XML Technologies

XQuery: The XML Query Language

Christian Grün
Database & Information Systems Group
Universität Konstanz

Introduction

What is XQuery 3.0?

- (*more than a*) query language for XML data
- (*more than a*) counterpart to SQL
- *functional* language
- *extensible* via user-defined functions and modules
- general purpose language?

☞ Information Processing Language

Extensions

- Full Text
- Update Facility
- Scripting Extension

Alternatives

- XSLT: comparable features, but specialized on transformations
- SQL 2005:
  aims to combine SQL and XQuery

Introduction

Functional Language

- prominent examples: Haskell, Scheme, OCAML, R
- based on the evaluation of *functions*: one input, one output
- no *side effects* (that would change the state of a program)
- *higher-order* functions: functions as arguments
- iteration ⇢ recursion

☞ XQuery 3.0 is a fully functional language

✎ How would a functional version of this code look like?

```
c=0; for(i=1; i<10; i++) { c+=i; }; print c
```

XQuery Data Model

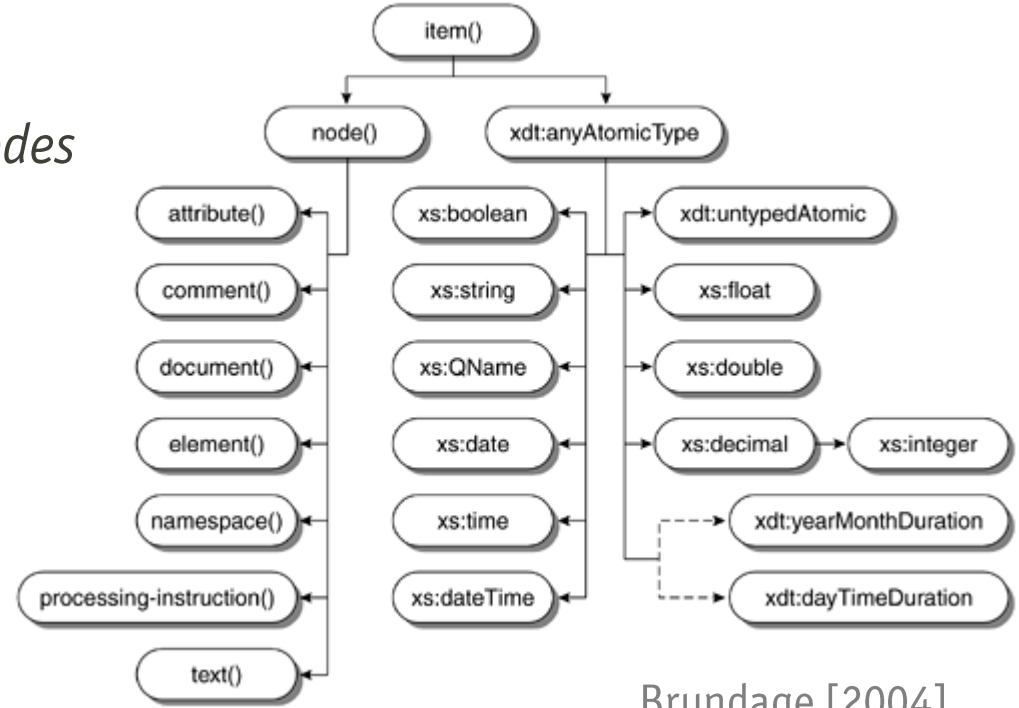
Sequences

- *values* are the result of evaluated expressions
- all values are *sequences*
- sequences are lists of zero or more *items*:
  `<H>U</H>, ("different", 2, <a/>), ()`
- sequences with zero items are called *empty sequences*
- sequences are *ordered*: `(1,2)` ≢ `(2,1)`
- they may contain *duplicates* (...in contrast to XPath results!)
- special feature: sequences will automatically be *flattened*:
  `(1,(2,3),(),4)` ⇢ `(1,2,3,4)`

XQuery Data Model

Data Types

- items are *atomic values* or *nodes*
- nodes are XML constructs: elements, attributes, ...
- atomic values are booleans, integers, strings, dates, ...



Brundage [2004]

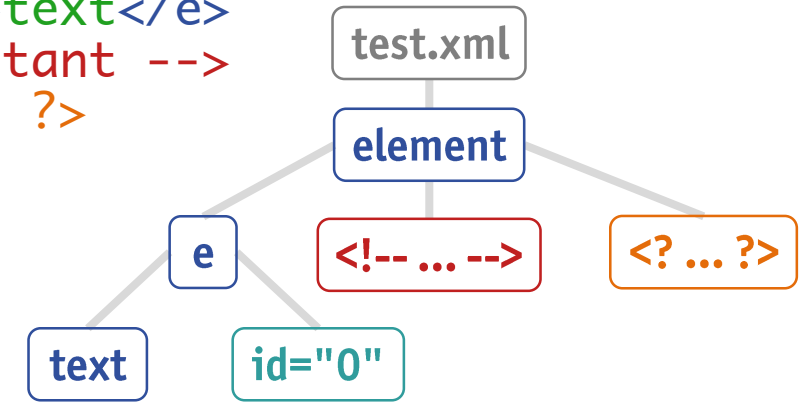
XQuery Data Model

XML Node Types

Already introduced with the XML data model:

- document nodes
- elements
- attributes
- texts
- comments
- processing instructions

```
<element>
  <e id="0">text</e>
  <!-- important -->
  <?php code ?>
</element>
```



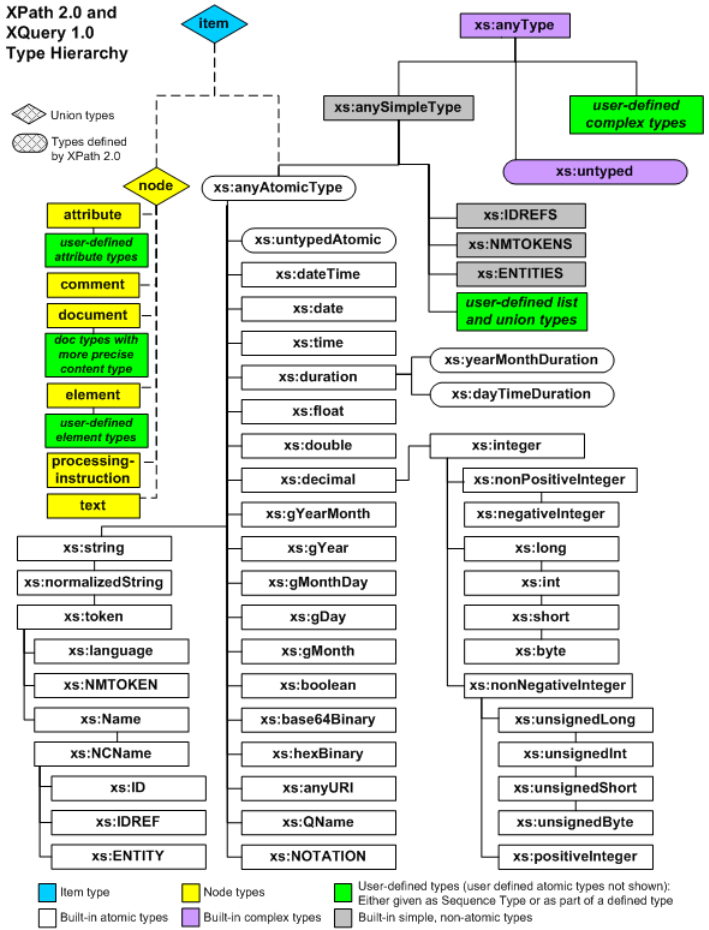
XQuery Data Model

Atomic Values

- the complete picture...
- usually, you will only work with the most common types:
  - `xs:integer`, `xs:decimal`, `xs:double`
  - `xs:string`
  - `xs:boolean`
  - `xs:date`, `xs:time`, `xs:duration`

✎ What is `xs:untypedAtomic`?

XQuery Data Model, 2010



XQuery Data Model

Untyped Data

- if no schema is available, XML nodes have no atomic type
- as a result, the atomized value will be untyped:
  `data(<xml>text</xml>) instance of xs:untypedAtomic` ⇢ `true`

Typed Data

- typed items:
  - `xs:decimal`: `19.84`
  - `xs:string`: `"!"`
  - `xs:date`: `xs:date("2000-01-01")`

XQuery Data Model

Creating Atomic Values

- using literals:
  `"string"`, `0.98`
- explicit type assignment (cast):
  - `xs:untypedAtomic("string")`
  - `xs:byte(127)`
  - `"2019-12-31T23:59:59" cast as xs:dateTime`
- resulting from a function call:
  - `true()` ⇢ `xs:boolean`
  - `max( (1,2,5,10,20,50,100) )` ⇢ `xs:integer`

Expressions

Primary Expressions

- literals, variables, function calls, parenthesized expressions

Arithmetics, Comparisons

- numeric and date comparisons
- comparisons based on values or document order

Logical Expressions

- Boolean operators

Conditional Expressions

- if/then/else, switch, typeswitch

Sequence Operators

- set-based operations

Constructors

- direct and computed node construction

FLWOR Expressions

- XQuery-specific loops

Primary Expressions

Literals (Constants)

- string: `"example"`
- integer: `1234`
- decimal (exact): `12.34`
- double (floating): `12.3e4`

Variables

- dollar syntax with optional prefixes or namespaces: `$item`, `$pf:var`, `$Q{www}count`

Function Calls

- function name, arguments:
  - `doc("input.xml")`
  - `local:function(99)`

Comments

- enclosed by

```
(: smileys
  (: can be nested :)
:)
```

Arithmetic Expressions

Addition, Subtraction

- numbers: `1 + 2 - 3`
- dates: `date("2001-01-01") + xs:duration("P9Y")`
- strings: `"and " || "some " || "more"`

Multiplication, (Integer) Division, Modulo

- `(1 + 2) * 3`
- `4 * 5 div 6`
- `7 idiv 8`
- `9 mod 10`

Comparison Expressions

Value Comparisons

- available operators:
  `eq`, `ne`, `lt`, `le`, `gt`, `ge`
- restricted to the comparison of *atomic items* (single values)
- the values of both operands must be of the *same type*
- if that's not the case, a *type error* will be raised
- if any of the operands yields an *empty sequence*, () is returned

What results do you expect?

- `3 gt 4`
- `"a" le "b"`
- `"123" eq 123`
- `<x>1</x> lt 2`
- `(1,2) gt (2,1)`
- `() eq ()`
- `() ne ()`

Comparison Expressions

General Comparisons

- available operators:
  `=, !=, <, <=, >, >=`
- *existential semantics*: compares all items of a sequence to each other
- returns true if *at least one* comparison yields `true`
- untyped values...
  - are *implicitly cast* to the type of the other item
  - if both items are untyped, their values are compared as *strings*

What results do you expect?

- `3 > 4`
- `"a" <= "b"`
- `"123" = 123`
- `<x>1</x> < 2`
- `(1,2) > (2,1)`
- `() = ()`
- `() != ()`

Comparison Expressions

Node Comparisons

- available operators: `is`, `<<`, `>>`
- compares *single nodes* based on their identity, i.e., their *document order* (✎ remember?)
- within in a query, documents that are accessed *multiple times* always have the same identity!

✎ What results do you expect?

- `doc("xml") is doc("xml")`
- `<x/> is <x/>`
- `doc("xml")/* << doc("xml")/*`
- `<x/> << <x/>`
- `doc("shop") >> doc("persons")`
- `<x/> >> doc("persons")`

- order between *different documents* and *constructed nodes* is implementation-defined (but must be unique)

Conditional Expressions

If/then/else Expression

- syntax:
  `if(`*test*`)`
  `then` *expr1* `else` *expr2*
- nested syntax:
  `if(`*test1*`)`
  `then` *expr1*
  `else if(`*test2*`)`
  `then` *expr1*
  `else` *expr2*

Effective Boolean Value

Retrieved via `boolean()` (remember?):

- string ⇢ string length > `0` ?
- number ⇢ different to `0` or `NaN` ?
- boolean ⇢ adopted as is
- empty sequence ⇢ `false`
- non-empty node sequences ⇢ `true`
- other items ⇢ ↯
- sequences with more than one item ⇢ ↯

Conditional Expressions

Switch Expression

- tests the *value* of an expression
- syntax:

```
switch(test)
  case case1
  case case2
    return expr
  ...
  default
    return defaultExpr
```

Typeswitch Expression

- tests the *type* of an expression
- syntax:

```
typeswitch(test)
  case type
    return expr1
  ...
  default
    return defaultExpr
```

Sequence Types

Type Cast

- convert types to another type:
  `"1" cast as xs:integer` ≡ `xs:integer("1")` ⇢ `1`
  `"0815" cast as xs:byte` ⇢ ↯

Cast Check

- check if a cast is possible:
  `"0815" castable as xs:byte` ⇢ `false`

Instance Check

- check if an input belong to a given type:
  `(1,2,3) instance of xs:integer+` ⇢ `true`

Logical Expressions

And/or Expressions

- Syntax:
  expr1 `and` *expr2* ⇢ xs:boolean
  expr1 `or` *expr2* ⇢ xs:boolean
- computes the *effective boolean value* for the operands
- if possible, an implementation will *skip remaining tests* that will not change the final result

Functions

- `true()` / `false()` creates a boolean value
- `boolean(`*expr*`)` computes the effective boolean value of *expr*
- `not(`*expr*`)` inverses a boolean value
- `empty(`*expr*`)` / `exists(`*expr*`)` checks if a sequence contains items

✎ How many boolean values have to be computed for this expression?

```
'' or 'a' or 'b' or 0 or 1 or 2
```

Sequence Expressions

Quantifiers

- Syntax:
  `some` *var* `in` *expr*
  `satisfies` *test*
  `every` *var* `in` *expr*
  `satisfies` *test*

Example

```
some $v in (0, 2, 4, 5)
satisfies $v mod 2 = 1
```

Combining Sequences

- Syntax (remember?):
  seq1 `union` *seq2*
  seq1 `intersect` *seq2*
  seq1 `except` *seq2*

Range Expression

- Syntax: *min* `to` *max*
- Examples:
  - `1 to 10`
  - `reverse(80 to 90)`

Filters and Predicates

Predicates [...]

- used to *filter* expressions
- evaluated as *effective boolean values*
- applied from *left to right*
- order matters for *positional predicates*

Context Item .

- references currently processed item (one by one)

What results do you expect?

- `<A>X</A>[. = "X"]`
- `(1, 2, 3)[. mod 2 = 1]`
- `(1, 2, 3)[.*. > 8][.+. < 8]`
- `(1 to 3)[position() = 1 to 2]`
- `(1 to 3)[1 to 2]`
- `(1 to 3)[.]`
- `(0 to 2)[.]`
- `(0 to 2)[. + 1]`
- `(0 to 2)[boolean(.)]`
- `count( ("A","","B")[.] )`

FLWOR Expressions

- clauses (XQuery 1.0): `for`, `let`, `where`, `order by`, `return`
- step-by-step evaluation of sequences
- iterative binding of variables

Examples

```
for $n in reverse(1 to 4)
let $p := $n * $n
where $p > 8
return $p
```

```
for $n in 1 to 4
where $n * $n > 8
order by $n descending
return $n * $n
```

✎ What is the output of the two expressions?

FLWOR Expressions

Nested FLWOR Expressions

```
for $a in 1 to 3
for $b in 1 to 2
where $a * $b > 1
return
  for $c in 1
  let $d := ($a, $b, $c)
  return <X>{ $d }</X>
```

☞ TIMTOWTDI

Larry Wall, creator of Perl

Equivalent Example

```
for $c in 1
for $b in 1 to 2
for $a in 1 to 3
where $a * $b > 3
order by $a, $b, $c
return <X>{ $a, $b, $c }</X>
```

Result

```
<X>2 2 1</X>
<X>3 2 1</X>
```

FLWOR Expressions

XQuery vs XPath

XQuery
```
for $movie in doc("library.xml")//movie
return $movie
```
XPath
```
doc("library.xml")//movie
```

XQuery
```
for $movie in doc("library.xml")//movie
let $title := $movie/title
where $title = "Juha"
return $movie
```
XPath
```
doc("library.xml")//movie[title = "Juha"]
```

FLWOR Expressions

XQuery vs XPath

XQuery

```
for $movie in doc("library.xml")//movie
let $title := $movie/title
where starts-with($title, "La ")
order by $title
return <hit id="{ $movie/@id }">{ $title }</hit>
```

XPath ↯

```
doc("library.xml")//movie[starts-with(title, "La ")]
```

No XPath equivalents for:

- iterative binding of variables, order results
- multiple contexts, node constructions, ...

Paths, Map Operator

Location Paths

- already known from XPath
- in XQuery, path may contain of arbitrary expressions:
  `give / <xml>A</xml> / text() / string()`

Map Operator

- generalized expression for performing operations step by step
- items of the left-hand expression is bound to the *context item* one by one
- results will be not be *ordered*, *duplicates* are preserved

✎ What's the result?

```
('a','abc','ab') ! string-length(.) ! (. > 2)
```

Variable Assignments

Global Variables

- specified in the query header: `declare variable $v := expr;`
- may also be declared as `external`

Local Variables

- specified in FLWOR, quantifier, typeswitch and various other expressions

Shadowing

- all variables in XQuery are *final*!
- variables with the same name will be *shadowed*

What is the result of this query?

```
declare variable $a := 1;
let $a := 2
return (
 (let $a := $a * $a
  return $a), $a), $a
```

Constructors

Direct Constructors

- new XML fragments can be constructed on-the-fly
- simple XML snippets are valid XQuery expressions

Example

- creation of an element, attribute, and text node:
  `<node id="0">Direct Constructor</node>`

Restrictions

- no DTD handling or declaration of entities
- some characters within values have a special meaning, because...

Constructors

Embedding XQuery Expressions

- arbitrary XQuery expressions can be embedded in curly brackets

Example

```
<result>{
  for $n in 1 to 10
  return <number
    id = "{ $n }"
    roman = "{ format-integer($n, "I") }">{
    format-integer($n, "w")
  }</number>
}</result>
```

Constructors

Computed Constructors

- syntax is more verbose and less XML-like
- provides more flexibility, as new names can be dynamically generated

Example

- create elements with dynamically generated names:

```
for $i in 1 to 5
return element { "node" || $i } {
  attribute { "id" } { $i },
  text { "Text " || format-integer($i, "I") }
}
```

Functions

Function Libraries

- the XQuery core is extended by more than 150 functions:
  http://www.w3.org/TR/xpath-functions-31
- default functions use (optional) `fn` prefix:
  `fn:true()` ≡ `true()`
- additional function prefixes (XQuery 3.1): `math`, `map`, `array`
- BaseX 8.x offers more than 250 additional functions
  (some of them standardized in the EXPath/EXQuery projects):
  http://docs.basex.org/wiki/Module_Library
- Java™ bindings can be used to access arbitrary Java functions
  http://docs.basex.org/wiki/Java_Bindings

Input Functions

- Open document node
  `doc("doc.xml")`
- Open collection (sequence of document nodes)
  `collection("docs")`
- Open documents from database
  `db:open("database")`
- Convert XML string to node
  `parse-xml("<xml/>")`
- Parse text file
  `file:read-text("file.txt")`
- List entries of a ZIP file
  ```
  archive:entries(
    file:read-binary("a.zip"))
  ```
- Execute SQL query
  ```
  let $url := 'jdbc:... '
  let $id := sql:connect($url)
  return sql:execute($id,
    "SELECT ...")
  ```

Output Functions

- Serialize XML nodes to a string
  ```
  serialize(//product)
  ```
- Write to file
  ```
  file:write(
    "doc.xml", <x>A</x>)
  file:write-binary(
    "file.bin",
    xs:hexBinary("414243"))
  ```
- Raise error
  ```
  if($x >= 0) then $x else
  error((), "$x is negative")
  ```

Debugging information will be printed to STDERR or the BaseX Info View:

- Trace and pass on value:
  ```
  for $i in 1 to 10
  return trace($i, "$i: ")
  ```
- Dump (and swallow) value:
  ```
  for $i in 1 to 10
  return (prof:dump($i+1),$i)
  ```
- Measure performance
  ```
  prof:time(1 to 100000),
  prof:mem(1 to 1000)
  ```

String Functions

- Find substrings
  `contains("Chekhov", "hek")`
- Return string length
  `string-length("大江 健三郎")`
- Change case
  `lower-case("ДОСТОЕВСКИЙ")`
  `upper-case("albert camus")`
- Concatenate strings
  `concat("Nie","tz","sche")`
  ≡ `"Nie" || "tz" || "sche"`
- Regex match
  `matches("Kafka","Kaf.*")`
- Replace with regex
  `replace("AxB","[a-z]","")`
- Tokenize via regex
  `tokenize("a b c","\s")`
- Join strings
  `string-join(`
  `("a","b","c"),"  ")`

Sequence Functions

- Remove duplicate values
  `distinct-values(//year)`
- Find index positions
  `index-of(("o","X","o"),"X")`
- Remove items
  `remove(("o","X","o"), 2)`
- Return subsequence
  `subsequence((4,5,6), 2, 1)`
- Reverse sequence
  `reverse(1 to 100)`
- Return first item
  `head(1 to 100)`
- Second and following items
  `tail(1 to 100)`
- Compares items in depth
  `deep-equal($node1, $node2)`

☞ In XQuery, numbering always starts from 1!

Numeric Functions

- Count number of items
  `count(//title)`
- Minimum/maximum values
  `min(//title/year), max(...)`
- Calculate sum and average
  ```
  sum(for $i in 1 to 10
      return $i * $i)
  avg(//person/@age)
  ```
- Absolute and rounded number
  `abs(-1), round(2.345, 2)`
- Square root
  `math:sqrt(100)`
- Number π
  `math:pi()`
- Random numbers
  `random:double() * 5`

✎ What is the difference between the following queries:

- `min(1,2)` vs. `min((1,2))`
- `min()` vs. `min(())`

Functions

User-defined Functions

- existing functions can be extended by declaring your own functions
- makes the code better readable and reusable
- by default, the `local` prefix is reserved for user-defined functions

Example

```
declare function local:double($d as xs:double) {
  $d * 2
};
local:double(2.0)
```

Library Modules

- the executed query is called *main module*
- library modules are used to encapsulate functionality

☞ fundamental requirement in complex projects

File "op.xqm"

```
module namespace op = 'xq-operators';
declare function op:add($x, $y) { $x + $y };
```

File "test.xq"

```
import module namespace op = 'xq-operators' at 'op.xqm';
op:add(123, 456)
```

- depending on implementation, location may be omitted

Conclusion

- today, you have heard about the *nuts & bolts* of XQuery
- in the beginning, most developers used XQuery for simple XML queries
- today, it is *increasingly* used for writing *complex applications*
- we spend a lot time on *syntactical* and *semantical details*, but it gets more interesting once you use the language!

Impressive use cases

- Creating raytracer images: `http://files.basex.org/modules/raytracer.zip`
- XQuery Parser: the language itself can be used to parse arbitrary XQuery code (the parser module in question is appr. 700KB large...): `http://www.bottlecaps.de/rex/`