RC 21922 (98572) 28 December 2000 Computer Science

IBM Research Report

XAS: A System for Accessing Componentized, Virtual XML Documents

Ming-Ling Lo, Shyh-Kwei Chen,
Sriram Padmanabhan, Jen-Yao Chung
IBM Research Division
Thomas J. Watson Research Center
Yorktown Heights, NY 10598

XAS: A System for Accessing Componentized, Virtual XML Documents

Ming-Ling Lo, Shyh-Kwei Chen, Sriram Padmanabhan, Jen-Yao Chung
IBM Thomas. J. Watson Research Center
{mllo,skchen,srp,jychung}@us.ibm.com

Abstract

XML is emerging as an important format for describing the schema of documents and data to facilitate integration of applications in a variety of industry domains. An important issue that naturally arises is the requirement to generate, store and access XML documents.

It is important to reuse existing data management systems and repositories for this purpose. In this paper, we describe the *XML Access Server (XAS)*, a general purpose XML based storage and retrieval system which provides the appearance of a large set of XML documents while retaining the data in underlying federated data sources that could be relational, object-oriented, or semi-structured. XAS automatically maps the underlying data into virtual XML components when mappings between DTDs and underlying schemas are established. The components can be presented as XML documents or assembled into larger components. XAS manages the relationship between XML components and the mapping in the form of *Document Composition Logic*. The versatility in its ways to generate XML documents enables XAS to serve a large number of XML components and documents efficiently and expediently.

Index terms: XML, database systems, reuse, heterogeneous data processing, component-based applications, Web applications.

1 INTRODUCTION

The eXtensible Markup Language (XML) [3] has emerged as an important industrial standard and received much attention from the research community. It is increasingly used in e-business, e-commerce, application integration, and meta data management, to name a few applications.

In addition to being a good data representation and exchange format, one key advantage is that its structure and definition are very conducive to software processing. This suggests a future where the Internet carries a very large amount of XML traffic, generated and consumed primarily by software agents, and directed and monitored by human beings occasionally at the high level.

Given such a trend, the key issues include the sources of content of all the XML documents, and the software methodology for managing them. Part of the content will be generated by the new activities and stored in new types of data systems. But as a large amount of data already exists in non-XML formats, schemas and systems, how to efficiently reuse, convert and combine these data into XML components or documents becomes an important challenge.

In this paper, we design a middleware software, called XML Access Server (XAS) that answers this challenge. XAS maps distributed and heterogenous data to and from XML documents. It is designed with two guiding principles: flexibility and ease of use. Flexibility manifests in two aspects. First, as many types of data sources as possible should be allowed to be used as sources for XML components and documents. It requires XAS to handle both different system platforms and different data models. Second, the underlying data should be allowed to map to arbitrary types of XML documents based on different Document Type Definitions (DTDs) [3] in very flexible ways. Ease of use requires that the mapping logic between the underlying data and XML be specified only in high-level description, and that the associated software and data be very easy to deploy. In addition, XAS presents the underlying data as virtual XML components or documents to high-level applications, making them easier to reason about and process.

XAS supports three basic types of operations: *retrieval*, *deposit*, and *query* of virtual XML documents, on top of which higher level functionalities can be built. As the underlying data sources can be distributed and heterogeneous, upon a document retrieval request, XAS retrieves data items from the various data sources and assembles them into XML document(s). Upon a document deposit, XAS disassembles the document into pieces and stores them into appropriate locations in the underlying data sources. Similarly, when an XML query is received, XAS translates the query into a number of low-level queries to the underlying data sources and assembles the low-level answers into the final answer.

The actual data in XAS remains in the underlying data sources in native formats without explicitly converting or copying of data. Old applications written for the underlying data sources continue to run as usual, while new applications built on top of XAS access virtual XML components or documents that consist of logically related data from multiple data sources and are assembled on demand. XAS uses a component-based approach with simple interfaces that interacts well with other software modules to form solutions. Adding new data sources is a straightforward process incurring minor programming efforts and customization costs.

We have implemented a prototype of the XAS system using Java that retrieves and deposits

XML documents. Currently accepted underlying data sources include any JDBC [22] compliant data sources, Lotus Notes databases, and the IBM IMS hierarchical database systems. A snapshot of the prototype has been posted on the IBM alphaWorks Web site [13], and has received 9400 downloads in 8 months.

This paper is organized as follows. Section 2 analyzes the problem space of designing an XML access and management system. Section 3 presents the overall architecture and system design. The new XML component processing model and the detailed XAS functionalities are presented in Section 4. Section 5 describes the XAS mapping framework and the two-stage approach for overcoming the heterogeneity of hardware architectures. Section 6 discusses the related work, and Section 7 concludes this paper.

2 PRELIMINARIES

Middleware systems that map between XML and underlying data sources can be designed in many different ways, based on the following parameters: (1) how they reuse the existing resources, (2) the flexibility of the mapping, and (3) the complexity of required programming effort.

2.1 Reuse of Underlying Resources

The underlying resources can be divided into system, schema and data level resources. One may progressively reuse resources in one or more of these levels.

At the system level, one may reuse the robustness or performance features of the existing systems, such as transaction, recovery, and storage management, and create new schema and data to support XML. In addition, one may also reuse the schema and business logic in the existing systems. For example, an XML document may be decomposed and stored into an existing table in an underlying relational database system. Finally, one may also reuse existing data or application output as the raw materials for generating XML documents.

Although resource reuse at more levels makes better economic sense, it actually puts more constraints and requirements on design and renders it more difficult. However, in many situations reusing existing schema or data may be a user requirement. For instance, new data may continue to be generated by existing programs in relational data format, but need to be presented in XML format. Reusing at the schema and data level is essential for connecting old data and programs to the new XML world. XAS supports reuse at all three levels, with emphasis at the schema and data levels.

2.2 Flexibility of Mapping

Non-XML data can be mapped into XML documents or components in numerous ways. We categorize these solutions into four levels based on the flexibility of the schemes. At the level 0, the meta data of a data source is made available in XML format.

Level-1 solutions focus on one-to-one mapping. Every database or data source has some primitive units of data organization. For example, tables and rows are such units in relational databases, as classes and objects are in object-oriented databases. One may provide algorithms to map such units to XML documents in a fixed way. On many occasions, the XML formats useful to applications may not correspond naturally to any underlying data organization unit. In this case, more sophisticated mapping between underlying data and XML documents is necessary.

Level-2 mapping provides data in formats that are useful to the applications rather than being constrained by the fashion data are organized in the underlying systems. The flexibility of level-2 mapping can be achieved through a combination of one-to-many and many-to-one mappings. In a one-to-many mapping, a piece of underlying data may be mapped to XML documents in multiple ways. In many-to-one mapping, an XML document may consist of data from multiple underlying data organization units. One-to-many mapping is essential for implementing personalization and useful for application integration. Many-to-one mapping on the other hand helps in data integration.

Level-3 mapping is similar to level-2 mapping, but with underlying data coming from distributed and heterogeneous data sources, with different data models. Mapping at this level is the most difficult, and is the focus of XAS.

It should be noted that the purpose of mapping is to perform operations such as retrieval, deposit and query on the mapped XML documents. It is desirable that the mapping scheme be declarative, so that the same mapping description supports both the retrieval and deposit directions of the data flow.

2.3 Programming and Deployment Complexity

Another way to categorize the solutions is according to the complexity in the programming and deployment tasks required by the user. Logically a user should perform three tasks in order to map underlying data to XML documents.

First, as the same data might be mapped to XML in different ways, the users must express their intentions about how the underlying data should be mapped. We call this process the *authoring* *of the mapping logic*. This process can be accomplished through programming, scripting, or, some even simpler processes. If programming is used, there will be an additional compilation process, adding to the complexity of deployment. Regardless of the types of authoring, the description of the mapping logic can be either procedural or declarative in nature, with the latter being easier for the user to manipulate.

Once the mapping logic is authored, it needs to be deployed in the mapping systems. For one solution, each set of mapping logics is embedded in a dedicated set of objects. Therefore, as more mapping logics are authored, more objects are generated, and need to be loaded and linked into the systems making use of them. A different approach is to have an interpretive engine that stores various mapping logic descriptions simply as data. This approach is hard to design and implement because of the complexity of the engine, but easy for end users as little or no deployment effort is required.

Since ease of use is a guidance principle for the XAS design, we provide a high level and declarative approach for the users to author the mapping logic. In fact, the user is only required to annotate existing DTD or XML schema with very simple mapping constructs, since a large portion of the mapping complexity is already captured by the DTD or XML schema.

For deployment of the mapping logic, an interpretive engine is designed. Thus there are no deployment steps required by the users beyond authoring their intentions. XAS can be deployed either as a standalone application, or as a service that can be nicely embedded in a web environment. XAS also has a simplified software interface for the users to add new data sources or to provide their own access methods to currently supported data sources, with limited customization and programming activities.

2.4 Other Schema Level Implications

Both DTD and XML schema are text-based specifications. A DTD or XML schema description is easy to read, create or modify. This may have three implications. First, despite the efforts to standardize DTDs in various industrial sectors, the number of DTD descriptions will continue to grow. Second, in systems that support XML mapping, the ratio between the numbers of XML views and underlying data containers will be much larger than before. Third, users might modify existing DTDs to produce their own versions, which means that a big portion of DTDs might differ with one another only partially. We thus aim at designing a system that supports a large number of frequently evolving virtual XML documents (or XML views) efficiently, with proper version control

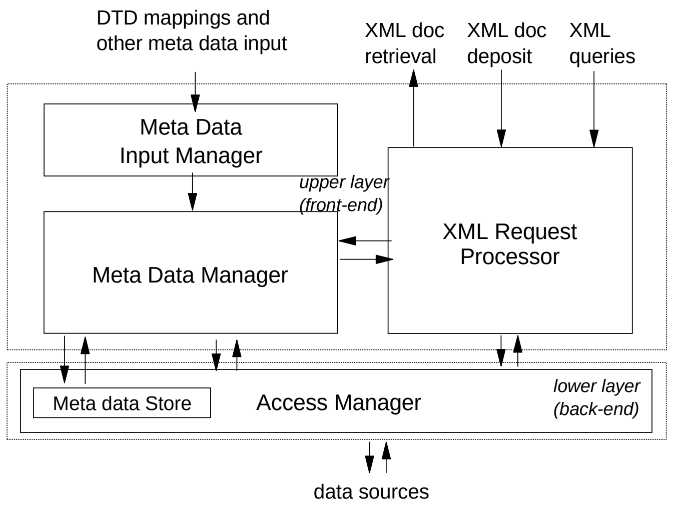


Figure 1: XAS architecture.

and practical sharing of components between XML documents.

3 SYSTEM DESIGN AND ARCHITECTURE

The XAS architecture (see Figure 1) consists of four main modules, a *Meta Data Input Manager*, a *Meta Data Manager* and a *Request Manager* in the upper layer, and an *Access Manager* in the lower layer. It accepts and interprets XML access requests, manages mapping related meta data, and handles other XML specific tasks.

The *Access Manager* implements the lower access layer, which encapsulates the data source heterogeneity and low level data access specifics. It implements the semantics of the various mapping functions so that the upper layer sees a much simplified schema. The interactions with the data sources on various communication, transaction, performance, reliability, and scalability issues are based on the connector and wrapper technologies, distributed systems, and federated databases[5, 18], including JDBC and ODBC for relational data sources [22, 16], CORBA for object technologies [23], and Component Broker for accessing enterprise data sources [11].

The *Meta Data Input Manager* accepts and processes two types of meta data. The mapping specific meta data describes how mapping can be accomplished between DTD and the underlying schema. Data source specific meta data describes which data sources, containers and data can be used, and various connection related information. The meta data input manager can interact with tools to facilitate meta data creation and input.

The *Meta Data Manager* organizes and manages the input meta data. Besides data source specific information, it maintains data structures representing the mapping logic, which can be used by the request manager to efficiently answer user requests.

The *Request Manager* processes the XML retrieval, deposit and query requests from users. It is responsible for consulting the meta data manager and translating each XML request into a set of level access requests for the access manager to process. It then assembles the results from the access manage into the answer of the user request. It may also access the meta data to generate a better processing strategy.

4 COMPONENT-BASED VIRTUAL XML DOCUMENTS

In general, reuse of the underlying schema design is the key to avoid explicit conversion of existing data into XML format. It not only saves time and space, but also alleviates the potential data consistency problem. There are four basic elements for managing virtual XML documents: the XML schema or DTD of the documents, the underlying schema supporting the document, the mapping logic between the two sets of schemas, and the data in underlying schema. The processes to build a component-based virtual XML document system include (1) identifying or creating the underlying and XML schemas, (2) creating the mapping logic, and (3) either reusing existing data in the underlying schema or creating new data.

The methods described in the paper are well applicable to both DTD and XML schema, but for the ease of discussion we refer to DTD only. We use the term *established in the XAS system* to denote that a given DTD is associated with a particular set of underlying schemas, and has a mapping logic constructed between them.

4.1 Identifying and Creating Schemas

In practical XML applications the DTDs are often given as requirements, either because of some agreements with trading partners or the need to conform to industrial standards. If the underlying data sources have been engaged in similar applications over time, most likely there would already be a set of native schema that serves similar purpose. In the enterprise environment it is often the case. For example, we may be given a DTD for invoices, while an accounting schema in an underlying relational database may already contain a set of invoice related tables. We may build a mapping logic between the given DTD and the underlying tables to reuse the schema and the data.

However, it is also possible that the XML applications have no counterpart in the non-XML domains. In such cases, no underlying schemas are available for reuse and underlying schema constructions are necessary. For example, when given a new DTD, we may be able to identify part of an underlying schema for reuse, while creating some additional underlying schemas is required for generating complete XML documents. The task is to build a set of underlying schemas that can support XML access most efficiently.

In some applications, we are given a set of schemas and data in certain underlying data sources which we would like to expose as XML data. There is no restriction in terms of the DTDs through which the underlying data can be exposed. Potentially a set of underlying schemas can be mapped to DTDs in numerous ways. To enable the users to build DTDs that are suitable for their applications is the main topic of *data source publication*.

Instead of working with underlying schemas directly, we may work with components of the created mapping logics. That is, we can reuse existing mapping logics to construct new mapping logics. Such method is called *componentized XML construction*. The mapping logic created during data source publication may also be used in componentized XML construction.

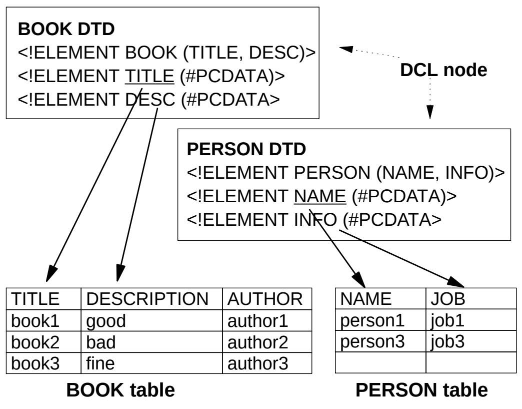
4.2 Creating Mapping Logic

Mapping logic plays an important role in reusing existing underlying schema and data, and in building the virtual XML documents. Once a mapping logic is created for a DTD, in principle, retrieval, deposit and query of XML documents are all performed using the same mapping description.

4.2.1 Mapping Logic and Virtual XML Components

Consider the two relational tables `BOOK` and `PERSON` in a relational database and the two corresponding DTDs (see Figure 2). We can create a mapping from the `TITLE` and `DESCRIPTION` columns of the `BOOK` table to the `BOOK` DTD, as shown by the four solid arrows. As soon as the mapping is created, each row in the `BOOK` table is automatically available as a `BOOK` XML component. These `BOOK` XML components are "virtual" because they are not stored in the system explicitly, but are generated only on demand. Similarly we can create person XML components through the mapping between the `PERSON` DTD and the `PERSON` table.

The key to the availability of these components is the ability to name or address them. The basic mechanism for naming virtual XML documents in XAS is through key values or object identities. When the mapping is established, one or a small number of elements are chosen as the keys of the documents of the DTD. The chosen elements map to some primary key values in the underlying


BOOK DTD
<!ELEMENT BOOK (TITLE, DESC)>
<!ELEMENT TITLE (#PCDATA)>
<!ELEMENT DESC (#PCDATA>
DCL node
PERSON DTD
<!ELEMENT PERSON (NAME, INFO)>
<!ELEMENT NAME (#PCDATA)>
<!ELEMENT INFO (#PCDATA>
BOOK table
PERSON table

TITLE	DESCRIPTION	AUTHOR
book1	good	author1
book2	bad	author2
book3	fine	author3

NAME	JOB
person1	job1
person3	job3

Figure 2: Mapping between the BOOK and PERSON DTDs and the BOOK and PERSON tables.

data sources. The virtual components are then addressed by the key values. In the example, the key of the BOOK DTD is TITLE, and we can ask for books with titles book1, and book2 etc.

Besides the basic key value naming, URLs or document names can also be assigned to documents or components. XAS maintains the mapping between the high level names and the key values. The key value mechanism is really a simplified query mechanism that XAS can support for identifying and accessing XML components and documents.

4.2.2 Document Composition Logic

The *Document Composition Logic (DCL)* is the logic that describes how the document is recursively composed of its child components and how the child components relate to one another. Each XML document or component in XAS can be viewed as a DCL definition recursively linking a set of child components.

There are two types of DCL. *Preexisting DCL* is implied in the relationships between data that has already existed in the underlying data sources, and is identified or discovered in the schema mapping process. In the previous example, the process of mapping the TITLE and DESCRIPTION columns of the BOOK table to the BOOK DTD actually involved identifying the relationship between the two columns as a preexisting DCL. Thus, the DCL that connects TITLE and DESCRIPTION with each BOOK component is preexisting, as is the DCL that links NAME and JOB with PERSON.

Preexisting DCLs is often due to the structure of schema or the foreign key relationships in the underlying data sources. For example, a preexisting DCL exists between two fields of a same relational row, or between two rows connected by a foreign key relationship. Preexisting DCLs may also exist in less obvious forms such as semantically related attribute or meta data values between the underlying data. Automatic identification or discovery of preexisting DCLs may become a useful research area in the future.

The other type of DCL is *newly created DCL*, which is supplied to XAS as part of the creation process of virtual XML documents, and connects previously unrelated components to form new logical entities. Consider the following book list DTD as an example:

```
<!ELEMENT BOOKLIST (PERSON, BOOKS)>
<!ELEMENT PERSON (NAME, INFO)>
<!ELEMENT NAME (#PCDATA)>
<!ELEMENT INFO (#PCDATA)>
<!ELEMENT BOOKS (BOOK)+>
<!ELEMENT BOOK (TITLE, DESC?)>
<!ELEMENT TITLE (#PCDATA)>
<!ELEMENT DESC (#PCDATA)>
```

We want to generate a book list XML that consists of a person's information followed by a list of books favored by the person. Although the information about people and books are stored in the `BOOK` and `PERSON` tables, no information exists in the database or any other sources about people's favorite book lists. To join these two tables, we need to create two new DCLs that represent the `BOOKLIST` and `BOOKS` components. The whole book list document then consists recursively of newly created and preexisting DCLs connecting various XML components together, with newly created DCLs at the higher levels, and preexisting DCLs at the lower levels.

Physically DCL is represented in XAS by *nodes*. A DCL node is a schema level entity that logically represents a set of XML components of certain type. It links to underlying schema or other DCL nodes which represent the child components for its instances. The implementation of preexisting DCL involves recording the corresponding linkages in the DCL node. The implementation of newly created DCL may require creation of new tables. These tables will contain the specific data used by the DCL nodes to associate previously unrelated components. For convenience, we call them *DCL tables*. Figure 3 describes the DCL nodes for the document type `BOOKLIST`. In this example, we must create two new DCL tables, `BOOKLIST` and `BOOKS`, for the newly created DCLs.

4.2.3 Components and Documents Deposit

After the DCL nodes and the schema for newly created DCL are set up, there are two ways to create new XML documents. The first is by inserting directly into the DCL tables. In the example, we can create the book list document for `person1` by inserting new rows into the `BOOKLIST` and `BOOKS` tables (see Figure 3). In this figure, preexisting tables and the DCL nodes for preexisting DCLs are shaded in white, while the DCL tables and the DCL nodes for newly created DCL are shaded in grey. As a result, previously unrelated `person1`, `book1` and `book2` are now connected as a new book list documents. This method is a very efficient way to create XML documents since we need only specify the new DCLs and not repeat the detailed descriptions of the child components. The newly created DCLs may later be reused as preexisting DCLs.

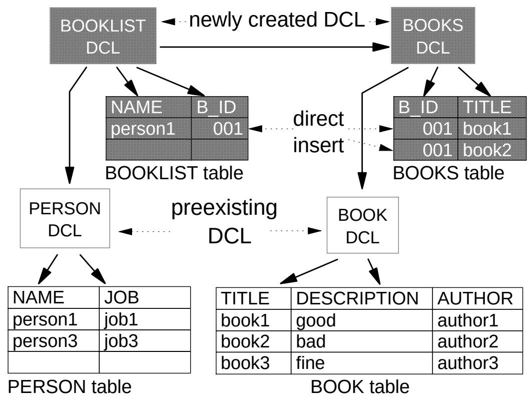


Figure 3: DCL nodes and the underlying tables for new and preexisting DCL.

XML data can also be populated by simply depositing XML documents into XAS. Although this scenario appears to be simple, there are several interesting issues. Consider again the previous `BOOKLIST` DTD. Suppose a new `BOOKLIST` document arrives after the DTD is established:

```
<BOOKLIST>
<PERSON>
  <NAME>person2</NAME>
  <INFO>job2</INFO>
</PERSON>
<BOOKS>
  <BOOK><TITLE>book4</TITLE></BOOK>
  <BOOK><TITLE>book5</TITLE></BOOK>
</BOOKS>
</BOOKLIST>
```

XAS breaks up the document into the `person2`, `book4` and `book5` components, inserts them into their respective tables, and records the DCL of the new document in the DCL tables. Now suppose another document arrives:

```
<BOOKLIST>
<PERSON>
  <NAME>person3</NAME>
  <INFO>job3</INFO>
</PERSON>
<BOOKS>
  <BOOK><TITLE>book2</TITLE><DESC>2nd</DESC>
  </BOOK>
  <BOOK><TITLE>book3</TITLE><DESC>3rd</DESC>
  </BOOK>
</BOOKS>
</BOOKLIST>
```

After breaking up the document, XAS recognizes that `person3`, `book2` and `book3` already exist in the `PERSON` and `BOOK` tables, and does not store the components redundantly. Its only action is to record the new DCL in the DCL tables. The result of depositing this document is the same as directly inserting into the DCL tables in the first place. If a new document contains both new and old components, XAS stores only the components that do not exist in the underlying data sources.

The advantages of XAS's component-based approach go beyond simply saving storage space. More importantly, business logic can be triggered as a result of component deposits. For instance, we can increment a reference count when another copy of the same component arrives, which can later be used for analysis. We can also check for consistency between copies of the same data when a redundant component arrives, and may reject the deposit, resort to the master copy, or request human intervention when inconsistency is detected. With this capability, all input XML documents are in a sense integrated, and are no longer isolated, monolithic copies of data with potential inconsistency. It is also much easier to perform analysis across documents by accessing the document components directly. Documents deposited this way do not lose their identities. They can later be retrieved as if each of them is kept in a separate file.

4.3 Populating Data

At the data level, XAS supports both new and old, XML and non-XML data in its underlying data source containers. The underlying data sources may also be XML-based systems. It requires the ability to map from one set of DTDs to another. With this ability, the XAS can leverage the XML data collected and managed by other XML systems, including another installation of XAS. The overall design of XAS enables a great deal of flexibility in data integration.

Virtual XML documents in XAS may come into existence in two ways. Existing underlying data may become virtual documents when new mapping logics are created. New XML documents can be entered into the system through the deposit process. When a mapping between a DTD and the underlying schema is established, all data already stored in the underlying schema automatically becomes virtual XML components in XAS.

4.4 Detailed Scenario

Consider the example in Figure 4. A DTD whose internal representation includes three DCL nodes, `parent`, `child1` and `child2`, has been established in XAS. There are three tables in the underlying data source corresponding to the three DCL nodes. The `parent` DCL node and table contain the high level DCL of this document. Two applications, `app1` and `app2`, are directly accessing the `parent` and `child2` tables, respectively. The following steps occur in this figure:

- Initially many data items have already existed in the tables before any XML documents are deposited, as shown by the lettered square blocks in each table. A virtual XML document was created as the result of mapping the underlying tables to the DTD. This document is represented by the square blocks labeled `a0`, `a` and `0` in the three tables. The rest of the

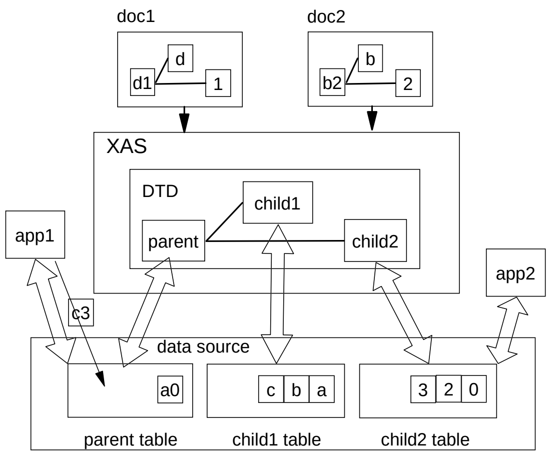


Figure 4: An example of the relationships between established DTD, underlying data containers, input XML documents and existing applications.

blocks in the `child1` and `child2` tables do not constitute virtual documents as they lack counterparts in the `parent` table.

- Two documents of the established DTD are about to be deposited into XAS. The input of `doc1` results in a brand new virtual XML document, with new data entering the underlying `parent`, `child1` and `child2` tables. The input of `doc2`, on the other hand, only causes a piece of newly created DCL `b2` to be recorded in the `parent` table. This new DCL causes two previously unrelated child components `b` and `2` to be connected in a virtual document.
- Application `app1` enters a piece of newly created DCL `c3` into the `parent` table, which results in a new virtual XML document.

Figure 5 shows the status of the system and the tables after the two XML documents are deposited and the input from `app1` is completed. Note that there is no redundant storage for the components of `doc2`. There are now four virtual XML documents in the system. `doc0` came into existence when the tables were mapped to the DTD. `doc1` and `doc2` are the results of the previous document deposits. `doc3` is the result of direct input from application `app1`. Regardless of how it is generated, each document can be accessed in the same way.

5 XAS MAPPING FRAMEWORK

As the main goal of XAS is to map distributed and heterogenous data into XML, XAS needs to deal with data not only coming from different systems, but also belonging to different types of data models. For example, some underlying data may be relational, some may be object-oriented, while yet some others might belong to new systems and data models.

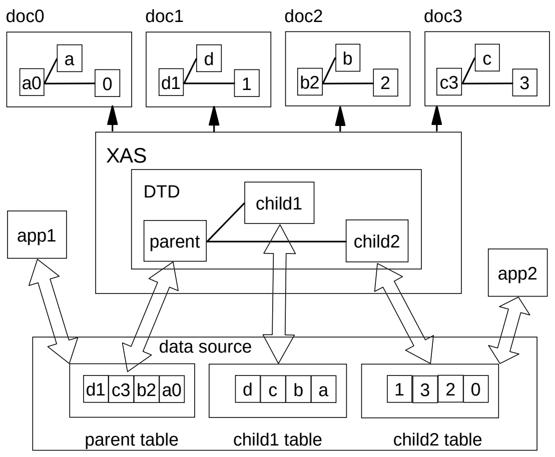


Figure 5: The configuration of XAS after the inputs in previous figure are completed.

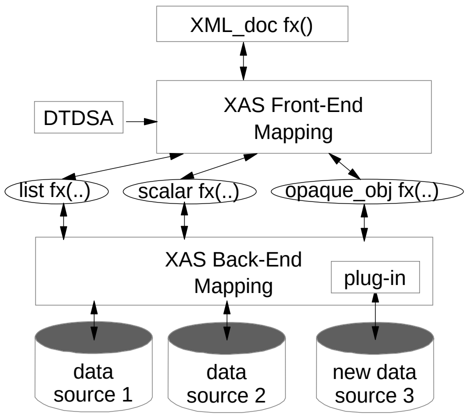


Figure 6: XAS mapping architecture.

The center of our mapping framework is a declarative mapping mechanism *DTD with Source Annotation (DTDSA)*. The DTDSA contains *mapping functions* that can return scalar values, opaque objects, and lists of either scalars or opaque objects. More precisely, it contains functions that take as input, and return, only such scalars, objects, and lists. Opaque objects are named because our framework need not care about its internal structure or semantics.

Our solution for overcoming the heterogeneity of hardware architectures includes a front-end stage and a back-end stage that correspond to the upper layer and the lower layer respectively in Figure 1. The front-end stage assumes a simple data model in which there are only lists, opaque objects and scalars. The task at this stage is to map from these items to XML documents. The back-end stage then focuses on the conceptually simpler task of mapping arbitrary data sources to scalars, lists, and opaque objects, thus encapsulating all heterogeneity mapping.

Figure 6 illustrates the mapping framework.

5.1 DTD with Source Annotation

The proposed DTDSA framework serves as a mapping mechanism that connects XML documents and underlying data sources. Given a DTD, we can obtain an XML document if all of the following questions are answered:

Number of occurrence: For each repetition symbol in the DTD ("*", "+" and "?"), the number of times the qualified DTD construct repeats.

Choice: For every choice declaration (choice list, terminal choice list and enumeration type), the chosen alternative.

Data content: For every value declaration (e.g., `#PCDATA`, `CDATA`, etc.), the assigned content.

We introduce two simple types of *mapping constructs* to express the missing information. *Value specifications* answer the "data content" and "choice" questions, and *binding specifications* answer the "number of occurrence" questions and bind values to the input parameters of other mapping functions. The mapping construct must associate with, or annotate, a particular DTD construct.

A DTDSA can represent a set of XML documents when there are undefined parameters associated with some mapping functions. An easy way to understand DTDSA is to consider during the retrieval process every annotated DTD construct as a function that takes zero or more arguments as input, calls its child DTD constructs recursively as subroutines, and returns a string. The root element of a DTDSA is also a DTD construct, and therefore is also a function, except that it takes as input all of the undefined parameters, and returns a valid XML document.

5.1.1 Value Specifications

A value specification annotates either a value or choice declaration. It assumes the syntax of a ':' symbol followed by a scalar function, and appears in DTDSA as follows:

$\boxed{\text{VCD} :sf}$

where `VCD` is a value or choice declaration, and sf a scalar-valued mapping function, called the *value function*.

If `VCD` is a value declaration, such as a `#PCDATA` or a `CDATA`, an invocation of VCD simply returns the value of sf. If `VCD` is a choice declaration, the value returned is determined as follows. If the value produced by `sf` is an integer `i`, with `i` between `1` and `n`, where `n` is the number of alternatives in the choice declaration, the ith alternative will be returned. If the value produced by `sf` is a string `Cj`, the alternative matching the string will be returned. Otherwise the returned value is undefined.

5.1.2 Binding Specifications

One or more binding specifications can annotate any DTD construct that is not a value or choice declaration. If a DTD construct contains an ending repetition symbol, i.e. '+', '*', or '?', annotation by binding specifications is mandatory. When more than one binding specifications annotate the same DTD construct, their order is significant. The syntax of a binding specification is a "::", followed by a variable, a ":=", and a list-valued mapping function. It appears in DTDSA as follows:

$\boxed{\text{DC} :: x_1 := vf_1 :: x_2 := vf_2 \ldots :: x_n := vf_n}$

where DC is a DTD construct which is not a value or choice declaration, x_i a variable and vf_i a binding function, for $i = 1, \ldots, n$. x_i is called a *binding variable*, and vf_i a *binding function.*

The basic semantics of a binding specification is that the list of values returned by the binding function is bound to the binding variable *in turn.* Binding specifications serve two purposes. When a binding specification is the closest mapping construct annotating a DTD construct with an ending repetition symbol, it determines the number of times the DTD construct is instantiated.

Binding specifications also determine how values are passed to the arguments of various mapping functions. Once a value is bound to a binding variable at certain DTD construct, the binding is visible to all instances of the DTD constructs called by the first DTD construct, unless the binding is overwritten by another binding with the same binding variable name.

Consider the following DTDSA:

```
<!ELEMENT A (B, C) ::x:=i1 ::y:=i2>
<!ELEMENT B (#PCDATA :y)>
<!ELEMENT C (D)* ::z:=intseq(x)>
<!ELEMENT D (#PCDATA :z)>
```

where the function intseq(x) produces a sequence of integers from 1 up to x. Assuming the input (or undefined) arguments to this DTDSA are i1=3 and i2=5, y will be bound to value 5, x to value 3, and z to 1, 2 and 3 in turn. The output virtual XML document is:

```
<A>
<B>5</B>
<C><D>1</D><D>2</D><D>3</D></C>
</A>
```

The detailed definition of the DTDSA is beyond the scope of this paper, and is described in [14].

5.2 Front-end Stage

The front-end processes the user requests and interpretes the mapping logic from the mapping functions in the DTDSA. This stage associates the lists, scalars, and opaque objects, returned after evaluating the mapping functions in the back-end stage, with the DTD constructs. It determines the evaluation sequence of the DTD constructs based on the results of calling mapping functions.

For example, in the retrieval process, XAS recursively instantiates the DTD constructs from the root elements using the DCL graph and user provided input values to construct the XML document. The results of mapping functions are bound to individual DCL constructs, and include scalars, opaque objects and lists returned from the back-end stage.

There may be more than one ways to implement mapping functions on top of a given data source. Instead of dictating the best way, XAS provides an option for the integrators of the data source to design their own access methods. Once the mapping functions are implemented, the front-end treats mapping functions with identical names and parameter types equally for all of the data sources that implement the functions. Users are free to mix mapping functions based on multiple data sources in a single DTDSA. Adding new data sources with new mapping functions involve providing new back-end access method implementations, and including the new mapping functions in the DTDSA.

5.3 Back-end Stage

The task at the back-end is to implement the mapping functions that map underlying data of arbitrary data model into the lists, scalars and opaque objects and interact with the front-end. The mapping functions can be simple arithmetic functions, as in the previous examples. However the real power lies in the functions implemented based on data in the underlying data sources. The motivation for using lists, scalars and opaque objects as the output of the mapping functions is to make back-end mapping conceptually straightforward.

Theoretically any type of data sources can serve as an underlying data source for XAS, such as relational, object-oriented, hierarchical, semi-structured databases, spreadsheets, flat files, or even other XML documents. The same mapping functions for different types of data sources differ only in back-end implementations, which effectively encapsulates the heterogeneity of the data sources.

As an example, consider a relational database. An SQL statement can be implemented as a mapping function that returns a list of rows, each of which is considered an opaque object in DTDSA to serve as binding functions. We can further provide a function that takes a row (an opaque object) and a column name and returns the value of the column, to serve as a value function.

6 RELATED WORK

Many relational and object-oriented database vendors are providing support for storing and retrieving XML documents [17, 12, 20]. The support tends to be monolithic (entire XML documents are stored and retrieved) and does not enable discovery and reuse of existing data in the repositories.

Our work is related to earlier efforts such as TSIMMIS [19, 8], Garlic [5, 21], DISCO [24] and Infomaster [9], which are based on mediators and wrappers to access data from various sources. Many of these projects are adapting to the use of XML schemas. The Virtual Database technology approach foresaw the use of XML for representing document schemas and enabling standard query mechanisms [10]. We believe that the distinction of XAS is its focus on XML schemas and XML query mechanisms from the outset. We are reusing the past work in terms of access methods and capabilities of underlying data sources. However, XAS addresses the challenging issues of sub-schema reuse as well as component and schema relationship discoveries.

There have been a number of papers discussing semi-structured data models [4, 1, 2] for integration of data from multiple data sources. Graph data structures such as ODMG's OEM have been proposed in the past for depicting and maintaining the schema of semi-structured data sources [15, 19, 7]. In contrast, XML is a non-typed tree data model for representing data schemas and hence provides more flexibility in associating with the base data types. The flexibility requires to be managed carefully and it is one of the requirements on the DCL mechanisms of XAS. Also, the DCL mechanism extends and tries to build larger schemas from the XML components when possible.

Finally, several papers are investigating the problem of discovering schema of semistructured data using data mining techniques [25, 6]. In [6], the authors present a technique that uses data mining and generates an RDBMS schema mapping. XAS focuses on the complementary problem. It requires the ability to understand existing schema and map it to new XML schemas as well as to map components of XML schemas to common underlying schemas to facilitate reuse.

7 CONCLUSION

We presented the XML Access Server (XAS) in this paper. By serving componentized, virtual XML documents, XAS provides to its applications the appear of a large set of XML data while retaining the real data inside the underlying data sources. XAS maps the underlying data into XML components, which can in turn be assembled into larger components. The mapping and assembly can be performed in an infinite number of ways. This component-based approach not only facilitates the efficient creation and access of XML documents but also enables the higher level analysis and knowledge mining activities.

XAS generates virtual XML documents in a variety of ways: by establishing mappings, by assembling previously unrelated XML components, and through regular document deposits. This versatility is the key to XAS's ability to serve a large number of XML documents efficiently and expediently.

We have implemented the XAS prototype in Java. The current prototype uses relational database systems as the underlying data sources and JDBC as the low level access technology. It can connect to IBM Universal Database Servers on the Windows, UNIX, and System 390 platforms and Microsoft Access on the Windows, and retrieve and deposit XML documents based on the schemas and data in these systems. A snapshot of XAS is posted on the IBM alphaWorks Web site under the name *XML Lightweight Extractor* (XLE) [13], which has been widely tested and used.

Future works include enhancing the capabilities of XAS along three directions. The first is to enable XAS to use more types of access technologies and data sources. In particular, we are working toward better integration with data sources that exposes blocks or streams of XML data. This capability would, among others, enable one XAS to serve as a data source of another and allow XAS to take as data sources the numerous new applications that output XML data. We also regard using non-relational enterprise data sources a high priority in supporting e-commerce applications, given the high availability and reliability of these data sources and a very large amount of data already stored in them. The second direction is XML queries. We believe that the internal mechanisms for supporting XML queries are similar and focus on the mechanisms at the current stage. As described earlier, the key value mechanism for addressing virtual XML components or documents is in fact simplified XML queries. A full query support will enable XAS with an even more flexible access capability. Internally, query support involves mapping incoming queries into sub-queries for the data sources and optimizing the query execution plans in the XAS context. The third direction is tool support for establishing DTDs in XAS, as described in Section 4.1, and for other mapping and component management tasks.

References

[1] S. Abiteboul. Querying semi-structured data. In *Proc. International Conference on Database Theory*, pages 1–18, 1997.

[2] C. Beeri and T. Milo. Schemas for integration and translation of structured and semi- structured data. In *Proc. International Conference on Database Theory*, pages 296–313, 1999.

[3] T. Bray et al. Extensible Markup Language (XML) 1.0. W3C Recommendation. `http://www.w3.org/TR`.

[4] P. Buneman. Semistructured data. In *Proc. 16th ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*, pages 117–121, 1997.

[5] M. Carey et al. Towards Heterogeneous Multimedia Information Systems: The Garlic Approach. In *Proc. the Fifth International Workshop on Research Issues in Data Engineering (RIDE): Distributed Object Management*, 1995.

[6] A. Deutsch, M. Fernandez, and D. Suciu. Storing semistructured data with STORED. In *Proc. ACM SIGMOD International Conf. Management of Data*, pages 431–442, 1999.

[7] M. Fernandez, D. Florescu, J. Kang, A. Levy, and D. Suciu. Catching the boat with Strudel: experience with a web-site management system. In *Proc. ACM SIGMOD International Conf. Management of Data*, pages 414–425, 1998.

[8] H. Garcia-Molina et al. The TSIMMIS approach to Mediation: Data Models and Languages. *Journal of Intelligent Information Systems*, 8(2):117–132, 1997.

[9] M. Genesereth, A. Keller, and O. Duschka. InfoMaster: An Information Integration System. In *Proc. of ACM SIGMOD Conference*, pages 539–542, 1997.

[10] A. Gupta, V. Harinarayanan, and A. Rajaraman. Virtual Database Technology. *SIGMOD Record*, 26(4):57–61, 1997.

[11] IBM Corp. Component Broker. `http://www.software.ibm.com/ad/cb/`.

[12] IBM Corp. DeveloperWorks: XML zone. `http://www.ibm.com/developer/xml`.

[13] IBM Corp. XML Lightweight Extractor (XLE). `http://www.alphaworks.ibm.com`.

[14] M.-L. Lo, S.-K. Chen, and S. Padmanabhan. Supporting XML views on distributed and heterogeneous data sources. *Tech. rep., IBM T. J. Watson Research Center*, August 2000.

[15] J. McHugh, S. Abiteboul, R. Goldman, D. Quass, and J. Widom. Lore: a database management system for semistructured data. In *Proc. ACM SIGMOD International Conf. Management of Data*, pages 54–66, 1997.

[16] Microsoft Corp. Open Database Connectivity. `http://www.microsoft.com/data/odbc/`.

[17] Oracle Corp. Oracle8i. `http: //www.oracle.com/database/oracle8i`.

[18] M. Ozsu and P. Valduriez. *The Computer Science and Engineering Handbook*, chapter Distributed and Parallel Database Systems. CRC Press, 1997.

[19] Y. Papakonstantinou, H. Garcia-Molina, and J. Widom. Object exchange across heterogeneous information sources. In *Proc. International Conference on Data Engineering*, pages 251–260, 1995.

[20] Poet Software Corp. `http://www.poet.com/ products/cms/xml_library`.

[21] M. Roth et al. The Garlic project. In *Proc. of ACM SIGMOD Conf.*, 1996.

[22] Sun Microsystems. JDBC Data Access API. `http://www.javasoft.com/products /jdbc/`.

[23] The Object Management Group. Corba. `http://www.corba.org/`.

[24] A. Tomasic, L. Raschid, and P. Valduriez. Scaling Heterogeneous Databases and the design of Disco. In *Proc. of Intl. Conf. on Distributed Computing Systems*, pages 449–457, 1996.

[25] K. Wang and H. Liu. Discovering Typical Structures of Documents: A Road Map. In *Proc. of ACM SIGIR Conf. on R& D in IR*, 1998.