Blooming FLWOR - An Introduction to the XQuery FLWOR Expression

By: Dr. Michael Kay

In the previous tutorial in this series I presented a quick 10-minute introduction to XQuery. I started with a number of XPath expressions — because every XPath expression is a valid XQuery — and then went on to introduce FLWOR expressions, the workhorse of the XQuery language.

FLWOR expressions are so powerful, and form such an important part of XQuery, that I think it's worth giving them an article in their own right: and here it is.

If it's not obvious from the title of the article, FLWOR is pronounced "flower". And since jokes don't always travel well around the world, I should explain that where I come from, "blooming" is a semi-polite adjective that can express everything from exasperation to admiration, but always refers to something that merits your particular attention.

Simple FLWOR Expressions

The simplest XQuery FLWOR expression is probably something like this:

```
for $v in $doc//video return $v
```

This simply returns all the video elements in the document $doc.

You can add a bit of substance with an XQuery `where` clause and a slightly more imaginative XQuery `return` clause:

```
for $v in $doc//video
where $v/year = 1999
return $v/title
```

This now returns the titles of all the videos released in 1999.

If you know SQL, you will probably find this reassuringly similar to the corresponding SQL statement:

```
SELECT v.title FROM video v WHERE v.year = 1999
```

On the other hand, if you know XPath, you might wonder why you can't simply write:

```
$doc//video[year=1999]/title
```

The answer is that you can: this XPath path expression is completely equivalent to the FLWOR expression above, and what's more, it's a legal XQuery query. In fact, every legal XPath expression is also legal in XQuery. The first query in this section can in fact be written:

```
$doc//video
```

Which style you prefer seems to depend on where you're coming from: if you've been using XML for years, especially XML with a deep hierarchy as found in "narrative" documents, then you'll probably be comfortable with path expressions. But if you're more used to thinking of your data as representing a table, then the FLWOR style might suit you better.

As we'll see, FLWOR expressions are a lot more powerful than path expressions when it comes to doing joins. But for simple queries, the capabilities overlap and you have a choice. Although it might be true that in SQL every query is a SELECT statement, don't imagine that in XQuery every query has to be a FLWOR expression.

The Principal Parts of an XQuery FLWOR Expression

The name FLWOR comes from the five clauses that make up a FLWOR expression: `for`, `let`, `where`, `order by`, and `return`. Most of these are optional: the only clause that's always present is the XQuery `return` clause (though there must be at least one XQuery `for` or `let` clause as well). To see how FLWOR expressions work, we'll build up our understanding one clause at a time.

F is for FOR

The behaviour of the `for` clause is fairly intuitive: it iterates over an input sequence and calculates some value for each item in that sequence, returning a sequence obtained by concatenating the results of these calculations. In simple cases there's one output item for every input item. So:

```
for $i in (1 to 10)
return $i * $i
```

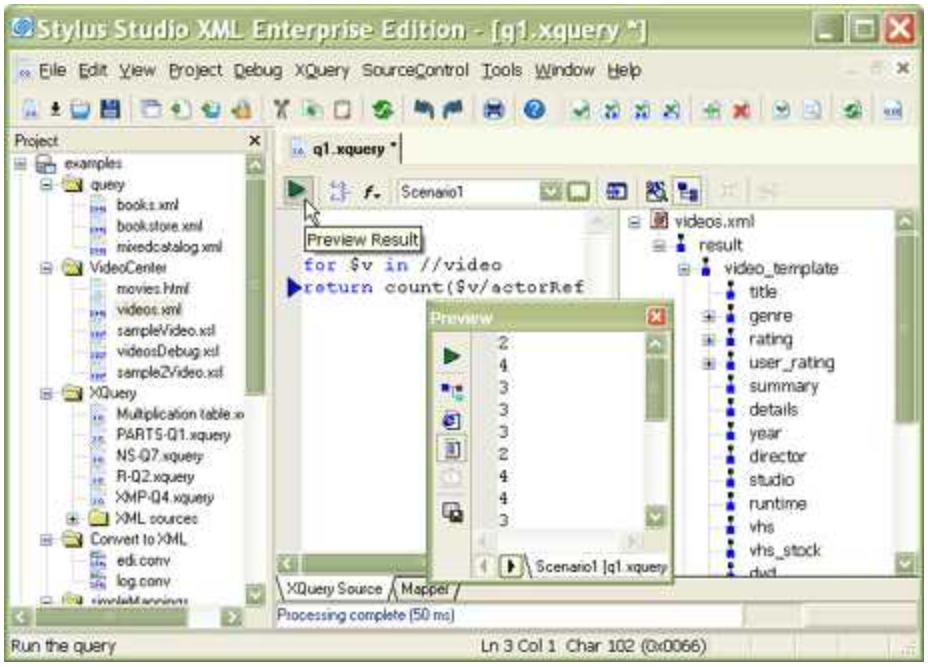
returns the sequence (1, 4, 9, 16, 25, 36, 49, 64, 81, 100).

In this example the input items are simple numbers, and the output items are also simple numbers. Numbers are an example of what XQuery calls *atomic values*: other examples are strings, dates, booleans, and URIs. But the XQuery data model allows sequences to contain XML nodes as well as atomic values, and the `for expression` can work on either.

Here's an example that takes nodes as input, and produces numbers as output. It counts the number of actors listed for each video in a data file:

```
for $v in //video
return count($v/actorRef)
```

You can run this in Stylus Studio or in Saxon, using the example videos.xml file on the Stylus Studio web site as input. You'll find some tips on setting up these tools in the previous XQuery tutorial. Here's the output from Stylus Studio:



The Preview pane shows the result: a rather boring sequence of numbers (2, 4, 3, 3, 3...).

This gives us a good opportunity to point out that a FLWOR expression is just an expression, and you can use it anywhere an expression is allowed: it doesn't have to be at the top level of the query. There's a function, avg( ), to compute the average of a sequence of numbers, and we can use it here to find the average number of actors listed for each of the movies in our data file:

```
avg(
    for $v in //video
    return count($v/actorRef)
)
```

The answer is 2.2941176 and a bit — the number of decimal places shown will depend on the XQuery processor that you're using. If you're only interested in the answer to two decimal places, try:

```
round-half-to-even(
   avg(
```

```
      for $v in //video
      return count($v/actorRef)
   ),
2)
```

which gives a more manageable answer of 2.29. (The strange name round-half-to-even is there to describe how this function does rounding: a value such as 2.145 is rounded to the nearest even number, in this case 2.14.) This is all designed to demonstrate that XQuery is a *functional language*, in which you can calculate a value by passing the result of one expression or function into another expression or function. Any expression can be nested inside any other, and the FLWOR expression is no exception.

If you're coming from SQL, your instinct was probably to try and do the averaging and rounding in the `return` clause. But the XQuery way is actually much more logical. The return clause calculates one value for each item in the input sequence, whereas the avg( ) function applies to the result of the FLWOR expression as a whole.

As with some of the examples in the previous section, XPath 2.0 allows you to write this example using path expressions alone if you prefer:

```
round-half-to-even(avg(//video/count(actorRef)), 2)
```

We've seen a `for` expression that produces a sequence of numbers from another sequence of numbers, and we've seen one that produces a sequence of numbers from a sequence of selected nodes. We can also turn numbers into nodes: the following query selects the first five videos.

```
for $i in 1 to 5 return (//video)[$i]
```

And we can get from one sequence of nodes to another sequence of nodes. This example shows all the actors that appear in any video:

```
for $actorId in //video/actorRef
return //actors/actor[@id=$actorId]
```

In fact, this last example is probably the most common kind of `for` expression encountered: but I introduced it last because I didn't want you to think that it's the only kind there is.

This example could once again be written as an XPath expression:

```
//actors/actor[@id=//video/actorRef]
```

However, this time the two expressions aren't precisely equivalent. Try them both in Stylus Studio: the FLWOR expression produces a list containing 38 actors, while the path expression produces only 36. The reason is that path expressions eliminate duplicates, and FLWOR expressions don't. Two actors are listed twice because they appear in more than one video.

The FLWOR expression and the "/" operator in fact perform quite similar roles: they apply a calculation to every item in a sequence and return the sequence containing the results of these calculations. There are three main differences between the constructs:

- The `for` expression defines a variable `$v` that's used in the `return` clause to refer to each successive item in the input sequence, while a path expression instead uses the notion of a context item, which you can refer to as "`.`" In this example, `//video` is short for `./root()//video`, so the reference to the context item is implicit.
- With the "/" operator, the expression on the left must always select nodes rather than atomic values. In the earlier example `//video/count(actorRef)` the expression on the right returned a number — that's a new feature in XPath 2.0 — but the left-hand expression must still return nodes.
- When a path expression selects nodes, they're always returned in document order, with duplicates removed. For example, the expression `$doc//section//para` will return each qualifying <para> element exactly once, even if it appears in several nested <section> elements. If you used the nearest-equivalent FLWOR expression, `for $s in $doc//section return $s//para`, then a <para> that appears in several nested sections would appear several times in the output, and the order of <para> elements in the output won't necessarily be the same as their order in the original document.

The `for` clause really comes into its own when you have more than one of them in a FLWOR expression. We'll explore that a little later, when we start looking at joins. But first, let's take a look at the other clauses: starting with `let`.

L is for LET

The XQuery `let` clause simply declares a variable and gives it a value:

```
let $maxCredit := 3000
let $overdrawnCustomers := //customer[overdraft > $maxCredit]
return count($overdrawnCustomers)
```

Hopefully the meaning of that is fairly intuitive. In fact, in this example you can simply replace each variable reference by the expression that provides the expression's value. This means that the result is the same as:

```
count(//customer[overdraft > 3000])
```

In a `for` clause, the variable is bound to each item in the sequence in turn. In a `let` clause, the variable only takes one value. This can be a single item or a sequence (there's no real distinction in XQuery — an item is just a sequence of length one). And of course the sequence can contain nodes, or atomic values, or (if you really want) a mixture of the two.

In most cases, variables are used purely for convenience, to simplify the expressions and make the code more readable. If you need to use the same expression more than once, then declaring a variable is also a good hint to the query processor to only do the evaluation once.

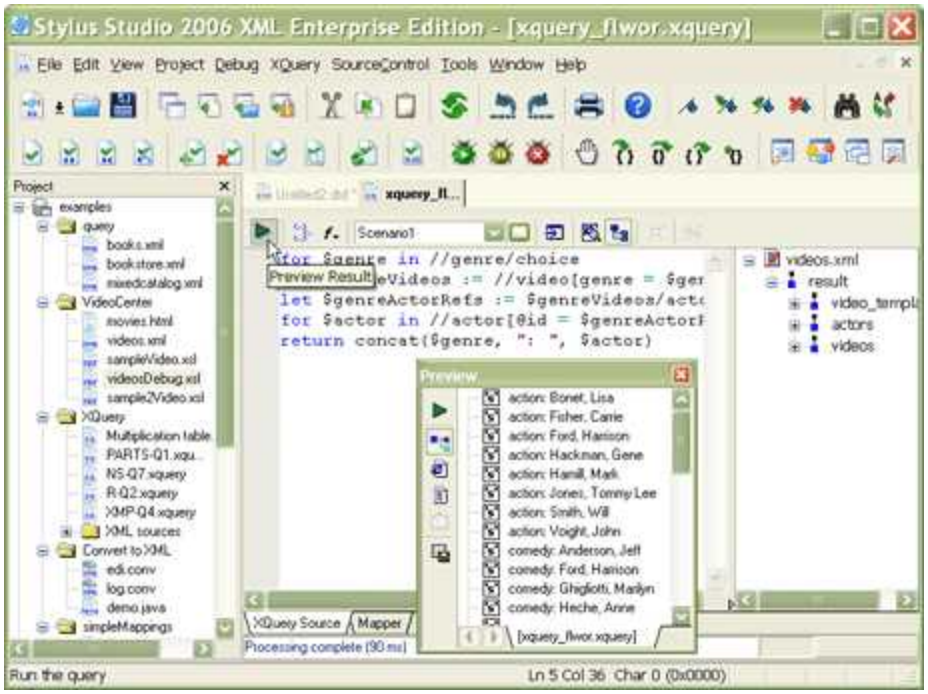
In a FLWOR expression, you can have any number of for clauses, and any number of let clauses, and they can be in any order. For example (we're returning here to the videos.xml data), you can do this:

```
for $genre in //genre/choice
let $genreVideos := //video[genre = $genre]
let $genreActorRefs := $genreVideos/actorRef
for $actor in //actor[@id = $genreActorRefs]
return concat($genre, ": ", $actor)
```

To understand this, just translate it into English:

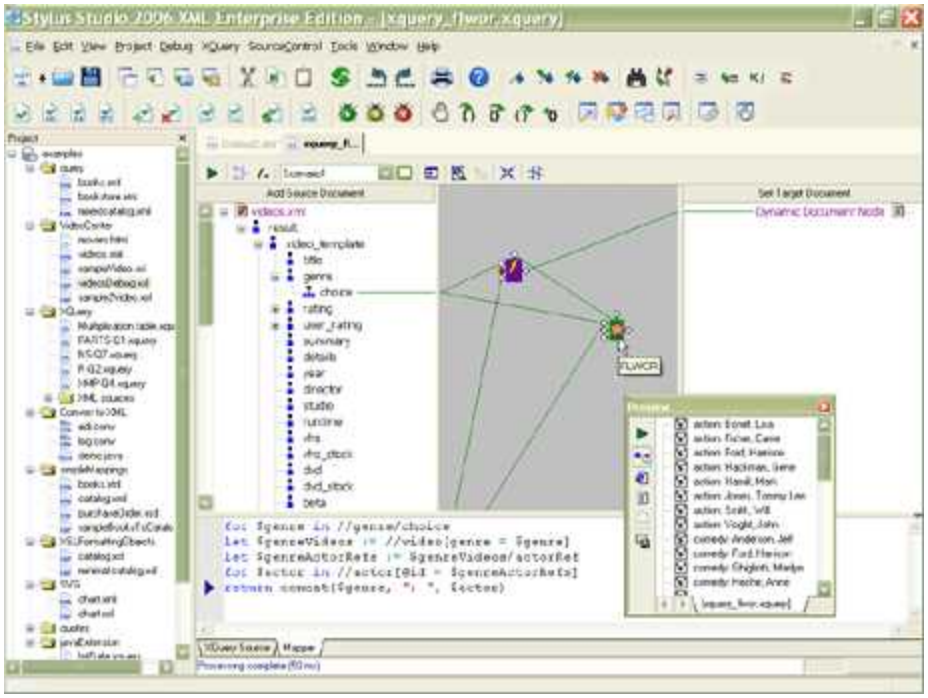
For each choice of genre, let's call the set of videos in that genre $genreVideos. Now let's call the set of references to all the actors in all those videos $genreActorRefs. For each actor whose ID is equal to one of the references in $genreActorRefs, output a string formed by concatenating the name of the genre and the name of the actor, separated by a colon.

Here's the output in Stylus Studio:



As a quick aside, Stylus Studio has an XQuery mapper that allows you to visually map from one or more XML input documents to any target output format. In a nutshell — click on the

"Mapper" tab on the bottom of the XQuery editing screen. Next, click on "Add Source Document" and add your source document(s). So for example, that last XQuery would look like this:



The FLWOR block is graphically represented as a function block with three input ports going into it on the the left (For, Where, Order By), a flow control port on the top, and an output port on the right. As you draw your XML mappings, the XQuery code is written for you; conversely, you can edit the XQuery code manually which will update the graphical model. To learn more, check out the " Introduction to the XQuery Mapper" and " Advanced XQuery Mapping" online video demonstrations. And now back to the regularly scheduled XQuery FLWOR tutorial ...

There's an important thing to note about variables in XQuery (you can skip this if you already know XSLT, because the same rule applies there). *Variables can't be updated*. This means you can't write something like `let $x := $x+1`. This rule might seem very strange if you're expecting XQuery to behave in the same way as procedural languages such as JavaScript. But XQuery isn't that kind of language, it's a declarative language and works at a higher level. There are no rules about the order in which different expressions are executed (which means that the little yellow triangle that shows the current execution point in the Stylus Studio XQuery debugger and XSLT debugger can sometimes behave in surprising ways), and this means that constructs whose result would depend on order of execution (like variable assignment) are banned. This constraint is essential to give optimizers the chance to find execution strategies that can search vast databases in fractions of a second. Most XSLT users

(like SQL users before them) have found that this declarative style of programming grows on you. You start to realize that it enables you to code at a higher level: you tell the system what results you want, rather than telling it how to go about constructing those results.

Isn't there a variable being updated when you write something like the following?

```
for $v in //video
let $x := xs:int($v/runtime) * xdt:dayTimeDuration("PT1M")
return concat($v/title, ": ",
      hours-from-duration($x), " hour(s) ",
      minutes-from-duration($x), " minutes")
```

(This query shows the running time of each video. It first converts the stored value from a string to an integer, then multiplies it by one minute (PT1M) to get the running time as a duration, so that it can extract the hours and minutes components of the duration. Try it.)

Here the variable $x has a different value each time around the XQuery for loop. This feels a bit like an update. Technically though, each time round the for loop you're creating a new variable with a new value, rather than assigning a new value to the old variable. What you can't do is to accumulate a value as you go around the loop. Try doing this to see what happens:

```
let $totalDuration := 0
for $v in //video
let $totalDuration := $totalDuration + $v/runtime
return $totalDuration
```

The result is not a single number, but a sequence of numbers, one for each video. This example is actually declaring two separate variables that happen to have the same name. You're allowed to use the same variable name more than once, but I don't think it's a good idea, because it will only get your readers confused. You can see more clearly what this query does if we rename one of the variables:

```
let $zero := 0
for $v in //video
let $totalDuration := $zero + $v/runtime
return $totalDuration
```

which is the same as this:

```
for $v in //video
return 0 + $v/runtime
```

Hopefully it's now clear why this gives a sequence of numbers rather than a single total. The correct way to get the total duration is to use the sum function: `sum(//video/runtime)`.

W is for WHERE

The XQuery where clause in a FLWOR expression performs a very similar function to the WHERE clause in a SQL select statement: it specifies a condition to filter the items we are interested in. The where clause in a FLWOR expression is optional, but if it appears it must only appear once, after all the for and let clauses. Here's an example that restates one of our earlier queries, but this time using a where clause:

```
for $genre in //genre/choice
for $video in //video
for $actorRefs in $video/actorRef
for $actor in //actor
where $video/genre = $genre
   and $actor/@id = $actorRefs
return concat($genre, ": ", $actor)
```

This style of coding is something that SQL users tend to be very comfortable with: first define all the tables you're interested in, then define a WHERE expression to define all the restriction conditions that select subsets of the rows in each table, and join conditions that show how the various tables are related.

Although many users seem to find this style comes naturally, I've personally been away from SQL for a while, and to me it often seems more natural to do the restriction in a predicate attached to one of the for clauses, like this:

```
for $genre in //genre/choice
for $video in //video[genre = $genre]
for $actorRefs in $video/actorRef
for $actor in //actor[@id = $actorRefs]
return concat($genre, ": ", $actor)
```

Perhaps there's a balance between the two; you'll have to find the style that suits you best. With some processors one style or the other might perform better, but a decent optimizer is going to treat the two forms as equivalent.

Do remember that in a predicate, you select the item that you're testing relative to the context node, while in the where clause, you select it using a variable name. A bare name such as genre is actually selecting `./child::genre` — that is, it's selecting a child of the context node, which in this case is a <video> element. It's very common to use such expressions in predicates, and it's very uncommon to use them (except by mistake!) in the where clause. If you use a schema-aware processor (the subject of another article) then you might get an error message when you make this mistake; in other cases, it's likely that the condition won't select anything. The where condition will therefore evaluate to false, and you will have to puzzle out why your result set was empty.

O is for ORDER BY

If there's no order by clause in a FLWOR expression, then the order of the results is as if the for clauses defined a set of nested loops. This doesn't mean they actually have to be evaluated as nested loops, but the result has to be the same as if they were. That's an important difference from SQL, where the result order in the absence of any explicit sorting is undefined. In fact XQuery defines an alternative mode of execution, *unordered mode*, which is similar to the SQL rules. You can select this in the query prolog, and the processor might even make it the default (this is most likely to happen with products that use XQuery to search a relational database). Some products (Saxon and Stylus Studio among them) give you exactly the same result whether or not you specify unordered mode — since the spec says that in unordered mode anything goes, that's perfectly acceptable.

Often however you want the query results in sorted order, and this can be achieved using the order by clause. Let's sort our videos in ascending order of year, and within that in decreasing order of the user rating:

```
for $x in //video
order by $x/year ascending, number($x/user-rating) descending
return $x/title
```

Note that we haven't actually included the sort keys in the data that we're returning (which makes it a little difficult to verify that it's working properly; but it's something you might well want to do in practice). We've explicitly converted the user-rating to a number here to use numeric sorting: this makes sure that 10 is considered a higher rating than 2. This isn't necessary if the query is schema-aware, because the query processor then knows that user-rating is a numeric field.

Ordering gets a little complicated when there is more than one for clause in the FLWOR expression. Consider this example:

```
for $v in //video
for $a in //actor
where $v/actorRef = $a/@id
order by $a, $v/year
return concat($a, ":", $v/title)
```

To understand this we have to stop thinking about the two for clauses as representing a nested loop. We can't compute all the result values and then sort them, because the result doesn't contain all the data used for sorting (it orders the videos for each actor by year, but only shows their titles). In this case we could imagine implementing the order specification by rearranging the for clauses and doing a nested loop evaluation with a different order of

nesting; but that doesn't work in the general case. For example, it wouldn't work if the order by clause changed to:

```
order by substring-after($a, ","),
      $v/year,
      substring-before($a, ",")
```

to sort first on the surname, then on the year, then on the first name (OK, that's crazy, but it's allowed).

At this stage I'll have to plead shortage of space before I go any further down this rat-hole. Suffice it to say that the spec introduces a concept of *tuples*, borrowed from the relational model, and describes how the sort works in terms of creating a sequence of tuples containing one value for each of the variables, and then sorting these notional tuples.

R is for RETURN

Every XQuery FLWOR expression has a return clause, and it always comes last. It defines the items that are included in the result. What more can one say about it?

Usually the XQuery return clause generates a single item each time it's evaluated. In general, though, it can produce a sequence. For example, you can do this:

```
for $v in //video[genre="comedy"]
return //actor[@id = $v/actorRef]
```

which selects all the actors for each comedy video. However, the result is a little unsatisfactory, because you don't know which actors belong to which video. It's much more common here to construct an element wrapper around each result:

```
for $v in //video[genre="comedy"]
return
  <actors video="{$v/title}">
    {//actor[@id = $v/actorRef]}
  </actors>
```

I haven't mentioned XQuery element and attribute constructors until now, because FLWOR expressions provide quite enough material for one article on their own. But in practice, a FLWOR expression without element constructors can only produce flat lists of values or nodes, and that's not usually enough. We usually want to produce an XML document as the output of the query, and XML documents aren't flat.

In practice this means that very often, instead of doing purely relational joins that generate a flat output, we want to construct hierarchic output using a number of nested FLWOR

expressions. Here's an example that (like the previous query) lists the videos for each actor, but with more structure this time:

```
for $v in //video[genre="comedy"]
return
   <actors video="{$v/title}">
      {for $a in //actor[@id = $v/actorRef]
         return
            <actor>
               <firstname>{substring-after($a, ",")}</firstname>
               <lastname>{substring-before($a, ",")}</lastname>
            </actor>
      }
   </actors>
```

Here we really do have two nested XQuery loops. The two queries below are superficially similar, and in fact they return the same result:

- ```
  for $i in 1 to 5
  for $j in ("a", "b", "c")
  return concat($j, $i)
  ```
- ```
  for $i in 1 to 5
  return
     for $j in ("a", "b", "c")
     return concat($j, $i)
  ```

But now add an order by clause to both queries so they become:

- ```
  for $i in 1 to 5
  for $j in ("a", "b", "c")
  order by $j, $i
  return concat($j, $i)
  ```
- ```
  for $i in 1 to 5
  return
     for $j in ("a", "b", "c")
     order by $j, $i
     return concat($j, $i )
  ```

The difference now becomes apparent. In the first case the result sequence is a1, a2, a3, ... b1, b2, b3,. In the second case it remains a1, b1, c1,... a2, b2, c2. The reason is that the first query is a single FLWOR expression (one return clause), and the order by clause affects the

whole expression. The second query consists of two nested loops, and the `order by` clause can only influence the inner loop.

So, the `return` clause might seem like the least significant part of the FLWOR, but a misplaced return can make a big difference! I recommend always aligning the F, L, O, W, and R clauses of a single FLWOR expression underneath each other, and indenting any nested expressions, so that you can see what's going on, and you can do this easily with the Stylus Studio XQuery Editor.

Other parts of the XQuery FLWOR Expression

We've explored the five clauses of the FLWOR expression that give it its name. But there are a few details we haven't touched on, partly because they aren't used very often. We'll summarize them here.

Declaring XQuery types

In the `for` and `let` clauses, you can (if you wish) declare the types of each variable. Here are some examples:

- `for $i as xs:integer in 1 to 5 return $i*2`
- `for $v as element(video) in //video return $v/runtime`
- `let $a as element(actor)* := //actor return string($a)`

Declaring types can be useful as a way of asserting what you believe the results of the expressions are, and getting an error message (rather than garbage output) if you've made a mistake. It helps other people coming along later to understand what the code is doing, and to avoid introducing errors when they make changes.

Unlike types declared in other contexts such as function signatures (and unlike variables in XSLT 2.0) the types you declare must be exactly right. The system won't make any attempt to convert the actual value of the expression to the type you declare, for example it won't convert an integer to a double, or extract the string value of an attribute node. If you declare the type as string but the expression delivers an attribute node, that's a fatal error.

XQuery Position variables

If you've used XSLT and XPath you've probably come across the `position()` function which enables you to number the items in a sequence, or to test whether the current item is the first or the last. FLWOR expressions don't maintain an implicit context in this way. Instead, you can declare an auxiliary variable to hold the current position, like this:

```
for $v at $pos in //video
   where $pos mod 2 = 0
   return $v
```

This selects all the even-numbered videos — useful if you are arranging the data in a table. You can use $pos anywhere where you might use the primary variable $v. Its value ranges from 1 to the number of items in the //video sequence. If there are no order by clauses, then the position variables in each for clause follow a nested-loop model as you would expect. If there *is* an order by clause, the position values represent the position of the items before sorting (which is different from the rule in XSLT).

There are various keywords in the order by clause that give you finer control over how the sorting takes place. The most important is the collation: unfortunately, though, the way collations work is likely to be very product-dependent. The basic idea is that if you're sorting the index at the back of a book, or the names in a phone directory, then you need to apply rather more intelligent rules that simply sorting on the numeric Unicode code value of each character. Upper-case and lower-case variants of letters may need to be treated the same way, and accents on letters have some quite subtle rules in many languages. The working group defining XQuery decided that standardizing collating sequences was too much like hard work, so they instead went for the simple rule that every collating sequence you might want has a name (specifically, a URI rather like a namespace URI) and it's up to each vendor to decide what collations to provide and how to name them.

Other things you can say in the order specification include defining whether empty values of the sort key (XQuery's equivalent of null values in SQL) should go at the start or end of the sequence, and whether the sort should be *stable*, in the sense that items with equal sort key values preserve their original order.

Multiple assignments

This is a bit of syntactic sugar, but I should mention it for completeness. Instead of writing

```
for $i in ("a", "b", "c")
for $j in 1 to 5
return concat($i, $j)
```

you can write

```
for $i in ("a", "b", "c"),
   $j in 1 to 5
return concat($i, $j)
```

The same applies to let clauses. The meaning is exactly the same.

What about Grouping in XQuery?

If you're used to SQL with its DISTINCT and GROUP BY keywords then you might have been wondering what the equivalent is in XQuery FLOWR expressions. Bad news, I'm afraid: there isn't one.

You can get a fair amount of mileage from the `distinct-values()` function. Here's a query that groups videos according to who directed them:

```
<movies>
   {for $d in distinct-values(//director) return
      <director name="{$d}">
      { for $v in //video[director = $d] return
         <title>{$v/title}</title>
      }
      </director>
   }
</movies>
```

This isn't an ideal solution: apart from anything else, it depends heavily on the ability of the query processor to optimize the two nested loops to give good performance. But for the time being, it's all there is. This is an area where vendors are very likely to offer extensions to the language as defined by W3C.

Grouping was a notoriously weak point of XSLT 1.0, and the problem has been addressed with considerable success in the 2.0 version of the language. I expect XQuery will follow suit; let's hope it doesn't take quite so long.

XQuery FLWOR Tutorial: Summary

Let's recap: In this FLWOR primer, you learned:

- FLWOR expressions have five clauses: `for`, `let`, `where`, `order by`, `return`. The first two can appear any number of times in any order. The `where` and `order by` clauses are optional, but if used they must appear in the order given. There is always a `return` clause.
- The semantics are similar to those of a SELECT statement in SQL. For most purposes it's possible to think of a FLWOR expression with multiple `for` clauses as a set of nested loops, but for sorting using `order by` a rather more complex execution model is needed.
- FLWOR expressions can be used anywhere that any other kind of expression can be used. This means they can be nested within each other, and they can appear in contexts such as an argument to a function like `count()` or `max()`. The only constraint

is that the type of value returned by the FLWOR expression (a sequence of items) must be appropriate to the context where the expression is used.

FLWOR expressions are the central feature of the XQuery language, in the same way as path expressions are at the heart of XPath. The other important feature in XQuery is the ability to construct an XML output document, and we'll explore that in more detail in another article to be featured in an upcoming issue of the Stylus Scoop.