# XQuery 3.0 is Nearing Completion

Andrew Eisenberg
IBM, Westford, MA 01886
andrew.eisenberg.ma@verizon.net

## Introduction

XQuery 3.0 was published in January as a W3C Candidate Recommendation [1]. It adds a number of new features to XQuery 1.0, which was published as a W3C Recommendation in Jan. 2007 and revised in Dec. 2010 [2]. In this article, I will describe some of these new features.

The XML Query Working Group is responsible for these specifications:

- XQuery 3.0: An XML Query Language
- XML Syntax for XQuery 3.0

The XML Query Working Group and the XSL Working Group are jointly responsible for these specifications:

- XPath 3.0
- XQuery and XPath Data Model (XDM) 3.0
- XQuery and XPath Functions and Operators 3.0
- XSL and XQuery Serialization 3.0

XQuery 3.0 was initially called XQuery 1.1, but the Working Groups later decided to align the names of these new versions of XPath and XQuery. The relationship between the two languages is unchanged; XPath 3.0 is largely a subset of XQuery 3.0.

The Working Groups decided not to publish a new version of XQuery 1.0 and XPath 2.0 Formal Semantics. The rules for static type inferencing are now implementation-defined.

Publication as a Candidate Recommendation means that XQuery 3.0 is largely complete and has been widely reviewed. The Working Groups are gathering implementation experience before requesting that the specification advance further.

## New Features in XQuery 3.0

Immediately after XQuery 1.0 was completed, the Working Groups published a set of requirements for XQuery 1.1. Positional grouping, windowing, error processing and recovery, and higher order functions were among these requirements. Usage scenarios were created for some of these features in the XQuery 3.0 Use Cases document [3]. As our work progressed, additional features were requested. Among these were the relaxed FLWOR expression, the simple map operator, and the string concatenation operator.

Before discussing these new features, I need to introduce a bit of terminology. The FLWOR expression is made up of `for`, `let`, `where`, `order by`, and `return` causes. All but the `return` clause can be thought of as generating ordered streams of tuples. Each tuple contains one or more bindings of variables to values. Consider this FLWOR fragment:

```
for $name in doc('employees.xml')
              //employee/data(name)
let $len := string-length($name)
let $flet := substring($name, 1, 1)
```

This fragment would produce the following tuple stream:

```
($name := "Jones",  $len := 5, $flet := "J")
($name := "Barnes", $len := 6, $flet := "B")
```

### *Group By*

The `group by` clause will be familiar to anyone that has used SQL. Tuples are grouped, and these groups can be acted upon by subsequent clauses.

This new clause in the FLWOR expression allows a user to specify one or more grouping variables.

```
let $emp := doc('employees.xml')
return
   for $e in $emp//employee
   let $state := $e/address/state
   let $name := $e/name
   group by $state
   return <state name="{$state}">
            {$name}
          </state>

→

<state name='NY'>
   <name>Jones</name>
   <name>Barnes</name>
</state>
.
.
.
```

In this example, `$state` is a grouping variable, while `$e` and `$name` are non-grouping variables.

This clause produces a post-grouping tuple from one or more pre-grouping tuples – tuples provided by previous FLWOR clauses. The variable names in the post-grouping tuples will be the same as those in the pre-grouping tuples, but they will be bound to new values.

Each grouping variable generates a grouping key, which is the atomized value of the grouping variable. The atomized value of an atomic value is the value itself, while the atomized value of a node is the typed value of that node. A type error is raised if any grouping key is a sequence of more than one item.

Each pre-grouping tuple is assigned to a group, based on the value of its grouping keys. The `fn:deep-equal` function is used to compare key values. This means that an empty sequence is equivalent to another empty sequence, a `NaN` is equivalent to another `NaN`, two `untypedAtomic` values are compared as strings, and that two values for which `eq` is not defined, such as integer and date, are not equivalent.

The tuples produced by the `group by` clause have bindings for all of the grouping and non-grouping variables. A tuple is produced for each group of input tuples. For this reason, the number of post-grouping tuples will be equal to or less than the number of pre-grouping tuples.

Each post-grouping tuple has bindings for each grouping variable to one of its corresponding grouping keys. I say “one of” because the grouping keys for a grouping variable must all be equivalent according to `fn:deep-equal`, but they may not be identical. Some collations consider `"Andrew"` and `"andrew"` to be equal, but they are certainly not identical.

The post-grouping tuple has bindings for each non-grouping variable to the concatenation of all of the values of the non-grouping variable for the tuples in the partition. When ordering mode is set to ordered, the concatenation preserves the order of the values in the pre-grouping tuple stream.

In the example above, the first post-grouping tuple might be:

```
($state := "NY",
 $e := <employee><name>Jones</name> ...,
       <employee><name>Barnes</name> ...,
 $name := <name>Jones</name>,
          <name>Barnes</name>
)
```

Grouping variables can be bound within the `group by` clause. Also, a `collation` clause can be specified for the comparison of string values:

```
group by $state := $e/address/state
            collation
            "us-english-case-insensitive",
         $city := $e/address/city
            collation
            "us-english-case-insensitive"
```

SQL’s `HAVING` clause allows a user to filter out some of the groups created by the `GROUP BY` clause. In XQuery 3.0, this is accomplished with a `where` clause following the `group by` clause.

## Count

The `count` clause has been added to the FLWOR expression to identify the position of tuples in the tuple stream.

```
for $x in (1, 2)
for $y in (10, 20)
count $c
return concat ($c, ": ", $x + $y)
```

$\rightarrow$

```
"1: 11", "2: 21", "3: 12", "4: 22"
```

The count clause adds a binding of `$c` to each tuple in the tuple steam. `$c` is bound to the position of the tuple in the tuple stream.

## Windowing

The `window` clause operates on a sequence and derives zero or more windows from that sequence. Each window is a sequence of consecutive items taken from the input sequence. Windows can contain different numbers of items. Each window can be identified by its start and end position in the input sequence.

Each `window` clause specifies either `tumbling` or `sliding`. A tumbling window clause generates windows that do not overlap one another. A sliding window clause allows the windows to overlap, but it does not allow two windows to start at the same position.

The `window` clause is similar to the `for` clause. It operates on an input sequence and binds its variable to successive windows, starting from the beginning of the input sequence. The `window` clause allows a user to bind additional variables and then use these variables to define the windows that are generated. These additional variables may be bound in the `start` clause and `end` clause:

- start/end item (`$v`)
- start/end item position (`at $v`)
- item previous to the start/end item (`previous $v`)
- item following the start/end item (`next $v`)

A user must pick distinct names for the variables defined within a single `window` clause.

Let's look at a simple example:

```
for sliding window $w in
  ("a","b","c","d","e","f","g","h","i","j")
start $s at $spos when $spos mod 2 eq 1
end $e at $epos when $epos eq $spos + 2
return <w start="{$s}" end="{$e}">{$w}</w>
```

→

```
<w start="a" end="c">a b c</w>
<w start="c" end="e">c d e</w>
<w start="e" end="g">e f g</w>
<w start="g" end="i">g h i</w>
<w start="i" end="j">i j</w>
```

For each window that is generated, `$s` is bound to the start item, `$spos` is bound to the position of the start item, `$e` is bound to the end item, and `$epos` is bound to the position of the end item. We start our windows at odd numbered positions and end them two positions later.

The final result element, starting at position 9, reflects a window with only two items. When no item satisfied the end condition, the last item in the input sequence was taken to be the end of this window. This last window can be skipped by starting the `end` clause with `only`. This prefix requires that each window have an end condition that is true.

Let's look at two, more realistic, examples. We'll start with a document that records trades, with each trade having a stock name, a price, and a timestamp. `trades.xml` contains the trades in time order, as follows:

```
<trades>
   <trade stock="ACO"
          price="200" time="12:00:00.1"/>
   <trade stock="ACO"
          price="202" time="12:00:00.2"/>
   <trade stock="BCO"
          price="200" time="12:00:00.3"/>
   <trade stock="CCO"
          price="300" time="12:00:00.4"/>
   <trade stock="CCO"
          price="299" time="12:00:00.5"/>
   <trade stock="CCO"
          price="290" time="12:00:00.6"/>
   <trade stock="ACO"
          price="200" time="12:00:00.7"/>
   <trade stock="ACO"
          price="205" time="12:00:03.1"/>
</trades>
```

The following query finds one second windows after the trade of a specific stock and then filters the windows, keeping those that contain more than one trade for the stock.

```
let $onesec := xs:dayTimeDuration('PT1S')
for sliding window $w
   in doc("trades.xml")//trade
start $s when $s/@stock eq "ACO"
end next $n
    when $n/@time > ($s/@time + $onesec)
let $occurrences
   := count ($w[@stock eq $s/@stock])
where $occurrences gt 1
return <run stock="{$s/@stock}"
            occurrences="{$occurrences}"
            time="{$s/@time}"
       />
```

→

```
<run stock="ACO"
     occurrences="3" time="12:00:00.1"/>
<run stock="ACO"
     occurrences="2" time="12:00:00.2"/>
```

The following query finds runs of trades on the same stock:

```
for tumbling window $w
   in doc("trades.xml")//trade
start $s when true ()
end next $n when $n/@stock ne $s/@stock
where count ($w) gt 1
return <run stock="{$s/@stock}"
            length="{count ($w)}"
            max="{max($w/@price)}"/>
```

→

```
<run stock="ACO" length="2" max="202"/>
<run stock="CCO" length="3" max="300"/>
<run stock="ACO" length="2" max="205"/>
```

Saying "`start $s when true()`" causes a window to start at the beginning of the input sequence and additional windows to start as soon as the previous window has ended. The end condition for the `window` clause is that the stock name of the next trade differs from the stock name of the initial trade. This means that every trade in the window will have the same stock name. We filter out those windows that have just a single trade, and then construct an element for each window that contains the stock name, the number of trades, and the maximum price.

A tumbling window can omit its `end` clause. This will end a window just before the start of the window that follows it. The example above can be rewritten as:

```
for tumbling window $w
   in doc("trades.xml")//trade
start $s previous $p
   when not ($p/@stock eq $s/@stock)
where count ($w) gt 1
return <run stock="{$s/@stock}"
            length="{count ($w)}"
            max="{max($w/@price)}"/>
```

## Relaxed FLWOR Expression

The FLWOR expression in XQuery 1.0 consisted of one or more `for` clauses and `let` clauses, followed by an optional `where` clause, an optional `order by` clause, and a `return` clause.

As we considered the `group by` clause, the `window` clause, and the `count` clause, we saw that each clause operates on a tuple stream and generates a new tuple stream. The rigid order of the clauses could be relaxed.

The XQuery 3.0 FLWOR expression begins with a `for` clause, a `let` clause, or a `window` clause. Any number of `for`, `let`, `window`, `where`, `group by`, `order by`, and `count` clauses can occur between the initial clause and the final `return` clause, and they can occur in any order.

```
let $emp := doc("employees.xml")
for $e in $emp//employee
let $state := $e/address/state
where starts-with ($state, 'M')
group by $state
order by count ($e) descending
count $c
where $c le 5
return <state
          name="{$state}"
          position="{$c}"
          emps="{count ($e)}">
       </state>
```

This query finds employees with a state that starts with "M", groups them by state, orders the groups based on the number of employees they contain, numbers the groups, keeps only the top 5 groups, and then reports the position and number of employees for each state.

## Try/Catch

In XQuery 1.0, an expression that raises a dynamic error causes the evaluation of the query that contains the expression to fail. The `castable` expression was provided to allow users to avoid errors when casting values, which is a common occurrence:

```
let $additionalCharge
       := if ($sale/weight
              castable as xs:decimal)
          then if ($sale/weight
                   cast as xs:decimal gt 10)
               then 1.50
               else 0
          else 0
```

If a value of weight is not castable to `xs:decimal`, `"light"` for example, then 0 is returned.

We've added the `try/catch` expression to XQuery 3.0. It is a more general expression than the `castable` expression. The example above could be rewritten as:

```
let $additionalCharge
       := try {
             if ($sale/weight
                 cast as xs:decimal gt 10)
             then 1.50
             else 0
          }
          catch err:FORG0001 {0}
```

`err:FORG0001` is raised when an input value cannot be converted to a value in the value space of the target datatype. XQuery defines many errors and XQuery implementations may define additional errors.

If the `try` clause evaluates without error, then it provides the result of the `try/catch` expression.

The `catch` clause allows a "`|`" separated list of QNames to identify specific dynamic errors that should be caught. Wildcards such as `*`, `myco:*`, and `*:FORG0001` can be specified as well.

Multiple `catch` clauses may be specified. If an error occurred in the `try` clause, then the first `catch` clause that matches the dynamic error is evaluated and produces the result for the `try/catch` expression.

Within a `catch` clause, the following variables are bound and available to the user:

- `$err:code`
- `$err:description`
- `$err:value`
- `$err:module`
- `$err:line-number`
- `$err:column-number`
- `$err:additional`

If the error is not matched by any `catch` clause, then the `try/catch` expression raises the error.

The `try/catch` expression does not catch static errors. Static errors are detected during the analysis phase and prevent the query from executing.

```
try {
   substring ($description, 1, 7, "orange")
}
catch * {"oops"}
```

This query raises static error `err:XPST0017`, because no substring function with 3 parameters exists in the static context.

## Higher-Order Functions

The XPath and XQuery data model has been extended, allowing function as a new primitive type. This allows functions to return functions and it allows functions to accept functions as arguments.

Function types can be written as:

```
function(*)

function(xs:string, xs:integer) as xs:string
```

The first of these identifies a function with any signature. The second identifies a function that accepts a string and an integer and returns a string.

An inline function expression allows a user to create an anonymous function item.

```
let $capitalize :=
       function ($s)
          {let $leading := substring($s,1,1)
           let $trailing := substring($s,2)
           return
              concat (upper-case($leading),
                      lower-case($trailing))
          }
return $capitalize ("association")
```

→

```
"Association"
```

This example binds a function to `$capitalize` and then returns an invocation of this function. Because a type was not specified for the `$s` parameter and the function return, these types default to `item()*`. The user could have been more precise and defined the inline function with:

```
function ($s as xs:string) as xs:string
{...}
```

A function returned by an inline function expression carries with it the static context and variable bindings that existed when it was created.

```
let $x := 7
let $f := function($i) {$x + $i}
return
   let $x := 12
   return $f(100)
```

→

```
107
```

A named function reference can be used to return functions by naming statically defined functions, including XQuery's built-in functions. Both the function name and the number of parameters must be specified, using *`name#arity`*.

The `fn:string-join` function, used in the examples below, accepts a sequence of strings and a string separator. It concatenates the strings in the sequence, adding the separator string between pair of strings.

```
let $f := fn:string-join#2
return $f(("a", "z"), " to ")
```

→

```
"a to z"
```

A function is partially applied when it is invoked with one or more argument placeholders, "?". This partial application of a function item returns a new function item, with as many parameters as there are argument placeholders.

```
let $dash-join := fn:string-join(?, "--")
let $ducks := ("huey", "duey", "louie")
return $dash-join($ducks)
```

→

```
"huey--duey--louie"
```

Functions can be defined that accept functions as their arguments and return functions as their results. A "top" function might accept a sequence, a ranking function and an integer n, returning the top n items in the sequence:

```
declare function local:top
   ($seq as xs:string*,
    $rank as
       function (xs:string) as xs:integer,
    $n as xs:integer)
{
   for $i in $seq
   order by $rank($i) descending
   count $c
   where $c le $n
   return $i
};

local:top (("red", "green", "blue"),
           string-length#1,
           2)
```

→

```
"green", "blue"
```

This returns the top 2 colors, ordered by the number of characters in each value.

XQuery provides several useful higher-order functions:

`fn:for-each ($seq, $f)`
: Applies the function item `$f` to every item from the sequence `$seq` in turn, returning the concatenation of the resulting sequences in order.

`fn:filter ($seq, $f)`
: Returns those items from the sequence `$seq` for which the supplied function `$f` returns true.

`fn:fold-left ($seq, $zero, $f)`
: Processes the supplied sequence from left to right, applying the supplied function repeatedly to each item in turn, together with an accumulated result value.

`fn:fold-right ($seq, $zero, $f)`
: Processes the supplied sequence from right to left, applying the supplied function repeatedly to each item in turn, together with an accumulated result value.

`fn:for-each-pair ($seq1, $seq2, $f)`

Applies the function item $f to successive pairs of items taken one from `$seq1` and one from `$seq2`, returning the concatenation of the resulting sequences in order.

A simple example using `fn:for-each` is:

```
let $ctof :=
       function($c) {9 * $c div 5 + 32}
return fn:for-each ((0, 100), $ctof)
```

→

```
32, 212
```

## *Annotations*

Annotations allow properties to be associated with both variables and functions when they are being declared.

```
declare namespace myXQuery
   ="http://www.example.com/performance" ;

declare %private variable $stack := ... ;

declare %myXQuery:optimize("max")
   function local:creditScore ... ;
```

Annotations are QName, value pairs introduced with "`%`". XQuery defines some annotations in the `http://www.w3.org/2012/xquery` namespace.

`%private` does not specify a namespace prefix, and so is considered to be in the `http://www.w3.org/2012/xquery` namespace. Private variables and functions are not visible to the modules that import the library module that contains them.

Implementations are free to define annotations in their own namespaces. The behavior of these annotations is completely up to the implementation. One might guess that the user intends to spend extra time and resources optimizing the `local:creditScore` function. An implementation that doesn't recognize that annotation is free to ignore it.

## *Some Additional Features*

XQuery 3.0 has many new features that are smaller and simpler than those shown above. In this section, I will describe some of them.

### Simple Map Operator

The simple map operator is similar to the path expression. It consists of a series of expressions, separated by "`!`". Each operation E1 ! E2 is evaluated by first evaluating E1, using each item in E1 as the inner focus for evaluating E2, and then concatenating each evaluation of E2. Unlike the path expression, there are no restrictions on the type of items in either E1 or E2, duplicate elimination does not take place, and reordering does not take place.

Some examples of this operator are:

```
(1 to 5) ! (. * .)
```

→

```
1, 4, 9, 16, 25
```

```
let $emp := doc('employee.xml')//employee
return $emp
       !string-join((name, dept), ", ")
       !upper-case(.)
```

→

```
"JONES, A00", "BARNES, B01"
```

### String Concatenation Expression

The string concatenation expression has been introduced as a short-hand for the `fn:concat` function. It uses "`||`" to identify the strings that will be concatenated.

```
$emp!(./name || " works for " || ./dept)
```

→

```
"Jones works for A00"
```

### URI Qualified Name

An expanded QName can be specified by with a URI qualified name, which is made up of a namespace URI and a local name:

```
doc("employees.xml")
//Q{http://www.example.com/emp}jobTitle
```

In XQuery 1.0, this would have been expressed as:

```
declare namespace e =
   "http://www.example.com/emp";

doc("employees.xml")//e:jobTitle
```

These URI qualified names can be used almost everywhere that a QName is allowed. They cannot be used in element names and attribute names in element constructors.

### Switch expression

The `switch` expression is similar to the switch expressions provided by most programming languages.

The following example uses the `switch` expression to normalize an element value that contains a customer's gender:

```
let $input
   := lower-case (
         normalize-space ($customer/gender)
      )
let $gender
       := switch ($input)
             case "man" return "M"
             case "boy" return "M"
             case "woman" return "F"
             case "girl" return "F"
             default return ()
```

### fn:analyze-string function

XQuery 1.0 provides several string functions that make use of regular expressions. `fn:matches` determines whether a string is matched by a regular expression. `fn:replace` replaces matching substrings with replacement strings. `fn:tokenize` breaks a string into multiple pieces, with separators identified by regular expressions.

`fn:analyze-string` takes `fn:match` a step further. Instead of just returning whether or not a string is matched by a regular expression, `fn:analyze-string` identifies which parts of the string did and did not match. It does so by returning an XML element that identifies the matching and non-matching parts of the regular expression.

```
analyze-string
   ("000-11-2222 is a good fellow",
    "(\d\d\d)-(\d\d)-(\d\d\d\d)")
→
<analyze-string-result
   xmlns=
   "http://www.w3.org/2005/xpath-functions">
   <match>
      <group nr="1">000</group>
      -
      <group nr="2">11</group>
      -
      <group nr="3">2222</group>
   </match>
   <non-match> is a good fellow</non-match>
</analyze-string-result>
```

Indentation has been added to this result to improve its readability.

Each character of the input string appears in either a `match` or a `non-match` element. This means that the string value of the result of `fn:analyze-string` will equal the input string.

## Still More Features

There isn't room to describe all of the features that have been added to XQuery 3.0. I'll mention several more of them, without providing detailed descriptions or examples. I encourage you to go directly to the XQuery 3.0 specification for more information on these features – it is quite readable.

External variables can now have default values that will be used if values are not provided by the query's host environment.

The context item can be declared. Both a type and a default value can be specified.

A computed namespace constructor allows a user to create a namespace node. This will most often be used to add to the in-scope namespaces when a user is creating element nodes.

A number of formatting functions have been defined that accept a "picture" argument to generate string representations of numbers, dates, etc.

Serialization declarations can be included to determine how XML results will be serialized.

A type name can be specified in a `validate` expression, requiring that a node, when validated, must be of that type.

Common trigonometric and exponential functions have been defined.

The `fn:parse-xml` function parses its string argument and returns a document node. The `fn:serialize` function serializes its argument and returns a string value.

The `fn:unparsed-text` function uses its URI argument to retrieve a resource and returns a string representation of the resource.

The `fn:available-environment-variables` and `fn:environment-variable` functions provide access to variables in the host environment.

## XQuery 3.0 Conformance

Minimal Conformance is the lowest level of conformance that can be claimed for XQuery 3.0. Minimal Conformance encompasses all XQuery 3.0 functionality, with the exception of the following optional features:

- Schema Aware Feature – allows the use of `import schema` in the prolog and allows the use of the `validate` expression.
- Static Typing Feature – requires XQuery to detect and report type errors during the static analysis phase. Some queries that might run successfully without static typing will return an error during static analysis.
- Module Feature – allows the use of `import module` in the prolog and allows library modules to be created.
- Serialization Feature – requires that an implementation provide a way to produce an XML serialization of the result of a query.

- Higher-Order Function Feature – allows dynamic function calls, inline functions, named function references, and partial application of functions.

## Completing XQuery 3.0

The XML Query and XSL WGs have published the XQuery 3.0 Test Suite [5] and asked for results from implementers of XQuery 3.0. Positive results from impementors will give us confidence that the specifications are complete and unambiguous. We will then be allowed to publish XQuery 3.0 as a W3C Recommendation.

Work on XQuery 3.1 has already started. The XML Query WG will make this work visible in the near future by publishing a Requirements and Use Cases document.

Jim Melton and I were co-chairing the XML Query WG when this work began. Several years ago I had to resign this position, leaving Jim to complete this work. He has done an excellent job and will deserve hearty congratulations when XQuery 3.0 is published as a Recommendation.

Congratulations will also be due to the editors of our specifications, and to the many members of the XML Query and XSL Working Groups that contributed proposals, raised issues, attended meetings, and wrote test cases.

## References

[1] XQuery 3.0: An XML Query Language, W3C Candidate Recommendation, Jonathan Robie, Don Chamberlin, Michael Dyck, John Snelson, 08 January 2013, http://www.w3.org/TR/2013/CR-xquery-30-20130108/.

[2] XQuery 1.0: An XML Query Language (Second Edition), W3C Recommendation, Scott Boag, Don Chamberlin, Mary F. Fernández, Daniela Florescu, Jonathan Robie, Jérôme Siméon, 14 December 2010, http://www.w3.org/TR/xquery/.

[3] XQuery 3.0 Use Cases, W3C Working Draft, Jonathan Robie, Tim Kraska, 08 January 2013, http://www.w3.org/TR/xquery-30-use-cases/.

[4] XQuery 1.0 is Nearing Completion, Andrew Eisenberg and Jim Melton, ACM SIGMOD Record, Vol. 34, No. 4, Dec. 2005, http://www.sigmod.org/publications/sigmod-record/0512/p78-column-eisenberg-melton.pdf.

[5] XQuery/XPath/XSLT 3.* Test Suite, http://www.w3.org/XML/Query/QT3-test-suite/.

[6] Query - W3C, http://www.w3.org/standards/xml/query.