March 2003

Propagating XML Constraints to Relations

Susan B. Davidson  
*University of Pennsylvania*, susan@cis.upenn.edu

Wenfei Fan  
*Temple University*

Carmem Hara  
*Universidade Federal do Parana*

Jing Qin  
*Temple University*

Follow this and additional works at: [http://repository.upenn.edu/cis_papers](http://repository.upenn.edu/cis_papers)

Recommended Citation  


This material is posted here with permission of the IEEE. Such permission of the IEEE does not in any way imply IEEE endorsement of any of the University of Pennsylvania's products or services. Internal or personal use of this material is permitted. However, permission to reprint/republish this material for advertising or promotional purposes or for creating new collective works for resale or redistribution must be obtained from the IEEE by writing to pubs-permissions@ieee.org. By choosing to view this document, you agree to all provisions of the copyright laws protecting it.

This paper is posted at ScholarlyCommons. [http://repository.upenn.edu/cis_papers/118](http://repository.upenn.edu/cis_papers/118)

For more information, please contact libraryrepository@pobox.upenn.edu.
Propagating XML Constraints to Relations

Abstract
We present a technique for refining the design of relational storage for XML data based on XML key propagation. Three algorithms are presented: one checks whether a given functional dependency is propagated from XML keys via a predefined view; the others compute a minimum cover for all functional dependencies on a universal relation given XML keys. Experimental results show that these algorithms are efficient in practice. We also investigate the complexity of propagating other XML constraints to relations, and the effect of increasing the power of the transformation language. Computing XML key propagation is a first step toward establishing a connection between XML data and its relational representation at the semantic level.

Comments

This material is posted here with permission of the IEEE. Such permission of the IEEE does not in any way imply IEEE endorsement of any of the University of Pennsylvania's products or services. Internal or personal use of this material is permitted. However, permission to reprint/republish this material for advertising or promotional purposes or for creating new collective works for resale or redistribution must be obtained from the IEEE by writing to pubs-permissions@ieee.org. By choosing to view this document, you agree to all provisions of the copyright laws protecting it.
Propagating XML Constraints to Relations

Susan Davidson*  
U. of Pennsylvania

Wenfei Fan†  
Bell Labs

Carmem Hara  
U. Federal do Parana, Brazil

Jing Qin  
Temple U.

Abstract

We present a technique for refining the design of relational storage for XML data based on XML key propagation. Three algorithms are presented: one checks whether a given functional dependency is propagated from XML keys via a predefined view; the others compute a minimum cover for all functional dependencies on a universal relation given XML keys. Experimental results show that these algorithms are efficient in practice. We also investigate the complexity of propagating other XML constraints to relations, and the effect of increasing the power of the transformation language. Computing XML key propagation is a first step toward establishing a connection between XML data and its relational representation at the semantic level.

1 Introduction

Over the past five years, XML has become enormously popular as a data exchange format. A common paradigm is for a data provider to export its data using XML; on the other end, the data consumer imports some or all of the XML data and stores it using database technology. Since the XML data being transmitted is often large in size and fairly regular in structure, the database technology used is frequently relational.

A problem with XML is that it is only syntax and does not carry the semantics of the data. To address this problem, a number of constraint specifications have recently been proposed for XML which include a notion of keys; such proposals have also found their way into XML-Data [18] and XML Schema [28]. A natural question to ask, therefore, is how information about constraints can be used to determine when an existing consumer database design is incompatible with the data being imported, or to generate de-novo a good consumer database. We illustrate the problem below.

Example 1.1: Suppose that the XML data (represented as a tree) in Fig. 1 is being exchanged and that the initial design of the consumer database has a single table Chapter with fields bookTitle, chapterNum

and chapterName (written Chapter(bookTitle, chapterNum, chapterName)). The table is populated from the XML data as follows: For each book element, the value of the title subelement is extracted. A tuple is then created in the Chapter relation for each chapter subelement containing the title value for bookTitle, the number value for chapterNum, and the name value for chapterName (see Fig. 2(a) for the resulting relational instance.) The key of the Chapter table has been specified as bookTitle and chapterNum. While importing this XML data, violations of the key are detected because two different books have the same title (“XML”) and disagree on the name of chapter one (“Introduction” versus “Getting Acquainted”). After digressing through the documentation accompanying the XML data, the database designers decide to change the schema to Chapter(isbn, chapterNum, chapterName) with a key of isbn and chapterNum (populated in the obvious way from the XML data). The resulting relational instance is shown in Fig. 2(b). While importing the XML data, no violations of the key constraint are detected. However, the designers are not sure whether they were lucky with this particular XML data set, or whether such violations will never occur.

It turns out that given the following keys on the XML data, the designers of the consumer database could prove that the key of Chapter in their modified design is correct:

1. isbn uniquely identifies a book element.
2. Within each book, number is a key for chapter, i.e., number is a key for chapter relative to book.
3. Each book has a unique title, and within each

<table>
<thead>
<tr>
<th>bookTitle</th>
<th>chapterNum</th>
<th>chapterName</th>
</tr>
</thead>
<tbody>
<tr>
<td>XML</td>
<td>1</td>
<td>Introduction</td>
</tr>
<tr>
<td>XML</td>
<td>10</td>
<td>Conclusion</td>
</tr>
<tr>
<td>XML</td>
<td>1</td>
<td>Getting Acquainted</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>isbn</th>
<th>chapterNum</th>
<th>chapterName</th>
</tr>
</thead>
<tbody>
<tr>
<td>123</td>
<td>1</td>
<td>Introduction</td>
</tr>
<tr>
<td>123</td>
<td>10</td>
<td>Conclusion</td>
</tr>
<tr>
<td>234</td>
<td>1</td>
<td>Getting Acquainted</td>
</tr>
</tbody>
</table>

(a) Chapter: the initial design

(b) Chapter: a refined design

Figure 2. Sample relational instances

*Research supported by NSF DBI-9975206.
†Research supported in part by NSF Career Award IIS-0093168. Currently on leave from Temple University.
book, each chapter has a unique name.

That is, if these XML keys hold on the data being imported, then isbn,chapterNum → chapterName is a functional dependency (FD) that is guaranteed to hold on the Chapter relation generated (in other words, isbn, chapterNum is a key of the relation). We refer to the FD as one that is propagated from these XML keys.

In general, given a transformation to a predefined relational schema and a set of XML keys, one wants to know whether or not an FD is propagated from Σ via the transformation. Let us refer to this problem as XML key propagation. The ability to compute XML key propagation is important in checking the consistency of a predefined relational schema for storing XML data.

On the other hand, suppose that the relational database is designed from scratch or can be re-designed to fit the constraints (and thus preserve the semantics) of the data being imported. A common approach to designing a relational database is to start with a rough schema and refine it into a normal form (such as BCNF or 3NF [1]) using FDs. In our scenario, we assume that the designer specifies the rough schema by a mapping from the XML document. The FDs over that rough schema must then be inferred from the keys of the XML document using the mapping. However, it is impractical to compute the set F of all the FDs propagated since F is exponentially large in the number of attributes. We would therefore like to find a minimum cover [1] of F, that is, a subset $F_m$ of F that is equivalent to F (i.e., all the FDs of $F_m$ can be derived from $F_m$ using Armstrong’s Axioms) and is non-redundant (i.e., none of the FDs in $F_m$ can be derived from other FDs in $F_m$).

**Example 1.2:** Returning to our example, suppose that the database designers decide to start from scratch and initially propose a schema of Chapter(isbn, booktitle, author, chapterNum, chapterName), and Author(isbn, author). Note that isbn → author is not mapped from the keys since a book may have several authors.

**Contributions.** In this paper, we propose a framework for improving consumer relational database design. Our approach is based on inferring functional dependencies from XML keys through a given mapping (transformation) of XML data to relations. The class of XML keys considered includes those commonly found in practice, and is a subset of those in XML Schema [27]. More specifically, we make the following contributions:

- A polynomial time algorithm for checking whether an FD on a predefined relational database is propagated from a set of XML keys via a transformation.
- A polynomial-time algorithm that, given a universal relation specified by a transformation rule and a set of XML keys, finds a minimum cover for all the functional dependencies mapped from XML keys.
- Undecidability results that show the difficulty of XML constraint propagation.
- Experimental results which show that the algorithms are efficient in practice.

Note that the polynomial-time algorithm for finding a minimal cover from a set of XML keys is rather surprising, since it is known that a related problem in the relational context – finding a minimum cover for functional dependencies embedded in a subset of a relation schema – is inherently exponential [16].

The undecidability results give practical motivation for the restrictions adopted in this paper. In particular, one result shows that it is impossible to effectively propagate all forms of XML constraints supported by XML Schema, which include keys and foreign keys, even when the transformations are trivial. This motivates our restriction of constraints to a simple form of XML keys. Another undecidability result shows that when the transformation language is too rich, XML constraint propagation is also not feasible, even when only keys are considered. Since XML to relational transformations are subsumed by XML to XML transformations expressible in query languages such as XQuery [8], this negative result applies to most popular XML query languages.
Related Work. In [14, 13], a chase/backchase method is presented which can be used for determining constraint propagation in a semistructured data model when views are expressed in CRPQ (conjunctive regular path queries) and dependencies are DERPDs (disjunctive embedded regular path dependencies). However, the method does not compute a minimum cover for propagated FDs; it is also too general to be efficient for checking propagation of XML keys. The CPI algorithm of [19] is orthogonal to our work and derives constraints from DTDs. Our work also parallels that of [2], which investigates propagation of type constraints through queries.

The problem of finding a cover for FDs embedded in a subset of a relational schema has been studied in [16] and shown to be inherently exponential. It is worth mentioning that the problem of computing embedded FDs cannot be reduced to ours since the XML key language cannot capture relational FDs, and vice versa.

Approaches for using a relational database to store XML data include [21, 24, 25, 5]. However, our framework and algorithms are the first results on mapping XML constraints through relational views. The transformation language developed in this paper is also similar to that of Stored [12] and aspects of the new release of Oracle (9i) [22].

Organization. The next section describes the class of XML keys considered and our transformation language. Section 3 states the constraint propagation problem and establishes the undecidability results. Sections 4 and 5 present algorithms for computing XML key propagation and minimum cover. Experimental results are given in Section 6, followed by our conclusions in Section 7. Complete details are given in the full version of the paper [11].

2 XML Keys and Transformations

XML keys. To define a key we specify three things: 1) the context in which the key must hold; 2) a target set on which we are defining a key; and 3) the values which distinguish each element of the target set. For example, the second key specification of Example 1.1 has a context of each element of the target set. For example, the second key value, "//" means descendant-or-self in XPath. To avoid confusion we write /P/Q for the concatenation of P, "//" and Q. A path ρ is a sequence of labels l1,...,ln. A path expression Q defines a set of paths, while “//” can match any path. We use ρ ∈ Q to denote that ρ is in the set of paths defined by Q. For example, book/author/name ∈ //name.

Following the syntax of [6] we write an XML key as:

\[ K = \{ (C, (T, \{P1, ..., P_r\})) \} \]

where K is the name of the key, path expressions C and T are the context and target path expressions respectively, and P1,...,P_r are the key paths. For the purposes of this paper, we restrict the key paths to be simple attributes @A1,...,@A_r, and denote this class of keys as \( K_a \). A key is said to be absolute if the context path C is the empty path ε, and relative otherwise.

Example 2.1: Using this syntax, the sample constraints from Section 1 and others can be written as follows:

1. \( KS_1 : (\epsilon, (//book[@isbn])): \) within the context of the entire document (ε denotes the root) a book element is identified by its @isbn attribute. The book node can occur anywhere in the tree.
2. \( KS_2 : (//book[chapter,@number]): \) within the context of any subtree rooted at a book node, a chapter is identified by its @number attribute. The chapter node must be immediately under the book node.
3. \( KS_3 : (//book[title,:]): \) each book has at most one title: similarly,
4. \( KS_4 : (//book[chapter,(name,:)]): \) for the name of a chapter, and
5. \( KS_5 : (//book[chapter,section,(name,:)]): \) for section name.
6. \( KS_6 : (//book[chapter,section,(section,:)]): \) within the context of a chapter of a book, each section is identified by its @number attribute.
7. \( KS_7 : (//book[author/contact,:]): \) a book can have multiple authors, but at most one has contact information (the contact author).

To define the meaning of an XML key, we use the following notation: in an XML document (tree), \( n[P] \) denotes the set of node identifiers that can be reached by following path expression P from the node with identifier n. \( [P] \) is an abbreviation for r[P], where r is the root node of the tree.

Example 2.2: In Fig. 1, [book] = {1,19} and [chapter] = {6,7} and [number] = {131,524,2832}.

Definition 2.1: An XML tree T satisfies an XML key \( \phi : (C, (T, \{A1,...,A_r\})) \), denoted \( T \models \phi \), iff for any n in C and any \( n_1, n_2 \) in n[T], (1) \( n_1 \) and \( n_2 \) each has a unique attribute \( A_i \) for all \( i \in [1,p] \), and (2) if \( val(n_1, A_i) = val(n_2, A_i) \) for all \( i \in [1,p] \) then \( n_1 = n_2 \), where \( val(n', A_i) \) denotes the text value associated with the attribute \( A_i \) of \( n' \).

Example 2.3: The XML tree of Fig. 1 satisfies our sample constraints. For example, \( KS_1 \) is satisfied since \( //book = \{1,19\} \) and \( val(16, @isbn) \neq val(19, @isbn) \). One can check \( KS_2 \) by verifying the absolute key

\footnote{We adopt this syntax for keys because it is more concise than that of XML Schema.}
This definition of keys has several salient features: First, keys can be scoped within the context of the entire document (an absolute key), or within the context of a subdocument (a relative key). Second, the specification of keys is orthogonal to the typing specification for the document (e.g. DTD or XML Schema). The type of documents will therefore be ignored throughout this paper. Combining keys with schema information, as is done in XML Schema, adds complexity to the inference problem. As demonstrated by [3], it is NP-hard even to check whether XML Schema keys are satisfiable, i.e., whether there exist any XML document which satisfies those keys. In contrast, the keys studied here are always satisfiable [7].

**Transformation Language.** The transformation language forms a core of many common transformations found throughout the literature, in particular those of [25].

**Definition 2.2:** A transformation $\sigma$ from XML data to relations of schema $R = (R_1, \ldots, R_n)$ is specified as $\langle Rule(R_1), \ldots, Rule(R_n) \rangle$, where each $Rule(R_i)$, referred to as the table rule for $R_i$, is defined with:

- a set $X_i$ of variables, in which $x_i$ is a distinguished variable, referred to as the root variable;
- a set of field rules $\{l : \text{value}(x) | l \in \text{att}(R_i)\}$, where $x$ is a distinct variable in $X_i$, and $\text{att}(R_i)$ denotes the set of attributes in the schema of relation $R_i$;
- a set of variable mapping rules of the form $x \leftarrow y/P$; where $x, y \in X_i$ and $P$ is a path expression.

In addition, each variable $x \in X_i$ is connected to the root $r$; that is, $x$ is specified with either $x \leftarrow x_r/P$ in the rule, or $x \leftarrow y/P$ and $y$ is connected to the root $r$; moreover, for any $x \leftarrow y/P$, $P$ is a simple path (i.e. without //) unless $y$ is $x_r$, and 2) no field rule is defined as $l : \text{value}(x)$ when there exists a variable $x$ specified with $x \leftarrow y/P$.

**Example 2.4:** Expanding on Example 1.1, consider the following schema $R$ (with keys underlined):

- book(isbn, title, author, contact),
- chapter(inBook, number, name),
- section(inChapt, number, name).

A transformation $\sigma$ from the XML data of Fig. 1 to $R$ could be specified as:

$\sigma = \langle \text{Rule(book), Rule(chapter), Rule(section)} \rangle$

$\text{Rule(book)} = \{ \text{isbn: value(z_1), title: value(z_2), author: value(z_3), contact: value(z_4)},$
- $x_b \leftarrow x_r/\text{book}, x_1 \leftarrow x_b/@\text{isbn}, x_2 \leftarrow x_b/\text{title},$
- $x_a \leftarrow x_b/\text{author}, x_3 \leftarrow x_a/@\text{name}, x_4 \leftarrow x_a/\text{name}; \}$

$\text{Rule(chapter)} = \{ \text{inBook: value(y_1), number: value(y_2), name: value(y_3)},$
- $x_c \leftarrow x_r/\text{chapter}, x_1 \leftarrow x_c/@\text{number}, x_2 \leftarrow x_c/\text{name}; \}$

**Table trees.** Throughout the remainder of the paper, we will use an abstract representation of a table rule called a table tree. The idea is that by treating "//" as a special node label, each table rule can be represented as a node-labeled tree. For example, Fig. 3 depicts the table trees for $\text{Rule(book)}$ and $\text{Rule(section)}$ in Example 2.4. In a table tree $TR$ representing $\text{Rule(R)}$, each variable in $\text{Rule(R)}$ corresponds to a unique node, and each node corresponds to at most one variable.

**Semantics.** Given an XML tree $T$, each $\text{Rule(R_i)}$ maps $T$ to an instance $I_i$ of $R_i$. More specifically, given a variable specification $x \leftarrow y/P$, $x$ ranges over $y[P]$; $x_r$ is always interpreted as the root $r$. A field rule $l : \text{value}(x)$ populates the $l$ field with values in $\{\text{value}(x) | x \in y[P]\}$, where function $\text{value}$ returns a string representing the pre-order traversal of the subtree rooted at $x$. Let $\text{att}(R_i) = \{l_1, \ldots, l_k\}$ and each variable $x$ be specified with $x \leftarrow l_i/P$. Then the instance $I_i$ is generated by $I_i = ((l_1 : \text{value}(x_1), \ldots, l_k : \text{value}(x_k)) | x_r = r, x \in x_r[P], x \in X_i)$.

**Example 2.5:** $\text{Rule(section)}$ is interpreted as:

$\{ (\text{inChapt: value(z_1), number: value(z_2),}$
- $\text{name: value(z_3)}, z_1 \in z_3[@\text{number}], z_2 \in z_3[@\text{section}],$
- $z_3 \in z_3[@\text{name}] \}$

Referring to the XML tree $T$ in Fig. 1, $\text{value(6)}$ returns $(@\text{number}: 1, \text{name: (S: Introduction)})$. The interpretation of the rule for section (Example 2.4) over $T$ generates following instance:

<table>
<thead>
<tr>
<th>section</th>
<th>inChapt</th>
<th>number</th>
<th>name</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>2</td>
<td>Fundamentals</td>
</tr>
<tr>
<td>1</td>
<td></td>
<td></td>
<td>Attributes</td>
</tr>
</tbody>
</table>
Several subtleties are worth mentioning. First, since
XML data is semistructured it is possible that for \( x \leftarrow y/P \),
\( y[P] \) is empty. In this case \( \text{value}(x) \) is defined to be \text{null}. Second, if \( y[P] \) has multiple elements, then to generate the
relation, an implicit Cartesian product is computed so that
all nodes in \( y[P] \) are covered in the relation.

3 Problem Statement and Limitations

**Key propagation.** The question of key propagation asks if
given a transformation \( \sigma \) from XML data to relations of a
fixed schema \( R \) and an XML tree \( T \) satisfying a set \( \Sigma \) of
XML keys, whether \( \sigma(T) \) satisfies an FD \( \varphi \) (on a schema \( R \)
in \( R \)). We write \( \Sigma \models_{\sigma} R : \varphi \) if the implication holds for all
XML trees satisfying \( \Sigma \), and refer to \( \varphi \) as an FD propagated
from \( \Sigma \). With respect to a transformation specification language,
the key propagation problem is to determine, given
any \( \sigma \) expressed in the language, any XML keys \( \Sigma \) and an
FD \( \varphi \), whether or not \( \Sigma \models_{\sigma} R : \varphi \). Note that we do not
require the XML data to conform to any type specification.

A subtle issue arises from \text{null} values in \( \sigma(T) \), the
relations generated from an XML tree \( T \) via \( \sigma \). In particular,
there may exist \( R \) tuples in \( \sigma(T) \) with FD \( X \rightarrow Y \) such that
their \( X \) or \( Y \) fields contain \text{null}. The presence of \text{null}
complicates FD checking since comparisons of \text{null} with any value do not evaluate to a Boolean value [23]. A brutal
solution is to restrict the semantics of the transformation \( \sigma \)
so that a tuple is not included if it has a \text{null} field. Since
XML is semistructured, this could exclude a large number of
“incomplete tuples” from \( \sigma(T) \). We therefore adopt the
following semantics of FDs: \( \sigma(T) \) satisfies FD \( X \rightarrow Y \),
denoted by \( \sigma(T) \models X \rightarrow Y \), if (1) for any tuple \( t \) in \( R \),
if \( \pi_X(t) \) contains \text{null} then so does \( \pi_Y(t) \); and (2) for
tuples \( t_1, t_2 \) in \( R \) if neither \( t_1 \) nor \( t_2 \) contains \text{null} and
\( \pi_X(t_1) = \pi_X(t_2) \), then \( \pi_Y(t_1) = \pi_Y(t_2) \). The motivation
behind the first condition is that an FD is possibly treated as
a key when normalizing the relational schema, and an
“incomplete key” \( X \) cannot determine complete \( Y \) fields.

Another issue we should address is the simplicity of the
transformation language, which can only express projection
(\( \pi \)), Cartesian product (\( \times \)) and a limited form of set union
(\( \cup \)). One might be tempted to develop a richer language
which can express all relational algebra operators: projection,
selection (\( \sigma \)), Cartesian product, set union and difference (\( - \)). Although these operators can be generalized to
XML trees, the following negative result holds:

**Theorem 3.1:** The key propagation problem from XML to
relational data is undecidable when the transformation language
can express all relational algebra operators.

The undecidability is established by reduction from the
equivalence problem for relational algebra queries (see [11]
for a proof); the latter is a well-known undecidable prob-
lem [1]. In contrast, for our transformation language there
is a polynomial time algorithm in the size of \( \Sigma \) and \( \sigma \).

**Example 3.1:** Recall the schema \( R \) and the transformation
given in Example 2.4. A universal relation \( U \) here is the collection
of all the fields of \( R \), defined as follows:

\[
U = \{ \text{bookIsbn, bookTitle, bookAuthor, authContact, chapNum, chapName, secNum, secName} \}
\]

\[
\text{Rule } (U) = \{ \text{bookIsbn: value(x1), bookTitle: value(x2), bookAuthor: value(x3), authContact: value(x4), chapNum: value(y1), chapName: value(y2), secNum: value(z1), secName: value(z2)} \}
\]

The table tree of Rule (\( U \)) is depicted in Fig. 4.

From the set of XML keys of Example 2.1 the following
minimum cover for the FDs on \( U \) can be computed:

- bookIsbn \rightarrow bookTitle,
- bookIsbn \rightarrow authContact,
- bookIsbn, chapNum \rightarrow chapName,
- bookIsbn, chapNum, secNum \rightarrow secName.

Guided by these FDs, we can decompose \( U \) into BCNF:

- book(bookIsbn, bookTitle, authContact),
- author(bookIsbn, bookAuthor),
- chapter(bookIsbn, chapNum, chapName),
- section(bookIsbn, chapNum, secNum, secName)

Although in the relational context algorithms have been
developed for computing a minimum cover for a set of
FDs [4, 16, 20], they cannot be used in our context since the
FDs must be computed from the XML keys \( \Sigma \) via the trans-
formation \( \sigma \), instead of being provided as input for those
relational algorithms. Furthermore, relational FDs are not
capable of expressing XML keys and vice versa.
Propagation of other XML constraints. XML Schema supports keys and foreign keys. Although it is tempting to develop algorithms to compute the propagation of both keys and foreign keys, we have the following negative result:

**Theorem 3.2:** The propagation problem for XML keys and foreign keys is undecidable for any transformation language that can express identity mapping.

The “identity” mapping is one in which the XML representation of relations is mapped to the same relations (in our language this corresponds to a small class of transformations defined with paths of length 3). The undecidability result is established by reduction from implication of relational keys and foreign keys, which is undecidable [15] (see [11] for a reduction). Because of this we restrict our attention to the propagation of XML keys.

4 Checking Key Propagation

Checking key propagation is nontrivial for a number of reasons: First, XML data is semistructured in nature, which complicates the analysis of key propagation by the presence of null values. Second, XML keys which are not in \( \Sigma \) but are consequences of \( \Sigma \) may yield FDs on a relational view. Thus key propagation involves XML key implication. Third, XML data is hierarchically structured and thus XML keys are relative in their general form – they hold on a sub-document. However, its relational view collapses the hierarchical structures into a flat table and thus FDs are “absolute” – they hold on the entire relational view. Thus one needs to derive a unique identification of a sub-document from a set of relative keys.

Before presenting our polynomial-time algorithm for checking XML key propagation (Algorithm propagation), we first discuss the notion of a “keyed” node and the implication of XML keys.

**Transitive set of XML keys.** To uniquely identify a node within the entire document we need a set of XML keys identifying unique contexts up to the root. To formalize this, we use the following notion [7]: \((Q_1, (Q'_1, S_1))\) immediately precedes \((Q_2, (Q'_2, S_2))\) if \(Q_2 = Q_1/Q'_1\). The precedence relation is the transitive closure of the immediately precedes relation. A set \( \Sigma \) of keys is transitive if for any relative key \((Q_1, (Q'_1, S_1))\) in \( \Sigma \) there is an absolute key \((e, (Q'_2, S_2))\) in \( \Sigma \) which precedes \((Q_1, (Q'_1, S_1))\). We say that a node is keyed if there exists a transitive set of keys to uniquely identify the node.

**Example 4.1:** The set \( \{K_{S_1}, KS_2\} \) is transitive since any chapter in the document can be identified by providing \#isbn of a book and \#number of a chapter. Thus every chapter node is keyed. In contrast, \( \{K_{S_2}\} \) is not transitive since with it alone there is no way to uniquely identify a book in the document, which is necessary before identifying a chapter of that book.

**Implication of XML keys.** One aspect of key propagation is to determine whether an XML key \( \varphi \) must hold provided that a set \( \Sigma \) of XML keys holds, denoted by \( \Sigma \models \varphi \). In other words, \( \Sigma \models \varphi \) iff for any XML tree \( T \), \( T \) satisfies \( \varphi \) as long as \( T \) satisfies all the keys in \( \Sigma \). An algorithm for implication analysis, implication, can be found in [11]. The algorithm takes as input a set \( \Sigma \) and \( \varphi \) of XML keys of \( K_a \) and returns true iff \( \Sigma \models \varphi \). It is based on a set of inference rules that, along the same lines as the Armstrong’s Axioms for implication of FDs in relational databases, allows one to derive key implication systematically. One example of the rules is target-to-context: if \((Q, (Q_1/Q_2, S))\) is a key then \((Q/Q_1, (Q_2, S))\). Intuitively, the rule states that if \( S \) can uniquely identify a set \( N \) of nodes in the entire tree \( T \), then it can also identify nodes of \( N \) in any subtree of \( T \); observe that for any nodes \( n \in [Q] \) and \( n' \in n[Q_1] \), the subtree rooted at \( n' \) is a subtree of the one rooted at \( n \). Another example of a trivial rule is epsilon: for any path \( Q \), it is true that \((Q, (\epsilon, \{\})\). Intuitively, it states that any subtree has a unique root node. Algorithm implication determines whether or not \( \Sigma \models \varphi \) in \( O(|\Sigma|^2 |\varphi|^2) \) time, where \( |\Sigma| \) and \(|\varphi|\) are the sizes of \( \Sigma \) and \( \varphi \).

**Table tree.** Algorithm Algorithm uses the tree representation of a transformation to bridge the gap between XML keys and the FD \( \phi \) to be checked. Without loss of generality, assume that \( \phi \) is of the form \( Y \rightarrow l \) with \( l \in attr(R) \) and \( Y \subseteq attr(R) \), and that for the relation \( R \), \( R[\alpha](R) \) is \( \{i : value(x_i) \mid i \in [1, n]\} \) along with a set \( X \) of variables and mappings \( x \leftarrow y/P \) for each \( x \in X \). In the table tree \( T_R \) representing \( R\[\alpha](R) \), any variable \( x \) in \( X \) has a unique node corresponding to it, referred to as the \( x \)-node. In particular, the \( x \)-node is the root of \( T_R \). Observe that for any \( x, y \in X \), if the \( x \)-node is a descendant of the \( y \)-node in \( T_R \), then there is a unique path in \( T_R \) from the \( y \)-node to \( x \)-node, which is a path expression. We denote the path by \( P(y, x) \), which exists only if there are variables \( x_1, \ldots, x_k \) in \( X \) such that \( x_1 = y \) and \( x_k = x \), and for each \( i \in [1, k-1] \), \( x_{i+1} \leftarrow x_i/P_i \) is a mapping in \( R \). We use \( \text{descendants}(y) \) to denote the set of all the variables that are descendants of \( y \); we define \( \text{ancestors}(y) \) similarly. In particular, if \( x \) is specified with \( x \leftarrow y/P \) then the variable \( y \) is called the parent of \( x \), denoted by \( \text{parent}(x) \). Referring to Fig. 3 (b), for example, \( x_r \) is the parent of \( Z_0 \), and \( P(x_r Z, c) \) is //book/chapter.

**Algorithm.** The intuition behind Algorithm propagation is as follows. Given an FD \( \phi \equiv Y \rightarrow l \) on \( R \), assume that \( l \) is specified with \( \text{value}(x) \), and that the table tree representing \( R[\alpha](R) \) is \( T_R \). Then \( \Sigma \models \phi \) iff (1) either \( \phi \) is trivial, that is, \( l \in Y \), or there exists an ancestor \( \text{target} \) of \( x \) in \( T_R \) such that \( \text{target} \) is keyed with fields of \( Y \) and moreover, \( x \) is unique under \( \text{target} \); that is, there is a set of transitive keys that uniquely identifies \( \text{target} \) with only those attributes which define
Algorithm propagation

Input: XML keys $\Sigma$, FD $\phi = Y \rightarrow l$ over $R$, and Rule($R$) in transformation $\sigma$, in which $l : value(x)$.
Output: true iff $\Sigma \models \exists R : \phi$.

1. ancestor[$x$] := nil;
2. $w$ := $x$;
3. while $w \neq x$, do
4. $w$ := parent($w$);
5. ancestor[$y$] := $w$ :: ancestor[$x$];
6. Ycheck := $Y$ := \{ $l$ \};
7. if $l \in Y$
8. then keyFound := true;
9. else keyFound := false;
10. context := $x$;
11. while ancestor[$x$] \neq nil do
12. target := head(ancestor[$x$]);
13. $S$ := \{ $\theta$ \mid $l' \in Y$, $l' : value(y) \in$ Rule($R$), $y$ \leftarrow target/$\theta$ is a variable mapping \} 
14. if not keyFound
15. then if implication($\Sigma$, (P($x$, context), (P(context, target), $S$))
16. then cont exit := target;
17. if implication($\Sigma$, (P($x$, target),
18. P(target, $x$), \{ \}));
then keyFound := true;
19. if exist(P($x$, target), $S$)
then $L$ := \{ $l'$ \mid $l' \in Y$, $l' : value(y) \in$ Rule($R$), $y$ \leftarrow target/$\theta$ is a variable mapping \}
20. Ycheck := Ycheck $\cup L$;
21. ancestor[$x$] := fail(ancestor[$x$]);
22. return keyFound and (Ycheck $\cup$ \{});

function exist(Q, S)
Input: Q: path expression; S: a set of attributes.
Output: true iff for all $l \in S$ and $n \in \{Q\}$, n $\theta$ exists.
1. $X$ := $S$;
2. for each key $\phi$ := (Q1, (Q1', S1)) in $\Sigma$ do
3. if $Q \subseteq Q1$/$Q1'$
4. then $X := X \cup S1$;
5. return (X $\subseteq$ \{ \});

Figure 5. XML key propagation algorithm

fields of $Y$, and $\Sigma \models (P(x_r, target), (P(target, x), \{ \}))$; (2) every field of $Y$ is defined with an attribute of some ancestor of $x$ that is required to exist. The first condition asserts that for any $R$ tuples $t_1$ and $t_2$, if they agree on their $Y$ fields and do not contain null, then they agree on their $l$ fields. The second condition excludes the possibility that in some $R$ tuple $t$, the $l$ field is defined while some of their $Y$ fields are null.

Putting everything together, Algorithm propagation is shown in Fig. 5. The algorithm first computes the list of all the ancestors of $x$ (Lines 1 to 5); it then traverses the table-tree $T_R$ top-down along the ancestor path from the root $x_r$ to $x$ (Lines 11 to 22), and for each ancestor $target$ in this path, checks if $target$ is keyed (Lines 15). The central part of the algorithm is to check whether there is a set of transitive keys for $target$. To do so, it uses variable context to keep track of the closest ancestor for which a key has been found, and collects the attributes of $target$ that populate fields in $Y$ in a set $S$. Thus $target$ is keyed iff $\Sigma \models (P(x_r, context), (P(context, target), S))$, i.e., $S$ is a key of $target$ relative to its closest ancestor with a key. XML key implication is checked by invoking Algorithm implication mentioned above. If it holds, the algorithm moves context down to target (Line 16; the correctness of this step is ensured by the target-to-context rule given above); then, it sets the Boolean flag key $Fo$ and to true if $x$ is unique under target (Line 17). To ensure that all the fields of $Y$ are defined with attributes of ancestors of $x$ that are required to exist, it uses a variable Ycheck (with an initial value of $Y$ := \{ $l$ \}) and removes from Ycheck the field names that correspond to the set $S$ of attributes (Lines 19 to 21). The algorithm returns true iff key $Fo$ and is true and Ycheck becomes empty, i.e., the two conditions given above are satisfied.

Example 4.2: To illustrate the algorithm, recall the transformation $\sigma$ of Example 2.4 and the set $\Sigma$ of XML keys of Example 2.1. Consider FD: isbn $\rightarrow$ contact over relation book defined by Rule(book), which is depicted in Fig. 3 (a). Note that the field contact in the FD is specified with variable $x_4$. Given $\Sigma$, $\sigma$ and the FD, the algorithm computes the ancestors of $x_4$, which consists of $x_r$, $x_b$ and $x_a$. Then, it first checks if $x_r$ is keyed by inspecting $\Sigma \models (e, \{ \})$. Since this holds by the epsilon rule given above, the algorithm then checks whether $x_b$ is keyed by inspecting $\Sigma \models (//book\{ @isbn \})$. Since this is also true, the algorithm proceeds to check whether $x_a$ is unique under $x_b$, i.e., whether $\Sigma \models (//book\{ (author/contact, \{ \}) \})$. This is also the case. In addition, the field isbn in the FD is defined in terms of an attribute of $x_b$ that is required to exist. That is, by the semantics of keys, $//book\{ @isbn \}$ requires every book element to have an @isbn attribute. Thus the algorithm concludes that the FD is derived from $\Sigma$ via $\sigma$ and returns true.

Next, let us consider Rule(section) of Example 2.4, represented by the table tree of Fig. 3 (b), and let $\phi$ be an FD: inChapt, number $\rightarrow$ name over relation section. After successfully verifying that $x_r$ is keyed, the algorithm checks whether its next ancestor is keyed, i.e., whether $\Sigma \models (//book/chapter, \{ @number \})$. This fails. Thus it attempts to verify another key relative to the root: $\Sigma \models (//book/chapter/section, \{ @number \})$, which fails again. At this point the algorithm concludes that the FD cannot be derived from $\Sigma$ and returns false. □
respectively (see [11] for details as well as for a proof of correctness of the algorithm).

5 Computing Minimum Cover

In this section we present two algorithms for finding a minimum cover for FDs propagated from XML keys. The first algorithm is a direct generalization of Algorithm propagation of Fig. 5, and always takes exponential time. We use this naive algorithm to illustrate the difficulties in connection with finding a minimum cover. The second algorithm takes polynomial time in the size of input, by reducing the number of FDs generated in the following way: a new FD is inserted in the resulting set only if it cannot be implied from the FDs already generated, using the inference rules for FDs. To the best of our knowledge, this is the first effective algorithm for finding a minimum cover for FDs propagated from XML keys.

A Naive Algorithm. Algorithm propagation given in the last section allows us to check XML key propagation. Thus a naive algorithm for finding a minimum cover is to generate each possible FD on \( U \), check whether or not it is in \( F^+ \), the set of all the FDs mapped from the XML keys, using Algorithm propagation, and then eliminate both extraneous attributes and redundant FDs from \( F^+ \) using standard relational database techniques; this yields a minimum cover \( F_m \) for \( F^+ \). The algorithm, Algorithm naive, can be found in [11]. It takes exponential time in the size of \( U \) for any input since it computes all possible FDs on \( U \).

It should be mentioned that the function invoked by the algorithm for eliminating redundancy, Function minimize, taken from below [4], takes quadratic time in the size of its input FDs, since FD implication can be checked in linear time using the Armstrong’s Axioms; but when invoked in naive, the set of input FDs is exponentially large.

```
function minimize(F)
Input: F: a set of FDs.
Output: A non-redundant cover of F.
1. for each \( (Y \rightarrow l) \in F \) do */ eliminate extra attributes */
2. for each \( l' \in Y \) do
3. if \( l \neq l' \)
4. then \( Y' := Y - \{l'\}; \)
5. \( G := F; \) /* eliminate redundant FDs */
6. for each \( \phi \in F \) do
7. if \( (G - \{\phi\}) \models \phi \)
8. then \( G := G - \{\phi\}; \)
9. return G;
```

Obviously, Algorithm naive is too expensive to be practical. The problem is that it needs to compute \( F^+ \), which is exponential in the size of \( U \) even with trivial FDs removed. This observation motivates us to develop an algorithm that directly finds \( F_m \) without computing \( F^+ \).

A Polynomial-Time Algorithm. We next present a more efficient algorithm for finding a minimum cover for all the propagated FDs. The algorithm takes \( O(mn) \) time, where \( m \) and \( n \) are the sizes of XML keys \( \Sigma \), and the transformation \( \sigma \), respectively. The algorithm works as follows. Recall that the transformation \( \text{tab}\text{le}(U) \) can be depicted as a table tree \( T \), in which each variable \( x \) in the set \( X \) of \( \text{tab}\text{le}(U) \) is represented by a unique node, referred to as the \( x \)-node. The algorithm traverses \( T \) top-down starting from the root of \( T \), \( x_r \), and generates a set \( F \) of FDs that is a cover of \( F^+ \), i.e., a superset of \( F_m \). More specifically, at each \( x \)-node encountered, it expands \( F \) by including certain FDs propagated from \( \Sigma \). It then removes redundant FDs from \( F \) to produce a minimum cover \( F_m \).

The obvious question is what new FDs are added at each \( x \)-node. As in Algorithm propagation, at each \( x \)-node a new FD \( Y \rightarrow l \) is included into \( F \) only if (1) \( x \) is key with a set of attributes that define the fields in \( Y \); (2) the field \( l \) is defined by the value of a node \( y \) and \( y \) is unique under \( x \).

**Example 5.1:** Recall the universal relation \( U \) defined by the transformation \( \sigma \) of Example 3.1, the table tree depicted in Fig. 4, and the set \( \Sigma \) of XML keys of Example 2.1. An FD derived from \( \Sigma \) at the \( z \)-node is \( \text{bok}\text{ib}\text{m} \), \( \text{chap}\text{Num} \), \( \text{sec}\text{Num} \rightarrow \text{sec}\text{Name} \). The left-hand side of the FD corresponds to a transitive set of keys for the \( z \)-node consisted of a section \( @\text{number} \) which is a chapter \( @\text{number} \) and a book \( @\text{isbn} \), which are a key of \( z \)-node’s ancestor \( y_c \). The right-hand side of the FD is defined by a node \( z_2 \) unique under \( z_2 \) by \( K \Sigma_0 \) in \( \Sigma \). Thus the key for the \( z_2 \)-node actually consists of the key of its ancestor \( y_c \) as well as a key for section \( @\text{number} \) relative to \( y_c \).

Critical to the performance of the algorithm is to minimize the number of FDs added at each \( x \)-node while ensuring that no FDs in \( F_m \) are missed. This is done in two ways: First, we reduce our search for candidate FDs to those whose left-hand side corresponds to attributes of keys in \( \Sigma \). Second, we observe that an ancestor \( \text{target} \) of an \( x \)-node may have several keys, but that in creating a transitive key for \( x \) only one of them needs to be selected as long as the following property is enforced: for any two transitive keys \( K_1 \) and \( K_2 \) of the \( x \)-node, \( F \) includes \( Y_1 \rightarrow l \) for each \( l \in Y_2 \) and \( Y_2 

\rightarrow \) \( l' \) for each \( l' \in Y_1 \), where \( Y_1, Y_2 \) are sets of \( U \) fields defined by \( K_1 \) and \( K_2 \), respectively. Given this, \( Y_1 \) and \( Y_2 \) are equivalent by Armstrong’s Axioms.

There is a subtlety caused by the troublesome null value. Let \( K_2 \) be a transitive key for an \( x \)-node, \( K_1 \) be a transitive key for an ancestor \( y \) of \( x \), \( Y_1 \) and \( Y_2 \) be the sets of \( U \) fields defined by \( K_1 \) and \( K_2 \), respectively, and \( Z \) be another set of \( U \) fields. Then the following is a rule for propagating \( F \): if \( (Y_1 \cup Z \rightarrow l) \) is in \( F \) and \( l \) is a \( U \) field defined by a descendant \( z \) of \( x \), then \( (Y_2 \cup Z \rightarrow l) \) should also be included in \( F \). The intuition behind this rule is that a key for \( x \) is also a key for its ancestor \( y \), provided that the existence of \( x \) under \( y \) is assured. This is because...
there is a unique ancestor $y$ of $x$ in a tree that connects to $x$ via the path $P(y, x)$. Thus, provided the existence of $x$ under $y$, we have $Y_2 \rightarrow l'$ for any $l' \in Y_1$. As a result, if $Y_1 \cup Z \rightarrow l$ then $Y_2 \cup Z \rightarrow l$, by the transitivity of FD implication. The existence of $x$ under $y$ is ensured by the existence of $z$: if $Y_2 \cup Z \rightarrow l$, then by the definition of FDs, $z$ must exist under $y$; hence, $x$ must be on the path from $y$ to $z$, i.e., $x$ exists. It is worth remarking that when $l$ is not defined by a descendant of $x$, the rule may not be sound since $Y_2 \rightarrow l'$ may not hold for an $l' \in Y_1$; more specifically, $Y_2$ consists of null if $x$ does not exist under $y$, while $l'$ may not be a key. As an example, consider two transitive keys at a node $x$: $K_1 = \{(Q_1, \{A_1\}), (Q_2, \{A_2\})\}$ and $K_2 = \{(Q_1/Q_2, \{A_1\})\}$, with each attribute $A_i$ populating a field $l_i$ in $U$. Note that $\{(Q_1, \{A_1\})\}$ is a key of the ancestor $y$ of $x$ that connects to $x$ via the path $Q_2$. Consider the following FDs: $f_1 = (l_1, l_2) \rightarrow l_3$, $f_2 = l_3 \rightarrow l_4$, and $f_3 = l_2 \rightarrow l_3$. Although $f_1$ and $f_2$ are indeed propagated from $K_1$ and $K_2$, $f_3$ is not. This is because the existence of attribute $A_1$ of node $y$ in $\{Q_1\}$ does not guarantee the existence of attribute $A_3$ of node $y$ in $\{Q_2\}$; therefore, $l_3$ can have a non-null value even when $l_3$ is null, violating the FD. Thus $f_3$ should not be included in $F$. However, if $F$ contains $l_1 \rightarrow l_4$ for some $l_4$ defined with a descendant of $x$, then $F$ should also include $l_3 \rightarrow l_4$.

To keep track of the information needed to generate FDs at each $x$-node, we associate the following with each variable $x$ in $\text{Rule}(U)$:

- $\text{keys}[x]$: a list of sets of $U$ fields, each set mapped from a transitive key of the $x$-node;
- $\text{keyAnce}[x]$: a set of sets of $U$ fields, each mapped from a transitive key of an ancestor of $x$;
- $\text{desc}[x]$: the set of all descendants of the $x$-node;
- $\text{unique}[x]$: the set of all the unique descendants of the $x$-node;
- $\text{att}[x]$: the set of attributes of the $x$-node;
- $\text{ancestor}[x]$: the list of all the ancestors of $x$ starting from the root;

Note that $\text{unique}[x]$ is a subset of $\text{desc}[x]$, and $\text{att}[x]$ is a subset of $\text{unique}[x]$ since any node in an XML tree has at most one attribute labeled with a particular name.

Using this notation, at each $x$-node, we expand the cover $F$ of FDs as follows: First, for each $(Q, (Q', S))$ in $\Sigma$ we compute $K$, the set of fields of $U$ defined by attributes in $S$. We check whether $S$ is contained in $\text{att}[x]$ and whether every attribute of $S$ defines a $U$ field (by comparing the cardinalities of $S$ and $K$). If it is the case then we traverse the ancestor path of $x$ starting from the root. We find the first ancestor target of $x$ that is keyed, and check whether $S$ is a key for the $x$-node relative to target using Algorithm implication. If these conditions are met we construct a transitive key $K'$ for $x$ by combining $S$ with an arbitrary transitive key for the ancestor target of the $x$-node, which is in $\text{keys}[\text{target}]$. We increment $\text{keys}[x]$ by adding

---

**Algorithm minimumCover**

**Input:** XML keys $\Sigma$, a universal relation $U$ defined by Rule($U$) along with a set $X$ of variables.

**Output:** a minimum cover $F_m$ for all FDs on $U$ propagated from $\Sigma$.

1. for each $x$ in $X$ do
2. $\text{keys}[x] := \text{nil}; \text{keyAnce}[x] := \{\}$;
3. $\text{unique}[x] := \{\}; \text{att}[x] := \{\};
4. for each $y$ in descendant($x$) do
5. if $l : \text{value}(y)$ is in Rule($U$) then
6. $\text{desc}[x] := \text{desc}[x] \cup \{l\};$
7. if implication($\Sigma$, ($P(x, x), (P(x, y), \{\})$))
8. then $\text{unique}[x] := \text{unique}[x] \cup \{l\}$;
9. if $y \leftarrow x/\text{att}'$ is a variable mapping then
10. $\text{att}[x] := \text{att}[x] \cup \{\text{att}'\};$
11. allVars := $x \cup \text{nil}; \text{keys}[x] := \{\text{nil};$
12. keyAnce[$x] := \{\}; \text{ancestor}[x] := \{\}; F := \{\};$
13. for each $l$ in $\text{unique}[x]$ do $F := F \cup \{l \rightarrow l\}$;
14. for each $x$ in $\text{children}(x)$ do genFDs($x$);
15. while allVars $\neq \text{nil}$ do
16. $x := \text{head}(\text{allVars})$;
17. for each $R$ in $\text{keyAnce}[x]$ do
18. for each $Y \rightarrow l$ in $F$ do
19. if $l \in \text{desc}[x]$ and $R$ is subset of $Y$ then
20. for each $K$ in $\text{keys}[x]$ do
21. $F := F \cup \{K \cup (Y \rightarrow R) \rightarrow l\}$;
22. allVars := tail(allVars);
23. $F_m := \text{minimize}(F)$; return $F_m$;

**procedure genFDs ($x$)**

**Input:** $x$: a variable in Rule($U$).

**Output:** expanded $F$.

1. allVars := $x$ := allVars; $w$ := parent($x$);
2. ancestor[$x] := ancestor[$w] + ($w$ := nil);
3. while keys[$w$] := nil do $w$ := parent($w$);
4. keyAnce[$x] := keyAnce[$w] \cup \text{head}(\text{keys[$w$]})$;
5. for each $(Q, (Q', S))$ in $\Sigma$ do
6. $K := \{l \mid \forall l' \in S, l : \text{value}(y) \in \text{Rule}(U)$,
7. $y \leftarrow x/\text{att}'$ is a variable mapping$\}$;
8. if $S \subseteq \text{att}[x]$ and $|K| = |S|$ then
9. traverse[$x$] := ancestor[$x$]\text{keyAnce} := false
10. while traverse[$x$] $\neq \text{nil}$ and not keyAnce do
11. target := head(traverse[$x$]);
12. if $\text{keys[target]} \neq \text{nil}$ and implication
13. ($\Sigma$, ($P(x, target), (P(target, x), S)\})$
14. then keyAnce := true
15. else traverse[$x$] := tail(traverse[$x$]);
16. if keyAnce then
17. $K := K \cup \text{head}(\text{keys[target]})$;
18. keys[$x$] := $K := keys[x]$;
19. keyAnce[$x$] := keyAnce[$x$] $\cup \text{head}(\text{keys[target]})$;
20. for each $l$ in $\text{unique}[x] = K$ do
21. $F := F \cup \{K \rightarrow l\}$;
22. for each $y$ in $\text{children}(x)$ do genFDs($y$);

Figure 6. Computing minimum cover
this transitive key, and insert \textit{keys[target]} in \textit{keyAnce[x]}. Second, we expand \textit{F} by including \textit{Y \rightarrow l} for each \textit{l} in \textit{unique[x]} (excluding \textit{Y}), where \textit{Y} is a set of \textit{U} fields defined by \textit{K'}. That is, the transitive key of the \textit{x}-node determines the unique descendants of \textit{x}. After the set \textit{F} is computed by traversing all variables \textit{x} in \textit{rule(U)}, it is expanded by applying transitivity on keys in \textit{keyAnce[x]}, which includes a key of \textit{x}'s closest keyed ancestor. That is, if \textit{K_1} \in \textit{keyAnce[x]}, we inspect each \textit{Y \rightarrow l} in \textit{F}, checking if \textit{K_1} is a subset of \textit{Y}, i.e., whether there exists \textit{Z} such that \textit{Y = K_1 \cup Z}. If this is the case and \textit{l} \in \textit{dese[x]}, then for each \textit{K_2} in \textit{keys[x]} we add \textit{(K_2 \cup Z) \rightarrow l} to \textit{F}. One can show that the sizes of \textit{keys[x]} and \textit{keyAnce[x]} are quadratic in the size of \textit{U}, \textit{unique[x]} is bounded by the size of \textit{rule(U)}, and the set \textit{X} is no larger than the size of \textit{rule(U)}; thus the set \textit{F} is bounded by \textit{m^3n^5}, i.e., the size (thus the cardinality) of \textit{F} is at most \textit{O(m^4n^3)}, a polynomial in the input size.

\textbf{Algorithm.} Based on these observations, we show Algorithm \textit{minimumCover} in Fig. 6. After computing \textit{dese[x]}, \textit{att[x]}, \textit{unique[x]} and initializing \textit{keys[x]}, and \textit{keyAnce[x]} for each variable \textit{x} in \textit{rule(U)} (Lines 1 to 10), the algorithm initializes these variables for the root node (Lines 11, 12), and inserts in \textit{F} FDS of the form \textit{Y \rightarrow l} for each unique field under the root (Line 13). It then invokes a recursive procedure \textit{genFDS} to process the children of the root node (Line 14). Procedure \textit{genFDS} expands \textit{F} given an input \textit{x}-node as described above, and recursively processes the children of the \textit{x}-node. After \textit{F} is computed, Algorithm \textit{minimumCover} expands it by applying the transitivity rule (Lines 15 to 22) and invokes function \textit{minimize} given in the last section to eliminate redundant FDS from \textit{F}, and thus yields a minimum cover \textit{F_m} (Lines 23). The correctness of the algorithm is established in [11].

\textbf{Example 5.2:} Given the transformation \textit{\sigma} of Example 3.1 and the set \textit{\Sigma} of XML keys of Example 2.1, Algorithm \textit{minimumCover} returns the FDS given in Example 3.1, which are a minimum cover for all the FDS propagated from \textit{\Sigma} via \textit{\sigma}. Specifically, the algorithm traverses the table tree of Fig. 4 (a) top-down starting at the root. At node \textit{x}_, two FDSs are generated: one is \textit{bookIsbn \rightarrow bookTitle}, and the other is \textit{bookIsbn \rightarrow authContact}. Here \textit{keys[x_3] = \{[@isbn]\}}. At node \textit{y}_, \textit{FD} \textit{bookIsbn, chapNum \rightarrow chapName} is included in \textit{F} and \textit{keys[y_3]} is changed to \textit{\{[@isbn, @number]\}}, which is constructed by combining \textit{@number}, a key of \textit{y} relative to \textit{x}, and the key in \textit{keys[x_3]}. Similarly, at node \textit{z}_, \textit{FD} \textit{bookIsbn, chapNum, secNum \rightarrow secName} is inserted into \textit{F}. No FDs are generated at any other nodes. \hfill \Box

The complexity of the algorithm is \textit{O(m^3n^5)} time, where \textit{m} and \textit{n} are the sizes of XML keys \textit{\Sigma} and table tree \textit{T_R}, respectively (see [11] for details). Since \textit{\Sigma} and \textit{rule(U)} are usually small, this algorithm is efficient in practice. The experimental results of the next section also show that it substantially outperforms Algorithm \textit{naive}.

A final remark is that, although one can generalize Algorithm \textit{minimumCover} to check XML key propagation instead of using Algorithm \textit{propagation}, there are good reasons for not doing so. The complexity of Algorithm \textit{minimumCover} is much higher than that of Algorithm \textit{propagation} (\textit{O(m^5n^3)} vs. \textit{O(m^4n^3)}). In short, Algorithm \textit{propagation} is best used to inspect a predefined relational schema, whereas Algorithm \textit{minimumCover} helps normalize a universal relation at the early stage of relational design.

\section{Experimental Study}

The various algorithms presented in this paper have been implemented, and a number of experiments performed. The results of these experiments show that despite their \textit{O(m^2n^3)} and \textit{O(m^3n^3)} worst-case performance, both Algorithms \textit{propagation} and \textit{minimumCover} work well in practice: they take merely a few seconds even given large transformation and XML keys. For computing minimum cover, Algorithm \textit{minimumCover} is several orders of magnitude faster than Algorithm \textit{naive}, and for checking key propagation Algorithm \textit{propagation} significantly outperforms the generalization of Algorithm \textit{minimumCover}. Our results also reveal that Algorithm \textit{minimumCover} is more sensitive to the number of XML keys than to the size of the transformation. This is nice since in many applications the number of keys does not change frequently, whereas a relational schema may define tables with a variety of different arities (number of fields). Our results also show that Algorithm \textit{propagation} has a surprisingly low sensitivity to the size of the transformation, and that its execution time grows linearly with the size of XML keys.

To perform these experiments, we synthetically generated transformations and XML keys based on the number of fields in a relation, the depth of a table-tree, and the number of XML keys. All experiments were conducted on the same 1.6GHz Pentium 4 machine with 512MB memory. The operating system is Linux RedHat v7.1 and the program was implemented in C++.

The first experiment evaluates the performance of the two algorithms for computing minimum cover (see Fig. 7(a)). These results tell us the following. First, the average complexity of Algorithm \textit{minimumCover} in practice is much better than its \textit{O(m^3n^3)} worst-case complexity. Consider, for example, the execution time of the algorithm for depth = 10 and key = 10. When the number of fields is increased (which corresponds roughly to increasing the size of the transformation), the execution time grows in the power of two in average instead.
of in the power of six. Second, the algorithm needs less than 35 seconds for 200 fields, and a little over 2 minutes even for 500 fields. Since in most applications the number of fields in a relation is much less than 500, we can say that Algorithm \textit{minimumCover} performs well in practice. Third, the performance of Algorithm \textit{minimumCover} is much better than Algorithm \textit{naive}. For example, when the number of fields is incremented by 5, the execution time of \textit{minimumCover} at most doubles, while for \textit{naive} it grows almost two-hundred-fold.

We next consider checking XML key propagation. An algorithm for doing so, Algorithm \textit{propagation}, was presented in Section 4. An alternative algorithm can also be developed by means of Algorithm \textit{minimumCover} as follows: Given a transformation $\sigma$, a set of keys $\Sigma$, and an FD $\phi = Y \rightarrow I$, the algorithm first invokes \textit{minimumCover}($\Sigma, \sigma$) to compute a minimum cover $F_m$ of all the FDs propagated; it then checks whether or not $F_m$ implies $\phi$ using relational FD implication, and whether all the fields in $Y$ are guaranteed to have a non-null value when $I$ is not null. It returns \textit{true} if these conditions are met. In what follows, we refer to this generalized algorithm as \textit{GminimumCover} since the performance is roughly comparable to the original algorithm.

Our second experiment serves two purposes: to compare the effectiveness of these two algorithms for checking key propagation, and to study the impact of the depth of table-tree (depth) on the performance of Algorithms \textit{propagation} and \textit{GminimumCover}. Fig. 7(b) depicts the execution time of these algorithms for field = 15 and keys = 10 with depth varying from 2 to 15. (These parameters were chosen based on the average tree depth found in real XML data [9].) The results in Fig. 7(b) reveal the following. First, Algorithm \textit{propagation} works well in practice: it takes merely 0.05 second even when the table tree is as deep as 15. Second, these algorithms are rather insensitive to the change to depth. Third, \textit{propagation} is much faster than \textit{GminimumCover} for checking key propagation, as expected. Although the actual execution times of the algorithms are quite different, the ratios of increase when the depth of the table-tree grows are similar. This is because in both algorithms the depth determines how many times Algorithm \textit{implication} is invoked, and because the complexity of Algorithm \textit{implication} is a function of the size of the XML keys, which grows when the depth of the table tree gets larger.

Our third experiment demonstrates how the number of XML keys (keys) influences the performance of Algorithms \textit{propagation} and \textit{GminimumCover} when checking key propagation. The results (Fig. 7(c)) show that increasing the number of keys has a bigger impact on Algorithm \textit{GminimumCover} than on \textit{propagation},

\textbf{Figure 7. Experimental results}
in which the growth of the execution time is almost linear. In fact, additional experiments tell us that for depth = 10 and keys = 50, Algorithm GminimumCover runs in under 2 minutes for 200 fields, but when increasing the number of keys to 100, its execution time is over 4 minutes for relations with 150 fields. In contrast, Algorithm propagation runs in both settings in less than 5 seconds. In addition, for 1000 fields, which is the maximum number of fields allowed by Oracle [22], the execution time of propagation is 85 seconds on average for 50 keys, and 142 seconds for 100 keys.

A closer look at Algorithm propagation reveals that the constant ratio of increase is based on the time needed for executing calls to Algorithm implication. That is, if the depth of the table-tree is fixed, the number of calls is roughly the same for the whole experiment; the increase in running time is based on the the performance of Algorithm implication, which depends on the size of the XML keys. The performance of implication also has an impact on the Algorithm GminimumCover. However, the number of keys has a bigger influence in this algorithm because for each node in the table-tree all the keys are analyzed. Also, by increasing the number of XML keys, the number of FDs in the resulting set is likely to grow, increasing the execution time for eliminating redundant FDs by calling minimize.

7 Conclusion

We have proposed a framework for refining the relational design of XML storage based on XML key propagation. For this purpose we have developed algorithms for checking whether a functional dependency is propagated from XML keys, and for finding a minimum cover for all functional dependencies propagated from XML keys, along with complexity results in connection with XML constraint propagation. Our experimental results show that these algorithms are efficient and effective in practice. These algorithms can be generalized and incorporated into relational storage techniques published in the literature (e.g. [25, 26, 22]). Our results are also useful in optimizing queries and in understanding XML to XML transformations.

Topics for future work include studying the propagation of other forms of integrity constraints, and re-investigating constraint propagation in the presence of types (e.g., XML Schema).

References