Chapter 10 – XQuery

Prof. Dr.-Ing. Stefan Deßloch
AG Heterogene Informationssysteme
Geb. 36, Raum 329
Tel. 0631/205 3275
dessloch@informatik.uni-kl.de

Inhalt

Why do we need a new query language?

- Relational Data, SQL
  - flat (rows and columns), use foreign keys, structured types for hierarchical data
  - data is uniform, repetitive
    - info schema for meta data
  - uniform query results
  - rows in a table are unordered
  - data is usually dense
    - NULL for missing/inapplicable data
- XML
  - nested, need to search for something at an arbitrary level (//*[@color = "Red"])
  - data is highly variable, self-describing
    - meta data distributed throughout doc
    - queries may need to access data and meta data: "tag name equals content" //*[name(.) = string(.)]
  - heterogenous query results
    - severe structural transformations required
      - invert a hierarchy
  - elements in document are ordered
    - needs to be preserved
    - query based on order, position
    - output order specification at multiple levels in the hierarchy
  - data can be sparse
    - empty or absent elements

XQuery: Language Requirements

- XML elements have attributes as well as content
- Content model of an XML element-type is much more flexible than in the relational setting (as described by an XML Schema)
  - Choices between alternative contents, variable numbers of repetitions
  - Mixed content (subelements mixed with text)

➔ An XML query language must be able to:

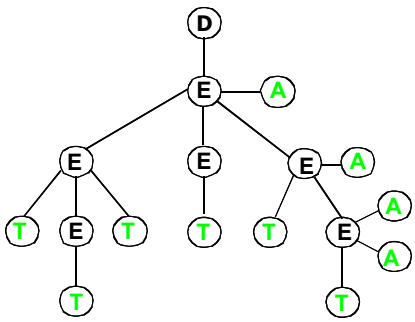
- Query deeply nested and heterogeneous structures
- Query metadata as well as user data
- Search for objects by absolute and relative order
- Preserve order of objects in input documents
- Impose new ordering at multiple levels of output
- Handle missing data and sparse data
- Preserve or transform the structure of a document
- Exploit references to unknown or heterogeneous types
- Easily define recursive functions
- Provide a very flexible data definition facility

XQuery

- XQuery is a general purpose query language for XML data
- Currently being standardized by the World Wide Web Consortium (W3C)
- XQuery is derived from
  - the **Quilt** ("Quilt" refers both to the origin of the language and to its use in "knitting " together heterogeneous data sources) query language, which itself borrows from
  - **XPath**: a concise language for navigating in trees
  - **XML-QL**: a powerful language for generating new structures
  - **SQL**: a database language based on a series of keyword-clauses: SELECT - FROM – WHERE
  - **OQL**: a functional language in which many kinds of expressions can be nested with full generality

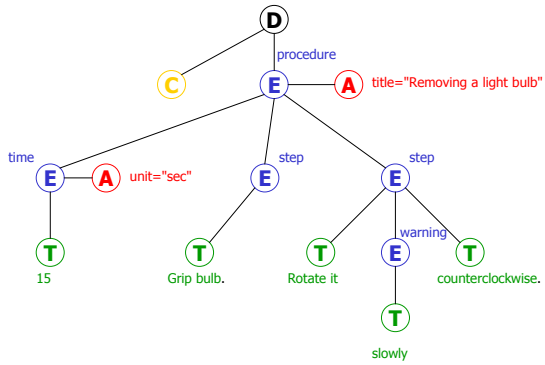
Tree Model of XML Data

- Query and transformation languages are based on a **tree model** of XML data
- An XML document is modeled as a tree, with **nodes** corresponding to elements and attributes
- Several types of nodes:
  - Document, Element, Attribute, Text, Namespace, Comment, Processing Instruction



Example

```
<?xml version = "1.0"?>
<!-- Requires one trained person -->
<procedure title = "Removing a light bulb">
  <time unit = "sec">15</time>
  <step>Grip bulb.</step>
  <step>
     Rotate it
     <warning>slowly</warning>
     counterclockwise.
  </step>
</procedure>
```



XQuery Data Model

- Builds on a tree-based model, but extends it to support sequences of items
  - A value is an ordered **sequence** of zero or more **items**
    - cannot be nested – all operations on sequences automatically "flatten" sequences
      - no distinction between an item and a sequence of length 1
    - can contain heterogenous values, are ordered, can be empty
    - may contain duplicate nodes (see below)
  - An **item** is a **node** or an **atomic value**
  - **Atomic values** are typed values
    - XML Schema simple types
  - There are seven kinds of **nodes** (see tree-based model)
    - nodes have an identity
    - each node has a typed value
      - sequence of atomic values
      - type may be unknown (anySimpleType)
    - element and attribute nodes have a type annotation
      - generated by validating the node
    - document order of nodes
- Closure property
  - XQuery expressions operate on/produce instances of the XQuery Data Model

General XQuery Rules

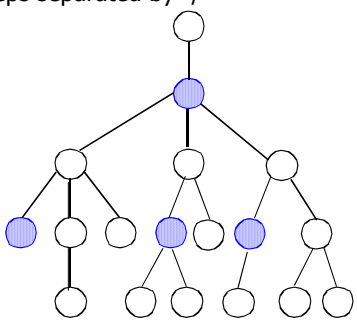
- XQuery is a case-sensitive language
- Keywords are in lower-case
- Every expression has a value and no side effects
- Expressions are fully composable
- Expressions can raise errors
- Expressions (usually) propagate lower-level errors
  - Exception: if-then-else
- Comments look like this
  - (: This is an XQuery comment :)

XQuery Expressions

- Literals: "Hello" 47 4.7 4.7E-2
- Constructed values: true() false() date("2002-03-15")
- Variables: $x
- Constructed sequences
  - $a, $b is the same as ($a, $b)
  - (1, (2, 3), (), (4)) is the same as 1, 2, 3, 4
  - 5 to 8 is the same as 5, 6, 7, 8

Path Expressions

- Inherited from XPath 1.0
  - is used to address (select) parts of documents using path expressions
- An XPath expression maps a node (the context node) into a set of nodes
  - always returns a set of distinct nodes
- A path expression consists of one or more steps separated by "/"
- Result of path expression: set of values that along with their containing elements/attributes match the specified path
  - E.g.: /bank-2/customer/customer-name evaluated on the bank-2 data returns
    - <customer-name> Joe </ customer-name>
    - < customer- name> Mary </ customer-name>
  - E.g.:/bank-2/customer/cust-name/text( ) returns the same names, but without the enclosing tags



Path Expressions (cont.)

- The initial "/" denotes root of the document (above the top-level tag)
- In general, a step has three parts:
  - The ***axis*** (direction of movement: child, descendant, parent, ancestor, following, preceding, attribute, … - 13 axes in all -)
  - A ***node test*** (type and/or name of qualifying nodes)
  - Some ***predicates*** (refine the set of qualifying nodes)
- Path expressions are evaluated left to right
  - Each step operates on the set of instances produced by the previous step
- Selection predicates may follow any step in a path, in [ ]
  - E.g. /bank-2/account[balance > 400]
    - returns account elements with a balance value greater than 400
    - /bank-2/account[balance] returns account elements containing a balance subelement
- Attributes are accessed using "@"
  - E.g. /bank-2/account[balance > 400]/@account-number
    - returns the account numbers of those accounts with balance > 400
  - IDREF attributes are not dereferenced automatically (more on this later)

XPath Axes Supported in XQuery

- Supported:
  - child
  - descendant
  - attribute
  - self
  - descendant-or-self
  - parent
- Optionally supported (full axis feature):
  - ancestor
  - ancestor-or-self
  - preceding
  - preceding-sibling
  - following
  - following-sibling
  - namespace

Path Expressions (cont.)

- The following examples use the abbreviated notation:
  - Find the first item of every list that is under the context node
    .//list/item[1]
  - Find the "lang" attribute of the parent of the context node
    ../@lang
  - Find the last paragraph-child of the context node
    para[last()]
  - Find all warning elements that are inside instruction elements
    //instruction//warning
  - Find all elements that have an ID attribute
    //*[@ID]
  - Find names of customers who have an order with today's date
    //customer [order/date = today ( )] / name

path expressions use a notation similar to paths in a file system:

/	means "child" or "root"
//	means "descendant"
.	means "self"
..	means "parent"
*	means "any"
@	means "attribute"

Node Tests

- Name test
  - **Element, attribute name**
    - **child::name, name** – Matches <name> element nodes
    - **child::*, *** - Matches any element node
    - **attribute::name, attribute::*, @*** for matching based on attribute name
  - **namespace:name** – Matches <name> element nodes in the specified namespace
  - **namespace:*** - Matches any element node in the specified namespace
    - **child::bank:*** - Matches any element node whose name is defined in bank namespace
- Node Type test
  - **comment()** – Matches comment nodes
  - **text()** – Matches text nodes
  - **processing-instruction()** – Matches processing instructions
  - **processing-instruction('target')** – Matches processing instructions with the specified target (<?target ...?>)
  - **element(), element(name), element(name, type)** – Matches element nodes
  - **node()** – Matches any node

Predicates

- Boolean expressions:
  book[author = "Mark Twain"]
- Numeric expressions:
  chapter[2]
- Existence tests:
  book[appendix]
  person[@married] (Tests existence, not value!)
- Predicates can be used in path expressions:
  //book[author = "Mark Twain"]/chapter[2]
  ...and in other kinds of expressions:
  (1to 100)[. mod 5 = 0]

XPath (cont.)

- XPath provides several functions
  - The function count() at the end of a path counts the number of elements in the set generated by the path
    - E.g. /bank-2/account[customer/count() > 2]
      Returns accounts with > 2 customers
  - Also function for testing position (1, 2, ..) of node w.r.t. siblings
- Boolean connectives and and or and function not() can be used in predicates
- IDREFs can be referenced using function id()
  - id() can also be applied to sets of references such as IDREFS and even to strings containing multiple references separated by blanks
  - E.g. /bank-2/account/id(@owners)
    returns all customers referred to from the owners attribute of account elements

Path Expressions and Functions

- Path expressions can be used in various places, e.g.:
  - In the For clause to bind variables
  - In the Let clause to bind variables to results of path expressions
- $c/text() gives text content of an element without subelements/tags
- XQuery path expressions support the "=>" operator for dereferencing IDREFs
  - Equivalent to the id( ) function of XPath, but simpler to use
  - Can be applied to a set of IDREFs to get a set of results
    - E.g.: "List hobbies of Denver employees"
      /emp[location = "Denver"]/@hobbies => *
      Results may include <skiing>, <hiking>, etc.
- The function **document(name)** returns root of named document
  - E.g. document("bank-2.xml")/bank-2/account

Expressions (cont.)

- Combining sequences: union intersect except
  - return sequences of distinct nodes in document order
- Arithmetic operators: + - * div idiv mod
  - Extract typed value from node
  - Multiple values => error
  - If operand is ( ), return ( )
  - Supported for numeric and date/time types
- Comparison operators
  - eq ne gt ge lt le compare single atomic values
  - = != > >= < <= implied existential semantics
  - is is not compare two nodes based on identity
  - << >> compare two nodes based on document order

Logical Expressions

- Operators: and or
- Function: not( )
- Return TRUE or FALSE (2-valued logic)
- "Early-out" semantics (need not evaluate both operands)
- Result depends on Effective Boolean Value of operands
  - If operand is of type boolean, it serves as its own EBV
  - If operand is ( ), zero, or empty string, EBV is FALSE
  - In any other case, EBV is TRUE
- Note that EBV of a node is TRUE, regardless of its content (even if the content is FALSE)!

Constructors

- To construct an element with a known name and content, use XML-like syntax:

```
<book isbn = "12345">
    <title>Huckleberry Finn</title>
</book>
```

- If the content of an element or attribute must be computed, use a nested expression enclosed in { }

```
<book isbn = "{$x}">
    {$b/title }
</book>
```

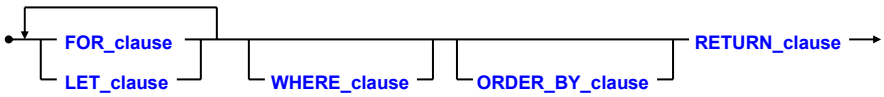
- If both the name and the content must be computed, use a computed constructor:

```
element {name-expr} {content-expr}
attribute {name-expr} {content-expr}
```


Validation of Constructed Elements

- An element constructor automatically validates the new element against "in-scope schema definitions"
  - Results in a **type annotation**
  - Can be generic: xs:anyType
- Validation mode (default = lax)
  - **Strict**: element must be defined in schema
  - **Lax**: element must match schema definition if it exists
  - **Skip**: ignore this element
  - Mode is set in Prolog or by explicit Validate expression
- Validation context:
  - Schema path inside which current node is validated
  - Each constructed element adds its name to the context
  - Can be overridden by an explicit Validate expression

XQuery: The General Syntax Expression FLWOR



- FOR clause, LET clause generate list of tuples of bound variables (order preserving) by
  - iterating over a set of nodes (possibly specified by an XPath expression), or
  - binding a variable to the result of an expression
- WHERE clause applies a predicate to filter the tuples produced by FOR/LET
- ORDER BY clause imposes order on the surviving tuples
- RETURN clause is executed for each surviving tuple, generates ordered list of outputs
- Associations to SQL query expressions
  for ⇔ SQL from
  where ⇔ SQL where
  order by ⇔ SQL order by
  return ⇔ SQL select
  let allows temporary variables, and has no equivalent in SQL

FLWOR - Examples

- Simple FLWR expression in XQuery
  - Find all accounts with balance > 400, with each result enclosed in an <account-number> .. </account-number> tag

```
for    $x in /bank-2/account
let    $acctno := $x/@account-number
where $x/balance > 400
return <account-number> {$acctno} </account-number>
```

- Let and Where clause not really needed in this query, and selection can be done in XPath.
  - Query can be written as:

```
for     $x in /bank-2/account[balance>400]
return <account-number> {$x/@account-number}
                                    </account-number>
```

Eliminating Duplicates

- Equality of elements
  - element name, attributes, content are identical
  - example: average price of books per publisher

```
FOR $p IN distinct-values(document("bib.xml")//publisher)
LET $a := avg(document("bib.xml")//book[publisher = $p]/price)
RETURN
        <publisher>
                <name> {$p/text()} </name>
                <avgprice> {$a} </avgprice>
        </publisher>
```


Nesting of Expressions

- Here: nesting inside the return clause
  - Example: inversion of a hierarchy

```
<book>
   <title>
   <author>
   <author>
</book>
<book>
   <title>
   <author>
   <author>
</book>
```

```
FOR $a IN distinct-values(//author)
RETURN
    <author>
        <name> { $a/text() } </name>
        { FOR $b IN //book[author = $a]
            RETURN $b/title }
    </author>
SORTBY(name)
```

```
<author>
   <name>
   <title>
   <title>
</author>
<author>
   <name>
   <title>
   <title>
</author>
```

Sorting of Results

- ORDER BY
  - Example: Sort the expensive books by first author name, book title

```
LET $b = document("bib.xml")//book[price > 100]
ORDER BY $b/author[1], $b/title
RETURN <expensive_books> $b </expensive_books>
```

- Ordering at various levels of nesting
  - Example: For all publishers, sorted by publisher name, list the title and price of all their books, sorted by price descending

```
<publisher_list>
{FOR $p IN distinct-values(document("bib.xml")//publisher)
  ORDER BY $p/name
  RETURN
        <publisher>
              <name> {$p/text()} </name>
              {FOR $b IN document("bib.xml")//book[publisher = $p]
              ORDER BY $b/price DESCENDING
              RETURN
                    <book>
                     {$b/title}
                     {$b/price}
                    </book>
              }
        </publisher>
}
</publisher_list>
```


Order Insignificance

- Indicate that the document order is insignificant
  - provides an opportunity for the optimizer
- Example:

```
fn:unordered(
    FOR $b IN document("bib.xml")//book,
              $a IN document("authors.xml")//author
    WHERE $b/author_id = $a/id
    RETURN
        <ps>
              { $b/titel, $a/name }
        </ps>)
```

Nesting and Aggregation

- Nesting of FLWOR expression in an element constructor
- Aggregation
  - Function over a set of elements
  - Example: List all publishers with more than 100 books

```
<BIG_PUBLISHERS>
  {
        FOR $p IN distinct(document("bib.xml")//publisher)
        LET $b := document("bib.xml")//book[publisher = $p]
        WHERE count($b) > 100
        RETURN $p
  }
</BIG_PUBLISHERS>
```

  - LET clause binds $b to a **set** of books

XQuery: Joins

- Joins are specified in a manner very similar to SQL

```
for $a  in  /bank/account,
        $c  in  /bank/customer,
        $d  in  /bank/depositor
where $a/account-number = $d/account-number
                and $c/customer-name = $d/customer-name
return <cust-acct>{ $c $a }</cust-acct>
```

- The same query can be expressed with the selections specified as XPath selections:

```
for $a in /bank/account
    $c in /bank/customer
    $d in /bank/depositor[
                account-number =$a/account-number and
                customer-name = $c/customer-name]
return <cust-acct>{ $c $a }</cust-acct>
```

XQuery: Outer Join

- Example: List all suppliers. If a supplier offers medical items, list the descriptions of the items

```
FOR $s IN document("suppliers.xml")//supplier
ORDER BY $s/name
RETURN
  <supplier>
    { $s/name,
      FOR $ci IN document("catalog.xml")//item[supp_no = $s/number],
          $mi IN document("medical_items.xml")//item[number = $ci/item_no]
      RETURN $mi/description
    }
  </supplier>
```

- Problem with full outer join: nesting forces asymmetric representation
  - produce a two-part document, enclosed by a <master_list> element
  - query needs a separate expression for computing the "orphan" items

Quantified Expressions

- Existential Quantification
  - Give me all books where "Sailing" and "Windsurfing" appear at least once in the same paragraph

```
FOR $b IN //book
WHERE SOME $p IN $b//para SATISFIES (contains($p, "Sailing")
                                     AND contains($p, "Windsurfing"))
RETURN $b/title
```

- Universal Quantification
  - Give me all books where "Sailing" appears in every paragraph

```
FOR $b IN //book
WHERE EVERY $p IN $b//para SATISFIES contains($p, "Sailing")
RETURN $b/title
```

Defining and Using Functions

- Predefined Functions
  - XPath/XQuery function library, e.g., document( )
  - aggregation functions: avg, sum, count, max, min
  - additional functions: distinct-values( ), empty( ), …
- User-defined Functions
  - Example: compute maximal path length in "bib.xml"

```
DECLARE FUNCTION local:depth($e AS node()) AS xs:integer
{
  (: A node with no children has depth 1 :)
  (: Otherwise, add 1 to max depth of children :)
  IF (empty($e/*))
            THEN 1
            ELSE 1 + fn:max( FOR $c IN $e/* RETURN local:depth($c) )
  };
LET $h := document("bib.xml")
RETURN
            <depth>{ local:depth($h) }</depth>
```


Function Definitions

- Function definitions may not be overloaded in Version 1
  - Much XML data is untyped
  - XQuery attempts to cast arguments to the expected type
  - Example: **abs($x)** expects a numeric argument
    - If **$x** is a number, return its absolute value
    - If **$x** is untyped, cast it to a number
    - If **$x** is a node, extract its value and treat as above
  - This "argument conditioning" conflicts with function overloading
  - XML Schema substitution rules are already very complex
    - two kinds of inheritance; substitution groups; etc.
  - A function can simulate overloading by branching on the type of its argument, using a **typeswitch** expression

Two Phases in Query Processing

- Static analysis (compile-time; optional)
  - Depends only on the query itself
  - Infers result type of each expression, based on types of operands
  - Raises error if operand types don't match operators
  - Purpose: catch errors early, guarantee result type
  - May be helpful in query optimization
- Dynamic evaluation (run-time)
  - Depends on input data
  - Computes the result value based on the operand values
- If a query passes static analysis, it may still raise an error at evaluation time
  - It may divide by zero
  - Casts may fail. Example:
    cast as integer($x) where value of **$x** is "garbage"
- If a query fails static type checking, it may still evaluate successfully and return a useful result.
  - Example (with no schema):
    $emp/salary + 1000
  - Static semantics says this is a type error
  - Dynamic semantics executes it successfully if **$emp** has exactly one salary subelement with a numeric value

XQuery - Status

- Some Recent Enhancements
  - Complete Specification of XQuery Functions and Operators
  - Joint XQuery/XPath data model
  - Type checking model
    - static vs. dynamic type checking as an option
    - with/without schema information
  - A lot of problems fixed
  - Current status: working draft under public review
    - fairly close to becoming a w3c recommendation
- Ongoing and Future Work
  - Full-text support
  - Insert, Update, Delete
  - View definitions, DDL
  - Host language bindings, APIs
    - JSR 225: XQuery API for JavaTM (XQJ)
    - problem to overcome: tradtional XML processing API is based on well-defined documents

Literature & Information

- Rahm, E., Vossen, G. (Eds.):
  Web & Datenbanken – Konzepte, Architekturen, Anwendungen,
  dpunkt-Verl., 2003
- Don Chamberlin: XQuery: A Query Language for XML, SIGMOD 2003 Tutorial
- Don Chamberlin: XQuery: An XML Query Language, IBM Systems Journal,
  vol. 41, no. 4, 2002
  - both available for download at http://www.almaden.ibm.com/u/chamberlin/
- http://www.w3.org/TR/xquery/