

Manipulating data with XSL

Make changes to database contents without stored procedures

Skill Level: Intermediate

[Nicholas Chase](mailto:nicholas@nicholaschase.com) (nicholas@nicholaschase.com)

Author

24 Oct 2001

What happens if you need to manipulate data in a database, but you can't create stored procedures? You pull the information, manipulate it, and put it back, that's what. One way to conveniently do that is to retrieve the data into XML structures and then use XSL to manipulate it. This tutorial teaches you to use XSLT to manipulate data that has been pulled from a database and then restore it to the database. Examples are in Java, but the principles apply in other languages.

Section 1. Introduction

Should I take this tutorial?

This tutorial is designed to assist Java developers who need to manipulate data in a database without the advantage of stored procedures. The tutorial demonstrates how to manipulate database data using XSLT (eXtensible Stylesheet Language Transformation) instead of stored procedures.

This tutorial assumes you are already familiar with the basics of XML in general, the Document Object Model (DOM) in particular, and with the basics of XSLT and XPath. All of the examples use Java, with JDBC for database access, but you can still get a basic grounding in the concepts without trying out the examples.

The links in [Resources](#) include referrals to tutorials on XML and DOM basics, XSL and XPath basics, and database access using JDBC (including the basics of SQL).

What is this tutorial about?

The examples used throughout this tutorial demonstrate the analysis and conversion of Web access logs stored in a database using XSLT. The application then adds the new data to separate database tables.

This tutorial explains the basics of retrieving data from a database into a DOM document, transforming the DOM document into a second document, and inserting the data from the second document into the database using updatable `ResultSet`s in Java. (Though the examples are all written in Java, the concepts are the same in any programming language and the tutorial can assist any developer who wants to learn how to manipulate data with XSLT.) Several of the more advanced features of XSLT and XPath are covered, demonstrating some of the ways that XSLT style sheets can be used to emulate the programming capabilities of database stored procedures.

Tools

This tutorial will help you understand the topic even if you read the examples rather than trying them out. If you do want to try the examples as you go through this tutorial, make sure you have the following tools installed and working correctly:

- A text editor: XML and Java source files are simply text. To create and read them, a text editor is all you need.
- A Java environment, such as the Java 2 SDK, which is available at <http://java.sun.com/j2se/1.3/> or the IBM JDK, which is available at <http://www.ibm.com/developerworks/java/jdk/index.html>.
- Any database that understands SQL, as long as you have an ODBC or JDBC driver. You can find a searchable list of more than 150 JDBC drivers at <http://industry.java.sun.com/products/jdbc/drivers>. (If you have an ODBC driver, you can skip this step and use the JDBC-ODBC bridge, which is included as part of the Java 2 SDK.) This tutorial uses JDataConnect, available at <http://www.jnetdirect.com/products.php?op=jdataconnect>.
- Java APIs for XML Processing, version 1.1 or later. Also known as JAXP, this is the reference implementation that Sun provides. You can download JAXP (you must have version 1.1 or later; the 1.0 release did not include support for XSLT) from http://java.sun.com/xml/xml_jaxp.html. This download also includes the Xalan 2 XSL Transformation engine. You can also download it from <http://xml.apache.org/xalan-j/index.html>. Xalan is updated more often than JAXP these days, and it also includes the JAXP

1.1.1, so you may prefer to download Xalan Java to obtain a current release of both JAXP and Xalan. If you're not sure whether you have the correct version of JAXP, check to see whether you have the `javax.xml.transform` package installed; if it's there, you have what you need.

Other XSLT processors: TraX was built in to JAXP beginning with JAXP version 1.1 to provide a simple way for Java developers to choose any XSLT processor without affecting the underlying code. So although the current version of JAXP includes Xalan, you can use a different XSLT processor if you prefer.

Other languages: If you want to adapt the examples, you can download a C++ implementation of Xalan from the Apache Project at <http://xml.apache.org/xalan-c/index.html>. Similarly, while the examples use TrAX as their transformation methodology, and TrAX is currently available only to Java developers, you can adapt the examples to use a native transformation method in your language of choice and still gain a thorough understanding of the concepts demonstrated in this tutorial.

Conventions used in this tutorial

There are several conventions used in this tutorial to reinforce the material at hand:

- Text that needs to be typed is displayed in a **bold monospace** font. In some code examples, bold is used to draw attention to a tag or element being referenced in the accompanying text.
- *Emphasis/Italics* is used to draw attention to windows, dialog boxes, and feature names.
- A `monospace` font is used for file and path names.
- Throughout this tutorial, code segments irrelevant to the discussion have been omitted and replaced with ellipses (. . .)

Section 2. The project

What is a stored procedure?

In addition to the actual storage of data, many databases support the ability to create *stored procedures*. A stored procedure is like an application in many ways in that it

can be executed to perform some action, usually on the data in the database. Stored procedures can be created in many different languages, depending on the database system in use.

Stored procedures are literally stored within the database system itself, as opposed to being implemented as an external application. This architecture offers the advantages of internal caching, and of independence from the greater application. For these reasons, among others, they are often used to manipulate data according to business logic.

Unfortunately, not all databases offer this ability. Even where it is available, compatibility between different database systems can be difficult to achieve.

In a situation where stored procedures aren't available, the business logic can be built into the application itself, or it can be accomplished in other ways. Using XSLT to emulate the stored procedures makes it possible to keep the logic separate, allowing for changes in logic without changes to the application.

The overall plan

This tutorial takes the simulated Web log data stored in the database and transforms it using the following steps:

1. Retrieve the data: Store all of the data in the `visit` table in a DOM document.
2. Transform the data: Process the DOM document using an external XSLT style sheet that creates a new document. This document contains a root node with child nodes for each table of new data. Each of these nodes contains elements that represent a row of data. The result of the transformation is sent to a new DOM document.
3. Insert the new data: Loop through each row and column element of the new document to add new rows to the database.

The application is not specific to a particular database.

Set up the database and tables

Because none of the logic is performed within the database, it makes little difference which database lies behind the system. The examples show database access using JDBC, which is a vendor-independent means for accessing a database, but you can use any language to extract the data into a `Document` and put it back again. (See

[Resources](#) for more information on JDBC.)

Choose a database system and make sure that the database itself is installed and running. Choose a JDBC driver and consult the documentation for information on the driver class name. Some drivers also have their own way of designating the URL that points to the database, so again, consult the documentation for the information you need to create the database connection (which is covered in the next section of this tutorial).

Once the database is up and running, create the tables. The examples are expecting only three tables, as shown in the following `create` scripts:

```
create table visit (  
    userid      text,  
    sessionid   text,  
    visitdate   text,  
    referrer    text,  
    page        text)
```

```
create table clickstream (  
    sessionid   text,  
    frompage    text,  
    topage      text)
```

```
create table visitinfo (  
    sessionid   text,  
    userid      text,  
    pagecount   number,  
    start       text,  
    end         text)
```

Create these tables using the appropriate method for your database.

Section 3. Extracting data into an XML document

Create the database connection

Creating a connection to the database is the first step toward populating a `Document` object with the original data.

The exact details for creating a connection to the database vary according to the JDBC driver, but the basic principle is always the same: load the driver, then use it

to create a connection.

```
import java.sql.Connection;
import java.sql.DriverManager;

public class TransformData extends Object {

    public static void main (String args[]){

        //Declare the connection
        Connection db = null;

        try {

            //Load the driver and use it to create a connection
            Class.forName("JData2_0.sql.$Driver");
            db =
            DriverManager.getConnection("jdbc:JDataConnect://127.0.0.1/logs");

        } catch (Exception e) {
            System.out.println("Error: "+e.getMessage());
        }
    }
}
```

In actuality, the `DriverManager` creates the connection, using the first available `Driver` that can successfully connect to the given URL.

(To use the JDBC-ODBC bridge for testing, substitute the driver class name of `sun.jdbc.odbc.JdbcOdbcDriver` and connection URL of `jdbc:odbc:logs`, assuming a System DSN of `logs`.)

Close the connection

Although Java automatically destroys objects it no longer needs, it's important to explicitly close the connection to free up database resources.

```
...
try {

    //Load the driver and use it to create a connection
    Class.forName("JData2_0.sql.$Driver");
    db = DriverManager.getConnection("jdbc:JDataConnect://127.0.0.1/logs");

} catch (Exception e) {
    System.out.println("Error: "+e.getMessage());
} finally {

    //Close the database connection
    try {
        db.close();
    } catch (Exception e) {
        System.out.println("Error closing connection: "+e.getMessage());
    }
}
}
```

```
}
```

It's important to make sure the connection is closed even if the application throws an exception, so add a `finally` block to the `try-catch`. Ironically, the `close()` method may also throw an exception, so it needs its own `try-catch` block.

Select the data

Once the `Connection` is in place, retrieve the original set of data.

Create a `Statement` object and use it to populate a `ResultSet`; then use the `ResultSet` methods to get the actual data.

```
...
import java.sql.Statement;
import java.sql.ResultSet;
...
//Declarations
Connection db = null;
Statement statement = null;
ResultSet resultset = null;

try {

    //Load the driver and use it to create a connection
    Class.forName("JData2_0.sql.$Driver");
    db = DriverManager.getConnection("jdbc:JDataConnect://127.0.0.1/logs");

    //Create the Statement and use it to populate the Resultset
    statement = db.createStatement();
    resultset = statement.executeQuery("select * from visit");

    //Loop through the ResultSet and output the data
    while (resultset.next()) {
        System.out.print(" ");
        System.out.print(resultset.getString("userid"));
        System.out.print(" | ");
        System.out.print(resultset.getString("sessionid"));
        System.out.print(" | ");
        System.out.print(resultset.getString("visitdate"));
        System.out.print(" | ");
        System.out.print(resultset.getString("referrer"));
        System.out.print(" | ");
        System.out.println(resultset.getString("page"));
    }

    //Close the statement
    statement.close();

} catch (Exception e) {
    System.out.println("Error: "+e.getMessage());
}
...
```

The `executeQuery()` method returns a `ResultSet` object that contains all of the data (if the query returns any). In this case, the query returns all of the data in all columns of the `visit` table. The "pointer" that indicates the current row starts off before the first row (if there is one), so the `next()` method returns `true` as long as it finds data.

Populate the `ResultSetMetaData` object with the `getMetaData()` method for the `ResultSet`. This prepares `resultSetmetadata` with all of the information the application needs.

Use the number of columns, retrieved with `getColumnCount()`, to loop through each of the columns in the row. For each column, retrieve the name using `columnName()` and the value using the original `getString()` method, this time using the index instead of the name.

Figure 2. Several rows of data

```
userid=nickc
sessionid=2453245as3q34
visitdate=2001-09-09 19:28:22.000
referrer=recs.html
page=books.html

userid=saraha
sessionid=45645623123as
visitdate=2001-09-09 19:29:33.000
referrer=outside.html
page=authors.html

userid=saraha
sessionid=45645623123as
visitdate=2001-09-09 19:29:53.000
referrer=authors.html
page=books.html
```

Next, use this data to populate the DOM Document.

Create the document

Now that the data is available, create a DOM Document to hold it.

```
...
import org.w3c.dom.Document;
import javax.xml.parsers.DocumentBuilder;
import javax.xml.parsers.DocumentBuilderFactory;

public class TransformData extends Object {

    public static void main (String args[]){

        //Declare the document
        Document dataDoc = null;

        try {
            //Create the document builder
            DocumentBuilderFactory dbfactory =
                DocumentBuilderFactory.newInstance();
            DocumentBuilder docbuilder = dbfactory.newDocumentBuilder();

            //Create the new document(s)
            dataDoc = docbuilder.newDocument();
        }
    }
}
```

```

    } catch (Exception e) {
        System.out.println("Problem creating document: "+e.getMessage());
    }

    //Declarations
    Connection db = null;
    Statement statement = null;
    ...

```

Even though there is no file to parse, creating a new Document object with JAXP requires the use of a DocumentBuilder. The DocumentBuilderFactory creates the DocumentBuilder, which creates a new Document. The dataDoc Document is declared outside the try-catch block to allow its use in other sections of code.

Populate the document

Once a Document exists, populate it with the data. Create an element for each row and add the columns to it, then add the row to the root element.

```

...
import org.w3c.dom.Element;
...
    //Declarations
    Connection db = null;
    Statement statement = null;
    ResultSet resultset = null;
    ResultSetMetaData resultmetadata = null;

    Element dataRoot = dataDoc.createElement("log");

    try {
...
        //Loop through the ResultSet and output the data
        while (resultset.next()) {

            //Create the element to hold the row
            Element rowEl = dataDoc.createElement("visit");

            //For each row, loop through each column
            for (int i=1; i <= numCols; i++) {
                //Get name and value
                String colName = resultmetadata.getColumnName(i);
                String colValue = resultset.getString(i);

                //Create a new element with the column name and
                //add the value, then add the element to the row
                Element columnEl = dataDoc.createElement(colName);
                columnEl.appendChild(dataDoc.createTextNode(colValue));
                rowEl.appendChild(columnEl);

            }
            //Add the row element to the root
            dataRoot.appendChild(rowEl);
        }

        //Add the root to the document
        dataDoc.appendChild(dataRoot);

        System.out.println(dataDoc.getDocumentElement().toString());

```

```

    //Close the statement
    statement.close();

    } catch (Exception e) {
    ...

```

First, create a root element for the document and call it `log`, then create an element for each row. This, too, has an arbitrary name (in this case, `visit`), representing a visit to the site. (Later, the style sheet will use these names.)

For each row, there is a set of columns. Rather than outputting the names and values, create a new element for each, and populate it with a new text node containing the value; then add each column element to the row element.

Once the row is complete, add it to the root element, `dataRoot`. Once the root element contains all of the rows, add it to the document itself.

Figure 3. Text output of the XML document

```

<log><visit><userid>nickc</userid><sessionid>2453245as3q34</sessionid><visitdate>
2001-09-09 19:27:03.000</visitdate><referrer>home.html</referrer><page>recs.htm
l</page></visit><visit><userid>nickc</userid><sessionid>2453245as3q34</sessionid>
<visitdate>2001-09-09 19:28:22.000</visitdate><referrer>recs.html</referrer><pa
ge>books.html</page></visit><visit><userid>saraha</userid><sessionid>45645623123
as</sessionid><visitdate>2001-09-09 19:29:33.000</visitdate><referrer>outside.ht
ml</referrer><page>authors.html</page></visit><visit><userid>saraha</userid><ses
sionid>45645623123as</sessionid><visitdate>2001-09-09 19:29:53.000</visitdate><r
eferrer>authors.html</referrer><page>recs.html</page></visit><visit><userid>nick
c</userid><sessionid>2453245as3q34</sessionid><visitdate>2001-09-09 19:31:14.000
</visitdate><referrer>books.html</referrer><page>authors.html</page></visit><vis
it><userid>seanac</userid><sessionid>535626411asd2</sessionid><visitdate>2001-09
-09 19:33:43.000</visitdate><referrer>home.html</referrer><page>recs.html</page>
</visit><visit><userid>nickc</userid><sessionid>62345235ser32</sessionid><visitd
ate>2001-09-09 19:34:58.000</visitdate><referrer>home.html</referrer><page>recs.
html</page></visit><visit><userid>seanac</userid><sessionid>535626411asd2</sessi
onid><visitdate>2001-09-09 19:50:22.000</visitdate><referrer>recs.html</referrer
><page>authors.html</page></visit></log>

```

The `Document` object is now ready to transform.

Section 4. Transforming the data

The transformation process

Part of the overall goal of this project is to take one `DOM Document` object and transform it into another `DOM Document` object according to an external XSL style sheet. Several XSL processors capable of handling this task exist, and the Transformation API for XML (TrAX) was designed to allow the maximum flexibility in choosing a processor by providing a common API that is processor-independent. TrAX is part of the JAXP 1.1 API, so it is readily available for Java developers to

use.

The process of transforming data with TrAX is straightforward:

1. Determine the source and result. These may be files, in-memory Documents, or even a stream of data, such as a SAX stream or output to the screen.
2. Create the Transformer. The application can use the Transformer to perform the transformation directly or to create a Template that can improve performance in situations where the same style sheet is applied multiple times.
3. Transform the data. This process sends the transformed data to the object designated as the result.

To use an XSLT processor directly, consult the documentation for the specific steps involved.

Create the sources and result

In the case of this tutorial, the source and result are simply DOM Document objects. The source is the Document object the application has already built and populated, and the result is an empty Document object.

```
...  
  
import javax.xml.transform.Source;  
import javax.xml.transform.Result;  
import javax.xml.transform.dom.DOMSource;  
import javax.xml.transform.dom.DOMResult;  
  
public class TransformData extends Object {  
    public static void main (String args[]){  
  
        //Declare the document  
        Document dataDoc = null;  
        Document newDoc = null;  
        try {  
            //Create the document builder  
            DocumentBuilderFactory dbfactory = DocumentBuilderFactory.newInstance();  
            DocumentBuilder docbuilder = dbfactory.newDocumentBuilder();  
  
            //Create the new document(s)  
            dataDoc = docbuilder.newDocument();  
            newDoc = docbuilder.newDocument();  
        } catch (Exception e) {  
            System.out.println("Problem creating document: "+e.getMessage());  
        }  
    }  
    ...  
  
    //Add the root to the document  
    dataDoc.appendChild(dataRoot);  
}
```

```

//Close the statement
statement.close();

//PERFORM THE TRANSFORMATION

//Set the source and result
Source source = new DOMSource(dataDoc);
Result result = new DOMResult(newDoc);

} catch (Exception e) {
    System.out.println("Error: "+e.getMessage());
...

```

Create the empty document just as the original document was created; with the `DocumentBuilder` object. These two documents become the parameters used to create a `DOMSource` and `DOMResult` object. (These objects correspond to the `StreamSource` and `StreamResult` used for transforming XML files.)

Set the style sheet

Next, set the style sheet.

In the case of this tutorial, the style sheet is a separate file. This allows a user to make changes to the resulting data while leaving the application intact.

```

...
import javax.xml.transform.stream.StreamSource;
...
    //Set the source and result
    Source source = new DOMSource(dataDoc);
    Result result = new DOMResult(newDoc);

    //Set the style sheet
    Source style = new StreamSource("logalyzer.xsl");

} catch (Exception e) {
    System.out.println("Error: "+e.getMessage());
...

```

Create the style sheet as a `StreamSource`. The next section details the building of the style sheet, but for now create a simple style sheet to act as a placeholder and save it as `logalyzer.xsl` in the same directory as the application.

```

<?xml version="1.0"?>
<xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform" version="1.0">

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

</xsl:stylesheet>

```

The style sheet must contain at least a single root node because the application is

sending the results to a `Document` object.

The application uses the style sheet in the creation of the `Transformer`.

Create the Transformer

The `Transformer` object is style sheet dependent, in that the `TransformerFactory` creates it based on the style sheet itself. Subsequent transformations require a new `Transformer` in order to use a different style sheet.

```
...
import javax.xml.transform.TransformerFactory;
import javax.xml.transform.Transformer;
...
    Result result = new DOMResult(newDoc);

    //Set the style sheet
    Source style = new StreamSource("logalyzer.xsl");

    //Create the Transformer
    TransformerFactory transFactory = TransformerFactory.newInstance();
    Transformer transformer = transFactory.newTransformer(style);

    } catch (Exception e) {
        System.out.println("Error: "+e.getMessage());
    }
...

```

First create the `TransformerFactory`; then use it to create the `Transformer` itself by feeding it the `Source` representing the style sheet.

The `Transformer` can now perform the transformation.

Execute the transformation

Performing the actual transformation is relatively straightforward. Simply execute the `transform()` method.

```
...
    //Set the source and result
    Source source = new DOMSource(dataDoc);
    Result result = new DOMResult(newDoc);

    //Set the style sheet
    Source style = new StreamSource("logalyzer.xsl");

    //Create the Transformer
    TransformerFactory transFactory = TransformerFactory.newInstance();
    Transformer transformer = transFactory.newTransformer(style);

    //Transform the Document
    transformer.transform(source, result);

    System.out.println(newDoc.getDocumentElement().toString());

    } catch (Exception e) {

```

```
System.out.println("Error: "+e.getMessage());
```

```
...
```

The `transform()` method takes the source and transforms it using the style sheet, then outputs it to the result. In this case, the result is another `Document` object, which can then be output to the screen.

Because the style sheet itself doesn't contain any templates to format the data, it appears in the result as a root element with one large text node.

Figure 4. One large root node

```
<newlog>nickc2453245as3q342001-09-09 19:27:03.000home.htmlrecs.htmlnickc2453245a
s3q342001-09-09 19:28:22.000recs.htmlbooks.htmlsaraha45645623123as2001-09-09 19:
29:33.000outside.htmlauthors.htmlsaraha45645623123as2001-09-09 19:29:53.000autho
rs.htmlrecs.htmlnickc2453245as3q342001-09-09 19:31:14.000books.htmlauthors.htmls
eanac535626411asd22001-09-09 19:33:43.000home.htmlrecs.htmlnickc62345235ser32200
1-09-09 19:34:58.000home.htmlrecs.htmlseanac535626411asd22001-09-09 19:50:22.000
recs.htmlauthors.html</newlog>
```

Creating the real style sheet solves this problem.

Section 5. Creating a simple XSLT style sheet

The basic style sheet

Now that the application is ready to begin transforming the data, it's time to start building the style sheet. To truly emulate the functionality afforded by stored procedures, the style sheet will need to directly manipulate the data. First you need to establish the basic data access.

The goal is to have an application that is completely generic when it comes to the data that is produced by the transformation. The application should be able to tell by the data itself where it goes in the database. The simplest way to do this is to create a document that organizes the data by table and by row. For example, consider this excerpt of the data destined for the `clickstream` table:

```
<?xml version="1.0"?>
<newlog>
  <clickstream>
    <row>
      <sessionid type="text">2453245as3q34</sessionid>
      <frompage type="text">home.html</frompage>
      <topage type="text">recs.html</topage>
    </row>
    <row>
      <sessionid type="text">2453245as3q34</sessionid>
      <frompage type="text">recs.html</frompage>
      <topage type="text">books.html</topage>
    </row>
  </clickstream>
</newlog>
```

```

    </row>
  </clickstream>
</newlog>

```

The first step is to send the original `visit` nodes to a template that ultimately formats them into rows, but for now just retrieves the data:

```

<?xml version="1.0"?>
<xsl:stylesheet
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform" version="1.0">

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

<xsl:template match="log">

  <xsl:apply-templates select="visit"/>

</xsl:template>

<xsl:template match="visit">
  <xsl:value-of select="sessionid" />
  <xsl:value-of select="referrer" />
  <xsl:value-of select="page" />
</xsl:template>

</xsl:stylesheet>

```

The root element, `log`, has its own template. Executed just once, this is where most of the coding will appear. For now, it simply sends only the `visit` nodes off to have other templates applied. The `visit` template then retrieves the values that belong in the `clickstream` table.

Figure 5. Unformatted clickstream data

```

<newlog>2453245as3q34home.htmlrecs.html2453245as3q34recs.htmlbooks.html456456231
23asoutside.htmlauthors.html45645623123asauthors.htmlrecs.html2453245as3q34books
.htmlauthors.html535626411asd2home.htmlrecs.html62345235ser32home.htmlrecs.html5
35626411asd2recs.htmlauthors.html</newlog>

```

This data must then be organized into rows.

Create rows of data

Organizing the data into rows involves creating a `row` element to hold it, then creating an element for each column of the table.

```

...
<xsl:template match="visit">
  <row>
    <sessionid><xsl:value-of select="sessionid" /></sessionid>
    <frompage><xsl:value-of select="referrer" /></frompage>
    <topage><xsl:value-of select="page" /></topage>
  </row>
</xsl:template>

```

```
</xsl:stylesheet>
```

When the application analyzes the data, it knows which data goes in which column by the name of the element.

Figure 6. Clickstream data formatted into rows

```
<newlog><row><sessionid>2453245as3q34</sessionid><frompage>home.html</frompage><
topage>recs.html</topage></row><row><sessionid>2453245as3q34</sessionid><frompage>
e>recs.html</frompage><topage>books.html</topage></row><row><sessionid>456456231
23as</sessionid><frompage>outside.html</frompage><topage>authors.html</topage></
row><row><sessionid>45645623123as</sessionid><frompage>authors.html</frompage><t
opage>recs.html</topage></row><row><sessionid>2453245as3q34</sessionid><frompage>
>books.html</frompage><topage>authors.html</topage></row><row><sessionid>5356264
11asd2</sessionid><frompage>home.html</frompage><topage>recs.html</topage></row>
<row><sessionid>62345235ser32</sessionid><frompage>home.html</frompage><topage>r
ecs.html</topage></row><row><sessionid>535626411asd2</sessionid><frompage>recs.h
tml</frompage><topage>authors.html</topage></row></newlog>
```

In order to know what `updateXXX()` method to use, however, the application still needs to know what type the data represents.

Add type information

Just as there are multiple `getXXX()` methods for pulling data out of the database based on its type, there are multiple `updateXXX()` methods for putting it back in again based on type. Because the application retrieves all data from the XML document as a `String`, information on what type to use must be explicitly added to the document.

```
...
<xsl:template match="visit">
  <row>
    <sessionid type="text"><xsl:value-of select="sessionid" /></sessionid>
    <frompage type="text"><xsl:value-of select="referrer" /></frompage>
    <topage type="text"><xsl:value-of select="page" /></topage>
  </row>
</xsl:template>
</xsl:stylesheet>
```

For simplicity's sake, this tutorial assumes only two basic types: text and number.

Figure 7. Two basic types: text and number

```
<newlog><row><sessionid type="text">2453245as3q34</sessionid><frompage type="text">home.html</frompage><topage type="text">recs.html</topage></row><row><sessionid type="text">2453245as3q34</sessionid><frompage type="text">recs.html</frompage><topage type="text">books.html</topage></row><row><sessionid type="text">45645623123as</sessionid><frompage type="text">outside.html</frompage><topage type="text">authors.html</topage></row><row><sessionid type="text">45645623123as</sessionid><frompage type="text">authors.html</frompage><topage type="text">recs.html</topage></row><row><sessionid type="text">2453245as3q34</sessionid><frompage type="text">books.html</frompage><topage type="text">authors.html</topage></row><row><sessionid type="text">535626411asd2</sessionid><frompage type="text">home.html</frompage><topage type="text">recs.html</topage></row><row><sessionid type="text">62345235ser32</sessionid><frompage type="text">home.html</frompage><topage type="text">recs.html</topage></row><row><sessionid type="text">535626411asd2</sessionid><frompage type="text">recs.html</frompage><topage type="text">authors.html</topage></row></newlog>
```

Now the application has all of the information it needs, except for the table to which the data should be added.

Add table information

If there were only one potential table involved, the application could be hard coded to enter the data into it. However, it makes much more sense to include this information in the new document along with the data itself.

While it certainly would be possible to add the table information to every row, it is much more efficient to have all of the rows contained within a single table element. To accomplish this, enclose the execution of the template affecting the `visit` elements with an element representing the `clickstream` table.

```
...
<xsl:template match="log">
  <clickstream>
    <xsl:apply-templates select="visit"/>
  </clickstream>
</xsl:template>

<xsl:template match="visit">
  <row>
    <sessionid type="text"><xsl:value-of select="sessionid" /></sessionid>
    <frompage type="text"><xsl:value-of select="referrer" /></frompage>
    <topage type="text"><xsl:value-of select="page" /></topage>
  </row>
</xsl:template>

</xsl:stylesheet>
```

Adding the `clickstream` tags around the `apply-templates` element ensures that they are only added to the new document once. The end result is a single `clickstream` element containing all the rows destined for that table.

Figure 8. Formatted clickstream data

```
<newlog><clickstream><row><sessionid type="text">2453245as3q34</sessionid><frompage type="text">home.html</frompage><topage type="text">recs.html</topage></row><row><sessionid type="text">2453245as3q34</sessionid><frompage type="text">recs.html</frompage><topage type="text">books.html</topage></row><row><sessionid type="text">45645623123as</sessionid><frompage type="text">outside.html</frompage><topage type="text">authors.html</topage></row><row><sessionid type="text">45645623123as</sessionid><frompage type="text">authors.html</frompage><topage type="text">recs.html</topage></row><row><sessionid type="text">2453245as3q34</sessionid><frompage type="text">books.html</frompage><topage type="text">authors.html</topage></row><row><sessionid type="text">535626411asd2</sessionid><frompage type="text">home.html</frompage><topage type="text">recs.html</topage></row><row><sessionid type="text">62345235ser32</sessionid><frompage type="text">home.html</frompage><topage type="text">recs.html</topage></row><row><sessionid type="text">535626411asd2</sessionid><frompage type="text">recs.html</frompage><topage type="text">authors.html</topage></row></clickstream></newlog>
```

This is certainly useful, but it could have been more easily accomplished with a single SQL statement. Now that the format has been established, however, the style sheet can be adapted to analyze and summarize the data in ways that would be difficult or impossible using SQL alone.

Section 6. Aggregating data with an XSLT style sheet

Get the basic visit information

Section 5, "Creating a simple XSLT style sheet," demonstrated the transformation of data using the typical method of creating templates for specific types of nodes. This section demonstrates a more direct means for dealing with data.

The `visitinfo` table contains information on each session. This information, which includes the start and stop time as well as basic information such as the session id and user information, may be spread across multiple rows. Handling the data directly makes it easier to combine this information into a single record.

Start by outputting just the basic visit information, including the table and type designations.

```
...
<xsl:template match="log">
<!--
<clickstream>
  <xsl:apply-templates select="visit"/>
</clickstream>
-->
      <visitinfo>
<row>
  <sessionid type="text">
    <xsl:value-of select="visit/sessionid" />
  </sessionid>
  <userid type="text">
    <xsl:value-of select="visit/userid" />
  </userid>
```

```

    </row>
  </visitinfo>

</xsl:template>

<xsl:template match="visit">
  ...

```

For clarity, comment out the `clickstream` information for now.

Figure 9. Basic visit information

```

<newlog><visitinfo><row><sessionid type="text">2453245as3q34</sessionid><userid
type="text">nickc</userid></row></visitinfo></newlog>

```

There are a number of things going on here. First, because this template is executed only once, only one value for the `sessionid` and `userid` appear in the final document.

Second, in this aggregating example the context node (the starting point) within this template is the `log` element, not the `visit` element as it was in the simpler example in Section 5. To retrieve the `sessionid` or `userid`, you must adjust the XPath expression that makes up the `select` attribute, unless you code the style sheet to loop explicitly through the visits.

Looping and sorting

One advantage of looping through the visits is the ability to sort the elements by a particular child. In this case, sort the visits by `sessionid`.

```

...
<xsl:template match="log">
  <!--
  <clickstream>
    <xsl:apply-templates select="visit"/>
  </clickstream>
  -->
  <visitinfo>
    <xsl:for-each select="visit">
      <xsl:sort select="sessionid" />

      <row>
        <sessionid type="text"><xsl:value-of select="sessionid" /></sessionid>
        <userid type="text"><xsl:value-of select="userid" /></userid>
      </row>

    </xsl:for-each>
  </visitinfo>

</xsl:template>
...

```

Compared to the example in Section 5, where the actual values are retrieved, the context node is now the `visit` element, so readjust the `select` attribute of the `value-of` elements.

The document now contains a row for each visit.

Figure 10. Basic visit information for all visits

```
<newlog><visitinfo><row><sessionid type="text">2453245as3q34</sessionid><userid
type="text">nickc</userid></row><row><sessionid type="text">2453245as3q34</sessi
onid><userid type="text">nickc</userid></row><row><sessionid type="text">2453245
as3q34</sessionid><userid type="text">nickc</userid></row><row><sessionid type="
text">45645623123as</sessionid><userid type="text">saraha</userid></row><row><se
ssionid type="text">45645623123as</sessionid><userid type="text">saraha</userid>
</row><row><sessionid type="text">535626411asd2</sessionid><userid type="text">s
eanac</userid></row><row><sessionid type="text">535626411asd2</sessionid><userid
type="text">seanac</userid></row><row><sessionid type="text">62345235ser32</ses
sionid><userid type="text">nickc</userid></row></visitinfo></newlog>
```

Selecting groups based on a value

Next, add the `pagecount` for each session.

Adding the `pagecount` value for each session should be easy. After all, there is a `count()` function built right into XPath. The `count()` function returns the number of nodes returned by an XPath expression passed to it as a parameter, so outputting the number of `visit` elements that contain a `sessionid` equal to, say, "5" could be accomplished with:

```
<xsl:value-of select="count(//visit[sessionid=5])" />
```

The expression

```
//visit[sessionid=5]
```

returns any `visit` node at any level of the document (hence the use of `//`) that satisfies the condition that its `sessionid` child has a value of "5". (See [Resources](#) for more information on XPath expressions.)

The difficulty lies in knowing what value to look for at any given time. After all,

```
//visit[sessionid=sessionid]
```

returns `true` for any `visit` with a `sessionid`. The solution is to use variables.

Variables in XPath

Fortunately, variables are not merely reserved for more "traditional" programming languages. Similar to XSLT parameters, XSLT variables are named values that are created and then referenced where needed as part of an XPath expression.

Variables, however, are set in a slightly different way. A parameter receives a value (which cannot be modified) either as part of the template itself or as part of the call to

the template. The value of a variable, on the other hand, can be directly set and modified from within a template.

```

...
<xsl:template match="log">
<!--
<clickstream>
  <xsl:apply-templates select="visit"/>
</clickstream>
-->
<visitinfo>
  <xsl:for-each select="visit">
    <xsl:sort select="sessionid" />
    <xsl:variable name="thisSession">
      <xsl:value-of select="sessionid" /></xsl:variable>
    <row>
      <sessionid type="text"><xsl:value-of select="sessionid" /></sessionid>
      <userid type="text"><xsl:value-of select="userid" /></userid>
      <pagecount type="number">
        <xsl:value-of select="count(//visit[sessionid=$thisSession])" />
      </pagecount>
    </row>
  </xsl:for-each>
</visitinfo>

</xsl:template>
...

```

Declare the variable and name it `thisSession`, assigning it a value of the `sessionid` for the current `visit`. The value of the variable is then substituted into the `count()` function (using the `$thisSession` syntax), returning only the number of visits that match the current `sessionid`.

Figure 11. Visit data with page counts

```

<newlog><visitinfo><row><sessionid type="text">2453245as3q34</sessionid><userid
type="text">nickc</userid><pagecount type="number">3</pagecount></row><row><sess
ionid type="text">2453245as3q34</sessionid><userid type="text">nickc</userid><pa
gecount type="number">3</pagecount></row><row><sessionid type="text">2453245as3q
34</sessionid><userid type="text">nickc</userid><pagecount type="number">3</page
count></row><row><sessionid type="text">45645623123as</sessionid><userid type="t
ext">saraha</userid><pagecount type="number">2</pagecount></row><row><sessionid
type="text">45645623123as</sessionid><userid type="text">saraha</userid><pagecou
nt type="number">2</pagecount></row><row><sessionid type="text">535626411asd2</s
essionid><userid type="text">seanac</userid><pagecount type="number">2</pagecoun
t></row><row><sessionid type="text">535626411asd2</sessionid><userid type="text"
>seanac</userid><pagecount type="number">2</pagecount></row><row><sessionid type
="text">62345235ser32</sessionid><userid type="text">nickc</userid><pagecount ty
pe="number">1</pagecount></row></visitinfo></newlog>

```

Select the first and last values

You can also use XPath expressions to determine the first and last values in a set.

Determining the start and stop times requires the ability to find the first and last value in a particular set of nodes. Accomplishing this involves the use of the `position()` function.

As in the above examples, the expression

```
//visit[sessionId=$thisSession]
```

returns a set of all `visit` nodes with a `sessionId` equal to `$thisSession`. The `position()` function allows the choice of a specific node. For example,

```
//visit[sessionId=$thisSession][position()=1]
```

and

```
//visit[sessionId=$thisSession][position()=3]
```

return just the first `visit` node that satisfies the `sessionId` criteria, and the third, respectively.

That makes it easy to select the first node. What about the last node? While it would be possible to set a variable using the `count()` function and so on, it's much more convenient to use the `last()` function, as in:

```
//visit[sessionId=$thisSession][position()=last()]
```

The `last()` function returns not the actual last node but its position, making it perfect for use here.

XPath syntax also allows the style sheet to omit the `position()` notation, as in:

```
//visit[sessionId=$thisSession][1]
```

and

```
//visit[sessionId=$thisSession][last()]
```

NOTE: The `position()` value is based on the position within the document, completely independent of any sorting done in a loop. If necessary, be sure to order the data appropriately when selecting it from the database in the first place.

Once the first or last `visit` is determined, retrieve the start and end times.

Add start and end times

Now you know the appropriate XPath expressions you need to select the first and last `visit` nodes for a session, but they still don't include the `visitdate` that shows when the sessions started and ended. Getting this information is simply a matter of selecting the proper `visitdate` child. The expression:

```
//visit[sessionId=$thisSession][1]
```

selects the `visit` element, so it follows that

```
//visit[sessionId=$thisSession][1]/visitdate
```

selects the `visitdate` child of that element. From there, adding the start and end times is straightforward:

```
...
<row>
  <sessionId type="text"><xsl:value-of select="sessionId" /></sessionId>
  <userid type="text"><xsl:value-of select="userid" /></userid>
  <pagecount type="number">
    <xsl:value-of select="count(//visit[sessionId=$thisSession])" />
  </pagecount>
  <start type="text">
    <xsl:value-of
      select="//visit[sessionId=$thisSession][1]/visitdate" />
  </start>
  <end type="text">
    <xsl:value-of
      select="//visit[sessionId=$thisSession][last()]/visitdate" />
  </end>
</row>
...
```

This completes the data for each visit.

Figure 12. Visit summary, complete with start and end times

```
<newlog><visitinfo><row><sessionId type="text">2453245as3q34</sessionId><userid
type="text">nickc</userid><pagecount type="number">3</pagecount><start type="tex
t">2001-09-09 19:27:03.000</start><end type="text">2001-09-09 19:31:14.000</end>
</row><row><sessionId type="text">2453245as3q34</sessionId><userid type="text">n
ickc</userid><pagecount type="number">3</pagecount><start type="text">2001-09-09
19:27:03.000</start><end type="text">2001-09-09 19:31:14.000</end></row><row><s
essionid type="text">2453245as3q34</sessionId><userid type="text">nickc</userid>
<pagecount type="number">3</pagecount><start type="text">2001-09-09 19:27:03.000
</start><end type="text">2001-09-09 19:31:14.000</end></row><row><sessionId type
="text">45645623123as</sessionId><userid type="text">saraha</userid><pagecount t
ype="number">2</pagecount><start type="text">2001-09-09 19:29:33.000</start><end
type="text">2001-09-09 19:29:53.000</end></row><row><sessionId type="text">4564
5623123as</sessionId><userid type="text">saraha</userid><pagecount type="number"
>2</pagecount><start type="text">2001-09-09 19:29:33.000</start><end type="text"
>2001-09-09 19:29:53.000</end></row><row><sessionId type="text">535626411asd2</s
essionid><userid type="text">seanac</userid><pagecount type="number">2</pagecou
nt><start type="text">2001-09-09 19:33:43.000</start><end type="text">2001-09-09
19:50:22.000</end></row><row><sessionId type="text">535626411asd2</sessionId><us
erid type="text">seanac</userid><pagecount type="number">2</pagecount><start typ
e="text">2001-09-09 19:33:43.000</start><end type="text">2001-09-09 19:50:22.000
</end></row><row><sessionId type="text">62345235ser32</sessionId><userid type="t
ext">nickc</userid><pagecount type="number">1</pagecount><start type="text">2001
-09-09 19:34:58.000</start><end type="text">2001-09-09 19:34:58.000</end></row>
</visitinfo></newlog>
```

Unfortunately, the style sheet adds this summary information for every visit. Because this is summary information, only unique sessions should show.

Save only unique sessions

To add only unique sessions to the new document, use this simple algorithm:

1. Save the current session id in a variable.
2. Check the current session id against the previous session id. If they match, do nothing. If not, add the session to the document.
3. Update the variable holding the last session id to match the current session id.
4. Repeat for all `visits`.

Build the logic into the style sheet as follows:

```

...
<xsl:template match="log">
  <!--
  <clickstream>
    <xsl:apply-templates select="visit"/>
  </clickstream>
  -->
  <visitinfo>
    <xsl:variable name="lastSession"></xsl:variable>
    <xsl:for-each select="visit">
      <xsl:sort select="sessionid" />
      <xsl:variable name="thisSession">
        <xsl:value-of select="sessionid" />
      </xsl:variable>

      <xsl:choose>
        <xsl:when test="$thisSession = $lastSession">
          <!-- This is not a new session. Do nothing. -->
        </xsl:when>
        <xsl:otherwise>
          <row>
            <sessionid type="text"><xsl:value-of select="sessionid" /></sessionid>
            <userid type="text"><xsl:value-of select="userid" /></userid>
            <pagecount type="number">
              <xsl:value-of select="count(//visit[sessionid=$thisSession])" />
            </pagecount>
            <start type="text">
              <xsl:value-of select="//visit[sessionid=$thisSession][1]/visitdate" />
            </start>
            <end type="text">
              <xsl:value-of select="//visit[sessionid=$thisSession][last()]/visitdate" />
            </end>
          </row>
        </xsl:otherwise>
      </xsl:choose>
      <xsl:variable name="lastSession">
        <xsl:value-of select="sessionid" />
      </xsl:variable>
    </xsl:for-each>
  </visitinfo>
</xsl:template>
...

```

For each `visit` update the `thisSession` variable before performing any other

work. Because the style sheet references the `lastSession` variable before updating it within the loop, it must be declared before the loop begins. The declaration contains no value, so that it can't possibly match the first session.

To determine whether to add the row to the document, use the `choose` element. Although similar to an `if` element, `choose` improves upon the functionality of `if` by allowing multiple conditions, similar to a traditional programming language's `if-then-else` or `select-case` statement.

In this example, if `thisSession` is the same as `lastSession`, the processor adds nothing to the document. If not, the processor adds the row normally.

Figure 13. Unique sessions only

```
<newlog><visitinfo><row><sessionid type="text">2453245as3q34</sessionid><userid
type="text">nickc</userid><pagecount type="number">3</pagecount><start type="tex
t">2001-09-09 19:27:03.000</start><end type="text">2001-09-09 19:31:14.000</end>
</row><row><sessionid type="text">45645623123as</sessionid><userid type="text">s
araha</userid><pagecount type="number">2</pagecount><start type="text">2001-09-0
9 19:29:33.000</start><end type="text">2001-09-09 19:29:53.000</end></row><row><
sessionid type="text">535626411asd2</sessionid><userid type="text">seanac</useri
d><pagecount type="number">2</pagecount><start type="text">2001-09-09 19:33:43.0
00</start><end type="text">2001-09-09 19:50:22.000</end></row><row><sessionid ty
pe="text">62345235ser32</sessionid><userid type="text">nickc</userid><pagecount
type="number">1</pagecount><start type="text">2001-09-09 19:34:58.000</start><en
d type="text">2001-09-09 19:34:58.000</end></row></visitinfo></newlog>
```

With the document now complete, it's time to add the data back into the database.

NOTE: Remember to uncomment the `clickstream` data in the style sheet.

Section 7. Inserting the data into the database

Create the Statement object

Just as the application utilized a `Statement` object to retrieve the data, it uses one to insert the data. This time, however, the creation involves additional parameters.

The `ResultSet` object used earlier was a read-only `ResultSet`, which was fine, but that won't do here. Because the application updates the `ResultSet` directly, and because it is the `Statement` that ultimately creates the `ResultSet`, the `Statement` needs to have parameters set when it is created.

```
...
//Transform the Document
transformer.transform(source, result);
```

```

//INSERT THE TRANSFORMED DATA INTO THE DATABASE

//Create the Statement as scrolling, insensitive to
//other changes, and updatable
Statement insertStatement =
    db.createStatement(ResultSet.TYPE_SCROLL_INSENSITIVE,
                       ResultSet.CONCUR_UPDATABLE);

} catch (Exception e) {
    System.out.println("Error: "+e.getMessage());
} finally {
    ...

```

Scroll-insensitive means that the pointer that represents the current row can be moved forward and backward, rather than just forward, as is the case with a default "forward-only" `ResultSet`. It also means that changes other users make to the database won't affect the `ResultSet`.

To say that the `ResultSet` is updatable means that columns can be updated directly using the `updateXXX()` methods.

Normalize the document

From here, the application should be able to take any data in the newly transformed `newDoc` and add it back into the appropriate table -- whether you have one table or a hundred -- and do it without any prior knowledge of what might be in the document.

Part of that process involves being absolutely certain of the format of the data, if not the data itself. While Xalan, the processor used in these examples, does not preserve whitespace by default, there is no guarantee that the style sheet has not been set to preserve it, changing a document from:

```

<newlog><visitinfo><row><sessionid
type="text">2453245as3q34</sessionid><userid
type="text">nickc</userid><pagecount
type="number">3</pagecount><start type="text">2001-09-09
19:27:03.000</start><end type="text">2001-09-09
19:31:14.000</end></row></visitinfo></newlog>

```

to:

```

<newlog>
  <visitinfo>
    <row>
      <sessionid type="text">2453245as3q34</sessionid>
      <userid type="text">nickc</userid>
      <pagecount type="number">3</pagecount>
      <start type="text">2001-09-09 19:27:03.000</start>
      <end type="text">2001-09-09 19:31:14.000</end>
    </row>
  </visitinfo>

```

```
</newlog>
```

While the data is certainly the same, the document is not. The DOM requires that each of those line feeds be considered a text node between element nodes, complicating the process of analyzing the data.

These extraneous nodes can be removed with a process called *normalization*, which combines adjacent text nodes into a single text node and removes any text nodes that contain nothing but whitespace.

Back in the TransformData application, normalize the document before doing any analysis on it.

```
...
    //INSERT THE TRANSFORMED DATA INTO THE DATABASE

    //Create the Statement as scrolling, insensitive to
    //other changes, and updatable
    Statement insertStatement =
        db.createStatement(ResultSet.TYPE_SCROLL_INSENSITIVE,
                           ResultSet.CONCUR_UPDATABLE);

    //Normalize the Document
    newDoc.normalize();

    } catch (Exception e) {
        System.out.println("Error: "+e.getMessage());
    } finally {
...

```

Loop through the tables

The first step in inserting the data is to determine which tables are affected. The data should be grouped within a single element for each table, though there is certainly nothing to require that. The only requirement is that each table element is a child of the root element, as shown below:

```
...
import org.w3c.dom.NodeList;
...
    newDoc.normalize();

    //Get table nodes
    NodeList tables = newDoc.getDocumentElement().getChildNodes();
    for (int i=0; i < tables.getLength(); i++) {
        //For each table, get the table element and name
        Element thisTable = (Element)tables.item(i);
        String tableName = thisTable.getNodeName();

        System.out.println(tableName);
    }

    } catch (Exception e) {
...

```

Retrieve a `NodeList` that consists of all of the children of the root element. If the style sheet built the document properly, this represents all of the table elements.

Figure 14. clickstream, visitinfo

```
clickstream
visitinfo
```

Loop through the rows

Once the table is determined, retrieve the rows for that table by looping through each one.

```
...
//Get table nodes
NodeList tables = newDoc.getDocumentElement().getChildNodes();
for (int i=0; i < tables.getLength(); i++) {
    //For each table, get the table element and name
    Element thisTable = (Element)tables.item(i);
    String tableName = thisTable.getNodeName();

    //Get the rows for this table
    NodeList rows = thisTable.getElementsByTagName("row");
    for (int j=0; j < rows.getLength(); j++) {
        Element thisRow = (Element)rows.item(j);
    }
}

} catch (Exception e) {
    System.out.println("Error: "+e.getMessage());
}
...
```

Because the document is normalized and the structure is predetermined, you could have accomplished this just as easily with `getChildNodes()`.

Create the ResultSet

At first glance, it would seem that the application should use an `INSERT` statement to add data to the database, but in actuality, it uses a `SELECT` statement.

```
...
for (int i=0; i < tables.getLength(); i++) {
    //For each table, get the table element and name
    Element thisTable = (Element)tables.item(i);
    String tableName = thisTable.getNodeName();

    //Create the ResultSet for this table
    ResultSet insertset =
        insertStatement.executeQuery("select * from "+tableName);

    //Get the rows for this table
    NodeList rows = thisTable.getElementsByTagName("row");
    for (int j=0; j < rows.getLength(); j++) {
        Element thisRow = (Element)rows.item(j);
    }
}
```

```
...
    //The table is done, so close the ResultSet
    insertset.close();
}
...
```

When the application executes the query, a copy of the data and its structure fills the `ResultSet` object, `insertset`. This data can be changed or deleted, and those modifications to the data set can then be sent back to the database.

After evaluating each row for a table, close the `ResultSet` object to prepare for the next table.

Create a new row

Adding new data to a `ResultSet` (as opposed to changing existing data) involves the use of the *insert row*. The insert row is a placeholder of sorts that allows an application to make changes to the `ResultSet` without disturbing existing data.

```
...
    //Create the ResultSet for this table
    ResultSet insertset =
        insertStatement.executeQuery("select * from "+tableName);

    //Get the rows for this table
    NodeList rows = thisTable.getElementsByTagName("row");
    for (int j=0; j < rows.getLength(); j++) {
        Element thisRow = (Element)rows.item(j);

        //Insert each row into the ResultSet
        insertset.moveToInsertRow();

        //Data will be added here

        //Send new row to the database
        insertset.insertRow();
    }

    //The table is done, so close the ResultSet
    insertset.close();
...

```

For each row of XML data, create a placeholder row by pointing the `ResultSet` to the insert row using `moveToInsertRow()`. After adding the data, send the new row to the database using the `insertRow()` method. Until `insertRow()` is called, the database remains untouched by the changes.

The data comes from the columns stored in each row.

Loop through the columns

After moving to the insert row, update each column with the appropriate data.

```

...
NodeList rows = thisTable.getElementsByTagName("row");
for (int j=0; j < rows.getLength(); j++) {
    Element thisRow = (Element)rows.item(j);

    //Insert each row into the ResultSet
    insertset.moveToInsertRow();

    //Get all columns for this row
    NodeList columns = thisRow.getChildNodes();
    for (int k=0; k < columns.getLength(); k++) {
        //For each column element, get the element name and value
        Element thisColumn = (Element)columns.item(k);
        String colName = thisColumn.getNodeName();
        String colValue = thisColumn.getFirstChild().getNodeValue();

        //Update this column in the ResultSet
        insertset.updateString(colName, colValue);
    }

    //Send new row to the database
    insertset.insertRow();
}
...

```

Get all the columns in each row, looping through each and retrieving the column name and value. Pass this information to the `updateString()` function. This causes the `ResultSet` to update the appropriate column with the appropriate value. When the `ResultSet` object sends the new row to the database, it includes these values.

(NOTE: Not all database-driver combinations support updatable `ResultSet`s. If this becomes a problem, use these values to build an `INSERT` statement and execute it instead.)

The application can also change the `updateXXX()` method used based on the type of data.

Insert the proper data type

Because the type information is specified on each column element, the application can use the appropriate `updateXXX()` method to update the database.

```

...
for (int k=0; k < columns.getLength(); k++) {
    //For each column element, get the element name and value
    Element thisColumn = (Element)columns.item(k);
    String colName = thisColumn.getNodeName();
    String colValue = thisColumn.getFirstChild().getNodeValue();
    String colType = thisColumn.getAttribute("type");

    //Update this column in the ResultSet
    if (colType.equals("text")) {
        insertset.updateString(colName, colValue);
    } else if (colType.equals("number")) {
        insertset.updateDouble(colName, new Double(colValue).doubleValue());
    }
}

```

```
    }  
  }  
  ...
```

Retrieve the type information from the `type` attribute of the column element and use it to determine which `updateXXX()` method to use. This example shows only two choices, `text` and `number`. In this case, `number` is treated as a `double` to avoid problems with integer values. Depending on the situation, more choices might be appropriate.

Section 8. XSLT data manipulation summary

Summary

This tutorial demonstrated an application that retrieves data from a database into a DOM `Document` object, then uses an XSL style sheet to transform the data into another DOM `Document`. The style sheet uses XSL variables, XPath expressions, and functions to simulate some of the operations that might be carried out by a database stored procedure. The application then inserts the newly transformed data inserted back into the database.

Resources

Learn

- For a basic grounding in XML, read through the [Introduction to XML](#) tutorial.
- For information on the Document Object Model, read the [Understanding DOM](#) tutorial.
- For an intro to programming XML, try Doug Tidwell's [XML programming in Java](#).
- Order [XML and Java from Scratch](#), by Nicholas Chase. It covers the use of XML and Java in general, and other data-centric views of XML, such as XML Query, and other uses for XML, such as SOAP. And besides, it's written by the author of this fine tutorial.
- Read G. Ken Holman's article for an [in-depth discussion of XSLT](#).
- Read [the XSL Recommendation](#) at the W3C.
- Read the [XPath Recommendation](#) at the W3C for a complete look at its capabilities.
- Read Benoit Marchal's tip on [using SAXTransformerFactory](#) to achieve greater flexibility and convenience with XSLT, or his feature on [SAX, the Power API](#).
- Get an overview of XSLT in [What kind of language is XSLT?](#) by Michael Kay.
- See [Sun Microsystems' information on JDBC](#).
- Review recent developerWorks articles on working with JDBC:
 - A July 2001 look at the latest JDBC: [What's new in JDBC 3.0](#) by Josh Heidebrecht
 - [An easy JDBC wrapper](#) by Greg Travis
- For information on using JDBC and XML, read the companion tutorials [Using JDBC to extract data into an XML document](#) and [Using JDBC to insert XML data into a database](#).
- Stay current with [developerWorks technical events and Webcasts](#).

Get products and technologies

- Download a [zip archive of the sample code](#) presented in this tutorial.
- Download [the Java 2 SDK](#), Standard Edition version 1.3.1.
- Download [JAXP 1.1](#), the Java APIs for XML Processing.
- Download [Xalan Java](#), an XSL Transformation engine supporting TrAX.

- Download [Xalan C++](#), an XSL Transformation engine supporting TrAX.
- Download [JDBC drivers](#) for your database.
- Download [JDataConnect](#), a JDBC driver that supports DB2, Oracle, Microsoft SQL Server, Access, FoxPro, Informix, Sybase, Ingres, dBase, Interbase, Pervasive and others.
- For an open-source database, download [MySQL](#) and [a MySQL JDBC driver](#).
- Build your next development project with [IBM trial software](#), available for download directly from developerWorks.

Discuss

- [Participate in the discussion forum for this content.](#)

About the author

Nicholas Chase

Nicholas Chase has been involved in Web site development for companies including Lucent Technologies, Sun Microsystems, Oracle Corporation, and the Tampa Bay Buccaneers. Nick has been a high school physics teacher, a low-level radioactive waste facility manager, an online science fiction magazine editor, a multimedia engineer, and an Oracle instructor. More recently, he was the Chief Technology Officer of Site Dynamics Interactive Communications in Clearwater, Fla. He is the author of three books on Web development, including the upcoming *XML Programming Primer Plus* (Sams). He loves to hear from readers and can be reached at nicholas@nicholaschase.com.