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:
  `for $i in //X 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 + (if($b) then 1 else "1")` ⇢ `1985` or ↯ (depends on `$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?

```
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

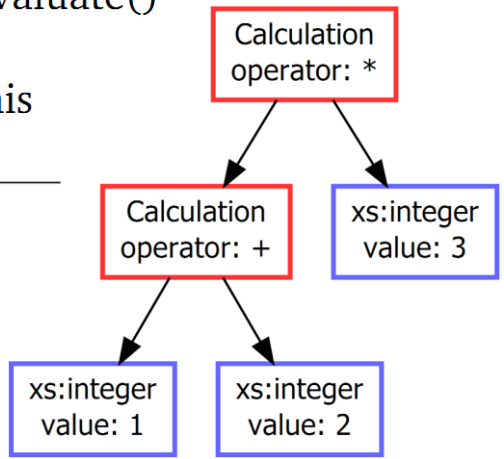
XQuery Optimization

Constant folding

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

Expression.Fold() : Expression

```
for a in ARGUMENTS do
    a = a.Fold()
end for
if all ARGUMENTS are values then
    return Evaluate()
else
    return this
end if
```



XQuery Optimization

Constant propagation

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

Enabling transformation

- single optimization triggers optimizations of parent expressions

Example

```
declare variable $π := 3.14159265
5 * $π
```

1. constant propagation:
   `$π` will be replaced with its value
2. constant folding:
   `5 * 3.1415…` will be pre-evaluated
3. remaining expression is a single
   xs:decimal value: `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 rewritings

- `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:

```
"↯" castable as xs:int
and xs:int("↯")
```

may, or may not, yield an error!

✎ What alternative expression can be used?

Equivalences

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

XQuery Optimization

General comparisons

- remember *existential semantics*!
  at least 1 comparison must be true:
  `() = ()` ⇢ `false`
  `(1,2) = (2,3)` ⇢ `true`
- can *only be negated* if operands will exactly yield one item:
  `not(1 != (1,2))` ≠ `1 = (1,2)`
- empty sequence always yields false:
  `() != $x` ≡ `false()`

Value comparisons

- defined for single items
- ☞ can always be negated (provided that input has correct type)
- simplify of boolean operation:
  `boolean($X) eq false()` ≡ `not($X)`
- simplify comparisons with empty sequence:
  `(1,2,3) eq ()` ≡ `()`

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 ≡ $x * $x
- multiplication ⇢ addition:
  $x * 2 ≡ $x + $x
- multiplication/division ⇢ bit shifting
  $x / 2^c ≡ $x >> c

✎ What limits exist when *shifting bits*?

General notes

- evaluation of duplicated operands may outweigh original costs
  $x * 12 ≡ $x + $x + $x + $x +
- many rewritings are not applicable to functional languages:

```
c = 0
for (i = 1 ... 100)
  c += 2
```

XQuery Optimization

Mathematical rewritings

Simplify algebraic expressions:

- remove neutral elements:
  `0 + $x - 0 ≡ $x`
  `1 * $x / 1 ≡ $x`
- remove absorbing elements:
  `$x * 0 ≡ 0`

☞ floating-point arithmetics may cause unexpected results:
`xs:float("NaN") * 0 ⇢ NaN`

- regroup terms:
  `($a*2) + 10 + ($a*3) + 5`
  `≡ ($a*2) + ($a*3) + 10 + 5`
- apply distributive law:
  `(3 * $a) - (2 * $a)`
  `≡ (3-2) * $a ≡ 1 * $a ≡ $a`
- merge identical expressions (*contrary* to strength reduction):
  `$a + $a + $a + $a ≡ 4 * $a`

XQuery Optimization

Function rewritings

count()

- pre-evaluate if result size is known at compile time:
  `count((1,"A"))` ≡ `2`
- use empty() and exists() to speed up evaluation:
  `count($x) = 0` ≡ `empty(X)`
  `count($x) > 0` ≡ `exists(X)`
  ✎ Why may 2nd version be faster?

boolean()

- get rid of redundant conversion:
  `if(boolean($x))` ≡ `if($x)`

not()

- simplify boolean tests:
  `not(not($x))` ≡ `boolean($x)`
  `not(empty($x))` ≡ `exists($x)`

empty()/exist()

- pre-evaluate if result size is known:
  `empty((1,"B"))` ≡ `false`

XQuery Optimization

Constant folding

- applied to built-in functions:
  `string-length('ABC')` ≡ `3`
  `substring('ABC',3)` ≡ `'C'`
  `contains('aa','a')` ≡ `true`
- utilize typing information:
  ```
  some $n in ('A','B')
  satisfies exists($n) ≡ true()
  ```
- utilize meta information:
  ```
  count(for $i in 1 to 3
        return $i)
  ```

Non-determinism

- constant folding is only applicable to deterministic functions!
  ```
  for $n in 1 to 3
  return math:random()
  ```
- evaluation order is important if functions have side effects:
  ```
  file:write("A", <a/>),
  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

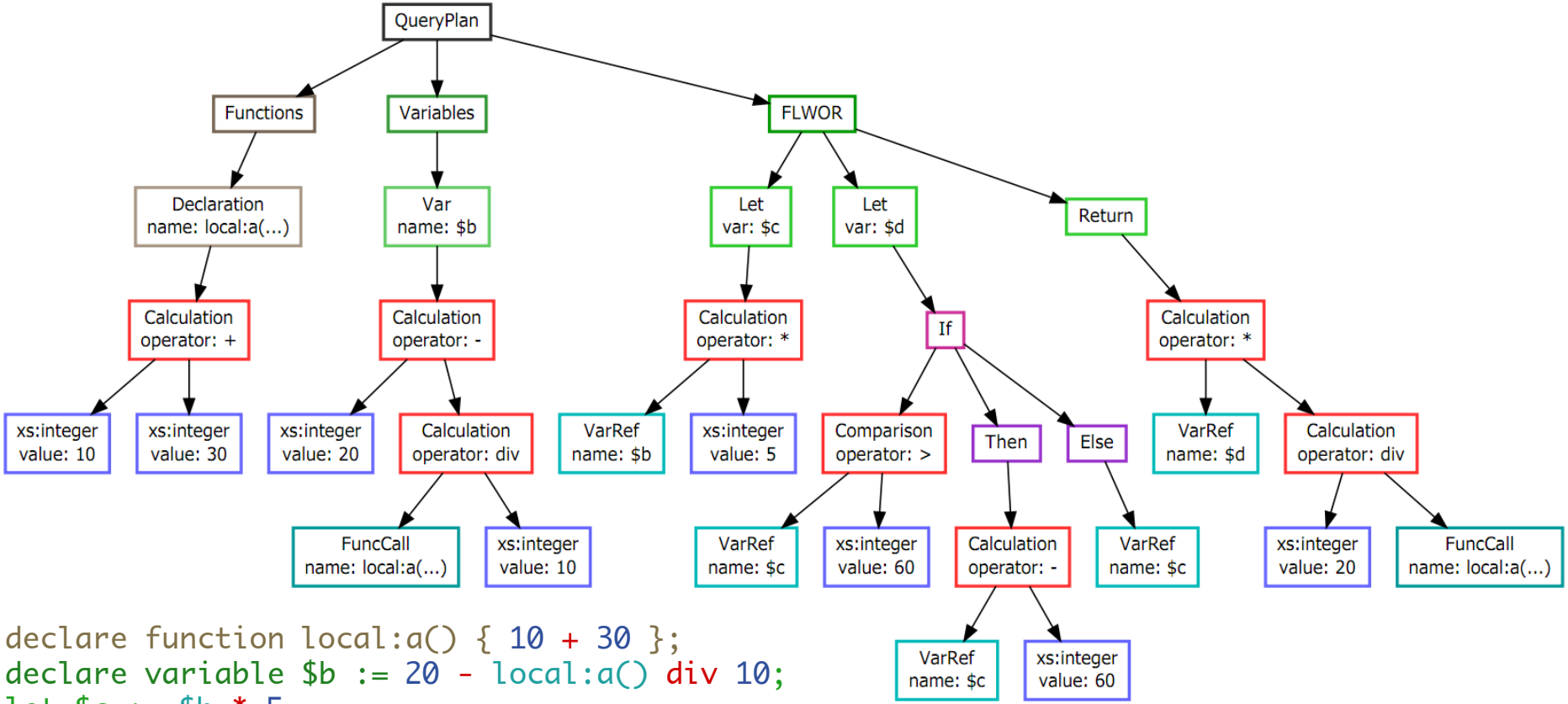
Merge identical expressions that would otherwise be evaluated multiple times:

```
for $a in 1 to 5
return ($a + 1) * ($a + 1)
```

Bind common expression to new variable:

```
for $a in 1 to 5
let $_ := ($a + 1)
return $_ * $_
```

✎ Why is CSE comparatively expensive?




```
declare function local:a() { 10 + 30 };
declare variable $b := 20 - local:a() div 10;
let $c := $b * 5
let $d := if($c > 60) then $c - 60 else $c
return $d * (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:
  `/books/book/ctx/title`
  ≡ `/books/book/()/title`
  ≡ `()`

☞ name index lookups are *local* operations (no path context needed)

Path summary rewritings

Contains information and statistics on *unique document paths*:

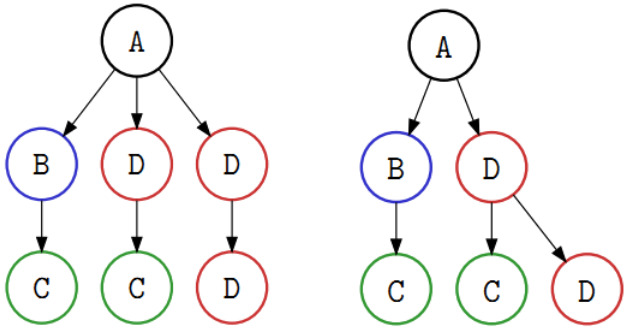
- count number of resulting nodes, e.g.: `count(/a/b)` ⇢ `123`
- skip paths that won't yield results:
  `/books/book/book` ≡ `()`
  `/books/text()/text()` ≡ `()`

✎ Which of the above queries can be optimized without index?

XQuery Optimization

Path summary rewritings

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



#	path	optimized
1	//B	/A/B
2	//C	/A/*/C
3	//D	/A/D | /A/D/D
4	//E	()
5	//A//C	/A/*/C

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

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

Optimized

```
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()

```
for i := 1 to #BINDINGS do
  inner := BINDINGS[i]
  if inner returns single, deterministic value then
    dest := null
    for o := i − 1 to 0 do
      outer := BINDINGS[o]
      break if outer declares variables used in inner
      dest := outer
    end for
    if dest is not null then
      move inner before dest
    end if
  end if
end for
```

XQuery Optimization

Predicate rewritings

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

Example

```
for $a in //code
where $a and $a = "JP"
return $a
```

Predicate rewriting

```
for $a in //code[.][. = "JP"]
return $a
```

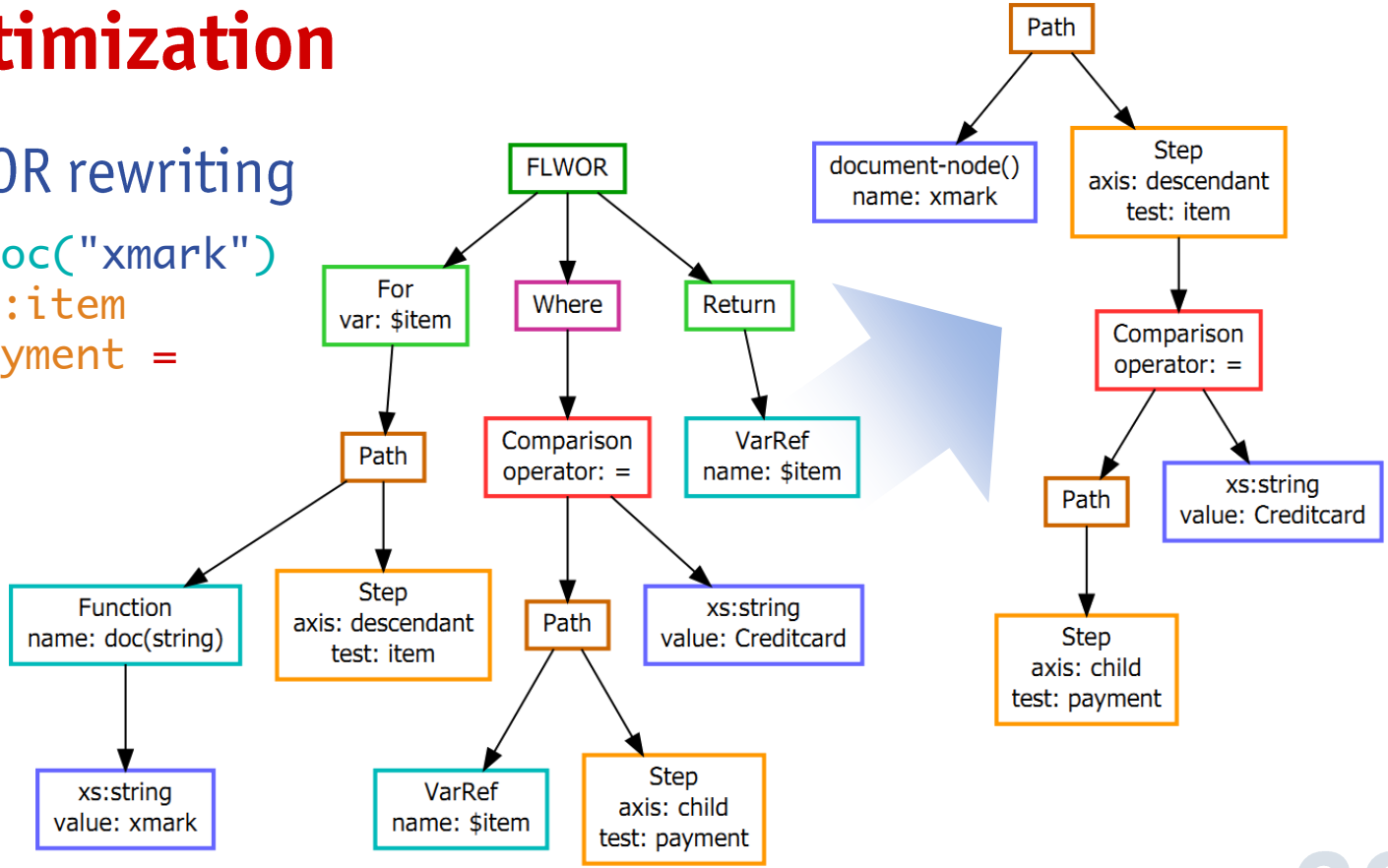
Simplification

```
//code[.][. = "JP"]
```

XQuery Optimization

Example: FLWOR rewriting

```
for $item in doc("xmark")
  /descendant::item
where $item/payment =
  "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*

Path.IndexContext(*data*: Data) : IndexContext

Require: STEPS = array with all path steps

```
context := null
for step in STEPS do
  continue if step is no AxisStep
  for pred in step.PREDICATES do
    c := new IndexContext(data, pred, step)
    continue if not pred.IndexAccessible(c)
    if context = null or c.costs < context.costs then
      context := c
      return context if context.costs = 0
    end if
  end for
end for
return context
```

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:
  `//keyword[text() = ("find","one","of","the","terms")]`
- highest #results will be returned if multiple terms are intersected
  `//full/text[. contains text ("Big" ftand "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

- descendant-or-self::m[child::n] ≡ descendant::n/parent::m
- p[self::n]/parent::m ≡ p/self::n/parent::m
- self::m[child::n] ≡ 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:

- /descendant::*m* [child::text() = *e*] ≡ TI(*e*)/parent::*m*
- /descendant::*m* [descendant::text() = *e*] ≡ TI(*e*)/ancestor::*m*
- /descendant::*m* [child::*n*/descendant::text() = *e*]
  ≡ TI(*e*)/ancestor::*n*/parent::*m*

Rewritings for attribute values:

- /descendant::*m* [attribute::*n* = *e*] ≡ AI(*e*, *n*)/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*] ≡ TI(*e*)/parent::*m*[parent::document-node()]
- /child::*m*/descendant::*n*[child::text() = *e*]
  ≡ TI(*e*)/parent::*n*[ancestor::*m*/parent::document-node()]

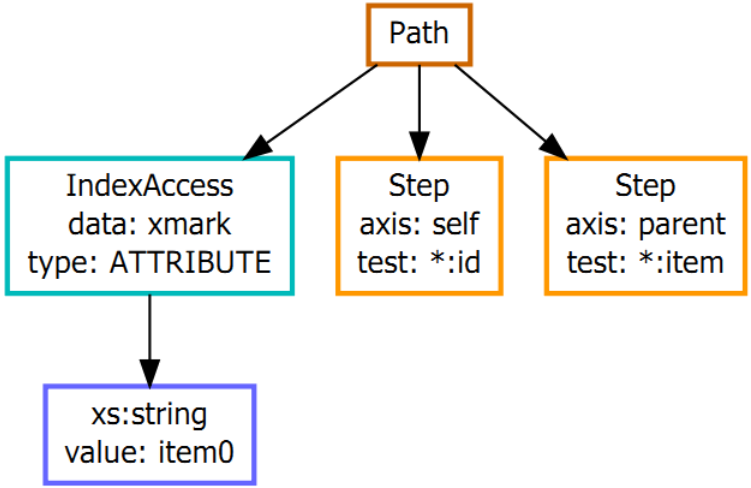
☞ document test is needed, as results might be placed somewhere deeper in the document.

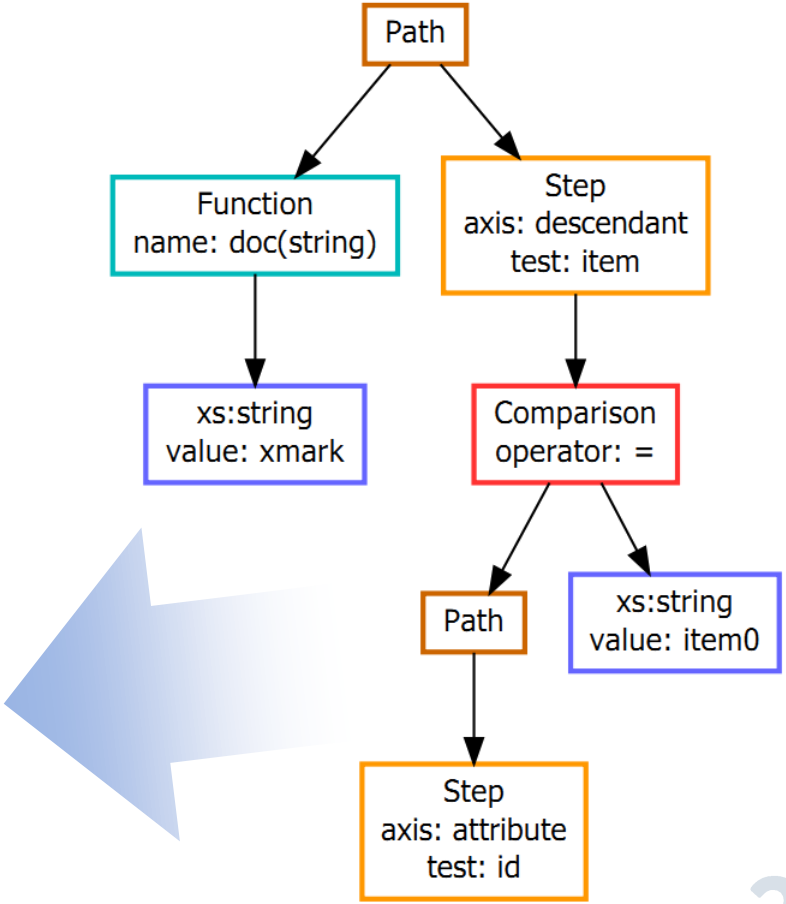
Document:	`<a>X<b>X</b></a>`
Query:	`/descendant::*[child::text() = 'X']`
Compiled:	`TI('X')/parent::*`
Result:	`<a>X<b>X</b></a>, <b>X</b>`
Query:	`/child::*[child::text() = 'X']`
Compiled:	`TI('X')/parent::*[parent::document-node()]`
Result:	`<a>X<b>X</b></a>`

Query Plans: Example 1

Attribute test

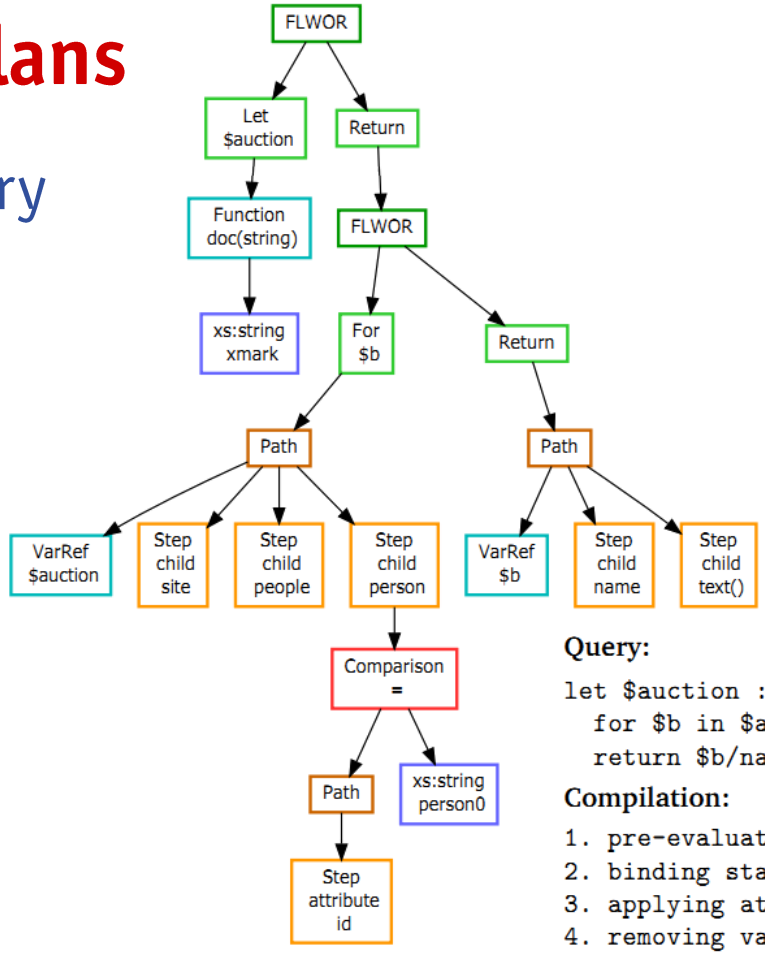
```
doc("xmark")/descendant::item
  [@id = "item0"]
```

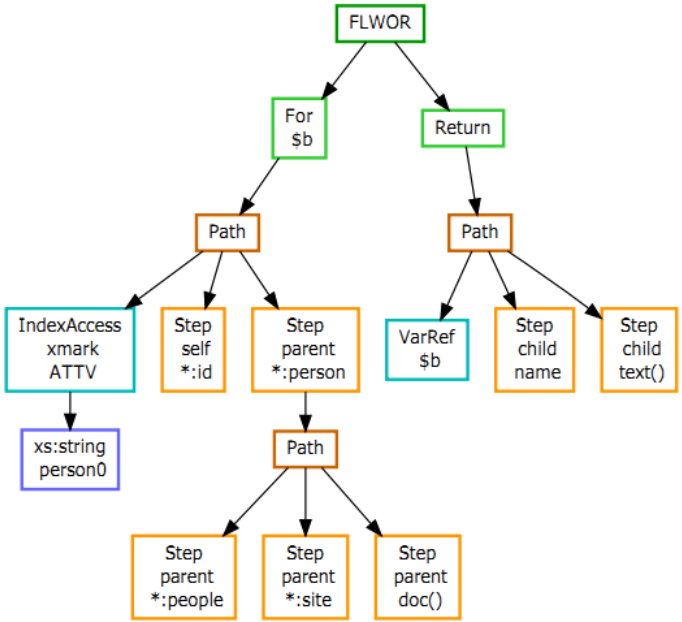




Query Plans

XMark query





Query:

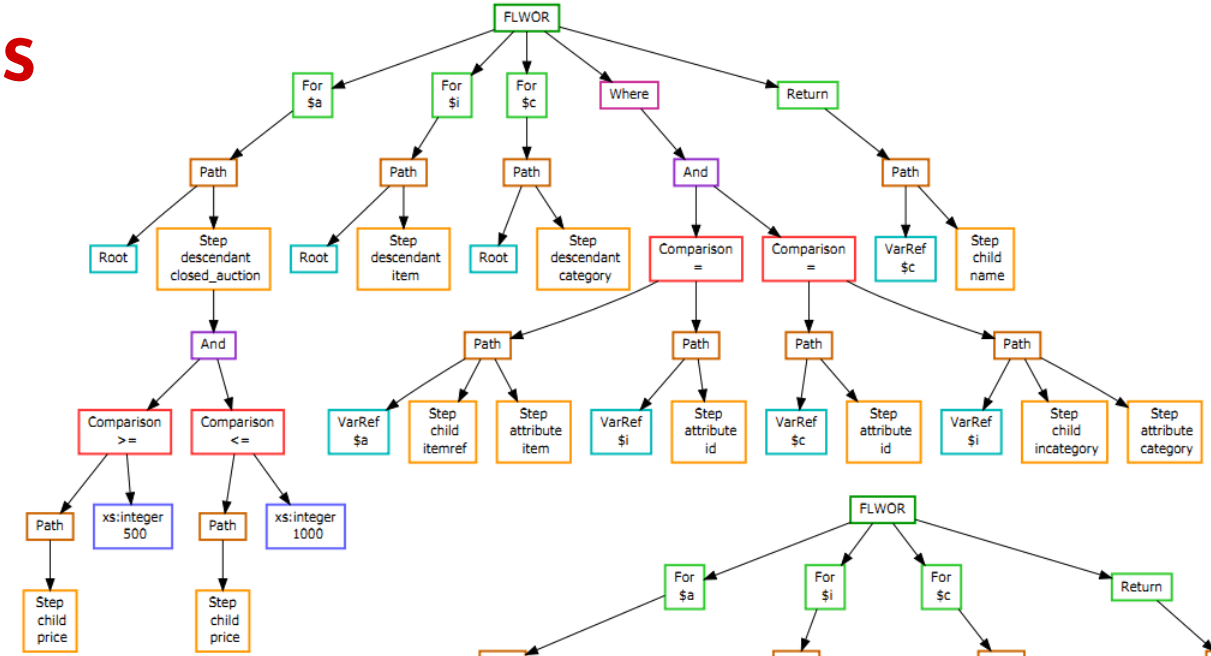
```
let $auction := doc("xmark") return
  for $b in $auction/site/people/person[@id = "person0"]
  return $b/name/text()
```

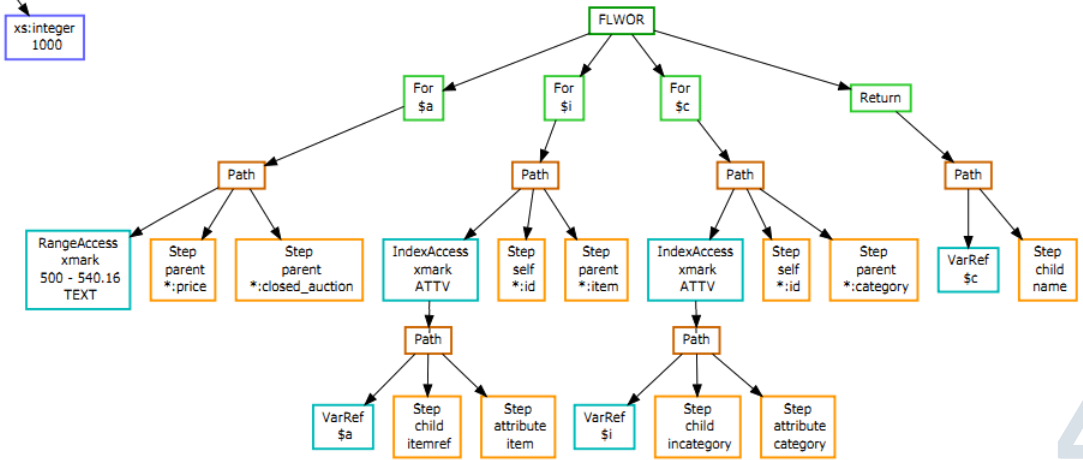
Compilation:

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, ...