Positional Grouping in XQuery

Michael Kay
Saxonica Limited
Reading, Berks
UK
+44 118 948 3589
mike@saxonica.com

ABSTRACT

This paper proposes an extension to the XQuery language to solve the problem of *positional grouping*: that is, problems in which it is necessary to convert a flat sequence into a hierarchy by recognizing patterns in the sequence of items. Positional grouping is contrasted with value-based grouping, where the allocation of items to groups is based on common values rather than on the positional relationships of the items in the sequence. The approach is based on analyzing a set of use cases, derived from real-world experience.

Categories and Subject Descriptors

D.3.3 [**Programming Languages**]: Language Contructs and Features – *data types and structures*

General Terms

Algorithms, Standardization, Languages.

Keywords

XML, XQuery, grouping.

1. INTRODUCTION

A recognized weakness of XQuery version 1.0 is the absence of facilities for grouping. In this paper, we use the term *grouping* to mean an operation that takes a "flat" sequence as input, and constructs a hierarchic arrangement of the items in the sequence, based on implicit structure found in the sequence of items. The input sequence is referred to as the *population*. The population will often be a sequence of sibling elements within an XML document, but in general (following the XQuery Data Model [1]) it may be any sequence of nodes or atomic values, regardless of any relationship these may have to each other within an XML tree structure.

In general it seems to be possible to classify grouping problems as being either value-based or positional. In value-based grouping, the primary criterion for placing two items in the same group is that they share common values for some *grouping key*: for example grouping books that have the same author, or employees based in the same location. With positional grouping, on the other hand, a significant factor in deciding how items are grouped is the position of the item relative to other items within the population. An example of a positional grouping problem is to take the input sequence (H2, P, P, P, H2, P, P) (think of these as the names of HTML elements) and group it into two `<section>` elements where each section contains an H2 element together with the following P elements up to the next H2. One possible solution to this problem in XQuery 1.0 is shown below. The complexity of this solution is evident.

```
declare function local:section($e as element(H2)) {
<section>{local:nextPara(
   $e/following-sibling::*[1][self::P])}
</section>};

declare function local:nextPara($p as element(P)?) {
  if ($p)  then ($p,
     local:nextPara($p/following-sibling::*[1][self::P]))
  else ()};

<out>{for $h in doc('doc.xml')//BODY/H2
        return local:section($h)}</out>
```

There have been a number of proposals for adding value-based grouping to XQuery, for example Borkar and Carey [2], and some XQuery implementations have "jumped the gun" by including such capabilities in advance of their standardization. Indeed, my own product, Saxon, includes such an extension, implemented in the form of a higher-order extension function in order to keep as close as possible to the conformance rules for vendor extensions: see [3].

By contrast, positional grouping in XQuery has received less attention. This may perhaps be due to the relational database tradition where all information is value-based and ordering plays no part. In XML, however, order is an intrinsic part of the data model, and many important relationships are expressed through the ordering of elements within an XML document. Positional grouping is particularly important when handling "narrative XML", that is, XML representing human-readable documents; however it also plays a role in applications that are more data-oriented, such as analysis of sequential log files, or up-conversion of legacy data formats.

The approach adopted in this paper is to start with a collection of use cases that collectively define the problem being tackled. Some of these use cases are taken from the XSLT 2.0 requirements document [4]. These were gathered (mainly by Steve Muench of

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

Oracle) by analyzing hundreds of real-world XSLT coding problems posted on the public xsl-list [5] between 1999 and 2001. Over time, some further examples of grouping problems have been observed, and representatives of these problems have been added to the list.

The XSLT 2.0 Candidate Recommendation [6] includes facilities designed to meet these requirements. These have proved very popular; the availability of the `<xsl:for-each-group>` instruction is often cited as the most important reason why people have moved forward from XSLT 1.0 even before 2.0 was finished. Any proposal for positional grouping in XQuery must therefore take the XSLT 2.0 facilities into account. On the other hand, they cannot be taken as the last word on the matter. There are some use cases that they handle more elegantly than others, and a few that they don't tackle at all. In addition, XQuery is a different language syntactically, and needs a different syntactic treatment.

I make no claim that the set of use cases is in any sense complete, or that the functionality proposed covers some theoretically well-defined problem space. Indeed, it does not: the problems addressed are a small subset of the problems that could be described with regular expressions. The aim here is not completeness in any theoretical sense, but usability: a simple facility for tackling a range of tasks that arise often in practice and are difficult to solve using XQuery 1.0 as it stands.

XSLT 2.0 tackles value-based and position-based grouping using a single construct, the `<xsl:for-each-group>` instruction. With hindsight, however, the two kinds of grouping have many differences and this may have overloaded the construct with non-orthogonal options. One important conceptual difference is that value-based grouping in XSLT assigns an item in the population to zero, one, or more groups: for example when books are grouped by author, a book will appear in a separate group for each of its several authors. By contrast the positional grouping facilities always perform a strict partitioning—each item in the population is assigned to exactly one group. Another difference is that (perhaps surprisingly) sorting of the results plays a bigger role in value-based grouping problems. With positional grouping, in nearly all cases the items in the output retain their ordering from the input. With value-based grouping, however, there is often a sorting requirement: the groups need to be sorted both with respect to other groups, and within themselves. In some cases this is based on input order (for example, listing the cast of a play in order of first appearance) but it is more commonly based on data values.

I have therefore chosen in this paper to consider positional grouping in isolation; there may be later opportunities to integrate the proposed solution with facilities for value-based grouping.

2. USE CASES

The use cases that follow define the scope of the positional grouping problem. Each use case is given a short name that is intended to be memorable, and will be referred to by this name in the rest of the paper. Each use case is defined by giving the input and the desired output; hopefully the relationship between the two is simple enough that the transformation from one to the other can be inferred. To illustrate the need for new language capabilities, I have also given an XQuery 1.0 solution to several problems, deliberately choosing a variety of coding styles.

2.1 Headings and Paragraphs

The problem here is to convert a structure with implicit sections, denoted by the presence of a header element (as used in XHTML) to a structure with explicit sections.

Input

```
<body>
  <h2>heading1</h2>
  <p>para1</p>
  <p>para2</p>
  <h2>heading2</h2>
  <p>para3</p>
  <p>para4</p>
  <p>para5</p>
</body>
```

Output

```
<chapter>
  <section title="heading1">
    <para>para1</para>
    <para>para2</para>
  </section>
  <section title="heading2">
    <para>para3</para>
    <para>para4</para>
    <para>para5</para>
  </section>
</chapter>
```

The XQuery 1.0 solution to this problem has already been given.

2.2 Adjacent Bullets

The problem here is to identify a sequence of adjacent `<bullet>` elements (among a sequence containing any other kind of element) and wrap them in a containing `<list>` element.

Input

```
<p/>
<q/>
<bullet>one</bullet>
<bullet>two</bullet>
<x/>
<y/>
```

Output

```
<p/>
<q/>
<list>
  <bullet>one</bullet>
  <bullet>two</bullet>
</list>
<x/>
<y/>
```

An XQuery 1.0 Solution

```
declare function local:item($e as element()?) {
if ($e) then
  if ($e[self::bullet])
    then (<list>{$e, local:followingBullets($e)}</list>,
     local:item($e/following-sibling::*[not(self::bullet)][1]))
    else ($e, local:item($e/following-sibling::*[1]))
else () };
declare function local:followingBullets($b as element()) {
  let $n := $b/following-sibling::*[1]
  where $n[self::bullet]
  return ($n, local:followingBullets($n))
};
<out>{local:item($input[1])}</out>
```

2.3 Term Definition Lists

Within a glossary in HTML, a defined term (`<dt>`) can be followed by a definition `<dd>`. The task is to group these together within a `<term>` element. To make things more complicated, a group can consist of one or more `<dt>` elements followed by one or more `<dd>` elements.

Input

```
<dt>XML</dt>
<dd>Extensible Markup Language</dd>
<dt>XSLT</dt>
<dt>XSL Transformations</dt>
<dd>A language for transforming XML</dd>
<dd>A specification produced by W3C</dd>
```

Output

```
<term>
  <dt>XML</dt>
  <dd>Extensible Markup Language</dd>
</term>
<term>
  <dt>XSLT</dt>
  <dt>XSL Transformations</dt>
  <dd>A language for transforming XML</dd>
  <dd>A specification produced by W3C</dd>
</term>
```

An XQuery 1.0 Solution

```
let $s := (for $e at $p in $input
           where $e[self::dt
                 and not(preceding-sibling::*[1][self::dt])]
           return $p, count($input)+1)
for $i in 1 to count($s) - 1
return <term>{
  for $j in $s[$i] to $s[$i + 1] - 1
  return $input[$j]
}</term>
```

2.4 Continuation Markers

Concatenate a sequence of fragments marked with the attribute `cont="yes"` to indicate that the next fragment is a continuation.

Input

```
<in cont="yes">One way to</in
<in cont="yes"> understand positional grouping is
<in> as an exercise in parsing.</in>
<in cont="yes">To get from a sequence of items</in>
<in cont="yes"> to a tree, we could use</in>
<in> some kind of grammar.</in>
```

Output

```
<para>One way to understand positional grouping is as an
exercise in parsing.</para>
<para>To get from a sequence of items to a tree, we could
use some kind of grammar.</para>
```

2.5 Page Ranges

Given a sequence of page references such as might occur in the index of a book, identify sub-sequences that denote continuous ranges of page numbers.

Input

```
4, 6, 9, 11, 12, 13, 18, 20, 21
```

Output

```
4, 6, 9, 11-13, 18, 20-21
```

2.6 Arrange in Rows

Arrange a sequence of items in fixed size rows of a table, say in three columns. (The same problem occurs when grouping records say ten to a page).

Input

```
"Green", "Pink", "Lilac", "Turquoise", "Peach", "Opal",
"Champagne"
```

Output

```
<table>
<tr>
  <td>Green</td><td>Pink</td><td>Lilac</td>
</tr>
<tr>
  <td>Turquoise</td><td>Peach</td><td>Opal</td>
</tr>
<tr>
  <td>Champagne</td><td> </td><td> </td>
</tr>
</table>
```

2.7 Level Numbers

In this problem, the hierarchic structure of the input is indicated by COBOL-like level numbers. I present a worked XSLT 2.0 solution to this problem in [7], in the context of up-conversion of genealogical data held in the non-XML GEDCOM format.

Input

```
<data>
<gedcom level="0"/>
<indi level="1"/>
<name level="2"/>
<first level="3">Michael</first>
<last level="3">Kay</last>
<email level="2">mike@saxonica.com</email>
<indi level="1"/>
<name level="2"/>
<first level="3">Norm</first>
<last level="3">Walsh</last>
<email level="2">norm@nwalsh.com</email>
</data>
```

Output

```
<gedcom>
  <indi>
    <name>
      <first>Michael</first>
      <last>Kay</last>
    </name>
    <email>mike@saxonica.com</email>
  </indi>
  <indi>
    <name>
      <first>Norm</first>
      <last>Walsh</last>
    </name>
    <email>norm@nwalsh.com</email>
  </indi>
</gedcom>
```

3. ANALYSIS OF THE PROBLEM

3.1 The Need for a Solution

The XQuery 1.0 solutions to these problems either involve recursive traversal of the sequence, or they involve the construction and manipulation of sequences of integers identifying subranges of items in the input sequence. Neither of these approaches makes for easy programming: even for an experienced user, it takes several attempts to construct correct solutions. Furthermore, both approaches strain the ability of the XQuery engine to deliver scalable performance. Recursive solutions have a tendency to blow the stack limit as the sequence length increases (tail call optimization can prevent this but is not always possible). Solutions based on integer indexing do not lend themselves well to streaming solutions. Yet, in the author's experience, these problems arise very frequently in practice.

In a functional programming language such as Haskell, the problem would be addressed using higher-order functions. This will be the conceptual basis of my approach also. However, to fit within the language style of XQuery and the concepts familiar to its typical users, a solution using custom syntax is preferred.

3.2 Positional Grouping as Parsing

One way to understand positional grouping is as an exercise in parsing. To get from a sequence of items to a tree that reflects their structure, we could use some kind of grammar that matches the items in the sequence as tokens in an alphabet. This might be a regular expression, or a BNF grammar; it might even be the schema for the document itself.

I have rejected approaches using formal grammars or regular expressions for two reasons. Firstly, such solutions are likely to be rather complex (it's not good enough, for example, to use element names as the symbols in the regular expression alphabet, since many positional grouping problems depend on criteria other than the element name). Secondly, the complexity is unnecessary: in all the use cases listed in the previous section, the criteria for partitioning the population can be defined in much simpler ways than with a full grammar.

3.3 Sequences of Sequences

Since the output of a grouping operation is a collection of groups, we immediately hit a design problem because the XQuery data model does not allow for sequences of sequences.

In practice the output of grouping is usually an XML tree, in which the hierarchic levels are represented by newly constructed element nodes. (Alternatively, the group may be summarized or aggregated so that the only output is a count or a total.) Therefore, it's best to think of grouping as being a higher-order operation in which a function is invoked to process each group as it is identified. The group itself is a simple sequence, and the groups are then processed conceptually one-at-a-time, so that there is no need ever to construct a sequence of sequences as an object.

3.4 XSLT 2.0 Facilities

XSLT 2.0 provides three variants of the `<xsl:for-each-group>` construct to handle positional grouping. These are:

- **group-adjacent**: defines a value-based grouping key (this can be any function of the items to be grouped, for example the element name). Adjacent items from the population go in the same group if they have the same value for the grouping key.
- **group-starting-with**: defines a pattern; items in the sequence that match this pattern form the first item of a new group.
- **group-ending-with**: defines a pattern; items in the sequence that match this pattern form the last item of a group.

Between them, these facilities enable most of the use cases in section 2 to be solved, though in some cases only by roundabout techniques. The solutions are along the following lines:

XSLT 2.0 Solutions to Use Cases

Use Case	XSLT 2.0 Solution
Headings and Paragraphs	group-starting-with
Adjacent Bullets	group-adjacent
Term Definition Lists	first group the <dt> and <dd> elements separately using group-adjacent; then merge a group of <dt> elements with the following group of <dd> elements using group-starting-with
Continuation Markers	group-ending-with
Page Ranges	group-adjacent, with a key of `(. - position())`
Arrange in Rows	group-adjacent, with a key of `(position() idiv 3)`
Level Numbers	group-starting-with, applied recursively at each level of grouping

So the facilities in XSLT 2.0 are sufficient to solve the problems in all cases; but in some cases, the solution is not easy.

3.5 Identifying Breaks

In all the use cases, the essential problem is to decide, given a pair of adjacent items in the population, whether to put the second item in the same group as the first, or to start a new group at that point.

The information needed to make this decision can in nearly all cases be made as a function of the two adjacent items. In one use case, *Arrange in Rows*, the decision does not depend on the items themselves, but on their position in the population.

This suggests that we can solve the positional grouping problem by providing a higher-order function with the following parameters:

1. The population to be grouped
2. A function to be called to process each identified group
3. A function that is called for each pair of adjacent items, and which is supplied with those two items plus the position of one of the items (let's say the second of the two) in the population; as output this function returns true *iff* the two items should go in different groups.

To see how this solves each of the use cases, let's write the determining function (item 3 above) as:

```
declare function break($first as item(), $second as item(),
$position as xs:integer) as xs:boolean {
....
}
```

and examine what goes in the body of the function for each of the use cases.

3.5.1 Headings and Paragraphs

In this case the function takes the form:

```
declare function break($first as item(), $second as item(),
$position as xs:integer) as xs:boolean {
   $second[self::h2]
}
```

That is, we break between two items if the second item is an h2 element.

3.5.2 Adjacent Bullets

For this use case we break between two items if they are not both bullet elements:

```
declare function break($first as item(), $second as item(),
$position as xs:integer) as xs:boolean {
   not($first[self::bullet] and $second[self::bullet])
}
```

3.5.3 Term Definition Lists

The break between groups occurs here when the first item is a <dd> element and the second is a <dt>:

```
declare function break($first as item(), $second as item(),
$position as xs:integer) as xs:boolean {
   $first[self::dd] and $second[self::dt]
}
```

3.5.4 Continuation Markers

In this case we start a new group after an item that does not specify `cont="yes"`:

```
declare function break($first as item(), $second as item(),
$position as xs:integer) as xs:boolean {
   not($first/@cont="yes")
}
```

3.5.5 Page Ranges

Two numbers go in the same group if they are consecutive:

```
declare function break($first as item(), $second as item(),
$position as xs:integer) as xs:boolean {
   $first + 1 ne $second
}
```

3.5.6 Arrange in Rows

A new row starts if the position of the second item in the pair is an integer multiple of the number of columns. Remember that in XQuery all indexing starts at one.

```
declare function break($first as item(), $second as item(),
$position as xs:integer) as xs:boolean {
   ($position - 1) mod 3 = 0
}
```

3.5.7 Level Numbers

As with the XSLT solution, this requires recursive application of positional grouping to handle the multiple levels. At a given level, say level $N, the solution is:

```
declare function break($first as item(), $second as item(),
$position as xs:integer) as xs:boolean {
   $second/@level=$N
}
```

4. A SYNTAX PROPOSAL

The analysis in the previous section gives the conceptual basis of the proposed approach. However, XQuery does not support higher-order functions. Instead, higher-order operations such as mapping and filtering are supported using custom syntax or operators: for example the "/" in a path expression plays the role of a higher-order *map()* or *apply()* function.

So we need to consider what surface syntax should be provided to support this proposed functionality.

The addition of new syntactic constructs to XQuery is constrained by the fact that the language has no reserved words. There are a variety of conventions used to disambiguate keywords by their syntactic context. The design below attempts to follow these rules, but it has not been verified that the resulting grammar is unambiguous.

As discussed in the introduction, I am proposing that positional grouping and value-based grouping should be kept entirely separate. Since the keyword "group" is likely to be used in the context of value-based grouping, I will use the keyword "partition" for positional grouping.

This suggests a syntax along the following lines:

```
partition $g in population
break
  after $a
  before $b
  at $p
  where condition
return action
```

In this structure:

- `$g` is a range variable bound to each group in turn, allowing the group to be referenced within the *action* expression.
- *population* is an arbitrary expression that selects the sequence of items to be grouped.
- `$a`, `$b`, and `$p` denote the three variables that are conceptually parameters to the function that decides whether two items that are adjacent in the population should go in the same group. Any of these that is not required may be omitted. The actual variable names of course may be user-chosen.
- *condition* is an expression, written in terms of `$a`, `$b`, and `$p`, that determines whether the second item `$b` should go in a different group from the first `$a`
- *action* is an expression, written in terms of `$g`, that determines how each group is processed: often this will construct elements to go on a result tree.

4.1 Use Cases with the Proposed Syntax

The following sections illustrate how the use cases may be solved using the proposed syntax.

4.1.1 Headings and Paragraphs

```
partition $section in *
break before $h2 where $h2[self::h2]
return <section title="{$section/h2}">
          {$section/p}
        </section>
```

4.1.2 Adjacent Bullets

```
partition $children in *
break after $a before $b
  where not($a[self::bullet] and $b[self::bullet])
return
  if ($children/self::bullet) then
    <list> {$children} </list>
  else $children
```

4.1.3 Term Definition Lists

```
partition $term in *
break after $a before $b
  where ($a[self::dd] and $b[self::dt])
return <term>{$term}</term>
```

4.1.4 Continuation Markers

```
partition $para in para
break after $a where not($a/@cont = "yes")
return <para>{$para}</para>
```

4.1.5 Page Ranges

```
partition $range in $page-numbers
break after $a before $b where ($b != $a + 1)
return if (count($range) = 1) then
          $range
        else
          concat($range[1], "-", $range[last()]
```

4.1.6 Arrange in Rows

```
partition $rows in $colours
break at $p where (($p - 1) mod 3 = 0)
return <tr> {
   for $i in $rows return <td>{$i}</td>
} </tr>
```

4.1.7 Level Numbers

Here we give the full recursive solution:

```
declare function f:group($items as element()*, $level as
xs:integer) as element() {
  partition $group in $items
  break before $b where $b/@level = $level
  return element {$group[1]/node-name()} {
     f:group(remove($group, 1), $level + 1)
   }};
f:group(/data/*, 0)
```

4.2 Grammar

Here is a slightly more formal presentation of the proposed grammar, using the conventions of the XQuery 1.0 specification, and linking in to its metasymbol names where appropriate.

The production `ExprSingle` is extended to allow the option of a `PartitionExpr`. We then define:

```
PartitionExpr ::= "partition" "$" VarName "in" ExprSingle
                  "break"
                  ("before" "$" VarName)?
                  ("after" "$" VarName)?
                  ("at" "$" VarName)?
                  "where" ExprSingle
                  "return" ExprSingle
```

4.3 Formal Semantics and Static Typing

I make no attempt at this stage to define a formal semantics for the new construct, in particular I have not attempted to define the type inferencing rules. As with a FLWOR expression, it is possible to deduce the static types of the range variables from the static types of the expressions to which they are bound; however, it may be desirable to add optional type declarations to the syntax, if only for consistency with other constructs in the language.

4.4 Performance

It should be self-evident that (like the XQuery 1.0 solutions to these problems) the performance of the proposed construct will be linear with respect to the size of the input sequence.

Because a range variable is bound to the contents of each group in turn, a straightforward implementation will need to allocate enough memory to hold the largest group. In Saxon, this approach has so far proved adequate for implementing the similar grouping constructs in XSLT 2.0. There is however potential for a smart implementation to optimize the use of memory further by analyzing how the range variable is actually used. In many cases the group is simply copied to the output of the partition expression. It is then quite feasible during a single pass through the input data both to identify the group boundaries and to copy the content of each group to the output.

Unlike the XQuery 1.0 solutions, no problems should arise as the number of groups increases.

5. CONCLUSIONS

I have presented a number of use cases for positional grouping problems: there is every reason to believe that these are representative of real-world problems that XQuery users will need to tackle, especially when handling narrative documents.

I then established that these can all be solved, at the conceptual level, by means of a user-defined function that is called once for each pair of adjacent items in the input sequence, and that determines whether the second item should be in the same group as the first, or should start a new group.

Based on this idea, I then proposed a syntactic extension to the XQuery grammar to provide a capability for positional grouping that meets all the use cases.

The proposed syntax tackles all the problems that the positional grouping facilities in XSLT 2.0 handle, and in some cases handles them more cleanly.

6. REFERENCES

[1] Fernandez, M. et al, XQuery 1.0 and XPath 2.0 Data Model (XDM), W3C Candidate Recommendation 3 Nov 2005. http://www.w3.org/TR/xpath-datamodel/

[2] Borkar, V., and Carey, M. *Extending XQuery for Grouping, Duplicate Elimination, and Outer Joins.* XML 2004, Idealliance (see http://www.idealliance.org/proceedings/xml04/papers/229/XQueryExtensionsFinal.html

[3] Kay, M.H. *saxon:for-each-group() extension function*, documented at http://www.saxonica.com/documentation/extensions/functions/for-each-group.html

[4] Muench, S. and Scardina, M. XSLT Requirements Version 2.0. W3C, 14 Feb 2001. http://www.w3.org/TR/xslt20req

[5] xsl-list Open Forum on XSL. http://www.mulberrytech.com/xsl/xsl-list/index.html

[6] Kay, M.H. *XSL Transformations (XSLT) Version 2.0.* W3C Candidate Recommendation, 3 Nov 2005. http://www.w3.org/TR/xslt20/

[7] Kay, M.H. *Up-conversion using XSLT 2.0.* XML 2004, http://www.idealliance.org/proceedings/xml04/papers/111/mhk-paper.html