FLWOR expressions

FLWOR expressions are probably the most powerful JSONiq construct and correspond to SQL's SELECT-FROM-WHERE statements, but they are more general and more flexible. In particular, clauses can almost appear in any order (apart that it must begin with a for or let clause, and end with a return clause).

Here is a bit of theory on how it works.

A clause binds values to some variables according to its own semantics, possibly several times. Each time, a tuple of variable bindings (mapping variable names to sequences) is passed on to the next clause.

This goes all the way down, until the return clause. The return clause is eventually evaluated for each tuple of variable bindings, resulting in a sequence of items for each tuple.

These sequences of items are concatenated, in the order of the incoming tuples, and the obtained sequence is returned by the FLWOR expression.

We are now giving practical examples with a hint on how it maps to SQL.

For clauses

For clauses allow iteration on a sequence.

For each incoming tuple, the expression in the for clause is evaluated to a sequence. Each item in this sequence is in turn bound to the for variable. A tuple is hence produced for each incoming tuple, and for each item in the sequence produced by the for clause for this tuple.

For example, the following for clause:

```
for $x in 1 to 3
```

produces the following stream of tuples (note how JSON syntax is so convenient that it can be used to represent these tuples):

```
{ "$x" : 1 }
{ "$x" : 2 }
{ "$x" : 3 }
```

The order in which items are bound by the for clause can be relaxed with unordered expressions, as described later in this section.

The following query, using a for and a return clause, is the counterpart of SQL's "SELECT display_name FROM answers". $x is bound in turn to each item in the answers collection.

Example 1. A for clause.

```
for $x in collection("answers")
return $x.owner.display_name
```

For clause expressions are composable, there can be several of them.

Example 2. Two for clauses.

```
for $x in ( 1, 2, 3 )
for $y in ( 1, 2, 3 )
return 10 * $x + $y
```

Example 3. A for clause with two variables.

```
for $x in ( 1, 2, 3 ), $y in ( 1, 2, 3 )
return 10 * $x + $y
```

A for variable is visible to subsequence bindings.

Example 4. A for clause.

```
for $x in ( [ 1, 2, 3 ], [ 4, 5, 6 ], [ 7, 8, 9 ] ), $y in $x()
return $y,
for $x in collection("faq"), $y in $x.tags()
return { "id" : $x.question_id, "tag" : $y }
```

It is also possible to bind the position of the current item in the sequence to a variable.

Example 5. A for clause.

```
for $x at $position in collection("answers")
return { "old id" : $x.answer_id, "new id" : $position }
```

JSONiq supports joins. For example, the counterpart of "SELECT q.title AS question, q.question_id FROM faq q JOIN answers a ON q.question_id = a.question_id" is:

Example 6. A join

```
for $question in collection("faq"),
    $answer in collection("answers")[ try { $$.question_id eq $questio
                                      catch * { false } ]
return { "question" : $question.title, "answer score" : $answer.score }
```

Note how JSONiq handles semi-structured data in a flexible way.

Outer joins are also possible with "allowing empty", i.e., output will also be produced if there is no matching movie for a captain. The following query is the counterpart of "SELECT q.title AS question, q.question_id FROM faq q LEFT JOIN answers a ON q.question_id = a.question_id".

Example 7. A join

```
for $question in collection("faq"),
    $answer allowing empty
        in collection("answers")[ try { $$.question_id eq $questio
                                  catch * { false } ]
```

```
return { "question" : $question.title, "answer score" : $answer.score }
```

Where clauses

Where clauses are used for filtering (selection operator in the relational algebra).

For each incoming tuple, the expression in the where clause is evaluated to a boolean (possibly converting an atomic to a boolean). if this boolean is true, the tuple is forwarded to the next clause, otherwise it is dropped.

The following query corresponds to "SELECT q.title as question, q.question_id as id FROM faq WHERE CONTAINS(question, 'NoSQL')".

Example 8. A where clause.

```
for $question in collection("faq")
where contains($question.title, "NoSQL")
return { "question" : $question.title, "id" : $question.question_id }
```

Order clauses

Order clauses are for reordering tuples.

For each incoming tuple, the expression in the where clause is evaluated to an atomic. The tuples are then sorted based on the atomics they are associated with, and then forwarded to the next clause.

Like for ordering comparisons, null values are always considered the smallest.

The following query is the counterpart of SQL's "SELECT a.display_name, a.score FROM answers a ORDER BY a.display_name".

Example 9. An order by clause.

```
for $answer in collection("answers")
order by $answer.owner.display_name
return { "owner" : $answer.owner.display_name, "score" : $answer.score }
```

Multiple sorting criteria can be given - they are treated like a lexicographic order (most important criterium first).

Example 10. An order by clause.

```
for $answer in collection("answers")
order by $answer.owner.display_name, $answer.score
return { "owner" : $answer.owner.display_name, "score" : $answer.score }
```

It can be specified whether the order is ascending or descending. Empty sequences are allowed and it can be chosen whether to put them first or last.

Example 11. An order by clause.

```
for $answer in collection("answers")
order by $answer.owner.display_name empty greatest, $answer.score
return { "owner" : $answer.owner.display_name, "score" : $answer.score }
```

An error is raised if the expression does not evaluate to an atomic or the empty sequence.

Group clauses

Grouping is also supported, like in SQL.

For each incoming tuple, the expression in the group clause is evaluated to an atomic. The value of this atomic is called a grouping key. The incoming tuples are then grouped according to the grouping key -- one group for each value of the grouping key.

For each group, a tuple is output, in which the grouping variable is bound to the group's key.

Example 12. An order by clause.

```
for $answer in collection("answers")
group by $question := $answer.question_id
return { "question" : $question }
```

As for the other (non-grouping) variables, their values within one group are all concatenated, keeping the same name. Aggregations can be done on these variables.

The following query is equivalent to "SELECT question_id, COUNT(*) FROM answers GROUP BY question_id".

Example 13. An order by clause.

```
for $answer in collection("answers")
group by $question := $answer.question_id
return { "question" : $question, "count" : count($answer) }
```

The following query is equivalent to "SELECT question_id, AVG(score) FROM answers GROUP BY question_id".

Example 14. An order by clause.

```
for $answer in collection("answers")
group by $question := $answer.question_id
return { "question" : $question, "average score" : avg($answer.score) }
```

JSONiq's group by is more flexible than SQL and is fully composable.

Example 15. An order by clause.

```
for $answer in collection("answers")
group by $question := $answer.question_id
return { "question" : $question, "scores" : [ $answer.score ]  }
```

Unlike SQL, JSONiq does not need a having clause, because a where clause works perfectly after grouping as well.

The following query is the counterpart of "SELECT question_id, COUNT(*) FROM answers GROUP BY question_id HAVING COUNT(*) > 1"

Example 16. An order by clause.

```
for $answer in collection("answers")
group by $question := $answer.question_id
where count($answer) gt 1
return { "question" : $question, "count" : count($answer)  }
```

Let clauses

Let bindings can be used to define aliases for any sequence, for convenience.

For each incoming tuple, the expression in the let clause is evaluated to a sequence. A binding is added from this sequence to the let variable in each tuple. A tuple is hence produced for each incoming tuple.

Example 17. An order by clause.

```
for $answer in collection("answers")
let $qid := $answer.question_id
group by $question := $qid
let $count := count($answer)
where $count gt 1
return { "question" : $question, "count" : $count  }
```

Note that it is perfectly fine to reuse a variable name and hide a variable binding.

Example 18. An order by clause.

```
for $answer in collection("answers")
let $qid := $answer.question_id
group by $qid
let $count := count($answer)
where $count gt 1
let $count := size(
  collection("faq")[ $$.question_id eq $qid ].tags
)
return { "question" : $question, "count" : $count }
```

Count clauses

For each incoming tuple, a binding from the position of this tuple in the tuple stream to the count variable is added. The new tuple is then forwarded to the next clause.

Example 19. An order by clause.

```
for $question in collection("faq")
order by size($question.tags)
count $count
return { "id" : $count, "faq" : $question.title }
```

Map operator

JSONiq provides a shortcut for a for-return construct, automatically binding each item in the left-hand-side sequence to the context item.

Example 20. A simple map

```
(1 to 10) ! ($$ * 2)
```

Example 21. An equivalent query

```
for $i in 1 to 10
return $i * 2
```

Composing FLWOR expressions

Like all other expressions, FLWOR expressions can be composed. In the following examples, a FLWOR is nested in an existential quantifier, nested in a FLWOR, nested in a function call, nested in a FLWOR, nested in an array constructor:

Example 22. Nested FLWORs

```
[
  for $answer in collection("answers")
  let $oid := $answer.owner.user_id
  where count(
      for $question in collection("faq")
      where some $other-answer in collection("answers")
                [$$.question_id eq $question.question_id
                 and
                 $$.owner.user_id eq $oid]
            satisfies
            $other-answer.score gt $answer.score)
       return $question
  ) ge 2
  return $answer.owner.display_name
]
```

Ordered and Unordered expressions

By default, the order in which a for clause binds its items is important.

This behaviour can be relaxed in order give the optimizer more leeway. An unordered expression relaxes ordering by for clauses within its operand scope:

Example 23. An unordered expression.

```
unordered {
  for $answer in collection("answers")
  where $answer.score ge 4
  return $answer
}
```

An ordered expression can be used to reactivate ordering behaviour in a subscope.

Example 24. An ordered expression.

```
unordered {
  for $question in collection("faq")
  where ordered { exists(for $answer at $i in collection("answers")
                         where $i eq 5
                         where $answer.question_id := $question.question_id
                         return $answer) }
  return $question
}
```