XQuery as a Tool for Liquid Data Integration— Some Design Considerations

Juan Andrade, Vadim Draluk, Dick Tsur

Abstract

This paper discusses the use of XQuery as a tool for liquid data integration within the BEA WebLogic Integration product. A liquid data integration facility would elevate the data integration task from the domain of application development to a domain of its own, in which it can be dealt with in a more cost effective way. The technical problem that this paper addresses is that of the application of XQuery to data integration in the context of web services and WLI application views. These views represent services in the form of XML input and output documents. The problem is the generation of XQuery forms that are consistent with this input output behavior. The paper proposes a hybrid, framework-based approach in which XQuery is used to perform translations from the inputs to outputs at the various stages of the process and a scripting language is used to embed these translations and to marshal the workflow. The approach is applied to two typical data integration scenarios: the 360° Customer View and to the so-called "banking problem" in which the integrated view represents the union of a set of information sources.

1. WLI and data sources

The WebLogic Application Integration (WLAI) subsystem is a component of the BEA WebLogic Integration (WLI) product. WLAI provides an adapter framework based on J2EE-JCA with extensions to allow bi-directional connections with another EIS (Enterprise Information System). In addition, it extends the standard client API with XML features based on an input/output document model.

Given that our data integration effort is centered on XML, three levels of interfaces within the existing WLI architecture need to be considered as viable candidates for interfacing with an underlying information source. All provide access to legacy and application data in XML format.

1. XCCI is the lowest-level interface. It is an extension of the standard CCI interface for JCA, providing XML-formatted input and output to the adapters. It also extends the standard DOM interface by supporting XPath functionality, thus making it convenient to use for navigating output documents.
2. Application Views are built on top of XCCI, adding
   - Design-time data vs. runtime data features
   - Asynchronous capabilities not specified by JCA 1.0, but expected in JCA 2.0.
3. Web Services, on top of application views, that provide SOAP and WSDL sugarcoating without any semantic enrichment.

Semantically input and output XML documents do not change their meaning between these levels of interface. Since the Web Service synchronous interface paradigm is sufficiently well known and understood, we will refer to all interfaces in question as Web Service Interfaces (WSIs).

Although this paper concentrates on an adaptor-based environment, we note that the same approach is valid for the integration of information coming from web services, or for that matter, of any other data source such as a system library, or Java class featuring the input/output document model.

2. To XQuery or not to XQuery?

When looking for expressive means to define integrated views over several independent Web Service Interfaced (WSI) information sources, one would naturally like to consider XQuery as a viable candidate. Indeed, it provides rich functionality for the integration of information originating in several XML documents, plus a very large number of features of a big-time (albeit not very mature) programming language.

A second glance though reveals some serious difficulties in using XQuery even in expressing requests to a single WSI source. These stem from an innate impedance mismatch between two paradigms. XQuery is designed to operate on *actual XML documents*; it takes as input a set of XML documents and composes an output or a result document using its own repertoire of selection, projection, join and other capabilities. WSI sources, on the other hand, take input, via an application view, of XML input documents and map these into output documents. In other words, WSI sources are characterized by their *functional behavior*, which is expressed by a *given map*. We may not know the details of this map nor can we control it. In order to express this behavior as an equivalent XQuery, we would have to:

- Define a virtual integrated document (VID) format upon which XQuery could operate. Intuitively, the VID would have to cover all possible output documents that can be produced by a WSI.
- Describe an XQuery composition of the VID that would produce as a result a valid input document to the WSI. A valid input document would be one that is acceptable by the application view of the WSI.
- Introduce a capability for an XQuery engine to *invoke* WSI sources rather than just read XML documents
- Add a parameterization mechanism to XQuery.

It's easy to overlook the necessity of the VID and related paraphernalia when considering certain trivial examples of data access, in which input data to be passed to a WSI source is as basic as a single string or number e.g., a call on *getCustomer( ),* in which a single key parameter is passed to denote a specific customer. In this case the WSI input document would be something simple as in:

```
<customer id = "12345" />
```

and the corresponding output document would be something like:

```
<customer>
   <name> … </name>
   <address> … </address>
   <orders>
      <order> … </order>
      …
   </orders>
   …
</customer>
```

An XQuery producing a result corresponding to the input document from the output documents would be a straightforward matter. But since WSI sources are not under our control, and their input document semantics can get fairly rich e.g., their structure may depend on the values of certain input parameters, the VIDs and their mappings to input documents can become complex and unintuitive. In section 5.3 we will consider these issues in greater detail when dealing with a specific example.

In order to support WSI source invocation we will have to develop and support our own XQuery engine with all the requisite extensions, which is neither an easy, nor a cheap thing to do.

VIDs can be introduced in several ways with different styles. We will consider different approaches of this kind, and see what the difficulties are in using them. Some will require introducing a separate VID for each source. In this case the approach is strongly procedural, since the extended XQuery used to define the integrated data view will handle the sources independently, typically in a sequence of nested queries, thus prescribing an execution plan. The second type is based upon a single global VID for the entire data view. It is more declarative, and requires a more sophisticated XQuery engine, capable of generating execution plans according to the usage information and meta-data available. We will not consider it in this paper due to its overwhelming complexity.

There are several per-source VID approaches. Consider a generic VID format as follows:

```
<Source>
      <RequestResponse>
            <Input>
            …
            </Input>
            <Output>
            …
            </Output>
      </RequestResponse>
      …
</Source>
```

This virtual document has a potentially infinite number of `<RequestResponse>` elements, each featuring `<Input>` and `<Output>` elements, the former being sort of a key for the

latter. Now, to access `<Output>`, we simply describe the corresponding `<Input>` in the WHERE clause. We will call this the explicit method.

Another multi-VID approach has VIDs defined to be "unfiltered" renditions of source output documents (SODs). Typically VIDs will have the same structure as SODs, but with many more elements and attributes in them. It leads to less artificial query syntax than that of the previous method, but requires potentially very complex mappings between VIDs and source input documents (SIDs) to be defined by the data designer as part of its meta-data. This will be hereafter referred to as the implicit method.

At the top level, the integrated view over the constituent information sources, presented to the application developer, consists of a set of parameters in the form of a main input document (MID) and the result produced by XQuery. The result is returned in form of the main output document (MOD). This mechanism allows for the parameterization of XQuery.

Both XQuery-based approaches along with parameterization will be considered in more detail later in this document when we apply them to a specific use case.

3. A Framework-based approach

When proposing a data integration solution, we have to keep following requirements in mind:

- Data integration is possible today within WLI, but it is very procedural and not simple for the user. Any proposed solution has to be specialized for data integration, and must be therefore simpler to use
- Solutions must as much as possible utilize technologies existing within BEA, or available to us from the open source market
- Solutions have to take into account both existing and emerging standards
- Access to integrated data views should not be very different from access of individual data sources

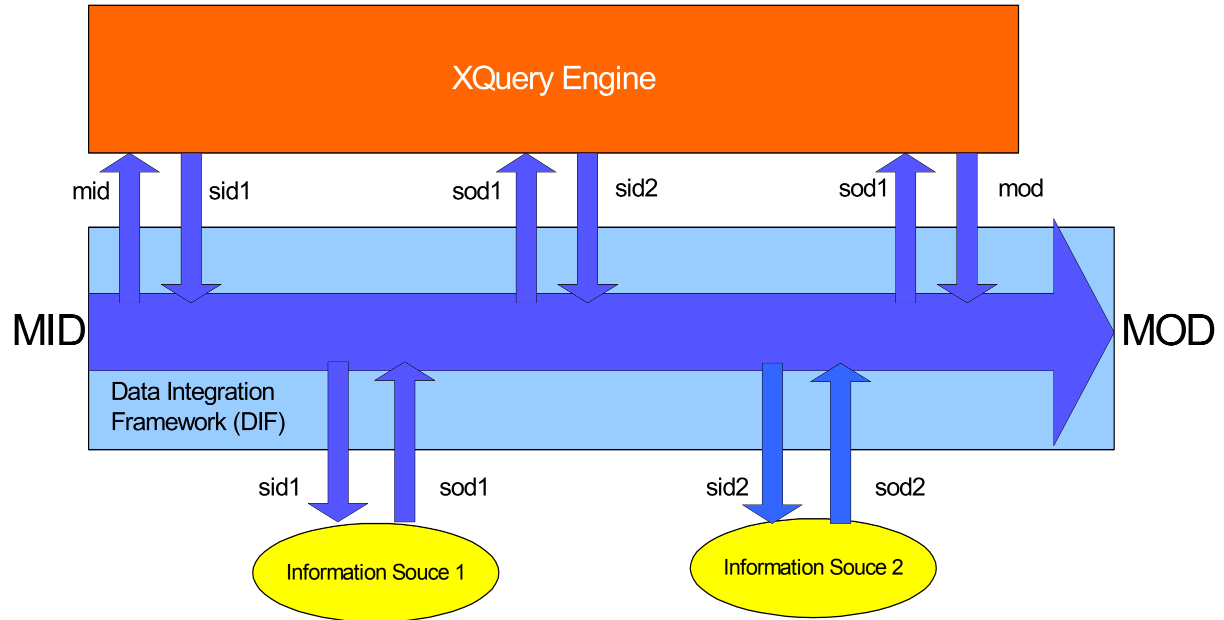
The last requirement in this list encourages us to use WSI as the paradigm for integrated data access. It means that the data integration framework (DIF) will take an XML document (the MID introduced earlier, to distinguish it from input documents used in individual source access, or SIDs) as its input and produce XML documents as output. Thus DIF functionality will have to include:

1. Generating SIDs, based on the MID and SODs of other adapters
2. Orchestrating invocation of individual adapters
3. Producing the MOD from the SODs of the participating sources.

We believe that an XML transformation language like XQuery can be easily and efficiently used to perform tasks (1) and (3), while a more flexible, extensible and

procedural language is better suited for (2). Such a combination can be achieved by introducing a framework approach, under which the procedural language is used at the core of framework, which invokes both WSI sources and a third-party XQuery engine to retrieve data and transform documents.

The DIF architecture is illustrated by the following diagram:



Data Integration Framework: work flow

There are several options as far as the implementation of the procedural part is concerned, including generating it in Java. We believe that ECMAScript from Crossgain is best positioned to address this task because it:

- Is very good at manipulating XML
- Has built-in capabilities of invoking XQuery engines

This approach can also be useful in the future when more complex update issues are addressed. It might even in some respect be simpler than the data retrieval case, since there will be no complex SODs involved, so all SIDs will have to be derived from a single MID (The orchestration has the potential of becoming more complex in this case).

This approach is also modular, so, if only a particular data source changes, only one XQuery dealing with this source will be affected.

4. *Enabling tools*

The framework by itself, while facilitating decomposition of one large task into a number of simpler ones, does not necessarily make the user's life significantly simple. Henceforth GUI tools are critical for attaining our goal of simple non-procedural data integration. Such tools will help:

- To build framework scripts orchestrating WSI sources and adapter invocations
- To define XQueries necessary to generate SIDs form MID and SODs

While not getting into any details of the GUI design, we will outline the steps a tool will have to support to generate framework scripts (visual XQuery tools are more generic, so that, at least initially, we would be able to leverage some third-party products.

Step 1: identifying the data sources. While we have agreed on WSI nature of all sources, their exact nature has to be specified by the developer. The information includes invocation paradigm (Web Service, XCCI interface, Java class method, system library function etc) and source parameters (URL for a Web service; Java class and method names, as well as constructor and method parameters; library and function names).

Step 2: pointing to the primary data source. We have to tell the tool which source will be used first. This source has to derive all its input information from the MID.

Step 3 (repeated multiple times): defining transformations of MID and SODs of sources processed earlier in the process to produce the source's SID. This includes defining an XPath expression for the node set in the input SOD which the framework will iterate over, passing it to the XQuery engine.

Step 4: transforming SODs into the MOD.

5. *Use patterns and examples*

In order to keep our examples simple, we will initially concentrate on only two data integration scenarios. We will call the first **360°** pattern (for the 360 degree view of, say, a Customer, an internal BEA use case), the second scenario will be called a **union** pattern (a use case from the banking industry).

The **360°** pattern is using one adapter as a primary data source, with additional data filled in from other data sources. When it is applied, the dependencies between input and output documents can be described as follows:

MID → SID1
SID1 → SOD1
MID, SOD1 → SOD2
MID, SOD1 → SOD3
…
SOD1, SOD2, SOD3, …, → MOD

The **union** pattern uses several data sources independently, constructing the MOD more or less as union of individual SODs. The use cased mentioned above produces an integrated view of the customer's accounts at different banks or branches:

MID → SID1
SID1 → SOD1
MID → SID2
SID2 → SOD2
…
SOD1 + SOD2 + … → MOD

5.1 A detailed 360º pattern example

This example deals with a 360º view of customer companies. It has following data sources:

Source 1: basic customer information in SAP, including company name, a numeric 7-digit id, address etc. Its SID has the format of

```
<Customer match = "…" compare = "…">
…
</Customer>,
```

with company name as its text, a `match` attribute specifying the type of matching (exact, case-insensitive), and a `compare` attribute describing comparison semantics (prefix, suffix, contains, contains-word). Its SOD format is

```
<Customers>
      <Customer name= "…" … />
      ...
</Customers>,
```

with all of the customer data returned as attributes.

Source 2: information on products sold to this customer in PeopleSoft. The SID format for this source is:

```
<customerProduct customerId = "… ">
      <field> … </field>
      …

</customerProduct>
```

Customer is uniquely identified by the 7-digit id. The request must specify explicitly the attributes ("field" element) of the product to be returned. The SOD format is:

```
<Products>
      <Product>
            <Field1> … <Field1>
            …
```

```
        </Product>
        …
</Products>
```

Source 3: database of CRs filed by customers in Clarify, with following SID:

```
<cr customer="…" severity="…"/>
```

Customers are identified by their id in form of "nnn-mmmm", derived from the id used by the previous sources. The SOD format is:

```
<CRs>
        <cr customer= "…" severity = "…" status = " " …>
                <message by = "… " date ="…">
                …
                </message>
        </cr>
</CRs>
```

The integrated view has to have a reasonably uniform MID, containing all information needed for SIDs, and a consistent MOD, hiding the individual SOD style differences as much as possible. Let's define the MID as follows:

```
<Customer name= "…" match= "…" compare= "…" crSeverity= "…"
  productData= "…"/>,
```

where `productData` is a list of values. We define the MOD to look like

```
<Customers>
        <Customer name= "…" id= "…" address= "…" >
                <Products>
                        <Product name= "…" … />
                        …
                </Products>
                <CRs>
                        <cr customer= "…" severity= "…" status= "" …>
                             <message by= "…" date= "…">…</message>
                             …
                        </cr>
                      …
                </CRs>
        </Customer>
        …
</Customers>
```

Let's designate the SAP source as the primary one in our **360°** schema, which means that the execution plan will be based on the iteration over elements returned from this source. We will also have to state explicitly the XPath expression for the iteration base, which is /Customers.

The first XQuery will convert MID into SID1:

```
LET $mid: = document(mid.xml)/Customer
RETURN
      <Customer match={$x/@match} compare={$x/@compare}>
          {$mid/@name}
      </Customer>
```

The second XQuery maps MID and SOD1 into SID2:

```
LET $mid := document(mid.xml)/Customer
LET $id := document(sod1.xml)/Customer/@id
RETURN
{
      FOR $field IN $mid/@productData
      RETURN
        <customerProduct customerId={$id}>
           {$field}
        </customerProduct>
}
```

The third query produces SID3 from MID and SOD1:

```
LET $mid := document(mid.xml)/Customer
LET $id := document(sod1.xml)/Customer/@id
RETURN
  <cr customer={xf:concat(xf:substr($id, 1, 3), "-",xf:substr($id,4)}
     severity={$mid/@severity}/>
```

Finally, the fourth and last XQuery converts SOD1, SOD2 and SOD3 into MOD:

```
LET $sod1 := document(sod1.xml)/Customer
LET $sod3 := document(sod3.xml)/CRs
RETURN
 <Customer name={$sod1/@name} id={$sod1/@id} address={$sod1/@address} >
   <Products>
   {
       FOR $sod2 IN document(sod2.xml)/Products/Product/*
       RETURN
          <Product>
            attribute {name($sod2)} {$sod2/data()}
          </Product>
   }
   </Products>
   {$sod3}
</Customer>
```

Please note that XQueries are used to contain several kind of difficulties:

- Filtering out input data relevant for one sources but not others
- Extracting input data for some sources from the output data of the others
- Dealing with individual data field format inconsistencies
- Addressing structural differences in data representation (elements vs. attributes)

All these issues can be addressed by using XSLT, or some other XML transformation structures are similar between the documents involved, these queries can be generated automatically using the GUI tool. This is even more likely to happen with the simpler **union** pattern, to be considered next.

5.2 A detailed union pattern example

This example deals with two merged banks, each with separate customer account data base. We would like to create a global view of all accounts across the two banks belonging to the same customer.

Assume that customers are identified by their SSNs, and both WSI data sources accept input documents in form of:

```
<SSN> … </SSN>
```

Naturally we keep the same format for the MID as well, so that there's no need for XQueries transforming MID into SIDs. The only one we will need is the query transforming SOD1 and SOD2 into a more uniform MOD.

Let's introduce SOD1 as having following format:

```
<Bank>
   <Branch>
      <Id> … </Id>
      <Location> … </Location >
      <Accounts>
         <Account>
           <Name> … </Name>
           <Type> … </Type>
           <Balance> … </Balance>
         </Account>
        …
     </Accounts>
     …
  </Branch>
  …
</Bank>
```

SOD2 has a simpler format, which we will also use for MOD:

```
<Accounts>
    <Account branch= "…" type= "…" name= "…" balance= "…"/>
    …
</Accounts>
```

So XQuery converting SOD1 and SOD2 into MOD will be:

```
<Accounts> {
      FOR  $sod2 IN document(sod2.xml)/Accounts/Account
      RETURN  $sod2;
```

```
      FOR $sod1 IN document(sod1.xml)/Bank/Branch/Accounts/Account
      RETURN
            <Account branch={$sod1/.../.../Id} type={$sod1/.../Type}
            name={$sod1/.../Name} balance={$sod1/.../Balance} />
   }
</Accounts>
```

5.3 360° pattern and global XQuery approach

First we address the explicit approach. Our example will retrieve all customer organization whose name contains the word "bank". For these customers we are interested in products they have (product name, price paid, date of contract), and all CRs with severity 3 and up.

```
<Customers>
{
    FOR $x IN source(source1.xml)/Source/RequestResponse
    LET $in1 := $x/Input/Customer
    LET $out1 := $x/Output/Customer
    WHERE  $in1[@match = "case-insensitive"][@compare="contains-word"]
    RETURN
      <Customer name={$out1/@name} id={$out1/@id}
      address={$out1/@address} >
        <Products>
           {
             FOR $y IN source(source2.xml)/Source/RequestResponse
             LET $in2 := $y/Input/customerProduct
             LET $out2 := $y/Output/Product
             WHERE $in2/@customerId = $out1/@id AND
                    $in2[field = "name"][field ="price"][field ="date"]
             RETURN
             <Product>
               {
                    FOR y1 IN $out2/*
                    RETURN
                       {attribute {name($y1)} {$y1/data()}
               }
             </Product>
           }
        </Products>
        {
          FOR $z IN source(source3.xml)/Source/RequesrResponse
          LET $in3 := $z/Input/cr
          LET $out3 := $z/Output
          LET $id = xf:concat(xf:substr($out1/@id, 1, 3), "-",
               xf:substr($out1/@id,4))
          WHERE $in3[@customer = {$id}] AND $in3[@severity = 3]
        }
    </Customer>
}
</Customers>
```

As we can see from the example, this approach has following characteristics:

- It is not significantly more non-procedural than the framework approach, since the execution plan is explicitly defined by nesting of FLWR operators
- It requires to introduce a proprietary function "source" into XQuery
- Parametrization is difficult to achieve in some cases, like in one dealing with product characteristics
- It preserves the letter of XQuery more than its spirit, by imposing special semantics on filtering of the Input components of the VID
- It does not encapsulate peculiarities and inconsistencies in formats of different SIDs

Now let's consider the implicit approach. Without actually constructing an integrated view, we will build queries for individual sources in order to see the difficulties presented by this solution.

We will begin with the source 3 as the simplest one. Its VID has exactly the same structure as SOD, but has instances of "cr" elements for all CRs in the data source. Thus the query will have to filter out the data from VID to arrive at the requested SOD:

```
 FOR $x IN vid(source3.xml)/CRs/cr
WHERE $z/@customer="123-4567" and  $z/@severity="3"
RETURN $z
```

We have to provide the XQuery engine with meta-data indicating that exact values of "customer" and "severity" attributes specified in the WHERE clause have to be generated into eponymous attributes of the SID3. While we will not concentrate on syntax of such a declaration, we can state that this a relatively straightforward task.

Now let's do the same for source 2. This case is different because it involves both filtering and projection from VID to SOD2:

```
FOR $y IN vid(source2.xml)/Products/customerProduct
WHERE $y/@customerId="1234567"
RETURN
      <Product>
         {$y/name}
         {$y/price}
         {$y/date}
      </Product>
```

As we can see, metadata describing SID2 in terms of VID2 will be significantly more complex than one in the previous case: in addition to filtering in customerId, we will have to tell the engine that sub-elements of "customerProduct" encountered in RETURN will have to be generated into "field" sub-elements of "Product" in SID2.
Finally consider SID1. Here we run into problems of a different kind, related to use of XQuery built-in functions. First, it is impossible to express the exact functionality of filtering supported in source 1 in terms of the standard XQuery: there's no function for case-insensitive comparison, nor is there a notion of "contains-word". Even if we

simplify the task significantly, and, say, request all customers whose organization name starts or ends with “Bank”, we will still face a task of providing meta-data mapping a particular built-in function applied to a particular attribute to some component in SID1.

So it looks like this approach is manageable when SODs can be derived from VIDs by filtering based on exact values of attributes and elements, as it’s the case with source 3. It appears to become unwieldy whenever relationship between SODs and VIDs involves projections or more complex type of filtering.

Now consider MID-based parametrization for the implicit case. Take source 2 as an interesting example (1 and 3 are simple from this point of view), based on MID as described in the framework example, only with customer name replaced by an ID. MID-parametrized query will be:

```
LET $mid := document(mid.xml)/Customer
FOR $y IN vid(source2.xml)/Products/customerProduct
   WHERE $y/@customerId=$mid/@id
   RETURN
      <Product>
     {
         FOR  $y1  IN $y/*
         WHERE SOME $y2 IN $mid/@productData/* SATISFIES $y1 = $y2
         RETURN
           {attribute {name($y1)} {$y1/data()}
      </Product>
```

6. Conclusions

The proposed data integration framework approach appears to be an easy and efficient way for BEA to strengthen its presence in the data integration market. Its runtime components are available right now within BEA or as open source software. Minimally credible tools can be implemented with very limited effort. A small team of 2-3 persons can develop a viable prototype in 2-3 month and we recommend that BEA adopt this approach

While a framework based approach is a good start, it is by no means the most sophisticated solution as it heavily relies on the tool user to produce the desired integration result. We believe that more can be achieved by adopting a top-down, declarative approach towards data integration. In this approach, certain extensions to XQuery in the form of parameterized view definitions will enable to view query results as XML documents. Other extension features will map XQuery views on WLAI application views and lastly, features will be included in XQuery to invoke specific underlying information sources and to obtain their outputs as views. We believe that these extensions will allow for a query specification that can be *compiled* into an execution plan, not unlike the one that can be manually produced using the frame-based approach. Additionally, a compiler would be able to optimize the plan, which is nearly impossible

using the frame-based approach. To adopt this declarative approach requires more research, in particular as far as the mapping capabilities are concerned. We also believe that XQuery as a language will evolve, and some of the contemplated extensions will be included as part of the standard. We recommend continuing the research into XQuery language extensions and once the framework-based system is built, to build a BEA proprietary XQuery processor that would include these.