Designing Functional Dependencies for XML

Mong Li LEE[1], Tok Wang LING[1], and Wai Lup LOW[2*]

[1] School of Computing, National University of Singapore
{leeml,lingtw}@comp.nus.edu.sg
[2] DSO National Laboratories, Singapore
lwailup@dso.org.sg

Abstract. Functional dependencies are an integral part of database theory and they form the basis for normalizing relational tables up to BCNF. With the increasing relevance of the data-centric aspects of XML, it is pertinent to study functional dependencies in the context of XML, which will form the basis for further studies into XML keys and normalization. In this work, we investigate the design of functional dependencies in XML databases. We propose FD_{XML}, a notation and DTD for representing functional dependencies in XML. We observe that many databases are hierarchical in nature and the corresponding nested XML data[1] may inevitably contain redundancy. We develop a model based on FD_{XML} to estimate the amount of data replication in XML data. We show how functional dependencies in XML can be verified with a single pass through the XML data, and present supporting experimental results. A platform-independent framework is also drawn up to demonstrate how the techniques proposed in this work can enrich the semantics of XML.

1 Introduction

Functional dependencies [Cod70] are an integral part of database theory, and they have been well studied for the past thirty years in the context of the relational data model. The concept of a *key* in databases is derived from functional dependencies. They also form the basis for the normalization process up to Boyce-Codd Normal Form (BCNF) [Cod72]. Studies on functional dependencies have also resulted in inference rules [Arm74], semantic data models [Wed92] and dependency-preserving decomposition techniques [Ber76,TF82].

The emergence of XML [BPSMM00] as a standard for data representation on the Web is fueling semistructured data models as strong contenders to the traditional relational data model. Although an XML document is not a database in the strictest sense, there is much work done in the research community to move XML towards a real data model (e.g., Lore [MAG+97], XML Schema [Fal00], XML data query model [FR01] etc.). These works deal with query languages,

* This work was done while the author was on a research scholarship from the National University of Singapore.

[1] For this paper, *XML data* refers to data represented in XML. It is not to be confused with the W3C Note XML-Data.

structural constraints, path consistency, and issues concerning the modeling of key and foreign key relationships. While it is equally important to see how the concept of functional dependencies can be applied to XML databases, there has been little work on functional dependencies representation and modeling in semistructured data models.

Figure 1 shows a typical Project-Supplier-Part semistructured database together with its DTD. In the instance diagram shown in Figure 1b, an entity (in the Entity-Relationship terminology) is denoted by a rectangle. Dark circles represent the identifier attributes of the entity, and hollow circles represent properties. The '@' symbols preceding the name of labeled edges indicate that the data is modeled as XML attributes. Unlabeled edges indicate that the data is modeled as XML elements. Entity types `Project`, `Supplier` and `Part` have the keys `PName`, `SName` and `PartNo` respectively. Suppose we have a constraint that a supplier must supply a part at the same price regardless of projects. Unfortunately, this information **cannot** be deduced from the DTD nor the database instance. This information is useful to anyone using this XML data as it can alert them to violations of this integrity constraint. For instance, knowledge of this constraint can help users to identify semantically incorrect transformations (e.g. transformations that result in a supplier supplying a part at different prices to different projects). How can we embed this knowledge into the XML data? Is this information best treated as meta-data or should it be embedded into the data itself? How should it be represented? And how can we efficiently check if the constraints are violated? We need a way to express functional dependency constraints in XML databases. These constraints will be part of the XML data interchange which governs the semantics of the data.

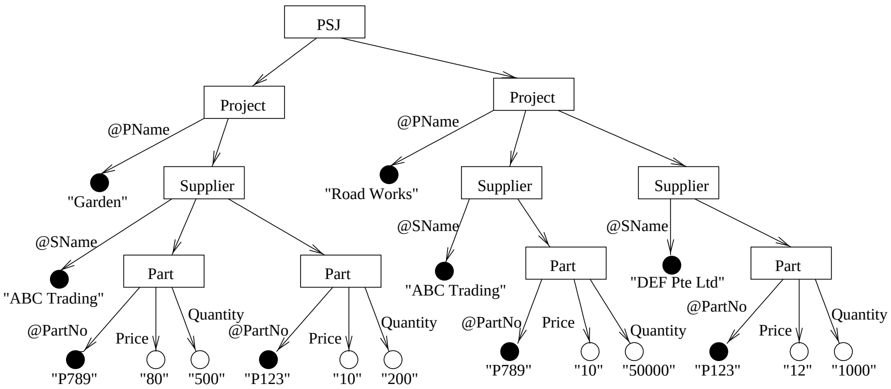
The contributions of this work are :

1. We propose FD_{XML}, a notation and DTD, for representing functional dependencies in XML. This notation takes into consideration the hierarchical nature of XML databases.
2. We formulate a replication cost model based on XML functional dependencies to measure the data replication factor of XML database designs.
3. We develop a scalable technique for verifying XML functional dependencies with a single pass through the XML database. This technique can be easily extended for efficient incremental verification.
4. We present a platform-independent framework to illustrate how functional dependencies can enhance semantics in the use of XML.

The rest of the paper is organized as follows. Related works are discussed in Section 2. In Section 3, we propose a notation for functional dependencies in XML. In Section 4, a model is developed to estimate the replication factor for different designs of XML functional dependencies. We develop in Section 5 a technique that verifies the functional dependencies on an XML database with a single pass through the data and present the experimental results. We conclude in Section 6.

```
<!ELEMENT PSJ (Project)*>
<!ELEMENT Project (Supplier*)>
<!ELEMENT Supplier (Part*)>
<!ELEMENT Part (Price?,Quantity?)>
<!ATTLIST Project PName IDREF #REQUIRED>
<!ATTLIST Supplier SName IDREF #REQUIRED>
<!ATTLIST Part PartNo IDREF #REQUIRED>
<!ELEMENT Price (#PCDATA)>
<!ELEMENT Quantity (#PCDATA)>
```

(a) DTD.



(b) Sample data instance.

```
<PSJ>
  <Project PName="Garden">
    <Supplier SName="ABC Trading">
      <Part PartNo="P789">
        <Price>80</Price>
        <Quantity>500</Quantity>
      </Part>
      <Part PartNo="P123">
        <Price>10</Price>
        <Quantity>200</Quantity>
      </Price>
    </Supplier>
  </Project>


  <Project PName="Road Works">
    <Supplier SName="ABC Trading">
      <Part PartNo="P789">
        <Price>10</Price>
        <Quantity>50000</Quantity>
      </Part>
    </Supplier>
    <Supplier SName="DEF Pte Ltd">
      <Part PartNo="P123">
        <Price>12</Price>
        <Quantity>1000</Quantity>
      </Part>
    </Supplier>
  </Project>
</PSJ>
```

(c) XML instance.

Fig. 1. A Project-Supplier-Part XML Database

2 Related Works

XML is now the de-facto standard for information representation and interchange over the Internet. The basic idea behind XML is surprisingly simple. Instead of storing semantics-related information in data dictionaries as in the relational data model, XML embeds tags within the data to give meaning to the associated data fragments. Although verbose, XML has the advantage of being simple, interoperable and extensible. However, XML is not a data model, and there are currently no notions in XML corresponding to functional dependencies in the relational data model [Wid99].

Schema languages for XML have been heavily researched [Don00], with the Document Type Definition (DTD) [BPSMM00] and XML Schema [Fal00] being the most popular currently. DTD has been the de-facto schema language for XML for the past couple of years. DTD's support for key and foreign key concepts come in the form of `ID` and `IDREF(s)`. However, there are many problems in using them as (database) key constructs. In this work, we use the `ID/IDREF(s)` mechanisms as "internal pointers" rather than key constructs. XML Schema is an effort by W3C to overcome some shortfalls of the DTD and is touted to eventually replace DTD. XML Schema's improvements over DTD include the ability to specify type constraints and more complex cardinality constraints. However, current XML schema languages do not offer mechanisms to specify functional dependency constraints (which are different from key constraints).

In a survey which takes a database-centric view of XML, [Wid99] noted that the concept of functional dependencies in the context of XML has not been explored. However, *keys*, which is a special case of functional dependencies, is studied in the context of XML by [BDF$^+$01]. It offers two basic definitions for keys in XML : strong keys and weak keys. At one end of the spectrum, strong keys constrain that all key paths must exist and be unique. For weak keys, no structural constraint is imposed on key paths. Weak key paths can be missing, which makes weak keys similar to null-valued keys in the relational context. [BDF$^+$01] also noted that there is a spectrum of other possibilities in between their strong and weak definitions. [BDF$^+$01] also introduced the concept of *relative keys*. Noting that keys in many data formats (e.g. scientific databases) have a hierarchical structure, relative keys provide hierarchical key structures to accommodate such databases. Such structures are similar to the notion of ID dependent relationships in Entity Relationship diagrams. [FS00] proposed constraints languages to capture semantics of XML documents using simply key, foreign key and inverse constraints. However, functional dependencies, which form the theoretic foundation for keys, were not addressed in these works.

The concept of allowing controlled data redundancy in exchange for faster processing was introduced in [LGL96]. In this work, we present analogous arguments in the XML context for allowing controlled data redundancy. Preliminary work has also been done on semantics preservation when translating data between the relational tables and XML [LC00].

3 Functional Dependencies in XML

The well-known definition of functional dependencies for the relational data model is :

> Let r be a relation on scheme R, with X and Y being subsets of attributes in R. Relation r *satisfies* the *functional dependency* (FD) $X \rightarrow Y$ if for every X-value x, $\pi_Y(\sigma_{X=x}(r))$ has at most one tuple, with π and σ being the projection and selection operators respectively. □

This notation is designed for use with flat and tabular relational data, and cannot be directly applied to XML data. We need another notation that takes into consideration XML's hierarchical characteristics.

Definition 1. *An* ***XML functional dependency,*** $\mathbf{FD_{XML}}$*, is an expression of the form :*

$$(Q, [P_{x_1}, \ldots, P_{x_n} \rightarrow P_y]) \tag{1}$$

where

1. *Q is the FD_{XML} header path and is a fully qualified path expression (i.e. a path starting from the XML document root). It defines the scope in which the constraint holds.*
2. *Each P_{x_i} is a LHS (Left-Hand-Side) entity type. A LHS entity type consists of an element name in the XML document, and the optional key attribute(s). Instances of LHS entity types are called LHS entities and they can be uniquely identified by the key attribute(s).*
3. *P_y is a RHS (Right-Hand-Side) entity type. A RHS entity type consists of an element name in the XML document, and an optional attribute name. Instances of the RHS entity type is called a RHS entity.*

For any two instance subtrees identified by the FD_{XML} header, Q, if all LHS entities agree on their values, then they must also agree on the value of the RHS entity, if it exists. □

Our notation makes use of XPath [CD99] expressions. Informally, the FD_{XML} header, Q, restricts the scope and defines the *node set* in which the functional dependency holds. The node set can be seen as a collection of "records" over which the FD_{XML} holds. P_{x_i}, $1 \leq i \leq n$, identify the LHS entity types of the functional dependency, and P_y identifies the RHS entity type. They are synonymous with the LHS and RHS expressions of the conventional functional dependency expressions respectively.

Figure 2 shows two FD_{XML}s defined on the DTD shown in Figure 1a. The FD_{XML} header path of the first FD_{XML} (Figure 2a) asserts that this functional dependency holds over the subtrees rooted at `/PSJ/Project`. This FD_{XML} states that a supplier must supply a part at the same price regardless of projects. In the relational form, this will be $SName, PartNo \rightarrow Price$. This FD_{XML} is violated in the instance in Figure 1b. Supplier "ABC Trading" sells

```
( /PSJ/Project , [ Supplier.SName,
                   Part.PartNo
                   → Price ] )
```

(a)

```
( /PSJ , [ Project.PName,
           Supplier.SName,
           Part.PartNo
           → Quantity ] )
```

(b)

Fig. 2. Examples of FD_{XML} from Figure 1a.

part "P789" at price "80" to project "Garden", but sells the same part to project "Road Works" at price "10". The second FD_{XML} (Figure 2b) states that the quantity of parts is determined by the project, supplier and part concerned. This FD_{XML} holds over the subtree(s) rooted at `/PSJ`.

The attributes `PName`, `SName` and `PartNo` are the key attributes of the elements `Project`, `Supplier` and `Part` respectively. If we assume that these key attributes are known, then the notation in Figure 2 can be simplified to that shown in Figure 3.

```
( /PSJ/Project , [ Supplier,
                   Part
                   → Price ] )
```

(a)

```
( /PSJ , [ Project,
           Supplier,
           Part
           → Quantity ] )
```

(b)

Fig. 3. A simplified notation for FD_{XML}.

Figure 4 shows another XML database instance conforming to the same DTD in Figure 1a. Suppose the FD_{XML} in Figure 2a holds on this database. In order not to violate the functional dependency, C_1 has to be of the same value as C_2 since they have the same values for the LHS entities of the FD_{XML}. Note that the absence of a `Price` element in the rightmost subtree does not violate the FD_{XML}.

3.1 Well-Structured FD_{XML}s

XML models hierarchical data naturally and such models are especially useful for modeling attributes of relationships (although such "attributes" may be modeled as elements in XML). As a result, many FD_{XML}s are also hierarchically structured. In this section, we define what it means to be *well-structured* for the hierarchically structured FD_{XML}s. We first introduce the concept of a lineage. A set of nodes, L, in a tree is a lineage if:

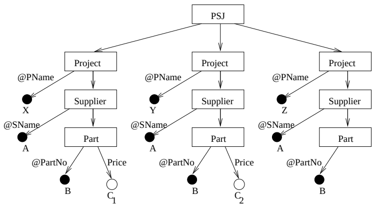


Fig. 4. Illustrating FD_{XML}

1. There is a node N in L such that all the nodes in the set are ancestors of N, and
2. For every node M in L, if the set contains an ancestor of M, it also contains the parent of M.

We use the DTD in Figure 1a to illustrate the concept of lineages. {`PSJ` , `Project`, `Supplier` , `Part`} is a lineage. `Part` satisfies condition 1 as all the other nodes are its ancestors. It can be verified that condition 2 for a lineage is satisfied by the nodes in this set. However, {`PSJ` , `Project` , `Part`} is not a lineage as condition 2 is not satisfied. Ancestors of `Part` (i.e. `PSJ` and `Project`) are in the set, but its parent (i.e. `Supplier`) is not[2].

Definition 2. *Consider the DTD :*

<!ELEMENT H_1 *(*H_2*)*>*
...
<!ELEMENT H_m *(*P_1*)*>*
<!ELEMENT P_1 *(*P_2*)*>*
...
<!ELEMENT P_{x-1} *(*P_x*)*>*

The FD_{XML}*,* $\mathcal{F}$=(Q,[P_1,...,$P_{x-1} \rightarrow P_x$]), *where* $Q = /H_1/\ldots/H_m$*, holds on this DTD.* $\mathcal{F}$ *is* ***well-structured*** *if :*

1. *there is a single RHS entity type.*
2. *the ordered XML elements in Q, LHS entity types and RHS entity type, in that order, form a lineage.*
3. *the LHS entity types are minimal (i.e. no redundant LHS entity types).* □

The FD_{XML} in Figure 2a is well-structured because:

1. There is a single RHS entity type (i.e. `Price`)
2. The XML elements in Q, LHS entity types, and RHS entity type (in that order) form a lineage. The RHS entity type satisfies condition 1 for a lineage. It can be easily verified that condition 2 is also satisfied.

[2] For the purpose of a lineage, the path needs not start from the document root, but can begin from any position in the XML document. This example can also be read as : "Since `//PSJ/Project/Part` (in XPath notation) is not a valid path in the DTD, it is not a lineage".

3. Both the supplier and part are required to determine the price (i.e. there is no redundant LHS entity types).

Well-structured FD_{XML}s are of specific interest as it presents the semantics of functional dependencies in a clear and succinct manner.

3.2 Non Well-Structured FD_{XML}s

The semantics for non well-structured FD_{XML}s cannot be generalised in a clear and consistent manner. For instance, the meaning of FD_{XML}s whose LHS entity types and RHS entity type do not form a lineage is ambiguous. Another case in point, the meaning of FD_{XML}s with LHS entities not sub-rooted under Q is unclear. Thus, as far as possible, functional dependencies in XML databases should be modeled as well-structured FD_{XML}s.

However, there is one special class of non well-structured FD_{XML}s, defined in *flat XML data*, that is meaningful. A characteristic of flat XML data is little or no nesting of elements and such databases model their data mainly as attributes. XML data with one level of element nesting is also known as *record-oriented* XML. Flat XML data is common because it is the simplest way to publish relational data as XML. In this section, we show how FD_{XML}s on them can be represented. Figure 5 shows a relational schema, its corresponding form in XML and the FD_{XML} representation for the functional dependency $city, state \rightarrow zipcode$ in the relation.

```
Student ( matric, name,
          street, city,
          state, zipcode)
FD : city, state → zipcode
```

(a) Relational schema.

```
<student_table>
 <student>
  <matric>...</matric> <name>...</name>
  <street>...</street> <city>...</city>
  <state>...</state> <zipcode>...</zipcode>
 </student>
 .......
</student_table>
```

(b) Data in XML.

```
( /student_table/student, [ city,
                            state
                            → zipcode])
```

(c) FD_{XML}.

Fig. 5. Example of flat XML data : A student database.

The FD_{XML} header path states that this constraint holds for the node set identified by the path `/student_table/student` (i.e. all student records). For every (student) node, if they agree on the values of `city` and `state`, they must also agree on the value of `zipcode`. The *well-structured* concept described previously does not apply to FD_{XML}s defined on flat XML data because all the LHS entity types are on the same level with no notion of a lineage. Thus, FD_{XML}s on flat XML data is a special case of non well-structured FD_{XML}s having clear semantics.

3.3 DTD for FD_{XML}

To facilitate interchange of the functional dependency constraints over the web, we propose a DTD for FD_{XML} (Figure 6a). We can easily to translate this DTD to other schema languages. The `Constraints` tag (line 1) will nest the functional dependencies, and can be extended to include other types of integrity constraints. The `Fid` (line 3) is the identifier of the functional dependency. Each `FD` will have a `HeaderPath` (line 6), at least one `LHS` (line 4) and one `RHS` (line 5), corresponding to Q, P_{x_i} and P_y respectively in FD_{XML}. The `ElementName` (line 7) child of both `LHS` and `RHS` elements contain element names in the XML database. The `Attribute` children of the `LHS` elements are the names of key attribute(s) of `ElementName`. Each `LHS` element can have multiple `Attribute` children to allow for multiple attributes from the same LHS element. The `ElementName` and `Attribute` children of `RHS` elements hold the name of the element/location which stores the value determined by the functional dependency. Each `RHS` element can have multiple `Attribute` children whose values are determined by the same LHS elements. Figure 6b shows how the FD_{XML} shown in Figure 2a is represented using our proposed DTD.

4 A Model for Measuring Replication in Well-Structured FD_{XML}s

Many databases are hierarchical in nature and nested XML data is well-suited for them. In contrast to the relational model, redundancy and replication are natural in hierarchical models (of which XML is one). In Figure 1, the price of each part is repeated each time the supplier supplies the part to a project. Although the schema can be carefully designed to remove redundancy and replication, this may destroy the natural structure and result in less efficient access. A possible "normalized" version of the database instance in Figure 1b is shown in Figure 7. This design does not replicate the price, but the database structure becomes less obvious since it makes extensive use of pointing relationships/references whose access are slower than containment/parent-child relationships [W$^+$00]. If re-ordering of the `Project-Supplier-Part` hierarchy is allowed, a better "normalized" version can be obtained by following the design rules in [WLLD01]. This design is not shown here due to lack of space, but the hierarchy in our running example becomes `Supplier-Part-Project`.

In the previous section, we have shown how FD_{XML}s can be modeled. In many cases, there will be replication of the same FD_{XML} instances due to the

```
<!ELEMENT Constraints (Fd*)>
<!ELEMENT Fd (HeaderPath,LHS+,RHS)>
<!ATTLIST Fd Fid ID #REQUIRED>
<!ELEMENT LHS (ElementName,Attribute*)>
<!ELEMENT RHS (ElementName,Attribute*)>
<!ELEMENT HeaderPath (#PCDATA)>
<!ELEMENT ElementName (#PCDATA)>
<!ELEMENT Attribute (#PCDATA)>
```

(a) DTD.

```
<Constraints>
  <Fd Fid="SP_Price_FD">
    <HeaderPath>/PSJ/Project</HeaderPath>
    <LHS>
      <ElementName>Supplier</ElementName>
      <Attribute>SName</Attribute>
    </LHS>
    <LHS>
      <ElementName>Part</ElementName>
      <Attribute>PartNo</Attribute>
    </LHS>
    <RHS>
      <ElementName>Price</ElementName>
    </RHS>
  </Fd>
</Constraints>
```

(b) A FD_{XML} for the Project-Supplier-Part Database conforming to the DTD.

Fig. 6. DTD for FD_{XML}

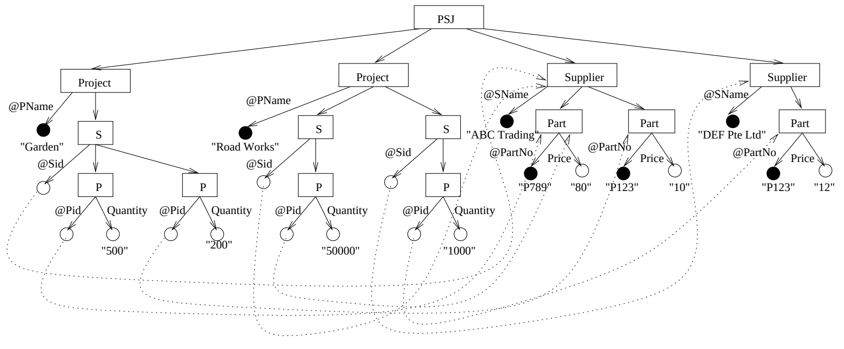
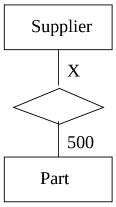


Fig. 7. A possible"normalized" instance of the Project-Supplier-Part database.

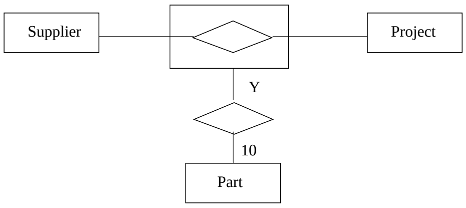
natural redundancy in XML data models. The degree of replication is of concern as it affects the effort needed to keep the database consistent. For example, an update to the value of a RHS entity must result in updating all RHS entities of the FD_{XML} instances to keep the data consistent. In this section, we present a model for estimating the degree of FD_{XML} replication. For simplicity, we limit our discussion of the model to **well-structured** FD_{XML}s. First, we introduce the concept of "*context cardinality*" of an element.

Definition 3. *The* ***context cardinality*** *of element X to element Y, denoted as* $Card_Y^X(\mathcal{D})$ *is the number of times Y can participate in a relationship with X in the context of X's entire ancestry in an XML document (denoted* $\mathcal{D}$*).* □

We can use various derivatives such as *maximum* or *average* context cardinalities. Suppose we have the following constraints for Figure 1: (1) Each supplier can supply at most 500 different parts. (2) Each supplier can supply at most 10 different parts to the same project. Then the maximum context cardinality of `Supplier` to `Part` is 10 since this cardinality is in the context of `Supplier`'s ancestry (i.e. `PSJ` and `Project`). We have $Card_{Part}^{Supplier}(\mathcal{D}) = 10$, where $\mathcal{D}$ is the context of our PSJ DTD. We use the Entity Relationship Diagram (ERD) in Figure 8 to illustrate the difference between traditional cardinality and the proposed context cardinality for XML. For simplicity, we omit the context parameter (i.e $\mathcal{D}$) in the notation when the context is obvious.



(a) ERD showing cardinalities of the relationship between Supplier and Part. (Each supplier can supply at most 500 different parts.)



(b) ERD showing **context cardinalities** of the relationship between Supplier and Part. (Each supplier can supply at most 10 different parts to the same project.)

Fig. 8. Context cardinality represented in ERD.

Suppose we have the following **well-structured** FD_{XML} , $\mathcal{F}$:

$$\mathcal{F} = (Q, [P_1, \ldots, P_{x-1} \rightarrow P_x]), \ where \ Q = /H_1/H_2/\ldots/H_m \tag{2}$$

The header path of $\mathcal{F}$ consists of the absolute path $/H_1/H_2/\ldots/H_m$. P_1 to P_{x-1} are LHS entity types and P_x is the RHS entity type. $\mathcal{F}$ is depicted graphically in Figure 9. Suppose this $\mathcal{F}$ holds on a DTD.

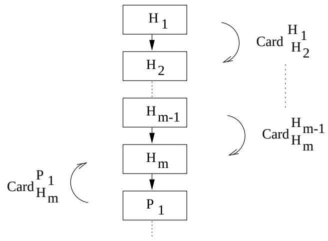


Fig. 9. Graphical depiction of $\mathcal{F}$.

Then the model for the replication factor, RF, for $\mathcal{F}$ is :

$$RF(\mathcal{F}) = min\left((\prod_{R=1}^{m-1} Card_{H_{R+1}}^{H_R}), Card_{H_m}^{P_1} \right) \tag{3}$$

This model obtains the frequency of repetition based on the context cardinalities of the elements. The frequency of repetition is determined by the smaller of two parameters. The first parameter is the product sum of the branching out factors of the header path elements (i.e. $(\prod_{R=1}^{m-1} Card_{H_{R+1}}^{H_R}))$. Intuitively, the number of times the data is replicated will be determined by how much "branching out" is caused by the elements in Q. In fact, if there are no other constraints, then this will be the replication factor. But the second parameter presents a constraint. This parameter $Card_{H_m}^{P_1}$ represents the number of times an element P_1 can be related to H_m in the context of P_1's ancestry. The final repetition factor of the FD_{XML} is constrained to be the smaller of the two parameters. Note that as $\mathcal{F}$ is well-formed, the entity types $P_2, P_3, \ldots, P_{x-1}, P_x$ are not involved in the model.

We illustrate the model using the DTD in Figure 1a. This example shows the process of estimating the number of times the price is replicated. Let the FD_{XML} in Figure 2a be $\mathcal{F}$ and assume it holds on the DTD in Figure 1a. Suppose we have at most 100 projects under the element (`/PSJ`) and each supplier can supply parts to at most 5 projects. Thus, we have :

$$\begin{aligned} \text{Constraint } 1: \ & Card_{Project}^{PSJ} = 100 \\ \text{Constraint } 2: \ & Card_{Project}^{Supplier} = 5 \end{aligned}$$

$$\begin{aligned} \text{RF}(\mathcal{F}) &= min\left((\textstyle\prod_{R=1}^{m-1} Card_{H_{R+1}}^{H_R}), Card_{H_m}^{P_1} \right) \\ &= min(100, 5) = 5 \end{aligned}$$

If we ignore constraint 2, then the repetition factor will be 100. Since a supplier supplies a part at the same price regardless of projects, the price will be repeated for every project the supplier supplies the part to. If this supplier supplies this part to all projects (i.e. 100), this price value will be repeated 100 times. However, constraint 2 states that a supplier can supply to a maximum of only 5 projects. Hence, this constraint limits the price value to repeat at most 5 times.

The replication factor can be used to gauge if the extra maintenance costs and increased effort to ensure data consistency is worth the faster response time for queries by replicating data. This factor obtained is the *maximum* replication factor as we used the maximum context cardinalities in the calculation. The *average* replication factor would have been computed if we have used the average context cardinalities. Numerical maximum (and minimum) context cardinalities can be obtained from some schema languages (e.g. XML Schema), but average context cardinalities may have to be estimated. As usual, the better the estimates are, the more accurate the model will be.

There are several design insights we can obtain from this model. One is that the FD_{XML} header path, Q, should be as short as possible (i.e. it should contain as few XML elements as possible). Each element in Q will increase the "branching out" and increase replication. Another insight is that by reducing the value of the second parameter in the model through careful design of the schema, the "branching out" of Q can be neutralized. For example, if the second parameter (i.e. $Card_{H_m}^{P_1}$) has the value 1, then we can ignore other constraints and be certain that there will not be any replication of FD_{XML} instances.

5 Verification of FD_{XML}s

We have presented FD_{XML} representations and redundancy considerations when designing XML databases with functional dependencies. In this section, we describe how FD_{XML}s can fit into a platform-independent framework that enriches the semantics of XML data. We also develop a scalable technique to verify FD_{XML}s with a single pass through the XML database.

In the framework, FD_{XML}s are first specified for an XML database. The XML database may then be distributed together with the FD_{XML} specification. The FD_{XML} specification is used to check if the distributed or updated XML data instances violate the specified constraints. If violations are detected, it may mean that the semantics of the data have changed, or that the data has undergone invalid transformations. The FD_{XML} specification is separate from the data, and it will not pose any overheads. It is introduced only to check the data for violations. Since the specification is also in XML, any existing XML tools or parsers can be used to process the specification.

Figure 10 gives the details of the verification process. The database and the FD_{XML} specification are parsed using an XML parser. The context information required is stored in state variables during parsing. The state variables provide context for the elements encountered during the parsing by retaining information such as the tag names and attribute values of their ancestors. The context information that needs to be stored is derived from the FD_{XML} specification.

When the parser encounters an element of the RHS entity type, we check the state variables to see if it occurs in the context as specified in the FD_{XML}. If the context is correct, the values of the LHS entities and the RHS entity are

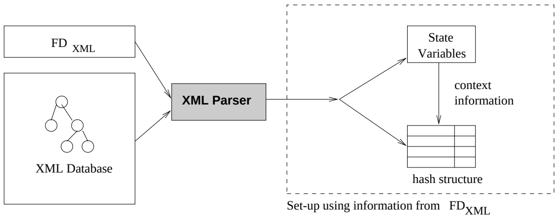


Fig. 10. Architecture for FD_{XML} verification.

stored into a hash structure. The hash structure maintains the values of the LHS entities, their RHS entities and the associated counts.

After the XML database is parsed, the entries in the hash structure which have more than one distinct RHS values are retrieved. These entries consist of those LHS entities who have different distinct RHS entity values, and hence, violate the FD_{XML}. The associated counts will be useful in determining the erroneous values. We use the XML database shown in Figure 1c to illustrate the approach. The hash structure after parsing this database is depicted in Figure 11. The first entry in the hash structure violates the FD_{XML} and subsequent differing RHS entity values and their counts are stored in a linked list.

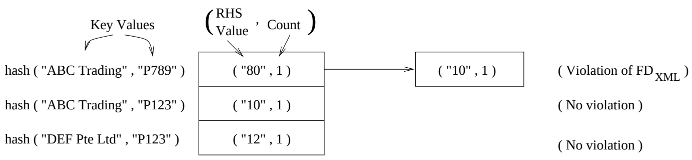


Fig. 11. State of hash structure after parsing the XML database in Figure 1c.

Note that only a single pass through the data is required for verification. If the hash structure is stored, future updates to the data will only require checking of possible violations against the data in the hash structure, and no file scan is necessary. This is an efficient way of performing **incremental** verification.

5.1 Experimental Results

We conducted experiments to evaluate our technique for FD_{XML} verification using two popular interfaces for XML parsing. We also test the scalability of the technique by measuring the response times and the size of the hash structure as we increase the size of the XML database. The journal article entries (about 80000) in the DBLP database [Ley01], an example of flat XML data, are used for the experiments. The experiments are performed on a Pentium III 850 MHz machine with 128MB of RAM running Windows NT 4.0. The verification program is implemented using Java and makes use of the Xerces parser [Pro01].

```
<article key="journals/cacm/Mayer79">
  <author>Richard E. Mayer</author>
  <title>A Psychology of Learning BASIC.</title>
  <journal>CACM</journal> <volume>22</volume> <number>11</number>
  <pages>589-593</pages> <year>1979</year>
  <url>db/journals/cacm/cacm22.html#Mayer79</url>
</article>
```

Fig. 12. A sample journal article entry in the DBLP database.

A sample article entry is shown in Figure 12. To illustrate how our technique works when violations occur, we assume that "*all articles in the same volume of the same journal should be published in the same year*"[3]. This FD_{XML} is denoted as :

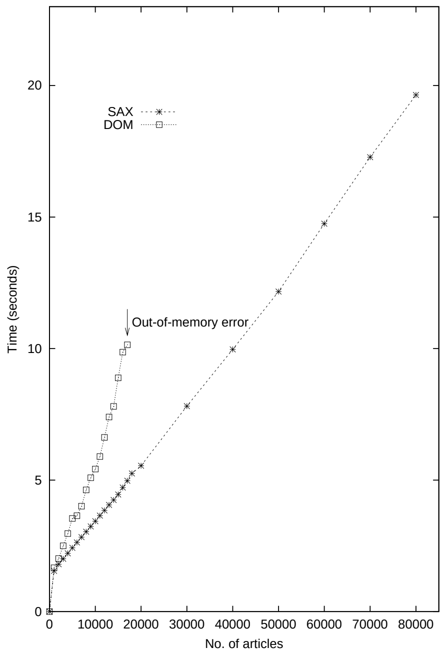
```
( /dblp/article, [ journal,
                   volume
                   → year])
```

Note that in this flat XML database, the order of occurrence of the `journal`, `volume` and `year` elements is not important. However, they have to be children of the path `/dblp/article`.

Experiment 1. We evaluate the performance and scalability of two popular interfaces for XML parsing : Simple API for XML (SAX) [Meg01] and the Document Object Model (DOM) [W3C01]. SAX uses event-driven parsing and DOM is a tree-based API. Figure 13 shows the runtime of the experiments using SAX and DOM parsers. Using a DOM parser, which builds an in-memory DOM tree of the articles, an out-of-memory error was encountered at about 18000 articles. Using a SAX parser, we are able to verify the FD_{XML} across all 80000 articles successfully. The runtime using SAX increases linearly with the number of articles, which is expected since the data needs to be parsed only once. Due to space constraints, we do not compare SAX and DOM further. But clearly, SAX is the more scalable interface for XML parsing.

Experiment 2. We measure the size of the hash structure as we increase the number of articles. The results are shown in Figure 14. Although there is a linear increase, the absolute numbers of hashed keys (i.e. {journal,volume}) is very much smaller than the number of articles. This is attributed to the fact that there are many journal articles with the same `journal` and `volume` values. In fact, for the worst case, the hash structure will only get as large as the number of articles. Only a small number of the hashed keys have "errors" or "violations" (i.e. a volume of a journal contains articles published in different years). Such "violations" will result in a linked-list containing the different RHS entity values and their counts. Further analysis shows that the average length of the linked lists of such "violations" is only about 2-3.

[3] This is typically not true as not all issues of the volume may be published in the same year. But this FD_{XML} is assumed so as to generate "errors".



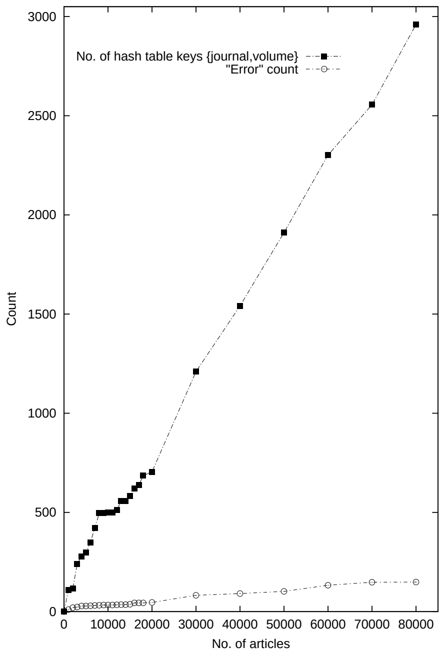


Fig. 13. Runtime vs. number of articles using SAX and DOM.

Fig. 14. Size of hash structure and number of key entries with linked-list.

```
Journal : CACM              Volume : 32
    Year : 1989   ( Count = 100 )
    Year : 1988   ( Count = 1 )
Journal : AI Magazine       Volume : 11
    Year : 1990   ( Count = 38 )
    Year : 1991   ( Count = 14 )
```

Fig. 15. Sample output of verification process.

Figure 15 shows sample output after verification of our assumed FD_{XML} (i.e. the journal name and volume number uniquely determines the year of publications). For Volume 32 of the journal "Communications of the ACM", 100 articles have the year "1989", while a single article has the year "1988". This seems to be an error, and a good guess for the correct year value will be "1989". However, if this FD_{XML} is to hold, the correct value for Volume 11 of the "AI Magazine" will not be so clear, as the counts of the different year values are not as indicative[4].

[4] This is not an error. Issues 1-4 of AI Magazine Volume 11 were published in 1990. Issue 5 of Volume 11 was published in 1991.

The experimental results show that functional dependency constraints in XML databases can be efficiently verified in a single pass through the database. Our technique of FD_{XML} verification does not take up much memory and scales up well, especially with the use of the SAX API for parsing the XML database. We have also shown how violations of a known FD_{XML} constraint are detected.

6 Conclusion

Functional dependency constraints have been an integral part of traditional database theory and it is important to see how this concept can be applied to XML. Existing works on XML keys define key dependencies within XML structures, but not relationships between these XML structures. This work fills this gap by proposing a representation and semantics for functional dependencies in XML databases to complement the work on XML keys. FD_{XML} is a schema language-independent representation for XML functional dependencies and is expressed in XML. Redundancy is natural in XML and we have developed a replication cost model to give a measure of data replication with well-structured FD_{XML}s. This model also provides insights into the design of FD_{XML}s to minimize redundancy. We also present a platform-independent framework for FD_{XML}'s use and deployment. We show how FD_{XML}s semantically enrich XML through the specification of functional dependencies on XML data. The specified constraints can be used for the verification of data consistency and to prevent illegal insert and update operations. We also present a technique for verifying FD_{XML}s which requires only a single pass through the XML database. Experimental results show that this technique is scalable with large real life XML databases. Our technique can be easily extended for efficient incremental verification.

There is much future work in this area. It is worthwhile to investigate if there exists other classes of non well-structured FD_{XML}s (besides those defined in flat XML data) which are meaningful. The cost model can then be extended to include such classes. In the relational data model, reasoning about functional dependencies have led to useful implication rules. It is interesting to see if such implication rules can be extended for FD_{XML}.

References

[Arm74] W. W. Armstrong. Dependency Structures of Database Relationships. In *Proceedings of the tri-annual IFIP Conf 74, N-H (Amsterdam)*, 1974.

[BDF+01] Peter Bunemana, Susan Davidson, Wenfei Fan, Carmem Hara, and Wang-Chiew Tan. Keys for XML. In *Proceedings of the WWW'10, Hong Kong, China*, 2001.

[Ber76] P. A. Bernstein. Synthesizing Third Normal Form Relations from Functional Dependencies. *ACM Transactions on Database Systems*, 1(4):277–298, Dec 1976.

[BPSMM00] Tim Bray, Jean Paoli, C. M. Sperberg-McQueen, and Eve Maler. Extensible Markup Language (XML) 1.0 (Second Edition). `http://www.w3.org/TR/2000/REC-xml-20001006`, 2000.

[CD99] James Clark and Steve DeRose. XML Path Language (XPath) Version 1.0. Available at `http://www.w3.org/TR/xpath`, 1999.

[Cod70] E. F. Codd. A Relational Model of Data for Large Shared Data Banks. *j-CACM*, 13(6):377–387, June 1970.

[Cod72] E. F. Codd. *Further Normalization of the Database Relational Model.* R. Rustin, Ed. Prentice-Hall, Englewood Cliffs, NJ, 1972.

[Don00] Dongwon Lee and Wesley W. Chu. Comparative Analysis of Six XML Schema Languages. *SIGMOD Record*, 29(3):76–87, 2000.

[Fal00] D. Fallside. XML Schema Part 0: Primer. Available at `http://www.w3.org/TR/xmlschema-0/`, 2000.

[FR01] Mary Fernandez and Jonathan Robie. XML Query Data Model. W3C Working Draft. Available at `http://www.w3.org/TR/query-datamodel/`, 2001.

[FS00] W Fan and J Siméon. Integrity Constraints for XML. In *Proceedings of the Nineteenth ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems, Dallas, Texas, USA*, pages 23–34. ACM, 2000.

[LC00] Dongwon Lee and Wesley W. Chu. Constraints-Preserving Transformation from XML Document Type Definition to Relational Schema. In *Proceedings of the 19th International Conference on Conceptual Modeling*, pages 323–338, 2000.

[Ley01] Michael Ley. DBLP Bibliography. Available at `http://www.informatik.uni-trier.de/ ley/db/`, 2001.

[LGL96] Tok Wang Ling, Cheng Hian Goh, and Mong Li Lee. Extending classical functional dependencies for physical database design. *Information and Software Technology*, 9(38):601–608, 1996.

[MAG$^+$97] J. McHugh, S. Abiteboul, R. Goldman, D. Quass, and J. Widom. Lore: A Database Management System for Semistructured Data. *SIGMOD Record*, 26(3), 1997.

[Meg01] David Megginson. SAX: The Simple API for XML. Available at `http://www.megginson.com/SAX/`, 2001.

[Pro01] The Apache XML Project. Xerces Java Parser. Available at `http://xml.apache.org/xerces-j/index.html`, 2001.

[TF82] Tsou and Fischer. Decomposition of a Relation Scheme into Boyce-Codd Normal Form. *SIGACTN: SIGACT News*, 14, 1982.

[W$^+$00] Kevin Williams et al. *Professional XML Databases.* Wrox Press Inc, 2000.

[W3C01] W3C DOM Working Group. Document Object Model (DOM). Available at `http://www.w3.org/DOM/`, 2001.

[Wed92] Grant E. Weddell. Reasoning About Functional Dependencies Generalized for Semantic Data Models. *ACM Transactions on Database Systems*, 17(1):32–64, Mar 1992.

[Wid99] Jennifer Widom. Data Management for XML: Research Directions. *IEEE Data Engineering Bulletin*, 22(3):44–52, 1999.

[WLLD01] Xiaoying Wu, Tok Wang Ling, Mong Li Lee, and Gillian Dobbie. Designing Semistructured Databases Using the ORA-SS Model. In *Proceedings of the 2nd International Conference on Web Information Systems Engineering (WISE).* IEEE Computer Society, 2001.