From XML to Relational Database

Yan Men-hin and Ada Wai-chee Fu
Department of Computer Science and Engineering
The Chinese University of Hong Kong
mhyan, adafu @cse.cuhk.edu.hk

1 Introduction

The Extensible Markup Language (XML) is a relatively new standard for structured documents and data on the Web [4]. It is expected that most XML documents will be accompanied by Document Type Definitions (DTDs) [4, 2]. DTD is essentially a grammar for restricting the tags and structure of a document.

Though an XML document can exist on its own, there is much interest in storing XML data as relational database, because it allows us to apply well-developed relational techniques on XML data. Database companies are working on how XML data can fit into their relational systems. However, most of the known approaches are manual. The problem is how this can be done automatically. One major issue in this is how to produce the relational schemas from the XML data, and this is the problem we tackle here. A most related work can be found in [13].

XML is a document markup language permitting tagged text (elements), element nesting, and element reference. An example of an XML representation of catalog information for a book is shown in Figure 1. Applications that operate on XML data often need additional guarantees on the structure and content of such data. Such constraints on document structure can be expressed using a `Document Type Definition (DTD)`. A DTD defines a class of XML documents using a language that is essentially a context-free grammar with several restrictions. Using the book example

```
<book>
  <title>Fables of the Green Forest</title>
  <author>
    <firstname>Henry G.</firstname>
    <lastname>George<lastname>
  </author>
  <author>
    <firstname>Hafner</firstname>
    <lastname>Pacman</lastname>
  </author>
  <price currency = "HKD">149.9</price>
  <bestseller authority="Times"/>
</book>
```

Figure 1: An XML representation example

```
<!ELEMENT book (title, author+,
                price, bestseller?)>
<!ELEMENT title (#PCDATA)>
<!ELEMENT author (#PCDATA|lastname|
                firstname|fullname)*>
<!ELEMENT price (#PCDATA)>
<!ATTLIST price currency CDATA "USD"
               source (list—regular—sale) list
               taxed CDATA #FIXED "yes">
<!ELEMENT bestseller EMPTY>
<!ATTLIST bestseller authority
                CDATA #REQUIRED>
```

Figure 2: A DTD example

in Figure 1, one may use the DTD declaration in Figure 2 to constrain the XML documents. For more details about the DTD, please refer to [4, 5].

2 Global Schema Extraction Algorithm

We propose two algorithms to create relational schemas from the XML data and the DTD that those XML data conform to. Although the algorithms we propose have different details, the global scheme is the same: (1) simplify DTD, (2) construct schema prototype trees, (3) generate relational schema prototypes, (4) detect functional dependencies and candidate keys, and (5) normalize the relational schema prototypes.

The first algorithm we propose is called the Global Schema Extraction algorithm.

Step 1: Simplify DTD

First, we need to simplify the DTD for the set of XML documents. DTD can be highly complex just like their counterpart for semistructured data [1]. It is possible to simplify the DTD without affecting the way we extract the relational schemas. To simplify the DTDs, we need to get rid of entity declarations first. Entity type declarations, which are used to abbreviate DTD components, are removed, and all the declarations referring to them are replaced with the DTD components they are representing. Then we apply transformations to the DTD. Part of our proposed transformations is similar to those presented in [13] and [6]. However, other than flattening the nested representation of DTDs as proposed by [13] and [6], our transformations also eliminate some operators in DTDs. Every element type declarations can be converted to the required form by performing the transformations shown below repeatedly (here `p`, `p'`, ...denote subelements within a given element type declaration).

```
p*  ──► p        p|p'      ──► p, p'
p+  ──► p        (p, p')   ──► p, p'
p?  ──► p        ..., p,..., p,...  ──► p
```

We only preserve the information that is useful in constructing schema prototype trees later. For instance, inside any *attribute type declaration*, the value types (e.g. `#IMPLIED, #FIXED`...etc) for the *character data (CDATA)* are removed from the DTD. Figure 4 shows the example of converting a DTD in Figure 3, which is a modification of [12], into a simplified DTD.

Step 2: Construct Schema Prototype Trees

With the simplified DTD, we then construct the *schema prototype trees* which represent the structure of the simplified DTD. The nodes can be elements or attributes specified in the DTD. Schema prototype trees will be used for generating schema prototypes in the next step (Step 3). They are constructed as follows.

First, we have to determine the root(s) of the trees from the DTD. There are several rules we have to follow when deciding the root:

Rule 1: *Only an element can become a root*

In XML, attributes cannot exist without following it's corresponding element. As a result, all attributes declared in the DTD can only be leaf nodes in the schema

```
<!ENTITY %txt "(#PCDATA)">
<!ELEMENT book(booktitle,price?,
                    author,authority*)>
<!ELEMENT authority (authname, country)>
<!ELEMENT authname %txt>
<!ELEMENT country %txt>
<!ELEMENT booktitle %txt>
<!ELEMENT price %txt>
<!ELEMENT monograph (title, author, editor)>
<!ELEMENT editor (monograph+)>
<!ATTLIST editor name CDATA #REQUIRED>
<!ELEMENT author (name, address)>
<!ATTLIST author id ID>
<!ELEMENT name (firstname, lastname)>
<!ELEMENT firstname %txt>
<!ELEMENT lastname %txt>
<!ELEMENT address %txt>
```

Figure 3: An DTD before simplification

```
<!ELEMENT book(booktitle,price,
                    author,authority) >
<!ELEMENT authority (authname, country)>
<!ELEMENT authname (#PCDATA)>
<!ELEMENT country (#PCDATA)>
<!ELEMENT booktitle (#PCDATA)>
<!ELEMENT price (#PCDATA) >
<!ELEMENT monograph (title, author, editor)>
<!ELEMENT editor (monograph)>
<!ATTLIST editor name CDATA >
<!ELEMENT author (name, address)>
<!ATTLIST author id ID >
<!ELEMENT name (firstname, lastname)>
<!ELEMENT firstname (#PCDATA)>
<!ELEMENT lastname (#PCDATA)>
<!ELEMENT address (#PCDATA)>
```

Figure 4: A simplified DTD

prototype trees. We can thus consider only elements when deciding the roots.
Rule 2: *For an element which does not appear in any other element declaration in the DTD, it becomes the root for a schema prototype tree*
Rule 3: *If there is no element in the DTD satisfying **Rule 2**, one of the elements is selected as the root*

When all elements in the DTD are the subelement of some other elements, we can be sure that recursion occurs in the DTD. Thus we have to arbitrarily break the loop in order to construct the schema prototype tree.

For all selected roots in the DTD, their schema prototype trees are constructed as follows:

Starting from the subelement(s) of the root, we try to scan the DTD in a depth-first style. When we first visit a subelement e' of an element e in the tree, where e' has not appeared in the schema prototype tree, we create a new node for e' as a child node of e. Apart from subelements, we need to take care of possible parsed character data (#PCDATA) and the attribute declarations for an element we are visiting. Any attributes declared for an element in the DTD is treated the same way as a subelement of the element. It is easy to see that the leaf nodes of the schema prototype tree are either elements declared as containing #PCDATA only, or attributes for their parent elements. If an element is declared as containing #PCDATA together with other subelement, i.e. it is a mixed element, we would mark the corresponding node with a "#" in the schema prototype tree. The marking would be useful in the following step.

We also need to handle the possible situation where recursion occurs while constructing the schema prototype tree. Consider a case when we visit an element which has already had a corresponding node X created in the schema prototype tree, we would create a leaf node with label $X.A$ which indicates a foreign key to its ancestor. The key can be discovered or arbitrarily assigned in Step 4 later. Then we would stop traveling down the subelement of that element to prevent an infinite recursion. Consider when the tree construction has come to the element declaration `<!ELEMENT`

`editor (monograph)>` where element `monograph` has already appeared in the tree. We would create a new node `monograph.A`. An edge pointing from `editor` to it is created as well. The example schema prototypes tree for the DTD would look like the one in Figure 5.

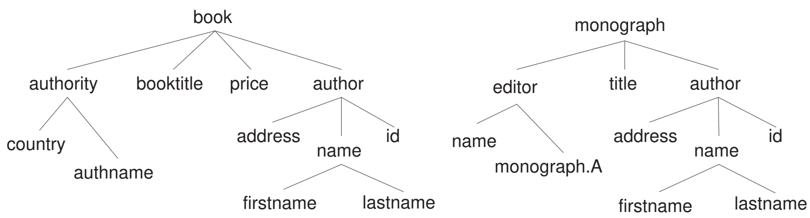

Figure 5: The relational schema prototype trees constructed from the example DTD

Step 3: Generate Relational Schema Prototype

Given a schema prototype tree, the corresponding relational schema prototype is generated as follows. The basic idea is to regard all the necessary attributes and elements in the simplified DTD as the "attributes" in an ER-Model. The schema prototype is thus generated by inlining all the necessary descendants of the schema prototype tree starting from the root. The necessary descendants refer to all the leaf nodes in the schema prototype tree, and the nodes marked with a "#". The reason for doing this is because not all the elements in an XML document contain real data. If an element is not declared to contain any #PCDATA in DTD then we can be sure that for any XML document conforming to that DTD, there is no parsed character data between any pairs of that element tag. As a result, we do not have to provide a field for that element in the relational schema prototype.

The relational schema prototype generated from the schema prototype tree presented in Figure 5 is shown in Figure 6. In order to uniquely specify the name for each attribute in the relational prototype schema, all attributes fields are named by the path from the root node of the tree.

table:book (	*table:monograph* (
book.booktitle, (A)	monograph.title, (A)
book.price, (B)	monograph.author.id, (B)
book.author.id, (C)	monograph.author.name.firstname, (C)
book.author.name.firstname, (D)	monograph.author.name.lastname, (D)
book.author.name.lastname, (E)	monograph.author.address, (E)
book.author.address, (F)	monograph.editor.name, (F)
book.authority.authname, (G)	monograph.editor.monograph.A (G)
book.authority.country (H)	)
)	

Figure 6: The relational schema prototypes generated from the tree in Figure 5

Step 4: Discover Functional Dependencies and Candidate Keys

With the generated schema prototypes, we can now apply traditional techniques of relational database design to produce the suitable relational schemas for the XML data. In order to reduce the data redundancy and inconsistency in the set of relational schemas for the XML data, we try to discover functional dependencies (FDs) and the candidate keys by analyzing the XML data. Then we can normalize the relational schema prototype.

Much work has been done on discovering FDs from relations in the past years. Recently a new algorithm was proposed [9], which has improved the efficiency of dependency inference by several orders of magnitude over the previous works. We have used this method in our prototype. Note that the ".A" attribute is not considered in this discovery process.

Let us assume the result of finding minimal sets of FDs and candidate keys for the tables in Figure 6 are as below:

`table:book` - FD(s): $A \rightarrow BC$, $DEF \rightarrow C$ and $C \rightarrow DEF$ Key(s): $\{AGH\}$

`table:monograph` - FD(s): $A \rightarrow BF$, $B \rightarrow CDE$, and $CDE \rightarrow B$ Key(s): $\{A\}$

Since `monograph.title` is identified as the key, we can assign the ".A" attribute as `monograph.editor.monograph.title`, a foreign key pointing to `monograph.title`. If we cannot find suitable keys (e.g. they are too lengthy), we would assign an artificial ID to the relation and the ".A" attribute would point to that ID.

Step 5: Normalize the Relational Schema Prototypes

With the FDs and candidate keys, we can normalize the relational schema prototype to a set of new relations. In our example, we propose to use 3NF decomposition [3]. The data in the XML document can then be stored to the RDBMS according to the schema shown below. Note that since `table:book-3` and `table:monograph-3` are the same after comparing the attributes in them, they can be merged as one.

```
table:book-1 (                         table:monograph-1 (
book.booktitle, (A)                    monograph.title, (A)
book.price, (B)                        monograph.author.id, (B)
book.author.id (C)                     monograph.editor.name (F)
)                                      )
table:book-2 (                         table:monograph- 2(
book.booktitle, (A)                    monograph.editor.name, (F)
book.authority.authname, (G)           monograph.editor.monograph.title (G)
book.authority.country(H)              )
)                                      table:monograph-3 (
table:book-3 (                         monograph.author.id, (B)
book.author.id, (C)                    monograph.author.name.firstname, (C)
book.author.name.firstname, (D)        monograph.author.name.lastname, (D)
book.author.name.lastname, (E)         monograph.author.address (E)
book.author.address (F)                )
)
```

We call the above algorithm a *global* algorithm because we try to form relational schema prototypes which include as many elements in the DTD as possible, then we extract all the necessary information from the raw data in order to decompose the schema prototypes into the suitable relational schemas. As a result, the step of FD inference together with the characteristic of the actual XML data play a heavy role in this algorithm. Unlike the proposed schemas extraction algorithm by [7, 8], we do not have to introduce extra data fields.

However, one potential problem in the above proposed algorithm is that the cost of discovering FDs can be high since the number of possible minimal dependencies is exponential in the number of attributes [10, 11]. As a result, it might be better if we can reduce the size, i.e. the number of attributes, of the schema prototypes before the step of finding FDs. This will be the target of the second approach we present next.

3 DTD-splitting Schema Extraction Algorithm

In our second algorithm we try to predict some characteristics of the XML data from the DTD, and hence perform a certain level of schema decomposition (DTD split) before the step for finding FDs and keys.

Step 1: Simplify DTD

We have to simplify the possibly complicated structures of the DTD. In this algorithm we preserve some of the operators so as to preserve some subelement occurrence information. Every element type declarations can be converted to the required form by performing the transformations shown below. This is quite similar to what is used in [13]. For example, the DTD in Figure 3 will be simplified as the one in Figure 7.

```
p+ ──► p*        p|p'      ──► p, p'
p? ──► p         (p, p')   ──► p, p'
                 (p, p')*  ──► p*, p'*

   ..., p,..., p*,...  ──► p*
   ..., p,..., p,...   ──► p*
```

Step 2: Construct Schema Prototype Trees

With the simplified DTD, we then construct the *schema prototype trees* which represent the structure of the simplified DTD. The rules for determine the roots and the tree construction sequence are:

Root Determination

Rules 1 and 2 are similar to those stated in Section 2.

Rule 3: *For a non-#PCDATA element which appears in more than one other element declarations, it becomes a root for a schema prototype tree*

Let us assume that an element `C` is the subelement of both element `A` and `B` in the DTD. We would make element `C` a root for a schema prototype tree and the schema tree constructed from it would become a separate schema later in the following step. We use traditional relational database design theory to explain why we separate `C`. The relationship among `A, B, C` indicates some tendency for multiple elements of `A` and `B` to map to element of `C`. For example, both a_1 in `A` and b_1 in `B` are mapped to c_1 in `C`. Taking `C` as a root can reduce the redundancy of repeating the attributes of c_1 with both a_1 and b_1. If at most one element of `C` can be mapped to an element of either `A` or `B`, we have a many-to-one (*M:1*) mapping from **A,B** to **C**. This will be achieved later in Step 5. As a result, we would make element `C` a root for a schema prototype tree. The schema tree constructed from it would become a separate schema later in the following step.

For the case of many-to-many relationship ($M{:}N$), we can also decompose the relations into a relation containing A, a relation containing B, a relation containing C, a relation containing $K_{\mathtt{A}} \cup K_{\mathtt{C}}$ and a relation containing $K_{\mathtt{B}} \cup K_{\mathtt{C}}$, where $K_{\mathtt{A}}$, $K_{\mathtt{B}}$, $K_{\mathtt{C}}$ are the keys of A, B, C respectively. As a result, C can be separated as a root for another schema prototype tree.

Rule 4: *For an non-#PCDATA element* B *which ONLY appears in another non-root element declaration* A *in the DTD with a "*", it becomes the root for a schema prototype tree if it is NOT the only subelement of* A

We again use traditional relational database theory to explain why we separate elements with a "*" into another schema prototype tree: For an element declaration <!ELEMENT A (B*) > inside the DTD, the "*" has some indication of the tendency of a *1:M* relationship from A to B. For such *1:M* relationship, each value of $K_{\mathtt{B}}$ is associated with at most one value of $K_{\mathtt{A}}$. It nearly directly come to the idea that $K_{\mathtt{B}}$ should functionally determine $K_{\mathtt{A}}$. Since the FD $K_{\mathtt{B}} \rightarrow K_{\mathtt{A}}$ holds, B can be separated from A. In terms of relational database theory, we can decompose them into two relations: A and $\mathtt{B} \cup K_{\mathtt{A}}$.

For the case *M:N*, it is evident that we can always decompose the relation into a relation containing A, a relation containing B and a relation containing $K_{\mathtt{A}} \cup K_{\mathtt{B}}$. As a result, we are sure that B can be separated as a root for another schema prototype tree.

If B only appears in A and is the only subelement of A, it is not necessary to separate B into another relation.

Rule 3 and **rule 4** are similar to some suggestions in [13]. Here we explain the reasons to set such rules in our algorithm based on relational database design concepts. The desirable schemas depending on the *1:M, M:1, M:N* relationships will be discovered in the later steps.

Rule 5: *If recursion occurs in the DTD, one of the elements in the recursion is selected as the root*

Tree Construction

We propose a new method to construct the trees. Generally, the tree construction method is more or less the same as the one in Section 2. However, during the scan, we do not travel down any element which is determined as a root. For different kinds of roots which are determined by different rules above, their trees may be constructed in a slightly different way.

For roots determined by **rule 2, 3** or **5**, their tree construction processes are the same as the one in Section 2. However, during the tree construction if we visit an element declaration of a *root* element which is determined by **rule 3**, we would create a new node for that newly visited root element. This is because we expect a tendency of *M:1* relationship from the parent of the element to the root element. For example probably many books may be written by the same author, so it is likely to include the key of author as part of the relation for the book.

On the other hand, if we visit an element declaration of a *root* element which is determined by **rule 4**, we will not perform any node addition to the schema prototype tree. For roots determined by **rule 4**, we have to find out their only ancestor in the DTD, and add the corresponding nodes as the leaf nodes of the roots in the schema prototype trees. This is because we expect a tendency of *1:M* relationship from the parent of the element to the root element. For example we expect one book will likely be related to multiple authorities. Therefore it is likely to include a key of book as an attribute in the relation for authority.

For the case of recursion, if we revisit the element declaration of the root, we will find out the direct ancestor of the root inside the looping, and add the corresponding node as a leaf node of the root. We then stop traversing down to prevent infinite looping. The trees constructed from the DTD in Figure 7 is shown in Figure 8. Note that leaf nodes with bold names and ".A"s are the roots of other trees (ancestors). Bold names indicate keys to other relations (descendants) while ".A"s indicate foreign keys to other relations (ancestors). Trees will form relations and the keys and foreign keys of relations can be discovered or arbitrarily assigned in Step 4 later.

```
<!ELEMENT book(booktitle,price,
                    author,authority*) >
<!ELEMENT authority (authname, country) >
<!ELEMENT authname (#PCDATA) >
<!ELEMENT country (#PCDATA) >
<!ELEMENT booktitle (#PCDATA) >
<!ELEMENT price (#PCDATA) >
<!ELEMENT monograph (title, author, editor)>
<!ELEMENT editor (monograph*) >
<!ATTLIST editor name CDATA >
<!ELEMENT author (name, address) >
<!ATTLIST author id ID >
<!ELEMENT name (firstname, lastname)>
<!ELEMENT firstname (#PCDATA)>
<!ELEMENT lastname (#PCDATA)>
<!ELEMENT address (#PCDATA)>
```

Figure 7: An simplified DTD

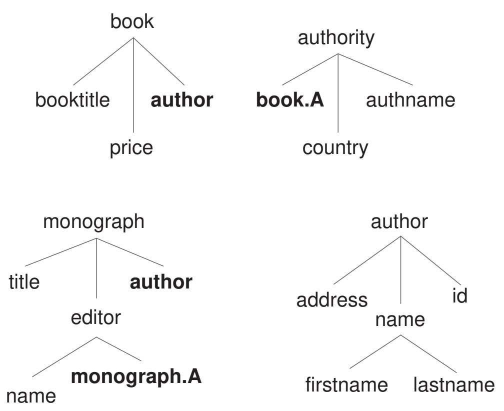

Figure 8: Schema prototype trees constructed from Figure 7

Up to this step our method is similar to [13]. However, [13] just joins all relations with foreign keys (by a parent.id) while our method, which is based on traditional database theory, efficiently uses both keys and foreign keys to join the relations. As a result, our method can produce resulting tables with less data redundancies. For example, given the above DTD, the shared inlining method in [13] will create an author relation with an parentID attribute pointing to books and monographs. Therefore the same author who has written k books and monographs will be repeated k times in the table, each time pointing to a different book or monograph. Similar redundancies occur in the hybrid inlining method. Note that even without the discovery of functional dependency, with only Steps 1 to 3, we can produce some pretty good design.

Step 3: Generate Relational Schema Prototype

Just as in Section 2, we generate schema prototype by inlining all the necessary descendants of the schema prototype tree, including leaf nodes and the node marked with a "#", starting from the root. However, we will not inline those key nodes or foreign key nodes in this step. We will decide how to add them (using found candidate keys or assigning a key attribute) into the resulting relational schema after we discover all the FDs and keys in Step 4. The relational schema prototypes generated from the schema prototype trees presented in the previous step are shown in the following.

```
table:book(booktitle,price)
table:authority(country,authname)
table:author(address,id,firstname,lastname)
table:monograph(title,name)
```

Note that we can be sure that no two nodes inside the same tree will have the same name. So we do not have to use the naming scheme used in Section 2.

Step 4: Discover Functional Dependencies and Candidate Keys

With the generated schema prototypes, we follow the process in Section 2's Step 4. In Step 2 and Step 3, we have actually *pre-decomposed* the DTD into smaller schema prototypes. As a result, the cost of inferring FDs and candidate keys would be much smaller. As mentioned in Step 3, we determine the candidate keys for the schema prototypes in this step so as to refine the schema prototypes. However, if a candidate key turns out to contain many attributes or is very lengthy, then we may also assign a new artificial ID field to serve as the key, unique ID's will be generated by the system for such a key. This method of an artificial ID is heavily used in other methods where functional dependencies are not utilized. This technique can also be used in the algorithm in Section 2.

Let us assume that the maximum number of attributes allowed for a key is 1, and all the candidate keys found for each schema prototype are listed as below:

```
table:book - {booktitle}
table:authority - {country, authname}
table:monograph - {title}
table:author - {id},{lastname, address}
```

According to the above procedure, we use `booktitle` as the key for `table:book`. We assign an `assignID` field to `table:authority`. `title` is used as `table:monograph`'s key while `id` is chosen as the key for `table:author`. All the keys or foreign keys to other relations are added in the format `table_name.table_key`. In our example, the relational schema prototypes generated from the trees in Figure 8 is:

table:book(*booktitle*, price, author.id)
table:authority(country, authname, *assignID*, book.booktitle)
table:author(address, *id*, firstname, lastname)
table:monograph(*title*, name, author.id, monograph.title)

Step 5: Normalize the Relational Schema Prototypes

With the FDs, candidate keys and the set of refined schema prototypes, we can simply normalize the relational schema prototype to a set of new relations, if the refined schema prototypes can be further decomposed. After normalization, we can then produce the resulting relational schemas for the XML and use them to map the XML data into relational database.

We have built a prototype and shown that the proposed mechanisms work quite well for real XML datasets. When the number of element and attribute declarations in the DTD is not too large, *DTD-splitting* algorithm can produce efficient resulting schema even without finding FDs and keys. When compared with previous methods, our methods can produce tables with less data redundancies.

Acknowledgments We thank the authors of [9] who have kindly provided us the source code of TANE for discovering functional dependencies. This research is supported by the RGC Earmarked Research Grant CUHK 4436/99E.

References

[1] C. Beeri and T. Milo. Schemas for integration and translation of structured and semi-structured data. In *Proc. of the Int. Conf. on Database Theory*, 1999.

[2] Jon Bosak and Tim Bray. Xml and the second-generation web. In *Scientific American*, May 1999.

[3] K. Brathwaite. Relational theory: concepts and application. In *Academic Press*, San Diego, 1991.

[4] T. Bray, J. Paoli, and C. Sperberg-Mcqueen. W3C Recommendation: Extensible Markup Language (XML) 1.0. http://www.w3c.org/TR/xml, February 1998.

[5] D. Connolly. Extensible Markup Language (XML). http://www.w3c.org/XML.

[6] A. Deuntsch, M Fernandez, and D. Suciu. Storing semistructured data with stored. In *Proceedings of ACM SIGMOD*.

[7] D. Florescu and D. Kossmann. A performance evaluation of alternative mapping schemes for storing xml data in a relational database. Technical report, INRIA, May 1999.

[8] D. Florescu and D. Kossmann. Storing and querying xml data using an rdbms. In *Data Engineering Magazine of Bulletin of the IEEE Computer Society Technical Committee on Data Engineering*, pages 27–34, September 1999.

[9] Ykä Huhtala, Juha Käkkäinen, Pasi Porkka, and Hannu Toivonen. Efficient discovery of functional and approximate dependencies using partitions. In *Proceedings of 14th International Conference on Data Engineering (ICDE'98)*, pages 392–401, Cambridge, MA, 1998.

[10] H. Mannila and K. Räihä. *The Design of Relational Database*. Addison-Wesley, Menlo Park, CA, 1992.

[11] H. Mannila and K. Räihä. On the complexity of inferring functional dependencies. In *Discrete Applied Mathematics*, volume 40, pages 237–243, 1992.

[12] P. Merialdo. ACM SIGMOD Record: XML Version. http://www.acm.org/ sigmod/record/xml/, December 1999.

[13] J. Shanmugasundaram, Tufte K, G. He, C. Zhang, D. DeWitt, and J. Naughton. Relational database for querying xml documents: limitations and opportunities. In *Proceedins of the 25th VLDB Conference*, Edinburgh, Scotland, 1999.