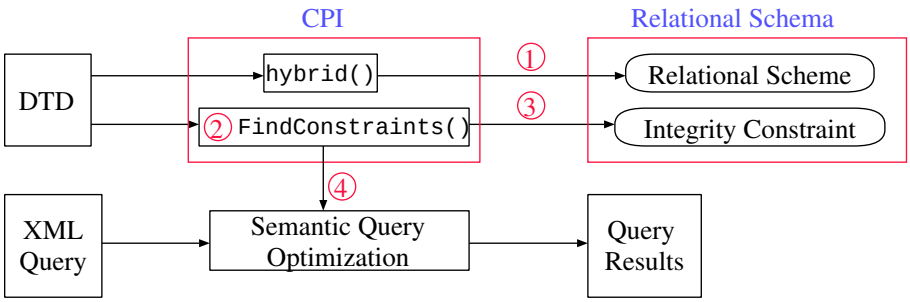

Figure 1: Overview of our approach. Numbers 1) to 4) specify: 1) transforming schema, 2) discovering constraints (i.e., `FindConstraints()`, 3) preserving constraints (i.e., `RewriteConstraints()`, and 4) applying constraints.

semantic constraints in XML notation. Third, we re-write the discovered constraints to conform to relational notation. Finally, in addition, we show 2 motivating examples which utilize the discovered constraints in step 2.

This paper is organized as follows. Section 2 gives a brief introduction of XML and DTD. In Section 3, one transformation algorithm is discussed in detail. Section 4 presents various semantic constraints that are hidden in DTD. Section 5 proposes our algorithm to preserve such constraints during transformation. Section 6 reports some experimental results that we have conducted. Section 7 shows examples that use the constraints found by our method. Related work is given in Section 8.

2 Background

2.1 Relational Schema

In general, the overall design of the database is called the *database schema*. We define a relational schema $\mathcal{R}$ to be composed of a *relational scheme* ($\mathcal{S}$) and *semantic constraints* (Δ). That is, $\mathcal{R} = (\mathcal{S}, \Delta)$. In turn, the relational scheme $\mathcal{S}$ is a collection of table schemes such as $r(a_1, ..., a_k)$, where a_i is the i-th attribute in the table r and the semantic constraints Δ is a collection of various semantic knowledge such as domain constraints, inclusion dependency, equality-generating dependency, tuple-generating dependency, etc.

2.2 XML and DTD

XML is a textual representation of the hierarchical data that is being defined by the World-Wide Web Consortium [5]. The meaningful piece of the XML document is bounded by matching starting and ending *tags* such as `<name>` and `</name>`. In XML, tags are defined by users while in HTML, permitted tags are pre-defined. Thus, XML is a meta-language that can be used for defining other customized languages. Using the Document Type Definition (DTD), users can define the structure of the XML document of particular interest. A DTD in XML is very similar to a schema in a relational database. The main building blocks of DTD are *elements* and *attributes*, which are defined by the keywords `<!ELEMENT>` and `<!ATTLIST>`, respectively. In general, components in DTD are specified by the following BNF syntax:

```
<!ELEMENT> <element-name> <element-type>
<!ATTLIST> <attribute-name> <attribute-type> <attribute-option>
```

Table 1: A DTD for `Conference`.

```
<!DOCTYPE Conference [
  <!ELEMENT conf    (title,date,editor?,paper*)>
  <!ATTLIST conf    id     ID          #REQUIRED>
  <!ELEMENT title   (#PCDATA)>
  <!ELEMENT date    EMPTY>
  <!ATTLIST date    year   CDATA       #REQUIRED
                    mon    CDATA       #REQUIRED
                    day    CDATA       #IMPLIED>
  <!ELEMENT editor  (person*)>
  <!ATTLIST editor  eids   IDREFS      #IMPLIED>
  <!ELEMENT paper   (title,contact?,author,cite?)>
  <!ATTLIST paper   id     ID          #REQUIRED>
  <!ELEMENT contact EMPTY>
  <!ATTLIST contact aid    IDREF       #REQUIRED>
  <!ELEMENT author  (person+)>
  <!ATTLIST autho   id     ID          #REQUIRED>
  <!ELEMENT person  (name,email?)>
  <!ATTLIST person  id     ID          #REQUIRED>
  <!ELEMENT name    EMPTY>
  <!ATTLIST name    fn     CDATA       #IMPLIED
                    ln     CDATA       #REQUIRED>
  <!ELEMENT email   (#PCDATA)>
  <!ELEMENT cite    (paper*)>
  <!ATTLIST cite    id     ID          #REQUIRED
                    format (ACM|IEEE)  #IMPLIED>
]>
```

For instance, Table 1 shows a DTD for `Conference` which states that a `conf` element can have four sub-elements: `title`, `date`, `editor` and `paper` in that order. As common in regular expression, 0 or 1 occurrence (i.e., *optional*) is represented by the symbol `?`, 0 or more occurrences is represented by the symbol `*`, and 1 or more occurrences is represented by the symbol `+`. A sub-element without any such symbols (e.g., `title`) represents a *mandatory* one.

Keywords `#PCDATA` and `CDATA` are used as *string* types for elements and attributes, respectively. For instance, the type of the `title` element is defined as `#PCDATA` so that `title` element can be arbitrary character data. `<attribute-option>` can be either `#REQUIRED` or `#IMPLIED`. An attribute with a `#REQUIRED` option is a *mandatory* one while an attribute with a `#IMPLIED` option is an *optional* one. `<attribute-type>` keywords `ID` and `IDREF` are used for the pointed and pointing attributes, respectively. `IDREFS` is a plural form of `IDREF`. For instance, the `author` element must have a mandatory `id` attribute and this attribute is used when other attributes point to this attribute. On the other hand, the `contact` element has a mandatory `aid` attribute that must point to the `id` attribute of the contacting `author` of the current paper. One interesting definition in Table 1 is the `cite` element; it can have zero or more `paper` elements as sub-elements, thus creating a cyclic definition.

Table 2: A valid XML document conforming to the DTD for `Conference` in Table 1.

```
<conf id="er99">
  <title>Int'l Conference on Conceptual Modeling (ER)</title>
  <date>
    <year>1999</year> <mon>May</mon> <day>20</day>
  </date>
  <editor eids="sheth bossy">
    <person id="klavans">
      <name fn="Judith" ln="Klavans" />  <email>klavans@cs.columbia.edu</email>
    </person>
  </editor>
  <paper id="p1">
    <title>Indexing Model for Structured Documents</title>
    <contact aid="dao"/>
    <author>
      <person id="dao"> <name fn="Tuong" ln="Dao" /> </person>
    </author>
  </paper>
  <paper id="p2">
    <title>Logical Information Modeling of Heterogeneous Digital Assets</title>
    <contact aid="shah"/>
    <author>
      <person id="shah">
        <name fn="Kshitij" ln="Shah" />
      </person>
      <person id="sheth">
        <name fn="Amit" ln="Sheth" />
        <email>amit@cs.uga.edu</email>
      </person>
    </author>
    <cite id="c100" format="ACM">
      <paper id="p3">
        <title>Making Sense of Scientific Information on World Wide Web</title>
        <author>
          <person id="bossy">
            <name fn="Marcia" ln="Bossy" />
          </person>
        </author>
      </paper>
    </cite>
  </paper>
</conf>
<paper id="p7">
  <title>Constraints-preserving Transformation from the XML...</title>
  <contact aid="lee"/>
  <author>
    <person id="lee">
      <name fn="Dongwon" ln="Lee" />  <email>dongwon@cs.ucla.edu</email>
    </person>
  </author>
  <cite id="c200" format="IEEE" />
</paper>
...
```

Table 2 shows a valid XML document conforming to the DTD for `Conference`. The document represents a portion of the fictional ER conference held in 1999. The first two `paper` elements are described with `id="p1"` and `id="p2"`, respectively. The `paper` element with `id="p2"` further has a `cite` element that describes the references in the paper. The `paper` element with `id="p7"` shows an example of the valid XML document that is *not* rooted at `conf` element. Note that a valid XML document can be rooted at any level of the DTD hierarchy as long as their sub-elements and attributes follow the DTD syntax.

2.3 Assumptions

Without loss of generality, to simplify our presentation, we assume that XML documents have the following properties:

1. The XML documents are all *valid*. That is, the document has a DTD and conforms to that DTD. Elements may be nested only in the way described by the DTD and may only have attributes allowed by the DTD. When a DTD is not available, we assume that we can infer a DTD by applying DTD inference algorithms such as [10, 14].

2. Since the focus of this paper is on the *data* aspect, XML features such as *entities* or *notations* are not covered.

3. In general, DTD can be complex. Especially, the sub-elements definition part can be declared in a complicated and *redundant* way. For instance, a definition `<!ELEMENT parent (child**|child?*)>` is valid, but can be further simplified to `<!ELEMENT parent (child*)>`. We assume that input DTD has been already simplified using a technique in [18].

3 Transforming DTD to Relational Schema

Transforming a hierarchical XML model to a flat relational model is not a trivial task. There are several difficulties including non 1-to-1 mapping, set values, recursion, and fragmentation issues [18]. Recently, the idea of using relational databases for XML storage system has attracted a lot of attention and several transformation algorithms have been proposed (e.g., [18, 11, 10, 4]). For a better presentation, we chose one particular transformation algorithm, called the *hybrid inlining algorithm* [18]. It is chosen since it exhibits the pros of the other two competing algorithms in [18] without severe side effects and it is a more generic algorithm than those in [4, 10]. Since issues of discovering and preserving semantic constrains in this paper is orthogonal to that of transformation algorithms, our technique can be applied to other transformation algorithms easily.

3.1 Hybrid Inlining Algorithm

The *hybrid* algorithm essentially does the following[1]:

1. Given a DTD, create a *DTD graph* that represents the structure of the DTD. A DTD graph can be constructed when parsing the given DTD. Its nodes are elements, attributes, or operators in DTD.

[1]Note that we modified the hybrid algorithm a bit for a better presentation and corrected a few minor problems, but the crux of the algorithm still remains intact.

Each element appears exactly once in the graph, while attributes and operators appear as many times as they appear in the DTD. Further, attributes with `#IMPLIED` or `IDREFS` type are converted to an operator node "`?`" or "`+`" in a DTD graph.

2. Identify *top nodes* in a DTD graph that are the nodes satisfying any of the following conditions: 1) not reachable from anyone (e.g., source node), 2) direct child of "`*`" or "`+`" operator node, 3) recursive node with indegree > 1, or 4) one node between two mutually recursive nodes with indegree $= 1$. Then, starting from a top node T, *inline* all the elements and attributes at *leaf nodes* reachable from T unless they are other top nodes themselves.

3. Attribute names are composed by the concatenated path from the top node to the leaf node using "`_`" as a delimiter.

4. Use an attribute with `ID` type as a key if provided. Otherwise, add a system-generated integer key[2].

5. If a table corresponds to the shared element with indegree > 1 in DTD, then add a field `parent_elm` to denote the parent element to which the current tuple belongs. Further, for each shared element, a new field `fk_$X$` is added as a *foreign key* to record the key values of parent element X. If X is inlined into another element Y, then record the Y's key value in the `fk_$Y$` field instead.

6. Inlining an element Y into a table corresponding to another element X (i.e., top node) creates a problem when an XML document is rooted at the element Y. To facilitate queries on such elements, a new field `root_elm` is added to a table r.

7. If an *ordered* DTD model is used, a field `ordinal` is added to record position information of sub-elements in the element. (For a simpler presentation, we did not show the `ordinal` field in this paper.)

For further details of the algorithm, refer to [18]. Figure 2 illustrates a DTD graph that is created from the DTD in Table 1. Table 3 shows the output of the transformation by the hybrid algorithm.

Among 11 elements in the DTD in Table 1, 4 elements – `conf`, `paper`, `person`, and `eids` – are top nodes and thus chosen to be mapped to the different tables. For the top node `conf`, the elements `date`, `title`, and `editor` are reachable and thus inlined. Then, the `id` attribute is used as a key and the `root_elm` field is added. For the top node `paper`, the elements `title`, `contact_aid`, `author`, `cite_format` and `cite_id` are reachable and inlined. Since the `paper` element is shared by the `conf` and `cite` elements (two incoming edges in a DTD graph), new fields `parent_elm`, `fk_conf` and `fk_cite` are added to record who and where the parent node was. Note that in the `paper` table of Table 1, a tuple with `id="p7"` has the value `"paper"` for the `root_elm` field. This is because the element <`paper id="p7"`> is rooted in the DTD in Table 1 without being embedded in other element. Consequently, its `parent_elm`, `fk_conf` and `fk_cite` fields are null. For the top node `person`, the elements `name_fn`, `name_ln` and `email` are reachable and inlined. Since the `person` is shared by the `author` and `editor` elements, again, the `parent_elm` is added. Note that in the `person` table of Table 1, a tuple with `id="klavans"` has the value `"editor"`, not `"paper"`, for the `parent_elm` field. This implies that "klavans" is in fact an `editor`, not an `author` of the paper.

[2]Note that in practice, even if there is an attribute with `ID` type, one may decide to have a system-generated key for better performance.

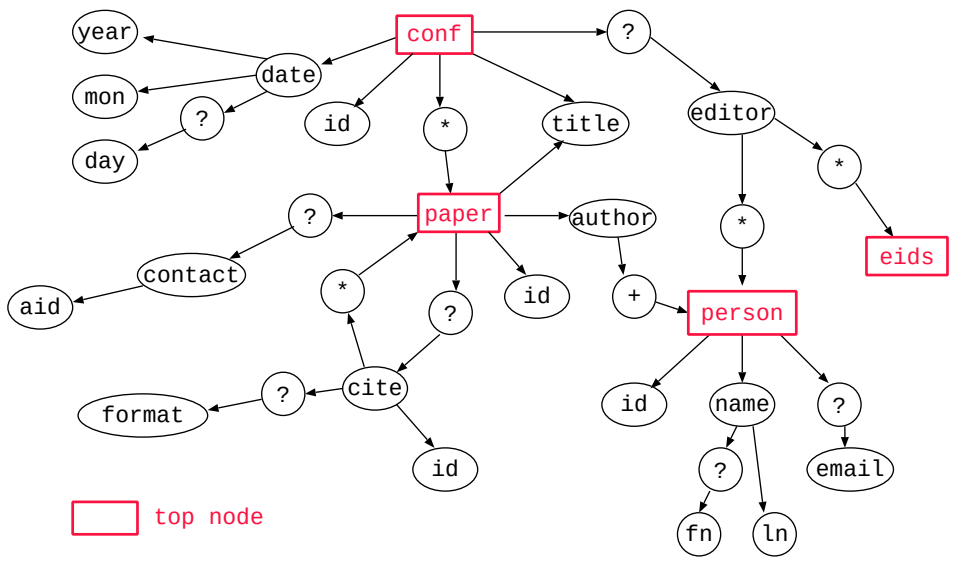

Figure 2: A DTD graph for the DTD in Table 1.

Table 3: A relational scheme ($\mathcal{S}$) along with the associated data that are converted from the DTD in Table 1 and XML document in Table 2 by the hybrid algorithm. Note that the hybrid algorithm does not generate *semantic constraints* (Δ).

conf

id	root_elm	title	date_year	date_mon	date_day
er99	conf	ER	1999	May	20

conf_editor_eids

id	root_elm	fk_conf	eids
100001	conf	er99	sheth
100002	conf	er99	bossy

paper

id	root_elm	parent_elm	fk_conf	fk_cite	title	contact_aid	cite_id	cite_format
p1	conf	conf	er99	–	Indexing ...	dao	–	–
p2	conf	conf	er99	–	Logical ...	shah	c100	ACM
p3	conf	cite	–	c100	Making ...	–	–	–
p7	paper	–	–	–	Constraints ...	lee	c200	IEEE

person

id	root_elm	parent_elm	fk_conf	fk_paper	name_fn	name_ln	email
klavans	conf	editor	er99	–	Judith	Klavans	klavans@cs.columbia.edu
dao	conf	paper	–	p1	Tuong	Dao	–
shah	conf	paper	–	p2	Kshitij	Shah	–
sheth	conf	paper	–	p2	Amit	Sheth	amit@cs.uga.edu
bossy	conf	paper	–	p3	Marcia	Bossy	–
lee	paper	paper	–	p7	Dongwon	Lee	dongwon@cs.ucla.edu

4 Semantic Constraints in DTD

In this section, we present the types of semantic constraints hidden in DTD and show how to preserve them into relational schema.

4.1 Domain Constraints

When the domain of the attributes is restricted to a certain specified set of values in DTD, it is called *Domain Constraints*. For instance, in the following DTD, the domain of the attributes `gender` and `married` are restricted.

```
<!ATTLIST author gender  (male|female) #REQUIRED
                 married (yes|no)      #IMPLIED>
```

In transforming such DTD into relational schema, we can enforce the domain constraints using SQL `CHECK` clause as follows:

```
CREATE DOMAIN gender VARCHAR(10) CHECK (VALUE IN ("male", "female"))
CREATE DOMAIN married VARCHAR(10) CHECK (VALUE IN ("yes", "no"))
```

When the mandatory attribute is defined by the `#REQUIRED` keyword in DTD, it needs to be forced in the transformed relational schema as well. That is, the attribute `ln` cannot be omitted below.

```
<!ELEMENT person    EMPTY>
<!ATTLIST person    fn   CDATA  #IMPLIED
                    ln   CDATA  #REQUIRED>
```

We use the notation "$X \nrightarrow \emptyset$" to denote that an attribute X cannot be null. This kind of domain constraint can be best expressed by using the `NOT NULL` clause in SQL as follows:

```
CREATE TABLE person (fn VARCHAR(20), ln VARCHAR(20) NOT NULL, ...)
```

More complex kinds of domain constraints can be inferred from the specification of the cardinality of the sub-elements. For instance, below, the `conf` element must have `title` and `date` sub-elements. Further, the `date` element must have `year` and `mon` sub-elements, but not necessarily `day` sub-element.

```
<!ELEMENT conf    (title,date,editor?,paper*)>
<!ELEMENT date    (#PCDATA)>
<!ATTLIST date    year CDATA  #REQUIRED
                  mon  CDATA  #REQUIRED
                  day  CDATA  #IMPLIED>
```

Therefore, the corresponding attributes in the transformed table `conf` in Table 3 should have been defined as `NOT NULL` as follows:

```
CREATE TABLE conf (
  title     VARCHAR(50) NOT NULL,
  date_year NUMBER      NOT NULL,
  date_mon  VARCHAR(3)  NOT NULL, ...
)
```

4.2 Cardinality Constraints

In DTD declaration, there are only 4 possible cardinality relationships between an element and its sub-elements as illustrated below:

```
<!ELEMENT article (title, author+, reference*, price?)>
```

1. 1-to-{0,1} mapping ("at most" semantics): An element can have either zero or one sub-element. (e.g., sub-element `price`)
2. 1-to-{1} mapping ("only" semantics): An element must have one and only one sub-element. (e.g., sub-element `title`)
3. 1-to-{0, ...} mapping ("any" semantics): An element can have zero or more sub-elements. (e.g., sub-element `reference`)
4. 1-to-{1, ...} mapping ("at least" semantics): An element can have one or more sub-elements. (e.g., sub-element `author`)

For convenience, let us call each cardinality relationship type A, B, C, and D, respectively. From these cardinality relationships, mainly three constraints can be inferred. First, whether or not the sub-element can be null. This constraint is easily enforced by the `NULL` or `NOT NULL` clause. Second, whether or not more than one sub-elements can occur. This is also known as *singleton constraint* in [21] and is one kind of equality-generating dependencies and further discussed in Section 4.4. Third, given an element, whether or not its sub-element should occur. This is one kind of tuple-generating dependencies and is further discussed in Section 4.5.

4.3 Inclusion Dependencies (IDs)

An *Inclusion Dependency* assures that values in the columns of one fragment must also appear as values in the columns of other fragment and is a generalization of the notion *referential integrity*.

Trivial form of IDs found in DTD is that "given an element X and its sub-element Y, Y must be included in X (i.e., $Y \subseteq X$)". For instance, from the `conf` element and its four sub-elements in DTD, the following IDs can be found as long as `conf` is not null: {`conf.title` $\subseteq$ `conf`, `conf.date` $\subseteq$ `conf`, `conf.editor` $\subseteq$ `conf`, `conf.paper` $\subseteq$ `conf`}. Another form of IDs can be found in the attribute definition part of DTD with the use of the `IDREF(S)` keyword. For instance, consider the `contact` and `editor` elements in the DTD in Table 1 shown below:

```
<!ELEMENT person  (name,email?)>
<!ATTLIST person  id   ID     #REQUIRED>
<!ELEMENT contact EMPTY>
<!ATTLIST contact aid  IDREF  #REQUIRED>
<!ELEMENT editor  (person*)>
<!ATTLIST editor  eids IDREFS #IMPLIED>
```

The DTD restricts the `aid` attribute of the `contact` element such that it can only point to the `id` attribute of the `person` element[3]. Further, the `edis` attribute can only point to multiple `id` attributes of the `person`

[3]Precisely speaking, the DTD does not tell whom the `aid` attribute should point to. This information is available only by human expert.

element. As a result, the following IDs can be derived: {`editor.eids` $\subseteq$ `person.id`, `contact.aid` $\subseteq$ `person.id`, `cite.pids` $\subseteq$ `paper.id`}. IDs can be best enforced by the "foreign key" concept if the attribute being referenced is a primary key. Otherwise, it needs to use the `CHECK`, `ASSERTION`, or `TRIGGERS` facility in SQL.

4.4 Equality-Generating Dependencies (EGDs)

The *Singleton Constraint* [21] restricts an element to have "at most" one sub-element. When an element type X satisfies the singleton constraint towards its sub-element type Y, if an element instance x of type X has *two* sub-elements instances y_1 and y_2 of type Y, then y_1 and y_2 must be the same. This property is known as *Equality-Generating Dependencies (EGDs)* and denoted by "$X \rightarrow Y$" in database theory. For instance, two EGDs: {`conf` $\rightarrow$ `conf.title`, `conf` $\rightarrow$ `conf.date`} can be derived from the `conf` element in Table 1. This kind of EGDs can be enforced by SQL `UNIQUE` construct. In general, EGDs occur in the case of the 1-to-{0,1} and 1-to-{1} mappings in the cardinality constraints.

4.5 Tuple-Generating Dependencies (TGDs)

Tuple-Generating Dependencies (TGDs) in relational model require that some tuples of a certain form be present in the table and use the "$\twoheadrightarrow$" symbol. Two useful forms of TGDs from DTD are the *child* and *parent constraints* [20].

1. **Child constraint:** `"Parent` $\twoheadrightarrow$ `Child"` states that every element of type *Parent* must have at least one child element of type *Child*. This is the case of the 1-to-{1} and 1-to-{1,...} mappings in the cardinality constraints. For instance, from the DTD in Table 1, since the `conf` element must contain the `title` and `editor` sub-elements, the child constraint `conf` $\twoheadrightarrow$ {`title, editor`} holds.

2. **Parent constraint:** `"Child` $\twoheadrightarrow$ `Parent"` states that every element of type *Child* must have a parent element of type *Parent*. According to XML specification, there is no notion of *root* in DTD. That is, XML documents can start from any level of elements without necessarily specifying its parent element. Therefore, parent constraints cannot be assured simply by looking at DTD structure. Rather, it requires some semantic knowledge. In the DTD in Table 1 again, for instance, the `editor` and `date` elements can have the `conf` element as their parent. Further, if we know that all XML documents were started at the `conf` element level rather than the `editor` or `date` level, then the parent constraint {`editor, date`} $\twoheadrightarrow$ `conf` holds. Note that the `title` $\twoheadrightarrow$ `conf` does not hold since the `title` element can be a sub-element of either the `conf` or `paper` element.

5 Discovering and Preserving Semantic Constraints

In this section, we describe how to systematically discover semantic constraints from DTD and how to preserve and re-write them in relational terms.

5.1 Data Structures

To help find semantic constraints, we use the following data structure:

Definition 1. *An* **annotated DTD graph (ADG)** $\mathcal{G}$ *is a pair* ($\mathcal{V}$, $\mathcal{E}$)*, where* $\mathcal{V}$ *is a finite set and* $\mathcal{E}$ *is a binary relation on* $\mathcal{V}$*. The set* $\mathcal{V}$ *consists of element and attributes in a DTD. Each edge* $e \in \mathcal{E}$ *is*

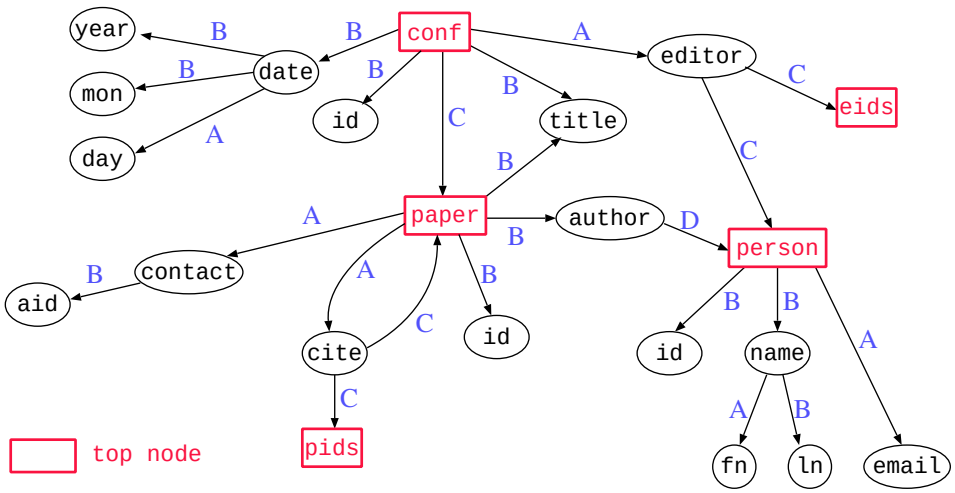

Figure 3: An *Annotated DTD graph* for the DTD in Table 1. The associated values (i.e., indegree, type, tag, and status) for the nodes are not shown.

labeled with the cardinality relationship types (A to D) as defined in Section 4.2. In addition, each vertex $v \in \mathcal{V}$ *carries the following information:*

1. `indegree` *stores the number of incoming edges.*
2. `type` *contains the element type name in the content model of the DTD (e.g.,* `conf` *or* `paper`*).*
3. `tag` *stores a flag value whether the node is an element or attribute (if attribute, it contains the attribute keyword like* `ID` *or* `IDREF`*, etc.).*
4. `status` *contains "visited" flag if the node was visited in a depth-first search or "not-visited".*

■

Note that the cardinality relationship types in ADG considers not only element vs. sub-element relationships but also element vs. attribute relationships. For instance, from the DTD `<!ATTLIST X Y #IMPLIED Z #REQUIRED)>`, two types of cardinality relationships (i.e., type A between element X and attribute Y, and type B between element X and attribute Z) can be derived. Figure 3 illustrates an example of ADG for the DTD in Table 1.

5.2 Discovering Semantic Constraints

The cardinality relationships can be used to find semantic constraints in a *systematic* fashion. Table 4 summarizes 3 main semantic constraints that can be derived from cardinality relationships.

Then, the following `FindConstraints()` algorithm immediately follows from Table 4.

5.3 Preserving Semantic Constraints

Semantic constraints discovered by `FindConstraints()` have additional usage as we shall show in Section 7 shortly. However, to enforce correct semantics in the newly generated relational schema, we need to re-write the semantic constraints in XML terms to ones in relational terms. Details are illustrated in Algorithm `RewriteConstraints()`.

Algorithm 1: `FindConstraints`

Input : Node v and w

switch `edge`(v,w) **do**
- **case** *type A* $v \rightarrow w$;
- **case** *type B* w is not null; $v \rightarrow w$; $v \twoheadrightarrow w$;
- **case** *type C* `/* empty */`;
- **case** *type D* w is not null; $v \twoheadrightarrow w$;

Algorithm 2: `RewriteConstraints`

Input : Constraints Δ' in XML notation

Output: Constraints Δ in relational notation

switch Δ' **do**
- **case** $X \nrightarrow \emptyset$
  - If X is mapped to attribute X' in table scheme A, then $A[X']$ cannot be null. (i.e., "`CREATE TABLE` A (...X' `NOT NULL`...)") ;
- **case** $X \subseteq Y$
  - If X and Y are mapped to attributes X' and Y' in table scheme A and B, respectively, then re-write it as $A[X'] \subseteq B[Y']$. (i.e., If Y' is a primary key of B, then "`CREATE TABLE` A (...`FOREIGN KEY` (X') `REFERENCES` B(Y')...)". Else "`CREATE TABLE` A (...(X') `CHECK` (X' IN (SELECT Y' FROM B))...") ;
- **case** $X \rightarrow X.Y$
  - If element X and Y are mapped to the same table scheme A (i.e., since Y is not a top node, Y becomes an attribute of table A) and Z is the key attribute of A, then re-write it as $A[Z] \rightarrow A[Y]$. (i.e., "`CREATE TABLE` A (...`UNIQUE` (Y), `PRIMARY KEY` (Z)...)");
- **case** $X \twoheadrightarrow X.Y$
  - **if** (element X and Y are mapped to the same table) **then**
    - Let A be the table and Z be the key attribute of A. Then re-write it as $A[Z] \twoheadrightarrow A[Y]$. (i.e., "`CREATE TABLE` A (...Y `NOT NULL`, `PRIMARY KEY` (Z)...)") ;
  - **else**
    - Let the tables be A and B, respectively and Z be the key attribute of A. Then re-write it as $B[fk_A] \subseteq A[Z]$. (i.e., "`CREATE TABLE` B (...`FOREIGN KEY` (fk_A) `REFERENCES` A(Z)...)")

return Δ;

Table 4: Cardinality relationships and their corresponding semantic constraints in DTD

Relationship	Type	Symbol	Semantics	not null	EGDs	TGDs
1-to-{0,1}	A	?	at most	no	yes	no
1-to-{1}	B		only	yes	yes	yes
1-to-{0,...}	C	*	any	no	no	no
1-to-{1,...}	D	+	at least	yes	no	yes

5.4 CPI: Constraints-preserving Inlining Algorithm

We shall now describe our complete DTD-to-relational schema transformation algorithm: *CPI (Constraints-preserving Inlining) algorithm* is a combination of the hybrid inlining algorithm, `FindConstraints()` and `RewriteConstraints()` algorithms. The CPI algorithm is illustrated in `CPI()` and `BuildTable()`. Functions used in the algorithm are: $Adj[v]$ returns all nodes adjacent to v in graph, `add`(A, B) adds an attribute A into a table B, `concat`(A, ..., Z) concatenates strings from A to Z, `topnode`(G) finds all top nodes in graph G, and `edge`(v, w) returns the edge from v to w.

The algorithm first identifies all the top nodes from the ADG. This can be done using algorithms to find sinks or strongly-connected components in a graph [8]. Then, for each top node, the algorithm generates a corresponding table scheme using `BuildTable()`. The associated constraints are found and re-written in relational terms using `FindConstraints()` and `RewriteConstraints()`. Algorithm `BuildTable()` scans an ADG in a depth-first search manner while finding constraints and inlines a new field in the leaf node. The final output schema is the union of all the table schemes and semantic constraints.

Algorithm 3: `CPI`

Input : Annotated DTD Graph $\mathcal{G} = (\mathcal{V}, \mathcal{E})$
Output: Relational Schema $\mathcal{R}$

$\mathcal{V} \leftarrow$ `topnode`($\mathcal{G}$);
for *each* $v \in \mathcal{V}$ **do**
  table_def $\leftarrow \{\}$;
  if *v.tag = 'element'* **then** `add`('root_elm', *table_def*); `/* start where? */`
  if *v.indegree > 1* **then**
    `add`('parent_elm', *table_def*); `/* shared elements case */`
    `add`(concat('fk_', parent(v)), *table_def*);
  $\mathcal{W} \leftarrow Adj[v]$; w $\in \mathcal{W}$;
  if *any w.tag =* `'ID'` **then** `add`(w.*type*, *table_def*);
  else `add`('id', *table_def*); `/* system-generated primary key */`
  $\mathcal{R} \leftarrow \mathcal{R}$ + `hybrid`(v, *table_def*, $\emptyset$);
return $\mathcal{R}$;

Table 5 contains the semantic constraints that are discovered during the transformation by the CPI algorithm. Table 6 contains the semantic constraints that are re-written in relational format. As an example, the CPI algorithm will eventually spit out the following SQL `CREATE` statement for the `paper` table. Note that not only is the relational scheme provided, but the semantic constraints are also ensured by use of the `NOT NULL`, `KEY`, `UNIQUE` or `CHECK` constructs.

Table 5: A partial list of semantic constraints in XML notation found from the DTD in Table 1. Additional semantic constraints can be derived by applying Armstrong's axioms (e.g., transitivity or augmentation, etc). The last row titled "semantic knowledge" shows the constraints that are inferred by human experts from both structural constraints as well as semantic knowledge discussed in Sections 4.3 and 4.5.

Edge Type	Semantic Constraints
A	conf $\rightarrow$ conf.editor conf.date $\rightarrow$ conf.date.day conf.paper $\rightarrow$ {conf.paper.contact,conf.paper.cite} conf.paper.cite $\rightarrow$ conf.paper.cite.format conf.paper.author.person $\rightarrow$ conf.paper.author.person.email conf.paper.author.person.name $\rightarrow$ conf.paper.author.person.name.fn
B	conf $\rightarrow$ {conf.date,conf.id,conf.title} conf.date $\rightarrow$ {conf.date.year,conf.date.mon} conf.paper $\rightarrow$ {conf.paper.id,conf.paper.author,conf.title} conf.paper.cite $\rightarrow$ conf.paper.cite.id conf.paper.contact $\rightarrow$ conf.paper.contact.aid conf.paper.author.person $\rightarrow$ {conf.paper.author.person.id, conf.paper.author.person.name} conf.paper.author.person.name $\rightarrow$ conf.paper.author.person.name.ln conf.editor.person $\rightarrow$ {conf.editor.person.id,conf.editor.person.name} conf.editor.person.name $\rightarrow$ conf.editor.person.name.ln conf $\twoheadrightarrow$ {conf.date,conf.id,conf.title} conf.date $\twoheadrightarrow$ {conf.date.year,conf.date.mon} conf.paper $\twoheadrightarrow$ {conf.paper.id,conf.paper.author,conf.title} conf.paper.cite $\twoheadrightarrow$ conf.paper.cite.id conf.paper.contact $\twoheadrightarrow$ conf.paper.contact.aid conf.paper.author.person $\twoheadrightarrow$ {conf.paper.author.person.id, conf.paper.author.person.name} conf.paper.author.person.name $\twoheadrightarrow$ conf.paper.author.person.name.ln conf.editor.person $\twoheadrightarrow$ {conf.editor.person.id,conf.editor.person.name} conf.editor.person.name $\twoheadrightarrow$ conf.editor.person.name.ln {conf.id,conf.date,conf.date.year,conf.date.mon,conf.title} $\nrightarrow \emptyset$ {conf.paper.id,conf.paper.title,conf.paper.author,conf.paper.cite.id,conf.paper.contact.aid} $\nrightarrow \emptyset$ {conf.paper.author.person.id,conf.paper.author.person.name,conf.paper.author.person.name.ln} $\nrightarrow \emptyset$ {conf.editor.person.id,conf.editor.person.name,conf.editor.person.name.ln} $\nrightarrow \emptyset$
D	conf.paper.author $\twoheadrightarrow$ conf.paper.author.person conf.paper.author.person $\nrightarrow \emptyset$
Semantic knowledge	conf.editor.eids $\subseteq$ conf.paper.author.person.id conf.paper.contact.aid $\subseteq$ conf.paper.author.person.id {conf.date,conf.id,conf.editor} $\twoheadrightarrow$ conf {conf.date.year,conf.date.mon,conf.date.day} $\twoheadrightarrow$ conf.date conf.editor.eids $\twoheadrightarrow$ conf.editor {conf.paper.author,conf.paper.id,conf.paper.contact} $\twoheadrightarrow$ conf.paper conf.paper.contact.aid $\twoheadrightarrow$ conf.paper.contact {conf.paper.cite.id,conf.paper.cite.format} $\twoheadrightarrow$ conf.paper.cite {conf.paper.author.person.id,conf.paper.author.person.name, conf.paper.author.person.email} $\twoheadrightarrow$ conf.paper.author.person {conf.paper.author.person.name.fn,conf.paper.author.person.name.ln} $\twoheadrightarrow$ conf.paper.author.person.name

Algorithm 4: `hybrid`

Input : Vertex v, TableDef $table_def$, string $attr_name$
Output: Relational Schema $\mathcal{R}$

$v.status \leftarrow$ 'visited';
for *each* $w \in Adj[v]$ **do**
  if $w.status$ = *'not-visited'* **then**
    $\Delta' \leftarrow$ `FindConstraints`(v, w);
    $\Delta \leftarrow$ `RewriteConstraints`(Δ');
    `hybrid`(w, $table_def$, `concat`($attr_name$, '_', $w.type$));
`add`($attr_name$, $table_def$);
$\mathcal{R} \leftarrow table_def + \Delta$;
return $\mathcal{R}$;

```
CREATE TABLE paper (
    id              NUMBER        NOT NULL,
    title           VARCHAR(50)   NOT NULL,
    contact_aid     NUMBER,
    cite_id         NUMBER,
    cite_format     VARCHAR(50)   CHECK (VALUE IN ("ACM", "IEEE")),
    root_elm        VARCHAR(20)   NOT NULL,
    parent_elm      VARCHAR(20),
    fk_cite         VARCHAR(20)   CHECK (fk_cite IN (SELECT cite_id FROM paper)),
    fk_conf         VARCHAR(20),
    PRIMARY KEY (id),
    UNIQUE (cite_id),
    FOREIGN KEY (fk_conf)      REFERENCES conf(id),
    FOREIGN KEY (contact_aid)  REFERENCES person(id)
);
```

6 Experimental Results

We have implemented the CPI algorithm in Java using the IBM XML4J package. Table 7 shows a summary of our experimentation. We gathered test DTDs from "http://www.oasis-open.org/cover/xml.html" and [17]. Since some DTDs had syntactic errors caught by the XML4J, we had to modify them manually. Note that people seldom used the `ID` and `IDREF(s)` constructs in their DTDs except the `XMI` and `BSML` cases. The number of the tables generated in relational schema was usually smaller than that of elements/attributes in DTD due to the inlining effect. The only exception to this phenomenon was the `XMI` case, where extensive use of type C and D cardinality relationships resulted in a lot of top nodes in the ADG.

The number of semantic constraints had close relationship with the way DTD hierarchy was designed and the type of cardinality relationship used in DTD. Since the `XMI` DTD had a lot of type C cardinality relationship, which could not contribute to the semantic constraints at all, the number of semantic constraints at the end was small compared to that of elements/attributes in DTD. This was also true for the `OSD` case. On the other hand, in the `ICE` case, since it used type B cardinality relationship a lot, it resulted in relatively abundant semantic constraints at the end.

Table 6: The semantic constraints in relational notation re-written from the semantic constraints in XML notation in Table 5.

Type	Semantic constraints in relational notation
ID	conf_editor_eids[eids] $\subseteq$ person[id] paper[contact_aid] $\subseteq$ person[id]
EGD	conf[id] $\rightarrow$ conf[title,date_year,date_mon,date_day] paper[id] $\rightarrow$ conf[title,contact_aid,cite_id,cite_format] person[id] $\rightarrow$ conf[name_fn,name_ln,email]
TGD	conf[id] $\twoheadrightarrow$ conf[title,date_year,date_mon,date_day] paper[id] $\twoheadrightarrow$ conf[title,contact_aid,cite_id,cite_format] person[id] $\twoheadrightarrow$ conf[name_fn,name_ln,email] conf_editor_eids[fk_conf] $\subseteq$ conf[id] paper[fk_conf] $\subseteq$ conf[id] paper[fk_cite] $\subseteq$ paper[cite_id] person[fk_conf] $\subseteq$ conf[id] person[fk_paper] $\subseteq$ paper[id]
not null	conf[id,title,date_year,date_mon,root_elm] $\nrightarrow \emptyset$ conf_editor_eids[id,root_elm] $\nrightarrow \emptyset$ paper[id,title,root_elm] $\nrightarrow \emptyset$ person[id,name_ln,root_elm] $\nrightarrow \emptyset$

Further detailed reports on the experimentation and the implementation of the CPI algorithm is available from [12].

7 Application of the Semantic Constraints

The constraints that are discovered during the transformation have two distinct usages: 1) they are converted into relational database format and used to ensure correct semantics in the resulting relational schema, and 2) they can be used as *semantic knowledge* in a variety of areas [1, 2, 13, 21]. Since the focus of this paper is not on the application of the constraints, in this section, we shall show a few motivating examples for the area of future research.

7.1 Semantic Query Optimization

The most common usage of the constraints occurs in semantic query optimization where a user's query is typically re-written using constraints to a simpler form to minimize the processing cost of the query. For instance, consider the following query Q_1: "Find titles of the paper that has at least an author with non-null last name". In XQL [16] notation, this query can be written as follows:

```
XQL:    /paper[author/person/name/ln]/title
```

The [ ] notation in XQL is called the *filter expression*. That is, the given query Q_1 finds all `paper` elements that have at least one sub-element `author` x, such that x has a sub-element `person` y as a child, such that y has a sub-element `name` z as a child, such that z has a sub-element `ln` as a child. When Q_1 is translated to SQL based on the relational schema in Table 3, it will be as follows:

```
SQL:    SELECT  P.title
```

Table 7: Results of CPI algorithm against the DTDs downloaded from "http://www.oasis-open.org/cover/xml.html" and [17].

DTD		DTD Schema				Relational Schema				
Name	Domain	Elm	Attr	ID	IDREF(S)	Table	Attr	$\rightarrow$	$\twoheadrightarrow$	$\nrightarrow \emptyset$
novel	literature	10	1	1	0	5	13	6	9	9
play	Shakespeare	21	0	0	0	14	46	17	30	30
tstmt	religious text	28	0	0	0	17	52	17	22	22
vCard	business card	23	1	0	0	8	19	18	13	13
ICE	content syndication	47	157	0	0	27	283	43	60	60
MusicML	music description	12	17	0	0	8	34	9	12	12
OSD	s/w description	16	15	0	0	15	37	2	2	2
PML	web portal	46	293	0	0	41	355	29	36	36
Xbel	bookmark	9	13	3	1	9	36	9	1	1
XMI	metadata	94	633	31	102	129	3013	10	7	7
BSML	DNA sequencing	112	2495	84	97	104	2685	99	33	33

```
FROM    paper P, person Q
WHERE   P.id = Q.fk_paper        AND
        Q.parent_elm = 'paper'   AND
        Q.name_ln != null
```

Note that the filter expression in XQL had to be translated to join expressions between the `paper` and `person` tables in SQL. However, if we had used the semantic constraints in query formation stage, we could have first created the following XQL query:

```
XQL:    /paper/title
```

Intuitively, this makes sense since the filter expression in Q_1 is satisfied by all the `paper` elements. That is, according to the DTD, all papers must have at least one `author` sub-element (paper $\twoheadrightarrow$ paper.author), `author` must have at least one `person` sub-element (author $\twoheadrightarrow$ author.person), `person` must have one `name` sub-element (person $\twoheadrightarrow$ person.name), and `name` must have one `ln` attribute (name $\twoheadrightarrow$ name.ln). Therefore, the filter expression is redundant and does not have to enforced in the translated SQL, resulting in the following SQL at the end:

```
SQL:    SELECT  title
        FROM    paper
```

7.2 Semantic Caching

In a client and server architecture, client caching is commonly used to speedup query response time and to prepare for unexpected network partition. When such client caching uses the user's query description as key value to local cache, it is called *semantic caching*. To maximize the usage of such client caching, we recently proposed a technique called *query matching* in [13]. In the query matching technique, a user's query is examined to determine if it can be answered from any of the locally stored answers with the help of semantic knowledge to avoid unnecessary access to the server.

Suppose a client cache stored the following query Q_1 that selects `person` elements that are directly or indirectly related to ER `conf` element:

```
XQL:    /conf[title='ER']/*/person
```

This query can be translated to the following SQL query based on the relational schema in Table 3:

```
SQL:    SELECT  P2.id, P2.name_fn, P2.name_ln, P2.name_email
        FROM    conf C, conf_editor_eids C2, paper P, person P2
        WHERE   C.title = 'ER'  AND
                (C.id = P.fk_conf    AND  P.id = P2.fk_paper)
        OR      (C.id = C2.fk_conf   AND  C2.eids = P2.id)
```

Now the user asks the second query Q_2 that selects the editor's names of the ER `conf` element:

```
XQL:    /conf[title='ER']/editor/person/name
```

Then, Q_2 does not have to be shipped to the server to select answers since $Q_2 \subseteq Q_1$. This is intuitively true since in both XQL queries Q_1 and Q_2, `person.name` $\subseteq$ `person` and `editor` $\subseteq$ `*`. Therefore, given the cached answer A_1 to the query Q_1, answers A_2 to the query Q_2 can be obtained by computing "$A_2 = A_1 \wedge Q_2$" on the client side, which is more efficient than sending Q_2 to and receiving answers from the server.

8 Related Work

Constraints and semantic knowledge play an important role in XML query processing [1, 2, 6, 20, 21]. Since our CPI algorithm provides a systematic way of finding and preserving constraints from a DTD, ours is an improvement to the existing transformation algorithms (e.g, [18, 11]). Work done in STORED [10] deals with *non-valid* XML documents. When input XML documents do not conform to the given DTD, STORED uses a data mining technique to find a representative DTD whose support exceeds the predefined threshold. Since our algorithm to find and preserve constraints are not directly tied to a single transformation algorithm, ours can be applied to this algorithm as well. [14] also presents a DTD inference algorithm when it is not known. [4] discusses template language-based transformation from XML DTD to relational schema which requires human expert to write an XML-based transformation rule.

Some work has been done in [19] dealing with the transformation from relational tables to XML documents. There has been some transformation work in OODB area as well [7]. Since OODB is a richer environment than RDB, their work is not readily applicable to our application. The logical database design methods and their associated transformation techniques to other data models have been extensively studied in ER research. For instance, [3] presents an overview of such techniques. However, due to the differences between ER and XML models, those transformation techniques need to be modified substantially.

9 Conclusion

Since the schema design in relational databases greatly affects the query processing efficiency, how to transform the XML DTD to its corresponding relational schema is an important problem. Further, due

to XML DTD's peculiar characteristics and its incompatibility between the hierarchical XML and flat relational model, the transformation process is not a straightforward task.

After showing a variety of semantic constraints hidden implicitly or explicitly in DTD, we presented two algorithms on: 1) how to discover the semantic constraints using one of the existing transformation algorithms, and 2) how to re-write the semantic constraints in relational notation. Then, using a complete example developed through the paper, we showed semantic constraints found in both XML and relational terms. The final relational schema transformed from our CPI algorithm not only captures the *structure*, but also the *semantics* of the given DTD. Further research direction of using the semantic constraints towards *query optimization* and *semantic caching* is also presented.

References

[1] Abiteboul, S., Buneman, P., Suciu, D. "Data on the Web: From Relations to Semistructured Data and XML", *Morgan Kaufmann Publishers*, 2000

[2] Böhm, K., Aberer, K., Öszu, M. T., Gayer, K. "Query Optimization for Structured Documents Based on Knowledge on the Document Type Definition", *Proc. IEEE Advances in Digital Libraries (ADL)*, Los Alamitos, California, April, 1998

[3] Batini, C., Ceri, S., Navathe, S. B. "Conceptual Database Design: An Entity-Relationship Approach", *The Benjamin/Cummings Publishing Company, Inc.*, 1992

[4] Bourret, R. "XML and Databases", *Internet Document*, September, 1999 (http://www.informatik.tu-darmstadt.de/DVS1/staff/bourret/xml/XMLAndDatabases.htm)

[5] Bray, T., Paoli, J., Sperberg-McQueen, C. M. (ed.), "Extensible Markup Language (XML) 1.0", *W3C Recommendation*, Feburary, 1998 (http://www.w3.org/TR/REC-xml)

[6] Che, D., Aberer, K. "A Heuristics-Based Approach to Query Optimization in Structured Document Databases", *IEEE Int'l Database Engineering and Application Symp.*, Montreal, Canada, August, 1999

[7] Christophides, V., Abiteboul, S., Cluet, S., Scholl, M. "From Structured Document to Novel Query Facilities", *Proc. ACM SIGMOD*, Minneapolis, Minnesota, May, 1994

[8] Cormen, T. H., Leiserson, C. E., Rivest, R. L. "Introduction to Algorithms", *The MIT Press*, 1992

[9] Deutsch, A., Fernandez, M. F., Florescu, D., Levy, A., Suciu, D. "XML-QL: A Query Language for XML", *Proc. The Query Language Workshop (QL)*, December, 1998 (http://www.w3.org/TR/NOTE-xml-ql)

[10] Deutsch, A., Fernandez, M. F., Suciu, D. "Storing Semistructured Data with STORED", *Proc. ACM SIGMOD*, Philadephia, Pennsylvania, June, 1998

[11] Florescu, D., Kossmann, D. "Storing and Querying XML Data Using an RDBMS", *IEEE Data Engineering Bulletin*, 22(3), September, 1999.

[12] Lee, D. "XPRESS Home Page" (http://www.cs.ucla.edu/~ dongwon/xpress/), 2000.

[13] Lee, D., Chu, W. W. "Semantic Caching via Query Matching for Web Sources", *Proc. ACM CIKM*, Kansas City, MO, 1999

[14] Ludäescher, B., Papakonstantinou, Y., Velikhov, P., Vianu, V. "View Definition and DTD Inference for XML", *Proc. Post-ICDT Workshop on Query Processing for Semistructured Data and Non-Standard Data Formats*, 1999

[15] Raggett, D., Hors, A. L., Jacobs, I. (ed.), "HTML 4.0 Specification", *W3C Recommendation*, April, 1998 (http://www.w3.org/TR/REC-html40/)

[16] Robie, J., Lapp, J., Schach, D. "XML Query Language (XQL)", *WWW The Query Language Workshop (QL)*, December, 1998 (http://www.w3.org/TandS/QL/QL98/pp/xql.html)

[17] Sahuguet, A. "Everything You Ever Wanted to Know About DTDs, But Were Afraid to Ask", *Proc. 3rd Int'l Workshop on the Web and Databases (WebDB)*, Dallas, TX, 2000

[18] Shanmugasundaram, J., Tufte, K., He, G., Zhang, C., DeWitt, D., Naughton, J. "Relational Databases for Querying XML Documents: Limitations and Opportunities", *Proc. VLDB*, Edinburgh, Scotland, 1999

[19] Turau, V. "Making Legacy Data Accessible for XML Applications", *Internet Document*, 1999 (http://www.informatik.fh-wiesbaden.de/~ turau/veroeff.html)

[20] Wood, P. T. "Optimizing Web Queries Using Document Type Definitions", *Proc. Int'l Workshop on Web Information and Data Management (WIDM)*, 1999

[21] Wood, P. T. "Rewriting XQL Queries on XML Repositories", *Proc. 17th British National Conf. on Databases*, 2000