Experiences with XQuery Processing for Data and Service Federation

Michael Blow, Vinayak Borkar, Michael Carey, Daniel Engovatov,
Dmitry Lychagin, Panagiotis Reveliotis, Joshua Spiegel, Till Westmann [*]

Abstract

In this paper, we describe our experiences in building and evolving an XQuery engine with a focus on data and service federation use cases. The engine that we discuss is a core component of the BEA AquaLogic Data Services Platform product (recently re-released under the name Oracle Data Service Integrator). This XQuery engine was designed to provide efficient query and update capabilities over various classes of enterprise data sources, serving as the data access layer in a service-oriented architecture (SOA). The goal of this paper is to give an architectural overview of the engine, discussing some of the key implementation techniques that were employed as well as several XQuery language extensions that were introduced to address common data and service integration problems and challenges.

1 Introduction

The advent of relational databases in the 1970's ushered in a productive era in which developers of data-centric applications could work more efficiently than ever before. Instead of writing procedural programs to access and manipulate data, declarative queries could accomplish the same tasks. With physical schemas hidden by the relational model, developers spent less time worrying about performance, as physical changes no longer implied program changes. Simplified views could be defined and used with confidence because rewrite optimizations ensured that queries over views are just as performant as queries over base data. The relational revolution was a huge success and led to many commercial database products. Almost every enterprise application developed in the past 15-20 years uses a relational database for persistence, and all enterprises run major aspects of their operations on relationally-based packaged applications like SAP, Oracle Financials, PeopleSoft, Siebel, Clarify, and SalesForce.com.

Today, developers of data-centric enterprise applications face a new challenge. There are many different relational database systems (Oracle, DB2, SQL Server, MySQL, ...) and a given enterprise is likely to have a number of different relational databases within its corporate walls; information about key business entities like customers or employees commonly exists in multiple databases. Also, while most "corporate jewels" are stored relationally, they are often relationally inaccessible because the applications enforce the business rules

[*]Work was done while authors were at BEA Systems, Inc. Current affiliations: Michael Blow, Dmitry Lychagin, and Joshua Spiegel - Oracle Corporation. Vinayak Borkar - Black Titan Software, LLC. Michael Carey - University of California, Irvine. Daniel Engovatov - Stanford University. Panagiotis Reveliotis - Composite Software, Inc. Till Westmann - SAP

and control the business logic. Meaningful access must thus come through the “front door” via application APIs. Because of this, developers of new applications face a major integration challenge: bits and pieces of a given business entity will live in a mix of relational databases, packaged applications, files, legacy mainframe systems, and/or home-grown applications. New “composite” applications need to somehow be created from these parts.

Composite application development is the goal of the service-oriented architecture (SOA) movement [2]. XML-based Web services are one piece of the puzzle, providing physical normalization for intra- and inter-enterprise service invocations and data exchange. Web service orchestration languages [3] are another piece of the puzzle, but are procedural by nature. At BEA, we felt that a declarative approach was needed for creating *data services* [8] for use in composite applications. We chose to ride the wave created by Web services and the associated XML standards, using XML, XML Schema, and XQuery to knit together a standards-based foundation for data services development [9, 10]. The BEA AquaLogic Data Services Platform (ALDSP), introduced in mid-2005, has XQuery in the leading role as the language for accessing and composing information from sources including relational databases, Web services, packaged applications, and files. This paper reviews the ALDSP XQuery implementation and some of the key challenges that we addressed during its development.

2 Background

The types of data models employed by enterprise data sources span from semi-structured to fully-structured, from flat to hierarchical to graph-based, and from untyped to loosely-typed to strictly-typed. For example, relational databases contain structured, flat data while XML documents contain semi-structured, hierarchical data. Some backend sources may require input or provide output in the form of flat, structured data (e.g. stored procedures), or hierarchical, semi-structured data (e.g. Web services). Given the vast heterogeneity found in enterprise data models, a data federation approach should support access to as many different kinds of data sources as possible and employ a rigorous yet versatile data model and type system.

In our approach, the XML data model [11], XML Schema [4, 5], and the XQuery language [13] serve as a solid foundation for integrating diverse data sources. XML provides a flexible way of describing many different types of data representations, while XML Schema offers a standard facility for the formal definition of both simple and complex, hierarchical types. The combination of XML Schema types and the concept of sequence type, specified by the XQuery type system, facilitates the specification of data models that go beyond document types, admitting collections of heterogeneous, arbitrarily shaped data items, and providing additional constructs for advanced usages [12].

XQuery has been specifically designed to query XML documents while paying a lot of attention to many details of XML-centric data processing. XQuery supports both typed and untyped data, focusing on structured as well as semi-structured use cases [14]. The language itself is declarative, enabling many rewriting and optimization opportunities for the compiler and runtime engine, many of which have been extensively researched over the past years (e.g., [6, 7]). XQuery is relatively easy to use, with simple constructs for node construction, XPath-based navigation, and flexible FLWOR expressions for joining and ordering of XML data. While currently focusing on declarative query processing, the language roadmap includes the XQuery Update Facility extension [15], for handling data modifications in a declarative fashion, as well as the XQuery Scripting Extension [16], for imperative programming when strict evaluation order is needed and side-effects may be present. The XQuery language has an active community of users and is gaining adoption across many commercial software vendors. All these factors make it an excellent language choice for building a complex data federation system.

Figure 1 illustrates how a complex data federation problem of assembling a single view of customer information is easily accomplished in an XQuery-capable system. It demonstrates a scenario where the data is assembled from three different data sources: two relational databases containing customer information along with the orders, and a Web service used to obtain the credit rating. Access to relational tables is modeled via

```
declare namespace db_customer = 'urn:CUSTOMER';
declare namespace db_order = 'urn:ORDER';
declare namespace websrv_credit_check = 'urn:CREDIT_CHECK';

declare function getProfile() as element(customer_profile)*
{
  for $customer in db_customer:CUSTOMER()
  return
    <customer_profile>
      <customer_id>{ data($customer/cid) }</customer_id>
      <name>
        <first>{ data($customer/first_name) }</first>
        <last>{ data($customer/last_name) }</last>
      </name>
      <credit_rating>{
        let $ssn := data($customer/ssn)
        return websrv_credit_check:GET_CREDIT_RATING($ssn)
      }</credit_rating>
      <orders>{
        for $order in db_order:ORDER()
        where $order/customer_id eq $customer/cid
        order by $order/order_date descending
        return
          <order>
            <order_id>{ data($order/order_id) }</order_id>
            <date>{ data($order/order_date) }</date>
            <total>{ data($order/total_amount) }</total>
          </order>
      }</orders>
    </customer_profile>
};
```

Figure 1: XQuery example

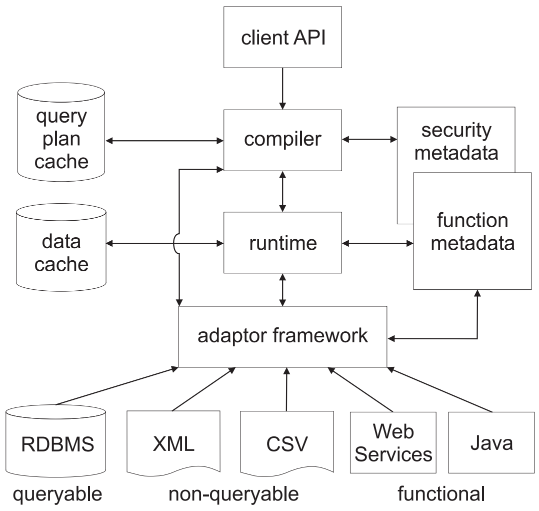

Figure 2: Overview of the ALDSP engine architecture

XQuery function calls (db_customer:CUSTOMER() and db_order:ORDER()), as is a parameterized invocation of the Web service (websrv_credit_check:GET_CREDIT_RATING()). Note that, due to the usage of XML, the result has a natural nested structure, allowing for convenient client data consumption and simple bindings to other programming environments and data models, such as Service Data Objects (SDO) [17] and the Java Architecture for XML Binding (JAXB) [18].

3 XQuery Language Extensions

While our experience has shown XQuery to be an excellent choice for a data federation language, we also found it necessary to extend the language in certain ways in order to support advanced querying capabilities and to make existing features easier to use. This section describes some of the language extensions that have been implemented in ALDSP for these purposes.

• **Metadata.** In ALDSP, enterprise information sources are modeled as external XQuery functions whose actual implementations are transparently provided by the system. Early in the design of ALDSP we were faced with the need to capture and store metadata pertaining to external data sources. The solution we adopted was to extend XQuery prolog declarations with a flexible concept of annotations, which are XML fragments augmenting either an individual function declaration or a whole prolog/module in general [19]. They are defined using "pragma" directives that either precede a function declaration or appear at the beginning of a module/prolog definition:

```
(::pragma name <XML_content/> ::)
```

As the content of an annotation is XML, it can easily hold various kinds of information. One of the usages of annotations in ALDSP is to describe data source binding properties such as relational database connectivity configurations, Web service definition and endpoint locations, delimited file format properties, etc. Over time, ALDSP's usage of annotations has evolved to store many other details of a function/prolog configuration in the product, such as function visibility scope, modeling kinds, update configuration information, and key specifications. In retrospect, this powerful annotation framework minimized the overall number of artifacts in the system and allowed us to quickly introduce new concepts and features as ALDSP evolved.

• **Optional node constructors.** Renaming elements and attributes is a common operation performed in queries that integrate data. In the following example, an XQuery expression is used to rename the customer's "last_name" element to "last", creating a new element with the new name and copying the typed value of the input element:

```
<last>{ data($customer/last_name) }</last>
```

Per XQuery semantics, this expression calls for the construction of an empty element in the event that the input is the empty sequence. But what if the user wants to create the new element with the new name only if the input is non-empty? One can express that logic in XQuery 1.0 as follows:

```
if (exists(data($customer/last_name))) then <last>{ data($customer/last_name) }</last> else ()
```

Given the occurrence frequency of this sort of scenario in data integration use cases, a less verbose approach was required. We extended the direct element and attribute constructors of XQuery with a ? modifier, so the same logic can be expressed as follows in ALDSP:

```
<last?>{ data($customer/last_name) }</last>
```

To optionally create attributes based on the input, one would write

```
<customer last?="{ data($customer/last_name) }" />
```

• **Group by.** Grouping data is an important operation in query processing but, unfortunately, the standard XQuery 1.0 provides no concise way to do so. In our XQuery engine, we added a GROUP BY clause to the FLWOR expression [1]. The following query constructs sequences of customer names grouped by their zip codes.

```
for $customer in db_customer:CUSTOMER()
group $customer as $c-group by $customer/zip_code as $zip
return <group zip="{ $zip }">{ $c-group/last_name }</group>
```

• **XQSE.** Although any computation can be expressed in XQuery, some processing is easier to express in an imperative manner (like in Java, C++, etc). This is also relevant when the steps in a program have side effects beyond the state of the program itself, as XQuery is a side-effect free language. We introduced the XQuery Scripting Extension (XQSE), described in detail in [20], to overcome this limitation of XQuery. XQSE is a proper superset of XQuery based on statements. XQuery expressions can be used anywhere in an XQSE program where an expression is expected. Some of the constructs supported in XQSE are "while" and "iterate" loops, variable assignment with "set" statements, conditional "if" statements, and "try/catch" based error-handling, which is commonplace in popular programming languages.

• **Typing extensions.** The XQuery standard includes an optional feature for statically typing expressions. We found it necessary to extend the XQuery type inferencing rules to meet users' requirements, as requiring query writers to explicitly request revalidation on node construction in order to stay in the typed world was producing a poor user experience. To work around this issue, we implemented a structural form of type inferencing; types in ALDSP are represented by their structure rather than by their schema type name. This is also absolutely essential for view unfolding, which needs to preserve type information through the process of node construction and subsequent drill-down [21].

4 Implementation Techniques

Figure 2 gives an overview of the ALDSP query engine. Queries are submitted for execution through the client API, compiled and optimized, then evaluated by the runtime subsystem, utilizing the adaptor framework for external data source connectivity. Assisting in query processing are metadata context providers, which keep track of various configuration parameters and other properties, as well as caching components for improving overall system performance.

Efficient query execution is crucial in data integration scenarios. Our experience has shown that layers of XQuery functions are quite common in federated data views. In ALDSP, users start with XQuery functions representing physical data sources, then create functions for logical transformations, and finally specialize them for publishing through client APIs. User-defined XQuery functions can be reused in each step during this process, selection predicates can be applied at various layers, and code reuse could potentially result in subparts of a function not being required for a final result. The ALDSP engine performs efficient query evaluation by using standard optimization techniques such as function inlining, unnesting, dead code elimination, and many others [21]. All non-recursive functions are inlined in the beginning of the rewriting process, thus enabling the optimizer to have a global view of the whole query. Subsequent optimization stages rely on this global view to rewrite parts of the plan to a more efficient form, eliminate expressions that were determined to be unnecessary for the result, and choose optimal implementations for runtime operators.

Another important feature of our engine is the inclusion of relational operators in its core XML query algebra. During the query compilation phase, these operators enable well-known relational optimizations such as join reordering, predicate pushdown, transitive condition inference, and many others. At runtime, relational operators are evaluated on tuple streams in a traditional (relational database like) manner. Efficient join processing is vital to overall system performance. The ALDSP query compiler detects inner, outer, and semi-joins patterns in XML queries and the execution engine implements them using well-known join algorithms. When it comes to combining data from relational sources, ALDSP employs a distributed join method internally called *clustered parameter passing join*. It significantly reduces the number of accesses to the underlying database sources and leads to a very efficient query evaluation. Grouping and aggregation operations require special attention in data integration use cases and have always been at the focus of ALDSP query processing. First of all, as described in the previous section, ALDSP introduces an additional “group-by” clause in the FLWR expression, which is backed up by optimizer and runtime support. During query compilation the optimizer may choose to split group-by into two operations: clustering and pre-clustered grouping. Clustering is a weaker form of sorting which may be merged with adjacent order-by clauses or eliminated altogether if the optimizer can prove that the input is already clustered on the required field. The grouping operation is then executed in a streaming fashion on pre-clustered input.

Relational database systems play a central role in the information federation architecture, typically storing most of the enterprise data. For this reason, the ALDSP engine specifically focuses on optimizing database access patterns. We designed and implemented ALDSP’s XQuery to SQL translation framework to identify XQuery subexpressions and patterns that can be translated into equivalent SQL queries and pushed down to underlying database sources for native execution. A key feature of the SQL generator is its broad support of different SQL dialects found in modern database systems, which is also customizable by users. The XQuery to SQL translation process is driven by the ALDSP query optimizer. First of all, it relies on the join identification performed in previous optimization stages. Using join blocks in the plan, the optimizer then re-arranges expressions to maximize SQL-able fragments. Finally, there’s a SQL text generation stage which emits SQL queries and replaces XQuery fragments with database invocation expressions which will be executed at runtime. The key problem we faced at this stage is how to preserve the semantic equivalence between a generated SQL statement and the actual XQuery expression given by the user. Unfortunately, we found that in some cases preservation may not be possible or may lead to highly suboptimal query execution plans. In these relatively rare cases, the query optimizer is designed to prefer overall query performance over adhering exactly to precise XQuery semantics, while also providing query architects with flexible mechanisms to control which parts of the query are executed by the underlying databases and which are evaluated in the middle tier by the ALDSP engine. An example of such a semantic mismatch is when a database does not properly distinguish between an empty string and a NULL value, or if it has some special rules for string comparison operations on certain character data types.

The major challenge in executing queries efficiently in the middleware is to avoid data materialization, as it usually impacts performance negatively. The ALDSP runtime engine meets this challenge by processing data in a streaming fashion, thus preventing materialization whenever possible. XML data is represented as a stream of small tokens, each corresponding to a part of an XML data item [22]. These tokens flow through the runtime system and are discarded as soon as possible. The ALDSP's internal XML data model extends the XQuery Data Model with support for tuple tokens which serve as typed containers for various data items. Having tuple tokens greatly simplifies implementation of joins and grouping operators, at the same time natively matching relational data obtained from back-end database systems during query execution. In cases when large data sets are unavoidable during query execution, the ALDSP runtime supports such time-tested memory management techniques as external merge sorting and secondary storage join operators.

5 Updates

We now turn our attention to the ALDSP update model. ALDSP's API enables a client to execute a query, operate on the results, and then submit the modified data back to persist the changes. Changes on the client side are transmitted using Service Data Objects (SDO) [17]. On the server side we have extended the XQuery Data Model (XDM) with an SDO-like ability to carry changes. The result, eXtended XDM, or XXDM for short, is a proper superset of XDM in terms of information content. In other words, XXDM can model everything that XDM can model, and it can also model changes to XDM instances.

XXDM nodes share the same data model attributes as XDM nodes (see [11]) and have an additional attribute called "state" which is used to indicate if the node has been changed or not, and if so, how. This state attribute can have one of four values: CREATED, DELETED, MODIFIED, or NONE. A newly created XXDM node has a value of CREATED, a node to be deleted has a value of DELETED, a node that has been modified has a value of MODIFIED, and a node that has not been altered has a value of NONE. Like nodes, atomic values have state as well but their attribute may not have a value of MODIFIED. Modified atomic values are represented by a DELETED value (the old value) followed by a CREATED value (the new value). We use this finer-grained indicator for modification of simple content so that changes in sequences of atomic values can be captured more efficiently.

XXDM is similar, at least abstractly, to the pending update list (PUL) concept in the XQuery Update Facility (XUF) [15]. While conceptually related, the goal of XXDM is different. The PUL is used to explain the semantics of various XUF constructs, and is used only implicitly for that purpose. In contrast, XXDM is a concrete extension to XDM that provides programmatic access to data and changes.

Changes to a result set need to be translated to the underlying data sources, and ALDSP provides the user with two tools for doing this: automatic update maps and XQSE (see Section 3). Update maps are an internally generated description of how to map values from target to source. ALDSP generates them automatically by analyzing the XQuery source for a data service definition and essentially inverting the query. The mapping is described using an internal language that the user can inspect, fix, and augment using a graphical editor. For cases where the update map is insufficient or unavailable, the XQSE scripting capabilities can be used to decompose the changes manually. For this purpose, ALDSP provides a built-in library of mutator functions for working with XXDM instances. XQSE can also be used in combination with update maps, allowing the user to inject complex business logic or error handling without having to hand code the basic "mapping" logic. We refer the reader to [23] and [24] for more information.

6 Conclusion

In this paper we have explained how we utilized XQuery at BEA as the core technology for a modern information integration product (ALDSP, now called ODSI – for Oracle Data Service Integrator). We discussed how we implemented the full XQuery language in that context at BEA, covering some of the techniques used to ensure efficiency and some problems that we faced along the way. Key techniques included the use of efficient and streamable internal data formats, much like those in commercial relational query engines, and a strong focus on delegating query processing to the underlying data containers whenever possible. We also briefly described how ALDSP handles updates. Based on our experiences to date with XML and XQuery, as well as with the diversity of enterprise data sources, we are very optimistic about the future of XML and XQuery as the “right” fit for information integration in the SOA era.

References

[1] V. Borkar, M. Carey, *Extending XQuery for Grouping, Duplicate Elimination, and Outer Joins*, in XML Conference and Expo., Nov, 2004.

[2] M. Huhns, M. Singh, *Service-Oriented Computing: Key Concepts and Principles*, in IEEE Internet Computing (9):2, 75–81, 2005.

[3] M. Singh, M. Huhns, *Service-Oriented Computing: Semantics, Processes, Agents*, Wiley, 2005.

[4] World Wide Web Consortium, *XML Schema Part 1: Structures Second Edition*, http://www.w3.org/TR/2004/WD-xquery-20040723/, 28 Oct, 2004.

[5] World Wide Web Consortium, *XML Schema Part 2: Datatypes Second Edition*, http://www.w3.org/TR/2004/WD-xquery-20040723/, 28 Oct, 2004.

[6] V. Braganholo, S. Davidson, C. Heuser, *From XML View Updates to Relational View Updates: Old Solutions to a New Problem*, in Proc. of the Int’l Conference on Very Large Data Bases, 276–287, 2004.

[7] Y. Papakonstantinou, V. Borkar, M. Orgiyan, K. Stathatos, L. Suta, V. Vassalos, P. Velikhov, *XML Queries and Algebra in the Enosys Integration Platform*, in Data & Knowledge Engineering, (44):3, 299–322, 2003.

[8] M. Carey, *Data Services: This is Your Data on SOA*, in Business Integration Journal, Nov/Dec, 2005.

[9] V. Borkar, M. Carey, N. Mangtani, D. McKinney, R. Patel, S. Thatte, *XML Data Services*, in International Journal of Web Services Research, (3):1, 85–95, 2006.

[10] M. Carey, the AquaLogic Data Services Platform Team, *Data Delivery in a Service-Oriented World: The BEA AquaLogic Data Services Platform*, in Proc. of the ACM SIGMOD Int’l Conference on Management of Data, 695–705, 2006.

[11] World Wide Web Consortium, *XQuery 1.0 and XPath 2.0 Data Model (XDM)*, http://www.w3.org/TR/2007/REC-xpath-datamodel-20070123/, 23 Jan, 2007.

[12] World Wide Web Consortium, *XQuery 1.0 and XPath 2.0 Formal Semantics*, http://www.w3.org/TR/2007/REC-xquery-semantics-20070123/, 23 Jan, 2007.

[13] World Wide Web Consortium, *XQuery 1.0: An XML Query Language*, http://www.w3.org/TR/2007/REC-xquery-20070123/, 23 Jan, 2007.

[14] World Wide Web Consortium, *XML Query Use Cases*, http://www.w3.org/TR/2007/NOTE-xquery-use-cases-20070323/, 23 Mar, 2007.

[15] World Wide Web Consortium, *XQuery Update Facility 1.0*, http://www.w3.org/TR/2008/CR-xquery-update-10-20080801/, 28 Aug, 2008.

[16] World Wide Web Consortium, *XQuery Scripting Extension 1.0*, http://www.w3.org/TR/xquery-sx-10/, 28 Mar, 2008.

[17] Adams et al., *Service Data Objects For Java Specification*, in Service Data Objects For Java Specification, Ed. 2.1, 2006.

[18] S. Vajjhala, J. Fialli, *The Java Architecture for XML Binding (JAXB) 2.0*, http://jcp.org/en/jsr/detail?id=222, 19 Apr, 2006.

[19] P. Reveliotis, M. Carey, *Your Enterprise on XQuery and XML Schema: XML-based Data and Metadata Integration*, in Proc. of the Int'l. Workshop on XML Schema and Data Management (XSDM), 2006.

[20] V. Borkar, M. Carey, D. Engovatov, D. Lychagin, T. Westmann, W. Wong, *XQSE: An XQuery Scripting Extension for the AquaLogic Data Services Platform* in Proc. of the Int'l Conference on Data Engineering, 1229–1238, 2008.

[21] V. Borkar, M. Carey, D. Lychagin, T. Westmann, D. Engovatov, N. Onose, *Query Processing in the AquaLogic Data Services Platform* in Proc. of the 32nd Int'l Conference on Very Large Data Bases, 1037–1048, 2006.

[22] D. Florescu, C. Hillery, D. Kossmann, P. Lucas, F. Riccardi, T. Westmann, M. Carey, A. Sundararajan, *The BEA Streaming XQuery Processor*, in The VLDB Journal, (13):3, 294–315, 2004.

[23] V. Borkar, M. Carey, D. Lychagin, R. Preotiuc-Pietro, P. Reveliotis, J. Spiegel, T. Westmann, *XDM + SDO = XXDM: Getting Change Back From XDM*, in Proc. of the Int'l Workshop on XQuery Implementation, Experience and Perspectives <XIME-P/>, 2008.

[24] M. Blow, V. Borkar, M. Carey, C. Hillery, A. Kotopoulis, D. Lychagin, R. Preotiuc-Pietro, P. Reveliotis, J. Spiegel, T. Westmann, *Updates in the AquaLogic Data Services Platform*, in Proc. of the Int'l Conference on Data Engineering, 2009.