XQueryP: Programming with XQuery

Don Chamberlin
IBM Almaden Research Center
chamberlin@almaden.ibm.com

Michael Carey
BEA Systems
mcarey@bea.com

Daniela Florescu
Oracle
dana.florescu@oracle.com

Donald Kossmann
ETH
kossmann@inf.ethz.ch

Jonathan Robie
Data Direct
jonathan.robie@datadirect.com

Abstract

XQuery is a declarative language for querying and updating XML data sources. Interfacing XQuery to a host programming language is difficult because of the type system mismatch, and global optimization is difficult in a mixed-language environment. In this paper, we investigate a small extension called XQueryP that enables XQuery expressions to exchange state information through variables. This extension makes it easier to develop applications in XQuery without relying on a host programming language. We draw an analogy between the proposed extension and similar extensions that have been added over the years to the SQL query language for similar reasons.

1. Introduction

XML continues to gain importance as a data interchange format. Application-specific XML vocabularies such as XBRL [1] and HL7 [2] continue to emerge. XML is used in web services both for service descriptions [3] and for data exchange [4]. It is also used as a storage format for the latest generation of Microsoft Office documents [5].

Given the proliferation of XML data, tasks such as merging and transforming XML data sources are increasingly important. Today, these tasks are often accomplished by using programming languages such as Java, Perl, or PHP. The application program captures XML data using an XML query language such as XPath or XQuery or a programming interface such as DOM or SAX. Before processing, the XML data is converted into objects that are native to the programming language. If persistence is necessary, these objects are converted (again) into a storage format and stored in a file system or relational database. After processing, the data is often converted back into XML and passed to another program where it undergoes another series of conversions.

The problems with this approach are obvious. Transforming data from one environment to another impacts performance and adds unnecessary complexity. No programming language supports the type system of XML Schema [6] or provides direct support for recursively nested XML structures. XPath and XQuery are declarative and set-oriented, whereas host programming languages are procedural and scalar. Global optimization in a mixed-language environment is difficult or impossible. This well-known problem is often referred to as "impedance mismatch."

The impedance mismatch problem has existed in the relational database world for many years. SQL is a set-oriented declarative language with its own type system, often used together with a host programming language. The approaches that have been proposed for eliminating the mismatch between SQL and its host languages fall into two general categories: (1) Enhance a procedural language with relational query operators so that it no longer needs to call SQL; or (2) Enhance SQL with extensions to support application logic without relying on a host language.

The approach of enhancing a programming language with relational query operators is exemplified by languages such as Pascal/R [7]. While these languages represent interesting research, they have had little impact on the database industry. One possible reason is that procedural languages do not offer the same opportunities for optimization that are available in declarative languages such as SQL. Another possible reason is that users prefer to have a query language that can be used in multiple environments rather than restricting data access to a single programming environment. SQL also provides functionality, such as authorization and definition of views and constraints, that does not fit well into a typical programming language. A more recent example of adding database operators to a procedural language is Microsoft's DLinq [8]. It will be interesting to see whether this work is more successful than its predecessors.

Historically, the approach of extending SQL to make it independent of a host language has been more successful. Several commercial implementations of this approach exist, including Oracle's PL/SQL [9] and IBM's SQL PL [10]. In 1996, a set of programming extensions called PSM were adopted as part of the ANSI/ISO SQL standard [11]. Today, programming extensions are supported by most SQL implementations. SQL as a programming language has been used extensively in building many commercial applications including salesforce.com [12] and the Oracle application suite. In general, industry experience suggests that it is easier to add a few carefully-selected control flow operations to a database query language than it is to embed a foreign type system and persistence model into a procedural programming language.

The available approaches for dealing with impedance mismatch in the XML world fall into the same general categories as the older relational database approaches. One possibility is to add XML types and retrieval/update operations to an existing programming

XIME-P 2006: 3rd International Workshop on XQuery Implementation, Experience, and Perspectives, June 30, 2006, Chicago, Illinois

language. This is the approach taken by Microsoft XLinq [13]. This approach tends to limit opportunities for optimization, and does not address the basic mismatch between the primitive types of XML Schema and the host language.

In this paper, we examine the alternative approach: extending XQuery with operations that support application development. Our goal is to find the minimum extension to XQuery that is sufficient for this purpose, and to explore the benefits of this extension. We will refer to this extension as XQueryP. We believe that the case for extending XQuery with programming operators is even stronger than it was for SQL, for two reasons: (1) XQuery already has a form of iteration and conditional expressions, so the extensions needed for application development are minimal; and (2) The mismatch with the type system of a typical host language is even greater for XML data than for relational data.

As a starting point for XQueryP, we take XQuery Version 1.0 [14] as extended by the XQuery Update Facility [15]. With the Update Facility, XQuery is close to a native XML application language. In Sections 2 and 3, we propose two small extensions that further support application development using XQuery. In Section 4 we present an illustrative example, and in the following sections we present related work, conclusions, and future work.

This paper assumes a familiarity with XQuery 1.0 and the XQuery Update Facility. Grammar productions used in this paper use the same format as those in the XQuery specification and in some cases rely on symbols defined in the XQuery grammar. We also use some terms, such as "effective boolean value" and "SequenceType matching," that are defined in the XQuery specification. All the grammar productions in this paper have been merged into an experimental XQuery parser and used to validate the examples in the paper.

2. XQueryP

Our goal is to define the smallest extension to XQuery that makes development of simple applications reasonably feasible. Of course, this is a matter of judgment. We realize that some of the features described here are not strictly necessary--for example, a while-loop might be replaced by a recursive function. Nevertheless, we describe here a small extension called XQueryP, without which we believe that application developers would be reluctant to use XQuery as a stand-alone development language. XQueryP consists of four parts: sequential execution mode, blocks, assignments, and while expressions.

2.1 Sequential Execution Mode

In XQuery, as in SQL, data is passed from one expression to another by physical nesting of the expressions. Side effects are saved on "pending update lists" and made effective only as the last step in query processing. As a result, the side effects of one expression cannot be seen by other expressions. This approach does not scale well to complex applications. Application development would be made easier by the notion of a "state" that could be modified by successive updating expressions. The idea of a state, represented by a set of variables, is a natural extension to XQuery since the language already has variables that are bound by various kinds of expressions such as `for`, `let`, and `some`.

The XQuery Update Facility defines an internal "apply" operation that makes updates visible, and invokes this operation at the root of the expression tree. If the "apply" operation were invoked at earlier stages in query processing, expressions would be able to see the side effects of other expressions. In order to make the result well-defined, it would be necessary to define an ordering on the evaluation of expressions in the expression tree. We propose to define such an ordering and to make it effective by means of a prolog declaration called an execution declaration, with the following syntax:

Setter ::= (all existing options) | ExecutionDecl

ExecutionDecl ::= "declare" "execution" "sequential"

In the absence of an execution declaration, the semantics of XQuery are exactly as specified by XQuery 1.0 and the XQuery Update Facility. If an execution declaration is present, the query is said to be in sequential mode, and a sequential ordering is defined on the expression tree. The semantics of updating expressions are changed to invoke the "apply" operation immediately rather than returning a pending update list. In sequential mode, each expression can see the side effects of previous expressions.

In sequential mode, the effect of evaluating an expression tree must be the same as though the expressions were evaluated in sequential order. If none of the expressions have side effects, sequential mode is not relevant, and all the optimizations available in XQuery 1.0 still apply. In the presence of side effects, the evaluation order of expressions is effectively constrained by the interdependencies among the expressions, which can be detected by standard data-flow optimization techniques [16]. Because of the constraints defined by the Update Facility on where updating expressions can be used, the actual limitations imposed by sequential mode on optimization are minimal.

An additional consequence of sequential mode is that three new kinds of expressions are made available: blocks, assignments, and while expressions. When a query is not in sequential mode, these expressions raise static errors. The syntax and semantics of the new expressions are described in the following sections.

The evaluation order of expressions in sequential mode is defined as follows (a rule is provided for every kind of expression that can have updating subexpressions):

- FLWOR: The for, let, where, and order-by clauses are evaluated first, in order of appearance, generating a tuple stream. Then the return clause is evaluated multiple times in the order of the tuple stream, and "apply" is invoked after each evaluation.
- if-then-else: The if-clause is evaluated first. Next, either the then-clause or the else-clause (but not both) is evaluated.
- typeswitch: The switch-expression is evaluated first. Then, exactly one of the return-clauses is evaluated.
- transform: No change to existing semantics is needed. The copy clause is evaluated first; then the modify clause is evaluated and applied; then the return clause is evaluated.
- comma: Operand expressions are evaluated from left to right, and "apply" is invoked after each.
- function call: Argument expressions are evaluated before the function body.

2.2 Blocks

Blocks provide a new way of combining expressions that takes advantage of sequential mode. A block consists of one or more expressions, enclosed in curly braces and optionally preceded by one or more declarations. Inside a block, the declarations and expressions are separated by semicolons. A block can appear anywhere a primary expression is expected.

Syntax: (see Section 3 for an addition to the Block production)

PrimaryExpr ::= (all existing options) | Block

Block ::= "{" (BlockDecl ";")* Expr (";" Expr)* "}"

BlockDecl ::=
"declare" "$" VarName TypeDeclaration? (":=" ExprSingle)?
("," "$" VarName TypeDeclaration? (":=" ExprSingle)?)*

The semantics of a block are as follows: If execution mode is not sequential, a block raises a static error. If execution mode is sequential, the declarations (if any) are processed in order, bringing new variables into scope inside the block. Then the expressions are evaluated from left to right, with the side effects of each expression made effective immediately. The results of all the expressions except the last one are ignored. The result of the last expression is returned as the result of the block. A block may contain a mixture of updating and non-updating expressions. In common usage, it is expected that all the expressions in a block, possibly excepting the final one, will be updating expressions. A block is defined to be an updating expression if any of its contained expressions is an updating expression.

Inside a block, the syntax and semantics of a declaration are very similar to those of a let-clause in a FLWOR expression. Each declaration names one or more variables and can provide a type and an initializing expression for each variable. The semantics are as follows: The individual variable declarations are processed in order from left to right. For each variable declaration, the initializing expression (if any) must be a non-updating expression, and its type must match the declared type (using the rules for SequenceType matching). The initializing expression (if any) is evaluated and the resulting value is bound to the declared variable. The scope of each declared variable is the remainder of the block, including the initializing expressions of subsequent declared variables, but not including its own initializing expression. Each variable declaration occludes any existing in-scope variable with the same name.

If a declaration inside a block does not specify a type for a variable, the implicit type of the variable is xs:anyType. If a declaration does not specify an initializing expression for a variable, the variable is not bound to a value. Any reference to such a variable (other than on the left-hand-side of an assignment expression, as described in the next section) is a dynamic error.

The following example illustrates usage of a block. The example increases the price of all items in a catalog whose price is less than 100 by ten percent, and returns the updated items. It is often desirable for an expression to both update an element and return it (or some value computed from it), both for efficiency and because it might (as in this case) be impossible to find the updated elements by a subsequent query.

```
for $item in /catalog/item[price < 100]
return
  {do replace value of $item/price with $item/price * 1.1; $item}
```

Inside a direct element constructor, two adjacent curly braces are ordinarily interpreted as a single literal curly brace. Therefore, in this context, if two adjacent curly braces are intended as delimiters, they must be separated by whitespace.

2.3 Assignment

We define a new kind of expression, called an assignment expression, to bind a variable to a new value. Assignment expressions allow different parts of a computation to exchange information through side effects rather than by returning values. Like the insert, delete, replace, and rename expressions, the assignment expression is an updating expression that returns an empty sequence.

Syntax:

ExprSingle ::= (all existing options) | AssignExpr

AssignExpr ::=
"set" "$" VarName ":=" ExprSingle

Example:

set $y := <holiday>{xs:date("2006-07-04")}</holiday>

The semantics of an assignment expression are as follows: If execution mode is not sequential, a static error is raised. If execution mode is sequential, the variable named on the left-hand-side of the assignment expression must have been declared in the query prolog or in a containing block, and must not have been overridden by a variable binding in a for-clause, let-clause, typeswitch, or quantified expression.

The expression on the right-hand side of the assignment expression must be a non-updating expression, and its type must match the type of the variable on the left-hand-side, using the rules for SequenceType matching. The right-hand-side expression is evaluated and the resulting value is bound to the named variable, replacing its existing value (if any).

The following example illustrates the use of a block containing an assignment expression. The example generates a list of project elements in document order, showing the individual cost and cumulative cost of each project in the year 2005.

```
{ declare $total-cost as xs:decimal := 0;
  for $p in /project[year eq 2005]
  return
    {  set $total-cost := $total-cost + $p/cost;
       <project>
          <name>{$p/name}</name>
          <cost>{$p/cost}</cost>
          <cumulative-cost>{$total-cost}</cumulative-cost>
       </project>
    }
}
```

2.4 Discussion of binding semantics

Various expressions in XQuery 1.0, as well as the assignment expression introduced by XQueryP, can bind a variable to a value

that may include one or more nodes. Sequential execution mode requires us to examine the nature of this binding closely, since a node may be modified after being bound.

Nodes are bound to variables using reference semantics--that is, the value that is bound to a variable may include one or more node references. Multiple variables may contain references to the same node. If a referenced node is modified, the modifications are visible through all variables that reference the node. If a node is deleted, any existing references to the node remain bound to their respective variables and can be used to access the node, but the node no longer has a parent.

It is important to note that, in a FLWOR expression, the variable bindings in the for and let clauses are evaluated before the first iteration and remain constant throughout all the iterations. Updating expressions in the return clause may modify or delete nodes that were bound to variables by the for and let clauses. In sequential mode, each iteration can see the modifications made by the previous iteration, even though the bound variables (which contain node references) are not re-evaluated.

In the following example, $books is bound to a sequence of references to red books, and the referenced books are changed from red to blue by successive iterations of the return clause. In this example, every iteration returns the same average price because the original set of books remains bound to the variable even as their colors change. This illustrates the fact that the predicate used in a let clause does not necessarily remain invariant during execution of a FLWOR expression.

```
let $books := /book[color eq "Red"]
for $i in (1 to length($books))
return
  {do replace value of $books[i]/color with "Blue";
   avg($books/price) }
```

2.5 Functions and Blocks

In a function declaration, the body of a function is an expression enclosed in curly braces. In order to allow a function body to include a block without requiring an additional set of curly braces, we replace EnclosedExpr with Block in the grammar production for FunctionDecl. This is a compatible change.

The execution mode of a function is determined by the presence or absence of an execution declaration in the prolog of its module. The body of a nonsequential function may not contain an assignment expression, while expression, semicolon, or nested block.

Since a block may contain updating expressions and also return a value, the semantic rule that the declaration of an updating function must not declare a result type is relaxed. The following example illustrates an updating function that returns a value. It deletes all mail messages that are older than a given date, and returns a count of the deleted messages.

```
declare updating function local:prune($d as xs:date) as xs:integer
  { declare $count as xs:integer := 0;
    for $m in /mail/message[date lt $d]
    return
      { do delete $m; set $count := $count + 1 };
    $count
  };
```

2.6 While expression

A FLWOR expression iterates over a predefined set of values. In sequential mode, successive iterations may see a different dynamic context, and there is a need to terminate an iteration based on the dynamic context. This is accomplished by introducing a new iterator called a while expression.

Syntax:

ExprSingle ::= (all existing options) | WhileExpr

WhileExpr ::= "while" "(" ExprSingle ")" "return" ExprSingle

The semantics of a while expression are as follows: If execution mode is not sequential, a static error is raised. We will refer to the expression following "while" as the test expression and to the expression following "return" as the body expression. The test expression must not be an updating expression. The test expression is evaluated. The following two-step process is then repeated as long as the effective boolean value of the test expression is true:

- The body expression is evaluated and its side effects are made effective.
- The test expression is re-evaluated.

The result of a while expression is the concatenation of all the values returned by its body expression, in iteration order. A while expression is an updating expression if its body expression is an updating expression; otherwise it is a non-updating expression.

3. Explicit Atomicity

Many applications need to preserve atomicity for a series of updates. For example, an application may withdraw money from one account and deposit it into another; the semantics of the application require all or none of the updates to be applied. In this section, we propose an XQuery extension that permits users to control the atomicity of their updates without recourse to a host programming language.

The natural unit of atomicity is a block of expressions. Therefore we propose to extend the syntax of a block with an optional keyword denoting atomicity, as follows:

Block ::= "atomic"? "{" (BlockDecl ";")* Expr (";" Expr)* "}"

Note that the XQuery Update Facility defines the unit of atomicity to be an entire query. In sequential mode, default atomicity is at the individual expression level. Explicit atomic blocks introduce a unit of atomicity that is intermediate between these extremes. If atomic blocks are nested, or if a function containing an atomic block is called from inside an atomic block, only the largest containing atomic block is effective.

If all the expressions in an atomic block are evaluated successfully, execution proceeds as usual (the side-effects of these expressions become visible to expressions outside the atomic block.) However, if any expression in an atomic block results in an error, the side-effects of all expressions in the atomic block are rolled back (made ineffective), and the atomic block raises one of the errors resulting from its internal expressions. If multiple errors occur inside an atomic block, selection of which error is raised by the block is implementation-dependent.)

XQuery is designed for use in many different environments, some of which may have no requirement for atomicity. Therefore we propose that explicit atomicity be an optional feature that is separate from XQueryP. As in the case of other optional features, an implementation that does not support explicit atomicity should raise a static error if an atomic block is encountered.

Note that, in this proposal, snapshot granularity and atomicity are controlled independently. Snapshot granularity is controlled by a prolog declaration (execution mode), but atomicity is controlled by expression syntax. In sequential mode, snapshot granularity is at the single-expression level but an atomic block can contain multiple expressions. This follows the precedent of SQL in which several SQL statements, with side effects, may be grouped into a single transaction.

The following is an example of a function containing an atomic block. If the transfer is successful, the function returns zero; otherwise it returns -1.

```
declare updating function local:transfer
    ($from-acctno as xs:string, $to-acctno as xs:string,
        $amount as xs:decimal) as xs:integer
    { declare $from-acct as element(account)
            := /bank/account[acctno eq $from-acctno],
          $to-acct as element(account)
            := /bank/account[acctno eq $to-acctno];
      if ($from-acct/balance ge $amount)
      then atomic {
          do replace value of $from-acct/balance
              with $from-acct/balance - $amount;
          do replace value of $to-acct/balance
              with $to-acct/balance + $amount;
          0 }   (: end of atomic region :)
      else -1
    };
```

4. Example

Consider a distributed music-sharing club in which each member has a personal XML document with the structure illustrated below. The documents of the club members are distributed across the Internet. Each document, called a member file, contains its own URI, the URIs of the member files of "friends" known to this member, and the URIs of songs offered for sharing by this member. The following is an example of a member file for a member named Fred who has one friend and one song to share:

```
<member>
    <name>Fred</name>
    <uri>http://fred.name/member.xml</uri>
    <friends>
        <friend>        (!-- repeats --)
            <name>Tom</name>
            <uri>http://tom.name/member.xml</uri>
        </friend>
    </friends>
    <songs>
        <song>          (!-- repeats --)
            <title>Copperhead Road</title>
            <uri>http://fred.name/songs/001.wma</uri>
        </song>
    </songs>
</member>
```

The following example query finds a URI for the song with title "My Favorite Song" by performing a breadth-first search of member files, stopping when a copy of the desired song is found. The query assumes that $myfile is bound to a member file that provides a starting point for the search.

```
declare execution sequential;
{ declare $prospect-list as element(member)* := $myfile/member,
      $prospect as element(member),
      $visited as element(uri)* := ( ),
      $index as xs:integer := 1,
      $done as xs:boolean := false( ),
      $theSong as element(song);
  while (not($done) and $index <= fn:count($prospect-list) )
  return {
      set $prospect := $prospect-list[$index];
      if ($prospect/uri = $visited)
      then set $index := $index + 1
      else {
          set $visited := $visited, $prospect/uri;
          set $theSong :=
              $prospect/songs/song[title eq "My Favorite Song"];
          if (exists($theSong))
          then {set $done := true( ); $theSong/uri}
          else {
              set $prospect-list := $prospect-list,
                  $prospect/friends/friend/uri/doc(.)/member;
              set $index := $index + 1
              }
          }
      }
}
```

This example illustrates the use of blocks, assignments, and the while expression. We believe that the example would have been significantly more difficult to write without these features.

5. Related Work

XQueryP is built on the foundation of XQuery 1.0 and the XQuery Update Facility. Like Quilt [17], which served as the basis for XQuery, XQueryP attempts to find synergy in a collection of well-known programming techniques.

Ghelli, Ré, and Siméon have proposed a set of side-effecting extensions to XQuery called XQuery! [18]. Sequential mode resembles the "ordered semantics" option described in XQuery!. However, unlike XQuery! (but like the XQuery Update Facility), sequential mode distinguishes between updating and non-updating expressions, and restricts updating expressions from appearing in certain places such as in predicates and inside other updating expressions. Also, our proposal introduces some concepts not included in XQuery!, including assignment, while expressions, and independent control over atomicity.

Florescu, Grünhagen, and Kossmann have proposed a web services implementation language named XL [19] that preceded the XQuery Update Facility and provided its own update syntax, together with features for exception handling, assertions, triggers, parallel execution, and asynchronous interactions among web services. The design philosophy of XQueryP is quite different from that of XL. Whereas XL covers a very broad range of web

service functionality, we have attempted to define the minimum extension to XQuery that is useful for application development.

Wadler has also proposed a language for web applications called Links [20]. In Wadler's approach, a high-level application description is translated into separate parts that run on a client, a midtier server, and a database server, using appropriate target languages such as Javascript, SQL, and XQuery. As in the case of XL, Wadler's approach is much more ambitious than ours, involving creation of a completely new programming language rather than a small extension to XQuery.

An XQuery implementation released by Mark Logic Corporation has extended XQuery for application development, supporting update functionality and variable assignment by means of an extended function library [21]. This work predates the XQuery Update Facility and does not provide an integrated syntax for queries and updates. Nevertheless, the Mark Logic implementation has proved the usefulness of XQuery as an application development language.

6. Conclusions and Future Work

We have investigated a proposal for XQuery language extensions to support the development of simple applications without relying on a host programming language. The extensions needed for this purpose are small and mostly involve reusing existing semantic concepts in different ways--for example, applying updates immediately rather than holding them on pending lists.

Our proposal has two parts called XQueryP and explicit atomicity. XQueryP introduces sequential execution mode, blocks, assignment expressions, and while expressions. Explicit atomicity introduces the notion of an atomic block. XQueryP is a prerequisite for explicit atomicity, and the Update Facility is a prerequisite for both features. The proposed extensions are upward-compatible with XQuery 1.0.

We believe that the development language chosen for a given application should suit the principal kind of processing done by that application. If processing XML is a small part of the application, transforming XML data into another type system may be justified. On the other hand, if XML processing is the core of the application, a native XML-based language has important advantages. Many examples of such applications can be found in web services, processing messages and transforming or integrating various XML information sources. XQuery can also support web-based user interfaces, dynamically generating XHTML or serving as a scripting language to coordinate side-effecting external functions. Writing these applications in XQuery eliminates the impedance-mismatch problem and provides maximum opportunities for global optimization.

XQueryP plays roughly the same role in XQuery that is played by Persistent Stored Modules (PSM) in the SQL Standard [11]. PSM and its commercial variations have been a successful part of SQL. Compared with PSM, XQueryP is a smaller extension and its benefits are greater. Based on industry experience with SQL, it seems inevitable that XQuery will be extended to serve as a development language for simple applications.

In order to realize the potential of XQuery as a programming language, the work described in this paper should be extended in several ways. Usability should be improved by adding `break` and `continue` constructs to the iteration expressions, and by adding a `return` expression to facilitate early exit from a function body. XQuery also needs an error-handling facility, for query expressions as well as for updates. Such a facility might be modeled on the `try/catch` facility in Java.

XQuery as an application language might also be extended with facilities for saving data persistently across invocations and for invoking (and being invoked by) web services.

7. Acknowledgments

The authors wish to acknowledge many helpful discussions with the members of the XQuery Update Task Force.

We are also grateful to Scott Boag at IBM for testing the grammar and to Zhen Hua Liu, Muralidhar Krishnaprasad, and Anguel Novoselsky at Oracle and Vinayak Borkar at BEA for their helpful suggestions.

8. References

[1] Extensible Business Reporting Language. See http://www.xbrl.org/home.

[2] Health Level 7 XML Special Interest Group (R. Dolin and P. Biron, editors). *Using XML as a Supplementary Messaging Syntax for HL7 Version 2.3.1.* See http://www.hl7.org/special/committees /sgml/hl7v231xmlFINAL.zip.

[3] E. Christensen et al. *Web Services Definition Language (WSDL) 1.1*. W3C Note. See http://www.w3.org/TR/wsdl.

[4] W3C XML Protocol Working Group (M. Gudgin et al, editors). *SOAP Version 1.2 Part 1: Messaging Framework.* See http://www.w3.org/TR/soap12.

[5] Microsoft Corporation. *Microsoft Office Open XML Formats Overview.* See http://www.microsoft.com/office /preview/developers/fileoverview.mspx (Sept. 2005).

[6] W3C Schema Working Group (H. Thompson et al, editors). *XML Schema, Parts 0, 1, and 2*. See http://www.w3.org/TR/xmlschema-0, -1, and -2.

[7] M. Jarke and J. Schmidt. Query Processing Strategies in the Pascal/R Relational Database Management System. *Proc. ACM SIGMOD Conference,* Orlando, FL, June 1982.

[8] *DLinq: .NET Language Integrated Query for Relational Data.* Microsoft Corp., Sept. 2005. See http://download.microsoft.com/download/c/f/b/cfbbc093-f3b3-4fdb-a170-604db2e29e99/DLinq%20Overview.doc

[9] Steven Feuerstein. *Oracle PL/SQL Programming, 4th Ed.* O'Reilly & Associates, 2005.

[10] Z. Janmohamed, C. Liu, D. Bradstock, R. F. Chong, M. Gao, F. McArthur, and P. Yip. *DB2(R) SQL PL : Essential Guide for DB2 UDB on Linux, UNIX, Windows, i5/OS, and z/OS.* IBM Press, 2004.

[11] *Database Languages--SQL--Part 4: Persistent Stored Modules (SQL/PSM).* ANSI/ISO/IEC 9075-4-1999.

[12] Salesforce.com. See http://www.salesforce.com/company/.

[13] *XLinq: .NET Language Integrated Query for XML Data.* Microsoft Corp., Sept. 2005. See http://download.microsoft.com/download/c/f/b/cfbbc093-f3b3-4fdb-a170-604db2e29e99/XLinq%20Overview.doc.

[14] W3C XML Query Working Group (S. Boag et al, editors). *XQuery 1.0: An XML Query Language* (03 Nov 2005). See http://www.w3.org/TR/xquery.

[15] W3C XML Query Working Group (D. Chamberlin et al, editors). *XQuery Update Facility* (8 May 2006). See http://www.w3.org/TR/xqupdate.

[16] S. Muchnick. *Advanced Compiler Design and Implementation.* Morgan Kaufmann, 1997.

[17] D. Chamberlin, J. Robie, and D. Florescu. Quilt: An XML Query Language for Heterogeneous Data Sources. Proceedings of WebDB 2000 Conference, in *Lecture Notes in Computer Science,* Springer-Verlag, 2000).

[18] G. Ghelli, C. Ré, and J. Siméon. *XQuery!: An XML Query Language with Side Effects.* Second International Workshop on Database Technologies for Handling XML Information on the Web (DataX 2006) March 2006, Munich, Germany. To appear in in *Lecture Notes in Computer Science,* Springer-Verlag.

[19] D. Florescu, A. Grünhagen, and D. Kossmann. XL: An XML Programming Language for Web Service Specification and Composition. *Proceedings of WWW2002*, Honolulu, HI, May 2002.

[20] P. Wadler, University of Edinburgh. *Links: Linking Theory to Practice for the Web--Case for Support.* See http://homepages.inf.ed.ac.uk/wadler/links/epsrc05/case.pdf.

[21] Mark Logic Corporation. Mark Logic Server, XQuery API Documentation. See http://xqzone.marklogic.com/pubs/3.0 /apidocs/UpdateBuiltins.html and http://xqzone.marklogic .com/pubs/3.0/apidocs/Extension.html.