Chapter 7: XQuery

References:

- Scott Boag, Don Chamberlin, Mary F. Fernńndez, Daniela Florescu, Jonathan Robie, Jérôme Siméon (Eds.):
  XQuery 1.0: An XML Query Language.
  W3C Recommendation 23 January 2007.
  [http://www.w3.org/TR/xquery/]
- Wolfgang Lehner, Harald Schöning:
  XQuery. Grundlagen und fortgeschrittene Methoden.
  dpunkt.verlag, 2004, ISBN 3-89864-266-6, 290 Seiten.
  [http://www.xquery-buch.de/]
- Howard Katz (Ed.), Don Chamberlin, Denise Draper, Mary Fernández, Michael Kay, Jonathan Robie, Michael Rys, Jérôme Siméon, Jim Tivy, Philip Wadler:
  XQuery from the Experts. A Guide to the W3C XML Query Language.
  Pearson Education Inc., 2004, ISBN 0-321-18060-7, 484 pages.
- Jim Melton, Stephen Buxton:
  Querying XML: XQuery, XPath, and SQL/XML in Context.
  Morgan Kaufmann/Elsevier, 2006, ISBN 1-55860-711-0, 815 pages.
- Rudolf Jansen:
  XQuery: Eine praxisorientierte Einführung.
  Software & Support Verlag GmbH, 2004, ISBN 3-935042-65-5, 167 Seiten.

Objectives

After completing this chapter, you should be able to:

- read and understand queries in XQuery.
- write queries to XML data in XQuery.
- enumerate and explain the clauses of FLWOR expressions.
- explain the use of direct and computed constructors.
- compare XQuery with SQL.

Overview

Introduction (1)

- One can view XML as a data model, and every data model should have a query language.
- XPath permits to
  - ◇ select nodes in a given XML document, and to
  - ◇ compute values from the data in a document,

  but it does not permit to generate new nodes.
- An XML query language should be able to produce new XML documents as result.

  I.e. to transform given XML documents into documents that are differently structured, contain only a subset of the data, or additional derived data.

Introduction (2)

- Joins are very limited in XPath (semijoins) or can be formulated only procedurally (with `for`-loops).

  Note that many powerful constructs such as `for`-loops got into XPath only in Version 2.0 during the development of XQuery. Although today, XPath is sufficient for quite a lot of queries, the original XPath was much more restricted.

- One cannot sort in XPath.

  Sometimes, although the result set can be determined by a simple Path expression, one must use the more advanced `FLWOR`-expression (an XQuery construct, see below) only for the purpose of sorting.

- Of course, a good XML query language must be at least as powerful as SQL.

Introduction (3)

- The XML format is a common interface to a lot of different data sources (documents, relational databases, object repositories).

  The data might be physically stored as XML, or might be only viable as XML via a middleware.

- An XML query language permits to combine data from different sources.

  This integrating function of an XML query language is natural and important. While also SQL can be used in distributed databases, and there exist relational interfaces to non-relational data, this is much more vendor-dependent (and typically expensive).

History (1)

- In December 1998, the W3C organized a workshop about query languages for XML.

  [http://www.w3.org/TandS/QL/QL98/].

- There was a lot of research about query languages for semi-structured data models and XML in particular (e.g., Lorel, XQL, XML-QL, YATL, Quilt).

  See, e.g.: XML Query Langauges, Experiences and Exemplars.
  [http://www.w3.org/1999/09/ql/docs/xquery.html]

- The XML Query Working Group started in 1999 the work on a W3C standard XML query language.

  [http://www.w3.org/XML/Query]

History (2)

- XPath 1.0 and XSLT 1.0 became a W3C Recommendation in November 1999.
- While it seemed natural that XPath-like expressions should be used also in XQuery, the XQuery committee had quite different ideas for the exact details of syntax and semantics.

  XPath 1.0 came from the document processing community, not from the database community. But having two similar languages that differed in important details was obviously not good. This lead to difficult negotiations and ultimately the development of XPath 2.0.

History (3)

- XML Schema became a W3C recommendation in May 2001.
- Steps of the XQuery standardization:
  - First Working Draft: February 15, 2001.
  - Last Call Working Draft: November 12, 2003

    The last call period ended on February 15, 2004. Several updates were published afterwards.
  - W3C Candidate Recommendation: Nov. 3, 2005

    Update: June 8, 2006
  - W3C Proposed Recommendation: Nov. 21, 2006
  - W3C Recommendation: January 23, 2007

History (4)

- The following eight documents were developed together:
  - XQuery 1.0
  - XQueryX 1.0: XML Syntax for XQuery 1.0
  - XPath 2.0
  - XSLT 2.0
  - XQuery 1.0/XPath 2.0 Data Model
  - XQuery 1.0/XPath 2.0 Formal Semantics
  - XQuery 1.0/XPath 2.0 Functions and Operators
  - XSLT 2.0/XQuery 1.0 Serialization

History (5)

- Related documents (Working Group Notes):
  - XML Query Use Cases

    [http://www.w3.org/TR/xquery-use-cases/]
  - XML Query (XQuery) Requirements

    [http://www.w3.org/TR/xquery-requirements/]
- Extensions (Candidate Rec. May/Aug. 2008):
  - XQuery and XPath Full Text 1.0

    [http://www.w3.org/TR/xpath-full-text-10/]
  - XQuery Update Facility 1.0

    [http://www.w3.org/TR/xquery-update-10/]

XQuery vs. XSLT

- The two languages have overlapping, but not identical goals.
- XSLT was developed by the document processing community. Main use: rendering XML documents.

  Although it can also be used for selecting and restructuring data.
- XQuery is a database language.
- Databases store very many / very large documents: indexes and query optimization are important (the data does not fit completely into main memory).

  The data is also more regularly structured (in most cases).

XQuery Implementations (1)

- IPSI XQ

  Written in Java. [http://sourceforge.net/projects/ipsi-xq]
  [http://www.ipsi.fraunhofer.de/oasys/projects/ipsi-xq/]

- AltovaXML

  The engine used in XMLSpy is free (contains validator: DTD/Schema, XSLT 1.0/2.0, XQuery). [http://www.altova.com/altovaxml.html]

- Galax

  Open source, from some authors/editors of the XQuery Specification.
  [http://www.galaxquery.org/]

- eXist (open source native XML database)

  [http://exist.sourceforge.net/]
  Online demo: [http://demo.exist-db.org/sandbox/sandbox.xql]

XQuery Implementations (2)

- X-HIVE

  Commercial XML-DBMS, Online demo evaluator.
  [http://support.x-hive.com/xquery/] (select first any demo, then "your own"). [http://support.x-hive.com/xquery/basicservlet?demo=demo0&xquery=xquery&todo=showframes]

- Saxon (from Michael Kay)

  M. Kay is editor of the XSLT 2.0 Spec. Basic version (without static type checking and XQuery→Java compiler) is open source. Supports XSLT 2.0, XPath 2.0, XQuery 1.0. [http://saxon.sourceforge.net/]

- Qizx/open (open source Java implementation)

  In Java. Limited version is free. [http://www.axyana.com/qizxopen/]
  Online demonstration:
  [http://www.xmlmind.com:8080/xqdemo/xquery.html]

Example Document (1)

STUDENTS			
<u>SID</u>	FIRST	LAST	EMAIL
101	Ann	Smith	···
102	Michael	Jones	(null)
103	Richard	Turner	···
104	Maria	Brown	···

RESULTS			
<u>SID</u>	<u>CAT</u>	<u>ENO</u>	POINTS
101	H	1	10
101	H	2	8
101	M	1	12
102	H	1	9
102	H	2	9
102	M	1	10
103	H	1	5
103	M	1	7

EXERCISES			
<u>CAT</u>	<u>ENO</u>	TOPIC	MAXPT
H	1	Rel. Algeb.	10
H	2	SQL	10
M	1	SQL	14

Example Document (2)

- Translation to XML with data values in elements:

```
<?xml version='1.0' encoding='ISO-8859-1'?>
<GRADES-DB>
  <STUDENTS>
    <STUDENT>
      <SID>101</SID>
      <FIRST>Ann</FIRST>
      <LAST>Smith</LAST>
    </STUDENT>
    ...
  </STUDENTS>
  ...
</GRADES-DB>
```

First Example (1)

- Print all results for Homework 1:

```
for $s in /GRADES-DB/STUDENTS/STUDENT,
    $r in /GRADES-DB/RESULTS/RESULT
where $s/SID = $r/SID and
      $r/CAT = 'H' and
      $r/ENO = 1
return <h1>{ $s/LAST, $r/POINTS }</h1>
```

- Result:

```
<h1><LAST>Smith</LAST><POINTS>10</POINTS></h1>
<h1><LAST>Jones</LAST><POINTS>9</POINTS></h1>
<h1><LAST>Turner</LAST><POINTS>5</POINTS></h1>
```


First Example (2)

- A characteristic feature of XQuery are "FLWOR-expressions" (pronounced "Flower-expressions").
- This name is derived from the keywords `for`, `let`, `where`, `order by`, `return`.

  They are written "FLWOR-expressions", and pronounced "Flower", although two characters are exchanged: `where` comes before `order by`.
- The `for`-clause corresponds to `FROM` in SQL: It generates a sequence of variable bindings.
- The `return`-clause corresponds to `SELECT` in SQL: It produces a piece of output for each variable binding that satisfies the `where`-clause.

First Example (3)

- Note that the order of the clauses in XQuery fits better with the evaluation sequence: Also in SQL, `FROM` is conceptually evaluated first, and `SELECT` last.
- In XQuery, all keywords are written in lower case.

  In contrast, SQL is case-insensitive. However, since case is important in XML, the choice for XQuery seems natural. As XPath, XQuery has no reserved words. It is possible to name an element `for`.
- XPath 2.0 is a subset of XQuery, i.e. FLWOR-expressions are not the only type of queries.

  The `for`-loop in XPath is a simplified version (special case) of the FLWOR-expression in XQuery.

First Example (4)

- Expressions can be arbitrarily nested, also inside explicitly given XML (direct element constructors).
- < starts literal XML mode, and `{...}` marks sections that must be evaluated:

```
<result>{ for ... where ... return ... }</result>
```

- Result:

```
<result>
  <h1><LAST>Smith</LAST><POINTS>10</POINTS></h1>
  <h1><LAST>Jones</LAST><POINTS>9</POINTS></h1>
  <h1><LAST>Turner</LAST><POINTS>5</POINTS></h1>
</result>
```

First Example (5)

- Of course, the value of an XQuery expression is an XDM sequence.
- How this is printed ("serialized") depends on the implementation (most have options to control this).
- E.g., it could be written into one long line, or indented with one element per line:

```
<result>
    <h1>
        <LAST>Smith</LAST>
        <POINTS>10</POINTS>
    </h1>
    ...
</result>
```


Overview

Overall Syntax (1)

- The basic XQuery unit is a module.
- A module can be
  - ◇ a library module (contains mainly function declarations),
  - ◇ a main module (contains mainly the query).
- Each module may optionally start with a version declaration:

```
xquery version "1.0";
```

One can also specify the encoding, but the treatment of this is implementation-dependent: `xquery version "1.0" encoding "utf-8";`

Overall Syntax (2)

- A main module consists of a prolog (which can be empty) and the query ("QueryBody").
- A library module consists of a module declaration and a prolog.
- The prolog can contain
  - ◇ First an arbitrary sequence of namespace declaractions, module import commands (for schemas and modules), and XQuery parameter settings,
  - ◇ and then an arbitrary sequence of variable, function, and option declarations.

Overall Syntax (3)

- The query itself (“QueryBody”) is an expression.
- XPath-expressions are also XQuery-expressions.

  However, the grammar in the XQuery standard completely defines expressions. Basically, XPath is a restricted version of XQuery. Since XQuery has extensions in many places, it was not possible to simply embedd an XPath expression as defined in the XPath standard.
- As in XPath, all data values are sequences of items, where items are atomic values or nodes.
- Expressions can be arbitrarily nested.

  While only recent SQL DBMS support the use of an SQL query with one result value as a term, the arbitrary nesting was a basic design principle in XQuery. It is sometimes called a funtional language.

Expressions (1)

- On the top level, an expression consists of one or more subexpressions (“ExprSingle”) separated by “,” (sequence concatenation operator).
- On the next level, an expression (“ExprSingle”) is
  - ◇ a FLWOR-expression,
  - ◇ a quantified expression (`some`, `every`)
  - ◇ a typeswitch expression (see below)
  - ◇ an `if`-expression,
  - ◇ or an expression with the usual logical, comparison and arithmetic operators (see below).

Expressions (2)

- In comparison, the `XPath 2.0` grammar has
  - ◇ a `for`-expression instead of the `FLWOR`-expression,
  - ◇ no `typeswitch` expression.
- Note that the `for`-expressions in XPath 2.0 are valid FLWOR-expressions in XQuery:
  - ◇ They have only the `for` and the `return` part.

    It is legal in XQuery to leave out the other parts.
  - ◇ The `for`-clause is simplified: XQuery permits to declare a type for the variable, and to add a positional variable (see below).

Expressions (3)

- The grammars for XQuery and XPath 2.0 are very similar (they are generated from a single source, only some possibilities are missing in XPath or replaced by simpler mechanisms).
- Continuing the comparison, one finds that
  - ◇ Quantified expressions (`some`, `every`) permit a type declaration for the variable in XQuery.

    In XPath, no such type declaration is possible. In XQuery, it is optional (thus, XPath is still a subset of XQuery).

Expressions (4)

- The "valueExpression" (Argument of unary + and -, i.e. at the end of the operator hierarchy) is a path expression in XPath. In XQuery, there are two additional possibilities:
  - ◇ `validate (strict|lax) {` ⟨Expression⟩ `}`

    The expression must evaluate to exactly one document or element node. It is treated as an XML infoset (i.e. existing type annotations are ignored), validated according to the "in-scope schema definitions", and a new tree is built from the PSVI. However, the "Schema Import Feature" is optional in XQuery.
  - ◇ An "extension expression" with a pragma: `(# ...#) {` ⟨Expression⟩ `}`

Expressions (5)

- As explained above, the XPath grammar permits the `namespace` axis, with is not supported in XQuery.

  But because it can be supported only in an inefficient way, it is anyway no longer recommended to use it.
- The next difference is in the "Primary Expression":
  - ◇ Both languages permit numeric and string literals, variable references, expressions in `(...)`, the context item ".", and function calls.
  - ◇ XQuery permits in addition constructors (see below), and "`ordered|unordered {` ⟨Expression⟩ `}`".

Operator Precedences (1)

Prio	Operator	Assoc.
1	`,` (comma)	left
2	`:=` (assignment)	right
3	`for, some, every, typeswitch, if`	left
4	`or`	left
5	`and`	left
6	`eq,ne,lt,le,gt,ge,=,!=,<,<=,>,>=,is,<<,>>`	left
7	`to`	left
8	`+, -`	left
9	`*, div, idiv, mod`	left
10	`union, |`	left

(continued on next slide)

Operator Precedences (2)

(continued from previous slide)

Prio	Operator	Assoc.
11	`intersect, except`	left
12	`instance of`	left
13	`treat as`	left
14	`castable`	left
15	`cast`	left
16	`-` (unary), `+` (unary)	right
17	`?, *, +` (Occurrence Indicators)	left
18	`/, //`	left
19	`[ ], ( ), {}`	left

Only differences (additions) to XPath: `:=`, `typeswitch`.

typeswitch-Expression (1)

- The typeswitch-expression permits to check the dynamic type of an expression, and to distinguish different cases based on this type:

```
typeswitch($cust/address)
  case $a as element(*,USAddr)  return $a/state
  case $a as element(*,CanAddr) return $a/prov
  case    element(*,GermanAddr) return ()
  default return fn:error("Unknown address type")
```

- element(*,USAddr) matches any non-nilled element node with type annotation USAddr.

  Or a type derived from that. This example needs schema validation.

typeswitch-Expression (2)

- The first case-clause with a matching type is selected, or the default clause if non matches.
- A variable must be declared in the case only if the value of the original expression is needed to compute the return value.

  The scope of this variable declaration is this single case. Different cases can declare variables with the same name.

- The same effect can be achieved with conditional expressions (if) and “instance of”.

  “treat as” is necessary in addition to use the value as a value of its real type. So in the end, the typeswitch simplifies the expression.

Constructors: Overview

- An important difference between XPath and XQuery is that XQuery can generate new nodes, XPath can only select nodes from given documents.
- Creating new nodes is done by constructors.
- There are two types of constructors in XQuery:
  - ◇ Direct constructors, which look like XML text. There e.g. the node name is explicitly given.
  - ◇ Computed constructors, which have a new syntax, and permit to compute e.g. the node name by an expression.

Direct Constructors (1)

- A direct constructor looks like XML text that is directly copied to the output, but one can embed XQuery expressions to be evaluated in {...}.
- For example, the XQuery expression

```
<a b="Aufg.{1*1}">1+1={1+1}</a>
```

  is evaluated to 1+1=2.

- Direct constructors are parsed by XQuery, they are not copied character by character to the output.

  The internal XDM representation of the output is constructed, thus e.g. information about extra whitespace/line breaks inside tags is lost.

Direct Constructors (2)

- If one needs curly braces "{" or "}" in the data, one must double them: "{{" or "}}".

  Alternatively, one can use character references: { and }.

- Because the direct constructor only mimics XML, but is defined in the XQuery grammar, there is a slight difference: When one encloses an attribute value inside ", one can use "" to denote the character " inside the string.

  Correspondingly, when one encloses it in ', the apostrophe is written ''. This is the XQuery/XPath convention, not the XML convention. There one must use an entity reference or a character reference.

Direct Constructors (3)

- Furthermore, entity references and character references are expanded, not copied to the output:
  - ◇ Only the predefined entities (<, >, &, ", ') can be used in entity references.

    It might be that the output serialization uses these entity references again if the character itself would be invalid (e.g. " inside a "-delimited attribute value). But if it is not necessary to use the entity reference, it will be printed in expanded form.
  - ◇ In the same way, character references are expanded (e.g., a is replaced by "a").

    Again, the output must of course be valid XML, which might require some form of escaping (entity / character references).

Direct Constructors (4)

- XQuery comments (: ... :) cannot be used in the direct element constructor, neither in the tags nor in the content (except of course inside {...}).

  In the tags they are a syntax error, in the content they are considered as text data. Formally, comments can appear everywhere where "ignorable whitespace" can appear. A few productions in the XQuery grammar are marked with /* ws:explicit */. Inside these productions, the nonterminal S (known form the XML grammar) is used to mark explicitly where whitespace is allowed. This nonterminal does not match the XQuery comment. The productions for the direct element constructor have this explicit whitespace. In this way they are made more compatible with the real XML grammar, although I personally do not see the advantage of forbidding comments inside tags.

Direct Constructors (5)

- Allowed occurrences of embedded XQuery expressions ({...}) inside direct element constructors:
  - ◇ The element name (element type) and the attribute names must be given explicitly (a QName), and cannot be computed with {...}.

    If one wants to compute these, one must use the computed element constructor (see below).
  - ◇ Embedded XQuery expressions ({...}) can be used only inside the attribute value (inside "..." or '...') and in the element content.

Direct Constructors (6)

- If {...} is used in an attribute value, the constructed attribute value is computed as follows:
  - ◇ The expressions inside {...} are evaluated and atomization is applied to the result.
  - ◇ Thus, one gets a sequence of atomic values for each {...}. These values are converted (with a cast) to strings, and concatenated with a single space between each pair.

    At the beginning and the end of the sequence, no space is inserted, thus the empty sequence gives the empty string.

Direct Constructors (7)

- Computation of attribute value, continued:
  - ◇ Then the explicitly given characters and the strings resulting from each {...} are concatenated without adding spaces.
  - ◇ Example:

```
<a b="xy{ 1 to 3 }z{ 3, 4 }" />
```

    is evaluated to <a b="xy1 2 3z3 4"/>.
  - ◇ If the attribute name is xml:id, the attribute value is treated specially (as an ID).

Direct Constructors (8)

- The content of a direct element constructor can contain (between start tag and end tag):
  - ◇ Literal text (without the characters <, {, }, &),
  - ◇ entity references for the five predefined entities,
  - ◇ character references,
  - ◇ CDATA sections: <![CDATA[...]]>,
  - ◇ enclosed expressions: {...},
  - ◇ other direct constructors (for element, comment, and processing instruction nodes).

Direct Constructors (9)

- Even variable references are not interpreted inside the content (or attribute value) of a direct element constructor: The "$"-sign is treated as literal text.
- For example

```
for $i in (1, 2, 3) return <a>$i</a>
```

  gives

```
<a>$i</a>
<a>$i</a>
<a>$i</a>
```

- Inside the constructor, one must write {$i} to get the value of the variable $i.

Direct Constructors (10)

- A sequence of whitespace characters (e.g. spaces, line breaks) within the content of a direct element constructor is considered "boundary whitespace" if it is delimited on both sides by
  - ◇ the start or end of the content (i.e. the start tag or end tag of the direct element constructor), or
  - ◇ and enclosed direct constructor (e.g. start and end tags of direct element constructors), or
  - ◇ an enclosed expression `{...}`.

    Space characters generated by character references, CDATA sections, or enclosed expressions do not count as whitespace here.

Direct Constructors (11)

- Boundary whitespace is
  - ◇ eliminated if the boundary whitespace policy in the static context is "`strip`",
  - ◇ and it is copied to content of the construced element node if the boundary whitespace policy is "`preserve`".
- The boundary whitespace policy can be set with a declaration in the prolog:

```
declare boundary-space preserve;
```

- The default is implementation-defined.

Direct Constructors (12)

- Exercise: How does this XQuery expression look like without the boundary whitespace?

```
<a>
    <b> xy </b>
    <c> {"xy"} </c>
    <d> &#x20; <!-- This is a space --> </d>
    (: Be careful here! :)
</a>
```

- The example shows also a direct comment constructor.

  One cannot use enclosed expressions `{...}` in a direct comment constructor. One must write the comment explicitly. But there is of course also a computed comment constructor (see below).

Direct Constructors (13)

- The content of a direct element constructor is evaluated to a sequence of nodes as follows:
  - ◇ Each consecutive sequence of literal characters (including characters from entity/character references and CDATA sections) evaluates to a single text node.
  - ◇ Each nested direct constructor is evaluated, resulting in a new node.

    The parent property of this new node is set to the element node that is currently being constructed. The standard also explains how the `base-uri`-property is set (see Section 3.7.1.3).

Direct Constructors (14)

- Evaluation of content of a direct element constructor, continued:
  - ◇ Each enclosed expression `{...}` is evaluated to a sequence of items.
  - ◇ For each subsequence of adjacent atomic values, a single text node is constructed, containing the values converted to strings with a single space inserted between each pair.
  - ◇ For each node in the sequence returned by `{...}`, a new copy is made of this node and the entire subtree below it. (see details below).

Direct Constructors (15)

- Eval. of content of direct element constr., cont.:
  - ◇ A document node is replaced by its children.
  - ◇ Now there might be again adjacent text nodes, which are merged into a single text node.
- It is permitted that the resulting sequence contains attribute nodes, but only at the very beginning.
- These become attributes of the constructed element nodes (in addition to the attributes explicitly specified in the direct element constructor), the remaining nodes become its children.

Direct Constructors (16)

- The construction and copying of nodes is influenced by the construction mode, which can be:
  - ◇ `strip`: a new document is constructed without the information generated only by the validation.

    So the new node as well as the copied element nodes receive the type `xs:untyped`, and copied attribute nodes are treated as `xs:untypedAtomic`. Properties `nilled`, `is-id`, and `is-idrefs` are all set to false (except for attribute nodes called `xml:id`). All typed values stored in the original nodes are converted to strings.
  - ◇ `preserve`: information from schema-validation of the original document is preserved.

    The new node gets the type `xs:anyType`, but all copied nodes retain their original type. Properties like `nilled` are copied.

Direct Constructors (17)

- Another parameter is the copy-namespaces mode. It contains two components:
  - ◇ `preserve` means that the in-scope namespaces of the original node are copied to its copy.

    `no-preserve`: only namespaces used in the element name or its attributes (i.e. the necessary namespaces) are copied. But if then the typed value of the element or one of its attributes is of type `QName` or `NOTATION` (“namespace sensitive”), an error occurs.
  - ◇ `inherit` means that in-scope namespaces from the constructed node are inherited to its contents (the copied nodes).

    Possibly overridden by namespaces copied from original node.

Direct Constructors (18)

- The following example shows that nodes are indeed copied, getting a new identity:

```
let $x := <a/>
let $y := <b>{$x}</b>
let $z := $y/a
return if($x is $z) then "yes" else "no"
```

- `let` is a clause of the FLWOR-expression that binds a variable to the sequence on the right hand side.
- The result is `"no"`: Although `$z` is `<a/>` constructed from `$x`, it has a new identity.

Computed Constructors (1)

- A computed constructor starts with a keyword that indicates the type of node to be constructed: `element`, `attribute`, `text`, `processing-instruction`, `comment`, `document`.
- For node types with a name (element, attribute, PI), a name specification follows. This can be an explicitly given QName or an enclosed expression `{...}` ("name expression of the constructor").
- Next, the content is defined by an expression in `{...}` ("content expression").

Computed Constructors (2)

- E.g. the XQuery expression

```
element STUDENT {
    element SID { 101 }
    element FIRST { "Ann" }
    element LAST  { "Smith" }
}
```

gives

```
<STUDENT>
    <SID>101</SID>
    <FIRST>Ann</FIRST>
    <LAST>Smith</LAST>
</STUDENT>
```


Computed Constructors (3)

- One can also compute the element (type) name:

```
element { concat("S", "ID") } { 100+1 }
```

- Atomization is applied to the name expression.

  afterwards it must be of type `xs:QName`, `xs:string`, or `xs:untypedAtomic`.

- Otherwise, the processing is done as for direct constructors.

  Especially, if the result of evaluating the content expression of an element constructor starts with attribute nodes, these are assigned to the constructed element node.

- If the content expression is missing, the content is empty. One must still write `{}` in this case.

Computed Constructors (4)

- For attribute, text, comment, and PI constructors, atomization is applied to the result of evaluating the content expression.
- The resulting atomic values are cast into strings and concatenated with a single space inserted between each pair (empty sequence → empty string).
- For constructed attribute nodes the type annotation is `xs:untypedAtomic`.
- Constructed text nodes are automatically deleted when their text is the empty string.

Overview

FLWOR-Expressions (1)

- An important construct of XQuery are FLWOR-expressions (pronounced "Flower-expressions"):

```
for $⟨var⟩ in ⟨ExprSingle⟩, ...
let $⟨var⟩ := ⟨ExprSingle⟩, ...
where ⟨ExprSingle⟩
[stable] order by ⟨OrderSpecList⟩
return ⟨ExprSingle⟩
```

- One can use `for` and `let` multiple times in arbitrary order. At least one of the two is required.

  ExprSingle is an XQuery expression without the "," outside (...).

- `where` and `order by` are optional.

FLWOR-Expressions (2)

- The expressions in the `for` and `let` clauses are evaluted to produce a sequence.

  In case of the `for` clause, this is called the "binding sequence" for the variable.

- The `for`-clause iterates over the elements of sequence, e.g.

```
for $i in (1, 2, 3) return <a>{$i}</a>
```

gives

```
<a>1</a>
<a>2</a>
<a>3</a>
```

FLWOR-Expressions (3)

- In contrast, the `let`-clause assigns the entire sequence to the variable, e.g.

```
let $i := (1, 2, 3) return <a>{$i}</a>
```

gives

```
<a>1 2 3</a>
```

Here the sequence of atomic values is mapped to a single text node as explained above for the constructors. But e.g. `let $i := (<a/>, <b/>, <c/>) return <x>{$i}</x>` gives `<x><a/><b/><c/></x>`. In contrast, `for` gives `<x><a/></x><x><b/></x><x><c/></x>`.

FLWOR-Expressions (4)

- Semantically, it makes no difference whether several variables are bound in a single `for`/`let`-clause, or whether the keyword is repeated each time.

  This is of course the same rule as for the `for`-expressions in XPath.

- For instance,

```
for $i in ('a', 'b'), $j in (1, 2)
return element {$i} {$j}
```

is equivalent to:

```
for $i in ('a', 'b')
for $j in (1, 2)
return element {$i} { $j }
```


FLWOR-Expressions (5)

- Both of the above queries produce the following result (if ordering mode is `ordered`, see below):

```
<a>1</a>
<a>2</a>
<b>1</b>
<b>2</b>
```

- This fits well with the nested `for`-loops: For each value of the variable `$i` in the outer for loop (each element name), the inner for loop (over `$j`) (the element content) is repeated once.

FLWOR-Expressions (6)

- Thus, the sequence in which variables are declared in the `for`-clause is important.

  In contrast, in SQL the sequence of variable declarations under `FROM` is semantically not important.

- If one exchanges the two variable declarations, i.e.

```
for $j in (1, 2), $i in ('a', 'b')
return element {$i} {$j}
```

the output is produced in a different order:

```
<a>1</a>
<b>1</b>
<a>2</a>
<b>2</b>
```

FLWOR-Expressions (7)

- Together, the `for` and `let` clauses produce a sequence of variable assignments (mapping each variable to a value, i.e. an XDM sequence).
- The XQuery standard uses the word "tuple" instead of "variable assignment".

  I consider that unfortunate, because it makes the comparison with SQL more difficult (where each variable is bound to a "tuple"). The standard uses the word "variable binding" for the association between a single variable and its value. A "tuple" consists of "variable bindings". Of course, it is formally true, that a tuple is the generalization of pair, triple, and so on, and here several "variable bindings" are combined (a tuple can also be seen as a mapping from names to values, which fits here, too, but usually the names are column names).

FLWOR-Expressions (8)

- The sequence of variable assignments generated by `for`/`let` is called "tuple stream" in the standard.
- There is a parameter called "ordering mode", which can be "`ordered`" or "`unordered`".

  This parameter is explained in detail on Slide 7-80 and following.
- If the ordering mode is `ordered`, XQuery guarantees that the tuple stream is in the sequence that corresponds to the nested `for`-loops (see above).
- Then the document order of the original document is retained in the output of the query.

FLWOR-Expressions (9)

- Next, the `where`-clause acts as a filter on the "tuple stream" (sequence of variable assignments).
- For each variable assignment, the expression under `where` is evaluated, and its effective boolean value is determined.
- If it is false, the variable assignment is deleted from the sequence.

  I.e. the remaining sequence of variable assignments contains only those variable assignments for which the `where`-condition is true. The variable assignments remain in the same relative order in which they were generated by the `for` and `let`-clauses.

FLWOR-Expressions (10)

- Because the effective boolean value is automatically determined, one can easily check the existence of a node:

```
for $s in //STUDENT
where //RESULT[SID=$s/SID and CAT='H']
return $s/LAST
```

  (students who submitted at least one homework).

  Remember that the effective boolean value of a sequence that starts with a node is true, whereas the effective boolean value of the empty sequence is false. Other possible cases are singleton sequences of boolean type, of string type including `anyURI` and `untypedAtomic` (only the empty string is considered false), and of numeric type (only NaN and 0 are treated as false). In all other cases, a type error occurs.

FLWOR-Expressions (11)

- The `where`-clause is optional. As in SQL, it defaults to "`true`" (no variable assignments are deleted).
- If an "`order by`" clause is specified, the remaining sequence of variable assignments is then sorted.

  The `order by`-clause is explained on Slide 7-86 and following.
- The last step is the `return`-clause (required): For each variable assignment, the `return`-expression is evaluated, and the resulting item sequences are concatenated in the order given by the current sequence of variable assignments.

FLWOR-Expressions (12)

- The length of the result sequence can differ from the number of variable assignments considered, because the `return`-expression can evaluate to a sequence of arbitrary length (0, 1, or more).
- In the following query, there is one variable assignment per SQL exercise (2), but the output sequence contains one entry per solution ($5 = 2+3$):

```
for $e in //EXERCISE[TOPIC='SQL']
return //RESULT[CAT=$e/CAT and ENO=$e/ENO]
```


FLWOR-Expressions (13)

- All subexpressions of the FLWOR-expression are "ExprSingle", thus the operator "," for sequence concatenation can be used only inside `(...)`.
- The entire FLWOR-expression has a higher priority than the comma operator, thus

```
for $i in (1, 2, 3)
return <a> {$i} </a>, <b/>
```

  is not a syntax error, but returns

```
<a>1</a>
<a>2</a>
<a>3</a>
<b/>
```


FLWOR-Expressions (14)

- If one uses constructors under `return`, all nodes are new (entire subtrees are copied).
- If one uses only standard XPath-expressions (e.g. variable names), no copying is done.

  This also necessary for the compatibility with XPath, which has simple `for`-loops and never constructs new nodes (only new sequences).
- For example, the following returns `"yes"`:

```
let $x := (<a/>)
let $y := (for $x1 in $x return $x1)
let $z := (for $x2 in $x return $x2)
return if($y is $z) then "yes" else "no"
```

for-Clause: Details (1)

- The scope of a variable declared with `for` or `let` extends from the point just after the binding expression (which defines the values for the variable) to the end of the FLWOR-expression.
- Thus, the variable can already be used in binding expressions for other variables declared later in the same `for`-clause:

```
for $s in //STUDENT, $r in //RESULT[SID=$s/SID]
return element solved {$s/LAST, $r/CAT, $r/ENO}
```


for-Clause: Details (2)

- This rule for the scope of variables fits with the equivalence with nested `for`-loops.
- It differs from SQL: There the variable declarations in the `FROM`-clause are conceptually done in parallel.

  Thus, one cannot use a tuple variable in a subquery later in the same `FROM`-clause. This gives the query optimizer more freedom to determine the join order.
- It is legal (but bad style) to declare several variables with the same name in a FLWOR-expression: Each new declaration shadows the previously declared variable for the rest of the `FLWOR`-expression.

for-Clause: Details (3)

- One can define a "positional variable" associated with a variable declared in a `for`-clause, e.g.

```
for $s at $i in //STUDENT
return element STUD {
                attribute ID {$i},
                $s/concat(LAST, ", ", FIRST)
          }
```

- `$i` contains the position of the current value for `$s` in the binding sequence, i.e. the value of `//STUDENT`.
- Positions are counted from `1`. The result of the query is shown on the next slide.

for-Clause: Details (4)

- In the example, the positional variable is used to generate new unique IDs for the students:

```
<STUD ID="1">Smith, Ann</STUD>
<STUD ID="2">Jones, Michael</STUD>
<STUD ID="3">Turner, Richard</STUD>
<STUD ID="4">Brown, Maria</STUD>
```

- Other applications of positional variables include:
  - ◇ First-n queries (see below).
  - ◇ Sampling: E.g. take only every 10-th student:

    `$i mod 10 = 1`

for-Clause: Details (5)

- It is also possible to declare a type for the variable:

```
for $p as xs:decimal
        in //RESULT[CAT='H' and ENO=1]/POINTS
return $p div 10
```

- As I understand the standard, this is a type assertion (like "treat as"), so it should give an error here (an element node is not a decimal value).

  Type assertions might be necessary to permit static type checking.

- AltovaXML complains only if it cannot convert the value to the required type (treats it as type cast).
- Positional variables always have type `xs:integer`.

for-Clause: Details (6)

- The `for`-clause consists of a comma-separated list of one or more variable declarations, each consisting of:
  - ◇ The name of the variable (starting with "$"),
  - ◇ optionally, a type declaration, consisting of the keyword "as" and a sequence type,

    In the `for`-clause, the occurrence indicators ?, +, * are not relevant because the variable is bound to single sequence elements.
  - ◇ optionally, a positional variable declaration, consisting of the keyword "at" and a variable name,
  - ◇ the keyword "in", and an "ExprSingle".

let-Clause: Details

- The `let`-clause consists of a comma-separated list of one or more variable declarations with a slightly different syntax than under `for` (to emphasize that the entire sequence is bound to the variable):
  - ◇ Name of the variable (starting with "$"),
  - ◇ optionally, a type declaration, consisting of the keyword "as" and a sequence type,
  - ◇ the symbol ":=", and an "ExprSingle".
- Of course, positional variables make no sense in the `let`-clause (and are therefore not permitted).

Ordering Mode (1)

- The ordering mode has an important influence on the semantics of XQuery expressions:
  - ◇ If it is `ordered`, the sequence of variable assignments constructed by `for`/`let` is as above.

    I.e. it corresponds to nested loops in the order or variable declarations, and respects the document order.
  - ◇ If it is unordered, the implementation has more freedom for query optimization: Especially, the sequence of variable assignments generated by `for` and `let` is in an implementation-defined order (unless the `order by` clause is used).

Ordering Mode (2)

- But the consequences of ordering mode `unordered` are even more drastic, because
  - ◇ in XPath expressions, document order does not have to be respected,
  - ◇ thus selecting specific positions becomes more or less meaningless (nondeterministic).
  - ◇ E.g. `/a/b[1]` gives any `b`-child of `a`, not necessarily the first.

    But `/a/b[3]` is non-empty only if there are at least three `b`-children.

Ordering Mode (3)

- Thus, ordering mode `unordered` is not only a question of the output sequence, but can modify also the selected values.

  Actually, that is not so astonishing, because the arbitrary nesting of XQuery expressions means that as soon as one allows a different result sequence in FLWOR-expressions, one could anyway get an entirely different result for the entire query, not only a permutation. Everything in XQuery is a sequence, and the exact order matters in many places.

- When one uses the XPath function `unordered(...)` only at this single point an arbitrary permutation is allowed (not everywhere inside as with the ordering mode). E.g. positions inside remain meaningful.

Ordering Mode (4)

- Whether the nondeterminism is bad, depends on the data (it might have been better to declare this in the schema instead of in the query):
  - ◇ If, e.g., the data are a dump from a relational database, the order of the rows is meaningless.

    It only occurs because the data must be written to a file in some order. The problem is that XML permits to use this order and that XML makes the impression that the order might mean something.

  - ◇ Then nobody would use positions in the query or other constructs that depend on the order.

    And one does not want to pay a price for getting a specific order if that order is anyway irrelevant.

Ordering Mode (5)

- The ordering mode is part of the static context and be set in the prolog, e.g.

  ```
  declare ordering ordered
  ```

  and locally inside the query with the expressions
  - ◇ `ordered {...}`, and

    I.e. for evaluating "...", the ordering mode is set to "`ordered`".

  - ◇ `unordered {...}`.

    Note the difference to `unordered(...)`, the XPath function.

- The default value is implementation-defined.

  This seems unfortunate, because it immediately causes portability problems: The ordering mode is important for nearly every query.

Ordering Mode (6)

- E.g. suppose that the homework results are stored in the document in order of submission (i.e. new entries are always appended at the end).
- If one wants to print all student names in the sequence in which they submitted Homework 1, this can be done as follows:

```
ordered {
    for $r in //RESULT, $s in //STUDENT
    where $r/CAT = 'H' and $r/ENO = 1
          and $r/SID = $s/SID
    return $s/LAST
}
```


`order by` Clause (1)

- With the `order by` clause, one can sort the tuple stream (variable assignments) generated by `for`/`let`.

  After passing the filter of the `where`-clause and before the `return`-clause is evaluated.

- This is done by defining one or more expressions, the values of which are used for sorting, e.g.

```
for $s in //STUDENT, $r in //RESULT[SID=$s/SID]
where $r/CAT = 'H' and $r/ENO = 1
order by $r/POINTS
return $s/LAST
```


`order by` Clause (2)

- More specifically, the expression(s) are evaluated for each variable assignment, and atomization is applied.

  After that, each expression must return a sequence of length ≤ 1 (for a given variable assignment), i.e. a single value or the empty sequence. Otherwise (longer sequence) a type error occurs.

- Values of type `untypedAtomic` are treated as `string`.
- Then the values of each expression (for all variable assignments) must be comparable: The comparison is done with the operator `gt` of the least common supertype that has such an operator.

`order by` Clause (3)

- Suppose that n expressions are used as sort criteria, and the values for variable assignment $\mathcal{A}$ are $(x_1, \ldots, x_n)$, and for variable assignment $\mathcal{B}$, the values are $(y_1, \ldots, y_n)$.
- Then $\mathcal{A}$ comes after $\mathcal{B}$ in the sort order if there is $i \in \{1, \ldots, n\}$ such that:
  - ◇ neither $x_j > y_j$ nor $y_j > x_j$ for $j = 1, \ldots, i-1$,
  - ◇ $x_i > y_i$.

  I.e. the result of the first expression has highest priority, the second expression decides the relative order of two variable assignments when the values of the first expression are equal/uncomparable, and so on.

`order by` Clause (4)

- For each expression, one can specify `ascending` or `descending`:
  - ◇ `ascending` is the default: The least value is listed at the beginning, the greatest at the end.
  - ◇ `descending` selects the inverse order: The maximum value is listed first, and then successively smaller values, until the minimum value.
- The abbreviations `asc` and `desc` known from SQL are not supported in XQuery.

`order by` Clause (5)

- Another difference to SQL is that XQuery does not require that the values used for sorting are also printed.

  However, modern SQL DBMS do not actually have this requirement.
- For each column, one can specify
  - ◇ `empty greatest`: The empty sequence is listed last in ascending order (first in descending order).

    And NaN comes immediately before the empty sequence in ascending order (immediately after it in descending order).
  - ◇ `empty least`: The empty sequence comes first in ascending order, last in descending order.

`order by` Clause (6)

- Thus, the `order by` clause consists of a comma-separated list of one or more "order specs", each of which consists of
  - ◇ an "ExprSingle" (values used for sorting),
  - ◇ optionally, one of the keywords "`ascending`" or "`descending`",
  - ◇ optionally, one of the phrases "`empty greatest`" or "`empty least`",
  - ◇ optionally, the keyword "`collation`" and an URI literal (this defines the sort order for strings).

`order by` Clause (7)

- Finally, instead of "`order by`", one can also write "`stable order by`".
- This means that if the sort criteria give no decision, the original order of the variable assignments must be kept (derived from the document order and the sequence/nesting of `for`-loops).
- Of course, if an `order by` clause is specified, this takes precedence over the ordering mode "`unordered`".

`order by` Clause (8)

- Strings are sorted differently from numbers, e.g. the string `"12"` comes before the string `"9"`.

  Already the first character is different and decides the sort order.

- If one uses a document without schema, or an XQuery system without validation, the type of (attribute or element) values is `untypedAtomic`.
- This is converted to `string` for comparison.
- Thus, “`order by $r/POINTS`” will not work. One has to write “`order by number($r/POINTS)`”.

  Alternative: `xs:integer(...)`.

Overview

Example Document (1)

STUDENTS

SID	FIRST	LAST	EMAIL
101	Ann	Smith	···
102	Michael	Jones	(null)
103	Richard	Turner	···
104	Maria	Brown	···

EXERCISES

CAT	ENO	TOPIC	MAXPT
H	1	Rel. Algeb.	10
H	2	SQL	10
M	1	SQL	14

RESULTS

SID	CAT	ENO	POINTS
101	H	1	10
101	H	2	8
101	M	1	12
102	H	1	9
102	H	2	9
102	M	1	10
103	H	1	5
103	M	1	7

Example Document (2)

- Version 1 (data in elements, structured by relation):

```
<?xml version='1.0' encoding='ISO-8859-1'?>
<GRADES-DB>
  <STUDENTS>
    <STUDENT>
      <SID>101</SID>
      <FIRST>Ann</FIRST>
      <LAST>Smith</LAST>
    </STUDENT>
    ...
  </STUDENTS>
  ...
</GRADES-DB>
```

Simple Query (1)

- Print all homework results of student 101:

```
SELECT ENO, POINTS
FROM   RESULTS
WHERE  CAT = 'H' AND SID = 101
```

ENO	POINTS
1	10
2	8

- Solution 1 in XQuery:

```
<STUD_101>{
    for $r in /GRADES-DB/RESULTS/RESULT
    where $r/CAT = 'H' and $r/SID = 101
    return <HW>{$r/ENO, $r/POINTS}</HW>
}</STUD_101>
```

Whereas the result table structure in SQL is fixed, there are many ways to structure the output in XML. The query has to specify this.

Simple Query (2)

- Result:

```
<STUD_101>
    <HW>
        <ENO>1</ENO>
        <POINTS>10</POINTS>
    </HW>
    <HW>
        <ENO>2</ENO>
        <POINTS>8</POINTS>
    </HW>
</STUD_101>
```

Of course, the line breaks and indentation depend on the output serialization. This is the result of BaseX. AltovaXML by default writes everything in one line, but `/oi yes` ("outputindent") gives the above.

Simple Query (3)

- Solution 2 (predicate under `for` instead of `where`-clause, requires fewer variable references):

```
<STUD_101>{
    for $r in //RESULT[CAT = 'H' and SID = 101]
    return <HW>{$r/ENO, $r/POINTS}</HW>
}</STUD_101>
```

- Solution 3 (with context instead of variable):

```
<STUD_101>{
    //RESULT[CAT = 'H' and SID = 101]/
        <HW>{ENO, POINTS}</HW>
}</STUD_101>
```


Simple Query (4)

- The last example shows that the use of constructors is not limited to the `return`-clause of FLWOR-expressions.
- In the grammar, constructors are "Primary Expressions", i.e. on the same level as datatype literals or variable references.

  [https://www.w3.org/TR/xquery/#nt-bnf]
  As in XPath, a "StepExpr" in a path expression can not only be an "AxisStep", but also a "FilterExpr", which is a "PrimaryExpr" optionally followed by predicates. This again shows why the grammar rules of XPath had to be repeated in the XQuery grammar: Constructors in primary expressions are new in XQuery, but this has consequences for standard path expressions.

Problem with Namespaces (1)

- Input document with link to XML Schema:

```
<GRADES-DB xmlns:xsi=
    "http://www.w3.org/2001/XMLSchema-instance"
  xsi:noNamespaceSchemaLocation="grades.xsd">
```

- The namespace definition for `xsi` implicitly applies to all element nodes of the input document.
- Since the `ENO` and `POINTS`-nodes are copied from the input document, each such node shows this namespace explicitly in the output, e.g.

```
<ENO xmlns:xsi="...">1<ENO>
<POINTS xmlns:xsi="...">10<POINTS>
```


Problem with Namespaces (2)

- It should be possible to avoid this as follows:

```
declare copy-namespaces no-preserve, no-inherit;
<STUD_101>{
    for $r in //RESULT[CAT = 'H' and SID = 101]
    return <HW>{$r/ENO, $r/POINTS}</HW>
}</STUD_101>
```

`inherit` or `no-inherit` is actually not important for this task, but the syntax forces one to specify it together with `no-preserve`. The standard says: "If `copy-namespaces` mode specifies `no-preserve`, the new copy retains only those in-scope namespaces of the original element that are used in the names of the element and its attributes."

- But neither BaseX nor AltovaXML remove the namespace declarations.

Problem with Namespaces (3)

- If one specifies this namespace in the outer element of the query, the output serialization does not repeat it in each copied element:

```
<STUD_101 xmlns:xsi=
  "http://www.w3.org/2001/XMLSchema-instance">{
    for $r in //RESULT[CAT = 'H' and SID = 101]
    return <HW>{$r/ENO, $r/POINTS}</HW>
}</STUD_101>
```

Now the namespace is declared for all nodes in the output document, so when the nodes from the input document are copied, they are in scope of the namespace declaration, and it is not necessary to explicitly repeat the inherited declaration.

Problem with Namespaces (4)

- A solution is to construct new element nodes with the same name and the same contents:

```
<STUD_101>{
    for $r in //RESULT[CAT = 'H' and SID = 101]
    return
        <HW>
            <ENO>{data($r/ENO)}</ENO>
            <POINTS>{data($r/POINTS)}</POINTS>
        </HW>
}</STUD_101>
```

With the same query structure, one could also generate an HTML table: Replace `STUD_101` bei `table`, `HW` by `tr`, and the new `ENO` and `POINTS` tags by `td`. One could even add a headline.

Joins (1)

- “Print the names of all students who have at least 9 points in both, Homework 1 and Homework 2”:

```
SELECT S.FIRST, S.LAST
FROM   STUDENTS S, RESULTS H1, RESULTS H2
WHERE  S.SID = H1.SID AND S.SID = H2.SID
AND    H1.CAT = ’H’ AND H1.ENO = 1
AND    H2.CAT = ’H’ AND H2.ENO = 2
AND    H2.POINTS >= 9 AND H2.POINTS >= 9
```

FIRST	LAST
Michael	Jones

Joins (2)

- XQuery:

```
<ANSWER>{
for   $s  in //STUDENT,
      $h1 in //RESULT, $h2 in //RESULT
where $s/SID = $h1/SID and $s/SID = $h2/SID
and   $h1/CAT = ’H’ and $h1/ENO = 1
and   $h2/CAT = ’H’ and $h2/ENO = 2
and   $h1/POINTS >= 9 and $h2/POINTS >= 9
return <ROW FIRST = "{$s/FIRST}"
            LAST  = "{$s/LAST}" />
}</ANSWER>
```

```
Query Result: <ANSWER>
                   <ROW FIRST="Michael" LAST="Jones"/>
              </ANSWER>
```


Joins (3)

- One can move parts of the condition to predicates in the `for`-clause:

```
<ANSWER>{
for   $s  in //STUDENT,
      $h1 in //RESULT[CAT = ’H’ and ENO = 1],
      $h2 in //RESULT[CAT = ’H’ and ENO = 2]
where $s/SID = $h1/SID and $s/SID = $h2/SID
and   $h1/POINTS >= 9 and $h2/POINTS >= 9
return <ROW FIRST = "{$s/FIRST}"
            LAST  = "{$s/LAST}" />
}</ANSWER>
```


Joins (4)

- One can put the entire `where`-condition into predicates in the `for`-clause (similar to joins under `FROM`):

```
<ANSWER>{
for $s  in //STUDENT,
    $h1 in //RESULT[CAT = ’H’] [ENO = 1]
                   [SID = $s/SID]
                   [POINTS >= 9],
    $h2 in //RESULT[CAT = ’H’ and ENO = 2 and
                    SID = $s/SID and
                    POINTS >= 9]
return <ROW FIRST = "{$s/FIRST}"
            LAST  = "{$s/LAST}" />
}</ANSWER>
```

Joins (5)

- An XML document that directly corresponds to the table structure of a relational database does not make optimal use of XML's tree structure.
- If the `RESULT` elements of a student were nested inside the `STUDENT` element, one would not need an explicit join on `SID`:

```
for $s  in //STUDENT,
    $h1 in $s/RESULT[CAT = 'H'] [ENO = 1]
                    [number(POINTS) >= 9],
    $h2 in $s/RESULT[CAT = 'H'] [ENO = 2]
                    [number(POINTS) >= 9]
...
```


Numeric Comparisons (1)

- Who has at least as many points for Homework 1 as Student 101?

```
<ANSWER>{
for   $s in //STUDENT,
      $x in //RESULT[CAT='H' and ENO=1],
      $y in //RESULT[CAT='H' and ENO=1]
where $x/SID = $s/SID and $s/SID != 101
and   $y/SID = 101
and   number($x/POINTS) >= number($y/POINTS)
return <ROW FIRST = "{$s/FIRST}"
            LAST  = "{$s/LAST}" />
}</ANSWER>
```


Numeric Comparisons (2)

- If the document is not validated, attribute values and values of elements with simple content are of type `untypedAtomic`.
- When the comparison is done with a value of numeric type (e.g. a constant as in earlier examples), a numeric comparison is done.

  The `untypedAtomic` value is converted to the more specific type.
- If two `untypedATomic` values are compared, a string comparison is done.

  To get a numeric comparison, one must use `number(...)` on at least one side of the comparison.

NOT EXISTS (1)

- "Print the names of all students who have not yet submitted a homework":

```
SELECT S.FIRST, S.LAST
FROM   STUDENTS S
WHERE  NOT EXISTS(SELECT *
                  FROM   RESULTS R
                  WHERE  R.SID = S.SID
                  AND    R.CAT = 'H')
```

FIRST	LAST
Maria	Brown

NOT EXISTS (2)

- Note that `not(...)` computes the effective boolean value of its argument, and thus can be used to check for existence of a node:

```
<ANSWER>{
for $s in //STUDENT
where not(//RESULT[SID=$s/SID and CAT='H'])
return <ROW FIRST = "{$s/FIRST}"
            LAST  = "{$s/LAST}" />
}</ANSWER>
```

```
Query Result: <ANSWER>
                  <ROW FIRST="Maria" LAST="Brown"/>
              </ANSWER>
```


NOT EXISTS (3)

- Instead of `not(...)` one can also use

```
count(//[RESULT[SID=$s/SID][CAT='H']]) = 0
```

For sequences of nodes, this is equivalent (not for atomic values).

- One can also use an explicit quantifier:

```
<ANSWER>{
for $s in //STUDENT
where every $r in //RESULT[SID = $s/SID]
            satisfies CAT != 'H'
return <ROW FIRST = "{$s/FIRST}"
            LAST  = "{$s/LAST}" />
}</ANSWER>
```


Universal Quantification (1)

- "Print the names of all students who have solved all homeworks in the database":

```
SELECT S.FIRST, S.LAST
FROM   STUDENTS S
WHERE  NOT EXISTS (SELECT *
       FROM  EXERCISES E
       WHERE E.CAT = 'H'
       AND   NOT EXISTS (SELECT *
             FROM  RESULTS R
             WHERE R.SID = S.SID
             AND   R.CAT = 'H'
             AND   R.ENO = E.ENO))
```


Universal Quantification (2)

- In XQuery, the "for all" can be directly expressed:

```
<STUDENTS_WITH_ALL_HOMEWORKS>{
  for $s in //STUDENT
  where every $e in //EXERCISE[CAT='H'] satisfies
    //RESULT[SID=$s/SID][CAT='H'][ENO=$e/ENO]
  return <ROW FIRST = "{$s/FIRST}"
              LAST  = "{$s/LAST}" />
}</STUDENTS_WITH_ALL_HOMEWORKS>
```

```
Query Result: <STUDENTS_WITH_ALL_HOMEWORKSANSWER>
                  <ROW FIRST="Ann" LAST="Smith"/>
                  <ROW FIRST="Michael" LAST="Jones"/>
              </STUDENTS_WITH_ALL_HOMEWORKSANSWER>
```

LIKE (1)

- "Print the names of all students who have an email address from `acm.org`":

```
SELECT FIRST, LAST
FROM   STUDENTS
WHERE  EMAIL LIKE '%@acm.org'
```

FIRST	LAST
Ann	Smith

- The function library contains
  [https://www.w3.org/TR/xpath-functions/]
  - ◇ `contains(`s_1`, `s_2`)`: Substring test.
  - ◇ `starts-with(`s_1`, `s_2`)`: Prefix test.
  - ◇ `ends-with(`s_1`, `s_2`)`: Suffix test (this example).
  - ◇ `matches(`s`, `p`)`: Regular expression test.

LIKE (2)

- Solution with `ends-with`:

```
for $s in //STUDENT[ends-with(EMAIL, '@acm.org')]
return ...
```

- Solution with `matches`:

```
for $s in //STUDENT
          [matches(EMAIL, '^.*@acm.org$')]
return ...
```

`$` matches the end of the string (or a line end in multi-line mode). It is necessary, because otherwise the regular expression could match any substring. In the same way, `^` ensures that the match must begin at the start of the string. Of course, `^.*` could be left out, but in this way, the example demonstrates a match of the entire string. Case-insensitive matching is done with a third argument `'i'` ("flags").

Duplicate Elimination (1)

- "Print the numbers of all homeworks for which there is at least one graded submission (result)":

```
SELECT DISTINCT ENO
FROM   RESULTS
WHERE  CAT = 'H'
```

ENO
1
2

- Duplicates in a sequence of atomic values can be removed with the function `distinct-values`.
- Often, existential quantification helps.

Duplicate Elimination (2)

- Solution with `distinct-values`:

```
<RESULT>{
let $s := //RESULT[CAT = 'H']/ENO
for $n in distinct-values($s)
return <HOMEWORK>{$n}</HOMEWORK>
}</RESULT>
```

Note that `distinct-values` applies atomization to its argument.

- Solution with existential quantification:

```
for $h in //EXERCISE[CAT = 'H']
where exists(//RESULT[CAT = 'H'][ENO = $h/ENO])
return ...
```

Simple Aggregations (1)

- "How many students are in the database?":

```
SELECT COUNT(*)
FROM   STUDENTS
```

COUNT(*)
4

- The library offers the usual aggregation functions:
  - ◇ `count`(s): Number of items in a sequence
  - ◇ `sum`(s[,z]): Sum (with result z for empty seq.)
  - ◇ `avg`(s): Average.
  - ◇ `min`(s): Minimum.
  - ◇ `max`(s): Maximum.

Simple Aggregations (2)

- This is simple in XQuery, too:

```
<NUM_STUDENTS>
  {count(//STUDENT)}
</NUM_STUDENTS>
```

- Note that in XQuery, aggregation functions can be used under `where`, which would be forbidden in SQL.

  The reason is that in XQuery, the argument of the aggregation function computes the set (sequence) of values to be aggregated. In SQL, the argument is only an attribute, and the aggregation is over variable assignments generated by the `FROM`-clause.

Simple Aggregations (3)

- How many distinct topics are there?

```
SELECT COUNT(DISTINCT TOPIC)
FROM   EXERCISES
```

COUNT(...)
2

- In XQuery:

```
count(distinct-values(//EXERCISE/TOPIC))
```

- In SQL, null values are not counted:

```
SELECT COUNT(EMAIL)
FROM   STUDENTS
```

COUNT(EMAIL)
3

- In XQuery, this happend automatically.

  The path `//STUDENT/EMAIL` selects only existent email elements.

GROUP BY (1)

- "Print for every student the total number of homework points."

```
SELECT S.FIRST, S.LAST, SUM(R.POINTS)
FROM   STUDENTS S, RESULTS R
WHERE  S.SID = R.SID AND R.CAT = 'H'
GROUP  BY S.SID, S.FIRST, S.LAST
```

FIRST	LAST	SUM(POINTS)
Ann	Smith	18
Michael	Jones	18
Richard	Turner	5

GROUP BY (2)

- If we want the same in XQuery, we must explicitly exclude students without homeworks:

```
<STUDENTS_WITH_SUM_HW_POINTS>
{
  for $s in //STUDENT
  let $p := //RESULT[SID=$s/SID][CAT='H']/POINTS
  where exists(//RESULT[SID=$s/SID and CAT='H'])
  return <ROW FIRST="{$s/FIRST}" LAST="{$s/LAST}"
              SUM="{sum($p)}" />
}
</STUDENTS_WITH_SUM_HW_POINTS>
```


Restructuring the Data (1)

- Suppose we want to remove the elements for the relations (like STUDENTS), and put the tuple elements directly below GRADES-DB:

```
<GRADES-DB>{
    for $e in /GRADES-DB/*/*
    return $e
}</GRADES-DB>
```

- This gives

```
<GRADES-DB>
    <STUDENT>
        <SID>101</SID>
        ...
```


Restructuring the Data (2)

- The opposite transformation (grouping tuple elements by relation) is also possible:

```
<GRADES-DB>
    <STUDENTS>{
        for $s in /GRADES-DB/STUDENT
        return $s
    }</STUDENTS>
    <EXERCISES>{
        for $e in /GRADES-DB/EXERCISE
        return $e
    }</EXERCISES>
    ...
</GRADES-DB>
```


Restructuring the Data (3)

- Nesting results under students (data in attributes):

```
<GRADES-DB>{
    for $s in //STUDENT
    return element STUDENT {
        for $d in $s/*
        return attribute {name($d)} {data($d)},
        for $r in //RESULT[SID=$s/SID]
        return element RESULT {
            for $a in $r/*
            where name($a) ne "SID"
            return attribute {name($a)} {data($a)}
        }
    },
    ...(: Copy/transform EXERCISE data :)
}</GRADES-DB>
```

Restructuring the Data (4)

- The output looks as follows:

```
<?xml version='1.0' encoding='UTF-8'?>
<GRADES-DB>
  <STUDENT SID='101' FIRST='Ann' LAST='Smith'>
    <RESULT CAT='H' ENO='1' POINTS='10'/>
    <RESULT CAT='H' ENO='2' POINTS='8'/>
    <RESULT CAT='M' ENO='1' POINTS='12'/>
  </STUDENT>
  <STUDENT SID='102' FIRST='Michael' LAST='Jones'>
    ...
  </STUDENT>
  ...
</GRADES-DB>
```


Overview

1. Introduction
2. Basic Syntax, Constructors
3. FLOWR-Expressions
4. Examples, Comparison with SQL
5. Prolog, Functions

Namespaces

- Namespaces can be defined in the Prolog:
  - ◇ `declare namespace Prefix = "URI";`
  - ◇ `declare default element namespace "URI";`
  - ◇ `declare default function namespace "URI";`
- The following namespace prefixes are predeclared:
  - ◇ `xml = http://www.w3.org/XML/1998/namespace`
  - ◇ `xs = http://www.w3.org/2001/XMLSchema`
  - ◇ `xsi = http://www.w3.org/2001/XMLSchema-instance`
  - ◇ `fn = http://www.w3.org/2005/xpath-functions`
  - ◇ `local = http:// www.w3.org/2005/xquery-local-functions`

User-Defined Functions (1)

- One can declare functions in the prolog of the main module (i.e. before the query) and library modules.
- Functions must be in a namespace, but for functions declared in the main module XQuery defines the namespace prefix `local`.
- A simple example is:

```
declare function local:inc($n as xs:integer)
        as xs:integer
        { $n+1 };
local:inc(1) (: This is the query :)
```

User-Defined Functions (2)

- Thus, a function declaration consists of:
  - The keywords "`declare function`",
  - the name of the function with namespace prefix,
  - a comma-separated list of parameter declarations in `(...)`, each consisting of a variable and optionally the keyword "`as`" and a sequence-type,
  - optionally, a specification of the return type: the keyword "`as`" and a sequence-type,
  - and body of the function: an expression in `{...}`
  - and finally a "`;`".

User-Defined Functions (3)

- If the types are not specified, `item()*` is assumed (the most general type).
- Instead of a function body, one can also specify the keyword "`external`".

  It is implementation-dependent if an how functions written in some other language (e.g., C) can be linked to the XQuery evaluator.

- Functions can be recursive.

  And also mutually recursive. XQuery becomes in this way computationally complete, but then it cannot guarantee termination.