# Chapter 6
# XML Query Languages

- XPath is not a query language:
  selects only sets of nodes
- additional functionality of query languages:
  - composition of tuples/structures from several nodes of a path
  - joins
  - dereferencing
    * via joins
    * via direct resolving of IDs (seen as values)
    * via dereferencing of ID attributes
  - aggregations
  - formatting and restructuring of results
  - operations on the order of nodes!



## XML Query Languages

Collected experiences from SQL, OQL, OEM/WSL/MSL, F-Logic and some more ...

- predecessors of XPath: XSL Patterns/XPointer/XQL (1998)
- XQL extended the early "basic form" to a query language
  - adding several constructs to the path expressions
  - increasingly complicated
  - still not sufficiently expressive
  - showed the limits and requirements
- XML-QL (1998): pattern-matching-based "extraction language"
  - not path-based, but XML-pattern/template-based binding of variables
  - semantics by a clause-construct
  - generation and structuring of the result by an XML pattern with variables

## XML Query Languages (cont'd)

- Quilt (2000): SQL-style extension of XPath
  - binding of variables by XPath expressions
  - nested loops by "for"-clauses
  - additional conditions in a "where"-clause
  - structuring of the result by a "return"-clause
- XQuery (2001): "official" version of Quilt
  - now integrated in the W3C-Draft XPath/XQuery



## 6.1 XQL

XQL (XML Query Language; 1998) is a simple query language based on early constructs of XPath:

- all XPath expressions that can be expressed without the use of "*axis*::" (cf. Slide 197 - axes have only been added later).
- text() was a function,
- function applications have been expressed by "!" at the end of the path expression: //country/name!text()

Further querying functionality was integrated syntactically into the path expressions.

## XQL: Boolean Operations and Set Operations

- $q_1$ union $q_2$, $q_1 \mid q_2$
- $q_1$ intersect $q_2$
- $q_1 \sim q_2$ (union, in case that both are non-empty)
- $q_1$ or $q_2$
- $q_1$ and $q_2$



## XQL: Return Operators (Projections on the path)

Operators that output the node that is addressed at the given position:

??: the complete node is added to the output structure (including attributes and subelements)

?: only the element “hull” is added to the output

- country/city[@isCountryCap]/name

```
<name>Berlin</name>
<name>Rome</name>
```

- country?/city[@isCountryCap]/name

```
<country> <name>Berlin</name> </country>
<country> <name>Rome</name> </country>
```

- country?[@car_code?]/city[@isCountryCap]/name

```
<country car_code="D"> <name>Berlin</name> </country>
<country car_code="I"> <name>Rome</name> </country>
```

- country?[@car_code?]/city?[@isCountryCap]/name!text()

```
<country car_code="D"> <city>Berlin</city> </country>
<country car_code="I"> <city>Rome</city> </country>
```

## XQL: GROUPING

- copy a part of the original document structure:
  $path_1 \{ path_2 \}$
- without grouping:
  country?[@car_code?]/city?/name!text()

```
<country car_code="D"> <city>Berlin</city> </country>
<country car_code="D"> <city>Hamburg</city> </country>
<country car_code="D"> <city>Munich</city> </country>
```

- with grouping:
  country?[@car_code?] {/city?/name!text()}

```
<country car_code="D">
  <city>Berlin</city>
  <city>Hamburg</city>
  <city>Munich</city>
</country>
```



## XQL: SEMIJOINS

- Semi-joins via subqueries in the condition:

$$\pi[A](r \rtimes s) \;, \quad A \subset \text{attr}(r)$$

Query: name of the continent where Germany is located:

```
/mondial/continent[@id =
                /mondial/country[@car_code="D"]
                          /encompassed/@continent]
               /name!text()
```

**Problems**

- full joins with join conditions not possible
- no restructuring/generation of answer structure

## XQL: Joins

- Asymmetric full joins expressed by *correlating variables* and “alternative”-construct: Filters may contain variable assignments of the form

  [$\$var$ := $expr$]

  that are the used in another condition

  [$expr'$ = $\$var$]

  //organization?[$s := @headq] {name?? | abbrev?? | member?? | //city[@id=$s]?? }

```
<organization>
 <name>European Union</name>
 <abbrev>EU</abbrev>
 <member type="member" country="GR F E A D I B L NL DK SF S IRL P GB"/>
 <member type="membership applicant"
    country="AL CZ H SK LV LT PL BG RO EW M CY"/>
 <city> <name>Brussels</name> ... </city>
</organization>
```



## XQL: Conclusion

- Ad-hoc-constructs (in different versions)
- insufficient restructuring functionality
  - tree structure of the input is in principle retained
- insufficient join functionality
- no clear semantics for the result format
- queries cannot be nested (cf. SQL, OQL: results are again relations); here is even no notion of a subquery
- one of the reasons: no variable concept
- implemented and used up to 2002 in the “Tamino” system of Software AG.

## 6.2 Query Languages: Requirements

Requirements on XML Query Languages [David Maier and W3C XML Query Requirements]

- closedness: output must be XML
- orthogonality/composability: everywhere where a set of XML elements is required, also a query is allowed.
- clean definition and nesting of operations: selection, extraction/projection, restructuring, combination/join, fusion of elements,
- applicable without presence of schema, but can use a schema,
- retaining the order of nodes,
- [queries should have an XML representation, especially, XML documents should be able to contain embedded queries]
- resolving of XPointer and XLink
- formal semantics: deriving structure of the result, equivalence and query containment



## 6.3 XML-QL

- `http://www.w3.org/TR/1998/NOTE-xml-ql-19980819/`
- simple, pattern-based XML query language:

  WHERE *xml-pattern* IN *url* CONSTRUCT *result*
- usage of variable bindings:
  *xml-pattern* contains variables that can be used in *result*,
- declarative,
- “relationally complete”, i.e., joins can be expressed.

Example:

```
WHERE <country car_code=$id>
        <name>$name</name>
      </country>
IN "http://www.../mondial.xml"
CONSTRUCT <country car_code=$id name=$name/>
```

## XML-QL: JOINS

Joins are expressed as a list of
WHERE ($expr_1$ IN $doc_1$, ... , $expr_n$ IN $doc_n$)-clauses:

- equijoin inside a document:

```
WHERE
  <country car_code=$c></country>
  IN mondial.xml,
  <organization abbrev=$org>
    <membership type=$type country=$c/>
  </organization>
  IN mondial.xml,
CONSTRUCT ...
```



## XML-QL: JOINS

- Joins that combine multiple documents:

```
WHERE
  <city name=$name1>
  IN http://www.../europe.xml,
  <city name=$name2>
  IN http://www.../america.xml,
  <connection from=$name1 to=$name2>
  IN http://www.../lufthansa.xml
CONSTRUCT
 <connection>
   <from continent="europe" city="name1"/>
   <to continent="america" city="name2"/>
 </connection>
```

## XML-QL: Nested Queries

WHERE *xml-pattern* IN *url* CONSTRUCT *result*

- *result* can contain nested WHERE ... IN ... CONSTRUCT statements.

## Further Functionality

- tag-variables: WHERE <$tag> ... </>
- regular path expressions: instead of XPath's "//", <*> ... </> is used.

## Data Model

- Graph-based: XML-tree with IDREF edges:

```
WHERE
  <country car_code=$cc>
    <capital><name>$name</name><population>$pop</population></capital>
  </country>
IN ... CONSTRUCT ...
```



## XML-QL: Conclusion

- clause-based high-level language
- selection and construction pattern-based (by binding variables in the patterns; similar to Logic Programming)
- join conditions: not in a WHERE clause, but implicitly expressed by *join variables* (like in Logic Programming)
- graph data model; no difference between tree edges and reference edges
- has been implemented
- used in different projects (e.g. MIX – Mediation in XML; UC San Diego 1999/2000)
  - allows for access and combination of different HTML/XML-sources in a query.

## 6.4 SQL, OQL etc.

- set-oriented (sets of tuples or objects) language
- implicit iteration over sets:

  SELECT ... FROM *relation-or-extent* c
- variable c ranges over *data items*
- join: use several such variables and correlate them
- WHERE and SELECT part: use these variables

Similar constructs for XML?

- variables range over sets of nodes
- ... sets of nodes can be addressed by XPath
- straightforward and intuitive:

```
for $c in //country
where $c.population > 1000000
return $c.name
```



## 6.5 Quilt and XQuery

- a Quilt is a “Flickenteppich” ...
- IBM, Software AG, INRIA; literature: WebDB2000-Workshop
- Structure similar to SQL/OQL: clause-based, functional language (arbitrary nesting of FLWR expressions allowed),
- Use of variables similar to SQL/OQL,
- based upon XPath (previously XQL/XSL Patterns) in the selection part and upon XML-QL (XML patterns) in the construction part:
- For Let Where Return-clauses

  for *variable* in *xpath-expr*     // from XQL/XPath and XML-QL
  let *additional_variable* := *xpath-expr*
  where *condition*
  return *xml-expr*     // from XML-QL
- has been moved into W3C’s “XML Query” in 2001 with only small changes.
- Remark: XQuery is case-sensitive.
  ALL KEYWORDS MUST BE WRITTEN WITH non-capital LETTERS!

## XQuery: Example

- for-clause: binding of variables (cf. SQL: FROM)
- where-clause: evaluation of conditions
- return-clause: generation of the result (cf. SQL: SELECT)

```
<result>
 { for $c in /mondial/country
   where $c/population > 10000000
   return <bigcountry>
            { $c/name }
            <area> { string($c/@area) } </area>
          </bigcountry>
 } </result>
```

[Filename: first-example.xq]

generates

<result><bigcountry><name>France</name><area>547030</area></bigcountry>
<bigcountry><name>Spain</name><area>504750</area></bigcountry>
:
<result>



## Aside: Tools – XQuery as a Database and Web Query Language

### XML Databases

- local repository of XML documents
- adding documents to the Database
- access only against locally stored documents
- presence of access paths like indexes etc
- manipulation of documents

Tool: a local eXist repository; see Web page
paths: /mondial//country/name or doc('/db/xmlcourse/mondial.xml')/mondial/country/name

### Queries against the Web

- querying the whole Web
- documents not locally stored; only on-the-fly-indexing possible
- access to remote documents by their url

Tool: saxonXQ; see Web page
paths: doc('*filename or url*')//country/name

## XQuery: Overview of Functionality

- for-clause: defines nested loops where each of the variables runs over the set of selected values
- variables in XPath expressions: bound in for/let (or by surrounding statements), they are used as starting points for paths and in conditions
- joins:
  - multiple variables in a for-clause:
    for *\$var*$_1$ in *doc*$_1$*/path*$_1$, ..., *\$var*$_n$ in *doc*$_n$*/path*$_n$
  - correlated definition of the variables in the for-clause:
    for *\$var*$_1$ in *doc*$_1$*/path*$_1$, *\$var*$_2$ in *\$var*$_1$*/path*$_2$, ...
- let-clause for definition of "constants":
  let *\$var := expr*
  binds *\$var* to the *whole result* of *expr* (in general, a node set).
- nested/iterated for-let-for-let-clauses allowed
- generation of nested and grouped structures:
  the return-clause may contain further FLWR-clauses (which can contain variables from the outer clause).



## Simplest XQuery Queries: XPath

- Each XPath query is also an XQuery query
  result: a sequence of nodes or literal values

```
doc('mondial.xml')//country/name
```

Note: different behavior when returning attribute nodes!

```
doc('mondial.xml')//country/@area
```

## XQuery: For-clause

for \$*var* = *xpath-expr*

- iterates over the result of *xpath-expr*

```
for $c in /mondial//country/name
return $c
```

[Filename: for-example.xq]

## XQUERY: RETURN-CLAUSE

Output of all statements must be XML.

- simple case: content of a variable

```
for $c in /mondial//country/name
return $c
```

- and generation of structured results (cf. OQL)

**Generation of Structures**

- literal XML
- computed element- and attribute constructors (later)

**Use of Computed Values/Structures**

- enclosed between "{" . . . "}"
- evaluation of variables and XPath expressions
- nested FLWR-clauses



## RETURN-CLAUSE: CONSTRUCTION OF RESULT ELEMENTS

- literal XML, values of variables and results of XPath expressions

```
<html><table>
<tr><th>Name</th><th>Area</th><th>Population</th></tr>
{ for $c in /mondial/country
  return
    <tr><td>{$c/name/text()}</td>
        <td>{string($c/@area)}</td>
        <td>{$c/population/text()}</td>
    </tr>
}
</table></html>
```

[Filename: table-example.xq]

returns one table row for each country.

# XQuery: For-Clause

## Multiple Variables in a For-Clause

- cartesian product
  (cf. FROM-clause in SQL)

```
for $c in /mondial//country,
    $o in /mondial//organization
where $c/@capital = $o/@headq
return
  <answer>
     <country>{$c/name/text()}</country>
     <organization>{$o/name/text()}</organization>
  </answer>
```

[Filename: cartesian-example.xq]



# XQuery: For-Clause

## Multiple Variables in a For-Clause

- "correlated" Join
  (cf. FROM-clause in Schema-SQL and OQL)
- subset of the cartesian product

```
for $c in /mondial/country,
    $p in $c/province
return
  <answer>
     <country>{$c/name/text()}</country>
     <prov>{$p/name/text()}</prov>
  </answer>
```

[Filename: correlated-join-example.xq]

## Return-Clause with Nested FLWR-Clause

- inner query used in the outer return-clause (cf. OQL)

```
for $c in /mondial/country
where $c/province
return
  <answer>
     {$c/name}
     { for $p in $c/province
       return
       <prov>{$p/name/text()}</prov>
     }
  </answer>
```

[Filename: nested-flwr-example.xq]

generates for each country that has provinces an <answer> element that contains a <name> element and a sequence of <prov> elements.



## Let-Clause

let $*var* := *xpath-expr*

- does not iterate over the result of *xpath-expr*
- but binds the complete result of *xpath-expr* as sequence of nodes to the variable:

```
for $c in /mondial/country
let $cities := $c//city/name
return
  <country>
     {$c/name}
     {$cities}
  </country>
```

[Filename: let-example.xq]

- useful for keeping intermediate results for reuse (often missed in SQL)

## WHERE-CLAUSE: CONDITIONS

Similar to XPath's conditions (same predicates etc):

- logical "and" and "or"
- "not(...)" as a boolean function
- Comparisons: "==" for node identity, "<<" and ">>" for document order, "follows" and "precedes"
- Quantifiers: where some|every $*var* in *expr* satisfies *condition*

```
for $c in /mondial/country
where some $city in $c//city satisfies $city/population > 1000000
return $c/name
```

```
for $c in /mondial/country
where every $city in $c//city satisfies $city/population > 1000000
return $c/name
```

[Filenames: some-example.xq and every-example.xq]



## CONDITIONAL EVALUATION AND ALTERNATIVES

- if-then: alternative choice of subelements
  if (*expr*) then *expr* else *expr*

```
<result>
 { for $c in /mondial/country
   return
     <country>
       {$c/name}
       {if ($c/province) then $c/province/city else $c/city}
     </country>
  }
</result>
```

[Filename: if-else-example.xq]

## Order of Result Set

XPath: the result is *always* returned in *document order*:

- purely navigational access:

  ```
  //country/city/name
  ```

- even when a backward axis is used during navigation, the nodes are enumerated in document order:

  ```
  //country[name='Germany']/province[last()]/preceding-sibling::*/name
  ```

  (backward axis is only relevant for context functions in immediate conditions)

- or when id-referencing is used:

  ```
  id(//organization/@headq)/name
  ```

  (note: cities are *not* ordered according to the order of the organizations!)

XQuery: result set is ordered according to for-clause:

```
for $c in //organization
return id($c/@headq)/name
```

let-clause: binds the result set according to the respective order.



## Sorting

- order by: *expr* order by (*expr* [ascending|descending])

  ```
  <result>
    { for $c in //country
      order by $c/name
      return $c/name }
  </result>
  ```

  [Filename: orderby-example.xq]

- note that the interpreter must be told whether the values should be regarded as numbers or as strings (default: alphanumerical)

  ```
  <result>
    { for $c in //country
      where $c/population > 0
      order by number($c/population)
      return $c/name    }
  </result>
  ```

  [Filename: orderby-num-example.xq]

## Grouping and Aggregation

- aggregate functions over result sets (avg, sum)
- bind variable with “for”-clause
- assign group with “let” (dependent on the current value in the for-clause) to a variable
- apply aggregate function to a nodeset

```
<result>
{ for $c in /mondial/country
  let $cities := $c//city
  where sum($cities/population) > 10000000
  return
    <answer>
       {$c/name}
       {sum($cities/population)}
    </answer>
}
</result>
```

[Filename: aggr-1-example.xq]



## Aggregation

- aggregation over result of a FLWR subquery
- bind (single) intermediate result by “let”

```
<result>
{ for $c in /mondial/country
  let $maxpop := max( for $citypop in $c//city/population/text()
                      return $citypop)
  return
    <answer>
       {$c/name}
       {$maxpop}
    </answer>
}
</result>
```

[Filename: aggr-2-example.xq]

## ATTRIBUTES IN THE RETURN-CLAUSE

- note that expressions the form “@bla” return *attribute nodes* - these are (AttrName,value)-pairs:

```
<result>
  {//country[name='Germany']/@car_code}
</result>
```

  generates <result car_code=“D”/>.

- attribute nodes are always added to the surrounding element.
- if only their value is needed, apply string().

```
for $c in /mondial/country
return
  <country>
    {$c/@area}
    {string($c/@car_code)}
  </country>
```

Result:

<country area=“28750”>AL</country>

<country area=“131940”>GR</country>

:

[Filename: attribute-example.xq]



## COMPUTED ELEMENT- AND ATTRIBUTE NAMES

- explicit constructors
  - element *expr attrs-and-content*
    the evaluation of *expr* yields the name of the element, the result of *attrs-and-content* is then inserted as attributes and content
    Note: content is a node sequence, separated by “,”
  - attribute *expr expr-value*
    the evaluation of *expr* yields the name of the attribute, *expr-value* yields its value.

```
<result>
 { for $c in doc('mondial.xml')//country
   where $c/encompassed
   return
   element { $c/@car_code }
           { attribute {$c/encompassed[1]/@continent} {"yes"},
             $c/name
           }
 } </result>
```

A result node:

<B europe=“yes”>
  <name>Belgium</name>
</B>

[Filename: computed-constructors-example.xq]

## Functions and Operators

- Recall Slide 211 for string() and name(), and Slide 208 for id().
- See “XML Query Functions and Operators” for predefined functions
- e.g. removal of duplicate values: distinct-values(...)

```
distinct-values(doc('...')//SPEAKER)
```

  How many speeches has each of the speakers in “Hamlet”?

```
for $a in distinct-values(doc('/db/xmlcourse/hamlet.xml')//SPEAKER)
let $n := count(//SPEECH[SPEAKER = $a])
order by $n descending
return
  <answer>
    {$a}
    {$n}
  </answer>
```

  [Filename: distinct-values.xq]



## User-Defined Functions

- User defined functions are declared in the prolog:
  declare function *func_name* ([*$var*$_1$, ..., *$var*$_n$]) [as *returnType*]
  {
  *expr that uses $var*$_1$, ..., *$var*$_n$
  }
- Parameters: $*var*$_i$ [as *paramType*], default for parameter and return types is item()* (i.e. a sequence of nodes, literals etc.),
- Any sequence type may be used for *paramType* and *returnType* (cf. XML Schema),
- Any XQuery expression is allowed in the function body.

## USER-DEFINED FUNCTIONS: EXAMPLES

- A function computing the population density for a given country:

```
declare function local:density ($name as xs:string) as element(density)
{
  for $c in doc('/db/xmlcourse/mondial.xml')//country[name=$name]
  let $density := if ($c/@area > 0) then $c/population div $c/@area else 0
  return <density>{$density}</density>
};
local:density('Germany')
```

[Filename: function-density.xq]

- Example for a recursive function:

```
declare function local:depth($e as node()) as xs:integer
{
   if (fn:empty($e/*)) then 1
   else fn:max(for $c in $e/* return local:depth($c)) + 1
};
local:depth(doc('/db/xmlcourse/mondial.xml'))
```

[Filename: function-depth.xq]



## USER-DEFINED FUNCTIONS: EXAMPLE

Ignoring the FLWR, XQuery can even be used as a common functional language:

- every (arithmetic + if) expression is a valid XQuery expression

```
(:call saxonXQ faculty.xq x=5 :)
declare variable $x external;
declare function local:faculty($n as xs:integer) as xs:integer
{  if ($n=1) then 1
   else $n* local:faculty($n - 1)
};
local:faculty($x)
```

[Filename: faculty.xq]

## Operations on Sets/Collections

Comparisons are instance-based: if one operand is a set, each value is compared

- ... as we have seen for XPath: country[.//city/name = "Cordoba"]/name
  country[.//city/population > 1000000]/name
- but somewhat surprising when using a "let"-view:

```
let $europcountries := //country[encompassed/@continent="europe"]
for $country in //country
where not ($country = $europcountries)
return $country/name
```

  [Filename: set-comparison-example.xq]

  outputs all names of non-european countries.
  (note that "where not ($country/name = $europcountries/name)" would have the same effect)

- selection from let-collections is also instance-based:

```
let $europcountries := //country[encompassed/@continent="europe"]
return $europcountries[@area>300000]/name
```

  [Filename: set-selection-example.xq]



## Flexibility

For each task, there is a multitude of possible solutions ...

Example: Uncorrelated Subqueries

Names of all countries that are larger than Germany:

- XPath:

```
//country[@area > number(//country[@car_code='D']/@area)]/name
```

- XQuery and SQL: uncorrelated subquery/semijoin

```
for $c in //country
where $c/@area >
 number(//country[@car_code='D']/@area)
return $c/name
```

```
SELECT c.name
FROM country c
WHERE c.area > (SELECT c2.area
                FROM country c2
                WHERE c2.code = 'D')
```

- binding the uncorrelated subquery to a variable:

```
let $germanyarea := number(//country[@car_code='D']/@area)
for $c in //country
where $c/@area > $germanyarea
return $c/name
```

## EXERCISES

... see Web.

**Exercise 6.1**
Determine the lowest mountain that is the highest mountain of the continent where it is located.

Solve the problem for the relational Mondial-DB in SQL, and for XML in XQuery. □



## XQUERY: CONCLUSION

### Design and Functionality

- combines the positive experiences of previous approaches
- avoids their drawbacks
- intuitively clear syntax and semantics
- declarative, orthogonal, functional style: every expression is a function on nodesets that also returns a nodeset
  - explicit, variable-based iteration: "for *var* in *expression*"
  - implicit iteration: "*collection*[*condition*]" or "*collection*/*path*"
- Theoretical background (see W3C XML Query Formal Semantics; datatypes of the XML Schema and XML Query Data Model)
  - for each expression (and thus also for its result), the formal type (according to the XML Schema datatypes) can be determined.
  - the type of each variable is determined in the same way.
  - formal, denotational semantics of queries:
    "what is the answer set of a given expression?"

## XQuery: Conclusion (Cont'd)

W3C XML Query Formal Semantics:

- XPath/XQuery is a functional language.
- is built from expressions, rather than statements. Every construct in the language (except for the XQuery query prolog) is an expression and expressions can be composed arbitrarily.
- The result of one expression can be used as the input to any other expression, as long as the type of the result of the former expression is compatible with the input type of the latter expression with which it is composed.
- Another characteristic of a functional language is that variables are always passed by value, and a variable's value cannot be modified through side effects.



## XQuery: Conclusion (Cont'd)

- Note: XQueryX provides a syntax that is formulated in XML

### Restrictions

- up to now no resolving of XLink/XPointer (see later)
- only a *query language*:
  decision of the W3C: first complete XQuery 1.0 as a query language and make it consistent with XML Schema and XML Query Data Model as a "Recommendation", and then start official thoughts about updates in XQuery 2.0.

## General Design Patterns for Database Query Languages

SQL, OQL, XML-QL, XQuery (and many others) use the same underlying principle:

- binding variables
- evaluating a condition
- generating a result (which is a set of data items of the underlying data model)

Note: XQL did not follow this idea → restricted expressiveness and clarity

... let's now have a look on one more XML query language

- the underlying principle is the same
- → everything else is "just syntax"!



# 6.6 Further (Academic) Query Languages

## XPathLog

- Prolog-/Datalog-style (May, DBPL and VLDB 2001; TPLP 2004)
- based on F-Logic
  - path syntax changed from *step.step.step* to *step*/*step*/*step*
  - same syntax for conditions as for F-Logic: "[...]" could be reused
  - F-Logic semantics (1989) closely related with XPath semantics
  - new: distinction between attributes/subelements
- Binding of variables at *arbitrary* positions of an expression
- joins as conjunction (as in Prolog/Datalog)

## XPathLog

- implicit resolving of multi-valued attributes
- implicit resolving of reference attributes

```
?- //country->C[name->N and @membership->O/name->A].
```

- access to signature/metadata

```
?- //country[name="Germany"]/M.
?- //country[name="Germany"]/@A.
```

- class membership and -hierarchy

```
?- C isa country[name->N]/M.
?- _C isa country/@A->_O, _O isa X.
?- country[@M=>C].      % from DTD
```



## XPathLog: Overview

- declarative language
- implicit iteration (fixpoint semantics)
- (equi-)join variables
- XPath-style semantics in rule heads for *generation* and *manipulation* of XML data
- first implementation of an update language for XML (Demo VLDB 2001)
- fixpoint semantics for execution
- can compute transitive closure etc.

## GENERAL DESIGN PRINCIPLES FOR DATABASE QUERY LANGUAGES

SQL, OQL, XML-QL, XQuery (and many others) use the same underlying principle:

- binding variables
- evaluating a condition
- generating a result (which is a set of data items of the underlying data model)

| | SQL/OQL | XML-QL | XQuery | XPathLog |
|---|---|---|---|---|
| variables: | 1-step-navig. | XML patterns | XPath navig. | XPath navig.+ |
| | SQL: flat data model | | | XPath Patterns |
| | OQL: + path navig. | | | |
| conditions: | WHERE clause | Patterns | XPath fragment | XPath filters |
| | | (Equality join conds) | (only non-join-conds) | (Join conds) |
| | | WHERE clause | WHERE Clause | separate conjuncts |
| | | (comparisons+joins) | (all) | (comparisons+joins) |

- the underlying Logic Programming fixpoint semantics enables XPathLog to compute the transitive closure
- ... but it does not allow for syntactically nested statements



## FURTHER (ACADEMIC) QUERY LANGUAGES

- XML-GL (Comai, Politecnico Milano, 1999): graphical “language”
- Lorel-XML (Stanford Univ., 1999): OQL-style language, migration of Lorel
- YATL-XML (Cluet, INRIA, 2000): term-based language, migration of YATL
- Lixto/Elog (Gottlob, TU Wien, 2001): graphical tool for data extraction from the Web, Datalog-based internals
- Xcerpt, XChange (Bry et al, LMU München, 2002): term- and unification-based language

... many different approaches to the same goal (mainly in Europe).
Overview in (May, TPLP 2004).

# Chapter 7
# Manipulating XML Data

- XML data in files:
  - usually no changes (except manually or by scripts)
  - transformations XML $\rightarrow$ HTML etc: XSLT
- XML data in application systems
  - inside the application programming language; mostly by the DOM-API
  - no special data manipulation language necessary (cf. OQL)?
- different proposals
  - pre-XQuery commercial area:
    - XMLDB: XUpdate (1999)
    - eXcelon (2000; XUL as extension of XSLT)
  - academic area:
    - “Updating XML” (Halevy et al, SIGMOD 2001) as an extension to XQuery
    - XPathLog (May, VLDB 2001): Prolog-style query- and manipulation language



## EXTENDING XQUERY WITH UPDATES – CONCEPTS

In the meantime consensus about which operations is reached. Syntax is still open.

- always wrt. a context node
- base operations:
  - delete *node*
  - rename *node* as *name*
  - insert *node/nodes* before|after|into *node*
- combined operations:
  - replace *node* with *node*
  - move *node* before|after|into *node*

## 7.1 XML:DB Initiative's XUpdate

- XML:DB Initiative founded in late 1999
  Goal: interface for storing XML in databases
- Low-level API (Java etc., using DOM + XPath ...)
- an update concept: XUpdate
- Implementation:
  dbXML Core XML Database released as Open Source software in Sept. 2000
  transferred to the Apache Software Foundation ("Xindice")
- `http://xmldb-org.sourceforge.net/`
  (inactive?)
- The XML:DB database API is implemented in several systems:
  eXist, X-Hive, Tamino, XML:DB Lexus, ...

... but here we are mainly interested in XUpdate ...
(note that XUpdate (1999) is not related with XQuery (2001))



## XML:DB XUPDATE

Situation in 1999: XML, XPath, XSLT [see later], low-level APIs

- Requirement: "The XML Update specification MUST be an XML element"
  i.e., the language is itself in XML syntax (like XSLT and XML Schema)
- XUpdate: a very basic description of update operations:
  - which node (elements, attributes)
  - which operation (delete, update value, append/insert to contents)
  - new value (in case of update/append/insert)

Basic structure:

```
<xu:modifications xmlns:xu= "http://www.xmldb.org/xupdate">
  <xu:operation select= "xpath-expression">
    contents (e.g. new value)
  </xu:operation>
</xu:modifications>
```

... submit such an element as a kind of a "message" to the DB and get the update.

## XUpdate: Example

```
<?xml version="1.0"?>
<xu:modifications version="1.0" xmlns:xu="http://www.xmldb.org/xupdate">
  <xu:append select="/mondial/country[name='Germany']">
    <xu:element name='localname'>Deutschland</xu:element>
  </xu:append>
</xu:modifications>
```

[Filename: XUpdate/append.xu]

Calling eXist with (see `client.sh -h`)

`/bin/gen_client.sh -u` *user* `-P` *password* `-c /db/may -f mondial.xml -X append.xu`

executes the update.

- `select= "`*xpath*`"` is the same as in XSLT (see later), XML Schema etc. – a widely used concept in the XML world.
  (if multiple nodes are addressed, each one is modified)
- `<xu:element>` constructor is the same as in XSLT (1998) and later in XQuery's RETURN clause
- analogously insert-before and insert-after.



## XUpdate: Examples (Cont'd)

```
<?xml version="1.0"?>
<xu:modifications version="1.0" xmlns:xu="http://www.xmldb.org/xupdate">
  <xu:remove select="/mondial/country[name='Germany']/localname"/>
</xu:modifications>
```

[Filename: XUpdate/remove.xu]

```
<?xml version="1.0"?>
<xu:modifications version="1.0" xmlns:xu="http://www.xmldb.org/xupdate">
  <xu:update select="/mondial/country[name='Germany']/population">
     80000000
  </xu:update>
</xu:modifications>
```

[Filename: XUpdate/update.xu]

## XUpdate: Examples (Cont'd)

- get the new value from the database:

```
<?xml version="1.0"?>
<xu:modifications version="1.0" xmlns:xu="http://www.xmldb.org/xupdate">
  <xu:update select="/mondial/country[name='Germany']/population/text()">
     <xu:value-of select="/mondial/country[name='Germany']/@area"/>
  </xu:update>
</xu:modifications>
```

[Filename: XUpdate/update-select.xu]

note: the inner `select` cannot depend on the current node.

```
<?xml version="1.0"?>
<xu:modifications version="1.0" xmlns:xu="http://www.xmldb.org/xupdate">
  <xu:variable name="bla"
      select="/mondial/country[name='Germany']/gdp_total/text()"/>
  <xu:update select="/mondial/country[name='Germany']/population/text()">
     <xu:value-of select="$bla"/>
  </xu:update>
</xu:modifications>
```

[Filename: XUpdate/update-variable.xu]



# XUPDATE: CONCLUSION AND COMMENTS

- XML-syntax of the language strongly influenced by XSLT (1998)
  - elements as commands
  - `select="..."` selects nodes to which the command is applied
  - use of variables `select="$variable"` as in XSLT, and later also in XQuery
  - element/command contents specifies what is to be done
  - element generation by literal XML (also in XSLT and later XQuery)
- only very simple functionality
  - no way to compute the inner value,
  - no iteration etc.
- same time: combination with XSLT and XUpdate to XUL (XML Update Language/Updategrams [Excelon]):
  XSLT program structures + XUpdate operations, applied to "current node" of XSLT.

## 7.2 XQuery with Updates (2001)

- extend a *declarative query language* with updates
- based on *variable bindings*
- SQL: FROM-WHERE for selecting nodes ...
  ... that are then modified.
- XQuery: FOR-LET-WHERE for selecting nodes ...
  ... that are then modified.
- update instead of the return-clause (cf. SQL: UPDATE vs. SELECT)
- XQuery expected to reach recommendation state in 2005 ... as a query language still without updates.



## XQuery with Updates – Example: QuiP

- QuiP: the 2001/02 XQuery prototype of Software AG [DA Lehti 2001], later integrated into the Tamino system (before: XQL).
- calling `quip filename.xq > bla.xml` writes the modified XML to a file.

```
update
for $c in document("twocountries.xml")//country
let $area := string($c/@area)
delete $c/@area
insert <area>{$area}</area> after $c/name
rename $c//city[@id=$c/@capital] as capital
replace $c/@car_code with
    attribute code {concat($c/name/text(), ":", string($c/@car_code))}
replace $c/population/text() with
   $c/population/text() * (1 + $c/population_growth div 100)
insert "biggest city" into
   $c//city[population = max(for $citypop in $c//city/population/text()
                             return int($citypop))]
```

[Filename: XQuery/update.quip]