Part IX

XML Processing
Limitations of the Relational Model

Suppose a shop sells **digital cameras**:

<table>
<thead>
<tr>
<th>ProdID</th>
<th>Name</th>
<th>Price</th>
<th>Resol.</th>
<th>Memory</th>
<th>Lens</th>
</tr>
</thead>
<tbody>
<tr>
<td>0815</td>
<td>SuperCam 2000</td>
<td>199.90</td>
<td>12 MP</td>
<td>512 MB</td>
<td>24mm</td>
</tr>
<tr>
<td>4200</td>
<td>CoolPhoto 15XT</td>
<td>379.98</td>
<td>12 MP</td>
<td>2 GB</td>
<td>22mm</td>
</tr>
<tr>
<td>4711</td>
<td>Foo Pix FX13</td>
<td>249.00</td>
<td>8 MP</td>
<td>4 GB</td>
<td>28mm</td>
</tr>
</tbody>
</table>

Or a shop might sell **printers**:

<table>
<thead>
<tr>
<th>ProdID</th>
<th>Name</th>
<th>Price</th>
<th>Color</th>
<th>Speed</th>
<th>Resol.</th>
</tr>
</thead>
<tbody>
<tr>
<td>1734</td>
<td>ePrinter R300c</td>
<td>499.90</td>
<td>yes</td>
<td>12 ppm</td>
<td>600 dpi</td>
</tr>
<tr>
<td>1924</td>
<td>PrintJet Duo</td>
<td>629.00</td>
<td>yes</td>
<td>14 ppm</td>
<td>1200 dpi</td>
</tr>
<tr>
<td>4448</td>
<td>OfficeThing Vlx</td>
<td>299.98</td>
<td>no</td>
<td>20 ppm</td>
<td>600 dpi</td>
</tr>
</tbody>
</table>
Limitations of the Relational Model

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

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

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.:

```xml
<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 an ordered, unranked tree.

■ 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 provides an encoding for **trees**.

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 -->`. 
<?xml version='1.0' encoding='utf-8'?><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>
- 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**: &amp;, &apos;, &gt;, &lt;, and &quot;.

- It is perfectly legal to have both, text and element children, under the same parent (→ “**mixed content**”).
**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/)
XPath expressions are built from

- **the path operator** ‘/’

\[
e_1 / e_2 \\
\equiv \\
\text{distinct-document-order}\left(\text{for } . \text{ in } e_1 \text{ return } e_2\right)
\]

- **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:
  - Duplicates are removed based on node identity.
  - Output is returned in document order.

\[^{13}\text{Strictly speaking, duplicate removal and document ordering are only performed if the right-hand expression returns only nodes.}\]
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

---

14 Only elements and attributes have a name!
Selected node sets, assuming context node $h$ 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 \( h \) is bound to \( h \):

- \( h / \text{parent::*} = \{b\} \)
- \( h / \text{ancestor::*} = \{a, b\} \)
- \( h / \text{ancestor-or-self::*} = \{a, b, h\} \)
- \( h / \text{preceding-sibling::*} = \{c, g\} \)
- \( h / \text{preceding::*} = \{c, d, e, f, g\} \)
- \( h / \text{attribute::*} = \{\text{attributes of } h\} \)
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/...’)

© Jens Teubner · Information Systems · Summer 2014
Predicates can be used to filter an item sequence:

\[/\text{descendant::tshirt}[\text{attribute::code = 'T1534017']\]

Semantics for \( expr[p] \):

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

\( \cdot \) binds context item ‘.’ for evaluation of \( p \).

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

- \( ebv(()) \rightarrow \text{false} \)
- \( ebv((x, \ldots)) ; x \) is a node \( \rightarrow \text{true} \)
- \( ebv(x) ; x \) is of type \( \text{xs:boolean} \) \( \rightarrow x \)
- \( ebv(x) ; x \) is a string \( \rightarrow \text{false} \) if \( x \) is empty, \text{true} otherwise
Predicates where $p$ evaluates to a **singleton numeric value** are treated in a special way:

\[
\text{for . at $\pos$ in } expr \text{ return }
\begin{align*}
\text{if ($p = \pos$) then . else (} & )
\end{align*}
\]

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!
1. [·] binds stronger than `/`.

What does `/descendant::*/*[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]`
The basic XPath/XQuery type is the item sequence.

- All sequences are flat.
  - Nested sequences are automatically flattened:
    
    \[(42, ("foo", 7), "bar") \rightarrow (42, "foo", 7, "bar")\]

  - A one-item sequence and that item are the same: \(42 \equiv (42)\)
  - 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.
Use **FLWOR expressions** to work with sequences:

```xml
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.,

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

Resulting tuple stream:

```plaintext
  ( ⟨ $x = 1$, $y = ("foo", 4)$, $z = "a"$ ⟩ )
  ⟨ $x = 1$, $y = ("foo", 4)$, $z = "b"$ ⟩
  ⟨ $x = 2$, $y = ("foo", 8)$, $z = "a"$ ⟩
  ⟨ $x = 2$, $y = ("foo", 8)$, $z = "b"$ ⟩
```
where/order by/return Clauses

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?

```xquery
let $x := 1
for $i in (1,2,3,4)
  let $x := $x * 2
return $x
```
We’ve now seen two notions of order:

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

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

\[
\cdots /\text{descendant::foo} \leftrightarrow \text{for } x \text{ in } \cdots \text{ return } x /\text{descendant::foo}
\]

Most operators have a precise semantics with respect to order.

- But that order can be **relaxed**.
- **unordered** {· }, **fn:unordered** (·), default ordering mode
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** $\rightarrow$ **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`, ...
XQuery contains operators to **construct new nodes**.

→ Useful, *e.g.*, to format output:

```xquery
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?
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

\[
\begin{align*}
    \text{let } \$\text{foo} & := \text{element } \text{foo} \{ \} \\
    \text{let } \$\text{bar} & := \text{element } \text{bar} \{ \$\text{foo} \} \\
    \text{return } \$\text{foo} \text{ is } \$\text{bar/child::foo} \ ?
\end{align*}
\]
Because of identity creation, node construction contains a side effect.

Result of

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

What about

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

XQuery is “almost” a functional language, but does not allow variable substitution if the bound expression contains node construction.
Three abbreviations may be used in XPath:

1. The `axis::` part in a location step can be omitted and defaults to `child::`, *e.g.*,
   
   \[
   \text{doc('cat.xml')/catalog/tshirt/descendant::html:p}
   \]

2. Two slashes `//` instead of a single slash `/' expand to `/descendant-or-self::node()/`.
   
   \[
   \text{doc('cat.xml')/catalog//price}
   \]
   
   expands to
   
   \[
   \text{doc('cat.xml')/catalog/descendant-or-self::node()/price}
   \]

3. An `@` sign instead of the `axis::` expands to `attribute::`.
   
   \[
   \text{doc('cat.xml')/catalog/tshirt/@code}
   \]
   
   expands to
   
   \[
   \text{doc('cat.xml')/catalog/tshirt/attribute::code}
   \]
Direct constructors are a more intuitive way to express node construction:

```xquery
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 in XQuery have to be embraced by \((: \cdots :)\).

\[
<!-- \cdots -->
\]

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?

```xml
<foo>
    \textit{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.

```sql
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>'));
```
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")
```
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
Conversely, XML data can be queried as relational tables, e.g.,

```sql
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'
```