Extending XQuery for Analytics

Kevin Beyer, Don Chamberlin, Latha S. Colby, Fatma Özcan, Hamid Pirahesh, Yu Xu*

IBM Almaden Research Center
650 Harry Rd
San Jose, CA 95120

Abstract

XQuery is a query language under development by the W3C XML Query Working Group. The language contains constructs for navigating, searching, and restructuring XML data. With XML gaining importance as the standard for representing business data, XQuery must support the types of queries that are common in business analytics. One such class of queries is OLAP-style aggregation queries. Although these queries are expressible in XQuery Version 1, the lack of explicit grouping constructs makes the construction of these queries non-intuitive and places a burden on the XQuery engine to recognize and optimize the implicit grouping constructs. Furthermore, although the flexibility of the XML data model provides an opportunity for advanced forms of grouping that are not easily represented in relational systems, these queries are difficult to express using the current XQuery syntax. In this paper, we provide a proposal for extending the XQuery FLWOR expression with explicit syntax for grouping and for numbering of results. We show that these new XQuery constructs not only simplify the construction and evaluation of queries requiring grouping and ranking but also enable complex analytic queries such as moving-window aggregation and rollups along dynamic hierarchies to be expressed without additional language extensions.

1. Introduction

As XML gains acceptance as a standard format for data storage and exchange, XML data repositories need to deal with data generated by both document-centric applications and transaction-centric applications. XML query languages must, therefore, be well-suited for expressing queries that require searching and navigating through multiple levels of XML objects as well as relating and combining objects as required for analysis of transaction-oriented data. This includes queries that involve grouping, ranking, moving-window aggregation, and comparisons across different levels of partitions, in addition to navigation, extraction, filtering, and construction.

XQuery [15][16][17], a query language developed by the W3C XML Query Working Group [14], is emerging as the standard language for querying XML data. XQuery provides constructs for expressing a large class of queries, but it does not include an explicit grouping construct comparable to the `group by` clause in SQL. Lack of an explicit grouping construct makes certain classes of common business analytic queries needlessly difficult to express and execute efficiently. Futhermore, business objects often have complex structures and varying schemas demanding advanced analytic capabilities that are even more difficult to express in the current XQuery syntax. In this paper, we examine these classes of queries and propose extensions to XQuery. We show how these extensions make analytic queries on XML data easier to express and optimize. These extensions include features that enable users to clearly express their intent for grouping complex data and for numbering of results. Moreover, the extensions enable new types of powerful analytic queries like rollup queries on "ragged-hierarchies" that are difficult to express in XQuery and even in SQL, its more mature relative.

The remainder of this paper is organized as follows: Section 2 explores some of the limitations of the existing XQuery syntax for queries requiring grouping. Section 3 introduces a proposal for an explicit grouping construct to remove these limitations with several examples that illustrate this proposal. Section 4 introduces a companion proposal for output numbering. Section 5 illustrates advanced OLAP-style queries using the extended constructs. Section 6 presents the results of some preliminary measurements of the impact of explicit grouping on the performance of an XQuery engine [2]. Related work and our conclusions are presented in Sections 7 and 8, respectively.

2. The Grouping Problem

We motivate the problem with some illustrative examples. We begin with an example query based on an input document consisting of a bibliography containing many books. The structure of a book is represented by the following example instance. A book may have zero or more authors and zero or one publisher.

---

*Work performed at IBM Almaden. Current Address: University of California, San Diego, La Jolla, CA 92093.

SIGMOD 2005, June 14–16, 2005, Baltimore, Maryland, USA.

```
<book>
  <title>Transaction Processing</title>
  <author>Jim Gray</author>
  <author>Andreas Reuter</author>
  <publisher>Morgan  Kaufmann</publisher>
  <year>1993</year>
  <price>59.00</price>
  <discount>6.00</discount>
</book>
```

Suppose that we need to find the average net price (after discount) of books for each publisher and year. This query may be expressed in the current XQuery syntax, following the style of the grouping examples shown in the XQuery specification [15] and in [1], as follows:

(Q1) *Find the average net price of books for each publisher and year.*

```
for $p in distinct-values(//book/publisher),
    $y in distinct-values(//book/year)
let $netprices :=
  //book[publisher = $p and year = $y]/(price – discount)
where fn:exists($netprices)
order by $p, $y
return
  <group>
    <publisher>{$p}</publisher>
    <year>{$y}</year>
    <avg-net-price>{avg($netprices)}</avg-net-price>
  </group>
```

This XQuery expression involves computing the set of distinct publishers and the set of distinct years, then finding the set of books corresponding to each (publisher, year) pair and computing the average net price for each non-empty set of books. As pointed out in [1], this method of expressing grouping has several problems. Arguably, this is not the most natural way for users to express the intent of grouping books by publisher and year. More seriously, a straightforward execution of this query would be very inefficient because it would involve many passes over the set of books resulting in expensive and redundant navigation and self-joins. In order to produce an efficient plan, an XQuery optimizer would need to detect the grouping implied by the query, which might be quite difficult to do in complex cases. The expression shown above also suffers from the problem of missing rows for books that do not have any publishers. This is a consequence of the fact that the non-existence of a publisher element for a book will not be represented in the sequence of publisher elements produced by the expression //book/publisher in the first for clause.

Another serious problem with the current XQuery syntax for grouping applications is illustrated by the following query:

(Q2) *Find the average price of books for each author.*

```
for $a in distinct-values(//book/author)
let $b := //book[author = $a]
return
  <group>
    {$a}
    <avg-price> {avg($b/price)} </avg-price>
  </group>
```

In our example input data set, each book can have multiple authors. The result of Q2 will contain a group for each individual author, paired with the average price of all books co-authored by that individual, regardless of other co-authors. For example, the output element with author Gray will contain the average price of all books by Gray, Gray and Reuter, etc. This may or may not capture the user's intent. A user might reasonably want to find the average price of books by each distinct set of authors. In this alternative query, books by Gray would be in a different group from books by Gray and Reuter. This alternative query is quite difficult to express using the current XQuery syntax, but can be easily expressed using a syntactic extension described in Section 3.3.

Difficulties with the current XQuery syntax are even more pronounced in the case of analytic queries that involve partitioning data in different ways in order to perform moving aggregations or comparisons of aggregations at different levels of a multi-dimensional hierarchy. For example, consider a document containing many `sale` elements. An example `sale` element is shown below.

```
<sale>
  <timestamp>2004-01-31T11:32:07</timestamp>
  <product>Green Tea</product>
  <state>CA</state>
  <region>West</region>
  <quantity>10</quantity>
  <price>9.99</price>
</sale>
```

Now consider the following query:

(Q3) *For each year and state, compare the total yearly sales in that state to the total yearly sales in the region containing the state, ordered by year, region, and state.*

The following example output element represents the result of this query:

```
<summary>
  <year>2004</year>
  <region>West</region>
  <state>OR</state>
  <state-sales>42500.00</state-sales>
  <region-sales>105860.00</region-sales>
  <state-percentage>40.1</state-percentage>
</summary>
```

In the current XQuery syntax, this query might be expressed as follows:

```
for $year in
    distinct-values(//sale/year-from-dateTime(timestamp))
```

```
for $region in distinct-values(//sale/region)
let $region-sales := //sale[region = $region and
                           year-from-dateTime(timestamp) = $year]
let $region-sum := sum( $region-sales/(quantity * price) )
for $state in distinct-values($region-sales/state)
let $state-sales := $region-sales[state = $state]
let $state-sum := sum( $state-sales/(quantity * price) )
order by $year, $region, $state
return <summary>
          <year>{ $year }</year>
          <region>{ $region }</region>
          <state>{ $state }</state>
          <state-sales>{ $state-sum }</state-sales>
          <region-sales>{ $region-sum }</region-sales>
          <state-percentage>
              { $state-sum * 100 div $region-sum }
          </state-percentage>
       </summary>
```

This query as written suggests seven passes over the data for computing the two levels of aggregations, but these can be computed in two passes if the system is clever enough to convert the multiple self-joins into a grouping query and combine the `sum` aggregate computations with the generation of groups. Notice how different Q1 and Q3 are: Q1 uses a cross-product of grouping values and an `exists` predicate to eliminate empty groups, while Q3 uses correlated expressions. Crafting a system that detects grouping queries in their varied forms is extremely difficult. We illustrate, in Section 3.1, how the constructs that we propose allow a more direct translation of this query into an efficient evaluation plan.

3. Explicit Grouping

3.1 Basic Syntax and Semantics

In XQuery, the term *tuple* denotes a set of named values that are related in some way. For example, a tuple might consist of several values that describe a book, such as the title, author, publisher, and price of the book. Tuples are not part of the XQuery data model [16], but are used as an intermediate result during the processing of XQuery expressions such as FLWOR expressions and quantified expressions. The `for` and `let` clauses within a FLWOR expression generate an ordered sequence of tuples. Each tuple, in turn, consists of one or more named variables that are bound to values. All the tuples generated during processing of a FLWOR expression contain the same set of variables.

Grouping, in general, is an operation that applies to a homogeneous sequence of tuples such as that found inside an XQuery FLWOR expression. Some subset of the variables, called *grouping variables*, are used to define the groups—one group for each distinct combination of values for the grouping variables. Within each group, the values of the remaining (non-grouping) variables are aggregated in some way. Because it provides a natural environment for a grouping operation, the FLWOR expression is the obvious place to add a grouping extension to the XQuery syntax. Grouping can be defined within a FLWOR expression without making any changes to the XQuery data model.

The following is a simplified version of the FLWOR syntax in the current XQuery specification, using the same EBNF notation but omitting or simplifying certain features that are not relevant to this discussion:

```
FLWORExpr ::= (ForClause | LetClause)+ WhereClause?
        OrderByClause? ReturnClause
ForClause ::= "for" "$" VarName ("at" "$" VarName)? "in" Expr
        ( "," "$" VarName ("at" "$" VarName)? "in" Expr)*
LetClause ::= "let" "$" VarName ":=" Expr
        ( "," "$" VarName ":=" Expr)*
WhereClause ::= "where" Expr
OrderByClause ::= "stable"? "order" "by" OrderSpec
        ("," OrderSpec)*
OrderSpec ::= Expr ("ascending" | "descending")?
ReturnClause ::= "return" Expr
```

We propose to extend the FLWOR syntax by introducing an optional `group by` clause following the current optional `where` clause. If present, the `group by` clause may be followed by optional `let` clauses and an optional `where` clause. The resulting syntax is as follows:

```
FLWORExpr ::= (ForClause | LetClause)+ WhereClause?
        (GroupByClause LetClause* WhereClause?)?
        OrderByClause? ReturnClause
GroupByClause ::= "group" "by"
        Expr "into" "$" VarName
        ( "," Expr "into" "$" VarName )*
        ( "nest" Expr "into" "$" VarName
        ( "," Expr  "into" "$" VarName)* )?
```

The `group by` clause can be thought of as operating on an input tuple stream that is generated by the clauses that precede the `group by`, and in turn generating an output tuple stream that is processed by the clauses that follow the `group by`. Both the input and the output tuple streams consist of tuples of bound variables, but the variables in the output tuple stream are not in general the same as the variables in the input tuple stream. The cardinality of the output tuple stream is less than or equal to the cardinality of the input tuple stream.

The expressions in the `group by` clause (immediately following the `group by` keyword) are called *grouping expressions* and the variables they are bound to are called *grouping variables*. The expressions in the `nest` clause are called *nesting expressions* and the variables they are bound to are called *nesting variables*. Each tuple in the output stream represents one group and consists of a value for each grouping variable and each nesting variable.

A group is formed for each distinct set of values for the grouping expressions. Section 3.3 details the semantics of equality used in computing the groupings. In the output tuple stream generated by a `group by` clause, each grouping variable is bound to a value

that is representative of its group. If the group contains nodes, the particular node that is chosen as the representative is implementation-dependent. Each nesting variable is bound to the sequence of values returned by a nesting expression for all the input tuples in the group.

Using the extended FLWOR syntax, we can express the first query in Section 2 as follows:

(Q1) *Find the average net price of books for each publisher and year.*

```
for $b in //book
group by $b/publisher into $p, $b/year into $y
nest $b/price - $b/discount into $netprices
return
  <group>
    {$p, $y}
    <avg-net-price>{avg($netprices)}</avg-net-price>
  </group>
```

In the above query, in the input tuple stream that is seen by the `group by` clause, each tuple contains one variable: `$b`, bound to one book. In the output tuple stream generated by the `group by` clause and used by the `return` clause, each tuple contains three variables: `$p`, bound to a `publisher` element, `$y`, bound to a `year` element, and `$netprices`, bound to a sequence of atomic values that represent net prices of books. Figure 1 shows an example of the tuple stream after the `group by` (and before the `return`) for the above query.

Since an empty sequence is considered to be a distinct value for grouping purposes, books with no publisher are present in the result of Q1 when it is expressed with explicit grouping.

$p	$y	$netprices
<publisher> Morgan Kaufmann </publisher>	<year> 1993 </year>	(65.00, 43.00, 57.00)
<publisher> Morgan Kaufmann </publisher>	<year> 1995 </year>	(34.00, 75.00)
<publisher> Addison-Wesley </publisher>	<year> 1993 </year>	(48.00)

Figure 1: Example variable bindings after group by in Q1

The optional `let` clauses following the `group by` clause allow group properties to be computed and reused in multiple places. The optional `where` clause, applied after the `let` clauses, allows filtering of tuples resulting from the `group by` operation (in effect, eliminating groups that do not satisfy a predicate).

The usefulness of these clauses is illustrated by the following example:

(Q4) *List all the publishers whose average book price is more than 100, in descending order by average price.*

```
for $b in //book
group by $b/publisher into $pub  nest $b/price into $prices
let $avgprice := avg($prices)
where $avgprice > 100
order by $avgprice descending
return
  <expensive-publisher>
    { $pub }
    <avg-price> {$avgprice} </avg-price>
  </expensive-publisher>
```

In this example, the `let` clause following the `group by` clause permits the average price of books by a given publisher to be computed once and then used in both a predicate and an element constructor.

The `nest` clause in our proposed syntax allows all of the information from the sequence of tuples contributing to a group to be retained for use in further computations after the `group by`. We illustrate this with the following example, which shows how query Q3 from Section 2 may be expressed with the extended `group by` syntax.

(Q3) *For each year and state, compare the total yearly sales in that state to the total yearly sales in the region containing the state, ordered by year, region, and state.*

```
for $s in //sale
group by $s/region into $region,
       year-from-dateTime($s/timestamp) into $year
  nest $s into $region-sales
let $region-sum := sum( $region-sales/(quantity * price) )
order by $year, $region
return
  for $s in $region-sales
  group by $s/state into $state
  nest $s into $state-sales
  let $state-sum := sum( $state-sales/(quantity * price) )
  order by $state
  return
    <summary>
          {$year, $region, $state}
          <state-sales>{ $state-sum }</state-sales>
          <region-sales>{ $region-sum }</region-sales>
          <state-percentage>
              { $state-sum * 100 div $region-sum }
          </state-percentage>
    </summary>
```

In the expression shown above, the first `group by` clause groups the sale elements by region and year and binds the sequence of sales in each group into the nesting variable `$region-sales`, and then the subsequent `let` clause binds the aggregated total sales for the region into the `$region-sum` variable. An example output tuple in the result of this grouping is shown below.

$region	$year	$region-sales	$region-sum
<region> West </region>	1993	(<sale> … </sale>, <sale> … </sale>)	124.90

Figure 2: Example bindings after group by region and year

For each tuple in this intermediate result, the nested FLWOR expression further partitions the `sale` elements in `$region-sales` by state into the nesting variable `$state-sales`, which is immediately aggregated into `$state-sum` using a `let` clause. The final `return` clause then formats the result, constructing new elements as needed. The result is the same sequence of `summary` elements produced by Q3 in Section 2.

Note that a `group by` clause may not necessarily contain a `nest` subclause, as illustrated by the following example, which computes the distinct pairs of publishers and titles in the input bibliography document. This query is similar in intent to the SELECT DISTINCT feature in SQL.

(Q5) *List the distinct pairs of publishers and titles.*

```
for $b in //book
group by $b/publisher into $pub, $b/title into $title
order by $pub, $title
return <pair> {$pub, $title} </pair>
```

In XQuery without an explicit `group by` operator, this query would be expressed by computing a Cartesian product of all publishers and titles and then filtering out the combinations that are not present in the input document. Apart from being awkward to write, this solution becomes rapidly more inefficient as the number of variables increases. In addition, the query expressed without explicit grouping would ignore books that have a publisher but no title, or a title but no publisher.

As a group is formed from a collection of tuples, the values returned by a given nesting expression are concatenated together into a sequence that is bound to a nesting variable. In this process, the sequences returned by the nesting expression for individual tuples are merged and lose their individual identity. This is necessary because the XQuery data model does not support nested sequences. One consequence is that the cardinalities of the sequences bound to two different nesting variables after grouping can be different. A second consequence is that any nesting expression that evaluates to an empty sequence will not appear in the resulting nested sequence. This has implications for count-related optimizations. Consider the following example query:

(Q6) *For each year display the number of books published and the list of book titles published that year.*

```
for $b in //book
group by $b/year into $year  nest $b/title into $titles
return
  <yearly-report>
    { $year}
    <book-count> {count($titles)} </book-count>
    <title-list> {$titles} </title-list>
  </yearly-report>
```

Since we know that each book has exactly one title, counting titles instead of books will produce the same result. If books were allowed to be published without titles (or with multiple titles) then we would need a second nesting variable to aggregate and count the books themselves. Alternatively, aggregating and counting books could be replaced by aggregating and counting a literal such as 1 (either explicitly by the user or by an optimizer). In SQL, such count optimizations may be applied based purely on the (non) nullability of the input arguments. In XQuery, one has to be aware of the number of allowed occurrences of child elements (derived potentially from schema information) in order to enable such optimizations in general.

3.2 Variable Scoping Rules

The values bound to variables in the output tuple stream of a `group by` are properties of groups, not of individual tuples. Therefore, these variables are not in scope until after the formation of groups is complete. For this reason, a grouping expression may not reference a grouping variable defined by another grouping expression.

In the clauses that follow the `group by` clause of a grouped FLWOR expression (i.e., the `let`, `where`, `order by`, and `return` clauses that follow `group by`), the scoping rules for variable names are as follows:

1. Grouping and nesting variables are in scope. Each of these variables is bound to a value derived from a group. If a grouping or nesting variable has the same name as a previously bound variable, it overrides the previous binding.

   In the following example, which illustrates the use of `group by` in inverting a hierarchy, the variable-name `$b` is iteratively bound to each book in the input stream. The same variable-name is then rebound explicitly to the sequence of all the books in the group corresponding to a given publisher in the output stream of the `group by`.

   (Q7) *Create a list of publisher elements, each containing the name of a publisher and a nested list of book elements representing books published by that publisher.*

   ```
   for $b in //book
   group by $b/publisher into $pub  nest $b into $b
   order by $pub
   ```

```
return
  <publisher>
    <name> {string($pub)} </name>
    <books> {$b} </books>
  </publisher>
```

2. Variables bound outside the FLWOR and not overridden inside the FLWOR remain in scope. However, variables that are bound in the FLWOR before the `group by` clause (i.e. variables bound by `for` and `let` clauses that precede `group by`) are not in scope. A reference to one of these variable names is a static error (unless the name has been rebound as a grouping or nesting variable). This is because the individual tuples of the input stream no longer exist in the output tuple stream. An alternative design might preserve the names of the variables in the input stream, automatically rebinding each of these variables to a sequence representing all the values for that variable in a given group. This design was considered and rejected on the grounds that implicit rebinding of variables is confusing and unnecessary. We believe that users should be very much aware that the variables in scope after grouping are different from the variables in scope before grouping, and that this awareness can be reinforced by giving different names to the post-grouping variables.

3.3 Equality

As mentioned in Section 2, the current XQuery syntax does not provide a way to express grouping based on equality of sequences. In our `group by` extension proposal, we allow grouping of sequences using `fn:deep-equal` [17] as the default comparison function. This default comparison function has the following properties:

1. If a grouping expression returns a sequence of items, each permutation is considered a distinct value.
2. An empty sequence is treated as a distinct value for grouping purposes.

Using the default comparison for grouping we can express the variant of query Q2 from Section 2 as follows:

(Q2a) *Find the average price of books for each distinct set of authors.*

```
for $b in //book
group by $b/author into $a
   nest $b/price into $prices
return
  <group>
    {$a}
    <avg-price> {avg($prices)} </avg-price>
  </group>
```

The above formulation of Q2a uses the default `deep-equal` function for grouping books by their authors; therefore it will generate an output element for each distinct permutation of authors. For example, a book by Gray and Reuter will not be in the same group as a book by Reuter and Gray.

In order to allow comparisons based on other definitions of sequence equality to be used instead of the `fn:deep-equal` function, we extend the `group by` clause with `using` subclauses as follows:

```
GroupByClause ::= "group" "by"
        Expr "into" "$" VarName ("using" QName)?
        ( "," Expr "into" "$" VarName ("using" QName)? )*
        ( "nest" Expr "into" "$" VarName
        ( "," Expr "into" "$" VarName)* )?
```

The `using` subclause specifies the comparison function to be used in comparing the grouping expressions. In order to be useful, such a function must be deterministic, transitive, and free of side effects. For example, a `set-equal` function might be defined that compares two sequences and returns `true` if one sequence is a permutation of the other. The following example defines such a function which is then used in the `using` subclause of the previous example to group by set semantics. Of course, this query would execute more efficiently if the `set-equal` function were built-in rather than user-defined.

```
declare function local:set-equal
  ($arg1 as item()*, $arg2 as item()*) as xs:boolean
  { every $i1 in $arg1 satisfies
      some $i2 in $arg2 satisfies $i1 eq $i2
    and every $i2 in $arg2 satisfies
      some $i1 in $arg1 satisfies $i1 eq $i2
  }
for $b in //book
group by $b/author into $a using local:set-equal
nest $b/price into $prices
return
  <group>
    {$a}
    <avg-price> {avg($prices)} </avg-price>
  </group>
```

The ability of the proposed syntax to form groups based on multi-valued grouping keys stands in contrast to the `group by` operator of SQL, in which each grouping key has a single value. The proposed XQuery grouping operator takes advantage of the flexibility of XML, in which a given path may return zero, one, or many elements.

3.4 Ordering

3.4.1 Ordering within Groups

In the tuple stream resulting from a `group by`, each nesting variable is bound to the sequence of values returned by the nesting expression for all the tuples in the group. We show, in this section, how the ordering of the items in this nesting sequence can enable order-based aggregations over the sequence to be ex-

pressed easily. The ordering of values in this nested sequence is, by default, dependent on the *ordering mode* in the Static Context [15]. Ordering mode is an XQuery feature that permits a user to specify whether the ordering of results is significant. In the current XQuery specification, ordering mode controls whether the result of a FLWOR expression with no `order by` clause preserves the binding order in which the tuples were generated by the `for` and `let` clauses. Since the values in a nested sequence come from tuples that have a well-defined ordering in the input stream, it is appropriate for ordering mode to control the ordering of these values. If ordering mode is `ordered`, the nested sequence preserves the ordering of the input tuple stream. If ordering mode is `unordered`, the ordering of values in the nested sequence is undefined.

Allowing the user to order the nested sequence with an ordering that is different from the default ordering would enable several types of "windowing" queries to be expressed more succinctly. In order to allow the nested sequence to be ordered in a different way we extend the `nest` subclause with `order by` specifications as follows:

GroupByClause ::= "group" "by"
Expr "into" "$" VarName ("using" QName)?
("," Expr "into" "$" VarName ("using" QName)?)*
("nest" Expr **OrderByClause?** "into" "$" VarName
("," Expr **OrderByClause?** "into" "$" VarName)*)?

If `order by` is specified for a given nesting variable, it controls the ordering of nested values in the same way that an `order by` clause orders the result of a FLWOR expression. When an `order by` clause is used inside a `nest` clause, the variables in scope for that `order by` clause are the variables of the input tuple stream (before groups are formed).

The following "moving window" example illustrates the usefulness of ordering the nested results within groups.

(Q8) *Within each region, order the sales by timestamp; then for each sale, show the total amount of the sale and the total amount of the previous ten sales in that region.*

The following example output element illustrates the expected result from this query:

```
<region name = "West">
  <sale>
    <timestamp>2004-04-01T11:32:07</timestamp>
    <sale-amount>25.00</sale-amount>
    <previous-ten-sales>180.00</previous-ten-sales>
  </sale>
  (... more <sale> elements ...)
</region>
```

The XQuery representation of this query is as follows:

```
for $s in //sale
group by $s/region into $region
nest $s order by $s/timestamp into $rs
order by $region
return
  <region name = string($region)>
    {for $s1 at $i in $rs
     return
       <sale>
         {$s1/timestamp}
         <sale-amount>{$s1/quantity * $s1/price}</sale-amount>
         <previous-ten-sales>
           {sum(for $s2 at $j in $rs
                where $j < $i and $j >= $i – 10
                return $s2/quantity * $s2/price)}
         </previous-ten-sales>
       </sale>
    }
</region>
```

In the above query, the nesting variable `$rs` is bound to the sequence of all the sales within each region ordered by the timestamp of the sale. In the `return` clause, the first `for` clause iterates through the sales in this ordered nested sequence binding `$s1` to each sales and `$i` to the position of `$s1` within the sequence. The `for-where-return` within the `sum` function performs a second iteration through the nested sequence to retrieve the ten previous sales, i.e., sales whose position in the ordered nested sequence is within 10 slots from `$s1`'s position (i.e., `$i`) in the sequence.

3.4.2 Ordering of Groups

In an ordinary FLWOR expression, the binding order in which tuples are generated by the `for` and `let` clauses is significant. If the FLWOR expression has no `order by` clause and ordering mode is `ordered`, the result of the FLWOR expression preserves binding order. If the FLWOR expression has an `order by` clause that includes the keyword `stable`, binding order is preserved among results that have equal ordering-keys. However, in a grouped FLWOR expression, the results represent properties of groups rather than individual input tuples, and groups have no meaningful input sequence that can be preserved. Therefore, in a grouped FLWOR expression that has no `order by` clause, the ordering of the result sequence is undefined. Similarly, the keyword `stable` is ignored in the `order by` clause of a grouped FLWOR expression.

3.5 Putting it all together

Since both the input and the output of a `group by` clause are tuple streams, in principle a `group by` clause could be followed by another `group by` clause in the same FLWOR. But there seems to be little motivation for this. The output tuple stream of a `group by` consists of grouping variables (which have already been grouped) and nesting variables (whose values are sequences, so it is difficult to perform any further grouping on them). Therefore, this proposal allows only one `group by` clause in a

FLWOR expression. Users can specify additional grouping operations within nested FLWORs (by including, for example, a FLWOR with a `group by` inside a `return` clause).

The combined EBNF syntax for FLWOR expressions with our proposed extensions is as follows:

```
FLWORExpr ::= (ForClause | LetClause)+ WhereClause?
              (GroupByClause LetClause* WhereClause?)?
              OrderByClause? ReturnClause
GroupByClause ::= "group" "by"
              Expr "into" "$" VarName ("using" QName)?
              ( "," Expr "into" "$" VarName ("using" QName)? )*
              ( "nest" Expr OrderByClause? "into" "$" VarName
              ( "," Expr OrderByClause? "into" "$" VarName)* )?
```

4. Output Numbering

In the current syntax of XQuery, the `for` clause of a FLWOR expression may contain a *positional variable*, identified by the keyword `at`, which is bound to an increasing sequence of ordinal numbers as the `for` clause iterates over its input sequence. For example, the following FLWOR expression might generate a numbered list of book titles authored by Jim Melton:

(Q9) *List the books authored by Jim Melton, including the title and ordinal number of each book.*

```
for $b at $i in //book[author = "Jim Melton"]
return
  <book>
    <number>{$i}</number>
    {$b/title}
  </book>
```

The result of Q9 might be as follows:

```
<book>
  <number>1</number>
  <title>Understanding the New SQL</title>
</book>
<book>
  <number>2<number>
  <title>Understanding SQL and Java Together</title>
</book>
```

It is important to understand that the positional variable in an `at` subclause is bound to the ordinal number of each input item in the binding order of the `for` clause, which is not necessarily the same as the order in which these items appear in the result of the FLWOR expression. In the above example, Jim Melton's books are numbered according to their position in the input document ("document order"). However, suppose that the query were modified slightly to list Jim Melton's books in order by price:

(Q9a) *List the books authored by Jim Melton in order of increasing price, with an ordinal number for each book.*

```
for $b at $i in //book[author = "Jim Melton"]
order by $b/price ascending
return
  <book>
    <number>{$i}</number>
    {$b/title, $b/price}
  </book>
```

The result of Q9a might be as follows:

```
<book>
  <number>2<number>
  <title>Understanding SQL and Java Together</title>
  <price>49.95</price>
</book>
<book>
  <number>1</number>
  <title>Understanding the New SQL</title>
  <price>54.95</price>
</book>
```

The result of Q9a illustrates that positional variables in the `for` clause represent input ordering and do not reflect the output order of a FLWOR expression that contains an `order by` clause. It would be useful in many cases to bind another kind of positional variable to represent output ordering. For this purpose, a positional variable could be added to the `return` clause, using syntax consistent with the existing syntax in the `for` clause, as follows:

ReturnClause ::= "return" **("at" "$" VarName)?** Expr

The semantics of a positional variable in a `return` clause are as follows: Each time the `return` clause is executed, the positional variable is bound to an integer representing the ordinal number of that execution, starting with 1.

Output positional variables are useful both for numbering output streams and for filtering them, as illustrated by the following example:

(Q9b) *Find the three most expensive books written by Jim Melton, in order of decreasing price, and generate an ordinal number for each.*

```
for $b in //book[author = "Jim Melton"]
order by $b/price descending
return at $rank
  if ($rank <= 3) then
    <book>
      <rank>{$rank}</rank>
      {$b/title, $b/price}
    </book>
  else ( )
```

In the existing XQuery syntax, which does not have a positional variable in the `return` clause, this query requires an additional FLWOR expression to reorder the input stream, as follows:

```
let $ranked-books :=
  (for $b in //book[author = "Jim Melton"]
   order by $b/price descending
   return $b)
return
  (for $b at $i in $ranked-books
   where $i <= 3
   return
     <book>
       <rank>{$i}</rank>
       {$b/title, $b/price}
     </book> )
```

Note that the proposal for a positional variable in the `return` clause of a FLWOR expression is completely independent of the proposal for a `group by` clause. The proposals are compatible and complimentary, but either can be adopted and used without the other. An example query that uses both grouping and output ordering is given in Q10 below.

(Q10) *For each month, show the monthly sales ranked by region, including the rank of each region for that month.*

The following is an example output element from Q10:

```
<monthly-report year="2004" month="10">
  <regional-results>
    <rank>1</rank>
    <region>West</region>
    <total-sales>3950.00</total-sales>
  </regional-results>
  (... more <regional-results> elements ...)
</monthly-report>
```

Q10 can be expressed using the proposed XQuery extensions as follows:

```
for $s in //sale
group by year-from-dateTime($s/timestamp) into $year,
         month-from-dateTime($s/timestamp) into $month
 nest $s into $month-sales
order by $year, $month
return
  <monthly-report year="{$year}" month="{$month}">
    {for $ms in $month-sales
     group by $ms/region into $region
       nest $ms/quantity * $ms/price into $sales-amounts
     let $sum := sum($sales-amounts)
     order by $sum descending
     return at $rank
       <regional-results>
         <rank> {$rank} </rank>
         { $region }
         <total-sales> {$sum} </total-sales>
       </regional-results>}
  </monthly-report>
```

5. Advanced Grouping

By exploiting the ability to group on complex objects, we can write OLAP cube and rollup queries [9] without further extension. Moreover, we can use the power of XML and XQuery to write queries that are nearly impossible in a relational system. As a first example, consider a rollup query on an unbounded "ragged-hierarchy", which is difficult for SQL to handle because it requires a fixed number of columns. The hierarchy is captured by the <categories> element in an extended collection of books:

```
<book>
  <title>Transaction Processing</title>
  <publisher>Morgan Kaufmann</publisher>
  <year>1993</year>
  <price>59.00</price>
  <categories>
     <software><db><concurrency/></db>
               <distributed/></software>
  </categories>
</book>
<book>
  <title>Readings in Database Systems</title>
  <publisher>Morgan Kaufmann</publisher>
  <year>1998</year>
  <price>65.00</price>
  <categories>
     <software><db/></software>
     <anthology/>
  </categories>
</book>
```

Inside the <categories> element is an arbitrary forest of elements that represent the book's categorical membership. For example, *Transaction Processing* is in four categories: "software", "software/db", "software/db/concurrency", and "software/distributed"; *Readings in Database Systems* is in three categories: "software", "software/db", and "anthology". We can answer rollup queries over such a hierarchy, for example:

(Q11) *Find the average price of books in each category.*

Example output:

```
<result><category>software</category>
        <avg-price>62.00</avg-price></result>
<result><category>software/db</category>
        <avg-price>62.00</avg-price></result>
<result><category>software/db/concurrency</category>
        <avg-price>59.00</avg-price></result>
```

```
<result><category>software/distributed</category>
        <avg-price>59.00</avg-price></result>
<result><category>anthology</category>
        <avg-price>65.00</avg-price></result>
```

To express Q11, we cannot directly use a group by on the category because that would group books by their entire categorization as a whole, rather than each individual category. Therefore, we define an auxiliary function that produces all individual categories of a book as follows:

```
declare function local:paths($x as element()*) as xs:string* {
  for $i in $x
  let $name := fn:local-name-from-QName(fn:node-name($x))
  return ($name,
          for $j in local:paths($i/*)
          return fn:concat($name, '/', $j))   }
```

The function recursively produces all the paths below the input elements by concatenating the name of each input element to the paths of its children. When used in the query below on our books, it produces the list of categories described above. The following query places each book into every category group to which it belongs then aggregates the prices:

```
for $b in //book
for $c in local:paths($b/categories/*)
group by $c into $category
nest $b/price into $prices
return <result><category>{$category}</category>
<avg-price>{avg($prices)}</avg-price></result>
```

As a second example, consider a datacube query:

(Q12) *Find the average price of books overall, by publisher, by year, and by (publisher, year) combinations.*

Example output:

```
<result><group/>
         <avg-price>63.50</avg-price></result>
<result><group><publisher>Morgan Kaufmann</publisher>
                        </group>
        <avg-price>63.50</avg-price></result>
<result><group><year>1993</year></group>
        <avg-price>59.00</avg-price></result>
<result><group><year>1998</year></group>
        <avg-price>65.00</avg-price></result>
<result><group><publisher>Morgan Kaufmann</publisher>
                    <year>1993</year></group>
        <avg-price>59.00</avg-price></result>
<result><group><publisher>Morgan Kaufmann</publisher>
                    <year>1998</year></group>
        <avg-price>65.00</avg-price></result>
```

To write Q12, we again use the "membership function" paradigm. In this case, we define a function that takes a sequence as input, which represents the cube dimensions, and produces the powerset of the sequence:

```
declare function local:cube($dims as item()*) as item()* {
  if empty($dims) then <group/>
  else for $subgroup in local:cube(fn:subsequence($dims, 2)))
       return ($subgroup,
               <group>{$dims[1], $subgroup/*}</group>)  }
```

The query again looks nearly identical to the previous two:

```
for $b in //book
let $pub := <publisher>{$b/publisher/*}</publisher>
for $cell in local:cube( ($pub, $b/year) )
group by $cell into $cell
nest $b/price into $prices
return <result>{$cell }
          <avg-price>{avg($prices)}</avg-price></result>
```

The only wrinkle to contend with is the optional publisher element, which is handled in the `let` clause by creating an empty publisher when one does not exist. If we omit this step, then books without a publisher are not reported when we are grouping by publisher, but this is another useful query to pose.

Of course, the `paths` and `cube` functions are not easy to write, and we expect that a common set of such membership functions will be provided by the implementations and eventually standardized. Furthermore, the system should understand the common membership functions to optimize the query. In these two examples, we replicate the input item into every group to which it belongs, which substantially increases in both storage and time requirements. However, many algorithms exist to process such queries much more efficiently [3].

The membership function paradigm generalizes SQL's `cube`, `rollup`, and `grouping-set` constructs through complex-object grouping without adding new constructs. Even though our datacube example is easily performed on relational data in SQL today, this proposal is substantially more powerful because the dimensions themselves can be complex objects including sets of items and the dimension may even be derived by other membership functions. For example, the `cube` function works with an unspecified number of dimensions and can, therefore, perform dynamic, data-driven datacube queries.

6. Experiments

One of the motivations for adding an explicit `group by` syntax to XQuery is that this syntax enables users to express their intent very clearly. Explicit grouping relieves the XQuery optimizer from the difficult task of recognizing a grouping "pattern" in a query that superficially looks like something more complex, such as nested iterations with a self-join. We believe that explicit grouping will enable systems to significantly improve performance by choosing better plans in cases where they would other-

wise fail to recognize the grouping "pattern." We implemented our proposed group by extension in System RX [2] which is a system supporting native XML storage and query processing described elsewhere in these proceedings. To investigate the performance impact of explicit grouping, we conducted an experiment with two equivalent sets of queries expressed in XQuery with and without explicit group by.

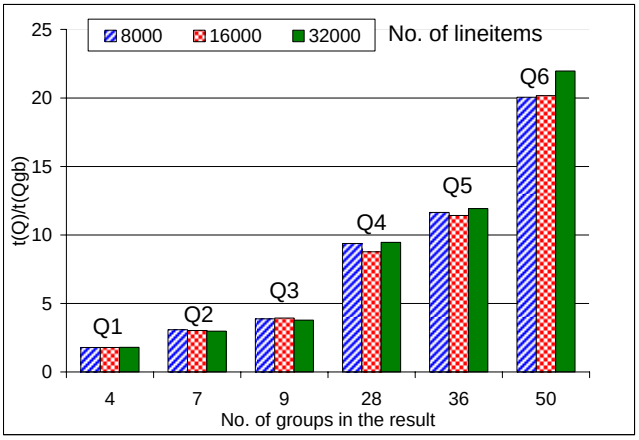
The experiments were based on XML documents containing purchase order data with each order containing detailed lineitem information about several items purchased, customer information, and other order information. Each `order` element had an average of four `lineitem` elements. Each `lineitem` element contained many child elements. The textual representation of each order document was about 3K bytes in size.

The queries used in this experiment were simple grouping queries. They involved extracting all the `lineitem` elements from all the `order` elements in the input collection and then grouping these `lineitem` elements by different combinations of child elements. The query templates with and without explicit grouping are presented in Table 1.

Table 1

	Query With Explicit Group By **(Qgb)**	Query Without Explicit Group By **(Q)**
Group By One Element (a)	for $litem in //order/lineitem group by $litem/**a** into $a nest $litem into $items return <r>{$a, count($items)}</r>;	for $a in distinct-values(//order/lineitem/**a**) let $items := for $i in //order/lineitem where $i/**a** = $a return $i return <r>{$a, count($items)}</r>;
Group By Two Elements (a, b)	for $litem in //order/lineitem group by $litem/**a** into $a, $litem/**b** into $b nest $litem into $items return <r>{$a, $b, count($items)}</r>;	for $a in distinct-values(//order/lineitem/**a**), $b in distinct-values(//order/lineitem/**b**) let $items := for $i in //order/lineitem where $i/**a** = $a and $i/**b** = $b return $i where exists($items) return <r>{$a, $b, count($items)}</r>;

Six different pairs of queries were generated from the templates by substituting different child elements of `lineitem` for the grouping terms "**a**" and "**b**" in order to vary the number of groups produced in the result. Each grouping element occurred exactly once in its parent (i.e., `lineitem`) element. Queries Q1, Q2, Q3, and Q6 denote the single element group by queries grouping by `shipinstruct`, `shipmode`, `tax`, and `quantity`, respectively. Queries Q4 and Q5 denote the two-element group by queries grouping by (`shipinstruct`, `shipmode`) and by (`shipinstruct`, `tax`), respectively. These queries were run against input collections containing different numbers of order documents. The number of lineitems that were aggregated ranged from 8K to 32K. No indexes were used in the experiments and no rewrites were performed to detect the group-by implied in the query without the explicit group by.



The chart shown above summarizes the results of the group by experiments. The numbers along the X-axis denote the number of groups in the result. The Y-axis shows the comparison of the execution times for the two different versions of the queries computed as t(Q)/t(Qgb) where t(Q) and t(Qgb) are the execution times (averaged over different runs) for the query expressed without and with the explicit group by, respectively. The performance of the query without the explicit group by, relative to that of the query with the explicit group by, deteriorates as the number of distinct values (or distinct pairs of values) of the grouping child elements increases. This is because it results in corresponding increases in the number of scans over the input collection.

7. Related Work

The problems associated with the lack of explicit constructs for grouping in XQuery have previously been identified by other researchers [1][4][6][12]. In [1], the authors point out many of the subtleties involved in correctly expressing grouping queries in the current XQuery syntax including the challenges associated with recognizing and optimizing the execution of such queries. In addition to the difficulties of dealing with grouping complex objects and empty sequences, they point out several issues involved in the reliance on element construction to represent "tuples" that arise in grouping and other analytic queries. In this paper, we have provided a concrete proposal for overcoming the limitations resulting from the lack of explicit grouping constructs. Although some of the issues related to the lack of explicit tuples such as the loss of types resulting from element construction have since been mitigated by the inclusion of a construction mode that preserves types in the current XQuery specification [15], other issues such as the loss of node identities through element construction still remain. While the addition of tuples to the XQuery data model is a potentially interesting area of research, it is beyond the scope of this paper. Some of the problems with lack of better support for group by, distinct, and outer-join in XQuery are also mentioned in [4] and proposed extensions are presented. Their proposed group by extension is similar to ours in spirit but complete details of syntax and semantics are not provided. They also do not consider options for specifying equality, ordering within groups, and more advanced forms of aggregation.

Other researchers have proposed methods to provide better support for grouping operations on XML data either at the logical algebraic level or at the physical operator level. Natix [8] provides a tuple-based algebra that includes grouping operators for the efficient construction of XML elements. Techniques for using rewrite rules to transform queries with grouping operations, expressed as nested queries in the current syntax, into efficient plans using group by operators have been explored by researchers in [6][7][11] and [12]. This is a promising optimization similar to the techniques for de-correlating nested queries into group-by and outer-join operators that have been developed for relational [10][13] and object-oriented [5] systems.

These transformations, are however, more difficult when grouping complex objects and were not considered in the papers mentioned above. Furthermore, it may not always be possible to detect the implied grouping operations and apply such transformations. Detecting a basic "group by" pattern in a query expressed without explicit grouping would involve identifying the pattern suggested by the combination of constructs (distinct-values, self-joins, existence tests on empty groups, etc.) that are required to express grouping. This combination is very difficult to detect in complex analytic queries that compute rollups and moving aggregates across deeply nested data. Furthermore, if any of these constructs is omitted in the query expression then the expression cannot simply be converted into a single grouping operation. The missing constructs would need to be compensated, potentially through correlated outer-joins (probably based on node-ids) and other operations. This would result in a more complex query expression which would be harder to further optimize through rewrite transformations.

While the rewrite transformations described in [6][7][11][12] are useful optimization techniques, we believe that they are complimentary and additional constructs are needed at the language level to enable easier expressivity and more efficient execution of analytic queries.

8. Conclusions

Analytical queries written using in the current XQuery draft are difficult to read, write, and process efficiently. We propose a series of XQuery extensions that greatly simplify analytical queries. The extensions cover not only basic grouping queries, but also advanced analytic queries like ranking, moving window queries, and allocation queries.

The proposal enables new features that go significantly beyond what is offered today, even from mature technologies like SQL. Most obvious is the ability to group by complex objects, including permutations and sets of items, and to override the default notion of group membership. By careful crafting of complex grouping values, the language can express rollup and cube queries without further extension, and can even handle "ragged-hierarchies," a notoriously difficult problem for SQL. Moreover, the new syntax is also useful for common tree transformations such as hierarchy inversions.

In summary, the proposed XQuery extensions offer a powerful and flexible set of tools that benefit the user and implementor alike.

9. Acknowledgments

We are grateful to Bobbie Cochrane for her valuable insights during the early stages of this work.

10. References

[1] K. Beyer, R.J. Cochrane, L.S. Colby, F. Özcan, H. Pirahesh, "XQuery for Analytics: Challenges and Requirements", XIME-P 2004.

[2] K. Beyer, et al, "System RX: One Part Relational, One Part XML", SIGMOD 2005.

[3] K. Beyer and R. Ramakrishnan, "Bottom-Up Computation of Sparse and Iceberg CUBEs", SIGMOD 1999, pages 359-370.

[4] V. Borkar and M. Carey, "Extending XQuery for Grouping, Duplicate Elimination, and Outer Joins", XML 2004.

[5] S. Cluet and G. Moerkotte, "Nested Queries in Object Bases", DBPL 1993.

[6] A. Deutsch, Y. Papakonstantinou and Y. Xu, "The NEXT Framework for Logical XQuery Optimization", VLDB 2004, pages 168-179.

[7] L. Fegaras, D. Levine, S. Bose, and V. Chaluvadi, "Query processing of streamed XML data", CIKM 2002, pages 126-133.

[8] T. Fiebig and G. Moerkotte, "Algebraic XML Construction in Natix", WISE(1), 2001, pages 212-221.

[9] J. Gray, A. Bosworth, A. Layman, and H. Pirahesh, "Data Cube: A Relational Aggregation Operator Generalizing Group-By, Cross-Tab, and Sub-Totals", ICDE 1996, pages 152-159.

[10] W. Kim, "On Optimizing an SQL-like Nested Query", ACM TODS, 7(3), pages 443-469, 1982.

[11] N. May, S. Helmer, and G. Moerkotte, "Three cases for query decorrelation in XQuery", XSym 2003, pages 70-84.

[12] S. Paparizos, et al, "Grouping in XML", EDBT 2002 Workshop, LNCS 2490, pages 128-147.

[13] P. Seshadri, H. Pirahesh and T. Y. Cliff Leung, "Complex Query Decorrelation", ICDE 1996, pages 450-458.

[14] World Wide Web Consortium (W3C). XML Query Working Group. See http://www.w3.org/XML/Query.

[15] World Wide Web Consortium (W3C). *XQuery 1.0: An XML Query Language.* W3C Working Draft, Apr. 4, 2005. See http://www.w3.org/TR/xquery/.

[16] World Wide Web Consortium (W3C). *XQuery 1.0 and XPath 2.0 Data Model.* W3C Working Draft, Apr. 4, 2005. See http://www.w3.org/TR/xpath-datamodel/.

[17] World Wide Web Consortium (W3C). *XQuery 1.0 and XPath 2.0 Functions and Operators.* W3C Working Draft, Apr. 4, 2005. See http://www.w3.org/TR/xpath-functions/.