XML Data Streaming for Enterprise Applications Using PL/SQL and SAX

April 3, 2002
By Jinyu Wang and Mark Scardina

Dealing with XML in enterprise applications, the challenges are how to efficiently handle the huge amount of XML documents too large to be processed in memory and how to allow scalable access and processing of these documents. In this paper, we will analyze the fundamental strategies and discuss an efficient XML data-streaming solution that makes use of the SAX, PL/SQL interfaces and Oracle8i/9i Database.

XML, Database and Enterprise Applications

Database management systems (DBMSs) like Oracle9i/8i have dominated enterprise data management for years, as they offer facilities that are critical for enterprise data management. These include the highly tuned query engine that supports transaction processing of large amounts of data, data backup and recovery, security, multi-user access, triggers and so on.

Now, as XML becomes the standard portable data format to share data and metadata between enterprise applications, the XML data processing metaphor is introduced and receiving attention.

Native XML support for managing the XML data, such as schemas (DTDs, XSDs), query languages (XQuery, XPath, etc.) and programming interfaces (SAX, DOM, JDOM, etc.) is extensive. However, for building an enterprise application that manipulates massive data and needs to utilize extensive parallel processing, network distributed resources, and complex logic, we still need to store and manage the XML data with the DBMS systems.

Native XML Processing in Database

Now, the question is “What is the best way to process the XML data and apply the application logic?” Well, the answer depends on the way you implement this application logic.

Since PL/SQL runs natively inside the database and tends to run faster than other languages in the database server, many database applications use PL/SQL stored procedures to implement the business logic.

Figure-1 shows the normal approach for developing native XML processing inside an Oracle database using PL/SQL stored procedures. In this implementation, we read the XML document from database table and parse it to build a DOM tree object. Then, we run the application logic process by retrieving the data from the DOM tree using the DOM APIs provided by the PL/SQL XML Parser (included in the Oracle XDK for PL/SQL).

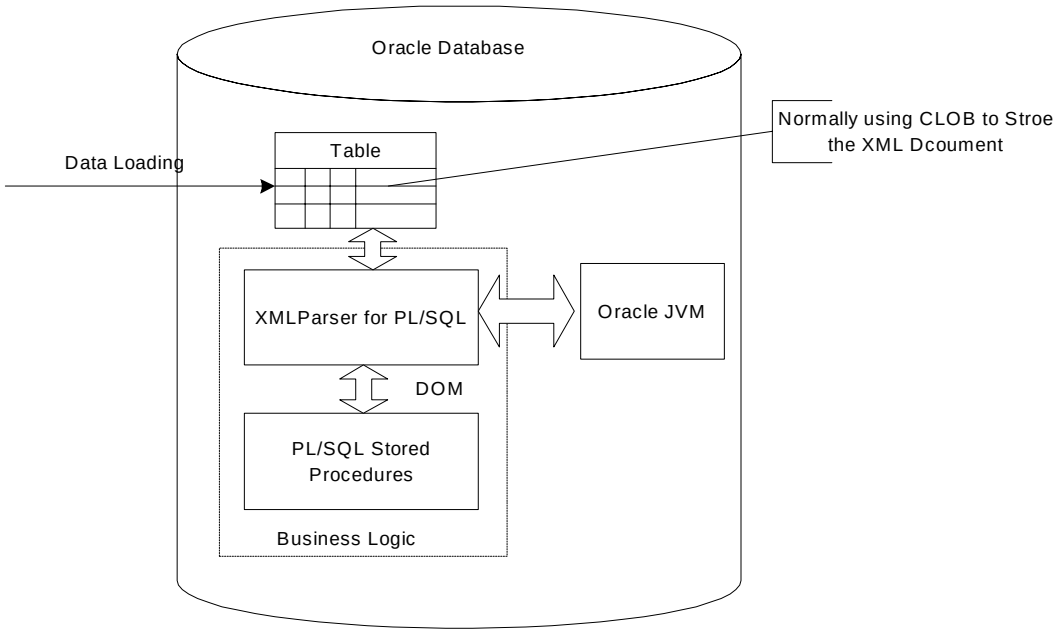

Figure 1: Native XML Processing using XDK for PL/SQL DOM APIs

As shown in Figure 1, this approach has the following overhead negatively impacting the performance of the application:

- Overhead for the Calls between PL/SQL and Java

  As the current XML Parser for PL/SQL provided by Oracle XDK is built based on Java stored procedures, every procedure call to the DOM APIs, involves a round trip between PL/SQL Engine and Oracle JVM with the resulting overhead. Typical applications need to retrieve all of the data from a document thereby making dozens if not hundreds of DOM calls. This will negatively impact the performance of the application.

- DOM processing overhead

  As we are dealing with large XML documents, maintaining a DOM object in the database session also consumes too much memory for the task at hand. The DOM APIs were designed to be used when the application requires the entire document to be in memory at the same time. Large document processing using the DOM will lead to the “out of memory” exceptions.

Therefore, to properly implement enterprise applications needing to process large XML documents with complex application logic, we should reduce the calls between PL/SQL and Java and avoid building the DOM tree.

From a basic XML processing point of view, we should avoid build DOM unless we need to do XSLT transformation or DOM-based editing of the XML document. Otherwise, we should think about using the SAX interface, which is an event-driven data-streaming interface.

Figure 2 shows an alternative approach. We build the Java stored procedures using the SAX interface to stream the XML data into database tables, which is then indexed by its corresponding XPATH string.

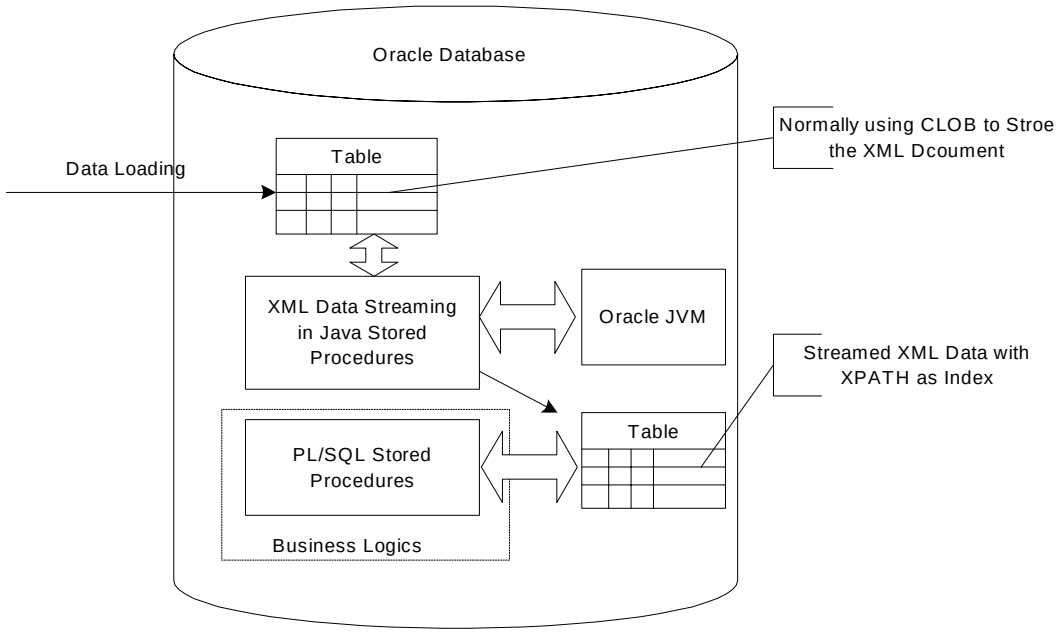

Figure 2: Native XML Processing using SAX-based XML Streaming

For example, if we have the XML document shown in Listing 1.

```
<?xml version="1.0" encoding="UTF-8"?>
<transaction type="PPSR_REG_PRIOR_FS" client_ref="B2B TUNING">
  <finance_statement seq="1" expiry_date="12-10-2004"
prior_reg_date="12-10-1999" prior_reg_legislation="1989">
    <secured_party_details>
      <new_sp_group>
        <new_secured_party type="O" seq="1">
          <email_address>John.King@companies.govt.cn</email_address>
          <fax_country>+86</fax_country>
          <fax_area>09</fax_area>
          <fax_number>9134225</fax_number>
          <contact_phone>(09)913-4245</contact_phone>
          <address_details>
            <contact_address>
              <line1>162-186 Grafton Road</line1>
              <suburb>Grafton</suburb>
              <city_town>Beijing</city_town>
              <postcode>1001</postcode>
              <country_code>CN</country_code>
            </contact_address>
            <mailing_address>
              <line1>162-186 Grafton Road</line1>
              <suburb>Grafton</suburb>
              <city_town>Bejing</city_town>
              <postcode>1001</postcode>
              <country_code>CN</country_code>
            </mailing_address>
          </address_details>
        </new_secured_party>
      </new_sp_group>
    </secured_party_details>
  </finance_statement>
</transaction>
```

Listing 1: Sample XML Document

After the SAX-based XML data streaming of the document, we populate a database table with the XML data and its XPATH, as shown in Table 1.

XPATH Index	**Data Content**
/transaction/@type	PPSR_REG_PRIOR_FS
/transaction/@client_ref	B2B TUNING
/transaction/finance_statement/@seq	1
/transaction/finance_statement/@expiry_date	12-10-2004
/transaction/finance_statement/@prior_reg_date	12-10-1999
/transaction/finance_statement/@prior_reg_legislation	1989
/transaction/finance_statement/secured_party_details/new_sp_group/new_secured_party/@type	O
/transaction/finance_statement/secured_party_details/new_sp_group/new_secured_party/@seq	1
/transaction/finance_statement/secured_party_details/new_sp_group/new_secured_party/email_address/	John.King@companies.govt.cn
…	…

Table 1: Table for Streamed XML Data

After the SAX-based XML data streaming, the application processing will use this streamed data by retrieving it from the table using either SQL or PL/SQL. In this way, we both avoid building the DOM tree and reduce the round trip overhead of the calls from PL/SQL to Java.

The provided sample implementation associates a database session id with the streamed data which allows multiple users from different database sessions to access and process the streamed data while sharing the same table.

In summary, the benefits of this strategy are:

- SAX interfaces allows efficient data streaming for large XML documents
- XPATH-based XML Mapping provides XML data indexing
- SQL data table access makes use of the efficient data management capability of the Oracle database in implementing the application.

XML Data Streaming Using PL/SQL and SAX

In this section, I'll describe the details of how to write stored procedures in Java for streaming the XML Document using SAX APIs and deploy it into the Oracle9*i*/8*i* database.

Before trying to develop the program, you need to setup the environment by downloading the latest version of the Oracle XML Developers' Kit for Java at the following web site:

http://otn.oracle.com/tech/xml/xdk_java.

The component you'll need is XML Parser for Java (xmlparserv2.jar). The Oracle Database (8.1.6 version and above) is also required, as we need to create Java Stored Procedures to setup the streaming process on the database server.

After you download the XDK for Java and have an Oracle 8.1.6 Database or above installed, you can follow these steps to build the Java Stored Procedures to implement XML streaming into database tables:

- Developing the Java Class using SAX-based XML Data Streaming
- Loading and resolving the Java Classes into the Oracle Database
- Publishing to PLSQL the Java Classes by defining the specifications

Developing the Java Classes for SAX-based Data Streaming

First, we must implement the Java programs to stream the XML document using SAX APIs and insert the data into the database through JDBC. We shall implement these two Java classes:

- ExDocumentHandler class (ExDocumentHandler.java): streams the XML Document by processing SAX events and inserting the streamed data into a database table.
- ExElement class(ExElement.java): its instance object is used to push into the XPATH stack sufficient information for XPATH generation.

ExElement object has the following three member variables:

```
public int nodePos; //current element's position under its parent node
public int childNum; //the number of child elements
public String nodeName; //the element's name
```

In order to understand the XPATH generation process, let us examine the process through an example.

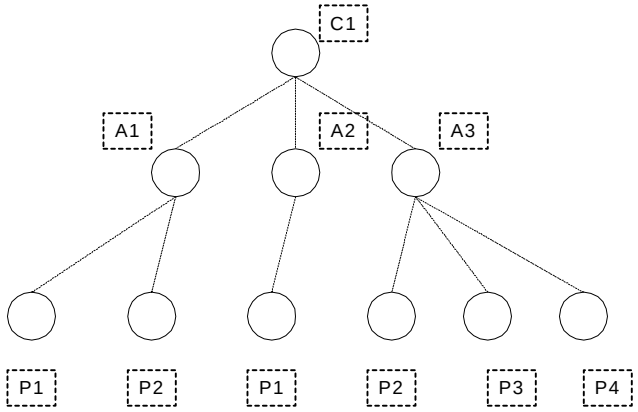

Figure 3: Sample XML Document

Figure 3 shows a graphical example XML document. To generate the XPATH for each XML Element with its corresponding data, we will process the following SAX events when parsing it:

- `StartElement`
- `endElement`
- `characters`
- `endDocument`

During this processing, we will maintain two repositories. One is a STRING buffer where we will keep the content of the current XML element. The other one is a STACK where we keep track of the current element's XPATH information.

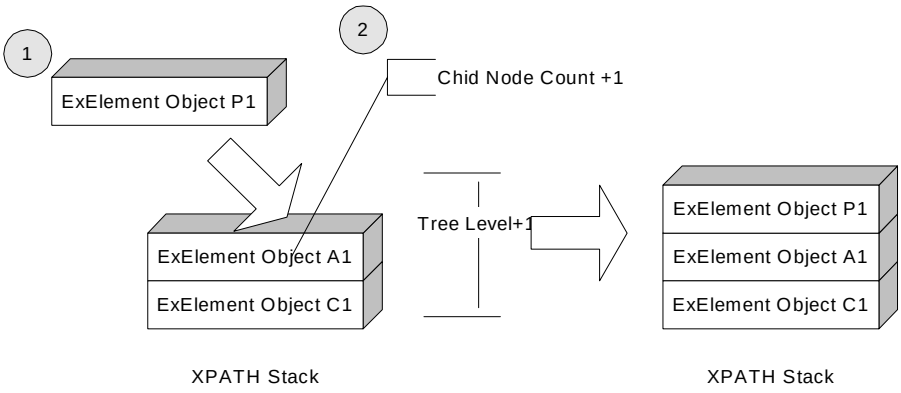

Figure 4: XPATH Stack Management When Receiving a startElement() Event

In the StartElement event process, as shown in Figure 4, we will:

- Push a new ExElement object onto the XPATH Stack
- Increase the child element count in the parent ExElement Object by 1

In addition, we will:

- Insert previous XPATH and Element content into database
- Setup the mapping of all of the current element's attributes

The following list shows the source code of this process. Note that, since the program will be load and run in the database server, we use Oracle's JDBC KBPR driver in the sample program. This JDBC Driver uses the default/current database session and thus requires no additional database username, password or URL.

```
public void startElement(NSName name, SAXAttrList attrlist)
  throws SAXException
  {
    ExElement parent = null;

    // Get the XPATH and update the XPATH stack
    int length = m_xpathNum.size();
    if(length != 0)
      parent = (ExElement)m_xpathNum.get(length-1);
    if(parent != null)
    {
      if(isName)
      {
        m_xpathNum.add(new
ExElement(parent.childNum+1,0,name.getExpandedName()));
      }
      else
      {
        m_xpathNum.add(new ExElement(parent.childNum+1,0));
      }
    }
    else
    {
```

```
      if(isName)
      {
        m_xpathNum.add(new ExElement(1,0,name.getExpandedName()));
      }
      else
      {
        m_xpathNum.add(new ExElement(1,0));
      }
    }

    // Add the xpath attribute to the current element
    String elem_path = getCurrentXpath();

    // Print out the result
    try
    {
      // Close the parent Node
      if(length > 0)
      {
        if(buf.length()>0)
        {
          // Insert the Elements into the Database (now flush ahead)
          insertIndexData(parentXpath, buf.toString());

          write(parentXpath);
          write(":");
          flushChar();
          write("\n");
        }
     }

      parentName = name.getExpandedName();
      parentXpath = getCurrentXpath();

      int len = attrlist.getLength();
      for(int i=0; i< len; i++)
      {
        write(elem_path+"@"+attrlist.getExpandedName(i));
        write(":");
        String value = attrlist.getValue(i);
        write(value);
        write("\n");

        // Insert Attribute Element into the Database Table
       insertIndexData(elem_path+"@"+attrlist.getExpandedName(i),
value);

      }
    }
    catch(IOException ex)
    {
      throw new SAXException(ex);
    }
 }
```

Listing 2: startElement Event Processing

Figure 5 shows when receiving the endElement SAX event, we will pop the top element off of the XPATH stack.

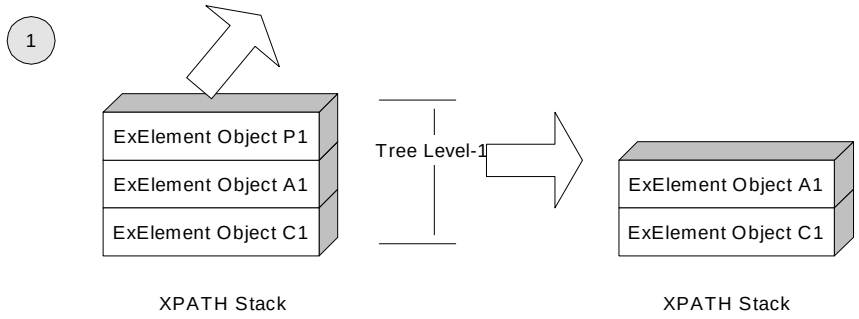

Figure 5: XPATH Stack Management When Receiving endElement() Event

The source code is show in Listing 3.

```
  public void endElement(NSName name)  throws SAXException
  {

    int length = m_xpathNum.size();
    if(length > 1)
    {
      ExElement parent = (ExElement)m_xpathNum.get(length-2);
      parent.childNum = parent.childNum+1;
      m_xpathNum.setSize(length-1);
    }

}
```

Listing 3: endElement Processing

Listing 4 shows that in processing the endDocument() events, we close the XPATH Stack and finish processing the data streaming.

```
public void endDocument() throws SAXException
  {
    int length = m_xpathNum.size();
    try
    {
      // Close the parent Node
      if(length > 0)
      {
        // Insert the Elements into the Database (now flush ahead)
        insertIndexData(parentXpath, buf.toString());
        conn.close();

      }
    }
    catch(Exception ex)
    {
      throw new SAXException(ex);
    }
  }
```

Listing 4: endDocument Process

To compile the classes, you need to include `xmlparserv2.jar` and `classes12.zip` in the Java CLASSPATH. Note that, we assume you are using JDK 1.2.x and above. If you are using JDK 1.1.x, please use the JDBC library `classes111.zip`. The following command line will

compile and produces the compiled java classes - `ExDocumentHandler.class` and `ExElement.class`.

```
javac ExDocumentHandler.java
```

Now, we are ready to load the Java classes into the database server.

Loading and Resolving the Java Class

After compiling the java classes, you can use the `setup.csh` (for Solaris) or `setup.bat` (for Windows) to test the downloaded sample code discussed in the following sections by:

```
csh setup.csh or setup.bat
```

However, you can still follow the description in this paper to try the provided sample step by step.

With the utility `loadjava`, you can upload the Java source, class, and resource files into an Oracle database, where they are stored as Java schema objects. You can run `loadjava` from the command line or from an application, and you can specify several options including a resolver. Make sure you have `$ORACLE_HOME\bin` in your System Path to be able to run `loadjava`.

After using the `DBUser.sql` to create the xdkplsql user in the database, we can load the ExDocumentHandler.class and ExElement.class by running the following commands:

```
loadjava -resolve -verbose -user xdkplsql/xdkplsql ExElement.class
loadjava -resolve -verbose -user xdkplsql/xdkplsql
ExDocumentHandler.class
```

Publishing the Java Class by Defining the Specification

For each Java method callable from SQL or PL/SQL, you must write the call specification in PL/SQL. This exposes the method's top-level entry point to the Oracle server, as below:

```
CREATE OR REPLACE FUNCTION XMLDocumentValidation(xml IN VARCHAR2,dtd IN
VARCHAR2,root IN VARCHAR2 )return varchar2
IS LANGUAGE JAVA NAME
'oracle.xml.pm.sample.ExDocumentHandler.docValidationwithDTD(java.lang.S
tring,java.lang.String,java.lang.String) returns java.lang.String';
/

CREATE OR REPLACE FUNCTION XMLDocumentExpansionDTD(session_id IN NUMBER,
xml IN VARCHAR2,dtd IN VARCHAR2,root IN VARCHAR2 )return varchar2
IS LANGUAGE JAVA NAME
'oracle.xml.pm.sample.ExDocumentHandler.docExpandwithDTD(int,
java.lang.String,java.lang.String,java.lang.String) returns
java.lang.String';
```

After the Java stored procedure specifications are created, they can be called using either SQL statements or PL/SQL procedures as if they were built-in PL/SQL functions.

Example Using the Stored Procedures

You can call Java stored procedures from SQL DML statements, PL/SQL blocks, and PL/SQL subprograms. Using the SQL `CALL` statement, you can also call them from the top level (from SQL*Plus, for example) and from database triggers. The following example shows how to do XML data streaming using the created Java stored procedures.

1. Creating Database tables to store XML mapped data and the XML/DTD Documents.

```
create table xml_index
(
  session_id NUMBER,
  name varchar2(4000),
  value varchar2(4000)
);

create table xml_tbl
(
  id number,
  xml CLOB
);

create table dtd_tbl
(
  id number,
  dtd CLOB
);
```

2. Loading the XML Document into the Database

You can use the SQL commands to insert the data shown in `DBData.sql`:

```
insert into xml_tbl values(1,fileToClob('test.xml'));
insert into dtd_tbl values(1,fileToClob('test.dtd'));
```

3. Calling the Java Stored Procedure to stream the input XML Document into the xml_index table

```
CREATE OR REPLACE PROCEDURE docExpand(xmlid IN NUMBER, dtdid IN NUMBER)
IS
  p_xml varchar2(32767);
  p_dtd varchar2(32767);
  p_out varchar2(4000);
  p_type varchar2(300);
  p_dtdid NUMBER;

begin
  select xml into p_xml from xml_tbl where id=xmlid;
  select dtd into p_dtd from dtd_tbl where id=dtdid;

  ---------------------------------------------------------------------
  -- Expand the document with DTD validation
  ---------------------------------------------------------------------
  p_out := XMLDocumentExpansionDTD(2, p_xml, p_dtd,'transaction');
  printBufferOut(p_out);

  ---------------------------------------------------------------------
  -- Select DTD for Validation
  ---------------------------------------------------------------------
  select value into p_type
  from xml_index
  where session_id=2 and name='/transaction/@type';

  if p_type = 'PPSR_REG_PRIOR_FS'
    then p_dtdid :=2;
    else p_dtdid :=1;
  end if;

  select dtd into p_dtd from dtd_tbl where id=p_dtdid;
  p_out := XMLDocumentValidation(p_xml, p_dtd,'transaction');
```

```
end;
```

You now can use the Java Stored Procedure to stream the XML document from PL/SQL and do the DTD validation as the same time After the data is streamed into the database table with the associated database session id and XPATH, multiple users can easily access/process the XML data by querying the table with SQL using the XPATH information.

Conclusion

This approach efficiently extracts all of the data out of an XML document with a single PL/SQL java stored procedure call and uses very little memory and high throughput when compared to approaches that use the PL/SQL DOM APIs.