Constraint Preserving XML Storage in Relations

Yi Chen, Susan B. Davidson and Yifeng Zheng
Dept. of Computer and Information Science
University of Pennsylvania
yicn@saul.cis.upenn.edu, susan@cis.upenn.edu, yifeng@saul.cis.upenn.edu

Abstract

As XML becomes a standard for data representation on the internet, there is a growing interest in storing XML using relational database technology. To date, none of these techniques have considered the semantics of XML as expressed by keys and foreign keys. In this paper, we present a storage mapping which preserves not only the content and structure of XML data, but also its semantics.

1 Introduction

Over the past several years, there has been a tremendous interest in using relational databases to store XML documents[1, 2, 3, 4, 5], thus leveraging a well-developed technology for data management and query processing. There have also recently been several proposals for capturing constraints beyond DTDs, in particular keys [6] and foreign keys. Aspects of these proposals are finding their way into XMLSchema [7] where, following the older notions of ID and IDREF, foreign keys are termed "keyrefs". The question naturally arises as to how to capture XML key and keyref constraints in the relational mapping.

For example, consider a community of biomedial researchers who are performing gene expression experiments and exchange data using the XML standard MAGE-ML [8]. This standard includes a specification of keys and keyrefs, and each group is expected to produce data that is correct with respect to these keys. Upon the recommendation of their bioinformatic experts, they use relational database technology to store their experimental data (e.g. the Stanford Microarray Database [9], which uses Oracle). Since the data is already stored in a relational database, they wish to ensure that the data in relational form is correct with respect to the XML keys using relational technology.

Key and foreign key constraints can always be expressed as queries which produce an empty result when evaluated against a correct instance and a non-empty result otherwise. Thus XML constraints can always be checked using triggered procedures in a relational database. However, triggers are very inefficient compared to key and foreign key constraints in relational databases [10]. To fully leverage database technology for constraint checking, we therefore wish to map XML key and keyref constraints to relational key and foreign key constraints.

In this paper, we address the problem of mapping an XML document together with its constraints into a relational schema so as to check XML key and keyref constraints using key and foreign key constraints. The mapping should preserve three kind of information: 1) the *content* of the document, i.e. each node of the document should appear in the target database; 2) the *structure* of the document, i.e. the parent-child relationship of the nodes; [1] and 3) the *semantics* of the document as captured by XML key and keyref constraints. We will call such a mapping an *information preserving mapping*, and focus on mappings in which the schema is available.

This work makes the following contributions:

1. We distinguish three categories of information which need to be preserved in an information lossless mapping.

2. We propose a notion of *constraint relations* which explicitly capture XML key and keyref constraints.

3. We present a mapping from an XML document to a relational instance which extends constraint relations to capture the complete content and structure of the document.

4. XML key/keyref constraints can be checked using key and foreign key constraints in the constraint relations.

The rest of the paper is organized as follows: A definition of XML keys and keyrefs is given in Section 2. After introducing constraint relations in Section 3, we give an information preserving storage mapping algorithm called $X2R$. We close by reviewing related work and discussing future directions.

[1] Note that we are not considering other structure information such as cardinality constraints.

```
<root>
    <city>
    <name> Philadelphia </name>
    <state> PA </state>
       <restaurants>
          <cuisine type=''French''>
             <restaurant>
                <name> Le Soir </name>
                <entree>
                  <name>
                  Braised Cod Loin and Squid
                  </name>
                  <price>$24</price>
                </entree>
                <dessert>
                  <name>
                  Apple French Toast
                  </name>
                  <price>$10</price>
                </dessert> ...
             </restaurant> ...
          </cuisine> ...
       </restaurants>
       <reviews>
          <review restaurant="Le Soir">
             Excellent
          <review>  ...
       </reviews>
    </city> ...
</root>
```

Figure 1: Sample XML document

2 Constraints: Keys and keyrefs

Before giving a formal definition of keys for XML, consider the sample document "restaurants.xml" shown in Figure 1. The document contains information about restaurants and reviews of restaurants by cities. Within a city, the restaurants are grouped by their cuisine. As for the semantics of the document, one might wish to assert that a city is identified by its name and state, and that within the context of a city a restaurant is uniquely identified by its name. For example, since there is already a restaurant named *Le Soir* in Philadelphia PA, we could not add another with the same name in Philadelphia PA; however, we could add another restaurant named *Le Soir* in Seattle WA. We might also wish to assert that each menu item in a restaurant (which can be either an appetizer, entree, salad or dessert) is uniquely identified by its name. For example, since there is an dessert named *Apple French Toast* at the restaurant *Le Soir*, there can not also be an entree with that name. Finally, we might wish to assert that each review refers to a restaurant by name, i.e. the value of the restaurant attribute of a review is the name of some restaurant.

To define a key for XML we specify three things: 1) the context in which the key must hold; 2) a set on which we are defining a key; and 3) the values which distinguish each element of the set. Since we are working with hierarchical data, specifying the context, set and values involve path expressions. In what follows, we adopt the syntax of [6] for keys[2] and use the following notation: $n[\![P]\!]$ denotes the set of node identifiers in the XML tree representation of the document that can be reached by following the path expression P from the node with identifier n. We also use $[\![P]\!]$ as an abbreviation for $r[\![P]\!]$, where r is the root node.

An *XML key* can be written as

$$K : (C^K, (T^K, \{P_1^K, \ldots, P_p^K\}))$$

where K is the name of the key, path expressions C^K is called the *context path*, T^K the *target path*, and $P_1^K, ..., P_p^K$ the *key paths* for the key. For a *context node* $c \in [\![C^K]\!]$, relative to a *target node* $t \in c[\![T^K]\!]$, key path $P_i^K (i = 1, ..., p)$ must identify a single *key node* (either an element or attribute) whose value must be of a simple type. That is, following XMLSchema we require that each P_i^K exist and be unique (*strong* keys, [6]); furthermore, the value at the end of a key path must be of a simple type (rather than an XML tree value as in [6]). The idea is that within the scope of a context node, the key constraint must hold on the set of target nodes.

Definition 2.1: An XML tree T is said to *satisfy* a key $K : (C^K, (T^K, \{P_1^K, \ldots, P_p^K\}))$ if and only if $\forall c \in [\![C^K]\!]$, $\forall t_1, t_2 \in c[\![T^K]\!]$, $t_1[\![P_i^K]\!] = t_2[\![P_i^K]\!](i = 1, \ldots, p) \rightarrow t_1 = t_2$. ■

In relational databases, a *foreign key* allows a list of attributes in one table to reference a list of attributes in another table (which must form the key for the referred table). Since XML is hierarchical, we must additionally specify the context in which the references are made. Our notation for a keyref is therefore similar to that of a key:

$$R : (C^R, (T^R, \{P_1^R, \ldots, P_p^R\})) \text{ KEYREF } K$$

where K is the name of the key being referenced, R is the name of the keyref, C^R is the context path, T^R is the target path, and $P_1^R, ..., P_p^R$ are the *key paths* for the keyref. The concatenation of the context and target paths locates the referencing node. As in the relational case, each key path P_j^R must match the key path P_j^K of key K, $j = 1, ..., p$, i.e., although the key path expressions may be different they must have compatible data types. Following XMLSchema, the referencing node and referenced node must be within the same context node, thus the path expression C^R must be the same as C^K.[3]

Definition 2.2: An XML tree T is said to *satisfy* a keyref $R : (C, (T^R, \{P_1^R, \ldots, P_p^R\}))$ KEYREF K,if and only if $\forall c \in [\![C]\!]$, for $\forall t^R \in c[\![T^R]\!]$, $\exists t^K \in c[\![T^K]\!]$, $t^R[\![P_i^R]\!] = t^K[\![P_i^K]\!](i = 1, \ldots, p)$. ■

Adopting XPath notation for paths, let "/" denote the root or be used to concatenate two path expressions, "."

[2] We adopt this because its syntax is more concise than that of XMLSchema.

[3] To be precise, C^R should be equivalent to C^K when this can be efficiently decided.

denote the current context, "//" match a sequence of labels, and @ be an attribute name. All paths must end in either a single label or a disjunction of labels. The keys and keyrefs for our sample XML document can be written as:

$K_0 : (/, (.//city, \{./name, ./state\}))$
A city is identified by its name and state.

$K_1 : (//city, (.//restaurant, \{./name\}))$
Within the context of a city, each restaurant is uniquely identified by its name.

$K_2 : (//restaurant, (./appetizer|entree|salad|dessert, \{./name\}))$
Within a restaurant, an appetizer, entree, salad or dessert is identified by its name.

$R_0 : ((//city, (./reviews/review, \{./@restaurant\}))$ KEYREF K_1
Within a city, each review refers to a restaurant in that city by its name.

Before moving on to storage mapping, we comment on the restriction that the context node of the node being referenced must be the same node as the context node of the referencing node. Although this is done in XMLSchema to simplify the referencing scheme, it is not necessary. In [6], the authors defined a notion of "transitive keys", which guarantees that the context node of any key can itself be identified by some key value. In this case, the target node of any key can be identified by some path of key values from the root. Using this, the notion of foreign key can be generalized so that the context nodes may differ.

3 An Information Preserving Mapping

We now present an algorithm which, given the schema of an XML document, generates an information preserving relational schema. The XML schema consists of information about the structure of the document (i.e. the DTD) and XML key and keyref information as described in the previous section. The relational schema generated consists of a collection of relations together with their key and foreign key constraints.

3.1 Constraint preservation via constraint relations

At the heart of the schema-generation technique is a set of relations corresponding to the given XML key and keyref constraints. To capture the internal identifier for a node n in the document, we use the notation *nodeid(n)*. We use *text()* to grab the value of an attribute or simple element (text) node.

For each XML key $K : (C^K, (T^K, \{P_1^K, \ldots, P_p^K\}))$, we create a *key relation* $KR(tid, cid, P_1, \ldots, P_p)$, where for every $c \in [\![C^K]\!]$ and $t \in c[\![T^K]\!]$, a tuple (*nodeid(t), nodeid(c),* $t/P_1^K.text(), ..., t/P_p^K.text()$) is inserted into KR. We also assert the following functional dependency (key) in KR: $cid, P_1, \ldots, P_p \rightarrow tid$. Each target node is in a single context and hence occurs exactly once in KR. There are therefore two keys for KR, $(cid, P_1, \ldots, P_p)$ and (tid).

Similarly, for each XML keyref $R : (C^R, (T^R, \{P_1^R, \ldots, P_p^R\}))$ KEYREF K we create a *keyref relation* $RR : (tid, cid, P_1, \ldots, P_p)$, where for every $c \in [\![C^R]\!]$ and $t \in c[\![T^R]\!]$, a tuple (*nodeid(t), nodeid(c),* $t/P_1^K.text(), ..., t/P_p^K.text()$) is inserted into RR. We also assert the foreign key $(cid, P_1, \ldots, P_p)$ REFERENCES $KR(cid, P_1, \ldots, P_p)$. As with the key relation, (tid) is a key for RR. Both the key and key reference mappings rely crucially on the fact that each key path terminates in an attribute or simple element (text). We will refer to these key and keyref relations as **constraint relations**.

Note that since we insert a tuple in KR (RR) for every target node of every context node, the mapping from the XML document to these constraint relations is complete.

Proposition 3.1: An XML document satisfies the XML key $K : (C^K, (T^K, \{P_1^K, \ldots, P_p^K\}))$ if and only if the corresponding key relation $KR(tid, cid, P_1, \ldots, P_p)$ satisfies its key $(cid, P_1, \ldots, P_p)$.
Sketch of proof: By construction, there is a one to one correspondence between tuples in KR and matches for the context node, target node, and key path values of K in the XML document. Since the functional dependency (key) in KR is equivalent to the following assertion:
$\forall t_1, t_2 \in KR,$
$t_1.cid = t_2.cid \wedge t_1.P_1 = t_2.P_1 \wedge \ldots \wedge t_1.P_p = t_2.P_p$
a violation of the relational key implies a violation of the XML key and vice versa. ∎

Proposition 3.2: An XML document satisfies the XML keyref
$R : (C^R, (T^R, \{P_1^R, \ldots, P_p^R\}))$ KEYREF K
if and only if the corresponding foreign key relation $RR(tid, cid, P_1, \ldots, P_p)$ satisfies its foreign key $(cid, P_1, \ldots, P_p)$ REFERENCES $KR(cid, P_1, \ldots, P_p)$.
Sketch of proof: By construction, there is a one to one correspondence between tuples in RR and matches for the context node, target node, and key path values of R in the XML document. The foreign key constraint in RR is equivalent to the following assertion:
$\forall t_1 \in RR, \exists t_2 \in KR.\ t_1.cid = t_2.cid \wedge t_1.P_1 = t_2.P_1 \wedge \ldots \wedge t_1.P_p = t_2.P_p$
where KR is the key relation formed from the XML key K. Since the referring and referred nodes are by definition within the same context, $t_1.cid = t_2.cid$ is always true, and a violation of the relational foreign key constraint implies a violation of the XML key reference constraint and vice versa. ∎

It is therefore enough to check key and foreign key constraints in the relational instance to guarantee that the key and keyref constraints are satisfied in the source XML.

Returning to our example of the previous section, the constraint relations to be created are:

1. *city(cid, name, state)* for K_0, with (*name,state*) and (*cid*) as keys.

The attribute *cid* stores the identifier of a node tagged with *city*, and attributes *name* and *state* store the value of its *name* and *state* child elements, respectively. Since the context node of the key is the root of the document it is omitted in the relation.

2. *restaurant(rid,cid,name)* is created for K_1, with (*cid, name*) and (*rid*) as keys.

3. *menuitem(iid,rid, a|e|s|d.name, type)* is created for K_2, with keys (*rid, a|e|s|d.name*) and (*iid*). Attribute *type* can have value "appetizer", "entree", "salad" or "dessert".

4. *review(rvid, cid, @restaurant)* is created for R_1, with key (*rvid*) and foreign key (*cid, @restaurant*) REFERENCES *restaurant(cid,name)*.

Before we leave the issue of constraint preservation, it is important to consider the effect of updates to the source XML document. Following [12, 10], we assume two forms of updates, *insert(content)* and *delete(child)*.[4] We also assume that the path from the root to the update point is either specified or recoverable (e.g., using a parent relation).

Thus an insertion can be considered as an XML document with the same root as the original document in which the content to be inserted is marked as "new" and the existing content (i.e. the path from the root to the update point) is marked "old". Using the mapping to constraint relations described earlier, a tuple produced from a node marked "new" causes an insertion. Tuples produced from nodes marked "old" are ignored.

Assuming that the content of the subtree rooted at *child* is either specified or recoverable (for example, through a content and structure preserving mapping of the document), an analogous technique can be used for deletion: a tuple produced from a node marked "new" causes a deletion, and tuples produced from nodes marked "old" are ignored.

In this way, every insertion (deletion) can be mapped to a set of inserts (deletes) to the constraint relations via the mappings described earlier to populate the constraint relations. Note that since a single XML update may correspond to several tuples in the target, transactions must be used to prevent anomalies.

Let δ be the content to be inserted or the contents of the subtree to be deleted, "+" be the effect of inserting or deleting content, Σ be the XML constraints, Σ' be the key and foreign key constraints on the constraint relations, and M be the mapping which populates the constraint relations from the XML input.

Proposition 3.3: $I + \delta$ satisfies Σ if and only if $M(I) + M(\delta)$ satisfies Σ'.
Sketch of proof: According to Propositions 3.1 and 3.2, $I + \delta$ satisfies Σ if and only if $M(I + \delta)$ satisfies Σ'. By the definition of M, $M(I + \delta) = M(I) + M(\delta)$. ∎

[4] We do not consider the operators rename and replace since the authors acknowledge that they have problems on REF.

1. Create constraint relations. If any target path ends with a disjunction of tags, then add a new attribute in the relation to record the tag of the target node.
2. Create a DTD graph that represents the structure of a given XML-Schema.
3. For each XML key or key reference constraint, mark the edges at the end of the target path in the schema graph.
4. If a target path ends with a single tag (as opposed to a disjunction), then inline any attribute or non-empty text descendent connected by non-star path as well as the content of the target node (if it exists).
5. If any target path ends with a disjunction tag, inline the common attributes or non-empty text descendents connected by a non-star path as well as the content of the target node (if it exists).
6. If all incoming edges of a node are marked, the non-star edge connecting this node and the inlined node(including that on the key path) are marked. Do this recursively until no more edges need to be marked.
7. If there are unmarked non-star edges in the schema graph, find one whose source vertex connects with its parent by a star edge. We name this node a *master node*. Create a table for the master node and inline any descendant nodes connected by a non-star path. Mark all incoming edges of this master node.
8. Repeat the last two steps until there are no unmarked non-star edge in the schema graph.
9. Ensure that all parent-child relations are recorded by adding a parent id (and parent code) in each child relation to link with its parent.

Figure 2: The $X2R$ storage mapping algorithm

3.2 Content and Structure Preservation

Since only some of the nodes in the source are involved in a key or key reference relation, we need to consider how the rest of the document can be captured to preserve content, structure and semantic information. We can use arbitrary XML storage system together with the constraint relations to preserve them, but this approach brings redundancy in the generated relational schema. In the remainder of this section, we present an algorithm called $X2R$ (see Figure 2) which generates an information preserving mapping without redundancy.

The algorithm extends an inlining strategy [4] as follows: First, we create a DTD graph(as defined in [4]) as well as the constraint relations defined in the previous subsection. We then map the nodes not captured in the constraint mapping, either by inlining them into the constraint relations or by creating seperate relations. Finally, we add any missing parent-child information in the constructed relations.

The salient differences between the $X2R$ algorithm and hybrid-inlining and its variants [4, 5] are as follows: 1) We

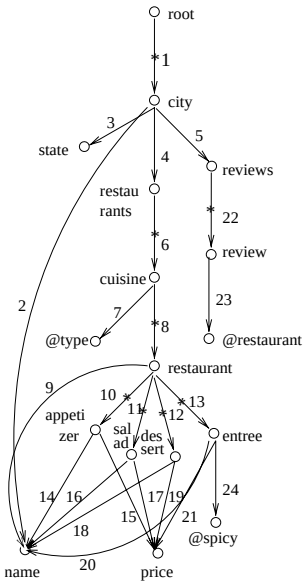

Figure 3: DTD graph

start from a set of key and reference relations which capture semantic information; 2) Relations that are separated in hybrid inlining may be coalesced by paths that end with a disjunction; and 3) The key and reference relations may inline ancestors other than the parent, i.e. the context node may be a non-parent ancestor.

Proposition 3.4: The $X2R$ mapping from the source XML schema X to R is information lossless.

Sketch of proof: Every node in the source XML document is mapped into some relation (content preservation), the structure information is captured either explicitly in *pid* attributes or via the DTD (inlining), and the semantics is captured in the constraint relations (proposition 3.1 and 3.2). ■

3.3 An example

We now illustrate how the $X2R$ mapping algorithm works on our example. The DTD graph assumed is presented in Figure 3 (see the Appendix for the XML schema specification) and the final relations created are as follows:

- *city(cid, name, state)* with keys (*name, state*) and (*cid*).
- *cuisine(csid, type, pid)* with key (*csid*) and foreign key (*pid*) REFERENCES *city(cid).*
- *restaurant(rid, cid, name, pid)* with keys (*cid, name*) and (*rid*), and foreign key (*pid*) REFERENCES *cuisine(csid).*
- *menuitem(iid,rid, a|e|s|d.name, price, type)* with keys (*rid, a|e|s|d.name*) and (*iid*), and foreign key (*rid*) REFERENCES *restaurant(rid).*
- *spicy(eid, spicy)* with key (*eid*) and foreign key (*eid*) REFERENCES *menuitem(iid).*
- *review(rvid, cid, @restaurant, TEXT, pid)* with key (*rvid*), and foreign keys (*cid, @restaurant*) REFERENCES *restaurant(cid, name)* and (*pid*) REFERENCES *city(cid).*

In step 1, we build constraint relations *city(cid, name, state), restaurant(rid, cid, name), menuitem(iid,rid, a|e|s|d.name,type),* and *review(rvid, cid, @restaurant).* In step 4 we inline the *TEXT* information into the *review* relation, and in step 5 we inline the *price* information into table *menuitem.* Relation *cuisine (csid,type, pid)* and relation *spicy(eid, spicy)* are created in step 7. All the *pid* information is inlined in step 9. Note that in some relations (e.g. table *menuitem*) the parent node coincides with the context node. Details of this construction are deferred to the Appendix.

4 Related works and conclusions

This paper proposes a novel XML storage strategy using relational databases which preserves the content, structure and semantic information as expressed in key and foreign key constraints. In contrast, other strategies that have been proposed [1, 2, 3, 4, 5] only preserve the content and structure of XML documents. Our storage mapping is guided by the XML key and keyref constraints, and is based on the notion of constraint relations. Although the storage strategy in this paper is integrated with hybrid-inlining, constraint relations can be combined with virtually any other strategy. A direct benefit of our storage mapping is the ability to efficiently check XML constraints using relational key and foreign key constraints; furthermore, the technique is incremental. In some ways, our approach is analogous to the initial stage of database design based on functional dependencies. In future work, we plan to consider how the conceptual schema design can be refined according to the query workload.

LegoDB [5] considers the issue of finding an optimal relational schema in a space of storage mappings according to the query workload for some cost model. The procedure is analogous to the tuning procedure for physical database design in the relational database[13]. Specifically, it tunes the conceptual schema according to the query and update workload to achieve better performance. In this paper we produce a relational mapping which is optimized for updates(enforcing the constraints efficiently), but do not consider queries. Considering queries would obviously affect the design: For example, if finding the price and spiciness of an entree is a frequent query we may wish to extend the schema of *menuitem* to include attribute *spicy*; the attribute is null for appetizers, salads and dessert.

We classify nodes by considering both their tag and their keys (if any), while [1, 2, 3, 4, 5] only consider the former. For example, in our sample XML file, we can define a parent type *menuitem* for types *appetizer,entree,salad* and *dessert* based on keys and then store type *menuitem* into one table. In other words, we use the semantic information in keys to guide the schema design.

We do not address the issue of mapping queries on the XML document to the relational representation in this paper. Several general mapping algorithms from XML query languages to SQL [14, 15, 16] have been proposed and could be used for the X2R mapping.

Another related issue not addressed in this paper is how to map constraints into a relational schema that has already been created. For example, in the Clio system [17] the target schema is fixed. In general, it is impossible for a fixed target to preserve all the information in the source, especially the structure and constraints information. [18] begins to address the question of mapping constraints to a fixed target schema by giving an algorithm to answer the following question: Given a set of XML keys Σ and a mapping to a fixed relational schema $\mathcal{R}$, is a functional dependency f on $\mathcal{R}$ provable from Σ? [19, 20] focuses on the more general question of how to use the constraints in the target to optimize constraint checking defined in source.

There are several native XML Schema validators[21, 10], to our knowledge no other work has been done on automatically enforcing XML constraints using relational database technologies.

Acknowledgments. We are grateful to Val Tannen, Wang-Chiew Tan, Byron Choi, Wenfei Fan and Carmem Hara for helpful discussions.

References

[1] A. Deutsch, M. Fernandez, and D. Suciu. Storing semistructured data with STORED. In *In Proceedings of the Workshop on Query Processing for Semistructured Data and Non-Standard Data Formats*, pages 431–442, 1999.

[2] D. Florescu and D. Kossmann. Storing and querying XML data using an RDBMS. In *Bulletin of the Technical Committee on Data Engineering*, pages 27–34, September 1999.

[3] A. Schmidt, M. Kersten, M. Windhouwer, and F. Waas. Efficient relational storage and retrieval of XML documents. In *WebDB*, pages 47–52, 2000.

[4] 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 *The VLDB Journal*, pages 302–314, 1999.

[5] P. Bohannon, J. Freire, P. Roy, and J. Simeon. From XML-Schema to Relations: A Cost-Based Approach to XML Storage. In *ICDE*, 2002.

[6] P. Buneman, S. Davidson, W. Fan, C. Hara, and W. Tan. Keys for XML. In *WWW10*, pages 201–210, 2001.

[7] H. Thompson, D. Beech, M. Maloney, and N. Mendelsohn. XML Schema Part 0: Primer , May 2001. http://www.w3.org/TR/xmlschema-0/.

[8] MAGE-ML. http://www.mged.org/Workgroups/MAGE/mage-ml.html.

[9] G. Sherlock and et al. T. Hernandez-Boussard. The Stanford Microarray Database. In *Nucleic Acids Research, 29(1):152–155, 2001.*, 2001. http://daisy.stanford.edu/MicroArray/SMD.

[10] Y.Chen, S. Davidson, and Y. Zheng. Validating Constraints in XML. 2002.

[11] R. J. Miller, Y. E. Ioannidis, and R Ramakrishnan. The use of information capacity in schema integration and translation. In *Proc. 19th International VLDB Conference*, pages 120–133, August 1993.

[12] A. Halevy I. Tatarinov, Z. Ives and D. Weld. Updating XML. In *Proceedings of ACM SIGMOD Conference on Management of Data*, 2001.

[13] J. Gehrke R. Ramakrishnan. *Database Management Systems*. McGraw-Hill, 2000.

[14] J. Shanmugasundaram, E. J. Shekita, J. Kiernan, R. Krishnamurthy, S. Viglas, J. F. Naughton, and I. Tatarinov. A general techniques for querying XML documents using a relational database system. *SIGMOD Record*, 30(3):20–26, 2001.

[15] M. F. Fernandez, W. C. Tan, and D. Suciu. Silkroute: trading between relations and XML. *WWW9 / Computer Networks*, 33(1-6):723–745, 2000.

[16] M. J. Carey, J. Kiernan, J. Shanmugasundaram, E. J. Shekita, and S. N. Subramanian. XPERANTO: Middleware for publishing object-relational data as XML documents. In *The VLDB Journal*, pages 646–648, 2000.

[17] R. J. Miller, M. A. Hernandez, L. M. Haas, L. Yan, C. T. H. Ho, R. Fagin, and L. Popa. The Clio project: Managing heterogeneity. *SIGMOD Record*, 30(1):78–83, 2001.

[18] S.B. Davidson C. Hara W. Fan. Propagating XML keys to relations. Technical Report MS-CIS-01-31, University of Pennsylvania, Computer and Information Science Department, 2001.

[19] L. Popa and V. Tannen. An Equational Chase for Path-Conjunctive Queries, Constraints, and Views. In *International Conference on Database Theory (ICDT)*, Jerusalem, Israel, January 1999.

[20] L. Popa and A. Deutsch and A. Sahuguet and V. Tannen. A Chase Too Far? In *Proceedings of ACM SIGMOD International Conference on Management of Data*, Dallas, USA, May 2000.

[21] Microsoft XML Parser 4.0(MSXML). Available at: http://msdn.microsoft.com.