XML Schemata Inference and Evolution

I. Sanz, J. M. Pérez, R. Berlanga and M. J. Aramburu

Departament de Llenguatges i Sistemes Informàtics
Departament de Ingenieria y Ciencia de los Computadores
Universitat Jaume I, E–12071 Castellón. Spain.
email:{isanz,martinej,berlanga,aramburu}@uji.es

Abstract. This work addresses the automatic generation of conceptual models for XML-oriented databases, which in many cases have little or no support for schemata. Our techniques are based on both an incremental clustering algorithm, which groups together the incoming XML documents according to their structural similarities, and a schema inference method, which maintains dynamically the schema of each detected document cluster. Our proposal takes into consideration the schema evolution. For this purpose, we have adapted the TOODOR document model that describes the temporal properties of the XML document types.

Keywords: Schema Inference, Document Clustering, XML Databases.

1 Introduction

The fast-paced adoption of XML as a language for data interchange among diverse applications and web services, is opening the way towards the real integration of heterogeneous information sources. In our opinion, the success of this integration will reside in three key factors. Firstly, it will be necessary to develop repositories for the massive storage of XML data and documents coming from several sources (i.e. Web warehouses). Secondly, query languages providing homogeneous access to these data are also needed. Finally, the availability of (semi-)automatic methods for the creation, management and integration of the conceptual schemata that describe the data in XML repositories is also an important factor.

Regarding the first two factors, currently there exist several native XML-oriented databases, as well as extensions of commercial databases, that allow for the storage and retrieval of XML data. XQuery [1] and XML-QL [2] are two relevant proposals of query languages for this type of data. Nevertheless, not all of these approaches are valid for the integration of repositories in XML. An important requisite is that they should be flexible enough to accept any XML document, regardless of the existence of an accompanying type definition or schema. It should be taken into account that many XML documents come from applications that do not provide such definitions.

This requisite provides the rationale for the third key factor in the integration of information sources. The possible lack of typing, or the excessive presence of

different type definitions, require the development of new mechanisms for the automatic definition of conceptual schemata. It should be noticed that current approaches to the semantic integration of data assume the existence of this kind of conceptual schemata (sometimes an ontology), and apply them to the search of relationships of equivalence or association; see [3, 4] for some examples.

In this paper we address the issue of the automatic generation of conceptual schemata by discovering the structural similarities between XML documents, and their clustering into classes. Our proposal has been implemented over the G commercial database system [5], which has been designed for the development of web applications that integrate heterogeneous and highly dynamic information sources. The G system transforms the information extracted from different sources into linked XML documents, which are stored in a native format. The techniques discussed in this paper obtain, in a dynamic and incremental way, a conceptual schema for this database, and represents it by means of the XML Schema Language (XMLS) [6].

The remainder of the paper is organized as follows. Section 2 gives a brief description of the G system. In Section 3 we define a structural similarity measure for XML documents, a clustering algorithm based on this measure, and the XMLS generation mechanism for the classes obtained by the clustering algorithm. Section 4 shows some preliminary results of this approach, and in Section 5 some conclusions are presented.

2 Context of the Work

This work is part of a project with the objective of automatically generating web-based applications for integrated repositories of XML documents. The system is called G, and its architecture is shown in Figure 1. The main contribution of this paper is the *Schema Manager* module, which is depicted at the right side of the figure. This module is in charge of monitoring the input stream of documents in order to classify them by their structural properties, and of inferring for each detected class an XML schema.

One relevant feature of this process is that the inferred schemata can evolve along time so that only the up-to-date schemata must be regarded in the generation of the database indexes and their applications. The evolution of schemata is guided by the structural changes detected by the clustering algorithm. On the other hand, once a schema becomes historical, it will be not considered by the clustering routine anymore. In this way, the time and space complexity of this module is reduced considerably.

In the next sections we describe the two components of the *Schema Manager* module, namely: the clustering routine and the schema inference and evolution component.

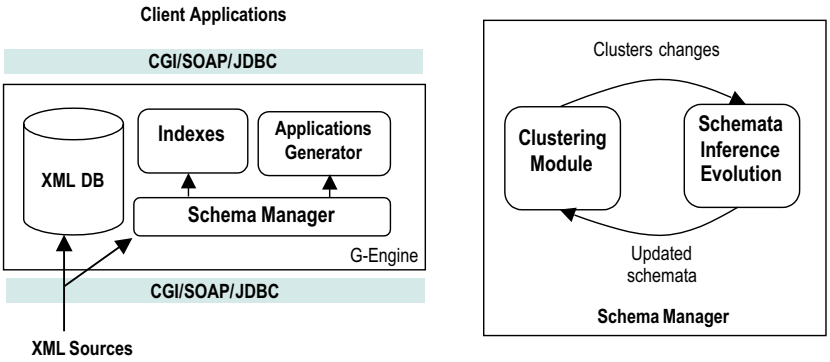

Fig. 1. Architecture of the G system

3 The Clustering Module

The clustering module is in charge of classifying each incoming XML document according to its structural properties. We assume that this clustering process is mainly non-supervised, since most XML documents in the web do not provide a known schema nor DTD. Another important requirement for the clustering routine is that it must be incremental, that is, for each incoming document the algorithm must adjust the classes involved according to their structure.

In the next sections we first describe the similarity measure used to cluster documents by their structure, and then the clustering routine.

3.1 Structural Similarity

In the literature there exist several proposals to determine whether two documents have a similar structure or not. Most of these approaches rely on the tree edit distance [7] and usually have high complexity costs. In the context of our application, we need to evaluate this similarity function over a large set of documents. That is why we need to find out a new similarity function with a lower temporal cost.

Thus we propose a different approach based on the similarity between document paths. We consider that each document is represented by the set of all the paths that go from its root element to each one of its leaves (text or attribute). Let us denote this set as $pathSet(d)$, where d is an XML document.

We also define the similarity between two paths, p_1 and p_2 as follows:

$$pathSim(p_1, p_2) = \frac{|elements(p_1) \cap elements(p_2)|}{max(|elements(p_1)|, |elements(p_2)|)}$$

where the function $elements$ returns the set of node elements of the given path.

As it would be expected, the more elements the paths share, the higher value the proposed measure returns. However, this measure does not take into account the relative order of the elements in the paths. With this simplification it is possible to return a high similarity value for a pair of incompatible paths (i.e. if the paths state different orderings for the same pair of elements). This drawback can be avoided by introducing the following compatibility function:

$$comp(p_1, p_2) = \begin{cases} false & \text{if } \exists e_1, e_2 \in elements(p_1) \cap elements(p_2) \\ & \text{such that } match(p_1, \text{“}//e_1//e_2//\text{”}) \\ & \wedge\ match(p_2, \text{“}//e_2//e_1//\text{”}) \\ true & \text{otherwise} \end{cases}$$

Here, the function $match(p, pe)$ returns $true$ if the path p matches the path expression pe.

Starting from these similarity and compatibility functions, we define the following global function to measure the similarity between the structure of two documents, d_1 and d_2:

$$Sim(d_1, d_2) = \frac{\sum_{p_i \in pathSet(d_1)} max_{p_j \in pathSet(d_2)} (pathSim(p_i, p_j))}{|pathSet(d_1)|}$$

$$docSim(d_1, d_2) = \begin{cases} 0 & \text{if } \exists p_1 \in d_1, p_2 \in d_2, \\ & \text{such that } \neg comp(p_1, p_2) \\ \frac{Sim(d_1,d_2)+Sim(d_2,d_1)}{2} & \text{otherwise} \end{cases}$$

With this formula, two documents have a similar structure if all their paths are compatible, and most of their paths are similar.

3.2 Clustering Routine

To describe the clustering routine, the following definitions are needed.

- Each document class C is a set of documents whose structural similarities with respect to the class representatives are greater than a given threshold β_{sim}. The class representatives, denoted $repSet(C)$, are themselves documents of the class. The similarity between two representatives of the class must not be greater than a given threshold β_{rep} ($\beta_{rep} \geq \beta_{sim}$). In this way, each class will represent a set of common structural properties, which are established by the intersection of all its representatives, as well as a set of structural particularities, which are stated by its representatives. Consequently, the threshold β_{sim} determines the degree of optionality of the associated class schema, whereas the number of representatives determines the degree of heterogeneity of the class schema.
- A document d *structurally subsumes* to another document d' if the path set of the former includes that of the latter, formally: $pathSet(d) \supset pathSet(d')$.
- The class to which a document d belongs is denoted with $class(d)$.

Algorithm 1 Clustering Routine

Require: d_{new}, β_{sim}, β_{rep}, InvertedFile, Classes
{ d_{new}: new incoming document;
β_{sim}: similarity threshold for documents;
β_{rep}: similarity threshold for representatives;
InvertedFile: inverted file for document class representatives;
Classes: set of currently detected classes; }
Ensure: Classes, InvertedFile
1: Select from the InvertedFile the representatives whose similarity with d_{new} is greater than β_{sim}, and put them into the set $Docs$.
2: $New = \emptyset$ {Auxiliary class that will contain the union of those classes similar to d_{new}}
3: **for all** $d \in Docs$ **do**
4: **if** $New = \emptyset$ **then**
5: $New = class(d) \cup \{d_{new}\}$
6: $New.repSet = repSet(class(d))$
7: **else**
8: $New = New \cup class(d)$
9: $New.repSet = New.repSet \cup repSet(class(d))$
10: **end if**
11: **if** $\exists d' \in repSet(New)$ such that d_{new} subsumes d' **then**
12: Remove all the representatives of New subsumed by d_{new}
13: Add d_{new} to the set of representatives of New
14: **end if**
15: Remove the class $class(d)$ from Classes, and all its representatives from the InvertedFile.
16: **end for**
17: **if** $New = \emptyset$ **then**
18: Add to Classes the new class $\{d_{new}\}$
19: Update the InvertedFile with d_{new}
20: **else**
21: **if** $\nexists d' \in repSet(New)$ such that $docSim(d', d_{new}) > \beta_{rep}$ **then**
22: add d_{new} to the set of representatives of New
23: **end if**
24: Add to Classes the updated class New.
25: Add the the representative set of New to InvertedFile.
26: **end if**

The clustering routine (see Algorithm 1) uses an inverted file over the representatives of the classes in order to efficiently calculate the structural similarity of each incoming document. Each entry of the inverted file represents a path element, and its associated value is the list of representatives having that path element.

The algorithm basically updates the inverted file and the set of current document classes according to the structure of each new incoming document. It takes into consideration the following situations:

- When the similarity between the new document and several representatives of different classes is greater than the given threshold β_{sim}, all the involved classes must be joined into one single class. The variable New is used for this purpose, which incrementally aggregates the involved classes (lines 3–16). Additionally, a representative set must be revised for the resulted class (lines 11–15 and lines 21–23).
- When the similarity between the new document and all the current representatives is not greater than the given threshold β_{sim}, the document belongs to a new class, whose representative is itself.
- The representative set of a class must be updated in the following two cases: when the new document structurally subsumes some of its current representatives, and when the similarity between the new document and all the class representatives is not greater than the threshold β_{rep}. In the first case, the new document replaces all the representatives that are subsumed by it. In the second case, the new document becomes a new representative of the class.

4 Inference and Evolution of XML Schemata

The second module of the *Schema Manager* is in charge of inferring a schema for each document class, and of deciding how an existing schema evolves over time. This module deals with the small changes produced by the arrival of new documents into the system, as long as they do not alter the cluster structure. For instance, the appearance of a new optional attribute will probably not change the cluster representative set, but it needs to be taken into account when reporting the current cluster schema. In any case, the *Schema Inference and Evolution component* is tightly coupled with the Clustering Routine, since the schemata need to be refreshed each time some structural change occurs.

The type model we have adopted for schema modelling is an extension of the XML Schema standard to include the TOODOR model (Temporal Object Oriented Document Organization and Retrieval) [8]. More specifically, we have defined an RDF[1] notation for expressing the evolution of an XML Schema according to the rules of the TOODOR model.

TOODOR combines a static XML-like type system with a set of temporal primitives to represent the evolution of schemata. The model defines $\mathcal{CI}$ as a set

[1] `http://krono.act.uji.es/toodor.rdfs`

of class identifiers, $\mathcal{OI}$ as a set of object unique identifiers and $\mathcal{AN}$ as a set of attribute names. As in other object-oriented data models, TOODOR allows class identifiers from $\mathcal{CI}$ to be used in the definition of types, being considered each class identifier as an object type. This type system is extended with two types for time expressions: $Time$ to denote time instants, and $Period$ to denote time periods. The document type system is denoted by $\mathcal{DOCTYPES}$, and the domain of time values is $\mathcal{PDATE}$.

In TOODOR a *class* is a 4-tuple whose components are as follows:

- $class\_id \in \mathcal{CI}$ is the class identifier,
- $lifespan \in \mathcal{PDATE}$ is the time period during which the class is defined,
- h_type is a sequence of pairs (p_i, T_i) with $i \geq 1$, where $p_i \in \mathcal{PDATE}$ and $T_i \in \mathcal{DOCTYPES}$,
- and finally, $h\_population$ is a sequence of pairs (p_i, I_i) with $i \geq 1$, where $p_i \in \mathcal{PDATE}$ and I_i is the subset of $\mathcal{OI}$ that coincides with the set of the class's instances created during the time period p_i.

4.1 Inference of types and temporal properties

The Schema Manager maintains one TOODOR schema for each cluster detected by the clustering routine. In order to keep these schemata up-to-date as new documents arrive to the system, the Schema Inference and Evolution component performs two tasks:

- The inference of types describing the static structure of the documents within a class at a given time.
- The tracking of the schema as it evolves along time, and in particular the *lifespan* of each static type.

In order to deal with the changes produced by the incoming documents, we use a label-relaxation algorithm. We represent TOODOR types as DAGs, in which nodes are class identifiers of $\mathcal{CI}$ and arcs are labelled to express the composition relationships between the types they represent. The allowed labels are analogous to those found in XML DTDSs: 1 (exactly one), ? (zero or one), $*$ (zero or more) and + (one or more).

The process begins with an initial sample document, usually a representative chosen by the clustering routine, which is transformed into a labelled DAG. When new documents are added to a cluster, its schema is modified according to the label transition rules of Table 1. In order to update the DAG, these rules take into account the labels in the existing schema plus the multiplicity of the composition relationships (0, 1 or N) of the new document.

Additionally, the inference routine tries to infer the basic type of the elements of the class (e.g. numeric, date or string) by applying lexical patterns to their values.

All these changes are time-stamped and kept into a log. This will allow the Schema Manager to reconstruct a schema at any given time point. Note that no

Initial label	Multiplicity	Resulting label
1	0	?
1	1	1
1	N	+
?	0, 1	?
?	N	*
+	0	*
+	1, N	+
*	0, 1, N	*

Table 1. Label transition rules

arcs or nodes are ever deleted in this phase, because nodes can only be added, and arcs only relaxed. Thus, the schema is just made more general.

A schema is considered completely evolved when the clustering routine detects a change in its structure (i.e. its representative set has changed). In this case the schema can no longer be considered valid, and it needs to be moved to a new *stable* state. For this purpose, the following procedure is applied:

- The schema is marked as obsolete and its associated change logs is closed.
- A new schema is created for the class according to the changes notified by the clustering routine.
- The new schema is initialized with a set of documents that the clustering routine has identified as belonging to its class. In order to create a schema that is as relevant as possible, these documents are chosen using a criterion of temporal closeness, within a temporal window with a length that depends on the rate of change of the schema (that is, the frequency of changes as detected by the clustering routine).
- The resulting schema is stored and marked as current.

Summarizing, no all of the past states of a schema are actually stored in the database, only the *stable* states created after each change in the clustering are. It is possible to reconstruct any intermediate state by following the changes logs, applying the necessary steps in order to obtain the schema that was current at the given time.

5 Evaluation

In order to evaluate our automatic validation system for XML document schemata, various experiments were designed. In this section we explain one of these tests and how its results prove the effectiveness of the system. In this particular test we start from four different document types whose DTDs are presented in Table 2. As can be noted, DTDs 1 and 2 are very similar, so they could be integrated under the same schema. By analyzing DTD 3 we can see that it is incompatible with the previous two, so they should not be fused together. Finally, DTD 4

is completely different from the other three. The objective of the experiment is to start from a repository of documents with instances of these four DTDs to check whether our system is able to group them correctly, and to infer the proper schemata that defines their structure. Although our system is also able to infer XML Schemata, in order to simplify the example, here we prefer to express the structure of documents by using DTDs.

DTD 1	**DTD 2**
`<!ELEMENT term (code, lecturers)>`	`<!ELEMENT term (lecturers)>`
`<!ELEMENT code (#PCDATA)>`	`<!ELEMENT lecturers (lecturer+)>`
`<!ELEMENT lecturers (lecturer+)>`	`<!ELEMENT lecturer (name, subjects)>`
`<!ELEMENT lecturer (name, subjects)>`	`<!ELEMENT name (#PCDATA)>`
`<!ELEMENT name (#PCDATA)>`	`<!ELEMENT subjects (subject*)>`
`<!ELEMENT subjects (subject+)>`	`<!ELEMENT subject (#PCDATA)>`
`<!ELEMENT subject (#PCDATA)>`	
DTD 3	**DTD 4**
`<!ELEMENT term (code, subjects)>`	`<!ELEMENT book (code, authors, title, place, publisher)>`
`<!ELEMENT code (#PCDATA)>`	`<!ELEMENT authors (name, address?)+>`
`<!ELEMENT subjects (subject+)>`	`<!ELEMENT code (#PCDATA)>`
`<!ELEMENT subject (lecturers, code)>`	`<!ELEMENT title (#PCDATA)>`
`<!ELEMENT lecturers (lecturer+)>`	`<!ELEMENT place (#PCDATA)>`
`<!ELEMENT lecturer (name)>`	`<!ELEMENT publisher (#PCDATA)>`
`<!ELEMENT name (#PCDATA)>`	`<!ELEMENT name (#PCDATA)>`
	`<!ELEMENT address (#PCDATA)>`

Table 2. DTDs of the input documents

To populate our repository with instances of the previous DTDs, we applied the IBM XML Generator[2]. With this tool we generated 20 document instances of each DTD, so that in this experiment a total of 80 documents constituted the input of our system.

Table 3 presents the average similarity between the types of documents as returned by the system. Obviously, the table shows that the similarity rates between documents of the same DTD are very high. For documents of the DTDs 1 and 2 the returned similarity rates are also high as they share very similar structures. However, the returned similarity rates for any of the previous documents and the instances of the DTDs 3 (incompatible) and 4 (completely different) are null or nearly null.

Finally, the clustering and the schema inference modules were tested with two different similarity thresholds: $\beta_{sim} = \beta_{rep} = 0.7$ and $\beta_{sim} = \beta_{rep} = 0.95$. With the threshold of 0.7, the documents of DTDs 1 and 2 were grouped together, and a new DTD which integrated their schemata was inferred. However, the documents of DTDs 3 and 4 were not combined with the rest, and their inferred schemata were equivalent to those initially used for generating the document instances. In the case of the most restrictive similarity threshold (0.95), the documents of DTDs 1 and 2 were not grouped together and their schemata were separately inferred.

[2] `http://www.alphaworks.ibm.com/`

	DTD 1	DTD 2	DTD 3	DTD 4
DTD 1	1	0.85	0	0.17
DTD 2	0.85	0.98	0	0.08
DTD 3	0	0	1	0.2
DTD 4	0.17	0.08	0.2	0.98

Table 3. Average similarity between the different types of documents

6 Conclusions

In this work we have presented some mechanisms for the automatic generation and upkeep of schemata for XML-oriented document repositories. Our approach assumes that there are no type or schema definitions available for the incoming XML documents, as occurs in many web-based applications. This paper describes two unsupervised algorithms for both, the *Clustering Routine* that detects significant changes in the structure of the repository, and for the *Schema Manager* that keeps the schemata up-to-date as new documents arrives to the system. We have implemented these methods over G, a commercial semi-structured database system.

We are exploring a number of possibilities for future research. First of all, we need to assess the efficiency of the proposed solution; we also need to understand better how the arrival order of the documents affects the performance of the clustering routine. In addition to this, there are some useful applications that can be built on top of the foundations presented in this paper, such as the analysis of the changes in the schemata as they evolve over time. We are also considering the possibility of enhancing the system by means of semantic relationships defined in ontologies.

References

1. W3C Consortium: XQuery 1.0: An XML Query Language. In: `http://www.w3.org/xquery`. (2002)
2. Chamberlin, D., Robie, J., Florescu, D.: Quilt: An XML query language for heterogeneous data sources. In: WebDB 2000. (2000) 53–62
3. Cluet, S., Veltri, P., Vodislav, D.: Views in a large scale XML repository. In: VLDB 2001. (2001) 271–280
4. Mena, E., Illarramendi, A., Kashyap, V., Sheth, A.P.: OBSERVER: An approach for query processing in global information systems based on interoperation across pre-existing ontologies. Distributed and Parallel Databases **8** (2000) 223–271
5. Hélide: The G Web Applications Platform. In: `http://www.helide.com`. (2002)
6. W3C Consortium: XML schema. In: `http://www.w3.org/XML/Schema`. (2002)
7. Zhang, K., Shasha, D.: Simple fast algorithms for the editing distance between trees and related problems. SIAM Journal of Computing **18** (1989) 1245–1262
8. Aramburu, M.J., Berlanga, R.: A temporal object-oriented model for digital libraries of documents. Concurrency: Practice and Experience **13** (2001)