

XQuery

- **XQuery is to XML what SQL is to relational databases**
- **XQuery is designed to query XML data - not just XML files, but anything that can appear as XML, including databases**
- XQuery is built on XPath expressions
- XQuery is defined by the W3C
- XQuery is supported by all the major database engines (IBM, Oracle, Microsoft, etc.)
- XQuery is a W3C standard - and developers who correctly implement its specs are guaranteed that the code will work among different products

XQuery has many implementations

Just to mention one:

BaseX

<http://basex.org/>

Data model

- Instance of the data model:
 - a sequence composed of zero or more items
- The empty sequence often considered as the “null value”
- Items
 - nodes or atomic values
- Nodes
 - document | element | attribute | text | namespaces | PI | comment
- Atomic values
 - Instances of all XML Schema atomic types
 - string, boolean, ID, IDREF, decimal, QName, URI, ...
 - untyped atomic values
- Typed (I.e. schema validated) and untyped (I.e. non schema validated) nodes and values

Sequences

- Can be heterogeneous (nodes and atomic values)
 - ($\langle a \rangle$, 3)
- Can contain duplicates (by value and by identity)
 - (1,1,1)
- Are not necessarily ordered in document order
- Nested sequences are automatically flattened
 - (1, 2, (3, 4)) = (1, 2, 3, 4)
- Single items and singleton sequences are the same
 - 1 = (1)

Atomic values

- The values of the 19 atomic types available in XML Schema
 - E.g. `xs:integer`, `xs:boolean`, `xs:date`
- All the user defined derived atomic types
 - E.g. `myNS:ShoeSize`
 - `xdt:untypedAtomic`
- Atomic values carry their type together with the value
 - `(8, myNS:ShoeSize)` is not the same as `(8, xs:integer)`
- XQuery grammar has built-in support for:
 - Strings: `"125.0"` or `'125.0'`
 - Integers: `150`
 - Decimal: `125.0`
 - Double: `125.e2`
 - 19 other atomic types available via XML Schema
 - Values can be constructed by
 - constructors in F&O doc: `xf:true()`, `xf:date("2002-5-20")`
 - Casting
 - schema validation

XML nodes

- 7 types of nodes:
 - document | element | attribute | text | namespaces | PI | comment
- Every node has a unique node identifier
- Nodes have children and an optional parent
 - conceptual "tree"
- Nodes are ordered based of the topological order in the tree ("document order")

Combining sequences

- Union, Intersect, Except
- Work only for sequences of nodes, not atomic values
- Eliminate duplicates and reorder to document order
 $\$x := \langle a \rangle, \$y := \langle b \rangle, \$z := \langle c \rangle$
 $(\$x, \$y) \text{ union } (\$y, \$z) \Rightarrow (\langle a \rangle, \langle b \rangle, \langle c \rangle)$
- F&O specification provides other functions & operators; eg `xf:distinct-values()` and `xf:distinct-nodes()` particularly useful.

XQuery type system

- XQuery's has a powerful (and complex!) type system
- XQuery types are imported from XML Schemas
- Every XQuery expression has a static type
- Every XML data model instance has a dynamic type
- The goal of the type system is:
 1. detect statically errors in the queries
 2. infer the type of the result of valid queries
 3. ensure statically that the result of a given query is of a given (expected) type if the input dataset is guaranteed to be of a given type

XQuery Structure

- An XQuery basic structure:
 - prologue + expression
- Role of the prologue:
 - Populates the context where expressions are compiled and evaluated
 - Contains:
 - namespace definitions
 - schema imports
 - default element and function namespace
 - function definitions
 - collations declarations
 - function library imports
 - global and external variables definitions
 - etc.

XQuery expressions

XQuery Expr := Constants | Variable | FunctionCalls |
PathExpr | ComparisonExpr |
ArithmeticExpr | LogicExpr |
FLWRExpr | ConditionalExpr |
QuantifiedExpr | TypeSwitchExpr |
InstanceofExpr | CastExpr | UnionExpr |
IntersectExceptExpr | ConstructorExpr |
ValidateExpr

Expressions can be nested with full generality!

Variables

- \$ + Name
- bound by expression evaluation
- created by let, for, some/every, type-switch expressions, or as function parameters
- example:
 - let \$x := (1, 2, 3)
return count(\$x)
- above scoping ends at conclusion of return expression

Functions

- XQuery uses functions to extract data from XML documents
- The doc() function is used to open the "books.xml" file
- empty(item*) => boolean
- index-of(item*, item) => xs:unsignedInt?
- distinct-values(item*) => item*
- distinct-nodes(node*) => node*
- union(node*, node*) => node*, except(node*, node*) => node*
- string-length(xs:string?) => xs:integer?
- contains(xs:string, xs:string) => xs:boolean
- date(xs:string) => xs:date
- add-date(xs:date, xs:duration) => xs:date
- In-place XQuery functions, such as:
declare function ns:foo(\$x as xs:integer) as element()
{ <a>{\$x+1} }
- Can be recursive and mutually recursive

Arithmetic expressions

- Apply the following rules:
 - atomize all operands
 - if every item in the input sequence is either an atomic value or a node whose typed value is a sequence of atomic values, then return it
 - if an operand is untyped, cast to `xs:double` (if unable, => error)
 - if the operand types differ but can be promoted to common type, do so (e.g.: `xs:integer` can be promoted to `xs:decimal`)
 - if operator is consistent w.r.t. types, apply it; result is either atomic value or error
 - otherwise, throw type exception
- Examples:
 - $-1 - (4 * 8.5)$
 - $\$b \bmod 10$
 - `<a>42 + 1`

Logical expressions

- They are:
 - `expr1` and `expr2`
 - `expr1` or `expr2`
- Return true, false (two value logic, not three value logic like SQL!)
- Rules:
 - first compute the Boolean Effective Value (BEV) for each operand:
 - if `()`, `""`, `0`, zero length string then return false
 - if the operand is of type boolean, its BEV is its value;
 - else return true
 - then use standard two value Boolean logic on the two BEV's as appropriate
- false and error => false or error! (non-deterministic: it is impossible to foresee the result!!!)

XML Data Management

Comparison

Value	for comparing single values	eq, ne, lt, le, gt, ge
General	Existential quantification + automatic type coercion	=, !=, <=, <, >, >=
Node	for testing identity of single nodes	is, isnot
Order	testing relative position of one node vs. another (in document order)	<<, >>

Values and comparisons

- `<a>42 eq "42"` true
- `<a>42 eq 42` error
- `<a>42 eq 42.0` error
- `<a>42 = 42` true
- `<a>42 = 42.0` true
- `<a>42 eq 42` true
- `<a>42 eq 42` false
- `<a>baz eq 42` type error
- `() = 42` false
- `(<a>42, 43) = 42` true
- `ns:shoesize(5) eq ns:hatsize(5)` true
- `(1,2) = (2,3)` true
- `(1,2) != (2,3)` true
- `(1,2) != (1,2)` true

Running example

```
<bib>
  <book year="1994">
    <title>TCP/IP Illustrated</title>
    <author><last>Stevens</last><first>W.</first></author>
    <publisher>Addison-Wesley</publisher>
    <price>65.95</price>
  </book>
  <book year="2000">
    <title>Data on the Web</title>
    <author><last>Abiteboul</last><first>Serge</first></author>
    <author><last>Buneman</last><first>Peter</first></author>
    <author><last>Suciu</last><first>Dan</first></author>
    <publisher>Morgan Kaufmann Publishers</publisher>
    <price>39.95</price>
  </book>
  <book year="1999">
    <title>The Economics of Technology and Content for Digital TV</title>
    <editor>
      <last>Gerbarg</last><first>Darcy</first>
      <affiliation>CITI</affiliation>
    </editor>
    <publisher>Kluwer Academic Publishers</publisher>
    <price>129.95</price>
  </book>
</bib>
```

Running example

```
<!ELEMENT bib ( book* )>
```

```
<!ELEMENT book ( title,  
  ( author+ | editor+ ),  
  publisher,  
  price )>
```

```
<!ATTLIST book  
  year CDATA #REQUIRED >
```

```
<!ELEMENT author ( last, first )>
```

```
<!ELEMENT editor ( last, first, affiliation )>
```

```
<!ELEMENT title (#PCDATA)>
```

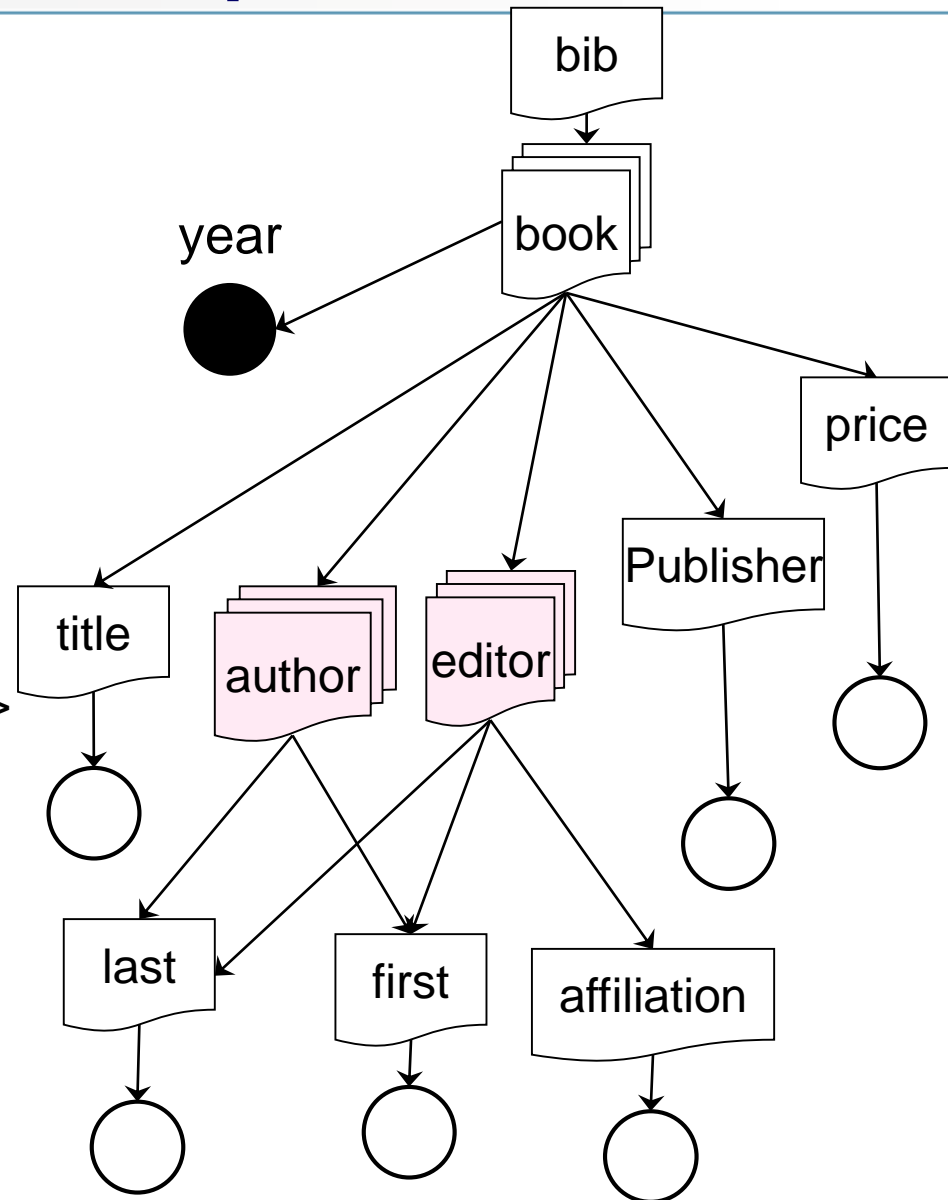
```
<!ELEMENT last (#PCDATA)>
```

```
<!ELEMENT first (#PCDATA)>
```

```
<!ELEMENT affiliation (#PCDATA)>
```

```
<!ELEMENT publisher (#PCDATA)>
```

```
<!ELEMENT price (#PCDATA)>
```



FLWOR expressions

- 5 clauses:
 - FOR
 - LET
 - WHERE
 - ORDER BY
 - RETURN

Simple iteration expressions

- **for** *variable* **in** *expression1*
return *expression2*
- Example: for \$x in doc("bib.xml")/bib/book
 return \$x/title
- Semantics :
 - iteratively bind the variable to each root node of the forest returned by expression1
 - for each such binding, evaluate expression2
 - concatenate the resulting sequences
 - nested sequences are automatically flattened

LET Expression

- *let variable := expression1*
return expression2
- Example : `let $x :=doc("bib.xml")/bib/book`
 `return count($x)`
- Semantics:
 - bind the variable to the result of the expression1 (in the example, an entire sequence of values)
 - add this binding to the current environment
 - evaluate and return expression2

FOR vs LET

- `let $x := doc("bib.xml")/bib/book`
`return count($x)`
- `for $x in doc("bib.xml")/bib/book`
`return count($x)`

WHERE




- WHERE clause express a condition over nodes; only nodes satisfying the condition are further considered.
- Conditions can contain AND and OR.
- **not()** is implemented by a function
- Example:

```
for $book in doc ("books.xml")//book  
where $book/publisher="Bompiani"  
        and $book/@avalilable="Y"  
return $book
```
- Same as:

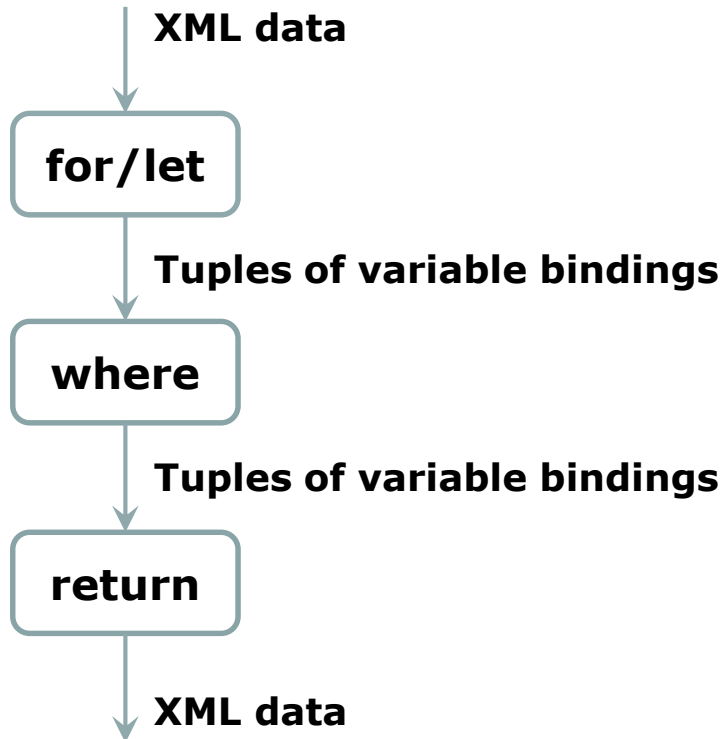
```
doc("books.xml")//book[publisher="Bompiani" and @avalilable="Y"]
```

XML Data Management

RETURN

- This clause can generate:
 - A node (or tree) 
 - An ordered "forest" (of nodes or trees) 
 - A textual value (PCdata) 
- It can contain node constructors
 - `<result>`
literal text content
`</result>`
 - `<result>`
{ \$x/name } {-- Braces "{" used to delineate evaluated content --}
`</result>`
 - `<result>`
some content here { \$x/name } and some more here
`</result>`

FLWR expressions: evaluation



- **for** : iteration, “individual” bindings
 - Every item in a sequence generates a different binding (whose value is that item)
- **let** : “collective” binding
 - A sequence of items is collectively bound to one variable (whose value is the whole sequence)
- **where** : filtering expressions
 - Independently evaluated on each tuple of bindings to keep or discard it
- **return** : construct results
 - Executed once for each tuple of bindings

A FWR query

```
for $book in doc("books.xml")//book
where $book/price>60
return   <expensiveBook>
           { $book/title }
           </expensiveBook>
```

```
<expensiveBook><title>TCP/IP Illustrated</title></expensiveBook>
<expensiveBook>
  <title>The Economics of Technology and Content for Digital TV</title>
</expensiveBook>
```

XML Data Management

Similar query with use of text() function

```
for $book in doc("books.xml")//book
where $book/price>60
return <expensiveBook>
      { $book/title/text() }
      </expensiveBook>
```

```
<expensiveBook>TCP/IP Illustrated</expensiveBook>
<expensiveBook>
  The Economics of Technology and Content for Digital TV
</expensiveBook>
```

Similar query with LET

```
let $books := doc("books.xml")//book[price>60]
return <expensiveBooks>
      { $books/title }
      </expensiveBooks>
```

```
<expensiveBooks>
  <title>TCP/IP Illustrated</title>
  <title>The Economics of Technology and Content for Digital TV</title>
</expensiveBooks>
```

XML Data Management

Similar (but unusual) query with LET

```
let $books := doc("books.xml")//book
where $books/price>60
return <expensiveBooks>
      { $books/title }
      </expensiveBooks>
```

```
<expensiveBooks>
  <title>TCP/IP Illustrated</title>
  <title>Data on the Web</title>
  <title>The Economics of Technology and Content for Digital TV</title>
</expensiveBooks>
```

Order by

```
for $book in doc("books.xml")//book
order by $book/title
return
  <book>
    { $book/title,
      $book/publisher }
  </book>
```

Counting combined with let clause

```
for $e in doc("books.xml")//publisher
let $book := doc("books.xml")//book[publisher = $e]
where count($book) > 100
return $e
```

- Each publisher is in the result more than 100 times

distinct-values()

```
for $e in distinct-values( doc("books.xml")//publisher )
let $book := doc("books.xml")//book[publisher = $e]
where count($book) > 100
return $e
```

Each publisher is in the result only once

Conditional expressions

- if (\$book/@year <1980)
then (<old-book> {\$x/title} </old-book>)
else (<new-book> {\$x/title} </new-book>)

FLWR expressions

- **FLWR expression:**
for \$x in //book
let \$y := \$x/author
where \$x/title="Ulysses"
return count(\$y)
- **Equivalent to:**
for \$x in //bib/book
return (let \$y := \$x/author
return if (\$x/title="Ulysses")
then count(\$y)
else ())

FLWR expressions

- **Selections**

```
for $b in doc("bib.xml")//book
where $b/publisher = "Springer Verlag" and
      $b/@year = "1998"
return $b/title
```

- **Joins**

```
for $b in doc("bib.xml")//book,
     $p in doc("pubs.xml")//publisher
where $b/publisher = $p/name
return ( $b/title , $p/address)
```

Order by

- for \$b in doc("bib.xml")//book
where \$b/publisher = "Springer Verlag"
order by \$b/@year
return \$b/title

List books published by Addison-Wesley after 1991, including their year and title.

```
<bib>
{
  for $b in doc("bib.xml")/bib/book
  where $b/publisher = "Addison-Wesley"
      and $b/@year > 1991
  return
    <book year="{ $b/@year }">
      { $b/title }
    </book>
}
</bib>
```

List the titles and years of all books published by Addison-Wesley after 1991, **in alphabetic order**

```
<bib>
  {
    for $b in doc("bib.xml")//book
      where $b/publisher = "Addison-Wesley"
        and $b/@year > 1991
      order by $b/title
      return
        <book>
          { $b/@year }
          { $b/title }
        </book>
  }
</bib>
```

Create a flat list of all the title-author pairs, with each pair enclosed in a "result" element.

```
<results>
{
  for $b in doc("bib.xml")/bib/book,
    $t in $b/title,
    $a in $b/author
  return
    <result>
      { $t }
      { $a }
    </result>
}
</results>
```

Compute the full Cartesian product of all title and all authors, disregarding the actual authorship relationship.

```
<results>
{
  for $t in doc("bib.xml")/bib/book/title,
    $a in doc("bib.xml")/bib/book/author,
  return
    <result>
      { $t }
      { $a }
    </result>
}
</results>
```


For each book in the bibliography, list the title and authors, grouped inside a "result" element

```
<results>
{
  for $b in doc("bib.xml")/bib/book
  return
    <result>
      { $b/title }
      { $b/author }
    </result>
}
</results>
```

XML Data Management

For each author in the bibliography, list the author's name and the titles of all books by that author, grouped in a "result" element.

```
<results> {  
  for $a in doc("bib.xml")//author  
  order by $a/last, $a/first  
  return  
    <result>  
      <author> { $a/last } { $a/first }</author>  
      {  
        for $b in doc("bib.xml")/bib/book  
        where $b/author=$a  
        return $b/title  
      }  
    </result>  
}</results>
```

XML Data Management

For each author in the bibliography, list the author's name and the titles of all books by that author, grouped in a "result" element.

```
<results> {  
  for $a in distinct-values(doc("bib.xml")//author)  
  order by $a/last, $a/first  
  return  
    <result>  
      <author> { $a/last } { $a/first }</author>  
      {  
        for $b in doc("bib.xml")/bib/book  
        where $b/author=$a  
        return $b/title  
      }  
    </result>  
} </results>
```

XML Data Management

For each book that has at least one author, list the title and first two authors, and an empty "et-al" element if the book has additional authors.

```
<bib> {  
  for $b in doc("bib.xml")//book  
  where count($b/author) > 0  
  return  
    <book> { $b/title }  
            { $b/author[position()<=2] }  
            { if (count($b/author) > 2)  
              then <et-al/>  
              else () }  
    </book>  
} </bib>
```

Find books in which the name of some element ends with the string "or" and the same element contains the string "Suciu" somewhere in its content. For each such book, return the title and the qualifying element.

```
for $b in doc("bib.xml")//book
let $e := $b/*[contains(string(.), "Suciu")
               and ends-with(local-name(.), "or")]
where exists($e)
return
  <book>
    { $b/title }
    { $e }
  </book>
```

XML Data Management

For each book with an author, return the book with its title and authors. For each book with an editor, return a reference with the book title and the editor's affiliation.

```
<bib>
{   for $b in doc("bib.xml")//book[author]
    return <book>
        { $b/title }
        { $b/author }
    </book>
}
{   for $b in doc("bib.xml")//book[editor]
    return <reference>
        { $b/title }
        {$b/editor/affiliation}
    </reference>
}
</bib>
```

XML Data Management

Find pairs of books that have different titles but the same set of authors (possibly in a different order).

<bib>

```
{ for $book1 in doc("bib.xml")//book,  
  $book2 in doc("bib.xml")//book  
  let $aut1 := for $a in $book1/author  
              order by $a/last, $a/first  
              return $a
```

```
  let $aut2 := for $a in $book2/author  
              order by $a/last, $a/first  
              return $a
```

```
  where $book1 << $book2 and not($book1/title = $book2/title)  
        and deep-equal($aut1, $aut2)
```

```
  return <book-pair>  
        { $book1/title }  
        { $book2/title }  
  </book-pair>
```

} </bib>

XML Data Management

```
<book>
  <title>Data on the Web</title>
  <author>Serge Abiteboul</author>
  <author>Peter Buneman</author>
  <author>Dan Suciu</author>
  <section id="intro" difficulty="easy" >
    <title>Introduction</title>
    <p>Text ... </p>
  <section>    <title>Audience</title> <p>Text ... </p>  </section>
  <section>
    <title>Web Data and the Two Cultures</title> <p>Text ... </p>
    <figure height="400" width="400">
      <title>Traditional client/server architecture</title>
      <image source="csarch.gif"/>
    </figure>
    <p>Text ... </p>
  </section>
  (...)
</section>
</book>
```


XML Data Management

Prepare a (nested) table of contents for Books, listing all sections with titles. Preserve the original attributes of each <section> element, if any.

```
declare function local:toc($book-or-section as element()) as element()* {  
  for $section in $book-or-section/section  
  return  
    <section>  
      { $section/@* ,  
        $section/title ,  
        local:toc($section) }  
    </section>  
};  
  
<toc> {  
  for $s in doc("bib.xml")//book  
  return local:toc($s)  
} </toc>
```

Prepare a (flat) figure list for Books, listing all the figures and their titles. Preserve the original attributes of each <figure> element, if any.

```
<figlist>
{
  for $f in doc("bib.xml")//figure
  return
    <figure>
      { $f/@* }
      { $f/title }
    </figure>
}
</figlist>
```

How many sections are in Books, and how many figures?

```
<section_count>
{
  count(doc("bib.xml")//section)
}
</section_count>
```

```
<figure_count>
{
  count(doc("bib.xml")//figure)
}
</figure_count>
```

How many top-level sections are in Books?

```
<top_section_count>
{
  count(doc("bib.xml")/bib/book/section)
}
</top_section_count>
```

Make a flat list of the section elements in Books. In place of its original attributes, each section element should have two attributes, containing the title of the section and the number of figures immediately contained in the section.

```
<section_list>
{
  for $s in doc("bib.xml")//section
  let $f := $s/figure
  return
    <section title="{ $s/title/text() }" figcount="{ count($f) }"/>
}
</section_list>
```

XML Data Management

Make a nested list of section elements in Books, preserving their original attributes and hierarchy. Inside each section element, include the title of the section and an element that includes the number of figures immediately contained in the section.

```
declare function local:summary($bookOrSec as element()) as element()* {
  for $section in $bookOrSection/section
  return <section>
    { $section/@* }
    { $section/title }
    <figcount>
      { count($section/figure) }
    </figcount>
    { local:summary($section/section) }
  </section>
};
<toc> { for $s in doc("bib.xml")/bib/book
  return local:summary($s)
} </toc>
```

Missing functionalities

- Standard semantics for Web services invocation
- Try-catch mechanism
- Group by
- Distinct by
- Full text search
- Integrity constraints / assertions
- Metadata introspection

RECENTLY ADDED

- Updates

XML Data Management Technology

- Two families of XML stores:
- **Native XML dbs**
 - Use XML-specific storage management
 - Support only XML query languages
- **Relational databases with XML support**
 - Based on relational storage systems which are suitably extended
 - Integrate SQL with XQuery

Native XML databases

- Use data models which are not relational and can be standard or proprietary
 - ES: DOM, XPath Data Model, XML Information Set
- Use proprietary physical data stores, which are document-centric
- Organize databases as document collections
- Examples: Tamino, Xyleme, eXist, Galax, ...

Relational DBMSs with XML support

- Use relational data stores heavily, but support as well native XML data
- When they map XML data to relational structures, may have rather different mapping to relational schemas:
 - Fixed and DTD independent (canonic)
 - Variable and DTD specific (custom)
 - Decided by the physical DB optimizer (e.g. with mapping rules)

How to pass "input" data to a query ?

- External variables (bound through an external API)
 - declare variable \$x as xs:integer external
- Current item (bound through an external API)
 - . (a dot)
- External functions (bound through an external API)
 - declare function bea:foo() as document * external
- Specific built-in functions
 - **xf:doc(uri)**, xf:collection(uri)

XQuery Update Facility

XML Data Management

DTD for users.xml

```
<!ELEMENT users (user_tuple*)>
<!ELEMENT user_tuple (userid, name, rating?)>
<!ELEMENT userid (#PCDATA)>
<!ELEMENT name (#PCDATA)>
<!ELEMENT rating (#PCDATA)>
```

DTD for items.xml

```
<!ELEMENT items (item_tuple*)>
<!ELEMENT item_tuple (itemno, description, offered_by, start_date?, end_date?,
                                                                reserve_price?)>
<!ELEMENT itemno (#PCDATA)>
<!ELEMENT description (#PCDATA)>
<!ELEMENT offered_by (#PCDATA)>
<!ELEMENT start_date (#PCDATA)>
<!ELEMENT end_date (#PCDATA)>
<!ELEMENT reserve_price (#PCDATA)>
```

DTD for bids.xml

```
<!ELEMENT bids (bid_tuple*)>
<!ELEMENT bid_tuple (userid, itemno, bid, bid_date)>
<!ELEMENT userid (#PCDATA)>
<!ELEMENT itemno (#PCDATA)>
<!ELEMENT bid (#PCDATA)>
<!ELEMENT bid_date (#PCDATA)>
```

Add a new user (with no rating) to the users.xml view.

```
do insert
  <user_tuple>
    <userid>U07</userid>
    <name>Annabel Lee</name>
  </user_tuple>
into doc("users.xml")/users
```

Enter a bid for user Anna Lee on February 1st, 1999 for 60 dollars on item 1001.

```
let $uid := doc("users.xml")/users/user_tuple[name="Anna Lee"]/userid
return do insert <bid_tuple>
    <userid>{data($uid)}</userid>
    <itemno>1001</itemno>
    <bid>60</bid>
    <bid_date>1999-02-01</bid_date>
</bid_tuple>
into doc("bids.xml")/bids
```

Insert a new bid for Anna Lee on item 1002, adding 10% to the best bid received so far for this item.

```
let $uid := doc("users.xml")/users/user_tuple[name="Anna Lee"]/userid
let $topbid := max(doc("bids.xml")/bids/bid_tuple[itemno=1002]/bid)
return do insert    <bid_tuple>
                    <userid>{data($uid)}</userid>
                    <itemno>1002</itemno>
                    <bid> {$topbid*1.1} </bid>
                    <bid_date>1999-02-01</bid_date>
                    </bid_tuple>
into doc("bids.xml")/bids
```


Place a bid for Anna Lee on item 1002, adding 10% to the best bid received so far on that item, **but only if the bid amount does not exceed a given limit (200).**

```
let $uid := doc("users.xml")/users/user_tuple[name="Anna Lee"]/userid
let $topbid := max(doc("bids.xml")/bids/bid_tuple[itemno=1002]/bid)
where $topbid*1.1 <= 200
return do insert <bid_tuple>
    <userid>{data($uid)}</userid>
    <itemno>1007</itemno>
    <bid>{$topbid*1.1}</bid>
    <bid_date>1999-02-01</bid_date>
</bid_tuple>
into doc("bids.xml")/bids
```

Set Annabel Lee's rating to B.

```
let $user := doc("users.xml")/users/user_tuple[name="Annabel Lee"] return do replace  
value of $user/rating with "B"
```

Erase user X Y and the corresponding associated items and bids.

```
let $user := doc("users.xml")/users/user_tuple[name="X Y"]
let $items := doc("items.xml")/items/item_tuple[offered_by=$user/userid]
let $bids := doc("bids.xml")/bids/bid_tuple[userid=$user/userid]
return (
    do delete $user,
    do delete $items,
    do delete $bids
)
```

An alternative solution is:

```
let $user := doc("users.xml")/users/user_tuple[name="X Y"]
let $items := doc("items.xml")/items/item_tuple[offered_by=$user/userid]
let $bids := doc("bids.xml")/bids/bid_tuple[userid=$user/userid]
return do delete $user, $items, $bids
```

Add the element

```
<comment>This is a bargain !</comment>
```

as the last child of the `<item>` element describing item 1002.

```
do insert <comment>This is a bargain !</comment>
```

```
as last into doc("items.xml")/items/item_tuple[itemno=1002]
```

XML Data Management

Place a bid for Anna Lee on item 1010, which does not exist in "items.xml". An application constraint requires that no bid be placed on an item which was not previously registered in the database.

```
let $uid := doc("users.xml")/users/user_tuple[name="Anna Lee"]/userid
```

```
return do insert <bid_tuple>
```

```
    <userid>{data($uid)}</userid>
```

```
    <itemno>1010</itemno>
```

```
    <bid>60</bid>
```

```
    <bid_date>2006-04-23</bid_date>
```

```
  </bid_tuple>
```

```
into doc("bids.xml")/bids
```

Expected resulting content of bids.xml: the same as the original contents.

This update violates an application constraint. Therefore, its execution raises a **dynamic error**.

XML Data Management

Document "part-tree.xml":

```
<parttree>
  <part partid="0" name="car">
    <part partid="1" name="engine">
      <part partid="3" name="piston"/>
    </part>
    <part partid="2" name="door">
      <part partid="4" name="window"/>
      <part partid="5" name="lock"/>
    </part>
  </part>
  <part partid="10" name="skateboard">
    <part partid="11" name="board"/>
    <part partid="12" name="wheel"/>
  </part>
  <part partid="20" name="canoe"/>
</parttree>
```

XML Data Management

Document "partlist.xml":

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<partlist>
  <part partid="0" name="car"/>
  <part partid="1" partof="0" name="engine"/>
  <part partid="2" partof="0" name="door"/>
  <part partid="3" partof="1" name="piston"/>
  <part partid="4" partof="2" name="window"/>
  <part partid="5" partof="2" name="lock"/>
  <part partid="10" name="skateboard"/>
  <part partid="11" partof="10" name="board"/>
  <part partid="12" partof="10" name="wheel"/>
  <part partid="20" name="canoe"/>
</partlist>
```

Delete all parts in "part-tree.xml".

```
do delete doc("part-tree.xml")//part
```


Delete all parts belonging to a car in "part-tree.xml", leaving the car itself.

```
do delete doc("part-tree.xml")//part[@name="car"]/part
```

Delete all parts belonging to a car in "part-list.xml", leaving the car itself.

```
for $pt in doc("part-tree.xml")//part[@name="car"]/part,  
    $pl in doc("part-list.xml")//part  
where $pt/@partid eq $pl/@partid  
return do delete $pl
```

or...

Alternative Solution (using a recursive function):

```
declare updating function local:delSubtree($p as
element(part)) {
    for $schild in doc("part-list.xml")//part
    where $p/@partid eq $schild/@partof
    return local:delSubtree($schild) , do delete $p
};
```

```
for $p in doc("part-tree.xml")//part[@name="car"]/part
return local:delSubtree($p)
```

Add a radio to the car in "part-tree.xml", using a part number that hasn't been taken.

```
let $next := max(doc("part-tree.xml")//@partid) + 1
return do insert <part partid="{ $next}" name="radio"/>
into doc("part-tree.xml")//part[@partid=0 and @name="car"]
```

Note: The position of the new element with respect to its siblings is implementation-dependent. If position is significant, and the user wants to ensure the element appears last, "as last" should be used, as in the following query:

```
let $next := max(doc("part-tree.xml")//@partid) + 1
return do insert <part partid="{ $next}" name="radio"/>
as last
into doc("part-tree.xml")//part[@partid=0 and @name="car"]
```

The head office has adopted a new numbering scheme. In "part-tree.xml", add 1000 to all part numbers for cars, 2000 to all part numbers for skateboards, and 3000 to all part numbers for canoes.

```
for $keyword at $i in ("car", "skateboard", "canoe"),
    $parent in doc("part-tree.xml")//part[@name=$keyword]
let $descendants := $parent//part
for $p in ($parent, $descendants)
return do replace value of $p/@partid with $i*1000+$p/@partid
```