

Bridging XML-Schema and relational databases. A system for generating and manipulating relational databases using valid XML documents.

Iraklis Varlamis
Dept of Informatics,
Athens University of Economics & Business,
Patision 76, 10434,
Athens, HELLAS
varlamis@aueb.gr

Michalis Vazirgiannis
Dept of Informatics,
Athens University of Economics & Business,
Patision 76, 10434,
Athens, HELLAS
mvazirg@aueb.gr

ABSTRACT

Many organizations and enterprises establish distributed working environments, where different users need to exchange information based on a common model. XML is widely used to facilitate this information exchange. The extensibility of XML allows the creation of generic models that integrate data from different sources. For these tasks, several applications are used to import and export information in XML format from the data repositories. In order to support this process for relational repositories we developed the X-Database system. The base of this system is an XML-Schema file that describes the logical model of interchanged information. Initially, the system analyses the syntax of the XML-Schema file and generates the relational database. Then it handles the decomposition of valid XML files according to that Schema and the composition of XML documents from the information in the database. Finally the system offers a flexible mechanism for modifying and querying database contents using only valid XML documents, which are validated over the XML-Schema file's rules.

Keywords

Metadata, XML, Relational Databases, Querying, Document Storage and Retrieval.

1. INTRODUCTION

XML was initially designed for the exchange of information electronic "documents". Rapidly, enterprises begin to adopt the new model for information exchange and to develop applications that support XML. Assorted by a set of supportive technologies for data presentation, transformation, querying and validation,

XML is becoming the standard format for data exchange among distributed applications components or co-operating applications. The use of XML for information interchange among different enterprises and organizations evokes the need for a common schema that the information must follow.

The most widely supported technologies for describing the schema of XML documents and validating their contents, are Document Type Definitions (DTDs) and XML-Schema [1]. Domain experts aim to precisely define the rules that must be followed by the exchanged documents, creating a communication channel among co-operating enterprises and organizations.

With the use of XML, communication and information exchange can be established regardless of the underlying storage platform. However, the different applications that communicate using XML have to transform XML to the underlying information model, which is usually a relational DBMS. These applications contain mechanisms for exporting data from the database and generating XML documents as well as for storing XML documents in the databases. As stated in [2], such mechanisms must be *general*, *dynamic* and *efficient*. The application that satisfies these criteria will support a wide range of information interchange tasks.

In the video industry, for example, companies possess a huge amount of digitized video information and meta-information, which must be made available to their customers and collaborators. Video information usually includes programs, movies, documentaries, advertisements, video clips etc. This information is stored, in digital format, in a video database and is decomposed in scenes and plans. Meta information refers to the contributors, the technical specifications, the existing copies of a program and their maintenance state, the concept and content of an audiovisual program, and is equally important to the digitized audiovisual program. MPEG-7, the new format for video meta-information is developed to describe such information and certain efforts have been made to create an XML equivalent of it. [3]

A schema describes the structure of an XML document. It indicates which elements appear in the document and which sub-elements, attributes, and relations are allowed within each element. Authors can invent their own schemata and share them with other authors or readers. The schema can be used to validate the structure of the XML document automatically and also to

decompose an XML document to the pieces of information it comprises of.

Co-operative, distributed systems are developed that control the digitization of audiovisual information, the extraction of content features, the recording of additional information, and the information retrieval and re-synthesis tasks [4]. The various components of such systems need a common schema for the exchanged XML information.

The purpose of this paper is to present the X-Database system that utilizes the structure of the XML documents, as it is described in XML-Schema:

- to generate a relational database schema
- to store XML documents' information in the database
- to create XML documents from the database contents
- to update and query the contents of the database using exclusively XML documents

X-Database has been developed to support a co-operative video annotation, storage and query process, and tested with video metadata. However, its generic design enables the generation and manipulation of a relation database according to any XML information schema. The information transferred among the various co-operating processes is modeled using XML documents that comply with the strict rules expressed in XML-Schema.

In order to demonstrate the architecture and mechanisms employed by our system, we use the video information paradigm throughout the paper. The same example is used to illustrate the advanced capabilities of XML-Schema in describing the XML documents' structure and its representation in a relational database schema. The process of inserting, deleting, modifying and querying relational data using XML documents is also demonstrated for the video metadata case.

In the following section the information flow of the whole annotation and querying system is depicted and the system architecture is presented. Section 3 presents the various entities and structures of the XML-Schema file used in this system. In Section 4 a detailed description of the X-Database module, is performed and certain issues that were confronted during the database design are illustrated. Section 5 presents related work on the area of XML storage and retrieval from relational DBMSs, attempting a comparison with commercial DBMSs tools and other relevant research work. In section 6, some experimental results are demonstrated and the final conclusions are drawn in section 7.

2. The implemented information system

2.1 Information flow

The video information annotation and querying system is presented in Figure 1. The information flow among the various system components is described in the following.

- 1) The administrator of the Video-Information Model, analyses the information to be stored in the database. The analysis process results to an XML-Schema file. This file is used as a validating mechanism for the XML documents that will be created.
- 2) The X-Database module receives the XML-Schema file and automatically generates the database.

- 3) The annotator creates XML documents containing information about audiovisual contents.

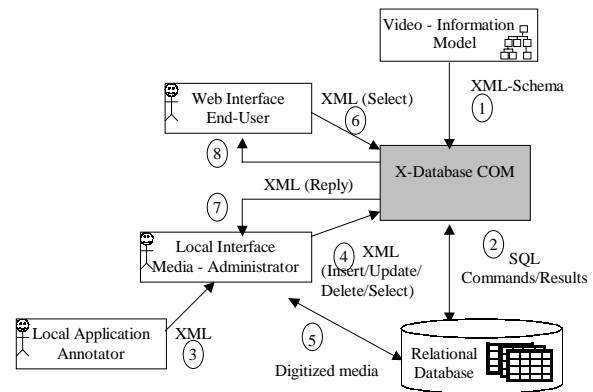


Figure 1. Information Flow

- 4, 5) The media administrator matches the digitized program descriptions with real media or media copies and provides all the media information related to each audiovisual object. The administrator can insert, update, delete and retrieve information from the database (step 4) and store the digitized video files into the database (step 5)

6) The web-users retrieve information from the database. User queries are translated to the appropriate XML documents before being sent to the X-Database module.

7,8) The X-Database module receives XML documents created during steps 4 and 6 and converts them to SQL commands. It also retrieves the query results and constructs the XML-Reply documents, which are sent either to the media administrator (step 7) or to the web-users (step 8).

2.2 System architecture

The generic system architecture is presented in the following figure (). The included components are described in the following paragraphs.

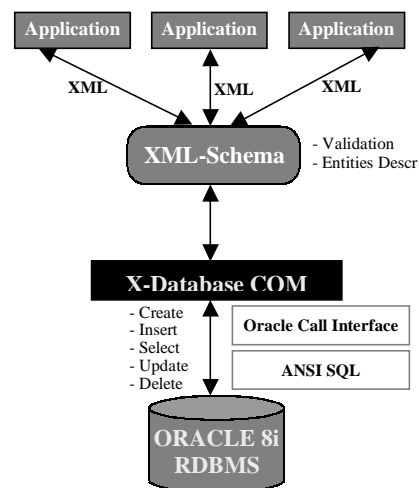


Figure 2. System Architecture

2.2.1 The database

The system was developed over the Oracle 8i RDBMS and the Oracle Call Interface was used to access the database. However, in an effort to keep the system generic, we addressed only ANSI SQL commands to the database so that it works with any relational DBMS. The schema of the database is not predefined. All the tables and relationships are created by the X-Database system according to the XML-Schema structure.

2.2.2 Applications

The system users are divided into two categories: professional users, such as video editors, archivists and producers, who access the audiovisual information in a local environment and non-professional users who search the information using a web based application.

A web application, implemented using ActiveX components, allows users to compose their queries. These queries are converted into proper XML documents and sent to the X-Database module. The retrieved information is sent back to the application, as an XML document and is presented to the users.

A local application developed in C++ enables the media administrators to modify the database contents. Video annotators use an additional application to describe video contents and produce XML documents, which are then forwarded to the media administrators.

The web application performs only *select* statements to the database while the media administrator's application can perform *insert*, *update*, *delete*, *select* and in special cases *create* statements to the database. In addition to this, several types of connection privileges have been created to establish different access levels to the information stored in the database for web and local users.

2.2.3 Database Interface

All applications communicate with each other and the database using XML documents. This communication is supported by an extra application, the X-Database, which acts as interface between applications and database. This architecture will allow future applications to interact with the database through the same interface, without knowing anything about the database structure. The X-Database module is implemented as a COM object developed in C++ and can be accessed both by web based and local applications.

3. The X-Database module

3.1 Advantages from the use of XML-Schema language

The implemented data model is based on the directives of MPEG-7 [5] format for video metadata and is expressed using the emerging XML-Schema notation.

The use of XML-Schema induces many advantages to the system:

- It offers a strict way of defining the structure of interchanged information
- It can be easily produced from the analysis of the physical model of information. Once the information model is described in an object-oriented notation, it can be mapped

into an XML-Schema and consequently into a database schema.

- In opposition to DTDs, it has an object-oriented nature. It is easier for object oriented analysis tools to export the information structure to an XML-Schema file, which can then be used by our system.

The major innovations of the X-Database module are:

- It takes as input an XML-Schema file, which describes the logical structure of information, maps it to a relational database schema and automatically creates the database.
- It uses the same XML-Schema file to validate the structure of XML documents transferred between the database and the system applications.
- The XML-Schema file defines the structure of the four basic database commands (insert, update, delete, select). This enables the database contents' manipulation exclusively through the use of valid XML documents.

Several issues on mapping object-oriented information to a relational schema have been faced and solved during the development of the database interface. Problems and solutions considering the database architecture will be discussed in detail in the following.

3.2 The entities of the XML-Schema file

The XML-Schema file contains five different types of sub-elements. The *xsd:* prefix in XML Schema terms is used to avoid name conflicts with the respective XML terms.

- *xsd:element*

They define the name and type of each XML element.

For example, the first of the following statements defines that XML files may contain an element named *Comments* with *text* content. The second defines that XML files may contain an element named *AudioVisual* with complex content as described elsewhere in the XML-Schema.

```
<xsd:element name="Comments"
type="xsd:string"/>
<xsd:element name="AudioVisual"
type=" AudioVisualDS "/>
```

- *xsd:attribute*

They describe the name and type of attributes of an XML element. They may contain an attribute named *use* with value *"required"*, which states that these attributes are mandatory for this XML element. They can be of simple (integer, float, string, etc.) or complex type (i.e. enumerations, numerical range etc.)

To give an example, the first *xsd:attribute* definition states that the *TrackLeft* attribute may take an integer value, whereas the second notes that the *AVType* attribute may take a value whose format is described elsewhere in the XML-Schema.

```
<xsd:attribute name="TrackLeft"
type="xsd:integer" use="required"/>
<xsd:attribute name="AVType"
type="AVTypeD" use="required"/>
```

– *xsd:simpleType*

They define new datatypes that can be used for XML attributes. To simplify the XML-Schema structure we only use simpleType to define enumerations of strings. An example of a simpleType follows:

```
<xsd:simpleType name="AVTypeD">
  <xsd:restriction base="string">
    <xsd:enumeration value="Movie" />
    <xsd:enumeration value="Picture" />
  </xsd:restriction>
</xsd:simpleType>
```

– *xsd:attributeGroup*

They group xsd:attribute definitions that are used by many XML elements. An example of an attributeGroup is the following:

```
<xsd:attributeGroup name="AnnotationD">
  <xsd:attribute name="annotation" type="xsd:string"/>
  <xsd:attribute name="who" type="xsd:string"/>
  <xsd:attribute name="what" type="xsd:string"/>
  <xsd:attribute name="place_where" type="xsd:string"/>
  <xsd:attribute name="time_when" type="xsd:string"/>
  <xsd:attribute name="why" type="xsd:string"/>
</xsd:attributeGroup>
```

– *xsd:complexType*

They represent the various entities of the metadata model. Complex types comprise of xsd:attributes, or group of xsd:attributes definitions and sequences of xsd:elements. Once defined, they can be used as the type of more complex XML elements. They contain:

- one or more *<xsd:attribute>* tags
- one or more *<xsd:element>* tags that describe the sub-elements of a given complex element.

Each sub-element has a “*type*” or “*ref*” attribute. This means that an element can contain a sub-element as a whole, or can refer to the sub-element using its id.

Definition 1: The complex types that are only declared as “*ref*” by other complex types are referred as *Top-level* types.

The following XML-Schema fraction gives an example of defining the internal structure of an XML element, named AudioVisual.

```
<xsd:element name="AudioVisual"
  type="AudioVisualDS" />
<xsd:complexType name="AudioVisualDS">
  <xsd:attribute name="id" type="ID"
    use="required"/>
  <xsd:attribute name="AVType"
    type="AVTypeD" use="required" />
  <xsd:sequence>
    <xsd:element maxOccurs="1" minOccurs="0"
      name="Syntactic" type="SyntacticDS" />
    <xsd:element maxOccurs="1" minOccurs="0"
      name="Semantic" type="SemanticDS" />
    <xsd:element maxOccurs="1" minOccurs="0"
      ref="MetaInfoDS" />
  </xsd:sequence>
</xsd:complexType>
<xsd:simpleType name="AVTypeD">
  <xsd:restriction base="string">
    <xsd:enumeration value="Movie" />
    <xsd:enumeration value="Picture" />
    <xsd:enumeration value="Document" />
  </xsd:restriction>
</xsd:simpleType>
```

According to the XML-Schema the AudioVisual element:

- has two attributes (“id” which is a number and “AVType”, which can take one of the values Movie, Picture or Document),
- may contain two sub-elements namely Syntactic and Semantic (with their sub-elements) and
- may contain a reference to a MetaInfoDS element.

A valid XML file according to the above schema would be:

```
<AudioVisual id="1" AVType="Movie">
  <Syntactic>
    ...
  </Syntactic>
  <Semantic>
    ...
  </Semantic>
  <MetaInfoRef>a2<MetaInfoRef>
</AudioVisual>
```

In addition to this, since XML-Schema supports inheritance, certain entities of the schema extend the features of other entities. This is performed with the use of the *<xsd:extension>* tag.

The extension entities contain all the features of their parent entity.

```
<xsd:complexType name="AudioSegmentDS">
  <xsd:complexContent>
    <xsd:extension base="SegmentDS">
      <xsd:sequence>
        <xsd:element name="Position"
          type="MediumTimeD" />
      </xsd:sequence>
      <xsd:attribute name="FirstFrame"
        type="xsd:integer" use="required"/>
      <xsd:attribute name="LastFrame"
        type="xsd:integer" use="required"/>
      <xsd:attribute name="AudioType"
        type="AudioTypeD" use="required"/>
    </xsd:extension>
  </xsd:complexContent>
</xsd:complexType>
```

3.3 The structure of the XML-Schema file

The XML-Schema file used to describe the structure of XML information is two-folded. The first part of the file contains the definition of the various entities of information and of some supportive structures, while the second part describes the structure of database commands.

Applications interact with the database using strictly defined XML documents. These documents contain the data that will be inserted into the database and also describe the database actions that will be performed.

Definition 2: The complex types that represent the information entities involved in our model are referred as *base elements*.

Definition 3: The complex types that are used to group base elements before they are sent for a transaction to the database are referred as *extension elements*.

The extension elements used are:

- **DBCommand**

This is the root element of the XML documents that are sent to the database. It may contain zero to many Insert, Update, Delete or Select elements.

```
<xsd:complexType name="DBCommand">
  <xsd:sequence>
    <xsd:element name="Insert" type="DBInsert"
      minOccurs="0" maxOccurs="unbounded"/>
    <xsd:element name="Update" type="DBUpdate"
      minOccurs="0" maxOccurs="unbounded"/>
    <xsd:element name="Delete" type="DBDelete"
      minOccurs="0" maxOccurs="unbounded"/>
    <xsd:element name="Select" type="DBSelect"
      minOccurs="0" maxOccurs="unbounded"/>
  </xsd:sequence>
</xsd:complexType>
```

- **DBInsert, DBUpdate**

They contain the elements to be inserted or updated. In order to maintain the consistency of the database, only certain elements can be inserted or updated. The definition of **DBInsert** or **DBUpdate** in XML-Schema marks out which elements can be inserted or updated. For example if an element A contains an element of type B, only A can be sub-element of a **DBInsert** element. The B element can be inserted or updated in the database only as sub-element of an A.



- **DBDelete**

Contains references to elements that can be deleted. Only the ids of the elements to be deleted appear into a Delete element/command.

- **DBReply**

This is the root element used to enclose the retrieved information, which is sent to the applications. DBReply may contain all the base elements.

- **DBSelect**

This element has three parts: a) a **"return"** element that contains the base element(s) to be returned, b) a **"from"** element that contains the element(s) that will be used as criteria for the query and c) a **"where"** attribute that contains the query's conditions. Detailed examples of the Select process will be presented in the following paragraphs.

The set of Complex types that represent information entities and database commands is stored in an XML-Schema file (namely xsdsource.xsd), which is used as initialization file for the X-Database module.

From the two groups of types only the base types are used for the database creation. The extension types are ignored during this phase. These are used only for the validation of the commands that are sent to and from the database.

4. From XML to Relational

4.1 Mapping XML-Schema entities and structures into a Relational schema

The X-Database module can be divided in two parts.

- The first part concerns the analysis of the XML-Schema file that contains information about the structure of the exchanged XML documents. This information is used to generate the relational database structures. This occurs in the initialization phase of the module.
- The second part parses XML documents and constructs the appropriate SQL commands that are processed by the database. It also takes database results and formulates valid XML documents as a reply to user queries.

a) Mapping attributes and attributeGroups

Each **"attribute"** or **"attributeGroup"** in XML-Schema is mapped to the database to a field or set of fields respectively. The required attributes correspond to NOT NULL constraints on the relevant fields. The following example presents a complexType that contains a set of attributes and its representation in the database.

```
<xs:complexType name="DigitalStorageDS">
  <xs:attribute name="id" type="ID"
    use="required"/>
  <xs:attribute name="name" type="string"
    use="required"/>
</xs:complexType>
```

```
CREATE TABLE DigitalStorageDS (
  id NUMBER NOT NULL,
  name VARCHAR2(20) NOT NULL,
  PRIMARY KEY (id));
```

b) Mapping simpleTypes

Each **"simpleType"** in XML schema contains an enumeration of strings that represent the possible values an attribute can have. To give an example, the **complexType** AudioVisualDS has an attribute named AVType of type AVTypeD, where AVTypeD is a **simpleType** containing three different values (Movie, Picture, Document). This attribute is mapped to an AVType field of type VARCHAR2 in the AudioVisualDS table. The following constraint is attached to this field:

```
AVType VARCHAR2(20) NOT NULL CONSTRAINT
CHECK (AVType IN ('Movie','Picture','Document'))
```

c) Mapping complexTypes

Each **"complexType"** in XML-Schema is mapped to a separate table in the database. Additional tables are created to represent many-to-many relations between complexTypes.

Two important issues that arise from the mapping of complexTypes to database tables are *containment* and *reference* to elements of another complexType. As mentioned before, one element of a certain complexType may refer to its sub-elements in two ways: as **"type"** (sub-elements are contained inside the parent element) or as **"ref"** (previously created sub-elements are referenced by their parent element).

On creating the XML-Schema all the physical relations among the various entities have been expressed as relations of containment or reference among the respective complex types. Therefore, when the complex types are converted to database tables it is essential for their relations to be correctly expressed into database constraints.

Sub-elements may appear zero to many (unbounded) times inside an element. The amount of possible “*ref*” or “*type*” sub-elements is defined by the *maxOccurs* attribute. The *minOccurs* attribute defines whether a field can be null (minOccurs=0) or not.

Combining the amount of occurrences of a sub-element with the different types of relations results in the following cases:

- ❖ Complex type A has *exactly one reference* to a complex type C.

This means that an element C must have been created before any elements of type A can refer to it. This is a one-to-many relation so an intermediate table is not required. For complex types A and C, two tables are created in the database. When an element of type C is deleted then all references to it must be removed.

Constraints

table A:
 CONSTRAINT A_C_ref FOREIGN KEY Cid REFERENCES C(id) ON DELETE SET NULL
 table C:
 CONSTRAINT C_Pkey id PRIMARY KEY.

- ❖ Complex type A has *unbounded* number of *references* to a complex type D.

This means that elements of type D must have been created before being referenced by elements of type A. This is a many-to-many relation so an intermediate table (A_D_link) is required, which will keep record of the order of the references inside A. So for complex types A and D three tables are created in the database. When an instance of A or D is destroyed then all related records in A_D_link are deleted.

Constraints

table A:
 CONSTRAINT A_Pkey id PRIMARY KEY.
 table D:
 CONSTRAINT D_Pkey id PRIMARY KEY.
 (if minOccurs=1 then Did NOT NULL.)
 table A_D_link:
 CONSTRAINT D_Pkey (Aid,Did) PRIMARY KEY.
 CONSTRAINT A_D_ref FOREIGN KEY Aid REFERENCES A(id) ON DELETE CASCADE
 CONSTRAINT A_D_ref FOREIGN KEY Did REFERENCES D(id) ON DELETE CASCADE

- ❖ Complex type A contains *exactly one element* of complex type B.

This means that the element B can exist only inside A. This is a one-to-one relation so for complex types A and B two tables are created in the database. When an instance of A is destroyed then the B contained in it must be destroyed.

Constraints

table A:
 CONSTRAINT A_Pkey id PRIMARY KEY.
 table B:
 CONSTRAINT B_Pkey id PRIMARY KEY.
 CONSTRAINT B_A_ref FOREIGN KEY Aid REFERENCES A(id) ON DELETE CASCADE
 CONSTRAINT Un_Aid Aid UNIQUE

- ❖ Complex type A contains *unbounded* number of *elements* of complex type E.

This means that elements of type E are declared only inside the container element A. The relation is one-to-many, since different elements of type A cannot share the same sub-elements of type E. The order in which elements of type E appear inside an element of type A is important, so it is stored in table E. When an A element is deleted then all the E elements contained in it must be removed from the database.

Constraints

table A:
 CONSTRAINT A_Pkey id PRIMARY KEY.
 table E:
 CONSTRAINT E_Pkey id PRIMARY KEY.
 CONSTRAINT E_A_ref FOREIGN KEY Aid REFERENCES A(id) ON DELETE CASCADE

d) Handling inheritance of types

An interesting issue encountered during the generation of the relational schema has been the preservation of the object-oriented nature of XML. The users should be able to access the database contents based on the logical structure of information rather than the database schema.

An object-oriented feature of XML-Schema is inheritance between types. The following example describes a typical case of inheritance and reference. The entities “Video_Tape” and “DVD” are declared as extensions of the abstract entity “Medium” so they inherit all the attributes and elements of “Medium” along with their own attributes and elements. A “Program” entity references the “Medium” in which it is stored (Figure 3).

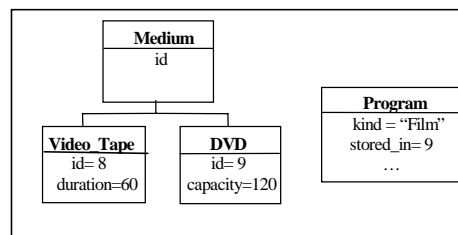


Figure 3. Inheritance and reference

```
<xsd:complexType name="Program">
  <xsd:attribute name="kind" type="string"/>
  ...
  <xsd:sequence>
    <xsd:element name="storedin" ref="Medium"/>
    ...
  </xsd:sequence>
</xsd:complexType>
```

The reference is made to the “Medium” entity but the referenced id may be the id of a “Video_Tape” or a “DVD”. The relational database system must be adapted to support this feature and the application must designate the entity actually referenced. In order to maintain such behavior a table named OBJECTIDS is created in the database keeping record of the **id** and **type** of each element that participates in a hierarchy. A group of triggers ensures that the reference is assigned to the correct element of the hierarchy.

An example of a trigger, which validates that the reference id corresponds to a record in MediumDS table follows:

```
CREATE TRIGGER trig_Program_Medium
BEFORE INSERT OR UPDATE OF
    Medium ON Program
FOR EACH ROW
CALL CheckKindOf(:new.Medium,
    'MediumDS')
```

4.2 Database manipulation using XML documents

An XML document that is sent to the X-Database module contains a root DBCommand element, which may contain many insert, update, delete or select sub-elements. The module parses this element and generates a set of insert, update, delete or select queries that contain the information of the complex type.

The information of an XML element is usually stored in many different tables. Therefore, a set of constraints, both database and programming ones, are required to guarantee the integrity of the database into the aforementioned actions. Some of the constraints are:

a) Certain entities exist only as *part of* other entities. The respective complex types appear only as sub-elements of other complex types. As a consequence these complex types cannot appear inside a **DBInsert** command. To give an example: the video segments information is related to a video object, so VideoSegmentDS elements appear only inside a VideoDS element. This guarantees that no video segment information that does not belong to a Video can be stored into the database.

b) Certain entities can be *re-used by more than one* entity. The complex types that correspond to such entities must appear as sub-elements of **DBInsert**. When a complex type refers to another complex type, the latter must already exist into the database.

c) Certain entities contain *ordered instances* of other entities, which in XML-Schema terms means that a complex type may have more than one sub-elements of another type. The order in which these sub-elements appear is important, hence ordering information must be stored in the database during the XML document parsing.

d) One or more commands can be send into the X-Database module at the same time. These commands usually evoke a reply from the database, so the DBReply module must be able to group the database replies for each command.

These constraints are incorporated into the XML-Schema file in the definition of *extension elements*. A pre-parsing of the XML-Schema document gives to the X-Database module all the information needed for parsing the XML documents.

The four database commands supported by the module are:

1) **Insert**: One or more *top-level complex types* can be found inside a **DBInsert** element. The parsing starts from the top-level element and continues recursively to all sub-elements, thus generating and executing a series of SQL INSERT statements.

2) **Update**: One or more *top-level complex types* can be found inside a **DBUpdate** element. A *negative id* has been given to all new elements, so these are inserted into the database. The rest of the elements, which have a positive number as id (this is the id provided by the database) are updated. The ids are returned (as in **DBInsert**)

3) **Delete**: Only *top-level complex types* can be found inside a DELETE element. The database deletes the information of all sub-elements for this element.

4) **Select**:

A DBSelect element has three parts:

- A **where** part that contains the filter of the Select query.
- A **from** part where the elements that contain the attributes to be filtered appear.
- A **return** part that contains the element(s) to be returned.

The whole element is returned after a select statement. The following XML fraction is an example of the **DBSelect** command.

```
<DBCommand>
<Select where="@-100@ = 1234">
<from>
  <AudioVisual id="-96" Name="Gladiator"
    AVType="Movie">
    <MediaInfo id="-97">
    <MediaProfile>
    <MediumRef>-100</MediumRef>
    </MediaProfile>
    </MediaInfo>
  </AudioVisual>
</from>
<return>
  <AudioVisualRef>-98</AudioVisualRef>
</return>
</Select>
</DBCommand>
```

This command selects all the AudioVisualDS entities that refer to a MediumDS with id=1234. The values of various attributes that appear inside the “from” tags are not taken into account during the parsing of the document. These values are randomly assigned to the **required** attributes, to maintain the validity of the XML document. Only the value of -100 field is important because -100 appears in the where clause.

The value of the *where* attribute is the “**where clause**” of the query. From the nested structure of elements inside the “from” part of the command, the parser is able to create all the “**join conditions**” among the tables that participate to the query. From the elements that appear in the “from” and “return” elements the “**list_of_associated_tables**” is generated.

The return element has a reference to the complex type(s) that must be selected. These references give the name of the table that corresponds to the “**return entity**”.

So the first query that is sent to the database has the following structure:

```
SELECT return_entity.id FROM "return_entity",
"list_of_associated_tables"
WHERE ("join_conditions")
AND ("where clause")
```

The ids of the selected entities are returned. The module uses these ids to address to the database a set of recursive select commands to obtain the complete information of the selected entities. The complete elements are returned to the application inside a DBReply element.

4.3 Defining relational database features in XML-Schema.

Relational databases offer many features that minimize storage space (exact definition of fields type and size) and accelerate the information retrieval procedure (indexes and views). Such features are not supported directly in XML-Schema. However we can extend the XML-Schema file of a metadata model to enable such features.

In the attribute definition we can replace the primitive types (i.e. string, integer etc) with simpleTypes that contain information concerning the size or list of values etc.

For example, the attribute VideoTapeCode which is of type xsd:string can become of type VTCode, where VTCode is defined as follows:

```
<xsd:simpleType name="VTCode">
  <xsd:restriction base="xsd:string">
    <xsd:length value="8"/>
    <xsd:pattern value="\d{3}-\d{4}"/>
  </xsd:restriction>
</xsd:simpleType>
```

The *base* attribute will be used to define the field type, whereas the *value* attribute of the *length* element will be used as the field size.

In order to define indexes in the RDBMS we must define the attributes of XML-Schema that will be indexed. A simple solution is adding an attribute named *indexed* that will appear and have the value "yes" only in attributes that should be indexed.

```
<xsd:attribute name="id" type="xsd:ID"
  use="required" db:indexed="yes"/>
```

However, the *indexed* attribute is not supported by the W3C's definition of XML-Schema and normally cannot appear inside an xsd:attribute, so we declare it to another namespace (as db:indexed).

5. Related Work

The X-Database module achieved to keep the query mechanism fairly simple, by combining the default structure of XML documents with the logic of SQL "Select" commands. The query

mechanism provided may not be very powerful as those of other platforms (OQL-S of Ozone [6], WebOQL [7]) or XML query languages (X-Query [8]), but has certain other advantages. The query itself is an XML document, whose certain parts (*from*, *return*) have the same structure as the other XML documents that are inserted or updated and the where part can be easily expressed in an SQL-like manner. Most of the query categories proposed in [9] could be performed using our DBSelect element, such as Simple Visual Feature Query, Feature Combination Query, Query by Example etc.

Compared to the applications implemented in commercial relational database management systems, X-Database provides a complete solution in XML documents manipulation. DB2 XML extender [10] supports the use of XML DTDs only for describing the database schema but does not support XML-Schema, which is more powerful in schema definition. Microsoft SQL Server [11] uses specific template files to describe the database schema. Informix [12], Oracle [13] and Sybase [14] mainly support creation of XML files from database contents but did not take advantage of the capabilities of XML-Schema. All the above systems do not provide the ability to create the database schema based on the XML-Schema document and moreover to use the same document to validate all the XML documents and commands that are forwarded to the database.

Research results and initiatives as those of presented in [15] (concerning mapping of DTDs) and [16] are proved very useful for the design and evaluation of our system. Although, both the above approaches consider the process of mapping an XML-Schema or DTD model to Relational Schema, they do not discuss the idea of storing and querying the relational data using XML documents.

6. Experimental evaluation

In order to test the reliability and scalability of the X-Database module a series of test transactions is performed to the database. These transactions included database creation, multiple insertions and deletions, updates and selections of database contents. In all the transactions the total response time is measured. This includes the parsing of XML input documents the time for accessing the database and creating the XML reply document.

The system has been measured using a simple interface, where XML documents are inserted as text and the resulting XML is displayed in a web browser. The simulation was running in a Pentium II computer with 128MBytes of RAM and an IDE HD.

The X-Database algorithm is tested using different kinds of application loads in a simulation. The performance of the module in creating a small or larger database schema, in handling multiple insert, update or delete commands and in processing less or more complicated select queries is evaluated.

6.1 Database Complexity

The most critical section in the X-Database work is the creation of the database. The module must analyze the structure of XML-Schema document and create the appropriate number of tables along with the required foreign keys and triggers that will

guarantee the integrity of the database in the cascade insertions and deletions. Several parameters of the database schema have been measured, such as the number of tables, foreign keys and triggers created, as well as the database creation time for XML-Schema files of different complexity. When the number of complexTypes, the number of references and the number of extensions in the XML-Schema increases, the number of tables in the database schema increases respectively. The results are presented in Figure 4.

Complex Types	Unbounded refs	Extensions	Tables	Triggers	Foreign keys	Creation time (sec)	Drop time (sec)
68	35	18	140	77	260	52	31
63	31	14	111	51	197	41	20
56	29	14	99	44	172	38	20
50	27	13	88	40	154	31	20
45	26	9	81	36	139	29	11
37	23	8	69	30	93	19	9
28	18	5	50	21	64	15	8
15	9	5	26	10	31	7	4
5	5	0	11	6	12	3	3

Figure 4. Database Performance for XML-Schemas of different complexity

6.2 Insertion - Selection time

In order to test the efficiency of our system, the time needed for a set of insertions and selections from the database has been measured. The database schema used for testing was created using an XML-Schema file of increased complexity, with 68 complexTypes and 18 extensions. It is explicit in the resulting graphs that both insertion and selection time are linear to the number of retrieved elements given that all the elements are of the same type (Figure 5). When elements of different type are inserted, the number of consequent insertions differs and the relation to time is not linear.

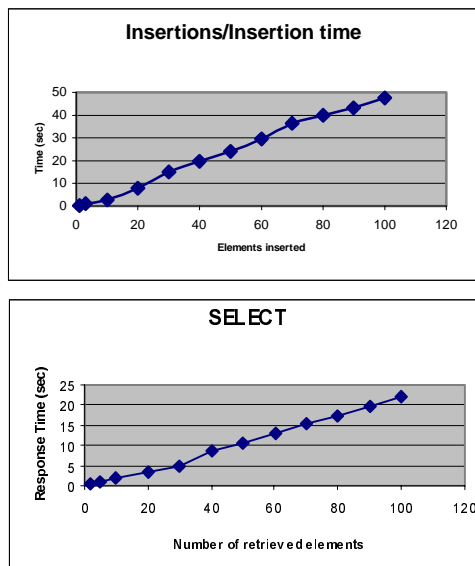
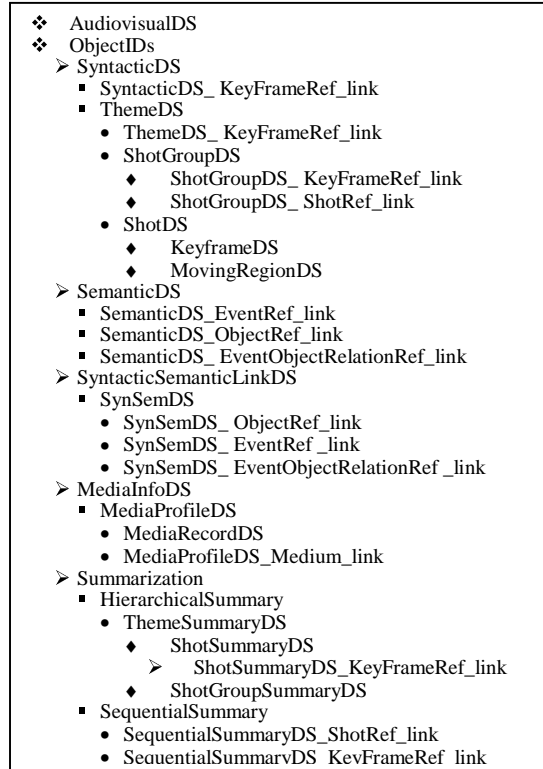


Figure 5. "Insert", "Select" Test Results

An audiovisual object stores its information into 34 different tables. As a result, an attempt to insert an audiovisual object into

the relational database evokes 34 insert SQL commands, which are executed recursively into the database.

In the following figure, the tables where the information of an audiovisual object is stored are listed.



7. Conclusions and further work

Attempting to create a platform that handles digitized video data and meta-information, where different modules will access the same database, XML is chosen to be the wrapper of the information transferred among them. A set of rules expressed in an XML-Schema notation guarantees the validity of transferred documents. The explicitness of XML-Schema has been exploited to build the relational database that stores the information.

Compared to the object-oriented one, the relational model is not the most appropriate for storing XML data. Attempting to map XML structures to tables and relations, we confirmed that it is much easier to use objects instead. Similarly, Bourret in [16] uses object schemas as intermediate between XML and relational. However, the storage of XML into relational DBMS is an important issue for many information systems that arises many non-trivial problems.

The X-Database module acts as an interface between the applications and the database. Client applications interact with the relational database system using only XML documents indifferently to the underlying database schema. This provides a simple and database independent mechanism for storing and retrieving video meta-information that can be easily applied to any information model. Fault tolerance is achieved by adding

appropriate control procedures to the database, such as triggers that check the validity of inserted values and reference constraints that guarantee the cascade removal of an object and its content objects from the database. The later is very critical since information for an object may be stored in more than one table in our database.

Further research has to be focused into information retrieval and advanced query tasks. The retrieval process can be easily accelerated if the appropriate indexes are created. For this task the current XML-Schema can be enriched to precisely define the indexes to be created, the information entities on which to perform similarity search etc. Finding an efficient notation to describe such necessities in XML-Schema is crucial in creating a database schema that will serve advanced retrieval needs.

Additional effort is needed to solve problems that concern schema evolution and its adoption to the database schema. We consider that the XML-Schema is defined once in the beginning of the whole process and never changes. In the majority of applications, the schema changes over time; elements are added, deleted and modified. The X-Database module is responsible for mapping these changes to the database.

8. Acknowledgments

The authors would like to thank Panayiotis Poulos for his help during the development phase. Many thanks to Nancy Routzouni for many helpful discussions on this work.

References

- [1] "XML Schema Part 0: Primer," W3C Working Draft, Sept. 2000 (<http://www.w3.org/TR/xmlschema-0>)
- [2] M. Fernandez, W.C. Tan, D. Suci, SilkRoute: Trading between Relations and XML, *WWW9 / Computer Networks* 33(1-6): 723-745 (2000)
- [3] G. Akrivas, S. Ioannoy, E. Karakoulakis, K. Karpouzis, Y. Avrithis, A. Delopoulos, S. Kollias, I. Varlamis, M. Vazirgiannis. An Intelligent System for Archiving and Retrieval of Audiovisual Material Based on the MPEG-7 Description Schemes, 5th WSES Multiconference on Circuits, Systems, Communications & Computers (CSCC 2001)
- [4] I. Varlamis, M. Vazirgiannis, P. Poulos, G. Akrivas, S. Ioannoy. X-Database: A middleware for collaborative video annotation, storage and retrieval, to appear in the 8th Panhellenic Conference in Informatics (2001).
- [5] ISO/IEC JTC1/SC29/WG11, "MPEG-7 Overview (v. 1.0)," Doc. N3158, Dec. 1999.
- [6] S. Abiteboul, J. Widom, T. Lahiri, A Unified Approach for Querying Structured Data and XML. The Query Languages Workshop, QL'98.
- [7] G. Arocena, A. Mendelzon, WebOQL: Restructuring Documents, Databases, and Webs, Proc. ICDE'98, Orlando, February 1998.
- [8] D. Chamberlin, J. Clark, D. Florescu, J. Robie, J. Simeon, M. Stefanescu. XQuery 1.0: An XML Query Language. W3C working draft, June 2001. <http://www.w3.org/TR/xquery>.
- [9] Y. Alp Aslandogan and Clement T. Yu, "Techniques and Systems for Image and Video Retrieval", *IEEE Transactions on Knowledge and Data Engineering*, vol. 11, no. 1, Jan.-Feb. 1999.
- [10] IBM's DB2 extender for XML (<http://www-4.ibm.com/software/data/db2/extenders/xmlxt.html>)
- [11] Microsoft SQL Server XML support, (<http://msdn.microsoft.com/msdnmag/issues/0300/sql/sql.asp>)
- [12] Informix and XML, (<http://www.informix.com/xml/>)
- [13] Steve Muench, Using XML and Relational Databases for Internet Applications, Oracle Corporation (<http://technet.oracle.com/tech/xml/info/htdocs/relational/index.htm#ID795>)
- [14] Sybase SQL server, (<http://www.sybase.com/products/databaseservers/ase/whitepapers/L01041.pdf>)
- [15] D. Florescu, D. Kossmann. Storing and querying XML Data using an RDBMS. Bulletin of the IEEE Computer Society Technical Committee on Data Engineering. (1999)
- [16] R. Bourret, Mapping W3C Schemas to Object Schemas to Relational Schemas, <http://www.rpbouret.com/xml/SchemaMap.html> (March 2001)