XML Technologies

XQuery: Compilation

Christian Grün
Database & Information Systems Group
Universität Konstanz
XQuery Compilation

Introduction

• our Abstract Syntax Tree will now be recursively traversed
• expressions will be decorated with certain properties:
  • data type of resulting values (→ type checking)
  • expected number of results
  • accessed databases
  • optimization flags (e.g.: iterable, ordered, duplicate-free)
• expressions will be optimized and rewritten

BaseX: check query info! COMPPLAN=false shows plan before compilation
XQuery Typing

Sequence types

- consists of item type and cardinality (constraining the number of items)
- cardinality is specified by a wildcard (? , *, + , or nothing)
- types can be explicitly specified in variable declarations:
  `let $x as xs:integer+ := (1, 2, 3) return $x`
- types can also be enforced by using casts:
  `xs:double(1), xs:byte(2), xs:string(<a>3</a>)`
- otherwise, the type will be derived from the input or the expected result
- if sequences contain different types, the most general type will be assigned:
  `(xs:integer(1), xs:string("a")) → item()+`
XQuery Typing

Type checking
• process of *statically* assigning and verifying types to expressions
• raise errors at *compile time* if wrong types are found: "16" + 1984
• in a *strongly typed* language, no type errors will occur at evaluation time
  ➔ Is XQuery, Java, PHP strongly or weakly typed?

Schema types
• if a *schema* is available, types will be assigned to all XML nodes
• example: the following query will only succeed if $X$ is of numeric type:
  
  \[
  \text{for } \$i \text{ in } //X \text{ return } \$i + 23 \]
  

XQuery Typing

Pessimistic vs. optimistic typing

- *pessimistic* typing rejects any expression that might fail due to type issues
- too restrictive for XQuery, as many operations on untyped data would fail
- example: \((<x>10</x>, <x>ABC<x>) = 10\) ✎ Result?
- with *optimistic* typing, an operation may succeed even if not all items have valid types:
  \[1984 \ldotp\ldotp\ldotp (\text{if}($b)\text{ then }1\text{ else }"1") \rightarrow 1985\text{ or }\n\] (depends on $x$)
- drawback: requires additional type checks at evaluation time
XQuery Optimization

Idea: expression tree is rewritten into a (hopefully) cheaper plan.

Example: how can the following (imperative) code be simplified?

```plaintext
code
zero = true
s = 0
for i = 0 ... 100:
  s += zero ? 0 : i
```

Motivation

- programmers have *too less experience* with low-level operations
- high-level languages *don’t even allow* low-level optimizations
- *in-depth knowledge* on the processor may be needed to write optimal code
XQuery Optimization

Evaluating XQuery

Performance of Java-based XQuery processors is influenced by multiple layers:
1. XQuery processor code is compiled to Java byte code
2. XQuery is compiled to an expression tree
3. AST is optionally transformed to Java byte code as well (Saxon-EE, Qexo)
4. Resulting expression is further optimized by Just-In-Time compiler

Observations

- code that will eventually be executed is *hardly predictable*!
- but: most *high-level optimizations* are also beneficial on lower levels
XQuery Optimization

Constant folding

- some sub-expressions will always evaluate to the same result
- such values can be \textit{pre-evaluated} at compile time
- constant folding is \textit{recursively} performed on all nodes (→ same for other optimizations)
- original or optimized expression is returned as \textit{new expression}
XQuery Optimization

Constant propagation

- variables with constant values are \textit{statically bound} to their references
- straightforward in functional languages, as all global variables are \textit{immutable}

Enabling transformation

- single optimization triggers optimizations of parent expressions

Example

\begin{verbatim}
declare variable $\pi := 3.14159265
5 * $\pi
\end{verbatim}

1. constant propagation:
   \( \pi \) will be replaced with its value
2. constant folding:
   \( 5 * 3.1415... \) will be pre-evaluated
3. remaining expression is a single
   \texttt{xs:decimal} value: \texttt{15.70796326793966}
XQuery Optimization

Copy propagation

- replaces targets of direct assignments with their values
- reduces redundant operations
- equivalent expressions are often rewritten to identical expressions (💡 What’s the difference?)
- allows simplified detection and elimination of common sub-expressions (see later)

Example

```
let $a := 1
let $b := $a
return ($a + 1) * ($b + 1)
```

Copy propagation

```
let $a := 1
return ($a + 1) * ($a + 1)
```

Constant propagation

```
(1 + 1) * (1 + 1)
```
XQuery Optimization

Boolean rewrites

- *and/or* are commutative
  - Short-circuiting: stop after partial evaluation
    - `true() or $x ≡ true()`
    - `false() and $x ≡ false()`
    - `true() and $x ≡ boolean($x)`

Order of operands

- note that logical expressions are *not deterministic* in XQuery:
  - "𐀣" castable as `xs:int`
  - and `xs:int("𐀣")`
  - may, or may not, yield an error!

Equivalences

- `A and B ≡ if A then boolean(B) else false()`
- `A or B ≡ if A then true() else boolean(B)`

What alternative expression can be used?
XQuery Optimization

General comparisons

- remember *existential semantics*! at least 1 comparison must be true:
  \( () = () \Rightarrow \text{false} \)
  \((1,2) = (2,3) \Rightarrow \text{true} \)
- can *only be negated* if operands will exactly yield one item:
  \( \text{not}(1 \neq (1,2)) \neq 1 = (1,2) \)
- empty sequence always yields false:
  \( () \\neq \$x \equiv \text{false}() \)

Value comparisons

- defined for single items
- can always be negated (provided that input has correct type)
- simplify of boolean operation:
  \( \text{boolean}($X) \text{ eq } \text{false}() \equiv \text{not}($X) \)
- simplify comparisons with empty sequence:
  \( (1,2,3) \text{ eq } () \equiv () \)
XQuery Optimization

Logical equivalencies

Numerous optimizations exist for if(...) then ... else:
1. if condition is a constant value, return resulting branch:
   if(1) then $a$ else $b$ ≡ $a$
2. if both branches are identical, skip condition:
   if($a$) then $b$ else $b$ ≡ $b$
3. swap branches:
   if(not($a$)) then $a$ else $b$ ≡ if($a$) then $b$ else $a$
4. return simplified condition:
   if($a$) then true() else false() ≡ boolean($a$)
5. return negated condition:
   if($a$) then $b$ else true() ≡ not($a$) or $b$
XQuery Optimization

Strength reduction
Replace expensive operations with cheap equivalents:

- exponentiation → multiplication:
  \( x^2 \equiv x \times x \)
- multiplication → addition:
  \( x \times 2 \equiv x + x \)
- multiplication/division → bit shifting
  \( x / 2^c \equiv x \gg c \)

⚠️ What limits exist when shifting bits?

General notes

- evaluation of duplicated operands may outweigh original costs
- \( x \times 12 \equiv x + x + x + x + x + x + x + x + x + x \)
- many rewritings are not applicable to functional languages:
  \[ c = 0 \]
  \[ \text{for } (i = 1 \ldots 100) \]
  \[ c += 2 \]
XQuery Optimization

Mathematical rewritings

Simplify algebraic expressions:

• remove neutral elements:
  \[ 0 + x - 0 \equiv x \]
  \[ 1 \times x / 1 \equiv x \]

• remove absorbing elements:
  \[ x \times 0 \equiv 0 \]

-floating-point arithmetics may cause unexpected results:
  \[ \text{xs:float("NaN")} \times 0 \rightarrow \text{NaN} \]

• regroup terms:
  \[ (a^2) + 10 + (a^3) + 5 \]
  \[ \equiv (a^2) + (a^3) + 10 + 5 \]

• apply distributive law:
  \[ (3 \times a) - (2 \times a) \]
  \[ \equiv (3-2) \times a \equiv 1 \times a \equiv a \]

• merge identical expressions (contrary to strength reduction):
  \[ a + a + a + a \equiv 4 \times a \]
XQuery Optimization

Function re rewritings

\textbf{count()} \\
• pre-evaluate if result size is known at compile time: \\
  \texttt{count((1,"A"))} \equiv 2 \\
• use \texttt{empty()} and \texttt{exists()} to speed up evaluation: \\
  \texttt{count($x) = 0} \equiv \texttt{empty(X)} \\
  \texttt{count($x) > 0} \equiv \texttt{exists(X)} \\

\texttt{empty((1,"B"))} \equiv \texttt{false}

\textbf{boolean()} \\
• get rid of redundant conversion: \\
  \texttt{if(boolean($x))} \equiv \texttt{if($x)} \\
\textbf{not()} \\
• simplify boolean tests: \\
  \texttt{not(not($x))} \equiv \texttt{boolean($x)} \\
  \texttt{not(empty($x))} \equiv \texttt{exists($x)}
XQuery Optimization

Constant folding

- applied to built-in functions:
  \[
  \text{string-length('ABC')} \equiv 3 \\
  \text{substring('ABC',3)} \equiv 'C' \\
  \text{contains('aa','a')} \equiv \text{true}
  \]

- utilize typing information:
  \[
  \text{some } n \text{ in ('A','B')} \\
  \text{satisfies exists}(n) \equiv \text{true()}
  \]

- utilize meta information:
  \[
  \text{count(for } i \text{ in 1 to 3} \\
  \text{return } i)
  \]

Non-determinism

- constant folding is only applicable to deterministic functions!
  \[
  \text{for } n \text{ in 1 to 3} \\
  \text{return } \text{math:random()}
  \]

- evaluation order is important if functions have side effects:
  \[
  \text{file:write("A", <a/>)}, \\
  \text{file:delete("A")}
  \]
XQuery Optimization

Function inlining

• function call is replaced with body
• function arguments are bound as variables and wrapped into FLWOR expression
• only reasonable for functions of moderate size

Can function inlining be applied to recursive functions?

Example

declare function
local:dbl($x) { $x + $x };
local:dbl(4) + 2

Inline function:
≡ (let $X := 4 return $X+$X) + 2

Propagate and fold constants:
≡ (4 + 4) + 2
≡ 10
XQuery Optimization

Dead code elimination
Progressively clean up obsolete subexpressions:
• dead code that will never be evaluated (e.g. `then/else` branch)
• (type)switch branches
• propagated variables
• inlined functions

Common subexpression elimination
Merge identical expressions that would otherwise be evaluated multiple times:
```xquery
for $a in 1 to 5
return ($a + 1) * ($a + 1)
```
Bind common expression to new variable:
```xquery
for $a in 1 to 5
let $x := ($a + 1)
return $x * $x
```
⚠️ Why is CSE comparatively expensive?
declare function local:a() { 10 + 30 }
declare variable $b := 20 - local:a() \div 10$
let $c := b \times 5$
let $d := \begin{cases} 
c - 60 & \text{if } c > 60 \\
c & \text{otherwise} 
\end{cases}$
return $d \times (20 \div local:a())$
XQuery Optimization

**XPath: descendant-or-self steps**

- the two slashes `//` are a shortcut for `/descendant-or-self::node()/`
- `//x` is resolved to `/descendant-or-self::node()/child::x`
- expensive construct, as *all document nodes* are addressed by first step!
- effective rewriting: `/descendant::*x`
- *not possible* if step is followed by position predicate (remember?): `//x[1]`
- (probably needed) alternative: enclose path by parentheses: `(//x)[1]`

What’s the difference between the two writings?

How can `/descendant-or-self::node()/self::*x` be rewritten?
XQuery Optimization

Name index rewritings

The index remembers which *element* and *attribute names* exist:
- if a single name does not exist, the whole path might disappear:
  \[
  /\text{books/book/ctx/title} \\
  \equiv /\text{books/book/()/title} \\
  \equiv ()
  \]
  - name index lookups are *local* operations (no path context needed)

Path summary rewritings

Contains information and statistics on *unique document paths*:
- count number of resulting nodes, e.g.: \(\text{count}(/a/b) \rightarrow 123\)
- skip paths that won’t yield results:
  \[
  /\text{books/book/book} \equiv () \\
  /\text{books/text()/text()} \equiv ()
  \]

\(\checkmark\) Which of the above queries can be optimized without index?
XQuery Optimization

Path summary re rewritings

```xml
<A>
  <B><C></B>
  <D><C></D>
  <D><D></D>
</A>
```

<table>
<thead>
<tr>
<th>#</th>
<th>path</th>
<th>optimized</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>//B</td>
<td>/A/B</td>
</tr>
<tr>
<td>2</td>
<td>//C</td>
<td>/A/*/C</td>
</tr>
<tr>
<td>3</td>
<td>//D</td>
<td>/A/D</td>
</tr>
<tr>
<td>4</td>
<td>//E</td>
<td>()</td>
</tr>
<tr>
<td>5</td>
<td>//A//C</td>
<td>/A/*/C</td>
</tr>
</tbody>
</table>

1. descendant steps are rewritten to cheaper *child steps*
2. multiple matches are masked with an *element wildcard*
3. *union operator* combines paths with matching nodes on different levels
XQuery Optimization

FLWOR expressions

- most powerful construct in XQuery
- opens doors to many (also competing) optimizations
  - finding the best representation is often impossible

Code elimination

A complete FLWOR expression can be replaced by an empty sequence if...

- any for clause loops over an empty sequence
- the where clause would yield false for all iterations
- the return clause yields an empty sequence
XQuery Optimization

Constant propagation

FLWORs contain for and let clauses:
- for binds all items one by one
- let binds all items at at time
- the expression of a let variable can be statically bound to its reference

Can be counter-productive: should be avoided if variable is used more than once, or if it is expensive to compute

Example

```xml
for $i in 1 to 10
let $a := "got it"
return <a>{ $a }</a>
```

Optimized

```xml
for $i in 1 to 10
return <a>{ "got it" }</a>
```
XQuery Optimization

Hoisting invariant variables

- move variables outside the loop that do not depend on variables that have been bound before
- avoids repeated evaluation of variables that will always yield the same result
- does not work with non-deterministic operations or node constructors (Why?)

Example

```
for $a in 1 to 4
  for $b in 1 to 100
    let $c := math:sqrt($a)
    return $b * $c
```

Optimized

```
for $a in 1 to 4
  let $c := math:sqrt($a)
  for $b in 1 to 100
    return $b * $c
```
XQuery Optimization

Algorithm

1. loops over bindings, starting from second most outer variable (first binding cannot be moved)
3. selects each binding that returns a single, deterministic value
5. loops backwards through all bindings that don’t declare any variables of the current binding
10. if a destination is found, the binding is moved

```
FLWOR.HoistInvariantVariables()
1  for i := 1 to #BINDINGS do
2     inner := BINDINGS[i]
3     if inner returns single, deterministic value then
4         dest := null
5         for o := i - 1 to 0 do
6             outer := BINDINGS[o]
7             break if outer declares variables used in inner
8             dest := outer
9         end for
10        if dest is not null then
11            move inner before dest
12        end if
13 end if
14 end for
```
XQuery Optimization

Predicate rewritings

- normalizes path representation by rewriting \textit{where} clause to predicates whenever possible
- a \textit{where} clause may contain multiple tests, separated by \textit{and}
- resulting representation will be further rewritten by XPath and FLWOR optimizations

Example

\begin{verbatim}
for $a$ in //code
  where $a$ and $a$ = "JP"
  return $a$
\end{verbatim}

Predicate rewriting

\begin{verbatim}
for $a$ in //code[.][. = "JP"]
  return $a$
\end{verbatim}

Simplification

//code[.][. = "JP"]
XQuery Optimization

Example: FLWOR rewriting

```xml
for $item in doc("xmark")
  /descendant::item
where $item/paid = "Creditcard"
return $item
```
XQuery Optimization

Value index rewritings in BaseX

Complex Operation, divided into three steps:
1. Retrieve database context for path expression
2. Analyze all predicates for index access
3. Rewrite path: introduce index access, followed by inverted steps

Comparison with relational databases

- Indexes are uniquely coupled with specific columns of a table
- Table data is flat, no hierarchies need to be resolved
  - Steps 1 and 3 can be skipped
XQuery Optimization

1. Retrieve database context

The context of a path can be specified...

1. globally: /based/on/global/context
   - Take global context (if it refers to a database)

2. via functions (doc(), collection(), ...): doc("based")/on/function
   - Pre-evaluate function and take resulting database node

3. as input of other expressions:
   - for $n in doc("files.xml")//name return doc($n)/path
   - Cannot be retrieved (context is not available at compile time)
XQuery Optimization

2. Analyze predicates

Expressions suitable for index rewritings may occur at various positions:

- Equality test: //medium[type = ("Journal","Paper")]
- Range test: /books/book/year[text() > 2000]/../title
- Full-text: //medium/title[text() contains text "L’Étranger"]
- Logical operations, multiple predicates:
  //medium[year = 1929][(type = "DVD" or type = "Video") and title contains text "chien andalou"]/description

Challenge: choose predicate(s) that will be evaluated faster than others!
XQuery Optimization

2. Analyze predicates

Best average results are obtained if a single predicate is rewritten:
2. parse all axis steps
4. parse all predicates
6. check if predicate can be rewritten
8. choose predicate with lowest costs
9. if costs are 0, predicate will yield no results: skip remaining tests
13. return index context, containing cheapest index candidate
XQuery Optimization

2. Analyze predicates: an example

The IndexAccessible method of the General Comparison (⚠️ remember?) ensures that the following conditions are fulfilled:

- the equality operator is used (invalid: //*[text() != "FAIL"])
- one of the operands contains only axis steps, concluding with a text() or attribute step (invalid: //*[substring(text(),4)) = "FAIL"])
- the other operand contains no reference to the current context (invalid: //*[text() = name(.)])
- all resulting items of the second operand must strings or untyped (invalid: //*[text() = ("OK", 9999, "...FAILED")])
XQuery Optimization

2. Analyze predicates: costs

Calculated by requesting the expected number of results from the index:

- sum of #results will be returned if multiple terms are requested:
  \[//\text{keyword}[\text{text()} = ("\text{find}", "\text{one}", "\text{of}", "\text{the}", "\text{terms}")]\]

- highest #results will be returned if multiple terms are intersected
  \[//\text{full/text[. contains text ("\text{Big} \text{ ftand \text{Bang}}")]}\]

- number of results will be estimated if real assessment is too expensive
  e.g.: fuzzy, wildcard, or range queries

- some indexes will be privileged if sequential costs are higher than others
  (as is the case, e.g., for full-text requests due to tokenization)
XQuery Optimization

3. Rewrite path

XPath axes embrace certain symmetries:
Olteanu et al. [2002]: XPath: Looking Forward

- \( \text{descendant-or-self::m[child::n]} \equiv \text{descendant::n/parent::m} \)
- \( \text{p[self::n]/parent::m} \equiv \text{p/self::n/parent::m} \)
- \( \text{self::m[child::n]} \equiv \text{child::n/parent::m} \)

Observations:

- some steps are wrapped into predicates
- last name test outside predicate will never change
XQuery Optimization

3. Rewrite path

Rewritings for text nodes:
- \(/\text{descendant}::m \ [\text{child}::\text{text}() = e] \equiv \text{Tl}(e) / \text{parent}::m\)
- \(/\text{descendant}::m \ [\text{descendant}::\text{text}() = e] \equiv \text{Tl}(e) / \text{ancestor}::m\)
- \(/\text{descendant}::m \ [\text{child}::n / \text{descendant}::\text{text}() = e] \equiv \text{Tl}(e) / \text{ancestor}::n / \text{parent}::m\)

Rewritings for attribute values:
- \(/\text{descendant}::m \ [\text{attribute}::n = e] \equiv \text{Ai}(e, n) / \text{parent}::m\)

document test is skipped, as all results have the same root node.
XQuery Optimization

3. Rewrite path

Rewritings of paths with child steps:

- \(/child::m[child::text() = e] \equiv TI(e)/parent::m[parent::document-node()]\)
- \(/child::m/descendant::n[child::text() = e] \equiv TI(e)/parent::n[ancestor::m/parent::document-node()]\)

A document test is needed, as results might be placed somewhere deeper in the document.
Query Plans: Example 1

Attribute test

\[
doc("xmark")/descendant::item[@id = "item0"]
\]
Query Plans

XMark query

Query:
1st $auction := doc("xmark") return
   for $b in $auction/site/people/person[@id = "person0"]
   return $b/name/text()

Compilation:
1. pre-evaluating doc("xmark")
2. binding static variable $auction
3. applying attribute index
4. removing variable $auction, simplifying flwor
Query Plans

XMark query

Query:

for $a$ in /descendant::closed_auction
    [price >= 500 and price <= 1000]
for $i$ in /descendant::item
for $c$ in /descendant::category
where $a$/itemref/item = $i$/@id
and $c$/@id = $i$/incategory/category
return $c$/@name
XQuery Optimization

Summary

- traditional compiler construction comes with a wide range of optimization techniques, many of which can be applied to XQuery
- additional tree- and index-based rewritings are needed to do full justice to the specifics of XML
- most optimizations reveal their full power in the interplay with others
- some queries can be fully evaluated in the compilation step

What’s next

Query evaluation! Cached vs. iterative processing, lazyness, ...