Journal of Southeast University (English Edition) Mar. 2002 Vol.18 No.1 ISSN 1003—7985

Lossless Mapping from Semi-Structured Data to Structured Data*

Li Wenwu** Jin Yuanping Tong Mina

(Department of Computer Science and Engineering, Southeast University, Nanjing 210096, China)

Abstract: Most semi-structured data are of certain structure regularity. Having been stored as structured data in relational database (RDB), they can be effectively managed by database management system (DBMS). Some semi-structured data are difficult to transform due to their irregular structures. We design an efficient algorithm and data structure for ensuring lossless transformation. We bring forward an approach of schema extraction through data mining, in which different kinds of elements are transformed respectively and lossless mapping from semi-structured data to structured data can be achieved.

Key words: semi-structured data, DTD, RDB, schema mapping, DOM, overflow data

The Web has been becoming an important approach to information acquiring, transmitting and exchanging. The Web data are of certain structure regularity. But different from structured data in traditional database, they are not in accordance with a concrete data model. They have the properties of self-description, dynamic changing and semi-structure. Extensible markup language (XML) provides well support for semi-structured data, and becomes the standard of data exchange in the Web. Self-description of semi-structured data provides flexibility, but it also takes up a large quantity of storage space and processing time. Meanwhile, we are short of efficient and suitable management mechanisms and query functions for semi-structured data.

Based on a mapping query language semi-structured to relational data(STORED)[1], we developed a prototype system which can automatically and losslessly map XML semi-structured data to structured relational data and the corresponding overflow data collection. Thereby we can efficiently manage semi-structured data with mature RDBMS and promote the application of database technology in the Web. The transformation from XML data to relational database data is lossless in the sense that we can expediently transform RDB data back to original XML data.

1 Lossless Mapping of DTD (or XML Schema)

Document type definition (DTD) formulates the pattern model of all the element contents in XML documents. It helps authors to write validated XML documents. We map DTD to relational database in such a way that we can check data integrity on operations using relational DBMS to manage semi-structured data and get the original DTD whenever it is needed in data exchange processing.

The mapping scheme maps DTD data into the following three relational tables:

1) Element table, stores element information such as father element name, element type, element relative position, occurrence times, and so on.

2) Attribute table, stores attribute information such as attribute name, name of the element that the attribute belongs to, attribute type, default attribute value, occurrence of the attribute.

3) Attribute_value table, stores enumerated values of enumerated type attribute.

We use the following faculty.dtd as an example to describe the mapping process. Due to the space limitation, we omit element type declaration of most "#PCDATA" type elements:

```
〈?xml version = "1.0" encoding = "UTF-8"?〉
〈!ENTITY % sexalign "sex ( male | female)
   # REQUIRED"〉
〈!ELEMENT faculty (teacher * )〉
〈!ELEMENT teacher (name, address +, phone *, student *, major * )〉
〈!ATTLIST teacher
      teacherid ID # REQUIRED
      % sexalign;〉
〈!ELEMENT name (firstname, lastname)〉
  〈!ELEMENT address (city, street, ((apartment, roomnumber) | dwelling))〉
〈!ATTLIST address
      zip CDATA # REQUIRED
      addresssort (home | office | lab) "home"〉
〈!ATTLIST phone
      areacode CDATA # REQUIRED〉
〈!ELEMENT student (studentname)〉
```

Received 2001 - 12 - 04.

* The project supported by the plan of key university faculty members of State Education Ministry and "333" Talent Plan of Jiangsu Province.

** Born in 1978, male, graduate.

```
〈!ATTLIST student
    studentid ID # REQUIRED
    % sexalign;〉
〈!ELEMENT major (field)〉
〈!ELEMENT firstname ( # PCDATA)〉
…
```

- Create element table (Tab.1). There are seven columns in the table with column "elementname" for element name, "fathername" for father element name, "datatype" for element type. And the column "minOccur" represents the minimum occurrence times of the corresponding element, and "maxOccur" the maximum occurrence times. The "- 1" stands for "+ ∞". Columns of "sequence_num" and "is_choice" indicate the relative position of an element within its father element and whether the element is from a choice list. These two columns differentiate choice lists of content particles from sequence lists of content particles and record relative positions of elements in sequence lists, and therefore ensure losslessness of element transformation. Sometimes the relative positions in some complex elements are difficult to record. In these cases, we create transitional elements in place of the complex elements. The field "datetype" of transitional elements is null.

Tab.1 Element table

elementname	fathername	datatype	sequence_num	is_choice	minOccur	maxOccur
address	teacher	children	2	False	1	- 1
address # 3.1 #	address	null	3	Ture	1	1
apartment	address # 3.1 #	# PCDATA	1	False	1	1
city	address	# PCDATA	1	False	1	1
dwelling	address	# PCDATA	3	Ture	1	1
faculty	# root #	children	1	False	1	1
roomnumber	address # 3.1 #	# PCDATA	2	False	1	1
street	address	# PCDATA	2	False	1	1
teacher	faculty	children	1	False	0	- 1
⋮	⋮	⋮	⋮	⋮	⋮	⋮

For instance, when the element type declaration of "〈!ELEMENT address (city, street, ((apartment, roomnumber) | dwelling))〉" is transformed, the values of field "sequence_num" of elements of "city" and "street" are "1" and "2" respectively, and the values of field "is_choice" of both of them are "False". The values of field "sequence_num" of elements of both "(apartment, roomnumber)" and "dwelling" are "3", and the values of field "is_choice" of them are "True". The elements of "apartment" and "roomnumber" have their own sequence numbers ("1" and "2" respectively) within their complex father element "(apartment, roomnumber)". We create a transitional element "address # 3.1 #" in place of the complex element "(apartment, roomnumber)". The elements of "address # 3.1 #" and "dwelling" are regarded as choice list of child elements of the element "address"; and the elements of "apartment" and "roomnumber" are regarded as sequence list of child elements of the element "address # 3.1 #" (Tab.1).

- Create attribute table (Tab.2) for storing attribute information of elements. There are five columns in the table: "attributename" for the name of attribute, "elementname" for the name of elements associated by attributes, "attributetype" for the data type of attribute, "attributeDefault" for the default value of attribute and "attibutePresence" for whether the attribute is required, implied or fixed.

Tab.2 Attribute table

attributename	elementname	attributetype	attributeDefault	attributePresence
addresssort	address	EnumeratedType	home	# REQUIRED
areacode	phone	CDATA	null	# REQUIRED
sex	student	EnumeratedType	null	# REQUIRED
sex	teacher	EnumeratedType	null	# REQUIRED
studentid	student	ID	null	# REQUIRED
teacherid	teacher	ID	null	# REQUIRED
zip	address	CDATA	null	# REQUIRED

The field "attributeDefault" is set to null if the corresponding attribute has no default value. Although many attributes might have no default values and therefore result in many nulls in the column "attributeDefault", the space cost for these nulls would not be very high because the attribute table is just a small static table in the whole system.

- Create Attribute_value table (Tab.3). There are

three columns in the table: "attributename", "elementname", "attributevalue"(enumerated value of attribute).

Tab.3 Attribute _ value table

attributename	elementname	attributevalue
addresssort	address	home
addresssort	address	lab
addresssort	address	office
sex	student	female
sex	student	male
sex	teacher	female
sex	teacher	male

Attribute _ value table has two usages: ① Store enumerated values of enumerated type attributes to achieve lossless mapping of attributes; ② When we need bind constraints with attribute column on creating relational table during transformation of XML data, we can obtain the required information by directly accessing Attribute _ value table instead of traversing DTD once more (see 3.1.1).

2 Selection of Mapping Scheme

We illustrate the selection strategy through the example of faculty.xml.

```
〈?xml version = "1.0" encoding = "UTF - 8"?〉
〈!DOCTYPE faculty SYSTEM
    "E: \ Documents \ faculty.dtd"〉
〈faculty〉
  〈teacher teacherid = "1003" sex = "male"〉
    〈name〉
      〈firstname〉Leon〈/firstname〉
      〈lastname〉Smith〈/lastname〉
    〈/name〉
    〈address addresssort = "home" zip = "210000"〉
      〈city〉Nanjing〈/city〉
      〈street〉Suzhou Road〈/street〉
      〈apartment〉3 # 〈/apartment〉
      〈roomnumber〉301〈/roomnumber〉
    〈/address〉
    〈phone areacode = "025"〉4567891〈/phone〉
    〈phone areacode = "025"〉5678912〈/phone〉
    〈phone areacode = "025"〉6789123〈/phone〉
    〈student studentid = "2003" sex = "male"〉
      〈studentname〉Mike〈/studentname〉
    〈/student〉
    〈student studentid = "2004" sex = "female"〉
      〈studentname〉David〈/studentname〉
    〈/student〉
    〈major〉
      〈field〉Pattern Recognization〈/field〉
    〈/major〉
    〈major〉
      〈field〉Artificial Intelligence〈/field〉
    〈/major〉
  〈/teacher〉
  〈teacher teacherid = "1004" sex = "male"〉
    〈name〉
      〈firstname〉Tom〈/firstname〉
      〈lastname〉White〈/lastname〉
    〈/name〉
    〈address addresssort = "lab" zip = "210000"〉
      〈city〉Nanjing〈/city〉
      〈street〉Yunnan Road〈/street〉
      〈apartment〉4 # 〈/apartment〉
      〈roomnumber〉404〈/roomnumber〉
    〈/address〉
    〈address addresssort = "home" zip = "210000"〉
      〈city〉Nanjing〈/city〉
      〈street〉Taiping Road〈/street〉
      〈dwelling〉1 # 〈/dwelling〉
    〈/address〉
    〈phone areacode = "025"〉1023456〈/phone〉
    〈phone areacode = "025"〉1203456〈/phone〉
    〈student studentid = "2005" sex = "male"〉
      〈studentname〉Wilfred〈/studentname〉
    〈/student〉
    〈major〉
      〈field〉Local Area Network Techonology
      〈/field〉
    〈/major〉
  〈/teacher〉
...
  〈/faculty〉
```

2.1 Mapping scheme of individual element table

One of the simplest mapping schemes is to transform every element into its own separate table, and the text contents and attributes of the element are stored in the columns of the relational table. Two fields ("ElementID" and "ParentID") are used to associate parent elements and child elements (Tab.4 – Tab.7).

Tab.4 Teacher table

ParentID	ElementID	teacherid	sex
0	3	1003	male
0	4	1004	male
⋮	⋮	⋮	⋮

Tab.5 Name table

ParentID	ElementID
3	9
4	10
⋮	⋮

Tab.6 Firstname table

ParentID	ElementID	firstname
9	15	Leon
10	16	Tom
⋮	⋮	⋮

Tab.7 Lastname table

ParentID	ElementID	lastname
9	21	Smith
10	22	White
⋮	⋮	⋮

Using this scheme, every element instance is stored as a record in its corresponding table. Its advantage is that no matter how many elements are in the XML document, no extra null except the nulls within XML document is produced in the relational table. But from Tab.4 – Tab.7, we can see that there is a one-to-one correspondence between teacher and name, and name and (firstname, lastname) as well. They can obviously be merged into one table. If we create a table for every individual element with two columns of "ElementID" and "ParentID", then the two columns may contain redundant data in most cases. Furthermore, a large number of connections of tables are required on the query and update actions, resulting in substantially decreased access performance. In addition, the number of the tables can be too big to manage efficiently due to the one table for one element transformation.

2.2 Mapping scheme based on a template

We distinguish two classes of elements: simple elements and complex elements. A simple element is an EMPTY element or an element only contains text, whereas a complex element contains child elements. For example, in faculty.dtd, "firstname", "lastname" and "phone" are simple elements, and "name", "address", "student" and "major" are complex elements. We can store simple elements that have one-to-one or deterministic-many-to-one correspondences with their father elements in their father element tables. This eliminates a part of redundant connections of individual element tables. Meanwhile, some simple elements such as "phone" have non-deterministic-many-to-one correspondences with their father elements. We call these elements non-deterministic simple elements. They can't be stored directly into the tables of their father elements because we don't know how many columns we need to store them. Besides, the complex elements must be dealt with.

2.2.1 Mapping method of non-deterministic simple elements

We intend to store non-deterministic simple elements into the tables of their father elements to improve the access performance. The key problem is how many more columns should be created in the father tables for these child elements. On one hand, if the number of the columns is set too big, a wide, sparse table is generated, resulting in low storage usage. On the other hand, if the number is too small, many data may overflow, resulting in the decreased access performance. We expect to get a most highly supported template according to which the number of the columns can be decided. A schema extraction method based on WL's data mining algorithm[2] is designed for this purpose.

We get all element nodes from document object model (DOM) tree and encode the elements, e.g., name[1], phone[1], phone[2]. The encoded elements are regarded as different elements during data mining. We obtain every path from root node to leaf node from DOM tree, e.g.,

faculty[1].teacher[1].name[1].firstname[1]
faculty[1].teacher[3].phone[3]
faculty[1].teacher[4].address[2].dwelling[1]

and so on. Given a minimum support (Supp), the algorithm discard the paths whose matching numbers are below Supp, and then merge the remaining paths to a template. The template produced for faculty.xml is as follows:

faculty(teacher(name[1](firstname[1],lastname[1]),
address[1](city[1],street[1],apartment[1],roomnumber[1]),
phone[1],phone[2],
student[1](studentname[1]),
major[1](field[1])))

According to the template, five more columns are required in the teacher table to store the element "phone": two columns (one column for text content and another for attribute "areacode") for each of the phone1 and phone2, and one column for overflow indicating flag (see 3.1).

2.2.2 Mapping method of complex elements

A convenient mapping method of complex elements is to create an individual table for every complex element. The elements in the table keep relationships with their father elements through columns of "ParentID" and "ElementID".

With a further observation, we find that for the element "major", there are a one-to-non-deterministic-many correspondence between teacher and major and a one-to-one correspondence between major and field. Therefore the correspondence between teacher and field is also one-to-non-deterministic-many. According to the template for faculty.xml, most teachers have one and only one field, in other words, one field is most highly supported by the teachers. So we can directly add a column of "field" in the teacher table (see Tab.8). As another example, there is a one-to-one correspondence between teacher and name, and name

and (firstname, lastname) as well. So we can add column "firstname" and column "lastname" in the teacher table.

Tab.8 Teacher table (inverse)

ElementID	5	10	…
teacherid	1003	1004	…
sex	male	male	…
firstname	Leon	Tom	…
lastname	Smith	White	…
phone1	4567891	1023456	…
phone1 _ areacode	025	025	…
phone2	5678912	1203456	…
phone2 _ areacode	025	025	…
phone _ overflow	True	False	…
student _ studentid	2003	2005	…
student _ sex	male	male	…
studentname	Mike	Wilfred	…
student _ overflow	True	False	…
field	Pattern Recognization	Local Area Network Techonology	…
major _ overflow	True	False	…

We distinguish two classes of complex elements. A unary complex element such as "student" or "major" contains only one child element. A multiple complex element such as "name" or "address" contains two or more child elements. Unary complex elements can be stored in their father element tables, whereas multiple complex elements are usually stored in their own separate tables. If the correspondence between a multiple complex element and its father element is deterministic-many-to-one, the multiple complex element can also be stored in its father element table, e.g., the correspondence between (firstname, lastname) and teacher is two-to-one, so (firstname, lastname) can be stored in the teacher table.

3 Data Transformation

3.1 Creating relational table

We use the mode abstraction method in 2.2.1 to get the template of XML document before creating relational table.

3.1.1 Common relational table

First we choose and tune two parameters A and C: Each table usually can have at most A columns; C is the collection size threshold indicating whether the collection elements need to be stored in nested or separate relational tables. Collection elements are the elements that occur at least twice in the template. This occurrence number multiplied the number of its child elements and attributes is the collection size, e.g., "phone" is a collection element with its collection size being 4. For non-deterministic simple elements, if we store them in their father element tables, we should increase the number of collection size of columns so as to balance the reduction of overflow data and the decrease of nulls. The larger the collection size of a collection element is, the bigger the occurrence probability of nulls is. When the collection size is bigger than or equals to C, we store the collection element in a separate relational table.

According to the correspondence between child elements and father elements in DTD (or XML schema), we divide child elements into three classes: Class I, the correspondence is one-to-one; Class II, the correspondence is deterministic-many-to-one, e.g., "name"; Class III, the correspondence is non-deterministic-many-to-one, e.g., "address", "phone", "student" and "major". Then we calculate the numbers of columns to be taken up by every element and sort them in ascending order in each class. Note if the maximum element occurrence time in DTD is bigger than the occurrence time in the template for faculty.xml, we append a column for the flag indicating the overflow of the element, e.g., "dwelling", which is the child element of "address", its maximum occurrence time in DTD is 1, whereas the occurrence time in the template is 0, so element "address" should append a column of "dwelling _ overflow". We can figure out the columns of the child elements of the element "teacher" are: Class II, name: 2; Class III, major: 2, student: 4, phone: 5, address: 8.

The processing algorithm to these three classes of elements is as follows:

1) Create separate tables for the multiple complex elements in Class III in order to prevent the loss of relations between their child elements when the complex elements overflow. Then remove these elements from Class III.

2) Create separate tables for the collection elements whose collection sizes are bigger than or equal to threshold C in Class III so as to decrease the nulls in the father element tables. Then remove these elements from Class III.

3) Increase the number of columns in the father element tables to accommodate all occurrences for the elements in Class I.

4) Calculate the sum of the number of columns of

attributes of the father element and that of all elements in Class I as Total _ fields. If Total _ fields is bigger than or equals to A, create separate tables for all the elements in Class II and Class III and jump to 6).

5) If Total _ fields is smaller than A, increase the number of columns in the father element tables taking turns of sorted elements of Class II and Class III as long as the sum of Total _ fields and that number of columns is still not bigger than A; the Total _ fields is updated with the sum simultaneously. Create separate tables for the remaining elements in Class II and Class III.

6) Use steps 1) – 5) for each of the elements to be stored in separate tables.

In the example of faculty.xml, A is set to 30, and elements of "name", "phone", "student" and "major" are stored in teacher Table (Tab.8), whereas "address" is stored in a separate table (Tab.9).

Tab.9 Address table

ParentID	ElementID	Zip	addresssort	city	street	apartment	room number	dwelling _ overflow
5	6	210000	home	Nanjing	Suzhou Road	3 #	301	False
10	11	210000	lab	Nanjing	Yunnan Road	4 #	404	False
10	12	210000	home	Nanjing	Taiping Road	null	null	True
⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮

We append a column of "ElementID" in every common relational table. For the elements whose child elements are stored in separate tables, "ElementID" is used to associate their records with the corresponding records in child element table; for the elements having columns of overflow indicating flag, "ElementID" is used to associate their records with the corresponding records in the overflow _ element table (see 3.1.2 and Tab.10). In every child element table we also append a column of "ParentID" as a foreign key, which corresponds to the "ElementID" in its father element table. "ParentID" is used to associate the records in the table with the corresponding records in the father element table.

Tab.10 Overflow _ element table

ParentID	ElementID	elementname	elementvalue
5	7	phone	6789123
5	8	student	null
8	null	studentname	David
5	9	major	null
9	null	field	Artificial Intelligence
12	null	dwelling	1 #

When adding attribute columns in the common relational table, we set the rules and constraints of those columns referring to the attribute _ value table (Tab.3). For instance, the column "addresssort" in address table (Tab.9) is bound to the rule "@addresssort in ('home', 'office', 'lab')" and is evaluated the default value ("home"). In addition, the default values of all the overflow indicating flags are "False", or non-overflow.

3.1.2 Overflow relational table

There are often a few data that can't be stored in the common relational tables. We use the overflow _ element table and the overflow _ attribute table to store these data.

- Overflow _ element table (Tab.10): has four columns of "ParentID", "ElementID", "elementname" and "elementvalue".

"ParentID" in overflow _ element table associates with "ElementID" in common relational tables. We set the values of "ElementID" unique in all common tables to ensure every record can exactly match its overflow elements. We design a set of data structure in 3.2 to implement this.

The overflow _ element table and all the common relational tables have the column "ElementID" and the unique property of its values provide scalability of the system (See 4). The function of the column "ElementID" in overflow _ element table is similar to that in common relational table. But different from the latter "ElementID" created as primary key, the former "ElementID" isn't required if the overflow element has no attributes and child elements. So the column "ElementID" in the overflow _ element table allows "null" value.

The column "elementname" in the overflow _ element table is created as a foreign key of the element table (Tab.1).

The column "elementvalue" need to store all kinds of type of element values, so we set its data type "ntext" and allows "null" value.

- Overflow _ attribute table (Tab.11): have three columns of "ElementID", "attributename", and "attributevalue". The data type of column "elementvalue" is set to "ntext" and it allows "null" value.

Tab.11 Overflow _ attribute table

ElementID	attributename	attributevalue
7	areacode	025
8	studentid	2004
8	sex	female

3.2 Loading the relational tables

We can expediently load the relational tables with XML data after the tables have been created. Store texts and attributes of elements in the relational tables through depth-first-traverse DOM tree. If we can't find the corresponding field for one element, there are two possibilities: ① A separate table has been created for the element; ② The element is overflow. We can distinguish these two cases from overflow indicating flag. If there is no corresponding overflow indicating flag for the element, it's the former case. We find the corresponding relational table according to the element name and then fill in the table with its attributes and child elements. Otherwise it's the latter case. We directly fill in the overflow _ element and overflow _ attribute tables with the element, its child elements and their attributes. In both of two cases, we fill the field "ParentID" of the child record with the value of "ElementID" of the father record.

We design the following data structure to make the values of "ElementID" unique. We use a variable CurrentID and a linked list RecycleID _ list to control the value of "ElementID". We set 0 as the initial value of CurrentID and null as the initial value of RecycleID _ list. When inserting a new record with a field "ElementID", we check if RecycleID _ list is null. If it's null, then we increase CurrentID by 1 and evaluate the field "ElementID" with CurrentID. Otherwise we evaluate the field "ElementID" with the value of the first node of RecycleID _ list and then remove the first node. When deleting a record with a field "ElementID", we append a new node with the value of the "ElementID" into RecycleID _ list.

The faculty.xml is transformed to four relational tables (Tab.8 – Tab.11) after the processing above.

4 Related Work and Discussion

Object oriented databases[3] can store XML documents without explicitly storing their schema, but a DTD is required to get the object-oriented schema. If DTD (or XML Schema) is known, it's an effective storage scheme for semi-structured data. But DTD (or XML Schema) is unknown in some applications. Our mapping scheme uses a schema extraction method based on WL's data mining algorithm and stores data in RDBMS instead of ODBMS. It is independent of DTD (or XML Schema).

We didn't deal with the mapping of comments, processing instructions, CDATA sections, prolog and standalone document declarations. We plan to use another special relational table to store these data.

The mapping scheme is well scalable. It can also support XML namespace by using the following two tables: ① ElementID-namespace table: generally speaking, most namespaces are attached to the complex elements that have unique "ElementID" values in the relational tables. Only three columns are required to record the information of this kind of namespaces: "ElementID", "Namespace _ name" and "Namespace _ value"; ② General namespace table: store the namespaces of attributes and comparatively simple elements. The table has four columns: "ParentID", "name" (name of element or attribute), "Namespace _ name" and "Namespace _ value".

The lossless mapping from semi-structured data to structured data is a challenging task, because they are obviously incompatible. Our hypothesis is that many semi-structured data sources have a mass regular structure with few outliers so that we can extract the schema to store data. After the semi-structured data have been transformed to structured data in RDB, they can be efficiently managed by traditional DBMS technology.

References

[1] Deutsch A, Fernandez M, Suciu D. Storing Semistructured Data with STORED[A]. *ACM SIGMOD International Conference on Management of Data*[C]. Philadelphia, 1999, **28**(2): 431 - 442.

[2] Wang K, Liu H. Discovering Typical Structures of Documents: a road map approach[A]. *ACM SIGIR Conference on Research and Development in Information Retrieval*[C]. New York, 1998.146 - 154.

[3] Christophides V, Abiteboul S, Cluet S, et al. From Structured Document to Novel Query Facilities[A]. In: Snodgrass R, Winslett M, eds. *Predeedings of* 1994 *ACM SIGMOD International Conference On Management of Data*[C]. Minneapolis, 1994.313 - 324.

半结构化数据到结构化数据的无损映射

李文武　金远平　童咪娜

（东南大学计算机科学与工程系，南京 210096）

摘　要　大多数半结构化数据都具有一定的结构规律，将它们转化为基于关系数据库存储的结构化数据，可有效地应用 DBMS 技术进行处理．部分不便于转化的数据作特殊处理，以保证整个数据的无损映射．本文在完成 DTD 的转换后，从一种最简单的映射方式入手，提出改进方案，利用一种基于数据挖掘的模式抽取方法，对不同类型的元素分别处理，设计了一套有效的溢出数据处理办法，实现了半结构化数据到结构化数据的无损映射．

关键词　半结构化数据，DTD，关系数据库，模式映射，DOM，溢出数据

中图分类号　TP311