WPI-CS-TR-02-18 May 2002

X-Cube: A fleXible XML Mapping System Powered by XQuery

by

Steffen Christ and Elke A. Rundensteiner

X-Cube: A fleXible XML Mapping System Powered by XQuery[*]

Steffen Christ and Elke A. Rundensteiner

Department of Computer Science

Worcester Polytechnic Institute

Worcester, MA 01609-2280

(stchrist | rundenst)@cs.wpi.edu

Abstract

Data management of XML using relational technology has received increased attention in recent years and a large variety of mapping strategies has been proposed for that purpose. However, there still is the lack of a generic mapping framework to support all possible mapping strategies and glue them together as desired by an XML database designer. Current solutions are often hard-coded or they are bound to one particular mapping strategy. We thus present the first *FleXible XML Mapping System Powered by XQuery*, called *X-Cube*. X-Cube supports the coordinated generation of the Default XML Schema as well as the actual XML Default View for loading XML data into a target data storage in a schema independent manner. The system is capable of executing both directions of the mapping process, loading and extraction.

X-Cube is a unified mapping approach that has several advantages, including (1) capture of all existing mapping strategies in a declarative, generic, and schema independent fashion, (2) support of mapping composition, (3) no demand for the definition of a new mapping language, (4) 100% based on XML technology, and (5) use of off-the-shelf tools only. Moreover, X-Cube is a standalone middlelayer system that can easily be used with other backend storage systems in terms of both data models (e.g., Object-Relational or Object-Oriented) *and* commercial database systems (e.g., Oracle 9i, IBM DB2, or Microsoft SQL Server). We have implemented a repository of the literature's most popular mapping strategies. X-Cube can now employ and compose them for actual data and schema mapping. Our experiments show the feasibility of X-Cube.

[*]This work was supported in part by NSF, namely, the NSF NYI grant IIS-979624, the NSF CISE Instrumentation grant IRIS 97-29878, and the NSF grant IIS-9988776. Steffen Christ wants to thank the DAAD for partial support. Steffen was on a one year exchange program funded by DAAD at WPI when this work was undertaken. His home university is the Department of Computer Science, Darmstadt University of Technology, D-64283 Darmstadt. In Darmstadt, Steffen was supervised by Prof. Alejandro P. Buchmann during this time period.

Contents

1 Introduction

In the last few years, XML has emerged as *the* data exchange format for business data on the web. The storage of XML data in a way that is flexible enough to support the semantics of XML and at the same time offers the services that are common for database systems, such as reliability, concurrency control and optimization of query processing, remains a partially unsolved issue.

Native XML database systems are still in their infancy and typically do not scale well [27, 4, 21]. Additionally, much of the data shared via XML often has to be stored in relational systems, because it is also used by relational-based software packages.

Most commercial database systems [17, 23, 24] currently support storage of XML data, meaning that XML data is either mapped to relational tables in a fixed and hardcoded manner that restricts the database design or that XML fragments are stored as *blobs* that do not allow the appliance of relational query optimization strategies. Hence, there clearly is the need for an middle layer application that can bridge the gap between XML and relational databases in a data storage independent and flexible way, allowing users of any relational data source to benefit from XML and XQuery capabilities.

The challenge of storing XML in relational database systems arises from basic differences in the data models. XML is ordered and semi-structured, that is the data is not strongly typed, arbitrarily nested, and implicitly structured. Relations on the other hand are typically unordered, strongly typed, and flat. These differences make it virtually impossible to implicitly define a simple algorithm for mapping data (and schema) between the two models. The power and flexibility of XML constitutes a much larger variety of data structures whose elements contain much deeper and more specific information then can be captured by a one-to-one mapping to a relational data type. Hence, a major challenge when mapping XML semantics to relational systems is to find an "adequate" way of storing XML's rich structure using relational data types, because there is no implicit and *"best"* way of mapping between XML data and relations.

Kossma et. al. [20] have done an analysis of various mapping strategies and clearly pointed out advantages and disadvantages of either approach, showing that there is no single and best choice of a appropriate strategy. Bohannon et. al. [6] favor a system where the "best" mapping is determined based on estimated query-costs based on extended XML Schema information and workload estimates. However, X-Cube is an open system that is capable of using any of those mapping strategies. It provides the necessary flexibility in choosing and properly composing arbitrary mapping strategies to eventually reach a mapping solution that is "best" for the particular application requirements. X-Cube moreover fits into middle layer XQuery engines like Rainbow [29] or XPERANTO [13] that eventually provide the possibility of actual XQuery processing over the loaded data.

1.1 Related Work

Deutsch et. al. [18] assume that there is no schema file provided with the XML data. Hence the proposed STORED system focuses on employing data mining techniques to solve schema discovery and storage mapping by identifying "highly frequent" tree patterns.

Shanmugasundaram et. al. [25] propose three strategies for mapping DTDs to relational schemas. Based on the schema knowledge they inline elements to reduce redundancy. Hereby, multi-valued elements and and such involved in recursive associations must be kept in separate tables.

Florescu and Kossman [20] investigate several alternatives for defining relational schemas from XML structures and evaluate them in terms of performance and fragmentation. They point out that there is no "best" mapping for a particular XML document to a relational representation and that the choice of the appropriate mapping strategy depends on the XML schema and expected query workload.

Bohannon et. al. [6] try to find the “most optimal” relational schema for a given XQuery workload based on incremental schema modifications. A modified XML Schema - named p-schema - is used to include statistical information about the XML data in the decision process of how to represent the particular XML structure with relational tables. Unlike the X-Cube approach, they assume that a basic (hard-coded) mapping strategy is used for the initial transformation step. Afterwards the resulting relational schema can be incrementally modified by inlining or partitioning certain element types.

XPERANTO [13] and Silkroute [19] focus on how to efficiently publish (extract) relational data into XML formats. SilkRoute proposed its own mapping language called RXL whereas XPERANTO utilizes XQuery.

Systems like XPERANTO [13] or Rainbow [29] should be seen as complementary systems to X-Cube, because those systems assume that some XML data has already been loaded into a relational database based on some mapping strategy. They then use features of the storage system to actually execute XQueries over the stored data. XPERANTO utilizes extracting XQuery expressions to construct virtual XML Views of the stored XML data. The extraction XQuery as well as the user XQuery are converted into an algebra tree representation of XQuery. After optimizing the concatenation of the two trees through rewrites and cuttings, as much of the remaining query as possible is pushed down to the underlying SQL database to do most of the actual query processing. This approach allows for high flexibility in two ways. First, the system is independent from the underlying database and second, the system can be used against data that was loaded with an arbitrary mapping strategy, as long as an extracting XQuery expression can be encoded for XPERANTO. X-Cube provides the necessary extraction XQuery even for composed mapping strategies and hence could directly be used in conjunction with systems like XPERANTO.

Most commercial database systems meanwhile also provide some support for XML to relational mapping. Most efforts such as IBM’s DB2 XML Extender [17], MS SQL Server’s XSD Schema [23], and Oracle’s XSU [24] center around the implementation of one particular mapping strategy, typically including the invention of a proprietary language or data structure to capture the mapping semantics. We overcomes both shortcomings in X-Cube, one by providing an open and geenric repository of mapping strategies to support any desired mapping and two by encoding all these strategies as XQuery expressions for openness, generality, and execution support.

IBM’s DB2 provides three fixed strategies to incorporate XML data into the relational database: (1) The XML file can be stored outside DB2, but still be accessable from within the database, (2) the whole XML data can be stored within one single column as an *XMLCLOB* or (3) as an *XMLVARCHAR*. All three possibilities are hard-coded and very limited as single XML data values inside the XML document are not really accessable via SQL.

Microsoft’s SQL Server provides a possibility of flexible data mapping that is based on a properietary mapping language (XSD mapping templates). The language borrows from XML Schema, but is much more restrictive, hence the flexibility of mappings that can be used on SQL Server is limited by its capabilities.

Oracle provides similar capabilities as IBM’s DB2 system. XML fragments can be stored as *CLOBS* or *VARCHAR2* columns. This again means that whole XML fragments are stored in one table cell, possibly as complex objects with nested object structures using OR extensions. Oracle’s XML to SQL utility offers additional support to shred individual XML elements and attributes into single table cells. However, XSU uses a fixed mapping that basically requires the XML document to be in the Default XML View format and hence is relatively strict on the format of XML data.

1.2 Our X-Cube Approach

In this paper, we describe X-Cube, the first mapping system that captures the semantics of all known mapping schemas in a generic and flexible manner. It uses the W3C standardized query language XQuery [5] for both mapping definition and execution.

X-Cube offers the following advantages:

1. **Declarative:** X-Cube is powerful enough to *describe all popular mapping strategies* from the literature. This description is *not* hard-coded, being limited only by the expressive power of XQuery.

2. **Executable:** X-Cube's mapping descriptions can directly be used to execute the mapping transformation, employing a standard and arbitrary XQuery engine.

3. **Generic:** The power of XQuery permits X-Cube to *exploit meta-querying* for XML Schema information for the mapping process, enabling X-Cube to support complex mapping strategies with arbitrary XML documents with zero additional effort.

4. **Composable:** X-Cube allows the seamless *composition of different mapping strategies* into one customized one, while still enabling both synchronized loading and extraction processing, following these composed mapping strategies.

5. **Extendable:** X-Cube's repository *can easily be extended* with new mapping strategies, allowing the customization and extension to *any imaginable mapping strategy* including hard-coded solutions of commercial database systems.

6. **Portable:** X-Cube is a *100% pure XML system* that is *portable to other data models* simply by introducing a Default XML View definition for them (e.g., Object Relational or Object Oriented systems). Moreover, it is not bound to any particular commercial database system.

7. **Optimizable:** Through the use of a standard XQuery language, X-Cube *can benefit from query optimization* advances applied to any XQuery engines.

8. **Non-proprietary:** X-Cube *does not define a new proprietary mapping language*, but rather relies on the W3C standard XQuery and *off-the-shelf technology* for the actual execution process.

X-Cube employs the concept of default XML views over target data sources, what makes X-Cube usable for middle-layer XQuery processing systems like XPERANTO and Rainbow [13, 29]. Those typically assume that XML data has been loaded into an underlying database system and use a extraction XQuery expression to construct a virtual view of the data for later query processing. The view of the data source as well as the user query are converted into algebra trees which then are connected to an overall query tree whose leaf nodes represent (possibly multiple) data sources. By cutting and rewriting the resulting tree, the actual query computation can be optimized and major parts of the remaining query execution can be pushed down to the database server. The use of X-Cube in conjunction with those systems results in an XQuery database system that has major advantages to commercial product extensions [24, 17, 23]: The system still uses an (maybe still commercial) relational or object-based database engine, but remains independent from the particular (usually limited) XML support of that system. Hence, the mapping of XML data can benefit from X-Cube's flexibility, while the system is still capable of executing XQuery expressions.

1.3 Outlook

The following section briefly describes the necessary background to XML and introduces our running example. Section 4 describes general issues when mapping XML to relational databases. Section 3 shows the general approach of X-Cube. Section 5 explains the use of XQuery for mapping and its advantages, showing diverse mapping strategies and how those are encoded for X-Cube. Section 6 defines how X-Cube can compose different mapping strategies.

Section 7 shows possibilities for instantiating mapping queries. Section 8 shows experimental results before finally section 9 concludes the paper. All XQuery expressions used by X-Cube can be found in the Appendix and on the X-Cube web-page (http://davis.wpi.edu/dsrg/xcube/).

2 Background

The Extensible Markup Language (XML) [10] – a subset of SGML [1] – has been standardized by the W3C as a "universal format for structured documents and data on the web". In recent years, XML has emerged as *the* generic data exchange format on the web. Its success is mainly based on the fact that it has retained main advantages of SGML, such as *extensibility, structure, and validation*, and hence is sufficiently flexible and powerful for broad web use.

2.1 Extensible Markup Language (XML)

Figure 1 shows our running XML example, which is a part of Example 1.5 of the W3C XQuery Use Cases [14]. It contains all main features of XML [10] that impact the mapping to relational databases. Elements like `<para>` that contain both nested sub-elements *and* `PCDATA` are referred to as *mixed-content* elements.[1]

```
<report>
  <title>Getting started</title>
  <chapter chapterid="chap1">
    <title>SGML</title>
    <intro>
      <para>While...<emph>markup</emph>...</para>
    </intro>
  </chapter>
</report>
```

Figure 1: Example "Report" XML Instance.

Characteristics steming from XML's semistructure that impact the mapping of XML files to relational databases include [3, 12]:

(a) XML's structure is irregular and not strongly typed: Elements in XML may not only be of one *simple type* (`STRING`, `INTEGER`, ...) like the `<title>` element in Figure 1, but can be of *complex type* (containing an arbitrary list of nested sub-elements as content) like the `<report>` element or *mixed type* (containing sub-elements and `PCDATA`) like the `<para>` element. Hence, it is not possible to identify elements as "of the same type" without a *type check*. Unlike relational systems where usually a simple column name depicts an element type, in XML the simple check for the identifier of an element is typically not sufficient as the type of particular elements may vary throughout the document.

(b) The structure is implicit: Although every XML document has a specific structure, it may not be known in advance and – caused by the descriptive nature of XML – may only be found encoded in the actual document. Such implicit structure information usually cannot be extracted without parsing the containing document completely. A special case of those implicit structures is the relative *order* of XML elements that can *only* be found when looking at an XML document instance.

[1] Strictly speaking, *mixed-content* means any content that contains `PCDATA`, hence even elements that only consist of `PCDATA`. In common usage, as in this paper, *mixed-content* refers to elements that contain a mixture of both sub-elements and `PCDATA`.

(c) The distinction between data and schema is blurred: Depending on the design of the particular XML document the boundaries between schema information and data are blurred. For example, replacing an element `<PRICE CURRENCY="USD"|"EUR">` with new price-tags `<US_PRICE>` and `<EU_PRICE>` includes parts of data in the tag-names that usually are assumed to contain schema information only.

XML is often used as basic syntax to define special purpose languages for data transfer between various applications. Those languages can be divided into two major categories. *Data-centric languages* are typically used to transfer data between applications and data storages. They describe discrete pieces of data with a usually fairly regular structure. The data is often fine-grained and contains little or no mixed-content. The order in which sibling elements and `PCDATA` occur is usually not significant; hence the mapping of those languages to a pure relational storage is relatively easy as it is not necessary to store additional order information inside the relational database.

Document-Centric languages on the other hand are intended for human consumption and hence have a less predictable structure and are coarser grained. The amount of mixed-content data is relatively high and the order in which elements and `PCDATA` occur is important. The mapping of this kind of data to relational databases can easily get fairly difficult and complex. As we assume that relational storage systems do not order columns or rows, the mapping of those data explicitly demands for separate treatment of order information [8].

2.2 XML Schema

XML Schema is a W3C XML standardized syntax for XML schema types that itself is based on the XML syntax [11]. Its use allows X-Cube to query over an XML instance and its corresponding XML Schema using only one XQuery expression that thus can be executed on a single XQuery engine. An XML instance is an XML document that corresponds to (is valid to) a certain XML Schema. We use the notation $xml^{<s>}$ to express that document *xml* is a valid instance of the XML Schema *s* [2]. XML Schema provides much broader possibilities of defining element types, minimum and maximum occurences of elements, and nesting structures than its predecessors such as DTD [7] and DCD [9]. Moreover it incorporates database modeling constructs like key and foreign key constraints. Figure 2 shows the XML Schema for the running example.

2.3 XQuery: XML Query Language

The XML Query language XQuery [5] was derived from a predecessor XML query language Quilt [15], along with useful features from several other languages (e.g. XPath [16]).

In XQuery, a query is composed of expressions. Main expression types include `FLWR` (pronounced "flower") *expressions* that build the framework of most XQueries (like SFW in SQL), *path expressions* that were derived from the XPath syntax [16] for navigating through the XML structure, *element constructors* for constructing new XML fragments, *functions* and *arithmetic operators* such as `SUM`, `COUNT`, and `AVG`, *conditional expressions* for joins of different sources and standard qualifying conditions, and expressions used for *modifying or testing data types* that are necessary as XML is not strongly typed.

Figure 3 shows an example XQuery expression, showing an outer *element constructor* (`<chapter-titles>`) and a shortened `FLWR` *expression* with an embedded *path expression* navigating to `/chapter`.

XQuery is a powerful XML query language with essential capabilities for XML data mapping: (a) it is capable of querying unknown or unpredictable data, (b) it allows querying for schema-information inside XML data, (c) it

[2]Throughout the paper we will use capital letters to describe sets of elements, e.g., *XML* for the set of all XML documents, *REL* for the set of all relational instances, *XMLSCHEMA* for the set of all XML Schemas.

```
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema" elementFormDefault="qualified">
 <xs:element name="report">
  <xs:complexType>
   <xs:sequence>
    <xs:element name="title" ref="title"/>
    <xs:element name="chapter" name="chapter">
     <xs:complexType>
      <xs:sequence>
       <xs:element name="title" ref="title"/>
       <xs:element name="intro" name="intro">
        <xs:complexType>
         <xs:sequence>
          <xs:element name="para" name="para">
           <xs:complexType mixed="true">
            <xs:choice minOccurs="0" maxOccurs="unbounded">
             <xs:element name="emph" name="emph" type="xs:string"/>
            </xs:choice>
           </xs:complexType>
          </xs:element>
         </xs:sequence>
        </xs:complexType>
       </xs:element>
      </xs:sequence>
      <xs:attribute name="chapterid" type="xs:string" use="required"/>
     </xs:complexType>
    </xs:element>
   </xs:sequence>
  </xs:complexType>
 </xs:element>
 <xs:element name="title" type="xs:string"/>
</xs:schema>
```

Figure 2: XML Schema for Example “Report”.

generates query results that even can be complex XML fragments, (d) it allows the construction of elements based on query results, (e) it allows the definition of *User Defined Functions* that may be recursively called, and (f) it is capable of simultaneous querying of multiple data sources, even both XML Schema and XML data.

2.4 XQueryX: XML Syntax for XQuery

Unlike XML Schema, XQuery has its own proprietary syntax that needs special parsers and engines for processing. In June 2001, the W3C has started an attempt to define an XML syntax for XQuery named XQueryX [22] to embed XQuery in the XML family in the same way as it did with defining an XML syntax for schema definitions.

The XML syntax of XQuery breaks the overall query expression into sub-expressions and those are again broken into their constituents. All variable specifications as well as path expressions are also broken into nested structures. This way specific parts of the path expressions or variables within path expressions are directly accessible as single XML elements. As XQueryX is a valid XML document, it is again queryable with XQuery. Hence it is possible to *query a query* and for example to replace variables or parts of a path expression. This feature can be used for example to incorporate schema knowledge in an XQuery expression that is used to query for things like tag and attribute names. Finally, tools are being developed that convert XQuery expressions into XQueryX and vice versa [26].

Figure 4 shows the example XQuery from Figure 3 in XQueryX notation.

```
<chapter-titles>
   FOR $chapter IN document("source.xml")/chapter
   RETURN
   {
      $chapter/title
   }
</chapter-titles>
```

Figure 3: Example XQuery Expression.

3 The X-Cube Approach

3.1 Architecture

Figure 5 shows that mapping using X-Cube consists of five steps. First, the user defines a particular mapping strategy by selecting and possibly combining mapping strategies from a *Mapping Repository* (this actually results in a pair of XQuery expressions for loading and extraction). In a second step, the combined *loading query* for the selected mapping is executed against the XML document and its corresponding XML Schema file to generate the appropriate XML Default View and its Default XML Schema. In a third step, the resulting Default XML View and its corresponding Default XML Schema are used by the *XML Source Wrapper* to generate the relational schema and to copy the data from the default XML view.

Steps four and five are used for extracting the data from the database. Step four generates a Default XML View and a corresponding XML Schema of the database instance, afterwards step five uses the (combined) *extracting query* that was selected by the user for transforming the Default XML View into the original XML document that had been loaded initially.

Figure 6 shows how X-Cube can be used with middle layer XQuery engines like XPERANTO or Rainbow [13, 29].

The processing of both, the XML document and the XML Schema, enables the system to be generic and at the same time (a) support mapping strategies that require full schema information, such as inline, (b) create correct data types on the relational side, (c) link the relational tables using foreign key constraints, such as for full shredding, and (d) allow data updates that conform to the given XML Schema without having to change the relational schema.

For all database systems that have a fixed and non-ambiguous definition of a Default XML View, the system can easily be ported. The required definition of a Default XML View ensures that all mapping decisions have to be captured in the transformation algorithm from the original XML document to the Default XML View. The remaining data loading part is non-ambiguous and hence straightforward.

3.2 Employing Default XML Views

When bridging XML and relational databases, there is one fixed mapping that is accepted throughout the community: The *Default XML View*. It is a direct and non-ambiguous XML representation of relational data. Consequently, for every relational database instance, there exists one unique Default XML View representation. As the reverse is true as well, the definition of any mapping strategy can be reduced to the specification of a transformation algorithm between the original XML data and the Default XML View of the desired relational instance to be created.

We use the notation $xml^{[rel]}$ ($xml \in XML$, $rel \in REL$) to express that the XML document *xml* is a Default XML View, representing the relational instance *rel*. Figure 7 shows the Default XML View of the running example after the Maximal Shredding mapping [28] (see Figure 8).

```
<?xml version="1.0"?>
  <q:query xmlns:q="http://www.w3.org/2001/06/xqueryx">
    <q:elementConstructor>
      <q:tagName>
        <q:constant q:datatype="xs:qname">chapter-titles</q:constant>
      </q:tagName>
      <q:constant q:datatype="xs:string">
      </q:constant>
      <q:flwr>
        <q:forAssignment q:variable="$chapter">
          <q:step q:axis="child">
            <q:function q:name="document">
              <q:constant q:datatype="xs:string">source.xml</q:constant>
            </q:function>
            <q:identifier>chapter</q:identifier>
          </q:step>
        </q:forAssignment>
        <q:return>
          <q:step q:axis="child">
            <q:variable>$chapter</q:variable>
            <q:identifier>title</q:identifier>
          </q:step>
        </q:return>
      </q:flwr>
      <q:constant q:datatype="xs:string">
    </q:constant>
    </q:elementConstructor>
  </q:query>
```

Figure 4: Example XQuery in XQueryX.

A Default XML View of relational data consists of a root-element named `<DB>` that contains a sub-element for every table of the relational data. The name of such elements is equal to the name of the corresponding table. For every row in a particular table, the corresponding table element contains a sub-element called `<table_name.ROW>`. Each row in turn contains an element for every column of the particular table that is named like the column. It contains the value of the particular cell that is defined by the combination of table_name, row, and column. Hence, the structure of a Default XML View is very similar to tables in HTML.

3.3 Using XQuery for Mapping

We use the term *mapping* to refer to the overall definition of a correspondence between XML data fragments and relations; *m: xml* $\Longleftrightarrow$ *rel* with *xml* $\in$ *XML and rel* $\in$ *REL*. Hence, *mapping* refers to both *loading* (*l: XML* $\longrightarrow$ *REL*) and *extraction* (*e: REL* $\longrightarrow$ *XML*). “Usable” mapping strategies should assure that $\forall$ *xml* $\in$ *XML: e(l(xml))=xml*, that is, the mapping is loss-less.[3]

XQuery is *closed* and *complete*. Hence, for every desired transformation of a particular XML fragment xml_1 into any arbitrary XML fragment xml_2, there always exists an XQuery expression *q*, so that *q(*xml_1*)* = xml_2. As Default XML Views are valid XML documents, the loading process can be reduced to executing a loading XQuery expression *l*, so that *l(*xml_1*)=*xml_2^{rel}, where xml_1 is the original XML Document and xml_2^{rel} is the Default XML View of the

[3] In this paper we assume that only mapping strategies that in fact are loss-less are used.

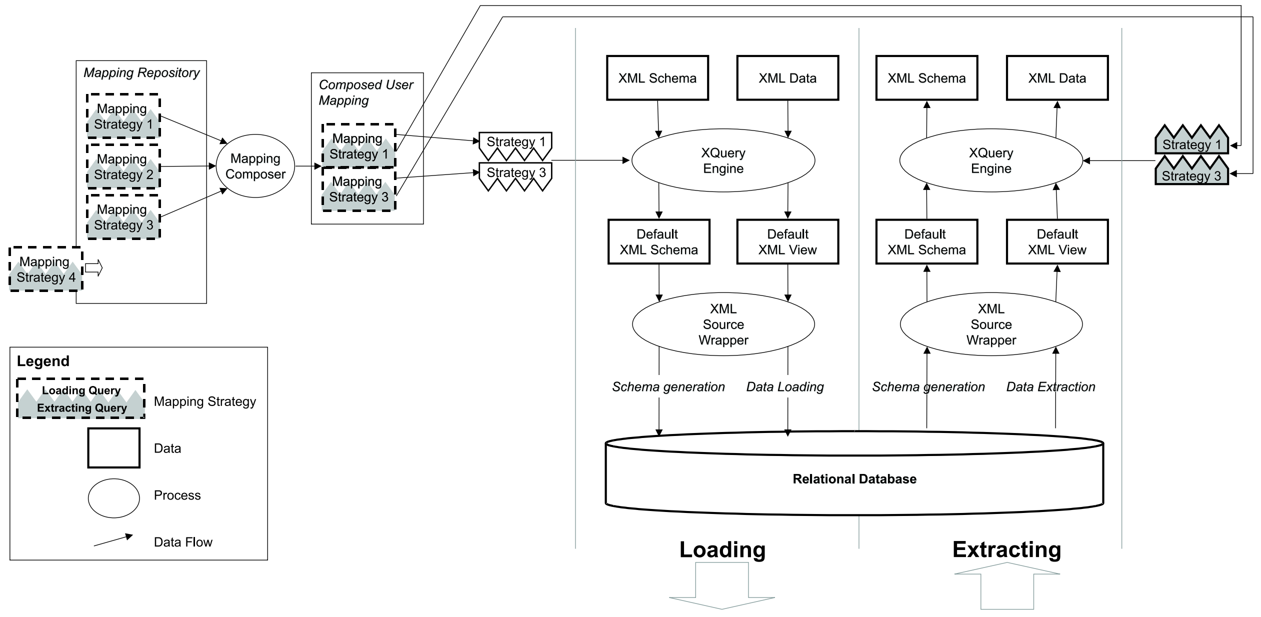


Figure 5: X-Cube Architecture.

desired relational schema *rel*. Similar, the extraction of XML can be reduced to executing an extraction query *e*, so that $e(xml_2^{rel})$=xml_1, where again xml_2^{rel} is the Default XML View of the desired relational schema *rel* and xml_1 is the original XML Document that was loaded using *l*.

For every mapping strategy *m* and a particular XML Schema *s* we can construct a loading XQuery expression $l^{m,s}$ so that $\forall\ xml_1^{<s>} \in XML : l^{m,s}(xml_1^{<s>})$=$xml_2^{rel}$ where *rel* is the desired relational schema under *m* and *s* (m: $xml_1^{<s>} \iff rel$). As $l^{m,s}$ can only be applied to XML documents that conform to the XML Schema *s*, we refer to $l^{m,s}$ as an *instantiated mapping*.

Based on XQuery's support of user defined functions and recursive function calls, we now propose that we can even construct generic loading query expressions l^m for every mapping *m* that are schema independent. That is, they work for any XML document as long as the appropriate XML Schema file is provided at execution time: $\forall\ s \in XMLSCHEMA, \forall\ xml_1^{<s>} \in XML : l^m(xml_1^{<s>}, s)$=$xml_2^{rel}$. As those query expressions can be applied in a general manner (to all XML documents no matter what schema they conform to), we refer to them as *generic mappings*. Figure 8 shows X-Cube's XQuery expression for the maximum shredding mapping [28]. This expression is encoded in a *generic* manner.

Similarly, extracting XQuery expressions can be constructed in two ways:
(1) XML Schema dependent, $e^{m,s}$ so that $\forall\ xml_2^{[rel]} \in \{xml \mid l^{m,s}(xml_1)=xml \lor l^m(xml_1, s)=xml \text{ with } xml_1 \in XML^{<s>}\} : e^{m,s}(xml_2^{[rel]})=xml_1$.
(2) XML Schema independent, e^m so that $\forall\ xml_2^{[rel]} \in \{xml \mid l^{m,s}(xml_1)=xml \lor l^m(xml_1, s)=xml \text{ with } xml_1 \in XML^{<s>}\} : e^m(xml_2^{[rel]})=xml_1$, no matter what XML Schema *s* was used.

The use of XQuery for the transformation process has several advantages:

1. XQuery is powerful enough to serve as description language for *capturing all common mapping strategies*.

2. XQuery allows the construction of *generic* schema independent loading expressions.

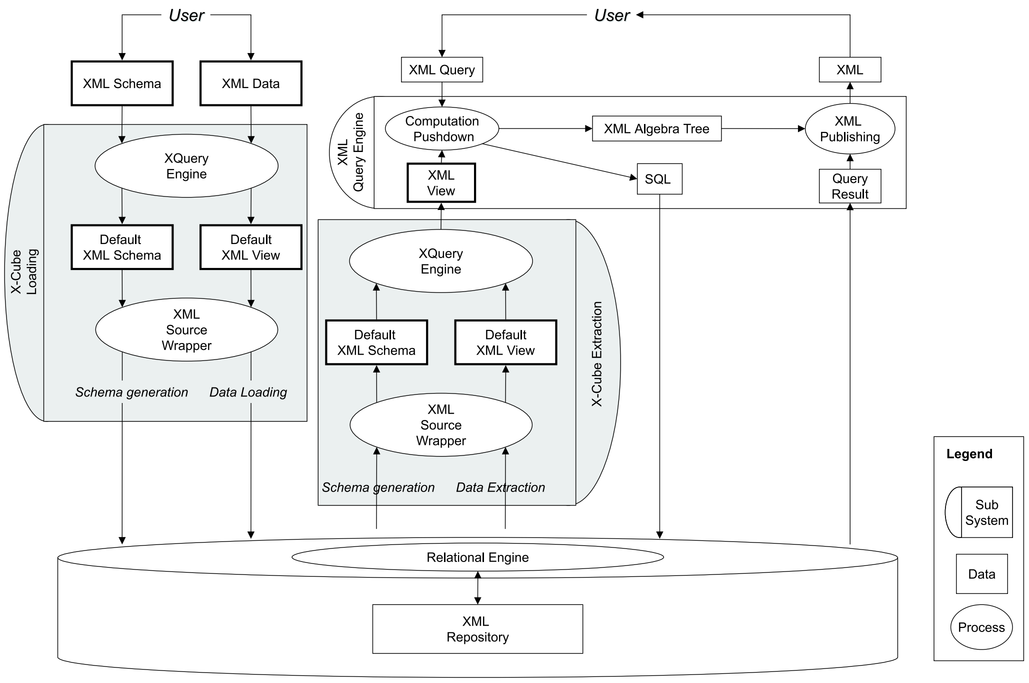


Figure 6: X-Cube's use in Middle Layer XQuery Engines.

3. The defined XQuery expressions can be used to directly *execute* the actual transformation.
4. XQuery is a W3C standard query language and hence can be processed using *off-the-shelf tools*.
5. The transformation process can benefit from any *optimization techniques* exploited by the used XQuery engine.

4 Bridging XML Data and Relational Databases

This section describes general issues when bridging XML data and relational databases. First, we describe XML features that directly impact the mapping between XML and relational systems and pinpoint those impacts. We then briefly discuss different conceptual data models for the representation of XML and show how popular mapping schemes from the literature utilize those to map XML data to relational representations.

4.1 Mapping Issues

This section describes issues that generally arise when mapping XML data to relational databases independent from the particular approach that is used. Those issues arise because of the structural differences between the simple and flat relational model and the complex, flexible and nested structure of XML and should be treated carefully, whatever mapping scheme is used.

4.1.1 Semi-Structure

XML's semi-structure in particular induces a set of untypical characteristics compared to relational data:

```
<DB>
  <report>
    <report.ROW>
      <IID>1</IID><ORDER>1</ORDER><PID>0</PID>
  </report.ROW>
  </report>
  <title>
    <title.ROW>
      <IID>2</IID><ORDER>1</ORDER><PID>1</PID>
    </title.ROW>
    <title.ROW>
      <IID>7</IID><ORDER>1</ORDER><PID>4</PID>
    </title.ROW>
  </title>
  <chapter>
    <chapter.ROW>
      <IID>4</IID><ORDER>2</ORDER><PID>1</PID><chapterid>chap1</chapterid>
    </chapter.ROW>
  </chapter>
  ...
```

Figure 7: Running Example's Default XML View for the Maximum Shredding Mapping [28].

Mixed Content and Typing. Pure relational systems only support simple data types (`String`, `Integer`, ...). This restriction is not of special relevance when mapping XML attributes or *simple type* XML elements, because those only contain simple data type values. However, a basic feature of XML are elements of *complex type*, meaning that the value of the element is no simple type, but consists of other elements and/or simple type values (`PCDATA`) in an intermixed fashion. Hence, not every element's value can be mapped directly into a relational value, that is, complex type elements have to be split into simpler types, which then eventually can be mapped to single relational values.

Structure is Implicit. The flexibility of XML allows that the structure of elements may vary throughout an XML instance. Optional sub-elements, sub-elements with an unbounded `maxOccurance`, and mixed-content elements may lead to various instances of an element. Hence, it is not possible to determine all possible structures by just reviewing one particular element instance. Either the XML data has to be parsed completely a priori or extra a priori schema information is necessary to determine all possible element types that the relational representation must be able to hold.

Distinction Between Schema and Data is Blurred. The final goal is to store XML data inside a relational database and to use its features for querying the data (either directly or by using middle layer XQuery processing systems). As relational databases are not capable of querying schema information, an important issue when mapping XML data to relations is to make sure that no XML data information that shall be queried later is stored as schema information in the relational database.

4.1.2 Order

A major discrepancy between XML and relational systems is order handling. XML data is ordered and hence it is possible to query for data based on specific order values. In contrary, relational databases are typically unordered and do not support the querying for columns or rows based on a certain order criteria. Mapping strategies have to define a treatment for the order information of individual XML elements. Most mapping strategies that try to implement order

```
<DB>
  FOR $tag IN DISTINCT (FOR $xxx IN document("res.xml")//*[./@type="ELEM"]
                        RETURN{<XXX>gettag($xxx)</XXX>})
  LET $elements := $root//*[name(.)=TRIM($tag/text())],
      $tagname := $tag/text()
  RETURN   <$tagname>
              FOR $elem IN $elements
              LET $atts := $elem/*[./@type="ATT"],
                  $rowtag := CONCAT($tagname, ".ROW")
              RETURN   <$rowtag>
                          <IID>$elem/@iid</IID>,
                          <ORDER>$elem/@porder</ORDER>,
                          <PID>$elem/@pid</PID>,
                          FOR $att IN $atts
                          LET $atttag := gettag($att)
                          RETURN   <$atttag>$att/CDATA/text()</$atttag>
                       </$rowtag>
           </$tagname>,
           <PCDATA>
              FOR $pcd IN $root//PCDATA
              RETURN   <PCDATA.ROW>
                          <IID>$pcd/@iid</IID>,
                          <ORDER>$pcd/@porder</ORDER>,
                          <PID>$pcd/@pid</PID>,
                          <VALUE>$pcd/text()</VALUE>
                       </PCDATA.ROW>
           </PCDATA>
</DB>
```

Figure 8: Maximum Shredding Mapping [28] in X-Cube (Generic).

handling – some of the strategies just ignore it – either define special order columns inside the data tables to store order information or create extra meta data tables for that purpose [20].

4.1.3 Attribute Element Distinction

The XML specification defines two types of data containers. Elements are ordered and allowed to be contained multiple times inside a parent element. Attributes are unordered and can only be associated once with a certain element type. As mapping strategies should be loss-less, it is necessary to also capture this distinction on the relational side. Typical relational databases have only one data container – a table cell – and hence the distinction of elements and attributes has to be treated in special ways.

4.2 XML Data Models

The mapping of XML data to a relational database requires us to determine a scheme that non-ambiguously defines a target location in the relational schema for every part of the XML data and vice versa. Most mapping strategies are based on rule-sets that determine how simple type elements, attributes and complex type elements are mapped to a relational schema.

XML is based on a tree-structured data model (ignoring IDs and IDREFs). In fact, existing mapping strategies use tree-structured data-models for representational purposes [20, 28, 6, 18, 19]. Depending on the particular mapping strategy, single edges or whole sub-trees are shredded into relational tables. This section describes the two different data models that are used by popular mapping strategies.

4.2.1 Elements and Attributes as Edges

Figure 9 shows the *Elements and Attributes as Edges* representation for the running example. Elements and attributes are represented as directed edges. Every edge that represents an element or attribute whose type is `[P]CDATA` ends in a leaf-vertex that directly contains the `[P]CDATA` value instead of an ID. The remaining vertices have been assigned a unique ID, such that a particular edge can be identified by its IDs of the start- and end-vertices respectively, or by the ID of the start-vertex and the content or ID of the end-vertex.

More formally, we represent an XML document *xml* by a directed ordered tree *G=(V, E)* – V is the set of vertices and E is the set of edges – where (∀ *elements and attributes in xml) (∃ an edge* $< v_i, v_j > \in E$).[4] Parent-child relationships, associations of attributes, and the relationship "is value of" are represented through vertices. If $v_x \in V$ (with $< v_i, v_x >, < v_x, v_j > \in E$) then either $< v_x, v_j >$ represents the sub-element of $< v_i, v_x >$ or $< v_x, v_j >$ represents an attribute of $< v_i, v_x >$, depending on the value of $type(< v_x, v_j >)$ (see below). If v_x is a leaf vertex (with $< v_i, v_x > \in E$), $value(v_x)$ (see below) returns the value of the attribute respectively element that is represented by $< v_i, v_x >$. The tree has a set of edge labels: *name*, *order*, *type*, and vertex labels *ID* and *value*:

- $name$: $< V \times V > \longrightarrow$ *String*
  Returns the name of the attribute respectively element that is represented by the argument edge.

- $order$: $< V \times V > \longrightarrow$ *Integer*
  Returns the value of the relative order of the argument edge to its siblings.

- $type$: $< V \times V > \longrightarrow$ {*ELEM, ATT*}
  Specifies if the edge represents an attribute or an element.

- ID: $V_{inner\_vertex} \longrightarrow$ *Integer*
  Returns the ID of inner vertex.

- $value$: $V_{leaf\_vertex} \longrightarrow$ *type_domains*
  Returns the value of the attribute or element if the argument is a leaf vertex.

This tree representation is simplified and hence has a major drawback: It cannot support mixed-content elements. The basic idea of mapping schemes that use this tree representation, such as *Attribute Tables*, *Edge Table*, or *Universal Table* [20] is to separately map every edge of the tree. The start-vertex's ID is referred to as the `SOURCE` and the end-vertex's content (`ID` or `[P]CDATA`) is referred to as `TARGET` of the element or attribute that is represented by the edge. When the local order of the outgoing edges of a particular vertex is kept, the tree can be reconstructed in a non-ambigious way.

4.2.2 Elements and Attributes as Vertices

Figure 10 shows the *Elements and Attributes as Vertices* representation for the running example. Elements, attributes and `[P]CDATA` values are represented as vertices. Every vertex represents three values: (1) a unique identifier (ID), (2) the type (attribute, element, `[P]CDATA`), and (3) the name or value of the attribute or element, respectively. Every leaf vertex represents either the value of an element or attribute (the name is replaced by the `[P]CDATA` value).

In more formal notation, we represent the XML document *xml* by a directed ordered tree *G=(V, E)*, where (∀ *elements, attributes, and* `[P]CDATA` *in xml) (∃ an vertex v ∈ V)*.[5] Parent-child relationships, associations of attributes,

[4] The predetermination on a tree induces that we assume that there are no recursions inside the XML data.
[5] Again, the predetermination on a tree induces that we assume that there are no recursions inside the XML data.

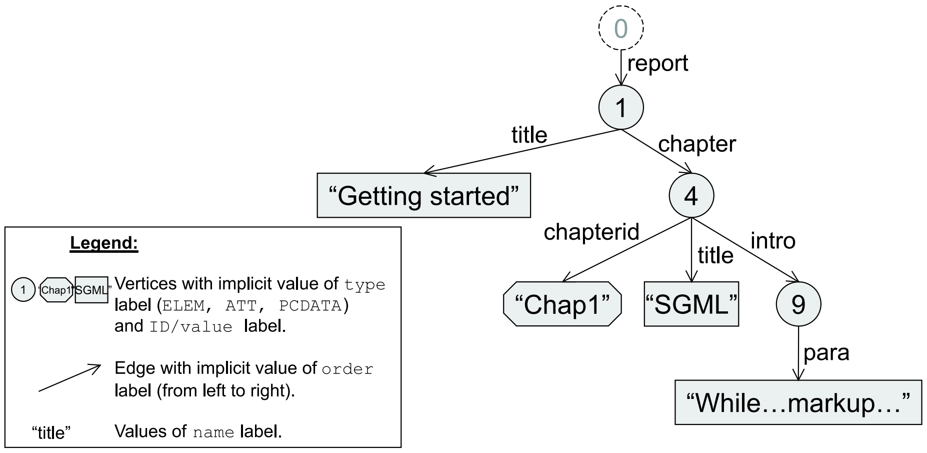

0
report
1
title
chapter
"Getting started"
4
chapterid
title
intro
"Chap1"
"SGML"
9
para
"While…markup…"
Legend:
1 Chap1 SGML
Vertices with implicit value of type label (ELEM, ATT, PCDATA) and ID/value label.
Edge with implicit value of order label (from left to right).
"title"
Values of name label.

Figure 9: Example Document in *Elements and Attributes as Edges* Representation.

and the relationship "is value of" are represented through edges. If $< v_1, v_2 > \in E$ then either v_2 represents the sub-element of v_1, v_2 represents an attribute of v_1, or v_2 stands for the value of v_1, depending on the values of the vertex labels *type(v_1)* and *type(v_2)* (see below). The tree has a set of vertex labels: *ID*, *name*, *type*, and *value* and an edge label *order*:

- ID: *V* $\longrightarrow$ *Integer*
  Returns the unique ID of the attribute, element or `[P]CDATA` that is represented by the argument vertex.
- $name$: V_{inner_vertex} $\longrightarrow$ *String*
  Returns the name of the attribute or element respectively that is represented by the argument vertex.
- $type$: *V* $\longrightarrow$ {*ELEM, ATT, [P]CDATA*}
  Specifies if the vertex represents an attribute or an element.
- $value$: V_{leaf_vertex} $\longrightarrow$ *String*
  Returns the value of the attribute or element respectively that is represented by the argument vertex.
- $order$: $< V \times V >$ $\longrightarrow$ *Integer*
  Returns the value of the relative order of the argument edge to its siblings.

The major advantage of this representation is that it can treat mixed-content elements, as element and `PCDATA` vertices can be linked to a parent-vertex in an intermixed fashion.

The basic idea of mapping strategies like *Clock* and *Inline* [28, 20] that use this tree representation is to separately map every vertex of the tree. The connection to the rest of the tree is usually captured by storing the ID value of the parent-vertex (then called *parent ID*) and the local order together with the particular element, attribute, or `[P]CDATA`.

4.3 Mapping Strategies

In the last few years, various strategies for mapping XML data to relational database systems have been proposed [20, 18, 19, 28]. This section describes five strategies (1) to show differences in strategies that are popular in the literature, (2) to illustrate the use of the two different data models, (3) to demonstrate that each strategy has its unique

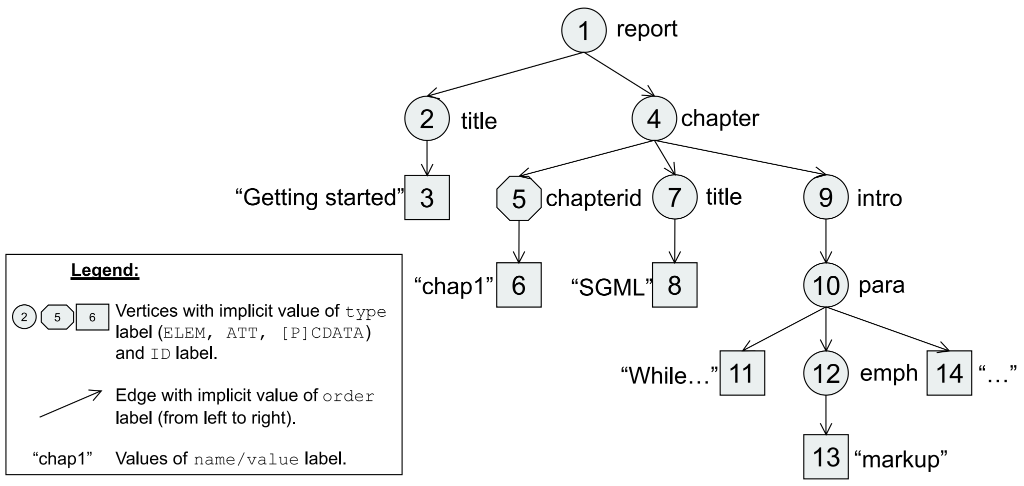

1 report
2 title
4 chapter
"Getting started" 3
5 chapterid
7 title
9 intro
"chap1" 6
"SGML" 8
10 para
"While…" 11
12 emph
14 "…"
13 "markup"
Legend:
Vertices with implicit value of type label (ELEM, ATT, [P]CDATA) and ID label.
Edge with implicit value of order label (from left to right).
"chap1" Values of name/value label.

Figure 10: Example Document in *Elements/Attributes as Nodes* Representation.

advantages for special XML fragments, and (4) give a basic understanding of different mapping strategies for later sections.

4.3.1 Edge Table

The *Edge Table* strategy [20] maps the whole XML document into one single table. It is based on the *Elements and Attributes as Edges* representation and hence does not support mixed-content elements and recursive XML structures (IDs and IDREFs). Another disadvantage of this strategy is that it does not distinguish between elements and attributes. Actually, attributes are stored as sub-elements and get a random order (usually they are inserted before the real sub-elements).

The constructed table has the following fixed structure:

```
EDGES( SOURCE Integer, ORDER Integer, NAME String, TARGET String)
```

Every edge of the data model tree is independently mapped to the table. The ID of the start-vertex is mapped to the `SOURCE` column, the ID resp. the value of the end-vertex is mapped to the `TARGET` column, the local order and the name of the edge are mapped to the `ORDER` and `NAME` column respectively. Table 1 shows the resulting Edge Table for the running example. As the Edge Table does not support mixed-content, the `<emph>` element has been ignored.

Edge			
SOURCE	ORDER	NAME	TARGET
0	1	report	1
1	1	title	Getting started
1	2	chapter	4
4	1	chapterid	chap1
4	2	title	SGML
4	3	intro	9
9	1	para	While ... markup ...

Table 1: Edge Table for the Running Example.

4.3.2 Attribute Tables

The *Attribute Tables* strategy [20] essentially splits up the Edge Table into separate tables for every element and attribute. Like the Edge Table, it is based on the *Elements and Attributes as Edges* representation and hence does not support mixed-content elements nor recursive XML structures (IDs and IDREFs). It does not distinguish between elements and attributes either. Attributes are stored in the same way as elements – in their own table – and get a random order (usually they are inserted before the real sub-elements).

The constructed tables have the following fixed structure (*$element_name* is the name of the particular element or attribute table):

```
$element_name( SOURCE Integer, ORDER Integer, TARGET String)
```

Every edge of the data model tree is independently mapped to the appropriate table (depending on its name). The ID of the start-vertex is mapped to the `SOURCE` column, the ID respectively the value of the end-vertex is mapped to the `TARGET` column, the local order of the edge is mapped to the `ORDER` column, and the name is captured by the table-name. Tables 2 and 3 show two example tables for the running example (In total there would be need for 6 tables to store the example document[6]).

report		
SOURCE	ORDER	TARGET
0	1	1

Table 2: Attribute Table *report* for the Running Example.

title		
SOURCE	ORDER	TARGET
1	1	Getting started
4	2	SGML

Table 3: Attribute Table *title* for the Running Example.

4.3.3 Universal Table

The *Universal Table* strategy [20] essentially outer joins the Attribute Tables based on the `SOURCE` and `TARGET` columns into one table. It is again based on the *Elements and Attributes as Edges* representation. Hence it has the same restrictions as the *Attribute Table* mapping. Attributes are stored in the same way as elements and again get a random order (usually they are inserted before the real sub-elements).

The constructed table has the following structure:

```
UNIVERSAL( SOURCE Integer, ELEM(1)_ORDER Integer, ELEM(1)_TARGET String, ...,
                ELEM(n)_ORDER Integer, ELEM(n)_TARGET String)
```

Every edge of the data model tree is mapped to the corresponding column of the Universal Table. The ID of the root-vertex is mapped to the `SOURCE` column, the ID resp. the value of the sub-vertices are mapped to the

[6] Assuming that the `<emph>` element would be removed, because the *Atribute Tables* approach does not support mixed-content.

corresponding `TARGET` columns, the local order of the edges is mapped to the corresponding `ORDER` columns, and the name is captured by the column names. Table 4 shows parts of the Universal Table for the running example.

Universal							
SOURCE	report_ORDER	report_TARGET	title1_ORDER	title1_TARGET	chapter_ORDER	chapter_TARGET	...
0	1	1	1	Getting started	2	4	...
...	...	...	...	...	...	...	

Table 4: Universal Table for the Running Example.

4.3.4 Full Shredding

The *Full Shredding* strategy [28] maps every element type into its own table. Attributes are mapped into the same table as their corresponding elements. The mapping strategy is based on the *Elements and Attributes as Vertices* representation and *does* support mixed-content elements *and* distinguishes between elements and attributes. However it *does not* support recursive XML structures (IDs and IDREFs), because it is again based on a tree structured data model.

The constructed tables have the following structure (*$element_name and $attribute1_name* stand for the name of the particular element respectively attribute):

```
$element_name( IID Integer, PID Integer, ORDER Integer, $attribute1_name
               String, $attribute2_name String, ...)
```

Every vertex of the data model tree is independently mapped to a table, depending on its name. The ID of the vertex is mapped to the `IID` column, the ID of the parent element is mapped to the `PID` column, the local order of the vertices is mapped to the `ORDER` column, and attribute values are directly mapped to the corresponding columns in the parent element table. `PCDATA` values of elements are stored in a separate `PCDATA` table and linked with the same `IID/PID` mechanism as used by elements. Tables 5 and 6 show two example tables for the running example.

report		
IID	PID	ORDER
1	0	1

Table 5: Full Shredding Table *report* for the Running Example.

chapter			
IID	PID	ORDER	chapterid
4	1	2	chap1

Table 6: Full Shredding Table *chapter* for the Running Example.

4.3.5 Inline

The *Inline* strategy [20] essentially uses the same strategy as the Full Shredding mapping. It maps every element into its own table, but elements whose occurence is one (or smaller than another threshold value) are *inlined* into the parent-element table. Attributes are again mapped into the same table as their corresponding elements. The mapping strategy is based on the *Elements and Attributes as Vertices* representation and – like the Full Shredding – it *does* support mixed-content elements and distinguishes between elements and attributes. However it *does not* support recursive XML structures (IDs and IDREFs), because it is again based on a tree structured data model.

The constructed tables have the following structure (*$element_name, $attribute1_name, and $inlined_element1_name* stand for the name of the particular elements and attributes respectively):

```
$element_name( IID Integer, PID Integer, ORDER Integer, $attribute1_name
String, ..., $inlined_element1_name type, $inlined_element2_name type,...)
```

Every vertex of the data model tree is independently mapped to a table or inlined into the table of the parent element. The ID of the vertex is mapped to the `IID` column, the ID of the parent element is mapped to the `PID` column, the local order of the vertices is mapped to the `ORDER` column, and attribute values are directly mapped to the corresponding columns. `PCDATA` values of elements are usually inlined. If an element contains mixed-content, the intermixed `PCDATA` is stored in a separate `PCDATA` table and linked with the same `IID/PID` mechanism used by elements. Table 7 shows an example table for the running example. To distinguish between elements and attributes, the column names end on the value of the type-function (`ELEM` or `ATT`). If an inlined element itself is a reference for other sub-elements (hence sub-elements need a ID value to reference), the column that contains the ID value ends on `IID`.

chapter					
IID	PID	ORDER	chapterid_ATT	title_ELEM	intro_para_IID
4	1	2	chap1	SGML	10

Table 7: Inline Table *chapter* for the Running Example.

5 Flexible yet Loss-Less Mapping

5.1 Loading of XML Data into a Relational Database

5.1.1 Pre-Processing - Converting Implicit Data into Explicit Data

To assure the loss-less storing of XML data inside a relational database, all mapping algorithms must store additional information (besides the raw data) in the relational database to be able to reconstruct the original XML document.[7] Basically, the stored information is data that is implicitly contained in the XML structures. Hence it consists at a minimum of (a) *unique IDs* for every element, attribute, and `PCDATA` that are used as relational keys to connect the fragment tables (via foreign key constraints) and (b) *order numbers* to record the relative order of siblings.

Based on the restrictions of a relational Database, this knowledge capture necessity is shared by all mapping schemas and hence it is a basic requirement for mapping composition. Also it would add complexity to the actual loading expression when done on the fly. As XQuery does not allow to retrieve the implicit `position()` information of an element, this information can only be extracted by using a `FOR` loop that iterates over all sub-elements on a certain nesting level to count their relative order.

X-Cube provides a *pre-processing step* that prepares the XML document for the actual loading query through converting implicit data into explicit data. Figure 11 shows parts of the running example after the pre-processing step.

[7] Remember: XML documents may be arbitrary nested and may consist of ordered sets.

```
<report type="ELEM" iid="1" pid="0" aorder="1" eorder="1" porder="1">
  <title type="ELEM" iid="2" pid="1" aorder="1" eorder="1" porder="1">
    <PCDATA type="PCDATA" iid="3" pid="2" aorder="0" eorder="0" porder="1">
      Getting started
    </PCDATA>
  </title>
  <chapter type="ELEM" iid="4" pid="1" aorder="2" eorder="2" porder="2">
    <chapterid type="ATT" iid="5" pid="4" aorder="1" eorder="0" porder="0">
      <CDATA type="CDATA" iid="6" pid="5" aorder="0" eorder="0" porder="0">
        chap1
      </CDATA>
    </chapterid>
    ...
```

Figure 11: Running Example After Pre-Processing.

The pre-processing step performs the following conversions:

- Exposing attributes as elements (with a special attribute TYPE set to ATT), so that attribute information is accessible and hence can be kept by mapping strategies that do not differentiate between elements and attributes.
- Exposing [P]CDATA as elements (with a special attribute TYPE set to [P]CDATA) for two reasons. First for the same reason of accessibility as above, second to be able to assign local order values to PCDATA (important in case of mixed-content data).
- Assignment of unique IDs to all (including new) elements (IID).
- Assignment of the direct parent's IID (PID) to all elements.
- Assignment of the relative order of sibling elements as attributes AORDER, EORDER, and PORDER.[8]

The pre-processing step is mandatory in X-Cube, because for later composition of different mapping strategies implicit and unique information like IDs, parent-child relationships, and relative order have to be explicitly available for every subtree of the initial source data. Thereafter, all mapping strategies of the Mapping Repository rely on the result of the pre-processing step as input. However, based on the particular mapping strategy, exposed attributes and/or [P]CDATA could be inlined again and dispensable order information could be dropped, if the particular mapping choice enables the capture of the desired semantics in some other fashion.

It is obvious that the conversions of the pre-processing step can be inverted easily, resulting in the original XML data. Hence the pre-processing done by X-Cube is also loss-less, what is heavily important to assure that the loading transformation as a whole is loss-less.

XQuery expressions for the actual loading after the pre-processing can be categorized along two dimensions (as in Figure 12), depending on how the mapping algorithm is modeled by the generic XQuery expression.

[8]AORDER keeps the relative order of elements and exposed attributes (used by mapping strategies that do not differentiate between elements and attributes), EORDER keeps the relative order of elements only, and PORDER keeps the relative order of elements and PCDATA (used by mixed-content sensitive mapping schemas).

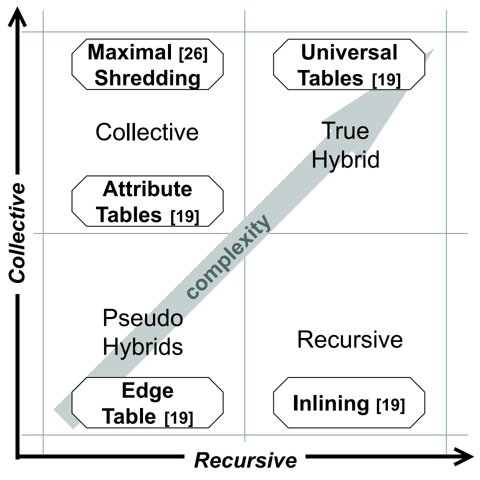


Figure 12: XQuery Expression Dimensions.

5.1.2 Collective Expressions

Several mapping strategies (e.g., Edge Table [20, 28]) treat every element (or attribute) independently from the structure and data of its context (parents, siblings, children). Those mapping strategies typically define a rule-set for the mapping process based solely on the particular element's local information. This rule-set can be applied directly on a particular element. The element's information is sufficient to directly determine how it is transformed.

```
FOR $distElem IN DISTINCT document("source.xml")//*
LET $elements := document("source.xml")//*
                 [name(.)=name($distElem)]
RETURN  FOR $element IN $elements
        RETURN Rule-Set($element)
```

Figure 13: Framework for Collective Expressions.

We refer to those mappings as *collective mappings* because they consist of an outer query in the form of a `FOR` loop that groups all distinct elements of the XML source together with an inner `LET` clause that *collects* all elements of the same type (see Figure 13). Eventually, the inner query applies the rule-set to every element in the collection one at a time, that is, the actual mapping of individual elements and attributes is completely independent.

5.1.3 Recursive Expressions

Some of the proposed mapping strategies (e.g., Inline [13, 20]) transform elements or attributes based on information of already transformed parent, child, or sibling elements, that is, the transformation is context-dependent. Hence, generic XQuery expressions have to be embedded in recursive function calls for traversing unknown XML structures (usually in breadth-first-order) and to pass information of the transformed parent-elements or siblings to sub-elements. Here, the actual rule-set is applied on both local element (or attribute) information together with information about the context that is passed (or computed) through recursive function calls (see Figure 14).

```
FUNCTION Q1($root, $parentinfo){
  LET $newparentinfo := ComputeParentInfo($root, $parentinfo)
  RETURN  FOR $child IN $root/*
          RETURN  ...
          Q1($child, $newparentinfo)
          ...
}
```

Figure 14: Framework for Recursive Expressions.

5.1.4 Pseudo Hybrids

Since recursive expressions are more powerful than collective ones, it is possible to describe collective expressions in a recursive manner. Typically the required recursive expressions are unnecessarily complex and include at least some collective part or a final ordering step to sort the result of the transformation into collections. Hence, the choice between recursive and collective expressions is not always straightforward. We call mapping strategies whose algorithm can be encoded in both recursive and collective fashion *Pseudo Hybrids*.

A typical example for a *Pseudo Hybrdid* is the Edge Table mapping [20]. A collective expression would just collect *all* elements in the source document and then transform the information of every element to the required table structure. A recursive expression would start at the root and then recursively traverse the virtual XML tree to its leaf-elements, transforming every element as it is being passed.

5.1.5 True Hybrids

Unlike Pseudo Hybrids, some mapping strategies require the use of both techniques in the loading XQuery expression. We call them *True Hybrids*.

A typical example for a *True Hybrid* is the Universal Table mapping [20]. The final table is basically the result of an outer join of attribute tables. To construct the Universal table, first attribute tables are build. Elements of the same type go into the same table, hence the required expression is collective. As a XML document can have an arbitrary amount of different element types (resulting in just as many attribute tables), the final outer join of the tables can only be done recursively.

5.2 Extraction of XML Data from a Relational Database

Section 4.2 has shown two major data models that are used by all popular mapping strategies described in the recent literature. Although both data models differ in how XML semantics are represented (by vertices or edges), both models assign unique ID values to XML elements and PCDATA and thereafter make use of those to link XML fragments. As the IDs are unique within the XML document (not only within elements or PCDATA of the same type), we distinguish between two possible ways to reconstruct the original data from a relational instance (or the Default XML View that represents it), namely schema independent or schema *de*pendent.

5.2.1 Schema Independent Extraction

The X-Cube system principally can be used in a generic manner. Hence, XML data that is already loaded in a relational database can be reconstructed solely from the information stored in the database. As X-Cube demands for mapping strategies that are loss-less, the database always (by definition) has to contain sufficient information for the correct reconstruction of the original XML data.

The generic extraction queries that we have encoded have basically a very similar framework that is independent from the categorization of the corresponding loading query (collective, recursive, hybrid). However, there is no general extraction query that could be used to extract XML data no matter what loading query has been used. Figure 15 depicts the typical framework of a generic extraction query.

```
FUNCTION Q1($root, $dxv){
  LET $elementName := getName($root),
      $elementOrder := getOrder($root)
  RETURN   <$elementName xcubesort=$elementOrder>
             FOR $childElement IN $dxv//* [./SOURCE=$root/TARGET] or [./PID=$root/IID]
             RETURN   Q1($childElement, $dxv)
                      SORTBY (@xcubesort)
           </$elementName>
}
```

Figure 15: Framework for Schema Independent Extraction Queries.

Extraction functions are recursive, because the final number of elements and their actual nesting structure cannot be determined in advance. The function is initially called with the original document root (identified by the fact that it has no parent element). Then the extraction function is called and processes the following steps:

1. Depending on the used mapping strategy, the tagname and the original relative order of the element is extracted from the Default XML View (depicted by the function `getName` and `getOrder` in Figure 15).

2. Using the retrieved information, an appropriate element is constructed.

3. Inside the new element a `FOR` loop iterates over all sub-elements that can be found (based on ID information) and calls the extraction function again to construct the found sub-elements.

4. Finally siblings are ordered based on the extracted local order information.

The major drawback of the schema independent extraction is that for every element the whole Default XML View has to be searched to determine possible sub-elements.

5.2.2 Schema Dependent Extraction

The previous section has explained the major drawback of the schema independent extraction, namely that the whole Default XML View has to be scanned for every element that is reconstructed. In cases where the original XML Schema of the loaded XML document is available, the knowledge of the XML structure that shall be reconstructed can be used to dramatically reduce the amount of the Default XML View that has to be scanned.

First, the XML Schema can help to identify leaf elements in the original XML data. As those elements do not have any sub-elements, the search for elements in the Default XML View that have matching IDs can be skipped completely. Second, the XML Schema can be used to identify possible sub-elements of a particular element and as a good set of the mapping strategies store different elements in different tables, the search space for elements with matching IDs is significantly reduced. Figure 16 shows the framework for schema independent extraction queries.

The framework for schema dependent extraction (Figure 16) differs from the one for schema independent extraction (Figure 15) only in the fact that the `FOR` loop does not search through the whole Default XML View to find

```
FUNCTION Q1($root, $dxv, $XMLschema){
  LET $elementName := getName($root),
      $elementOrder := getOrder($root),
      $candidates := getCandidates($root, $schema)
  RETURN   <$elementName xcubesort=$elementOrder>
             FOR $childElement IN $dxv/*[name(.) IN $candidates]/* [./SOURCE=$root/TARGET]
                                                               or [./PID=$root/IID]
             RETURN   Q1($childElement, $dxv, $XMLschema)
                      SORTBY (@xcubesort)
           </$elementName>
}
```

Figure 16: Framework for Schema Dependent Extraction Queries.

sub-elements of `$root` but only through a candidate list that is determined based on XML Schema information. The comparisons of computation time for the two explained extraction types has shown that the gain in performance through the use of XML Schema information can be up to a few hundred percent (depending on the schema of the XML data)!

6 Mapping Composition

Current XML mapping approaches typically consider one particular mapping strategy at a time [18, 20, 28, 19]. An exception to this are approaches like [6] that though they use a basic mapping strategy provide additional mechanisms that allow to continuously change the resulting relational structure. Simple mapping strategies have numerous restrictions concerning the semantics that can be captured, such as no mixed-content support, no attribute-element distinction, or no order-keeping. On the other side, even complex approaches may map data to relations that may not be "optimal". X-Cube allows the user to seamlessly compose multiple mapping strategies to achieve the desired semantic capture, allowing to construct relations that are suitable for the particular workload.

However, as X-Cube does not change or adjust the individual mapping strategies in any way (besides wrapping the particular XQuery expressions in a unified function call), the usability of a mapping strategy within a composition underlies the same restrictions as when used alone. This means that the following guidelines should be followed when composing mapping strategies:

- For every sub-tree of the XML document that is mapped individually, **use an appropriate mapping** (attribute-element differentiation, mixed-content treatment, etc.).
- **Never split up elements that are allowed to have mixed-content between them**.
- The XML Schema should have the root of the XML data file as first `<xs:element>`.

6.1 Unified Data Model

As already mentioned before, most existing mapping strategies use tree-structured data-models [20, 28, 6, 18, 19]. For X-Cube to be able to compose multiple mapping strategies, it must utilize a tree model that is a superset of the major alternative tree models used by different mappings. For this purpose we propose a *unified data model* that is essentially a composition of the two existing approaches from the literature that were presented in Section 4.2.

The XML document *xml* is represented by a directed ordered tree *G=(V,E)*, where ($\forall$ *elements, attributes, and* `[P]CDATA` *in xml)* ($\exists$ *a vertex* $v_i \in$ *V*).[9] Parent-child relationships, affiliations of attributes, and the relationship "is value of" (represented through edges ending in leaf-vertices) are represented through edges. If $< v_1, v_2 > \in E$ then either v_2 represents the sub-element of v_1, v_2 represents an attribute of v_1, or v_2 stands for the value of v_1, depending on the values of the vertex label *type*(v_1) and *type*(v_2) (see below). Additionally the tree has a set of vertex labels and edge labels: *name*, *order*, and *type*:

- $name^v$: $V \longrightarrow$ *String*
  $name^e$: $< V \times V > \longrightarrow$ *String*
  with $\forall$ *v* $\in$ *V*, $< x, v > \in$ *E*: $name^v$(*v*)=$name^e(< x, v >)$
- $order^v$: $V \longrightarrow$ *Integer*
  $order^e$: $< V \times V > \longrightarrow$ *Integer*
  with $\forall$ *v* $\in$ *V*, $< x, v > \in$ *E*: $order^v$(*v*)=$order^e(< x, v >)$
- $type^v$: $V \longrightarrow$ {*ELEM, ATT, [P]CDATA*}
  $type^e$: $< V \times V > \longrightarrow$ {*ELEM, ATT, [P]CDATA*}
  with $\forall$ *v* $\in$ *V*, $< x, v > \in$ *E*: $type^v$(*v*)=$type^e(< x, v >)$

Figure 17 shows the unified tree representation for the running example.

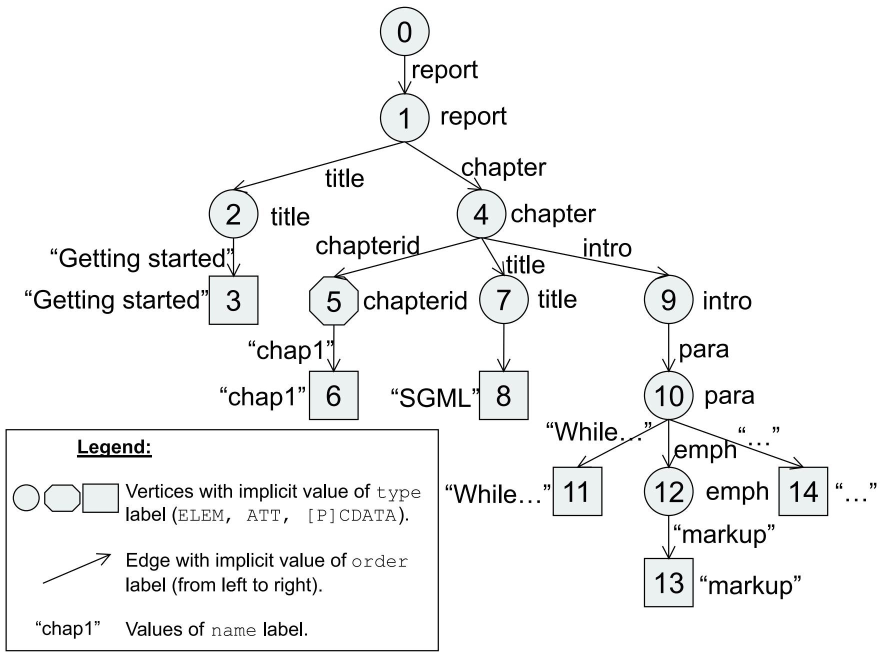


Figure 17: Data-Model for Example Document.

[9]The predetermination on a tree induces that we assume that there are no recursions inside the XML data.

6.2 Composed Loading

All mapping strategies available in X-Cube's Mapping Repository have to be encoded based on the above data model and in a generic (schema independent) fashion. They are always executed against the root of the XML document (or fragment) to be mapped.

Being a tree structure (Figure 17), every edge is a *cut set* that partitions the XML tree into two valid sub-trees. As long as it is assured that (1) the used mapping strategies are loss-less, (2) their encoding is based on the *unified data model* (defined in Section 6.1), and (3) that they store unique identification information for linking elements, we can iteratively compose mapping strategies in the following manner:

1. Split tree $G=(V,E)$ into two sub-trees $G_1=(V_1,E_1)$ and $G_2=(V_2,E_2)$ by removing the cut edge $e=\{v_1,v_2\}$, $v_1 \in V_1$ *and* $v_2 \in V_2$, so that afterwards v_2 is root of G_2.

2. Fire mapping m_1 against G_1 and m_2 against G_2 using the removed edge e as input parameter.

This method assures that finally the whole tree is mapped, because the information of the connecting edge is passed as input parameter and hence can also be stored. Assuming that the single mapping strategies have the postulated mapping properties *loss-less* and *preserving of linking information*, it is easy to see that the composed mapping is also loss-less and preserves the linking information.

In short, instead of splitting the XML data up-front into various parts that are loaded with different mapping strategies, X-Cube is thus able to execute XQuery expressions on whole XML trees. The loading expressions are extended to recognize and ignore the particular cut edge (and hence the whole linked sub-tree) when fired against G_1. Mapping m_2 is fired against the sub-tree G_2 only (and in turn can ignore sub-trees of G_2, if needed).

The above classification of the individual mapping strategies (recursive, collective, true or pseudo hybrid) determines how the XQuery has to be extended to skip those parts of the XML document that shall not be mapped with the current mapping strategy.

6.2.1 Order Shift

The variety of proposed mapping strategies and their particular restrictive assumptions raise a specific problem regarding order handling when composing different mappings. Depending on attribute-element differentiation and support of mixed-content elements by the selected mapping strategies, the order number that is finally stored in the relational database – after mapping with the composed strategy – may be shifted. For example, if a particular mapping strategy does not differentiate between attributes and elements (and hence inserts attributes as sub-elements), clearly the overall order of elements is shifted by the number of existing attributes that are now treated (and counted) as sub-elements.

To support the different ways of order-counting, the pre-processing step features several order values that facilitate the determination of the correct order value for different mapping strategies (see Section 5.1.1 for details). When using only one mapping strategy, the order shift problem does not occur because every mapping strategy constantly uses one specific order number (e.g., `aorder`) and those numbers are consistent.

The problem gets obvious when composing mapping strategies that use different order enumerations for the transformation of sibling elements: The first sub-element's mapping strategy may use the `aorder` value of the pre-processing step. Hence, it induces an order shift based on the number of existing attributes. Say the second sub-element's mapping strategy uses the `porder` value, that is, the order is *not* shifted. Treated as a whole, the correct ordering of the sub-elements is lost (see Figure 18).

To deal with this problem, we enhance every single mapping expression of the repository with an additional argument set consisting of *order_shift* and *shift_list* parameters. When composing different mappings, one can specify

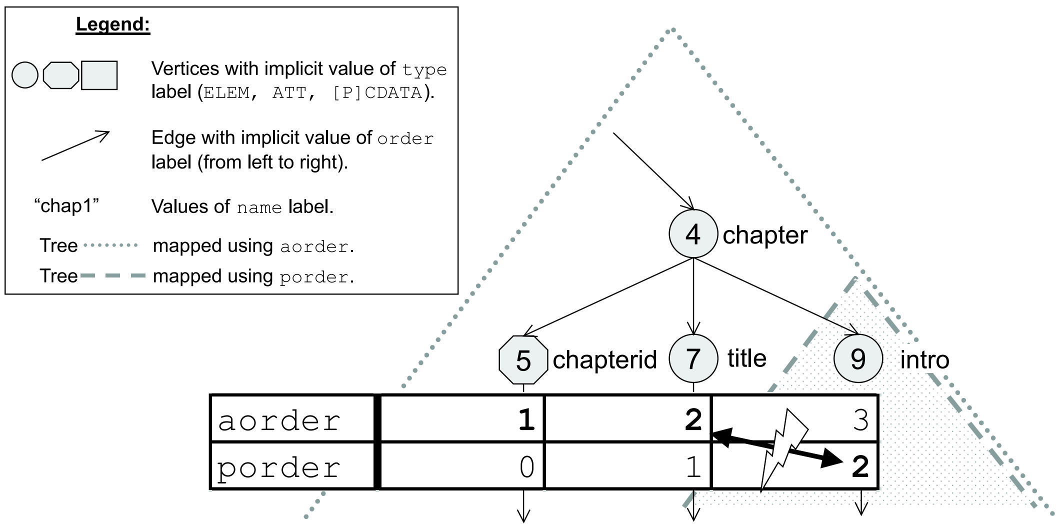

Legend:
Vertices with implicit value of type label (ELEM, ATT, [P]CDATA).
Edge with implicit value of order label (from left to right).
"chap1" Values of name label.
Tree mapped using aorder.
Tree mapped using porder.
4 chapter
5 chapterid
7 title
9 intro
aorder 1 2 3
porder 0 1 2

Figure 18: Order Shift Problem.

the particular element(s) where the order shift problem occurs. When defining what particular element shall be mapped using what strategy, the user can either manually correct the order shift for the selected mapping in form of an integer value, or use an XQuery expression for computing the appropriate shift on the fly.

6.3 Composed Extraction

Section 5.2 described that the extraction process can be done in a full-recursive (strategy independent) and a strategy-recursive (strategy dependent) manner. Based on the performance gains achieved when using XML Schema information, the mapping composition is implemented strategy dependent, although principally it could be implemented full-recursive as well.

With the loading composition, for any element the user has already specified what particular mapping was used for the loading transformation (and hence has to be used for the extraction). The composed extraction expression then consists of a huge wrapper function that contains an extraction XQuery for every mapping strategy that is supported. Based on the user input, an `IF THEN ELSE` chain navigates to the appropriate extraction expression for a particular element and extracts it. With help of the XML Schema, all (potential) sub-elements of the just extracted element are identified. The wrapper is then called recursively for those potential sub-elements with the `IID` of the last extracted element as additional argument. Based on the passed information, again the appropriate mapping strategy can be identified and the formerly passed `IID` then serves as `PID` to locate the needed information for the element to be extracted. Figure 19 shows the framework for the composed extraction.

Looking at the unified data model (Section 6.1), the composed extraction has to reconstruct the original XML tree with the hindrance that every edge or vertex might have to be reconstructed differently – depending on the mapping strategy that was employed during the loading. The root vertex is easily identified by the qualification `ID=0`. The XML Schema then identifies (potential) outgoing edges and their following vertices. Finally the user input associates a particular mapping strategy to them. The extraction function can then extract the needed information from the appropriate tables based on the identified components and the mapping strategy that was used and reconstruct the next level of vertices and edges.

```
FUNCTION Extract($pid, $root, $dxv, $strategy){
    LET $mapping := GetMapping($root)
    RETURN   IF ( $mapping="Edge" )
             THEN   ...
             ELSE IF ( $mapping="Attribute" )
                  THEN   LET $data := Retrieve-Attribute($pid, $root, $dxv),
                             $potential := Identify-Potential-SubElements($root,
                                                                            $strategy),
                             $iid := GetIID($data)
                         RETURN   ReconstructElement($data),
                                  FOR $element IN $potential
                                  RETURN   Extract($iid, $element, $dxv, $strategy)
                  ELSE IF  ...
}

Extract("0", Identify-Root(document("strategy.xsd")), document("dxv.xml"),
        document("strategy.xsd"))
```

Figure 19: XQuery Framework for Composed Extraction.

7 XQuery Instantiation for Extraction

X-Cube's uses generic XQuery expressions to make the system flexible and the mapping repository usable for all types of XML documents, independent from their particular schema. To map the XML document correctly, schema information has to be considered and – when using generic queries – has to be evaluated at runtime. This leads to a major drawback of X-Cube's flexibility: All queries of the mapping repository span over two data sources, (1) the XML data file and (2) the corresponding XML Schema file. Section 8 shows that there is indeed a trade-off between the high flexibility and transformation time. In cases where all XML documents to be loaded conform to one basic XML Schema the flexibility might well be traded against performance gains during the transformation.

For such purposes, we propose a way to instantiate mapping queries of X-Cube's query repository, again using only off-the-shelf tools (XQuery, XQueryX, and a standard XQuery engine).

7.1 Query Transformation with XQueryX

In Section 2.4 we have introduced the new XML syntax for XQuery expressions called XQueryX. For the instantiation process, X-Cube represents the XQuery in XQueryX format, achieving similar advantages as through the use of XQuery as mapping language:

- XQuery is powerful enough to serve as *description language* for the instantiation algorithm.
- The algorithm is *not hard-coded* (as when using a programming language) and can be adjusted and extended in an easy way.
- The defined XQuery expression can be used to directly *execute* the actual instantiation process.
- There is *no need for separate implementation* of an instantiation module.
- Even the instantiation process can profit from *future* XQuery *optimization techniques*.

The following sections describe what parts of a typical mapping expression can easily be considered while instantiating and how this can be done using XQueryX.

7.2 Recursive Function Replacement

An extensive study of recursive mapping expressions has shown that the recursive call respectively the base case of a recursive function is usually bound to the existence of further sub-elements of the current `$root` element and hence contained in a `FOR $child IN $root/*` loop. The argument of the recursive function call is typically the `$child` variable. That is why the replacement of a recursive function call consists of two major steps, namely the `FOR`-loop replacement and the recursive function call replacement.

The **first step** replaces the `FOR` loop depending on the XML Schema of the XML instance. We distinguish between two different cases:

- The current element has no further sub-elements. Hence, the loop and the inner expression (including the recursive function call) can be completely removed.
- The XML Schema indicates the existence of further sub-elements. In this case the loop and its inner expressions are replaced depending on the potential occurrences of the sub-elements:
  - For all sub-elements with `maxOccurence` $\leq$ `1` (including optional occurence), the `FOR` loop is removed and the particular sub-element identifier replaces the binding variable of the `FOR` loop in the inner expression.
  - For elements with an unbounded `maxOccurrence`, the `*`-quantifier of the `FOR` loop is replaced with the proper element identifier. In cases where multiple sub-elements with `maxOccurrence="unbounded"` exist, it is important to keep track of proper variable renaming during the replacement process.

Table 8 depicts the described instantiation cases using examples.

```
...
   FOR $child IN $root/*
   RETURN  innerExpression($child)
...
```

`$root` has no further sub-elements (no `$child` elements).	`FOR` loop is removed.	`...`
		`   - - -`
		`...`

`$child` has a `maxOccurence` of one.	`$child` is replaced by identifier (from XML Schema).	`...`
		`   innerExpression($root/title)`
		`...`

`$child`'s `maxOccurence` is unbounded (or bigger than threshold).	Replace `*`-quantifier with proper element identifier.	`...`
		`   FOR $child IN $root/intro`
		`   RETURN  innerExpression($child)`
		`...`

Mixed `FOR` loop: `title` with `maxOccurence=1` and `intro` with `maxOccurence=unbounded`.	Mix the above cases. Keep track of proper variable renaming.	`...`
		`   innerExpression($root/title),`
		`   FOR $var1 IN $root/intro`
		`   RETURN  innerExpression($var1)`
		`...`

Table 8: `FOR` Loop Replacement Rules.

The **second step** replaces the recursive function call that is part of the `FOR` loop's innerExpression. During the `FOR` loop replacement, a particular element (the sub-elements of `$root`) is detected as the argument for the recursive function call. Hence, the replacement of the recursive function call is done by replacing the recursive function call with the actual query of the function called and replacing the argument variable with the new replacement value of `$child` of step one. Additionally (although not always necessary) all variables inside the recursive query that were copied should be renamed to avoid potential name conflicts.

7.3 Non-recursive Function Replacement

Although the instantiation process is dependent on XML Schema information (the particular schema for that the instantiation shall be done), the replacement of non-recursive function calls can be done schema-independently. Whenever a function call in the XQuery expression is discovered, it is replaced with the complete body of the particular function. For easier variable handling (prevention of name-conflicts), our system replaces all variables besides the function arguments that are used in the function body with standard names. Before the actual replacement, the argument variables of the function have to be replaced throughout the whole function body with the variables that were used for the function call. If the function to be replaced contains further function calls, those can be replaced in the same way, but as the schema-independent replacement procedure can only be used for non-recursive functions, non-recursive function calls that result in call-circles cannot be treated either!

7.4 Schema Query Replacement

Section 7.2 described the replacement of recursive function calls and hand in hand with it the replacement of schema dependent `FOR` loops. The analysis of generic query expressions and the resulting schema dependent instances have shown that after the recursive function call is replaced, there are various expressions left that query for schema information only and – as we generated an instantiated expression – could easily be replaced or removed (see following table for typical examples).

Example Schema Query	**Replacement/Action**
`LET $var := gettag($root/title)`	Replace `$var` with `title` in following sub-expression.
`IF ($root/title) THEN ... ELSE ...`	Remove `IF` clause. Keep the proper part, depending on the existence of `$root/title`.
`<chapter $var/chapter/@* />`	List all attributes (`chapterid`) of `$var/chapter`.
`...`	...

Table 9: Examples of Schema Query Replacement Rules.

Although the replacement of those expressions is not necessary for the usability of the extraction query, it may influence the performance of the system, as a lot of unnecessary evaluations could be avoided. However, intelligent XQuery engines may discover those expressions on their own so that the impact on the overall performance might be minimal. A second aspect is that instantiated expressions may contain huge parts (especially inside `IF` clauses) that are never used, but simply enlarge the XQuery and may slow down the parsing process and impact the readability of the query.

8 Experimental Evaluation

We have implemented the X-Cube mapping manager based on the XQuery Engine *Kweelt* [2] and run various tests on Windows 98 with 128 MB memory and PIII 800 cpu. We have run various different data sets to evaluate the system in terms of correctness and performance. For all sets the following times were calculated:

- **Loading Transformation Time:** The time in milliseconds that is needed for the complete transformation of the document to the desired XML Default View plus the time that is required for the transformation of the XML Schema to the Default XML Schema that corresponds to the produced XML Default View.[10]
- **Loading Transformation Time in Composition:** The time in milliseconds that is needed for the complete transformation of the document to the desired XML Default View plus the time that is required for the transformation of the XML Schema to the Default XML Schema that corresponds to the produced XML Default View. In this case the composition module was used to perform the transformation although the whole document was loaded using only one mapping strategy. This way we can show the overhead time caused by the composition module itself.
- **Extraction Transformation Time (Schema Dependent):** The time in milliseconds that is needed to extract the data from the Default XML View to the original XML format, assuming that there *is* target schema information available that can be used to speed up the search process.[11]
- **Extraction Transformation Time in Composition (Schema Dependent):** The time in milliseconds that is needed to extract the data from the Default XML View to the original XML format, assuming that there *is* target schema information available that can be used to speed up the search process. In this case the composition module was used to perform the transformation although the whole document was extracted using only one mapping strategy. This way we can show the overhead time caused by the composition module itself.

Mac Beth Data Set. Results shown are based on the loading of "Mac Beth" of the Shakespeare 2.0 XML collection [12]. The XML Schema has 21 different element types, no attributes, and the data set has a sum of 3,975 elements with the deepest nesting level of 6. Table 10 shows some benchmark numbers and Figure 20 shows the comparison of the times for loading and extraction.[13]

Mapping	Attribute Tables	Full Shredding	Edge Table	Inline
Sum of Record Sets	3,963	7,262	3,963	6,353
Sum of Attributes	48	52	4	47
Number of Tables	16	17	1	12
Size Default XML View [byte]	877,942	1,534,165	1,051,089	1,532,941
Loading Time [ms]	63,042	86,400	17,310	954,245
Loading Time in Composition [ms]	69,320	147,810	28,930	1,032,870
Extraction Time (Schema Dependent) [ms]	303,623	1,034,774	1,002,059	921,504
Extraction Time in Composition (Schema Dependent) [ms]	460,940	1,153,270	1,018,810	977,630

Table 10: Loading and Extraction Results for the "Mac Beth" XML Data Set.

X Mark Data Set. Results shown are based on the loading of a 320KB XML file generated with the "X Mark" generator[14]. The XML Schema has 21 different element types, 3 attributes, and the data set has a sum of 3,213

[10] Every case was run 10 times to get representative loading times.
[11] Every case was run 10 times to get representative loading times.
[12] http://metalab.unc.edu/bosak/xml/eg/shaks200.zip
[13] For the test of the Edge Table and Attribute Tables mapping strategies the mixed-content inside `LINE` was removed lowering the total amount of elements to 3,963 (those mapping strategies do not support mixed-content).
[14] http://monetdb.cwi.nl/xml/

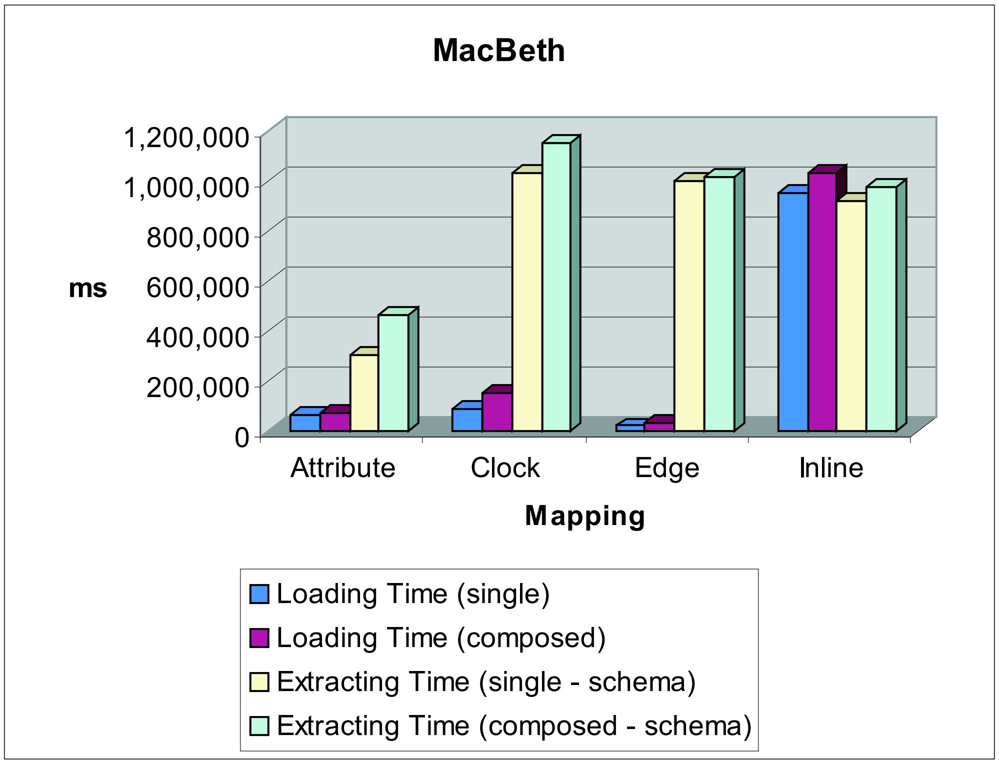


Figure 20: Loading and Extraction Time Comparison for MacBeth Data Set.

elements with the deepest nesting level of 10. Table 11 shows some benchmark numbers and Figure 21 shows the comparison of the times for loading and extraction.[15]

Mapping	Attribute Tables	Full Shredding	Edge Table	Inline
Sum of Record Sets	2,700	5,834	2,700	3,745
Sum of Attributes	66	72	4	49
Number of Tables	22	23	1	11
Size Default XML View [byte]	864,248	1,485,626	993,763	1,343,978
Loading Time [ms]	41,200	72,659	13,957	575,024
Loading Time in Composition [ms]	47,350	125,350	23,740	646,430
Extraction Time (Schema Dependent) [ms]	99,900	630,167	556,008	311,628
Extraction Time in Composition (Schema Dependent) [ms]	153,190	823,650	766,870	456,060

Table 11: Loading and Extraction Results for the “X Mark” XML Data Set.

Mapping Composition. This section employs a “home made” example document that is small enough so that we can show complete relational storage representations for it and “big” enough to demonstrate the advantages and the feasibility of mapping composition. The goal in this section is not to show performance numbers, but to demonstrate

[15] For the test of the Edge Table and Attribute Tables mapping strategies the mixed-content inside `text` was removed lowering the total amount of elements to 2,700 (those strategies do not support mixed-content).

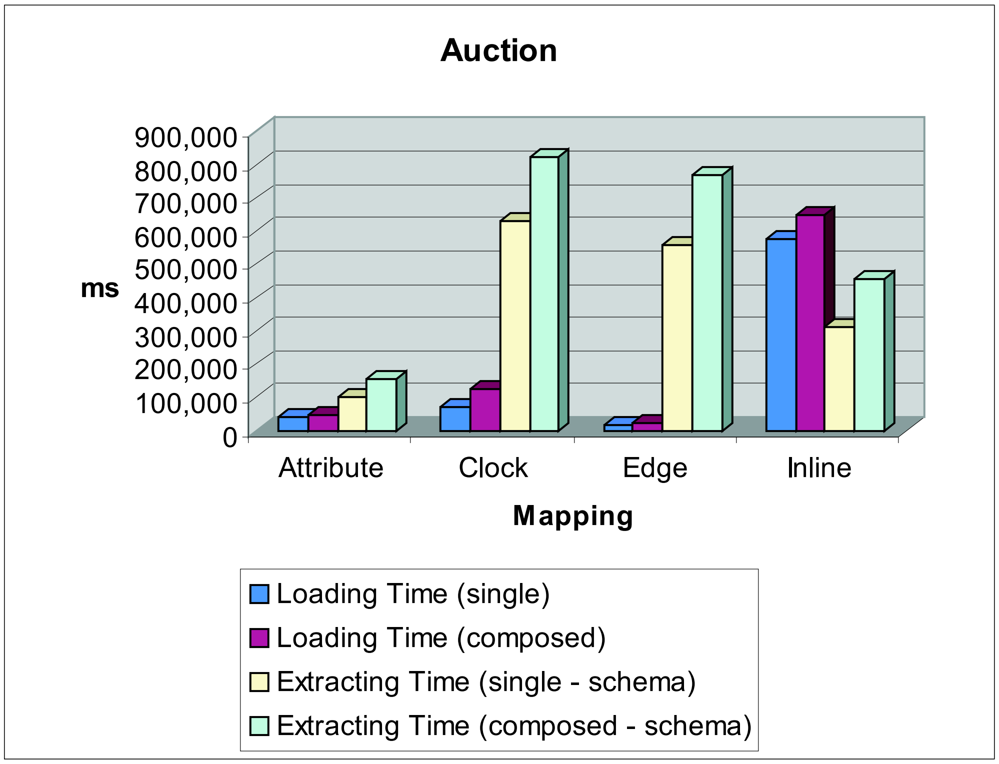


Figure 21: Loading and Extraction Time Comparison for MacBeth Data Set.

(a) that the presented composition approach really works and (b) how composition could be used to construct a possibly more desired relational representations for XML data.

Figure 22 shows an example XML document (BOOKLIST) that we will use to demonstrate the feasibility of mapping composition. Figure 23 shows the resulting relational structure when using the *Inline* mapping for the whole document. Figure 24 shows one example of a relational structure that can easily be achieved by combining mapping strategies. Here the `AUTHOR` and `NAME` elements are kept together in an *Edge Table*. This way queries that search for authors that wrote a book together do not require a join between two tables. However, this gives us just the overall notion of how combining multiple mapping strategies may work. In X-Cube all combinations are possible and so there is virtually no limit in designing target relations.

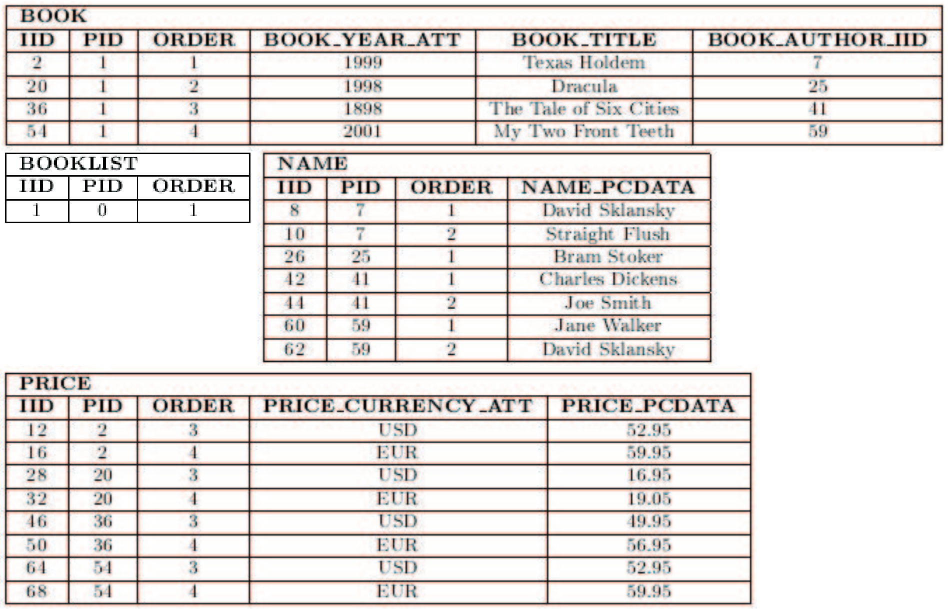
9 Conclusion

Although a lot of research has been done in terms of mapping XML data to relational databases, most approaches are still bound to one particular mapping strategy that is typically hard-coded inside the particular system. More flexible approaches often are based on a proprietary mapping language that has to be used in order to specify the desired mapping strategy. The mapping process is often still tied to one single mapping strategy. Even commercial systems do not provide the necessary flexibility for specifying mappings that map XML data to (for the particular case) "most optimal" relational representations.

Recent research (like XPERANTO or Rainbow) has proposed new middle layer systems that can be used on top

```
<BOOKLIST>
   <BOOK YEAR="1999">
      <TITLE>Texas Holdem</TITLE>
      <AUTHOR>
         <NAME>David Sklansky</NAME>
         <NAME>Straight Flush</NAME>
      </AUTHOR>
      <PRICE CURRENCY="USD">52.95</PRICE>
      <PRICE CURRENCY="EUR">59.95</PRICE>
   </BOOK>
   <BOOK YEAR="1998">
      <TITLE>Dracula</TITLE>
      <AUTHOR>
         <NAME>Bram Stoker</NAME>
      </AUTHOR>
      <PRICE CURRENCY="USD">16.95</PRICE>
      <PRICE CURRENCY="EUR">19.05</PRICE>
   </BOOK>
   <BOOK YEAR="1898">
      <TITLE>The Tale of Six Cities</TITLE>
      <AUTHOR>
         <NAME>Charles Dickens</NAME>
         <NAME>Joe Smith</NAME>
      </AUTHOR>
      <PRICE CURRENCY="USD">49.95</PRICE>
      <PRICE CURRENCY="EUR">56.95</PRICE>
   </BOOK>
   <BOOK YEAR="2001">
      <TITLE>My Two Front Teeth</TITLE>
      <AUTHOR>
         <NAME>Jane Walker</NAME>
         <NAME>David Sklansky</NAME>
      </AUTHOR>
      <PRICE CURRENCY="USD">52.95</PRICE>
      <PRICE CURRENCY="EUR">59.95</PRICE>
   </BOOK>
</BOOKLIST>
```

Figure 22: BOOKLIST Example.



BOOK

IID	PID	ORDER	BOOK_YEAR_ATT	BOOK_TITLE	BOOK_AUTHOR_IID
2	1	1	1999	Texas Holdem	7
20	1	2	1998	Dracula	25
36	1	3	1898	The Tale of Six Cities	41
54	1	4	2001	My Two Front Teeth	59

BOOKLIST

IID	PID	ORDER
1	0	1

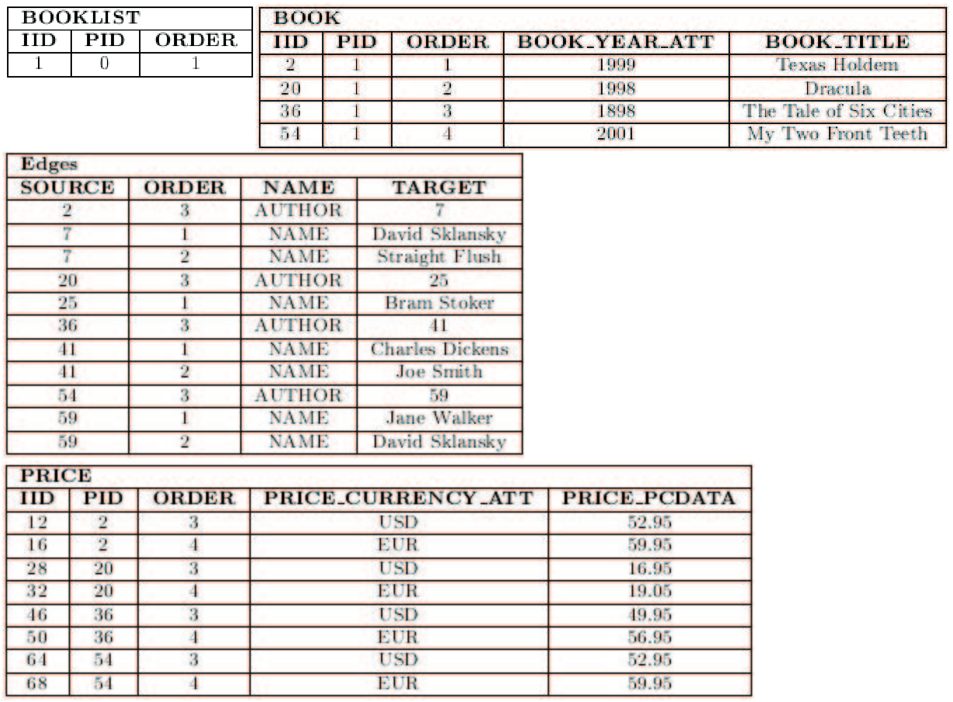
NAME

IID	PID	ORDER	NAME_PCDATA
8	7	1	David Sklansky
10	7	2	Straight Flush
26	25	1	Bram Stoker
42	41	1	Charles Dickens
44	41	2	Joe Smith
60	59	1	Jane Walker
62	59	2	David Sklansky

PRICE

IID	PID	ORDER	PRICE_CURRENCY_ATT	PRICE_PCDATA
12	2	3	USD	52.95
16	2	4	EUR	59.95
28	20	3	USD	16.95
32	20	4	EUR	19.05
46	36	3	USD	49.95
50	36	4	EUR	56.95
64	54	3	USD	52.95
68	54	4	EUR	59.95

Figure 23: Relational Structure for Example of Figure 22 after the Inline Mapping.



BOOKLIST		
IID	PID	ORDER
1	0	1

BOOK				
IID	PID	ORDER	BOOK_YEAR_ATT	BOOK_TITLE
2	1	1	1999	Texas Holdem
20	1	2	1998	Dracula
36	1	3	1898	The Tale of Six Cities
54	1	4	2001	My Two Front Teeth

Edges			
SOURCE	ORDER	NAME	TARGET
2	3	AUTHOR	7
7	1	NAME	David Sklansky
7	2	NAME	Straight Flush
20	3	AUTHOR	25
25	1	NAME	Bram Stoker
36	3	AUTHOR	41
41	1	NAME	Charles Dickens
41	2	NAME	Joe Smith
54	3	AUTHOR	59
59	1	NAME	Jane Walker
59	2	NAME	David Sklansky

PRICE				
IID	PID	ORDER	PRICE_CURRENCY_ATT	PRICE_PCDATA
12	2	3	USD	52.95
16	2	4	EUR	59.95
28	20	3	USD	16.95
32	20	4	EUR	19.05
46	36	3	USD	49.95
50	36	4	EUR	56.95
64	54	3	USD	52.95
68	54	4	EUR	59.95

Figure 24: Relational Structure for Example of Figure 22 after a Composed Mapping with Inline and Edge Table Strategies.

of any relational databases to query stored XML data. In conjunction with X-Cube, those systems can be used to *store and query* XML data inside relational databases in a flexible way. X-Cube's mapping XQuery expressions can directly be used by those middle layer systems to query the data stored inside the relational database even when *composing different mapping strategies*.

At the same time, the presented approach is *declarative* enough to describe all proposed mapping strategies in a *schema independent* way. The use of XQuery as mapping language provides X-Cube with the possibility to *directly execute* the encoded transformations using *off-the-shelf technology* only. This makes it easy to *port and extend* the system for the use with new mapping strategies and other storage systems and likely will even lead to future optimization when XQuery engines are improved.

References

[1] ISO 8879. Information processing - text and office systems - standardized generalized markup language (sgml), 1986.

[2] A. Sahuguet. Querying XML in the New Millennium. http://db.cis.upenn.edu/Kweelt, September 2000.

[3] Serge Abiteboul. Querying semi-structured data. In *Proc. of the Int. Conf. on Database Theory (ICDT)*, Delphi, Greece, 1997.

[4] B-Bop Associates, Inc. B-Bop Xfinity Server V. 2.0. http://www.b-bop.com/products_xfinity_server.htm, 2002.

[5] S. Boag, D. Chamberlin, M. F. Fernandez, D. Florescu, J. Robie, J. Siméon, and M. Stefanescu. XQuery 1.0: An XML Query Language. http://www.w3.org/TR/2001/WD-xquery-20011220, December 2001.

[6] Philip Bohannon, Juliana Freire, Prasan Roy, and Jme Simeon. From XML Schema to Relations: A Cost-based Approach to XML Storage. In *IEEE Int. Conf. on Data Engineering*, 2002.

[7] J. Bosak, T. Bray, D. Connolly, E. Maler, G. Nicol, C. M. Sperberg-McQueen, L. Wood, and J. Clark. Guide to the W3C XML Specification ("XMLspec") DTD, Version 2.1. http://www.w3.org/XML/1998/06/xmlspec-report-v21.htm, June 1998.

[8] Ronald Bourret, Christoph Bornhöved, and Alejandro P. Buchmann. A Generic Load/Extract Utility for Data Transfer Between XML Documents and Relational Databases. Technical Report DVS99-1, Darmstadt University of Technology, December 1999.

[9] T. Bray, C. Frankston, and A. Malhotra. Document Content Description for XML. http://www.w3.org/TR/NOTE-dcd, July 1998.

[10] T. Bray, J. Paoli, C. M. Sperberg-McQueen, and E. Maler. Extensible Markup Language (XML) 1.0 (Second Edition). http://www.w3c.org/TR/2000/REC-xml-20001006/, October 2000.

[11] A. Brown, M. Fuchs, J. Robie, and P. Wadler. XML Schema: Formal Description. http://www.w3c.org/TR/2001/WD-xmlschema-formal-20010320/, March 2001.

[12] Peter Buneman. Semistructured Data. In *Proc. of the ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems (PODS)*, pages 117–121, Tucson, Arizona, 1997.

[13] Michael J. Carey, Jerry Kiernan, Jayavel Shanmugasundaram, Eugene J. Shekita, and Subbu N. Subramanian. XPERANTO: Middleware for Publishing Object-Relational Data as XML Documents. In *The VLDB Journal*, pages 646–648, 2000.

[14] D. Chamberlin, P. Fankhauser, M. Marchiori, and J. Robie. XML Query Use Cases. http://www.w3.org/TR/2001/WD-xmlquery-use-cases-20011220, December 2001.

[15] D. Chamberlin, J. Robie, and D. Florescu. Quilt: An XML Query Language for Heterogeneous Data Sources. In Springer-Verlag, editor, *Proc. of WebDB 2000 Conference, in Lecture Notes in Computer Science*, 2000.

[16] J. Clark and S. DeRose. XML Path Language (XPath) Version 1.0. http://www.w3.org/TR/1999/REC-xpath-19991116, November 1999.

[17] DB2 UDB XML Extender. XML Extender Administration and Programming. http://www-4.ibm.com/software/data/db2/extenders/
xmlext/library.html, December 1999.

[18] Alin Deutsch, Mary F. Fernandez, and Dan Suciu. Storing Semistructured Data with STORED. In *Proceedings of ACM SIGMOD International Conference on Management of Data*, pages 431–442, Philadephia, USA, June 1999.

[19] Mary Fernandez, Wang-Chiew Tan, and Dan Suciu. SilkRoute: Trading between Relations and XML. http://www.www9.org/w9cdrom/202/202.html, May 2000.

[20] D. Florescu and D. Kossman. Storing and Querying XML Data using an RDBMS. In *IEEE Data Engineering Bulletin*, Vol. 22, No. 3, 1999.

[21] IXIA, Inc. TeXtML Server - The Native XML Database. http://www.ixiasoft.com/products/textmlserver, 2002.

[22] A. Malhotra, J. Robie, and M. Rys. XML Syntax for XQuery 1.0 (XQueryX). http://www.w3.org/TR/2001/WD-xqueryx-20010607, June 2001.

[23] Microsoft Inc. Microsoft SQL Server. http://www.microsoft.com/sql/default.asp.

[24] Oracle. Oracle XML SQL Utility for Java. http://technet.oracle.com/tech/xml/oracle_xsu/, 1999.

[25] J. Shanmugasundaram, G. He, K. Tufte, C. Zhang, D. DeWitt, and J. Naughton. Relational Databases for Querying XML Documents: Limitations and Opportunities. In *Proceedings of 25th International Conference on Very Large Data Bases (VLDB'99), Edinburgh, Scotland, UK*, pages 302–314, September 1999.

[26] Software AG. Quip Ver. 2.1. http://www.softwareag.com/developer/quip/download.htm, 2002.

[27] Software AG. Tamino XML Server Ver. 3.1. http://www.softwareag.com/tamino/default.htm, 2002.

[28] X. Zhang, G. Mitchell, W. Lee, and E. A. Rundensteiner. Clock: Synchronizing Internal Relational Storage with External XML Documents. In *RIDE-DM*, pages 111–118, April 2001.

[29] X. Zhang and E. A. Rundensteiner. Rainbow: Bridge over Gap of XML and Relational. Technical report, Worcester Polytechnic Institute, 2001. in progress.

A Appendix 1: Pre Processing

A.1 The Example XML Document

```
    <report>
      <title>Getting started</title>
      <chapter chapterid="chap1">
        <title>SGML</title>
        <intro>
          <para>While...<emph>markup</emph>...</para>
        </intro>
      </chapter>
    </report>
```

A.2 XML Schema for the Example Document

```
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema" elementFormDefault="qualified">
 <xs:element name="report">
  <xs:complexType>
   <xs:sequence>
    <xs:element name="title" ref="title"/>
    <xs:element name="chapter">
     <xs:complexType>
      <xs:sequence>
       <xs:element name="title" ref="title"/>
       <xs:element name="intro">
        <xs:complexType>
         <xs:sequence>
          <xs:element name="para">
           <xs:complexType mixed="true">
            <xs:choice minOccurs="0" maxOccurs="unbounded">
             <xs:element name="emph" type="xs:string"/>
            </xs:choice>
           </xs:complexType>
          </xs:element>
         </xs:sequence>
        </xs:complexType>
       </xs:element>
      </xs:sequence>
      <xs:attribute name="chapterid" type="xs:string" use="required"/>
     </xs:complexType>
    </xs:element>
   </xs:sequence>
  </xs:complexType>
 </xs:element>
 <xs:element name="title" type="xs:string"/>
</xs:schema>
```

A.3 The Pre-Processing Query

```
FUNCTION Q1($root, $pid, $aorder1, $eorder1, $porder1){
 LET $maintag := gettag($root),
     $iid := getiid(),
     $aorder2 := 0,
     $eorder2 := 0,
     $porder2 := 0,
     $pos := 0
 RETURN   <$maintag type="ELEM" iid=$iid pid=$pid aorder=$aorder1 eorder=$eorder1 porder=$porder1>
             FOR $attribute IN $root/@*
             LET $atttag := gettag($attribute),
                 $attiid := getiid(),
                 $pcdataiid := getiid(),
                 $aorder2 := $aorder2 + 1
             RETURN   <$atttag type="ATT" iid=$attiid pid=$iid aorder=$aorder2 eorder="0" porder="0">
                       <CDATA type="CDATA" iid=$pcdataiid pid=$attiid aorder="0" eorder="0" porder="0">
                        $attribute
                       </CDATA>
                      </$atttag>,
                      FOR $elem IN $root/*
                      LET $pos := $pos + 1,
                          $pcd := $root/text()[position()=$pos]
                      RETURN   IF (TRIM($pcd)="")
                               THEN   ""
                               ELSE   LET $pcdataiid := getiid(),
                                          $porder2 := $porder2 + 1
                                      RETURN  <PCDATA type="PCDATA"
                                                      iid=$pcdataiid
                                                      pid=$iid
                                                      aorder="0"
                                                      eorder="0"
                                                      porder=$porder2>
                                                TRIM($pcd)
                                              </PCDATA>
                               LET $aorder2 := $aorder2 + 1,
                                   $eorder2 := $eorder2 + 1,
                                   $porder2 := $porder2 + 1
                               RETURN   Q1($elem, $iid, $aorder2,
                                           $eorder2, $porder2)
                      LET $pos := $pos + 1,
                          $pcd := $root/text()[position()=$pos]
                      RETURN   IF (TRIM($pcd)="")
                               THEN   ""
                               ELSE   LET $pcdataiid := getiid(),
                                          $porder2 := $porder2 + 1
                                      RETURN   <PCDATA type="PCDATA"
                                                       iid=$pcdataiid
                                                       pid=$iid
                                                       aorder=$aorder2
                                                       eorder=$eorder2
                                                       porder=$porder2>
                                                 TRIM($pcd)
                                               </PCDATA>
          </$maintag>
}

Q1(document("source.xml"), 0, 1, 1, 1)
```

A.4 The Running Example After Pre-Processing

```
<report type="ELEM" iid="1" pid="0" aorder="1" eorder="1" porder="1">
  <title type="ELEM" iid="2" pid="1" aorder="1" eorder="1" porder="1">
    <PCDATA type="PCDATA" iid="3" pid="2" aorder="0" eorder="0" porder="1">
      Getting started
    </PCDATA>
  </title>
  <chapter type="ELEM" iid="4" pid="1" aorder="2" eorder="2" porder="2">
    <chapterid type="ATT" iid="5" pid="4" aorder="1" eorder="0" porder="0">
      <CDATA type="CDATA" iid="6" pid="5" aorder="0" eorder="0" porder="0">
        chap1
      </CDATA>
    </chapterid>
    <title type="ELEM" iid="7" pid="4" aorder="2" eorder="1" porder="1">
      <PCDATA type="PCDATA" iid="8" pid="7" aorder="0" eorder="0" porder="1">
        SGML
      </PCDATA>
    </title>
    <intro type="ELEM" iid="9" pid="4" aorder="3" eorder="2" porder="2">
      <para type="ELEM" iid="10" pid="9" aorder="1" eorder="1" porder="1">
        <PCDATA type="PCDATA" iid="11" pid="10" aorder="0" eorder="0" porder="1">
          While...
        </PCDATA>
        <emph type="ELEM" iid="12" pid="10" aorder="1" eorder="1" porder="2">
          <PCDATA type="PCDATA" iid="13" pid="12" aorder="0" eorder="0" porder="1">
            markup
          </PCDATA>
        </emph>
        <PCDATA type="PCDATA" iid="14" pid="10" aorder="1" eorder="1" porder="3">
          ...
        </PCDATA>
      </para>
    </intro>
  </chapter>
</report>
```

B Appendix 2: Loading

B.1 Maximum Shredding [28]

B.1.1 Step 1

```
FUNCTION Q1($root)
{
   <DB xmlns:xsi="http://www.w3.org/2000/10/XMLSchema-instance" xsi:noNamespaceSchemaLocation="schema.xsd">
      FOR $tag IN DISTINCT (FOR $xxx IN $root//*[./@type="ELEM"]
                  RETURN{<T>gettag($xxx)</T>})
      LET $elements := $root//*[name(.)=TRIM($tag/text())],
              $tagname := $tag/text()
      RETURN
      {
         <$tagname>
            FOR $elem IN $elements
            LET $atts := $elem/*[./@type="ATT"],
                $rowtag := CONCAT($tagname, ".ROW")
            RETURN
            {
               <$rowtag>
                  <IID>$elem/@iid</IID>,
                  <ORDER>$elem/@porder</ORDER>,
                  <PID>$elem/@pid</PID>,
                  FOR $att IN $atts
                  LET $atttag := gettag($att)
                  RETURN
                  {
                     <$atttag>$att/CDATA/text()</$atttag>
                  }
               </$rowtag>
            }
         </$tagname>
      },
      <PCDATA>
         FOR $pcd IN $root//PCDATA
         RETURN
         {
            <PCDATA.ROW>
               <IID>$pcd/@iid</IID>,
               <ORDER>$pcd/@porder</ORDER>,
               <PID>$pcd/@pid</PID>,
               <VALUE>$pcd/text()</VALUE>
            </PCDATA.ROW>
         }
      </PCDATA>
   </DB>
}

Q1({<XXX>document("result0.xml")</XXX>})
```

B.1.2 XML Schema

```
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema" elementFormDefault="qualified">
   <xs:element name="DB">
      <xs:complexType>
         <xs:sequence>
            FOR $elementname IN DISTINCT (FOR $xxx IN document("source.xsd")//*[name(.)="xs:element"]
                           RETURN{<T>$xxx/@name</T>})
            LET $elements := document("source.xsd")//*[name(.)="xs:element" AND ./@name=TRIM($elementname)
                             AND NOT ./@ref],
                $element := $elements[1],
                $attributes := $element/xs:complexType/xs:simpleContent/xs:restriction/xs:attribute UNION
                               $element/xs:complexType/xs:attribute,
                $rowname := CONCAT($element/@name, ".ROW")
            RETURN
            {
               <xs:element name=$elementname>
                  <xs:complexType>
                     <xs:sequence>
                        <xs:element name=$rowname minOccurs="1" maxOccurs="unbounded">
                           <xs:complexType>
                              <xs:sequence>
                                 <xs:element name="IID" type="xs:double"/>,
                                 <xs:element name="ORDER" type="xs:double"/>,
                                 <xs:element name="PID" type="xs:double"/>,
                                 FOR $attribute IN $attributes
                                 RETURN
                                 {
                                    <xs:element name=$attribute/@name type=$attribute/@type/>
                                 }
                              </xs:sequence>
                           </xs:complexType>
                        </xs:element>
                     </xs:sequence>
                  </xs:complexType>
               </xs:element>
            },
            <xs:element name="PCDATA" minOccurs="0">
               <xs:complexType>
                  <xs:sequence>
                     <xs:element name="PCDATA.ROW" minOccurs="1" maxOccurs="unbounded">
                        <xs:complexType>
                           <xs:sequence>
                              <xs:element name="IID" type="xs:double"/>,
                              <xs:element name="ORDER" type="xs:double"/>,
                              <xs:element name="PID" type="xs:double"/>,
                              <xs:element name="VALUE" type="xs:string"/>
                           </xs:sequence>
                        </xs:complexType>
                     </xs:element>
                  </xs:sequence>
               </xs:complexType>
            </xs:element>
         </xs:sequence>
      </xs:complexType>
   </xs:element>
```

```
</xs:schema>
```

B.2 Attribute Tables [20]

B.2.1 Step 1

```
FUNCTION Q1($root)
{
   FOR $tag IN DISTINCT (FOR $xxx IN $root//*[./@type="ELEM" OR ./@type="ATT"]
               RETURN{<T>gettag($xxx)</T>})
   LET $elements := $root//*[name(.)=TRIM($tag/text())],
           $tablename := $tag/text(),
           $rowtag := CONCAT($tag/text(), ".ROW")
   RETURN
   {
      <$tablename>
         FOR $elem IN $elements
         RETURN
         {
         <$rowtag>
            <SOURCE>$elem/@pid</SOURCE>,
            <ORDER>$elem/@eorder</ORDER>,
            IF ($elem/PCDATA)
            THEN
            {
               <TARGET>$elem/PCDATA/text()</TARGET>
            }
            ELSE
            {
               IF ($elem/CDATA)
               THEN
               {
                  <TARGET>$elem/CDATA/text()</TARGET>
               }
               ELSE
               {
                  <TARGET>$elem/@iid</TARGET>
               }
            }
         </$rowtag>
         }
      </$tablename>
   }
}

<DB xmlns:xsi="http://www.w3.org/2000/10/XMLSchema-instance" xsi:noNamespaceSchemaLocation="schema.xsd">
   Q1({<XXX>document("result0.xml")</XXX>})
</DB>
```

B.2.2 XML Schema

```
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema" elementFormDefault="qualified">
   <xs:element name="DB">
      <xs:complexType>
         <xs:sequence>
            FOR $elementname IN DISTINCT (FOR $xxx IN document("source.xsd")//*[name(.)="xs:element"]
                           RETURN{<T>$xxx/@name</T>})
            LET $elements := document("source.xsd")//*[name(.)="xs:element" AND ./@name=TRIM($elementname)
                             AND NOT ./@ref],
                $element := $elements[1],
                $attributes := $element/xs:complexType/xs:simpleContent/xs:restriction/xs:attribute UNION
                               $element/xs:complexType/xs:attribute,
                $rowname := CONCAT($element/@name, ".ROW")

            RETURN
            {
               <xs:element name=$element/@name>
                  <xs:complexType>
                     <xs:sequence>
                        <xs:element name=$rowname minOccurs="1" maxOccurs="unbounded">
                           <xs:complexType>
                              <xs:sequence>
                                 <xs:element name="SOURCE" type="xs:double"/>,
                                 <xs:element name="ORDER" type="xs:double"/>,
                                 <xs:element name="TARGET" type="xs:string"/>
                              </xs:sequence>
                           </xs:complexType>
                        </xs:element>
                     </xs:sequence>
                  </xs:complexType>
               </xs:element>,
               FOR $att IN $attributes
               LET $rowname := CONCAT($att/@name, ".ROW")
               RETURN
               {
                  <xs:element name=$att/@name>
                     <xs:complexType>
                        <xs:sequence>
                           <xs:element name=$rowname minOccurs="1" maxOccurs="unbounded">
                              <xs:complexType>
                                 <xs:sequence>
                                    <xs:element name="SOURCE" type="xs:double"/>,
                                    <xs:element name="ORDER" type="xs:double"/>,
                                    <xs:element name="TARGET" type="xs:string"/>
                                 </xs:sequence>
                              </xs:complexType>
                           </xs:element>
                        </xs:sequence>
                     </xs:complexType>
                  </xs:element>
               }
            }
         </xs:sequence>
      </xs:complexType>
   </xs:element>
</xs:schema>
```

B.3 Edge Table [20]

B.3.1 Step 1

```
FUNCTION Q1($root)
{
   LET $tag := gettag($root)
   RETURN
   {
      IF ($root/PCDATA)
      THEN
      {
         <EDGES.ROW>
            <SOURCE>$root/@pid</SOURCE>,
            <ORDER>$root/@eorder</ORDER>,
            <NAME>$tag</NAME>,
            <TARGET>$root/PCDATA/text()</TARGET>
         </EDGES.ROW>
      }
      ELSE
      {
         IF ($root/CDATA)
         THEN
         {
            <EDGES.ROW>
               <SOURCE>$root/@pid</SOURCE>,
               <ORDER>$root/@eorder</ORDER>,
               <NAME>$tag</NAME>,
               <TARGET>$root/CDATA/text()</TARGET>
            </EDGES.ROW>
         }
         ELSE
         {
            <EDGES.ROW>
               <SOURCE>$root/@pid</SOURCE>,
               <ORDER>$root/@eorder</ORDER>,
               <NAME>$tag</NAME>,
               <TARGET>$root/@iid</TARGET>
            </EDGES.ROW>,
            FOR $child IN $root/*[./@type="ELEM" OR ./@type="ATT"]
            RETURN
            {
               Q1($child)
            }
         }
      }
   }
}

<DB xmlns:xsi="http://www.w3.org/2000/10/XMLSchema-instance" xsi:noNamespaceSchemaLocation="schema.xsd">
   <EDGES>
      Q1({document("result0.xml")})
   </EDGES>
</DB>
```

B.3.2 XML Schema

```
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema" elementFormDefault="qualified">
   <xs:element name="DB">
      <xs:complexType>
         <xs:sequence>
            <xs:element name="EDGES">
               <xs:complexType>
                  <xs:sequence>
                     <xs:element name="EDGES.ROW" minOccurs="1" maxOccurs="unbounded">
                        <xs:complexType>
                           <xs:sequence>
                              <xs:element name="SOURCE" type="xs:double"/>,
                              <xs:element name="ORDER" type="xs:double"/>,
                              <xs:element name="NAME" type="xs:string"/>,
                              <xs:element name="TARGET" type="xs:string"/>
                           </xs:sequence>
                        </xs:complexType>
                     </xs:element>
                  </xs:sequence>
               </xs:complexType>
            </xs:element>
         </xs:sequence>
      </xs:complexType>
   </xs:element>
</xs:schema>
```

B.4 Inlining [20]

B.4.1 Step 1

```
FUNCTION Inlinable($root)
{
   LET $elements := document("source.xsd")//xs:element [./@name=$root AND NOT ./@maxOccurs!="1"]
   RETURN
   {
      IF ( COUNT($elements)=0 )
      THEN
      {
         FALSE
      }
      ELSE
      {
         LET $choices := document("source.xsd")//xs:choice [./xs:element/@name=$root] UNION
                         document("source.xsd")/xs:element [./@name=$root]
         RETURN
         {
            IF ( COUNT($choices)=0 )
            THEN
            {
               TRUE
            }
            ELSE
            {
               FALSE
            }
         }
      }
   }
}

FUNCTION PCDATAInline($root)
{
   LET $choices := document("source.xsd")//xs:element[./@name=$root AND NOT ./@ref]/xs:complexType/xs:choice
   RETURN
   {
      IF ( COUNT($choices)=0 )
      THEN
      {
         TRUE
      }
      ELSE
      {
         FALSE
      }
   }
}

FUNCTION Q1($root, $parentinlinable, $PCDATAInline)
{
   LET $tag := gettag($root)
   RETURN
   {
      IF ($root/@type="ELEM")
      THEN
      {
         <$tag INLINABLE=Inlinable($tag) $root/@*>
            FOR $child IN $root/*
                  RETURN
                  {
                     Q1($child, Inlinable($tag), PCDATAInline($tag))
                  }
               </$tag>
      }
      ELSE
      {
         IF ($root/@type="ATT")
         THEN
         {
            LET $tag := CONCAT($tag, "_ATT")
            RETURN
            {
               <$tag INLINABLE="TRUE" $root/@*>
                  $root/*[1]/text()
                     </$tag>
                  }
               }
               ELSE
               {
                  IF ($parentinlinable="TRUE" and $PCDATAInline="TRUE")
                  THEN
                  {
                     $root/text()
                  }
                  ELSE
                  {
                     <$tag INLINABLE=$PCDATAInline $root/@*>
                  $root/text()
                     </$tag>

                  }
               }
      }
   }
}

<DB>
   Q1({document("result0.xml")}, "FALSE", "TRUE")
</DB>
```

B.4.2 Step 2

```
FUNCTION Q2($root, $path)
{
   RETURN
   {
      LET $tag := CONCAT($path, "_", gettag($root), "_", NUMFORMAT("######", $root/@porder)),
          $elements := $root/*[(./@type="ELEM" OR ./@type="PCDATA") AND ./@INLINABLE="TRUE"]
      RETURN
      {
         IF (COUNT($elements) = 0)
         THEN
         {
            IF (TRIM($root/text())="")
            THEN
            {
               LET $iidtag := CONCAT($tag, "_IID")
               RETURN
               {
                  <$iidtag>$root/@iid</$iidtag>
               }
            }
            ELSE
            {
               <$tag>$root/text()</$tag>
            }
         }
         ELSE
         {
            FOR $elem IN $elements
            RETURN
            {
               Q2($elem, $tag)
            }
         }
      },
      FOR $attchild IN $root/*[./@type="ATT"]
      LET $atttag := CONCAT($path, "_", gettag($root), "_", NUMFORMAT("######", $root/@porder), "_",
                            gettag($attchild))
      RETURN
      {
         <$atttag>$attchild/text()</$atttag>
      }
   }
}

<DB xmlns:xsi="http://www.w3.org/2000/10/XMLSchema-instance" xsi:noNamespaceSchemaLocation="schema.xsd">
   FOR $tag IN DISTINCT (FOR $xxx IN document("result1.xml")//*[./@INLINABLE="FALSE"]
               RETURN{<T>gettag($xxx)</T>})
   LET $tables := document("result1.xml")//*[name(.)=TRIM($tag/text()) AND ./@INLINABLE="FALSE"],
       $tag := $tag/text()
   RETURN
   {
      <$tag>
         FOR $table IN $tables
         LET $tabletag := CONCAT($tag, ".ROW")
         RETURN
         {
            <$tabletag>
               <IID>$table/@iid</IID>,
               <PID>$table/@pid</PID>,
               <ORDER>$table/@porder</ORDER>,
               IF ($tag="PCDATA")
               THEN
               {
                  <VALUE>$table/text()</VALUE>
               }
               ELSE
               {
                  FOR $attchild IN $table/*[./@type="ATT"]
                  LET $atttag := CONCAT($tag, "_", gettag($attchild))
                  RETURN
                  {
                     <$atttag>$attchild/text()</$atttag>
                  },
                  FOR $elem IN $table/*[(./@type="ELEM" OR ./@type="PCDATA") AND ./@INLINABLE="TRUE"]
                  RETURN
                  {
                     Q2($elem, $tag)
                  }
               }
            </$tabletag>
         }
      </$tag>
   }
</DB>
```

B.4.3 XML Schema

```
FUNCTION Inlinable($root)
```

```
{
   LET $elements := document("source.xsd")//xs:element [./@name=$root AND NOT ./@maxOccurs!="1"]
   RETURN
   {
      IF ( COUNT($elements)=0 )
      THEN
      {
         FALSE
      }
      ELSE
      {
         LET $choices := document("source.xsd")//xs:choice [./xs:element/@name=$root] UNION
                         document("source.xsd")/xs:element [./@name=$root]
         RETURN
         {
            IF ( COUNT($choices)=0 )
            THEN
            {
               TRUE
            }
            ELSE
            {
               FALSE
            }
         }
      }
   }
}

FUNCTION GetChilds($root, $oldtag)
{
   LET $order := 0
   FOR $rootchild IN $root/xs:element [(NOT ./@maxOccurs!="1") AND (NOT ./@ref)]
   LET $order := $order + 1,
       $rootchildtag := CONCAT($oldtag, "_", $rootchild/@name, "_", NUMFORMAT("######", $order)),
       $rootchilds := $rootchild/xs:element UNION $rootchild/xs:choice
   RETURN
   {
      IF ( COUNT($rootchilds) = 0 )
      THEN
      {
         IF ($rootchild/@*)
         THEN
         {
            <xs:element name=$rootchildtag type="xs:string" />
         }
         ELSE
         {
            <xs:element name=$rootchildtag type=$rootchild/@type />
         },
         FOR $attribute IN $rootchild/xs:attribute
         LET $attributetag := CONCAT($rootchildtag, "_", $attribute/@name, "_ATT")
         RETURN
         {
            <xs:element name=$attributetag type=$attribute/@type />
         }
      }
      ELSE
      {
         IF ( (COUNT($rootchilds[(NOT ./@maxOccurs!="1") AND (NOT ./@ref)]) = 0) OR $rootchild/xs:choice )
         THEN
         {
            LET $rootchildtag := CONCAT($rootchildtag, "_IID")
            RETURN
            {
               <xs:element name=$rootchildtag type="xs:double"/>,
               FOR $attribute IN $rootchild/xs:attribute
               LET $attributetag := CONCAT($rootchildtag, "_", $attribute/@name, "_ATT")
               RETURN
               {
                  <xs:element name=$attributetag type=$attribute/@type />
               },
               GetChilds($rootchild, $rootchildtag)
            }
         }
         ELSE
         {

            FOR $attribute IN $rootchild/xs:attribute
            LET $attributetag := CONCAT($rootchildtag, "_", $attribute/@name, "_ATT")
            RETURN
            {
               <xs:element name=$attributetag type=$attribute/@type />
            },
            GetChilds($rootchild, $rootchildtag)
         }
      }

   }
}

FUNCTION Q1($root)
{
<xs:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema" elementFormDefault="qualified">
 <xs:element name="DB">
  <xs:complexType>
   <xs:sequence>
   FOR $elem IN $root//* [gettag(.)="xs:element" AND Inlinable(/@name)="FALSE" AND NOT ./@ref]
   LET $roottag := $elem/@name,
       $rowtag := CONCAT($roottag, ".ROW")
   RETURN
   {
      <xs:element name=$roottag>
       <xs:complexType>
        <xs:sequence>
         <xs:element name=$rowtag minOccurs="1" maxOccurs="unbounded">
          <xs:complexType>
           <xs:sequence>
         <xs:element name="IID" type="xs:double"/>,
         <xs:element name="PID" type="xs:double"/>,
         <xs:element name="ORDER" type="xs:double"/>,
         FOR $attribute IN $elem/xs:attribute
         LET $attributetag := CONCAT($roottag, "_", $attribute/@name, "_ATT")
         RETURN
         {
            <xs:element name=$attributetag type=$attribute/@type />
         },
         IF ( $elem/xs:element OR $elem/xs:choice)
         THEN
         {
            GetChilds($elem, $roottag)
         }
         ELSE
         {
            LET $tag := CONCAT($elem/@name, "_PCDATA_1")
            RETURN
            {
               <xs:element name=$tag type="xs:string"/>
            }
         }
```

```
          </xs:sequence>
         </xs:complexType>
        </xs:element>
       </xs:sequence>
      </xs:complexType>
     </xs:element>
   },
   IF ( $root//xs:choice )
   THEN
   {
      <xs:element name="PCDATA">
       <xs:complexType>
        <xs:sequence>
         <xs:element name="PCDATA.ROW">
          <xs:complexType>
           <xs:sequence>
         <xs:element name="IID" type="xs:double"/>,
         <xs:element name="PID" type="xs:double"/>,
         <xs:element name="ORDER" type="xs:double"/>,
         <xs:element name="VALUE" type="xs:string"/>
           </xs:sequence>
          </xs:complexType>
         </xs:element>
        </xs:sequence>
       </xs:complexType>
      </xs:element>
   }
   ELSE
   {
      ""
   }
   </xs:sequence>
  </xs:complexType>
 </xs:element>
</xs:schema>
}
```

```
Q1({<XXX>{document("source.xsd") FILTER ( self::xs:element OR self::xs:choice OR self::xs:attribute )}</XXX>})
```

B.5 Universal Table [20]

B.5.1 Step 1

```
FUNCTION Q1($root)
{
   FOR $tag IN DISTINCT (FOR $xxx IN $root//*[./@type="ELEM" OR ./@type="ATT"]
               RETURN{<T>gettag($xxx)</T>})
   LET $elements := $root//*[name(.)=TRIM($tag/text())],
           $tablename := $tag/text(),
           $rowtag := CONCAT($tag/text(), ".ROW")
   RETURN
   {
      <$tablename>
         FOR $elem IN $elements
         RETURN
         {
         <$rowtag>
            <SOURCE name=$tablename>$elem/@pid</SOURCE>,
            <ORDER name=$tablename>$elem/@aorder</ORDER>,
            IF ($elem/PCDATA)
            THEN
            {
               <TARGET name=$tablename>$elem/PCDATA/text()</TARGET>
            }
            ELSE
            {
               IF ($elem/CDATA)
               THEN
               {
                  <TARGET name=$tablename>$elem/CDATA/text()</TARGET>
               }
               ELSE
               {
                  <TARGET name=$tablename>$elem/@iid</TARGET>
               }
            }
         </$rowtag>
         }
      </$tablename>
   }
}

<DB>
   Q1({<XXX>document("result0.xml")</XXX>})
</DB>
```

B.5.2 Step 2

```
FUNCTION joinlist($root)
{
   LET $name := $root/@name
   RETURN
   {
      <$name/>,
      FOR $child1 IN $root/*[name(.)="xs:attribute"]
      RETURN
      {
         joinlist($child1)
      },
      FOR $child2 IN $root/*[name(.)="xs:element"]
      RETURN
      {
         joinlist($child2)
      }
   }
}

FUNCTION Q2($part1, $part2, $list)
{
   IF (COUNT($part2)=0)
   THEN
   {
      <UNIVERSAL>
         FOR $child IN $part1/*
         RETURN
         {
            <UNIVERSAL.ROW>
               $child/*
            </UNIVERSAL.ROW>
         }
      </UNIVERSAL>
   }
   ELSE
   {
      LET $firstlist := $list[1],
          $restlist := $list[position() .>. 1],
          $next := $part2[name(.)=gettag($firstlist)],
          $rest := $part2[name(.)!=gettag($firstlist)]
```

```
RETURN
{
   IF (COUNT($rest) .>. 0)
   THEN
   {
      Q2({<UNIVERSAL>
            FOR $row1 IN $part1/*
            LET $joins := $next/* [./SOURCE=$row1/TARGET]
            RETURN
            {
               IF ( COUNT($joins)=0 )
               THEN
               {
                  <UNIVERSAL.ROW>
                     FOR $child1 IN $row1/*
                     RETURN
                     {
                        $child1
                     },
                     FOR $child2 IN $next/*[1]/*[name(.)!="SOURCE"]
                     LET $tag := gettag($child2)
                     RETURN
                     {
                        <$tag $child2/@name />
                     }
                  </UNIVERSAL.ROW>
               }
               ELSE
               {
                  FOR $join IN $joins
                  RETURN
                  {
                     <UNIVERSAL.ROW>
                        FOR $child1 IN $row1/*
                        RETURN
                        {
                           $child1
                        },
                        FOR $child2 IN $join/*[name(.)!="SOURCE"]
                        RETURN
                        {
                           $child2
                        }
                     </UNIVERSAL.ROW>
                  }
               }
            }
      </UNIVERSAL>}, $rest, $restlist)
   }
   ELSE
   {
         <UNIVERSAL>
            FOR $row1 IN $part1/*
            LET $joins := $next/* [./SOURCE=$row1/TARGET]
            RETURN
            {
               IF ( COUNT($joins)=0 )
               THEN
               {
                  <UNIVERSAL.ROW>
                     FOR $child1 IN $row1/*
                     RETURN
                     {
                        $child1
                     },
                              FOR $child2 IN $next/*[1]/*[name(.)!="SOURCE"]
                              LET $tag := gettag($child2)
                              RETURN
                              {
                                 <$tag $child2/@name />
                              }
                           </UNIVERSAL.ROW>
                        }
                        ELSE
                        {
                           FOR $join IN $joins
                           RETURN
                           {
                              <UNIVERSAL.ROW>
                                 FOR $child1 IN $row1/*
                                 RETURN
                                 {
                                    $child1
                                 },
                                 FOR $child2 IN $join/*[name(.)!="SOURCE"]
                                 RETURN
                                 {
                                    $child2
                                 }
                              </UNIVERSAL.ROW>
                           }
                        }
                     }
                  </UNIVERSAL>
            }
         }
      }
}

Let $list := joinlist({document("source.xsd") FILTER self::xs:element or self::xs:attribute})
RETURN
{
   Q2(document("result1.xml")/*[1], document("result1.xml")/*[position() .>. 1], $list[position() .>. 1])
}
```

B.5.3 Step 3

```
FUNCTION Q3($root)
{
   IF (gettag($root)!="DB" AND gettag($root)!="UNIVERSAL" AND NOT CONTAINS(gettag($root),".ROW"))
   THEN
   {
      LET $tag := CONCAT($root/@name, "_", gettag($root))
      RETURN
      {
         IF (CONTAINS($tag, "SOURCE"))
         THEN
         {
            <SOURCE>
               $root/node()
            </SOURCE>
         }
         ELSE
         {
            <$tag>
               $root/node()
            </$tag>
         }
      }
   }
   ELSE
   {
      LET $tag := gettag($root)
      RETURN
      {
         <$tag>
            FOR $child IN $root/*
            RETURN
            {
               Q3($child)
            }
         </$tag>
      }
   }
}
<DB xmlns:xsi="http://www.w3.org/2000/10/XMLSchema-instance" xsi:noNamespaceSchemaLocation="schema.xsd">
   Q3(document("result2.xml"))
</DB>
```

B.5.4 XML Schema

```
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema" elementFormDefault="qualified">
   <xs:element name="DB">
      <xs:complexType>
         <xs:sequence>
            <xs:element name="UNIVERSAL">
               <xs:complexType>
                  <xs:sequence>
                     <xs:element name="UNIVERSAL.ROW" minOccurs="1" maxOccurs="unbounded">
                        <xs:complexType>
                           <xs:sequence>
                              <xs:element name="SOURCE" type="xs:double"/>,
                              FOR $element IN document("source.xsd")//* [name(.)="xs:element"]
                              LET $attributes := $element/xs:complexType/xs:simpleContent/xs:restriction/xs:attribute
                                                   UNION $element/xs:complexType/xs:attribute,
                                  $column1 := CONCAT($element/@name, "_ORDER"),
                                      $column2 := CONCAT($element/@name, "_TARGET")
                              RETURN
                              {
                                 <xs:element name=$column1 type="xs:double"/>,
                                 <xs:element name=$column2 type="xs:string"/>,
                                 FOR $attribute IN $attributes
                                 LET $column1 := CONCAT($attribute/@name, "_ORDER"),
                                         $column2 := CONCAT($attribute/@name, "_TARGET")
                                     RETURN
                                     {
                                        <xs:element name=$column1 type="xs:double"/>,
                                    <xs:element name=$column2 type="xs:string"/>
                                     }
                              }
                           </xs:sequence>
                        </xs:complexType>
                     </xs:element>
                  </xs:sequence>
               </xs:complexType>
            </xs:element>
         </xs:sequence>
      </xs:complexType>
   </xs:element>
</xs:schema>
```

B.6 Composed Loading

B.6.1 Step 1

```
FUNCTION checkskip($skiplist, $elem)
{
   LET $match := $skiplist[name(.)=$elem]
   RETURN
   {
      IF (COUNT($match) .>. 0)
      THEN
      {
         "TRUE"
      }
      ELSE
      {
         "FALSE"
      }
   }
}

FUNCTION tree($root)
{
   LET $element := document("source.xsd")//xs:element[./@name=$root],
       $elemname := $element/@name
   RETURN
   {
      <$elemname />,
      FOR $child IN $element//xs:element
      LET $childname := $child/@name
      RETURN
      {
         <$childname />
      }
   }
}

FUNCTION Att1($root, $skiplist, $orderlist, $ordershift)
{
   FOR $tag IN DISTINCT (FOR $xxx IN $root//*[./@type="ELEM" OR ./@type="ATT"]
               RETURN{<T>gettag($xxx)</T>})
   LET $elements := $root//*[name(.)=TRIM($tag/text())],
           $tablename := TRIM($tag/text()),
           $rowtag := CONCAT($tag/text(), ".ROW")
   RETURN
   {
      IF (checkskip($skiplist, $tablename)="TRUE")
      THEN
      {
         ""
      }
      ELSE
      {
         <$tablename>
            FOR $elem IN $elements
            RETURN
            {
            <$rowtag>
               <SOURCE>$elem/@pid</SOURCE>,
               IF (checkskip($orderlist, $tablename)="TRUE")
               THEN
               {
                  <ORDER>($elem/@eorder + $ordershift)</ORDER>
               }
               ELSE
               {
                  <ORDER>$elem/@eorder</ORDER>
               },
               IF ($elem/PCDATA)
               THEN
               {
                  <TARGET>$elem/PCDATA/text()</TARGET>
               }
               ELSE
               {
                  IF ($elem/CDATA)
                  THEN
                  {
                     <TARGET>$elem/CDATA/text()</TARGET>
                  }
                  ELSE
                  {
                     <TARGET>$elem/@iid</TARGET>
                  }
               }
            </$rowtag>
            }
         </$tablename>
      }
   }
}

FUNCTION Clock1($root, $skiplist, $orderlist, $ordershift)
{
   RETURN
   {
      FOR $tag IN DISTINCT (FOR $xxx IN $root//*[./@type="ELEM"]
                  RETURN{<T>gettag($xxx)</T>})
      LET $elements := $root//*[name(.)=TRIM($tag/text())],
              $tagname := TRIM($tag/text())
      RETURN
      {
         IF (checkskip($skiplist, $tagname)="TRUE")
         THEN
         {
            ""
         }
         ELSE
         {
            <$tagname>
               FOR $elem IN $elements
               LET $atts := $elem/*[./@type="ATT"],
                   $rowtag := CONCAT($tagname, ".ROW")
               RETURN
               {
                  <$rowtag>
                     <IID>$elem/@iid</IID>,
                     IF (checkskip($orderlist, $tagname)="TRUE")
                     THEN
                     {
                        <ORDER>($elem/@porder + $ordershift)</ORDER>
                     }
                     ELSE
                     {
                        <ORDER>$elem/@porder</ORDER>
                     },
                     <PID>$elem/@pid</PID>,
                     FOR $att IN $atts
                     LET $atttag := gettag($att)
                     RETURN
```

```
                        {
                           <$atttag>$att/CDATA/text()</$atttag>
                        }
                     </$rowtag>
                  }
               </$tagname>
            }
         },
         LET $pcdata := $root//*[checkskip($skiplist, name(.))="FALSE"]/PCDATA
         RETURN
         {
            IF (COUNT($pcdata) = 0)
            THEN
            {
               ""
            }
            ELSE
            {
               <PCDATA>
                  FOR $pcd IN $pcdata
                  RETURN
                  {
                     <PCDATA.ROW>
                        <IID>$pcd/@iid</IID>,
                        <ORDER>$pcd/@porder</ORDER>,
                        <PID>$pcd/@pid</PID>,
                        <VALUE>$pcd/text()</VALUE>
                     </PCDATA.ROW>
                  }
               </PCDATA>
            }
         }
      }
   }

FUNCTION Edge1($root, $skiplist, $orderlist, $ordershift)
{
   LET $tag := gettag($root)
   RETURN
   {
      IF ($root/PCDATA)
      THEN
      {
         <EDGES.ROW>
            <SOURCE>$root/@pid</SOURCE>,
            IF (checkskip($orderlist, $tag)="TRUE")
            THEN
            {
               <ORDER>($root/@eorder + $ordershift)</ORDER>
            }
            ELSE
            {
               <ORDER>$root/@eorder</ORDER>
            },
            <NAME>$tag</NAME>,
            <TARGET>$root/PCDATA/text()</TARGET>
         </EDGES.ROW>
      }
      ELSE
      {
         IF ($root/CDATA)
         THEN
         {
            <EDGES.ROW>
               <SOURCE>$root/@pid</SOURCE>,
               IF (checkskip($orderlist, $tag)="TRUE")
               THEN
               {
                  <ORDER>($root/@eorder + $ordershift)</ORDER>
               }
               ELSE
               {
                  <ORDER>$root/@eorder</ORDER>
               },
               <NAME>$tag</NAME>,
               <TARGET>$root/CDATA/text()</TARGET>
            </EDGES.ROW>
         }
         ELSE
         {
            <EDGES.ROW>
               <SOURCE>$root/@pid</SOURCE>,
               IF (checkskip($orderlist, $tag)="TRUE")
               THEN
               {
                  <ORDER>($root/@eorder + $ordershift)</ORDER>
               }
               ELSE
               {
                  <ORDER>$root/@eorder</ORDER>
               },
               <NAME>$tag</NAME>,
               <TARGET>$root/@iid</TARGET>
            </EDGES.ROW>,
            FOR $child IN $root/*[./@type="ELEM" OR ./@type="ATT"]
            LET $name := gettag($child)
            RETURN
            {
               IF (checkskip($skiplist, $name)="TRUE")
               THEN
               {
                  ""
               }
               ELSE
               {
                  Edge1($child, $skiplist, $orderlist, $ordershift)
               }
            }
         }
      }
   }
}

FUNCTION Univ1($root, $skiplist, $orderlist, $ordershift)
{
   FOR $tag IN DISTINCT (FOR $xxx IN $root//*[./@type="ELEM" OR ./@type="ATT"]
               RETURN{<T>gettag($xxx)</T>})
   LET $elements := $root//*[name(.)=TRIM($tag/text())],
           $tablename := TRIM($tag/text()),
           $rowtag := CONCAT($tag/text(), ".ROW")
   RETURN
   {
      IF (checkskip($skiplist, $tablename)="TRUE")
      THEN
      {
         ""
      }
      ELSE
      {
         <$tablename>
```

```
            FOR $elem IN $elements
            RETURN
            {
            <$rowtag>
               <SOURCE name=$tablename>$elem/@pid</SOURCE>,
               IF (checkskip($orderlist, $tablename)="TRUE")
               THEN
               {
                  <ORDER name=$tablename>($elem/@aorder + $ordershift)</ORDER>
               }
               ELSE
               {
                  <ORDER name=$tablename>$elem/@aorder</ORDER>
               },
               IF ($elem/PCDATA)
               THEN
               {
                  <TARGET name=$tablename>$elem/PCDATA/text()</TARGET>
               }
               ELSE
               {
                  IF ($elem/CDATA)
                  THEN
                  {
                     <TARGET name=$tablename>$elem/CDATA/text()</TARGET>
                  }
                  ELSE
                  {
                     <TARGET name=$tablename>$elem/@iid</TARGET>
                  }
               }
            </$rowtag>
            }
         </$tablename>
      }
   }
}

FUNCTION joinlist($root)
{
   LET $name := $root/@name
   RETURN
   {
      <$name/>,
      FOR $child1 IN $root/*[name(.)="xs:attribute"]
      RETURN
      {
         joinlist($child1)
      },
      FOR $child2 IN $root/*[name(.)="xs:element"]
      RETURN
      {
         joinlist($child2)
      }
   }
}

FUNCTION Univ2($part1, $part2, $list)
{
   IF (COUNT($part2)=0)
   THEN
   {
      <UNIVERSAL>
         FOR $child IN $part1/*
         RETURN
         {
            <UNIVERSAL.ROW>
               $child/*
            </UNIVERSAL.ROW>
         }
      </UNIVERSAL>
   }
   ELSE
   {
      LET $firstlist := $list[1],
          $restlist := $list[position() .>. 1],
          $next := $part2[name(.)=gettag($firstlist)],
          $rest := $part2[name(.)!=gettag($firstlist)]
      RETURN
      {
         IF (COUNT($rest) .>. 0)
         THEN
         {
            Univ2({<UNIVERSAL>
                  FOR $row1 IN $part1/*
                  LET $joins := $next/* [./SOURCE=$row1/TARGET]
                  RETURN
                  {
                     IF ( COUNT($joins)=0 )
                     THEN
                     {
                        <UNIVERSAL.ROW>
                           FOR $child1 IN $row1/*
                           RETURN
                           {
                              $child1
                           },
                           FOR $child2 IN $next/*[1]/*[name(.)!="SOURCE"]
                           LET $tag := gettag($child2)
                           RETURN
                           {
                              <$tag $child2/@name />
                           }
                        </UNIVERSAL.ROW>
                     }
                     ELSE
                     {
                        FOR $join IN $joins
                        RETURN
                        {
                           <UNIVERSAL.ROW>
                              FOR $child1 IN $row1/*
                              RETURN
                              {
                                 $child1
                              },
                              FOR $child2 IN $join/*[name(.)!="SOURCE"]
                              RETURN
                              {
                                 $child2
                              }
                           </UNIVERSAL.ROW>
                        }
                     }
                  }
            </UNIVERSAL>}, $rest, $restlist)
         }
         ELSE
         {
               <UNIVERSAL>
```

```
                  FOR $row1 IN $part1/*
                  LET $joins := $next/* [./SOURCE=$row1/TARGET]
                  RETURN
                  {
                     IF ( COUNT($joins)=0 )
                     THEN
                     {
                        <UNIVERSAL.ROW>
                           FOR $child1 IN $row1/*
                           RETURN
                           {
                              $child1
                           },
                           FOR $child2 IN $next/*[1]/*[name(.)!="SOURCE"]
                           LET $tag := gettag($child2)
                           RETURN
                           {
                              <$tag $child2/@name />
                           }
                        </UNIVERSAL.ROW>
                     }
                     ELSE
                     {
                        FOR $join IN $joins
                        RETURN
                        {
                           <UNIVERSAL.ROW>
                              FOR $child1 IN $row1/*
                              RETURN
                              {
                                 $child1
                              },
                              FOR $child2 IN $join/*[name(.)!="SOURCE"]
                              RETURN
                              {
                                 $child2
                              }
                           </UNIVERSAL.ROW>
                        }
                     }
                  }
               </UNIVERSAL>
         }
      }
   }
}

FUNCTION Univ3($root)
{
   IF (gettag($root)!="DB" AND gettag($root)!="UNIVERSAL" AND NOT CONTAINS(gettag($root),".ROW"))
   THEN
   {
      LET $tag := CONCAT($root/@name, "_", gettag($root))
      RETURN
      {
         IF (CONTAINS($tag, "SOURCE"))
         THEN
         {
            <SOURCE>
               $root/node()
            </SOURCE>
         }
         ELSE
         {
            <$tag>
```

```
               IF (TRIM($root/node())="")
               THEN
               {
                  $root/text()
               }
               ELSE
               {
                  $root/node()
               }
            </$tag>
         }
      }
   }
   ELSE
   {
      LET $tag := gettag($root)
      RETURN
      {
         <$tag>
            FOR $child IN $root/*
            RETURN
            {
               Univ3($child)
            }
         </$tag>
      }
   }
}

FUNCTION Inlinable($root)
{
   LET $elements := document("source.xsd")//xs:element [./@name=$root AND NOT ./@maxOccurs!="1"]
   RETURN
   {
      IF ( COUNT($elements)=0 )
      THEN
      {
         FALSE
      }
      ELSE
      {
         LET $choices := document("source.xsd")//xs:choice [./xs:element/@name=$root] UNION
                         document("source.xsd")/xs:element [./@name=$root]
         RETURN
         {
            IF ( COUNT($choices)=0 )
            THEN
            {
               TRUE
            }
            ELSE
            {
               FALSE
            }
         }
      }
   }
}

FUNCTION PCDATAInline($root)
{
   LET $choices := document("source.xsd")//xs:element[./@name=$root AND NOT ./@ref]/xs:complexType/xs:choice
   RETURN
   {
      IF ( COUNT($choices)=0 )
      THEN
      {
         TRUE
      }
      ELSE
      {
         FALSE
      }
   }
}

FUNCTION Inline1($root, $parentinlinable, $PCDATAInline, $skiplist)
{
   LET $tag := gettag($root)
   RETURN
   {
      IF ($root/@type="ELEM")
      THEN
      {
         <$tag INLINABLE=Inlinable($tag) $root/@*>
            FOR $child IN $root/*
                  RETURN
                  {
                     IF (checkskip($skiplist, gettag($child))="TRUE")
               THEN
               {
                  ""
               }
               ELSE
               {
                        Inline1($child, Inlinable($tag), PCDATAInline($tag), $skiplist)
                     }
                  }
               </$tag>
      }
      ELSE
      {
         IF ($root/@type="ATT")
         THEN
         {
            LET $tag := CONCAT($tag, "_ATT")
            RETURN
            {
               <$tag INLINABLE="TRUE" $root/@*>
                  $root/*[1]/text()
                     </$tag>
                  }
               }
               ELSE
               {
                  IF ($parentinlinable="TRUE" and $PCDATAInline="TRUE")
                  THEN
                  {
                     $root/text()
                  }
                  ELSE
                  {
                     <$tag INLINABLE=$PCDATAInline $root/@*>
                  $root/text()
                     </$tag>
                  }
               }
      }
   }
}
```

```
FUNCTION Inline2($root, $path, $orderlist, $ordershift)
{
   RETURN
   {
      LET $name := gettag($root),
          $elements := $root/*[(./@type="ELEM" OR ./@type="PCDATA") AND ./@INLINABLE="TRUE"]
      RETURN
      {
         IF (COUNT($elements) = 0)
         THEN
         {
            IF (TRIM($root/text())="")
            THEN
            {
               LET $iidtag := CONCAT($path, "_", $name, "_", NUMFORMAT("######", $root/@porder), "_IID")
               RETURN
               {
                  <$iidtag>$root/@iid</$iidtag>
               }
            }
            ELSE
            {
               IF (checkskip($orderlist, $name)="TRUE")
               THEN
               {
                  LET $tag := CONCAT($path, "_", $name, "_", NUMFORMAT("######", ($root/@porder + $ordershift)))
                  RETURN
                  {
                     <$tag>$root/text()</$tag>
                  }
               }
               ELSE
               {
                  LET $tag := CONCAT($path, "_", $name, "_", NUMFORMAT("######", $root/@porder))
                  RETURN
                  {
                     <$tag>$root/text()</$tag>
                  }
               }
            }
         }
         ELSE
         {
            FOR $elem IN $elements
            RETURN
            {
               Inline2($elem, $tag, $orderlist, $ordershift)
            }
         }
      },
      FOR $attchild IN $root/*[./@type="ATT"]
      LET $name := gettag($root)
      RETURN
      {
         IF (checkskip($orderlist, $name)="TRUE")
         THEN
         {
            LET $atttag := CONCAT($path, "_", $name, "_", NUMFORMAT("######", ($root/@porder + $ordershift)),
                                  "_", gettag($attchild))
            RETURN
            {
               <$atttag>$attchild/text()</$atttag>
            }
         }
         ELSE
         {
            LET $atttag := CONCAT($path, "_", $name, "_", NUMFORMAT("######", $root/@porder), "_",
                                  gettag($attchild))
            RETURN
            {
               <$atttag>$attchild/text()</$atttag>
            }
         }
      }
   }
}

FUNCTION Inline3($root, $skiplist, $orderlist1, $ordershift1, $orderlist2, $ordershift2)
{
   LET $step1 := {<DB>Inline1($root, "FALSE", "TRUE", $skiplist)</DB>}
   RETURN
   {
      FOR $tag IN DISTINCT (FOR $xxx IN $step1//*[./@INLINABLE="FALSE"]
                  RETURN{<T>gettag($xxx)</T>})
      LET $tables := $step1//*[name(.)=TRIM($tag/text()) AND ./@INLINABLE="FALSE"],
          $tag := $tag/text()
      RETURN
      {
         <$tag>
            FOR $table IN $tables
            LET $tabletag := CONCAT($tag, ".ROW")
            RETURN
            {
               <$tabletag>
                  <IID>$table/@iid</IID>,
                  <PID>$table/@pid</PID>,
                  IF (checkskip($orderlist1, $tag)="TRUE")
                  THEN
                  {
                     <ORDER>($table/@porder + $ordershift1)</ORDER>
                  }
                  ELSE
                  {
                     <ORDER>$table/@porder</ORDER>
                  },
                  IF ($tag="PCDATA")
                  THEN
                  {
                     <VALUE>$table/text()</VALUE>
                  }
                  ELSE
                  {
                     FOR $attchild IN $table/*[./@type="ATT"]
                     LET $atttag := CONCAT($tag, "_", gettag($attchild))
                     RETURN
                     {
                        <$atttag>$attchild/text()</$atttag>
                     },
                     FOR $elem IN $table/*[(./@type="ELEM" OR ./@type="PCDATA") AND ./@INLINABLE="TRUE"]
                     RETURN
                     {
                        Inline2($elem, $tag, $orderlist2, $ordershift2)
                     }
                  }
               </$tabletag>
            }
         </$tag>
      }
   }
```

```
}

FUNCTION Attribute($root, $skiplist, $orderlist, $ordershift)
{
   Att1({<XXX>$root</XXX>}, $skiplist, $orderlist, $ordershift)
}

FUNCTION Clock($root, $skiplist, $orderlist, $ordershift)
{
   Clock1({<XXX>$root</XXX>}, $skiplist, $orderlist, $ordershift)
}

FUNCTION Edge($root, $skiplist, $orderlist, $ordershift)
{
   <EDGES>
      Edge1($root, $skiplist, $orderlist, $ordershift)
   </EDGES>
}

FUNCTION Universal($root, $skiplist, $orderlist, $ordershift)
{
   LET $step1 := {<DB>Univ1({<XXX>$root</XXX>}, $skiplist, $orderlist, $ordershift)</DB>},
       $list := joinlist({document("source.xsd") FILTER self::xs:element or self::xs:attribute})
   RETURN
   {
      Univ3({Univ2($step1/*[1], $step1/*[position() .>. 1], $list[position() .>. 1])})
   }
}

FUNCTION Inline($root, $skiplist, $orderlist1, $ordershift1, $orderlist2, $ordershift2)
{
   Inline3($root, $skiplist, $orderlist1, $ordershift1, $orderlist2, $ordershift2)
}

RETURN
{
   <DB xmlns:xsi="http://www.w3.org/2000/10/XMLSchema-instance" xsi:noNamespaceSchemaLocation="schema.xsd">
      Inline({document("result0.xml")}, {<TITLE/>,<AUTHOR/>,<NAME/>}, {<XXX/>}, 0, {<XXX/>}, 0),
      FOR $title IN document("result0.xml")/BOOK/TITLE
      RETURN
      {
         Attribute($title, {<XXX/>}, {<XXX/>}, 0)
      },
      FOR $author IN document("result0.xml")/BOOK/AUTHOR
      RETURN
      {
         Edge($author, {<XXX/>}, {<XXX/>}, 0)
      }
   </DB>
}
```

B.6.2 Step 2

```
FUNCTION Q2($root)
{
   <DB xmlns:xsi="http://www.w3.org/2000/10/XMLSchema-instance" xsi:noNamespaceSchemaLocation="schema.xsd">
      FOR $table IN DISTINCT (FOR $xxx IN $root/*
                     RETURN{<T>gettag($xxx)</T>})
      LET $rows := $root/*[name(.)=TRIM($table/text())]/*,
          $tag := TRIM($table/text())
      RETURN
      {
         <$tag>
            FOR $row IN $rows
            RETURN
            {
               $row
            }
         </$tag>
      }
   </DB>
}

Q2(document("result1.xml"))
```

B.6.3 XML Schema

```
FUNCTION checkskip($skiplist, $elem)
{
   LET $match := $skiplist[name(.)=TRIM($elem)]
   RETURN
   {
      IF (COUNT($match) .>. 0)
      THEN
      {
         "TRUE"
      }
      ELSE
      {
         "FALSE"
      }
   }
}

FUNCTION tree($root)
{
   LET $element := document("source.xsd")//xs:element[./@name=$root],
       $elemname := $element/@name
   RETURN
   {
      <$elemname />,
      FOR $child IN $element//xs:element
      LET $childname := $child/@name
      RETURN
      {
         <$childname />
      }
   }
}

FUNCTION Attribute1($root, $skiplist)
{
   FOR $elementname IN DISTINCT (FOR $xxx IN $root//*[name(.)="xs:element" AND
                                 checkskip($skiplist, ./@name)="FALSE"]
                  RETURN{<T>$xxx/@name</T>})
   LET $elements := document("source.xsd")//*[name(.)="xs:element" AND ./@name=TRIM($elementname)
                             AND NOT ./@ref],
       $element := $elements[1],
       $attributes := $element/xs:complexType/xs:simpleContent/xs:restriction/xs:attribute UNION
                      $element/xs:complexType/xs:attribute,
       $rowname := CONCAT($element/@name, ".ROW")
   RETURN
   {
      <xs:element name=$element/@name>
         <xs:complexType>
            <xs:sequence>
               <xs:element name=$rowname minOccurs="1" maxOccurs="unbounded">
                  <xs:complexType>
```

```
                     <xs:sequence>
                        <xs:element name="SOURCE" type="xs:double"/>,
                        <xs:element name="ORDER" type="xs:double"/>,
                        <xs:element name="TARGET" type="xs:string"/>
                     </xs:sequence>
                  </xs:complexType>
               </xs:element>
            </xs:sequence>
         </xs:complexType>
      </xs:element>,
      FOR $att IN $attributes
      LET $rowname := CONCAT($att/@name, ".ROW")
      RETURN
      {
         <xs:element name=$att/@name>
            <xs:complexType>
               <xs:sequence>
                  <xs:element name=$rowname minOccurs="1" maxOccurs="unbounded">
                     <xs:complexType>
                        <xs:sequence>
                           <xs:element name="SOURCE" type="xs:double"/>,
                           <xs:element name="ORDER" type="xs:double"/>,
                           <xs:element name="TARGET" type="xs:string"/>
                        </xs:sequence>
                     </xs:complexType>
                  </xs:element>
               </xs:sequence>
            </xs:complexType>
         </xs:element>
      }
   }
}

FUNCTION Clock1($root, $skiplist)
{
   RETURN
   {
      FOR $elementname IN DISTINCT (FOR $xxx IN $root//* [name(.)="xs:element" AND
                                    checkskip($skiplist, ./@name)="FALSE"]
                     RETURN{<T>$xxx/@name</T>})
      LET $elements := document("source.xsd")//*[name(.)="xs:element" AND ./@name=TRIM($elementname)
                       AND NOT ./@ref],
          $element := $elements[1],
          $attributes := $element/xs:complexType/xs:simpleContent/xs:restriction/xs:attribute UNION
                         $element/xs:complexType/xs:attribute,
          $rowname := CONCAT($element/@name, ".ROW")
      RETURN
      {
         <xs:element name=$element/@name>
            <xs:complexType>
               <xs:sequence>
                  <xs:element name=$rowname minOccurs="1" maxOccurs="unbounded">
                     <xs:complexType>
                        <xs:sequence>
                           <xs:element name="IID" type="xs:double"/>,
                           <xs:element name="ORDER" type="xs:double"/>,
                           <xs:element name="PID" type="xs:double"/>,
                           FOR $attribute IN $attributes
                           RETURN
                           {
                              <xs:element name=$attribute/@name type=$attribute/@type/>
                           }
                        </xs:sequence>
                     </xs:complexType>
                  </xs:element>
```

```
                </xs:sequence>
             </xs:complexType>
          </xs:element>
       },
       <xs:element name="PCDATA" minOccurs="0">
          <xs:complexType>
             <xs:sequence>
                <xs:element name="PCDATA.ROW" minOccurs="1" maxOccurs="unbounded">
                   <xs:complexType>
                      <xs:sequence>
                         <xs:element name="IID" type="xs:double"/>,
                         <xs:element name="ORDER" type="xs:double"/>,
                         <xs:element name="PID" type="xs:double"/>,
                         <xs:element name="VALUE" type="xs:string"/>
                      </xs:sequence>
                   </xs:complexType>
                </xs:element>
             </xs:sequence>
          </xs:complexType>
       </xs:element>
    }
 }

 FUNCTION Edge($root, $skiplist)
 {
    <xs:element name="EDGES">
       <xs:complexType>
          <xs:sequence>
             <xs:element name="EDGES.ROW" minOccurs="1" maxOccurs="unbounded">
                <xs:complexType>
                   <xs:sequence>
                      <xs:element name="SOURCE" type="xs:double"/>,
                      <xs:element name="ORDER" type="xs:double"/>,
                      <xs:element name="NAME" type="xs:string"/>,
                      <xs:element name="TARGET" type="xs:string"/>
                   </xs:sequence>
                </xs:complexType>
             </xs:element>
          </xs:sequence>
       </xs:complexType>
    </xs:element>
 }

 FUNCTION Inlinable($root)
 {
    LET $elements := document("source.xsd")//xs:element [./@name=$root AND NOT ./@maxOccurs!="1"]
    RETURN
    {
       IF ( COUNT($elements)=0 )
       THEN
       {
          FALSE
       }
       ELSE
       {
          LET $choices := document("source.xsd")//xs:choice [./xs:element/@name=$root] UNION
                          document("source.xsd")/xs:element [./@name=$root]
          RETURN
          {
             IF ( COUNT($choices)=0 )
             THEN
             {
                TRUE
             }
             ELSE
```

```
            {
               FALSE
            }
         }
      }
   }
}

FUNCTION GetChilds($root, $oldtag, $skiplist)
{
   LET $order := 0
   FOR $rootchild IN $root/xs:element [(NOT ./@maxOccurs!="1") AND (NOT ./@ref)]
   LET $order := $order + 1,
       $rootchildtag := CONCAT($oldtag, "_", $rootchild/@name, "_", NUMFORMAT("######", $order)),
       $rootchilds := $rootchild/xs:element UNION $rootchild/xs:choice
   RETURN
   {
      IF ( checkskip($skiplist, $rootchild/@name)="FALSE" )
      THEN
      {
         IF ( COUNT($rootchilds) = 0 )
         THEN
         {
            IF ($rootchild/@*)
            THEN
            {
               <xs:element name=$rootchildtag type="xs:string" />
            }
            ELSE
            {
               <xs:element name=$rootchildtag type=$rootchild/@type />
            },
            FOR $attribute IN $rootchild/xs:attribute
            LET $attributetag := CONCAT($rootchildtag, "_", $attribute/@name, "_ATT")
            RETURN
            {
               <xs:element name=$attributetag type=$attribute/@type/>
            }
         }
         ELSE
         {
            IF ( (COUNT($rootchilds[(NOT ./@maxOccurs!="1") AND (NOT ./@ref)]) = 0) OR $rootchild/xs:choice )
            THEN
            {
               LET $rootchildtag := CONCAT($rootchildtag, "_IID")
               RETURN
               {
                  <xs:element name=$rootchildtag type="xs:double"/>,
                  FOR $attribute IN $rootchild/xs:attribute
                  LET $attributetag := CONCAT($rootchildtag, "_", $attribute/@name, "_ATT")
                  RETURN
                  {
                     <xs:element name=$attributetag type=$attribute/@type/>
                  },
                  GetChilds($rootchild, $rootchildtag, $skiplist)
               }
            }
            ELSE
            {
               FOR $attribute IN $rootchild/xs:attribute
               LET $attributetag := CONCAT($rootchildtag, "_", $attribute/@name, "_ATT")
               RETURN
               {
                  <xs:element name=$attributetag type=$attribute/@type/>
               },
              GetChilds($rootchild, $rootchildtag, $skiplist)
           }
        }
     }
     ELSE
     {
        ""
     }
  }
}

FUNCTION Inline1($root, $skiplist)
{
  RETURN
  {

     FOR $elem IN $root//* [gettag(.)="xs:element" AND Inlinable(/@name)="FALSE" AND (NOT ./@ref) AND
                          checkskip($skiplist, ./@name)="FALSE"]
     LET $roottag := $elem/@name,
         $rowtag := CONCAT($roottag, ".ROW")
     RETURN
     {
        <xs:element name=$roottag>
         <xs:complexType>
          <xs:sequence>
           <xs:element name=$rowtag minOccurs="1" maxOccurs="unbounded">
            <xs:complexType>
             <xs:sequence>
           <xs:element name="IID" type="xs:double"/>,
           <xs:element name="PID" type="xs:double"/>,
           <xs:element name="ORDER" type="xs:double"/>,
           FOR $attribute IN $elem/xs:attribute
           LET $attributetag := CONCAT($roottag, "_", $attribute/@name, "_ATT")
           RETURN
           {
              <xs:element name=$attributetag type=$attribute/@type/>
           },
           IF ( $elem/xs:element OR $elem/xs:choice)
           THEN
           {
              GetChilds($elem, $roottag, $skiplist)
           }
           ELSE
           {
              LET $tag := CONCAT($elem/@name, "_PCDATA_1")
                 RETURN
                 {
                    <xs:element name=$tag type="xs:string"/>
                 }
           }
             </xs:sequence>
            </xs:complexType>
           </xs:element>
          </xs:sequence>
         </xs:complexType>
        </xs:element>
     },
     IF ( $root//xs:element[checkskip($skiplist, ./@name)="FALSE"]/xs:choice )
     THEN
     {
        <xs:element name="PCDATA">
         <xs:complexType>
          <xs:sequence>
           <xs:element name="PCDATA.ROW">
            <xs:complexType>
```

```
              <xs:sequence>
            <xs:element name="IID" type="xs:double"/>,
            <xs:element name="PID" type="xs:double"/>,
            <xs:element name="ORDER" type="xs:double"/>,
            <xs:element name="VALUE" type="xs:string"/>
              </xs:sequence>
             </xs:complexType>
            </xs:element>
           </xs:sequence>
          </xs:complexType>
         </xs:element>
      }
      ELSE
      {
         ""
      }
   }
}

FUNCTION Universal1($root, $skiplist)
{
   <xs:element name="UNIVERSAL">
      <xs:complexType>
         <xs:sequence>
            <xs:element name="UNIVERSAL.ROW" minOccurs="1" maxOccurs="unbounded">
               <xs:complexType>
                  <xs:sequence>
                     <xs:element name="SOURCE" type="xs:double"/>,
                     FOR $element IN $root//* [name(.)="xs:element" AND checkskip($skiplist, ./@name)="FALSE"]
                     LET $attributes := $element/xs:complexType/xs:simpleContent/xs:restriction/xs:attribute
                                        UNION $element/xs:complexType/xs:attribute,
                         $column1 := CONCAT($element/@name, "_ORDER"),
                         $column2 := CONCAT($element/@name, "_TARGET")
                     RETURN
                     {
                        <xs:element name=$column1 type="xs:double"/>,
                        <xs:element name=$column2 type="xs:string"/>,
                        FOR $attribute IN $attributes
                        LET $column1 := CONCAT($attribute/@name, "_ORDER"),
                            $column2 := CONCAT($attribute/@name, "_TARGET")
                            RETURN
                            {
                               <xs:element name=$column1 type="xs:double"/>,
                           <xs:element name=$column2 type="xs:string"/>
                            }
                     }
                  </xs:sequence>
               </xs:complexType>
            </xs:element>
         </xs:sequence>
      </xs:complexType>
   </xs:element>
}

FUNCTION Attribute($root, $skiplist)
{
   Attribute1({<XXX>$root</XXX>}, $skiplist)
}

FUNCTION Clock($root, $skiplist)
{
   Clock1({<XXX>$root</XXX>}, $skiplist)
}

FUNCTION Inline($root, $skiplist)
```

```
{
   Inline1({<XXX>$root FILTER ( self::xs:element OR self::xs:choice OR self::xs:attribute )</XXX>}, $skiplist)
}

FUNCTION Universal($root, $skiplist)
{
   Universal1({<XXX>$root</XXX>}, $skiplist)
}

<xs:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema" elementFormDefault="qualified">
   <xs:element name="DB">
      <xs:complexType>
         <xs:sequence>
            Inline(document("source.xsd"), {<TITLE/>, <AUTHOR/>, <NAME/>}),
            Attribute(document("source.xsd")//xs:element[./@name="TITLE"], {<XXX/>}),
            Edge(document("source.xsd")//xs:element[./@name="AUTHOR"], {<XXX/>})
         </xs:sequence>
      </xs:complexType>
   </xs:element>
</xs:schema>
```

C Appendix 3: Extraction

C.1 Maximum Shredding [28]

C.1.1 Step 1

```
FUNCTION TrimIt($set)
{
   FOR $elem IN $set
   LET $tag := gettag($elem)
   RETURN
   {
      <$tag>TRIM($elem/text())</$tag>
   }
}

FUNCTION Q1($pid, $roottag, $dxv, $schema)
{
   FOR $root IN $dxv/*[TRIM(name(.))=TRIM($roottag)]/*[TRIM(./PID)=TRIM($pid)]
   LET $order := TRIM($root/ORDER/text()),
       $schemachilds := $schema//*[./@name=$roottag AND NOT ./@ref]/xs:element UNION
                        $schema//*[./@name=$roottag AND NOT ./@ref]/xs:choice/xs:element,
       $atts := TrimIt($root/*[name(.)!="ORDER" AND name(.)!="IID" AND name(.)!="PID"]),
       $iid := TRIM($root/IID/text())
   RETURN
   {
      <$roottag xcubesort=$order $atts>
         IF ( COUNT($schemachilds)=0 )
         THEN
         {
            LET $pcdata := $dxv/PCDATA/*[./PID=$root/IID],
                $pcdataorder := TRIM($pcdata/ORDER/text())
            RETURN
            {
               <PCDATA xcubesort=$pcdataorder>
                  TRIM($pcdata/VALUE/text())
               </PCDATA>
            }
         }
         ELSE
         {
            IF ( $schema//*[./@name=$roottag AND NOT ./@ref]/xs:choice )
            THEN
            {
               FOR $schemachild IN $schemachilds
               LET $tag := $schemachild/@name
               RETURN
               {
                  Q1($iid, $tag, $dxv, $schema)
               },
               FOR $pcdata IN $dxv/PCDATA/*[TRIM(./PID)=TRIM($iid)]
               LET $pcdataorder := TRIM($pcdata/ORDER/text())
               RETURN
               {
                  <PCDATA xcubesort=$pcdataorder>
                     TRIM($pcdata/VALUE/text())
                  </PCDATA>
               }
            }
            ELSE
            {
               FOR $schemachild IN $schemachilds
               LET $tag := $schemachild/@name
               RETURN
               {
                  Q1($iid, $tag, $dxv, $schema)
               }
            }
         }
      </$roottag>
   }
}

LET $dxv := document("dxv.xml"),
    $schema := {<XXX>document("source.xsd") FILTER ( self::xs:element OR self::xs:choice OR
                self::xs:attribute )</XXX>},
    $roottag := $schema/*[1]/@name
RETURN
{
   Q1("0.0", $roottag, $dxv, $schema)
}
```

C.2 Attribute Tables [20]

C.2.1 Step 1

```
FUNCTION Q1($pid, $roottag, $dxv, $schema)
{
   FOR $root IN $dxv/*[TRIM(name(.))=TRIM($roottag)]/*[TRIM(./SOURCE)=$pid]
   LET $order := TRIM($root/ORDER/text()),
       $schemachilds := $schema//*[./@name=$roottag AND NOT ./@ref]/xs:element UNION
                        $schema//*[./@name=$roottag AND NOT ./@ref]/xs:attribute UNION
                        $schema//*[./@name=$roottag AND NOT ./@ref]/xs:choice/xs:element,
       $iid := TRIM($root/TARGET/text())
   RETURN
   {
      <$roottag xcubesort=$order>
         IF ( COUNT($schemachilds)=0 )
         THEN
         {
            <PCDATA xcubesort="1">
               $iid
            </PCDATA>
         }
         ELSE
         {
            FOR $schemachild IN $schemachilds
            LET $childtag := $schemachild/@name
            RETURN
            {
               Q1($iid, $childtag, $dxv, $schema)
            }
         }
      </$roottag>
   }
}

LET $dxv := document("dxv.xml"),
    $schema := {<XXX>document("source.xsd") FILTER ( self::xs:element OR self::xs:choice OR
                self::xs:attribute )</XXX>},
    $roottag := $schema/*[1]/@name
RETURN
{
   Q1("0.0", $roottag, $dxv, $schema)
}
```

C.3 Edge Table[20]

C.3.1 Step 1

```
FUNCTION Q1($pid, $roottag, $dxv, $schema)
{
   FOR $root IN $dxv/EDGES/*[TRIM(./NAME)=TRIM($roottag) AND TRIM(./SOURCE)=$pid]
   LET $order := TRIM($root/ORDER/text()),
       $schemachilds := $schema//*[./@name=$roottag AND NOT ./@ref]/xs:element UNION
                        $schema//*[./@name=$roottag AND NOT ./@ref]/xs:attribute UNION
                        $schema//*[./@name=$roottag AND NOT ./@ref]/xs:choice/xs:element,
       $iid := TRIM($root/TARGET/text())
   RETURN
   {
      <$roottag xcubesort=$order>
         IF ( COUNT($schemachilds)=0 )
         THEN
         {
            <PCDATA xcubesort="1">
               $iid
            </PCDATA>
         }
         ELSE
         {
            FOR $schemachild IN $schemachilds
            LET $tag := TRIM($schemachild/@name)
            RETURN
            {
               Q1($iid, $tag, $dxv, $schema)
            }
         }
      </$roottag>
   }
}

LET $dxv := document("dxv.xml"),
    $schema := {<XXX>document("source.xsd") FILTER ( self::xs:element OR self::xs:choice OR
                self::xs:attribute )</XXX>},
    $roottag := $schema/*[1]/@name
RETURN
{
   Q1("0.0", $roottag, $dxv, $schema)
}
```

C.4 Inlining [20]

C.4.1 Step 1

```
FUNCTION GetAttributes($columnlist, $roottag)
{
   FOR $att IN $columnlist
   LET $name := TOKEN(gettag($att), "_", "2")
   WHERE gettag($att)=CONCAT($roottag, "_", TOKEN(gettag($att), "_", "2"), "_ATT")
   RETURN
   {
      <$name>TRIM($att/text())</$name>
   }
}

FUNCTION Q1($pid, $roottag, $dxv, $schema, $flag)
{
IF ( $flag=0 )
THEN
{
   LET $tag := TOKEN($pid, "_", "1"),
       $order := TOKEN($pid, "_", "2")
   RETURN
   {
      IF ( NOT (TOKEN($pid, "_", "3")) )
      THEN
      {
         <$tag inlineorder=$order>
            $roottag
         </$tag>
      }
      ELSE
      {
         IF ( TOKEN($pid, "_", "3")="IID" )
         THEN
         {
            <$tag inlineorder=$order>
               FOR $schemachild IN ($schema//xs:element[./@name=$tag AND NOT ./@ref]/xs:element) UNION
                         ($schema//xs:element[./@name=$tag AND NOT ./@ref]/xs:choice/xs:element)
               LET $schematag := TRIM($schemachild/@name)
               RETURN
                  {
                     Q1(TRIM($roottag), $schematag, $dxv, $schema, "1")
                  }
            </$tag>
         }
         ELSE
         {
            IF ( TOKEN($pid, "_", "4")="ATT" )
            THEN
            {
               LET $attname := TOKEN($pid, "_", "3"),
                   $att := {<$attname>$roottag</$attname>}
               RETURN
               {
                  <$tag inlineorder=$order $att />
               }
            }
            ELSE
            {
               <$tag inlineorder=$order>
                  Q1(CUTTOKEN(CUTTOKEN($pid, "_"), "_"), $roottag, $dxv, $schema, "0")
               </$tag>
            }
         }
      }
   }
}
ELSE
{
   FOR $root IN $dxv/*[TRIM(name(.))=TRIM($roottag)]/*[TRIM(./PID)=TRIM($pid)]
   LET $order := TRIM($root/ORDER/text()),
       $schemachilds := ($schema//xs:element[./@name=$roottag AND NOT ./@ref]/xs:element) UNION
                        ($schema//xs:element[./@name=$roottag AND NOT ./@ref]/xs:choice/xs:element),
       $atts := GetAttributes($root/*, $roottag)
   RETURN
   {
      <$roottag xcubesort=$order $atts>
            FOR $schemachild IN $schemachilds
            LET $tag := TRIM($schemachild/@name)
            RETURN
            {
               Q1(TRIM($root/IID), $tag, $dxv, $schema, "1")
            },
            FOR $pcdata IN $dxv/PCDATA/*[TRIM(./PID)=TRIM($root/IID)]
            LET $pcdataorder := TRIM($pcdata/ORDER/text())
            RETURN
            {
               <PCDATA xcubesort=$pcdataorder>
                  TRIM($pcdata/VALUE/text())
               </PCDATA>
            },
            FOR $inlinedelem IN $root/*[TOKEN(name(.), "_", "3")!="ATT" AND name(.)!="IID" AND
                                name(.)!="PID" AND name(.)!="ORDER"]
            LET $value := TRIM($inlinedelem/text())
            RETURN
            {
               Q1(CUTTOKEN(gettag($inlinedelem), "_"), $value, $dxv, $schema, "0")
            }
      </$roottag>
   }
}
}

LET $dxv := document("dxv.xml"),
    $schema := {<XXX>document("source.xsd") FILTER ( self::xs:element OR self::xs:choice OR
                self::xs:attribute )</XXX>},
    $roottag := $schema/*[1]/@name
RETURN
{
   Q1("0.0", $roottag, $dxv, $schema, "1")
}
```

C.4.2 Step 2

```
FUNCTION Q2($root)
{
   LET $tag := gettag($root),
       $childs1 := $root/*[NOT ./@inlineorder],
       $childs2 := $root/*[./@inlineorder],
       $order := NUMFORMAT("######", TRIM(CONCAT($root/@xcubesort, $root/@inlineorder))),
       $atts := $root/@*[name(.)!="xcubesort" AND name(.)!="inlineorder"]
   RETURN
   {
      <$tag $atts xcubesort=$order>
         IF ( COUNT($childs1)=0 )
         THEN
         {   IF ($tag="PCDATA" )
            THEN
            {
               $root/text()
            }
            ELSE
            {
               <PCDATA xcubesort="1">
                  $root/text()
               </PCDATA>
            }
         }
         ELSE
         {
            For $child1 IN $childs1
            RETURN
            {
               Q2($child1)
            }
         },
         FOR $dist IN DISTINCT (FOR $xxx IN $childs2
                      RETURN{<XXX><X1>gettag($xxx)</X1>,<X2>$xxx/@inlineorder/text()</X2></XXX>})
         LET $bag := $childs2[TRIM(name(.))=TRIM($dist/X1/text()) AND TRIM(./@inlineorder)=TRIM($dist/X2/text())],
             $tag2 := $dist/X1/text(),
             $atts := $bag/@*[name(.)!="xcubesort" AND name(.)!="inlineorder"],
             $text := CONCAT(TRIM($bag/text()))
         RETURN
         {
            <$tag2 xcubesort=$dist/X2 $atts>
                  IF ( $bag/* )
                  THEN
                  {
                     FOR $elem IN $bag,
                         $element IN $elem/*
                     RETURN
                     {
                        Q2($element)
                     }
                  }
                  ELSE
                  {
                     IF ( $tag2="PCDATA" )
                     THEN
                     {
                        $bag/text()
                     }
                     ELSE
                     {
                        <PCDATA xcubesort="1">
                           $bag/text()
                        </PCDATA>
                     }
                  }


            </$tag2>
         }
      </$tag>
   }

}

Q2(document("result1.xml"))
```

C.5 Universal Table [20]

C.5.1 Step 1

```
FUNCTION GetGroup($iid, $tag, $dxv, $schema)
{
   FOR $candidate IN $dxv/UNIVERSAL/UNIVERSAL.ROW[./*[CONTAINS(name(.), "_TARGET") OR name(.)="SOURCE"]/text()=$iid]
   LET $order := TRIM($candidate/*[TRIM(name(.))=CONCAT(TRIM($tag), "_ORDER")]),
       $target := TRIM($candidate/*[TRIM(name(.))=CONCAT(TRIM($tag), "_TARGET")])
   RETURN
   {
      <XXX>
         <ORDER>$order</ORDER>,
         <TARGET>$target</TARGET>
      </XXX>
   }
}

FUNCTION Q1($pid, $roottag, $dxv, $schema)
{
   FOR $candidate IN DISTINCT GetGroup($pid, $roottag, $dxv, $schema)
   LET $schemachilds := $schema//*[TRIM(./@name)=TRIM($roottag)]/xs:element UNION
                        $schema//*[TRIM(./@name)=TRIM($roottag)]/xs:choice/xs:element,
       $order := TRIM($candidate/ORDER/text()),
       $target := TRIM($candidate/TARGET/text())
   WHERE $order != ""
   RETURN
   {
      <$roottag xcubesort=$order>
            IF ( COUNT($schemachilds)=0 )
            THEN
            {
               <PCDATA xcubesort="1">
                  $target
               </PCDATA>
            }
            ELSE
            {
               FOR $schemachild IN $schemachilds
               RETURN
               {
                  Q1($target, TRIM($schemachild/@name), $dxv, $schema)
               }

            }

      </$roottag>
   }
}

LET $dxv := document("dxv.xml"),
    $schema := {<XXX>document("source.xsd") FILTER ( self::xs:element OR self::xs:choice OR
                self::xs:attribute )</XXX>},
    $roottag := $schema/*[1]/@name
RETURN
{
   Q1("0.0", $roottag, $dxv, $schema)
}
```

C.6 Composed Extraction

C.6.1 Step 1

```
FUNCTION TrimIt($set)
```

```
{
   FOR $elem IN $set
   LET $tag := gettag($elem)
   RETURN
   {
      <$tag>TRIM($elem/text())</$tag>
   }
}

FUNCTION GetAttributes($columnlist, $roottag)
{
   FOR $att IN $columnlist
   LET $name := TOKEN(gettag($att), "_", "2")
   WHERE gettag($att)=CONCAT($roottag, "_", TOKEN(gettag($att), "_", "2"), "_ATT")
   RETURN
   {
      <$name>TRIM($att/text())</$name>
   }
}

FUNCTION GetGroup($iid, $tag, $dxv, $schema)
{
   FOR $candidate IN $dxv/UNIVERSAL/UNIVERSAL.ROW[./*[CONTAINS(name(.), "_TARGET") OR name(.)="SOURCE"]/text()=$iid]
   LET $order := TRIM($candidate/*[TRIM(name(.))=CONCAT(TRIM($tag), "_ORDER")]),
       $target := TRIM($candidate/*[TRIM(name(.))=CONCAT(TRIM($tag), "_TARGET")])
   RETURN
   {
      <XXX>
         <ORDER>$order</ORDER>,
         <TARGET>$target</TARGET>
      </XXX>
   }
}

FUNCTION Reverse($pid, $roottag, $dxv, $schema, $mappinglist, $flag)
{
   LET $extramapping := $mappinglist/*[TRIM(name(.))=TRIM($roottag)]/@mapping,
       $treemapping := $mappinglist/XCUBEDOC/@mapping,
       $mapping := {IF ( COUNT($extramapping)=0 )
          THEN
          {
            $treemapping
          }
          ELSE
          {
            $extramapping
          }
         }
   RETURN
   {

IF ( $mapping="Inline" OR $flag=0 )
THEN
{

IF ( $flag=0 )
THEN
{
   LET $tag := TOKEN($pid, "_", "1"),
       $order := TOKEN($pid, "_", "2")
   RETURN
   {
      IF ( NOT (TOKEN($pid, "_", "3")) )
      THEN
      {
         <$tag inlineorder=$order>
            $roottag
         </$tag>
      }
      ELSE
      {
         IF ( TOKEN($pid, "_", "3")="IID" )
         THEN
         {
            <$tag inlineorder=$order>
               FOR $schemachild IN ($schema//xs:element[./@name=$tag AND NOT ./@ref]/xs:element) UNION
                         ($schema//xs:element[./@name=$tag AND NOT ./@ref]/xs:choice/xs:element)
               LET $schematag := TRIM($schemachild/@name)
               RETURN
                  {
                     Reverse(TRIM($roottag), $schematag, $dxv, $schema, $mappinglist, "1")
                  }
            </$tag>
         }
         ELSE
         {
            IF ( TOKEN($pid, "_", "4")="ATT" )
            THEN
            {
               LET $attname := TOKEN($pid, "_", "3"),
                   $att := {<$attname>$roottag</$attname>}
               RETURN
               {
                  <$tag inlineorder=$order $att />
               }
            }
            ELSE
            {
               <$tag inlineorder=$order>
                  Reverse(CUTTOKEN(CUTTOKEN($pid, "_"), "_"), $roottag, $dxv, $schema, $mappinglist, "0")
               </$tag>
            }
         }
      }
   }
}
ELSE
{
   FOR $root IN $dxv/*[TRIM(name(.))=TRIM($roottag)]/*[TRIM(./PID)=TRIM($pid)]
   LET $order := TRIM($root/ORDER/text()),
       $schemachilds := ($schema//xs:element[./@name=$roottag AND NOT ./@ref]/xs:element) UNION
                        ($schema//xs:element[./@name=$roottag AND NOT ./@ref]/xs:choice/xs:element),
       $atts := GetAttributes($root/*, $roottag)
   RETURN
   {
      <$roottag xcubesort=$order $atts>
            FOR $schemachild IN $schemachilds
            LET $tag := TRIM($schemachild/@name)
            RETURN
            {
               Reverse(TRIM($root/IID), $tag, $dxv, $schema, $mappinglist, "1")
            },
            FOR $pcdata IN $dxv/PCDATA/*[TRIM(./PID)=TRIM($root/IID)]
            LET $pcdataorder := TRIM($pcdata/ORDER/text())
            RETURN
            {
               <PCDATA xcubesort=$pcdataorder>
                  TRIM($pcdata/VALUE/text())
               </PCDATA>
            },
            FOR $inlinedelem IN $root/*[TOKEN(name(.), "_", "3")!="ATT" AND name(.)!="IID" AND
                                name(.)!="PID" AND name(.)!="ORDER"]
            LET $value := TRIM($inlinedelem/text())
            RETURN
            {
               Reverse(CUTTOKEN(gettag($inlinedelem), "_"), $value, $dxv, $schema, $mappinglist, "0")
            }
      </$roottag>
   }
}
}
ELSE
{ IF ( $mapping="Edge" )
  THEN
  {
   FOR $root IN $dxv/EDGES/*[TRIM(./NAME)=TRIM($roottag) AND TRIM(./SOURCE)=$pid]
   LET $order := TRIM($root/ORDER/text()),
       $schemachilds := $schema//*[./@name=$roottag AND NOT ./@ref]/xs:element UNION
                        $schema//*[./@name=$roottag AND NOT ./@ref]/xs:attribute UNION
                        $schema//*[./@name=$roottag AND NOT ./@ref]/xs:choice/xs:element,
       $iid := TRIM($root/TARGET/text())
   RETURN
   {
      <$roottag xcubesort=$order>
         IF ( COUNT($schemachilds)=0 )
         THEN
         {
            <PCDATA xcubesort="1">
               $iid
            </PCDATA>
         }
         ELSE
         {
            FOR $schemachild IN $schemachilds
            LET $tag := TRIM($schemachild/@name)
            RETURN
            {
               Reverse($iid, $tag, $dxv, $schema, $mappinglist, $flag)
            }
         }
      </$roottag>
   }
  }
ELSE
{ IF ( $mapping="Clock" )
  THEN
  {
   FOR $root IN $dxv/*[TRIM(name(.))=TRIM($roottag)]/*[TRIM(./PID)=TRIM($pid)]
   LET $order := TRIM($root/ORDER/text()),
       $schemachilds := $schema//*[./@name=$roottag AND NOT ./@ref]/xs:element UNION
                        $schema//*[./@name=$roottag AND NOT ./@ref]/xs:choice/xs:element,
       $atts := TrimIt($root/*[name(.)!="ORDER" AND name(.)!="IID" AND name(.)!="PID"]),
       $iid := TRIM($root/IID/text())
   RETURN
   {
      <$roottag xcubesort=$order $atts>
         IF ( COUNT($schemachilds)=0 )
         THEN
         {
            LET $pcdata := $dxv/PCDATA/*[./PID=$root/IID],
                $pcdataorder := TRIM($pcdata/ORDER/text())
            RETURN
            {
               <PCDATA xcubesort=$pcdataorder>
                  TRIM($pcdata/VALUE/text())
               </PCDATA>
            }
         }
         ELSE
         {
            IF ( $schema//*[./@name=$roottag AND NOT ./@ref]/xs:choice )
            THEN
            {
               FOR $schemachild IN $schemachilds
               LET $tag := $schemachild/@name
               RETURN
               {
                  Reverse($iid, $tag, $dxv, $schema, $mappinglist, $flag)
               },
               FOR $pcdata IN $dxv/PCDATA/*[TRIM(./PID)=TRIM($iid)]
               LET $pcdataorder := TRIM($pcdata/ORDER/text())
               RETURN
               {
                  <PCDATA xcubesort=$pcdataorder>
                     TRIM($pcdata/VALUE/text())
                  </PCDATA>
               }
            }
            ELSE
            {
               FOR $schemachild IN $schemachilds
               LET $tag := $schemachild/@name
               RETURN
               {
                  Reverse($iid, $tag, $dxv, $schema, $mappinglist, $flag)
               }
            }
         }
      </$roottag>
   }
  }
ELSE
{ IF ( $mapping="Attribute" )
  THEN
  {
   FOR $root IN $dxv/*[TRIM(name(.))=TRIM($roottag)]/*[./SOURCE=$pid]
   LET $order := TRIM($root/ORDER/text()),
       $schemachilds := $schema//*[./@name=$roottag AND NOT ./@ref]/xs:element UNION
                        $schema//*[./@name=$roottag AND NOT ./@ref]/xs:attribute UNION
                        $schema//*[./@name=$roottag AND NOT ./@ref]/xs:choice/xs:element,
       $iid := TRIM($root/TARGET/text())
   RETURN
   {
      <$roottag xcubesort=$order>
         IF ( COUNT($schemachilds)=0 )
         THEN
         {
            <PCDATA xcubesort="1">
               $iid
            </PCDATA>
         }
         ELSE
         {
            FOR $schemachild IN $schemachilds
            LET $childtag := $schemachild/@name
            RETURN
            {
               Reverse($iid, $childtag, $dxv, $schema, $mappinglist, $flag)
            }
         }
      </$roottag>
   }
  }
ELSE
{ IF ( $mapping="Universal" )
  THEN
  {
     FOR $candidate IN DISTINCT GetGroup($pid, $roottag, $dxv, $schema)
   LET $schemachilds := $schema//*[TRIM(./@name)=TRIM($roottag)]/xs:element UNION
                        $schema//*[TRIM(./@name)=TRIM($roottag)]/xs:choice/xs:element,
       $order := TRIM($candidate/ORDER/text()),
       $target := TRIM($candidate/TARGET/text())
   WHERE $order != ""
   RETURN
   {
      <$roottag xcubesort=$order>
            IF ( COUNT($schemachilds)=0 )
            THEN
            {
               <PCDATA xcubesort="1">
                  $target
               </PCDATA>
            }
            ELSE
            {
               FOR $schemachild IN $schemachilds
               RETURN
               {
                  Reverse($target, TRIM($schemachild/@name), $dxv, $schema, $mappinglist, $flag)
               }

            }

      </$roottag>
   }
  }
  ELSE
  {
   ""
}}}}}

}
}

FUNCTION Tree($root, $mapping )
{
   LET $element := document("source.xsd")//xs:element[./@name=$root],
       $elemname := $element/@name
   RETURN
   {
      <$elemname mapping=$mapping/>,
      FOR $child IN $element//xs:element
      LET $childname := $child/@name
      RETURN
      {
         <$childname mapping=$mapping/>
      }
   }
}


LET $dxv := document("dxv.xml"),
    $schema := {<XXX>document("source.xsd") FILTER ( self::xs:element OR self::xs:choice OR
                self::xs:attribute )</XXX>},
    $roottag := $schema/*[1]/@name,

    $mappinglist := {<ROOT>
             <XCUBEDOC mapping="Inline" />,
             <TITLE mapping="Attribute" />,
             <AUTHOR mapping="Edge" />,
             <NAME mapping="Edge" />
               </ROOT>}
RETURN
{
   Reverse("0.0", $roottag, $dxv, $schema, $mappinglist, "1")
}
```

C.6.2 Step 2

```
FUNCTION Q2($root)
{
   LET $tag := gettag($root),
       $childs1 := $root/*[NOT ./@inlineorder],
       $childs2 := $root/*[./@inlineorder],
       $order := NUMFORMAT("######", TRIM(CONCAT($root/@xcubesort, $root/@inlineorder))),
       $atts := $root/@*[name(.)!="xcubesort" AND name(.)!="inlineorder"]
   RETURN
   {
      <$tag $atts xcubesort=$order>
         IF ( COUNT($childs1)=0 )
         THEN
         {   IF ($tag="PCDATA" )
            THEN
            {
               $root/text()
            }
            ELSE
            {
               <PCDATA xcubesort="1">
                  $root/text()
               </PCDATA>
            }
         }
         ELSE
         {
            For $child1 IN $childs1
            RETURN
            {
               Q2($child1)
            }
         },
         FOR $dist IN DISTINCT (FOR $xxx IN $childs2
                    RETURN{<XXX><X1>gettag($xxx)</X1>,<X2>$xxx/@inlineorder/text()</X2></XXX>})
         LET $bag := $childs2[TRIM(name(.))=TRIM($dist/X1/text()) AND TRIM(./@inlineorder)=TRIM($dist/X2/text())],
             $tag2 := $dist/X1/text(),
             $atts := $bag/@*[name(.)!="xcubesort" AND name(.)!="inlineorder"],
             $text := CONCAT(TRIM($bag/text()))
         RETURN
         {
            <$tag2 xcubesort=$dist/X2 $atts>
                  IF ( $bag/* )
                  THEN
                  {
                     FOR $elem IN $bag,
                         $element IN $elem/*
                     RETURN
                     {
                        Q2($element)
                     }
                  }
                  ELSE
                  {
                     IF ( $tag2="PCDATA" )
                     THEN
                     {
                        $bag/text()
                     }
                     ELSE
                     {
                        <PCDATA xcubesort="1">
                           $bag/text()
                        </PCDATA>
                     }
                  }


            </$tag2>
         }
      </$tag>
   }

}

Q2(document("result1.xml"))
```

C.7   Final Ordering Steps (Similar for all Extractions)

C.7.1   Step 3

```
FUNCTION OrderIt($root)
{
   LET $tag := gettag($root)
   RETURN
   {
      <$tag $root/@*>
         IF ( $tag="PCDATA" )
         THEN
         {
            TRIM($root/text())
         }
         ELSE
         {
            FOR $child IN $root/*
            RETURN
            {
               OrderIt($child)
            }
            SORTBY (@xcubesort)
         }
      </$tag>
   }
}

OrderIt(document("result2.xml"))
```

C.7.2   Step 4

```
FUNCTION Q4($root)
{
   LET $tag := gettag($root),
       $args := $root/@*[name(.)!="xcubesort"],
       $childs := $root/*
   RETURN
   {
      IF ( $tag="PCDATA" )
      THEN
      {
         TRIM($root/text())
      }
      ELSE
      {
         <$tag $args>
            IF ( COUNT($childs)=0 )
            THEN
            {
               TRIM($root/text())
            }
            ELSE
            {
               FOR $child IN $childs
               RETURN
               {
                  Q4($child)
               }
            }
      </$tag>
      }
   }
}

Q4(document("result3.xml"))
```