From XML to Relational Database

By

YAN, Men-hin

Supervised By :

Prof. Ada Wai-chee Fu

A Thesis Submitted in Parital Fulfilment
of the requirements for the degree of
Master of Philosophy
in
Computer Science & Engineering

June 2001

From XML to Relational Database

submitted by

YAN, Men-hin

for the degree of Master of Philosophy
at the Chinese University of Hong Kong

Abstract

XML - the eXtensible Markup Language - is rapidly becoming a new standard for data representation and exchange on the Internet. When viewing XML from a database point of view, it is possible to manage the content of the XML document using databases. Most of the work has been concentrated on building a semistructured database system based on the semistructured-characteristics of XML. Besides adopting such a new and immature technique, using a traditional relational database system seems to be another option. As a result, we try to explore the possibility of storing XML data into a relational database instead of a semistructured database. Since XML data and relational data are vastly different in nature, we try to optimize the use of an relational database to store XML data by proposing a relational schema extraction algorithm. The general idea of our algorithm is to first extract the *schema prototypes* from the DTD (DTD is essentially a grammar for XML) of the XML documents, then apply an existing functional dependency discovery technique, *TANE*, on the prototype relations. With the found dependencies in the XML data, the schema prototypes can be further decomposed into better relational schema following the traditional relational database design theory. To reduce the cost of our algorithm due to the exponential complexity in the number of attributes during the dependency discovery, we further propose several approaches to extract possible characteristics

in the XML data according to the DTD (DTD splitting) before going to the step of dependency. For smaller size DTD, the DTD-splitting appoarches perform well even without the step of dependency discovery. In order to further improve the design of the relational schema, we propose a new algorithm based on the idea of partition refinement for finding possible multivalued dependencies in the XML data, thus providing more useful information for producing the relational schema. To reduce the search space of the multivalued dependency discovery algorithm, several effective pruning techniques are introduced. Experiments are carried out to show the effectiveness of all of our proposed algorithms.

論文題目：從 XML 到 關係數據庫

作者：殷民軒

學校：香港中文大學

學系：計算機科學及工程學系

修讀學位：哲學碩士

摘要：

XML – 可擴展標記語言(Extensible Markup Language) – 正迅速地成為互聯網(Internet)上數據代表及交換的新標準。從數據庫(Database)的觀點來看 XML，用數據庫來管理 XML 數據乃可能的做法。大部分的研究皆專注於利用 XML 半結構化(semistructured)的特徵來建立半結構化數據庫系統。除了採用如此新而不成熟的技術，利用傳統的關係數據庫系統似乎是另一個選擇。因此，我們嘗試用關係數據庫(relational database)替代半結構化數據庫來儲存 XML 數據，並探索其中的可能性。由於 XML 數據與關係數據本質上有非常大的差異，因此我們提出抽取關係模式(extracting relational schema)的演算法，來試著優化使用關係數據庫儲存 XML 數據的做法。我們的演算法大致的概念為: 首先從 XML 文件的 DTD (DTD 本質上乃 XML 的語法) 中抽取出模式原型(schema prototype)，然後應用現存的發現函數依賴(functional dependency)技術 – TANE 於原型關係上。利用從 XML 數據所找出的依賴，我們可依循傳統的關係數據庫設計理論，進一步將模式原型分解成較佳的關係模式。我們的演算法在發現依賴時，在屬性(attribute)數量上存在著指數複雜性(exponential complexity)。為了減少這種指數代價，我們提出數種方法從而在進行發現函數依賴的步驟之前，預先在 DTD 裡抽取出 XML 數據中可能擁有的特質。為了再進一步改良關係模式的設計，我們根據分區求精(partition refinement)的概念，提出另一新演算法來於 XML 數據中找出多值依賴(multivalued dependency)，從而提供更多有利於生產關係模式的資訊。為了減少此演算法的搜索空間，我們引入數種有效的剪枝(pruning)技術。我們並用實驗結果顯示所有建議的演算法的效能。

Acknowledgments

First of all, I would like to thank Prof. Ada Wai-Chee Fu, my supervisor, for her guidance and patience. My research could not have been done reasonably without her insightful advice. For the past two years, she gave me encouragement, support, and guidance on my research and my thesis.

My great gratitude goes to Prof. Man-Hon Wong and Prof. Michael Lyu, who marked my term paper and gave me valuable suggestions.

I would like to thank the Department of Computer Science & Engineering, CUHK. It provides the best equipment and office environment required for high quality research to our students.

Finally, I wish to express my thanks to my fellow colleagues, who helped me in solving the all kind of problems during my research, and enlightened me to new research ideas. They are (not in order) Willis Wai-To Chan, Yin Ling Cheung, Ham Siu-Ham Wong and Roy Sheun-Ti Chan.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

The Extensible Markup Language (XML) is the universal format for structured documents and data on the Web [15]. It is derived from SGML [3] and it is expected to rapidly become a new standard for data representation and exchange on the Internet. Because XML was defined as a textual language rather than data model, a XML document has implicit order. Although an XML document can have no restrictions on tags, attribute names, or nesting patterns, it is expected that most XML documents will be accompanied by Document Type Definitions (DTDs) [15, 21]. DTD is essentially a grammar for restricting the tags and structure of a document. The Internet community expects most of the XML documents on the web will conform to DTDs in order to make the XML data fully functional [12, 13].

It is clear that the fast emerging XML will soon become a dominant standard for representing data in the World Wide Web. When compared to HTML, it is obvious that XML encoding provides information in a far more convenient and usable format from a data management perspective. Being a document markup language (in some sense a meta language), XML is mainly used for representing data in the form of documents. However, in the database point of view, XML data stored in the document will have only limited usage unless the data is stored

and managed in a database system.

1.1 Storing XML in Database Systems

Due to the nature of information on the Web and the inherent flexibility of XML, data encoded in XML may be semistructured [16]. Work from the database community in the area of semistructured data corresponds closely to XML [4, 17]. As a result, storing the XML data into a semistructured database system seems to be a straightforward solution, and there have been considerable activities in the semistructured community focussed upon developing these kind of semistructured database systems [43, 24, 27].

In theory, semistructured system would clearly work and it should work best with the tailored features for handling XML data. However, is it the only approach to take? It is still unclear if the approach of using such systems is going to find widespread acceptance in the near future. The techniques in semistructured database are still new and under exploration, and it may take a long time for semistructured database systems to be as well developed as relational database system (RDBMS) is. As a result, we consider using an RDBMS to store XML data to be another possible approach. Using RDBMS to store XML data not only can let us apply well-developed relational techniques on XML data, but also can let existing traditional data coexist with the XML data which makes it possible to build applications that involve both kinds of data with little extra effort.

Since XML data and relational data are vastly different in nature (semistructured vs. structured), we have to explore new methods in order to optimize the use of an RDBMS to store XML. The main concern in this problem is how to produce the relational schema from the XML data. Recently, several approaches

have been proposed. One strategy is to infer from the DTDs of the XML documents how the XML elements should be mapped into tables [49]. Another option is to analyze the pattern of the XML data and then extract the schema from it [22]. Yet another option is to use simple ad-hoc schemes based on the widely accepted graph model for semistructured data [25, 26]. The three approaches presented above have one thing in common: they try to produce the relation schema solely based on the structure (or pattern) in the XML data. We propose a new approach which also takes the characteristics of the XML data into consideration except examining the structure of data. To do so, we introduce the traditional relational concepts like functional dependency [20] and multivalued dependency [53]. Our algorithm can thus produces relation schema that is better refined with and the produced tables should be more suitable for managing and querying.

The general idea of our algorithm is to first extract the *schema prototypes* from the DTD of the XML documents, then apply existing functional dependency discovery (inference) techniques like [29] on the prototype relations. With the found dependencies in the XML data, the schema prototypes can be further decomposed into better relational schema which follows the traditional relational database design theory. To reduce the cost of our algorithm due to the exponential complexity in the number of attribute during the dependency discovery, we propose several approaches to extract possible characteristics in the XML data according to the DTD before going to the step of performing dependency discovery. The new algorithms are presented in Chapter 3. In order to further improve the design of the relational schema, we propose a new algorithm for finding possible multivalued dependencies in the XML data, thus providing more useful information for producing the relational schema. The new algorithm is presented in Chapter 4.

1.2 Outline of the Thesis

The thesis is orgranised as follows.

In Chapter 2, We first give an overview of XML and also DTD, which is important in our proposed algorithms for producing relational schema of XML data. Then we review and compare the related work in storing XML data into database, including storing XML data using special-purpose database, relational database...etc. We focus more on work that uses relational database to store XML data.

In Chapter 3 introduce our proposed algorithms for extracting relational schema from DTDs and the XML documents. First we introduce the general approach for producing relational schema for XML data proposed by us. Then we describe Global Extraction Algorithm and DTD-split Extraction Algorithm, both of which rely mainly on DTD together with dependency discovering technique. And finally we compare and analyze the experiment results of our algorithms on real life XML data as well as sythetic XML data.

In Chapter 4, we introduce the new algorithm for discovering multivalued dependencies from relational data. First we introduce the partition technique involved in our algorithm. Then we describe the searching and pruning strategy used in our algorithm. And finally we provide the performance of our algorithm on benchmark databases together with the scalability test result on our algorithm. We also illustrate how the multivalued dependencies found in the XML data help refining the relational schema design.

We give a conclusion on our current work and discuss about our future work in Chapter 5.

Chapter 2

Related Work

2.1 Overview of XML

In this section, we give a brief overview of XML and DTD. (Note: only the concepts that are related to this paper will be introduced; for the formal specifications, see [15, 21])

2.1.1 Extensible Markup Language (XML)

Extensible Markup Language (XML) is simple, easily parsed and self-describing data format for representing and exchanging data on the web. At its most basic level, XML is a document markup language permitting tagged text (elements), element nesting, and element reference. Each tagged element has a sequence of zero or more attribute/value pairs, and a sequence of zero or more subelements.

Suppose there is an XML representation of catalog information for a book as shown in Figure 2.1.

Text delimited by angle brackets (<. . .>) is *markup*, while the rest is character data. Elements may contain a mix of character data and other elements; e.g.

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

Figure 2.1: An XML representation example

the `book` element contains the elements such as `title` and `price`. The element named `title` contains character data denoting the book title, and similarly, the element `price` contains character data denoting the book's price. This element also has an attribute named `currency` with the value `HKD`, represented using the syntax *attribute-name="attribute-value"* within the elements' start-tag.

In general, element names are unique; e.g., the `book` element in the example contains two `author` elements. However, attributes names are unique within an element; e.g., the `price` element cannot have another attribute named `currency`. The syntax also permits an empty element `<bestseller></bestseller>` to be represented more concisely as `<bestseller/>`. XML documents are called *well-formed* if they satisfy simple syntactic constrains, such as proper delimiting of elements names and attributes and proper nesting of start and end tags.

2.1.2 Data Type Definition (DTD)

XML provides a simple and general markup facility, which is useful for data interchange. The simple tag-delimited structure of well-formed XML makes parsing

extremely simple. However, applications that operate on XML data often need additional guarantees on the structure and content of such data. For example, a program that calculates the tax on the sale of a book may need to assume that each book element in its XML input includes a price subelement with a currency attribute and numeric content. Such constraints on document structure can be expressed using a `Document Type Definition (DTD)`. A DTD defines a class of XML documents using a language that is essentially a context-free grammar with several restrictions.

Using the book example in Figure 2.1, one may use the following DTD declaration in Figure 2.2 to constrain XML documents in our example.

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

Figure 2.2: An Document Type Definition(DTD) example

The first line of this declaration is an *element type declaration* that constrains the contents of the book element. Following common convention, the declaration syntax uses commas for sequencing, parentheses for grouping. Special operators like ? , * and + are used to denote different type of occurrences of the preceding construct as shown in Table 2.3

The second line of this DTD declares the type for the `title` element to be *parsed character data* (indicated by `#PCDATA` and implying an XML processor will parse the contents looking for markup). The declaration also indicates that the

<!ELEMENT...>	element type declaration
<!ATTLIST...>	attribute type declaration
<!ENTITY...>	entity type declaration
#PCDATA	parsed character data
CDATA	character data
?	zero or one
*	zero or more
+	one or more
|	or

Figure 2.3: Common declarations and operators used in an DTD

price element may have attributes currency, of type *character data* (indicated by CDATA) and default value USD; source, with one of the three values shown (an enumeration type) and default value list; and taxed, with the fixed (indicated by #FIXED) value yes. The fixed attribute type is a special case of the default attribute type; it mandates that the specified default value not be changed by and XML document conforming to the DTD. Our example DTD thus specifies that the book in our XML example in Figure 2.1 must be taxed. Note that the use of some element names without a corresponding declaration in the DTD is not an error. DTD is not a must for any XML document and such elements are simply not constrained by the DTD. However, it is expected that most of the practical XML documents on the web will conform to DTDs in order to make the XML data fully functional. For instance, any web application or a mobile agent encountering an XML file can interpret the file by consulting the DTDs to which the document conforms.

An XML document that satisfies the constraints of a DTD is said to be *valid* with respect to that DTD. The DTD associated with an XML document may be specified by the inclusion of *document type declaration*, e.g. <!DOCTYPE BOOKCATALOG SYSTEM "http://www.haha.com/ bookcatalog.dtd">, in a special section at the beginning of a document that called its *prolog*. The declara-

tion above indicates that the XML document claims validity with respect to the `BOOKCATALOG` DTD which may be found at the indicated location.

Apart from element type and attribute type declaration, there are indeed some other types of declaration. For instance, *entity type declarations* are used for declaring entities as an abbreviation: users can define an abbreviation with its corresponding full term, and then use this abbreviation in the XML document (or DTD). For more details about the DTD, please refer to [15, 21]. The data modelling provided by DTDs may not be sufficient for some applications and the XML Schema [2] proposal defines facilities that address the needs that cannot be provided by DTD. XML schema was accepted recently (2001-05-02) as a W3C [1] Recommendation. Still, DTD is more commonly used right now and the work on the XML Schema is still undergoing.

2.1.3 ID, IDREF and IDREFS

ID, IDREF and IDREFS are special attributes which are need for referencing element/elements from another element. The attribute ID can occur once for each element. ID uniquely identifies an element within a XML document and can be referenced through an IDREF field in another element. IDREFS is used when more than one IDREF field are referenced by the element. Consider the following example in Figure 2.4 with its DTD at Figure 2.5.

```
<Person Id='P1' Name='Ham' Friend='P2' />
<Person Id='P2' Name='Roy' Friend='P1' />
<Course Title='Introduction to Computing' Tutor='P1 P2' />
```

Figure 2.4: An example XML document fragment

From the DTD in Figure 2.5, it is clear that the attribute `Id` is of type ID,

```
<!ELEMENT Person EMPTY>
<!ATTLIST Person Id ID #REQUIRED Name CDATA #REQUIRED
          Friend IDREF #IMPLIED>
<!ELEMENT Course EMPTY>
<!ATTLIST Course Title CDATA #REQUIRED
          Tutor IDREFS #IMPLIED>
```

Figure 2.5: DTD for Figure 2.4

`Friend` is of type IDREF and `Tutor` is of type IDREFS. Thus attributes `Friend` and `Tutor` serve as references to `Person` elements.

2.2 Using Special-Purpose Database to Store XML Data

Most of the work on storing XML data uses semistructured database system. For such a special-purpose database system such like Lore [43, 27] or Strudel [24], it is particularly tailored to store and retrieve XML data, using specially designed structures and indices [45], query languages [5, 48, 34, 18] and particular query optimization techniques [44]. However, it is still unclear if the approach of using special-purpose system is going to find widespread acceptance. Despite that the special-purpose should work best, it is going to take a long time before such systems are mature and scale well for large amount of data. On the other hand, relational database systems are mature and scale very well, and they have the additional advantage that in a relational database XML data and traditional (structured) data can co-exist making it possible to build applications that involve both kinds of data with little extra effort. As a result our approach in this thesis is to explore the use of an RDBMS to manage the XML data.

2.3 Using Relational Databases to Store XML Data

In the approach of using an RDBMS to store and query XML data, XML data is mapped into relational tables and queried by SQL. As the requirements of processing XML data are very different from requirement to process traditional (structured) data, recent work has concentrated on models and algorithms to convert XML documents to relational tuples, and the main concern is how to produce the relational schemas from the XML data. We state some of the recent work in below.

2.3.1 Extracting Schemas with STORED

Deutisch et al. proposed STORED [22] approach which use a combination of semistructured and relational techniques. First all the XML data are mapped into semistructured model which is similar to the graph model used in special-purpose databases. Then [22] uses *frequently pattern discovery* to produce the relational schemas. The most frequently-appeared patterns found in the semistructured model of XML are used to produce relational schemas while the least frequently-appeared patterns are stored into overflow graphs.

can be stored into RDBMS entirely under STORED. [22] claims that under reasonable assumptions, the generated schemes can cover a large percentage of the XML data (at approximately 90 %) while the remaining data have to be managed separately by overflow graph, making the data hard to be managed and queried.

2.3.2 Using Simple Schemes Based on Labeled Graph

Florescu and Kossmann proposed simple ad-hoc schemes based on the widely accepted graph model for semistructured data [25, 26]. In the labeled graph model, each XML element is represented by a node in the graph; the node is labeled with the oid of the XML object. Element-subelement relationships are represented by edges in the graph and labeled by the name of the subelement. The order of subelements of an element is represented by ordering every outgoing edges of a node in the graph. Values of an XML document are represented as leaves in the graph. Various ways to store the edges of the graph, as well as ways to store the leaves of the graph, are proposed.

The approach used in [25, 26] focused much at preserving of the order and structure of the original XML data and has to create a lot of extra data . With only a small portion of the attributes in the table storing the actual XML data, the produced tables are apparently much larger in size than the original XML documents, making the approach least attractive. This approach is not suitable for direct queries on the data as well since too many attributes are unknown the the users.

2.3.3 Generating Schemas from DTDs

The most related work will be from Shanmugasundaram et al. who proposed creating relational schemas according to the DTDs the XML document conforming to [49]. XML data is not involved in the process at all. The produced table can then be used for semistructured-queries-like SQL queries. First, DTD graphs are created from the DTDs of the XML data. A DTD graph represents the structure of a DTD. Its nodes are elements, attributes and operators in the DTD. Each element appears exactly once in the graph, while attributes and operators appear as many times as they appear in the DTD. Relational Schema then can be gen-

erated from the DTD graph by inlining the elements and attributes following a set of rules. Several schema conversion techniques are proposed and discussed in [49]. Some of the recent research on XML and relational database [33, 31, 42, 52] also adopt the technique proposed in [49] for generating the relational schema.

Just like [25, 26], the resulting relation schemas are specifically designed that having many of it's attributes unrelated to the actual XML data but serving for the special purpose only, e.g. attributes are added for joining the tables only. In one of our proposed algorithm introduced in Chapter 3, we enhanced [49]'s approach based on relational database theory.

The three approaches presented above have one thing in common: they try to produce the relation schema solely based on the structure (or pattern) in the XML data without considering the characteristics of the data and the possible dependencies in the data. In our proposed work, we try to produce the relational schemas of XML based on both the structure of the XML data, and the characteristics of the XML data.

2.3.4 Commercial Approaches

Database companies are working to figure out how XML data can fit into their systems. For example, commercial product like Oracle 8i or 9i [51] or IBM DB2 XML Extender [30] provides a primitive solution which is to ask the user or a system administrator in order to decide how XML elements and attributes are stored in relational tables. It requires the user to have enough knowledge on the XML data and the user has to define the relational mappings of the XML data based on the special definition languages provided in the database system. Our proposed approaches are automatic.

2.4 Discovering Functional Dependencies

2.4.1 Functional Dependency

A *functional dependency* over a relation schema R is an expression $X \rightarrow Y$, where $X \subseteq R$ and $Y \in R$. The dependency *holds* or is *valid* in a given relation r over R if for all pairs of tuples $t, u \in r$ we have: if $t[A] = u[A]$ fro all $A \in X$, then $t[Y] = u[Y]$, i.e. t and u *agree* on X and Y. A functional dependency $X \rightarrow Y$ is minimal (in r) if A is not functionally dependent on any proper subet of X, i.e. if $Z \rightarrow Y$ does not hold in r for any $Z \subset X$. The dependency $X \rightarrow Y$ is trivial if $Y \in X$. Functional dependency is originally defined in [20]. The axioms for functional dependencies are introduced in [8]. The theory of functional dependencies is discussed in [36]. Functional dependency is one of the most important constraints in relational database design and analysis.

2.4.2 Finding Functional Dependencies

In our proposed algorithms, one of the important steps would be using the existing functional dependency inference technique to find out the functional dependencies inside the XML data. Much work has been done on discovering functional depedencies from relations in the past years [39, 32, 47, 35, 29]. It is called functional dependency inference problem: *Given a relation r, find a set of functional dependencies that is equivalent with the set of all functional dependencies holding in r.* As the problem can have a probabilistic nature, some of the recent works have been focused on approximate functional dependency inference [32, 47]. In order to improve the efficiency of the dependency inference, some of the recent works have been focused on using parallel approaches [47, 35]. Recently a new algorithm called *TANE* for discovering functional and approximate dependencies was proposed [29], which has improved the efficiency of dependency inference by

several orders of magnitude over the previous work. We find that it is also possible to apply this existing techniques in finding the functional dependencies for our relational schema prototypes for XML data. Moreover, a new algorithm for finding *multivalued dependencies* proposed by us is based on the idea of partition refinement used in *TANE*.

2.4.3 TANE and Partition Refinement

TANE finds functional dependencies based on the concept of *partition refinement*.

For a relation schema R, given a relation (or table) r, two rows (or tuples) t and u are **equivalent** with respect to a given set $X \subset R$ if attributes $t[A] = u[A]$ for all $A \in X$. Any attribute set X partitions the tuples of the relation into **equivalence classes**. We denote the equivalence class of a tuple $t \in r$ with respect to a given set $X \subseteq R$ by $[t]_X$, i.e. $[t]_X = \{u \in r \mid t[A] = u[A] \text{ for all } A \in X\}$. The set $\pi_X = \{[t]_X \mid t \in r\}$ of equivalence classes is a **partition** of r under X. That means π_X is a collection of disjoint sets (equivalence classes) of tuples, such that each set has a unique value for the attribute set X, and the union of the sets equals the relation r.

Tuple ID	A	B	C	D
1	1	1	2	3
2	1	2	1	4
3	1	2	2	2
4	1	1	1	4
5	1	1	2	2
6	1	2	2	3
7	2	3	1	3
8	2	3	2	4

Table 2.1: An example relation

For example, consider the relation in Table 2.1. Attribute A has value 1 for tuples 1 to tuples 6, so they form an equivalence class $\{1, 2, 3, 4, 5, 6\}$ (here we use tuple identifiers to denote tuples). Attribute A has value 2 in tuple 7 and tuple 8, so they form another equivalence class $\{7, 8\}$. The whole partition with respect to A is $\pi_{\{A\}} = \{\{1, 2, 3, 4, 5, 6\}, \{7, 8\}\}$. The partitions for other attributes are $\pi_{\{B\}} = \{\{1, 4, 5\}, \{2, 3, 6\}, \{7, 8\}\}$, and the partition with respect to $\{CD\}$ is $\pi_{\{CD\}} = \{\{1, 6\}, \{2, 4\}, \{3, 5\}, \{7\}, \{8\}\}$.

A partition π is a **refinement** of another partition π' if every equivalence class in π is a subset of some equivalence class of π'.

Let t_i be the tuple with Tuple ID $= i$. $\pi_{\{CD\}}$ refines $\pi_{\{A\}}$ since each equivalence class in $\pi_{\{CD\}}$ is totally contained by some equivalence class in $\pi_{\{A\}}$. On the other hand, $\pi_{\{CD\}}$ dose not refine $\pi_{\{B\}}$ since some equivalence classes in $\pi_{\{CD\}}$ are not contained in any equivalence class in $\pi_{\{B\}}$. For instance, $[t_1]_{\{CD\}} = \{1, 6\}$ in $\pi_{\{CD\}}$ is not contained in any equivalence class in $\pi_{\{B\}}$.

It is easy to see that the partitions can be computed as a product of two previously computed partitions. As shown in *TANE*, the **product** of two previously computed partitions π' and π'', denoted by $\pi' \cdot \pi''$, is the least refined partition π that refines both π' and π'': For all $X, Y \subseteq R, \pi_X \cdot \pi_Y = \pi_{\{X \cup Y\}}$.

According to TANE, a functional dependency $X \rightarrow Y$ holds if and only if π_X refines π_Y. Thus the concept of partition refinement gives almost direct functional dependencies, i.e. we can determine if a functional dependency $X \rightarrow A$ holds by simply checking if $\mid \pi_X \mid = \mid \pi_{X \cup \{A\}} \mid$.

To find all minimal non-trivial functional dependencies. *TANE* starts the search from singleton sets of attributes and works its way to larger attribute sets through the set containment lattice level by level. When the algorithm is processing a set X, it tests dependencies of the form $X/\{A\} \rightarrow A$, where $A \in X$. This guarantees that only non-trivial dependencies are considered while pruning

the search space effectively, as shown in Figure 2.6.

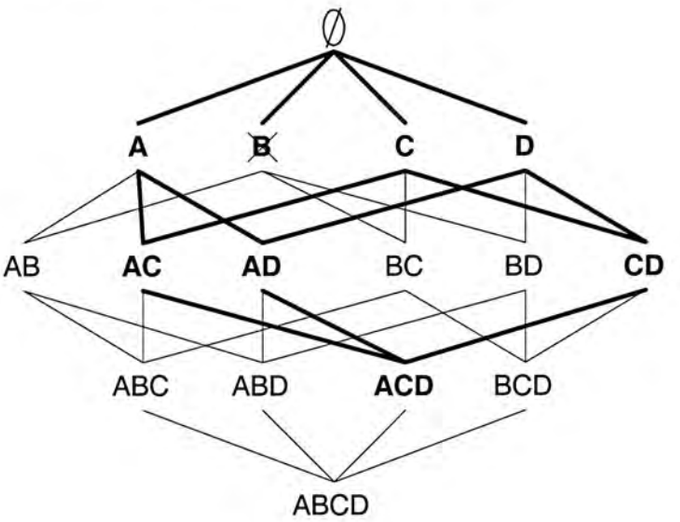

Figure 2.6: A pruned set containment lattice for $\{A, B, C, D\}$. Due to the deletion of B, only the bold parts are accessed by the levelwise algorithm

TANE also adopted levelwise strategy [40]to discover the functional dependencies level by level while pruning much of the search space. As a result, the algorithm can outperform the previous algorithms by several orders of magnitude. For more details please refer to [29].

2.5 Multivalued Dependencies

Multivalued dependency was first discussed in [53]. A set of axioms are given in [9] for multivalued dependency where the axioms are proved to be sound and complete. The notion of fourth normal form (4NF), which is based on multivalued dependency, was proposed in [23].

We assume the usual interpretation of a relation (or table) in the relational database model where no duplicate tuples are allowed. The definition of multivalued dependency is given below:

Let R be a relation schema and let $X = X_1, X_2, ..., X_n$ be a subset of R , let

$Y = Y_1, Y_2, ..., Y_n$ be a subset of R and let $Z = R - Y - X$. The **multivalued dependency** $X \rightarrow\rightarrow Y$ holds in R if, in any legal relation $r(R)$, for all pairs of tuples t_1 and t_2 in r such that $t_1[X] = t_2[X]$, there exists tuples t_3 and t_4 in r such that:

$$t_1[X] = t_2[X] = t_3[X] = t_4[X]$$

$$t_3[Y] = t_1[Y]$$

$$t_3[Z] = t_2[Z]$$

$$t_4[Y] = t_2[Y]$$

$$t_4[Z] = t_1[Z]$$

Given a relation schema R, a multivalued dependency $X \rightarrow\rightarrow Y$ is said to be ***non-minimal*** if there exists a multivalued dependency $X' \rightarrow\rightarrow Y$ where X' is a proper subset of X; $X \rightarrow\rightarrow Y$ is said to be ***trivial*** if $Y \subset X$ or $X \cup Y = R$. It is obvious that in order to have a *non-trivial* multivalued dependency $X \rightarrow\rightarrow Y$, all X, Y and $Z = R - X - Y$ cannot be $\emptyset$.

2.5.1 Example of Multivalued Dependency

Table 2.2 shows a simple example to demonstrate the occurrence of multivalued dependency. Consider a relation schema with three attributes namely `Course`, `Teacher`, and `Text Book`. Suppose that for a course MVD1110 taught in a certain semester, there are two teachers (A and B) sharing the teaching and the course requires three text books (Book1, Book2 and Book3). There is no reason to associate a `Teacher` with one `Text Book` but not the others. As a result, the only way to express the fact that `Teachers` and `Text Books` of a `Course` are independent of each other is to have each `Teacher` associate with each `Text`

`Book`, and the tuples for the course MVD1110 are shown in Table 2.2.

`Course`	`Teacher`	`Text Book`
MVD1110	A	Book1
MVD1110	A	Book2
MVD1110	A	Book3
MVD1110	B	Book1
MVD1110	B	Book2
MVD1110	B	Book3

Table 2.2: An multivalued dependency example

It is obvious that redundancy exists in the table. However, there is no functional dependency. The way to remove the redundancy is to consider multivalued dependency. With the definition of multivalued dependency we can see that `Course` $\rightarrow\!\!\rightarrow$ `Teacher` and `Course` $\rightarrow\!\!\rightarrow$ `Text Book` hold in the example. For example, taking the *first* and the *last* tuple in our example table as t_1 and t_2 respectively, the corresponding t_3 and t_4 in the table should be the *third* and the *fourth* tuple respectively such that:

$$t_1[\texttt{ Course }] = t_2[\texttt{ Course }] = t_3[\texttt{ Course }] = t_4[\texttt{ Course }] = \text{MVD1110}$$

$$t_3[\texttt{ Teacher }] = t_1[\texttt{ Teacher }] = \text{A}$$

$$t_3[\texttt{ Text Book }] = t_2[\texttt{ Text Book }] = \text{Book3}$$

$$t_4[\texttt{ Teacher }] = t_2[\texttt{ Teacher }] = \text{B}$$

$$t_4[\texttt{ Text Book }] = t_1[\texttt{ Text Book }] = \text{Book1}$$

Chapter 3

Using RDBMS to Store XML Data

As mentioned before in section 1, XML data and relational data are vastly different in nature (semistructured vs. structured) thus we directly store XML data into RDBMS. We have to come up with the suitable relation schemas and use them for mapping the data in the XML documents into the RDBMS accordingly. The general flow of generating the suitable relational schemas is shown in Figure 3.1.

After suitable DTD simplification, prototype schemas are extracted from the simplified DTD. The relational schemas are then further decomposed from the prototype schemas according to the functional dependencies discovered in the XML data.

Based on this general flow, we propose several algorithms to create relational schemas from the XML data and the DTD those XML data conforming to. Although the algorithms we propose have different details, the global scheme is the same, as shown in Figure 3.2.

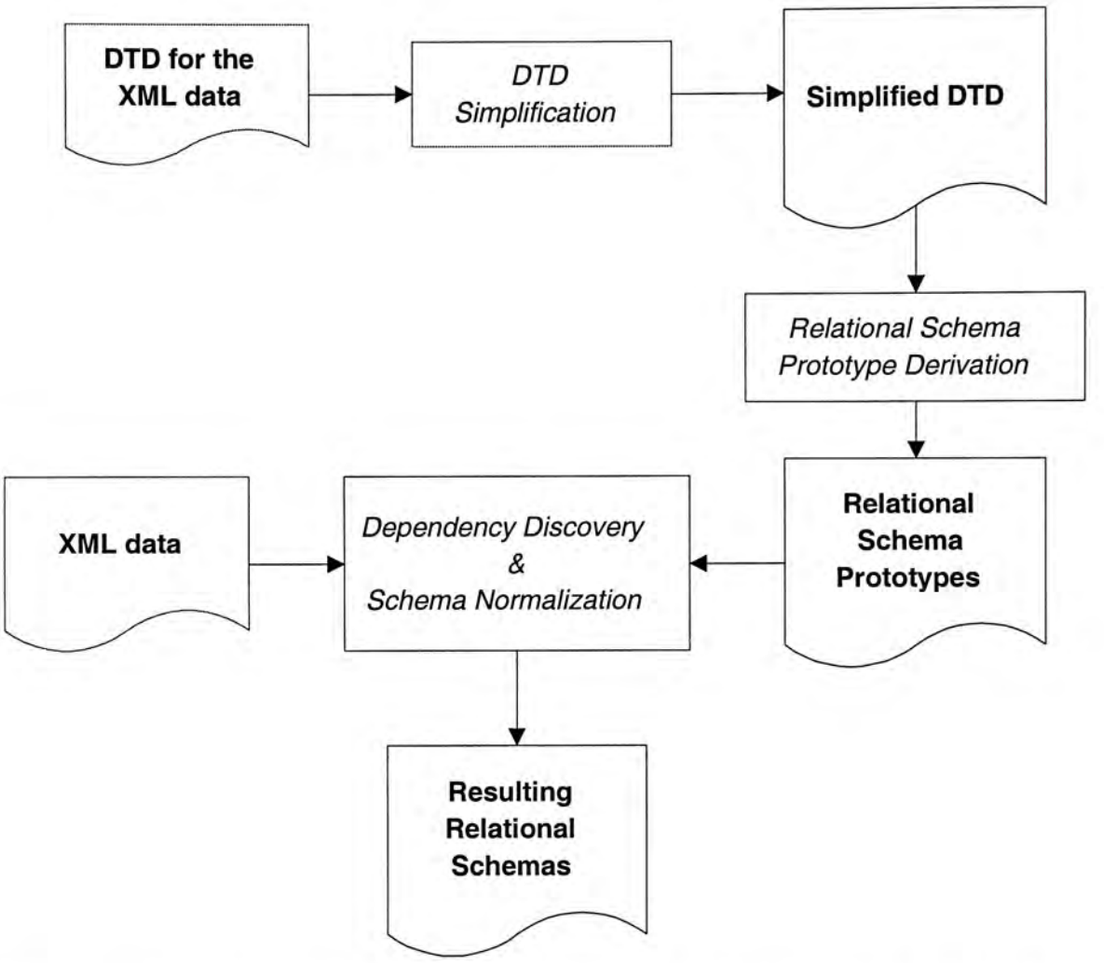

Figure 3.1: General flow of generating relational schemas from XML

Algorithm GENERAL_SCHEMA_EXTRACTION_ALGORITHM

1 **INPUT** :
2 ▷ Set of XML documents conforming to the same DTD
3 ▷ DTD used by the set of XML documents
4 **OUTPUT**:
5 ▷ Set of relational schemas for the set of XML documents
6 **METHOD**:
7 Simplify DTD
8 Construct schema prototype trees
9 Generate relational schema prototypes
10 Detect possible functional dependencies and candidate keys
11 Normalize the relational schema prototypes

Figure 3.2: Algorithm for extracting relational schemas from XML

3.1 Global Schema Extraction Algorithm

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

3.1.1 Step 1: Simplify DTD

First, we need to simplify the DTD for the set of XML documents. Being the schema of XML, DTD can be very high in complexity just like their counterpart for semistructured data [10]. Even we expect that DTDs designed for real-life applications would not have extremely complicated structures, an ordinary nested DTD consist of entity type declarations still possesses high complexity. Attempts for constructing schema prototype trees (which will be described in Step 2) of a DTD would likely be a hard job. However, it is possible to simplify the DTD and without affecting the way we extract the relational schemas. After all, we just want to take the DTD as a reference for generating required relational schemas that can be used for storing the data in the XML documents into an RDBMS.

To simplify the DTDs, we need to get rid of entity declarations first. They do not affect on the structure of the DTD. Rather, they are practical features for abbreviating frequently appeared DTD components, defining or referring to external or non-XML data...etc. For entity type declarations which are used to abbreviate DTD components, they are removed and all the declarations referring to them are replaced with the DTD components they are representing to. An example has been shown in Figure 3.3.

Besides removing entity type declarations, we need to deal with the possible complex element type declarations. In fact, we expect most of the complexity of DTDs should come from the complex structure of the element type declarations. For example, we could have an element p as `<!ELEMENT p (#PCDATA |(a+,(b*`

Original XML segment:

```
<!ENTITY %text "#PCDATA" >
<!ENTITY %text.includes "a | em" >
<!ELEMENT p (%text; | %text.includes;)* >
```

XML segment after removing entities and reference:

```
<!ELEMENT p (#PCDATA | a | em )* >
```

Figure 3.3: An example of removing entities declarations and references

| (c, (b, d?)*))*)> where a, b, c and d are p's subelements. The parenthesis indicate that element p would be highly nested. The binary operators "+", "*", "|" and "?" on any subelements increase the uncertainty on the occurrence of each subelements. However, what we concern in this global schema extraction algorithm about DTDs would be the presence of possible kind of subelements within the element only.

As a result, we propose a set of transformations which can convert the element type declarations into the required simplified forms. Part of our proposed transformations is similar to those presented in [49] and [22]. However, other than flattening the nested representation of DTDs as proposed by [49] and [22], our transformations also eliminate the binary operators in DTDs. Every element type declarations can be converted to the required form by performing the following transformations shown in Figure 3.4 repeatedly (here p, p', ...denote subelements within a given element type declaration).

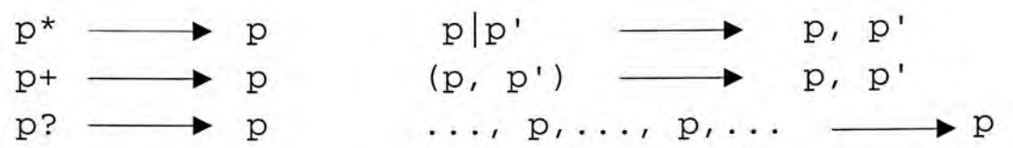

Figure 3.4: DTD transformations

After the transformation our example would now be: <!ELEMENT p (#PCDATA | a,b,c,d)>. The transformation would only preserve the *element-subelement(s)* relation in the element type declaration.

Moreover, anything in the DTD that is not related to the structure of the DTD is removed. We only preserve the information that is useful in constructing schema prototype trees later. For instance, inside any *attribute type declaration*, the value types (e.g. #IMPLIED, #FIXED...etc) for the *character data (CDATA)* are removed from the DTD. Also. Special attribute like ID or IDREF is regarded as normal character data as well since their possible characteristics (e.g. ID type data can be a key in the relational table), if there are any, can be discovered in the later step of finding functional dependency anyway.

Figure 3.5 and Figure 3.6 show the example of converting a DTD into a simplified DTD.

Figure 3.8 shows the example of converting a DTD in Figure 3.7, which is a modification of [46], into a simplified DTD.

3.1.2 Step 2: Construct Schema Prototype Trees

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

```
<!ENTITY %txt "#PCDATA" >
<!ENTITY %page "initPage?, endPage?" >

<!ELEMENT SigmodRecord (issue)* >
<!ELEMENT issue (volume,number,articles) >
<!ELEMENT volume (%txt;)>
<!ELEMENT number (%txt;)>
<!ELEMENT articles (article)+ >
<!ELEMENT article (title,%page;,authors) >
<!ELEMENT title (%txt;)>
<!ELEMENT initPage (%txt;)>
<!ELEMENT endPage (%txt;)>
<!ELEMENT authors (author)+ >
<!ELEMENT author (%txt;)>
<!ATTLIST author position CDATA #IMPLIED>
```

Figure 3.5: An example DTD before simplification

```
<!ELEMENT SigmodRecord (issue) >
<!ELEMENT issue (volume,number,articles) >
<!ELEMENT volume (#PCDATA)>
<!ELEMENT number (#PCDATA)>
<!ELEMENT articles (article) >
<!ELEMENT article (title, initPage, endPage, authors) >
<!ELEMENT title (#PCDATA)>
<!ELEMENT initPage (#PCDATA)>
<!ELEMENT endPage (#PCDATA)>
<!ELEMENT authors (author) >
<!ELEMENT author (#PCDATA)>
<!ATTLIST author position CDATA >
```

Figure 3.6: The simplified DTD converted from the DTD in Figure 3.5

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

Figure 3.7: An DTD before simplification

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

Figure 3.8: The simplified DTD converted from the DTD in Figure 3.7

constructed as follows.

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

Rule 1 *Only element can become a root*

In XML, attributes cannot exist without following it's corresponding element. We thus can regard *element-attribute(s)* relations as the same as *element-subelement(s)*. As a result, all attributes declared in the DTD can only be leaf nodes in the schema prototype trees. We can thus consider only elements but not attributes when deciding the roots.

Rule 2 *For an element which do not appear in any other element declaration in the DTD, it becomes the root for a schema prototype tree*

This rule is quite straightforward. An element would not appear in any other element declaration *if and only if* it is not a subelement of any other element, and it is the actual meaning of "root".

Rule 3 *If there is no element in the DTD satisfying* ***rule 2****, one of the element is selected as the root*

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

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

Starting from the subelement(s) of the root, we try to scan the DTD in a depth-first style. For a first-time visited subelement which do not appear in the schema prototype tree, we create a new node bearing the same name in the

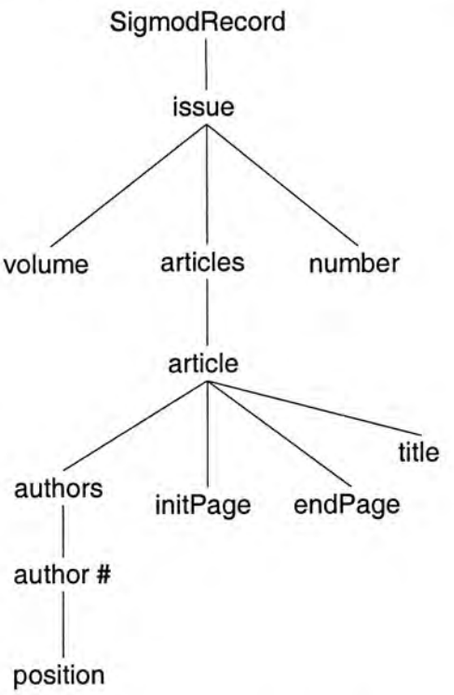

Figure 3.9: The schema prototype tree of the simplified DTD in Figure 3.6

schema prototype tree. Moreover, an edge is created from the parent node of the newly created node to the newly created node. Apart from subelements, we need to take care of possible parsed character data (#PCDATA) and the attribute declarations for an element we are visiting. Any attributes declared for an element in the DTD is treated the same way as a subelement of the element. It is easy to see that the leaf nodes of the schema prototype tree are either element declared as containing #PCDATA only, or attributes for their parent elements. If an element has declared as containing #PCDATA together with other subelement, we would mark the corresponding node with a "#" in the schema prototype tree. The marking would be useful in the following step.

The schema prototype tree corresponding to the above example is shown in Figure 3.9. Note that as `SigmodRecord` is the only element that is not referred by any other element, the schema prototype tree for `SigmodRecord` is the only tree that is constructed from our example DTD.

We also need to handle the possible situation where recursion occurs while

constructing the schema prototype tree. Consider a case when we visit an element which has already had a corresponding node X created in the schema prototype tree, we would create a leaf node with label $X.A$ which indicates a foreign key to its ancestor. The key can be discovered or arbitrarily assigned in Step 4 later. Then we would stop traveling down the subelement of that element to prevent an infinite recursion. Using Figure 3.8 as an example, consider when the tree construction has come to the element declaration <`!ELEMENT editor (monograph)`> where element `monograph` has already appeared in the tree. We would create a new node `monograph.A`. An edge pointing from `editor` to it is created as well.

The example schema prototype tree for the modified DTD would look like the one shown in Figure 3.10.

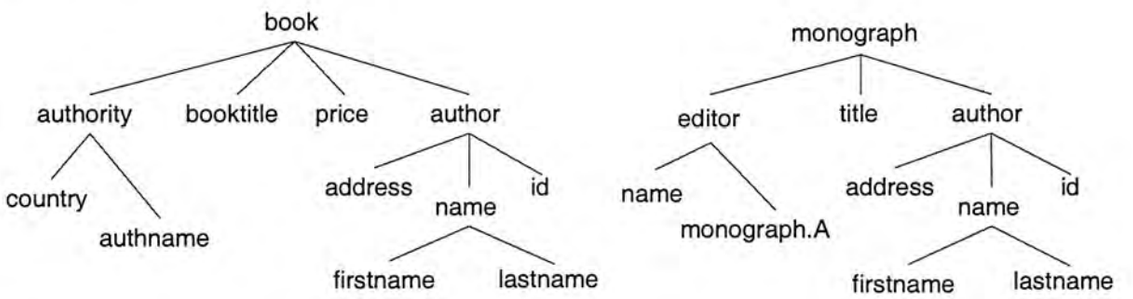

Figure 3.10: The schema prototype tree constructed from the example DTD in Figure 3.8

3.1.3 Step 3: Generate Relational Schema Prototype

Given a schema prototype tree, the corresponding relational schema prototype is generated as follows. The basic idea is to regard all the necessary attributes and elements in the simplified DTD as the "attributes" in an ER-Model. The schema prototype is thus generated by inlining all the necessary descendants of the schema prototype tree starting from the root. The necessary descendants refer to all the leaf nodes in the schema prototype tree, and the nodes marked

with a "#". The reason for doing this is because not all the elements in an XML document contain real data. We want to prevent creating unused fields for these elements in the relational schema prototype. Using Figure 3.9 as an example, the element `issue` is not declared to contain any #PCDATA in DTD. Thus we can be sure that for any XML document conforming to that DTD, there is no parsed character data exists between any pairs of <issue> and </issue> tag (which are used to represent element `issue` in XML). As a result, we do not have to provide a filed for the element `issue` in the relational schema prototype.

The relational schema prototype generated from the schema prototype tree presented in Figure 3.10 is shown in Figure 3.11. In order to uniquely specify the name for each attribute in the relational prototype schema, all attributes fields are named by the path from the root node of the tree.

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

table:monograph (
monograph.title, (A)
monograph.author.id, (B)
monograph.author.name.firstname, (C)
monograph.author.name.lastname, (D)
monograph.author.address, (E)
monograph.editor.name, (F)
monograph.editor.monograph.A (G)
)

Figure 3.11: The relational schema prototypes generated from the tree in Figure 3.10

3.1.4 Step 4: Discover Functional Dependencies and Candidate Keys

With the generated schema prototypes, we can now apply traditional techniques of relational database to produce the suitable relational schemas for the XML data.

In order to reduce the data redundancy and inconsistency in the set of relational schemas for the XML data, we have to discover a set of functional dependencies and the candidate keys by analyzing the XML data. Those constraints discovered from the XML data would be vital for us to normalize the relational schema prototype in an appropriate normal form.

We adopted a recently proposed technique for discovering functional dependencies, which is called *TANE* [29], in our algorithm. Before *TANE*, previous algorithms have invariably based on either repeatedly sorting the tuples of the relation or comparing every tuple to all other tuples which makes them inefficient for large relations. However, with respect to number of tuples, *TANE*'s' complexity is claimed to be linear. it formulated the dependency discovery task in terms of equivalence classes and partitions, together with efficient search space pruning techniques. We found that *TANE* is very suitable for the functional dependency discovering step in our algorithms.

Let's assume we have found the minimal set of functional dependencies of Figure 3.11 using *TANE*:

`table:book`

FD(s): $A \rightarrow BC$, $DEF \rightarrow C$ and $C \rightarrow DEF$

`table:monograph`

FD(s): $A \rightarrow BF$, $B \rightarrow CDE$, and $CDE \rightarrow B$

We can then easily obtain the candidate keys from the minimal set of func-

tional dependencies. A set of attribute $\{A_1, A_2...A_n\}$ in a relation r is a candidate key for the relation r iff the closure for that set of attributes, $\{A_1, A_2...A_n\}^+$, contains all the attributes in r. As a result, we can find $\{AGH\}$ being the possible candidate key for the relational schema prototype talbe `table:book`, and $\{A\}$ being the possible candidate key for `table:monograph` presented in Figure 3.11.

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

3.1.5 Step 5: Normalize the Relational Schema Prototypes

With the functional dependencies and candidate keys, we can simply normalize the relational schema prototype to a set of new relations. We use 3NF decomposition [14] as an example. 3NF decomposition algorithm is presented in appendix for readers' reference. The data in the XML document can then be stored to the RDBMS according to the schema shown in Figure 3.12. Note that since `table:book-3` and `table:monograph-3` are the same after comparing the attributes in them, they can be merged as one.

3.1.6 Discussion

We have proposed the global schema extraction algorithm in the above. We call it the *global* schema extraction algorithm because in the algorithm we try to form relational schema prototypes which include as much elements in the DTD as pos-

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

table:monograph-1 (
monograph.title, (A)
monograph.author.id, (B)
monograph.editor.name (F)
)
table:monograph- 2(
monograph.editor.name, (F)
monograph.editor.monograph.title (G)
)
table:monograph-3 (
monograph.author.id, (B)
monograph.author.name.firstname, (C)
monograph.author.name.lastname, (D)
monograph.author.address (E)
)

Figure 3.12: Relations decomposed from schema prototype for the XML data

sible, then we extract all the necessary information from the raw data in order to decompose the schema prototypes into the suitable relational schemas. As a result, the step of functional dependency inference together with the characteristic of the actual XML data play a heavy role in this algorithm. Since the relational schemas are created by using many traditional relational database methods in this algorithm, we can be sure that the schemas can make the XML data suit well into the relational database. Moreover, unlike the proposed schemas extraction algorithm by [25, 26], we do not have to introduce any extra data fields at all.

However, one of the potential problem in the above proposed algorithm is that the cost of discovering functional dependencies can be high since *the number of minimal dependencies must be exponential in the number of attributes* [37, 38] while a schema prototype could includes as many attributes as the total number of leaf nodes in the schema prototype trees created from the DTD (Consider the case when only one schema prototype tree is constructed from the DTD). As a result, when the structure of the XML is relatively large (having a large number

of different element and attribute types), it might be better if we can reduce the size, i.e. the number of attributes, of the schema prototypes before the step of finding functional dependencies. Another consideration for the *Global* algorithm is that when the characteristics of the XML data changed vastly, e.g. having a large scale update, the change might affect the resulting schema produced. If such kind of large scale change is predicted, it might be better to analyse more on the declared structure, i.e. the DTD, of the XML and produce relational schema that is more flexible to changes within the constraint of the DTD.

3.2 DTD-splitting Schema Extraction Algorithm

In section 3.1, the proposed algorithm emphasizes more on FDs and keys discovery from prototype schemas. In this section, we propose another schema algorithm - DTD-splitting Schema Extraction Algorithm. This algorithm also follows the steps in Figure 3.2. However, unlike the previous algorithm, this DTD-splitting algorithm relies more on the first 3 steps. In other words, instead of finding out all characteristics on the actual XML data, we have to determine some of them without referring to the XML data. In our second algorithm, we try to predict some characteristics of the XML data from the DTD, hence perform a certain level of schema decomposition (DTD split) before the step for finding functional dependencies and keys. In this case, the size of schema prototypes should be smaller than those in section 3.1, thus alleviating the possible cost in the functional dependency discovery steps.

Just like us, [33] shows interests in predicting some characteristics from the DTD. [33] tries to find out semantic constraints in DTD but those constraints are not for generating relational schemas - [33] just adopts the schema generating algorithm from [49]. Rather, they are just used to ensure the semantics for the relational schema generated from [49]. Also, in [33] actual XML data for the

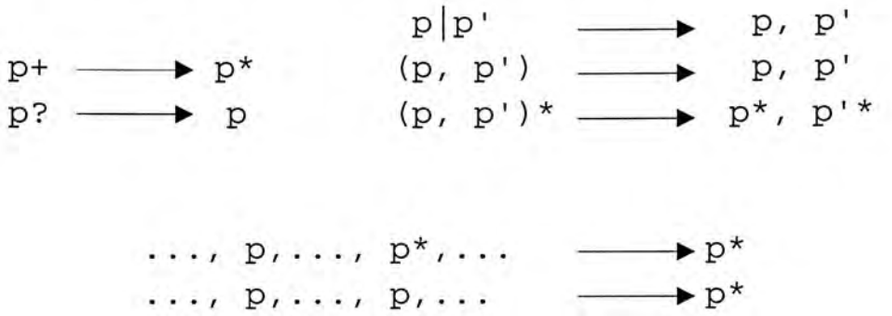

Figure 3.13: DTD transformations

DTD is not taking into consideration. Moreover, most of those constraints are based on the behaviour of attribute declaration only. When there are no rich attribute declarations in the DTD, those constraints cannot be determined by [33] at all.

3.2.1 Step 1: Simplify DTD

Just like the previous algorithm, we need to simplify the DTD for the set of XML data. We have to first remove all the entity declarations so as to review the actual structure of the DTD, as shown in the Figure 3.3. Then, we have to reduce the possible highly complicated structures of the DTD. In the previous algorithm, we consider only the possible kinds of element-subelement relations but not the binary operators "+", "*"," |" and "?" on the subelements. However, in this algorithm we preserve some of the binary operators so as to preserve some subelement occurrence information. Every element type declarations can be converted to the required form by performing the following transformations shown in Figure 3.13.

In the transformations, we simplify the occurrences of each subelement to either *one* or *more than one*. It is important to note that: the original meaning for "*" is *zero or more than one* but we convert it to *more than one* in our

```
<!ELEMENT SigmodRecord (issue*) >
<!ELEMENT issue (volume,number,articles) >
<!ELEMENT volume (#PCDATA)>
<!ELEMENT number (#PCDATA)>
<!ELEMENT articles (article*) >
<!ELEMENT article (title, initPage, endPage, authors) >
<!ELEMENT title (#PCDATA)>
<!ELEMENT initPage (#PCDATA)>
<!ELEMENT endPage (#PCDATA)>
<!ELEMENT authors (author*) >
<!ELEMENT author (#PCDATA)>
<!ATTLIST author position CDATA >
```

Figure 3.14: The simplified DTD converted from the DTD in Figure 3.5

transformation. For the case of "+" (*one or more than one*), we regard it as "*" (*more than one*) since there is a chance for the subelement to occur *more than one* times. For the case of "?" (*zero or more than one*), we simply remove the "?" since there is a chance for the subelement to occur once. Just like the corresponding step in the previous algorithm, special attribute type like ID or IDREF is treated as normal character data. However, since we have preserve the occurrence information this time, IDREFS type attribute is converted to normal character data with a * since IDREFS represents more than one IDREFs.

Figure 3.14 shows the example of converting a DTD in Figure 3.5 into a simplified DTD.

3.2.2 Step 2: Construct Schema Prototype Trees

With the simplified DTD, we then construct the *schema prototype trees* which represents the structure of the simplified DTD. As mentioned before, the nodes can be elements or attributes specified in the DTD. Schema prototype trees will

be used for generating schema prototypes in the next step (Step 3). However, the rules for determine the roots and the tree construction sequence is not the same as the one mentioned in section 3.1. Schema prototype trees are constructed as follows.

Root Determination

Again, we have to determine the roots of the trees from the DTD. There are several rules we have to follow when deciding the root:

Rule 1 and **rule 2** are similar to those stated in Section 3.1.

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

The rule is only applicable to *non-#PCDATA* elements because a *#PCDATA* element is definitely a leaf node in the schema prototype tree, as we saw in the previous algorithm. If we let a *#PCDATA* element become a root, the only element that is contained in its schema prototype tree would be the root itself. We do not want to create such kind of unnecessary schema prototype tree.

Let us first assume an element `C` being the subelement of both element `A` and `B` in the DTD. We would make element `C` a root for a schema prototype tree and the schema tree constructed from it would become a separate schema later in the following step. We can use traditional relational database theory to explain why we separate `C`.

There are four kinds of mapping cardinalities [50]: *one-to-one (1:1)*, *one-to-many (1:M)*, *many-to-one (M:1)* and *many-to-many (N:M)*. We can view element `A` and all its possible ancestor(s), form a relation **A**, while `B` and all its ancestor(s) form another relation **B**. For element `C` and all its possible subelement(s), we regard them as another relation **C**. We let the key of relation **A** be $K_{\mathtt{A}}$, the key

of **B** be $K_{\mathtt{B}}$ and the key of **C** be K_C. As element C is referred by both element A and B in the simplified DTD, we can eliminate the possible chances for *1:1* and *1:M* because each element A and B can only have one subelement C. So the possible mapping cardinality we have to consider for the relation A$-*-$B would be *M:1* and *M:N*.

The relationship among A, B, C indicates some tendency for multiple elements of A and B to map to element of C. For example, both a_1 in A and b_1 in B are mapped to c_1 in C. Taking C as a root can reduce the redundancy of repeating the attributes of c_1 with both a_1 and b_1. If at most one element of C can be mapped to an element of either A or B, we have many-to-one (*M:1*) mapping from **A** or **B** to **C**.

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

Thus we would make element C a root for a schema prototype tree and the schema tree constructed from it would become a separate schema later in the following step.

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

The rule is only applicable to *non-#PCDATA* elements and we the reason is the same as the one for **rule 3**. If the element B appears in more than one element declarations, it would fulfill **rule 3** and must be separated as a root. Thus we do not have to consider the nature of its ancestor - element A. Otherwise, when

B only appears in declaration of element A, we have to make sure B is not the ONLY subelement A before separate B as a root. If not, the separation would make the schema prototype tree of A contains A itself. We do now want to create such kind of unnecessary schema prototype tree.

We again use traditional relational database theory to explain why we separate elements with a "$*$" into another schema prototype tree:

For an element declaration <!ELEMENT A (B*) > inside the DTD, where the relation between A and B is A$-*-$B, we can view element A and all its possible ancestor(s), form a relation **A**. For element B and all its possible subelement(s), we regard them as another relation **B**. Again, we let the key of relation **A** be $K_{\mathtt{A}}$ and the key of **B** be $K_{\mathtt{B}}$. When we have a relation A$-*-$B in the DTD, as A is set to have more than one B subelements in the simplified DTD, we can eliminate the possible chances for *1:1* and *M:1* relationship from relation **A** to **B** . So the possible mapping cardinality we have to consider for the relation A$-*-$B would be *1:M* and *N:M*.

Here the "*" has some indication of the tendency of a *1:M* relationship from A to B. For this *1:M* case, each value of $K_{\mathtt{B}}$ is associated with at most one value of $K_{\mathtt{A}}$. It nearly directly come to the idea that $K_{\mathtt{B}}$ should functionally determine $K_{\mathtt{A}}$. Since the FD $K_{\mathtt{B}} \rightarrow K_{\mathtt{A}}$ holds, B can be separated from A. In terms of relational database theory, we can decompose them into two relations: A and B$\cup K_{\mathtt{A}}$.

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

Thus, we would make element B a root for a schema prototype tree. The schema tree constructed from it would become a separate schema later in the

following step.

Both **rule 3** and **rule 4** are similar to what suggested in [49]. In [49], if there is a relation A− ∗ −B where B is A's subelement, they create a new relation for B as they think B might correspond to the set-valued child of A; if a element B is more than one element's subelement, they create a new relation for b as they think B can be shared by relations. While [49] just simply make them as heuristics, we explain the reasons to set such rules in our algorithm based on relational database theory. Moreover, our rules can prevent creating unnecessary schema prototypes in the following step. The desirable schemas depending on the *1:M, M:1, M:N* relationships will be discovered in the later steps.

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

Just like the previous algorithm, we have to arbitrarily break the loop in order to construct the schema prototype tree when recursion occurs.

Tree Construction

For all selected roots in the DTD, we propose three different methods to construct the trees. The different methods might lead to slightly different resulting schemas. At later stage, when the different relational schemas are used to store the XML data, they might give different effects on join operation in actual data queries.

Generally, the tree construction method is more or less the same as the one in section 3.1. Starting from the subelement(s) of each root, we scan the DTD in a depth-first style and add all first-time visited subelement as a node into the tree. We mark all non-leaf node which has #PCDATA with "#" and handle the recursion the same way as in section 3.1. However, during the scan, we won't

travel down to any element which is determined as a root. For different kinds of roots which are determined by different rules above, their tree construction processes might not be the same, and there are some variations in their construction processes in each methods below:

Top-down Construction Method

In the top-down approach, for all kinds of roots (either determined by **rule 2, 3, 4** or **5**), their tree construction processes are the same as the one in section 3.1. However, during the tree construction if we visit an element declaration of a *root* element, we would create a new node for the newly visited root element. For the case of recursion, if we visit an element declaration which has been visited before, we would create a new node correspond to the visited element and stop traverse down to prevent infinite looping. To illustrate the idea, we use an example shown in Figure 3.15, which is simplified from Figure 3.7, to constructed the trees. The trees constructed by this method is shown in Figure 3.16.

Note that leaf nodes with bold names are the roots to other trees. Trees will form relations and the keys of relations (trees) can be discovered or arbitrarily assigned in Step 4 later. By joining the schema prototype trees through those keys in a top-down fashion, we could actually reconstruct larger schema trees, which are similar to those created using the algorithm stated in Section 3.1.

Bottom-up Construction Method

In the bottom-up approach, for roots determined by **rule 2**, their tree construction processes are the same as the one in section 3.1. For roots determined by **rule 3** or **4**, we have to find out all of their ancestors in the DTD, and add corresponding nodes as the leaf nodes of the roots in the schema prototype trees. For the case of recursion, if we revisit the element declaration of the root, we will find out the direct ancestor of the root inside the looping, and add the

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

Figure 3.15: Another simplified DTD example

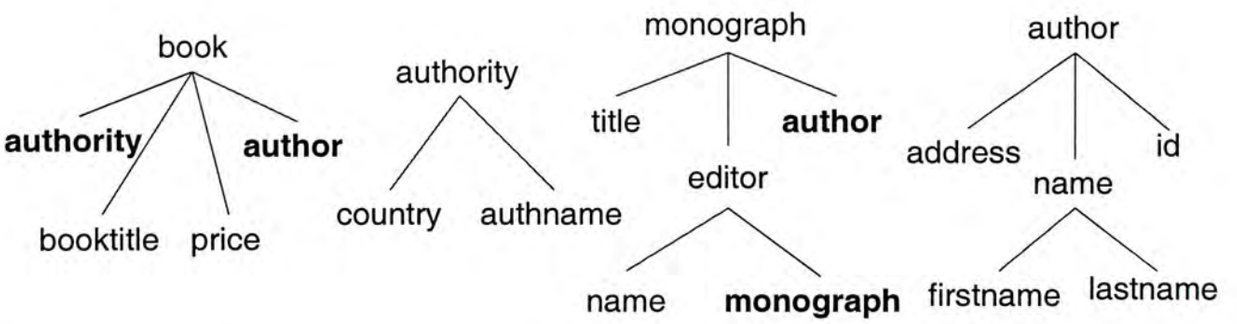

Figure 3.16: Schema prototype trees construction from Figure 3.15 using top-down construction method

corresponding node as a leaf node of the root. We then stop traverse down to prevent infinite looping. The schema prototype trees constructed by this method is shown in Figure 3.17.

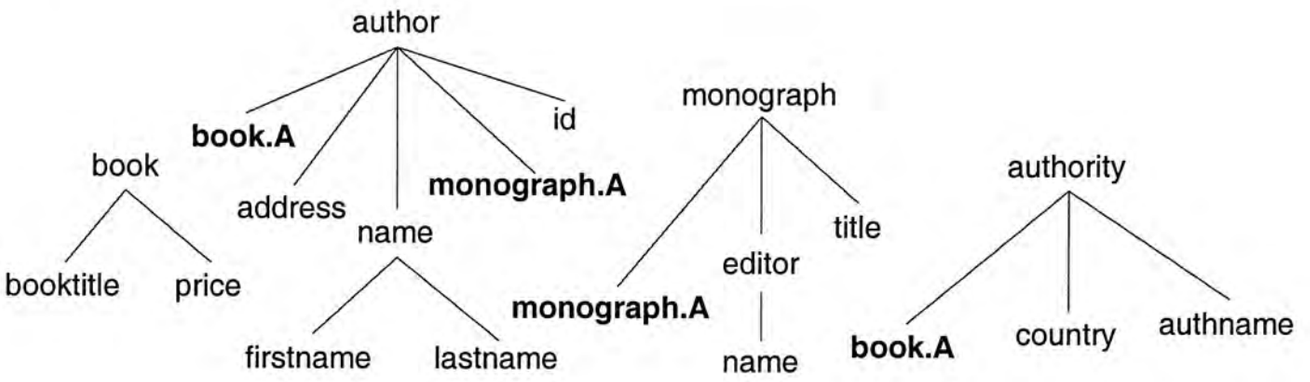

Figure 3.17: Schema prototype trees construction from Figure 3.15 using bottom-up construction method

Note that bold leaf nodes with a ".A" after each of their names is used to indicate the foreign keys to other relations. The keys can be discovered or arbitrarily assigned in Step 4 later. By joining the schema prototype trees through those foreign keys in a bottom-up fashion, we could actually reconstruct larger schema trees, which are similar to those created using the algorithm stated in section 3.1. The concept of *bottom-up construction* method is similar to the algorithm proposed in [49]. In other words, the technique used in [49] is the subset of the three methods stated by us.

Hybrid Construction Method

Hybrid construction method combines the techniques of handling roots while constructing tree from both top-down and bottom-up construction methods. We propose *hybrid* method by analysing the relation between DTD structure and the possible characteristics in the XML data. Based on the relational database theory, *hybrid* method should produce relational schema having least redundancy of data. For roots determined by **rule 2, 3** or **5**, their tree construction processes

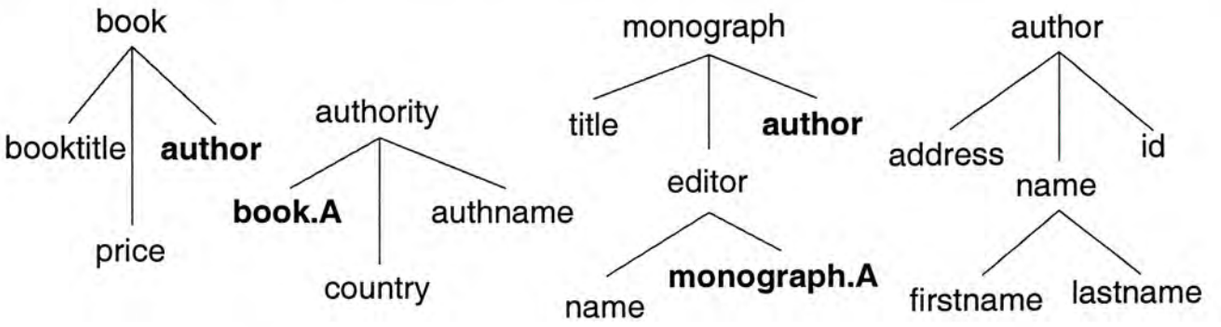

Figure 3.18: Schema prototype trees construction from Figure 3.15 using hybrid construction method

are the same as the one in section 3.1. However, during the tree construction if we visit an element declaration of a *root* element which is determined by **rule 3**, we would create a new node for that newly visited root element. This is because we expect a tendency of *M:1* relationship from the parent of the element to the root element. For example probably many books may be written by the same author, so it is likely to include the key of author as part of the relation for the book. On the other hand, if we visit an element declaration of a *root* element which is determined by **rule 4**, we will not perform any node addition to the schema prototype tree. For roots determined by **rule 4**, we have to find out their only ancestor in the DTD, and add the corresponding nodes as the leaf nodes of the roots in the schema prototype trees. This is because we expect a tendency of *1:M* relationship from the parent of the element to the root element. For example we expect one book will likely be related to multiple authorities. Therefore it is likely to include a key of book as an attribute in the relation for authority. The schema prototype trees constructed by this method is shown in Figure 3.18.

Note that both node of keys and nodes of foreign keys can be in the trees. Those keys can be discovered or arbitrarily assigned in Step 4 later.

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

Figure 3.19: The relational schema prototypes generated from the trees in Figure 3.16

3.2.3 Step 3: Generate Relational Schema Prototype

Just as section 3.1, we generate schema prototype by inlining all the necessary descendants of the schema prototype tree, including leaf nodes and the node marked with a "#", starting from the root. However, we will not inline those key nodes or foreign key nodes (depending on what tree construction method we have used in the previous step) in this step. We will decide how to add them (using found candidate keys or assigning a key attribute) into the relational schema after we discover all the functional dependencies and keys in Step 4. For all schema trees created from 3 different methods, their relational schemas are thus the same. The relational schema prototypes generated from the schema prototype trees presented in the previous step, regardless which construction method has been used, are shown in Figure 3.19.

Note that we can be sure that there not be two nodes having the same name inside the same tree. So we do no have to use the naming scheme used in section 3.1.

3.2.4 Step 4: Discover Functional Dependencies and Candidate Keys

With the generated schema prototypes, we follow exactly the process in section 3.1's Step 4. However, the main difference between the algorithm in section 3.1 and the algorithm proposed here is that: in the new algorithm during Step 2 and Step 3, we have actually *pre-decompose* the DTD into smaller schema prototypes. As a result, in this algorithm the cost of discovering functional dependencies and candidate keys, which is exponential to the number of attributes, would be smaller since the number of attributes in each schema prototypes is smaller compared with those more global schema prototypes in section 3.1.

As mentioned in Step 3, we have to determine the candidate keys for the schema prototypes in this step so as to refine the schema prototypes. However, if a candidate key turns out to contain many attributes or is very lengthy, then we may also assign a new artificial ID field to serve as the key, unique ID's will be generated by the system for such a key. This method of an artificial ID is heavily used in other methods where functional dependencies are not utilized. We only adopted the method when an artificial key is really needed since we want to prevent adding attributes that are unrelated to the actual XML data as much as possible. This technique can also be used in the algorithm in Section 3.1. The procedure is shown in Figure 3.20.

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

`table:book - {booktitle}`

`table:authority - {country, authname}`

`table:monograph - {title}`

Procedure REFINING_SCHEMA_PROTOTYPES

```
1  Set numAttr as maximum size of our required candidate key;
2  for each schema prototype S
3     FD_DISCOVERY_AND_CANDIDATE_KEY_DISCOVERY(S);
4     if no key found that has size ≤ numAttr
5     then begin
6        Arbitrarily assigns an ID field in S as the candidate key;
7     end
8     else begin
9        Assigns the one with minimum number of attributes as the candidate key
         of S;
10    end
11    for each other schema prototype S' which's schema prototype tree has a
      key/foreign key nodes of S
12       Adds the attributes(s) of the candidate key into S';
13    end for
14 end for
```

Figure 3.20: Procedure for deciding the candidate keys for the schema prototypes

`table:author` - {`id`},{`lastname`, `address`}

According to the procedure we stated in Figure 3.20, we use `booktitle` as the key for `table:book`. We assign an `assignID` field to `table:authority`. `title` is used as `table:monograph`'s key while `id` is chosen as the key for `table:author`. All the keys or foreign keys to other relations are added in the format `table_name.table_key`.

The relational schema prototypes generated by the three construction methods are shown in Figure 3.21, 3.22 and 3.23 respectively.

3.2.5 Step 5: Normalize the Relational Schema Prototypes

With the functional dependencies, candidate keys and the set of refined schema prototypes, we can simply normalize the relational schema prototype to a set of

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

Figure 3.21: The relational schema prototypes generated from the trees in Figure 3.16

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

Figure 3.22: The relational schema prototypes generated from the trees in Figure 3.17

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

Figure 3.23: The relational schema prototypes generated from the trees in Figure 3.18

new relations, if the refined schema prototypes can be further decomposed. This step is similar to the corresponding step in section 3.1. After normalization, we can then produce the relational schemas for the XML and use them to map the XML data into relational database.

3.2.6 Discussion

In this section, we have proposed another schema algorithm DTD-splitting Schema Extraction Algorithm. Unlike the previous algorithm, this DTD-splitting algorithm relies more on the first 3 steps. Based on the relational database theory, we try to predict some characteristics of the XML data from the DTD, hence perform a certain level of schema decomposition (DTD split) before the step for finding FDs and keys. As a result, the size of schema prototypes can be be smaller than those in section 3.1, thus alleviating the possible cost in the FD discovery steps, as mentioned before.

In this algorithm, we also proposed three different tree construction methods. Different tree construction methods (*Top-down, Bottom-up and Hybrid*) might lead to different relational schemas later. We think that *Hybrid* method should be the preferable methods as it combines the possible *1:M* handling from *Top-down* method, together with the possible *M:1* handling from *Bottom-up* method,

Even though *Hybrid* method is capable of handling both possible *1:M* and *M:1* mapping cardinalities in the relations, our algorithm is still unable to handle the case for *M:N* relation. Due to the fact that there will be a bigger chance for multivalued dependencies to hold inside a *M:N* relation, we proposed to discover if there is any multivalued dependencies inside the relation as well. The proposed algorithm is described in the following chapter.

3.3 Experimental Results

XML has become more popular in recent years. However, we found that most of the XML dataset available publicly are still *document-centric* while we have proposed algorithms to apply on *data-centric* XML. By *document-centric* we mean that XML is mainly used as a sophisticated version of HTML (e.g. for doing web document styling) while by *data-centric* we mean that XML is mainly used to describe data (e.g. for being the format of Electronic Data Interchange (EDI) or E-commerce Application). Readers can find a more detailed discussion about the difference between these two types of XML documents in Chapeter 5. We predict that more and more *data-centric* XML should be available on the WWW in the coming future. Right now, we illustrate the effects of our algorithms by applying them on a set of real-life XML data from *ACM SIGMOD Record: XML Version*, which is available at [46], and a set of synthetic XML data, which is generated according to the example DTD used in Figure 3.15. We implement each step of our algorithms in Perl except the step of discovering functional dependencies and candidate keys. For the step of discovering functional dependencies and candidate keys, we modify and use an implementation of *TANE* [29] written in C and compiled with a GNU C compiler. The original implementation of *TANE* is available at [28]. All the experiments were run in an isolated SUN Sparc Ultra1 workstation with SunOS 5.6.

3.3.1 Real Life XML Data: SIGMOD Record XML

In [46], there is a large XML document, `sigmodrecord.xml`, together with its DTD, `sigmodrecord.dtd`. `sigmodrecord.xml` contains information of more than 60 past issues of the magazine *SIGMOD Record* including information of about 1300 articles and information of more than 3000 authors. It is one of the largest *data-centric* XML document available on the WWW now. The

```
<!ELEMENT SigmodRecord (issue*) >
<!ELEMENT issue (volume,number,articles) >
<!ELEMENT volume (#PCDATA)>
<!ELEMENT number (#PCDATA)>
<!ELEMENT articles (article*) >
<!ELEMENT article (title, initPage, endPage, authors) >
<!ELEMENT title (#PCDATA)>
<!ELEMENT initPage (#PCDATA)>
<!ELEMENT endPage (#PCDATA)>
<!ELEMENT authors (author*) >
<!ELEMENT author (#PCDATA)>
<!ATTLIST author position CDATA >
```

Figure 3.24: `sigmodrecord.dtd`

`sigmodrecord.dtd` is shown in Figure 3.24, while a fraction of `sigmodrecord.xml` is shown in Figure 3.47.

The experimental results based on `sigmodrecord.dtd` and `sigmodrecord.xml` for both of our algorithms are presented in the following sections.

Experimental Result for Global Schema Extraction Algorithm

After the first three steps, the resulting schema prototypes are shown in Figure 3.25.

We then map the data in `sigmodrecord.xml` to produce a prototype table according to this schema prototype. Figure 3.26 shows a fraction of the mapped data in the prototype table.

The prototype table is then used in discovering the functional dependencies and candidate keys. The result is shown in Figure 3.27. It is interesting to note that the set of resulting functional dependencies is not exactly the same as

```
table(
SigmodRecord.issue.articles.article.authors.author.position, (1)
SigmodRecord.issue.articles.article.authors.author, (2)
SigmodRecord.issue.articles.article.title, (3)
SigmodRecord.issue.articles.article.initPage, (4)
SigmodRecord.issue.articles.article.endPage, (5)
SigmodRecord.issue.volume, (6)
SigmodRecord.issue.number (7)
)
```

Figure 3.25: The relational schema prototype for `sigmodrecord.xml`, which is generated by using Global Schema Extraction Algorithm

```
00,Catriel Beeri,A Note on Decompositions of Relational Databases.,33,37,12,1
01,Moshe Y. Vardi,A Note on Decompositions of Relational Databases.,33,37,12,1
00,Peter B. Miller,BUSINESS - An End-User Oriented Application Development Language.,38,69,12,1
01,Sergey Tetelbaum,BUSINESS - An End-User Oriented Application Development Language.,38,69,12,1
02,Kincade N. Webb,BUSINESS - An End-User Oriented Application Development Language.,38,69,12,1
00,Antonio L. Furtado,Horizontal Decomposition to Improve a Non-BCNF Scheme.,26,32,12,1
00,Kn. I. Kilov,Meta-Database Architecture for Relational DBMS.,18,25,12,1
01,I. A. Popova,Meta-Database Architecture for Relational DBMS.,18,25,12,1
00,James H. Burrows,Actual Conversion Experiences.,20,33,12,2
00,James P. Fry,Conversion Technology. An Assessment.,39,61,12,2
00,John L. Berg,Data Base Directions II: The Conversion Problem - Editorial.,3,3,12,2
00,Richard L. Nolan,Establishing Management Objectives.,9,19,12,2
00,Mayford L. Roark,Evolution in Computer Systems.,4,8,12,2
00,Milt Bryce,Standards.,34,38,12,2
00,Henry M. Walker,Administering a Distributed Data Base Management System.,86,99,12,3
00,Haran Boral,Database Research Activities at the University of Wisconsin.,19,26,12,3
01,David J. DeWitt,Database Research Activities at the University of Wisconsin.,19,26,12,3
02,Randy H. Katz,Database Research Activities at the University of Wisconsin.,19,26,12,3
03,Anthony C. Klug,Database Research Activities at the University of Wisconsin.,19,26,12,3
.
.
.
```

Figure 3.26: Fraction of the mapped data from `sigmodrecord.xml`, which is then used in functional dependency discovery step (*Global* algorithm)

```
Table1
No. of tuples:              3133
No. of attributes:          7
FDs found:                                  Keys found:

2 3 -> 1          3 6 7 -> 4 5              {2 3 7}
3 5 -> 4          5 6 7 -> 3 4              {2 4 5 6}
3 5 7 -> 6        1 2 4 6 -> 3              {2 4 5 6}
3 5 6 -> 7        1 2 5 7 -> 4              {2 4 6 7}
3 4 7 -> 5 6      1 2 4 7 -> 5
4 6 7 -> 3 5

============================================
```

Figure 3.27: Functional dependencies and candidate keys found from the prototype table in Figure 3.26

what we have predicted before the experiment. For example, we expect that in `sigmodrecord.xml`, `title` (3) can at least determine `initPage` (4) and `endPage` (5) since we think that there should not be two research articles with exactly the same title. However, the functional dependency $3 \rightarrow\rightarrow 4\ 5$ is not in the set of found dependencies. The reason is that for each different issue of SIGMOD Record magazine, there must be an article titled "Editor's Notes" at the beginning, thus breaking the functional dependency predicted by us. If we want to take those nearly-formed functional dependencies in the XML data into consideration, *TANE* can discover those nearly-formed ones using the concept of *approximate dependency* in [32]. Using 3NF decomposition as an example, one of the possible resulting relation schema for `sigmodrecord.xml` is shown in Figure 3.28

Note that Figure 3.28 is just one of the possible designs for the relational schema. With the functional dependencies found in the XML data, the user can decompose the schema prototypes into other good relational database designs for the XML data.

```
table1{
SigmodRecord.issue.articles.article.title , (3)
SigmodRecord.issue.articles.article.initPage, (4)
SigmodRecord.issue.articles.article.endPage, (5)
SigmodRecord.issue.volume, (6)
SigmodRecord.issue.number (7)
}

table2{
SigmodRecord.issue.articles.article.authors.author, (2)
SigmodRecord.issue.articles.article.title , (3)
SigmodRecord.issue.number (7)
}

table3{
SigmodRecord.issue.articles.article.authors.author.position, (1)
SigmodRecord.issue.articles.article.authors.author, (2)
SigmodRecord.issue.articles.article.initPage, (4)
SigmodRecord.issue.articles.article.endPage, (5)
SigmodRecord.issue.number (7)
}

table4{
SigmodRecord.issue.articles.article.authors.author.position, (1)
SigmodRecord.issue.articles.article.authors.author, (2)
SigmodRecord.issue.articles.article.title , (3)
SigmodRecord.issue.articles.article.initPage, (4)
SigmodRecord.issue.volume, (6)
}
```

Figure 3.28: Relational schemas produced for `sigmodrecord.xml` based on 3NF decomposition

Experimental Result for DTD-splitting Schema Extraction Algorithm

After the first three steps, the resulting schema prototypes are shown in Figure 3.29. Note that even after step 2, the tree structures of each construction methods should have slightly difference, the schema prototypes produced are the same.

table:issue(	table:article(	table:author(
volume, (1)	title, (1)	position, (1)
number, (2)	initPage, (2)	author, (2)
)	endPage, (3)	)
	)	

Figure 3.29: The relational schema prototype for `sigmodrecord.xml`, which is generated by using DTD-splitting Schema Extraction Algorithm

We then map the data in `sigmodrecord.xml` to produce a prototype table according to this schema prototype. Figure 3.30 shows a fraction of the mapped data in the prototype table.

The prototype tables are then used in discovering the functional dependencies and candidate keys. The results for each prototype tables are shown in Figure 3.31. For the procedure shown in Figure 3.20, we set *numAttr* as 1 only as we observe that the number of attributes in each schema prototypes are relatively small. The resulting relation schemas for `sigmodrecord.xml` are then produced. The schemas produced from *Top-down* method is shown in Figure 3.32. For *sigmodrecord.xml*, the relational schemas for both *Bottom-up* and *Hybrid* method are the same, as shown in Figure 3.33.

The three construction methods proposed in our *DTD-splitting* Schema Extraction Algorithm provide more flexibilities in producing the relational schema. With the support of the relational database concept, it is quite obvious that *Hybrid* methods should lead to a better relational schema design and have less redundancy of data in the resulting table. To better illustrate this, we run another experiment using a set of synthetic XML data.

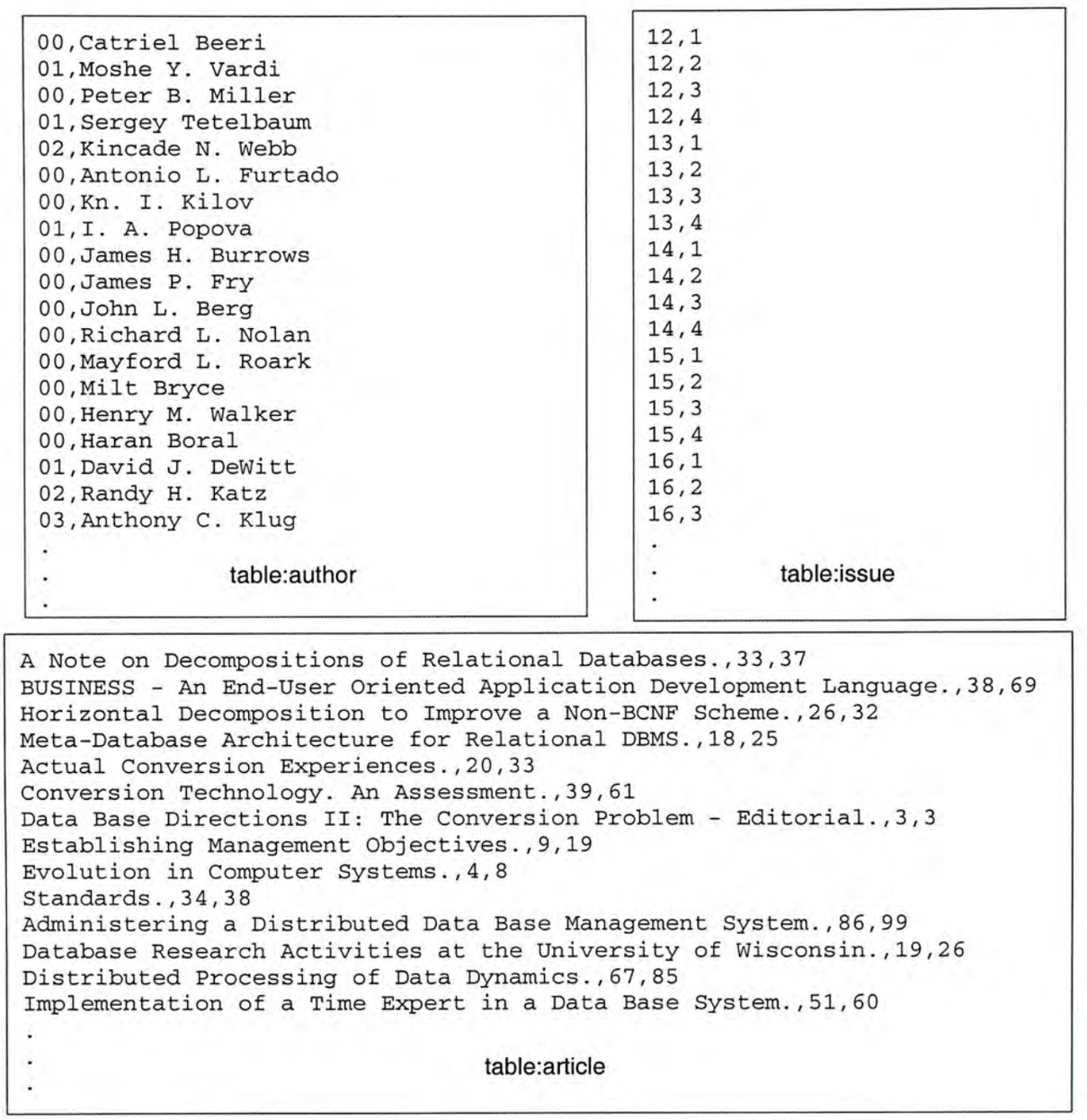

Figure 3.30: Fraction of the mapped table prototypes from `sigmodrecord.xml`, which is then used in functional dependency discovery step (*DTD-splitting* algorithm)

```
table:author
No. of tuples:              3113
No. of attributes:          2
FDs found:                          Keys found:

                                    {1 2}

=============================================

table:article
No. of tuples:              1248
No. of attributes:          3
FDs found:                          Keys found:

1 3 -> 2                            {1 3}

=============================================

table:issue
No. of tuples:              63
No. of attributes:          2
FDs found:                          Keys found:

                                    {1 2}

=============================================
```

Figure 3.31: Functional dependencies and candidate keys found from the prototype tables in Figure 3.30

```
table:issue(
volume, (1)
number, (2)
assignID , (3)
table:article.assignID    (4)
)

table:article(
title, (1)
initPage, (2)
endPage, (3)
assignID , (4)
table:author.assignID    (5)
)

table:author(
position, (1)
author, (2)
assignID , (3)
)
```

Figure 3.32: The relational schemas for `sigmodrecord.xml` by using *Top-down* method

```
table:issue(
volume, (1)
number, (2)
assignID (3)
)

table:article(
title, (1)
initPage, (2)
endPage, (3)
assignID, (4)
table:issue.assignID (5)
)

table:author(
position, (1)
author, (2)
assignID, (3)
table:article.assignID (4)
)
```

Figure 3.33: The relational schemas for `sigmodrecord.xml` by using *Bottom-up* method or *Hybrid* method

3.3.2 Synthetic XML Data

In our second experiment, we generate a set of XML data according to the example DTD used in Figure 3.15. To give reasonable characteristics to the XML data, we make some assumptions when generating the XML data:

(1)No two books or two monographs have the same title.

(2)No two authors, which have the same name, share the same address.

(3)Author has one address only.

(4)Author can appear in more than one books and/or monographs.

(5)Authority can appear in more than one books.

(6)Multiple subelement occurrences of authority and monograph range between 0 to 5.

In total, XML data for more than 200 books and monographs are generated according to the assumption stated. A fraction of our synthetic XML data set is shown in Figure 3.48.

table:book (	*table:monograph* (
book.booktitle, (1)	monograph.title, (1)
book.price, (2)	monograph.author.id, (2)
book.author.id, (3)	monograph.author.name.firstname, (3)
book.author.name.firstname, (4)	monograph.author.name.lastname, (4)
book.author.name.lastname, (5)	monograph.author.address, (5)
book.author.address, (6)	monograph.editor.name, (6)
book.authority.authname, (7)	monograph.editor.monograph.title (7)
book.authority.country (8)	)
)	

Figure 3.34: The relational schema prototype for synthetic XML data, which is generated by using Global Schema Extraction Algorithm

Experimental Result for Global Schema Extraction Algorithm

After the first three steps, the resulting schema prototypes for the set of synthetic XML data are shown in Figure 3.34.

We then map the synthetic XML data to produce a prototype table according to this schema prototype. Figure 3.35 shows a fraction of the mapped data in the prototype table.

The prototype table is then used in discovering the functional dependencies and candidate keys. The result is shown in Figure 3.36.

Using 3NF decomposition as an example, one of the possible resulting relation schema for the synthetic XML data is shown in Figure 3.37. Note that since `table:book-3` and `table:monograph-4` are the same after comparing the attributes in them, they are replaced by a common table `table:author`.

Experimental Result for DTD-splitting Schema Extraction Algorithm

After the first three steps, the resulting schema prototypes are shown in Figure 3.19. Note that even after step 2, the tree structures of each construction methods should have slightly difference, the schema prototypes produced are the same.

XML book, 19.9, 1, Men-Hin, Yan, "Hung Hom, Hong Kong", NY Times, US
XML book, 19.9, 1, Men-Hin, Yan, "Hung Hom, Hong Kong", PC Home, HK
XML book, 19.9, 1, Men-Hin, Yan, "Hung Hom, Hong Kong", PC Weekly, HK
XML in a nutshell, 99.9, 2, Ham, Wong, "CUHK, Hong Kong", Tokyo Times, JAP
XML in a nutshell, 99.9, 2, Ham, Wong, "CUHK, Hong Kong", PC Magazine, HK
XML in a nutshell, 99.9, 2, Ham, Wong, "CUHK, Hong Kong", DC Times, US
XML cook book, 99.9, 3, Roy, Chan, "CUHK, Hong Kong", PC Times, HK
XML cook book, 99.9, 3, Roy, Chan, "CUHK, Hong Kong", DC Times, US
XML cook book, 99.9, 3, Roy, Chan, "CUHK, Hong Kong", PC Zone, HK
.
.
.

table:book

XML monograph, 2, Ham, Wong, "CUHK, Hong Kong", Roy Chan, XML monograph
XML monograph, 2, Ham, Wong, "CUHK, Hong Kong", Roy Chan, DTD monograph
XML monograph, 2, Ham, Wong, "CUHK, Hong Kong", Roy Chan, mono XML
XSL monograph, 2, Ham, Wong, "CUHK, Hong Kong", Willis Chan, SGML monograph
XSL monograph, 2, Ham, Wong, "CUHK, Hong Kong", Willis Chan, XSL monograph
DTD monograph, 4, Brenda, Chan, "Choi Hung, Hong Kong", Roy Chan, XML monograph
DTD monograph, 4, Brenda, Chan, "Choi Hung, Hong Kong", Roy Chan, DTD monograph
DTD monograph, 4, Brenda, Chan, "Choi Hung, Hong Kong", Roy Chan, mono XML
monograph XSLT, 4, Brenda, Chan, "Choi Hung, Hong Kong", Henry Hui, SGML monograph
.
.
.

table:monograph

Figure 3.35: Fraction of the mapped data from synthetic XML data, which is then used in functional dependency discovery step (*Global* algorithm)

```
table:book
No. of tuples:              507
No. of attributes:   8
FDs found:                          Keys found:

1 -> 2 3          4 5 6 -> 3    {1 7 8}
3 -> 4 5 6


==========================================

table:monograph
No. of tuples:              487
No. of attributes:   7
FDs found:                          Keys found:

1 -> 2 6          7 -> 6        {1 7}
2 -> 3 4 5        3 4 5 -> 2

==========================================
```

Figure 3.36: Functional dependencies and candidate keys found from the prototype table in Figure 3.35

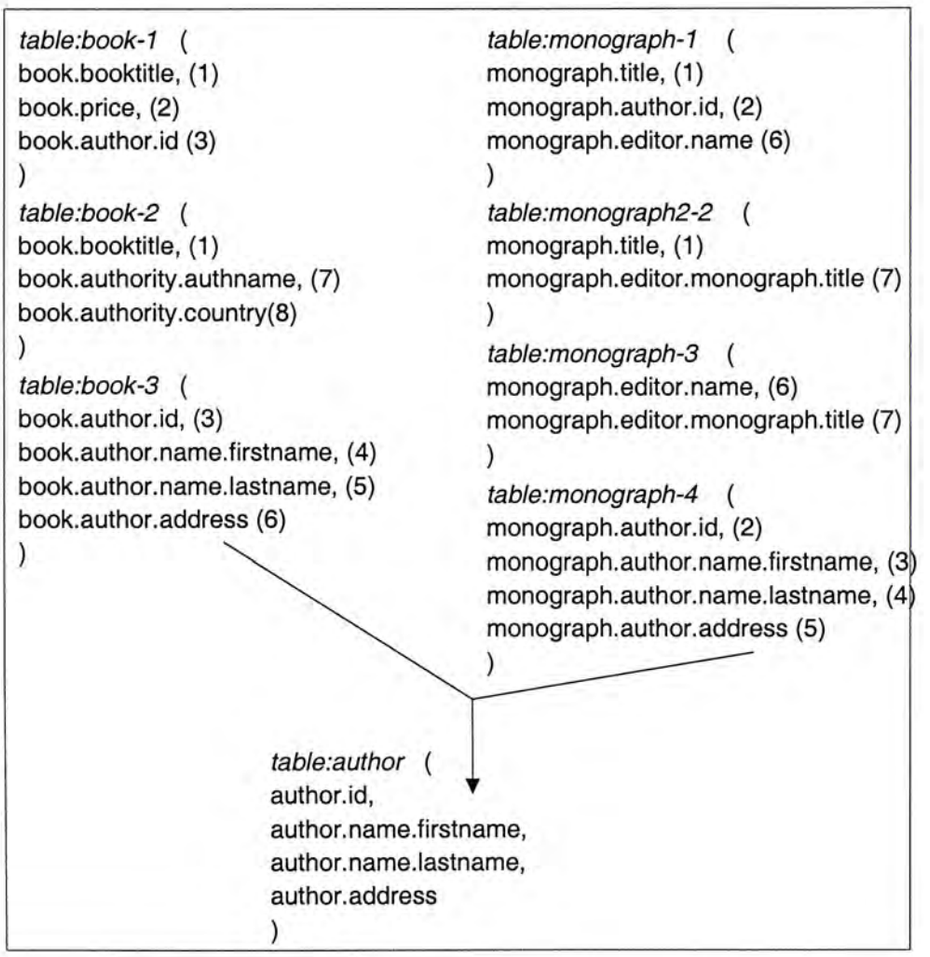

Figure 3.37: Relational schemas produced for the synthetic XML data based on 3NF decomposition

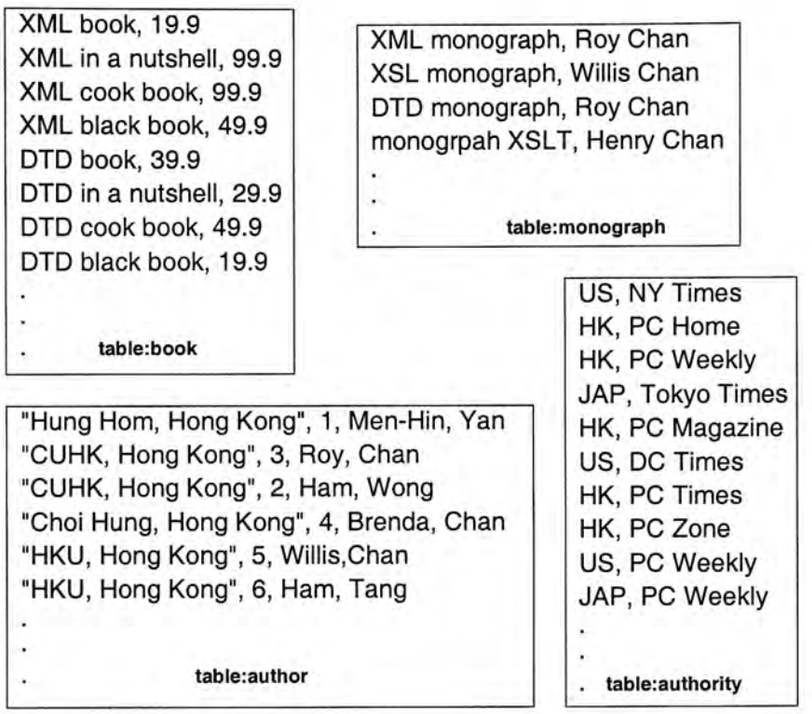

Figure 3.38: Fraction of the mapped table prototypes from the sythetic XML data, which is then used in functional dependency discovery step (*DTD-splitting* algorithm)

We then map the data to produce a prototype table according to this schema prototype. Figure 3.38 shows a fraction of the mapped data in the prototype table.

The prototype tables are then used in discovering the functional dependencies and candidate keys. The results for each prototype tables are shown in Figure 3.39. For the procedure shown in Figure 3.20, we set *numAttr* as 1 only as we observe that the number of attributes in each schema prototypes are relatively small. The resulting relation schemas for the synthetic XML data are then produced. The relation schemas for *Top-down*, *Bottom-up* and *Hybrid* methods are shown in Figure 3.40.

As we mentioned before, just like our *DTD-splitting* algorithm, the methods proposed in [49] also based on constructing schema from the characteristics in the DTD. And the *shared inlining method* and *hybrid inlining method* introduced in

```
table:book
No. of tuples:              117
No. of attributes:   2
FDs found:                          Keys found:
                                    {1}
==========================================
table:authority
No. of tuples:              312
No. of attributes:   2
FDs found:                          Keys found:
                                    {1 2}
==========================================
table:author
No. of tuples:              156
No. of attributes:   4
FDs found:                          Keys found:
1 4 -> 2 3                          {2}
==========================================
table:monograph
No. of tuples:              132
No. of attributes:   2
FDs found:                          Keys found:
                                    {1}
==========================================
```

Figure 3.39: Functional dependencies and candidate keys found from the prototype table in Figure 3.38

table:book (	*table:author* (	*table:authority* (	*table:monograph* (
booktitle (1),	address (1),	country (1),	title (1),
price (2),	id (2),	authname (2),	name (2),
authority.assignID (3),	firstname (3),	assignID (3)	monograph.title (3),
author.id (4)	lastname (4)	)	author.id (4)
)	)		)

Top-down Method

table:book (	*table:author* (	*table:authority* (	*table:monograph* (
booktitle (1),	address (1),	country (1),	title (1),
price (2),	id (2),	authname (2),	name (2),
)	firstname (3),	assignID (3)	monograph.title (3)
	lastname (4),	book.booktitle (4)	)
	monograph.title (5),	)	
	book.booktitle (6)		
	)		

Bottom-up Method

table:book (	*table:author* (	*table:authority* (	*table:monograph* (
booktitle (1),	address (1),	country (1),	title (1),
price (2),	id (2),	authname (2),	name (2),
author.id (3)	firstname (3),	assignID (3)	monograph.title (3),
)	lastname (4),	book.booktitle (4)	author.id (4)
	)	)	)

Hybrid Method

Figure 3.40: Relational schemas of the synthetic XML data produced by *DTD-splitting* algorithm

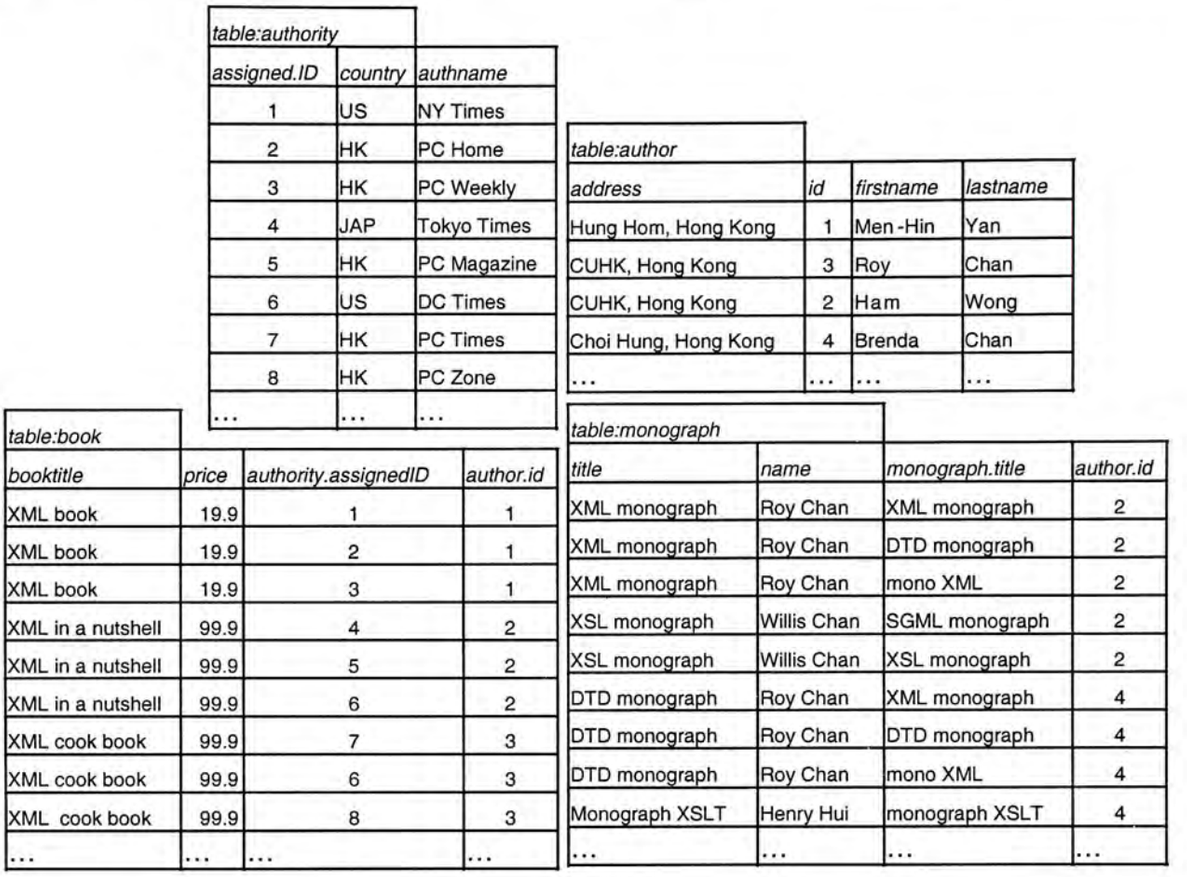

table:authority

assigned.ID	*country*	*authname*
1	US	NY Times
2	HK	PC Home
3	HK	PC Weekly
4	JAP	Tokyo Times
5	HK	PC Magazine
6	US	DC Times
7	HK	PC Times
8	HK	PC Zone
...	...	...

table:author

address	*id*	*firstname*	*lastname*
Hung Hom, Hong Kong	1	Men-Hin	Yan
CUHK, Hong Kong	3	Roy	Chan
CUHK, Hong Kong	2	Ham	Wong
Choi Hung, Hong Kong	4	Brenda	Chan
...	...	...	...

table:book

booktitle	*price*	*authority.assignedID*	*author.id*
XML book	19.9	1	1
XML book	19.9	2	1
XML book	19.9	3	1
XML in a nutshell	99.9	4	2
XML in a nutshell	99.9	5	2
XML in a nutshell	99.9	6	2
XML cook book	99.9	7	3
XML cook book	99.9	6	3
XML cook book	99.9	8	3
...	...	...	...

table:monograph

title	*name*	*monograph.title*	*author.id*
XML monograph	Roy Chan	XML monograph	2
XML monograph	Roy Chan	DTD monograph	2
XML monograph	Roy Chan	mono XML	2
XSL monograph	Willis Chan	SGML monograph	2
XSL monograph	Willis Chan	XSL monograph	2
DTD monograph	Roy Chan	XML monograph	4
DTD monograph	Roy Chan	DTD monograph	4
DTD monograph	Roy Chan	mono XML	4
Monograph XSLT	Henry Hui	monograph XSLT	4
...	...	...	...

Figure 3.41: Tables for top-down method

[49] are adopted in many current research projects like [33, 31, 42, 52]. To show how our proposed methods in *DTD-splitting* algorithm outperform the methods introduced in [49] in terms of minimizing data redundancy, we apply them on the synthetic XML data and present fraction of the relational tables for the three tree construction methods proposed by us, as well as those for the *shared inlining method* and *hybrid inlining method* in [49].

The tables produced from *Top-down* method is shown in Figure 3.41. The tables produced from *Bottom-up* method is shown in Figure 3.42. The schemas produced from *Hybrid* method is shown in Figure 3.43.

The relational schema extracted by *shared inlining method* and *hybrid inlining method* are shown in Figure 3.44. The tables produced from *shared inlining method* are shown in Figure 3.45. The tables produced from *hybrid inlining*

table:authority

assigned.ID	*country*	*authname*	*book.booktitle*
1	US	NY Times	XML book
2	HK	PC Home	XML book
3	HK	PC Weekly	XML book
4	JAP	Tokyo Times	XML in a nutshell
4	JAP	Tokyo Times	XML cook book
5	HK	PC Magazine	XML in a nutshell
6	US	DC Times	XML in a nutshell
7	HK	PC Times	XML cook book
8	JAP	PC Zone	XML cook book
...	...	...	...

table:monograph

title	*name*	*monograph.title*
XML monograph	Roy Chan	XML monograph
XML monograph	Roy Chan	DTD monograph
XML monograph	Roy Chan	mono XML
XSL monograph	Willis Chan	SGML monograph
XSL monograph	Willis Chan	XSL monograph
DTD monograph	Roy Chan	XML monograph
DTD monograph	Roy Chan	DTD monograph
DTD monograph	Roy Chan	mono XML
monograph XSLT	Henry Hui	monograph XSLT
...	....	...

table:author

address	*id*	*firstname*	*lastname*	*monograph.title*	*book.booktitle*
Hung Hom, Hong Kong	1	Men-Hin	Yan	-	XML book
CUHK, Hong Kong	3	Roy	Chan	-	XML cookbook
CUHK, Hong Kong	2	Ham	Wong	XML monograph	XML in a nutshell
CUHK, Hong Kong	2	Ham	Wong	XSL monograph	XML in a nutshell
Choi Hung, Hong Kong	4	Brenda	Chan	DTD monograph	-
...	...	...	...	...	...

table:book

booktitle	*price*
XML book	19.9
XML in a nutshell	99.9
XML cook book	99.9
...	...

Figure 3.42: Tables for bottom-up method

table:author

address	*id*	*firstname*	*lastname*
Hung Hom, Hong Kong	1	Men-Hin	Yan
CUHK, Hong Kong	3	Roy	Chan
CUHK, Hong Kong	2	Ham	Wong
Choi Hung, Hong Kong	4	Brenda	Chan
...	...	...	...

table:book

booktitle	*price*	*author.id*
XML book	19.9	1
XML in a nutshell	99.9	2
XML cook book	99.9	3
...	...	...

table:authority

country	*assigned.ID*	*authname*	*book.booktitle*
US	1	NY Times	XML book
HK	2	PC Home	XML book
HK	3	PC Weekly	XML book
JAP	4	Tokyo Times	XML in a nutshell
JAP	4	Tokyo Times	XML cook book
HK	5	PC Magazine	XML in a nutshell
US	6	DC Times	XML in a nutshell
HK	7	PC Times	XML cook book
JAP	8	PC Zone	XML cook book
...	...	...	...

table:monograph

title	*name*	*monograph.title*	*author.id*
XML monograph	Roy Chan	XML monograph	2
XML monograph	Roy Chan	DTD monograph	2
XML monograph	Roy Chan	mono XML	2
XSL monograph	Willis Chan	SGML monograph	2
XSL monograph	Willis Chan	XSL monograph	2
DTD monograph	Roy Chan	XML monograph	4
DTD monograph	Roy Chan	DTD monograph	4
DTD monograph	Roy Chan	mono XML	4
monograph XSLT	Henry Hui	monograph XSLT	4
...	...	...	...

Figure 3.43: Tables for hybrid method

method are shown in Figure 3.46.

We use the tables constructed by our *hybrid* method, as shown in Figure 3.43 to compare with those generated by [49]'s methods.

When compared with *shared inlining method* in Figure 3.45, it is obvious that the relational schemas constructed by our algorithm use less attributes. The main reason is that we prevent excessive use of artificial IDs for each table. Moreover, the tables produced by our algorithm has less data redundancy. For example, in the table `table:author`, all the data for `author` is repeated in *shared inlining method* for each different `book` and `monograph` foreign keys (`parentID` + `parentCODE`). While in our algorithm, we use key of `table:author` (`author.id`) in `table:book` and `table:monograph`, instead of using using foreign keys of `table:book` and `table:monograph` in `table:author`. As a result, redundancy of data can be prevented.

When compared with *hybrid inlining method* in Figure 3.46, it is obvious that relational schemas constructed by our algorithm use much less attributes. Based on the rule in *hybrid inlining method*, `author`'s attributes have to inlined all into `table:book` and `table:monograph`. The reason for [49] to propose that is to reduce the number of table joins in query. However, as shown in Figure 3.46, the method would create much redundancy among tables when the number of attributes for the extra inlining is large.

It is clear that our algorithm produces more efficient relational schema design than [49] does. Since the DTD of our synthetic XML data is not large, the number of attributes in each table is relatively small. As a result, the step of dependency discovery is not really significant in this case as no important functional dependencies are found. However, more reasonable and effective schemas can still be produced by our algorithm. This show that even without the step the dependency discovery, our algorithm still outperform the methods proposed

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

Shared Inlining Method

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

Hybrid Inlining Method

Figure 3.44: Relational schema for the methods proposed in [49]

by [49].

3.3.3 Discussion

The experiments on `sigmodrecord.xml` and our synthetic XML dataset illustrates the schema extraction effects for both of our schema extraction algorithms. From the relational schemas extracted, we would discuss some interesting observations below.

It is obvious that the relational schemas produced by global schema extraction algorithm (or *Global* algorithm) may not the same as those produced by DTD-splitting schema extraction algorithm (or *DTD-splitting* algorithm). The reason for their difference is what we have mentioned before: *Global* algorithm relies more on discovering dependencies in the XML data while *DTD-splitting* algorithm relies more on pre-decomposition of schema prototypes. With such a

table:book

assignID	*booktitle*	*price*
1	XML book	19.9
2	XML in a nutshell	99.9
3	XML cook book	99.9
...	...	...

table:author

id	*parentID*	*parentCODE*	*address*	*firstname*	*lastname*
1	1	table:book	Hung Home, Hong Kong	Men -Hin	Yan
3	3	table:book	CUHK, Hong Kong	Roy	Chan
2	2	table:book	CUHK, Hong Kong	Ham	Wong
2	1	table:monograph	CUHK, Hong Kong	Ham	Wong
2	2	table: monograph	CUHK, Hong Kong	Ham	Wong
4	3	table:monograph	Choi Hung, Hong Kong	Brenda	Chan
...	...	...	...	...	...

table:authority

assigned.ID	*parentID*	*parentCODE*	*authname*	*country*
1	1	table:book	NY Times	US
2	1	table:book	PC Home	HK
3	1	table:book	PC Weekly	HK
4	2	table:book	Tokyo Times	JAP
4	3	table:book	Tokyo Times	JAP
5	2	table:book	PC Magazine	HK
6	2	table:book	DC Times	US
7	3	table:book	PC Times	HK
8	3	table:book	PC Zone	JAP
...	...	...	...	...

table:monograph

assignedID	*parentID*	*parentCODE*	*title*	*editor.name*
1	1	table:monograph	XML monograph	Roy Chan
1	3	table:monograph	XML monograph	Roy Chan
1	9	table:monograph	XML monograph	Roy Chan
2	2	table:monograph	XSL monograph	Willis Chan
2	8	tab le:monograph	XSL monograph	Willis Chan
3	1	table:monograph	DTD monograph	Roy Chan
3	3	table:monograph	DTD monograph	Roy Chan
3	9	table:monograph	DTD monograph	Roy Chan
4	4	table:monograph	monograph XSLT	Henry Hui
...	...	...	...	....

Figure 3.45: Tables for [49]'s *shared inlining method*

table:authority

assigned.ID	parentID	parentCODE	*authname*	*country*
1	1	table:book	NY Times	US
2	1	table:book	PC Home	HK
3	1	table:book	PC Weekly	HK
4	2	table:book	Tokyo Times	JAP
4	3	table:book	Tokyo Times	JAP
5	2	table:book	PC Magazine	HK
6	2	tab le:book	DC Times	US
7	3	table:book	PC Times	HK
8	3	table:book	PC Zone	JAP
...	...	...	...	...

table:book

assignID	*booktitle*	*price*	*author.id*	*address*	*firstname*	*lastname*
1	XML book	19.9	1	Hung Hom, Hong Kong	Men -Hin	Yan
2	XML in a nutshell	99.9	2	CUHK, Hong Kong	Ham	Wong
3	XML cook book	99.9	3	CUHK, Hong Kong	Roy	Chan
...	...	...	...	...	...	...

table:monograph

assignedID	parentID	parentCODE	*title*	*editor. name*	*author.id*	*address*	*firstname*	*lastname*
1	1	table:monograph	XML monograph	Roy Chan	2	CUHK, Hong Kong	Ham	Wong
1	3	table:monograph	XML monograph	Roy Chan	2	CUHK, Hong Kong	Ham	Wong
1	9	table:monograph	XML monograph	Roy Chan	2	CUHK, Hong Kong	Ham	Wong
2	2	table:monograph	XSL monograph	Willis Chan	2	CUHK, Hong Kong	Ham	Wong
2	8	table:monograph	XSL monograph	Willis Chan	2	CUHK, Hong Kong	Ham	Wong
3	1	table:monograph	DTD monograph	Roy C han	4	Choi Hung, Hong Kong	Brenda	Chan
3	3	table:monograph	DTD monograph	Roy Chan	4	Choi Hung, Hong Kong	Brenda	Chan
3	9	table:monograph	DTD monograph	Roy Chan	4	Choi Hung, Hong Kong	Brenda	Chan
4	4	table:monograph	monograph XSLT	Henry Hui	4	Choi Hung, Hong Kong	Brenda	Chan
...	...	...	...	....	...	...	...	....

Figure 3.46: Tables for [49]'s *hybrid inlining method*

difference, both algorithms have their own trade offs:

Schemas created by *Global* algorithm are fully based on the real characteristics extracted from the XML data itself, thus ensuring all schema decompositions made in the algorithm are reasonably done based on relational database theory, Unlike the proposed schemas extraction by [25, 26], which need many extra data other than those in the XML file to maintain the schemas, it is possible for us to map all the data in the XML file into relational database without introducing any extra fields and data. However, since the schemas are decomposed totally based on discovered functional dependencies, the schemas might have to be updated when there is new XML data. The reason is that with any new additional XML data, some of the current functional dependencies might not hold anymore. Thus the schemas decomposed based on the old set of functional dependencies might be slightly different to the new set of functional dependencies found in the updated XML data, and we have to update the schemas again.

Schemas created by *DTD-splitting* algorithm are based more on the characteristics extracted from the DTD correspond to the XML data than the actual XML data. Because of the pre-decomposition of the schema prototypes, the cost of finding functional dependencies and keys are reduced. For XML data with relatively smaller DTD like the one of our synthetic XML data, even without the dependency discovery step, we still can produce reasonable relational schema design. Since the relational schemas depend more on DTD than the XML data, the relational schemas are less likely to be altered upon new addition of XML data. However, as the schemas are not produced based all on the functional dependencies and keys found inside the data, we might have to add an artificial key attribute into the schema. As a result, the resulting tables might contain fields that are unknown to users, and database users might not be able to use the tables directly. However, we do try to minimize the use of artificial keys as much as possible in our algorithm. Most likely those arbitrarily-added fields have to

be handled by the database system upon queries and updates.

Due to the exponential complexity in the number of attributes for functional dependency discovery, we suggest to use *Global* algorithm when the number of element and attribute declarations in DTD is not too large so that the number of attributes in the schema prototype is relatively smaller. When the DTD has a large number of element and attribute declarations, *DTD-splitting* algorithm should be used instead.

In the experiment of synthetic XML data, we observe that some of the relational schema generated by *DTD-splitting* algorithm could be further decomposed after the key/foreign key fields are added in them, i.e. we can undergo second round of functional dependency discovery process after the step of adding key/foreign key fields so as to further refine the schema design. On the other hand, *Global* algorithm ensures completed decomposition in one round of functional dependency discovery. Apart from further refine the schema design by another round of functional dependency discovery, we can also achieve that by finding multivalued dependencies in the XML data. We illustrate the use of multivalued dependencies in the experiments which are shown in the next chapter.

```
<SigmodRecord>
 <issue>
  <volume>12</volume>
  <number>1</number>
  <articles>
     <article>
          <title>A Note on Decompositions of Relational Databases.</title>
          <initPage>33</initPage>
          <endPage>37</endPage>
          <authors>
             <author position="00">Catriel Beeri</author>
             <author position="01">Moshe Y. Vardi</author>
          </authors>
     </article>
     <article>
          <title>BUSINESS - An End-User Oriented Application Development Language.</title>
          <initPage>38</initPage>
          <endPage>69</endPage>
          <authors>
             <author position="00">Peter B. Miller</author>
             <author position="01">Sergey Tetelbaum</author>
             <author position="02">Kincade N. Webb</author>
          </authors>
     </article>
          .
          .
          .
     <article>
          <title>Horizontal Decomposition to Improve a Non-BCNF Scheme.</title>
          <initPage>26</initPage>
          <endPage>32</endPage>
          <authors>
             <author position="00">Antonio L. Furtado</author>
          </authors>
     </article>
 </issue>
 <issue>
 .
 .
 .
 </issue>
</SigmodRecord>
```

Figure 3.47: Fraction of `sigmodrecord.xml`

```
<book>
    <booktitle>XML book</booktitle>
    <price>19.9</price>
    <author id="1">
        <name>
            <firstname>Men-hin</firstname>
            <lastname>Yan</lastname>
        </name>
        <address>Hung Hom, Hong Kong</address>
    </author>
    <authority>
        <authname>NY Times</authname>
        <country>US</country>
    </authority>
    <authority>
        <authname>PC Home</authname>
        <country>HK</country>
    </authority>
    <authority>
        <authname>PC Weekly</authname>
        <country>HK</country>
    </authority>
</book>
<book>
    <booktitle>XML in a nutshell</booktitle>
    <price>99.9</price>
    <author id="5">
        <name>
            <firstname>Willis</firstname>
            <lastname>Chan</lastname>
        </name>
        <address>CUHK, Hong Kong</address>
    </author>
    <authority>
        <authname>Tokyo Times</authname>
        <country>JAP</country>
    </authority>
    <authority>
        <authname>PC Magazine</authname>
        <country>HK</country>
    </authority>
    <authority>
        <authname>DC Times</authname>
        <country>US</country>
    </authority>
</book>
<book>
    <booktitle>XML cook book</booktitle>
    <price>99.9</price>
    <author id="3">
        <name>
            <firstname>Roy</firstname>
            <lastname>Chan</lastname>
        </name>
        <address>CUHK, Hong Kong</address>
    </author>
    <authority>
        <authname>PC Times</authname>
        <country>HK</country>
    </authority>
    <authority>
        <authname>Tokyo Times</authname>
        <country>JAP</country>
    </authority>
    <authority>
        <authname>PC Zone</authname>
        <country>HK</country>
    </authority>
</book>
<book>
    <booktitle>DTD book</booktitle>
    <price>39.9</price>
.
.
.
```

```
<monograph>
    <title>XML monograph</title>
    <author id="2">
        <name>
            <firstname>Ham</firstname>
            <lastname>Wong</lastname>
        </name>
        <address>CUHK, Hong Kong</address>
    </author>
    <editor name="Roy Chan">
        <monograph>
            <title>XML monograph</title>
        </monograph>
        <monograph>
            <title>DTD monograph</title>
        </monograph>
        <monograph>
            <title>mono XML</title>
        </monograph>
    </editor>
</monograoph>
<monograph>
    <title>XSL monograph</title>
    <author id="2">
        <name>
            <firstname>Ham</firstname>
            <lastname>Wong</lastname>
        </name>
        <address>CUHK, Hong Kong</address>
    </author>
    <editor name="Willis Chan">
        <monograph>
            <title>SGML monograph</title>
        </monograph>
        <monograph>
            <title>XSL monograph</title>
        </monograph>
    </editor>
</monograoph>
<monograph>
    <title>DTD monograph</title>
    <author id="4">
        <name>
            <firstname>Brenda</firstname>
            <lastname>Chan</lastname>
        </name>
        <address>Choi Hung, Hong Kong</address>
    </author>
    <editor name="Roy Chan">
        <monograph>
            <title>XML monograph</title>
        </monograph>
        <monograph>
            <title>DTD monograph</title>
        </monograph>
    </editor>
</monograph>
<monograph>
    <title>monograph XSLT</title>
    <author id="4">
        <name>
            <firstname>Brenda</firstname>
            <lastname>Chan</lastname>
        </name>
        <address>Choi Hung, Hong Kong</address>
    </author>
    <editor name="Henry Hui">
        <monograph>
            <title>monograph XSLT</title>
        </monograph>
    </editor>

.
.
.
```

Figure 3.48: Fraction of synthetic xml

Chapter 4

Finding Multivalued Dependencies

Apart from functional dependency, multivalued dependency is another important consideration in database design and analysis. We can detect multivalued dependencies to improve the database design by decomposition it to fourth normal form. As a result, apart from using functional dependencies, we consider using multivalued dependencies to improve the schema prototypes produced by our algorithms. As we mentioned in the Section 3.2.6 of Chapter 3,the motivation for using multivalued dependencies is that when we mapped the XML data according to the schema prototypes, it is possible to have *M:N* mapping situation as an element might allow more than one subelements to have multiple occurrences in it. Since it is possible for multivalued dependencies to hold inside *M:N*, for given XML data sets following the schema prototypes, we are also interested in discovering multivalued dependencies which might be useful for us to further refine the schema prototypes.

While extracting functional dependencies has received considerable attention [39, 32, 47, 29, 35, 37, 38] relatively less research effort has been put in finding multivalued dependencies. Based on the existing techniques of discovering

functional dependencies, especially [29], we propose a new algorithm for finding multivalued dependencies from a given dataset. This new approach not only can be used in our schema extraction algorithm. It is also applicable to any large relational database.

The algorithm is based on partitioning the set of tuples with respect to their attribute values. The use of partitions makes the validation of multivalued dependency simple and efficient. We propose several effective pruning methods based on the properties of multivalued dependency which greatly reduce the search space. Results show that our algorithm can correctly identify *minimal non-trivial* multivalued dependencies, providing useful dependency information to help us refining the schema prototypes. Moreover, the algorithm is also efficient for many existing benchmark databases, and has good scalability over the size of the dataset.

A new method for determining if a functional dependency holds or not was proposed by [29]. The method is based on representing attribute sets by equivalent class partitions of the set of tuples. We find that with suitable modification, the representation is also useful in discovering multivalued dependencies.

For reader's reference, here are some of the notations that we use in the following sections:

R	A given relation schema
r	A given relation over R
$t[A]$	The value of attribute A in tuple t
$[t]_X$	Equivalence class of tuple t with respect to $X \subseteq R$; a set of tuples whose X-values equals to the X-value of t
π_X	Partition of r under X under X
X_j	jth equivalence class in π_X
$\theta(Y, X_j)$	Number of different Y-value in equivalence class X_j

The relation in Table 4.1 is used for all the following examples concerning

Tuple ID	A	B	C	D
1	1	1	2	3
2	1	2	1	4
3	1	2	2	2
4	1	1	1	4
5	1	1	2	2
6	1	2	2	3
7	2	3	1	3
8	2	3	2	4

Table 4.1: An example relation

about validating multivalued dependencies with the use of partitions. For example, $\pi_{\{A\}} = \{\{1,2,3,4,5,6\},\{7,8\}\}$. The partitions for other attributes are $\pi_{\{B\}} = \{\{1,4,5\},\{2,3,6\},\{7,8\}\}$, $\pi_{\{C\}} = \{\{1,3,5,6,8\},\{2,4,7\}\}$ and $\pi_{\{D\}} = \{\{1,6,7\}, \{2,4,8\}, \{3,5\}\}$ respectively. The partition with respect to $\{CD\}$ is $\pi_{\{CD\}} = \{\{1,6\},\{2,4\},\{3,5\},\{7\}, \{8\}\}$.

4.1 Validation of Multivalued Dependencies

According to [29], a functional dependency $X \rightarrow Y$ holds if and only if π_X refines π_Y. Thus the concept of partition refinement gives almost direct functional dependencies. We find that we can also make use of the idea of partition refinement to find multivalued dependencies. With the concept of partition, each X-value actually forms an equivalence class. Thus we can see the validation of multivalued dependency in the way below:

Assume there is no duplicated tuples in the relation over the schema R with attribute sets X, Y, and Z (Z represents attributes in R other than X and Y, and $Z \neq \emptyset$). Arrange the equivalence classes in a partition by the smallest tuple ID in each equivalence class in ascending order. E.g. $\pi_{\{AB\}}$ is sorted as $\{\{1,4,5\}, \{2,3,6\}, \{7, 8\}\}$. We use X_j to represent the jth equivalence class in

π_X. E.g. for $\pi_{\{AB\}}$, $\{AB\}_1 = [t_1]_{\{AB\}} = \{1,4,5\}$, $\{AB\}_2 = [t_2]_{\{AB\}} = \{2,3,6\}$, while $\{AB\}_3 = [t_7]_{\{AB\}} = \{7,8\}$.

Lemma 1 Given a relation schema R with attributes X, Y and $Z = R - X - Y$. *The multivalued dependency* $X \rightarrow\rightarrow Y$ *holds in* R *iff for any valid relation* r, *for every equivalence class* X_j *in* π_X, *the number of different* Y*-values,* $\theta(Y, X_j)$, *times the number of different* Z*-values,* $\theta(Z, X_j)$, *equals to the size of* X_j, $\mid X_j \mid$, *i.e.*

$$\theta(Y, X_j) * \theta(Z, X_j) = \mid X_j \mid$$

Proof: First we assume $X \rightarrow\rightarrow Y$ holds (thus $X \rightarrow\rightarrow Z$ holds, by the complementation rule [9]). In the relation r, for a fixed value of X, say X_1, let the number of different Y-values be n and the number of different Z-values be m. We want to show that the number of tuples with the value of X in r equals $m * n$. This means that all combinations of the Y-values and Z values exist in the tuples having the given X-value.

Assume on the contrary that not all of these $n * m$ combinations of (Y-value, Z-value) pairs exist. Let (Y_1, Z_1) be the missing pair, i.e. the tuple (X_1, Y_1, Z_1) does not exist in the relation. Note that Y_1 exists with another Z value and X_1, Z_1 appears with another Y value and X_1. Then according to the definition of Multivalued Dependency, given two tuples $t_1 = (X_1, Y_1, Z_2)$ and $t_2 = (X_1, Y_2, Z_1)$ which exist in the relation, there should exist another two tuples t_3 and t_4 that fulfill the conditions stated in Section 2.5. $t_3[Y]$ should be equal to $t_1[Y] = Y_1$ and $t_3[Z]$ should be equal to $t_2[Z] = Z_1$. Thus t_3 should be (X_1, Y_1, Z_1), which is exactly the tuple that dose not exist in the relation, a contradiction. As a result, $X \rightarrow\rightarrow Y$ implies that all $n * m$ combinations of (Y-value, Z-value) pairs exist in the tuples for the fixed X-value.

Next consider the converse. Suppose that all the above mentioned $n * m$

combinations of (Y-value, Z-value) pairs exist. We want to show that $X \rightarrow\rightarrow Y$ holds (thus $X \rightarrow\rightarrow Z$ holds). That is, we want to show that the condition stated in Section 2.5 holds. Assume on the contrary that the condition does not hold. Then there exist 2 tuples $t_1 = (X_1, Y_1, Z_1)$, and $t_2 = (X_1, Y_2, Z_2)$, where X_1, Y_i, Z_i are values of X, Y, Z, respectively. And there does not exist 2 tuples $t_3 = (X_1, Y_1, Z_2)$ and $t_4 = (X_1, Y_2, Z_1)$. However, since all combinations of Y and Z values for given X_1 are found, t_3 and t_4 must exist, a contradiction. ∎

$\theta(Y, X_j)$ for each equivalence class X_j in π_X can be computed by comparing π_X and π_{XY}. *By checking how many equivalence classes in π_{XY} refine (are the subsets of) X_j*, we can obtain the $\theta(Y, X_j)$ for X_j. $\theta(Z, X_j)$ can be computed in the same way.

Example

Consider the relation schema R in Table 4.1 again. To test if $A \rightarrow\rightarrow B$ holds in R above, we need to compute $\pi_{\{A\}}$, $\pi_{\{AB\}}$ and $\pi_{\{ACD\}}$ (i.e. $\pi_{\{A \cup (R-B-A)\}}$).

With the computed $\pi_{\{A\}}$, $\pi_{\{B\}}$, $\pi_{\{C\}}$ and $\pi_{\{D\}}$, we can thus compute $\pi_{\{AB\}}$ and $\pi_{\{ACD\}}$. According to Table 4.1, $\pi_{\{AB\}} = \{\{1,4,5\}, \{2,3,6\}, \{7, 8\}\}$ and $\pi_{\{ACD\}} = \{\{1,6\}, \{2,4\}, \{3,5\}, \{7\}, \{8\}\}$. For the first equivalence class $\{A\}_1 = [t_1]_{\{A\}} = \{1,2,3,4,5,6\}$ in $\pi_{\{A\}}$, it is refined by equivalence classes $\{AB\}_1 = \{1,4,5\}$ and $\{AB\}_2 = \{2,3,6\}$ in $\pi_{\{AB\}}$. Thus we know that $\theta(\{B\}, \{A\}_1)$ is 2 because $\{A\}_1$ is refined by 2 equivalence classes in $\pi_{\{AB\}}$. For the second equivalence class $\{A\}_2 = [t_7]_{\{A\}} = \{7,8\}$ in $\pi_{\{A\}}$, it is refined by class $\{AB\}_2 = \{7,8\}$ in $\pi_{\{AB\}}$. Thus $\theta(\{B\}, \{A\}_2)$ is 1.

Similarly, $\{A\}_1$ is refined by equivalence classes $\{ACD\}_1 = \{1,6\}$, $\{ACD\}_2 = \{2,4\}$ and $\{ACD\}_3 = \{3,5\}$ in $\pi_{\{ACD\}}$. Thus we know that $\theta(\{CD\}, \{A\}_1)$ for $\{A\}_1$ is 3 because $\{A\}_1$ is refined by 3 equivalence classes in $\pi_{\{ACD\}}$. For $\{A\}_2$, it is refined by class $\{ACD\}_4 = \{7\}$ and $\{ACD\}_5 = \{8\}$. Thus $\theta(\{CD\}, \{A\}_2)$

is 2.

We can now check if the size of each equivalence class in $\pi_{\{A\}}$ equals to the product of its number of different $\{B\}$-values and its number of different $\{CD\}$-values, i.e. we check whether $\mid \{A\}_j \mid = \theta(\{B\}, \{A\}_j) * \theta(\{CD\}, \{A\}_j)$ for all $\{A\}_j$ in $\pi_{\{A\}}$. For $\{A\}_1$, $\theta(\{B\}, \{A\}_1)$ is 2 and $\theta(\{CD\}, \{A\}_1)$ is 3. $\mid \{A\}_1 \mid$ is 6. As $\mid \{A\}_1 \mid = \theta(\{B\}, \{A\}_1) * \theta(\{CD\}, \{A\}_1)$ $(6 = 2 * 3)$, $\{A\}_1$ can fulfill the requirement.

For $\{A\}_2$, $\theta(\{B\}, \{A\}_2)$ is 1 and $\theta(\{CD\}, \{A\}_2)$ is 2. $\mid \{A\}_2 \mid$ is 2. As $\mid \{A\}_1 \mid = \theta(\{B\}, \{A\}_1) * \theta(\{CD\}, \{A\}_1)$ $(2 = 1 * 2)$, $\{A\}_2$ can also fulfill the requirement.

Since all equivalence classes in $\pi_{\{A\}}$ ($\{A\}_1$ and $\{A\}_2$) can fulfill the requirement, we show that $A \rightarrow\rightarrow B$ holds in the relation.

4.2 Search Strategy and Pruning

The general search strategy for our algorithm is as follows. The search starts from singleton sets of right-hand side candidates for a multivalued dependency, and works its way to larger attribute sets of right-hand side candidates. For each right-hand side candidate, it is first validated with singleton sets of left-hand side candidates, and works its way to larger attribute sets of left-hand side candidates.

Similar to previous work in discovering functional dependencies, our focus is also to find only the *minimal non-trivial* multivalued dependencies by pruning the search space as much as possible. We find that there is some similarity between the properties of left-hand sides of functional dependency and those of multivalued dependency. Due to this similarity, we found that the small-to-large searching concept in previous functional dependency discovery algorithm can be

applied in discovering multivalued dependencies as well.

However, there is a crucial difference between the properties of left-hand sides of functional dependency and multivalued dependency which results in the increase of the search space for finding multivalued dependencies. We try to reduce the search space based on a unique property of multivalued dependency. The search strategies for both left-hand and right-hand sides candidates are discussed in the following sections.

4.2.1 Search Strategy for Left-hand Sides Candidates

For functional dependency, if $X \rightarrow Y$ holds then $XZ \rightarrow Y$ is also valid where Z represents some attributes in R other than X and Y. The dependency like $XZ \rightarrow Y$ above is not the minimal dependency found in the relation. Existing algorithms for finding functional dependencies tried to avoid considering those non-minimal left-hand side candidates, and we considered the possibility to do that in our algorithm for finding multivalued dependencies as well.

For multiple dependency if $X \rightarrow\rightarrow Y$ holds, $XZ \rightarrow\rightarrow Y$ would also be valid where Z represents some attributes in R other than X and Y. For a simple proof of this minimality rule, please refer to appendix. This minimality rule is a special case for *multivalued augmentation rule* [9]. The rule states that if $\alpha \rightarrow\rightarrow \beta$ holds and $\gamma \subseteq R$ and $\delta \subseteq \gamma$, then $\gamma\alpha \rightarrow\rightarrow \delta\beta$ holds. When substituting δ with $\emptyset$, we can get the rule we just stated.

With this property, we can prune the search space of left-hand side candidate as follows.

Rule 1 *Once we find a valid multivalued dependency $X \rightarrow\rightarrow Y$, we do not have to further validate any left-hand side which is a superset of X for a candidate multivalued dependency with Y as the right-hand side.*

4.2.2 Search Strategy for Right-hand Sides Candidates

For functional dependency, given $X \rightarrow A$ and $X \rightarrow B$ are valid dependencies, we can be sure that $X \rightarrow AB$ holds. Given $X \rightarrow AB$, we can be sure that $X \rightarrow A$ and $X \rightarrow B$ hold as well. The dependency like $X \rightarrow AB$ above is also called a non-minimal dependency, since obviously it can be easily deduced from $X \rightarrow A$ and $X \rightarrow B$. Existing algorithms for finding functional dependencies tried to avoid those non-minimal candidates by just considering the singleton sets as the possible right-hand side candidates.

Unfortunately, the above property for functional dependency is not always true for multivalued dependency, i.e. if $X \twoheadrightarrow AB$ holds, it may not be true that $X \twoheadrightarrow A$ and $X \twoheadrightarrow B$ hold as well. For instance, in the relation of Table 4.1, we can see that $A \twoheadrightarrow CD$ holds but neither $A \twoheadrightarrow C$ nor $A \twoheadrightarrow D$ holds. In this case, $A \twoheadrightarrow CD$ is actually a minimal non-trivial multivalued dependency. As a result, other than singleton sets, we have to also consider larger attribute sets as the possible right-hand side candidates.

Let us go back to the idea about validating multivalued dependencies introduced in Section 4.1. When we consider a possible dependency $X \twoheadrightarrow Y$, our validation must involve Z (Z equals $R - Y - X$). It is obvious that in order to have a valid *non-trivial* multivalued dependency $X \twoheadrightarrow Y$, all X, Y and Z cannot be $\emptyset$. For R with k attributes, the right-hand side candidates Y would be the largest when X and Z are singleton sets. Thus the largest Y would be the $(k-2)$-attribute sets, and the search space is expected to contain from singleton sets to all sets with $k-2$ attributes. Using Figure 4.1 as an example, where *level* refers to the number of attributes in an attribute set, we have to consider all the sets in level 1 and 2 (indicated by larger fonts) as right-hand side candidates only.

However, by applying the ***complementation rule*** [9] of multivalued depen-

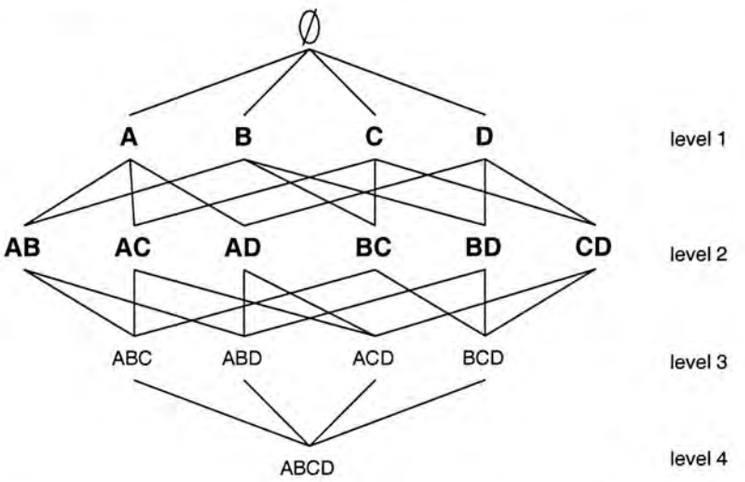

Figure 4.1: A set containment lattice for $R = \{A, B, C, D\}$

dency, we can actually reduce the search space to less than half. The complementation rule states that if $X \rightarrow\rightarrow Y$ holds, then $X \rightarrow\rightarrow Z$ (Z equals $R - Y - X$) also holds. From this rule, it is obvious that $X \rightarrow\rightarrow Z$ holds iff $X \rightarrow\rightarrow Y$ holds. Thus while we are validating $X \rightarrow\rightarrow Y$, we are actually validating $X \rightarrow\rightarrow Z$ at the same time. The size of Z is the largest when that of X is the smallest, i.e. when X is a singleton set. As a result, we do not have to consider any attribute set as a possible Y (right-hand side candidate) if that attribute set has already acted as a Z before. We apply the above idea to our search strategy as follows.

Rule 2 *A multivalued dependency candidate $X \rightarrow\rightarrow Y$ will not be validated if the level (size) of Y is greater than that of $Z = R - X - Y$.*

Using Figure 4.1 as an example where the number of attributes $k = 4$. First we consider a singleton set $\{A\}$, which is at level 1, as the Y-candidate. We consider the $X = \{B\}$. As a result, the corresponding Z is $\{CD\}$, which is in level 2. Since the level of the Y-candidate, $\{A\}$, is smaller than that of the corresponding Z, $\{CD\}$, the validation of the dependency $B \rightarrow\rightarrow A$ is performed. By the complementation rule, it is clear that we have also validated the dependency $B \rightarrow\rightarrow CD$ at the same time. Later the validation will come to take $\{CD\}$ as the Y-candidate, $\{B\}$ as X and $\{A\}$ as the corresponding Z. Since

the level of the Y-candidate, $\{CD\}$, is larger than that of the corresponding Z, $\{B\}$, the validation of the dependency $B \rightarrow\rightarrow CD$ will not be performed.

For the case of having candidate Y and the corresponding Z in the same level, we apply a similar strategy to avoid redundant validation.

Rule 3 *Assume a total order on the attributes (e.g. alphabetical order). Consider a multivalued dependency candidate $X \rightarrow\rightarrow Y$ where Y at the same level (has the same size) as $Z = R - X - Y$. The dependency candidate will be validated iff the smallest attribute (in alphabetical order) of Y is smaller than that of Z.*

Using Figure 4.1 as an example again. We consider a singleton set $\{A\}$, which is in level 1, as the Y-candidate. And we consider the X as $\{BC\}$. Thus the corresponding Z is $\{D\}$, which is in level 1 as well. Both $\{A\}$ and $\{D\}$ are in level 1. The smallest attribute in $\{A\}$ is A (since $\{A\}$ is a singleton set, the smallest attribute must be A) while that in $\{D\}$ is D. Since A is smaller than D in alphabetical order, the validation of the dependency $BC \rightarrow\rightarrow A$ is performed. By the complementation rule, it is clear that we have also validated the dependency $BC \rightarrow\rightarrow D$ at the same time. Later the validation come to taking $\{D\}$ as Y-candidate, $\{BC\}$ as X and $\{A\}$ as the corresponding Z. Since D is larger than A in alphabetical order, the validation of the dependency $BC \rightarrow\rightarrow D$ will not be performed.

By applying the above strategy, we can greatly reduce the number of candidates while still ensuring a complete search space. In other words, if R has k attributes, we can reduce the necessary right-hand side candidates from $k-2$ levels to only $\lceil (k-2)/2 \rceil$ levels. This is because from level $\lceil (k-2)/2 \rceil + 1$ to $(k-2)$, all the sets in them must have been a Z candidate for Y-candidate sets in level 1 to level $\lceil (k-2)/2 \rceil$, hence will not be validated according to our search rules. Using Figure 4.1 as an example again, $k = 4$. The necessary right-hand

side candidate(s) is/are from level 1 to level $\lceil (4-2)/2 \rceil = 1$ thus the only level we have to consider is level 1. The level(s) that we do no have to consider here should be from level $\lceil (4-2)/2 \rceil + 1 = 2$ to level $4 - 2 = 2$ so it is just level 2.

4.2.3 Other Pruning

Apart from the search space pruning proposed above, we also propose several other pruning rules based on the relation between multivalued dependency with key and functional dependency, as well as the characteristic of multivalued dependency in terms of partitions.

Left-hand Side Key pruning

It is well-known that a functional dependency holds whenever the left-hand side of the dependency is a key. By the ***replication rule*** [9] which states that if $X \rightarrow Y$ holds then $X \rightarrow\rightarrow Y$ also holds, obviously multivalued dependency holds under the same condition. With the above properties, we can produce the multivalued dependencies involving key or the superset of the key at the left-hand side without performing multivalued dependency validation at all. As mentioned in [29], an attribute set X is a key or a superset of a key if partition π_X consists of singleton equivalence classes only. The identification for keys is very simple and straightforward. As a result, we have the rule below:

Rule 4 *During validation, if the left-hand side candidate is identified as a key or superset of a key, we do not have to further verify it with our multivalued dependency validation.*

Functional Dependency Pruning

As introduced in [29], with the use of partition refinement concept, validating a functional dependency $X \rightarrow Y$ is simply checking if $\mid \pi_X \mid = \mid \pi_{X \cup Y} \mid$ or not. While validating a functional dependency involves single comparison, the validation for a multivalued dependency is more complicated as it includes comparisons between the sizes of equivalence classes for all π_X, $\pi_{X \cup Y}$ and $\pi_{X \cup Z}$. As a result, for each multivalued dependency candidate $X \twoheadrightarrow Y$, we first perform the simple validation for $X \rightarrow Y$ and see if it holds. By the ***replication rule*** if the functional dependency holds then the corresponding multivalued dependency must hold as well. In this case, we do not have to further perform the more complicated multivalued dependency validation process.

With *Rule 2* and *Rule 3*, given a relation of schema R we avoid validating any $X \twoheadrightarrow Z$ if $X \twoheadrightarrow Y$ has been validated before where $Z = R - Y - X$. However, in functional dependency pruning we should also consider if $X \rightarrow Z$ is identified as a functional dependency or not. Consider that for a candidate $X \twoheadrightarrow Y$, it's corresponding functional dependency $X \rightarrow Y$ does not hold while $X \rightarrow Z$ holds. By *replication rule*, $X \twoheadrightarrow Z$ should hold when $X \rightarrow Z$ holds. And by *complementation rule*, when $X \twoheadrightarrow Z$ holds, $X \twoheadrightarrow Y$ should hold as well. Thus in such a case, we still can avoid further verifying $X \twoheadrightarrow Y$ even $X \rightarrow Y$ is not a functional dependency.

Rule 5 *Given a relation of schema R and a multivalued dependency candidate $X \twoheadrightarrow Y$ and $Z = R - Y - X$. if either $X \rightarrow Y$ or $X \rightarrow Z$ is identified as a functional dependency, we do not have to further verify candidate $X \twoheadrightarrow Y$ with our multivalued dependency validation.*

Right-hand Side Key pruning

We discovered that when a dependency candidate has a key or a superset of a key as its right-hand side, it possesses a certain kind of property as well. We can use the property to increase the possibility of pruning more search space during validation.

Lemma 2 *Given a relation of schema R with attribute sets X, Y, and Z ($Z = R - Y - X$). When Y is a key, if π_{XZ} is not equal to π_X, then multivalued dependency $X \rightarrow\rightarrow Y$ cannot be valid.*

Proof: If Y is a key, the value of Y is unique for each tuple in R. In other words, for every equivalence class X_i in π_X, the number of different Y-values, $\theta(Y, X_i)$, must be the same as $\mid X_i \mid$. In this case, in order to fulfill the equality $\theta(Y, X_i) * \theta(Z, X_i) = \mid X_i \mid$, $\theta(Z, X_i)$ have to be 1 for each equivalence class X_i in π_X, i.e. π_{XZ} have to be equal to π_X. ∎

By applying the above property. we can save the validation process for certain candidates:

Rule 6 *Given a multivalued dependency candidate $X \rightarrow\rightarrow Y$ and $Z = R - Y - X$. When Y is a key or superset of a key, if the number of equivalence classes of π_X is not equal to that of π_{XZ}, i.e. $\mid \pi_X \mid \neq \mid \pi_{XZ} \mid$, then we do not have to perform dependency validation.*

4.3 Computing with Partitions

In [29], in order to reduce the time and space requirement of working with partitions, several techniques are introduced. We find that we can apply some of the

techniques when dealing with the partitions in our proposed work as well.

Similar to [29], we use 'stripped partitions' to replace the original partitions. A *stripped partition* is a partition with equivalence classes of size one removed. We can see stripped partitions as a more compact representation for the original partitions. For example, in Table 4.1 $\pi_{\{CD\}}$ is $\{\{1,6\},\{2,4\},\{3,5\},\{7\},\{8\}\}$. The stripped version for $\pi_{\{CD\}}$ is only $\{\{1,6\},\{2,4\},\{3,5\}\}$.

In [29], full partitions are still needed for computation of next level partitions as well as validating functional dependencies. However, in our algorithm we even apply stripped partitions for validating multivalued dependencies since our validation can be carried out without examining singleton equivalence classes.

As stated in Section 4.1 before, to check if a multivalued dependency $X \rightarrow\rightarrow Y$ holds, we need π_X, π_{XY} and π_{XZ}, and we have to obtain $\theta(Y, X_i)$ and $\theta(Z, X_i)$ for each X_i. Using $\theta(Y, X_i)$ as an example, if we use full versions of π_X and π_{XY}, we can obtain $\theta(Y, X_i)$ by checking how many equivalence classes in π_{XY} refine X_i. However, if we use stripped version of π_X and π_{XY}, the computation will become counting *the number of equivalence classes in* π_{XY} *which refine* X_i *+ the number of the remaining tuples in* X_i *that are not referred by any equivalence class in* π_{XY}. Since the Y-value and Z-value for a singleton equivalence class X_j must be 1, it must fulfill the equality $\theta(Y, X_j) * \theta(Z, X_j) =\mid X_j \mid (1 * 1 = 1)$. As a result, it is okay for us to just perform checking on stripped partition and omit all the singleton equivalence classes without affecting the results.

4.3.1 Computing Partitions

In [29], only partitions for the singleton attribute sets are computed by scanning the database. For partitions in higher levels, they are computed as a product of two previously computed partitions in lower levels. We used exactly the same method introduced by [29] to compute the partitions so that for all

$X, Y \subseteq R, \pi_X \cdot \pi_Y = \pi_{X \cup Y}$ starting from the second level partitions. Using Figure 4.1, $\pi_{\{C\}} = \{\{1,3,5,6,8\},\{2,4,7\}\}$ and $\pi_{\{D\}} = \{\{1,6,7\}, \{2,4,8\}, \{3,5\}\}$ respectively. $\pi_{\{CD\}}$ can be computed just from $\pi_{\{C\}}$ and $\pi_{\{D\}}$. Only tuples that are in the same equivalence class in both $\pi_{\{C\}}$ and $\pi_{\{D\}}$ forms a new equivalence class in $\pi_{\{CD\}}$ together. Other tuples forms singleton classes only. So the partition with respect to $\{CD\}$ is $\pi_{\{CD\}} = \{\{1,6\},\{2,4\},\{3,5\},\{7\},\{8\}\}$.

4.4 Algorithm

We try to find all minimal non-trivial multivalued dependencies level by level. For each candidate Y in the level, we try to find all valid minimal left-hand side candidates X so that $X \rightarrow\rightarrow Y$ holds. To ensure that we find only the minimal non-trivial candidates, we adopted the search strategies we described in Section 4.2 to avoid generating duplicated next-level candidates for both left-hand side candidates X and the right-hand side candidate Y.

The main algorithm for generating multivalued dependencies of the form $X \rightarrow\rightarrow Y$ is given in Figure 4.2. In the algorithm, $RHS(l)$ stores all right-hand side candidates for level l. Using Figure 4.1 as example, $RHS(1)$ for $R = \{A, B, C, D\}$ stores four right-hand side candidates $\{A\}, \{B\}, \{C\}$ and $\{D\}$. For a right-hand side candidate (Y-candidate) under validation, $LHS(k)$ stores all left-hand side candidates (X-candidates)to be verified where k is the level for right-hand side candidates. $LHSFAIL(k)$ stores all candidates that failed the validation. $LHSMVD(k)$ stores all successful candidates.

As shown in Figure 4.2, the input of the algorithm is all the tuples in r over the relation schema R. Step 5 to Step 23 are repeated for each attribute $Y \in RHS(l)$, and all valid dependencies for Y are recorded. In the algorithm, any left-hand side candidates to be verified are stored in $LHS(k)$. Failed candidates

Algorithm MVD_DISCOVERY(r, R)

```
1  N = number of attributes in R
2  l = 1
3  RHS(l) = {{Y} | Y ∈ R}
4  while l ≤ ⌈(N − 2)/2⌉
5      for each attribute Y ∈ RHS(l)
6          k = 1
7          put all X ∈ R/Y into LHS(k)
8          while LHS(k) is not empty
9              for each X in LHS(k)
10                 if VERIFY_MVD(X, Y) == FALSE        % X →→ Y dose
                   not hold
11                     enter X into LHSFAIL(k)         % Rule 1
12                 else                                % X →→ Y
                   holds
13                     output X →→ Y
14                     enter X into LHSMVD(k)
15             end for
16             Empty LHS(k)
17             if LHSFAIL(k) contains more than one element
18                 LHS(k + 1) = GENERATE_NEXT_LEVEL_L(k, LHSFAIL(k) )
19             Empty LHSFAIL(k)
20             k = k + 1
21         end while
22         Empty LHSMVD(k)
23     end for
24     RHS(l + 1) = GENERATE_NEXT_LEVEL_R(l, RHS(l))
25     l = l + 1
26 end while
```

Figure 4.2: Algorithm for discovering multivalued dependencies

in the current level are put into $LHSFAIL(k)$ for generating candidates for next level. The validation for a left-hand side candidate Y is completed when no more candidate is found in $LHS(k)$. The process is repeated for candidate set $RHS(l)$ of each level l until l reaches $(N-2)/2$ where N is the number of attributes in R. The reason for stopping at no more than level $(N-2)/2$ is stated in Section 4.2.2 before. Just like [29], we have implemented the attributes as bit vectors of words. All the left-hand and right-hand side candidates are represented as 32-bit bit vectors in our algorithm. All $LHSFAIL(k)$, $LHS(k)$,

$LHSMVD(k)$ and $RHS(l)$ are arrays storing the bit vectors representation of the candidates.

Figure 4.3 is the procedure for verifying if a multivalued dependency holds or not.

```
Procedure VERIFY_MVD(X, Y)
1  Z = R − X − Y
2  pruned = PRUNE(X, Y, Z)
3  if pruned ≠ CONTINUE
4      return pruned
5  Get π_XY
6  Get π_XZ
7  for each X_i ∈ π_X
8      Compute θ(Y, X_i) using π_XY
9      Compute θ(Z, X_i) using π_XZ
10     if | X_i | ≠ θ(Y, X_i) * θ(Z, X_i)        %Lemma 1
11         return FALASE
12 end for
13 return TRUE
```

Figure 4.3: Procedure for verifying a multivalued dependency

Procedure VERIFY_MVD takes in attribute set X and attribute Y and check if $X \rightarrow\rightarrow Y$ holds as shown in Figure 4.3. The details for computing each values in this procedure can be found in Section 4.1 and Section 4.1. Note that the validation would only be performed if the candidate can pass the procedure PRUNE.

The pruning procedure for our algorithm is given in Figure 4.4. In the procedure PRUNE, we implemented four pruning rules described in the previous sections: *Rule 2* and *Rule 3* for search space pruning described in Section 4.2.2, and *Rule 4 to 6* described in Section 4.2.3. We order the rules in this procedure according to the cost of computation involved in them in ascending order. In Figure 4.4, $Y > Z$ if either $\mid Y \mid > \mid Z \mid$ or the smallest attribute in Y is greater than the smallest attribute in Z.

Procedure PRUNE(X, Y, Z)
1 **if** $Y > Z$ %*Rule* 2, *Rule* 3
2 **return FALSE**
3 **if** X is a (super)key %*Rule* 4
4 **return TRUE**
5 **if** $(\pi_X == \pi_{XY})$ %*Rule* 5
6 **return TRUE**
7 **if** $(\pi_X == \pi_{XZ})$ %*Rule* 5
8 **return TRUE**
9 **if** Y is a (super)key and $\pi_X \neq \pi_{XZ}$ %*Rule* 6
10 **return FALSE**
11 **return CONTINUE**

Figure 4.4: Procedure for pruning a candidate set

4.4.1 Generating Next Level Candidates

The procedure of generating next level candidates is used for generating both next level left-hand side candidates (as in Step 18), as well as right-hand side candidates (as in Step 24). Figure 4.5 shows GENERATE_NEXT_LEVEL_L, which is the procedure for left-hand side candidates.

Procedure GENERATE_NEXT_LEVEL_L$(k, INQUEUE)$
12 **if** $k = 1$
2 **for each** pair $\{U, V\} \in INQUEUE$ where $U < V$
3 $W = U \cup V$
4 put W into $OUTQUEUE$
5 **end for**
6 **else**
7 **for each** $K \in$ PREFIX_BLOCK$(INQUEUE)$
8 **for each** pair $\{U, V\} \in K$ where $U < V$
9 $W = U \cup V$
10 **if** $W \not\supset E$ where $E \in LHSMVD(k)$ %*Rule* 1
11 put W into $OUTQUEUE$
12 **end for**
13 **end for**
14 **return** $OUTQUEUE$

Figure 4.5: Procedure for generating next level candidates for a set of input candidates

For generating left-hand side candidates, this procedure takes in the failed candidates of level k, $LHSFAIL(k)$, and use them to generate candidates for level $k+1$, $LHS(k+1)$. The candidates of level $k+1$ are the supersets of size $k+1$ of the failed candidates in level k. The procedure PREFIX_BLOCK partitions candidates in $LHSFAIL(k)$ into blocks so that the level k candidates in each block have same prefix of length $k-1$. To avoid generating duplicated $k+1$ candidates, only k candidates in the same prefix blocks are used to produce level $k+1$ candidates. By doing so we can ensure it generates a minimal but complete search space. Detailed information about PREFIX_BLOCK can be found in [7] and [41]. Consider the relation schema R in Table 4.1 again, the example search tree generated for finding valid left-hand side candidates for attribute A is shown in Figure 4.6.

For generating right-hand side candidates, it takes in the whole candidate sets, $RHS(l)$, in level l and use them to generate candidate set $RHS(l+1)$. The procedure is triggered in Figure 4.2 Line 24, and the name of the procedure is GENERATE_NEXT_LEVEL_R. This procedure is used for generating right-hand side candidates. GENERATE_NEXT_LEVEL_R is the same as GENERATE_NEXT_LEVEL_L (Figure 4.5) except that it neglects Line 10 in Figure 4.5.

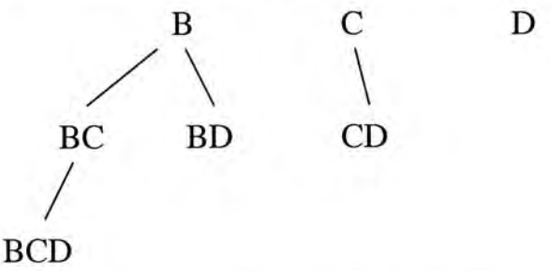

Figure 4.6: Search tree for attribute A in relation $R = \{A, B, C, D\}$

4.4.2 Computing Partitions

We did not have detailed description on how we generate the partitions in our algorithm in the sections above. We talk about it in this section.

In [29], validation of a functional dependency only involves two levels of partition candidates. However, in a multivalued dependency validation, we have to use partitions from different levels. E.g. for $R = \{A, B, C, D, E, F, G\}$, validating if $\{A\} \rightarrow\rightarrow \{B\}$ would require $\pi_{\{A\}}$ (level 1 partition), $\pi_{\{ABC\}}$ (level 3 partition) as well as $\pi_{\{ADEFG\}}$ (level 5 partition). As a result, unlike the strategy used in [29], which is computing new partitions during dependency validation, our algorithm computes all the partitions before validation.

The partitions in our algorithm are computed in a similar way to that in [29]: partitions for singleton attribute sets are computed from the relation, then the singleton equivalence classes are stripped off and forming the stripped version for those first-level partitions. For easy validation, we follow [29] to replace all original values in the database with integers while keeping the original equivalence relations in the database, i.e. same values are replaced by same integer values. We use a hash table to map the original data values to integers in incremental fashion (starting from 1). For partitions in higher levels, they are computed as a product of two previously computed partitions in lower levels. Thus starting from second-level candidate, we do no have to scan through the actual database anymore. For more detailed information, please refer to [29].

4.5 Experimental Results

We run experiments with our algorithm for discovering multivalued dependencies. We implemented our algorithm in C language and compiled our programs with GCC compiler under Unix environment. We implemented two versions for our algorithm: One works completely in main memory while the other stores all partitions on disk. With the two implementations, we carried out our experiments on Sun Ultra 5 workstations with 704 MB main memory. To illustrate the difference between memory and disk based approaches, we also used a Sun

Sparc 20 workstation with 128 MB main memory.

We tried our algorithm on a number of real life databases. The databases we used are available on the UCI Machine Learning Repository [11]. The datasets and the corresponding descriptions can be found in the url stated in [11]. We did not make any changes to any of the dataset except that we removed all duplicated tuples in the datasets. Table 4.2 shows the results of our algorithm on the example in Table 4.1 and 15 real life databases. Table 4.3 shows the pruning results of our algorithm on those benchmark databases.

4.5.1 Results of the Algorithm

The name of the database, the number of tuples in the database ($| r |$), the number of attributes of the database ($| R |$) are shown in the first 3 columns in Table 4.2. In the fourth column, we have the number of *pure* multivalued dependencies found. Here a *pure* multivalued dependency means that the found multivalued dependency is minimal nontrivial and not derived from functional dependencies or keys at all.

For Table 4.3, in the first column we have the number of multivalued dependency candidates that we have to verified, and the second column shows the number of candidates that we really have to verify by our multivalued dependency validation after pruning. Note that the second column actually indicates the number of candidates we have to consider after applying pruning rule *Rule 1*. From the third column to the last column we have the number of multivalued dependency candidates that is pruned by our proposed pruning rules *Rule 2* to *Rule 6*.

The first row of Table 4.2 and Table 4.3 shows the results for the example used in Table 4.1 used in this paper. The following 15 rows shows the results for all the real life databases picked from [11].

Databse Name	$\mid r \mid$	$\mid R \mid$	*Pure MVD*
example	8	4	2
servo	167	5	2
hayes-roth	132	6	0
shuttle-landing	15	7	22
bupa_data	341	7	0
post-operative	80	9	14
pima-indians-diabetes	768	9	0
yeast	1462	10	0
breast-cancer -wisconsin	691	11	84
glass	214	11	0
bridges	108	13	0
flare1	187	13	34
flare2	365	13	106
echodiogram	132	13	0
wine	178	14	0
housing	506	14	0

Table 4.2: Results of our algorithm on benchmark databases

4.5.2 Evaluation on the Results

In our experiment, we selected 15 real life databases, which are all from [11]. From the results shown in Table 4.2, among those 15 databases we used in our experiments, 6 of them consist of *pure* multivalued dependencies. The results indicates that multivalued dependency do exist in a portion of the real life databases, and our algorithm should be able to provide useful multivalued dependency information.

For each benchmark database, the number of *pure* multivalued dependencies found by our algorithm is actually exactly half of the values shown in Table 4.2. It is because that *Rule 2* and *3* prevent our algorithm from validating multivalued dependencies which's Y-candidates are Z-candidates before. However, when the multivalued dependency $X \rightarrow\rightarrow Y$ holds it's counterpart $X \rightarrow\rightarrow Z$ must hold as well. Thus we double the values of MVD found to obtain '*Pure* MVD'.

Database Name	MVD candidates	MVD-validated	Pruned by			
			Rule 2 & 3	Rule 4	Rule 5	by Rule 6
example	22	12	8	0	2	0
servo	144	75	69	0	0	0
hayes-roth	266	100	71	15	0	80
shuttle -landing	1464	789	650	0	25	0
bupa_data	1515	810	666	27	9	3
post-operative	16056	9075	6981	0	0	0
pima-indians-diabetes	13566	6539	5996	553	36	442
yeast	28646	10294	10018	255	65	8014
breast-cancer-wisconsin	147609	82056	65352	6	183	12
glass	73323	20076	34600	511	2674	15462
bridges	809209	221460	372061	4434	536	210718
flare1	1416667	784628	632028	0	11	0
flare2	1412446	780540	631893	0	13	0
echodiogram	874058	275570	426972	26763	550	144203
wine	892293	13113	469544	153873	45	255718
housing	1484093	248471	635597	41375	1658	556992

Table 4.3: Pruning results of our algorithm on benchmark databases

From Table 4.3, we observe that *Rules 2* and *3* have significant effects on most of the databases in our experiments. On the other hand, the effects of pruning *Rules 4* to *6* depend much on the characteristics of the databases. When there is key or superkey in the relation *Rule 4* and *6* (especially *Rule 6*), provide good pruning effects. In some of the benchmark databases, *Rule 6* provides pruning effects comparable to that from *Rule 2* and *3*. Overall speaking, our pruning rules have satisfactory effects on most of the databases in our experiments.

4.5.3 Scalability of the Algorithm

We evaluate the performances of our algorithm by the real time elapsed, instead of CPU time. The time is recorded by a Perl script. The reason for recording real time instead of CPU time is to have a clearer illustration on the cost of memory usage and the cost of I/O processing.

From the results in Table 4.2, it is obvious that the the search space increases enormously when the number of attribute increases. In fact, just like functional dependency, the search space for multivalued dependency is exponential in the number of attribute [37, 38]. Thus, similar to all algorithms for finding functional dependencies do, our algorithm for finding multivalued dependencies has exponential time and space scalabilities in the number of attributes. However, typically the number of attribute of a real life database is not really large, and our algorithm performs well. In our experiment, results for all 15 databases are obtained in a few minutes at most. For some of the smaller databases which have fewer attributes, the required time is just a few seconds.

Since the complexity for our algorithm is inevitably exponential in the number of attribute, we are interested in evaluating how the number of tuples affects the performance of our algorithm. To do that, we use a real life database, Wisconsin breast cancer database, obtained from UCI Machine Learning Repository [11]. Result of this database can be found in its corresponding rows in Table 4.2. Originally the data set consists of 699 tuples but we removed 8 duplicated tuples. To see how our algorithm scales over the number of tuples, we adopt the method used in [29] to enlarge the database for experimenting scalability. We duplicate the breast cancer data set multiple times and then merge them together to provide larger datasets. During data duplication, a new different set of attribute values is used so that we increase the database size but keep the same multivalued dependencies. We first run the experiment using memory version of

our algorithm. The result is illustrated in Figure 4.7.

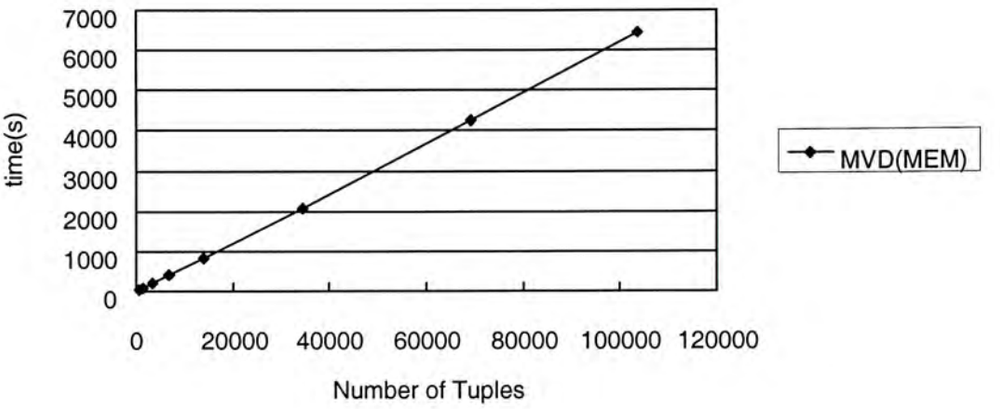

Figure 4.7: Scalability of our algorithm over large dataset

Our algorithm performs linearly in the number of tuples. The disk version of our algorithm performs also linearly except that the time required for disk version is longer than that for memory version. The difference between memory version and disk version is shown in Figure 4.8. MVD(MEM) represents the memory version while MVD(DISK) represents the disk version. The extra time required by disk version is caused by the disk access time for writing and reading partition files.

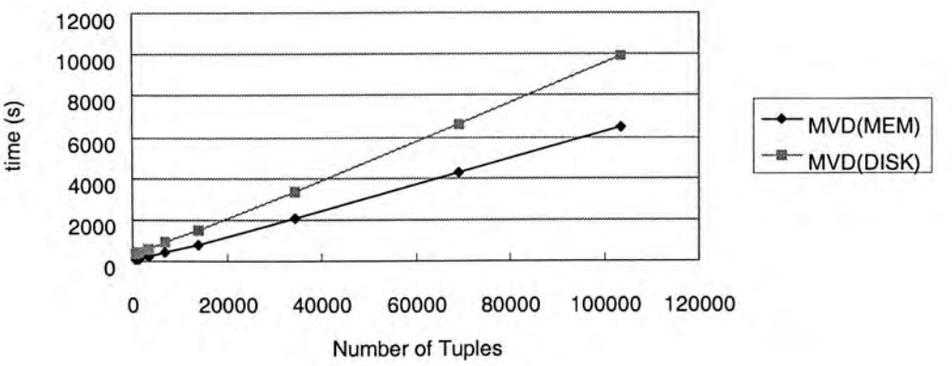

Figure 4.8: Scalability of our algorithm

To evaluate the situation when we run out of main memory while handling a very large datasets, we run both versions of our algorithm on a Sun Sparc 20

workstation which have poorer hardware specification when compared with a Sun Ultra Sparc workstation, including the size of memory. The Sparc 20 workstation we use for experiment has 128 MB main memory. The result is illustrated in Figure 4.9.

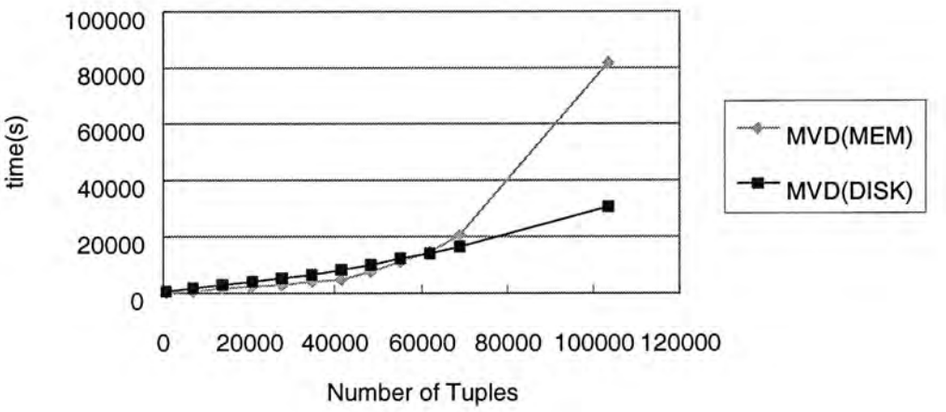

Figure 4.9: Scalability of our algorithm: with limited main memory

The performance of the memory version drops significantly when memory is running out. When the main memory runs out, the machine starts to use the reserved swap memory together with main memory. When the size of swap space increases, the performance for the main memory version drops. Memory swapping causes a sharp increase in running time, which is shown in the curve for the memory version. On the other hand, the disk version of our algorithm still performs nearly linearly as it relies much less on main memory while requires more temporary disk space. The memory / disk space usage for both versions of our algorithm is shown in Figure 4.10. The memory / disk space usage for our algorithm also has nearly linear scalability.

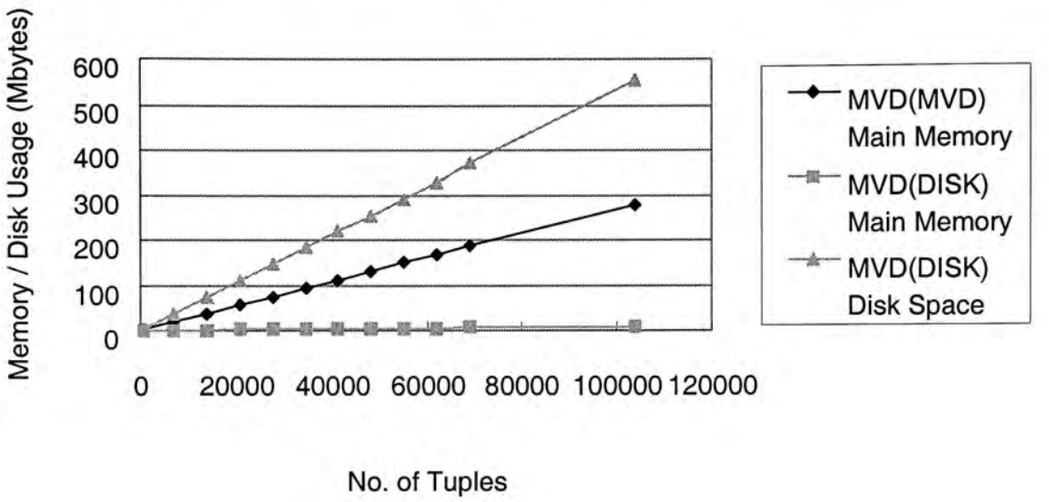

Figure 4.10: Memory and disk space usage of our algorithm

4.5.4 Using Multivalued Dependencies in Schema Extraction Algorithms

As we discussed in the previous chapter, we would like to introduce multivalued dependency discovery into our schema extraction algorithms to refine the relational schema of the XML data. As a result, the global scheme for our schema extraction algorithm should become:

(1) Simplify DTD
(2) Construct schema prototype trees
(3) Generate relational schema prototypes
(4) Detect possible **functional and multivalued dependencies** and candidate keys
(5) Normalize the relational schema prototypes

To illustrate how multivalued dependencies inside the XML data can help refining the relational schema, we repeat the experiment on the *SIGMOD Record* XML dataset and the synthetic XML dataset as we do in Chapter 3 Section 3.3. This time we apply our multivalued dependency discovery algorithm in Step 4

of our schema extraction algorithm as well.

Real Life XML Data: SIGMOD Record XML

In our previous experiment of using *DTD-splitting* Schema Extraction Algorithm on *SIGMOD Record* XML dataset, the schema prototype produced is relatively small in size (only 2 to 3 attributes per table). The need for further decomposition is rather small. As a result, we use schema prototype produced in Global Schema Extraction Algorithm to illustrate the effect to show that how we can improve the relational schema generated by our algorithms with the addition of multivalued dependency discovery.

After applying our multivalued dependency discovery algorithm, the result for the schema prototype of `sigmodrecord.xml` is shown in Table 4.4. The multivalued dependencies found are shown in Figure 4.11.

		Pure	MVD	MVD-	pruned by			
$\mid r \mid$	$\mid R \mid$	MVD	Candidates	validated	Rule 2 & 3	Rule 4	Rule 5	Rule 6
3113	7	8	1495	796	664	9	25	1

Table 4.4: Results of multivalued discovery algorithm on SIGMOD Record XML data (in Global Extaction Algorithm)

With the multivalued dependencies, we can further normalize the relational schema produced in Figure 3.28 using 4NF decomposition [23]. 4NF decomposition algorithm is presented in appendix for readers' reference. The relational schema produced is shown in Figure 4.12.

Note that Figure 4.12 is just one of the possible designs for the relational schema. With the functional and multivalued dependencies found in the XML data, the user can decompose the schema prototypes into other good relational database designs for the XML data.

```
Table1
No. of tuples:              3133
No. of attributes:          7
MVDs found:

1 3 5 ->-> 2                1 3 5 ->-> 2 4
1 3 5 ->-> 4 6 7            1 3 5 ->-> 6 7
3 5 ->-> 1 2                3 5 ->-> 6 7
3 5 ->-> 4 6 7              3 5 ->-> 1 2 4

=============================================
```

Figure 4.11: Multivalued dependencies found from the prototype table in Figure 3.26

Table Name	$\|r\|$	$\|R\|$	*Pure* MVD	MVD Candidates	MVD-validated	pruned by			
						Rule 2 & 3	Rule 4	Rule 5	Rule 6
book	507	8	0	3074	1924	1051	0	88	11
monograph	487	7	2	938	397	329	14	54	43

Table 4.5: Results of multivalued discovery algorithm on synthetic XML data (in Global Extaction Algorithm)

Synthetic XML Data

The result of applying multivalued dependency discovery algorithm on the tables produced by the *Global* Schema Extraction Algorithm is shown in Table 4.5, and the multivalued dependencies found are shown in Figure 4.13. However when applying 4NF decomposition on the schema prototypes, the multivalued dependencies found do not result in a more refined relatoinal schema in this experiment.

For applying multivalued dependency discovery algorithm on the tables produced by the *DTD-splitting* Schema Extraction Algorithm, since the schema prototype produced is relatively small in size (only 2 to 4 attributes per table). No multivalued dependencies are found in the mapped data for the schema pro-

```
table1-1{
SigmodRecord.issue.articles.article.title, (3)
SigmodRecord.issue.articles.article.initPage, (4)
SigmodRecord.issue.articles.article.endPage (5)
}

table1-2{
SigmodRecord.issue.articles.article.title, (3)
SigmodRecord.issue.articles.article.endPage (5)
SigmodRecord.issue.volume, (6)
SigmodRecord.issue.number (7)
}

table2{
SigmodRecord.issue.articles.article.authors.author, (2)
SigmodRecord.issue.articles.article.title, (3)
SigmodRecord.issue.number (7)
}

table3{
SigmodRecord.issue.articles.article.authors.author.position, (1)
SigmodRecord.issue.articles.article.authors.author, (2)
SigmodRecord.issue.articles.article.initPage, (4)
SigmodRecord.issue.articles.article.endPage (5)
SigmodRecord.issue.number (7)
}

table4{
SigmodRecord.issue.articles.article.authors.author.position, (1)
SigmodRecord.issue.articles.article.authors.author, (2)
SigmodRecord.issue.articles.article.title, (3)
SigmodRecord.issue.articles.article.initPage, (4)
SigmodRecord.issue.volume, (6)
}
```

Figure 4.12: Relational schemas produced for `sigmodrecord.xml` based on 4NF decomposition

```
table:book
No. of tuples:              507
No. of attributes:   8
MVDs found:


==========================================
table:monograph
No. of tuples:              487
No. of attributes:   7
MVDs found:

6 ->-> 7
6 ->-> 1 2 3 4 5

==========================================
```

Figure 4.13: Multivalued dependencies found from the synthetic XML data (in Global Extraction Algorithm)

$\mid r \mid$	$\mid R \mid$	*Pure* MVD	MVD Candidates	MVD-validated	pruned by			
					Rule 2 & 3	Rule 4	Rule 5	Rule 6
487	4	2	16	9	1	1	5	0

Table 4.6: Results of multivalued discovery algorithm on synthetic XML data (in DTD-splitting Extaction Algorithm)

totype. However, as we mentioned in Chapter 3 Section 3.3.3, for the schema generated by *DTD-splitting* Schema Extraction Algorithm we might be able to further refine the design by undergoing another round of dependency discovery. As a result, we try to discover multivalued dependencies in the resulting schema. Using the schema from *Hybrid* method as an example, we do find multivalued dependencies in the table `table:monograph`. The result is shown in Table 4.6 and the multivalued dependencies found are shown in Figure 4.14. The refined design of `table:monograph` and a fraction of the refined tables based on 4NF decomposition is shown in Figure 4.15 and Figure 4.16.

```
table:monograph
No. of tuples:              487
No. of attributes:   4
MVDs found:

2 ->-> 3                    2 ->-> 1 4

=====================================
```

Figure 4.14: Multivalued dependencies found from `table:monograph`

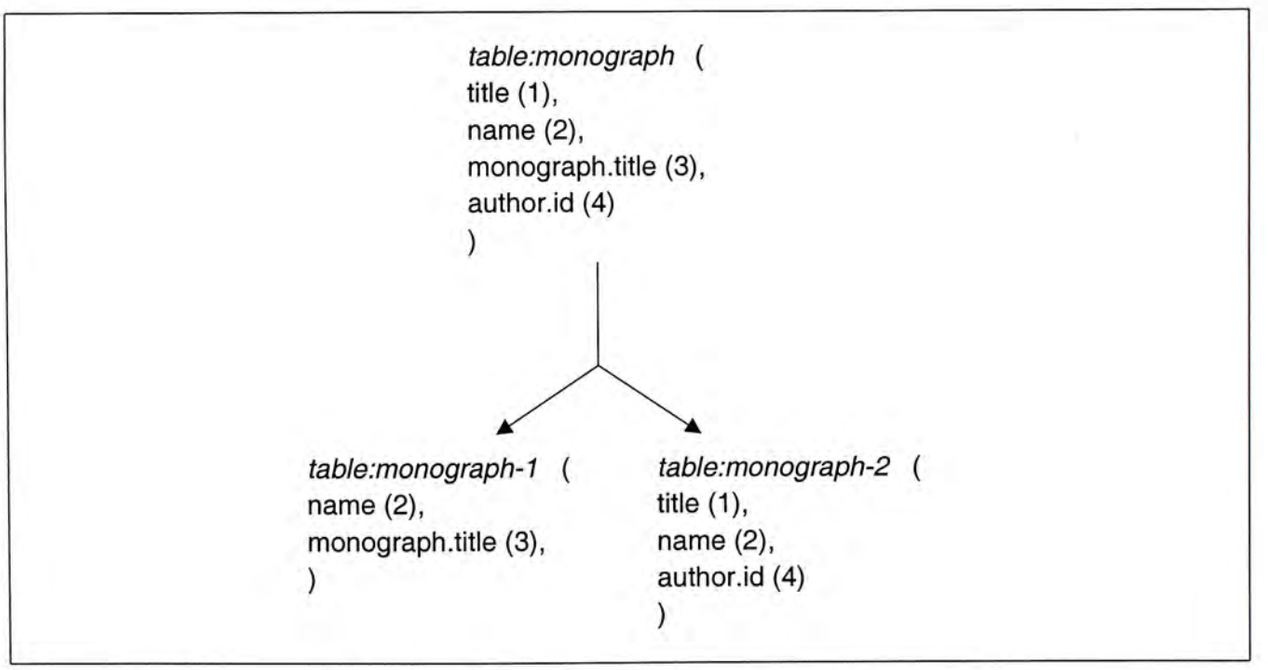

Figure 4.15: Refined design for table:monograph of synthetic XML data, which is produced by *hybrid* method, based on 4NF decomposition

table:monograph -1

name	*monograph.title*
Roy Chan	XML monograph
Roy Chan	DTD monograph
Roy Chan	mono XML
Willis Chan	SGML monograph
Willis Chan	XSL monograph
Roy Chan	XML monograph
Roy Chan	DTD monograph
Roy Chan	mono XML
Henry Hui	monograph XSLT
...	...

table:monograph -2

title	*name*	*author.id*
XML monograph	Roy Chan	2
XSL monograph	Willis Chan	2
DTD monograph	Roy Chan	4
monograph XSLT	Henry Hui	4
...	...	...

Figure 4.16: Tables of the refined `table:monograph`

Chapter 5

Conclusion

5.1 Discussion

It is clear that the fast emerging XML is becoming a dominant standard for representing data in the World Wide Web. When compared to HTML, it is obvious that XML encoding provides information in a far more convenient and usable format from a data management perspective. When viewing XML from a database point of view, it is possible to query the content of the XML documents. But what is the best way to provide this query capability over XML documents? The answer should depend on how we store the XML document into a database system. With XML documents having the characteristics of semistructured data, it seems that the recent research on storing and querying semistructured data can be easily applied to XML documents. And in fact, there has been great deal of activities exploiting new semistructured models and query languages for this purpose. A good example would be the Lore system which uses *OEM (Object Exchange Model)* [43] to store XML data, and use a semistructured query language Lorel for querying the XML data.

In theory, a semistructured system would clearly work and it should work best with the tailored features for handling XML data. However, is it the

only approach to take? Numerous researches in the past years on relational database have made today's RDBMS mature and well developed. The techniques in semistructured database are still in their early stage. It may take quite a while for semistructured database systems to be as well developed as RDBMS is. Before semistructured database system could be well prepared for XML data and while RDBMS is still the most dominant database system in the industry, we think it is quite reasonable to explore the possibility of leveraging relational database techniques to provide store and query capability over XML data. Before the development of semistructured database system is matured enough, relational database should be a very good alternative solution.

In this thesis, first we propose a *Global* schema extraction algorithm that relies on the functional and multivalued dependencies in the XML data. Unlike other previous work which only relies on the structure or the structure declaration (DTD)of the XML data, schemas created by *Global* algorithm are based on the real characteristics extracted from the XML data itself, thus ensuring all schema decompositions made in the algorithm are reasonably done based on relational database theory. In order to deal with the possible high exponential cost for finding dependencies when the structure of the XML is relatively large, as well as the change of the data characteristics when large scale update is performed which might affect the schema produced, we propose a *DTD-splitting* schema extraction algorithm that decompose the DTD of the XML data before schema extraction based on the relational database theory. Schemas created by *DTD-splitting* algorithm are based more on the characteristics extracted from the DTD correspond to the XML data than that of the actual XML data. Three different schema prototype construction methods are proposed with *Hybrid* method being the best one among them based on the studies using relational database concepts. In order to further enhance the design of the relational schema generated by our algorithms, a new algorithm for finding multivalued dependencies is proposed.

The algorithm shows nearly linear scalability in the number of tuples of the dataset and is very suitable for applying on a large set of data. Better relational schema for XML data can be produced with the information of multivalued dependencies in the XML data. With the extracted relational schema, the data in the XML document can be mapped into an RDBMS where relational techniques could be used to manage the XML data.

The astute reader may notice that some information about the XML document might be lost under our algorithms. This is indeed true: The relative order of each element is lost while constructing the schema prototype trees. However, we expect that this would not be a problem for storing XML documents in the field of e-commerce and EDI. The XML used in e-commerce and EDI fields are basically *data-centric* XML documents which have highly regular structure of text and low concern for the total order of elements. For *document-centric* XML documents which possesses less limit in the size of text and high concern in total order of elements, a content management system would be more suitable for maintaining the documents than RDBMS.

5.2 Future Work

We have some suggestion for the future direction of our research.

5.2.1 Translate Semistructured Queries to SQL

With our proposed algorithm, direct SQL queries are highly possible. When no artificial key is needed during the schema extraction process, all the data in the resulting tables should be from the original XML data thus direct SQL queries is possible. Users do not have to make queries using some artificial fields in the table that have no actual meaning to them. On the other hand, it is possible

for the users to use semistructured query language to make queries according to the actual structure of the XML data. In this case, The semistructured queries should be first translated to SQL statements, then queries are performed on the relational tables for the XML data. Many semistructured query languages are proposed for querying XML as a semistructured model [6, 19, 5, 48, 34, 18]. The main concept in these semistructured query language is the use of *path expression* which provides more flexibilities in querying than SQL. Based on the relational schema produced by our algorithms, it is possible to develop a set of query translation between semistructured queries and SQL queries for our relational tables of the XML data. We illustrate a possible translation by using the SIGMOD Record XML as example. Using the *hybrid* method relation schema in Figure 3.33 as the example relational schema here, if we have a semistructured query like the one shown in Figure 5.1, we can simply translate it into the SQL shown in Figure 5.2.

```
WHERE       <article>
                <title> From XML to Relational Database </title>
                <authors>
                    <author>$a</author>
                </authors>
            </article>
CONSTRUCT ALL <result>$a<result>
```

Figure 5.1: Example semistructured query

```
SELECT      X.author
FROM        table:author X, table:article Y
WHERE       X.Y.assignedID = Y.assignedID
AND         Y.title="From XML to Relational Database"
```

Figure 5.2: SQL translated from example semistructured query

However, as mentioned in [49], when the path expression is more complicated, e.g. with more operators in the path expression while having wild cards

loosening the constraints in the path expression, the translation would not be so straight-forward. As stated in [42], relatively less research efforts have been put on translating from XML queries to SQL queries. [42] proposed some general methods for it. We think that translating and optimizing the path expression into SQL for the relational schema generated by our algorithm would be an interesting topic to work on.

5.2.2 Improve the Multivalued Dependency Discovery Algorithm

We propose our algorithm for finding multivalued dependencies based on the concept of partition. Right now we have to generate partitions for all levels of candidates before validation for multivalued dependency candidates. The reason is that in a multivalued dependency validation, we have to use partitions from different levels. E.g. for $R = \{A, B, C, D, E, F, G\}$, validating if $\{A\} \rightarrow\rightarrow \{B\}$ would require $\pi_{\{A\}}$ (level 1 partition), $\pi_{\{ABC\}}$ (level 3 partition) as well as $\pi_{\{ADEFG\}}$ (level 5 partition). As a result, unlike the strategy used in [29], which is computing new partitions during dependency validation, our algorithm computes all the partitions before validation. We are interested to find out if there are other searching strategies which can reduce the level of partitions we have to used in each iteration of the validation process.

Moreover, for the disk version of our algorithm, we can further optimize the disk usage by introducing data compression techniques on the partition file. The trade-offs between disk compression and the disk access time will be studied.

5.2.3 Incremental Update of Resulting Schema

In our proposed algorithms, we produce the relational schema based on the characteristics, especially functional dependencies and multivalued dependencies, of the XML data. When the XML data is updated, it is possible that the set of dependencies in the XML data is changed too. As a result, the relational schema of the XML data may have to be updated as well. In *Global* algorithm, it is highly necessary to perform schema update since the schema is totally based on the dependencies in the XML data. In *DTD-splitting* algorithm, if the resulting schema depends more on the characteristics in the DTD, it is possible to avoid schema update even the XML data is updated. However, when the number of element and attributes declared in the DTD is large, it is still inevitable that the resulting schema depends on the dependencies in the XML data. Thus we have to propose some incremental update method for the resulting schema and try to reduce the cost of updating schema as much as possible.

One possible direction for the incremental update method is to try to minimize the number of tables in the resulting schema we have to update. The reason is that we expect that upon the update of XML data, usually only part of the dependencies discovered in the XML data would be updated. Thus just few resulting tables in the resulting schema might be affected and we should try to minimize the number in the update method. Another possible direction is to try to minimize the need for schema update. Here we can introduce the concept of *approximate functional dependency* [32] and use it to replace the role of functional dependency. The basic idea of approximate functional dependency is to define the *error* $e(X \rightarrow Y)$ for an functional dependency $X \rightarrow Y$. Upon the update of XML data, if $e(X \rightarrow Y)$ is still smaller than the threshold we set, we would still treat $X \rightarrow Y$ to be a valid dependency. Update of resulting schema is needed only when one of the approximate dependencies in XML data cannot hold, i.e. the error is larger than the threshold we set.

Bibliography

[1] World Wide Web Consortium (W3C). http://www.w3c.org/.

[2] D. Connolly and H. Thompson. XML Schema. http:// www.w3c.org/ XML/ Schema.

[3] ISO 8879. 1986 Information processing -Text and office systems - Standard Generalized Markup Language (SGML). http:// www.iso.ch/ cate/ d16378.html.

[4] S. Abiteboul. Querying semistructured data. In *Proceedings of the International Conference on Database Theory*, pages 1–18, Delphi, Greece, January 1997.

[5] S. Abiteboul, D. Quass, J. McHugh, J. Widom, and J. Wiener. The lorel query language for semistructured data. In *Journal of Digital Libraries*, volume 1(1), pages 68–88, April 1997.

[6] S. Adler, A. Berglund, J. Caruso, S. Deach, P. Grosso, E. Gutentag, A. Milowski, S. Parnell, J. Richman, and S. Zilles. Extensible Stylesheet Language (XSL) Version 1.0. Working Draft 27 available at http:// www.w3.org/ TR/ xsl/, March 2000.

[7] R. Agrawal, H. Mannila, R. Srikant, H. Toivonen, and A. Verkamo. Fast discovery of association rules. In *Advances in Knowledge Discovery and Data Mining*, pages 307–328, 1996.

[8] W. Armstrong. Dependency structures of data base relationships. In *Proceedings of the 1974 IFIP Congress*, pages 580–583, 1974.

[9] C. Beeri, R. Fagin, and J. Howard. A complete axiomatization for functional and multivalued dependencies. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 47–61, 1977.

[10] C. Beeri and T. Milo. Schemas for integration and translation of structured and semi-structured data. In *Proceedings of the International Conference on Database Theory*, 1999.

[11] C.L. Blake and C.J. Merz. UCI repository of machine learning databases, 1998. University of California, Irvine, Dept. of Information and Computer Sciences.

[12] Jon Bosak. Xml, java, and the future of the web. http:// metalab.unc.edu/ pub/ sun-info/ standards/ xml/ why/ xmlapps.htm.

[13] Jon Bosak and Tim Bray. Xml and the second-generation web. In *Scientific American*, May 1999. Web edition available at: http:// www.sciam.com/ 1999/ 0599issue/ 0599bosak.html.

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

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

[16] P. Buneman. Semistructured data. In *Proceedings on the Principles of Database System (PODS)*, pages 117–121, 1997.

[17] P. Buneman. Semisturctured data. In *Proceedings of the Sixth ACM SIGACT-SIGMOD-SIGART Symposium on Principle of Database Systems*, Tucson, Arizona, May 1997.

[18] S. Ceri, S. Comai, E. Damiani, P. Fraternali, S. Oaraboschi, and L. Tanca. XML-GL: a graphical language for querying and restructuring xml documents. In *Proceedings of the international WWW Conference*, 1999.

[19] J. Clark. W3C Recommendation: XSL Transformations (XSLT) Version 1.0. http:// www.w3.org/ TR/ xslt, November 1999.

[20] E. Codd. A relational model fro large shared data banks. In *Communications of the ACM*, volume 13(6), pages 377–387, 1970.

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

[22] A. Deuntsch, M Fernandez, and D. Suciu. Storing semistructured data with stored. In *Proceedings of ACM SIGMOD Conference on Management of Data*, Philiadelphia, PA, 1999.

[23] R. Fagin. Multivalued dependencies and a new normal form for relational databases. In *ACM Trasactions on Database Systems*, volume 2(3), pages 262–278, 1977.

[24] M. Fernadez, D. Florescu, J. Kang, A. Levy, and D. Suciu. Catching the boat with Strudel: Experiences with a web-based management system. In *Proceedings of ACM SIGMOD Conference on Management of Data*, Seattle, WA, September 1997.

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

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

[27] R. Goldman, J. McHugh, and J. Widom. From semistructured data to xml: Migrating the lore data model and query language. In *1999 WebDB Workshop*, 1999.

[28] Ykä Huhtala, Juha Käkkäinen, Pasi Porkka, and Hannu Toivonen. Tane program. http:// www.cs.Helsinki.FI/ research/ fdk/ datamining/ tane/.

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

[30] IBM. DB2 XML Extender. http:// www-4.ibm.com/ software/ data/ db2/ extenders/ xmlext/.

[31] Gerti Kappel, Elisabeth Kapsammer, S. Rausch-Schott, and Werner Retschitzegger. X-ray - towards integrating XML and relational database systems. In *International Conference on Conceptual Modeling / the Entity Relationship Approach*, pages 339–353, 2000.

[32] J Kivinen and H. Mannila. Approximate inference of functional dependencies from relations. In *Theoretical Computer Science*, September 1995.

[33] D. Lee and W. Chu. Constriants-preserving transformation from XML document type definition to relational schema. In *Proceeding of Proc. 19th International Conference on Conceptual Modeling (ER),*, 2000.

[34] A. Levy, M. Fernadez, D. Florescu, J. Kang, and D. Suciu. XML-QL: a query language for xml. In *Proceedings of the International WWW Conference*, 1999. Also available at http:// www.w3.org/ TR/ NOTE-xml-ql/.

[35] Wie Ming Lim and John Harrison. Parallel approaches for discovering functional dependencies from data for information system design recovery. In

Proceedings of the 1997 International Symposium on Parallel Architectures, Algorithms and Networks (ISPAN '97), Taipei, Taiwan, 1997.

[36] D. Maier. The theory of relational databases. In *Computer Science Press*, 1983.

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

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

[39] H. Mannila and K. Räihä. Algorithms for inferring functional dependencies. In *Data Knowledge Engineering*, 1994.

[40] H. Mannila and H. Toivonen. Levelwise search and borders of thories in knowledge discovery. In *Data Minging and Knowledge Discovery*, pages 241–258, 1997.

[41] H. Mannila, H. Toivonen, and A. Verkamo. Discovery of frequency episodes in event sequences. In *Data Minging and Knowledge Discovery*, pages 259–289, 1997.

[42] I. Manolescu, D. Florescu, and D. Kossmann. Pushing XML queries inside relational databases. Technical report, INRIA, 2001.

[43] J. McHugh, S. Abiteboul, R. Goldman, D. Quass, and J. Widom. Lore: A database management system for semistructured data. In *SIGMOD Record*, volume 26(3), pages 54–66, September 1997.

[44] J. McHugh and J. Widom. Query optimization for semistructured data. Technical report, Standford University, November 1997.

[45] J. McHugh, J. Widom, S. Abiteboul, Q.Luo, and A.Rajaraman. Indexing semistructured data. Technical report, Standford University, January 1988.

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

[47] S. Parikh, M Ganesh, and J. Srivastava. Parallel data mining for functional dependencies. Technical report, University of Minnesota, 1996.

[48] J. Robie, J. Lapp, and D. Scach. XML query language (XQL). http:// www.w3c.org/ TandS/ QL/ Ql98/ pp/ xql.html, 1998.

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

[50] A Silberschatz, H. Korth, and S. Sudarshan. *Database System Concepts*, chapter 2. McGraw Hill, 1996.

[51] Oracle Corporation Steve Muench. Using xml and relational databases for internet applications.

[52] Igor Tatarinov, Zachary G. Ives, Alon Y. Halevy, and Daniel S. Weld. Updating xml. In *ACM SIGMOD Conference 2001*, pages 413–424, 2001.

[53] C. Zaniolo. *Analysis and Design of Relational Schemata for Database Systems.* PhD thesis, University of California, Dept. of Computer Science, 1976.

Appendix A

Simple Proof for Minimality in Multivalued Dependencies

Assume that X, Y, Z, W partition the attributes in the relation R. We assume that the multivalued dependency $X \rightarrow\rightarrow Y$ hold for R.

To see if $XZ \rightarrow\rightarrow Y$ holds, we have to find out the tuple pairs having the same attribute values in X and Z, and see if their corresponding tuples are in R too. So if

(1) (x, y, z, w) and

(2) (x, y', z, w')

are tuples of R, by the definition of multivalued dependency, if $XZ \rightarrow\rightarrow Y$ holds we expect R contains corresponding tuples

(3) (x, y, z, w') and

(4) (x, y', z, w).

Since $X \rightarrow\rightarrow Y$ holds, when there is tuple (1) and tuple (2), there exist tuples

(5) (x, y', z, w) and

(6) (x, y, z, w')

in R too.

As tuple (3) = tuple (5) and tuple (4) = tuple (5) so R contains the corresponding tuples we expected as well.

Since all the expected tuples are contained by R, we prove that for a relation R if $X \twoheadrightarrow Y$ holds, $XZ \twoheadrightarrow Y$ also holds in R where Z represents attributes in R other than X and Y. Actually it is a special case for *multivalued augmentation rule.* The rule states that if $\alpha \twoheadrightarrow \beta$ holds and $\gamma \subseteq R$ and $\delta \subseteq \gamma$, then $\gamma\alpha \twoheadrightarrow \delta\beta$ holds. When substituting δ with $\emptyset$, we can get the minimality rule we just proved.

Appendix B

Third and Fourth Normal Form Decompositions

In our proposed algorithms, we decompose the schema prototypes into the relational schemas based on the functional dependencies or even the multivalued dependencies found in the XML data. In our examples and experiments, we normalized the schema prototypes based on Third Normal Form(3NF) and Fourth Normal Form(4NF) respectively. The decomposition algorithm for 3NF and 4NF are shown below.

B.1 3NF Decomposition Algorithm

Algorithm 3NF_DECOMPOSITION

1 F_c is the canonical cover of the set of functional dependencies;
2 $i := 0$;
3 **for** each functional dependency $X \rightarrow Y$ in F_c **do**
4 **if** none of the schemes R_j, $1 \leq j \leq i$ contains XY
5 **then** begin
6 $i := i + 1$;
7 $R_i := XY$;
8 **end**
9 **if** none of the schemes R_j, $1 \leq j \leq i$ contains a candidate key for R
10 **then** begin
11 $i := i + 1$;
12 $Ri :=$ any candidate key for R;
13 **end**
14 **Return** $(R_1, R_2, ..., R_i)$

B.2 4NF Decomposition Algorithm

Algorithm 4NF_DECOMPOSITION

1 $result := \{R\}$;
2 $done :=$ false;
3 compute F^+; F^+ is the closure of the set of functional dependencies
4 **while** (**not** *done*) **do**
5 **if** there is a scheme R_j in *result* that is not in 4NF
6 **then** begin
7 let $X \rightarrow\rightarrow Y$ be a nontrivial multivalued dependency that
8 holds on R_j such that $X \rightarrow R_j$ is not in F^+, and $X \cap Y = \emptyset$
9 $result := (result - R_j) \cup (R_j - Y) \cup (X, Y)$;
10 **end**
11 **else** *done* := true;
12 **then** begin