# Data Bases II
## XQuery

Michele Beretta

michele.beretta@unibg.it

## Query languages for XML

XML can be considered a *semi-structured data model*, andd a set of XML documents can be considered a *large data collection*.

We can **query** XML with specific languages:

- **XPath**, a simple selection language
- **XQuery**, a rich query language
- **XSLT**, used for document transformations (not discussed)

## 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

## Data model

We work with a **sequence of zero or more items**. The empty sequence is often considered as the “null” value.

Each item is either

- A **node**: `document | element | attribute | text | namespace | PI | comment`
- An **atomic value**: string, boolean, ids, and so on

## Sequences

- Can be heterogeneous
  - `(<a />, 3)`
- Can contain duplicates (by value and by identity)
  - `(1, 1, 1)`
- Are not necessarily ordered
- Nestes sequences are automatically flattened
  - `(1, 2, (3, 4)) = (1, 2, 3, 4)`
- A single item and singleton sequences are the same
  - `1 = (1)`
- Can be combined through `union`, `intersect`, and `except` (duplicates are removed)

## 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` and decimals `125.0`
- Doublese `125.e2`
- 19 other atomic types available via XML Schema

## XML Nodes

There are 7 types of nodes: `document | element | attribute | text | namespace | PI | comment`.

Every node has a unique node identifier, children and an optional parent. Hence, the document is **conceptually a tree**.

Node are ordered based on the topological order in tree, the “document order”.

## XQuery syntax

XQuery has the following basic structure

1. **Prologue**: populates the context where expressions are compiled and evaluated (namespace definitions, schema imports, function declarations, etc.)
2. **Expression**: the actual query, can be nested with full generality

```
Expr := Constants | Variable | FunctionCall
     | PathExpr | ComparisonExpr
     | ArithmeticExpr | LogicExpr
     | FLWRExpr | ConditionalExpr
     | ...
```

## Variables

Simply, `$ + NAME`. Created by `let`, `for`, `some`/`every`, type-switch expressions, or function parameters.

```
let $x := (1, 2, 3)
```

## Functions

XQuery uses functions to extract data from XML documents.

- The `doc()` function is used to open XML files
- `empty(item*) => xs: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`

We can also declare our own

```
declare function ns:foo($x as xs:integer) as element() {
  return (<a>{$x+1}</a>) (: Hi :)
}
```

## Arithmetic expressions

There are some rules:

1. Atomize all operands
2. If an operand is untyped, cast to `xs:double` (if unable, error)
3. If the operand types differ but can be promoted to a common type, do so
4. If the operator is consistent w.r.t. types, apply it, the result is either an atomic value or an error (otherwise, throw a type exception)

## Logical expressions

What you would expect, the typical `and` and `or`. A couple of particularities:

- It is a **two-value logic**, unlike SQL which is a three-value logic
- Sometime it can be non-deterministic: `false and error` → `false or error`

There is a conversion step, namely to compute the *Boolean Effective Value* (BEV) for each operand:

- If `()`, `""`, `0`, `false`, then `false`
- Everything else is `true`

## Comparison

| | | |
|---|---|---|
| **Value** | Comparing single values | `eq`, `ne`, `lt`, `le`, `gt`, `ge` |
| **General** | Existential quantification with automatic type coercion | `=`, `!=`, `<=`, `>=`, `<`, `>` |
| **Node** | Testing identity of single nodes | `is`, `isnot` |
| **Order** | Testing relative position of one node vs. another (document order) | `<<`, `>>` |

# Comparison – examples

| | |
|---|---|
| `<a>42</a> eq "42"` | `true` |
| `<a>42</a> eq 42` | error |
| `<a>42</a> eq 42.0` | error |
| `<a>42</a> = 42` | `true` |
| `<a>42</a> = 42.0` | `true` |
| `<a>42</a> eq <b>42</b>` | `true` |
| `<a>42</a> eq <b> 42</b>` | `false` |
| `<a>baz</a> eq 42` | type error |
| `() = 42` | `false` |
| `(<a>42</a>, <b>43</b>) = 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` |

# FLWOR Expressions

We have 5 clauses:

- FOR
- LET
- WHERE
- ORDER BY
- RETURN

```
<bookstore>
  <book available="Y">
    <title>Il Signore degli Anelli</title>
    <author>J.R.R. Tolkien</author>
    <publisher>Bompiani</publisher>
  </book>
  <book available="N">
    <title>Il nome della rosa</title>
    <author>Umberto Eco</author>
    <publisher>Bompiani</publisher>
  </book>
  <book available="Y">
    <title>Metamorfosi</title>
    <author>F. Kafka</author>
    <publisher>Feltrinelli</publisher>
  </book>
</bookstore>
```

## Simple iteration expression

```
for [VAR] in [EXP1]
return [EXP2]
```

Example:

```
for $x in doc("bookstore.xml")//book
return $x/title
```

Semantics:

1. Iteratively bind `VAR` to each root node of the forest returned by `EXP1`
2. For each such binding, evaluate `EXP2`
3. Concatenate the resulting sequence (nested sequences are flattened)

## LET expression

```
let [VAR] := [EXP1]
return [EXP2]
```

Example:

```
let $x := doc("bookstore.xml")//book
return count($x)
```

Semantics:

1. Bind `VAR` to the result of `EXP1`
2. Add this binding to the current environment
3. Evaluate and return `EXP2`

## WHERE clause

The `where` clause express a condition over nodes. Only nodes satisfying the condition are further considered.

You can use `and` and `or`, while `not()` is a function.

Example:

```
for $book in doc("books.xml")//book
where $book/publisher = "Bompiani" and $book/@available = "Y"
```

## RETURN clause

Not your average `return`. This clause can generate:

- A node (or a tree)
- An ordered «forest» (of nodes or trees)
- A textual value (PCData)

It can contain node constructors:

- `<result>literal text</result>`
- `<result>{ $x/name }</result>` (braces are for interpolation)

## FLWR expression

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

1. XML data is iterated by `for`, which generates different bindings
2. `let` possibly defines some variables
3. `where` filters the data, independently evaluated on each tuple of bindings
4. `return` constructs results, executed once for each tuple of binding

Lastly, you can `order by` before `return`.

## Conditional expressions

Simply:

```
if ([CONDITION])
then EXPR1
else EXPR2
```

## How to do JOINs

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

## Where to get more

For a full and deeper understanding, refer to old editions.

# Some examples

Given this document

```
<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>
```

## Example 1

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

```
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>
```

## Example 2

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

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

## Example 3

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

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

## Example 4

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.

```
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>
```

## Example with a recursive function

Prepare a (nested) table of contents for Books, listing all sections with titles. Preserve the original attributes of each 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>
```