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

Technische Universität Kaiserslautern

Chapter 8 – XQuery

Recent Developments for Data Models

Outline

Overview

I. Object-Relational Database Concepts

1. User-defined Data Types and Typed Tables
2. Object-relational Views and Collection Types
3. User-defined Routines and Object Behavior
4. Application Programs and Object-relational Capabilities

II. Online Analytic Processing

5. Data Analysis in SQL
6. Windowed Tables and Window Functions in SQL

III. XML

7. XML Data Modeling
8. **XQuery**
9. SQL/XML

IV. More Developments (if there is time left)
temporal data models, data streams, databases and uncertainty, …

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
      - e.g., 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

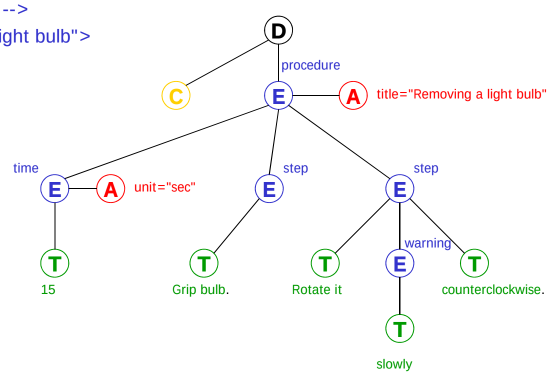
XQuery

- XQuery is a general purpose query language for XML data
- Standard developed by the World Wide Web Consortium (W3C)
  - W3C Recommendation since January 23rd, 2007
- 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, attributes, text, etc.
- 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 (XDM)

- Builds on a tree-based model, but extends it to support sequences of items
  - represent collections of documents and complex values
  - reflect (intermediate) results of query evaluation
  - closure property
    - XQuery queries and expressions operate on/produce instances of the XDM
- Based on XML Schema for precise type information
- XDM **instance**
  - ordered **sequence** of zero or more **items**
  - can contain heterogenous values
  - cannot be nested – all operations on sequences automatically "flatten" sequences
    - no distinction between an item and a sequence of length 1
  - may contain duplicate nodes (see below)
- An **item** is a **node** or an **atomic value**
- **Atomic values** are typed values
  - XML Schema simple types
  - important for representing results of intermediate expressions in the data model

XDM - Nodes

- There are seven kinds of **nodes**
  - Document, Element, Attribute, Text, Namespace, Comment, Processing Instruction
- Nodes form a tree
  - consisting of
    - root node
    - nodes directly or indirectly reachable from the root node via accessors
      - children
        - only element, processing instruction, comment and text nodes can **be** children
        - only document and element nodes **have** children
      - attributes
      - namespace nodes
  - trees are called
    - documents, if the root is a document node
    - fragments, otherwise
  - trees have exactly one root
  - a node belongs to exactly one tree

XDM – Nodes (cont.)

- A node has an identity (preserved by operations on nodes)
- 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
  - root < child < namespace < attribute < descendants
  - children and descendants < following siblings
  - order of siblings correspond to order in document

General XQuery Rules

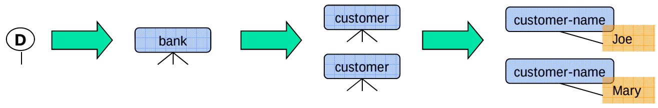
- XQuery is a case-sensitive language
- Keywords are in lower-case
- Every expression has a value and no side effects
- Expressions are fully composable
- Expressions can raise errors
- Expressions (usually) propagate lower-level errors
  - Exception: if-then-else
- Comments look like this
  - (: This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

- An XPath expression maps a node (the context node) into a sequence of nodes
  - consists of one or more steps separated by “/”
  - e.g.: return the names of all customers in bank
    /child::bank/child::customer/child::name
- Evaluation of path expression
  - step by step, from left to right
  - starting from an externally provided context node, or from document root
  - each step works on a sequence of nodes
    - for each node in the sequence, look up other nodes based on step expression
    - eliminate duplicates from result sequence
    - sort nodes in document order
  - empty result sequence does not result in an error

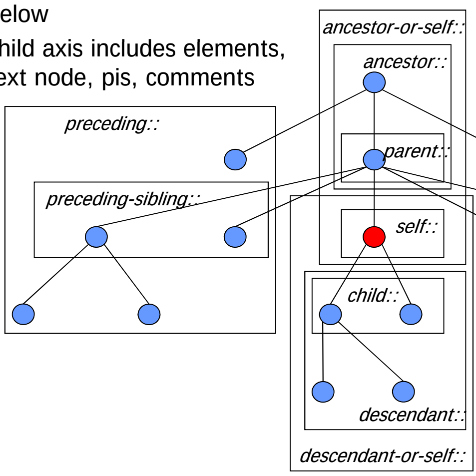


Path Expressions (cont.)

- 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)
  - Optional ***predicates*** (refine the set of qualifying nodes)
- Selection predicates may appear in any step in a path, in [ ]
  - Evaluated for each node qualified by axis/node test
  - E.g. /child::bank-2/child::account[child::balance > 400]
    - returns account elements with a balance value greater than 400
- Alternative: filter step
  - instead of axis::node-test, an expression can be used that locates nodes based on the context

Axis

- Result given in document order (exception: positional predicates)
- Axis for attributes and namespaces are available in addition to the ones listed below
- child axis includes elements, text node, pis, comments



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

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 to match nodes of a specific type
  - document-node()
  - comment()
  - text()
  - processing-instruction()
  - element(), element(name), element(name, type)
  - attribute(), attribute(name), attribute(name, type)
  - node() – matches any node

Node Test – Examples

- Find the names of all customers in bank
  /child::bank/child::customer/child::name
- Find all the element children of customers in bank
  /child::bank/child::customer/child::*
- Find all attributes of customer elements anywhere in the document
  /descendant::customer/attribute::*
- Find all attributes of customer elements having the type xs:string
  /descendant::customer/attribute::attribute(*, xs:string)
- Find all text nodes of the document
  /descendant::text()

Path Expressions – Abbreviated Notation

- Abbreviations
  - "."
    - current context node
  - ".."
    - "parent::node()"
  - "//"
    - "/descendant-or-self::node()/"
  - "@"
    - "attribute::"
  - axis missing
    - "child::"
    - (or "attribute::" with an attribute node type test)
- The following examples use the abbreviated notation:
  - Find the names of all customers in bank
    /bank/customer/name
  - Find all the element children of customers in bank
    /bank/customer/*
  - Find all attributes of customer elements anywhere in the document
    //customer/@*
  - Find all attributes of customer elements having the type xs:string
    //customer/attribute(*, xs:string)
  - Find all text nodes of the document
    //text()

Predicates

- Predicates can be used to apply additional filter conditions for the resulting nodes
  - Boolean expressions: selects all nodes for which expression returns "true"
    book[author = "Mark Twain"]
  - Numeric expressions: selects all nodes whose position is equal to the resulting value
    chapter[2]
  - Existence tests: selects nodes where expression does not result in empty sequence
    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:
  (1 to 100)[. mod 5 = 0]

Functions

- Context functions, e.g.
  - fn:last() returns the number of items in the current sequence
    - Find the last paragraph-child of the context node
      para[fn:last()]
  - fn:position() returns the position of the current item within the current sequence
    - Find the last paragraph-child of the context node (alternative query)
      para[fn:position()=fn:last()]
  - fn:current-date() returns the current date
    - Find names of customers who have an order with today's date
      //customer[order/date=fn:current-date()]/name
- Functions on nodes/items, e.g.
  - fn:string() returns the string value of an item
    - element nodes: concatenation of all descendant text nodes, in document order
- Functions and operators on sequences, e.g.
  - concatenation, distinct-values, subsequence
  - (deep) equal, union, intersect, except

Functions (cont.)

- IDREFs can be de-referenced using function fn:id()
  - fn: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/fn:id(@owners)
    returns all customers referenced by the owners attribute of account elements
- The function fn:doc(name) returns the root of the named document
  - E.g. fn:doc("bank.xml")/bank/account
- The function fn:collection(name) returns a sequence of nodes
  - E.g. fn:collection("myBankCollection")/bank/account

More Expressions

- 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>
```

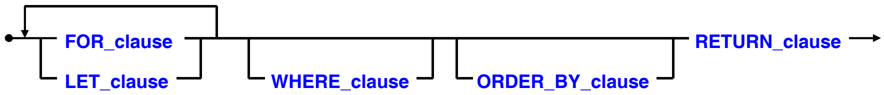
- If both the name and the content must be computed, use a computed constructor:

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


Validation of Constructed Elements

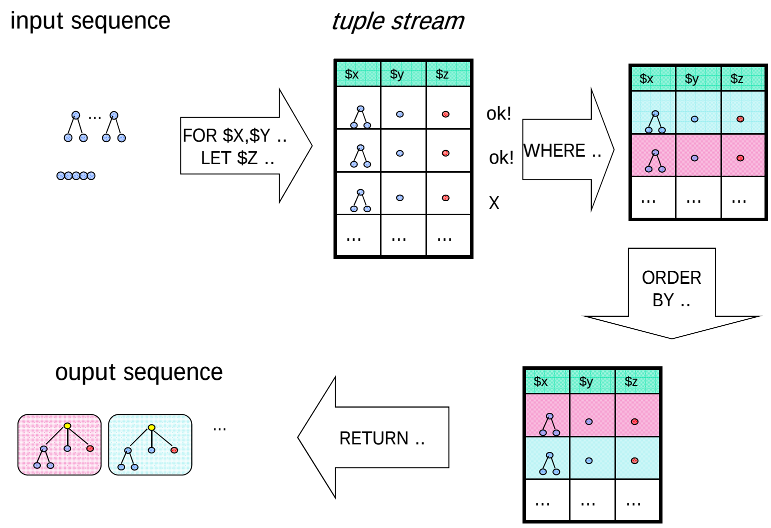
- An element constructor automatically validates the new element against "in-scope schema definitions"
  - Results in a **type annotation**
  - Can be generic: xs:anyType
- Validation mode (default = lax)
  - **Strict**: element must be defined in schema
  - **Lax**: element must match schema definition 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 a path 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

Evaluating FLWOR Expressions



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(doc("bib.xml")//publisher)
LET $a := avg(doc("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)
ORDER BY $a/name
RETURN
   <author>
      <name> { $a/text() } </name>
      { FOR $b IN //book[author = $a]
          RETURN $b/title }
   </author>
```

```
<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 = doc("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(doc("bib.xml")//publisher)
  ORDER BY $p/name
  RETURN
          <publisher>
                  <name> {$p/text()} </name>
                  {FOR $b IN doc("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 doc("bib.xml")//book,
              $a IN doc("authors.xml")//author
    WHERE $b/author_id = $a/id
    RETURN
        <ps>
              { $b/titel, $a/name }
        </ps>)
```


Nesting and Aggregation

- Aggregation
  - Function over a sequence of elements
    - count(), avg(), min(), max(), sum()
  - Example: List all publishers with more than 100 books

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

  - LET clause binds $b to a **sequence** 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 doc("suppliers.xml")//supplier
ORDER BY $s/name
RETURN
   <supplier>
     { $s/name,
       FOR $ci IN doc("catalog.xml")//item[supp_no = $s/number],
           $mi IN doc("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., doc( )
  - 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 $h := doc("bib.xml")
RETURN
          <depth>{ local:depth($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 API for Java$_{TM}$ (XQJ)

- Similar to JDBC, but for XQuery statements
  - data source, connection, (prepared) XQuery expression (statement)
    - XQuery variable identifier instead of parameter markers ("?")
- Query result is a sequence (XQSequence)
  - iterate through sequence items using XQSequence.next()
  - retrieve Java DOM objects using XQSequence.getObject()
  - retrieve atomic values as character string or mapped to Java data types
  - individual items or the complete stream can be "written" to the SAX API
- Support for "serializing" an XQuery result
  - to file, Java writer, string
  - as (X)HTML

XQuery Update Facility

- Introduces so-called updating expressions
  - potentially modify the state of an existing node
  - may occur on their own or nested inside other expressions
    - e.g., in the return clause of a FLWOR expression
- Update model: snapshot semantics
  - during query evaluation, updates are collected in a pending update list
    - contains update primitives, which have not been applied yet
      - update primitive identifies a target node, update operation
    - is returned by an XQuery expression, in addition to an XDM instance
  - only after the outermost expression has been evaluated, the updates in the list are applied

Insert and Delete Expression

- Insert copies of one or more nodes into designated position wrt. the target node
  - Syntax: **insert** <source-expression>
    ([**as** (**first** | **last**)] **into** | **after** | **before**) <target-expression>
    - target expression identifies a single element (or document) node
    - attribute nodes in source-expression result sequence have to appear before other nodes
  - before/after cause insertion as a preceding/following sibling of the target
  - into causes insertion as a child (or children) of the target
  - order of nodes in source-expression result sequence is preserved
  - Example: insert a year element after the publisher of the first book
    insert <year>2005</year> after fn:doc("bib.xml")/books/book[1]/publisher
- Delete zero or more nodes
  - Syntax: **delete** <target-expression>
  - Example: delete the last author of the first book
    delete fn:doc("bib.xml")/books/book[1]/author[last()]

Replace and Rename Expressions

- Replacing nodes or values
  - Syntax: **replace** [**value of**] <target-expression> **with** <new-expression>
  - can replace a node with a new sequence of nodes
    - node types must match (e.g., attribute can only be replaced by attribute(s))
    - Example: replace publisher of first book with publisher of second book
      replace fn:doc("bib.xml")/books/book[1]/publisher
      with fn:doc("bib.xml")/books/book[2]/publisher
  - can replace the value of a node using the 'value of' clause
    - replace attribute value or element content (text node)
    - Example: increase the price of the first book by 10 percent
      replace value of fn:doc("bib.xml")/books/book[1]/price
      with fn:doc("bib.xml")/books/book[1]/price * 1.1
- Rename an XDM node
  - Syntax: **rename** <target-expression> **as** <new-name-expr>
    - new-name-expr has to return an XML qualified name
  - Example: rename the first author element of the first book to 'principal-author'
    rename fn:doc("bib.xml")/books/book[1]/author[1] as "principal-author"

Transform Expression

- Creates modified copy of existing nodes
  - Syntax: **transform**

    copy <var> **:=** <expr> {**,** <var> **:=** <expr>}*
    modify <updating-expression>
    return <return-expression>
    - copy clause binds variable(s) to copied node sequence(s)
    - modify clause specifies updates to be performed on the copied nodes
    - return clause defines the result fo the transform expression
      - updates specified in the update clause have been performed, are visible
  - Example: return copies of all XML books with their price deleted

```
for $b in fn:doc("bib.xml")/books/book[contains(title, "XQuery")]
return
  transform
           copy $xb := $b
           modify do delete $xb/price
           return $xb
```

- Transform does not modify any existing nodes, is not an updating expression!

Evaluating Multiple Updating Expressions

- Compatibility
  - Within a given snapshot, a node may not be the target of
    - more than one rename expression
    - more than one replace expression
    - more than one replace value of expression
  - A replace value of $a expression wins over replace expressions of children of $a
- Update primitives in the pending update list identify nodes by their id
- Well-defined order of performing update primitives
  1. insertInto, insertAttributes, replaceValue, rename, delete (mark for deletion only!)
  2. insertBefore, insertAfter, insertIntoAsFirst, insertIntoAsLast
  3. replaceNode
  4. replaceElementContent
  5. delete (remove marked nodes)

XQuery - Status

- XQuery 1.0 is a w3c recommendation since January 2007
- XQuery API for Java$_{TM}$ (XQJ) is final (JSR) since 2009
- XQuery Update Facility 1.0 is a candidate recommendation
- XQuery 1.1 is in the making (working draft), work items include
  - value-based and positional grouping
  - outer join support
  - windowing
  - date and numeric value formatting
- Additional ongoing work
  - XQuery and XPath Full Text 1.0 (candidate recommendation)
    - adds support for text retrieval in XQuery
  - XQuery Scripting Extensions 1.0 (working draft)
    - adds procedural features

Summary

- Characteristics of XML (from a data modeling perspective)
  - data/meta-data integration, schema flexibility, heterogeneity, nesting, ordering, ...
- XQuery provides a powerful initial step towards an XML query language that reflect the above characteristics
- XQuery Data Model (XDM)
  - builds on XML tree structure, introduces sequences and atomic values
  - basis for XQuery processing, supports closure property
- Major query language constructs
  - path expressions
  - constructors
  - FLWOR expressions
- Problem: lack of an algebraic foundation