

XQJ - The JDBC for XQuery

Jonathan Bruce & Jonathan Robie

The XQuery language is designed specifically for XML programming and data integration, and programmers are more productive using XQuery for these tasks. However, many enterprise applications are built on the Java platform, and often require functionality not found in XQuery; for instance, many XML programs need to use the Web Services functionality of J2EE. On the Java platform, XML is accessed and manipulated as a DOM tree, a SAX stream, or as a StAX stream.

The XQuery API for Java, currently under development as JSR 225, lets programmers have the best of both worlds, using XQuery for XML programming and data integration, with full access to the J2SE and J2EE platforms. XQJ allows a Java program to connect to XML data sources, prepare and issue XQueries, and process the results as XML. This functionality is similar to that of JDBC® Java API for SQL, but the query language for XQJ is XQuery.

This article shows how XQJ is used to issue XQueries and obtain results. Next, it shows how XQJ can be used to query DOM trees, perform joins between XML and relational sources, obtain results using StAX, and issue prepared XQueries (similar to JDBC's prepared statements). Finally, we show four complete, working XQJ programs, including one that uses StAX to handle output. These programs are based on the Early Draft Review of JSR-225, released in May 2004. Code examples were tested with a pre-release version of DataDirect XQuery™, which implements XQuery and XQJ.

Querying Relational Data to Create XML

Although XQuery is a native XML query language, it is often used to query XML views of relational data to create XML results. Most Java programmers are familiar with the JDBC API. This example shows the similarities between JDBC and XQJ, using an XQJ connection to query a relational database and print out the results to the standard output.

SQL/XML Views of the Tables

The examples in this article are based on financial stock data, with a set of users and stock holdings for each user. Our implementation maps relational data using the SQL/XML mappings defined by SQL:2003. Here's the SQL/XML description of the USERS table (only a few rows are shown).

```
<USERS>
  <row>
    <USERID>Minollo</USERID>
    <FIRSTNAME>Carlo</FIRSTNAME>
    <LASTNAME>Innocenti</LASTNAME>
    <OTHERNAME/>
```

```

    <MEMBERSINCE>2004-07-16T00:00:00</MEMBERSINCE>
</row>
<row>
  <USERID>Jonathan</USERID>
  <FIRSTNAME>Jonathan</FIRSTNAME>
  <LASTNAME>Robie</LASTNAME>
  <OTHERNAME>William</OTHERNAME>
  <MEMBERSINCE>2004-04-03T00:00:00</MEMBERSINCE>
</row>
</USERS>

```

Here's the XML mapping of the HOLDINGS table, which contains the (fictional!) stock holdings for each user. Again, only a few rows are shown.

```

<HOLDINGS>
  <row>
    <USERID>Jonathan</USERID>
    <STOCKTICKER>AMZN</STOCKTICKER>
    <SHARES>3000.0000</SHARES>
  </row>
  <row>
    <USERID>Minollo</USERID>
    <STOCKTICKER>EBAY</STOCKTICKER>
    <SHARES>4000.0000</SHARES>
  </row>
  <row>
    <USERID>Jonathan</USERID>
    <STOCKTICKER>IBM</STOCKTICKER>
    <SHARES>2500.0000</SHARES>
  </row>
  <row>
    <USERID>Minollo</USERID>
    <STOCKTICKER>LU</STOCKTICKER>
    <SHARES>40000.0000</SHARES>
  </row>
  <!-- !!! SNIP !!! -->
</HOLDINGS>

```

Querying SQL/XML Views

For relational data, XQuery often is used as a reporting language, combining data to display on a web page or for use in a web message. Here's an XQuery that returns results that show the stocks held by each user.

```

for $u in collection('USERS')/USERS/row
return
  <user>
    <name>
      {
        $u/FIRSTNAME,
        $u/LASTNAME
      }
    </name>

```

```

{
  for $h in collection('HOLDINGS')/HOLDINGS/row
  where $h/USERID = $u/USERID
  return
    <stock>
      {
        $h/STOCKTICKER,
        $h/SHARES
      }
    </stock>
}
</user>

```

This query uses an XQuery function named `collection()` to address tables—the `collection()` function accepts a string and returns a sequence of nodes. In this sample query, we bind the variables `$u` and `$h` to the SQL/XML view of rows in the `USERS` and `HOLDINGS` tables.

Configuring Connections

The XQuery in the previous section uses the `collection()` function to access relational data, but XQuery has no notion of a relational database or a connection. To make this work, we use XQJ to specify the required database connections and associate these names with the tables.

A connection is created from a `DataSource`. Each vendor must provide some way of creating and parameterizing their own `XQDataSource` objects – the following two lines do this for DataDirect XQuery™, which uses a configuration file to describe connections.

```

DDXQDataSource dataSource = new DDXQDataSource();
dataSource.setConfigFile(configFile);

```

The `getConnection()` method can now be invoked to return a connection to the data source:

```

XQConnection connection = dataSource.getConnection();

```

The next step is to create an `XQExpression` object, which can execute an XQuery expression and return a sequence of results. An `XQConnection` can create an `XQExpression`.

```

XQExpression xqExpression = connection.createExpression();

FileReader fileReader = new FileReader("xquerySourceFile.xq");
XQSequence xqSequence =

```

```
xqExpression.executeQuery(fileReader);
```

For this example, we place the query in a separate file named `xquerySourceFile.xq` (another reasonable way would be to put the query in a `String` object).

Now that the query results are in a sequence, we can serialize this sequence using the `getSequenceAsString()` method.

```
System.out.println(xqSequence.getSequenceAsString());
```

Here's an excerpt from the result of this query.

NB: This is formatted for readability- whitespace has been modified

```
<user>
  <name>
    <FIRSTNAME>Jonathan</FIRSTNAME>
    <LASTNAME>Robie</LASTNAME>
  </name>
  <stock>
    <STOCKTICKER>AMZN</STOCKTICKER>
    <SHARES>3000.0000</SHARES>
  </stock>
  <stock>
    <STOCKTICKER>EBAY</STOCKTICKER>
    <SHARES>4000.0000</SHARES>
  </stock>
  <stock>
    <STOCKTICKER>IBM</STOCKTICKER>
    <SHARES>2500.0000</SHARES>
  </stock>
  <!-- !!! SNIP !!! -->
</user>
```

Querying Data from XML Files or Java XML APIs

In the previous query, our data was in a relational database. Now let's query an XML file.

Suppose we want to query the users from a file named 'holdings.xml', which looks like this.

```
<holdings>
  <holding>
    <USERID>Jonathan</USERID>
    <STOCKTICKER>PRGS</STOCKTICKER>
    <SHARES>23.0000</SHARES>
  </holding>
  <holding>
    <USERID>Minollo</USERID>
    <STOCKTICKER>PRGS</STOCKTICKER>
```

```
    <SHARES>4000000.0000</SHARES>
  </holding>
</holdings>
```

Here's an XQuery expression that returns the holdings for Jonathan.

```
doc("holdings.xml")//holding[USERID="Jonathan"]
```

Obviously, Jonathan is not the only person whose holdings might interest us. If we write our XQuery with an external variable, the Java program can specify the name of the user before it executes the query. If we also use an external variable to represent the document, the Java program can pass any document to the query at runtime:

```
declare variable $u as xs:string external;
declare variable $d as document-node() external;

$d//holding[USERID=$u]
```

Now let's write the Java code to create a DOM tree and bind it to the variable \$d. We start by creating a DOM tree.

```
// Establish DOM tree and load document into memory.
DocumentBuilderFactory factory =
    DocumentBuilderFactory.newInstance();
factory.setNamespaceAware(true);
DocumentBuilder parser = factory.newDocumentBuilder();

FileReader fileReader = new
    FileReader("holdings.xml");

Document document = parser.parse(fileReader);
```

Once we create a DOM tree, we can use XQJ to bind it to a variable and query it. First, let's create an expression object, and then bind the document to the variable \$d for this expression.

```
XQConnection connection = dataSource.getConnection();
XQExpression xqExpression = connection.createExpression();

xqExpression.bindNode(new QName("d"), document);
xqExpression.bindVariable(new QName("u"), "Jonathan");
```

Now we can execute the expression and output the result.

```
FileReader fileReader = new FileReader("xquerySourceFile.xq");

XQSequence xqSequence =
    xqExpression.executeQuery(fileReader);
System.out.println(xqSequence.getSequenceAsString());
```

Joining XML and Relational Sources

We have already explored how XQJ enables XQuery to operate on relational and XML file data stores. Now let's leverage that functionality to query both at the same time. We will use an incoming Web Service request that provides parameters for a query, and then query a database to create the Web Service response.

Consider the following Web Service request:

```
<request>
  <performance>
    <UserId>Jonathan</UserId>
    <start>2003-01-01</start>
    <end>2003-01-01</end>
  </performance>
</request>
```

The above example contains only the SOAP message payload – we left out the envelope to simplify the example, but the XQuery would be changed only slightly if it were in an envelope. This request asks for performance data on a user's portfolio within a specific date range.

Let's compose an XQuery that uses the parameters from the request to create a performance report, which will report the performance of each stock held by each user during the given range.

```
let $request := doc('request.xml')/request
for $user in $request/performance
return
  <portfolio UserID="{ $user/UserId}">
    { $request }
    {
      for $h in collection('HOLDINGS')/row
      where $h/USERID eq $user
      return
        <stock>
          {
            $h/STOCKTICKER,
            $h/SHARES
          }
        </stock>
    }
  </portfolio>
```

In keeping with our previous examples, we instantiate a XQDataSource instance and establish a connection to the data source.

```
XQConnection connection = dataSource.getConnection();
```

Again, we create an `XQExpression` object, which can execute the XQuery expression and return a sequence of results.

```
FileReader fileReader = new FileReader("xquerySourceFile.xq");

XQExpression xqExpression = connection.createExpression();
XQSequence xqSequence =
    xqExpression.executeQuery(fileReader);
```

With our query results in a sequence, we can again serialize this sequence using the `getSequenceAsString()` method.

```
System.out.println(xqSequence.getSequenceAsString());
```

The result looks like this (whitespace has been modified for readability):

```
<portfolio UserID="Jonathan">
  <request>
    <performance>
      <UserId>Jonathan</UserId>
      <start>2003-01-01</start>
      <end>2004-06-01</end>
    </performance>
  </request>
  <stock>
    <STOCKTICKER>PRGS</STOCKTICKER>
    <SHARES>23.0000</SHARES>
  </stock>
  <stock>
    <STOCKTICKER>AMZN</STOCKTICKER>
    <SHARES>3000.0000</SHARES>
  </stock>
  <stock>
    <STOCKTICKER>EBAY</STOCKTICKER>
    <SHARES>4000.0000</SHARES>
  </stock>
  <stock>
    <STOCKTICKER>IBM</STOCKTICKER>
    <SHARES>2500.0000</SHARES>
  </stock>
</portfolio>
```

Retrieving Results with Java XML APIs

Often, applications need to retrieve XQuery results as DOM, SAX, or StAX. The `XQSequence` interface has methods to support each of these APIs.

XQJ provides two ways by which the result of an XQuery can be processed. We have already used the `XQSequence` level access, which permits the result as a direct mapping of the XQuery sequence. Within a `XQSequence`, zero or more `XQItem` objects represent each component in an XQuery sequence. Note that instantiating each item in a `XQItem`

object is expensive because it requires the creation of multiple objects and should be used judiciously.

The `XQItemAccessor` interface allows an `XQuery` result sequence to be divided and processed item by item. This is particularly useful when processing large result sequences.

Importantly, both the `XQSequence` and `XQItemAccessor` result accessors interfaces allow processing using the SAX and StAX APIs.

For example, if you want to retrieve your results as a StAX stream rather than a string, you simply replace the last line from the previous section with this:

```
XMLStreamReader reader = xqSequence.getSequenceAsStream();
```

You can use this like any other StAX stream reader. For example, the following function reads one event at a time and prints the type of the event together with the associated names. Only a small excerpt is shown below—the complete program is [here](#).

```
private static void formatOutput(XMLStreamReader reader) throws
XMLStreamException {
    for (int event = reader.next();
        even != XMLStreamConstants.END_DOCUMENT;
        event = reader.next()) {
        switch (event) {
            case XMLStreamConstants.START_ELEMENT:
                System.out.println("Start tag: ");
                printNames(reader);
                break;
        }
    }
}
/* !!! SNIP !!! */
```

Preparing XQuery statements

When an `XQuery` is executed, the query generally is parsed and optimized before it is run. To avoid incurring this overhead each time the `XQuery` is used, `XQJ` allows queries to be prepared once and executed multiple times. Here's the code to create a prepared query—only the last line differs from the code used to create a query in our first example.

```
dataSource = new DDXQDataSource();
dataSource.setConfigFile(configFile);
connection = dataSource.getConnection();

FileReader fileReader = new
    FileReader("xquerySourceFile.xq");
```

```
preparedExpression = connection.prepareExpression(
    fileReader);
```

Once the query is prepared, it is executed using the `executeQuery()` call.

```
XQSequence xqSequence = preparedExpression.executeQuery();
System.out.println(xqSequence.getSequenceAsString());
```

Of course, queries often take parameters, and these parameters may need to be changed between executions. For example, we might want to prepare a query that selects items that match a particular value and change that value each time the query is executed. Suppose we want to use a query that returns the stock holdings for a given user. The user changes each time this XQuery is run.

```
declare variable $1 as xs:string external;

collection('HOLDINGS')/HOLDINGS[USERID=$1]
```

The value of `$1` is set using XQJ. Let's run this twice, each time for different users.

```
preparedExpression.bindString(new QName("1"), "Jonathan");
xqSequence = preparedExpression.executeQuery();
System.out.println("\n\nHoldings for Jonathan:\n\n");
System.out.println(xqSequence.getSequenceAsString());

preparedExpression.bindString(new QName("1"), "Minollo");
xqSequence = preparedExpression.executeQuery();
System.out.println("\n\nHoldings for Minollo:\n\n");
System.out.println(xqSequence.getSequenceAsString());
```

A complete program that prepares a query and executes with these values can be found [here](#).

Program Listings

XQJExecute

This example shows how to execute an XQuery contained in a file. It is called with two parameters: `:` the name of the file that specifies the connections for the query and the name of the file that contains the query. An optional, third parameter specifies the name of the file to which the output is written.

```
package com.ddtek.xqj.examples;

import java.io.*;

import javax.xml.xquery.XQConnection;
import javax.xml.xquery.XQException;
import javax.xml.xquery.XQExpression;
```

```

import javax.xml.xquery.XQSequence;
import com.ddtek.xquery.xqj.mediator.DDXQDataSource;

/**
 * This application is intended to be run as command-line Java,
 * the usage is:
 * java XQJExecute [configFile] [queryFile] <resultoutputfile>
 * There is an optional third argument that allows you
 * to specify an output file for the resulting XML:
 * java XQJExecute
 *
 */
public class XQJExecute {

    public static void main(String[] args) throws Exception {
        DDXQDataSource dataSource = null;
        XQConnection connection = null;
        XQExpression xqExpression = null;
        XQSequence xqSequence = null;
        String configFile = "";
        String xquerySourceFile = "";

        if (args.length < 2) {
            String usage = "Usage:\n\n java XQJExecute [configfile]
[queryfile]\n\n" +
                "          [configfile]      Source Configuration File\n" +
                "          [queryfile]      XQuery file" + "\n\n" +
                "or:\n" +
                "java XQJExecute [configfile] [queryfile]\n\n" +
                "          [configfile]      Source Configuration File\n" +
                "          [queryfile]      XQuery file" +
                "          [resultoutputfile]      result output file";

            System.out.println(usage);
            System.out.println("Exiting XQJExecute...");
            System.exit(1);
        }
        configFile = args[0];
        xquerySourceFile = args[1];

        try {
            /**
             * Prepare connection
             */
            dataSource = new DDXQDataSource();
            dataSource.setConfigFile(configFile);
            connection = dataSource.getConnection();

            /**
             * Create an XQExpression instance based on the connection
             */
            xqExpression = connection.createExpression();

            /**
             * Get an XQuery result sequence from XQuery
             * expression the specified file.
             */

```



```

import com.ddtek.xquery.xqj.mediator.DDXQDataSource;

/**
 * The purpose of this class is to serve as an example of using the XQJ
API to
 * query a stream from StAX (Streaming API for XML).
 *
 * This application is intended to be run as command-line Java,
 * the usage is:
 *
 * java StAXExample
 */
public class StAXExample {

    public static void main(String[] args) {

        DDXQDataSource dataSource = null;
        XQConnection connection = null;
        XQExpression xqExpression = null;
        XQSequence xqSequence = null;
        XMLStreamReader reader = null;
        String configFile = "";
        String xquerySourceFile = "";

        if (args.length != 2) {
            String usage = "Usage:\n\n java StAXExample [configfile]
[queryfile]\n\n" +
                "          [configfile]    Source Configuration File\n" +
                "          [queryfile]      XQuery file";

            System.out.println(usage);
            System.out.println("Exiting StAXExample...");
            System.exit(1);
        }
        configFile = args[0];
        xquerySourceFile = args[1];

        try {
            dataSource = new DDXQDataSource();
            dataSource.setConfigFile(configFile);
            connection = dataSource.getConnection();

            xqExpression = connection.createExpression();

            FileReader fileReader = new
                FileReader(xquerySourceFile);

            xqSequence =
                xqExpression.executeQuery(fileReader);

            while (xqSequence.next()) {
                reader = xqSequence.getSequenceAsStream();
                formatOutput(reader);
                System.out.println("");
            }
        }
        catch (Exception ex) {

```

```

        ex.printStackTrace();
    } finally {
        try {
            if (connection != null){
                reader.close();
                connection.close();
            }
        }
        catch (XQException xqEx) {
            xqEx.printStackTrace();
        }
    }
}

```

```

private static void formatOutput(XMLStreamReader reader) throws
XMLStreamException {

```

```

    for (int event = reader.next();
        event != XMLStreamConstants.END_DOCUMENT;
        event = reader.next()) {

        switch (event) {
            case XMLStreamConstants.START_ELEMENT:
                System.out.println("Start tag: ");
                printNames(reader);
                break;

            case XMLStreamConstants.END_ELEMENT:
                System.out.println("End tag");
                printNames(reader);
                break;

            case XMLStreamConstants.CHARACTERS:
                System.out.println("Text");
                printChars(reader);
                break;

            case XMLStreamConstants.CDATA:
                System.out.println("CDATA Section\n");
                break;

            case XMLStreamConstants.COMMENT:
                System.out.println("Comment");
                break;

            case XMLStreamConstants.DTD:
                System.out.println("Document type declaration\n");
                break;

            case XMLStreamConstants.ENTITY_REFERENCE:
                System.out.println("Entity Reference\n");
                break;

            case XMLStreamConstants.SPACE:
                System.out.println("Ignorable white space\n");

```

```

        break;

        case XMLStreamConstants.PROCESSING_INSTRUCTION:
            System.out.println("Processing Instruction\n");
            break;

        default:
            System.out.println("This can not happen.");
        }
    }
}

private static void printChars(XMLStreamReader aReader){
    String textString = aReader.getText();
    if(textString != null){
        System.out.println("\tValue:" + textString);
    }
}

private static void printNames(XMLStreamReader reader) {
    String localName = reader.getLocalName();
    String prefix = reader.getPrefix();
    String uri = reader.getNamespaceURI();

    if (localName != null){
        System.out.println("\tLocal name: " + localName);
    }
    if (prefix != null) {
        System.out.println("\tPrefix: " + prefix);
    }
    if (uri != null) {
        System.out.println("\tNamespace URI: " + uri);
    }
    System.out.println();
}
}
}

```

PreparedQuery

This example shows how to prepare a query and execute it with different variable bindings. It is called with two parameters: the name of the file that specifies the connections for the query and the name of the file that contains the query.

```

Package com.ddtek.xqj.examples;

import java.io.*;
import javax.xml.namespace.QName;

/* Import XQJ classes
*/
import javax.xml.xquery.XQConnection;
import javax.xml.xquery.XQException;
import javax.xml.xquery.XQPreparedExpression;
import javax.xml.xquery.XQResultSequence;
import javax.xml.xquery.XQSequence;

```

```

import com.ddtek.xquery.xqj.mediator.DDXQDataSource;

/**
 * The purpose of this class is to serve as an example of using the XQJ
API to
 * execute a prepared query.
 *
 * This application is intended to be run as command-line Java,
 * the usage is:
 *
 * java PreparedQuery
 */
public class PreparedQuery {

    public static void main(String[] args) throws Exception {
        DDXQDataSource dataSource = null;
        XQConnection connection = null;
        XQPreparedExpression preparedExpression = null;
        XQSequence xqSequence = null;
        String configFile = "";
        String xquerySourceFile = "";

        if (args.length != 2) {
            String usage = "Usage:\n\n java PreparedQuery [configfile]
"
                [queryfile]\n\n" +
"
                [configfile]    Source Configuration File\n" +
"
                [queryfile]    XQuery file";

            System.out.println(usage);
            System.out.println("Exiting PreparedQuery...");
            return;
        }
        configFile = args[0];
        xquerySourceFile = args[1];
        try {

            /* Get a connection, prepare the query
            */
            dataSource = new DDXQDataSource();
            dataSource.setConfigFile(configFile);
            connection = dataSource.getConnection();

            FileReader fileReader = new
                FileReader(xquerySourceFile);

            preparedExpression =
                connection.prepareExpression(fileReader);

            /* Bind variable $l to 'Jonathan' and execute
            */
            preparedExpression.bindString(new QName("l"), "Jonathan");
            xqSequence = preparedExpression.executeQuery();
            System.out.println("\n\nHoldings for Jonathan:\n\n");
            System.out.println(xqSequence.getSequenceAsString());
        }
    }
}

```



```

import javax.xml.xquery.XQException;
import javax.xml.xquery.XQExpression;
import javax.xml.xquery.XQSequence;
import com.ddtek.xquery.xqj.mediator.DDXQDataSource;

/**
 * The purpose of this class is to provide an example of how to
 * use the XQJ API to query XML with a variable bound to a DOM.
 *
 * To use this application, execute the following from the command
line:
 *
 * java BindDomToVariable
 *
 */
public class BindDomToVariable {

    /**
     * Execute BindDomToVariable as a java console application. See the
'args' param
     * for startup arguments.
     * @param args - a list of three strings that represent the path to
the
     * config file, the path to an XQuery source file, and the path to
a well-formed
     * XML data file.
     */
    public static void main(String[] args) {

        DDXQDataSource dataSource = null;
        XQConnection connection = null;
        XQExpression xqExpression = null;
        XQSequence xqSequence = null;
        String configFile = "";
        String xquerySourceFile = "";
        String xmlDataFile = "";

        //we need all three args to run this example...
        if (args.length != 3) {
            System.out.println("Usage: java BindDomToVariableconfigfile
xquery
                xmldatafile");
            System.out.println("Exiting BindDomToVariable...");
            return;
        }
        configFile = args[0];
        xquerySourceFile = args[1];
        xmlDataFile = args[2];

        try {
            /* Ask the DocumentBuilderFactory class for an instance
            * to help us make a DocumentBuilder parser.
            */
            DocumentBuilderFactory factory
            = DocumentBuilderFactory.newInstance();

            /* Establish the aspect of namespace awareness

```

```

        * before the parser is created.
        */
factory.setNamespaceAware(true);

/* Ask the DocumentBuilderFactory instance for a new
 * document builder parser
 */
DocumentBuilder parser = factory.newDocumentBuilder();

/* Create a DOM tree from the xml data file.
 */
Document document = parser.parse(xmlDataFile);

/* Create a connection, then an expression
 */
dataSource = new DDXQDataSource();
dataSource.setConfigFile(configFile);
connection = dataSource.getConnection();
xqExpression = connection.createExpression();

/* Bind the document to the external variable $d
 */
xqExpression.bindNode(new QName("d"), document);

/* Execute the query - print the result
 */

    FileReader fileReader = new
        FileReader(xquerySourceFile);

    xqSequence =
        xqExpression.executeQuery( fileReader);

    System.out.println(xqSequence.getSequenceAsString());

}
catch (SAXException e) {
    System.out.println(xmlDataFile + " is not well-formed." +
e);
}
catch (IOException e) {
    System.out.println(
        "Due to an IOException, the parser could not read "
+ xmlDataFile +
        ". " + e );
}
catch (FactoryConfigurationError e) {
    System.out.println("Could not locate a factory class" + e);
}
catch (ParserConfigurationException e) {
    System.out.println("Could not locate a JAXP parser" + e);
}
catch (TransformerConfigurationException e) {

```

