The VLDB Journal
DOI 10.1007/s00778-007-0063-0

SPECIAL ISSUE PAPER

XML schema refinement through redundancy detection and normalization

Cong Yu · H. V. Jagadish

Received: 20 February 2007 / Revised: 21 May 2007 / Accepted: 2 July 2007

Abstract As XML becomes increasingly popular, XML schema design has become an increasingly important issue. One of the central objectives of good schema design is to avoid data redundancies: redundantly stored information can lead not just only 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 strategy to avoid data redundancies is to design redundancy-free schema from the start on the basis of known functional dependencies. We observe that XML databases are often "casually designed" and XML FDs may not be determined in advance. Under such circumstances, discovering XML data redundancies from the data itself becomes necessary and is an integral part of the schema refinement (or re-design) process. 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. The XML data redundancies are defined on the basis of a new notion of XML functional dependency (XML FD) that (1) extends previous notions by incorporating set elements into the XML FD specification, and (2) maintains tuple-based semantics through the novel concept of *Generalized Tree Tuple* (GTT). Using this comprehensive XML FD notion, we introduce a new normal form (GTT-XNF) for XML documents, and provide comprehensive comparisons with previous studies. Given the set of data redundancies (in the form of redundancy-indicating XML FDs) discovered by *DiscoverXFD*, we describe a normalization algorithm for converting any original XML schema into one in GTT-XNF.

Keywords XML · Schema design · Functional dependency · Normal form · Data redundancy

C. Yu (✉) · H. V. Jagadish
Department of EECS, University of Michigan,
Ann Arbor, MI, USA
e-mail: congy@eecs.umich.edu

H. V. Jagadish
e-mail: jag@eecs.umich.edu

1 Introduction

Redundant data take up unnecessary storage, inflates data transfer cost, and can lead to update anomalies. A central objective of database design is to ensure that there are no unintended redundancies. As XML databases have become more common, good design of XML schemas has become increasingly important, especially in complex scientific databases. 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 schemas, while important for encouraging data generation, is likely to lead to many data redundancies in the resulting XML databases.[1] Designing a good XML schema is therefore often a two-stage process. In the first stage, data are generated according to the casually designed schema and redundancies are being discovered and recorded. In the second stage, the original schema is re-designed to eliminate unintended data redundancies, and the original data are transformed into the new schema.

The notion of functional dependency (FD) plays an important role in defining redundancies [7] in relational databases, and should play a correspondingly important role in XML

[1] Anecdotal examples include some large, heavily used community resources, such as PIR [21].

databases as well. Redundancies in XML data have several distinct features due to the heterogeneous nature of XML data, which makes them richer in semantics as compared with redundancies in relational data. As a result, standard relational FD discovery algorithms are insufficient to find all XML FDs (this is true whether we consider classic relational FD discovery algorithms such as [16], or more recent proposals such as Dep-Miner [15], TANE [11], and FUN [19]). In this paper, we develop a new algorithm *DiscoverXFD*, for efficient discovery of XML data redundancies in terms of redundancy-indicating XML FDs.

XML functional dependency (i.e., constraints which specify that values of certain XML elements are determined by others—to be formally defined later), and the related notion of XML normal form, have recently become an important research topic. In [1], Arenas and Libkin adopted a tree tuple-based approach and were the first to formally define XML FD and normal form (XNF). In [13,25], the authors took a path-based approach and built their XML FD notion in a fashion similar to the XML Key notion proposed in [4]. 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. Based on the new XML FD notion, we introduce a new XML normal form called GTT-XNF, and design an algorithm for converting any XML schema into one in GTT-XNF given a set of redundancy-indicating XML FDs.

The unique challenge in defining XML data redundancies can be illustrated using the example XML document shown in Fig. 1. The document 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 [1] and [25]. A third constraint is equally interesting: for any two books sold at the same store chain (i.e., stores with the same name), if they have the same ISBN, they will be sold at the same price. The price `79.90` of book `0072465638`, therefore, is stored redundantly for the store chain `Borders` (one in `Seattle` and the other in `Lexington`). Such a redundancy is special in that while it is the books on which the comparison is specified, the constraint actually involves an element (i.e., store name) that is not a descendant of book.

Main contributions and paper outline. We make the following main contributions: (1) We study the examples of redundancy-inducing (i.e., potentially redundancy indicating) constraints in the XML data model (Sect. 2.2) and show that existing XML FD notions are insufficient for capturing certain XML data redundancies (Sect. 2.3). (2) We propose a new *Generalized Tree Tuple*-based XML FD notion, which improves upon the notion introduced in [1]. We show that more XML data redundancies can now be effectively captured by *interesting* XML FDs (Sect. 3). (3) We introduce a new XML normal form (GTT-XNF) based on our proposed XML FD and XML Key notions and provide comprehensive comparisons with a previous XML Key notion [4] and a previous normal form notion XNF [1] (Sect. 4). (4) We design and implement the DiscoverXFD system, which employs a new XML data structure and several novel partition-based algorithms that can efficiently discover XML FDs and detect XML data redundancies (Sect. 5); (5) We describe a normalization algorithm for converting any XML schema into GTT-XNF given a set of redundancy-indicating XML FDs (Sect. 6). (6) We demonstrate the scalability and practicality of DiscoverXFD using a benchmark dataset and a variety of real life datasets (Sect. 7). We first present some necessary background.

2 Background and challenges

2.1 Schema and data tree

Figure 2 illustrates the schema of the example XML document in Fig. 1. It is shown in a nested relational representation [22] that is used as a common data model to represent both relational and hierarchical (XML) schemas. Intuitively, keyword Rcd is used to indicate *complex schema elements* (i.e., elements that have children elements, e.g., `contact`), and keyword SetOf is used to indicate *set schema elements* (i.e., elements that can have multiple matching data elements sharing the same parent in the data, e.g., `book`). A set element is not necessarily complex: e.g., `author` is a set element with a domain of string (str). Formally:

Definition 1 (*Schema*) A *schema* is defined to be $S = \langle E, T, r \rangle$, where:

- E is a finite set of element labels;
- T is a finite set of element types, and each $e \in E$ is associated with a $\tau \in T$, written as $(e : \tau)$, τ has the form: $\tau ::= \mathsf{str} \mid \mathsf{int} \mid \mathsf{float} \mid \mathsf{SetOf}\ \tau \mid \mathsf{Rcd}[e_1 : \tau_1, \ldots, e_n : \tau_n]$;

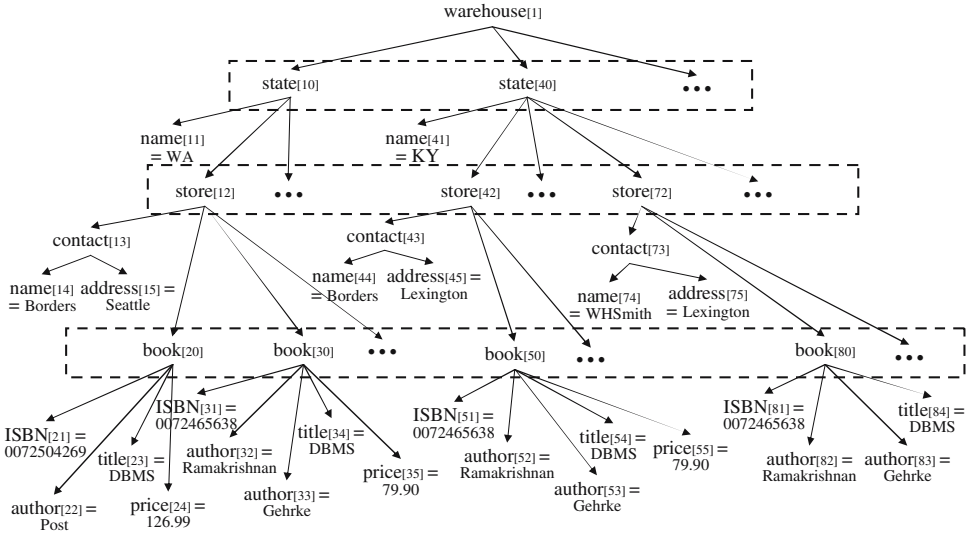

Fig. 1 Example XML document. Each node is assigned a key (shown in the *bracket*), which is referred to as `@key`, and the *dashed boxes* isolate data elements that correspond to complex set elements in the schema (Fig. 2)

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

Fig. 2 Example schema ($S_{\texttt{warehouse}}$) for the example XML document in Fig. 1

– $r \in E$ is the label of the root element, whose associated element type can not be SetOf τ.

Definition 1 corresponds to the "core" constructs in XML Schema [26]. Types str, int, and float are system defined *simple* types. Rcd is a *complex* type representing the "all" model-group in XML, respectively. Type SetOf is the *set* type associated with elements with `maxOccurs` greater than one in XML Schema. We ignore element order and represent the "sequence" model-group as the Rcd type. For simplicity, we treat attributes and elements in the same way, with a reserved "@" symbol to indicate attributes. For mixed-content elements, if there is exactly one textual value, we store it under a distinct new attribute "@value." Otherwise, we ignore the textual values and treat the mixed-content elements as regular *complex* elements.

A schema element e_k can be identified through a *path* expression, $path(e_k) = /e_1/e_2/\ldots/e_k$, where $e_1 = r$, and e_i is associated with type $\tau_i ::= \text{Rcd}[\ldots, e_{i+1} : \tau_{i+1}, \ldots]$ for all $i \in [1, k-1]$. Furthermore, if e_k is a set element, we call $path(e_k)$ a *repeatable path*, which is an important concept to be used later. Note that we do not consider $path(e_k)$ to be a repeatable path if e_k is not a set element, even if some $e_i (i < k)$ is a set element. For example, `/warehouse/state/name` is not a repeatable path while `/warehouse/state/store` is. For convenience, we adopt XPath steps "·" (self) and "··" (parent) to form a relative path given an anchor path. For example, if the anchor path is `/warehouse/state/store`, the relative path `../name` is equivalent to `/warehouse/ state/name`.

Definition 2 (*Data tree*) An XML database is defined to be a rooted labeled tree $T = \langle N, \mathcal{P}, \mathcal{V}, n_r \rangle$, where:

– N is a set of labeled data nodes, each $n \in N$ has a label e and a node key that uniquely identifies 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.

We assign a node key, referred to as `@key`, to each data node in the data tree in a pre-order traversal (gaps in the numbering in Fig. 1 indicate omitted elements). 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 [26] and assume that all given data trees conform to their schemas.

A data element n_k is a *descendant* of another data element n_1 if there exists a series of data elements n_i, such

that $(n_i, n_{i+1}) \in \mathcal{P}$ for all $i \in [1, k-1]$. Similar to schema elements, n_k can also be addressed using a *path* expression, $path(n_k) = /e_1/\ldots/e_k$, where e_i is the label of n_i for each $i \in [1, k]$, $n_1 = n_r$, and $(n_i, n_{i+1}) \in \mathcal{P}$ for all $i \in [1, k-1]$. It is possible that two distinct data elements can have the same path (e.g., node 11 and node 41). A data element 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 element n_a, if n_k is a descendant of n_a, $path(n_k) = \ldots/e_a/e_1/\ldots/e_{k-1}/e_k$, and e_i is not a set element for any $i \in [1, k-1]$. For example, node 21 (ISBN) is a direct descendant of node 20 (book), but not node 12 (store).

In considering data redundancy, it is important to determine the equality between the "values" associated with two data elements [4], instead of comparing their "identities," which is represented by `@key`. Therefore, we have:

Definition 3 (*Element-value equality*) Two data elements 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 *element-value equal (written as* $n_1 =_{ev} n_2$*)* if and only if:

- n_1 and n_2 both exist and have the same label;
- There exists a set M, such that for every pair $(n'_1, n'_2) \in M$, $n'_1 =_{ev} n'_2$, where n'_1, n'_2 are children elements of n_1, n_2, respectively. Every child element of n_1 or n_2 appears in exactly one pair in M.
- $(n_1, s) \in \mathcal{V}_1$ if and only if $(n_2, s) \in \mathcal{V}_2$, where s is a simple value.

Intuitively, two data elements (e.g., node 30 and 50) are element-value equal if and only if the subtrees rooted at those two elements are identical when the order among sibling elements is ignored. Based on element-value equality, we can now define the path-value equality:

Definition 4 (*Path-value equality*) Two data element paths p_1 on $T_1 = \langle N_1, \mathcal{P}_1, \mathcal{V}_1, n_{r1}\rangle$ and p_2 on $T_2 = \langle N_2, \mathcal{P}_2, \mathcal{V}_2, n_{r2}\rangle$ are *path-value equal* (written as $T_1.p_1 =_{pv} T_2.p_2$) if and only if there is a set M' of matching pairs where

- For each pair $m' = (n_1, n_2)$ in M', $n_1 \in N_1$, $n_2 \in N_2$, $path(n_1) = p_1$, $path(n_2) = p_2$, and $n_1 =_{ev} n_2$;
- All data elements with path p_1 in T_1 and path p_2 in T_2 participate in M', and each such data element participates in only one such pair.

Value equality between two paths is complicated by the fact that a single path can match multiple data elements 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 element-value equal.

Order consideration. In Definitions 3 and 4, we take extra steps to ignore the ordering among data elements. This is because ordering in a majority of the real life XML documents we have observed is not semantically significant. When order is important, Definitions 3 and 4 can be adjusted such that each matching pair always involves two data elements that have the same ordering position either among its siblings (Definition 3) or among the data elements with the same path (Definition 4).

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-inducing constraints. All the examples are based on the data tree in Fig. 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-inducing constraints later in Sect. 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 an ancestor nor a 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 a set of authors. The third important feature of XML constraints, therefore, is the involvement of *set elements*: each condition or implication element specification can, and often, resolve to a set of elements.

2.3 Previous proposals

The above constraints are essentially intuitive forms of functional dependencies (FDs). To capture redundancies indicated by those constraints, formal definitions of XML FD have been proposed and follow two main approaches: path-based approach and tuple-based approach. They differ in how the target elements of the constraint are specified: the former implicitly encodes the target elements inside the FD specification, while the latter specifies the target elements independent of each individual FD specification.

Path-based approach. Proposed by Vincent et al. [25] is representative of the path-based approach. An XML FD is of the form: $\{P_{x_1}, \ldots, P_{x_n}\} \rightarrow P_y$, where P_{x_i} (also called LHS) are the paths specifying the condition elements, P_y (also called RHS) is the path specifying the implication element, and the target elements are implicitly specified as the set of elements pointed to by P_y. For example, Constraint 1 can be expressed as: {`/warehouse/state/store/book/ISBN`}→`/warehouse/state/store/book/title`. The semantics of the FD is intuitively defined as the following: for any two distinct title nodes in the data tree, if the ISBN nodes they are associated with have the same value, then the title nodes themselves have the same value. A title node and an ISBN node are associated if they are the descendants of the same book node (book is chosen because its path is the longest common prefix of both title and ISBN). For example, the FD is satisfied in Fig. 1 because for any two titles (e.g., nodes 34 and 54), if their associated ISBNs (e.g., nodes 31 and 51, respectively) share the same value, they have the same value as well.

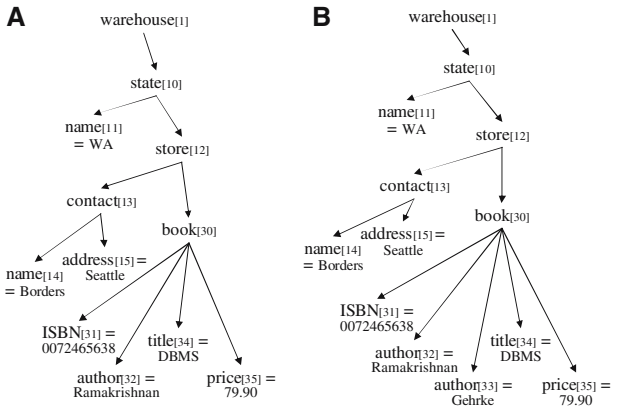

Fig. 3 (**a**) Original tree tuple example, and (**b**) generalized tree tuple example (`book` 30 is the pivot node)

Tuple-based approach. In [1], Arenas and Libkin proposed the first formal XML FD notion built upon the concept of *tree tuples*. Instead of specifying target elements from each individual FD as in [25], a set of tree tuples is defined independent of any FD and serves as the target for all FDs. Each tree tuple is a tree constructed by picking exactly one data node from the original data tree for each schema element and projecting away all the other nodes. Figure 3a illustrates one such tree tuple. XML FDs are subsequently defined based on this tree tuple notion and take a form similar to the one in the path-based approach. For example, Constraint 2 can be expressed as: {`/warehouse/state/store/contact/ name, /warehouse/state/store/book/ISBN`}→`/warehouse/state/store/book/price`. The semantics of the FD is defined as the following: for any two tree tuples, if they have the same values at the nodes specified in the LHS of the FD (i.e., the name and ISBN nodes), they will share the same values at their RHS nodes (i.e., the price nodes). It is worth noting that, if the original XML data tree is viewed as a set of nested relations [3], the set of tree tuples is conceptually equivalent to the set of fully unnested tuples. Compared with path-based approach, tuple-based XML FD notion has a semantics that is closer to the relational FD notion. It also suggests a natural technique for XML FD discovery: one can convert the XML data into a fully unnested relation and apply existing FD discovery algorithms directly.

Discussion: Both [25] and [1] effectively capture multi-hierarchical constraints like Constraint 2. In the former, elements from different hierarchies are associated with each other through the common ancestor node. In the latter, they are connected by belonging to the same tree tuple. Both proposals also adopt a similar semantics for missing elements, which roughly corresponds to the strong satisfaction of FD over incomplete relations as defined in [3].

However, neither notion can effectively capture constraints with set elements. Consider Constraint 3 for Fig. 1.

The closest form to which it can be expressed under both [25] and [1] is the following: {/warehouse/ state/store/ book/ISBN}→/warehouse/state/store/book/ author. It is not difficult to see that the semantics of this FD under either notion are not the same as the semantics of the original constraint. The semantics under [25] is that for any two authors, if they are associated with the same ISBN value, their values are the same. Under this semantics, the FD is violated since book 30 has two authors of different values and the two authors are clearly associated with the same ISBN value. The semantics under [1] is that for any two tree tuples, if their ISBN nodes share the same value, then they have the same value for their author nodes. According to the construction of tree tuple, author 32 and author 33 belong to two different tree tuples. Since the ISBN nodes of the two tuples have the same value while the author nodes of the two tuples differ, the FD is again violated. The original constraint, however, is satisfied in Fig. 1: two books with the same ISBN value always have the same *set* of authors. In the next section, we proposed *Generalized Tree Tuple*-based XML FD notion that overcomes the semantic limitations the previous proposals have in expressing constraints with set elements.

3 Capturing XML data redundancy

While both tuple and path-based approaches are valid ways for defining XML FDs, the tuple-based approach has a clearer semantics and is conceptually similar to the relational FD notion. As such, we follow the tuple-based approach. In Sect. 3.1, we introduce the notion of *Generalized Tree Tuple* (*GTT*), which improves upon the tree tuple notion in [1]. Based on this new tuple notion, we define GTT-Based XML FD and XML Key. In Sect. 3.2, we analyze the general form of XML FDs and show that only a subset of such XML FDs are considered *interesting*. Finally, in Sect. 3.3, XML data redundancy is defined formally.

3.1 GTT-based XML FD

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 element 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;
- $n \in N^t$ if and only if: 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;
- $(n_1, n_2) \in P^t$ if and only if $n_1 \in N^t, n_2 \in N^t, (n_1, n_2) \in P$;
- $(n, s) \in V^t$ if and only if $n \in N^t, (n, s) \in V$.

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 for data elements 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 Sect. 2.1) of those ancestor nodes, are preserved in the tuple. Figure 3b 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 Fig. 3a, 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 Fig. 3b belongs to the tuple class $C_{/\texttt{warehouse/state/store/book}}$.[2] Finally, we introduce the notion of XML FD-based on tuple class:

Definition 7 (*XML FD*) An *XML FD* is a triple $\langle C_p, LHS, RHS \rangle$, written as $LHS \rightarrow RHS$ 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) if and only if for any two generalized tree tuples $t_1, t_2 \in C_p$[3]

- $\exists\, i \in [1, n], t_1.P_{li} = \perp$ 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.

The expression $t.P$, where t is a tree tuple and P is a path expression, corresponds to (a set of) node(s) that are identified by following the path P starting with the pivot node of the tree t.

Because generalized tree tuples can be defined at any hierarchy level, with an appropriate tuple class specification,

[2] While the definitions and algorithms throughout the rest of paper handle both types, we largely omit the "choice" type for the simplicity of discussion.

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

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. $C_{\texttt{book}}$

Note that their semantics are exactly as expected. 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. $C_{\texttt{book}}$.

We treat missing elements (regarded as being null values) in the same way as in [25], where they are considered as different from each other and from all other existing elements (i.e., each FD must be strongly satisfied [3]). We also note that FDs involving set elements only on the RHS can also be captured by incorporating multivalued dependencies (MVD) [8] into the previous tuple-based approach. However, in general, FDs involving set elements cannot be captured using MVD. For example, FD 4 cannot be expressed using MVD because the set of author values must be considered together.

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 $\langle C_p, \text{LHS}\rangle$, where T satisfies the XML FD $\langle C_p, \text{LHS}, ./@\text{key}\rangle$.

For example $\langle C_{\texttt{state}}, \{./\texttt{name}\}\rangle$ is an XML Key for our running example, so is $\langle C_{\texttt{store}}, \{./\texttt{contact/name}, ./\texttt{contact/address}\}\rangle$.

This new notion of XML Key shares many similarities with the notion proposed by Buneman et al. in [4], 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). There are also important differences that we will explore in Sect. 4.

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 element.

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 [1], 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 Fig. 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 Sect. 2.1), and there exists a tuple class $C_{p'}$, where p' is the longest repeatable path that is a prefix of p, then each tuple in C_p corresponds to a distinct tuple in $C_{p'}$.*

Proof Each data element matching p' has at most one descendant matching p, and therefore each data element matching p has a distinct ancestor matching p'. Following Definitions 5 and 6, data elements matching p and p' have one-on-one correspondence to tuples in C_p and $C_{p'}$, respectively. Hence, each tuple in C_p corresponds to a distinct tuple in $C_{p'}$. □

A direct implication of Theorem 1 is that C_p is no longer necessary for the purpose of expressing FDs. Consider Fig. 4, which illustrates example tuples in both $C_{\texttt{contact}}$ and $C_{\texttt{store}}$. Each tuple in $C_{\texttt{contact}}$ has a distinct corresponding tuple in $C_{\texttt{store}}$. A comparison between two tuples in $C_{\texttt{contact}}$ can also be performed on their corresponding tuples in $C_{\texttt{store}}$, obtaining the same result. Therefore, all

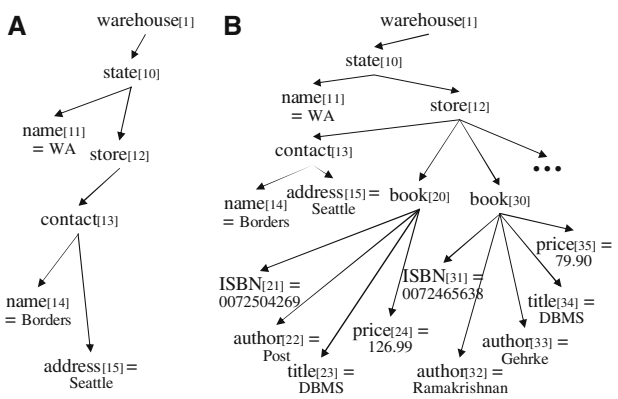

Fig. 4 Example tuples in *non-essential* tuple class $C_{\texttt{contact}}$ (**a**) and $C_{\texttt{store}}$ (**b**). In contrast, $C_{\texttt{book}}$ (with an example tuple in Fig. 3) is an essential tuple class

FDs under $C_{\texttt{contact}}$ can be expressed under $C_{\texttt{store}}$ with the same semantics (tuples in $C_{\texttt{store}}$ without contact do not affect this conclusion because missing elements are treated as unknowns). The reverse, however, is not true. FDs that are expressible under $C_{\texttt{store}}$ may refer to data elements that do not exist in tuples in $C_{\texttt{contact}}$, and are therefore not expressible under $C_{\texttt{contact}}$.

We call $C_{p'}$ the *lowest-repeatable-ancestor tuple class* of C_p. Since only tuple classes with repeatable pivot paths are essential for fully expressing all (non-redundant) XML FDs, we call them *essential tuple classes*. Intuitively, each essential tuple class corresponds to a distinct set element in the schema.

3.2.3 Structurally redundant XML FDs

Theorem 2 *Let $FD = \langle C_p, LHS, RHS\rangle$, if none of the paths in LHS and RHS specifies a data element that is a descendant of the pivot node in the tuple, then FD holds on a data tree T if and only if $FD' = \langle C_{p'}, LHS', RHS'\rangle$ holds on T, where $C_{p'}$ is the lowest-repeatable-ancestor tuple class of C_p, and paths in LHS' and RHS' are equivalent to paths in LHS and RHS (i.e., they correspond to the same absolute paths).*

Proof Each tuple in C_p has a corresponding tuple in $C_{p'}$ and tuples with sibling pivot nodes in C_p correspond to the same tuple in $C_{p'}$. If FD' is satisfied, then FD is not violated by two tuples with non-sibling pivot nodes. Because two tuples with sibling pivot nodes never violates FD (they share the same LHS and RHS elements), FD is satisfied. The reverse direction can be proved similarly. □

This *structural redundancy* is best illustrated by the following example FD, which is structurally redundant to FD 1 in Sect. 3.1:

FD 5 *{*`../ISBN`*}* → `../title` *w.r.t.* $C_{\texttt{author}}$

For the purpose of verifying FD, tuples in C_p with sibling pivot nodes are redundant to each other. Such redundancies are naturally eliminated in FD'. As a result, we consider FDs like FD 5 as structurally redundant and therefore uninteresting.

Another group of uninteresting FDs are those with an RHS path that does not match any descendant of the tuple pivot node, but with at least one LHS path that does match a descendant of the tuple pivot node. The satisfaction of such FDs either does not indicate redundancies or it indicates redundancies that are almost always indicated by other FDs. In the former case, for example, the satisfaction of {`../@key`, `./ISBN`} → `../contact/name` w.r.t. $C_{\texttt{book}}$ cannot cause any redundancy because for any two $C_{\texttt{book}}$ tuples with matching LHS, the RHS will always point to the same data element. In the latter case, for example, if {`./ISBN`} → `../contact/name` w.r.t. $C_{\texttt{book}}$ were satisfied, it would have meant that any two stores selling two books with the same ISBN would have the same name. In most scenarios, this means that all the stores in the database have the same name, which is easily detected through {`../../@key`} → `./contact/name` w.r.t. $C_{\texttt{store}}$.

3.2.4 Interesting XML FDs

Definition 10 (*Interesting XML FD*) An XML FD = $\langle C_p, LHS, RHS\rangle$ is interesting if it satisfies the following conditions:

- RHS ∉ LHS;
- C_p is an essential tuple class;
- RHS matches to descendant(s) of the pivot node.

In summary, an interesting XML FD is a non-trivial XML FD with an essential tuple class, and is not structurally redundant to any other XML FD. We note here that Definition 10 focuses on distinguishing interesting XML FDs from uninteresting ones based on the FD specification alone. As a result, the second condition of Definition 9 is not incorporated here: checking for triviality based on the second condition requires either examining the schema or checking the data directly.

3.3 XML data redundancy

Definition 11 (*XML data redundancy*) A data tree T contains a redundancy if and only if T satisfies an interesting XML FD $\langle C_p$, LHS, RHS$\rangle$, but does not satisfy the XML Key $\langle C_p$, LHS$\rangle$.

Intuitively, if $\langle C_p$, LHS$\rangle$ is not a key for T, then there exist two distinct tuples in C_p that share the same LHS. Since T satisfies $\langle C_p$, LHS, RHS$\rangle$, the RHS paths of the two tuples must be value equal. However, according to Definitions 5 and 10, the RHS paths match distinct data elements (because they are descendants of two distinct pivot nodes), which are therefore redundantly stored. For example, the data tree in Fig. 1 contains redundancies due to the satisfaction of FDs 1 and 3, where the book ISBN determines the title and author, but cannot uniquely identify an individual book in the set of books.

4 XML normal form: GTT-XNF

The definition of our new XML normal form, called GTT-XNF, naturally follows Definition 11 and is inspired by the XNF normal form defined in [1]. Formally, we have:

Definition 12 (*GTT-XNF*) An XML schema S is in GTT-XNF given the set of all satisfied interesting XML FDs if and only if for each such XML FD $(\langle C_p, LHS, RHS\rangle)$, $\langle C_p, LHS\rangle$ is an XML key.

Intuitively, GTT-XNF disallows any satisfied interesting XML FD that indicates data redundancies. The set of *all* satisfied interesting XML FDs needs to be either derived from an initial set of FDs specified independent of any database (XML FD inference) or extracted from the databases (XML FD detection). Section 5 describes XML FD detection, and here we briefly discuss XML FD inference.

The set of all interesting XML FDs for a given tuple class C_p can be derived from existing interesting XML FDs for C_p using the following inference rules. Those rules are similar to the Armstrong's axioms in the relational case, which are used to compute the closure of relational FDs [2].

Rule 1 (Reflexivity) $LHS \rightarrow P_1$ *w.r.t.* C_p *is satisfied if* $P_1 \subseteq LHS$.

Rule 2 (Augmentation) $LHS \rightarrow P_1$ *w.r.t.* $C_p \Rightarrow \{LHS, P_2\} \rightarrow P_1$ *w.r.t.* C_p.

Rule 3 (Transitivity) $LHS \rightarrow P_1$ *w.r.t.* $C_p \wedge \ldots \wedge LHS \rightarrow P_n$ *w.r.t.* $C_p \wedge \{P_1, \ldots, P_n\} \rightarrow P$ *w.r.t.* $C_p \Rightarrow LHS \rightarrow P$ *w.r.t.* C_p.

Deriving interesting XML FDs across different tuple classes, however, is more difficult. In general, no such axiom exists because the semantics of each path expression can change when their associated tuple class changes. For example, path `/warehouse/state/store/ book/title` matches a set of data elements within tuple class $C_{\texttt{store}}$, but a single data element within tuple class $C_{\texttt{book}}$. As a result, deriving *all* interesting XML FDs from an existing set of interesting XML FDs is often very difficult. This is in agreement with the nonaxiomatizability of XML FDs shown in [1]. Fortunately, XML FD inference is often not necessary since XML FD detection is more practical and fits the casual nature of XML schema design.

4.1 Comparison with previous key notion

Before we explore the differences between GTT-XNF and previously proposed normal forms, we analyze the difference between our XML Key notion and the notion proposed by Buneman et al. [4] first.

In [4], an XML Key is defined as $(Q, (Q', S))$, where Q is a path specifying a set of data elements that are the roots of data subtrees, on which the key (Q', S) holds. Within the key, Q' is a path specifying the set of data elements whose identities are to be compared (target elements), and S is a set of paths specifying the set of data elements whose values are to be compared (condition elements). For example, in Fig. 1, if we want to express that within each store, ISBN is a key for book, here is what we will have: (`/warehouse/state/store`, (`./book`, {`./ISBN`})).

Fig. 5 An alternative schema with multiple hierarchies

```
warehouse: Rcd
  state: SetOf Rcd
    name: str
    auction: SetOf Rcd
      contact: Rcd
        name: str
      book: SetOf Rcd
        ISBN: str
        au: SetOf str
  store: SetOf Rcd
    contact: Rcd
      name: str
      address: str
    book: SetOf Rcd
      ISBN: str
      author: SetOf str
      title: str
      price: str
```

When the paths in Q and Q' are all simple paths (i.e., they do not contain XPath step "//", which can potentially match multiple schema elements), an XML Key in [4] can be transformed into our GTT-based notion in the following way: first, set the tuple class to be $C_{Q'}$; second, set the LHS to be $\{S \cup Q/@key\}$. For example, the above relative key can be specified as $\langle C_{\texttt{book}}, \{\texttt{./ISBN}, \texttt{../@key}\}\rangle$ in our notion. Intuitively, the concept of "context" in [4] is transformed into one of the condition elements in LHS.

There are two main differences between the two key notions. First, in [4], Q' can involve an arbitrary path step and therefore specify a heterogeneous set of data elements. For example, consider an alternative schema in Fig. 5, Q' can be set to `/warehouse/state//book` and therefore match to book elements under both store and auction. In GTT-based XML Key notion, we only allow simple paths when specifying tuple classes, and therefore cannot express a key that holds for both sets of book elements. While allowing arbitrary paths for tuple classes gives us more powerful semantics, it introduces significant complexities into the specification of XML FDs. This is because a path in LHS that refers to ancestors of the pivot node can now match to a heterogeneous set of data elements as well. Furthermore, the redundancy detection algorithms will become significantly more complex when heterogeneous tree tuples need to be considered. We consider the benefits of this more flexible semantics not worth the significant complexities it introduces.

Second, the specification of the "context" of relative key is more flexible in GTT-based notion. While in [4], Q can only specify a path expression, we can in fact introduce more complex conditions. For example, consider the original example in Figs. 1 and 2, the key $\langle C_{\texttt{book}}, \{\texttt{./ISBN}, \texttt{../contact/name}\}\rangle$ specifies that ISBN is a key for book under all stores

with the same name. Such relative notion is difficult, if not impossible, to specify with the definition of [4].

4.2 Comparison with previous normal form notion

XNF is a normal form introduced by Arenas and Libkin [1] and our GTT-XNF generalizes XNF by allowing more flexible XML FD notions. As mentioned before, the main difference between the two XML FD notions is that XNF builds a single set of tree tuples by separating data elements of the same schema element at all levels, while GTT-XNF builds multiple sets of tree tuples, and within each set, data elements of the same schema element are only separated if they are not descendants of the pivot nodes.

Proposition 1 *For any given XML FD that is expressible under XNF, it is either expressible under GTT-XNF, or its satisfaction is trivially implied from the satisfaction of an XML FD that is expressible under GTT-XNF.*

Proof We prove the simple case first, where the XML schema contains a single hierarchy. In a "single-hierarchy" schema, every pair of set schema elements are in an ancestor-descendant relationship. For example, the schema in Fig. 2 is a single hierarchy schema. It is easy to see that, in this case, the set of tree tuples generated in XNF is exactly the set of generalized tree tuples within the essential tuple class whose pivot path matches the lowest set schema element (e.g., $C_{\texttt{author}}$). As a result, all FDs expressible under XNF is expressible under GTT-XNF, although not all such FDs are considered interesting (see Sect. 3.2.4).

When the schema contains more than one hierarchy, tree tuples generated in XNF do not appear as generalized tree tuples in GTT-XNF. For example, consider the multi-hierarchy schema in Fig. 5, both `auction` and `store` are set schema elements and they do not have an ancestor-descendant relationship. Each tree tuple generated in XNF will have exactly one store data element and exactly one auction data element. No tuple class, however, in GTT-XNF contains those tree tuples as defined in Definition 5.

We divide FDs under XNF in this case into two categories. First, we consider single-hierarchy FDs. A single-hierarchy FD is one in which all the path expressions in LHS and RHS only involve set schema elements within a single hierarchy. For example, {`/warehouse/state/ store/book/ISBN`}→`/warehouse/state/store/ book/title` is a single-hierarchy FD involving set schema elements `state`, `store`, `book`, all within a single hierarchy. Like in the simple case, those FDs are expressible in GTT-XNF within the appropriate tuple class (e.g., {`./ISBN`} → `./title` w.r.t. $C_{\texttt{/warehouse/state/store/book}}$).

Second, we consider FDs that involve set schema elements belong to multiple hierarchies. For example, {`/warehouse/state/name`, `/warehouse/state/ store/contact/name`}→`/warehouse/state/ auction/contact/name`. While we may be tempted to express this FD within tuple class $C_{\texttt{state}}$ since only tuples within $C_{\texttt{state}}$ contain both store and auction data elements, we in fact can not. Both `./store/contact/ name` and `./auction/contact/name` have set semantics (i.e., they match a set of data elements) for tuples within $C_{\texttt{state}}$, which is not the semantics of the FD under XNF because XNF, by definition, does not allow set semantics. However, the satisfaction of this FD, under XNF, can only mean one thing: *names of all the auctions under the same state are the same*. The reason is that according to Definitions 3.1 and 3.6 in [1], each store under a state will pair with every auction under the same state (and vice versa), and there is one tuple for each such pair. Consider the set of such tuples with the same store, but different auctions as shown in Fig. 6, the satisfaction of the above FD means those auctions all have the same name because they are all determined by the name of the same store. As a result, the satisfaction of this FD is trivially implied by the following FD under GTT-XNF: {`../name`} → `./contact/name` w.r.t. $C_{\texttt{auction}}$.

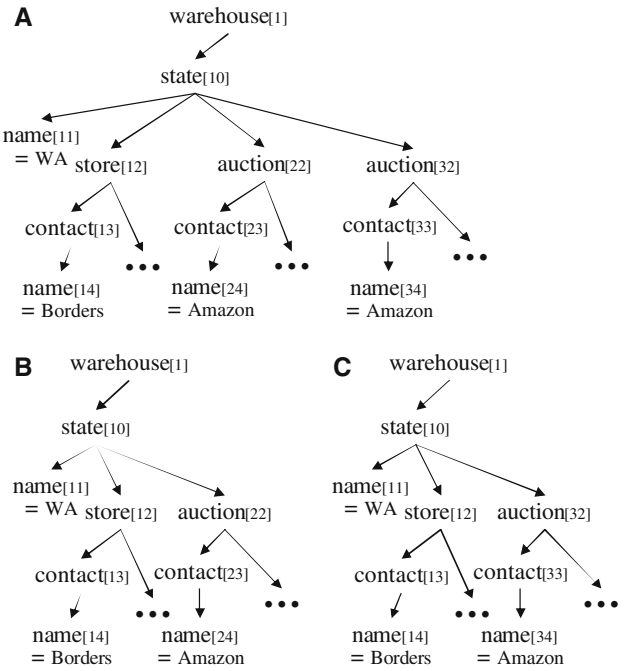

Fig. 6 Tuples under XNF with multiple hierarchies. (**a**) The partial data tree conform to the schema in Fig. 5. (**B**, **C**) Two of the tree tuples generated from the data tree that contain the same store

Let the multi-hierarchy FD expressible under XNF be $\{LHS_1 \cup LHS_2\} \rightarrow RHS$, where LHS_1 contains paths within the same hierarchy as RHS and LHS_2 contains all other paths in different hierarchies. Let FD $\{LHS_1\} \rightarrow RHS$ w.r.t. C_p, where p is the longest repeatable path that is a prefix of RHS, be an FD under GTT-XNF. The satisfaction of the former is always implied by the satisfaction of the latter.

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
...	...	...	...	...	...	...	...	...	...	...	...

Fig. 7 Example flat tuples in the flat representation of the XML data in Fig. 1

(If LHS_1 is empty in the original FD, we set LHS_1 in the new FD to be `/@key`, meaning the RHS is a constant throughout the entire database).

Theorem 3 *Any data redundancy that can be detected based on the XML FD notion in XNF can be detected based on the XML FD notion in GTT-XNF.*

Proof According to Definition 5.1 in [1], the RHS of a satisfied non-trivial FD is redundantly stored if the LHS of the FD does not determine the identity (i.e., `@key`) of the parent data element of the RHS. If the FD is expressible under both XNF and GTT-XNF, and it is interesting under GTT-XNF, then this definition and our Definition 12 express the same semantics. If the FD is not expressible under XNF, we identify a new FD with the same RHS under GTT-XNF that implies this FD (as discussed in the previous paragraph), and the redundancy of the RHS can then be detected based on this new FD (see Sect. 5). □

5 Detecting XML data redundancy

In this section, we show how data redundancies in XML can be efficiently detected through the discovery of satisfied interesting XML FDs and Keys. Based on Definition 11, we design *DiscoverXFD*, an algorithm to discover interesting XML FDs with non-key LHSs, and several related algorithms.

5.1 XML data representation

Flat Representation. The XML FD notion proposed in [1] 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 Fig. 7, 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 [1]. For example, the tuple in Fig. 3a is represented by the second tuple in Fig. 7.

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

R_{root}

@key	*parent*
1	⊥

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	⊥

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

Fig. 8 Example essential tuples in the hierarchical representation of the XML data in Fig. 1

(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 Fig. 7 would double.

Hierarchical representation. Inspired by the notion of essential tuple class (Sect. 3.2.2), and the concept of nested relation [18], a more compact representation of the XML data can be adopted. As shown in Fig. 8, the original XML data tree can be converted into a set of relations based on the original XML schema, 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 Fig. 3b, 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*.

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 Fig. 7. Therefore, each individual relation in Fig. 8 is considerably smaller than the single relation in Fig. 7 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 Sect. 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 Sect. 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 (Sect. 5.3) and FDs involving set elements (Sect. 5.4) based on the concepts of *partition target* and *set partition*. Section 5.5 analyzes the complexities of those algorithms. We first briefly describe how relational algorithms are applied to discover intra-relation FDs.

5.2 Discovering intra-relation FDs

The algorithm for discovering intra-relation FDs is adopt-ed from existing partition-based algorithms, e.g., TANE [11], Dep-Miner [15], and FUN [19]. There are two main data structures: *attribute partition* and *attribute set lattice*.

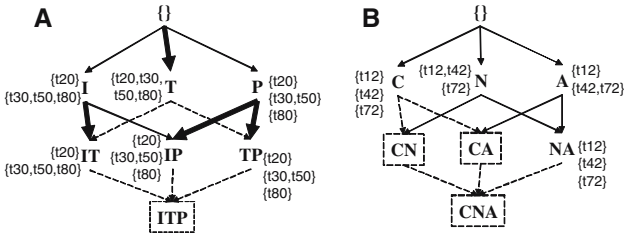

Fig. 9 Example attribute set lattices for R_{book} (**a**) and R_{store} (**b**). I, T, P, C, N, A stand for `ISBN`, `title`, `price`, `contact`, `contact/name`, `contact/address`, respectively. Shown along selected nodes are the attribute partitions. *Bold edges* correspond to satisfied FDs. *Dashed 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 that 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 if and only if $\Pi_{LHS} \hookrightarrow \Pi_{RHS}$ in R_p.*

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

Lemma 1 is straightforward and Lemma 2 is true because $\Pi_X \hookrightarrow \Pi_Y$ if and only if $\Pi_{X \cup Y} = \Pi_X$. Intuitively, Lemmas 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 Fig. 9, 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 Fig. 10) 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

[4] All examples are based on the data in Fig. 8.

[5] If a group contains only one tuple, it can be removed from the partition, resulting in a *striped* partition [11]. 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 Π_∅, Π_{a_1}, ..., Π_{a_n} from R_p
 2. Init. Keys = ∅, FDs = ∅, Queue Q = ∅, AttributeSet A = ∅;
 3. for i = 1, ..., n: Q.enqueue({a_i}); // the single attribute nodes
 4. while Q ≠ ∅:
 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 ∈ Ls: Π_A = Π_{A−A_1} • Π_{A_1};
10.     if Ls.size>=2: let A_1, A_2 ∈ Ls: Π_A = Π_{A_1} • Π_{A_2};
11.   if Π_A.maxGrpSize==1: Keys.add(A); continue;
12.   foreach A_L ∈ Ls: // A_L is the LHS
13.     let r = A−A_L; // r is the RHS
14.     if Π_{A_L} == Π_A: FDs.add(A_L→r);
15.   let a_k be the last attribute in A;
16.   for i = k+1, ..., n:
17.     A' = A∪a_i;
18.     if there is no K∈Keys, such that K⊂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 = ∅, the set of LHSs to be returned
20. foreach a ∈ A:
21.   let A_L = A-a;
22.   foreach L→r ∈ FDs:
23.     if a==r and A_L⊂L: continue;
24.     else if A⊂L: continue;
25.     else Ls.add(A_L);
26. return Ls
```

Fig. 10 Algorithm DiscoverFD

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, see [19]).

Since the objective 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 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 Fig. 9a, 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 Fig. 9b, nodes CN, CA, and CNA are removed because C is an XML Key.

5.3 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 $fd_1 = LHS' \cup \{\texttt{p'/parent}\} \rightarrow RHS$ 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} 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 (Fig. 11).

Algorithm DiscoverXFD is designed based on the above lemmas. 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 (lines 5–6: children relations are visited before the parent relation). At each relation, the algorithm accomplishes two things by employing the data structure *partition target* (shown in Fig. 12): First, it detects 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, it detects 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 Fig. 12) and updated as

Algorithm DiscoverXFD:
Input: R_p and pointers to its children relations,
Keys, the set of discovered keys so far,
FDs, the set of discovered FDs so far
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;

// bottom up recursively visit all relations
5. **foreach** child relation R of R_p:
6. PTs.addSet(DiscoverXFD(R, Keys, FDs));
// Q maintains the attribute sets to be examined
7. **for** $i = 1, ..., n$: Q.enqueue($\{a_i\}$);
// convert children PartitionTargets into the current level
8. **foreach** pt∈PTs:
9. pt′ = updatePT(ID_p, pt, $\Pi_\emptyset$); // Figure 12
10. **if** pt′ ≠ NULL: resultPTs.add(pt′);

// examine each generated attribute set in the while loop
11. **while** Q ≠ ∅:
12. A = Q.dequeue();
// generate Π_A if the partition is not materialized yet
// this block is the same as Figure 10 line 6-10
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 (no group in A contains multiple tuples),
// 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**;

// A is not a Key,
// check if Π_A satisfy any PT from children 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, Π_{A_L}, Π_A, C_p); // C_p: the tuple class
37. **if** pt ≠ NULL: resultPTs.add(pt);

// generate additional attribute sets
// this block is the same as Figure 10 line 15-18
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 10 without line 24.

Fig. 11 Algorithm DiscoverXFD

Function createPT(ID_p, Π_{A_L}, Π_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** $g_1 \in \Pi_{A_L}$:
5. **foreach** $g_2 \in \Pi_A$ 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** $t_1 \in g_1$, $t_2 \in g_2$:
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** $(t_1 \neq t_2) \in$ 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** $(t_1 \neq t_2) \in$ 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

Fig. 12 Utility functions

the algorithm moves up the hierarchies (Function updatePT in Fig. 12).

The details of the algorithm are shown in Figs. 11 and 12. 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 Fig. 8. When visiting R_{book}, the algorithm detects that $\Pi_{\{ISBN\}}$ is not the same as $\Pi_{\{ISBN, price\}}$ (see Fig. 9a), which means that {`./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} is created, with its FDTarget being {`t12≠t72`, `t42≠t72`} and KeyTarget being {`t12≠t42`}. The algorithm then visits R_{store} and examines its attribute partitions. In particular, in $\Pi_{contact/name}$ (see Fig. 9b), `t72` is separated

from t12 and t42, which means that 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 that the KeyTarget is not satisfied. As a result, {../contact/name, ./ISBN} → ./price w.r.t. C_{book} is reported as an inter-relation FD.

5.4 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 (Fig. 13) through a simple example based on the data in Fig. 8. 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} (lines 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 Sect. 5.2, 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.

Algorithm CreateSetPartition
Input: Π^{child}_A, the partition on attribute A in R_{child}
ID, the index maps *@key* to *parent* in R_{child}
1. Init. Π^{parent}_A as a single group of all distinct *parent*s in R_{child}
2. **foreach** $g \in \Pi^{child}_A$:
3. **foreach** $t \in g$: $t = ID$.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 if and only if cnt(t_1) = cnt(t_2) in g
5. **foreach** $g' \in \Pi^{parent}_A$, $g'' \in G$:
6. divide g' into g'_1, g'_2, where $g'_1 = g''$ and $g'_2 = g' - g''$
Output: Π^{parent}_A, the set partition on A in R_{parent}

Fig. 13 Algorithm CreateSetPartition

5.5 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 Sect. 7), 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 cannot 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.

6 Schema normalization

Given the set of discovered redundancies, our next objective is to eliminate those redundancies from the database. This redundancy elimination process involves three phases. The first phase is *schema normalization*, where the original schema is refined into a new schema such that all redundancy-indicating FDs are *eliminated*.[6] The second phase is *mapping generation*, where a schema mapping is established between the original schema and the new schema. The last phase is *data transformation*, where the original database is transformed into the new format based on the schema mapping. The last two phases are studied extensively in the context of data integration [22], and therefore we only focus on *schema normalization* in this paper. In Sect. 6.1, we describe how individual redundancy-indicating XML FDs can be eliminated by modifying the schema. In Sect. 6.2, we present the overall algorithm for schema normalization, which uses a bottom-up strategy to systematically group and eliminate redundancy-indicating FDs.

For the rest of this section, we use the XML document and schema in Figs. 1 and 2 as an example, and assume that we have discovered a set Σ of redundancy-indicating FDs that contains the following two FDs, both of which are interesting (Definition 10) and minimal (i.e., not implied by other FDs).

$F_1 = \{./\texttt{ISBN}\} \rightarrow ./\texttt{title}$ w.r.t. $C_{\texttt{book}}$;

$F_2 = \{../../\texttt{name}, ../\texttt{contact/name}, ./\texttt{ISBN}\} \rightarrow ./\texttt{price}$ w.r.t. $C_{\texttt{book}}$;

$F_3 = \{./\texttt{ISBN}\} \rightarrow ./\texttt{author}$ w.r.t. $C_{\texttt{book}}$;

We further assume that $\langle C_{\texttt{store}}, \{../\texttt{name}, ./\texttt{contact/name}\}$ is an XML Key in our example.

6.1 Eliminating redundancy-indicating FDs

As defined in Definition 11, if an XML FD, $\langle C_p, LHS, RHS\rangle$, holds on a database, while the XML Key $\langle C_p, LHS\rangle$ does not, the data elements that are represented by RHS are redundantly stored. To eliminate such an FD, we move the schema element corresponding to RHS into an appropriate new schema location, such that those data elements are no longer redundantly stored. We first classify those FDs into two categories: *local* and *global*.

Definition 13 (*Local/global XML FD*) An XML FD, $\langle C_p, LHS, RHS\rangle$, is *local* if there exists $LHS' \subset LHS$ such that $\langle C_{p'}, LHS'\rangle$ is an XML Key, where $C_{p'}$ is an ancestor tuple class of C_p (i.e., p' is a prefix of p). Otherwise, the FD is *global*.

[6] Eliminate here means adjusting the schema such that those FDs no longer indicate redundancy. FDs are a property of the database and, as a result, can never be eliminated.

The distinction between global and local FDs can be illustrated through some examples. Consider F_1, which is a global FD because no subset of its LHS is a key for any tuple class above $C_{\texttt{book}}$. Intuitively, this FD means any two `book`s, *regardless whether they are under the same* `store` *or* `state`, if they have the same `ISBN`, then they will have the same `title`. Consider F_2, which is a local FD because {`../name, ./contact/name`} is a key for $C_{\texttt{store}}$. Because the state `name` and store `name` uniquely identifies each `store`, this FD intuitively means any two `book`s, if they have the same `ISBN`, they will have the same `price`, *as long as they are under the same* `store`. In other words, unlike global FDs, local FDs are satisfied within a single subtree, instead of across multiple subtrees. Because of this distinction, we adopt different procedures for eliminating global and local FDs, with the goal of minimal changes to the original schema. Both procedures de-couple the RHS schema element of the redundancy-indicating FD from the other schema elements within the tuple class, and create a new schema element that contains the de-coupled RHS schema element. The differences are *where is the new schema element created* and *what are the type of this new schema element*.

Procedure 1 (*Eliminate global FD*) *Let* $F = \{P_1, \ldots, P_n\} \rightarrow P_r$ *w.r.t.* C_p *be a redundancy-indicating global FD on Schema* $S_{\texttt{root}}$*;* $\{e_i \mid i \in [1,n]\}$ *and* $\{\tau_i \mid i \in [1,n]\}$ *be the sets of schema element labels and types, respectively, associated with each* P_i*;* e_r *and* τ_r *be the schema element label and type, respectively, associated with* P_r*;* $\tau_{\texttt{parent}}$ *be the schema element type of the parent element of* P_r*;* $\tau_{\texttt{root}} = \mathsf{Rcd}[e'_1 : \tau'_1, \ldots, e'_m : \tau'_m]$ *be the element type of the* `root` *element. We eliminate the redundancy by the following procedure:*

- *Create a new schema element with label* $e_{\texttt{new}}$ *and type* $\tau_{\texttt{new}} = \mathsf{SetOf}\ \mathsf{Rcd}[e_1 : \tau_1, \ldots, e_n : \tau_n, e_r : \tau_r]$
- *Set* $\tau_{\texttt{root}} = \mathsf{Rcd}[e'_1 : \tau'_1, \ldots, e'_m : \tau'_m, e_{\texttt{new}} : \tau_{\texttt{new}}]$
- *Remove* $(e_r : \tau_r)$ *from* $\tau_{\texttt{parent}}$.

To eliminate a global FD (e.g., F_1), we create a new schema element containing both its LHS elements (e.g., `ISBN`) and RHS element (e.g., `title`), and put this new element under the root. The RHS element is then removed from its original position. Figure 14 illustrates the new schema after eliminating F_1.

Creating inclusion constraint. To avoid the loss of associations between P_r and paths other than P_i ($i \in [1,n]$), an implicit value-based inclusion constraint will now be created on the generalized tree tuples in C_p and $C_{e_{\texttt{new}}}$: Let P'_i ($i \in [1,n]$) be the set of paths (relative to $e_{\texttt{new}}$) for the schema elements e_i ($i \in [1,n]$) under $e_{\texttt{new}}$; for each tuple $t \in C_p$, there exists a tuple $t' \in C_{e_{\texttt{new}}}$, such that $t.P_i =_{nv} t'.P'_i$

Fig. 14 Resulting schema after eliminating F_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
        price: str
  new-book: SetOf Rcd
    ISBN: str
    title: str
```

Fig. 15 Resulting schema after eliminating F_2

```
warehouse: Rcd
  state: SetOf Rcd
    name: str
    store: SetOf Rcd
      contact: Rcd
        name: str
        address: str
      book: SetOf Rcd
        ISBN: str
        title: str
        author: SetOf str
      new-book: SetOf Rcd
        ISBN: str
        price: str
```

for all $i \in [1, n]$. This constraint is very much like the inclusion constraints that are adopted in relational data model.

Adjusting FD. We first remove F from Σ. The semantics of F is now captured by the new FD: $\{P_1, \ldots, P_n\} \rightarrow P_r$ w.r.t. $C_{\texttt{new}}$. However, since $\{P_1, \ldots, P_n\}$ is a key for $C_{\texttt{new}}$, this new FD is not redundancy indicating and therefore does not need to be added to Σ. We then remove all FDs in Σ that are affected by the move of P_r. For example, if the FD $F_4 =$ {`./author, ./title`} $\rightarrow$ `./ISBN` w.r.t. $C_{\texttt{book}}$ is in Σ, it will be removed because it is no longer valid. It is safe to do so because `ISBN` is no longer redundant and the semantics of this FD is now captured through the above-mentioned inclusion constraint.

Procedure 2 (*Eliminate local FD*) *Let* $F = \{P_1, \ldots, P_{k-1}, P_k, \ldots, P_n\} \rightarrow P_r$ *w.r.t.* C_p *be a redundancy indicating local FD on Schema* $S_{\texttt{root}}$, *where* $\{P_1, \ldots, P_{k-1}\}$ *is the key for* $C_{p'}$, $C_{p'}$ *is an ancestor tuple class of* C_p, *and there is no other subset* L *of* $\{P_i \mid i \in [1, n]\}$ *such that* L *is a key for* $C_{p''}$, $C_{p''}$ *is an ancestor tuple class of* C_p *and a descendant tuple class of* $C_{p'}$ *(i.e.,* $C_{p'}$ *is the lowest tuple class that can be identified*);

Furthermore, let $\{e_i \mid i \in [k, n]\}$ *and* $\{\tau_i \mid i \in [k, n]\}$ *be the sets of schema element labels and types, respectively, associated with each* P_i; e_r *and* τ_r *be the schema element label and type, respectively, associated with* P_r; $\tau_{\texttt{parent}}$ *be the schema element type of the parent element of* P_r; $\tau_{\texttt{p}'} = \mathsf{Rcd}[e'_1 : \tau'_1, \ldots, e'_m : \tau'_m]$ *be the element type of the schema element corresponding to the pivot path of* $C_{p'}$. *We eliminate the redundancy by the following procedure:*

- *Create a new schema element with label* $e_{\texttt{new}}$ *and type* $\tau_{\texttt{new}} = \mathsf{SetOf\ Rcd}[e_k : \tau_k, \ldots, e_n : \tau_n, e_r : \tau_r]$
- *Set* $\tau_{\texttt{p}'} = \mathsf{Rcd}[e'_1 : \tau'_1, \ldots, e'_m : \tau'_m, e_{\texttt{new}} : \tau_{\texttt{new}}]$
- *Remove* $(e_r : \tau_r)$ *from* $\tau_{\texttt{parent}}$.

To eliminate a local FD (e.g., F_2), we create a new schema element containing the subset of its LHS elements (e.g., `ISBN`) that are not part of the key for the ancestor tuple class (e.g., $C_{\texttt{store}}$) and RHS element (e.g., `title`), and put this new element under the schema element corresponding to the pivot path of the ancestor tuple class (e.g., `/warehouse/state/store`). The RHS element is then removed from its original position. By creating the new schema element under the non-`root` ancestor, fewer elements needs to be copied under the new schema element and the overall schema becomes simpler. Figure 15 illustrates the new schema after eliminating F_2.

Just like in the case of eliminating local FDs, after the modification of the schema, we impose a value-based inclusion constraint on tuples in C_p and $C_{\texttt{new}}$, and remove any FD that is affected by the move of P_r.

Special case for Procedure 2. There is a special case for eliminating local FDs, that is when the entire LHS of the FD is a key for some ancestor tuple class. A classic example is the DBLP schema, where the `year` of an `article` (tuples in $C_{\texttt{article}}$) is determined by the identity (i.e., `@key`) of the `issue` containing the `article`. In this case, instead of creating a new schema element containing a single element `year`, we can simply move `year` directly under `issue`, making the result schema simpler. Note that by not creating a new (set) schema element, we can potentially introduce new redundancy-indicating FDs into Σ. For example, `year` can be determined by some non-key elements under `issue`, and therefore still be redundant. This redundancy will then be eliminated when we analyze FDs for $C_{\texttt{issue}}$.

6.2 Normalization algorithm

Figure 16 illustrates the overall schema normalization algorithm. The bulk of the algorithm is modifying the schema using either Procedures 1 or 2, which are described extensively before. There are several things worth mentioning here. First, we group FDs according to their LHS so that no unnecessary schema element is created. For example, consider our example FDs F_1 and F_3, both of which have the LHS {`./ISBN`}. If they are dealt with separately, two new schema element will be created: one contains `ISBN` and `title`, the other contains `ISBN` and `author`. By grouping these two FDs, only one new schema element needs to be created: one contains `ISBN`, `title`, and `author`.

Algorithm SchemaNormalization:
Input: Schema S, a set Σ of redundancy-indicating FDs,
a set Υ of XML Keys (to determine local vs. global FD).
1. Group FDs in Σ based on tuple class C_p and LHS,
order them according to decreasing depth of C_p (lowest first)
and increasing number of paths in LHS second (fewest first);
2. **while** Σ is not empty:
3. **let** $\mathcal{F}$ be the first set of FDs in Σ with the same LHS and C_p;
4. **let** F be the first FD in $\mathcal{F}$;
5. **if** F is local: Modify S by applying Procedure 2;
6. **else** Modify S by applying Procedure 1; // F is global
7. **foreach** additional $F' \in \mathcal{F}$:
8. Modify S in the same way by applying procedures 1 or 2,
but using the new schema element already created in
dealing with F'
9. remove all FDs in $\mathcal{F}$ from Σ;
10. **foreach** $F \in \Sigma$:
11. **if** F is no longer valid: remove F from Σ;
12. **if** F is now structurally-redundant:
13. convert F into its equivalent F' that is not structurally
redundant and add F' to Σ (see Theorem 2);
Output: Schema S, the modified redundancy-free schema

Fig. 16 Algorithm SchemaNormalization

Second, we process FDs according to the number of paths in their LHS as a heuristic strategy to reduce the storage cost. Consider F_1 and F_3, plus a new FD $F_5 = \{$`./title, ./author`$\} \rightarrow$ `./ISBN` w.r.t. $C_{\texttt{book}}$. If F_5 is processed first, then the elements `title` and `author`, instead of `ISBN`, will remain under `book`. Even though there will be no data redundancy, it is intuitively undesirable since `title` and `author` are more complex elements being stored at a lower hierarchy (`book` instead of `root`). Ordering FDs according to the number of paths in LHS ensures that F_1 and F_3 will be processed before F_5 and therefore only `ISBN` will remain under `book`.

Third, we process FDs according to the hierarchy depth of their tuple class (i.e., in a bottom-up fashion). This is because during the process of FDs for a lower hierarchy tuple class, redundancy-indicating FDs for a higher hierarchy tuple class may be created. Processing FDs in the hierarchical order allows us to deal with each tuple class at most once.

Finally, the algorithm terminates because each application of Procedures 1 and 2 either removes at least one redundancy-indicating FD, or converts one redundancy indicating FD into another one with a tuple class at a higher hierarchy (i.e., the special case for Procedure 2).

Figure 17 illustrates the final redundancy-free schema after eliminating all three FDs. Notice that eliminating F_1 and F_3 together results in only one new schema element (`new-book`), and that after the elimination of F_1 and F_3, F_2 is no longer redundancy indicating.

7 Experimental evaluation

We implemented the redundancy detection algorithm, DiscoverXFD, on top of Berkeley DB [23] using Java. The main data structures, including attribute partitions, partition

Fig. 17 Resulting schema after eliminating all three FDs

warehouse: Rcd
  state: SetOf Rcd
    name: str
    store: SetOf Rcd
      contact: Rcd
        name: str
        address: str
      book: SetOf Rcd
        ISBN: str
        price: str
  new-book: SetOf Rcd
    ISBN: str
    title: str
    author: SetOf str

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 2.0-GHz P4 CPU and 1 GB RAM, running Windows XP (SP2) and JRE 1.4.2. The JVM memory was 512 MB and the Berkeley DB cache size was 128 MB. For timing measurements, each experiment was run three times and the average reading was recorded.

7.1 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 [17] geography dataset; the human subset of PIR protein information dataset from protein information resource [21]; the DBLP [14] 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 skipping 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

Table 1 Statistics of real life datasets

	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	1, 001.1	3, 736.4
Document size (in MB)	1.2	31.8	133.8

Table 2 Analyzing real life datasets

	Mondial	PIR	DBLP
Loading time (seconds)	0.6	18.0	55.9
Partition time (seconds)	6.0	20.2	79.9
Discovery time (seconds)	11.1	253.6	1,093.5
Intra-relation Keys	98	41	205
Inter-relation Keys	25	6	5
Intra-relation FDs	21 (0)	73 (12)	313 (16)
Inter-relation FDs	9 (0)	8 (2)	0 (0)
Total discovered FDs	30	81	313

When counting FDs, only the ones whose LHS is not a key are counted—the numbers of discovered FDs therefore refer to only redundancy indicating FDs. Numbers in parentheses are the number of FDs that involve set elements

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}

Fig. 18 Partial schema of the Mondial dataset and an example redundancy-indicating inter-relation FD

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 s for Mondial to 20 min for DBLP, demonstrating the practicality of the system.

More importantly, data redundancies were detected in all three datasets, as shown in Table 2. An example redundancy-indicating FD in Mondial is shown in Fig. 18. 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. It is also worth noting that, for the PIR dataset, a significant number of discovered redundancy-indicating FDs (17% of all discovered FDs) involve set elements—those FDs can not be

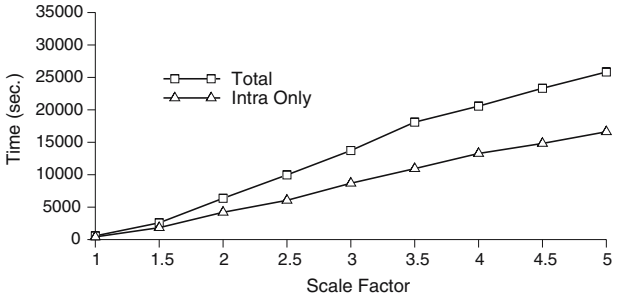

Fig. 19 Cost of redundancy detection on XMark datasets with increasing scale factors

captured by previous XML FD proposals. Such *set* FDs are less common for Mondial and DBLP, presumably because the data those are representing inherently do not contain many such redundancies. 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 104,507 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.[7]

We also analyzed the effect of FD grouping (Step 1 in Fig. 16) of the SchemaNormalization algorithm by normalizing the PIR schema (given the set of discovered redundancy-indicating FDs) with or without FD grouping. While a total of 63 new schema elements are produced if we do not adopt FD grouping, only 23 new schema elements are produced when it is adopted.

7.2 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 100 MB. As shown in Fig. 19, the total time for redundancy detection (line Total) increases linearly with the scale factor, indicating that the system scales well with increasing data

[7] PIR has been replaced by the new UniProt database, whose design has taken into consideration our suggestions. A rough comparison between the two schemas show that an estimated 20% of the discovered FDs pointing to unintended redundancies, 30% are intended redundencies, the rest of FDs are no longer applicable in the new schema.

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

	S1	S2	S3	XMark (sf = 0)
Data elements	118	153	192	331
Rel. algorithm	0.9	6.0	128.2	>10,000
DiscoverXFD	2.1	2.3	2.5	4.2

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 (Fig. 19 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 Sect. 5.1, 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 objective, we implemented an alternative redundancy detection system, which converts XML data into flat representation and adopts the algorithm FUN [19][8] for FD and Key discovery (we made minor adjustments to avoid recording un-*interesting* FDs). We performed redundancy detection with this system on all three real-life datasets. Not surprisingly, it did not finish detection (within 24 h) 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.

8 Related work

Designing XML Keys and XML FDs were first addressed in [4,5,13], respectively. Formal definitions of XML FDs and Normal Forms were later proposed in [1] and [25], providing significant improvements over relational FDs in capturing XML data redundancies. However, as discussed at length in Sect. 4, those proposals are limited in their ability to capture redundancies involving set elements and many do not utilize the formal tuple-based semantics. The redundancy detection problem, one of our main contributions, is not addressed in any of the above studies. Integrity constraints (including keys) in XML were studied extensively in [4,5,9,10], which proposed several notions (e.g., element-value equality) that are used here and in many other studies. However, they do not adopt the tree tuple notion that we and [1] adopt here.

The hierarchical representation of XML data shares many similarities with nested relations [3]. In [18,20], data redundancies in nested relations are characterized using relational FDs and MVDs. However, as mentioned in Sect. 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 [11,15,19] 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 [6,24], which is a considerably simpler problem than our problem of redundancy detection through the discovery of FDs and Keys.

Designing normalization algorithms for schema refinement is an important problem and was studied in [1,25]. We follow their directions in this paper and design normalization algorithms that are based on the our proposed normal form GTT-XNF.

In many real world scenarios, FDs are not always satisfied—an FD may hold a large subset of the data, but not on a small number of tuples. Several studies have looked at this, most recently in [12]. While we do not consider this in the current study, data structures similar to those in [12] can potentially be adopted for the discovery of approximate XML FDs.

9 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 proposals and capture a comprehensive set of XML data redundancies, including in particular redundancies involving set elements. Based on those new notions, we proposed a new XML normal form GTT-XNF. We designed and implemented DiscoverXFD, the first XML data redundancy detection system through the discovery of XML FDs and Keys. We further designed a normalization algorithm that converts any XML schema into one in GTT-XNF given

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

the set of detected redundancy-indicating XML FDs. Experimental evaluation demonstrates that the system is practical in detecting redundancies in real datasets and scales well with increasing dataset size.

Acknowledgements This work was supported in part by the United States National Science Foundation (NSF) under grant IIS- 0438909 and by National Institutes of Health (NIH) under grant 1-U54- DA021519. We would like to thank the anonymous reviewers for their constructive suggestions.

References

1. Arenas, M., Libkin, L.: A normal form for XML documents. TODS **29**(1), 195–232 (2004)
2. Armstrong, W.: Dependency structures of database relationships. In: Proc. IFIP, pp. 580–583. North Holland (1974)
3. Atzeni, P., DeAntonellis, V.: Foundations of Databases. Benjamin Cummings (1993)
4. Buneman, P., Davidson, S., Fan, W., Hara, C., Tan, W.-C.: Keys for XML. In: Proc. WWW, pp. 201–210. Hong Kong, China (2001)
5. Buneman, P., Davidson, S., Fan, W., Hara, C., Tan, W.-C.: Reasoning about keys for XML. Inf. Syst. **28**(8), 1037–1063 (2003)
6. Chen, Y., Davidson, S., Zheng, Y.: XKvalidator: a constraint validator for XML. In: Proc. CIKM, pp. 446–452. McLean, VA (2002)
7. Codd, E.F.: A relational model of data for large shared data banks. Commun. ACM **13**(6), 377–387 (1970)
8. Fagin, R.: Multivalued dependencies and a new normal form for relational databases. TODS **3**, 262–278 (1977)
9. Fan, W., Libkin, L.: On XML integrity constraints in the presence of DTDs. J. ACM **49**(3), 368–406 (2002)
10. Fan, W., Simeon, J.: Integrity constraints for XML. JCSS **66**, 2554–291 (2003)
11. Huhtala, Y., Karkkainen, J., Porkka, P., Toivonen, H.: TANE: an efficient algorithm for discovering functional and approximate dependencies. Comput. J. **42**(2) (1999)
12. Ilyas, I., Markl, V., Haas, P., Brown, P., Aboulnaga, A.: CORDS: automatic discovery of correlations and soft functional dependencies. In: Proc. SIGMOD, pp. 647–658. Paris, France (2004)
13. Lee, M.L., Ling, T.W., Low, W.L.: Designing functional dependencies for XML. In: Proc. EDBT, pp. 124–141. Prague, Czech Republic (2002)
14. Ley, M.: DBLP Computer Science Bibliography. http://dblp.uni-trier.de/
15. Lopes, S., Petit, J.-M., Lakhal, L.: Efficient discovery of functional dependencies and Armstrong relations. In: Proc. EDBT, pp. 350–364. Konstanz, Germany (2000)
16. Mannila, H., Raiha, K.-J.: Dependency inference. In: Proc. VLDB, pp. 155–158. Brighton, England (1987)
17. May, W.: Information extraction and integration with Florid: the Mondial case study, (1999). http://www.dbis.informatik.uni-goettingen.de/lopix/lopix-mondial.html
18. Mok, W.Y., Ng, Y.-K., Embley, D.: A normal form for precisely characterizing redundancy in nested relations. TODS **21**(1), 77–106 (1996)
19. Novelli, N., Cicchetti, R.: Functional and embedded dependency inference: a data mining point of view. Inf. Syst. **26**, 477–506 (2001)
20. Ozsoyoglu, Z.M., Yuan, L.-Y.: A new normal form for nested relations. TODS **12**(1), 111–136 (1987)
21. PIR International Protein Sequence Database. http://pir.georgetown.edu/pirwww/search/textpsd.shtml (2006)
22. Popa, L., Velegrakis, Y., Miller, R., Hernández, M., Fagin, R.: Translating Web data. In: Proc. VLDB, pp. 598–609. Hong Kong, China (2002)
23. Sleepycat Software. http://www.sleepycat.com/ (2006)
24. Vincent, M., Liu, J.: Checking functional dependency satisfaction in XML. In: Proc. XSym, pp. 4–17. Trondheim, Norway (2005)
25. Vincent, M., Liu, J., Liu, C.: Strong functional dependencies and their application to normal forms in XML. TODS **29**(3), 445–462 (2004)
26. W3C. XML Schema. http://www.w3.org/TR/xmlschema-0/ (2004)