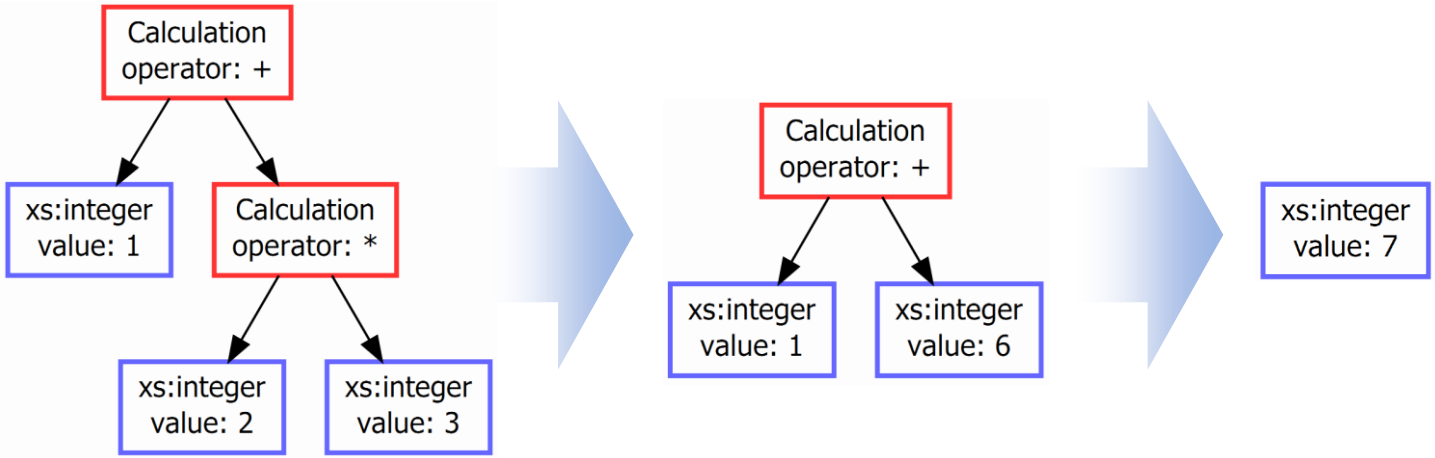
XML Technologies
XQuery: Evaluation

Christian Grün
Database & Information Systems Group
Universität Konstanz

XQuery Evaluation

Introduction

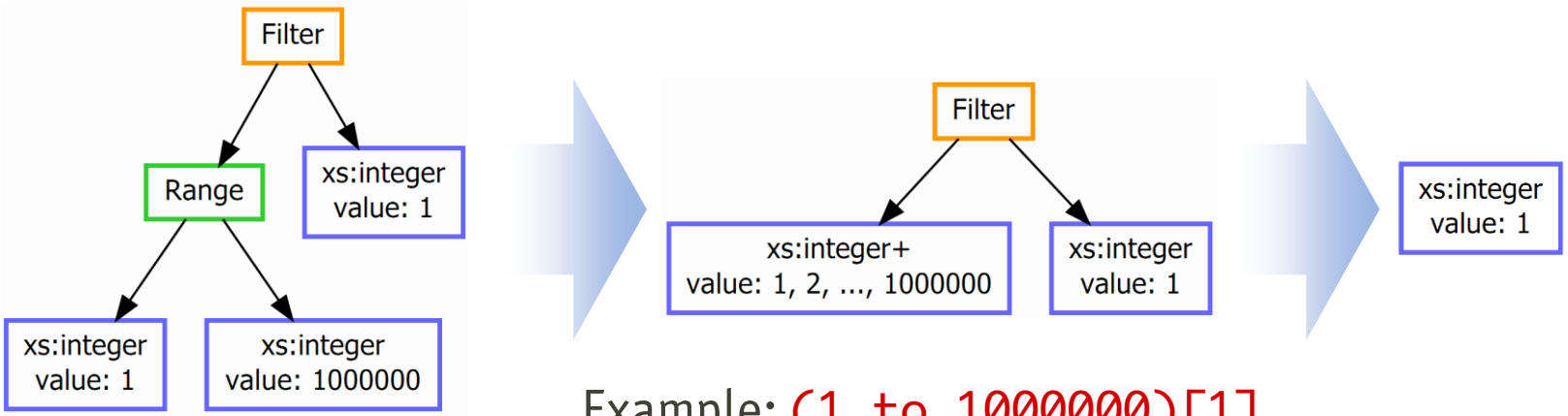
- our expression tree will now be *evaluated*
- all nodes are recursively traversed and replaced by their computed value



XQuery Evaluation

Challenge

- conventional evaluation may produce huge intermediate results
- final result is possibly much smaller



Example: `(1 to 1000000)[1]`

XQuery Evaluation

Iterative processing

- idea: request, process and pass on query results *one by one*
- main advantage: *constant* memory usage
- alternative wording: *streaming*, *pipelining*

Lazy evaluation

- computation of an expression is delayed until its value is required
- some values in an expression may not need to be computed at all

✎ How many items need be computed for `(1 to 1000000)[1]`?

XQuery Evaluation

Iterator pattern

- instead of `Evaluate()`, expressions get a `Next()` function, which returns a single result item
- a `null` value is returned if an expression is *exhausted*, i.e., if all values have been returned

Alternative

- expressions can be equipped with an `Iterator()` function, which creates and returns a new iterator
- advantage: the expression itself will be *immutable* (need not be reset, etc.)

XQuery Evaluation

Iterator examples

IndexOf.Iterator.Next() : Item

```
Require:
    ITERATOR := iterator on the input sequence ($seq)
    ITEM := item to be found ($item)
    POS := 0 (current iterator position)
loop
    POS := POS + 1
    item := ITERATOR.Next()
    if item = null then
        return null
    else if ITEM equals item then
        return new Integer(POS)
    end if
end loop
```

`fn:index-of` function

Intersect.Iterator.Next() : Node

```
Require: ITERATORS := array with iterators on all operands
nodes := array with cached nodes
for i := 0 to #ITERATORS−1 do
    nodes[i] := ITERATORS[i].Next()
    return null if nodes[i] is null
end for
i := 1
while i < #nodes do
    diff := nodes[0].ID − nodes[i].ID
    if diff < 0 then
        nodes[0] :=ITERATORS[0].Next()
        return null if nodes[0] is null
        i := 1
    else if diff > 0 then
        nodes[i] :=ITERATORS[i].Next()
        return null if nodes[i] is null
    else
        i := i + 1
    end if
end while
return nodes[0]
```

`intersect` operator (for *ordered* input nodes)

XQuery Evaluation

Iterator: meta data

The result size of an expression can be help to speed up evaluation:

- `Size()` returns the number of iterated items, or `-1` if the number is unknown
- `Get(i)` retrieves the specified item
- `Get-Remaining()` returns remaining (possibly all) items
- `Reset()` resets the iterator

Example

Count.Item() : Item

Require: EXPR := argument of the count function

```
iter := EXPR.Iterator()
c := iter.Size()
if c = −1 then
    repeat
        c := c + 1
    until iter.Next() = null
end if
return new Integer(c)
```

`fn:count` function, handling both known and unknown sizes

XQuery Evaluation

Blocking operators

- not all expressions can be evaluated in an iterative way
- *blocking operators* require all items to be cached before the actual result can be computed and returned

Examples

- `for $i in 1 to 100 order by $i descending return $i`
- `for $i in 1 to 5 let $j := "A" group by $j return $j`
- `/descendant::A/descendant::B`

✎ Can you explain why there expressions are *blocking*?

XQuery Evaluation

Location path blocking

Optimized node caching:

- cache *all* incoming nodes
- only sort nodes, and remove duplicates, if necessary:
  - *diff* = 0: new node equals last entry ⇢ node is ignored
  - *diff* < 0: new node appears before last entry ⇢ ORDERED flag is invalidated

NodeIterator.Add(Node n)

Require:
NODES := array with cached nodes
ORDERED indicates if nodes are ordered and duplicate-free

```
iter := EXPR.Iterator()
c := iter.Size()
if ORDERED and #NODES > 0 then
  diff := n.ID − NODES[#NODES - 1].ID
  if diff = 0 then
    return
  else if diff < 0 then
    ORDERED = false
  end if
end if
add n to NODES
```

XQuery Evaluation

Iterative processing: summary

- iterative query evaluation is an elegant approach to process large data sets
- it facilitates both *constant* memory usage and *faster* execution times
- iterative processing is limited by *blocking operators*
- iteration slows down querying if values are requested *multiple times*

☞ best results are gained by *combining* iterative and caching algorithms

☞ all existing information on data (number of results, positional access) should be utilized!