

The SOA Magazine

Feature Article



SOA and the Importance of XQuery

by Carlo Innocenti

Published: September 3, 2007 (SOA Magazine Issue X: September 2007, Copyright © 2007)

[Download this article as a PDF document.](#)

Abstract: The fundamental goal of SOA is to facilitate business-level software modularity and allow for rapid reuse of application and data logic, enabling the enterprise to be responsive and agile. In reality, most SOA projects are focused on existing application logic, and therefore fail to consider the key implications of integrating data. The common difficulties associated with accessing a large number of heterogeneous data sources represents a level of design complexity that, when ignored, will only get worse over time.

The XQuery language has become an integral technology for connecting heterogeneous data sources within service-oriented solutions. It's a natural fit for the typical XML data interchange format used in most SOA implementations and it provides powerful, cross-repository data access features without compromising scalability or performance. This article explores how the XQuery feature-set can be leveraged in support of SOA.

Introduction: SOA and Data Management

SOA has been around for a number of years earning acceptance as a solid approach for systems management – one that allows for the broad reuse of existing software assets, provides a sound architectural model for the federation of disparate IT systems, and supports the automation of abstract business processes via a range of programming paradigms.

But how does data management fit in? Guidelines for service-oriented data access and management techniques are sparse. Those that are available have typically been formulated by SOA experts, not data management experts. As a result, different understandings of the same problems turn into a constant source of confusion and headaches.

XQuery and Data Management Interfaces

Most SOA data management solutions currently in use rely on traditional, well defined APIs including ODBC, JDBC, OCI, ADO.NET, OLE DB, and others. All of these APIs share similar concepts, but most of them fail to capture the differences between traditional data access architecture characteristics (tightly coupled, complex state machine, connection based, and relational model driven) and characteristics associated with SOA (loosely coupled, stateless, message-centric, and typically XML-based data interchange).

XQuery, the XQuery for Java API, and Web services provide a great way to bridge the data disparity with service-orientation. XQuery still exposes an interface against which users submit queries and from which they process results, but at the same time it is easily exposed via a Web service, and it further provides abstraction between the consumer of the data and the physical details about how the data is stored. XQuery is designed to give language implementations the possibility to execute queries against heterogeneous data sources, interpreting (but not necessarily materializing) all of them as XML.

XQuery is based on an XML data model, providing smooth integration in today's Web service-centric infrastructures.

When you consider the service-orientation design paradigm, it becomes evident that XQuery features are very much in alignment with the goals of service-oriented computing.

Some XQuery Examples

Consider a relatively simple use case: an IT organization needs to build a few applications that provide a self-service interface for employees to track their time-off and to find information about the location of employees on the company's campus. These types of applications are expected to grow over time, and the strategy of the IT department is to establish a set of data services that can be reused and augmented as requirements dictate.

Personal employee data and time-off information are available in a relational database server, and information about offices (and map diagrams) is available in a variety of XML documents and Web services.

XQuery can be used to easily and efficiently access the data sources involved, and to provide responses in the form of XML messages. An over simplified XQuery retrieving time-off details for the month could be designed as follows:

```
declare variable $employeeEmail as xs:string external;
declare variable $month as xs:string external;

let $employeeData := collection("emp.dbo.personnel")/personnel[emp_email = $employeeEmail]
return
  <user name="{ $employeeData/emp_name }">
    <timeoff month="{ $month }"> {
      for $monthData in collection("emp.dbo.timeoff")/timeoff[emp_id = $employeeData/emp_id and month_id = $month]
      return
        <event type="{ $monthData/type }" hours="{ $monthData/hours }"/>
    } </timeoff>
  </user>
```

Notice how, from the XQuery's point of view, the RDBMS data source looks like a regular XML structure. The join between the two relational tables involved in the query is resolved by an XPath expression (`collection("emp.dbo.timeoff")/timeoff[emp_id = $employeeData/emp_id]`).

The XQuery author doesn't need to worry about the underlying database being used to store the "personnel" and "timeoff" tables, or about which particular SQL dialect is supported. And, should the underlying physical storage of the information change (for example, if "timeoff" data was stored as an XML document on the file system, or maybe as an XML type in the database), the impact would be limited to very simple tweaks in the XQuery statement exposing the data service (for example, changing `collection('emp.dbo.timeoff')/timeoff` into `doc('timeoff.xml')//timeoff`).

An additional benefit of XQuery is that building the XML result is a natural operation. XML fragments are typically valid XQuery expressions, which means you don't need to do anything more than type XML markup code to make sure XQuery outputs the structure you need. For example, you can modify the previous example so that instead of returning a generic XML structure, it returns an HTML document that can be rendered by any Web browser without further processing, as follows:

```
declare variable $employeeEmail as xs:string external;
declare variable $month as xs:string external;

<html> {
let $employeeData := collection("emp.dbo.personnel")/personnel[emp_email = $employeeEmail]
return
  <body>Timeoff for { $employeeData/emp_name } in { $month }:
    <table style="border-style: outset;border-width: 2pt;width:100%"> {
      for $monthData in collection("emp.dbo.timeoff")/timeoff[emp_id = $employeeData/emp_id and month_id = $month]
      return
        <tr>
          <td style="border-style: inset;border-width: 1pt">
```

```

        {$monthData/type}
      </td>
      <td style="border-style: inset;border-width: 1pt">
        {$monthData/hours}
      </td>
    </tr>
  } </table>
</body>
} </html>

```

The two external variables at the top of this XQuery example are used to provide parameters to run the XQuery; a simple Web service implementation could automatically expose the XQuery using the external variables as arguments for the Web service operation:

```

<wsdl:types>
  <xs:schema elementFormDefault="qualified" targetNamespace="myNamespace">
    <xs:element name="getEmpTimeoff">
      <xs:complexType>
        <xs:sequence>
          <xs:element name="employeeEmail" type="xs:string"/>
          <xs:element name="month" type="xs:string"/>
        </xs:sequence>
      </xs:complexType>
    </xs:element>
    ...
  </xs:schema>
</wsdl:types>

```

Clearly, that's just one of the many options for exposing an XQuery as a Web service operation. The reason it's easier doing this with XQuery rather than with other programming languages is that XQuery and WSDL share the same underlying data model (XML), and that makes the task of creating a WSDL definition for XQuery a natural operation with no data type mapping required.

Exposing an XQuery that merges RDBMS and native XML data is not any different, thanks to the fact that XQuery abstracts low level data implementation details. Suppose that the IT organization has also developed a Web service that is able to create a map highlighting a specific office via its ID number. XQuery can be used not only to expose a data service, but also to access the existing Web services.

For example:

```

declare variable $employeeEmail as xs:string external;

let $employeeData := collection("emp.dbo.personnel")/personnel[emp_email = $employeeEmail]
let $officeData := doc("offices.xml")//officeData[emp_id = $employeeData/emp_id]
return
  <user name="{ $employeeData/emp_name }">
    <office number="{ $officeData/id }"> {
      ddtek:wscall(
        <ddtek:location
          address="http://myServer/XQuery"
          soapaction="getOfficeMap"/>,
        <getOfficeMap>
          <officeID>{ $officeData/id }</officeID>
        </getOfficeMap>
      )
    } </office>
  </user>

```

This apparently simple XQuery statement is actually integrating three very different data sources: an RDBMS, an XML file (“offices.xml”) and a Web service result (“getOfficeMap”). Again, notice how the fact that XML, as the XQuery data model, makes such tasks simple to code and maintain without the need to worry about data type mappings.

These days, the most common alternative to using XQuery is to write a combination of Java and SQL code accessing JDBC and XML APIs. Trying that approach even in relatively simple cases like the ones just described makes it clear as to why XQuery is preferable. The amount of code that needs to be written is smaller while the readability and maintainability of this code is higher. The Java/SQL/XML approach inevitably runs into problems dealing with multiple data models (SQL and XML) through a language relying on yet another data model (Java). Even just creating an XML result can be a painful experience with this combination.

XQuery Performance and Scalability Considerations

When you talk about a language that is able to access heterogeneous data sources (RDBMS, flat files, EDI, XML, Web services), and possibly large data sets (such as over-sized XML documents), the first obvious question asked by a data management expert is: “What about performance and scalability?” The W3C XQuery language specification doesn’t broach such implementation details, as you might imagine. However, if you survey the current marketplace you’ll notice that performance and scalability are exactly what continue to distinguish XQuery products.

Some products only work with in-memory data, while others are able to translate relevant portions of the query into SQL and take advantage of RDBMS performance tuning. Still others are able to work in a pure streaming fashion and even implement sophisticated techniques such as XML projection and in-memory indexing. Some XQuery products offer automatic Web service integration (both in terms of consuming and providing Web services), while others support this only within the context of a specific application server (or provide no support at all).

The XQuery API for Java (XQJ) is a specification (JSR 225) currently under public review as part of the Java Community Process (JCP) [REF-1]. You can think about XQJ in the XQuery world as the equivalent to JDBC for SQL. One of the major benefits of XQJ is that it’s very well suited for implementing streaming-based processing, assuming the underlying XQuery engine supports this capability.

XQJ further provides the option to bind data (like XML documents) to external variables and the initial context item through a variety of interfaces, including I/O streams, SAX, StAX or DOM. Similarly, XQuery results can be consumed from XQJ through the same rich set of interfaces. StAX in particular (the XML streaming “pull” API) is very well suited for exposing streaming based processing, and even allows multiple processing steps to be chained together (often referred to as “XML pipelines”) for “pure” streaming. The ability to process streaming across multiple XQueries is extremely powerful and allows for the creation of sophisticated and complex data services that can be exposed as services without compromising performance or scalability.

Despite the performance and scalability benefits, attention must still be paid to an XQuery data access infrastructure in order to fully realize the potential of data services exposed as part of an SOA. This is because SOA implementations tend to amplify the deficiencies of traditional data access middleware, putting more congestion into a system that is already bottlenecked at the network management layer.

Many service-oriented solutions, for example, use SOAP (XML-based) messaging, which can result in increased network traffic and verbose network shipments. Furthermore, the emphasis that service-orientation places on service reuse can lead to ever-increasing usage demands for data services. To address this design consideration, it is important to plan for scalable data access, not only using highly scalable data access middleware, but also by relying on data architects to design (or at least review) data access code.

Conclusion

XQuery provides an easy, efficient way to create data services, exposing XML-based, abstracted interfaces to a variety of heterogeneous data sources. The availability of robust, scalable, and performant implementations makes XQuery one of the best options in the data management area in support of SOA and service-oriented computing. The same level of modularity, reuse, and business agility that IT organizations have already started to appreciate within the business service layer of established service inventories can now be realized within the data service layer as well.

References

[REF-1] JSR 225: XQuery API for Java™ (XQJ) (<http://www.jcp.org/en/jsr/detail?id=225>)

THE PRENTICE HALL SERVICE-ORIENTED COMPUTING SERIES FROM THOMAS ERL



[Home](#) [Past Issues](#) [Contributors](#) [What is SOA?](#) [SOA Books](#) [Legal](#) [RSS](#)

Copyright © 2006-2007 SOA Systems Inc.
All Rights Reserved