**The Florida State University**

**College of Arts and Sciences**

# An XML–Database Interface System

**By**

**Lakshmi Valsalakumari**

TR-020401

April 2002

A project submitted to the Department of Computer Science in partial fulfillment of the

requirements for the degree of Master of Science in Computer Science

**Major Professor**
**Dr. Greg Riccardi**
Supervisory Committee Members
Dr. Lois Hawkes
Dr. Robert Van Engelen

# TABLE OF CONTENTS

# Abstract

This paper describes an XML-Database interface system that transforms information from a hierarchical XML representation into persistent, relational database data and vice-versa. Information in XML documents is used to update the status of a corresponding database, making the database consistent with the XML file content. Additionally, database queries submitted via an XML document are processed and the resulting information output as XML files. The transformations in both directions (XML-database and database-XML) are done using an XML-Database interface library developed in Java, designed with the special needs and features of scientific systems in mind. The library is database-independent. It is designed to achieve the required transformations, irrespective of the database schema. State-of-the-art concepts in processing XML have been used to implement the library. SAX technology is used to process and transform the data, and DOM to generate the transaction/query results information. The transformations preserve the context of the information. Only recognized elements in recognized contexts are processed and transformed. The remaining elements are passed on, untransformed. The Apache Cocoon pipelining strategy has been used to achieve this. All database operations are done using standard JDBC classes to ensure database independence.

# Acknowledgements

# 1. Introduction

My objective in implementing this project was to develop an XML-Database interface system that transforms information from a hierarchical XML representation into persistent, relational database data and vice versa. The system would accept information from XML files and use it to update a corresponding relational database. I also had to come up with a method to query the database and report the results in XML.

XML is in the forefront of technologies for exchanging information between systems [9]. Key players in the technology industry rely on XML to launch some of the latest technologies and business models involving information transfer [2]. Web Services, Microsoft's SOAP, and the newly launched .Net are just a few examples.

With XML so widely used to transfer information, systems to process XML and transform it into persistent database data are of immense relevance. This is reflected in the number of XML-enabled databases and middleware technologies to process XML being made available in the market [10].

I started with a survey of current technologies used to process XML. Parsers to process XML documents were readily available. Extensible Stylesheet Language (XSL) was a powerful technology used to transform XML documents [4].

I had to take into account the inherent differences between XML and relational databases in building an interface system between the two. XML is hierarchical in nature and represents information as a tree of nested elements. Relational databases, on the other hand, are flat, and use shared attributes between tables, to represent relationships.

As the first step in implementing the system, I designed *dbtxns* schema—an XML schema for arbitrary relational database information. Any application-specific XML document would first be transformed into dbtxns.xml. I used XSL to achieve this transformation. The dbtxns.xml document

would be the input to the XML-Database interface library. Figure 1.1 shows an overview of the transformations.
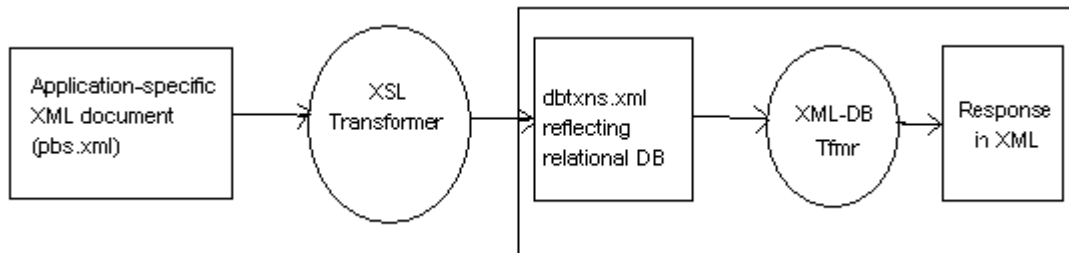


Figure 1.1 Overview of the transformations

The next step was to develop the interface library itself, in Java, to do the actual XML–Database transformations. I designed a *DBTransactions Transformer (DbTxnsTxmr)* to achieve this. The transformer connects to the database and verifies whether the information represented by the document is present in the database or not. It inserts new records if the information is not present. Otherwise, it updates the existing records. The transformer also reports the database transaction results as an XML document. The transformer uses standard JDBC classes to perform the database operations, thus ensuring database independence.

I also had to develop a method to support queries against the database. I extended the XML-Database interface library to include a *DBQuery Transformer (DbQueryTxmr)* component. The query transformer would transform SQL queries embedded in XML documents into query results.

Figure 1.2 outlines the architecture of the system.
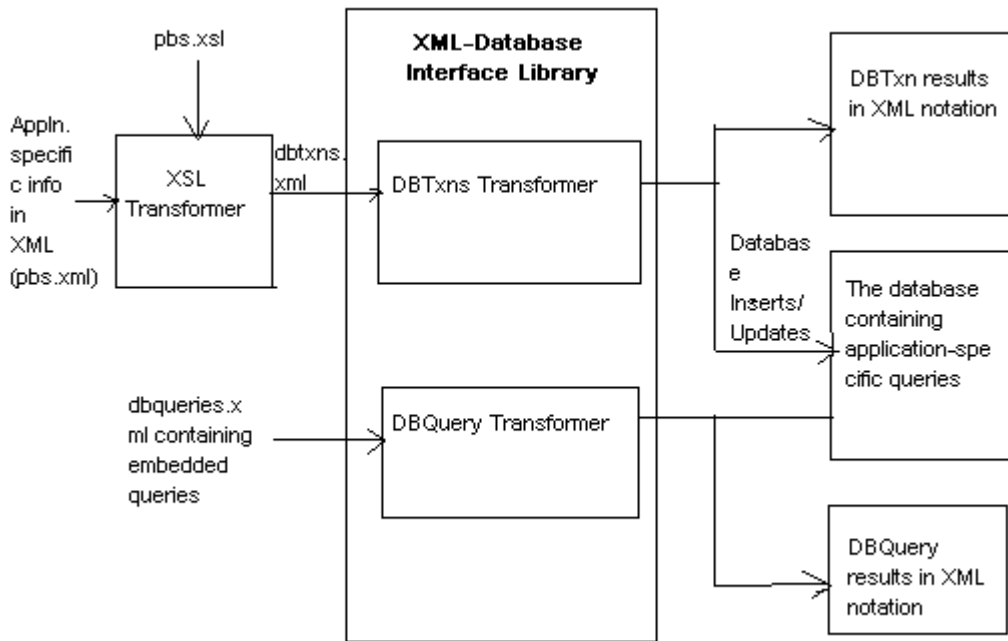
Figure 1.2 The XML-Database Interface System

This paper describes in detail how the XML-Database interface system has been implemented. It starts with a brief introduction to the technologies used. Chapter 3 describes a sample application on which the XML-Database transformations were carried out. Chapters 4 and 5 go into the implementation aspects in detail. Chapter 6 sums up the project and outlines possible future extensions.

# 2. An Overview of Technologies Used

## 2.1 XML—A Brief Introduction

XML, Extensible MarkUp Language, is a widely accepted standard for exchanging information between systems [1]. XML is a markup language like HTML.

Figure 2.1 shows an XML document, describing an employee.

```
1      <employee id="10093">
2              <name>John Smith</name>
3              <department>IT</department>
4              <address>
5                  <apt>230 City Towers </apt>
6                  <street>Newport Avenue</street>
7                  <city>Tallahassee</city>
8                  <state>FL</state>
9              </address>
10     </employee>
```

Figure 2.1 An XML document–employee.xml

Like HTML, XML makes use of tags (words bracketed by '<' and '>') and attributes (of the form name="value") to mark up text.

Line 1 of Fig. 2.1 is a tag, with employee as the tag name. It has an attribute id, with the value "10093". <employee> is a begin tag. </employee> in line 10 is its corresponding end tag.

The free text that occurs outside the tags is character data, or content. "John Smith" in Line 2 of Fig. 2.1 is an example of character data.

Unlike in HTML, XML allows the use of one's own tags for markup.

XML enforces strict rules for a document to be well-formed. A well-formed XML document is one that corresponds to the XML 1.0 grammar specified by W3C [1]. A well-formed XML document has exactly one root element. `<employee>` is the root element of the XML document in Fig. 2.1.

Each start element (begin tag) in XML should have a corresponding end element (end tag). The elements should be nested within one another. The tags and nesting rules allow XML to represent information in a hierarchical manner [9].

## 2.2 Processing XML

XML documents are read and processed with XML parsers. Parsers provide access to the documents' content and structure, and ensure that the documents are well-formed [2]. Parsers can be validating or non-validating. A validating XML parser ensures that the XML document is valid, following rules specified in an associated Document Type Definition (DTD) or XML schema [1].

The Simple API for XML (SAX) and the Document Object Model (DOM) are two common techniques to access and process the information content of an XML document [2].

### 2.2.1 SAX

SAX, although not an official W3C recommendation, has become a de facto industry standard for parsing XML, due to its simplicity and processing speed. SAX is a lightweight event-driven XML processing model [6]. SAX events include element starts and ends, character data, and processing instructions among others.

Table 2.1 lists the partial sequence of SAX events triggered on parsing employee.xml.

| SAX Event | On Encountering |
|---|---|
| startDocument | the start of the XML document |
| startElement | the start tag of element employee |
| startElement | the start tag of element name |
| characters | parseable character data inside name |

| endElement | the end tag of element name |
|---|---|
| ... | ... |
| endElement | the end tag of element employee |
| endDocument | the end of the XML document |

Table 2.1 Partial sequence of SAX events

SAX allows the application to receive extracted data directly from the parser [6]. Figure 2.2 shows how an application can process SAX events.

```
1   public void startElement(String uri, String local, String raw, Attributes
    attrs) {
2       System.out.println("Encountered start of element: " + local);
3   }
4   public void endElement(String uri, String local, String raw) {
5       System.out.println("Encountered end of element: " + local );
6   }
7   public void characters(char[] ch, int start, int length) {
8       System.out.println("Encountered text: ");
9       for (int index=0; index < length; index++) {
10          System.out.print(ch[index]);
11      }
12  }
```

Figure 2.2 An application processing the SAX events

The methods startElement, endElement and characters are examples of Application Programming Interfaces (APIs) defined by SAX [6]. Lines 1 to 3 of Figure 2.2 define the startElement interface. This defines the action to be taken on encountering the start tag of an element. Line 2 of the method specifies that the element name is to be printed out to standard output.

Lines 4 to 6 specify the action to be taken on encountering the end tag of an element. This is done through the endElement method.

Lines 7 to 12 specify the action to be taken on encountering character data, through the characters method.

Figure 2.3 shows the output of this piece of code, when used to process employee.xml.

```
Encountered start of element: employee
Encountered start of element: name
Encountered text: John Smith
Encountered end of element: name
Encountered start of element: department
Encountered text: IT
Encountered end of element: department
Encountered start of element: address
Encountered start of element: apt
Encountered text: 230 City Towers
Encountered end of element: apt
Encountered start of element: street
Encountered text: Newport Avenue
Encountered end of element: street
Encountered start of element: city
Encountered text: Tallahassee
Encountered end of element: city
Encountered start of element: state
Encountered text: FL
Encountered end of element: state
Encountered end of element: address
Encountered end of element: employee
```

Figure 2.3 Output of processing employee.xml

Attractive features of SAX include its small memory footprint, and fast response time for processing, since application processing is concurrent with parsing.

**2.2.2 DOM**

DOM, the official W3C Document Object Model, is the second technique applications can use to access and process data inside an XML document. The document model is an in-memory representation of an XML document as a tree of connected nodes [8].

Figure 2.4 shows the DOM tree for employee.xml.
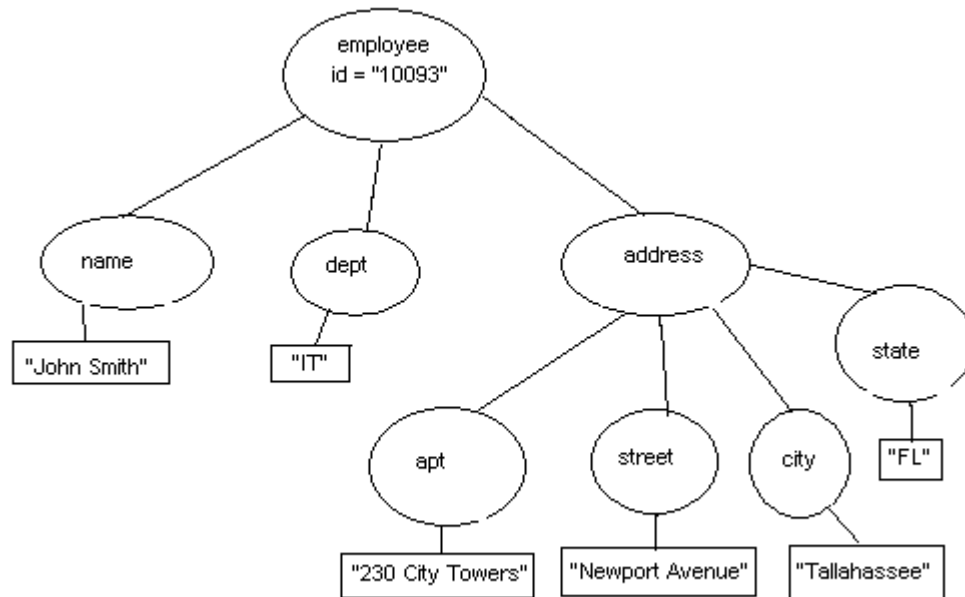


Figure 2.4 DOM tree for employee.xml

The DOM parser performs three functions—it represents the XML document in memory as a tree, provides methods to manipulate the in-memory tree, and provides methods to create new XML elements [3].

The primary benefit of DOM over SAX is that the entire document information is available in memory in context. Hence arbitrary processing and transformation of elements is possible. However, a drawback of this approach is that application processing has to wait till the entire document is parsed and the document tree constructed in memory.

This project uses both SAX and DOM technologies to process and transform the XML data.

## 2.3 Validating The XML Documents

XML allows the user to specify the structure of an XML document by specifying an associated Document Type Definition (DTD). A DTD identifies the elements that can be present in an XML

document, the attributes an element can have and their default values, and the hierarchical relationships between the different elements [1].

Figure 2.5 shows the DTD for employee.xml.

```
1          <?xml version='1.0' encoding='UTF-8' ?>
2          <!ELEMENT employee (name, dept, address)>
3          <!ATTLIST employee  id CDATA  #REQUIRED >
4          <!ELEMENT name (#PCDATA)>
5          <!ELEMENT dept (#PCDATA)>
6          <!ELEMENT address(apt, street, city, state)>
7          <!ELEMENT apt (#PCDATA)>
8          <!ELEMENT street (#PCDATA)>
9          <!ELEMENT city (#PCDATA)>
10         <!ELEMENT state (#PCDATA)>
```

Figure 2.5 A DTD for employee.xml

Line 2 specifies that `employee` is an element containing the elements `name`, `dept`, `address`. Line 2 states `employee` has a required attribute `id`. Line 3 specifies that `name` is an element containing parseable character data (PCDATA).

This project uses DTDs to define the structure of XML documents. A validating parser can validate an XML document against its specified DTD.

We anticipate our input documents to contain several unrecognized elements which are not of interest to us, but which nevertheless, should not hamper our processing of recognized elements. We therefore follow a *weak validation strategy* in our project, wherein we identify and transform recognized elements alone, and just pass on the remaining unrecognized elements. This is done recognizing and preserving the context of various elements.

`<dbtxns>` and `<dbqueries>` are the elements the XML-Database library recognizes and processes, transforming them into `<dbTxnsResults>` and `<dbQueryResults>` respectively.

Figure 2.6 shows an XML document, containing a `<dbtxns>` element, together with other unrecognized elements. Line 1 shows an unrecognized element, `<foo>`. Line 2 shows the start of `<dbtxns>`, which ends at Line 12 with `</dbtxns>`. Line 13 has `</foo>`, the end tag of `foo`.

```
1   <foo>
2       <dbtxns>
3           <dbtxn>
4               <dbtable name="JOB">
5                   <dbrow>
6                       <dbfield name="id" value="20692@i5" key="true"
                        />
7                       <dbfield name="jobName" value="b6p45" />
8                       <dbfield name="status" value="R" />
9                   </dbrow>
10              </dbtable>
11          </dbtxn>
12      </dbtxns>
13  </foo>
```

Figure 2.6 XML File with recognized and unrecognized elements

DbTxnsTxmr transforms `<dbtxns>` into `<dbTxnsResults>`. Figure 2.7 shows the output XML document after the transformations.

```
1   <foo>
2       <dbTxnsResults>
3           <updateResult>
4               <Success>1 row(s) updated in JOB</Success>
5           </updateResult>
6       </dbTxnsResults>
7   </foo>
```

Figure 2.7 The Transformed Output

`<dbtxns>` has been transformed into `<dbTxnsResults>` preserving its context. The unrecognized elements are not transformed.

## 2.4 XSL and XSLT

XSL (Extensible Stylesheet Language) is a powerful language that allows one to rearrange elements, and transform XML documents in almost arbitrary ways [3]. XSL is a language that defines the transformations to be applied. XSLT, the XSL Transformer, does the transformations.

Figure 2.8 shows how XSL transformations work. The XSL transformer reads both an XML document and an XSL style sheet. Based on instructions found in the style sheet, XSLT outputs a new, transformed XML document.
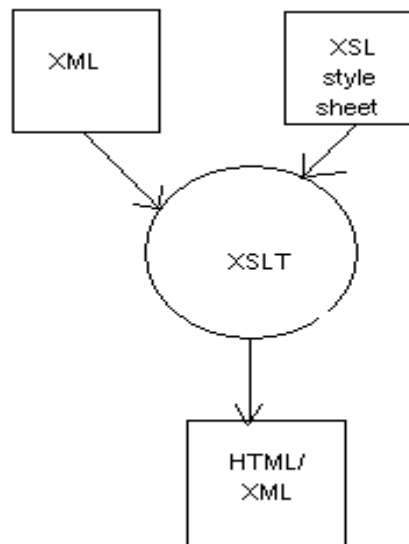
Figure 2.8 Transformations with XSL

An XSL stylesheet is a well-formed XML document. The basic building block of an XSL style sheet is a *template rule*. A template rule consists of two parts: 1) A pattern that identifies an element in the XML document; and 2) an action that specifies the transformation and rendering of the identified node [3].

A typical XSLT construct would be:

```
<xsl:template match = "/">
    <xsl:apply-templates/>
```

```
</xsl:template>
```

XSLT allows elements to be identified (matched) in various ways as outlined below [5].

- by name: <xsl:template match="employee" >

- by ancestry: <xsl:template match = "employee//state" >

- by attributes:  <xsl:template match = "employee[@id="1234"]" >

- by children: <xsl:template match = "employee[name]" >

- by position: <xsl:template match = "employee[lst-of-type()]" >

- by matching several element names: <xsl:template match="employee|manager" >

A key feature of XSLT is its recursive nature. Starting from the root element of the XML document, each node is inspected to see if there is an applicable template rule. In case of more than one rule being applicable to a particular node, XSLT allows priorities to be specified for the transformation. The relevant rule is invoked. The inspection then continues on the children of the new node. `<xsl:apply-templates>` is used to apply this recursive action.

XSLT allows conditional processing of elements with `xsl:if` and `xsl:choose` constructs. `<xsl:apply-templates select="xxx"/>` can be used to process selected children alone of a matched element. `<xsl:for-each>` construct can be used to process each of repetitive elements. Sorting and numbering of elements are other manipulations that can be performed.

This project uses the transformation capabilities of XSL to transform application-specific input XML files into `dbtxns.xml`. This is outlined in detail in Chapter 3.

## 2.5 Cocoon and The XML Pipeline Concept

Apache Cocoon, an open source project under the Apache Software Foundation, is an XML publishing framework that uses XML and XSL technologies. Cocoon 2 aims at completely separating

the content, style and logic of creating web documents—allowing sites to be independently designed, created and managed [7].

Cocoon uses the notion of an XMLProducer and an XMLConsumer. An XML Producer generates XML. A class implementing Cocoon's XMLProducer interface generates SAX events. Such a class will have a `setConsumer` method, which can be used to configure an XMLConsumer.

An XMLConsumer *consumes* XML data. Such a class receives notification of SAX events, and acts on them. This sets up a very simple and basic XML pipeline—a producer generating SAX events and sending them down to a consumer, which acts on receiving them.

Cocoon relies on a pipeline model of processing XML. An XML document is pushed through a pipeline. The pipeline consists of several transformation steps applied to the document, as shown in Fig 2.9. Every pipeline begins with a generator, continues with zero or more transformers, and ends with a serializer.
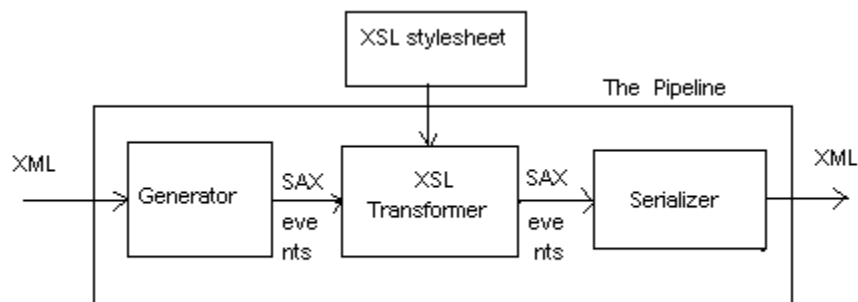


Figure 2.9 A Cocoon Pipeline

The Generator is the starting point for the pipeline. The simplest Generator in Cocoon is the FileGenerator. It accepts a local XML document or URL, parses it, and sends SAX events down the pipeline.

A Transformer receives SAX events, transforms it, and generates SAX events, which it sends down the pipeline. A transformer is thus a consumer as well as a producer of XML. Its function is

17

similar to XSL. The Cocoon project includes the XalanTransformer, which applies XSL to the SAX events it receives.

A Serializer transforms a sequence of SAX events into a presentation format. Cocoon provides serializers for generating HTML, XML, PDF, etc., as well as the ability to develop other serializers.

This project has borrowed Cocoon's pipeline concept and extended several of its source classes to achieve the required transformations. This is elaborated in detail in the implementation sections.

# 3. An Example Application

This chapter describes how XML-Database transformations are carried out on a sample application.

The Portable Batch System (PBS) is a batch queuing and workload management system, which executes jobs on various system nodes upon user requests. It operates on networked, multi-platform UNIX environments, including heterogeneous clusters of workstations, supercomputers, and massively parallel systems [11].

Jobs are submitted to PBS with the `qsub` command. PBS maintains information about each job while it is executing. The `qstat -f` command provided by PBS can be used to output the status of each job. This command provides information about the job requested. The information provided includes its status, the nodes on which it is executing, the configuration and environment variables under which the job is being executed, and so on.

As part of prior work done on PBS, Perl Scripts have been written to output the PBS job information as XML documents. XML files containing job information from PBS were thus available. These files, containing PBS-specific markup tags, would be the input XML files for the XML-Database Interface library to transform.

## 3.1 The Input XML Files

Figure 3.1 shows a sample PBS file for the job, '20692@inter5'.

```
1       <job id="20692@inter5">
2               <jobName>b6p45</jobName>
3               <userName>heller</userName>
4               <resourceUsed>
5                   <cpuTimeUsed>02:27:15</cpuTimeUsed>
6                   <memoryUsed>20524kb</memoryUsed>
7               </resourceUsed>
8               <status>R</status>
```

```
9            <queue>medium</queue>
10           <resourceRequested>
11                <cpuTimeList>24:00:00</cpuTimeList>
12           </resourceRequested>
13           <nodes>
14               <node>
15                    <nodeId>cp201</nodeId>
16                    <cpu>0</cpu>
17               </node>
18               <node>
19                    <nodeId>cp112</nodeId>
20                    <cpu>0</cpu>
21               </node>
22           </nodes>
23           <sessionId>17596</sessionId>
24           <variableList>
25                <variable name="PBS_O_HOME">/home/heller</variable>
26                <variable name="PBS_O_LANG">en_US</variable>
27           </variableList>
28           <comment>Job started on Fri Sep 28 at 12:58</comment>
29           <eligibilityTime>Fri Sep 28 12:59:54 2001</eligibilityTime>
30           </job>
```

Figure 3.1 A PBS XML File

The PBS file of Figure 3.1 contains information on a particular job, 20692@inter5, represented by the <job> root element in Line 1.

Most child elements of <job>, (e.g. jobName, userName), are simple text-only elements, with PCDATA (parseable character data) content. <resourceUsed>, <nodes>, <node>, <resourceRequested>, and <variableList> are the only exceptions, containing child elements within. <nodes> [Line 13] contains repeating <node> elements, and <variableList> [Line 24] has repeating <variable> elements. <job> and <variable> are the only elements with attributes.

## 3.2 The Database

Figure 3.2 shows the database model for PBS. The database has been defined using Oracle 8i on Linux.
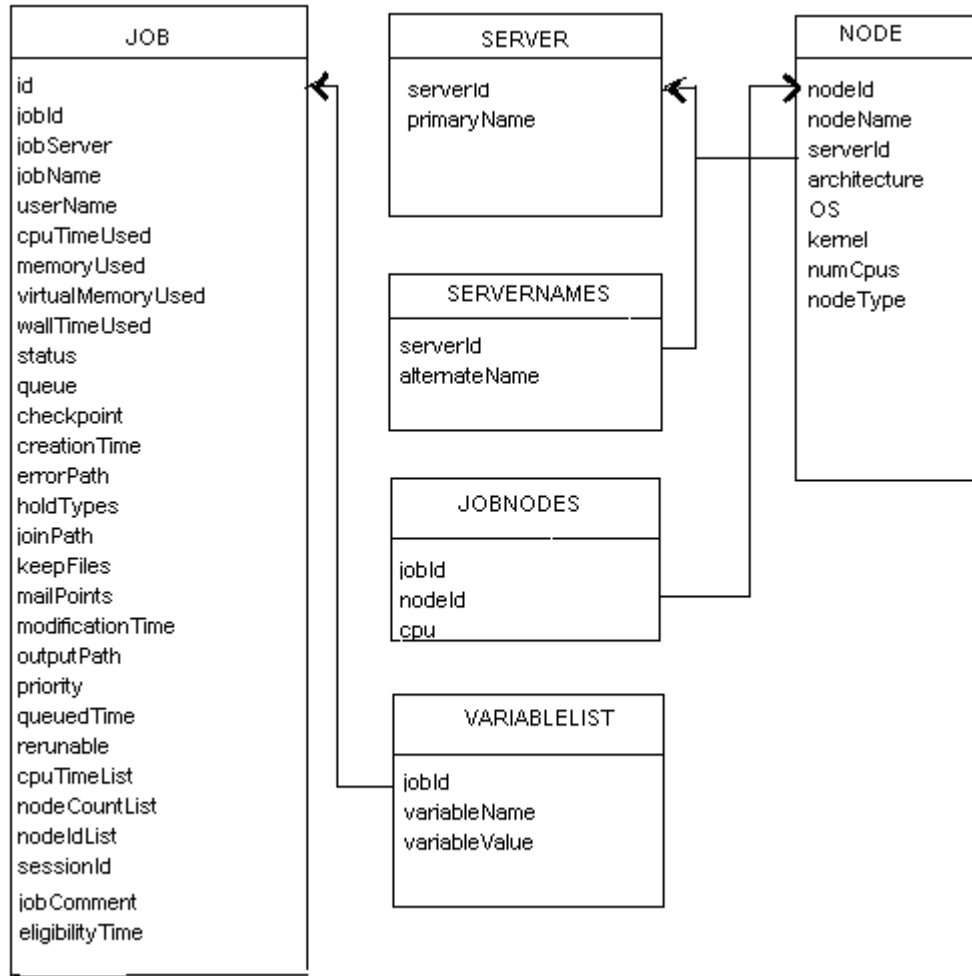


Figure 3.2 The Database Model

The JOB table in Fig 3.2 corresponds to the `<job>` element in the input XML documents. It has `id`, the job Id, as the primary key attribute. Each record in the JOB table corresponds to a single `<job>` element.

JOBNODES and VARIABLELIST tables correspond to <nodes> and <variableList> elements. These contain information corresponding to the repeating <node> and <variable> elements respectively.

SERVER, SERVERNAMES, and NODES are master tables with static information.

The <job> information in the input PBS XML file is to be used to update the corresponding information in the PBS database. For each <job> element in the input file, the corresponding record in JOB table is to be updated, if a record exists for it. If no record exists in the database, a new record is to be inserted. Corresponding updations / insertions are to be done in JOBNODES and VARIABLELIST tables as well.  These steps will bring the database state consistent with the input XML information.

## 3.3 Transformation to dbtxns.xml

A two-step process is followed to achieve the transformation from the PBS XML files into the database.

The first step transforms the PBS XML file into dbtxns.xml. Figure 3.3 shows the result of this transformation.

```
1   <dbtxns>
2       <dbtxn action="insertOrUpdate">
3           <dbtable name="JOB">
4               <dbrow>
5                   <dbfield name="id" value="20692@inter5" key="true" />
6                   <dbfield name="jobName" value="b6p45" key="false" />
7                   <dbfield name="userName" value="heller" key="false" >
8                   <dbfield name="cpuTimeUsed" value="02:27:15"
9                     key="false" />
10                  <dbfield name="memoryUsed" value="20524kb"
11                    key="false" />
12                  <dbfield name="status" value="R" key="false" />
```

```
13              </dbrow>
14          </dbtable>
15          <dbtable name="JOBNODES">
16              <dbrow>
17                  <dbfield name="jobId" value="20692@inter5"
18                    key="true" />
19                  <dbfield name="nodeId" value="cp201" key="true" />
20              <dbfield name="cpu" value="0" key="true" />
21              </dbrow>
22              <dbrow>
23                  <dbfield name="jobId" value="20692@inter5"
24                    key="true" />
24                  <dbfield name="nodeId" value="cp112" key="true" />
25                  <dbfield name="cpu" value="0" key="true" />
26              </dbrow>
27          </dbtable>
28          <dbtable name="VARIABLELIST">
29              <dbrow>
30                  <dbfield name="jobId" value="20692@inter5"
31                    key="true" />
32                  <dbfield name="variableName" value="PBS_O_HOME"
33                    key="true" />
34                  <dbfield name="variableValue" value="/home/heller"
35                    key="false" />
36              </dbrow>
37              <dbrow>
38                  <dbfield name="jobId" value="20692@inter5"
39                    key="true" />
40                  <dbfield  name="variableName" value="PBS_O_LANG"
41                    key="true" />
42                  <dbfield name="variableValue" value="en_US"
43                    key="false" />
44              </dbrow>
45          </dbtable>
46      </dbtxn>
```

47    &lt;/dbtxns&gt;

Figure 3.3 dbtxns.xml for pbs.xml

The PBS markup tags have been transformed into a form representing relational database information, as can be seen. The &lt;dbtxn&gt; element [Line 2] represents a single database transaction. The `action` attribute of &lt;dbtxn&gt; indicates whether the transaction is to be an insert or an update. The &lt;dbtable&gt; elements [Lines 3, 15, 28] indicate the database tables where the action is to take place. &lt;dbfield&gt; elements indicate the database fields to be updated. The `key` attribute of &lt;dbfield&gt; contains information as to whether the database field represents a primary key of the table or not.

## 3.4 Into The Database

dbtxns.xml serves as the input to the second step–to the XML-Database Interface library.

The DB Transactions Transformer component of the XML-Database Interface library handles the &lt;dbtxns&gt; information. It transforms the information within &lt;dbtxns&gt; into database insert and update transactions.

DBTxnsTxmr uses the &lt;dbtable&gt; and &lt;dbfield&gt; information to verify whether a corresponding database table exists in the database, with keys as in the XML document. It then checks whether a corresponding record exists in the database table. This is to decide whether an insert, or an update operation is to be performed.

In the example above, an `insertOrUpdate` action is to be performed on JOB, JOBNODES and VARIABLELIST tables. The primary key of JOB is id. The transformer checks if the job table in the database contains a record with id="20692@inter5". If it does so, it updates this job record with the information in the other &lt;dbfield&gt; elements. If no record with id= "20692@inter5" is found, it inserts a new record in JOB table. Similar logic is used to insert or update information in the VARIABLELIST and JOBNODES tables.

The transformer also outputs the transaction results as an XML document. Figure 3.4 shows the XML document output by the above transformation.

```
1  <dbTxnsResults>
2      <updateResult>
3          <Success>1 row(s) updated in JOB</Success>
4      </updateResult>
5      <updateResult>
6          <Success>1 row(s) updated in JOBNODES</Success>
7      </updateResult>
8      <updateResult>
9          <Success>1 row(s) updated in JOBNODES</Success>
10     </updateResult>
11     <updateResult>
12         <Success>1 row(s) updated in VARIABLELIST</Success>
13     </updateResult>
14     <updateResult>
15         <Success>1 row(s) updated in VARIABLELIST</Success>
16     </updateResult>
17 </dbTxnsResults>
```

Figure 3.4 dbTxnsResults.xml

Line 3 indicates that the record in JOB table corresponding to 20692@inter5 has been updated. Two records each in JOBNODES and VARIABLELIST tables corresponding to the same job id, have been updated as well. The transaction results are reported under <dbTxnsResults> context. <dbtxns> has thus been transformed into <dbTxnsResults>.

## 3.5 Querying From The Database

The project needs to support query facilities against the database as well. Database queries are submitted embedded in XML files. These queries are extracted, executed against the database, and the query results output as XML trees.

Figure 3.5 shows an XML file containing queries against the PBS database.

```
1  <?xml version="1.0" encoding="UTF-8" ?>
2  <dbqueries>
3      <dbquery queryname="job_query" rowname="job">
4          select * from JOB
5      </dbquery>
6      <dbquery queryname="jobnodesFor20692" rowname="jobnode">
7          select * from JOBNODES where jobId='20692@inter5'
8      </dbquery>
9      <dbquery queryname="variablesFor20692" rowname="variable"
10        conn="test">
11        select * from VARIABLELIST where jobId='20692@inter5'
12     </dbquery>
13  </dbqueries>
```

Figure 3.5 dbqueries.xml

The query file in Figure 3.5 conforms to dbqueries schema, explained subsequently.

The Query Transformer component of the XML-Database Interface Library, DbQueryTxmr, transforms <dbqueries> into <dbQueryResults>. Figure 3.6 shows the output of this transformation.

```
1  <dbQueryResults>
2      <dbQueryResult>
3          <job_query>
4              <job>
5                  <ID>20692@inter5</ID>
6                  <JOBNAME>b6p45</JOBNAME>
7                  <USERNAME>heller</USERNAME>
8                  <CPUTIMEUSED>02:27:15</CPUTIMEUSED>
9                  <MEMORYUSED>20524kb</MEMORYUSED>
10                 <SESSIONID>17596</SESSIONID>
11                 <JOBCOMMENT>Job started on Fri Sep 28 at
                       12:58</JOBCOMMENT>
12             </job>
13         </job_query>
```

```
14    </dbQueryResult>
15    <dbQueryResult>
16      <jobnodesFor20692>
17              <jobnode>
18                  <JOBID>20692@inter5</JOBID>
19                  <NODEID>cp107</NODEID>
20                  <CPU>0</CPU>
21              </jobnode>
22              <jobnode>
23                  <JOBID>20692@inter5</JOBID>
24                  <NODEID>cp112</NODEID>
25                  <CPU>0</CPU>
26              </jobnode>
27        </jobnodesFor20692>
28    </dbQueryResult>
29    <dbQueryResult>
30        <variablesFor20692>
31            <variable>
32                <JOBID>20692@inter5</JOBID>
33                <VARIABLENAME>PBS_O_HOME</VARIABLENAME>
34                <VARIABLEVALUE>/home/heller</VARIABLEVALUE>
35            </variable>
36        </variablesFor20692>
37    </dbQueryResult>
38 </dbQueryResults>
```

Figure 3.6 dbQueryResults.xml

# 4. Project Implementation

This chapter describes how the transformations described in Chapter 3 are brought about. It explains the design and implementation of the various transformation components in detail. This chapter does not describe the XML-Database Interface library—the heart of the transformation system. The library implementation is explained in Chapter 5.

## 4.1 dbtxns Schema

The design of the intermediate XML representation—dbtxns—has been a crucial component in achieving the database-independent nature of the XML-DB interface library.

Figure 4.1 lists the dbtxns.dtd schema in its entirety.

```
1         <?xml version='1.0' encoding='UTF-8' ?>
2         <!ELEMENT dbtxns (dbtxn+)>
3         <!ELEMENT dbtxn (dbtable+)>
4         <!ATTLIST dbtxn action  "insertOrUpdate">
5         <!ATTLIST dbtxn conn   CDATA  #IMPLIED>
6         <!ELEMENT dbtable (dbrow+)>
7         <!ATTLIST dbtable  name CDATA  #REQUIRED >
8         <!ELEMENT dbrow (dbfield+)>
9         <!ELEMENT dbfield EMPTY>
10        <!ATTLIST dbfield name  CDATA   #REQUIRED
11                          value CDATA  #REQUIRED
12                          key   (true | false )  'false' >
```

Figure 4.1 dbtxns schema

<dbtxns> is the root element of the schema, containing multiple <dbtxn> elements [Line 2]. Each <dbtxn> element corresponds to a single database transaction, to be acted on the database accordingly, to preserve database consistency.

<dbtxn> has an action attribute, defaulting to insertOrUpdate [Line 4]. This implies that the transaction to be performed can be either an update or an insert. Nothing more is known at this stage. In other words, the transformer has to figure this information out by itself.

<dbtxn> has an optional connection attribute, conn [Line 5]. This attribute has been included with future extensions in mind, wherein, a particular connection can be specified to connect to a specific database. At present, this attribute is not processed, and a default connection to a default database instance is used every time.

A <dbtxn> element consists of multiple <dbtable> elements [Line 3]. Each <dbtable> corresponds to a database table. The name of the table is represented by dbtable's mandatory name attribute [Line 7].

A <dbtable> element consists of multiple <dbrow> elements [Line 6]. Each <dbrow> element corresponds to one database record in the database table represented by <dbtable>.

Each <dbrow> element consists of multiple <dbfield> elements [Line 8]. A <dbfield> element corresponds to a database table field, and has name—the field name, value—the field value, and key— whether the field is part of the primary key of the table or not, as its attributes.

DBTxnsTxmr processes the XML content it finds inside the <dbtxns> context, transforming it into flattened relational database data. The transformer uses the <dbtable> and <dbfield> information to verify whether a corresponding record exists in a database table. Primary keys of the table in question are retrieved using a metadata query. These primary key values are then compared against the <dbfield> information to verify whether the database operation to be done is an insert or an update. If no record exists for the primary key values, a database insert is done. Else, the

29

corresponding record is updated. The transformer updates fields indicated in the XML document alone. The database state is thus brought in tune with the input XML information.

## 4.2 XML-XML Translation With XSL

The transformation from application-specific input files such as pbs.xml into dbtxns.xml is done using XSLT. This step is application-specific, as the XSL stylesheet needs to know the structure of the document to be transformed.

**pbs.xsl** is the stylesheet defined to transform pbs.xml to dbtxns.xml. Figure 4.2 shows part of pbs.xml.

```
1   <?xml version="1.0" ?>
2   <xsl:stylesheet version="1.0"
        xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
3   <xsl:output method="xml" omit-xml-declaration="no" />
4   <xsl:output doctype-system="./dbtxns.dtd" />
5   <xsl:output indent="yes" />
6   <xsl:template match="/">
7       <dbtxns>
8           <dbtxn>
9               <xsl:apply-templates select="/job" />
10              <xsl:apply-templates select="/job//nodes" />
11              <xsl:apply-templates select="/job//variableList" />
12          </dbtxn>
13      </dbtxns>
14  </xsl:template>
15  </xsl:stylesheet>
```

Figure 4.2 Parts of pbs.xsl

The `method` attribute of `<xsl:output>` in Line 3 specifies that the output of the transformation is to be an XML document. The `doctype-system` attribute of `<xsl:output>` specifies the schema to

which the output document should conform. This is crucial in filling out the default attributes of elements.

Various <xsl:template> and <xsl:apply-templates> tags now specify the transformations to be applied. The first rule is applied to the root element / with <xsl:template match = "/">. Figure 4.3 outlines this.

```
1   <xsl:template match="/">
2       <dbtxns>
3           <dbtxn>
4                   <xsl:apply-templates select="/job" />
5                   <xsl:apply-templates select="/job//nodes" />
6                   <xsl:apply-templates select="/job//variableList" />
7           </dbtxn>
8       </dbtxns>
9   </xsl:template>
```

Figure 4.3 Transformation Rule for the Root Element

The rule specifies that on encountering the document root element / [Line 1], the elements <dbtxns> and <dbtxn> are to be printed out. <xsl:apply-templates> tag is then used to apply template rules to the <job> element, and to the <nodes> and <variableList> elements within <job>. The // in job//nodes indicates that <nodes> and <variableList> can occur at any depth within <job>, following Xpath terminology.

Figure 4.4 specifies the transformation rules to be applied on encountering <nodes> under <job>.

```
1   <xsl:template match="/job//nodes">
2       <dbtable name="JOBNODES">
3       <xsl:for-each select="./node">
4           <dbrow>
5                   <dbfield key="true" value="{/job/@id}" name="jobId" />
6                   <dbfield key="true" value="{./nodeId}" name="nodeId" />
```

```
7                <dbfield key="true" value="{./cpu}" name="cpu" />
8           </dbrow>
9       </xsl:for-each>
10      </dbtable>
11  </xsl:template>
```

Figure 4.4 Transformation Rule for the Nodes Element

The line <dbtable name= "JOBNODES"> is output first [Line 2], specifying that the database table applicable is going to be JOBNODES. <xsl:for-each> construct [Line 3] is now used to handle repeating node information. For each <node> element encountered, a <dbrow> element is output [Line 4], along with the corresponding <dbfield> elements for the jobId, nodeId, and cpu attributes of JOBNODES table [Lines 5, 6, 7]. The syntax {./nodeId} indicates that the value of <nodeId> element is to be substituted by XSLT. {/job/@id} specifies that the value of id attribute of <job> element is to be substituted. The <xsl:for-each> construct ensures that the transformation is done for each <node> element encountered.

Figure 4.5 shows a similar transformation for <variableList>. The database table, in this case, is VARIABLELIST.

```
1   <xsl:template match="/job//variableList">
2       <dbtable name="VARIABLELIST">
3        <xsl:for-each select="./variable">
4               <dbrow>
5           <dbfield key="true" value="{/job/@id}" name="jobId" />
6           <dbfield key="true" value="{./@name}" name="variableName" />
7           <dbfield value="{.}" name="variableValue" />
8         </dbrow>
9        </xsl:for-each>
10      </dbtable>
11  </xsl:template>
```

Figure 4.5 Transformation Rule for the VariableList Element

Figure 4.6 outlines the transformations to be applied on encountering the root element <job>.

```
1  <xsl:template match="/job">
2      <dbtable name="JOB">
3          <dbrow>
4              <dbfield key="true" value="{/job/@id}" name="id" />
5              <xsl:apply-templates select="./jobName" />
6              <xsl:apply-templates select="./userName" />
7              <xsl:apply-templates select =
8                  "./resourceUsed/cpuTimeUsed" />
9              <xsl:apply-templates
10                 select="./resourceRequested/cpuTimeList" />
11             <xsl:apply-templates select="./sessionId" />
12             <xsl:apply-templates select="./comment" />
13             <xsl:apply-templates select="./eligibilityTime" />
14         </dbrow>
15     </dbtable>
16 </xsl:template>
17 <xsl:template match="/job/jobName">
18     <dbfield value="{.}" name="jobName" />
19 </xsl:template>
20 <xsl:template match="/job/userName">
21     <dbfield value="{.}" name="userName" />
22 </xsl:template>
23 <xsl:template match="/job/resourceUsed/cpuTimeUsed">
24     <dbfield value="{.}" name="cpuTimeUsed" />
25 </xsl:template>
26 <xsl:template match="/job/resourceRequested/cpuTimeList">
27     <dbfield value="{.}" name="cpuTimeList" />
28 </xsl:template>
29 <xsl:template match="/job/sessionId">
30     <dbfield value="{.}" name="sessionId" />
31 </xsl:template>
32 <xsl:template match="/job/comment">
33     <dbfield value="{.}" name="jobComment" />
34 </xsl:template>
35 <xsl:template match="/job/eligibilityTime">
```

```
36      <dbfield value="{.}" name="eligibilityTime" />
37  </xsl:template>
```

Figure 4.6 Rule for the Job Element

The database table is JOB, so <dbtable name = "JOB"> is output [Line 2], with a single <dbrow> element [Line 3]. Line 4 outputs a <dbfield> element with name attribute set to the string id. The value attribute of <dbfield> is set to the id attribute of <job>, and the key attribute is set to true, indicating that the field is a primary key.

Template rules are now applied for each of the child elements of job, with <xsl:apply-templates select = "./jobName" />, <xsl:apply-templates select = "./userName" />, <xsl:apply-templates select = "./resourceUsed/cpuTimeUsed" /> etc. This approach ensures that only the elements we are interested in processing, get transformed, and appear in the output dbtxns.xml.

## 4.3 dbqueries Schema—The Query Transformer Input

The project supports query facilities against the data in the database. Database queries are submitted embedded in XML files. DbQueryTxmr extracts these queries, executes them, and outputs the query results as an XML document.

Figure 4.7 shows the schema expected by the query transformer.

```
1       <?xml version='1.0' encoding='UTF-8' ?>
2       <!ELEMENT dbqueries (dbquery+)>
3       <!ELEMENT dbquery (#PCDATA)>
4       <!ATTLIST dbquery queryname CDATA  #REQUIRED>
5       <!ATTLIST dbquery rowname   CDATA  #REQUIRED>
6       <!ATTLIST dbquery conn      CDATA  "default">
```

Figure 4.7 dbqueries.dtd

Line 1 shows <dbqueries> as the root element, containing one or more <dbquery> elements. The query to be executed, in SQL syntax, is contained as character data inside <dbquery>.

Queryname and rowname are mandatory attributes of <dbquery>. The values of these attributes are used to mark up the query results. The value of rowname is used to mark up each row of the query result. The value of queryname is used to mark up the overall query result.

conn is an optional attribute of <dbquery> with a default value of default [Line 6]. This attribute has been included with future extensions in mind–to enable the query to be submitted to a specific connection. Currently, a default connection to the Oracle database instance on enpdata is used every time.

Figure 4.8 shows an example dbqueries.xml document.

```
1   <dbqueries>
2   <dbquery queryname="jobnodesFor20692" rowname="jobnode">select * from
    JOBNODES where jobId='20692@inter5'</dbquery>
3   </dbqueries>
```

Figure 4.8 dbqueries.xml

DbQueryTxmr transforms dbqueries.xml into dbQueryResults.xml. Figure 4.9 shows the output of the transformation.

```
1   <dbQueryResults>
2       <dbQueryResult>
3           <jobnodesFor20692>
4               <jobnode>
5                   <JOBID>20692@inter5</JOBID>
6                   <NODEID>cp107</NODEID>
7                   <CPU>0</CPU>
8               </jobnode>
9               <jobnode>
10                  <JOBID>20692@inter5</JOBID>
11                  <NODEID>cp112</NODEID>
12                  <CPU>0</CPU>
13              </jobnode>
14          </jobnodesFor20692>
```

```
15      </dbQueryResult>
16 </dbQueryResults>
```

Figure 4.9 dbqueryResults.xml

The query transformer transforms `<dbqueries>` into `<dbQueryResults>`. It currently supports

only distinct individual queries inside `<dbquery>`.

# 5. The XML-Database Interface Library

The XML-Database interface library, implemented in Java, is the core of the system, performing the XML-database transformations and vice versa. The library had to be application-independent and database-independent. Java, with its strong credentials for portability, platform-independence, its support for XML, and its database-independent JDBC classes, provided an ideal choice.

An important concern was the *weak validation and processing* strategy we required. The library had to process and transform only recognized elements in recognized contexts, and preserve and pass on the remaining elements. The Apache Cocoon pipelining strategy has been used to achieve this.

The two principal transformer components of the library are:

➤ **The DB Transactions Transformer** (`DbTxnTxmr`)—which transforms the information in dbtxns.xml into database data through database insert/update transactions, and generates the transaction results in XML under `<dbTxnsResults>`, and

➤ **The Query Transformer** (`DbQueryTxmr`)—which transforms the XML queries embedded in dbqueries.xml into query results presented in XML under `<dbQueryResults>`.

Both these transformers derive from a parent transformer class—`XmlDbTxmr`.
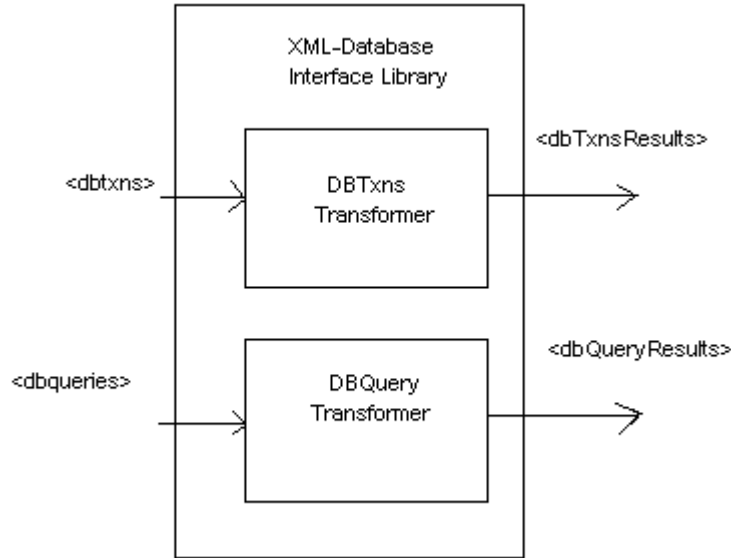
Figure 5.1 The Transformer Components of the Library

The library uses SAX to parse and process the XML documents input to the library. DOM technology has been used to generate and output the transformation/query results as XML documents.

Apache Xerces Parser, which supports the most recent versions of both SAX and DOM - SAX 2 and DOM 2, is the default XML parser used by this project.

The classes that constitute the XML-Database Interface library are explained in subsequent sections. The standard interfaces and classes used to implement the library are explained first. The processing logic is explained after introducing all component classes.

**Naming Convention Used**: Classes beginning with `Xml` (eg. `XmlDbProcessor, XmlDbTxmr` ) have been developed as part of this library. Classes beginning with `XML` (eg. `XMLReader`) are standard interfaces / classes provided as part of Apache Xerces / SAX / Cocoon.

## 5.1 Standard Interfaces And Classes Used

This section explains some of the important standard interfaces and classes used by the library.

**XMLReader:** `XMLReader` is the main interface defined in SAX2 for reading an XML document using SAX callbacks [6]. An XML parser that supports SAX2 should implement this interface. Apache Xerces Parser implements `XMLReader`.

The `setContentHandler` method of `XMLReader` allows an application to register a content event handler. This sets up the processing to respond to SAX events. The `setErrorHandler` method allows registration of an error handler, to handle SAX parsing errors encountered. This is again a mandatory step for a SAX application, as parsing errors will go unreported otherwise.

The `parse` method initiates the XML document parsing.

The `XMLReader` interface also has the `setFeature` and `setProperty` methods to set various parser features and properties.

**DefaultHandler:** `DefaultHandler` is a helper class provided as part of SAX2 to act as a convenience base class for SAX2 applications [6]. It provides default implementations for the callbacks in the four core SAX2 handler classes: `EntityResolver`, `DTDHandler`, `ContentHandler`, `ErrorHandler` [6]. All consumer classes of our library extend the `DefaultHandler` class.

**DOMStreamer:** `DOMStreamer` is a utility class provided by Cocoon that generates SAX events from a W3C DOM document [7]. The `stream` method starts the production of SAX events.

## 5.2 The XML-DB Interface Library Classes

### 5.2.1 The XmlDbProcessor Class

The `XmlDbProcessor` class is the starting point for invoking the library. It is the control program that accepts the input files to process. It processes the command-line arguments, sets up the XML parser, and invokes the transformer to process the input file.

Figure 5.2 shows how `XmlDbProcessor` does this.

```
1  import org.xml.sax.*;
2  import org.apache.cocoon.xml.*;
3  public class XmlDbProcessor implements XMLProducer {
4      private XMLReader parser = null;    // XMLReader instance.
5      protected XMLConsumer consumer = null; //The XML Consumer
6      private XmlDbTxmr dbTxmr = null;    // The transformer class
7      //Instantiating the parser
8      parser = (XMLReader) Class.forName(parserName).newInstance();
9      //Instantiating our transformer class
10     dbTxmr = new XmlDbTxmr();
11     //Sets the consumer of this class to the instantiated transformer.
12     this.setConsumer(dbTxmr); //The starting step of our pipeline
13     //Registering the content and error handlers
14     parser.setContentHandler(dbTxmr);
15     parser.setErrorHandler(dbTxmr);
16     //Invoking parse
17     parser.parse(uri);
18 }
```

Figure 5.2 XmlDbProcessor.java

XmlDbProcessor is an XML producer class [Line 3]. Its consumer is set to dbtxmr—the instance of XmlDbTxmr, the main transformer class. Line 8 instantiates the XML parser. Line 10 instantiates our transformer class—XmlDbTxmr as dbtxmr. Line 12 sets up dbtxmr as the consumer. Lines 14 and 15 configure dbtxmr to respond to the SAX events generated by XmlDbProcessor.

Line 17 invokes the parse method of the parser. This results in the processor parsing the input XML document. SAX events are thereby reported to the consumer, dbtxmr.

**5.2.2 The XmlDbTxmr Class —The Transformer**

The XmlDbTxmr class is our primary transformer class. It is a parent transformer class for child transformers that do specific transformations on recognizing specific contexts. It is a transformer. Hence it is an XMLProducer as well as an XMLConsumer [7].

XmlDbTxmr is instantiated by XmlDbProcessor, which configures it as its consumer. Figure 5.3 shows the implementation of XmlDbTxmr.

```
1   import org.xml.sax.*;
2   import org.w3c.dom.*;
3   import org.apache.cocoon.xml.*;
4   public class XmlDbTxmr extends DefaultHandler implements XMLConsumer,
            XMLProducer {
5       // The DBConnect class used to provide database connection
6       protected DBConnect db = null;
7       // The generator class. Generates the DOM tree for outputting XML
8       protected XmlDomGenerator gen = null;
9       // The SAX Consumer that consumes events generated by this class
10      protected XmlSaxConsumer consumer = null;
11      // Attribute indicating context
12      protected String context = null;
13      // Child txmr. Processes each recognized context
14      protected XmlDbTxmr txmr = null;
15  }
```

Figure 5.3 XmlDbTxmr.java

Line 4 of Figure 5.3 shows XmlDbTxmr extending the standard DefaultHandler helper class. This is required, as it is an XML consumer, and needs to respond to the SAX events generated by XmlDbProcessor. XmlDbTxmr implements the startElement, endElement, characters and comment methods of the ContentHandler interface of SAX , and the error, warning and fatalError methods of its ErrorHandler interface [6].

XmlDbTxmr also implements Cocoon's XMLConsumer interface [Line 4], as it is a consumer to the SAX events generated by XmlDbProcessor.

XmlDbTxmr implements Cocoon's XMLProducer interface as well [Line 4]. It implements the XMLProducer interface, as it needs to attend to unrecognized elements that need to be passed on. When XmlDbTxmr comes across unrecognized elements, it passes it down the pipeline—to a consumer

of its own. In this role, XmlDbTxmr acts as a SAX producer, sending SAX events down the pipeline. An XmlSaxConsumer class instance (explained subsequently) is configured to be the consumer to SAX events generated by XmlDbTxmr.

The transformer contains an XmlDomGenerator attribute [Line 8]. The transformer class constructs an XML DOM tree in memory to store the results of the XML transformations. This DOM tree is then converted into a SAX stream and fed to the XmlSaxConsumer class—in effect, hooking the generated XML DOM tree into the SAX pipeline.

The DBConnect member attribute [Line 6] handles the database connections required by the transformer. A context String [Line 12] stores the context being processed.

### 5.2.3 The XmlSaxConsumer Class—The Final Sax Consumer

The XmlSaxConsumer class is the last step of our pipeline. This class is again a SAX Consumer class, and uses the standard SAX APIs to consume SAX events received from up the pipeline. Figure 5.4 shows its implementation.

```
1   import org.xml.sax.*;
2   import org.apache.cocoon.xml.*;
3   public class XmlSaxConsumer extends DefaultHandler implements XMLConsumer {
4       private PrintWriter out;
5       XmlSaxConsumer() {
6           out = new PrintWriter(new FileWriter ( XmlDbConstants.OUTPUT_FILE
                    ));
7               out.print("<?xml version=\"1.0\" encoding=\"UTF-8\"?>\n");
8       }
9   }
```

Figure 5.4 XmlSaxConsumer.java

XmlSaxConsumer is a consumer to the SAX events generated by XmlDbTxmr, the transformer. Note that XmlDbTxmr is itself a consumer to SAX events generated by XmlDbProcessor.

The following step in `XmlDbProcessor`, instantiating `XmlSaxConsumer`, sets up the last stage of the pipeline.

```
1        //Sets consumer of dbtxmr, the instantiated XmlDbTxmr,
2        // to XmlSaxConsumer
3        dbTxmr.setConsumer(new XmlSaxConsumer());
```

`XmlSaxConsumer` has a `PrintWriter` member attribute [Line 6 of Fig. 5.4]. This is initialized to a default output file, `out.xml`.

Figure 5.5 shows the SAX pipeline we have in place now.
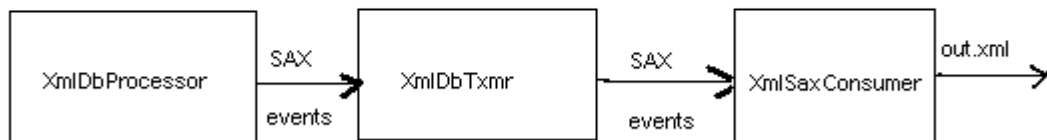


Figure 5.5 The SAX Pipeline

`XmlSaxConsumer` is an XML consumer, and needs to process the SAX events it receives. For content handling, all it needs to do is merely echo out whatever it receives, to the `PrintWriter` attribute `out`, to be output to `out.xml`.

Figure 5.6 shows how the `startElement`, `endElement`, `characters` methods are implemented in `XmlSaxConsumer`.

```
// On receiving element start SAX event
1  public void startElement(String uri, String local, String raw, Attributes
2      attrs) {
3      //Prints out the name of the element received, '>'
4      out.print("<" + local );
5      //Attributes, if any, need to be printed out as well
6      if (attrs != null) {
```

```
7           for(int i = 0; i < attrs.getLength(); i++) {
8               out.print(" " + attrs.getLocalName(i) + "=\""
9                   + attrs.getValue(i) + "\"");
10          }
11      }
12      // Closing the element tag
13      out.print(">");
14 }
// On receiving element end SAX event
15 public void endElement(String uri, String local, String raw) {
16      //Printing out the end element tag
17      out.println("</" + local + ">");
18      out.flush();
19 }
// On receiving character data inside an element
20 public void characters(char[] ch, int start, int length) {
21      for (int i = 0; i < length; i++) {
22              out.print(ch[i]);
23      }
24 }
```

Figure 5.6 ContentHandler Methods of XmlSaxConsumer

The implementations of `startElement, endElement, characters` given above ensure that all XML content received by `XmlSaxConsumer` is echoed out onto `out.xml` in well-formed XML syntax.

### 5.2.4 The XmlDomGenerator Class —the XML Generator

`XmlDomGenerator` is our XML Generator class, generating an XML document as an in-memory DOM tree. It uses standard DOM 2 APIs to do so, making use of Apache Xerces parser's DOM implementation classes [8].

Apache Xerces provides the `DOMImplementationImpl` class, implementing the standard DOM2 `DOMImplementation` interface. `DOMImplementation` provides a `createDocument` method to create

an XML document object. Xerces also provides the `DocumentImpl` class, which implements the `Document` interface.The `Document` interface represents an XML document in memory, and contains various methods to create other DOM objects—elements, text nodes, processing instructions, comments etc.

Figure 5.7 shows `XmlDomGenerator` using the above classes and interfaces to create a DOM tree in memory.

```
1   import org.w3c.dom.*;
2   import org.apache.xerces.dom.*;
3   import org.apache.xml.serialize.*;
4   public class XmlDomGenerator {
5       protected DocumentImpl xmldoc; // the XML document object
6       protected Element root; // The root element
7       public XmlDomGenerator(String docName) {
8           // The DOM implementation class provided by Apache Xerces
9           DOMImplementationImpl impl =  (DOMImplementationImpl)
                    DOMImplementationImpl.getDOMImplementation();
10          // Creating an empty DocumentType node
11          DocumentType type = impl.createDocumentType(docName,
                null, null);
12          // Creating the XML Document object
13          xmldoc = (DocumentImpl) impl.createDocument(null, docName,
14              type);
15      }
16   }
```

Figure 5.7 XmlDomGenerator.java

`XmlDomGenerator` has a `DocumentImpl` attribute, `xmldoc` [Line 5]. This represents the root of the XML document tree, and provides access to the document data – to the various nodes.  The generator uses the `createElement`, `createComment`, `createTextNode` methods provided by `DocumentImpl` to create various nodes of the DOM tree.

The `Element`[8] attribute, `root`, is the topmost element of the DOM tree we create. `XmlDomGenerator` provides a `setRoot` method to create this root element. A `getRoot` method provided, returns this root element attribute.

`XmlDomGenerator` also provides various `addElement` methods to append nodes to specific elements in the document tree. These use `DocumentImpl`'s `createElement`, `appendChild` and `Element`'s `setAttribute` methods internally to do so. The public `addText` and `addComment` methods are provided to append text elements and comments to various nodes of the DOM tree.

`XmlDomGenerator` also provides a `serialize` method, to do a serialization of the generated DOM tree. It uses the `XmlSerializer` class to achieve this.

**5.2.5 The XmlSerializer Class —The Serializer**

The `XmlSerializer` class is provided to generate the XML document as a serialized stream. In the current implementation, this class just extends Cocoon's `XMLSerializer`, providing just an additional constructor to serialize the XML document to standard output. Figure 5.8 shows the implementation of `XmlSerializer`.

```
1  import java.io.*;
2  import org.w3c.dom.*;
3  import org.apache.xerces.dom.*;
4  import org.apache.xml.serialize.*;
5  public class XmlSerializer extends XMLSerializer {
6      public XmlSerializer(java.io.OutputStream out) {
7          super(out, null);
8      }
9  }
```

Figure 5.8 XmlSerializer.java

Figure 5.9 shows how `XmlDomGenerator` uses the `XmlSerializer` class, to serialize the DOM tree.

```
1  public void serialize() {
2      try {
3          XmlSerializer ser = new XmlSerializer(System.out);
4          OutputFormat format = new OutputFormat(xmldoc);
5          ser.setOutputFormat(format);
6          ser.serialize(root);
7      } catch (IOException e) {
8          System.err.println("Error while serializing!!");
9          System.exit(-1);
10     }
11 }
```

Figure 5.9 The `serialize()` method of the DOM generator

### 5.2.6 DOM-SAX Conversion

Cocoon's `DOMStreamer` class is used to convert the generated DOM tree into a stream of SAX events.

Figure 5.10 shows how the DOM-SAX conversion is done. This step retrieves the root of the DOM tree generated, converts it into a SAX stream with the `stream()` call, and feeds it to the consumer, the `XmlSaxConsumer` instance.

```
1  DOMStreamer stream = new DOMStreamer(consumer);
2  stream.stream((Node)gen.getRoot());
```

Figure 5.10 DOM-SAX conversion

With this step, the transformation results are also hooked onto the pipeline, and so, our pipeline takes its final form, as shown in Figure 5.11.
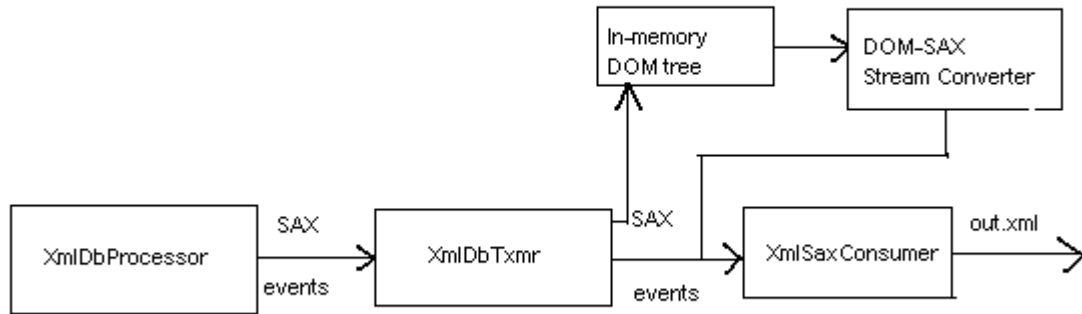
Figure 5.11 The Final Pipeline

## 5.2.7 DBConnect and XmlDbConnect

DBConnect and XmlDbConnect are the database connection classes used by the Xml-Database interface library. DBConnect is the database connection library class developed by Dr.Riccardi, to connect to any database on any server, given the database driver, URL, and the user/password information. It uses JDBC classes to achieve this.

XmlDbConnect provides the specific connection details to connect to the info Oracle 8i database instance on enpdata.csit.fsu.edu using Oracle JDBC driver. Figure 5.12 shows how the connection is specified.

```
1   public class XmlDbConnect extends DBConnect {
2       public XmlDbConnect() {
3           super(DBINSTANCE, DBUSER, DBPWD);
4           this.setDriver("oracle.jdbc.driver.OracleDriver");
5           this.setUrl("jdbc:oracle:thin:@enpdata.csit.fsu.edu:1521:");
6       }
7   }
```

Figure 5.12 XmlDbConnect.java

`XmlDbConnect` represents the only database-specific component of the library. To connect to a JDBC-compliant database of choice, all that needs to be changed is the driver and url information in `XmlDbConnect`.

The child transformer classes instantiate `XmlDbConnect` on encountering `<dbtxns>` and `<dbqueries>`. The connection details are stored in the `DBConnect` and `connection` member attributes of the parent transformer, and reused for each database transaction.

### 5.2.8 The XmlDbException Class

`XmlDbException` is the exception class defined, extending Java's standard `Exception` class. This class is used to enable the library to throw its own exceptions when required.

### 5.2.9 The XmlDbConstants Class

`XmlDbConstants` class defines the constants used by the library. The default parser name, and the default input and output file names are defined in this class.

## 5.3 The Child Transformer Classes

`DbTxnTxmr` and `DbQueryTxmr` are our two child transformer classes, designed to handle the `<dbtxns>` and `<dbqueries>` contexts respectively. `DbTxnTxmr` handles the `<dbtxns>` context, transforming the XML information within, into database insert and update transactions. `DbQueryTxmr` handles the `<dbqueries>` context, extracting the queries specified within, executing it, and generating the query results as `<dbQueryResults>`. Both transformers preserve the context, and ignore unrecognized elements within the contexts being transformed.

### 5.3.1 The DBTxnTxmr Class —The Transactions Transformer

`DbTxnTxmr` is the transactions transformer, transforming the XML information within `<dbtxns>` into database insert/update transactions. This class also generates the transaction results as an XML DOM tree in memory. The DOM tree is then serialized into a SAX stream.

`DBTxnTxmr` extends `XmlDbTxmr`, the main transformer class.

```
1   import org.w3c.sax.*;
2   import org.w3c.dom.*;
3   import org.apache.xml.serialize.*;
4   import org.apache.cocoon.xml.*;
5   import org.apache.cocoon.xml.dom.*;
6   public class DbTxnTxmr extends XmlDbTxmr {
7       private XmlDbTxn dbtxn = null;        //Corresponds to dbtxn element
8       private int action;  //Attribute indicate database insert/update
9       private XmlDbTable table = null;      // Corresponds to dbtable
10      private String tablename = null;
11      private Vector primaryKeys = null; //Vector storing primary keys
12      private XmlDbRow row = null; //Corresponds to dbrow element
13      //Ctor
14      public DbTxnTxmr(XmlDomGenerator gen, XmlSaxConsumer cons) {
15          super(gen, cons); // Sets the corr.attributes in the parent.
16      }
17  }
```

Figure 5.13 DbTxnTxmr.java

`XmlDbTxmr` instantiates `DbTxnsTxmr`, on encountering <dbtxns>.

```
1           txmr = (XmlDbTxmr) new DbTxnTxmr(gen,consumer);
2           context = new String(XmlDbConstants.DBTXNS_ELEMENT);
3           txmr.processStartElement(uri, local, raw, attrs);
```

Figure 5.14 Instantiating DbTxnsTxmr

To process and transform the <dbtxns> content into database transactions, DbTxnTxmr uses the classes–XmlDbTxn, XmlDbTable, XmlDbRow, XmlDbField and XmlField. The processStartElement method of DbTxnsTxmr handles the start of <dbtxns>, <dbtxn>, <dbtable>, <dbrow> and <dbfield> elements. A processEndElement method similarly handles the end tags of these elements. Any other elements encountered are ignored.

On encountering <dbtxns>, the transformer instantiates an `XmlDomGenerator` class to store the in-memory DOM tree. The root of the tree is set to <dbTxnsResults>. A database connection is established, instantiating `XmlDbConnect`. The connection is stored in the parent transformer class, and reused until the end of <dbtxns> is encountered. Figure 5.15 shows how this is done.

```
1      gen = new XmlDomGenerator("DbTxnResults");
2      gen.setRoot("dbTxnsResults");
3      //Initializing the default database connection
4      DBConnect db = (DBConnect) new XmlDbConnect();
5      this.setDB(db);
6      conn = db.getConnection();
```

Figure 5.15 Processing of <dbtxns>

On encountering <dbtxn>, an `XmlDbTxn` class is instantiated and stored in the `dbtxn` member attribute. The `processAttribs` method of dbtxn is invoked to determine if the action is an insert or an update. The connection is made non-auto-committing, indicating the start of a database transaction.

On encountering <dbtable>, an `XmlDbTable` class is instantiated and stored in the `table` member attribute. The `getPrimaryKeyofTable()` method of `XmlDbTable` is invoked to retrieve the primary keys of the table. This is done by connecting to the database and executing a metadata query to retrieve the primary keys. The table field names returned are stored in the `primaryKeys` vector.

On encountering <dbrow>, an `XmlDbRow` class is instantiated. This will create a hashtable within to store all the <dbfield> elements encountered.

On encountering each <dbfield> tag, an `XmlDbField` class is instantiated. This validates the <dbfield> attributes and stores the field name, value and whether it is a primary key or not, in an `XmlField` class instance. This `XmlField` is then added to the hashtable contained in `XmlDbRow`.

On encountering the end of <dbrow>, all <dbfield> elements encountered would be present in the hashtable within. This is cross-verified to check if all the primary key fields of the table are indeed present, and an exception raised if not. The insertOrUpdate() method of XmlDbRow is invoked to determine if the database action is to be an insert or an update. XmlDbRow does this by connecting to the database and checking if a record exists in the database table, corresponding to the primary key column values encountered. The appropriate insertRow() / updateRow() method of XmlDbTable is then called to perform the actual database insert, or update.

On encountering the end of <dbtxn>, the transaction is committed to the database. The database connection is closed on encountering the end of <dbtxns>. The DOMStreamer's stream method is invoked to serialize the <dbTxnsResults> DOM tree into a SAX stream to be fed to the SAX consumer class of the transformer.

A DbTxnConstants class contains the constants used by the various DbTxnTransformer classes.

An XmlDbElement class is defined as a generic parent class for the XmlDbTxn, XmlDbTable, XmlDbRow, XmlDbField classes.

```
1   public class XmlDbElement
2   {
3       // The DBConnection object
4       protected DBConnect db;
5       // String containing localname of the element encountered
6       protected String local;
7       // Attributes of the element
8       protected Attributes attrs;
9       // Statement object for database queries
10      protected Statement stmt;
11      //Constructor.
12      public XmlDbElement(DBConnect db, String local, Attributes attr)
13      throws XmlDbException {
```

```
14          this.db = db;
15          this.local = local;
16          this.attrs = attr;
17          this.validate();//Validate the element.
18          //Implemented in child classes
19      }
20 }
```

Figure 5.16 XmlDbElement.java

**5.3.2 The DBQueryTxmr Class —The Query Transformer**

DBQueryTxmr is our other child transformer class, responding to <dbqueries> context, and transforming the queries encountered therein.

```
1   public class DbQueryTxmr extends XmlDbTxmr
2   {
3       // Attribute storing query name
4       private String queryName = null;
5       // Row name used in the generated output
6       private String rowName = null;
7       //Ctor
8       public DbQueryTxmr( XmlDomGenerator gen, XmlSaxConsumer cons){
9       super(gen, cons);
10      }
11 }
```

Figure 5.17 DbQueryTxmr.java

DbQueryTxmr is instantiated by XmlDbTxmr. XmlDbTxmr executes the following code, on encountering <dbqueries> element start.

```
1       txmr = (XmlDbTxmr) new DbQueryTxmr(gen,consumer);
2       context = new String(XmlDbConstants.DBQUERIES_ELEMENT);
3       txmr.processStartElement(uri, local, raw, attrs);
```

Figure 5.18 Invoking the Query Transformer

A `processStartElement` method handles the start of <dbqueries> and <dbquery> elements. A `processEndElement` method handles the end tags of these elements. Any other elements encountered are ignored.

On encountering <dbqueries>, an `XmlDomGenerator` class is instantiated to store the in-memory DOM tree. The root of the tree is set to `dbQueryResults`. A database connection is established, by instantiating `XmlDbConnect`.

Figure 5.19 shows how this is done.

```
1       gen = new XmlDomGenerator("DbQueryResults");
2       gen.setRoot("dbQueryResults");
3       //Initializing the default database connection
4       DBConnect db = (DBConnect) new XmlDbConnect();
5       this.setDB(db);
6       conn = db.getConnection();
```

Figure 5.19 Invoking the Query Transformer

The connection is stored in the parent transformer class, and reused for each query encountered within <dbquery>.

On encountering <dbquery>, its attributes are validated.

The actual query to be executed is contained as text data (PCDATA) within the `dbquery` element. This is processed by `processCharacters` method. A private `processQuery` method, invoked by `processCharacters`, executes the SQL query, and appends the results to the `dbQueryResults` DOM tree, using the specified `queryName`, `rowName`, and the query column names and values.

Figure 5.20 shows a sample dbqueries.xml.

```
1   <?xml version="1.0" encoding="UTF-8" ?>
2   <dbqueries>
```

```
3       <dbquery queryname="job_query" rowname="job">
4           select * from JOB
5       </dbquery>
6       <dbquery queryname="jobnodesFor20692" rowname="jobnode">
7           select * from JOBNODES where jobId='20692@inter5'
8       </dbquery>
9   </dbqueries>
```

Figure 5.20 dbqueries.xml

Figure 5.21 shows the output of the query transformation.

```
1   <dbQueryResults>
2       <dbQueryResult>
3           <job_query>
4               <job>
5                   <ID>20692@inter5</ID>
6                   <JOBNAME>b6p45</JOBNAME>
7                   <USERNAME>heller</USERNAME>
8                   <CPUTIMEUSED>02:27:15</CPUTIMEUSED>
9                   <MEMORYUSED>20524kb</MEMORYUSED>
10                  <SESSIONID>17596</SESSIONID>
11              </job>
12          </job_query>
13      </dbQueryResult>
14      <dbQueryResult>
15          <jobnodesFor20692>
16              <jobnode>
17                  <JOBID>20692@inter5</JOBID>
18                  <NODEID>cp107</NODEID>
19                  <CPU>0</CPU>
20              </jobnode>
21              <jobnode>
22                  <JOBID>20692@inter5</JOBID>
23                  <NODEID>cp112</NODEID>
24                  <CPU>0</CPU>
25              </jobnode>
```

```
26          </jobnodesFor20692>
27      </dbQueryResult>
28 </dbQueryResults>
```

Figure 5.21 dbqueryResults.xml

On encountering </dbqueries>, the database connection is closed. The dbQueryResults DOM tree is serialized into a SAX stream to be fed to the SAX consumer class of the transformer.

A DbQueryConstants class contains the constants used by the various DbQueryTransformer classes.

## 5.4 Transformations Preserving The Context

The XML-Database interface library needs to transform only recognized elements in recognized contexts. The remaining elements need to be passed on, untouched. In other words, the transformations should be done, preserving the context of the input information. How the transformer does this is explained below.

XmlDbTxmr, the primary transformer, maintains a context attribute—a String—that represents the current contribute being processed. XmlDbTxmr recognizes and processes the contexts—dbtxns and dbqueries. XmlDbTxmr merely echoes on all XML content it receives, until it encounters the start element of either of these recognized contexts.

Figure 5.22 shows how XmlDbTxmr handles the startElement SAX event when context is not set.

```
1  public void startElement(String uri, String local, String raw, Attributes
   attrs) {
2      //If context is null.
3      //Unrecognized tag in unrecognized context. Hence  just echoed out
4      // to the consumer , XmlSaxConsumer
5      consumer.startElement(uri, local, raw, attrs);
6  }
```

Figure 5.22 startElement of XmlDbTxmr

Similar calls—consumer.endElement(uri, local, raw), consumer.characters(ch, start, length)—are made within the endElement and characters methods when context is null. XmlSaxConsumer, the consumer, echoes out the content received in XML notation onto the output file.

This ensures that XML content in unrecognized contexts are passed down the pipeline, untouched by the transformer.

When XmlDbTxmr encounters a recognized context, the context string is appropriately set, and the corresponding child transformer initialized.

Figure 5.23 shows how this is done.

```
1   if (local.equals(XmlDbConstants.DBTXNS_ELEMENT) ) {
2         //Instantiating the appr.transformer and setting the context
3         txmr = (XmlDbTxmr) new DbTxnTxmr(gen,consumer);
4         context = new String(XmlDbConstants.DBTXNS_ELEMENT);
5         txmr.processStartElement(uri, local, raw, attrs);
6   } else if (local.equals(XmlDbConstants.DBQUERIES_ELEMENT)) {
7       //Instantiating the query transformer and setting the context
8       txmr = (XmlDbTxmr) new DbQueryTxmr(gen,consumer);
9       context = new String(XmlDbConstants.DBQUERIES_ELEMENT);
10      txmr.processStartElement(uri,local, raw, attrs);
11  }
```

Figure 5.23 Setting The Context String

Both DbTxnTxmr and DbQueryTxmr are child transformer classes extending XmlDbTxmr. The child processor takes over the processing of elements, once the context has been set.

Figure 5.24 shows how the child transformers are invoked.

```
1       //One of the recognized contexts that we process
2       if (context.equals(XmlDbConstants.DBTXNS_ELEMENT) ||
3           (context.equals(XmlDbConstants.DBQUERIES_ELEMENT))) {
```

```
4               //The respective child txmr will do the processing
5                   txmr.processStartElement(uri,local, raw, attrs);
6       }
```

Figure 5.24 Invoking The Child Transformer

All code snippets explained above have been from the startElement method of XmlDbTxmr.

Similar processing takes place in the other ContentHandler methods as well.

Figure 5.25 shows how characters is implemented.

```
1  public void characters(char[] ch, int start, int length) {
2      if (context == null) {
3          //Echoing the characters down the pipeline to XmlSaxConsumer
4          consumer.characters(ch, start, length);
5      } else {
6          //The child transformer does the processing
7          txmr.processCharacters(ch, start, length);
8      }
9  }
```

Figure 5.25 On encountering character data

Figure 5.26 shows how the endElement method is implemented.

```
1  public void endElement(String uri, String local, String raw) {
2      //Recognized context
3      if (context.equals(XmlDbConstants.DBTXNS_ELEMENT) ||
4          (context.equals(XmlDbConstants.DBQUERIES_ELEMENT))) {
5          //Invoke endElement processor of child txmr
6          txmr.processEndElement(uri, local, raw);
7          //Reached the end tag of the current context.
8          //Destroy child txmr.
9          // Re-initialize context to null
10         if (local.equals(context)) {
11             if (txmr != null) {txmr = null;}
12                 context = null;
13             }
14         }else {
```

```
15                //Unrecognized element. Just pass it through
16                consumer.endElement(uri, local, raw);
17           }
18      }
```

<center>Figure 5.26 On encountering element ends</center>

Figure 5.26 shows how the child transformer is destroyed, and the context re-initialized, on encountering </dbtxns> or </dbqueries>.

## 5.5 The Final Result

A consolidated input file, containing both <dbtxns> and <dbqueries> contexts, and the resulting output file after all the transformations, is reproduced below.

Figure 5.27 shows the input file.

**The Input File**

```
1       <?xml version="1.0" encoding="UTF-8" ?>
2       <foo>
3            <dbtxns>
4                 <dbtxn>
5                      <dbtable name="JOB">
6                           <dbrow>
7                                <dbfield name="id"
8                                     value="20692@inter5" key="true" />
9                                <dbfield name="jobName" value="b6p45" />
10                               <dbfield name="userName" value="heller" />
11                               <dbfield name="status" value="R" />
12                          </dbrow>
13                     </dbtable>
14                     <dbtable name="JOBNODES">
15                          <dbrow>
16                               <dbfield name="jobId" value="20692@inter5"
17                                    key="true" />
18                               <dbfield name="nodeId" value="cp201"
```

```
19                              key="true" />
20                          <dbfield name="cpu" value="0" key="true"
21                              />
22                      </dbrow>
23                  </dbtable>
24              </dbtxn>
25          </dbtxns>
26          <dbqueries>
27              <dbquery queryname="job_query" rowname="job">
28                  select * from JOB
29              </dbquery>
30          </dbqueries>
31      </foo>
```

Fig 5.27 The Consolidated Input File

Figure 5.28 shows the transformed output.

**The Transformed Output File**

```
1       <?xml version="1.0" encoding="UTF-8" ?>
2       <foo>
3         <dbTxnsResults>
4               <updateResult>
5                   <Success>1 row(s) updated in JOB</Success>
6               </updateResult>
7               <updateResult>
8                   <Success>1 row(s) updated in JOBNODES</Success>
9               </updateResult>
10        </dbTxnsResults>
11        <dbQueryResults>
12              <dbQueryResult>
13                  <job_query>
14                      <job>
15                          <ID>20692@inter5</ID>
16                          <JOBNAME>b6p45</JOBNAME>
17                          <USERNAME>heller</USERNAME>
```

```
18                              <CPUTIMEUSED>02:27:15</CPUTIMEUSED>
19                              <MEMORYUSED>20524kb</MEMORYUSED>
20                              <WALLTIMEUSED>04:27:11</WALLTIMEUSED>
21                              <STATUS>R</STATUS>
22                              <QUEUE>medium</QUEUE>
23                              <CHECKPOINT>u</CHECKPOINT>
24                      </job>
25                  </job_query>
26              </dbQueryResult>
27          </dbQueryResults>
28      </foo>
```

Fig 5.28 The Transformed Output

The <dbtxns> context is transformed into <dbTxnsResults> , and <dbqueries> into <dbQueryResults>. The transformations are done, preserving the context, foo, which remains untransformed.

# 6. Conclusions

## 6.1 Achievements

The objective of this project—to develop an XML-Database interface system that transforms information from a hierarchical XML representation into persistent, relational database data and vice versa—has been achieved.

The XML-Database interface library performs the transformations from XML files into the database. It performs database insert/update operations to bring the database state in sync with the input XML information. The library also provides a method to query the database, reporting the query results in XML.

The library has been designed and implemented to be database-independent. The XSL transformation from the application-specific tags of the input XML files into dbtxns.xml is the only application-specific step. Once the transformation into the dbtxns schema is done, the remaining transformations and database insertions/updations are completely independent of the database model. The use of JDBC classes ensures database-independence in performing the database interactions.

The second concern of dealing with unexpected and unrecognized input has also been achieved. The library processes and transforms recognized elements alone, preserving the context of the transformed elements. Unrecognized elements are passed on unchanged.

The library has been built with state-of-the-art XML processing technology. The most recent versions of SAX and DOM technologies have been used to process, and to generate XML. Apache Cocoon 2's pipelining strategy has been used to achieve the library's filtering and transforming capabilities.

## 6.2 Future Extensions

The library has been developed and implemented as a stand-alone back-end package. Deploying it in a servlet-like environment and providing user interaction is something that needs to be explored.

The XSL transformation from the application-specific files into dbtxns.xml is currently being performed as a separate step, and is not linked with the library. Extending the library to include this step needs to be looked into.

The query transformer, in its current implementation, expects ANSI SQL queries embedded in the input XML document. The queries are distinct and are processed and transformed individually. Supporting nested queries, and using the results of one query in another, (as scripting languages do), is a possible extension to make the query transformer much more powerful and feature-rich.

The query processing part could also be re-designed to expect and process database object names rather than queries in SQL syntax. The query processor could, for example, encounter `job` as an element or text, look for a corresponding `JOB` table, and retrieve information, rather than execute an explicit `select * from JOB` as is currently done.

# 7. Appendix

## 7.1 Abbreviations

XML : Extensible Markup Language

DTD : Document Type Definition

SAX : Simple API for XML

DOM : Document Object Model

XSL : Extensible StyleSheet Language

W3C : The World-Wide Web Consortium

# 7.2 User Manual

**Required Software:**

The following software needs to be installed for the XML-DB Interface library.

- ➢ Sun Microsystems Java Development Kit - JDK 1.2.2
- ➢ Oracle 8i (Or any JDBC-compliant relational database)
- ➢ Apache Xerces Parser (Xerces 1.4.3 is the version used in this project)
- ➢ Apache Cocoon 2

CLASSPATH needs to have the following jar files.

- ➢ Xerces.jar
- ➢ Avalon-excalibur-4.0.jar
- ➢ Avalon-framework-4.0.jar
- ➢ Logkit-1.0b5.jar
- ➢ Xalan.jar

The last four jar files are required by Cocoon classes.

The following Cocoon classes need to be present in the CLASSPATH.

- ➢ org.apache.cocoon.Constants.java
- ➢ org.apache.cocoon.caching.*
- ➢ org.apache.cocoon.serialization.*
- ➢ org.apache.cocoon.sitemap.*
- ➢ org.apache.cocoon.util.*
- ➢ org.apache.cocoon.xml.*

JDBC classes need to be present in CLASSPATH (/oracle/home/classes and /oracle/product/jdbc/lib/classes12.zip for the Oracle database on enpdata).

Compiling and Invoking The Library:

1. The source java files are organized within the following packages: enpcs.xmldb, enpcs.xmldb.util, enpcs.xmldb.dbtxn, enpcs.xmldb.dbquery

2. Use "javac *.java" to compile the various .java source files in these packages.

3. Use "java enpcs.xmldb.XmlDbProcessor  username –i input_file.xml" to invoke the library.

4. The output will be generated by default as out.xml in the current directory.

# References

1.  The Official W3C XML Site—*http://www.w3.org/XML*

2.  JP Morgenthal with Bill La Forge, `Enterprise Application Integration with XML and Java`

3.  Cay Horstmann and Gary Cornell, `Core Java Volumes 1 and 2`—from The Sun Microsystems Java Series

4.  Elliotte Rusty Harold , `The XML Bible`: esp. The chapter on XSL transformations

5.  Don Box, Aaron Skonnard, John Lam, `Essential XML : Beyond Markup`

6.  The Official SAX 2.0 Site —*http://www.saxproject.org/*

7.  The Apache Cocoon Project Site—*http://xml.apache.org/cocoon/apidocs*

8.  *DOM Level 2 Core Specification —http://www.w3.org/TR/2000/REC-DOM-Level-2-Core-20001113*

9.  The XML Industry Portal—*http://www.xml.org/xml*

10. XML Database Products—*http://www.rpbourret.com/xml/XMLDatabaseProds.htm*

11. *The Portable Batch System—http://www.openpbs.org/about.html*