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/Lorel, 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
  - W3C Working Draft XQuery first version from 15 February 2001
  - XQuery 1.0: W3C Recommendation since 23.1.2007
  - `http://www.w3.org/TR/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 199 - 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 or q_2
- q_1 and q_2
- 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)

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>
```


XPATH: SEMIJOINS ARE POSSIBLE

- Semi-joins via subqueries in the condition:

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

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 then 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>
```

Equivalent:

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

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 clean variable concept
  (the variable concept of XPath 2.0 motivated by logical quantifiers is clean, but also leads to complex expressions)
- 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 [never completed since XLink is rarely used]
- 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 $xml\text{-}pattern$ IN url CONSTRUCT $result$
- usage of variable bindings:
  $xml\text{-}pattern$ 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>
    <members 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 Recall basic concepts from 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/text()
```


6.5 XQuery

- First proposed as “Quilt” by IBM, Software AG, INRIA at WebDB2000-Workshop
- a Quilt is a “Flickenteppich” ...
- 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 $xpath\text{-}expr$ // from XQL/XPath and XML-QL
  let $additional\_variable$:= $xpath\text{-}expr$
  where $condition$
  return $xml\text{-}expr$ // 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/@area > 500000
   return <bigcountry>
             { $c/name }
             <area> { string($c/@area) } </area>
          </bigcountry>
 } </result>
```

[Filename: XQuery/first-example.xq]

generates

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


SOME NOTES ON THE MONDIAL DATABASE

- Cities can have more than one name (“München”,“Munich”,. . .).
  (this is important for data integration when combining Mondial with external sources)
  Access the first one with //city[name='Munich']/name[1].
- Countries, provinces and cities usually have several
  <population year="...">*value*</population>
  subelements.
  These subelements are in temporal order, the last one ist the newest.
  Access the last one with //country[name='Germany']/population[last()].
  Aside: note that this is much more longwinded with SQL:

```
select country, year, population
from countrypops p1
where year = ( select max(year)
               from countrypops p2
               where p2.country=p1.country )
```

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

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

path expressions: doc('*filename or url*')//country/name

6.5.1 XQuery: the basis

- 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 $doc_1/path_1, \ldots, \$var_n$ 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 $\$var_1/path_2, \ldots$
- let-clause for definition of "constants":
  let *$var := expr*
  binds *$var* to the *whole result* of *expr* (in general, a sequence of nodes).
- nested/iterated for-let-for-let-clauses allowed
- where clause: conditions on the variables defined so far
- generation of nested structures:
  the return-clause may contain further FLWR-clauses (which can use 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 $x in /mondial//country/name
return $x
```

[Filename: XQuery/for-example.xq]

XQUERY: RETURN-CLAUSE

Output of all statements must be XML.

- simple case: content of a variable

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

- 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[last()]/text()}</td>
    </tr>
}
</table></html>
```

[Filename: XQuery/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[1]/text()}</country>
       <organization>{$o/name/text()}</organization>
    </answer>
```

[Filename: XQuery/cartesian-example.xq]

- compare where clause with equivalent
  where $c/id(@capital) is $o/id(@headq)
  on node level (“=” would also be correct here, taking the string value of the nodes).

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: XQuery/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: XQuery/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 := $c//city/name[1]      (: first name of each city :)
return
  <country>
     {$c/name}
     {$cities}
  </country>
```

[Filename: XQuery/let-example.xq]

- useful for keeping intermediate results for reuse (often missed in SQL)
  - Note: if a fragment with idref-attributes is created, these cannot be dereferenced during later use (the created fragment is not part of the “main” tree)

WHERE-CLAUSE: CONDITIONS

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

- logical “and” and “or”
- “not(...)” as a boolean function
- Comparisons: “is” 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[last()] > 1000000
return $c/name
```

```
for $c in /mondial/country
where every $city in $c//city satisfies $city/population[last()] > 1000000
return $c/name
```

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

USE CASE: JOIN BETWEEN DIFFERENT DOCUMENTS

- doc(...) function to access files (local or from the Web)
- here: join by a subquery

```
for $c in doc(concat('http://www.dbis.informatik.uni-goettingen.de',
                     '/Mondial/mondial-europe.xml'))/mondial/country
where some $l in doc('hamlet.xml')//LINE
      satisfies contains($l, $c/name[1])
return
  <country>
    {$c/name}
  </country>
```

[Filename: XQuery/join-web-documents.xq]

- "contains" requires (node,node) as arguments, does not accept (node, node+). Exercise: compare not only with name[1] but with all of them.

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: XQuery/attribute-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.

Ordering

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

```
for $c in //country
order by $c/name[1]
return $c/name[1]
```

[Filename: XQuery/orderby-example.xq]

- note that the interpreter must be told whether the values should be regarded as numbers or as strings (default: alphanumerical) (see below).
- Note: ordering cannot only be used for finally presenting the output (like in SQL), but also to create an ordered sequence to bind it with “let” and do something with it:

```
let $ordered := (for $c in //country
                 order by number($c/@area) descending
                 return $c/name[1] )
return $ordered[position() = (1 to 4)]
```

[Filename: XQuery/bigcountries.xq]

GROUPING (BY "LET") AND AGGREGATION

- aggregate functions over result sets (avg, sum, min, max, count).
- bind group-by variable(s) with "for"-clause,
- assign group with "let" (dependent on the current value in the for-clause) to a variable,
- apply aggregate functions to the nodesets bound by the let.

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

[Filename: XQuery/aggr-1-example.xq]

AGGREGATION

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

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

[Filename: XQuery/aggr-2-example.xq]

Conditional Evaluation and Alternatives

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

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

[Filename: XQuery/if-else-example.xq]

- same as SQL's CASE ... WHEN ...
- since XQuery 3.0: "switch/case+/default" expression.

6.5.2 XQuery: Further Functionality

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.

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

[Filename: XQuery/computed-constructors-example.xq]

```
<B europe="yes">
  <name>Belgium</name>
</B>
```

... for *one* continent for each country

Computed Element- and Attribute Names: Another example

- the same, iterating over *all* "encompassed" elements of a country:

```
for $c in doc('mondial.xml')//country
order by count($c/encompassed) descending
return
element { $c/@car_code }
        { for $e in $c/encompassed
          return attribute {string($e/@continent)} {$e/@percentage},
          $c/name
        }
```

[Filename: XQuery/computed-constructors-example2.xq]

- returns e.g.

```
<R europe="25" asia="75"> <name>Russia</name> </R>
```

- note: `text { $car_code }` can be used to create text nodes, e.g. from a string bound to a variable (when operating on sequences, a sequence of text nodes is different from a sequence of strings. E.g., union and except are only applicable on sequences of nodes).

Handling Duplicates

- recall from XPath: results (and intermediate results) of XPath expressions are *node sets* in document order
  ⇒ for $x in *xpath-expr*, let $y := *xpath-expr*
  always results in a set (i.e., duplicates removed)
- recall Slide 219 for 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]

- takes only the string values (⇒ no further navigation applicable)

Handling Duplicates in XQuery(cont'd)

- FLWR expressions (e.g., for $c in ... return $c) do *not* eliminate duplicates automatically
- for $o in //organization return $o/id(@headq)
  returns duplicates
- distinct-values(for $o in //organization return $o/id(@headq))
  returns only the string values
- so it must be done programmatically (often, specific for the given problem: iterate over the target set and do the test in a subquery) – cf. SQL:
  `select * from <table-of-entity-tuples> where <condition>`
- or by a generic function – see Slide 300

OPERATING WITH SEQUENCES

Comparisons are existentially quantified and instance-based: if one operand is a sequence, each value is compared, and if one value satisfies the condition, the whole filter is satisfied:

- ... as we have seen for XPath: country[.//city/name = "Cordoba"]/name
  country[.//city/population > 1000000]/name
- the same holds when comparing with a sequence bound to a variable by a "let"-view:

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

  [Filename: XQuery/seq-comparison-example.xq]

  outputs all names of non-european countries.

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

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

  [Filename: XQuery/seq-selection-example.xq]

Operations on Nodes and Node Sequences

- "=" compares the string-values of nodes, not "correct" if node identity has to be checked
- "is" compares node identity:

```
for $c in //country,
    $o in //organization
where $c/id(@capital) is $o/id(@headq)
return <pair country='{$c/@car_code}' org='{$o/abbrev}'/>
```

[Filename: XQuery/node-comparison.xq]

- "is" is not allowed for node sequences:

```
let $caps := //country/id(@capital)
for $hq in //organization/id(@headq)
where $hq is $caps       (: not allowed :)
return $hq/name
```

[Filename: XQuery/nodes-comparison-example-1.xq]

⇒ Error: "A sequence of more than one item is not allowed as the second operand of "is" "

Operations on Nodes and Node Sequences

- explicit iteration via some:

```
let $caps := //country/id(@capital)
for $hq in //organization/id(@headq)
where some $cap in $caps satisfies $hq is $cap
return $hq/name
```

[Filename: XQuery/nodes-comparison-example-2.xq]

- index-of(*sequence(atomic type)*,*item*) → integer

```
let $caps := //country/id(@capital)
for $hq in //organization/id(@headq)
where index-of($caps,$hq)      (: checks if $caps contains $hq :)
return $hq/name
```

[Filename: XQuery/nodes-comparison-example-3.xq]

- ... or "where $caps intersect $hq", or even shorter:

```
let $caps := //country/id(@capital)
let $hqs := //organization/id(@headq)
return ($hqs intersect $caps)/name
```

[File: XQuery/nodes-comparison-example-4.xq]

- "intersect", "union", and "except" are only allowed on sequences of *nodes*, not on sequences of literals (like e.g. strings).

GROUP BY (SINCE XQUERY 3.0)

- At first sight like SQL . . .
- recall SQL: only group-by variables and aggregate function applications over the group can be selected.
- group by variable: uses/binds the *string value* of the path expression!
- XQuery: all non-group-variables are bound to the sequence of all values of that variable (can be used later for aggregate functions).

```
for $c in //city
group by $cc := $c/ancestor::country/name[1]
return
<answergroup country="{$cc}">
  { $c/name[1] }
  <sumpop> {sum($c/population[last()])} </sumpop>
</answergroup>
```

[Filename: XQuery/group-by-simple.xq]

Group By (cont'd) 3.0)

consider again

```
for $c in //city
group by $cc := $c/ancestor::country/name[1]
return
<answergroup country="{$cc}">
  { $c/name[1] }
  <sumpop> {sum($c/population[last()])} </sumpop>
</answergroup>
```

[Filename: XQuery/group-by-simple.xq]

- one <answergroup> per country (in arbitrary order if the countries),
- note: $c is *implicitly* reassigned after the group-by, holding the sequence of all city elements in this country;
- $c/name[1] results in the sequence of all their (first) name subelements,
- sumpop holds then the sum of their newest population counts.
- similar (and preferable?): for $c in //country let $cities := $c//city ... (cf. Slide 282)

Group By – Danger!

- Recall: all non-group-variables are bound to the sequence of all values of that variable (can be used later for aggregate functions).

```
let $x := 3, $y := ("a","b")
for $c in //city
group by $cc := $c/ancestor::country/name[1]
return
<answergrp country="{$cc}"> {$x} {$y}</answergrp>
```

[Filename: XQuery/group-by-ugly.xq]

- after the group-by, $x and $y are a *sequences* that consist of as many 3's and so many a-b-a-b-... as the country has cities!
- if a non-group-by-variable is not used later, the optimizer will (hopefully) never instantiate its sequence, but
- if such a non-group-by-variable $x is used later, one should consider to access it by $x[1],
- if such a non-group-by-variable $y contains a sequence before, there is no simple way to "keep it" (the duplicated sequence is flattened)!

⇒ in general, prefer to use
for *groupingvar* in ... let *groupvar* := *correlatedpath* ... (cf. Slide 282)

Group By – a statistical use case

- Sometimes,
for *groupingvar* in ... let *groupvar* := *correlatedpath* ... is not more intuitive:

```
for $c in //city[population]
group by $pp := round($c/population[last()] div 10000)
where count($c) > 0     (: another where, not "having" :)
order by $pp descending
return
<answergrp pp="{$pp * 10000} to {$pp * 10000 + 10000}" count="{count($c)}">
  {$c/name[1]} </answergrp>
```

[Filename: XQuery/group-by-popgrps.xq]

```
let $allpps := distinct-values(
     for $p in //city/population[last()] return round(/$p div 10000))
for $pp in $allpps
let $c := //city[round(population[last()] div 10000) = $pp]
where count($c) > 0
order by $pp descending
return
<answergrp pp="{$pp * 10000} to {$pp * 10000 + 10000}" count="{count($c)}">
  {$c/name[1]} </answergrp>
```

[Filename: XQuery/group-by-popgrps2.xq]

FUNCTIONS AND OPERATORS

XPath and XQuery Standard Operators

- Recall Slide 213 for string() and name(), and Slide 210 for id().
- Mathematical functions have been added as builtins in XQuery 3.0:
  - `declare namespace math="http://www.w3.org/2005/xpath-functions/math";`
  - use math:sqrt, math:sin, ...
- See "W3C XML Query Functions and Operators 3.0" for predefined functions:
  `https://www.w3.org/TR/xpath-functions-30/`
  - functions/operators using strings (cf. Slide 217);
    for splitting and re-constructing IDREFS attributes, especially,
    `tokenize`(*string-to-be-tokenized*, *separator*) $\rightarrow$ xs:string* and
    `string-join`(xs:string*, *separator*) $\rightarrow$ xs:string are useful.
  - functions/operators using numbers,
  - functions/operators using date and time (cf. Slide 309),
  - functions/operators on sequences of nodes.

6.5.3 User-Defined Functions/Functional Programming

- definition and reuse of functions simplifies daily work with XML in general,
- up to a full-fledged functional programming language (like Lisp or Haskell)

Syntax

- 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)?
{                                          (: returns zero or one element :)
  for $c in doc('mondial.xml')//country[name=$name]
  let $density :=
      if ($c/@area > 0) then $c/population[last()] div $c/@area else 0
  return <density>{$density}</density>
};
local:density('Germany')
```

[Filename: XQuery/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(/)
```

[Filename: XQuery/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
- example: the "factorial" function (german: "Fakultät"): $n! := \begin{cases} 1 & \text{if } n = 1 \\ n \cdot (n-1)! & \text{if } n > 1 \end{cases}$

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

[Filename: XQuery/factorial.xq]

- saxon from command line: for strings as parameters use
  `saxonXQ ?foo='xs:string("blabla")' anyprogram.xq`

User-Defined Function: a Useful Example

Remove duplicates from a node set (taken from the example Section from W3C XPath/XQuery Functions and Operators):

```
declare function distinct-nodes-stable ($arg as node()*) as node()*
{
   for $a at $apos in $arg
   let $before_a := fn:subsequence($arg, 1, $apos - 1)
   where every $ba in $before_a satisfies not($ba is $a)
   return $a
}
```


XQuery Exception Handling (since XQuery 3.0/2014)

- like in Java: try-catch, embedded as functional programming syntax.
- example: catch any exception and report it (as result!)

```
declare function local:trycatchtest($x as xs:integer) {
  try { 1000 div $x }
  catch * {
    $err:code, $err:description, $err:value,
    if ($err:module) then (" module: " || $err:module) else (),
    "(line " || $err:line-number || ", col " || $err:column-number || ")"
  }};
local:trycatchtest(0)
```

[Filename: XQuery/trycatch.xq]

- example: catch div by zero (which is exception err:FOAR0001)

```
declare function local:catchdivby0($x as xs:integer) {
  try { 1000 div $x }
  catch err:FOAR0001 { 12345 }};
local:catchdivby0(0)
```

[Filename: XQuery/catchdivby0.xq]

- List of all error codes: <https://www.w3.org/2005/xqt-errors/>

Web Access requires try-catch

- many Web pages are not available or not valid XML

⇒ raises exceptions (different types)

```
declare function local:getwebpage($x as xs:string) {
  try { doc("http://www." || $x || ".de") }
  catch * { " not found or invalid as XML: " || $x}
};
for $n in //province[name='Niedersachsen']//city/name
return local:getwebpage($n)
```

[Filename: XQuery/catchwebaccess.xq]

Throwing an exception: fn:error

- call fn:error;
- this will not create a result, but throw an exception:

```
declare function local:throwexc($x as xs:integer) {
  try { if ($x = 0)
        then fn:error(fn:QName('http://www.w3.org/2005/xqt-errors',
                               'err:FOER0000'))
        else if ($x = 1)
        then fn:error(fn:QName('http://www.semwebtech.org/foo/', 'err:foo'),
                      "I don't like that!")
        else $x
      }
  catch * {
    $err:code, $err:description, $err:value,
    if ($err:module) then (" module: " || $err:module) else (),
    "(line " || $err:line-number || ", col " || $err:column-number || ")"
  }};
local:throwexc(0)
```

[Filename: XQuery/throw.xq]

QUERYING FOR CONTEXT POSITIONS BY POSITION()

- position() is always evaluated *for a given node* wrt. *a given context (sequence)*
- in Mondial, languages, religions, and ethnic groups in each country are ordered by percentage.
  What is the ranking of the french language in each country?

```
let $lgs := //country[@car_code='CH']/language
return
<result allpositions = '{$lgs/position()}'
  everypos='{for $x in $lgs return $x/position()}'
  frenchpos1 = '{$lgs[text()='French']/position()}'
  frenchpos2 = '{for $x in $lgs return if ($x/text()="French")
                    then $x/position() else ()}'            />
```

[Filename: XQuery/queryposwrong.xq]

- allpositions is '1 2 3 4', everypos is '1 1 1 1', the others are both '1'.

⇒ the "context" is always only the individual node bound to $x, not the sequence bound to $l

- see next slide for the workaround design in XQuery.

Querying for context positions by position() - "Positional variables"

- The XQuery language introduced the so-called "positional variables":

  for $x at $i in *expr*

  $i is then bound to the position of $x in the context given by *expr*:

```
for $c in //country[language/text()='French']
let $lgs := $c/language
return <result country='{$c/@car_code}'
               rank= '{ for $x at $pos in $lgs
                        return if ($x/text()="French") then $pos else ()}'/>
```

[Filename: XQuery/frenchpos.xq]

- As shown on Slide 393, this is solved better in XSLT.
- an XQuery workaround is shown on the next slide.

FINDING IN ANOTHER SEQUENCE — THE INDEX-OF() FUNCTION

- index-of(*sequence of atomic type*,*value*) → integer
- for querying the position of a *(atomic!) value* in a sequence of *(atomic!) values*,
- the sequence can be obtained directly representing the "context", or can be computed in any other way.

```
for $c in //country[language/text()='French']
let $l := $c/language
return <result country='{$c/@car_code}'
               rank= '{ index-of($l/text(), "French") }'/>
```

[Filename: XQuery/frenchpos2.xq]

FUNCTIONAL PROGRAMMING (INCOMPLETE LIST)

- "simple" map operator ("!"): syntactic sugar for
  sequence ! *expr* → for $x in *sequence* return *expr*($x)
  //country[name='Germany']//city ! concat(name[1], " has pop ", population[last()])
- handling sequences for rewriting iteration into recursion: fn:head, fn:tail
  e.g. sum(*sequence*) = head(sequence) + sum(tail(sequence)),

```
declare function local:sum($x as xs:integer*) as xs:integer {
    if (not(fn:empty($x))) then fn:head($x) + local:sum(fn:tail($x)) else 0 };
local:sum((1,2,3,4,5))
```

[Filename: XQuery/sum.xq]

- Note: function references and inline functions require the saxon EE version
  - Lisp-style: fold-left, fold-right
  - Inline functions, Dynamic function calls

```
let $f := function ($x) { 2 * $x },
for $c in (1 to 10)
return $f($c)
```

[Filename: XQuery/inlinefct.xq]

```
declare namespace math="http://www.w3.org/2005/xpath-functions/math";
let $f := math:sqrt#1     (: $f unary function math:sqrt :)
for $c in (1 to 10) return $f($c)
```

[Filename: XQuery/dynfctcall.xq]

6.5.4 Miscellaneous

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. □

SPECIALIZED DATATYPES FOR TIME ETC.

The datatypes specified by XML Schema are used in XPath/XQuery (and in XSLT)

- Syntax: constructors like xs:dateTime('*syntactical representation*')
- syntactical representations:
  - xs:dateTime: *yyyy-mm-dd* T*hh:mm:ss*[.*xx*][{+|-}*hh:mm*]
  - xs:date: *yyyy-mm-dd* and xs:time: *hh:mm:ss* [{+|-}*hh:mm*]
  - xs:duration: P [*n* Y][*n* M][*n* D][T[*n* H][*n* M][*n* [.*n*] S]], where *n* can be any natural number
  - xs:dayTimeDuration, xs:yearMonthDuration: restrictions of xs:duration.

```
let $x := xs:dateTime('2009-08-01T13:51:20.99'),
    $y := xs:date('2008-12-31'),
    $t1 := xs:time('12:50:00+01:00'),     (: timezone +1 = Frankfurt :)
    $t2 := xs:time('15:35:00.50-05:00')   (: timezone -5 = New York  :)
return <e const="{$x}" diff="{$t2 - $t1}" d="{$y + xs:yearMonthDuration("P1Y2M")}"
 sum1="{xs:time("11:12:00") + xs:dayTimeDuration("PT1H75M")}"
 sum2="{xs:dateTime("2009-01-10T11:12:00") + xs:dayTimeDuration("P3DT26H40M")}"/>
```

[Filename: XQuery/datetime-test.xq]

- resulting diff = "PT8H45M0.5S" (an xs:duration), sum1 = "13:27:00" (an xs:time), sum2= "2009-01-14T13:52:00" (an xs:dateTime), d= "2010-02-28" (an xs:date)

Actual Usage of xs:date etc.

... is often simple (recall that any text contents in an XML file is PCDATA or CDATA):

```
<country car_code="B">
   <indep_date>1830-10-04</indep_date>
</country>
```

```
for $c in //country[indep_date < '1900-01-01']
return concat($c/name, $c/indep_date)
```

[Filename: XQuery/simple-date-example.xq]

note: explicit `[indep_date < xs:date('1900-01-01')]` would be more explicit, but string comparison has the same result for the standard textual representation ('0' $<$ '1').

- functions current-date() and current-time(),
- extraction functions like days-from-duration, minutes-from-dateTime etc. (see W3C XQuery and XPath Functions and Operators)

```
for $c in //country[indep_date]
order by xs:date($c/indep_date) ascending
return
  concat($c/name, " is ", days-from-duration(current-date() - xs:date($c/indep_date)),
         " days old.")
```

[Filename: XQuery/current-date-example.xq]

Durations and Time Differences

- difference between two times is a duration
- add $x to a time: $x must be a duration (SQL: DATE + number means days)
  - Note: in XPath's built-in functions there is also nothing like SQL's ADD_MONTHS(date, number) function.
- duration serialization format: like "PnDTnHnMdecS", n positive integers, m positive decimal, any n, mX might be missing, dec might be a decimal with a decimal point. Negative durations: "-P..."
- functions adjust-dateTime-to-timezone($dt, $timezone) and adjust-time-to-timezone($dt, $timezone);
  $timezone must hold a dayTimeDuration!
  not allowed: adjust-time-to-timezone(xs:time("12:00:00"), 1)
  adjust-time-to-timezone(xs:time("12:00:00"), xs:dayTimeDuration("PT1H30M")) $\rightarrow$ 12:00:00+01:30
- Mondial has airport timezones (gmtOffset elements) as numbers, which might be negative, and even non-integer:

```
<airport iatacode="YYT" city="cty-Canada-Saint-Johns" country="CDN">
  <name>St Johns Intl</name>  <latitude>47.61861</latitude> <longitude>-52.751945</longitu
  <gmtOffset>-3.5</gmtOffset> <located_on island="island-Newfoundland"/>  </airport>
```

Durations and Time Differences: example

```
let $flights :=  (   (: all times in localtime :)
   <flight nr="LH123" from="FRA" to="LIS" departure="12:30:00" arrival="14:40:00"/>,
   <flight nr="LH124" from="FRA" to="JFK" departure="12:40:00" arrival="15:10:00"/>,
   <flight nr="LH125" from="FRA" to="YYT" departure="12:50:00" arrival="14:20:00"/> )
for $fl in $flights[position() = (1 to 2)]     (: not for YYT: offset 3.5h :)
let $dur1 := xs:time($fl/@arrival) - xs:time($fl/@departure),
    $deptoffset := //airport[@iatacode = $fl/@from]/gmtOffset,
    $arroffset := //airport[@iatacode = $fl/@to]/gmtOffset,
    $deptOffsetDur := if ($deptoffset >= 0) then
            xs:dayTimeDuration(concat('PT', $deptoffset, 'H'))
       else xs:dayTimeDuration(concat('-PT', abs($deptoffset), 'H')),
    $arrOffsetDur :=   if ($arroffset >= 0) then
            xs:dayTimeDuration(concat('PODT', $arroffset, 'H'))
       else xs:dayTimeDuration(concat('-PODT', abs($arroffset), 'H')),
    $depteff := adjust-time-to-timezone(xs:time($fl/@departure),$deptOffsetDur),
    $arreff := adjust-time-to-timezone(xs:time($fl/@arrival),$arrOffsetDur),
    $dureff := $arreff - $depteff (: XQuery knows that these are xs:time :)
return
<res dur1="{$dur1}" dDur="{$deptOffsetDur}" aDur="{$arrOffsetDur}"
     depteff="{$depteff}" arreff="{$arreff}" dureff="{$dureff}"/>
```

[Filename: XQuery/flights-duration-example1.xq]

Converting numbers to durations

- there seems to be no built-in function to turn a number into a duration.
  (number can mean hours/minutes/seconds)
  So, let's write one.

```
declare function local:numToDur($x as xs:decimal, $unit as xs:string)
   as xs:duration {
   xs:dayTimeDuration(local:ntd($x, $unit, false())) };

declare function local:ntd($x as xs:decimal, $unit as xs:string,
   $rec as xs:boolean?) as xs:string {
 let $sign := if($x<0) then "-" else "",
     $pre := if (not($rec)) then "P" else ""
 return if ($unit = "H") then
   concat($sign, $pre, "T", floor(abs($x)),
                           "H", local:ntd((abs($x)-floor(abs($x)))*60,"M", true()))
   else if ($unit = "M") then
     concat($sign, $pre,      floor(abs($x)),
                           "M", local:ntd((abs($x)-floor(abs($x)))*60,"S", true()))
     else if ($unit = "S") then concat(abs($x), "S") else ()
};
```

```
let $flights :=  (
   <flight nr="LH123" from="FRA" to="LIS" departure="12:30:00" arrival="14:40:00"/>,
   <flight nr="LH124" from="FRA" to="JFK" departure="12:40:00" arrival="15:10:00"/>,
   <flight nr="LH125" from="FRA" to="YYT" departure="12:50:00" arrival="14:20:00"/> )
   (: all times in localtime :)
for $fl in $flights
let $deptOffsetDur := local:numToDur(//airport[@iatacode = $fl/@from]/gmtOffset,"H"),
    $arrOffsetDur := local:numToDur(//airport[@iatacode = $fl/@to]/gmtOffset,"H"),
    $depteff := adjust-time-to-timezone(xs:time($fl/@departure),$deptOffsetDur),
    $arreff := adjust-time-to-timezone(xs:time($fl/@arrival),$arrOffsetDur),
    $dureff := $arreff - $depteff (: XQuery knows that these are xs:time :)
return
<res dOffsetDur="{$deptOffsetDur}" aOffsetDur="{$arrOffsetDur}"
     depteff="{$depteff}" arreff="{$arreff}" dureff="{$dureff}"/>
```

[Filename: XQuery/flights-duration-example2.xq]

```
<res dOffsetDur="PT1H" aOffsetDur="PT0S" depteff="12:30:00+01:00"
                       arreff="14:40:00Z" dureff="PT3H10M"/>
<res dOffsetDur="PT1H" aOffsetDur="-PT5H" depteff="12:40:00+01:00"
                       arreff="15:10:00-05:00" dureff="PT8H30M"/>
<res dOffsetDur="PT1H" aOffsetDur="-PT3H30M" depteff="12:50:00+01:00"
                       arreff="14:20:00-03:30" dureff="PT6H"/>
```

The third flight takes 6hrs from +1 to the -3.5h-timezone

HANDLING XML PROCESSING INSTRUCTIONS IN XQUERY

- PIs are also children of their parent element in the XML tree (cf. Slide 186):
- can be detected with nodetest “processing-instruction()”,
- access: use name() and string()

```
let $d :=
 doc("https://www.dbis.informatik.uni-goettingen.de/Teaching/SSD/XML-DTD/html-php.xml")
for $n in $d//processing-instruction()
return ($n, " --- ",  name($n), " --- ", string($n))
```

[Filename: XQuery/pis.xq]

PRACTICAL HINTS: OUTPUT

When creating output, most XQuery engines generate the XML declaration, and output "<" and ">" in text nodes as "<" and ">", respectively.

Output Options

- W3C XSLT and XQuery Serialization 3.1:
  declare namespace output = "http://www.w3.org/2010/xslt-xquery-serialization";
  declare option output:method "xml";
  declare option output:encoding "UTF-8"; (or e.g. "ISO-8859-1")
  declare option output:indent "yes";
- for saxon, look at its own extensions
  declare namespace saxon="http://saxon.sf.net/";
  declare option saxon:output "saxon:indent-spaces=1";
- In case it is intended to generate e.g. LaTeX, SQL input statements, RDF/Turtle/N3 or whatever plain text output, the XML declaration and the "<"/">"-conversion must be avoided.
  W3C: declare option output:method "text";
  saxon: declare option saxon:output "method=text";

Add Doctype Declaration to the Output (saxon)

- XQuery engines output only the XML structure itself
- how to add the <!DOCTYPE mondial SYSTEM "mondial.dtd"> preamble?
  With saxon, use
  declare namespace saxon="http://saxon.sf.net/";
  declare option saxon:output "doctype-system=mondial.dtd";

Generation of Multiple Instances (and for Debugging)

- fn:put(*node,uri*) (belongs to XQuery Update Facility; requires saxonEE)
- side-effect during executing the program,
- also possible with dynamically computed filenames:
  Generate a file for each country:

```
(: saxonXQEE -update:on \!indent=yes redirected-output-countries.xq  :)
  declare namespace saxon="http://saxon.sf.net/";
  declare option saxon:output "doctype-system=mondial.dtd";
  declare option saxon:output "saxon:indent-spaces=1";
 for $c in doc("file:mondial.xml")//country
 return put($c, concat("tmp/",$c/@car_code, ".xml"))
```

[Filename: XQuery/redirected-output-countries.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
```


6.5.5 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 nodes/sequences of nodes that also returns a sequence of nodes
  - 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 (like SQL, LISP, Haskell, ...).
- 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 consider updates in XQuery 2.0.
- started as an "XML Query Language" . . .
- . . . XQuery 3.0 became a full-fledged functional programming language.

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 $\Rightarrow$ restricted expressiveness and clarity

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

- the underlying principle is the same

$\Rightarrow$ 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

- declarative language
- (equi-)join variables

```
?- //country->_C[name->N and @capital->_X[name->XN],
   //organization->_O[@abbrev->A and @headq->_X].
N/"Belgium", A/"EU" X/"Brussels"
N/"Austria", A/"OSCE" X/"Vienna"
   :           :          :
```

- XPath-style semantics in rule heads for *generation* and *manipulation* of XML data
- first implementation of an update language for XML (Demo VLDB 2001)
  generation of XML in rule heads:
  C[density -> D] :- C isa country[population -> P; @area -> A], D is P div A.
- fixpoint semantics for Datalog-style rules
  ⇒ possible to compute transitive closure etc.

```
R[tr_flows_into -> S]   :-  R isa river, R/to[@water -> S], S isa water.
R[tr_flows_into ->S]    :-  L isa lake, L/to[@water -> R[tr_flows_into -> S]].
R[tr_flows_into ->S]    :-  R isa river, R/to[@water -> W[tr_flows_into -> S]].
```

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. SQL: flat data model OQL: + path navig.	XML patterns	XPath navig.	XPath navig.+ XPath patterns
conditions:	WHERE clause	Patterns (equality join conds) WHERE clause (comparisons+joins)	XPath fragment (only tree-style join-conds) WHERE clause (all)	XPath filters (join conds) separate conjuncts (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

- 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"), abandoned in 2011.
- The XML:DB database API is implemented in several systems:
  eXist (`http://exist-db.org`; open-source), Tamino (Software AG)

... 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 message (e.g., HTTP POST) 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, 2002 bought by Progress Software]): XSLT program structures + XUpdate operations, applied to "current node" of XSLT.

7.2 XQuery with Updates

- 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.
- execute update instead of the return-clause (cf. SQL: UPDATE/DELETE vs. SELECT)?

ASIDE: XQUERY WITH UPDATES – AN EARLY PROPOSAL

- QuiP: the 2001/02 XQuery prototype of Software AG, later integrated into the Tamino system (before: XQL).
- calling `quip filename.xq > bla.xml` wrote 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]

XQuery with Updates

- XQuery reached recommendation state in 2007 ... as a query language still without updates.
- "XQuery Update Facility", first W3C Working Draft has been published 27 January 2006; `http://www.w3.org/TR/xqupdate`
- XQuery Update Facility 1.0, W3C Recommendation 17 March 2011; `https://www.w3.org/TR/xquery-update-10/`

Update Expressions

delete (node|nodes) *TargetExpr*
insert node|nodes *SourceExpr* (([as (first | last)] into) | after | before) *TargetExpr*
rename node *TargetExpr* as *Expr* *// Expr must result in a qname*
replace [value of] node *TargetExpr* with *Expr*

- Syntax: "node"/"nodes" does not matter.
- for insert, rename, and replace, *TargetExpr* must evaluate to a single node.

Simple Updates in XQuery

- not implemented in saxonHE, only in (commercial) saxonEE (download 30 days eval)
- changes the XML file,
- in an XML database system, the internal database would be changed.
- call `saxonXQEE -update:on update.xq`
  changes the document (if given as local file; not via http:. . .)
  (use `-update:discard` just for testing syntax without changing the document)
- call `saxonXQEE -update:on -tree:linked` *`filename`* for (required for transformation commands, cf. Slide 341)

```
(: saxonXQEE -update:on \!indent=yes simpleinsert.xq  :)
insert node <bla/>
after fn:doc("mondial.xml")//country[name='Albania']
```

[Filename: XQuery/simpleinsert.xq]

```
delete nodes fn:doc("mondial.xml")//bla
```

[Filename: XQuery/simpledelete.xq]

```
 replace value of node fn:doc("mondial.xml")//country[name='Germany']/name
         with 'Deutschland'
```

[Filename: XQuery/simpleupdate.xq]

XQuery with Updates: Embedding into FLWR Clauses

- can be embedded into the return part of FLWR clauses when variables should be used (note: the return must still be there!)
- update statements return nothing
- sequence for executing multiple updates
- no queries allowed in the same sequence

```
for $c in  fn:doc("mondial.xml")//country
return (
  insert node <bla/> after $c/name,
  delete node $c/@area
  (: ,  $c/@car_code          not allowed :)
)
```

[Filename: XQuery/updateseq.xq] (changes the mondial.xml file)

XQuery with Updates – Copy-Modify Expressions

1. copy: assign variable(s) to fragments if the tree,
2. modify: update things bound to the copy-variable(s),
3. return: return something generated from the copied+updated variables.

- not an update! – (often called “hypothetical update”)
- rather belongs to the querying/constructing functionality; sometimes shorter than reconstructing as a new fragment

Syntax:

copy $*VarName* := *Expr* (, $*VarName* := *Expr*)*
modify *UpdateExprSeq*
return *Expr*

```
(:  saxonXQEE -update:on \!indent=yes -tree:linked copymodify.xq :)
for $c in fn:doc("mondial.xml")//country[@area > 1000000]
return
  copy $cc := $c
  modify delete nodes $cc/(province|city)
  return $cc
```

[Filename: XQuery/copymodify.xq] / / (outputs the result doc.)

XQuery with Updates – how to perform complex tasks

(see example next slide)

- embed into a copy-modify-return-statement
  - ⇒ cannot write file directly
  - ⇒ pipe output
- works differently than expected when knowing SQL updates:
  - collect *"pending updates"* as a sequence in the modify statement
  - use "for ... return *updates*" to collect them

XQuery with Updates – code example next slide

- for *each* country: update most recent population (+1 year, using population growth rate),
- turn Bavaria into an independent country (element), and delete it as province (semantically absolutely incomplete update)

XQuery with Updates – complex task example

```
(:  saxonXQEE -update:on \!indent=yes -tree:linked updmondial.xq > bla.xml :)
let $mm := fn:doc("mondial.xml")
return
  copy $m := $mm
  modify
  ( for $c in $m//country[population_growth and population]
    return ( replace value of node $c/population[last()]
               with $c/population[last()] * (1 + $c/population_growth div 100),
             replace value of node $c/population[last()]/@year
               with $c/population[last()]/@year + 1 ),
    insert node
      ( copy $bav := $m//country[name='Germany']/province[name='Bayern']
        modify ( rename node $bav as 'country',
                 insert node attribute area{string($bav/area)} into $bav,
                 delete node $bav/area )
        return $bav )
    after $m//country[name='Germany'],
    delete node $m//country[name='Germany']/province[name='Bayern']
  )
  return $m
```

[Filename: XQuery/updmondial.xq]

Updating Functions

- declare updating function ... { returns *update statements* };

```
(:  saxonXQEE -update:on \!indent=yes -tree:linked updfct.xq :)
declare updating function local:growpop($tree, $n as xs:string) {
  let $c := $tree//country[name=$n]
  return
  ( replace value of node $c/population[last()]
      with round($c/population[last()] * (1 + $c/population_growth div 100)),
    replace value of node $c/population[last()]/@year
      with $c/population[last()]/@year + 1 ) };

let $m := doc("mondial.xml")
return (local:growpop($m, "Germany"),
        local:growpop($m, "Austria") )
```

[Filename: XQuery/updfct.xq] updates mondial.xml

XQuery with Updates 3.1 – Transform Expressions NOT YET SUPPORTED

- apply a transformation to a single node
- shorthand for copy-modify-return

```
(:  saxonXQEE -update:on \!indent=yes -tree:linked transform.xq :)
let $x := fn:doc("mondial.xml")
return
  $x transform with
  { modify delete nodes $x//country }
```

[Filename: XQuery/transform.xq]