
MarkLogic Server

Introduction to XQuery

Release 3.2
May, 2007

Copyright

© Copyright 2002-2007 by Mark Logic Corporation. All rights reserved worldwide.

This Material is confidential and is protected under your license agreement.

Excel and PowerPoint are either registered trademarks or trademarks of Microsoft Corporation in the United States and/or other countries. This document is an independent publication of Mark Logic Corporation and is not affiliated with, nor has it been authorized, sponsored or otherwise approved by Microsoft Corporation.

Contains LinguistX, from Inxight Software, Inc. Copyright © 1996-2006. All rights reserved. www.inxight.com.

Antenna House OfficeHTML Copyright © 2000-2006 Antenna House, Inc. All rights reserved.

Argus Copyright ©1999-2005 Icen Technology Ltd. All rights reserved.

Table of Contents

Introduction to XQuery

Copyright	2
1.0 Acknowledgements	5
2.0 Introduction	6
2.1 Objectives	6
2.2 Audience	6
2.3 Scope	6
2.4 Requirements	6
3.0 History	7
3.1 Motivations for XQuery	7
3.2 Roots: Leveraging What Came Before	7
3.2.1 XPath and XQL	8
3.2.2 XML-QL	8
3.2.3 SQL and OQL	9
4.0 XQuery Basics	10
4.1 XQuery Expressions	10
4.2 Element Constructors	12
4.3 FLWOR Expressions	13
4.3.1 The for Clause	13
4.3.2 The let Clause	15
4.3.3 The where Clause	16
4.3.4 The order by Clause	17
4.3.5 The return Clause	18
4.3.6 Summary of FLWOR Data Flow	18
4.4 The typeswitch Expression	19
4.5 The if Expression	20
5.0 Applied XQuery	22
5.1 An Example XML Document	22
5.2 Simple XQuery programs	23
5.3 Nested FLWOR expressions	24
5.4 Embedding Conditional Expressions Within FLWOR Expressions	26
5.5 Quantified Expressions	28

5.6	Making Decisions Based on Content Order	29
5.7	Defining and Using Functions	31
5.7.1	Defining Functions	31
5.7.2	Simple Function Usage	32
5.7.3	Using Recursive Functions	34
5.8	Catching Exceptions	35
6.0	Namespaces	37
6.1	How Namespaces Work	37
6.2	Using Namespaces in XQuery	39
6.3	Namespaces in MarkLogic Server	40
6.4	Juggling Multiple Namespaces to Write Web Applications	41
6.5	Recommended Practice with Multiple Namespaces	43
7.0	XML and SQL	49
7.1	XML Views of Relational Data	49
7.1.1	A Simple Example	49
7.2	SQL vs. XQuery	50
7.2.1	A Simple Query	50
7.2.2	GROUP BY and HAVING	50
7.2.3	Inner Joins	51
7.2.4	Outer Joins	51
7.3	SQL to XQuery, not XQuery to SQL	52
8.0	Learning the Language	53
8.1	XQuery Use Cases	53
8.2	Digging Deeper	53
8.3	Additional Resources	54
9.0	References	55
9.1	References Specific to MarkLogic Server	55
9.2	General References	55
	Technical Support	56

1.0 Acknowledgements

Portions of this document are adapted with permission from the presentation “A Quilt, not a Camel”, by Don Chamberlin, Jonathan Robie and Daniela Florescu, dated May 19, 2000.

While many individuals and their associated companies and institutions have contributed to the development of the XQuery draft standard through their participation in W3C XQuery Working Group, the original Quilt proposal is the most direct predecessor of the XQuery effort. Almost two years later, “A Quilt, not a Camel” still contains the most concise and effective presentation of the origins, motivations and basic concepts that lie behind much of today’s XQuery standard.

2.0 Introduction

XQuery is a powerful new query language specifically designed for posing queries against XML data sets. The XQuery specification is under active development by the W3C XQuery Working Group.

2.1 Objectives

This document provides an introduction to the XQuery language, describing its origins and some of the basic language features.

2.2 Audience

This introductory document is intended for a technical audience, specifically developers with good working knowledge of XML and of the XPath standard. The document assumes a basic understanding of database concepts, but does not require direct experience with SQL queries.

2.3 Scope

This document provides a general introduction to the XQuery language. For the most part, this document's content is implementation-independent.

2.4 Requirements

There are no pre-requisites to reading this document.

To be able to experiment with the code samples provided in this document, you will need to have previously installed the MarkLogic Server software. If you have not as yet installed this software and wish to do so, see the books entitled *Getting Started with MarkLogic Server* and *Installation Guide*.

3.0 History

The World Wide Web has made a vast assortment of information available to any person at any time from just about any location.

By putting publishing in the hands of any end user – be they an individual, an institution or an enterprise—the web has enabled the creation of an enormous, unstructured, worldwide dataset. The combination of information available through public and private web presences dwarfs any collection of information ever assembled by humankind.

In the midst of this massively distributed publishing ecosystem, XML has emerged as the leading candidate for a universal language for information interchange. Its popularity has grown both as a standard format for structuring complex document-oriented data and as a standard interchange format for inter-application communication.

This chapter describes some of the history of how XQuery came to be, and includes the following sections:

- [Motivations for XQuery](#)
- [Roots: Leveraging What Came Before](#)

3.1 Motivations for XQuery

The originators of XQuery strongly believed that in order to realize its full potential, XML needed to be matched with a query language of comparable power and flexibility. By early 2000, a number of XML query languages had been proposed or implemented, including XPath, XQL, XML-QL, Lorel and YATL. Most of these languages are oriented towards a specific problem domain, such as working with semi-structured documents, or with well-structured databases.

The goals of the XQuery language development effort are simply:

- to leverage the most effective features of the existing and proposed query languages of the time,
- to design a small, clean, implementable language,
- to cover the functionality required by the complete set of XML Query use cases in a single language, and
- to be able to write concise, tight, understandable queries.

3.2 Roots: Leveraging What Came Before

The first priority of the draft recommendation was not to reinvent the wheel, but rather to take advantage of the best features of the work that had already been completed by a variety of efforts. One of the earliest challenges facing the XQuery Working Group was determining what was best about each these efforts and deciding how to weave these features into a single coherent proposal.

3.2.1 XPath and XQL

XPath and XQL are closely-related languages that address the specific issue of navigating in a hierarchical data model.

Their basic unit of expression is the *path expression*, which is treated as a series of *steps*. Each step moves along a particular *axis* in the data (towards children, ancestors, attributes, etc.) and may apply a *predicate* to qualify the specific axis to traverse.

XPath has an abbreviated syntax, which is adapted from XQL's model:

```
/book[title = "War and Peace"]  
  /chapter[title = "War"]  
    //figure[contains(caption, "Guns")]
```

This example specifies a particular node or nodes in the XML document, as follows:

1. Starting at the top of the document hierarchy, select all book nodes that have a title element whose value is “War and Peace”.
2. Within these book nodes (admittedly, there is likely only one such node), step into the chapter nodes that have a title element whose value is “War”.
3. Within those chapter nodes, select all the figure nodes – at any level below the chapter node – which have a caption element containing the word “Guns”.

As this simple example demonstrates, XPath provides a powerful and concise specification for traversing hierarchical data models. XQL has some additional operators of interest, including the BEFORE and AFTER operators.

3.2.2 XML-QL

The XML-QL language was originally proposed by Alin Deutsch, Mary Fernandez, Daniela Florescu, Alon Levy and Dan Suciu.

Of particular note in the XML-QL proposal are the WHERE and CONSTRUCT clauses. The WHERE clause binds variables to specific parts of XML documents according to a pattern specification. The CONSTRUCT clause provides a general-purpose specification of the output document that is to be generated.

The following example shows how WHERE and CONSTRUCT are used:

```
WHERE
  <part pno = $pno> $pname </> in "parts.xml",
  <supplier sno = $sno> $sname </> in "suppliers.xml",
  <sp pno = $pno sno = $sno> </> in "sp.xml"
CONSTRUCT
  <purchase>
    <partname> $pname </>
    <suppliername> $sname </>
  </purchase>
```

For a given part number `$pno` and a given supplier number `$sno`, this example:

1. binds `$pname` to the name of the part specified,
2. binds `$sname` to the name of the supplier specified, and
3. generates a purchase document for any cases in which the given part is supplied by the given supplier, based on the supplier-parts list provided in the document `sp.xml`.

3.2.3 SQL and OQL

SQL and OQL are powerful database query languages. SQL is the standard for expressing queries across relational databases. OQL maps relational queries into a functional language better suited for tight integration with object-oriented programming languages.

SQL queries derive result sets from other tables by specifying a stylized series of clauses: SELECT... FROM... WHERE...

OQL is a functional language, in which a query is an expression. Expressions can take several forms, including the SELECT... FROM... WHERE... form. Expressions can be freely nested and combined.

4.0 XQuery Basics

XQuery is a powerful and concise language.

Imagine you were working with a set of XML documents that describe parts and list orders, and that you wanted to express a query to find the description and average price of each red part that has at least 10 orders outstanding.

Using XQuery, you could write the following:

```
<example>

{

  for $part in doc("parts.xml")/parts/part[color = "red"]
  let $order := doc("orders.xml")/orders/order[partno = $part/partno]
  where count($order) >= 10
  return
    <important_red_part>
      { $part/description }
      <avg_price>{ avg($order/price) }</avg_price>
    </important_red_part>

}

</example>
```

This section uses the query above to highlight some basic concepts of the XQuery language, and contains the following sections:

- [XQuery Expressions](#)
- [Element Constructors](#)
- [FLWOR Expressions](#)
- [The typeswitch Expression](#)
- [The if Expression](#)

4.1 XQuery Expressions

Like OQL, XQuery is a functional language. Queries are expressions, and expressions can be combined, creating extremely powerful queries. An introductory list of XQuery expressions includes:

- Path expressions, which use abbreviated XPath syntax and are detailed extensively in the W3C XPath Standards document (see “References” on page 55):

```
bids/bid[itemno = "47"]/bid_amount
```

- Expressions that use operators and functions:

```
($base + $options) * tax-rate($state)
```

- Element constructors, which can combine declarative XML structure with embedded expressions:

```
<bid>
  <itemno>{ $i }</itemno>
  <userid>{ $u }</userid>
  <bid_amount>{ $a }</bid_amount>
</bid>
```

- For-let-where-order-by-return expressions, colloquially referred to as “FLWOR” expressions:

```
for $part in doc("parts.xml")/parts/part[color = "red"]
let $order := doc("orders.xml")/orders/order[partno = $part/partno]
where count($order) >= 10
order by $part/partno
return
  <important_red_part>
    { $part/description }
    <avg_price>{ avg($order/price) }</avg_price>
  </important_red_part>
```

- Typeswitch, which permits conditional evaluation based on expression type:

```
typeswitch ($input)
case $a as element(*, USAddress) return $a/state
case $a as element(*, CanadaAddress) return $a/province
case $a as element(*, JapanAddress) return $a/prefecture
default return "unknown"
```

- If expressions, which conditionally evaluate one of two sub-expressions:

```
if ($answer < 42 or $answer > 42)
then
  <correct>false</correct>
else
  <correct>true</correct>
```

4.2 Element Constructors

A key building block in XQuery is the ability to natively embed XML structures inline. Just as other programming languages can apply operators and functions natively to types such as integers, characters and strings, XQuery allows query developers to work directly with XML structures.

Element constructors allow you to embed XML structures directly into your XQuery programs. For example, the following XML represents a perfectly good XQuery expression:

```
<bid>
  <itemno>47</itemno>
  <userid>tsmith</userid>
  <bid_amount>34.25</bid_amount>
</bid>
```

Frequently, however, the contents of the XML structure need to be constructed dynamically. In this case, XQuery expressions can be embedded in the element constructor by surrounding them with curly braces.¹ This is how element constructors are most frequently utilized in the return clause of a FLWOR expression:

```
<bid>
  <itemno>{ $i }</itemno>
  <userid>{ $u }</userid>
  <bid_amount>{ $a }</bid_amount>
</bid>
```

The embedded XQuery can be any valid XQuery expression, which means that element constructors can get quite complex, with embedded FLWOR expressions, nested constructors, and more.

1. In XQuery, curly braces are used for mode switching between declarative XML structures and XQuery expressions to be evaluated. There is no explicit or implicit scoping associated with curly braces as there is in many other languages. Curly braces are also used in the definition of functions, as detailed in section 4.

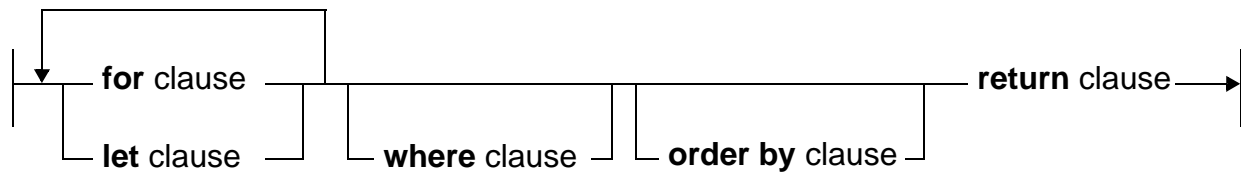
The power of element constructors lies in their ability to be used anywhere an expression is expected:

```
if (deep-equal($author, <author>
    <last>{ $lastname }</last>
    <first>{ $firstname }</first>
</author>)) then .....
```

4.3 FLWOR Expressions

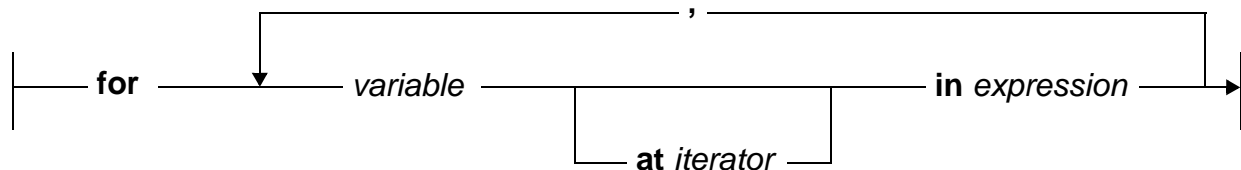
The heart of XQuery is the FLWOR expression. A FLWOR expression binds variables, applies a predicate, orders the data set, and constructs a new result:

We will now examine each of these five clause types in more detail.



4.3.1 The for Clause

The `for` clause is used for iterating over one or more sequences:



The `for` clause declares an ‘implicit iteration’, because each *variable* is understood to be bound in order to the successive values in the corresponding *expression* sequence. The variables appearing in the `for` clause are bound sequentially (for example, for documents, in depth-first order; for sequences, from left to right).

Each *expression* evaluates to a sequence of nodes. For example, the expression in the following `for` clause evaluates to all `<part>` nodes in the `parts.xml` document that have a child element `<color>` whose value is “red”:

```
for $part in doc("parts.xml")/parts/part[color = "red"]
```

The complete `for` clause creates a sequence of tuples that are passed for evaluation into the subsequent clauses of the FLWOR expression. In the example above, the tuples bind the variable `$part` to a `<part>` node and its descendents. There will be as many tuples as there are `<part>` nodes that match the XPath expression. Tuples are ordered based on the depth-first order in which `<part>` nodes occur in the document `parts.xml`.

As specified in the syntax diagram above, `for` clauses can have multiple bindings:

```
for $individual in /students/student,
    $course in /curriculum/course,
    $teacher in /faculty/instructor
```

In this case:

- `$individual` is bound in depth-first order to the sequence of `<student>` nodes,
- `$course` is bound in depth-first order to the sequence of `<course>` nodes, and
- `$teacher` is bound in depth-first order to the sequence of `<instructor>` nodes.

The subsequent clauses in the FLWOR expression are passed a set of binding-tuples based on the Cartesian product of these sequences:

```
($individual = <student>1, $course = <course>1, $teacher =
<instructor>1)
($individual = <student>1, $course = <course>1, $teacher =
<instructor>2)
....
($individual = <student>1, $course = <course>1, $teacher =
<instructor>n) ($individual = <student>1, $course = <course>2,
$teacher = <instructor>1)
($individual = <student>1, $course = <course>2, $teacher =
<instructor>2)
....
($individual = <student>1, $course = <course>2, $teacher =
<instructor>n)
....
($individual = <student>1, $course = <course>m, $teacher =
<instructor>1)
....
($individual = <student>1, $course = <course>m, $teacher =
<instructor>n)
....
($individual = <student>p, $course = <course>m, $teacher =
<instructor>n)
```

Each successive *expression* may refer to previously declared variables. For example,

```
for $a in doc("mydoc.xml")/a[@id > 10],
    $b in $a/b[@id > 100]
return $b/c
```

is equivalent to the single path expression:

```
doc("mydoc.xml")/a[@id > 10]/b[@id > 100]/c
```

The optional *iterator* variable allows a bound variable to be related to its indexed order in the sequential list:

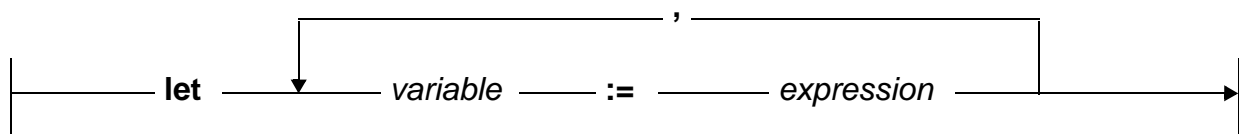
```
for $part at $index in doc("parts.xml")/parts/part[color = "red"]
```

In the example above, *\$index* is bound to the position in the sequence of part elements which is returned. The first part returned is bound to an *\$index* with value 1, and the last to an *\$index* with value *n*, where *n* is the number of matching parts.

The order of the tuples preserves document order unless some *expression* contains a non-order-preserving function such as `distinct-nodes()` or `distinct-values()`.

4.3.2 The let Clause

The `let` clause is used for binding variables (without iteration) to single values or sequences of values:



A `let` clause produces *one* binding for each variable. Consequently, `let` clauses do not affect the number of binding tuples evaluated in a FLWOR expression.

The *variable* is bound to the value of *expression*, which may be a single value or an entire sequence of values.

From the sample query describe above, the `let` clause augments each tuple with additional data:

```
for $part in doc("parts.xml")/parts/part[color = "red"]
let $order := doc("orders.xml")/orders/order[partno = $part/partno]
```

The `for` clause defines a sequence of `<part>` nodes, based on the conditions encapsulated in its *expression*:

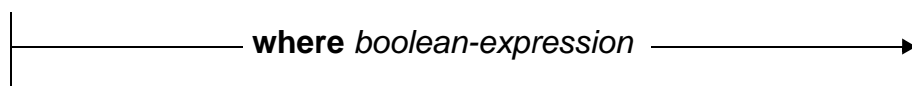
```
($part = <part>1)
($part = <part>2)
....
($part = <part>n)
```

The `let` clause augments these tuples with additional bindings, without increasing the total number of tuples. The bindings may be a single value, or an entire sequence of values. In the case of our example, each part may be featured in zero, one or multiple orders:

```
($part = <part>1, $order = (<order>12, <order>19, ....))
($part = <part>2, $order = (<order>4))
($part = <part>3, $order = ())
....
($part = <part>n, $order = (<order>3, <order>25, ....))
```

4.3.3 The where Clause

The `where` clause specifies a filter condition on the tuples emerging from the `for-let` portion of a FLWOR expression:



Only tuples for which the *boolean-expression* evaluates to *true* will contribute to the result sequence of the FLWOR expression. The `where` clause preserves the order of tuples, if any. *boolean-expression* may contain `and`, `or` and `not`, among other operators.

Where clauses apply scalar conditions to scalar variables:

```
$color = "red"
```

Where clauses can also apply set conditions to variables that are bound to sets:

```
avg($emp/salary) > 10000
```

From the example above:

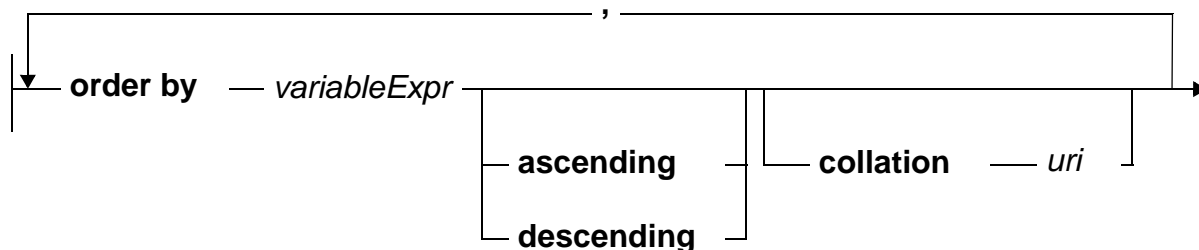
```
for $part in doc("parts.xml")/parts/part[color = "red"]
let $order := doc("orders.xml")/orders/order[partno = $part/partno]
where count($order) >= 10
```

only those tuples with more than ten `<order>` nodes in the `$order` sequence will contribute to the result sequence:

```
($part = <part>_1, $order = (<order>_12, <order>_19, ...))
($part = <part>_2, $order = (<order>_4))
($part = <part>_3, $order = ())
....
($part = <part>_n, $order = (<order>_3, <order>_25, ...))
```

4.3.4 The order by Clause

The `order by` clause specifies the order (ascending or descending) to sort items returned from a FLWOR expression, and also provides an option to specify a collation URI with which to determine the order:



The `order by` clause can be used to specify an order in which the tuple sequence will be passed to the return clause:

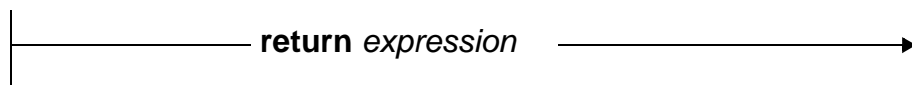
```
order by $auction_item/itemno
return
  <auction_item>
    { $auction_item/itemno }
    <avg_bid>{ avg($b/bid_amount) }</avg_bid>
  </auction_item>
```

The `order by` clause can specify any sort key, regardless of whether that sort key is contained in the result sequence. Sequences can be reordered on an ascending or descending basis:

```
order by $auction_item/itemno descending
return
  <auction_item>
    { $auction_item/itemno }
    <avg_bid>{ avg($b/bid_amount) }</avg_bid>
  </auction_item>
```

4.3.5 The return Clause

The `return` clause constructs the result of a FLWOR expression:



The return expression is evaluated once for each tuple of bound variables. This evaluation preserves the order of tuples, if any, or it can impose a new order using the `order by` clause.

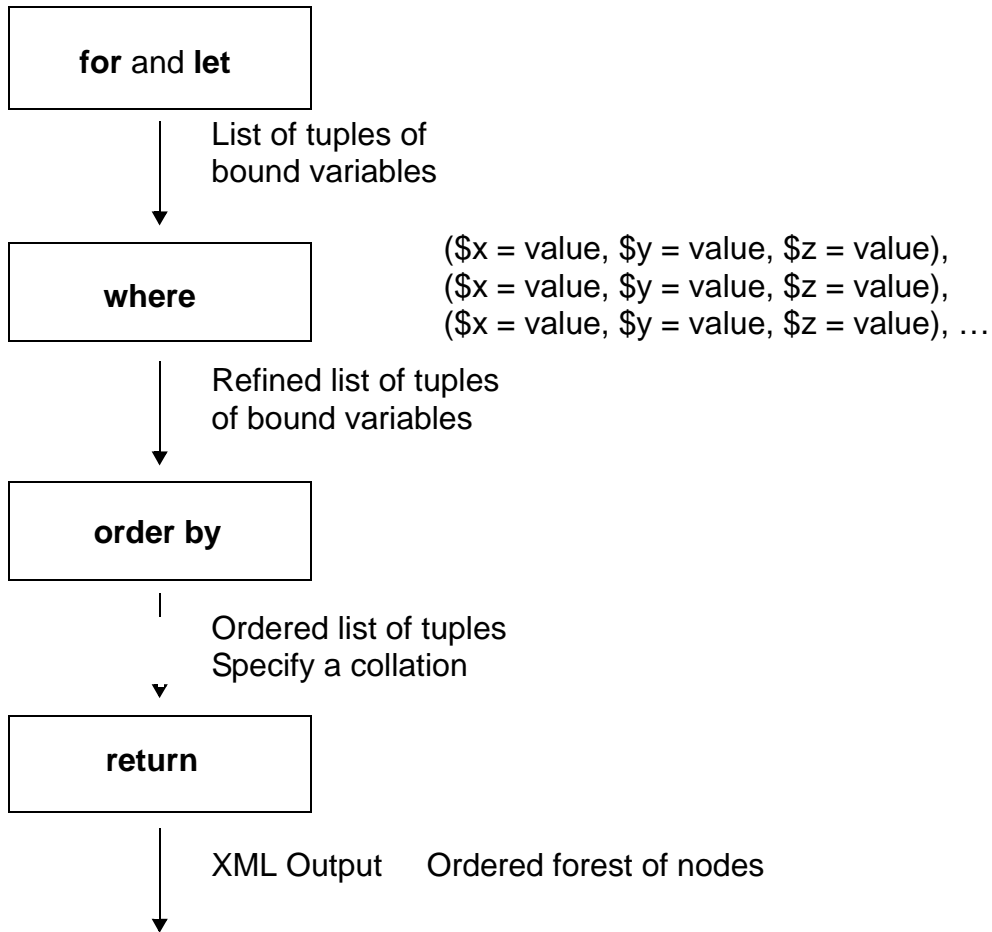
Often, the `return` clause uses an element constructor:

```
return
  <auction_item>
    { $auction_item/itemno }
    <avg_bid>{ avg($b/bid_amount) }</avg_bid>
  </auction_item>
```

Because `return` specifies an expression, any legal XQuery expression can be used to construct the result, including another FLWOR expression.

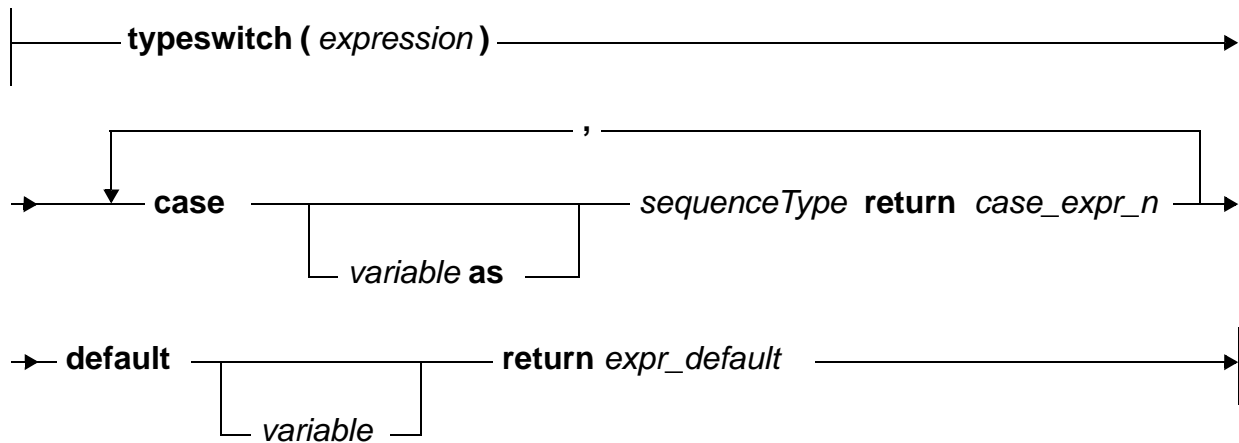
4.3.6 Summary of FLWOR Data Flow

Data flows through a FLWOR expression as shown in the following figure.



4.4 The typeswitch Expression

The `typeswitch` expression allows conditional evaluation of a set of sub-expressions based on the type of a specified expression:



A `typeswitch` expression evaluates the first `case_expr` whose `sequenceType` matches the type of the specified `expression`. If there is no `sequenceType` match, `expr_default` is evaluated.

Typeswitch provides a powerful mechanism for processing node contents:

```
typeswitch ($address)
  case $a as element(*, USAddress) return handleUS($a)
  case $a as element(*, CanadaAddress) return handleCanada($a)
  default return handleUnknown($address)
```

This code snippet determines the `sequenceType` of the variable `$address`, then evaluates one of three sub-expressions. In this case:

- If `$address` is of type `USAddress`, the function `handleUS($a)` is evaluated.
- If `$address` is of type `CanadaAddress`, the function `handleCanada($a)` is evaluated.
- If the type of variable `$address` matches none of the above, the function `handleUnknown($a)` is evaluated.

It is possible to construct `case` clauses in which a particular `expression` matches multiple `sequenceTypes`. In this case, the `case_expr` of only the first matching `sequenceType` is evaluated.

4.5 The if Expression

The `if` expression allows conditional evaluation of sub-expressions:

```
|————— if expr_c1 then expr_r1 else expr_r2 —————>|
```

If expression `expr_c1` evaluates to true, then the value of the `if` expression is the value of expression `expr_r1`, otherwise the value of the `if` expression is the value of `expr_r2`. The `else` clause is not optional – if no action is to be taken, a null sequence should be used for `expr_r2`. There is no “end if” or similar construct in XQuery.

The extent of `expr_r1` and `expr_r2` is limited to a single expression. If a more complex set of actions are required, an element constructor, sequence, or function call must be used.

If expressions can be nested:

```
if ($year < 1994)
then
  <available>archive</available>
else if ($year = $current_year) then
  <available>current</available>
else
  <available>inventory</available>
```

5.0 Applied XQuery

The previous section introduced some of the most critical constructs in XQuery. Many of the others are introduced in this section through concrete examples. For an understanding of complete set of XQuery expressions and a formal explanation of the language grammar, readers should consult the current version of the XQuery Working Group’s Draft Recommendation (see “References” on page 55).

This section draws upon a number of the formal use cases that define key aspects of XQuery functionality. These use cases are detailed in the XQuery Use Cases (see “References” on page 55) document, and are installed as part of the MarkLogic Server installation process. The following sections are included:

- [An Example XML Document](#)
- [Simple XQuery programs](#)
- [Nested FLWOR expressions](#)
- [Embedding Conditional Expressions Within FLWOR Expressions](#)
- [Quantified Expressions](#)
- [Making Decisions Based on Content Order](#)
- [Defining and Using Functions](#)
- [Catching Exceptions](#)

5.1 An Example XML Document

Many of these examples work with a hypothetical XML document named `bib.xml`, which has the following structure:

```
<bib>
  <book>
    <title>      .... </title>
    <author>     .... </author>
    ....
    <publisher> .... </publisher>
    <year>       .... </year>
    <price>      .... </price>
  </book>
  <book>
    ....
  </book>
  ....
</bib>
```

5.2 Simple XQuery programs

Given the structure of `bib.xml`, imagine you wanted to find all the books published in 1998 by Penguin. The following XQuery program would resolve that query, returning the matching sequence of book nodes sorted by title within author:

```
for $b in /bib/book
where $b/year = "1998" and $b/publisher = "Penguin"
order by $b/author, $b/title
return
  $b
```

This query uses a single FLWOR expression. The optional `let` clause is omitted. The optional `order by` clause is included to override the default document order of the query response.

If instead you wanted to find all the books that specify no authors and return those books' titles:

```
<orphan_books>
{
  for $b in /bib/book
  where empty($b/author)
  order by ($b/title)
  return
    $b/title
}
</orphan_books>
```

This query is formed of a pair of nested expressions. The query itself is composed of an element constructor – the outer `<orphan_books>` declaration. The curly braces are used to escape from XML back into XQuery mode, whereupon a FLWOR expression is evaluated. The result of this nested expression is the generation of a completely new XML structure, with the following format:

```
<orphan_books>
  <title>      .... </title>
  <title>      .... </title>
  <title>      .... </title>
  ....
</orphan_books>
```

The generated output is sorted alphabetically by title.

Both examples have demonstrated results that incorporate XML elements from the queried documents. Sometimes it is important to be able to generate output that incorporates not entire XML elements, but only values from those elements.

The following query remaps `<title>` elements into `<booktitle>` elements on output generation:

```

<orphan_books>
{
  for $b in /bib/book
  where empty($b/author)
  order by $b/title
  return
    <booktitle>{ $b/title/text() }</booktitle>
}
</orphan_books>

```

This query generates output in the following form:

```

<orphan_books>
  <booktitle>      .... </booktitle>
  <booktitle>      .... </booktitle>
  <booktitle>      .... </booktitle>
  ....
</orphan_books>

```

The `text()` filter in the XPath expression `$b/title/text()` maps to a sequence of text values for all matching `<title>` elements. In this case, since there is only one `<title>` element in each specified `$b` node, `$b/title/text()` simply returns the name of the current book `$b`.

It is instructive to review these three queries to understand their underlying construction. The queries are quite similar, and yet they demonstrate the variety of ways in which you can use different types of XQuery expressions:

- The first query is a simple FLWOR expression, with the return clause incorporating the simplest of XPath expressions.
- The second query is an element constructor expression containing a FLWOR expression that returns a slightly more complex XPath expression, thereby generating a new XML structure.
- The final query is an element constructor expression that contains a FLWOR expression that returns an element constructor expression that contains an even more complex XPath expression.

5.3 Nested FLWOR expressions

As you start to create more powerful queries that work with more complex XML structures, one of the first things you will likely need to do is nest FLWOR expressions.

Imagine you want to create a matching document to the “bib.xml” document, but rather than having `<book>` nodes as the primary nodes of the document, you want to have `<publisher>` nodes as the primary nodes.

To do this, you need to invert the “bib.xml” document. The following query achieves this goal:

```
<catalog>
{
  for $p in distinct-values(/bib/book/publisher)
  order by $p
  return
    <publisher>
      <name>{ $p/text() }</name>
      {
        for $b in /bib/book[publisher = $p]
        order by $b/price descending
        return
          <book>
            { $b/title }
            { $b/price }
          </book>
      }
    </publisher>
}
</catalog>
```

This query generates output in the following form:

```
<catalog>
  <publisher>
    <name> .... </name>
    <book>
      <title> .... </title>
      <price> .... </price>
    </book>
    ....
  </publisher>
  ....
</catalog>
```

The query works in the following way:

1. The outer FLWOR expression produces a sequence of unique `<publisher>` elements, relying on the `distinct-values()` function to eliminate redundant entries in the sequence.
2. The `publisher` nodes are sorted alphabetically by the publisher’s name.

3. For each `<publisher>` element in that sequence, the inner FLWOR expression produces a sequence of `<book>` nodes, selecting only those books whose `<publisher>` element matches the current `<publisher>` element in the sequence.
4. For that subset of books, the inner FLWOR returns the book's title and its price, and sorts the resulting output XML by `<price>` element so that the highest-priced book comes first in each publisher's extracted XML.

The inner FLWOR expression uses the **for** clause:

```
for $b in /bib/book[publisher = $p]
```

to select all `<book>` nodes from the current document which have a `<publisher>` child element whose value is equal to `$p`.

This clause is semantically equivalent to the following **for-where** clause:

```
for $b in /bib/book
where $b/publisher = $p
```

Robust XQuery implementations (such as MarkLogic Server) will optimize the two different constructs into exactly equivalent evaluation strategies.

5.4 Embedding Conditional Expressions Within FLWOR Expressions

In some situations, you will want the results returned by a FLWOR expression to be formatted differently based on the content of nodes being evaluated. To do this, you can embed **if** expressions within the **return** clause of a FLWOR expression.

Consider an XML document which describes the holdings of a library – a variant on the “bib.xml” document introduced earlier. The holdings are comprised of books and journals – and in this simplistic view of the world, journals have editors instead of authors.

We want to create an extract of this XML document that includes the appropriate content based on the type of holding – be it book or journal:

```

for $h in /library/holding
order by $h/title
return
  <holding>
    { $h/title }
    {
      if ($h/@holding_type = "Journal")
      then
        $h/editor
      else
        $h/author
    }
  </holding>

```

Using XPath's @ syntax, the `if` expression evaluates the `holding_type` attribute of the current `<holding>` node to see if it is a journal, and if so, returns the `<editor>` node rather than the `<author>` node.

The XML structure generated by this query will contain a sequence of `<holding>` nodes, each of which has a `<title>` node, and either an `<editor>` or an `<author>` node, depending on the type of holding.

Suppose instead the goal had been to normalize the output. In this case, all `<holding>` nodes in the generated output should have simply `<title>` and `<author>` children. For journal holdings, the `<author>` element's value should be mapped directly from the corresponding `<editor>` element in the source document. To achieve this goal, the following query is a good start:

```

for $h in /library/holding
order by $h/title
return
  <holding>
    { $h/title }
    {
      if ($h/@type = "Journal")
      then
        <author>{ $h/editor/text() }</author>
      else
        $h/author
    }
  </holding>

```

This query assumes that there is only one `<editor>` node per holding.

Given that many books have more than one author, the latter assumption seems unwise. Consequently, let us rewrite the query as follows:

```

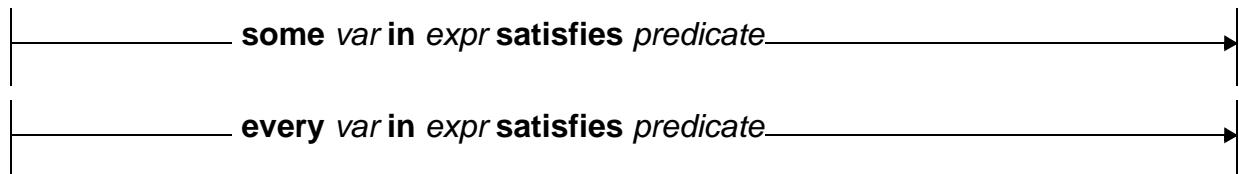
for $h in /library/holding
order by $h/title
return
  <holding>
    { $h/title }
    {
      if ($h/@type = "Journal")
      then
        for $e in $h/editor
        return
          <author>{ $e/text() }</author>
      else
        $h/author
    }
  </holding>

```

If the `<author>` node is not a leaf node (e.g. `<author>` nodes contain `<lastname>` and `<firstname>` elements), this query will need to be further refined to generate the proper output.

5.5 Quantified Expressions

XQuery provides predicates that simplify the evaluation of quantified expressions. The basic syntax for these expressions follows:



These expressions are particularly useful when trying to select a node based on a condition satisfied by at least one or alternatively all of a particular set of its children.

Imagine an XML document containing log messages. The document has the following structure:

```

<log>
  <event>
    <program>    .... </program>
    <message>    .... </message>
    <level>      .... </level>
    <error>
      <code>      .... </code>
      <severity>  .... </severity>
      <resolved>  .... </resolved>
    </error>
    <error>
      ....
    </error>
    ....
  </event>
  ....
</log>

```

Every `<event>` node has `<program>`, `<message>`, and `<level>` children. Some `<event>` nodes have one or more `<error>` children.

Consider a query to report on those events that have unresolved errors:

```

for $event in /log/event
where some $error in $event/error satisfies $error/resolved = "false"
return
  $event

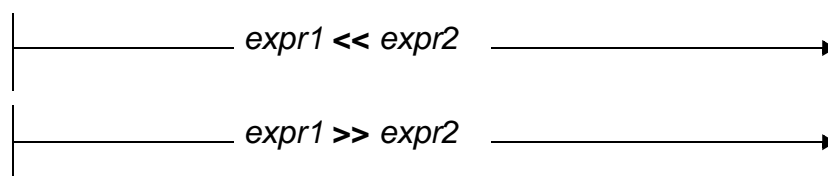
```

This query returns only those `<event>` nodes in which there is an `<error>` node with a `<resolved>` element whose value is “false”.

5.6 Making Decisions Based on Content Order

The concept of content order is fundamental to XQuery. Up to this point, we have worked with XML content in document order, and shown how to re-order result sequences using the `order by` clause.

The `<<` (before) and `>>` (after) operators allow you to select content based on an ordering condition:



Expr1 and *expr2* must be either single nodes or empty sequences. If either is empty the value is the empty sequence. Otherwise the value of *expr1* << *expr2* is true exactly when the value of *expr1* is a node which precedes the node value of *expr2* in document order. The value of *expr1* >> *expr2* is true exactly when the value of *expr1* is a node which follows the node value of *expr2* in document order.

Starting with the global ordering of the document content against which *expr1* and *expr2* will be evaluated, *before* returns nodes described by *expr1* that come before some node described by *expr2*. *after* returns nodes described by *expr1* that come after some node described by *expr2*.

Consider an XML document on the subject of cardiology. Such a document might contain different types of information, including diagnosis, treatment strategies and detailed medical procedures. With such content, it may be instructive to use XQuery to search for exception cases – for example, a medical procedure in which the first incision into the patient is made before anesthesia is administered.

Let us assume that the detailed medical procedures can be identified by looking for <section> elements whose <title> child has the value “Procedure”. Let us further assume that the steps in these procedures are each formally defined in XML. Given these assumptions, we might use the following query:

```
for $proc in /book/section[title = "Procedure"]
where not (some $a in $proc//anesthesia
           satisfies $a << ($proc//incision) [1])
return $proc
```

This FLWOR expression starts by identifying a sequence of <section> nodes – those sections which are titled “Procedure”). That sequence is further qualified by the **where** clause.

Let’s take a closer look at the where clause. The first expression:

```
$proc//anesthesia
```

identifies every <anesthesia> element within the current <procedure> node. The second expression:

```
($proc//incision) [1]
```

identifies every <incision> element within the current <procedure> node, and then selects the first such node, if there is one, using standard XPath expression syntax.

Consequently, the compound expression:

```
some $a in $proc/anesthesia satisfies $a << ($proc/incision) [1])
```

filters the <procedure> nodes \$proc for those having some <anesthesia> child element preceding the first <incision> child element.

If there is no <incision> child element this expression will return the empty sequence.

With the not operator inverting the results, the **where** clause:

```
where not (some $a in $proc/anesthesia
           satisfies $a << ($proc/incision) [1])
```

evaluates to true only when there is no anesthesia applied before the first incision.

5.7 Defining and Using Functions

Good programming practice as well as the functional nature of the XQuery language demands the ability to encapsulate expressions in functions. These strongly-typed functions act as expressions, and can be used anywhere expressions are appropriate in an XQuery program, assuming proper type usage.

5.7.1 Defining Functions

Queries can define their own local functions, using syntax as follows:

```
define function function_name($var1 as type1, $var2 as type2, ....)
as return_type
{
    expression
}
```

Basic rules of thumb are:

- The `function_name` must be unique.
- Functions can take zero or more parameters, which may be strongly typed.
- Types are as specified in XML, and include such atomic types as integer, string and decimal, as well as more XML-centric types, including item, element and node.
- Function results may be strongly typed, with the type specified in the **as** clause of the function header.
- Functions can call themselves recursively, either directly or indirectly.

The following are all acceptable function declarations within the XQuery language specification:

```
define function factorial($n)
{
  typeswitch($n)
  case $a as xs:integer return
    if ($n < 0) then 0
    else if ($n = 0) then 1
    else $n * factorial($n - 1)
  default return "Runtime error: expecting integer"
}
```

This function will return a static type error if called with a non-integer argument:

```
define function factorial($n as xs:integer)
as xs:integer
{
  if ($n < 0) then 0
  else if ($n = 0) then 1
  else $n * factorial($n - 1)
}
```

This function incorporates both typed arguments and typed results:

```
define function max-length($seq as xs:string*)
as xs:integer
{
  if (empty($seq)) then 0
  else max((string-length($seq[1]), max-length($seq[2] to
count($seq))))
}
```

5.7.2 Simple Function Usage

To understand why we might want to encapsulate certain functionality in a function call, imagine an XML document that specifies authors using the following construct:

```
<author>
  <lastname>    .... </lastname>
  <firstname>   .... </firstname>
</author>
```

Any given book in the XML document may have many authors.

Consider a query in which the query writer needs to transform a set of these `<author>` nodes into a human-readable list of authors. To do this for a given `<book>` node, one might write:

```

for $author in $book/author
return
  <span>{
    if ($author/preceding-sibling::author) then
      if ($author/following-sibling::author) then
        ","
      else
        " and"
    else
      ()
  }
  { $author/firstname/text () }
  { $author/lastname/text () }
}</span>

```

This query returns a set of `` nodes that, when displayed in order, displays a list of the authors for a given book in normal English format, with commas separating individual authors and the word “and” preceding the last author in the list.

Key to this query are the **preceding-sibling** and **following-sibling** predicates in the XPath expressions evaluated by each **if** expression. These predicates select the immediately preceding and following siblings of the current `<author>` nodes – and by doing so, evaluate to true if there is such a sibling and false if there is not.

Consequently, the nested **if** expressions can be read as:

```

if this is not the first author then
  if this is not the last author then
    insert a comma (between authors)
  else
    insert an “and” (before the last author)

```

If the results of this query are directed to a web browser, the author list will correctly render in a pleasant, human-readable format. The unusual placement of some of the braces avoids unwanted white space in the XML/XHTML output – white space that will be result in undesirable spacing in the browser display.

With this functionality under our belt, consider a query in which this mapping needs to be accomplished in more than one location. Good programming practice dictates that we should not maintain the same code body in two different locations.

Consequently, we encapsulate our author transformation function into a function body as follows:

```

define function author-list($book as node)
as node*
{
  for $author in $book/author
  return
    <span>{
      if ($author/preceding-sibling::author) then
        if ($author/following-sibling::author) then
          ","
        else
          " and"
      else
        ()
    }
    { $author/firstname/text () }
    { $author/lastname/text () }
  }</span>
}

```

The function header defines a function named `author-list`, and specifies that it takes a single parameter called `$pBook` which is of type **node**. Strong type checking will result in errors if there is a type mismatch at parse- or run-time.

The `*` immediately following the type in the **as** clause specifies that the function returns a sequence of one or more items of type **node**.

The `author-list` function is an expression like any other in XQuery, and may be used in any place where it is appropriate for the expression to evaluate to a sequence of nodes. This function can then be called as needed from deeper in the query:

```

for $book in sequence-expression
return
  {
    ....
    author-list($book)
    ....
  }

```

5.7.3 Using Recursive Functions

Functional languages such as Lisp, Smalltalk and Scheme have a rich tradition of relying on recursion to accomplish the processing of data of unknown depth or length. XQuery is no different in this regard.

Consider a set of strings. Imagine that your objective is to return the “longest” item in the set. If two items are the same length, the return the first one, assuming sequence order. The following query would accomplish the trick:

```

define function longest($items as item()*)
as item()
{
  let $count := count($items)
  return
    if ($count <= 1) then $items
    else
      let $first := item-at($items, 1),
          $rest := longest(subsequence($items, 2, $count - 1))
      return
        if (string-length($first) >= string-length($rest))
        then $first
        else $rest
}

....
longest($stringSequence)
....

```

This is a classic example of sequence processing in XQuery, and maps directly to Lisp’s list processing model.

The same recursive properties, however, are frequently useful to process and parse recursive XML structures. Imagine processing an XML document that describes a complex part assembly. Such a document might describe assemblies that contain sub-assemblies that contain further sub-assemblies – and the depth of the nesting is unknown because the schema is recursive. Processing queries against such a document may require walking the XML document to unknown depths, processing parts within parts within parts. Recursive functions are essential to writing robust queries that need to deal with such documents.

5.8 Catching Exceptions

MarkLogic Server includes many extensions to the XQuery language. The try/catch extension allows you to catch and handle exceptions.

MarkLogic Server exceptions are thrown in XML format, and you can write an XPath statement on the exception if there is a particular part of the exception you want to show. The exception is bound to the variable in the `catch` clause.

```

|----- try { expression } ----- catch ( variable ) ----- { expression } -----
|

```

The following code sample uses a try/catch block to catch exceptions upon loading a document, and prints out the filename if an exception occurs.

```
try {  
    xdm:load($filename, $uri)  
}  
catch ($exception) {  
    "Problem loading file ", $filename }
```

6.0 Namespaces

One of XML's great advantages is that both the content and the structure of information stored in XML is generally human readable. XML's self-describing and plain language characteristics makes this possible.

Unfortunately, these characteristics can also introduce confusion. If someone were to hand you an arbitrary XML structure, there would be no real way to know what specific data format you have received. For example, an XML structure with that is rooted at a `<bug>` element could easily contain any of the element sets:

```
<bug><bug><bug>
  <description> <description> <species>
  <contact> <module> <common-name>
  <date> <type> <description>
  <status> <source> <range>
  <date>
  <owner>
  <history>
    <status>
    <author>
    <comment>
    <date>
```

The `<bug>` structure on the left might be used to describe software bugs at ABC Corp. The middle structure might be used for bugs that are found in software under development at XYZ Corp. And the structure on the right might be used to specify a catalog of insects—a different kind of bug entirely!

The problem is that simply by looking at the root element's `<bug>` tag, there is no way to know which structure you are dealing with. The XML solution to this problem is *namespaces*. This chapter describes namespaces and includes the following sections:

- [How Namespaces Work](#)
- [Using Namespaces in XQuery](#)
- [Namespaces in MarkLogic Server](#)
- [Juggling Multiple Namespaces to Write Web Applications](#)
- [Recommended Practice with Multiple Namespaces](#)

6.1 How Namespaces Work

Every XML element is associated with a particular namespace. The namespace is used to differentiate between tags with the same name but different meanings.

Consider the example above. Rather than simply knowing that the structure you have been handed is rooted at a `<bug>` element, namespaces can let you know that the structure is in fact rooted at a `<bug>` element *as defined by ABC Corp.* This immediately lets you know that you should be expecting that element to have children as outlined in the leftmost structure from the example above.

Every XML element that you work with in XQuery has a namespace associated with it. This association can be specified explicitly for that particular element or be implicitly inherited from surrounding context. The syntax for explicitly associating a namespace with a particular element and its children is as follows:

```
< tagname xmlns = " namespaceURI ">
```

Namespaces are identified with Universal Resource Indicators (URIs). If you are creating your own XML data format, you may wish to create a namespace for it. To do so, choose a URI within your own domain (for example, `http://www.myco.com/ns/newnamespace`) in order to make sure that no one else accidentally uses the same URI to describe a completely different data format.

It is important to recognize that the URI acts only as a unique *identifier*. Nothing need be “stored” at this location; the URI is simply used to differentiate between one namespace and another. Because of this, the protocol prefix used in the URI has no real significance, beyond its use to identify a specific URI. `http://` is most frequently used, but you are free to use any alternative prefix you want (for example, `ftp:`).

Returning to our bug example, here are three sample explicit namespace declarations for each of the three `<bug>` structures outlined above:

```
<bug xmlns="http://www.abc.com/ns/bugns">
<bug xmlns="http://www.xyz.com/bugreports">
<bug xmlns="http://zoology.localuniv.edu/namespaces/entymology">
```

Clearly, it would be tiresome to have to explicitly provide a namespace for each XML element you specify. Implicit namespace association is achieved through inheritance. Consequently, in the XML structure:

```
<bug xmlns="http://www.abc.com/ns/bugns">
  <description>Order entry fails if order quantity is 5</description>
  <contact>Mary Lozares</contact>
  <date>2002/11/15</date>
  <status>Open</status>
</bug>
```

all four child elements (<description>, <contact>, <date> and <status>) are associated with the `www.abc.com/ns/bugns` namespace (which we will refer to as the `bugns` namespace, for convenience), because that is the namespace of their parent node.

If a namespace is not specified (explicitly or implicitly) for a given XML element, that element is associated with a universal *unnamed* namespace. We will refer to this namespace as the unnamed namespace, for convenience. The unnamed namespace can be explicitly specified using the null string:

```
<element xmlns="">
  <childelement1>....
```

6.2 Using Namespaces in XQuery

Every XML element and XPath component that is referenced in an XQuery expression is associated with some namespace.

Not surprisingly, since XQuery allows you to declare and manipulate XML structures, explicit namespace associations can be made as outlined above. For instance, if you wanted to bind an ABC Corp. <bug> node to a variable in XQuery, you could write a **let** clause as follows:

```
let $newBug := <bug xmlns="http://www.abc.com/ns/bugns">
  <description>Order entry fails if .... </description>
  <contact>Mary Lozares</contact>
  <date>2002/11/15</date>
  <status>Open</status>
</bug>
```

In this structure, <description>, <contact>, <date> and <status> are all associated with the `bugns` namespace by inheritance from their <bug> parent.

However, the use of XML elements within XQuery expressions introduces some new twists for implicit namespace association.

If a namespace is not explicitly declared for an XML structure or XPath expression component in an XQuery expression, the namespace for that structure or component is inherited from the static context of the surrounding XQuery code.

For this purpose, XQuery allows you to explicitly set a default namespace for the expressions (including XML structures) contained in an XQuery program. If the default namespace is not set explicitly, it is assumed to be the universal unnamed namespace.

In the following example, the default namespace is set to be the `bugns` namespace:

```
default element namespace = "http://www.abc.com/ns/bugns"

let $newBug := <bug>
    <description>Order entry fails if .... </description>
    <contact>Mary Lozares</contact>
    <date>2002/11/15</date>
    <status>Open</status>
</bug>

return
....
```

In the above example, the `<bug>` element that is bound to `$newBug` is associated with the `bugns` namespace, as that is the default namespace for the XQuery.

Namespace association for XPath components in XQuery programs follows a similar rule. The `<bug>` element in the **for** clause and the `<status>` element in the **where** clause of the query below are both associated with the `bugns` namespace:

```
default element namespace = "http://www.abc.com/ns/bugns"

for $bug in /bug
where $bug/status = "Open"
return
....
```

If a default namespace had not been set for this query, the `<bug>` and `<status>` elements would have been associated with the unnamed namespace.

6.3 Namespaces in MarkLogic Server

Just as every XML element and XPath component in every XQuery expression is associated with some namespace, so is every XML element stored within MarkLogic Server's datastore. This can lead to great frustration for developers.

Hidden namespace conflicts are one of the most common challenges encountered early in the XQuery learning curve. It is quite common for a query to mistakenly reference a different namespace than the data it is querying – usually because the XPath expressions in the query are not referencing the same namespace under which the data was loaded. In these situations, the XPath expressions do not match any data in the database, and consequently an empty (and seemingly incorrect) result is returned.

Consider the following example. Imagine that you had used the following XQuery to load a set of sample bug data into MarkLogic Server:

```
xdmp:document-insert ("bugs.xml",
  <bug xmlns="http://www.abc.com/ns/bugns">
    <description>....
```

Now you compose the following query in order to report all the outstanding bugs:

```
for $bug in /bug
where $bug/status = "Open"
return
  $bug
```

Mysteriously, this query returns no results – despite the fact that you have just carefully loaded the sample bug data using the preceding XQuery!

The explanation for this conundrum is simple.

The data was loaded in the `bugns` namespace. The query is operating in the unnamed namespace.

In fact, the query is correctly reporting that there are no `<bug>` nodes in the database that are associated with the unnamed namespace. (If there were, it would then be checking to see if those nodes had `<status>` elements associated with unnamed namespace that have values of “Open”.)

Fortunately, resolving the problem is also simple. Either load the bug data in the unnamed namespace, or associate the XPath components of the report query with the `bugns` namespace. If we were to do the latter, the query might look as follows:

```
default element namespace "http://www.abc.com/ns/bugns"

for $bug in /bug
where $bug/status = "Open"
return
  $bug
```

6.4 Juggling Multiple Namespaces to Write Web Applications

Using namespaces properly can get particularly complex when you are writing web applications.

This is because the query results must be associated with the `http://www.w3.org/1999/xhtml` namespace (which we will refer to as the XHTML namespace, for convenience) in order for the returned XML to be correctly rendered as HTML by the browser. But the content being incorporated into that returned dataset is almost certainly associated with a different namespace – either the unnamed or a format-specific namespace.

Making this all work out properly involves juggling multiple namespaces within your application. XQuery puts three tools at your disposal:

1. You can use the `xmlns` attribute to set the namespace for an XML structure and its enclosed XQuery expressions.
2. You can use the **default element namespace** syntax to set a default namespace for the entire XQuery program.
3. XQuery lets you define namespace prefixes, which can be used to concisely associate namespaces with specific XML elements and specific XPath components.

The syntax for defining these prefixes is as follows:

```
declare namespace prefix_name = " namespace_uri "
```

For example, the following XQuery code specifies that the XHTML namespace should be the default namespace, and that the `bugs` namespace can be referenced using the `bugs:` prefix:

```
default element namespace = "http://www.w3.org/1999/xhtml"  
declare namespace bugs = "http://www.abc.com/ns/bugs"
```

With this approach in mind, the following query could be used to return key data from each `<bug>` node in an XHTML `<div>` structure:

```
default element namespace = "http://www.w3.org/1999/xhtml"  
declare namespace bugs = "http://www.abc.com/ns/bugs"  
  
for $bug in /bugs:bug  
where $bug/bugs:status = "Open"  
return  
  <div>  
    { $bug/bugs:description } [{ $bug/bugs:date } ]  
  </div>
```

This code will be evaluated as follows:

1. The XHTML namespace is set as the default namespace, and the prefix `bugs` is defined to reference the `bugs` namespace.

2. The **for** clause creates a sequence of all `<bug>` elements in the database that are associated with the `bugns` namespace.
3. The **where** clause selects only those `<bug>` elements that have a `<status>` child element that is associated with the `bugns` namespace and has a value of “Open”
4. The **return** clause generates a sequence of `<div>` elements, one for each selected `<bug>` element. Each `<div>` element will be correctly associated with the XHTML namespace. Within each `<div>` element will be some free text that corresponds to a description of each open bug and the date at which the bug was first submitted.

It is important to recognize that using namespace prefixes only affects the specific element prefixed. Namespace prefixes have no impact on the namespace associations of child elements or contained XQuery expressions.

6.5 Recommended Practice with Multiple Namespaces

With all these namespace tools at your disposal, there are quite a number of different ways in which you can express the same query. For any given query, what’s the best way to approach it?

Choosing the “best” way is really just a question of programming preference. The same result can be obtained from many different permutations of the same basic code. To illustrate this, we provide a number of different variations of a sample query, culminating in our recommended programming practice.

Consider the following sample code included in the manual *Getting Started with MarkLogic Server*:

```
default element namespace = "http://www.w3.org/1999/xhtml"
declare namespace bk = "http://www.marklogic.com/ns/gs-books"

<html>
  <head>
    <title>Database dump</title>
  </head>
  <body>
    <b>XML Content</b>
    {
      for $book in doc("books.xml")/bk:books/bk:book
      return
        <pre>
          Title: { $book/bk:title/text() }
          Author: { ($book/bk:author/bk:first/text(), " ",
                    $book/bk:author/bk:last/text()) }
          Publisher: { $book/bk:publisher/text() }
        </pre>
    }
    <a href="update-form.xqy">Update Publisher</a>
  </body>
</html>
```

This is a perfectly good query. It sets the default namespace to the XHTML namespace, and then associates each reference to the books data using a prefix that references the `gs-books` namespace.

Here's a simple variation:

```
declare namespace bk = "http://www.marklogic.com/ns/gs-books"

<html xmlns="http://www.w3.org/1999/xhtml">
  <head>
    <title>Database dump</title>
  </head>
  <body>
    <b>XML Content</b>
    {
      for $book in doc("books.xml")/bk:books/bk:book
      return
        <pre>
          Title: { $book/bk:title/text() }
          Author: { ($book/bk:author/bk:first/text(), " ",
                    $book/bk:author/bk:last/text()) }
          Publisher: { $book/bk:publisher/text() }
        </pre>
    }
    <a href="update-form.xqy">Update Publisher</a>
  </body>
</html>
```

In the first case, the XHTML namespace is set as a default. In this case, the `<html>` element (and its enclosed XML structure and XQuery expressions) is being associated explicitly with the XHTML namespace.

Here is an approach that sets `gs-books` as the default namespace and explicitly prefixes every XHTML tag:

```
default element namespace = "http://www.marklogic.com/ns/gs-books"
declare namespace xhtml = "http://www.w3.org/1999/xhtml"

<xhtml:html>
  <xhtml:head>
    <xhtml:title>Database dump</xhtml:title>
  </xhtml:head>
  <xhtml:body>
    <xhtml:b>XML Content</xhtml:b>
    {
      for $book in doc("books.xml")/books/book
      return
        <xhtml:pre>
          Title: { $book/title/text() }
          Author: { ($book/author/first/text(), " ",
                    $book/author/last/text()) }
          Publisher: { $book/publisher/text() }
        </xhtml:pre>
    }
    <xhtml:a href="update-form.xqy">Update Publisher</xhtml:a>
  </xhtml:body>
</xhtml:html>
```

Sometimes it is convenient to separate the components of the XQuery. Then you can take advantage of two different implicit namespace associations. While no consensus “best practice” has emerged from the community at large, this is our recommended programming practice:

```
default element namespace = "http://www.marklogic.com/ns/gs-books"
declare namespace xhtml = "http://www.w3.org/1999/xhtml"

define function book-query()
returns element*
{
  for $book in doc("books.xml")/books/book
  return
    <xhtml:pre>
      Title: { $book/title/text() }
      Author: { ($book/author/first/text(), " ",
                $book/author/last/text()) }
      Publisher: { $book/publisher/text() }
    </xhtml:pre>
}

<html xmlns="http://www.w3.org/1999/xhtml">
  <head>
    <title>Database dump</title>
  </head>
  <body>
    <b>XML Content</b>
    { book-query() }
    <a href="update-form.xqy">Update Publisher</a>
  </body>
</html>
```

In this case, we have taken advantage of the static namespace association property of XQuery to implicitly associate the `gs-books` namespace with the XQuery expressions within the `book-query()` function, and the implicit inheritance association property of XQuery to associate the XHTML namespace with the children of the `<html>` element.

Let’s walk through how this works:

1. The default element namespace for the entire query is set to the `gs-books` namespace.
2. At parse time, the XPath expression contents of the function `book-query()` are statically associated with the default element namespace `gs-books`.
3. Within the `book-query` function, the `<pre>` elements need to be associated with the XHTML namespace, so they are specified using the explicit `xhtml:` prefix. Using a prefix to associate a particular namespace with a specific XML element does not implicitly associate that same namespace with child elements or XQuery expressions contained within that element.

4. The `<html>` element is explicitly associated with the XHTML namespace using the `xmlns` construction.
5. All of the children of the `<html>` element inherit their parent's namespace.
6. The fact that the `book-query()` function is referenced from within a structure associated with the XHTML namespace has no impact on the namespace association of the expressions encapsulated by the `book-query()` function. The namespace associations for these expressions are determined statically based on the position of those expressions within the entire XQuery code block, not dynamically based on the locations where the function is referenced.

7.0 XML and SQL

XML and XQuery are not intended to replace relational data formats and the SQL query language. Just as tabular data and relations are very well suited to solving a certain set of business problems, hierarchical data – such as XML – and a hierarchical query language—such as XQuery—are well suited to a different set of problems.

However, there is always interest in understanding how the two approaches – SQL with tabular data and XQuery with XML data – can be compared. This section provides a short introduction to the subject and includes the following sections.

- [XML Views of Relational Data](#)
- [SQL vs. XQuery](#)
- [SQL to XQuery, not XQuery to SQL](#)

7.1 XML Views of Relational Data

Relational database systems can relatively easily export data in XML format using a number of steps:

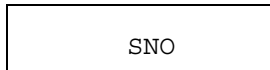
1. Use a SQL query to define (in tabular form) the data to be exported.
2. Use a simple default mapping from each exported table to an XML tree.
3. Use an XQuery query to compose the trees into an XML view with any desired structure.
4. Use arbitrary XQuery queries against this view.

Steps 1 through 4 can be executed in sequence, storing the data at each step. Alternatively, steps 1 through 4 can be composed into a single application that dynamically extracts, remaps and then queries the resulting dataset.

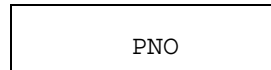
7.1.1 A Simple Example

Any table can be represented by a simplistic XML structure. The table itself is the root of the XML document. Each row in the table is a nested element under the root. Each column (each data value) in the row becomes a further nested element underneath the row.

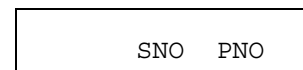
SUPPLIERS



PARTS



CATALOG



<suppliers>

<s_tuple>

<parts>

<p_tuple>

<catalog>

<c_tuple>

Taking this approach, the following simple tables map into simple XML structures:

These XML structures are much simpler than the deep and ragged structures XQuery was designed to work with.

7.2 SQL vs. XQuery

Working with this dataset as an example, we can now examine a number of simple comparisons between SQL and XQuery.

7.2.1 A Simple Query

Given the datasets outlined above, imagine you wanted to extract the part numbers of any gear parts, and order the results in ascending numeric order.

In SQL, you might write the following query:

```
SELECT pno
FROM parts
WHERE descrip LIKE 'Gear'
ORDER BY pno;
```

In XQuery, given the dataset mappings we have outlined above, the query might look like this:

```
for $p in /parts/p_tuple
where contains($p/descrip, "Gear")
order by $p/pno
return
  $p/pno
```

7.2.2 GROUP BY and HAVING

GROUP BY and HAVING are powerful SQL operators. Imagine you wanted to find the part numbers and average prices for parts that have at least three suppliers.

In SQL, you might write the following query:

```
SELECT pno, avg(price) AS avg_price
FROM catalog
GROUP BY pno HAVING count(*) >= 3
ORDER BY pno;
```

In XQuery, given the dataset mappings we have outlined above, the query might look like this:

```
for $p in distinct-values(/parts/p_tuple/pno)
let $c := /catalog/c_tuple[pno = $p]
where count($c) >= 3
order by ($p/pno)
return
  <well_supplied_part>
    { $p }
    <avg_price>{ avg($c/price) }</avgprice>
  </well_supplied_part>
```

The denormalization of XML requires the use of `distinct-values()` as well as the join between `<parts>` and `<catalog>`. We cannot go to the catalog document and simply group by `<pno>` and keep the groups bigger than three. Instead, for each distinct `<pno>` `$p` in `<parts>` for which there are more than 3 elements in the sequence `/catalog/c_tuple[pno = $p]`, we use the average over the `<price>` elements of the matching `<catalog>` sequence.

7.2.3 Inner Joins

Inner joins are accomplished using multiple `for` clauses, and relying on the FLWOR expression to generate the Cartesian product of the associated tuples.

For example, imagine you wanted to return a completely flat list of the supplier names and the descriptions of the parts they supply. In XQuery, the query might look like this:

```
for $c in /catalog/c_tuple,
  $p in /parts/p_tuple[pno = $c/pno],
  $s in /suppliers/s_tuple[sno = $c/sno]
order by $s/sname, $p/descrip
return
  <offering>
    { $s/sname }
    { $p/descrip }
  </offering>
```

7.2.4 Outer Joins

Outer joins are accomplished using nested FLWOR expressions.

For example, imagine you wanted to list the names of all suppliers in alphabetic order, and within each supplier, list the descriptions of the parts it supplies, if any. In XQuery, the query might look like this:

```
for $s in /suppliers/s_tuple
order by $s/sname
return
  <supplier>
  {
    for $c in /catalog/c_tuple[sno = $s/sno],
      $p in /parts/p_tuple[pno = $c/pno]
    order by $p/descrip
    return
      $p/descrip
  }
</supplier>
```

7.3 SQL to XQuery, not XQuery to SQL

It is important to note that all the examples in this section of the document have been based on a dataset that originated in a relational data model. The dataset was transformed into XML's hierarchical model using a number of simple steps.

It is convenient to compare XQuery with SQL based on this type of dataset, because it can be quite difficult to move in the other direction.

SQL has no mechanism for expressing hierarchy. It is very difficult, if not pragmatically infeasible, to take the full range of XQuery expression and port it into SQL. In particular, when dealing with ragged and potentially recursive XML structures, any attempt to port such data into tabular form and construct SQL queries against it is fraught with difficulties. The development effort to normalize such data is substantial. The performance impact of requiring joins of arbitrary dimension can be severe.

XQuery may be an interesting language with which to express queries against the dataset outlined above. However, we would not recommend porting this dataset to an XQuery-enabled XML database. This dataset is exactly the kind of dataset with which the relational model excels.

On the other hand, it is also easy to find datasets that do not lend themselves to the relational model. Large, complex, ragged and recursive XML documents are exactly the sort of thing that will give a relational model fits and starts. These sorts of documents are perfect for XQuery, and perfect for native XML databases. SQL cannot even begin to process this sort of data in any efficient manner.

Just as we would not port relational data into an XML database for use with XQuery, we would not port complex document data into a relational database for use with SQL. XQuery itself can travel between the two models, but sometimes the data cannot.

8.0 Learning the Language

While this document provides a brief introduction to the XQuery language, there is no substitute for hands-on experimentation. This chapter provides the following starting points for learning XQuery:

- [XQuery Use Cases](#)
- [Digging Deeper](#)
- [Additional Resources](#)

8.1 XQuery Use Cases

With that in mind, the MarkLogic Server installation includes a key learning aid. The W3C XQuery Working Group has constructed, at length, a number of use cases which demonstrate how a significant number of core tasks can be implemented using the XQuery language. These use cases are documented in the [XQuery Use Cases](#) document (see “References” on page 55).

When you install MarkLogic Server, an online version of these use cases is automatically installed on a server running at port 8000. If you have MarkLogic Server installed on your local machine, enter the following URL in a browser to access the use cases:

<http://localhost:8000/>

This online tool provides an initial opportunity to get your hands on the code – looking at sample XQuery, executing it against sample datasets, and even modifying it in place to see what happens when you start to change the XQuery code snippets yourself and then execute them. *Getting Started with MarkLogic Server* walks you through this process in some detail.

8.2 Digging Deeper

This document provides enough information to get started writing XQuery. It is not, however, a complete language reference.

There is no formal discussion of types, namespaces or schema. There is no in-depth discussion of sequences presented. Basic operators are presented only through the examples, and many are omitted (for example, `eq` versus `=`, `eq` versus `is`). And the more advanced operators and functions are hardly covered at all.

As the XQuery draft recommendation solidifies into the first release of the standard, resources that address all of these will start to appear. Books will be written and published. Online tutorials will flourish.

In the meantime, you are an early adopter, and the resources at your disposal are still sparse.

8.3 Additional Resources

Key resources are included in the [References](#) section at the end of this document. They include:

- The current XQuery draft language recommendation.
- The current draft recommendation for XQuery Functions and Operators.
- The XML Schema standard – required reading for both type definitions and to understand the schema definitions that can be used in MarkLogic Server.
- The XPath standard – useful reading to understand how XPath expressions are constructed and evaluated.

In addition, Mark Logic provides a number of extensions to XQuery that address functional limitations in the current proposal. For example, the current proposal does not address how XML should be updated or inserted into an underlying datastore. Mark Logic's built-ins address this and other issues. Documentation addressing these built-ins can be found online at Mark Logic's website.

9.0 References

The following references were used for this document.

9.1 References Specific to MarkLogic Server

Getting Started with MarkLogic Server, Mark Logic Corporation. See: <http://support.marklogic.com>.

Installation Guide, Mark Logic Corporation. See: <http://support.marklogic.com>.

“XML Query Use Cases”, W3C Working Draft. See: <http://www.w3.org/TR/xmlquery-use-cases>.

“XML Path Language (XPath) 2.0”, W3C Working Draft. See: <http://www.w3.org/TR/xpath20/>.

“XML Schema Parts 0, 1 and 2 (Primer, Structures and Datatypes)”, W3C Recommendation. See: <http://www.w3.org/TR/2001/REC-xschema-0-20010502/>, <http://www.w3.org/TR/2001/REC-xschema-1-20010502/>, and <http://www.w3.org/TR/2001/REC-xschema-2-20010502/>.

“XQuery 1.0: An XML Query Language”, 15 November 2002 W3C Draft (implemented in MarkLogic Server 2.0). See: <http://www.w3.org/TR/2003/WD-xquery-20030502/>.

“XQuery 1.0 and XPath 2.0 Functions and Operators”, 15 November 2002 W3C Working Draft (implemented in MarkLogic Server 2.0). See: <http://www.w3.org/TR/2003/WD-xpath-functions-20030502/>.

9.2 General References

“A Quilt, Not a Camel”, by Don Chamberlin, Jonathan Robie and Daniela Florescu. Presentation dated May 19, 2000.

“XQuery 1.0: An XML Query Language”, W3C Working Draft. See: <http://www.w3.org/TR/xquery/>.

“XQuery 1.0 and XPath 2.0 Functions and Operators”, W3C Working Draft. See: <http://www.w3.org/TR/xquery-operators/>.

Technical Support

Mark Logic provides technical support according to the terms detailed in your Software License Agreement. For evaluation licenses, Mark Logic may provide support on an “as possible” basis.

For registered customers, we invite you to visit our support website at <http://support.marklogic.com> to access our full suite of documentation and help materials. For all customers, including community licensed customers, visit the Mark Logic Developer’s site at <http://developer.marklogic.com>, which includes full product documentation, downloads, and developer community open-source projects.

If you have questions or comments, you may contact Mark Logic Technical Support at the following email address:

support@marklogic.com

If reporting a query evaluation problem, please be sure to include the sample XQuery code.