

The W3C XML Query Working Group

Paul Cotton
Jonathan Robie

Abstract

The W3C XML Query Working Group is developing a query language for XML documents. This talk will provide an update on the work of the XML Query WG. In particular this talk will cover the proposed syntax for the XQuery language.

1. The W3C XML Query Working Group

In 1998, when the W3C organized a workshop on XML query languages, a high degree of interest was shown by the XML, database, and full-text search communities. As a result, the W3C started the XML Query Working Group, which has produced an initial design of an XML query language and an XML query algebra. This talk describes the work of the W3C XML Query Working Group, as expressed in the five documents released by the Working Group:

- XML Query Use Cases <http://www.w3.org/TR/xmlquery-use-cases>
- XML Query Requirements <http://www.w3.org/TR/xmlquery-req>
- XML Query Data Model <http://www.w3.org/TR/query-datamodel>
- The W3C XML Query Algebra <http://www.w3.org/TR/query-algebra>
- XQuery: A Query Language for XML <http://www.w3.org/TR/xquery>

The purpose of this talk is to provide an introduction, giving the motivation for XQuery and the XML Query Algebra, and illustrating how they are used with a few examples. Much more detail is available in the documents themselves, and the most recent version of each W3C XML Query document is always found at the URLs we have cited.

1.1. Why XML Query?

XML queries are important because they meet everyday practical needs. Recently, one survey <http://www.nwfusion.com/news/2001/0226xml.html> concluded that the three top reasons that enterprise IT managers use XML are:

- Users can search more corporate data and search it more accurately.
- Process the data in new ways.
- Speed up application development.

An expressive XML query language plays an important role in searching and repurposing data across the enterprise, and can significantly speed up application development in many domains.

XML is now used to represent every conceivable type of data, including documents, relational databases, object repositories, XML repositories, messages, and graphics. Some of this data is stored natively in XML. Much more data is stored in other formats or systems, but can be viewed as XML via middleware.

Traditional query languages were designed for specific kinds of data structures, and are not well suited for the rich mix of data sources found in XML. XQuery, the W3C XML Query Language, is an expressive query language that uses the structure of XML to intelligently query any data represented in XML.

The XML Query Algebra allows a precise specification of the values, data types and structures returned by each construct in XQuery. Much of the type checking can be performed before the query is executed, providing early detection of errors, and resulting in a high degree of safety. The precise model of the XML Query Algebra makes it easier to translate queries in middleware applications; for instance, an XQuery may be translated into SQL to allow queries against an XML view of a relational database. In addition, the XML Query Algebra is based on list comprehensions and set comprehensions, which can provide a basis for query optimization.

1.2. XQuery: the W3C XML Query Language

XQuery is a small, implementable language, and queries in this language are concise and easily understood. It is also flexible enough to query a broad spectrum of XML information sources, including both databases and documents. XQuery is designed to be easy to read and write, and it uses a keyword-oriented syntax similar to that of other query languages. The Query Working Group requirements specify that there will also be an XML-based syntax for queries.

XQuery is derived from an XML query language called Quilt [[Quilt: an XML Query Language for Heterogeneous Data Sources](#)], which in turn borrowed features from several other languages. From XPath [[W3C XPath 1.0](#)] and XQL [[XML Query Language \(XQL\)](#)] it took a path expression syntax suitable for hierarchical documents. From XML-QL [[A Query Language for XML](#)] it took the notion of binding variables and then using the bound variables to create new structures. From SQL [[SQL](#)] it took the idea of a series of clauses based on keywords that provide a pattern for restructuring data (the SELECT-FROM-WHERE pattern in SQL). From OQL [[The Object Database Standard: ODMG-93](#)] it took the notion of a functional language composed of several different kinds of expressions that can be nested with full generality. From XML [[W3C XML 1.0.](#)] it borrowed the syntax for constructing elements and attributes. Quilt was also influenced by reading about other XML query languages such as Lorel [[Lorel](#)] and YATL [[YATL](#)].

In this paper, we provide a brief introduction to the XQuery language by presenting several kinds of expressions and showing how they can be combined. For the examples presented here, we will assume that the file "xmp-data.xml" contains the following data:

```
<bib>
  <book year="1994">
    <title>TCP/IP Illustrated</title>
    <author>
      <last>Stevens</last>
      <first>W.</first>
    </author>
    <publisher>Addison-Wesley</publisher>
    <price> 65.95</price>
  </book>
  <book year="1992">
    <title>Advanced Programming in the Unix
```

```

environment</title>
    <author>
        <last>Stevens</last>
        <first>W.</first>
    </author>
    <publisher>Addison-Wesley</publisher>
    <price>65.95</price>
</book>
<book year="2000">
    <title>Data on the Web</title>
    <author>
        <last>Abiteboul</last>
        <first>Serge</first>
    </author>
    <author>
        <last>Buneman</last>
        <first>Peter</first>
    </author>
    <author>
        <last>Suciu</last>
        <first>Dan</first>
    </author>
    <publisher>Morgan Kaufmann
Publishers</publisher>
    <price> 39.95</price>
</book>
<book year="1999">
    <title>The Economics of Technology and
Content for Digital TV</title>
    <editor>
        <last>Gerbarg</last>
        <first>Darcy</first>
        <affiliation>CITI</affiliation>
    </editor>
    <publisher>Kluwer Academic
Publishers</publisher>
    <price>129.95</price>
</book>
</bib>

```

1.2.1. Path Expressions

Path expressions are well suited to querying the tree structure of XML documents, and are currently the most common way to query documents in the XML community. XQuery allows path expressions using the abbreviated syntax of XPath 1.0 <http://www.w3.org/tr/xpath>, with a few extensions. The following XQuery expression is also a valid XPath 1.0 expression, and has the same meaning:

```
document("xmp-data.xml")/bib/book[publisher =
```

```
"Addison-Wesley" AND @year > 1991]
```

The result of the above query is as follows:

```
<book year="1992">
  <title>Advanced Programming in the Unix
environment</title>
  <author>
    <last>Stevens</last>
    <first>W.</first>
  </author>
  <publisher>Addison-Wesley</publisher>
  <price>65.95</price>
</book>
```

It is not necessary to use the document() function when querying collections of documents. In many cases, a query is written and applied to one or more collections. If our sample data were stored in a collection that contained only the one document, the following query, applied to the collection, would yield the same result as the previous one:

```
/bib/book[publisher = "Addison-Wesley" AND @year > 1992]
```

If the collection contains more than one document, the query returns the result of applying the query to each document in the collection.

1.2.2. Element Constructors

XQuery allows elements and attributes to be constructed using the same syntax as XML. For instance, the following expression is a valid XQuery:

```
<bib id="a123">
  </bib>
```

Element constructors are often used to create literal XML that is used in query results. For instance, the following example combines the path expression from the first example with the element constructor we have just seen:

```
<bib id="a123">
  {
```

```

    /bib/book[publisher = "Addison-Wesley" AND @year
> 1992]
    }
</bib>

```

The above query creates a new element named "bib", and places the results of the path expression into this new element. Here are the results of the query:

```

<bib>
  <book year="1992">
    <title>Advanced Programming in the Unix
environment</title>
    <author>
      <last>Stevens</last>
      <first>W.</first>
    </author>
    <publisher>Addison-Wesley</publisher>
    <price>65.95</price>
  </book>
</bib>

```

The "<" character indicates the start of an element constructor. The "{" character indicates an escape from literal XML. If the curly braces were removed, the path expression would be interpreted as literal XML text. Therefore, the following is both a valid XQuery and the result of executing the same query:

```

  <bib id="a123">
    /bib/book[publisher = "Addison-Wesley" AND @year
> 1992]
  </bib>

```

As we will see, most XQuery expressions can easily be composed with any other XQuery expression. For element constructors, any XQuery expression can occur within curly braces. The curly braces can also be used to allow an expression to compute the name or value of an attribute. Outside of an element constructor, any XQuery expression can stand by itself, without the need for curly braces - these are used only to distinguish literal data from query expressions in an element constructor.

1.2.3. FLWR Expressions

XQuery's FLWR expressions are similar to SQL's SELECT-FROM-WHERE. The acronym FLWR, which is pronounced "flower", stands for the keywords FOR, LET, WHERE, and RETURN, which are the keywords that can occur in a FLWR expression. FOR iterates over lists of nodes, LET binds variables, WHERE filters out nodes that do not pass a condition, and RETURN constructs a result. A FLWR expression can contain any sequence of FOR, LET, and WHERE clauses, provided at least one such clause is present, followed by a RETURN clause. It is easiest to explain each of these clauses using simple examples.

A FOR clause iterates over a list of document nodes, assigning a variable to a given node for each iteration. For instance, the following expression creates one author element for each author found in the source:

```
FOR $a IN document("data/xmp-data.xml")/bib/book/author
RETURN
  <author>
    {
      $a/first,
      $a/last
    }
  </author>
```

To eliminate duplicates, FOR is often used together with the distinct() function, which returns a set of nodes whose values are unique. The following query returns the set of authors with no duplicates:

```
FOR $a IN
distinct(document("data/xmp-data.xml")/bib/book/author )
RETURN
  <author>
    {
      $a/first,
      $a/last
    }
  </author>
```

The FOR clause can assign more than one variable. It creates one set of variable bindings for every tuple in the Cartesian cross-product of the lists to which variables are bound. Consider the following query:

```
FOR $l IN
```

```

distinct(document("data/xmp-data.xml")/bib/book/author/last),
      $f IN
distinct(document("data/xmp-data.xml")/bib/book/author/first)
RETURN
  <name>
    {
      $l, $f
    }
  </name>

```

In our data, we have four unique last names and four unique last names. The above query creates sixteen names by producing one name with each available combination of first and last names.

The LET clause binds a variable to a list of nodes, and does not iterate over the nodes to which the variable is bound. For instance, the following query returns the number of author elements found in the source:

```

LET $a := document("data/xmp-data.xml")/bib/book/author
RETURN
  <count>
    {
      count($a)
    }
  </count>

```

The above query returns one <count> element, containing the number 5. If we had used a FOR instead of a LET, it would have returned five <count> elements, and each would contain the number 1.

The WHERE clause filters out tuples that do not meet a condition. It is most frequently used together with FOR or LET clauses. The following query returns a book list for each author who has written more than one book:

```

FOR $a IN
distinct(document("data/xmp-data.xml")/bib/book/author)
LET $b := document("data/xmp-data.xml")/bib/book[author
= $a]
WHERE count($b) > 1
RETURN
  <booklist>
    {
      $a, $b/title
    }

```

```
</booklist>
```

The output of the above query is as follows:

```
<cv>
  <author>
    <last>Stevens</last>
    <first>W.</first>
  </author>
  <title>TCP/IP Illustrated</title>
  <title>Advanced Programming in the Unix
environment</title>
</cv>
```

1.2.4. Other Expressions

This section explores some of the other kinds of expressions found in XQuery. Like most languages, XQuery allows expressions using functions and operators. The aggregate operators `min()`, `max()`, `sum()`, `count()`, and `avg()` are defined in the language. The following query returns books that are more than twice the price of the average book:

```
[
                                LET $p :=
avg(document("data/xmp-data.xml")/bib/book/price)
  FOR $b IN document("data/xmp-data.xml")/bib/book
  WHERE $b/price > 2 * $p
  RETURN
    $b/title
```

To allow conditions involving sequence, XQuery has the operators `BEFORE` and `AFTER`. Consider the following example:

```
FOR $b IN document("data/xmp-data.xml")/bib/book
WHERE $b/author[last = "Abiteboul"]
      AND NOT empty($b/author BEFORE
$b/author[last="Abiteboul"])
  RETURN
    $b/title
```

The `BEFORE` operator is an infix operator that takes two node lists. It returns the set of nodes from the left hand list that occur before the first node of the right hand list. In

the above example, it returns the set of authors that occur before the first author whose last name is "Abiteboul". The `empty()` function evaluates a node list and returns TRUE if it is an empty list. Therefore, the above query returns those books where one of the authors has the last name "Abiteboul", and the first author of the book does not have this last name.

XQuery also supports conditional logic:

```
FOR $b IN /bib/book
RETURN
    IF $b/price > 100
        THEN <wishlist> $b/title </wishlist>
        ELSE <order> $b/title </order>
```

`SORTBY` is used to sort a list. The following query produces a list of books sorted by the author's name, then by the book title.

```
document("data/xmp-data.xml")/bib/book
    SORTBY (author/last, author/first, title)
```

Functions may be declared in XQuery, then used in the body of the query. The following function takes an author element, and returns an element containing the author and the set of books by that author:

```
FUNCTION booksBy(ELEMENT $a) RETURNS ELEMENT
{
    <booksBy>
        $a,
        //book[author=$a]/title
    </booksBy>
}
</code.block>
<para>The following query uses the above function:</para>
<code.block>
<![CDATA[
    FOR $a IN distinct(//author)
    RETURN booksBy($a) SORTBY (author/last, author/first)
```

XQuery also has quantifiers, casting expressions, typeguards, and an `INSTANCEOF` operator similar to that of Java. The reader is encouraged to read the [\[XQuery: A Query Language for XML\]](#) specification for details.

1.2.5. XQuery and the W3C

Important issues remain open in the design of XQuery. Many of these issues deal with relationships between XQuery and other XML activities, for example:

- The type system of XQuery is not yet completely aligned with the type system of the formal semantics, and there are open questions about the goals of the type system.
- The details of the operators supported on simple XML Schema datatypes will be defined by a joint XSLT/Schema/Query task force.
- XQuery relies on path expressions for navigating in hierarchic documents. XQuery expects these path expressions to conform to the semantics of XPath 2.0, as defined by a joint XSLT/Query task force (see [[W3C XPath 2.0 Requirements](#)]).

1.3. XML Query Algebra

The XML Algebra proposes a set of core operations to manipulate XML documents and a formal semantics based on the XML Query Data Model. The formal semantics of XQuery is defined by mapping XQuery expressions to the Algebra core operations. Two kinds of semantics are provided. Static semantics is used to infer the type for a query before the query is executed. Because static semantics is based only on the type of each input document and does not require examination of the data, this permits early detection of errors in queries. The dynamic semantics determines the value returned by a query. This is based on the content of the documents being queried, and can not be determined without executing the query. In this section we will attempt to give a flavor of the XML Query Algebra by giving a few simple examples of the static and dynamic semantics, and how these are used to provide the semantics for XQuery.

1.3.1. XML Query Algebra Operations, Values and Types

The XML Query Algebra is composed of a set of core operations to manipulate, construct, iterate over XML documents, etc. The following BNF gives a subset of

these expressions.

```

Expr ::= Const atomic constant
      | @Name[Expr] attribute
      | Name[Expr] element
      | Expr, Expr sequence
      | () empty sequence
attribute | children(Expr) children of an element or
attribute | value(Expr) textual value of an element or
expression | for v in Expr do Expr iteration
            | if Expr then Expr else Expr iteration
            | match Expr
              | case v : Type do Expr
              | case v : Type do Expr
              | ..
              | else Expr type matching
            ...

```

Data values are defined as the subset of expressions that consists only of scalar constant, attribute, element, sequence, and empty-sequence expressions. Because XML syntax is somewhat awkward in a formal mathematical notation, XML Query Algebra has its own way of representing an XML document. For instance, here is the XML Algebra representation of the XML document for the result of the first query:

```

book [ @year [ "1992" ],
       title [ "Advanced Programming in the Unix
environment" ],
       author [ last [ "Stevens" ],
                first [ "W." ] ],
       publisher [ "Addison-Wesley" ],
       price [ 65.95 ] ]

```

The XML Query Algebra also provide a formal representation for types which uses regular expression grammars, as given by the following BNF.

```

Type ::= String | Integer ... atomic type
      | @Name[Type] attribute type
      | Name[Type] element type
      | Type, Type sequence
      | () empty sequence
      | Type | Type choice
      | 0 empty choice
      | Type{m,n} repetition (kleene star)

```

These type constructs are used to capture XML Schema constructs. For instance choice correspond to choice groups, repetitions captures the minOccurs and maxOccurs properties of XML Schema, etc.

The operations of the XML Query Algebra can be used to capture the dynamic semantics of XQuery expressions, while the types are used for the static semantics. For instance, assuming that variable 'doc' contains an element, the following XQuery navigation:

```
$doc/book
```

can be mapped to the algebra in the following way:

```
for v0 in children(doc) do
  match v0
    case v1 : book[AnyTree] do v1
    else ()
```

This algebra expression first gets all the children of the element 'doc'. The match expression is then used to select only those children whose type is an element book with any content.

As a second example, here is how a FLWR expression can be mapped into the Algebra.

```
FOR $b IN $doc/book
WHERE $b/price > 100
RETURN $b/author
```

assuming that path expressions are mapped as indicated above, this corresponds to:

```
for b in doc/book
  if empty(for v0 in b/price do (if b > 100 then b else
  ()))
  then b/author
  else ()
```

Here, note that the where clause is mapped to a conditional expression and that the predicate $b > 100$ is mapped to an iteration that performs existential quantification. This is an example of implicit semantics in XQuery which are made explicit in the mapping to the XML Query Algebra. On the other hand, the XML Query Algebra expression is significantly more verbose than the original XQuery.

1.3.2. Static Semantics

Static semantics are defined using type inference rules. Type inference rules are used to infer the type of an expression, given the type of its sub-expressions. For instance, here is the inference rule for the conditional expression.

```

Type2      Expr3: Type3      Expr1: Boolean      Expr2:
-----
| Type3)      if Expr1 then Expr2 else Expr3: (Type2

```

The statement "Data : Type" should be read as "the value Data has type Type". The statements above the rule are the conditions that must hold in order to deduce the statement below the rule. In the above example, it means that if Expr1 has type boolean, and Expr2 and Expr3 have types Type2 and Type3 respectively, then the type of the conditional expression is (Type2 | Type3). In other words, the type of a conditional expression allows either the type of the THEN expression or the type of the ELSE expression.

These rules allow to detect errors at compile time. For instance, if Expr1 above has type string instead of Boolean, then no inference rule will match and this will result in an error.

Each of the core algebra expression has a corresponding typing rule. As an additional example, here is the rule for children:

```

-----
Expr1: a[Type1]
children(Expr1): Type1

```

And here is the rule for sequence:

	Expr1: Type1	Expr2:
Type2	(Expr1, Expr2): (Type1, Type2)	

The first rule states that the type of the children of an element is the content of the corresponding type. The second rule states that the type of a sequence of expressions is the sequence of their respective types. As we have seen if these rules cannot be applied, it would result in a type error. If no error is detected, then the rules will give a type for the whole query expression. Suppose that the type for variable doc is expressed in the algebra expression as:

```
doc : book [ title [ String ],
              price [ Integer ],
              author [ String ]* ]*
```

Then the type of the XQuery above is inferred by the system as:

```
author [ String ]*
```

1.3.3. Dynamic Semantics

Just as the static dynamics provides a set of inference rules that compute type, the dynamic semantics provides a set of inference rules that compute values. This is done at the time that the query is executed; in fact, the dynamic semantics can be seen as a formal specification for computation of query results. The inference rules for dynamic semantics are similar to those for the static semantics. For instance, the following rule evaluates a conditional expression. If the conditional's boolean expression Exp1 evaluates to true, Exp2 is evaluated and its value is produced. If the conditional's boolean expression evaluates to false, Exp3 is evaluated and its value is produced.

E - Exp1 => true	E - Exp2 => v2
E - if Exp1 then Exp2 else Exp3 => v2	E - Exp1 => false
E - if Exp1 then Exp2 else Exp3 => v3	

The discussion of the algebra in this section has been quite brief. For more information on the algebra, see the XML Query Algebra document [[W3C XML Query Algebra](#)]

1.4. Conclusion

XQuery is a small, expressive language that intelligently uses the structure of XML to allow queries on many kinds of data. Experience shows that XQuery is easy to learn and to use, and it is extremely compositional, making it easy to create very powerful queries. The Query Algebra provides a precise specification of the results of a query, and also makes it possible to discover equivalences so that queries can be rewritten flexibly based on the performance parameters of various kinds of access. In addition, it allows XQuery to be a strongly typed language.

Bibliography

[W3C XML 1.0.] World Wide Web Consortium. Extensible Markup Language (XML) 1.0. W3C Recommendation, Feb. 10, 1998. Available at "<http://www.w3.org/TR/1998/REC-xml-19980210>".

[W3C XML Query Requirements] World Wide Web Consortium. XML Query Requirements. W3C Working Draft, Feb. 15, 2001. Available at "<http://www.w3.org/TR/xmlquery-req>".

[W3C XML Query Use Cases] World Wide Web Consortium. XML Query Use Cases. W3C Working Draft, Feb. 15, 2001. Available at "<http://www.w3.org/TR/xmlquery-use-cases>".

[W3C XML Query Data Model] World Wide Web Consortium. XML Query Data Model. W3C Working Draft, Feb. 15, 2001. Available at "<http://www.w3.org/TR/query-datamodel>".

[W3C XML Query Algebra] World Wide Web Consortium. XML Query Algebra. W3C Working Draft, Feb. 15, 2001. Available at "<http://www.w3.org/TR/query-algebra>".

[XQuery: A Query Language for XML] World Wide Web Consortium. XQuery: A

Query Language for XML. W3C Working Draft, Feb. 15, 2001. Available at "<http://www.w3.org/TR/xquery>".

[MSL- A Model for W3C XML Schema] ##### Available at "<http://cm.bell-labs.com/cm/cs/who/wadler/papers/msl/msl.pdf>".

[The Object Database Standard: ODMG-93] Rick Cattell et al. Release 1.2. Morgan Kaufmann Publishers, San Francisco, 1996.

[Quilt: an XML Query Language for Heterogeneous Data Sources] Available at "http://www.almaden.ibm.com/cs/people/chamberlin/quilt_Incs.pdf".

[SQL] International Organization for Standardization (ISO). Information Technology-Database Language SQL. Standard No. ISO/IEC 9075:1999. (Available from American National Standards Institute, New York, NY 10036, (212) 642-4900.)

[W3C XPath 2.0 Requirements] Available at "<http://www.w3.org/TR/xmlquery-req###>".

[XML Query Language (XQL)] Available at "<http://www.w3.org/TandS/QL/QL98/pp/xql.html>".

[Wadler2000] Philip Wadler, Avaya Labs. "Proofs are Programs: 19th Century Logic and 21st Century Computing". Available at "<http://cm.bell-labs.com/cm/cs/who/wadler/papers/frege/frege.pdf>".

[A Query Language for XML] Alin Deutsch, Mary Fernandez, Daniela Florescu, Alon Levy, and Dan Suciu. Available at "<http://www.research.att.com/~mff/files/final.html>"

[W3C XPath 1.0] World Wide Web Consortium. XML Path Language (XPath) Version 1.0. W3C Recommendation, Nov. 16, 1999. Available at "<http://www.w3.org/TR/xpath.html>".

["Survey shows XML use growing fast in enterprises."] Network World Fusion. Available at "<http://www.nwfusion.com/news/2001/0226xml.html>".

[Lorel] Serge Abiteboul, Dallan Quass, Jason McHugh, Jennifer Widom, and Janet L. Wiener. The Lorel Query Language for Semistructured Data. International Journal on Digital Libraries, 1(1):68-88, April 1997. See "<http://www-db.stanford.edu/~widom/pubs.html>"

[YATL] S. Cluet, S. Jacqmin, and J. Simeon. The New YATL: Design and Specifications. Technical Report, INRIA, 1999.

Biography

Paul **Cotton**

Microsoft
Canada

Paul Cotton - Paul Cotton joined Microsoft in May 2000 and is currently Program Manager of XML Standards. Paul has 28 years of experience in the Information Technology industry. He has been involved in computer standards work since 1988 when he began representing Fulcrum Technologies in the SQL Access Group where he was heavily involved in the development of the SQL Call-Level Interface (CLI), which is the de jure standard based on ODBC. Paul has represented his employer and Canada on the ISO SQL and SQL/Multi-Media committees since 1992 and was the Editor of the SQL/MM Full-Text and Still Image documents from 1995 until joining Microsoft. Paul was a founding member in the consortium efforts to standardize JDBC and SQLJ, which provide interfaces to SQL for the Java language. Paul has been participating in the W3C XML Activity since mid-1998 when he became IBM's prime representative on the XML Linking and Infoset Working Groups. Paul has been chairperson of the XML Query Working Group and a member of the XML Coordination Group since September 1999. Paul was elected to the W3C Advisory Board in June 2000 soon after joining Microsoft. Paul is also Microsoft's alternate on the XML Protocol Working Group.

Jonathan **Robie**

Software AG
USA

Jonathan Robie - Jonathan Robie is an XML Research Specialist at Software AG. He is an editor of XQuery, the W3C XML Query Language, and also an editor of the W3C XML Query Requirements, Use Cases, and Data Model. He was also a co-author of two earlier XML query languages: Quilt, a precursor of XQuery, and

XQL, a precursor of XPath. In addition to his work on XML query languages, he is a member of the W3C XML Schema Working Group, where he is an editor of the Schema Formalization document, and was an editor of the W3C Document Object Model for both Level 1 and Level 2.

Mr. Robie has been on the architectural team for XML databases or repositories at three different companies - Software AG, Texcel Research, and POET Software. He has been a regular speaker at SGML and XML conferences since 1996.

Prior to his work with XML, Mr. Robie was an object database specialist at POET Software, where he implemented transactions in the kernel, designed and presented workshops on object database programming, and provided key-customer support.