Making XML Technology Easier to Use

An Oracle White Paper
March 2005

ORACLE

Marking XML Technology Easier to Use

Executive Overview 3
Introduction 3
XML Parsing 5
  DOM XML Parsing 5
    DOM 3.0 Load and Save .... **Error! Bookmark not defined.**
    DOM 3.0 Validation 8
  SAX XML Parsing 10
    SAX Output Serialization .... **Error! Bookmark not defined.**
XSL Transformation 12
  Built-in Grouping 13
  Temporary Trees 13
  Multiple Result Documents 13
  Character Mapping 14
XML Schema Validation 14
  XML Schema Validation Interfaces 15
JAXB Class Generator 16
XML Pipeline Processor 17
Conclusion 18

Making XML Technology Easier to Use

EXECUTIVE OVERVIEW

Over the past 5 years, the acceptance of XML has grown in modern business applications. XML technology gives businesses the infrastructure to satisfy their increasing demands for accessing and exchanging business data. However, the complexity of XML technology may result in high application development and maintenance cost.

To simplify the use of XML technology so as to reduce the cost, Oracle XML Developer's Kit 10g (XDK) extends the existing XML standards support and introduces new technologies and features that simplify XML creation, access, transformation, and validation.

INTRODUCTION

When companies expand their business by merges, acquisitions or increased engagements with their partners by building business partner networks, they need to bring applications developed by different people, at different times and different locations, together. Therefore, a standard data format is required to establish a framework for exchanging the business data. Extensible Markup Language, XML, the widely accepted industry standard, offers the real paradigm to meet these needs for building such Business-to-Business (B2B), Business-to-Customer (B2C) and Enterprise Application Integration (EAI) applications.

On the other hand, when moving the business forward by adopting new technologies, companies cannot throw away their legacy applications and replace them with the new ones. XML is a data format that incorporates metadata with data thereby facilitating the use of this legacy data for new systems and sharing across platforms and applications.

The Internet has brought competition into local businesses from across the nation and even the world. Today's business can no longer depend upon paper to convey transactions. Moving the business over the Internet, XML provides a powerful data abstraction for transaction data management for the enterprise.

Modern content publishing cannot be limited to print and Web pages. There is an increasing need to make the content available across media and device types. By setting up "data islands" over the networks in XML, companies can have flexible content representations with easy transformations to wireless devices in

WML, printable format in PDF, and even the interactive graphics in the format of Scalar Vector Graphics (SVG).

XML is widely used by companies to expand the business boundary, integrate legacy applications, and create new business systems. However, the current XML technology is not easy to use.

First, XML sounds simple while it is actually not because creating and reading an XML document is not limited to just specifying start tags, end tags and attributes for the XML elements. The internal entities, external entities, parameter entities, the PCDATA, CDATA and all of the new concepts in XML require significant time to pick up. Needless to say this is compounded by the bewildering array of related standards such as XML Schema, XSLT, XPath, Namespaces and XQuery standards.

Second, the generic XML programming interfaces, such as DOM and SAX, focus on XML document manipulation instead of business logic. Therefore, it is not straightforward for application developers to process business data.

On the other hand, because of the extensible characteristics and lack of design guidance in the industry for XML, there are many different ways represent the same business data. Therefore, processing XML, on a per document basis is not efficient as there is no effective strategy for XML metadata processing. Oracle XDK 10g addresses these issues by providing a set of new XML development features and technologies to simplify XML application development, which includes:

- DOM 3.0 and SAX serialization interfaces that simplify the XML Parsing
- XSLT and XPATH 2.0 support that simplifies XSL transformations through higher order functionality
- Java Architecture for XML Binding (JAXB) support that simplifies XML processing by binding XML data to objects which can be directly manipulated by application logic
- XML Schema validation interfaces that allow applications to query the metadata definitions during the SAX XML parsing and provide high efficiency stream-based XML metadata processing. This new technology facilitates the “open” content processing for XML documents in heterogeneous formats.
- XML Pipeline Processor that establishes a reusable component framework supporting declarative pipelining of XML resources for business applications

XML PARSING

XML parsers are the components that read in XML documents and provide the programmatic access to the content and structure of XML. It is the first step when users need to process and access XML data. The Oracle XDK 10g extends the existing support for both the Document Object Model (DOM) and Simple API for XML (SAX) standards in its XML parsers.

DOM XML Parsing

DOM is arguably the most popular API for manipulating XML in use today. It presents an XML document in an object-based form as shown in Figure 1.

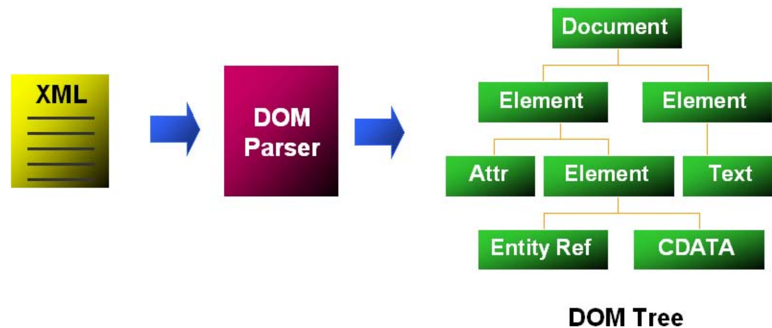

Figure 1: DOM XML Parsing

DOM interfaces are a set of object-based APIs to access XML data and they work across languages, including Java, C, C++, JavaScript and PL/SQL. The current XDK DOM implementation supports both W3C DOM Level 1 and DOM Level 2 recommendations in Java, C and C++.

Though the in-memory DOM object tree leads to high memory costs from processing large XML documents, it is still widely used in the production applications all over the world. This is because it allows quick access and dynamic updates on the XML content and structure, which is needed by the XML editing and transformations. Oracle XDK 10g extends the DOM standards support by introducing DOM 3.0 Load and Save and the DOM 3.0 Validation.

DOM 3.0 Load and Save

Because of the high cost of migrating applications, portability is critical for modern business systems. Because there is no definition for loading XML data and serializing the DOM objects in W3C DOM Level 1 and DOM Level 2 standards, developers have to rely on vendor-specific code when handling the input and output for DOM parsers. This makes XML applications no longer portable. Additionally, applications need a flexible DOM building process to

reduce the XML footprint in memory, as the DOM tree can be many times the size of the input XML.

DOM 3.0 Load and Save solves these problems by standardizing the input and serialization process for DOM parsers thus making it possible to build fully standards compliant and thus portable DOM XML applications.

Oracle XDK 10g provides the new **createLSInput** and **createLSParser** interfaces for reading in XML data sources. For output DOM objects can then be serialized using the new **LSOutput**, and **LSSerializer** interfaces.

The new **DOMParserFilter** interface in the XDK DOM 3.0 Load and Save allows filtering the input XML Content to build a customized DOM.

To improve the scalability of DOM XML applications, Oracle XDK 10g also provides asynchronous XML DOM parsing in DOM 3.0 Load and Save. The following example illustrates the use of asynchronous DOM parsing:

```
import org.w3c.dom.ls.LSParser;
import org.w3c.dom.ls.LSInput;
import org.w3c.dom.ls.LSLoadEvent;
import oracle.xml.parser.v2.XMLLSParser;
import org.w3c.dom.events.Event;
import org.w3c.dom.events.EventListener;
import oracle.xml.parser.v2.XMLDocument;
import oracle.xml.parser.v2.XMLDOMImplementation;
 …
public class DOMAsynLoading implements EventListener {
  static DOMAsynLoading test;
  boolean parseFlag=true;
  public static void main (String[] args) {
    test = new DOMAsynLoading();
    test.testParse("src/xml/book.xml");
  }

  public void testParse(String input) {
    short mode;
    DOMImplementationLS impl = new
XMLDOMImplementation();
    mode = DOMImplementationLS.MODE_ASYNCHRONOUS;
    LSParser parser = impl.createLSParser(mode, null);
     try {
      ((XMLLSParser)parser).addEventListener("ls-load",
(EventListener)test, false);
    } catch(Exception e) {
      e.printStackTrace();
    }
    LSInput inp = impl.createLSInput();
    try {
      URL url = createURL(input);
      inp.setSystemId(url.toString());

      System.out.println("Asynchronous DOM parsing...");
      parseFlag = false;
      Document doc = parser.parse(inp);
```

```
        // Other Application Code
        // Neen DOM Document
        while(!parseFlag) {
          try {
            System.out.println("Waiting for the DOM
Parsing...");
            Thread.sleep(10);
          } catch(Exception e) {
          }
        }

      } catch(Exception e) {
        e.printStackTrace();
      }  }

  public void handleEvent(Event evt) {
      Document doc = ((LSLoadEvent)evt).getNewDocument();
      try {
        ((XMLDocument)doc).print(System.out);
        parseFlag = true;
      } catch(Exception e) {
        e.printStackTrace();
      }
  }
```

To set up the asynchronous DOM parsing, applications need to register an **EventListener** to **DOMbuilder** using the **DOMBuilder.addEventListener()** function and set the processing mode to be **DOMImplementationLS.MODE_ASYNCHRONOUS**. Then, the DOM XML parsing can be executed in a background process. After the DOM XML parsing is completed, the DOM parser will call the **HandleEvent()** callback function to notify the application and pass back the content handler for the DOM document in memory.

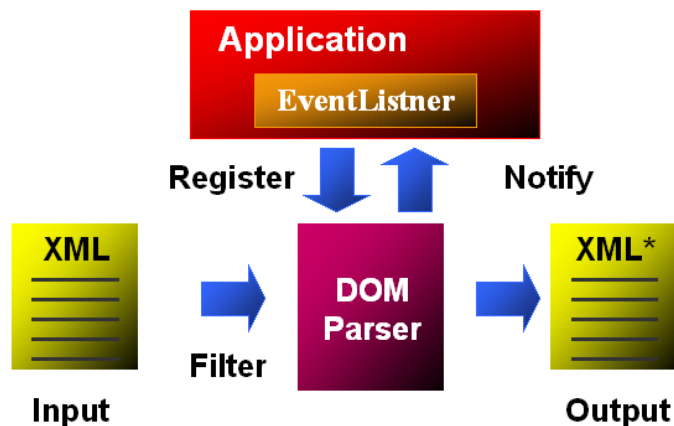

Figure 2: DOM 3.0 Load and Save

Figure 2 illustrates the DOM 3.0 Load and Save function flow that permits portable XML implementations by standardizing the DOM input and output

process and allows content filtering to reduce the DOM object footprint for the scalability. Additionally, the asynchronous DOM parsing in DOM 3.0 Load and Save supports multi-threading of the XML processing for scalability.

DOM 3.0 Validation

In enterprise applications, it is important that the XML documents are created following company or industry standards and be able to catch errors as early as possible. DOM 3.0 Validation, as shown in Figure 3, allows users to retrieve the metadata definitions from XML schemas, query the validity of DOM operations and validate the DOM documents or sub-trees against the XML schema.

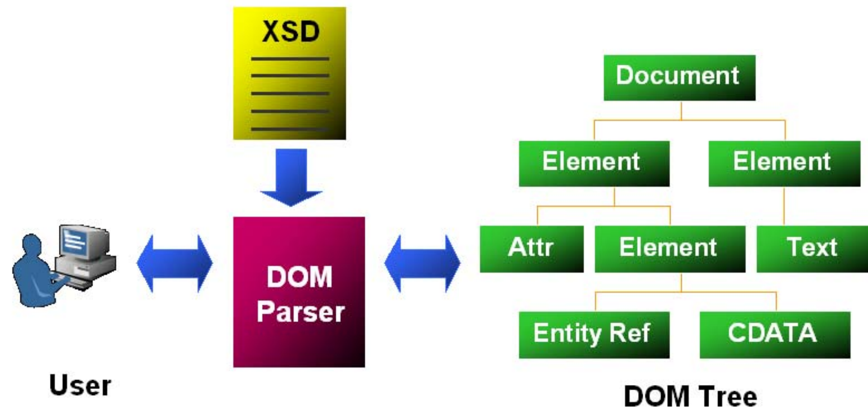

Figure 3: DOM 3.0 Validation

Figure 4 shows an example XML Schema (**book.xsd**) as defined for a book-listing document:

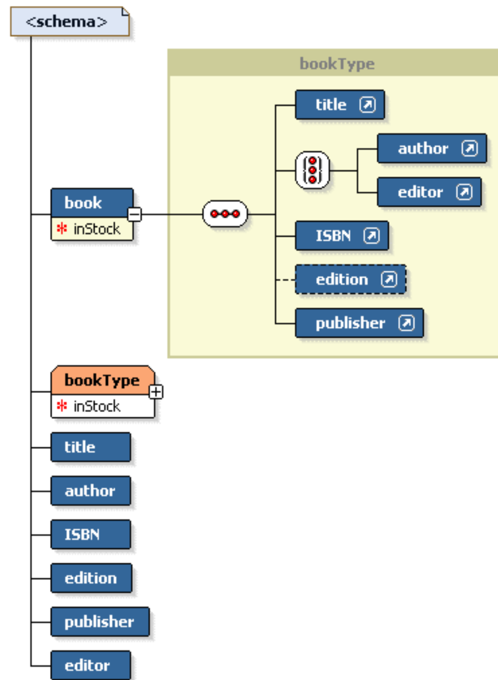

Figure 4: XML Schema for Book.xml

With the DOM 3.0 Validation APIs, applications can parse the XML document and query the metadata when updating XML DOM tree. For example, if we have the document as shown bellow:

```
<?xml version = '1.0'?>
<book inStock="Yes">
 <title>Compilers: Principles, Techniques, and
Tools</title>
  <author>Alfred V.Aho, Ravi Sethi, Jeffrey D.
Ullman</author>
  <ISBN>0-201-10088-6</ISBN>
  <edition>Second</edition>
  <publisher>Addison Wesley</publisher>
</book>
```

An application can be built to provide an interactive DOM 3.0 Validation. The command is formatted as "*Element_Name DOM_Functions*". For example, in order to find out "*What are the defined elements in the XML schema?"* a user can submit the following query to the DOM parser , as shown below, where the **doc** refer to the DOM document:

```
?doc getDefinedElements
publisher
author
edition
editor
book
title
ISBN
```

The queries such as *"what are the allowed parent or child elements*", "*what is the content type for the element*", "*what are the allowed sibling elements*" are

all available for use. The following example checks what are the allowed children, what are the required attributes for the **<book/>** element:

```
?book getAllowedChildren
publisher
ISBN
editor
title
edition
author

?book getRequiredAttributes
inStock
```

With the XML schema definitions, we can also list the enumerated values for the XML elements:

```
?edition getEnumeratedValues
First
Second
Third
Fourth
Fifth
```

Before a DOM operation, users can query the DOM parser and ask if the operation is legal according the metadata definitions. The following example asks "*Can the **<author/>** element can be remove from its parent **<book/>**?*"

```
?book canRemoveChild author
VAL_FALSE
```

Based on our schema the DOM parser would report that this operation is illegal.

Finally, users would like to validate the edited or newly constructed document before serializing it or passing to another XML process as shown in the following example:

```
?doc validateDocument
VAL_TRUE
?edition nodeValidity
VAL_TRUE
```

DOM 3.0 validation provides the APIs to validate the XML document or sub-tree. With all of the guidance from the DOM parser, XML editing is much more robust.

SAX XML Parsing

SAX Parsing is event-based XML parsing, which reports the events, such as **startDocument** and **endDocument** events to a set of callback functions, which is shown as follows:

Figure 5: SAX XML Parsing

Compared to DOM XML parsing, SAX is a steam-based XML process that does not build in-memory objects. This lightweight XML parsing is efficient when dealing with large XML documents. However, users cannot update the XML content in place and it is not efficient for dynamic access. Therefore, SAX parsing is normally useful when retrieving, filtering, and searching the content from large XML documents.

SAX Output Serialization

Currently, users have to deal with a set of callback functions when using a SAX parser, which requires a certain amount of work. When submitting SQL queries to the database and receiving the SAX streaming output from XSU as shown in Figure 6, this additional work is tiresome due to the number of handlers that must be managed.

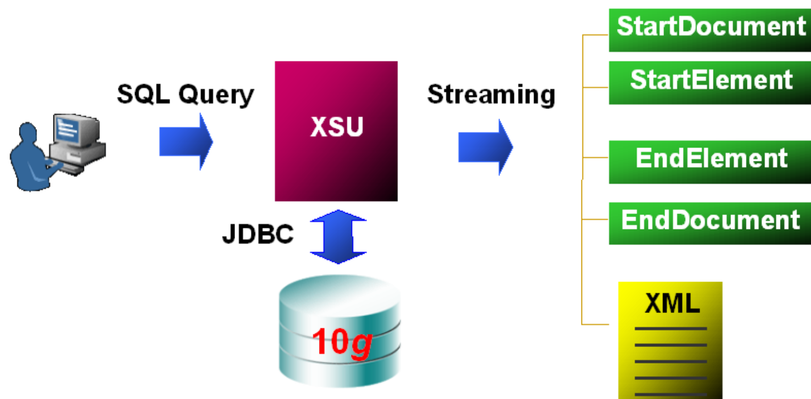

Figure 6: Streaming Output for Database Queries

Oracle XDK 10g simplifies this process by introducing a new Java interface, **oracle.xml.parser.v2.XMLSAXSerializer**, to serialize the SAX output. This new interface also provides extensive output options that further simplify the content delivery process. The output options allow users to specify if the pretty printing format is needed, what the XML declaration and encoding information

is, which if any are the elements whose content needs to be set as CDATA sections and what the DTD **system-id** and **public-id** are to be. To use this new feature, one simply uses it as another type of SAX content handler. For example, one can register it to the XSU SAX output interface as follows:

```
OracleXMLQuery.getXMLSAX(sample);
```

This would generate unbounded XML documents from result sets returned from queries where in the past XSU need to create a DOM to produce the same document.

XSL TRANSFORMATION

XSL Transformation (XSLT), as shown in Figure 7, applies the XSLT Stylesheets to the input XML documents to transform and apply formatting semantics on the text outputs.

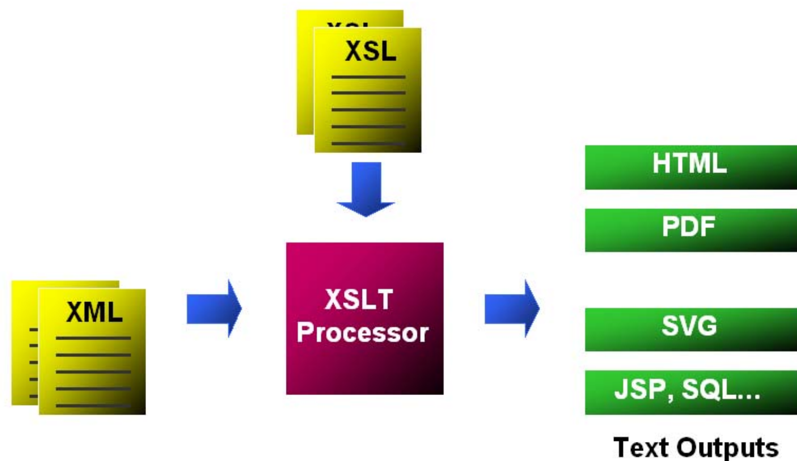

Figure 7: XSL Transformation

In the Oracle XDK 10g, the Java XSL processor provides a wide range of new features and enhancements defined in the W3C XSLT and XPATH 2.0 working drafts. In this paper, we only list some of the easy-to-use features that enable efficient and powerful XSL transformations:

- The new built-in grouping support that allows simplified and efficient content grouping
- The new temporary trees support that eliminates extra work for node-set conversions
- The new support to create multiple output documents in one XSL transformation
- The new character mapping support that eliminates the error-prone character escaping process

These new XSLT 2.0 features enrich the functionality while simplifying the XSLT transformations thereby optimizing the design and processing.

Built-in Grouping

In XSLT 2.0, users can use the **<xsl:for-each-group>** element to select the items to be grouped, which can contain a number of **<xsl:sort>** elements (for sorting the groups) followed by the instructions that create content for the particular group. The **<xsl:for-each-group>** instruction can take one of four attributes, which are used to determine how the selected items are grouped together:

- **group-by**, which ignores the order in which the items appear in the selected sequence.
- **group-adjacent**, which only groups together adjacent items with the same value.
- **group-starting-with**, which holds a pattern that matches the first node in each group.
- **group-ending-with**, which holds a pattern that matches the last node in each group.

The members of the group can be returned with the **current-group()** function.

Temporary Trees

In XSLT 1.0, the XSL variables and intermediate XSL transformation results are simply strings. Therefore, a user cannot access the content using the XPath expressions and it is difficult to modularize the XSL processing. As a result, XSL processors provide extension functions to support such operations, such as the Oracle XDK's **ora:node-set()** function which converts the result fragments into a document tree. The problem with using vendor-provided extensions is that the XSL Stylesheets are no longer portable.

XSLT 2.0 solves this problem by introducing *Temporary Trees*, which are constructed by evaluating an **<xsl:variable>**, **<xsl:param>** or **<xsl:with-param>** elements. Because temporary trees are document nodes, they can be accessed with XPath expressions and processed by the **<xsl:apply-templates>**, **<xsl:for-each>** elements and the **Key()** and **id()** functions.

This feature eliminates node set conversions and allows XSLT users to break up complex transformations into several steps and providing iterative processing of XML documents.

Multiple Result Documents

Without the ability to create multiple result documents from a single transformation, XSLT 1.0 applications usually use parameters to indicate which page should be created from a particular transformation, and use separate

transformations each time. This can take more time, particularly when the same calculations must be repeated for each page.

Oracle XDK has provided the **<ora:output>** extension to simplify these kinds of operations. However, there is still the issue of breaking the portablility of the XSL stylesheets.

In Oracle XDK 10g, one can instead utilize the new **<xsl:result-document>** element from XSLT 2.0 for such operations. This allows paginating the XSLT outputs, so that each document only includes part of the result of the transformation. For example, in transforming a CD catalog, each **<CD/>** element can be created as separate result document. With this feature, users can easily generate pages that use Hypertext Markup Language (HTML) frames. It's also possible to create supplementary files that are referenced by the main output, for example Scalable Vector Graphics (SVG) graphics, Cascading Style Sheets (CSS) stylesheets, or files containing meta information about the main output.

Character Mapping

In XSLT 1.0, specifying the character escaping is difficult to use and error prone. In XSLT 2.0, the problem is solved by allowing users to declare mapping characters with an **<xsl:character-map>** element at the top level of the stylesheet. The **<xsl:output-character>** element within the **<xsl:character-map>** is used to define the mapped characters that should not be escaped on output.

These characters usually come from the Unicode private use area (between **#xE000** and **#xF8FF**). For example, you can set up a character map to generating JSP code by stating that whenever the character **#xE001** is encountered in a text node or attribute node, it should be replaced in the output by the string, **<%**. Similarly, any occurrence of the character **#xE002** should be replaced in the output by the string, **%>**.

Using character maps is much more robust than using disable-output-escaping because the unescaped characters are guaranteed to persist even when a text node or attribute is copied in a temporary tree. (For this reason d-o-e has been deprecated for XSLT 2.0.) In addition, the XSL processors are more likely to produce consistent results with the character maps.

XML SCHEMA VALIDATION

The Java XML Schema Processor in Oracle XDK 10g fully supports the W3C XML Schema 1.0 recommendation and provides both LAX and STRICT mode XML Schema validation by the **XMLParser.SCHEMA_LAX_VALIDATION** or **XMLParser.SCHEMA_STRICT_VALIDATION** options for the XML parser. However, the XML schema processor is not limited to document validations. Since rich metadata information are provided in the XML schema

documents, users can extend it to perform more complex XML document processing.

XML Schema Validation Interfaces

In the Oracle XDK 10g, the Java XML Schema processor is re-architected to provide new XML Schema validation interfaces allowing synchronous retrieval of the metadata information and the validation processing status from the XML Schema processor during the SAX XML parsing.

Users can get the current validating status (**LAX** and **STRICT**) through the **XSDValidator.getCurrentMode()**, look up the XML elements and attribute content types by the **XSDValidator.getElementDeclaration()** and the **XSDValidator.getAttributeDeclarations**() APIs, and obtain the associated XML Schema annotations with **XSDValidator.getAnnotation().**Figure 8 illustrates that the new functionality that enables building lightweight and high-performance XML metadata processing applications using the SAX streaming.

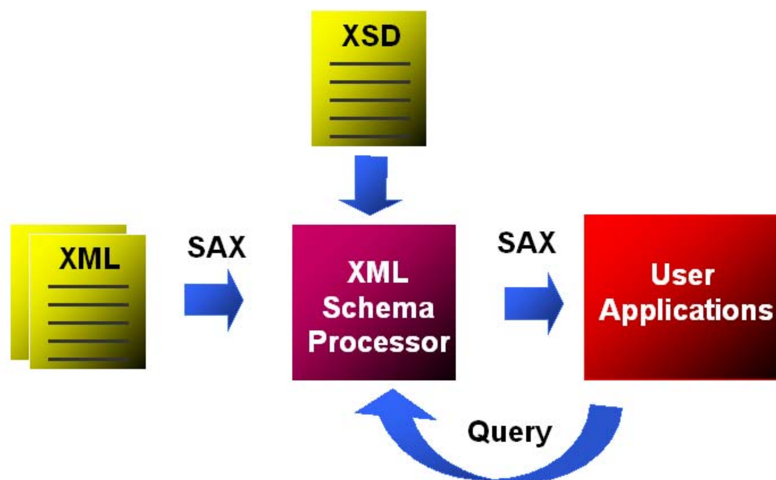

Figure 8: Streaming XML Metadata Processing

Utilizing the new metadata-processing model, an XML messaging application, as shown in Figure 9, is built to allow "open" XML processing of XML messages.

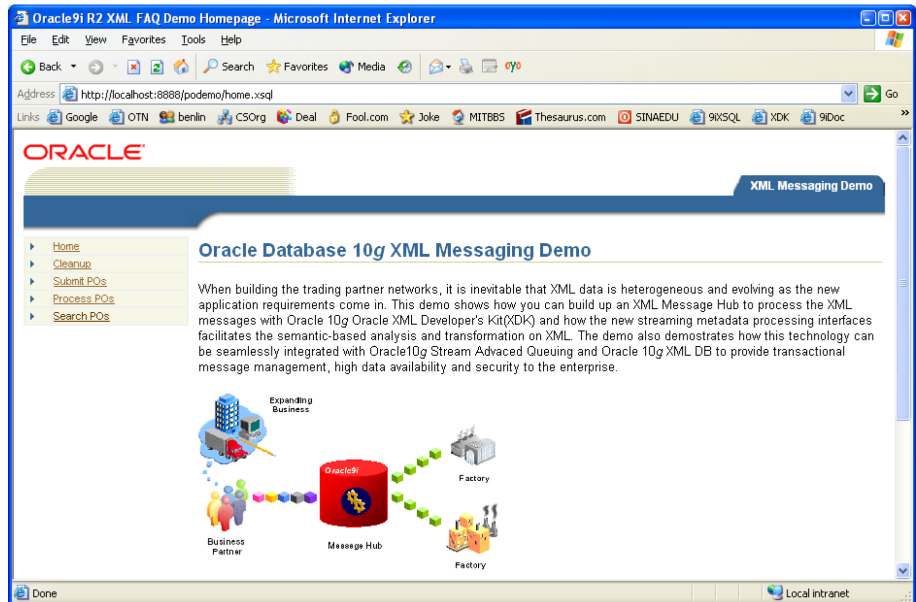

Figure 9: XML Messaging Demo

When building trading partner networks, it is inevitable that XML data is heterogeneous and evolving as new application requirements come in. Dealing with such a variety of XML documents individually is inefficient.

This application shows that the new streaming metadata processing interfaces in the XDK facilitates open content analysis and transformation for such XML documents. The demo also demonstrates that this technology can be seamlessly integrated with Oracle Streams Advanced Queuing 10g and Oracle XML DB 10*g* to provide transactional, high available and secure XML message management for the enterprise.

JAXB CLASS GENERATOR

In Oracle XDK 10g, the Java Architecture of XML Binding (JAXB) 1.0 standard is fully supported by allowing users to generate Java Classes based on XML Schemas and marshal/unmarshal XML documents through the Java object interfaces with options for XML Schema validation. The new JAXB Class Generator allows user to customize the generated Java classed using both XML schema annotations and through an internal customization files.

Figure 10: Create XML Element in DOM vs. JAXB XML Unmarshalling

As shown in Figure 10, the JAXB object binding interfaces directly with the business logic for the XML content rather than the document structure thus greatly simplifies the XML processing.

XML PIPELINE PROCESSOR

XML applications may involve many different processes, such as XML parsing, XML Schema validation, XSL Transformations and so on. To deal with different sets of interfaces for each process requires additional development time and possibly a learning curve.

To help XML developers save time to process XML, Oracle XDK 10g provides a new XML Pipeline processor that is complaint to the W3C XML Pipeline Definition Language Version 1.0 Note.

Using the Oracle XML Pipeline Processor, a user just needs to create a pipeline document in XML. This document declares the use of the available XML processing components, specifies the inputs and outputs for XML processes, and describes the processing relationships. In the current Oracle XML Pipeline Processor, the available components include the DOM and SAX XML parsers for parsing the XML documents, the XML Schema processor for the XML Schema validations, XSL processor for transforming XML documents and the XML compressor to compress XML into a binary format for serialization.

The run-time pipeline controller is built based on a reusable component framework. It executes the chain of XML processing according to the descriptions in the pipeline document and returns a particular result as shown in Figure 11.

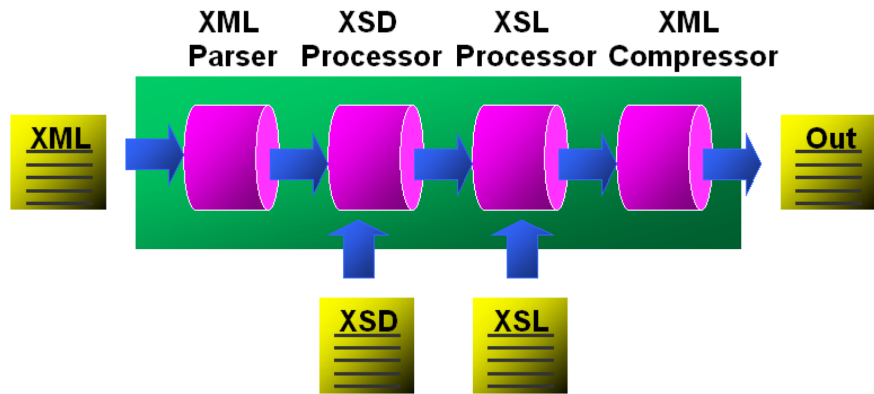

Figure 11: XML Pipeline Processing

The built-in optimization in the Oracle XML Pipeline Processor further saves a developer's time by optimizing the XML processing.

CONCLUSION

Due to the flexibility and broad applicability of XML technology, businesses are starting to use an XML infrastructure for their enterprise applications. Over the past 5 years, both software vendors and end-user companies around the world have made a huge investment in building XML-enabled database applications.

Representing the data as XML makes it more useful and flexible for business use. However, the overhead introduced by XML processing along with the myriad of ways to represent the same business data further complicate XML data processing and sharing. Additionally, current XML processing interfaces and functionality provided by existing standards can be cumbersome or insufficient when trying to meet increasing business application requirements.

Oracle XDK 10g provides XML infrastructure components that meet and extend the standards while introducing features that greatly simplify the use of XML technology. Thus, the Oracle XDK 10g can meet the requirements, and serve as the infrastructure, for current and future enterprise applications.

Making XML Technology easier to use
march 2005
Author: Jinyu Wang
Contributing Authors: Mark Scardina

Oracle Corporation
World Headquarters
500 Oracle Parkway
Redwood Shores, CA 94065
U.S.A.

Worldwide Inquiries:
Phone: +1.650.506.7000
Fax: +1.650.506.7200
www.oracle.com

Copyright © 2005, Oracle. All rights reserved.
This document is provided for information purposes only and the
contents hereof are subject to change without notice.
This document is not warranted to be error-free, nor subject to any
other warranties or conditions, whether expressed orally or implied
in law, including implied warranties and conditions of merchantability
or fitness for a particular purpose. We specifically disclaim any
liability with respect to this document and no contractual obligations
are formed either directly or indirectly by this document. This document
may not be reproduced or transmitted in any form or by any means,
electronic or mechanical, for any purpose, without our prior written permission.
Oracle, JD Edwards, and PeopleSoft are registered trademarks of
Oracle Corporation and/or its affiliates. Other names may be trademarks
of their respective owners.