Efficient Discovery of XML Data Redundancies *

Cong Yu
Department of EECS
University of Michigan
congy@eecs.umich.edu

H. V. Jagadish
Department of EECS
University of Michigan
jag@eecs.umich.edu

ABSTRACT

As XML becomes widely used, dealing with redundancies in XML data has become an increasingly important issue. Redundantly stored information can lead not just to a higher data storage cost, but also to increased costs for data transfer and data manipulation. Furthermore, such data redundancies can lead to potential update anomalies, rendering the database inconsistent.

One way to avoid data redundancies is to employ good schema design based on known functional dependencies. In fact, several recent studies have focused on defining the notion of XML Functional Dependencies (XML FDs) to capture XML data redundancies. We observe further that XML databases are often "casually designed" and XML FDs may not be determined in advance. Under such circumstances, discovering XML data redundancies (in terms of FDs) from the data itself becomes necessary and is an integral part of the schema refinement process.

In this paper, we present the design and implementation of the first system, *DiscoverXFD*, for efficient discovery of XML data redundancies. It employs a novel XML data structure and introduces a new class of partition based algorithms. DiscoverXFD can not only be used for the previous definitions of XML functional dependencies, but also for a more comprehensive notion we develop in this paper, capable of detecting redundancies involving set elements while maintaining clear semantics. Experimental evaluations using real life and benchmark datasets demonstrate that our system is practical and scales well with increasing data size.

1. INTRODUCTION

Redundant data takes up unnecessary storage, inflates data transfer cost, and can lead to update anomalies. A central goal of database design is to ensure that there are no unintended redundancies. As XML databases have become more common, good design of XML schema has become increasingly important, especially in scientific databases with complex structures. Furthermore, one of the benefits of XML (whether intended or not) is the ease of generating XML data: compared with relational data, XML data can be created by ordinary users (e.g. individual scientists) with minimal training in database schema design. Such casual design of XML schema is likely to lead to many data redundancies in the resulting XML databases[1]. Discovery of redundancies and schematic constraints based on the data will provide the critical first step for analyzing and refining such schemas.

The notion of functional dependency (FD) plays a central role in defining redundancies [8] in relational databases, and should play a correspondingly important role in XML databases as well. Although similar to their relational counterparts, redundancies in XML data have several distinct features due to the heterogeneous nature of XML data. In consequence, standard relational FD discovery algorithms are both inefficient and insufficient to find all XML FDs. (This is true whether we consider classic relational FD discovery algorithms such as [17], or more recent proposals such as Dep-Miner[16], TANE[13], and FUN[20]). In this paper, we develop a new algorithm *DiscoverXFD*, for efficient discovery of XML FDs and data redundancies.

The notion of XML functional dependency (XML FD), and the related notion of XML normal form, have recently become an important research topic. In [3], Arenas and Libkin adopted a tuple based approach and were the first to formally define XML FD and Normal Form. In [14, 24], the authors took a path based approach and built their XML FD notion in a fashion similar to the XML Key notion proposed in [5]. In this paper, we show that these XML FD notions are insufficient, and propose a *generalized tree tuple* based XML FD notion that can fully capture XML data redundancies with unambiguous semantics. Our algorithm, DiscoverXFD, can find all XML FDs in accordance with either of the previous definitions, or according to our new, more general, definition.

Consider the example XML document in Figure 1, which maintains information about books sold at various book stores within a book warehouse, grouped by states. Each store records its contact information and the books it is selling, and for each book, the ISBN, author, title, and price are maintained. Two intuitive constraints, which the example satisfies, are the following: two books with the same ISBN must have the same title and the same set of authors; and likewise, two books with the same set of authors and the same title must share the same ISBN. Both constraints cause some information in the XML document to become redundant (e.g., the title `DBMS` and the set of authors `Ramakrishnan` and `Gehrke` are stored multiple times for ISBN `0072465638`, and vice versa). One important characteristic that distinguishes such XML redundancies from their relational counterparts is the involvement of set elements: it is the *set of authors*, rather than an individual author, that are being compared and duplicated. This class of FDs is not covered by the definitions in [3] and [24]. A third constraint is equally interesting: for any two books sold at the same store chain (i.e., stores with the same name), if

*Supported in part by NSF under grant IIS-0438909.

Permission to copy without fee all or part of this material is granted provided that the copies are not made or distributed for direct commercial advantage, the VLDB copyright notice and the title of the publication and its date appear, and notice is given that copying is by permission of the Very Large Data Base Endowment. To copy otherwise, or to republish, to post on servers or to redistribute to lists, requires a fee and/or special permission from the publisher, ACM.
VLDB '06, September 12-15, 2006, Seoul, Korea.
Copyright 2006 VLDB Endowment, ACM 1-59593-385-9/06/09.

[1]Anecdotal examples include some large, heavily used community resources, such as PIR [1].

warehouse: Rcd
  state: SetOf Rcd
    name: str
    store: SetOf Rcd
      contact: Rcd
        name: str
        address: str
      book: SetOf Rcd
        isbn: str
        author: SetOf str
        title: str
        price: str

Figure 2: Example schema for Figure 1.

and a node key that uniquely identify it in T;
– $n_r \in N$ is the root node;
– $\mathcal{P}$ is a set of parent-child edges, there is exactly one $p = (n', n)$ in $\mathcal{P}$ for each $n \in N$ (except n_r), where $n' \in N, n \neq n'$, n' is called the parent node, n is called the child node;
– $\mathcal{V}$ is a set of value assignments, there is exactly one $v = (n, s)$ in $\mathcal{V}$ for each leaf node $n \in N$, where s is a value of simple type.

In Figure 1, node keys are assigned in pre-order traversal (gaps in the numbering indicate omitted elements), and we use "@key" to refer to the node keys. Parent-child edges are represented as directed lines between two data nodes (with arrow pointing to the child node). Value assignments are represented as equality between the node label and the value. We adopt the notion of conformance as defined in [25] and assume that all given data trees conform to their schemas.

A node (or data element) n_k is a *descendant* of another node n_1 if there exists a series of nodes n_i, such that $(n_i, n_{i+1}) \in \mathcal{P}$ for all $i \in [1, k\text{-}1]$. Similar to schema elements, n_k can also be addressed using a *path* expression, $path(n_k) = /e_1/.../e_k$, where e_i are labels of n_i for all $i \in [1, k]$, $n_1 = n_r$, and $(n_i, n_{i+1}) \in \mathcal{P}$ for all $i \in [1, k\text{-}1]$. Clearly, it is possible that two distinct data nodes will have the same path (e.g., node 11 and node 41). Furthermore, n_k is called *repeatable* if e_k corresponds to a set element in the schema. Finally, n_k is called a *direct descendant* of node n_a, if n_k is a descendant of n_a, $path(n_k) = .../e_a/e_1/.../e_{k-1}/e_k$, and e_i is not a set element for any $i \in [1, k\text{-}1]$. For example, node 21 (ISBN) is a direct descendant of node 20 (book), but not node 12 (store).

In considering data redundancy, it is also important to determine the equality between the "values" associated with two data nodes (as proposed in [5]), instead of comparing their "identities" (as represented by @key):

Definition 3 (Node-Value Equality). *Two data nodes n_1 of $T_1 = \langle N_1, \mathcal{P}_1, \mathcal{V}_1, n_{r1} \rangle$, and n_2 of $T_2 = \langle N_2, \mathcal{P}_2, \mathcal{V}_2, n_{r2} \rangle$ are* node-value equal (written as $n_1 =_{nv} n_2$) *iff:*
– n_1 and n_2 both exist and have the same label;
– There exists a set M of matching pairs: each pair $m = (n'_1, n'_2)$ indicates that $n'_1 =_{nv} n'_2$, where n'_1, n'_2 are child nodes of n_1, n_2, respectively. All child nodes of n_1 (n_2) participate in M and each child node participates in only one such pair.
– $(n_1, s) \in \mathcal{V}_1$ iff $(n_2, s) \in \mathcal{V}_2$, where s is a simple value.

Intuitively, two data nodes (e.g., node 30 and 50) are node-value equal iff the two subtrees rooted at the two nodes are identical without considering the order among sibling nodes. Because data nodes can also be addressed by paths, we define the path-value equality based on Definition 3:

Definition 4 (Path-Value Equality). *Given paths p_1 on $T_1 = \langle N_1, \mathcal{P}_1, \mathcal{V}_1, n_{r1} \rangle$, and p_2 on $T_2 = \langle N_2, \mathcal{P}_2, \mathcal{V}_2, n_{r2} \rangle$, p_1, p_2 are* path-value equal *(written as $T_1.p_1 =_{pv} T_2.p_2$) iff: for T_1, for each $n_1, n_1 \in N_1, path(n_1) = p_1$, there exists corresponding $n_2, n_2 \in N_2, path(n_2) = p_2$, $n_1 =_{nv} n_2$, vice versa for T_2, and the correspondence is one-on-one.*

Value equality between two paths is complicated by the fact that a single path can match multiple nodes in a data tree. Definition 4 requires that, for two paths to be considered value equal, each node that is pointed to by one path must have a corresponding node that is pointed to by the other path, where the two nodes are node-value equal.

2.2 Example XML Data Redundancies

We now illustrate data redundancies that can be caused by constraints on the XML data and describe the features of those redundancy-indicating constraints. All the examples are based on the data tree in Figure 1.

Constraint 1. *Whenever two books (e.g., nodes 30 and 50) agree on their ISBN values, they will have the same title.*

It is clear that Constraint 1 leads to redundancies if there are two distinct books in the data with the same ISBN value: their titles are redundantly stored. Intuitively, such XML constraints consist of three components. First, *target elements*, which is the set of data elements (e.g., the books) on which the constraints are imposed. Second, *condition elements*, which are the elements (e.g., ISBN) specified in the condition of the constraint. Third, *implication elements*, which are the elements (e.g., title) whose equality is implied if the condition is met. It is worth noting that not all constraints correspond to redundancies. For example, if each distinct book in the data has a unique ISBN value, then Constraint 1 will not result in any redundancy. We will explore the properties of redundancy-indicating constraints later in Section 3.3.

Constraint 1 is straight-forward because both ISBN and title are subelements of book, and each book has exactly one ISBN and one title. However, constraints on XML data can become more complicated. Consider:

Constraint 2. *Whenever two books are on sale at stores with the same name, if they agree on their ISBN values, they will have the same price.*

Again, Constraint 2 indicates redundancies if there exist two distinct books that share the same ISBN value and that are being sold at the same store or at two stores with the same name. More importantly, Constraint 2 illustrates two important features for XML constraints. First, constraints can involve elements from *multiple hierarchies*. In this case, while the target elements are the set of books, the condition elements include not only a descendant element of book (i.e., title), but also a store name element that is neither ancestor nor descendant of book. Second, constraints can involve *missing elements*. Often, either the condition elements or the implication elements can be missing in the data instances. For example, the price of the book node 80 is not recorded. The following constraints illustrate yet another important feature of XML constraints:

Constraint 3. *Whenever two books agree on their ISBN values, they have the same set of authors.*

Constraint 4. *Whenever two books share the same set of authors and the same title, they agree on their ISBN values.*

Constraints 3 and 4 indicate redundancies if there are two distinct books (e.g., book nodes 30 and 50) in the data with either the same ISBN values, or the same title values and the same set of author values. Most importantly, it is not any individual author, but rather the set of authors, that are being compared or redundantly stored because each book has

DEFINITION 5 (GENERALIZED TREE TUPLE). *A generalized tree tuple of data tree* $T = \langle N, \mathcal{P}, \mathcal{V}, n_r \rangle$*, with regard to a particular data node* n_p *(called pivot node), is a tree* $t^T_{n_p} = \langle N^t, \mathcal{P}^t, \mathcal{V}^t, n_r \rangle$*, where:*

- $N^t \subseteq N$ *is the set of nodes,* $n_p \in N^t$*;*
- $\mathcal{P}^t \subseteq \mathcal{P}$ *is the set of parent-child edges;*
- $\mathcal{V}^t \subseteq \mathcal{V}$ *is the set of value assignments;*
- n_r *is the same root node in both* $t^T_{n_p}$ *and* T*;*
- $n \in N^t$ *iff 1)* n *is a descendant or ancestor of* n_p *in* T*, or 2)* n *is a non-repeatable direct descendant of an ancestor of* n_p *in* T*;*
- $(n_1, n_2) \in P^t$ *iff* $n_1 \in N^t, n_2 \in N^t, (n_1, n_2) \in P$*;*
- $(n, s) \in V^t$ *iff* $n \in N^t, (n, s) \in V$*.*

Similar to an original tree tuple, a generalized tree tuple is a data tree projected from the original data tree. However, instead of separating sibling nodes with the same path at all hierarchy levels, a generalized tree tuple has an extra parameter called a *pivot* node, and the separation is done only at subtrees rooted above the pivot node. As a result, ancestor and descendant nodes of the pivot node, as well as all the non-repeatable direct descendant nodes (previously defined in Section 2.1) of those ancestor nodes, are preserved in the tuple. Figure 3(B) illustrates one such generalized tree tuple with node 30 as the pivot node. Note that both author nodes of the book are preserved in the tuple, while in Figure 3(A), only one is kept. Based on the pivot node, we can categorize all generalized tree tuples into tuple classes:

DEFINITION 6 (TUPLE CLASS). *A* tuple class C^T_p *of the data tree* T *is the set of all generalized tree tuples* t^T_n*, where* $path(n) = p$*. Path* p *is called the pivot path.*

For example, the generalized tree tuple in Figure 3(B) belongs to the tuple class $C_{/\texttt{warehouse/state/store/book}}$[3]. Finally, we introduce the notion of XML FD based on tuple class:

DEFINITION 7 (XML FD). *An* XML FD *is a triple* $\langle C_p, LHS, RHS \rangle$*, often written as* $\{P_{l1}, P_{l2}, ..., P_{ln}\} \rightarrow P_r$ *w.r.t.* C_p*, where* C_p *denotes a tuple class, LHS is a set of paths* $(P_{li}, i = [1, n])$ *relative to* p*, and RHS is a single path* (P_r) *relative to* p*.*

An XML FD holds on a data tree T *(or* T *satisfies an XML FD) iff for any two generalized tree tuples* $t_1, t_2 \in C_p$*:*

- $\exists\ i \in [1, n]$*,* $t_1.P_{li} = \perp$ *or* $t_2.P_{li} = \perp$*, or*
- *If* $\forall\ i \in [1, n]$*,* $t_1.P_{li} =_{pv} t_2.P_{li}$*, then* $t_1.P_r \neq \perp$*,* $t_2.P_r \neq \perp$*,* $t_1.P_r =_{pv} t_2.P_r$*. A null value,* $\perp$*, results from a path that matches no node in the tuple, and* $=_{pv}$ *is the path-value equality defined in Definition 4.*

Because generalized tree tuples can be defined at any hierarchy level, with an appropriate tuple class specification, this new XML FD notion can effectively capture constraints involving set elements. For example, Constraints 3 and 4 can now be expressed as:

FD 3: {`./ISBN`} → `./author` w.r.t. $C_{\texttt{book}}$

FD 4: {`./author, ./title`} → `./ISBN` w.r.t. $C_{\texttt{book}}$

with the expected semantics. And the other two example constraints (Constraints 1 and 2) can be expressed as:

FD 1: {`./ISBN`} → `./title` w.r.t. $C_{\texttt{book}}$

FD 2: {`../contact/name, ./ISBN`} → `./price` w.r.t. $C_{\texttt{book}}$.

Remarks: First, missing elements (regarded as being null values) are treated in the same way as in [24], where they are considered as different from each other and from all other existing elements (i.e., each FD must be strongly satisfied [4]). Second, we note that if we limit tuple classes to only those with pivot paths corresponding to leaf level elements and consolidate all tuple classes into one class, this notion has the same expressive power as the one proposed in [3]. Third, FDs involving set elements only on the RHS can also be captured by incorporating multivalued dependencies (MVD) [10] into the previous tuple based approach. However, in general, FDs involving set elements cannot be captured using MVD. For example, FD 4 can not be expressed using MVD because the set of `author` values must be considered together. Fourth, previous XML FD notions only consider one element at a time, and hence do not consider order between elements. Since we consider sets of elements, we could consider order between siblings, and treat each collection as a list rather than a set. But then we would miss redundancies where the element order was changed, something we believe is a common occurrence. As such, we have chosen to treat our collections as unordered sets, and to ignore order in XML. The impact of considering order in our system is discussed in Section 4.5.

When the RHS of an XML FD is `./@key`, the LHS then uniquely identifies each tuple in C_p because the pivot node (and hence its key) for each tuple is unique. This naturally leads us to the following XML Key notion:

DEFINITION 8 (XML KEY). *An* XML Key *of a data tree* T *is a pair* $\langle C_p, LHS \rangle$*, where* T *satisfies the XML FD* $\langle C_p, LHS, ./@key \rangle$*.*

This new notion of XML Key shares many similarities with the notion proposed by Buneman et al in [5], which contains a target path (which identifies a set of nodes) and a set of key paths (which uniquely identifies each node in the aforementioned set). In fact, our notion extends their notion by allowing the key paths to be arbitrarily relative to the target path while ensuring the semantics is still valid.

3.2 Interesting XML FD

The range of XML FDs expressible under the new notion are quite broad. However, not all expressible FDs are of interest. For example, some FDs may not be interesting because they are trivial or redundant with other FDs.

3.2.1 Trivial XML FDs

DEFINITION 9 (TRIVIAL XML FD). *An XML FD* $\langle C_p, LHS, RHS \rangle$ *is trivial if 1)* $RHS \in LHS$*, or 2) for any generalized tree tuple in* C_p*, there is at least one path in LHS that matches no data node.*

The definition of trivial XML FDs partly follows the relational semantics, where an FD is trivial if the LHS contains the RHS, and partly follows the strong satisfaction semantics, where an FD is trivial if the LHS always contains at least one null value. Such a situation can arise, as mentioned in [3], because of the existence of Choice elements. For example, if contact is a Choice element instead of a Rcd element (i.e., it can have either name or address as its child, but not both) in Figure 2, then the XML FD {`./contact/name`, `./contact/address`} → `./@key` w.r.t. C_{store} is trivial since no C_{store} tuple will have both LHS nodes.

3.2.2 Essential Tuple Classes

THEOREM 1. *Given a tuple class* C_p*, if* p *is not a repeatable path (see Section 2.1), and there exists tuple class* $C_{p'}$*, where*

[3]The superscript is often omitted for brevity. The same for the subscript, which in this case can be abbreviated as `book`.

warehouse	*state*	name	*store*	*contact*	contact/name	contact/address	*book*	ISBN	author	title	price
1	10	WA	12	13	Borders	Seattle	20	00...269	Post	DBMS	126.99
1	10	WA	12	13	Borders	Seattle	30	00...638	Rama...	DBMS	79.90
1	10	WA	12	13	Borders	Seattle	30	00...638	Gehrke	DBMS	79.90
...	...	...	...	...	...	...	...	...	...	...	...

Figure 5: Example flat tuples in the flat representation of the XML data in Figure 1.

4.1 XML Data Representation

Flat Representation: The XML FD notion proposed in [3] suggests a natural way of XML FD discovery: the original XML data tree can be represented as a single relational table, and existing relational FD discovery algorithms can be directly applied. As shown in Figure 5, the flat representation converts the XML data into a single relation of flat tuples, where each attribute in the relation corresponds to a distinct schema element and each tuple is generated by selecting one data value (or @key) from the data tree for each simple (or complex) element, following the notion of tree tuple in [3]. For example, the tuple in Figure 3(A) is represented by the second tuple in Figure 5.

There are, however, two major issues with applying existing relational FD discovery algorithms to this flat representation. First, it is not clear how certain interesting XML FDs (i.e., those involving set elements) can be discovered. For example, those algorithms cannot discover previously mentioned XML FDs like FD 3 and FD 4. Second, relational FD discovery algorithms have exponential complexity in the number of attributes they have to consider. As a result, this implementation does not scale well when the XML schema is complex: the more complex the XML schema is, the more attributes there are in the transformed relational schema. Furthermore, the number of tuples in the single relation will increase multiplicatively if the schema contains multiple set elements that have no ancestor-descendant relationship with each other. For example, if each book had two review elements, the total number of tuples in Figure 5 would double.

Hierarchical Representation: Inspired by the notion of essential tuple class (Section 3.2.2), and the concept of nested relation [19], a more compact representation of the XML data can be adopted. As shown in Figure 6, the original XML data tree can be converted into a set of relations, where each relation R_p (e.g., R_{book}) corresponds to an essential tuple class C_p (e.g., $C_{\texttt{book}}$). Attributes in each relation match distinct non-repeatable schema elements, whose longest repeatable prefix path is the pivot path of C_p. There are two additional attributes: 1) the *@key* attribute, which matches to the pivot path itself and serves as the key for the relation (since each generalized tree tuple has a unique pivot node); 2) the *parent* attribute, which matches to the pivot path of C_p's lowest-repeatable-ancestor tuple class (see Theorem 1). For example, the parent attribute of R_{book} corresponds to the path `/warehouse/state/store` since $C_{\texttt{store}}$ is the lowest-repeatable-ancestor tuple class of $C_{\texttt{book}}$. Each tuple (called *essential tuples*) in the relations corresponds to a partial generalized tree tuple in C_p. Any generalized tree tuple of an essential tuple class can be fully reconstructed by joining tuples from multiple relations (on the *parent* and *@key* attributes). For example, to generate the generalized tree tuple in Figure 3(B), one can join `t10` in R_{state} with `t12` in R_{store}, then with `t30` in R_{book}, then with `t32` and `t33` in R_{author}. We call R_{p1} a *parent relation* of R_{p2}, and R_{p2} a *child relation* of R_{p1}, if C_{p1} is the lowest-repeatable-ancestor tuple class of C_{p2}. For example, R_{store} is a parent relation of R_{book}. We can similarly define *ancestor relation* and *descendant relation*.

R_{root}

@key	*parent*
1	$\perp$

R_{state}

@key	*parent*	name
10	1	WA
40	1	KY

R_{store}

@key	*parent*	*contact*	contact/name	contact/addr.
12	10	13	Borders	Seattle
42	40	43	Borders	Lexington
72	40	73	WHSmith	Lexington

R_{book}

@key	*parent*	ISBN	title	price
20	12	00...269	DBMS	126.99
30	12	00...638	DBMS	79.90
50	42	00...638	DBMS	79.90
80	72	00...638	DBMS	$\perp$

R_{author}

@key	*parent*	author
22	20	Post
32	30	Ramakrishnan
33	30	Gehrke
52	50	Ramakrishnan
53	50	Gehrke
82	80	Ramakrishnan
83	80	Gehrke

Figure 6: Example essential tuples in the hierarchical representation of the XML Data in Figure 1.

Compared with the flat representation, hierarchical representation avoids many redundancies because the common part of different tree tuples is represented only once. For example, title and price about a single book is stored once (in R_{book}) throughout the entire database, instead of once for each author as in Figure 5. Therefore, each individual relation in Figure 6 is considerably smaller than the single relation in Figure 5 in terms of both the number of tuples it has and the number of attributes it maintains. Interesting XML FDs, whose LHS and RHS paths are in the same relation (e.g., FD 1 in Section 3.1), can be discovered efficiently by applying existing relational FD discovery algorithms to individual relations in isolation. The problem, however, is that not all interesting XML FDs contain only LHS or RHS paths within the same relation. For example, all the other three FDs (FD 2-4) in Section 3.1 contain paths that appear in multiple relations. We call XML FDs/Keys that involve a single relation *intra-relation FDs/Keys*, and those that involve multiple relations *inter-relation FDs/Keys*. The challenge is how to efficiently discover interesting inter-relation XML FDs/Keys. In the rest of the section, we present algorithms for discovering inter-relation FDs (Section 4.3) and FDs involving set elements (Section 4.4) based on the concepts of *partition target* and *set partition*. Section 4.5 analyzes the complexities of those algorithms. We first briefly describe how relational algorithms are applied to discover intra-relation FDs.

4.2 Discovering Intra-Relation FDs

The algorithm for discovering intra-relation FDs is adopted from existing partition-based algorithms, including TANE [13], Dep-Miner [16], and FUN [20]. There are two main data structures: *attribute partition* and *attribute set lattice*.

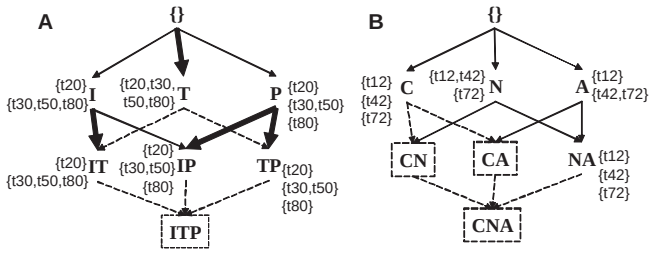

Figure 7: Example attribute set lattices for R_{book} (A) and R_{store} (B). I, T, P, C, N, A stand for `ISBN`, `title`, `price`, `contact`, `contact/name`, `contact/address`, respectively. Shown along selected nodes are the attribute partitions. Bold edges correspond to satisfied FDs. Dashed nodes and edges are those not visited in Algorithms DiscoverFD and DiscoverXFD.

Attribute Partition: An attribute partition of an attribute set X (Π_X) is a set of partition groups, where each group contains all tuples sharing the same values at X. For example, in R_{book}, $\Pi_{\{\texttt{ISBN,price}\}} = \{\{\texttt{t20}\}, \{\texttt{t30,t50}\}, \{\texttt{t80}\}\}$[4][5]. We say Π_X is a *refinement* of Π_Y ($\Pi_X \hookrightarrow \Pi_Y$) if whenever two tuples are in the same group in Π_X, they are in the same group in Π_Y, which leads to the following:

LEMMA 1. *A given intra-relation FD: $LHS \rightarrow RHS$ w.r.t. C_p holds iff $\Pi_{LHS} \hookrightarrow \Pi_{RHS}$ in R_p.*

LEMMA 2. *A given intra-relation FD: $LHS \rightarrow RHS$ w.r.t. C_p holds iff $\Pi_{LHS \cup RHS} = \Pi_{LHS}$ in R_p.*

Lemma 1 is straightforward and Lemma 2 is true because $\Pi_X \hookrightarrow \Pi_Y$ iff $\Pi_{X \cup Y} = \Pi_X$. Intuitively, Lemma 1 and 2 provide a more efficient way of determining the satisfaction of a given intra-relation FD.

Attribute Set Lattice: An attribute set lattice (in short, lattice) of relation R_p represents all intra-relation FDs in R_p (except those involving *@key* and *parent*). As shown in Figure 7, each node in the lattice corresponds to an attribute set, and an edge goes from node X to node Y if Y contains X and has exactly one more attribute than X. Each edge, in fact, corresponds to an intra-relation FD: let $Y = X \cup \{A\}$, edge (X, Y) corresponds to the intra-relation FD: $X \rightarrow A$ w.r.t. C_p.

The algorithm DiscoverFD (shown in Figure 8) aims to discover all intra-relation FDs that are not implied by other intra-relation FDs (i.e., minimal FDs). It traverses the lattice and discovers Keys and satisfied minimal FDs by constructing and comparing the attribute partitions. The lattice is simulated with queue Q, which produces the nodes from the lattice in level-order. For each node visited, the algorithm checks: 1) the associated partition to see if the attribute set is a Key. An attribute set is a Key if all groups in its partition contain exactly one tuple (line 11); 2) the set of associated edges to detect satisfied FDs. An FD corresponding to edge (X, XA) is satisfied if $\Pi_X = \Pi_{XA}$ (lines 12-14). The algorithm also produces new partitions by combining input partitions of smaller attribute sets (lines 9-10, details omitted due to lack of space).

Since the goal is to discover minimal FDs only, the algorithm adopts several optimization rules to remove certain nodes and edges from the lattice, which also improves performance because constructing and comparing partitions is costly. Assume

[4]All examples are based on the data in Figure 6.

[5]If a group contains only one tuple, it can be removed from the partition, resulting in a *striped* partition [13]. While we adopt striped partition in the implementation, we continue to use non-striped partition in the discussion for clarity.

Algorithm DiscoverFD:
Input: R_p with attributes $a_1, ..., a_n$
1. Generate $\Pi_\emptyset, \Pi_{a_1}, ..., \Pi_{a_n}$ from R_p
2. Init. Keys = $\emptyset$, FDs = $\emptyset$, Queue Q = $\emptyset$, AttributeSet A = $\emptyset$;
3. **for** $i = 1, ..., n$: Q.enqueue($\{a_i\}$); // the single attribute nodes
4. **while** Q $\neq \emptyset$:
5. A = Q.dequeue();
6. **if** Π_A does not exist: // Π_A needs to be generated
7. Ls = candidateLHS(A, FDs);
8. **if** Ls.size==0: **continue**; // No need to expand A
9. **if** Ls.size==1: **let** $A_1 \in$ Ls: $\Pi_A = \Pi_{A-A_1} \bullet \Pi_{A_1}$;
10. **if** Ls.size>=2: **let** A_1, $A_2 \in$ Ls: $\Pi_A = \Pi_{A_1} \bullet \Pi_{A_2}$;
11. **if** Π_A.maxGrpSize==1: Keys.add(A); **continue**;
12. **foreach** $A_L \in$ Ls: // A_L is the LHS
13. **let** r = A$-A_L$; // r is the RHS
14. **if** $\Pi_{A_L} == \Pi_A$: FDs.add($A_L \rightarrow r$);
15. **let** a_k be the last attribute in A;
16. **for** $i = k+1, ..., n$:
17. A$' =$ A$\cup a_i$;
18. **if** there is no K$\in$Keys, such that K$\subset$A$'$: Q.enqueue(A$'$);
Output: Keys–the set of intra-relation Keys w.r.t. C_p
FDs–the set of intra-relation FDs w.r.t. C_p

Function candidateLHS(A, FDs):
19. Init. Ls = $\emptyset$, the set of LHSs to be returned
20. **foreach** $a \in$ A:
21. **let** A_L = A-a;
22. **foreach** L$\rightarrow$r $\in$ FDs:
23. **if** a==r **and** $A_L \subset$L: **continue**;
24. **else if** A$\subset$L: **continue**;
25. **else** Ls.add(A_L);
26. **return** Ls

Figure 8: Algorithm DiscoverFD.

that X, Y are two possibly empty attributes sets, A, B are two single attributes, $A, B \notin X$, $A, B \notin Y$, and $X \cap Y = \emptyset$, the rules are: 1) Edge (XY, XYA) is removed if edge (X, XA) corresponds to a satisfied FD (line 23), because if $X \rightarrow A$ w.r.t. C_p holds, then $X \cup Y \rightarrow A$ w.r.t. C_p is implied; 2) Edge $(XYA, XYAB)$ is removed if edge (X, XA) corresponds to a satisfied FD (line 24). This is because if $X \rightarrow A$ w.r.t. C_p holds, then $X \cup Y \cup \{A\} \rightarrow B$ w.r.t. C_p is implied by $X \cup Y \rightarrow B$ w.r.t. C_p and thus it would not be minimal. For example, in Figure 7(A), after visiting edge (I, IT) and detecting $\{\texttt{ISBN}\} \rightarrow \texttt{./title}$ w.r.t. C_p is satisfied, edge (IP, ITP) is removed by the first rule, while edge (IT, ITP) is removed by the second rule; 3) If X is detected as an XML Key, the algorithm removes all nodes XY from the lattice (lines 11, 18). For example, in Figure 7(B), nodes CN, CA, and CNA are removed because C is an XML Key.

4.3 Discovering Inter-Relation FDs

The number of all possible inter-relation FDs is usually significantly larger than the number of all possible intra-relation FDs. Fortunately, the number of minimal inter-relation FDs is limited as Lemma 3 shows:

LEMMA 3. *Let $fd_0 = LHS \rightarrow RHS$ w.r.t. C_p be an inter-relation FD. For a given relation $R_{p'}$, where $R_{p'} = R_p$ or $R_{p'}$ is an ancestor relation of R_p (R_p is the relation corresponding to C_p), let $LHS' \subset LHS$ be the set of paths corresponding to attributes in $R_{p'}$ and descendant relations of $R_{p'}$. We have: 1) If $fd_1 = LHS' \cup \{$ p$'$/parent$\} \rightarrow RHS$ w.r.t. C_p does not hold, then fd_0 cannot be satisfied; 2) If FD $fd_2 = LHS' \rightarrow RHS$ w.r.t. C_p holds, then fd_0 is implied by fd_2.*

First, if an FD does not even hold for tuples with the same *parent* in a relation, any inter-relation FD that is generated by extending its LHS with attributes from ancestor relations cannot hold either. This is true because no ancestor attribute set can distinguish tuples with the same parent in the current relation. For example, $\{\texttt{./title}\} \rightarrow \texttt{./price}$ w.r.t. C_{book}

Algorithm DiscoverXFD(R_p, Keys, FDs):
1. Init. curKeys = ∅, curFDs = ∅, Q = ∅, AttributeSet A = ∅;
2. Init. PTs = ∅, resultPTs = ∅; // the set of PartitionTargets
3. Generate $\Pi_\emptyset, \Pi_{a_1}, ..., \Pi_{a_n}$ from R_p; // Attribute Partitions
4. Generate Index ID_p mapping *@key* to *parent* in R_p;
5. **foreach** child relation R of R_p:
6. PTs.addSet(DiscoverXFD(R, Keys, FDs));
7. **for** $i = 1, ..., n$: Q.enqueue($\{a_i\}$);
8. **foreach** pt∈PTs: // see Figure 10 for PartitionTarget
9. pt′ = updatePT(ID_p, pt, $\Pi_\emptyset$);
10. **if** pt′ ≠ NULL: resultPTs.add(pt′);
11. **while** Q ≠ ∅:
12. A = Q.dequeue();
// generate Π_A if needed
13. **if** Π_A does not exist:
14. Ls = candidateLHS2(A, curFDs);
15. **if** Ls.size==0: **continue**; // No need to expand A
16. **if** Ls.size==1: **let** A_1 ∈ Ls: $\Pi_A = \Pi_{A-A_1} \bullet \Pi_{A_1}$;
17. **if** Ls.size>=2: **let** A_1, A_2 ∈ Ls: $\Pi_A = \Pi_{A_1} \bullet \Pi_{A_2}$;
// A is a Key, the FDTarget of all PTs can be satisfied
18. **if** Π_A.maxGrpSize==1:
19. curKeys.add(A);
20. **foreach** pt∈PTs:
21. **if** pt.KeyTarget ≠ *invalid*:
22. Keys.add(pt.FD.L∪A w.r.t. pt.FD.C);
23. **else** // pt.KeyTarget can still be satisfied
24. FDs.add(pt.FD.L∪A → pt.FD.R w.r.t. pt.FD.C);
25. **continue**;
// not a Key, check if Π_A satisfy any PT from child relations
26. **foreach** pt∈PTs:
27. **if** Π_A does not satisfy pt.FDTarget:
28. pt′ = updatePT(ID_p, pt, Π_A);
29. **if** pt′ ≠ NULL: resultPTs.add(pt′);
30. **else if** Π_A satisfies pt.FDTarget & pt.KeyTarget:
31. Keys.add(pt.FD.L∪A w.r.t. pt.FD.C);
32. **else** // Π_A satisfies pt.FDTarget alone
33. FDs.add(pt.FD.L∪A → pt.FD.R w.r.t. pt.FD.C);
// generate potential PTs for parent relation
34. **foreach** A_L∈ Ls: // A_L is the LHS
35. **if** $\Pi_{A_L} == \Pi_A$: curFDs.add(A_l→A−A_L); **continue**;
36. pt = createPT($ID_p, \Pi_{A_L}, \Pi_A, C_p$); // C_p is R_p's tuple class
37. **if** pt ≠ NULL: resultPTs.add(pt);
// continue the traversal
38. **let** a_k be the last attribute in A;
39. **for** $i = k+1, ..., n$:
40. A′ = A∪a_i;
41. **if** there is no K∈curKeys, such that K⊂A′: Q.enqueue(A′);
42. **return** resultPTs;

Output: Keys–the set of inter-relation Keys under C_p
FDs–the set of inter-relation FDs under C_p

Function candidateLHS2(A, FDs):
same as Function candidateLHS in Figure 8 without line 24.

Figure 9: Algorithm DiscoverXFD.

Function createPT($ID_p, \Pi_{A_L}, \Pi_A, C_p$):
1. Init. new PartitionTarget pt;
2. pt.FD = A_L→A−A_L w.r.t. C_p;
3. pt.FDTarget = ∅; pt.KeyTarget = ∅;
4. **foreach** g_1∈Π_{A_L}:
5. **foreach** g_2∈Π_A and $g_2 \subseteq g_1$:
6. addKeyIneqs(ID_p, pt, g_2);
7. **if** $g_1 \neq g_2$: // need further separation
8. $g_1 = g_1 - g_2$;
9. **foreach** t_1∈g_1, t_2∈g_2:
10. $t'_1 = ID_p$.get(t_1); $t'_2 = ID_p$.get(t_2);
11. **if** $t'_1 == t'_2$: **return** NULL; // impossible separation
12. **else** pt.FDTarget.add($t'_1 \neq t'_2$);
13. **if** $g_1 \neq \emptyset$: addKeyIneqs(ID_p, pt, g_1);
14. **return** pt

Function updatePT(ID_p, pt, Π_A):
15. Init. new PartitionTarget pt′;
16. **foreach** $(t_1 \neq t_2)$ ∈ pt.FDTarget:
17. **if** Π_A does not satisfy $t_1 \neq t_2$:
18. $t'_1 = ID_p$.get(t_1); $t'_2 = ID_p$.get(t_2);
19. **if** $t'_1 == t'_2$: **return** NULL;
20. **else** pt′.FDTarget.add($t_1 \neq t_2$);
21. **foreach** $(t_1 \neq t_2)$ ∈ pt.KeyTarget:
22. **if** Π_A does not satisfy $t_1 \neq t_2$:
23. $t'_1 = ID_p$.get(t_1); $t'_2 = ID_p$.get(t_2);
24. **if** $t'_1 == t'_2$: pt′.KeyTarget = *invalid*; **break**;
25. **else** pt′.KeyTarget.add($t'_1 \neq t'_2$);
26. **return** pt′

Function addKeyIneqs(ID_p, pt, g):
27. **if** g.numTuples==1 **or** pt.KeyTarget == *invalid*: **return**;
28. **foreach** $t_1, t_2 \in g$ **and** $t_1 \neq t_2$:
29. $t'_1 = ID_p$.get(t_1); $t'_2 = ID_p$.get(t_2);
30. **if** $t'_1 == t'_2$: pt.KeyTarget = *invalid*; **return**;
31. **else** pt.KeyTarget.add($t'_1 \neq t'_2$);
32. **return**

struct PartitionTarget (i.e., PT):
FD: L→R w.r.t. C; // the FD this PT corresponds to
FDTarget; // inequalities needed for inter-relation FD
KeyTarget; // additional inequalities for inter-relation Key

Figure 10: Utility Functions.

does not hold for tuples `t20` and `t30`, which share the same parent `t12`. No matter what attributes from R_{store} and R_{state} are added to the LHS, `t20` and `t30` will always violate the resulting FD. Second, if an FD is already satisfied, extending its LHS with ancestor attributes produces only implied inter-relation FDs. Therefore, any minimal inter-relation FD is built upon an intra-relation FD that is satisfied under individual parents but not throughout the entire relation.

Algorithm DiscoverXFD is designed based on the above conclusions. It treats the entire collection of relations as a tree with edges corresponding to their parent/child relationships. It proceeds from leaf level to top level relations (line 5-6: child relations are visited before the parent relation). At each relation, the algorithm accomplishes two things by employing the data structure *partition target* (shown in Figure 10): First, detecting any intra-relation FD/Key that is satisfied under individual parents but not the entire relation. Those FDs/Keys will become *candidate partial FDs/Keys*. Second, detecting any attribute set in the relation that can form a satisfied inter-relation FD/Key with any candidate partial FD/Key from its descendant relations. A partition target, which is associated with a candidate partial FD and a candidate partial Key (the FD's LHS), contains two sets of inequalities: one corresponds to the FD satisfaction condition (FDTarget) while the other corresponds to the Key satisfaction condition (KeyTarget). The inequalities are constructed from partitions (Function createPT in Figure 10) and updated as the algorithm moves up the hierarchies (Function updatePT in Figure 10).

The details of the algorithm are shown in Figure 9 and 10. We illustrate how it works through a simple example: the discovery of FD 2 {`../contact/name, ./ISBN`} → `./price` w.r.t. C_{book} on data in Figure 6. When visiting R_{book}, the algorithm detects that $\Pi_{\{ISBN\}}$ is not the same as $\Pi_{\{ISBN,price\}}$ (see Figure 7(A)), which means {`./ISBN`} → `./price` w.r.t. C_{book} is not satisfied. In fact, for this FD to be part of some inter-relation FD, two inequalities must be satisfied, namely `t30`≠`t80` and `t50`≠`t80`. Because these inequalities will have to be satisfied in the parent relation, tuples in them are converted into their parent tuples, resulting in `t12`≠`t72` and `t42`≠`t72`. Often, two tuples in the same inequality are converted into the same parent tuple: the inequality can never be satisfied and the FD is not considered as a candidate partial FD. In this case, however, both inequalities can potentially be satisfied (i.e., the FD holds for tuples sharing the same parent), therefore, the FD is regarded as a candidate partial FD. Furthermore, for the LHS of a potential inter-relation FD to be a Key, the inequality `t30`≠`t50` must also be satisfied, which converts into `t12`≠`t42`. As a result, a partition target corresponding to {`./ISBN`} → `./price` w.r.t. C_{book}

Algorithm CreateSetPartition:
Input: Π_A^{child}, the partition on attribute A in R_{child}
ID, the index maps *@key* to *parent* in R_{child}
1. Init. Π_A^{parent} as a single group of all distinct *parent*s in R_{child}
2. **foreach** $g \in \Pi_A^{child}$:
3. **foreach** $t \in g$: $t = ID.\text{get}(t)$ // convert *@key* to *parent*
4. divide g into a set G of duplicates eliminated groups, such that $t_1, t_2 \in$ same group iff $\text{cnt}(t_1) = \text{cnt}(t_2)$ in g
5. **foreach** $g' \in \Pi_A^{parent}$, $g'' \in G$:
6. divide g' into g'_1, g'_2, where $g'_1 = g''$ and $g'_2 = g' - g''$
Output: Π_A^{parent}, the set partition on A in R_{parent}

Figure 11: Algorithm CreateSetPartition.

is created, with its FDTarget being {`t12`≠`t72`, `t42`≠`t72`} and KeyTarget being {`t12`≠`t42`}. The algorithm then visits its R_{store} and examines its attribute partitions. In particular, in $\Pi_{contact/name}$ (see Figure 7(B)), `t72` is separated from `t12` and `t42`, which means the FDTarget is satisfied by the partition. On the other hand, `t12` and `t42` remain in the same group in $\Pi_{contact/name}$, which means the KeyTarget is not satisfied. As a result, {`../contact/name, ./ISBN`} → `./price` w.r.t. C_{book} is reported as an inter-relation FD.

4.4 Handling Set Elements

Finally, to discover FDs involving set elements, like FD 3: {`./ISBN`} → `./author` w.r.t. R_{book}, we generate *set partitions*, which separate tuples according to those set attributes. We explain Algorithm CreateSetPartition (Figure 11) through a simple example based on the data in Figure 6. Consider attribute `author` in R_{author}, Π^{author}_{author} = {{`t22`}, {`t32,t52, t82`}, {`t33,t53,t83`}}. The initial Π^{book}_{author} is set as {{`t20, t30, t50,t80`}} (line 1). The first group in Π^{author}_{author} is converted into {`t20`} (line 3), and since there is only one tuple, no group division is needed (line 4). Applying {`t20`} to Π^{book}_{author} (line 5-6) results in a refined Π^{book}_{author} = {{`t20`}, {`t30,t50,t80`}}. Going through the next two groups in Π^{author}_{author} will not further refine Π^{book}_{author}. In a similar way, Π^{book}_{author} can be further turned into Π^{store}_{author} = {{`t12`}, {`t42, t72`}}. Each generated set partition, in fact, groups the tuples in the parent relation in the same way as an attribute partition, and can therefore be directly used in both discovery algorithms to detect satisfied FDs involving set elements. For example, Π^{book}_{author} is added to the attribute set lattice of R_{book}, and FD 3 and FD 4 can be discovered just like any other interesting FDs.

It is easy to see that, in the worst case, a top-level relation will have to deal with a large number of set partitions coming from its descendant relations. In practice, however, this is less of a concern for the following two reasons: 1) most of the set partitions quickly become *key partitions* (the higher the partition moves, the more likely it becomes a key partition), where each group in the partition contains only one tuple. As discussed in Section 4.2, such partitions are optimized and have little effect on the performance; 2) higher level relations contain significantly fewer tuples and are therefore less impacted by the increasing number of set partitions.

4.5 Complexity Analysis and Discussion

We briefly analyze the complexities of algorithms DiscoverFD and DiscoverXFD. For DiscoverFD, the number of edges in the attribute set lattice and the number of partitions at each relation R, are bounded by $O(R_k 2^{R_k})$ and $O(2^{R_k})$, respectively, where R_k is the number of attributes in R. For each edge visited in the lattice, a scan of the tuples in the relation is required. As a result, DiscoverFD has a worst case time complexity of $O(R_n R_k 2^{R_k})$, where R_n is the number of tuples in R. For DiscoverXFD, at each relation R, partition targets and set partitions from its descendant relations must be examined for each partition of R. Since the number of such partition targets and set partitions can be in the worst case $O(R_d 2^{R_d})$ (where R_d is the total number of attributes of all descendant relations of R), the worst case complexity for DiscoverXFD at each relation is $O(R_n R_k 2^{R_k} + R_n R_d 2^{R_k+R_d})$. This is in contrast with the complexity of $O(R_{n'}(R_k + R_d)2^{R_k+R_d})$, where $R_{n'}$ is the number of tuples in the flat representation, if we adopt the flat representation and use relational FD discovery algorithms. While the worst case complexity is only slightly better for DiscoverXFD and still exponential, the pruning strategies employed by DiscoverXFD can often reduce the number of partition targets and set partitions to be examined to near linear (see Section 5), reducing the time cost of DiscoverXFD close to that of DiscoverFD.

Discussion: First, order can be considered. It simply requires that, for two data nodes to be considered equal, their positions among the siblings (which can be obtained when establishing the hierarchical relations), in addition to their values, must be matched. While this increases the cost of computing partitions, it is also likely to produce more *key partitions*, which can be pruned away. As a result, we do not expect the impact of considering orders to be significant. We do not consider order in our system because we believe order-unaware redundancy is more meaningful in practice. Second, for XML data stored in native format, our algorithms can not be applied directly. However, the general pruning principles as shown in Lemma 3 still apply. Finally, We note here that FDs really depend on inherent properties of the world being represented. It is not possible to "prove" that there is an FD based purely on the data. In this sense, any FD discovery algorithm must be viewed as merely suggesting FDs, which hold in the current instance of the database, rather than establishing FDs. Some suggested FDs may turn out to be spurious – artifacts of the current database instance. Where data collections are large and representative, it is unlikely that too many spurious FDs will be suggested. Nonetheless, a final manual verification is often required.

5. EXPERIMENTAL EVALUATION

We implemented DiscoverXFD on top of Berkeley DB [2] using Java. The main data structures, including attribute partitions, partition targets, etc., are stored on disk and fetched when necessary, and only a single attribute partition of a single relation is required to fit in memory (for efficient generation of new partitions). This results in a small memory footprint. All experiments were conducted on a PC with a 2.0GHz P4 CPU and 1GB RAM, running Windows XP (SP2) and JRE 1.4.2. The JVM memory was 512MB and the Berkeley DB cache size was 128MB. For timing measurements, each experiment was run three times and the average reading was recorded.

5.1 Real Life Datasets

We first evaluated DiscoverXFD on three available real life datasets to examine its practicality and to verify the existence of data redundancies in real world datasets. The datasets include: the Mondial [18] geography dataset; the human subset of PIR protein information dataset from Protein Information Resource [1]; the DBLP [15] bibliography dataset. The statistics of each dataset are shown in Table 1: the schema and tuple class depth (the latter is usually smaller because of the skip-

	Mondial	PIR	DBLP
schema elements	152	114	331
max. schema depth	5	7	4
tuple classes	31	31	73
max. tuple class depth	5	5	3
avg. attributes per relation	4.9	3.7	4.5
max. attributes per relation	17	15	27
data elements (in 000s)	48.7	1001.1	3736.4
document size (in MB)	1.2	31.8	133.8

Table 1: Statistics of real life datasets.

	Mondial	PIR	DBLP
loading time (secs.)	0.6	18.0	55.9
partition time (secs.)	6.0	20.2	79.9
discovery time (secs.)	11.1	253.6	1093.5
intra-relation FDs	21	73	313
intra-relation Keys	98	41	205
inter-relation FDs	9	8	0
inter-relation Keys	25	6	5
redundancy-indicating FDs	30	81	313

Table 2: Performance and results on real life datasets

ping of non-set schema elements) affect the discovery of inter-relation FDs and Keys, and the average and maximum number of attributes per relation (in the hierarchical representation) affect the discovery of both intra-relation and inter-relation FDs and Keys. While Mondial and PIR datasets are similarly nested, the DBLP dataset stands out with a relatively flat structure (tuple class depth of 3) and with more complex relations (larger average and maximum number of attributes per relation). The size of each dataset is measured as the number of data elements (including both elements and attributes) it contains.

We performed redundancy detection on the datasets and measured three time costs: the loading time (parsing the document and converting it into the hierarchical representation), the partition time (creating partitions of single attributes for all the relations), and the discovery time (the time of actual FD and Key discovery i.e., Algorithm DiscoverXFD). As shown in the Table 2, redundancies in all datasets can be detected in a reasonable amount of time, ranging from 20 seconds for Mondial to 20 minutes for DBLP, demonstrating the practicality of the system.

More importantly, data redundancies were detected in all three datasets, as measured by the number of redundancy-indicating FDs in Table 2 (it is calculated as the sum of recorded intra-relation FDs and recorded inter-relation FDs, including FDs involving set elements, because an FD is recorded only when its LHS is not a Key). An example redundancy-indicating FD in Mondial is shown in Figure 12. Here, in C_{city}, the name element of province and the country attribute of city together determine the province attribute of city, but they are not an XML Key (e.g, they do not determine the name element of city). As a result, the province attribute of city is stored redundantly: once for each city in the same province. Furthermore, while the exact number of redundant elements is unknown, its lower bound can be estimated by checking only the recorded intra-relation FDs and measuring the average group size of their LHS partitions. We performed this estimation on the PIR dataset and found that intra-relation FDs alone caused 104507 data elements to be stored redundantly, about 10.4% of total data elements. We have proposed modifications to the PIR schema to avoid these redundancies, and communicated this to the owners of the database[6].

[6]PIR has been replaced by the new UniProt database, whose design has taken into consideration our suggestions.

Mondial: Rcd
...
province: SetOf Rcd
name: str
city: SetOf Rcd
name: str
@province: str
@country: str
...

FD: {../name, ./@country} → ./@province w.r.t. C_{city}

Figure 12: Partial schema of the Mondial dataset and an example redundancy-indicating inter-relation FD.

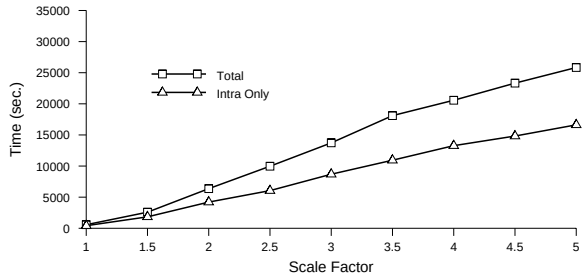

Figure 13: Cost of redundancy detection on XMark datasets with increasing scale factors.

5.2 Benchmark Dataset

We further evaluated DiscoverXFD on the XMark dataset to examine its scalability. The XMark schema shares similar schema characteristics with the nested real life datasets: 327 schema elements with a maximum depth of 9; 117 tuple classes with a maximum depth of 5; average and maximum number of attributes per relation at 2.8 and 17, respectively. The size of each dataset is linearly correlated to the scale factor used for its generation. At scale factor 1, the dataset contains about 2 million data elements and has a document size of 100MB. As shown in Figure 13, the total time for redundancy detection (line Total) increases linearly with the scale factor, indicating that the system scales well with increasing data size. To investigate whether detection of redundancies caused by inter-relation FDs is becoming more significant, we performed the detection for intra-relation FDs only (because inter-relation FDs cannot be discovered without incurring the cost of discovering intra-relation FDs). Again, the cost of detecting intra-relation FDs (Figure 13 line Intra Only) increases linearly to the scale factor and remains between 60-70% of the total (i.e, the cost of detecting redundancy-indicating inter-relation FDs stays about 30-40% of the total), indicating that manipulating partition targets and set partitions is efficient and does not dominate the overall detection process.

Comparison with relational algorithms: As mentioned in Section 4.1, XML FDs can also be discovered by applying relational algorithms on the flat representation of the XML data. While such an alternative implementation is limited due to its inability to discover FDs involving set elements, we nevertheless want to compare our system against it. Towards this goal, we implemented an alternative redundancy detection system, which converts XML data into flat representation and adopts the algorithm FUN [20][7] for FD and Key discovery (we made minor adjustments to avoid recording un-*interesting* FDs). We performed redundancy detection with this system

[7]FUN is chosen because it improves upon previous algorithms and is the fastest.

	S1	S2	S3	XMark (sf=0)
data elements	118	153	192	331
Relational Algorithm	0.9	6.0	128.2	>10000
DiscoverXFD	2.1	2.3	2.5	4.2

Table 3: Time cost (seconds) of redundancy detection on small XMark datasets using the alternative relational algorithm implementation and DiscoverXFD.

on all three real life datasets. Not surprisingly, it did not finish detection (within 24 hours) even on the Mondial dataset. In fact, redundancy detection using this system took hours for the smallest XMark dataset (scale factor 0), which contains only 331 elements. As a result, we created three more datasets (S1-S3) based on the smallest XMark dataset and compared the performance of our system against this alternative system on these. The results are shown in Table 3. As expected, while the alternative system performs well on very small datasets, it degrades rapidly as the size increases and performs much worse than our DiscoverXFD system for larger datasets.

6. RELATED WORK AND FUTURE WORK

Designing XML FDs was first addressed in [14], where the authors presented an intuitive way of expressing XML FDs. Formal definitions of XML FDs and Normal Forms were later proposed in [3] and [24], providing significant improvements over relational FDs in capturing XML data redundancies. However, as discussed at length in Section 2.3, both proposals are limited in their ability to capture redundancies involving set elements. The redundancy detection problem, one of our two main contributions, is not addressed in any of the above studies. Integrity constraints (including keys) in XML were first studied extensively in [5, 6, 11, 12], which proposed many notions being used here, as well as in many other studies. Their focus, however, is on reasoning about keys and integrity constraints. Furthermore, they do not adopt the tree tuple notion that we and [3] adopt here.

The hierarchical representation of XML data shares many similarities with nested relations [4]. In [21, 19], data redundancies in nested relations are characterized using relational FDs and MVDs. However, as mentioned in Section 3.1, MVDs cannot fully capture XML redundancies involving set elements on both sides of the dependency, and a more comprehensive notion of XML FD is therefore necessary.

Several algorithms [13, 16, 20] have been proposed for relational FD discovery. The intra-relation FD discovery algorithm is an extension of these algorithms, and their notion of partition is used extensively in the inter-relation FD discovery algorithm. Several recent studies have also focused on validating known XML Keys and FDs [7, 23], which is a considerably simpler problem than our problem of redundancy detection through the discovery of FDs and Keys.

While we focus on redundancy detection in this study, we are aware that defining XML Normal Form based on XML FDs and Keys, and designing normalization algorithms for schema refinement are equally important problems. Those problems were studied in [3, 24]. Future work can be done to improve upon their solutions based on our new notions and effectively remove redundancies described in this paper.

7. CONCLUSION

XML data redundancies have a richer semantics than redundancies in the relational context. We proposed generalized tree tuple based XML FD and Key notions that improve upon previous XML FD proposals and capture a comprehensive set of XML data redundancies, including in particular redundancies involving set elements. We designed and implemented DiscoverXFD, the first XML data redundancy detection system through the discovery of XML FDs and Keys. Experimental evaluation demonstrates that the system is practical in detecting redundancies in real datasets and scales well with increasing dataset size.

8. REFERENCES

[1] PIR International Protein Sequence Database. http://pir.georgetown.edu/pirwww/search/textpsd.shtml.

[2] Sleepycat Software. http://www.sleepycat.com/.

[3] M. Arenas and L. Libkin. A Normal Form for XML Documents. *TODS*, 29(1):195–232, 2004.

[4] P. Atzeni and V. DeAntonellis. *Foundations of Databases*. Benjamin Cummings, 1993.

[5] P. Buneman, S. Davidson, W. Fan, C. Hara, and W.-C. Tan. Keys for XML. In *WWW*, 2001.

[6] P. Buneman, S. Davidson, W. Fan, C. Hara, and W.-C. Tan. Reasoning About Keys for XML. *Inf. Syst.*, 28(8):1037–1063, 2003.

[7] Y. Chen, S. Davidson, and Y. Zheng. XKvalidator: A Constraint Validator for XML. In *CIKM*, 2002.

[8] E. F. Codd. A Relational Model of Data for Large Shared Data Banks. *Comm. of the ACM*, 13(6):377–387, 1970.

[9] E. F. Codd. Further Normalization of the Data Base Relational Model. *Data Base Systems*, 1972.

[10] R. Fagin. Multivalued Dependencies and a New Normal Form for Relational Databases. *TODS*, 3:262–278, 1977.

[11] W. Fan and L. Libkin. On XML Integrity Constraints in the Presence of DTDs. *Journal of the ACM*, 49(3):368–406, 2002.

[12] W. Fan and J. Simeon. Integrity Constraints for XML. *JCSS*, 66:2554–291, 2003.

[13] Y. Huhtala, J. Karkkainen, P. Porkka, and H. Toivonen. TANE: An Efficient Algorithm for Discovering Functional and Approximate Dependencies. *The Computer Journal*, 42(2), 1999.

[14] M. L. Lee, T. W. Ling, and W. L. Low. Designing Functional Dependencies for XML. In *EDBT*, 2002.

[15] M. Ley. DBLP Computer Science Bibliography. http://dblp.uni-trier.de/.

[16] S. Lopes, J.-M. Petit, and L. Lakhal. Efficient Discovery of Functional Dependencies and Armstrong Relations. In *EDBT*, 2000.

[17] H. Mannila and K.-J. Raiha. Dependency Inference. In *VLDB*, 1987.

[18] W. May. Information Extraction and Integration with Florid: The Mondial Case Study, 1999. http://www.dbis.informatik.uni-goettingen.de/lopix/lopix-mondial.html.

[19] W. Y. Mok, Y.-K. Ng, and D. Embley. A Normal Form for Precisely Characterizing Redundancy in Nested Relations. *TODS*, 21(1):77–106, 1996.

[20] N. Novelli and R. Cicchetti. Functional and Embedded Dependency Inference: A Data Mining Point of View. *Information Systems*, 26:477–506, 2001.

[21] Z. M. Ozsoyoglu and L.-Y. Yuan. A New Normal Form for Nested Relations. *TODS*, 12(1):111–136, 1987.

[22] L. Popa, Y. Velegrakis, R. Miller, M. Hernández, and R. Fagin. Translating Web Data. In *VLDB*, 2002.

[23] M. Vincent and J. Liu. Checking Functional Dependency Satisfaction in XML. In *XSym*, 2005.

[24] M. Vincent, J. Liu, and C. Liu. Strong Functional Dependencies and Their Application to Normal Forms in XML. *TODS*, 29(3):445–462, 2004.

[25] W3C. XML Schema. http://www.w3.org/TR/xmlschema-0/.