Information Systems (Informationssysteme)

Jens Teubner, TU Dortmund
jens.teubner@cs.tu-dortmund.de

Summer 2015

Part X

XML Processing

Limitations of the Relational Model

Suppose a shop sells **digital cameras**:

Products					
ProdID	Name	Price	Resol.	Memory	Lens
0815	SuperCam 2000	199.90	12 MP	512 MB	24mm
4200	CoolPhoto 15XT	379.98	12 MP	2 GB	22mm
4711	Foo Pix FX13	249.00	8 MP	4 GB	28mm

Or a shop might sell **printers**:

Products					
ProdID	Name	Price	Color	Speed	Resol.
1734	ePrinter R300c	499.90	yes	12 ppm	600 dpi
1924	PrintJet Duo	629.00	yes	14 ppm	1200 dpi
4448	OfficeThing VIx	299.98	no	20 ppm	600 dpi

Limitations of the Relational Model

What if a shop sells **both**? Fill with null values?

Products								
ProdID	Name	Price	Resol.	Memory	Lens	Color	Speed	Resol.
0815	SuperCam 2000	199.90	12 MP	512 MB	24mm	–	–	–
1734	ePrinter R300c	499.90	–	–	–	yes	12 ppm	600 dpi
1924	PrintJet Duo	629.00	–	–	–	yes	14 ppm	1200 dpi
4200	CoolPhoto 15XT	379.98	12 MP	2 GB	22mm	–	–	–
4448	OfficeThing VIx	299.98	–	–	–	no	20 ppm	600 dpi
4711	Foo Pix FX13	249.00	8 MP	4 GB	28mm	–	–	–

Now consider

- internet stores that sell **lots** of different products,
- multi-tenancy systems (*e.g.*, SalesForce),
- data that inherently has a flexible structure (*e.g.*, an OPAC).

Limitations of the Relational Model

The relational model is **highly structured and regular**.

→ Simple, good to optimize, efficient to implement.

→ For many use cases, also the data is like that.

But there are use cases for which this model is **too rigid**.

→ Would need

- either **many null values** (as shown before) or
- **very complex schemas** (decomposed tables).

→ Both are inefficient and error-prone.

XML to the Rescue?

XML provides the desired flexibility, *e.g.*:

```
<products>
  <camera prodId='0815'>
    <name>SuperCam 2000</name>
    <price currency='EUR'>199.90</price>
    <resolution unit='MP'>12</resolution>
    <memory unit='MB'>512</memory>
    <lens>24mm</lens>
  </camera>
  <printer prodId='1734'>
    <name>ePrinter R300c</name>
    ...
  </printer>
  ...
</products>
```

XML—eXtensible Markup Language

XML is a **syntax**.

- → "angle brackets",
- → character encoding and escaping, ...

XML is also a **data model**.

- → Underlying model is ✎ .
  - All tags must be properly **nested**.
- → XML comes with a complete **type system**.
  - **XML Schema** further allows to restrict XML instances to a particular shape and to assign types to XML pieces.

The beauty of XML is that there's a whole **stack of XML technologies**:

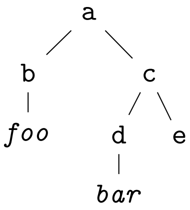
- → Parsing, character sets, etc. have all been taken care of.
- → Lots of tools available; clear interpretation across tools.

XML: Ordered, Unranked Trees

XML provides an encoding for **trees**.

```
<a>
  <b>foo</b>
  <c>
    <d>bar</d>
    <e/>
  </c>
</a>
```

$\rightsquigarrow$



Nodes in an XML tree are of different **node kinds**:

- **Element nodes** (here: `a`, `b`, ..., `e`) carry a **name** and may have any number of children (elements and/or text nodes).
- **Text nodes** (here: `foo`, `bar`) have an arbitrary text-only content; text nodes do not have children.

XML Node Kinds

In total, there are **seven node kinds**:

- Every XML document is encapsulated by a **document node**. Exactly one of its children must be an element node.
- We mentioned **element nodes** before. Elements may have elements, processing instructions, comments, and text nodes as children.
- Element nodes may own **attribute nodes**, which consist of a **name** and a **value**. Attribute names must be unique within one element.
- **Text nodes** contain character content.
- **Namespace nodes** contain prefix → URI bindings; they are mostly internal to XML processors.
- **Processing instruction nodes** are **target**/**content** pairs, represented as `<?target Content may be any string ?>`.
- **Comment nodes** contain text in (XML) comments: `<!-- This is a comment -->`.

Example

```
<?xml version='1.0' encoding='utf-8'?>
<!-- Example from www.w3.org -->
<?xml-stylesheet type='text/xsl'?>
<catalog xmlns='http://www.example.com/catalog'
         xmlns:xlink='http://www.w3.org/1999/xlink'
         xmlns:html='http://www.w3.org/1999/xhtml'>
  <tshirt code='T1534017' sizes='M L XL'
          xlink:href='http://example.com/0,,1655091,00.html'>
    <title>Staind: Been Awhile Tee Black (1-sided)</title>
    <description>
      <html:p>
        Lyrics from the hit song 'It's Been Awhile' are shown in
        white, beneath the large 'Flock &amp; Weld' Staind logo.
      </html:p>
    </description>
    <price currency='EUR'>25.00</price>
  </tshirt>
</catalog>
```

Notes

- Names in XML (*e.g.*, element or attribute names) are typically **QNames**:
  - → "qualified name"
  - → combination of a **prefix** (bound to a URI) and a local name, separated by `:`.
  - → **Namespaces** may help to mix different XML dialects (*e.g.*, an SVG graphic inside a HTML page).
- Use either double (`"`) or single (`'`) quotes for **attribute values**.
- There are exactly five pre-defined **character entities**: `&`, `'`, `>`, `<`, and `"`.
- It is perfectly legal to have both, text and element children, under the same parent (→ **"mixed content"**).

Navigating Through XML Trees

XPath is a language to select/address nodes in an XML document.

Idea:

- **Navigate** through the XML tree, like through a **file system**.

Example:

- `doc('cat.xml')/child::catalog/child::tshirt/descendant::html:p`

XPath is a subset of **XQuery**

→ Use an XQuery processor to experiment with XPath.

→ My favorite: BaseX (`http://www.basex.org/`)

Realization

XPath expression are built from

- **the path operator '/'**

$$e_1 \,/\, e_2$$
$$\equiv$$
$$\texttt{distinct-document-order}\,(\,\texttt{for . in}\ e_1\ \texttt{return}\ e_2\,)$$

- **step expressions** *axis*::*test*
  1. Start from the **context node** '.'.
  2. Navigate along *axis*.
  3. Return all nodes that meet the node test *test*.

The Path Operator /

- The / functions like a `map` operator.
- Input (left-hand side) of the / operator must be a **node sequence**.
- All evaluations of the right-hand expression are collected into a **single output sequence**:[16]
  - → Duplicates are removed based on **node identity**.
  - → Output is returned in **document order**.

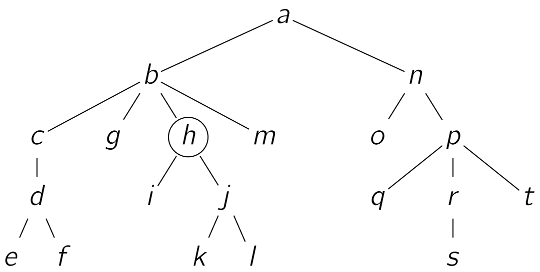
[16] Strictly speaking, duplicate removal and document ordering are only performed if the right-hand expression returns only nodes.

Step Expression *axis* : : *test*

- XPath defines **12 XPath axes**.
  - → Select nodes based on **XML tree structure**.
  - → See next slides for all axes.
- The **node test** *test* filters according to **name**, **node kind**, or **type**:
  - → `child::foo`: all child nodes with tag name `foo`
  - → `child::text()`: all children that are text nodes
  - → `ancestor::element(bar,shoeSize)`: all ancestor nodes with tag name `bar` and XML Schema type `shoeSize`
  - → `descendant::*`: all descendant nodes that have any name[17]

[17] Only elements and attributes have a name!

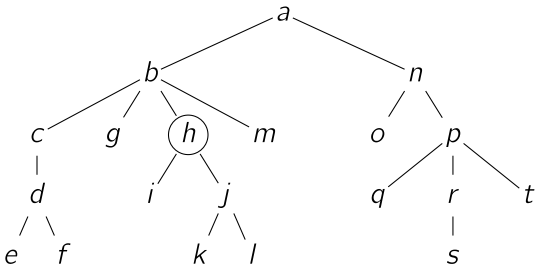
XPath Axes



Selected node sets, assuming context node `.` is bound to h:

- h/`child::*` $= \{i, j\}$
- h/`descendant::*` $= \{i, j, k, l\}$
- h/`self::*` $= \{h\}$
- h/`descendant-or-self::*` $= \{h, i, j, k, l\}$
- h/`following-sibling::*` $= \{m\}$
- h/`following::*` $= \{m, n, o, p, q, r, s, t\}$

XPath Axes (cont.)



Selected node sets, assuming context node `.` is bound to h:

- h/`parent::*` $= \{b\}$
- h/`ancestor::*` $= \{a, b\}$
- h/`ancestor-or-self::*` $= \{a, b, h\}$
- h/`preceding-sibling::*` $= \{c, g\}$
- h/`preceding::*` $= \{c, d, e, f, g\}$
- h/`attribute::*` $= \langle$attributes of $h\rangle$

Complete XPath Expressions

Use output of one '/' operator as input for the next.

⇝ "path expression"

Typical ways to **start** a path:

- Have initial context item **defined by query processor**
  - → *E.g.*, root of the given input document
- Use **built-in function** to retrieve document
  - → `doc (`*URL*`)`: XQuery built-in function
  - → `db:open (`*dbname*`,` *docname*`)`: BaseX: retrieve document *docname* from database *dbname*.
- A **rooted path expression** requires a context item, too, but starts from the document root associated with that context item.
  - → `/child::catalog/child::tshirt`
    (expands to '`root(self::node())/child::catalog/...`')

Predicate Operator [·]

Predicates can be used to **filter** an item sequence:

```
/descendant::tshirt[attribute::code = 'T1534017']
```

Semantics for *expr* [*p*] :

for . in *expr* return
  if (*p*) then . else ()

→ [·] binds **context item** '.' for evaluation of *p*.

→ Use **effective Boolean value** $ebv(\cdot)$ to decide:

- $ebv(\texttt{()})$ ⇀ false
- $ebv(\texttt{(}x\texttt{, ...)})$; x is a node ⇀ true
- $ebv(x)$; x is of type `xs:boolean` ⇀ x
- $ebv(x)$; x is a string ⇀ false if x is empty, true otherwise

Numeric Predicates

Predicates where *p* evaluates to a **singleton numeric value** are treated in a special way:

```
for . at $pos in expr return
  if (p = $pos) then . else ()
```

This is typically used for **positional predicates**...

→ `.../child::exam/child::date[2]`

...but can be used for very obscure queries, too:

→ `.../descendant::train[attribute::track + 3]`

→ Don't do this!

Predicate Semantics

1. [·] **binds stronger than /.**

   ✎ What does `/descendant::*/child::*[3]` return?

2. **Step expressions** return node sequences in **document order** ("forward axes") or **reverse document order** ("reverse axes").

   ✎ What about these expressions?

   - `descendant::a/preceding::*[3]`
   - `(descendant::a/preceding::*)[3]`
   - `descendant::a/(preceding::*)[3]`

XPath/XQuery Data Model

The basic XPath/XQuery type is the **item sequence**.

- All sequences are **flat**.
  - → Nested sequences are automatically flattened:

    `(42, ("foo", 7), "bar")` ⇾ `(42, "foo", 7, "bar")`

  - → A one-item sequence and that item are the same: `42` ≡ `(42)`
  - → Sequences are **ordered**. They may have **duplicates**.
- Items can be **nodes** or **atomic values**.
  - → Sequences can be **heterogeneous**.
  - → Valid types as specified by **XML Schema**.
  - → Implementations **may** use **static typing**.
- Construct sequences using '`,`' operator.

FLWOR Expressions

Use **FLWOR expressions** to work with sequences:

```
for $product in /child::catalog/child::*
where contains($product/attribute::sizes, "M")
order by $product/attribute::code
return $product/child::description
```

1. `for`/`let` **clause(s)**
2. `where` **clause** (optional)
3. `order by` **clause** (optional)
4. `return` **clause**

`for`/`let` Clauses

`for` $*var* `in` *expr*:

- **Iterate** over *expr*; create one binding of $*var* for each item in *expr*.
- Optional: bind a second variable to the **position** of $*var* in *expr*:

  `for` $*var* `at` $*pos* `in` *expr*

`let` $*var* `:=` *expr*:

- Create a **single binding** of $*var*: bind $*var* to the output of *expr*.

Multiple `for`/`let` clauses are allowed and can be **mixed**:

```
let $cat := /child::catalog
for $p in $cat/child::*
let $i := $cat/child::imprint
  ⋮
```

`for`/`let` Clauses; Tuple Stream

The `for`/`let` clauses produce a so-called **tuple stream**, *e.g.*,

```
for $x in (1,2)
let $y := ("foo",$x*4)
for $z in ("a","b")
   ⋮
```

Resulting tuple stream:

$$
\begin{array}{llll}
(& \langle\ \$x = 1,\ \ \$y = (\texttt{"foo"}, 4),\ \ \$z = \texttt{"a"}\ \rangle & \\
  & \langle\ \$x = 1,\ \ \$y = (\texttt{"foo"}, 4),\ \ \$z = \texttt{"b"}\ \rangle & \\
  & \langle\ \$x = 2,\ \ \$y = (\texttt{"foo"}, 8),\ \ \$z = \texttt{"a"}\ \rangle & \\
  & \langle\ \$x = 2,\ \ \$y = (\texttt{"foo"}, 8),\ \ \$z = \texttt{"b"}\ \rangle &)
\end{array}
$$

The tuple stream produced by the `for`/`let` clauses is

- **filtered** by the `where` clause
  - ⤳ effective Boolean value
- and **re-ordered** according to the `order by` clause.

Then, for each tuple in the stream, the `return` clause is evaluated and the result appended to the output.

XQuery is a **functional language**.

✎ What is the result of the following expression?

```
let $x := 1
for $i in (1,2,3,4)
  let $x := $x * 2
  return $x
```

Order

We've now seen two notions of **order**:

- **document order** and
- **sequence order**.

Both notions interact, but they are **not** the same. *E.g.*,

$\cdots$`/descendant::foo` $\leftrightarrow$

```
for $x in ···
  return $x/descendant::foo
```

Most operators have a precise semantics with respect to order.

$\rightarrow$ But that order can be **relaxed**.

$\rightarrow$ `unordered { · }`, `fn:unordered (·)`, default ordering mode

Types

XQuery is a **strongly typed language**.

But:

- There are many situations where data is implicitly type cast.
  - → *E.g.*, when using nodes in comparisons or arithmetic expr.
- The conversion **node** → **atomic value** is called **atomization**.
  - → If the node has an associated **typed value** (*e.g.*, as a consequence of schema validation), return that.
  - → Otherwise, return the node's **string value**, the **concatenation** of the contents of all descendant text nodes.
- To perform atomization explicitly, use the `fn:data(·)` built-in function.

More things about types:

- There are several operators that interact with XQuery's type system, *e.g.*, `cast as`, `instance of`, `typeswitch`, ...

Element Construction

XQuery contains operators to **construct new nodes**.

→ Useful, *e.g.*, to format output:

```
for $x in (1,2,3,4)
  return
    element number {
      attribute value { $x },
      element written-as {
        ("one", "two", "three", "four", "five")[$x]
      }
    }
```

✎ What is the output of this expression, written as XML?

Node Identity

Every node has a unique **identity**.

→ Test with operator `is`.

→ Two nodes may have same content and structure, but a different identity.

Node construction creates **new identities**.

→ Perform **deep copy** for nodes used in content expression.

→ ✎ What is the output of

```
let $foo := element foo { }
let $bar := element bar { $foo }
  return $foo is $bar/child::foo  ?
```

Node Identity (cont.)

Because of identity creation, node construction contains a **side effect**.

✎ Result of

```
let $a := element a { }
  return $a is $a
```
?

✎ What about

```
element a { } is element a { }
```
?

XQuery is "almost" a functional language, but does not allow variable substitution if the bound expression contains node construction.

More Syntax: Abbreviated XPath

Three abbreviations may be used in XPath:

1. The '*axis*`::`' part in a location step can be omitted and defaults to '`child::`', *e.g.*,

   `doc('cat.xml')/catalog/tshirt/descendant::html:p`

2. Two slashes '`//`' instead of a single slash '`/`' expand to '`/descendant-or-self::node()/`'.

   `doc('cat.xml')/catalog//price`
   expands to
   `doc('cat.xml')/catalog/descendant-or-self::node()/price`

3. An '`@`' sign instead of the '*axis*`::`' expands to '`attribute::`'.

   `doc('cat.xml')/catalog/tshirt/@code`
   expands to
   `doc('cat.xml')/catalog/tshirt/attribute::code`

Direct constructors are a more intuitive way to express node construction:

```
for $x in (1,2,3,4)
  return
    <number value='{ $x }'>
      <written-as>{
        ("one", "two", "three", "four", "five")[$x]
      }</written-as>
    </number>
```

→ Use **curly braces** {·} to “escape” back to XQuery.

Comments

Comments in XQuery have to be embraced by `(: ··· :)`.

`<!-- ··· -->` is the **direct comment constructor**.

→ Such "comments" will appear as comment nodes in the query result. In "XQuery mode" they likely lead to a syntax error.

Comments within direct constructors?

```
<foo>
  Would like to put some comment here.
  This is text content.
</foo>
```

SQL and XML

There are many ways how SQL and XML can interact.

E.g., **IBM DB2**:

- Special data type `XML`.
  - → Store XML documents as attribute values.

```
CREATE TABLE Employees (id      INT NOT NULL,
                        name    VARCHAR(30),
                        address XML);

INSERT INTO Employees (id, name, address)
  VALUES (42, 'John Doe',
          XMLPARSE (DOCUMENT '<address>'
                    || '<street>13 Main St</street>'
                    || '<zip>12345</zip>'
                    || '<city>Foo City</city>'
                    || '</address>'));
```

SQL and XML (cont.)

Access to `XML` content (syntactically) through **built-in functions**.

- `XMLEXISTS` (*XQueryExpr* `PASSING` *SQLExpr* `AS` *VarName*)
  - → Typically used as filter in `WHERE` clause.
  - → Pass attribute values of current row as variable to XQuery.

```
SELECT *
  FROM Employees
 WHERE name LIKE '%Doe'
   AND XMLEXISTS ('$a//pobox' PASSING address AS "a")
```

SQL and XML (cont.)

- `XMLQUERY` (*XQueryExpr* `PASSING` *SQLExpr* `AS` *VarName*)

  → Evaluate given query expression and return result as `XML`.

- `XMLCAST` (*XMLExpr* `AS` *DataType*)

  → Cast the result of the expression into an SQL data type.

Both are often used in combination:

```
SELECT id, name,
       XMLCAST (XMLQUERY ('$a//zip' PASSING address AS "a")
                AS integer) AS city
  FROM Employees
```

SQL and XML (cont.)

Conversely, XML data can be queried as relational tables, *e.g.*,

```
SELECT u."PO ID", u."Part #", u."Product Name", u."Quantity",
       u."Price", u."Order Date"
  FROM PurchasEorder p,
       XMLTABLE('$po/PurchaseOrder/item' PASSING p.POrder AS "po"
         COLUMNS "PO ID"        INTEGER      PATH '../@PoNum',
                 "Part #"       CHAR(10)     PATH 'partid',
                 "Product Name" VARCHAR(50)  PATH 'name',
                 "Quantity"     INTEGER      PATH 'quantity',
                 "Price"        DECIMAL(9,2) PATH 'price',
                 "Order Date"   DATE         PATH '../@OrderDate'
         ) AS u
 WHERE p.status = 'Unshipped'
```