Supplementary Material on XQuery

MANCHESTER 1824

The University of Manchester

An Introduction to XQuery

Acknowledgements [1]

- Some of these slides mostly contain text and examples that are most often taken *verbatim* from the article:
  - XQuery: An XML query language. Donald D. Chamberlin. IBM Systems Journal 41(4): 597-615 (2002) http://dx.doi.org/10.1147/sj.414.0597
- These slides were put together from the above publication for educational purposes only.
- Any changes made either reflect updates to XQuery since publication or are the result of minor editing, adaptation and extension for use in teaching.
- The author of these slides is solely responsible for any errrors they contain.
- No claims are made by the author of these slides with regards to any form of intellectual property rights wherever the above author is acknowledged.
- The author of these slides is very grateful to the author of the work above.


XQuery:
An XML query language

by D. Chamberlin

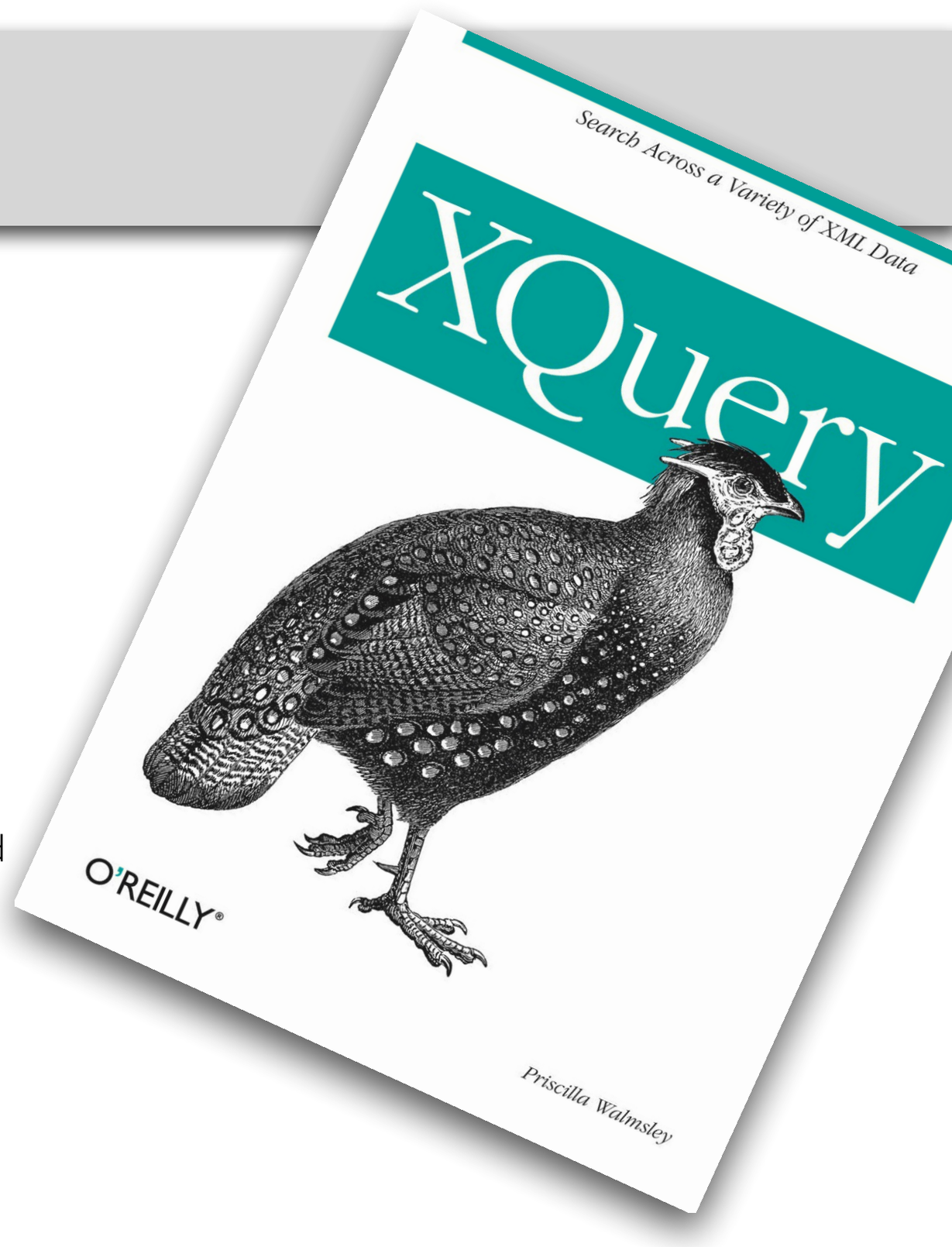
The World Wide Web Consortium has convened a working group to design a query language for Extensible Markup Language (XML) data sources. This new query language, called XQuery, is still evolving and has been described in a series of drafts published by the working group. XQuery is a functional language comprised of several kinds of expressions that can be nested and composed with full generality. It is based on the type system of XML Schema and is designed to be compatible with other XML-related standards. This paper explains the need for an XML query language, provides a tutorial overview of XQuery, and includes several examples of its use.

Acknowledgements [2]

- Some of these slides mostly contain text, examples and materials that are most often taken *verbatim* from the book::
  - An Introduction to XML and Web Technologies. Anders Møller and Michael Schwartzbach. Addison-Wesley, 2006. ISBN 978-0-321-26966-9. http://www.brics.dk/ixwt/ © Pearson Education Limited 2006
- These slides were put together from the above publication for educational purposes only.
- Any changes made either reflect updates to XQuery since publication or are the result of minor editing, adaptation and extension for use in teaching.
- The author of these slides is solely responsible for any errrors they contain.
- No claims are made by the author of these slides with regards to any form of intellectual property rights wherever the above authors are acknowledged.
- The author of these slides is very grateful to the authors of the work above.

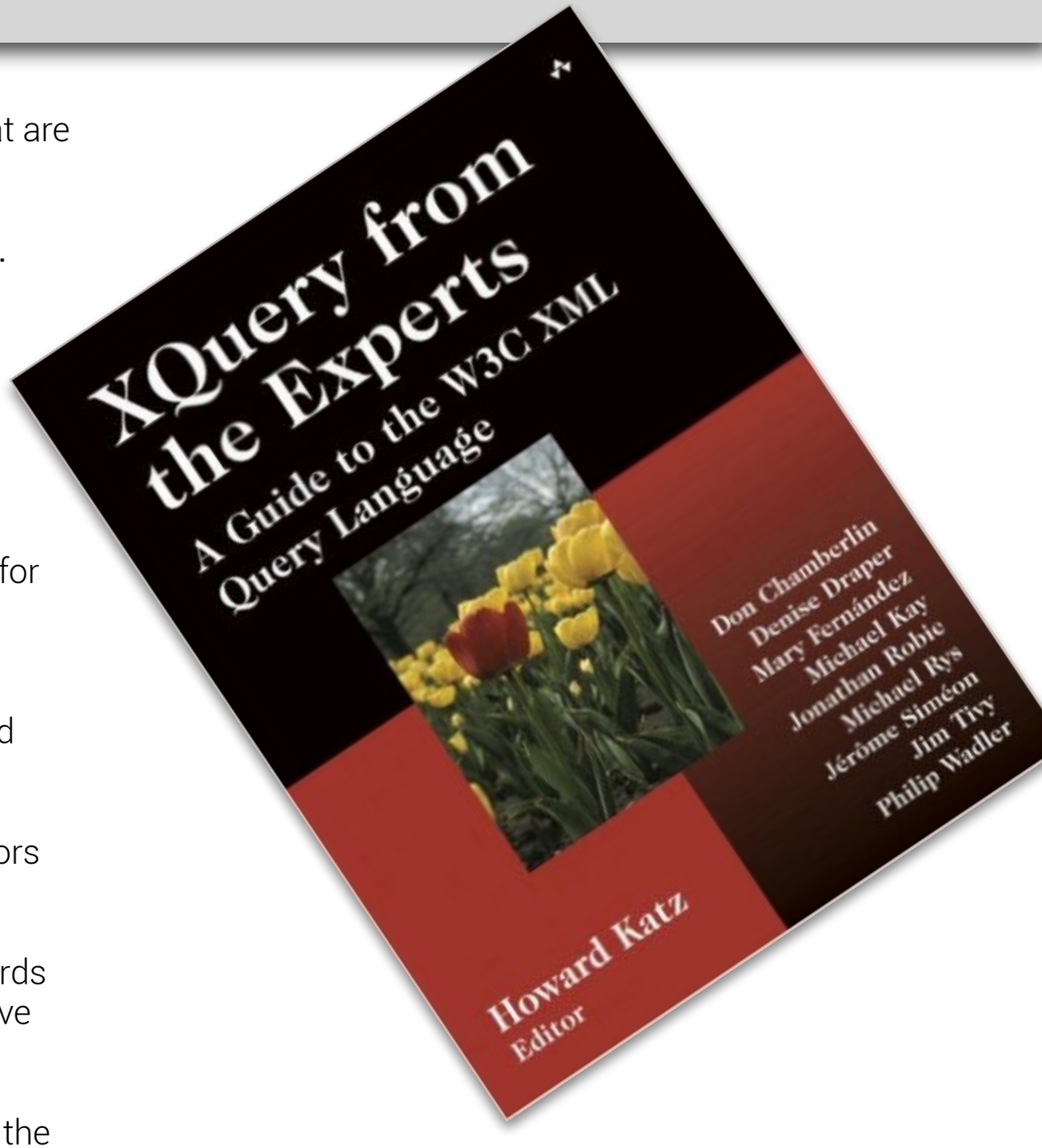
Acknowledgements [3]

- Some of these slides mostly contain text, examples and materials that are most often taken *verbatim* from the book:
  - XQuery. Priscilla Walmsley. O'Reilly, 2007. ISBN 978-0-596-00634-1. http://www.datypic.com/books/xquery/ © Priscilla Walmsley 2007
- These slides were put together from the above publication for educational purposes only.
- Any changes made either reflect updates to XQuery since publication or are the result of minor editing, adaptation and extension for use in teaching.
- The author of these slides is solely responsible for any errrors they contain.
- No claims are made by the author of these slides with regards to any form of intellectual property rights wherever the above author is acknowledged.
- The author of these slides is very grateful to the author of the work above.



Acknowledgements [4]

- Some of these slides mostly contain text and examples that are most often taken *verbatim* from the book:
  - XQuery from the Experts. D. Chamberlin, D. Draper, M. Fernández et al. Addison-Wesley, 2004. ISBN 0-321-18060-7. © Pearson Education Limited 2004
- In particular, from Chapter 1
  - XQuery: A Guided Tour. Jonathan Robie.
- These slides were put together from the above publication for educational purposes only.
- Any changes made either reflect updates to XQuery since publication or are the result of minor editing, adaptation and extension for use in teaching.
- The author of these slides is solely responsible for any errrors they contain.
- No claims are made by the author of these slides with regards to any form of intellectual property rights wherever the above authors are acknowledged.
- The author of these slides is very grateful to the authors of the works mentioned above.



The XQuery Data Model

Data: XML/Tree v. Relational/Tabular Form [1]

- XML data are different from relational data in several important respects that influence the design of a query language.
- For reasons including the ones below, existing relational query languages are not directly suitable for querying XML data.

- Relational data tend to have a regular structure, which allows the descriptive metadata for these data to be stored in a separate catalog, whereas XML data are often quite heterogeneous, and distribute their metadata throughout the document.

from: XQuery: An XML query language. Donald D. Chamberlin. IBM Systems Journal 41(4): 597-615 (2002)

Data: XML/Tree v. Relational/Tabular Form [2]

- XML documents often contain many levels of nested elements, whereas relational data are flat (in the sense of Codd's first-normal form constraint).

- XML documents have an intrinsic order, whereas relational data are unordered except where an ordering can be derived from data values.

- Relational data are usually dense (in the sense that nearly every column has a value), and relational systems often represent missing information by a special null value, whereas XML data are often sparse and can represent missing information simply by the absence of an element.

from: XQuery: An XML query language. Donald D. Chamberlin. IBM Systems Journal 41(4): 597-615 (2002)

Data Model [1]

- XQuery is defined in terms of a formal data model, not in terms of XML documents (or text).
- As in the relational languages, underlying an XQuery there is a guarantee of closure, in algebraic terms.
- Every input to a query is an instance of the data model and the output of every query is an instance of the data model.

Data Model [2]

- The data model provides an abstract representation of one or more XML documents or document fragments.
- The data model is based on the notion of a *sequence*.
- A sequence is an ordered collection of zero or more items.
- An item may be a node or an atomic value.
- An atomic value is an instance of one of the built-in data types defined by XML Schema, such as strings, integers, decimals, and dates.

from: XQuery: An XML query language. Donald D. Chamberlin. IBM Systems Journal 41(4): 597-615 (2002)

Data Model [3]

- A node conforms to one of several kinds, which are:
  - document
  - element
  - attribute
  - namespace
  - processing instruction
  - comment
  - text
- A node may have other nodes as children, thus forming one or more node hierarchies.

from: XQuery: An XML query language. Donald D. Chamberlin. IBM Systems Journal 41(4): 597-615 (2002)

Data Model [4]

- Some kinds of nodes, such as element and attribute nodes, have names or typed values, or both.
- A typed value is a sequence of zero or more atomic values.
- Nodes have identity (i.e., two nodes may be distinguishable even though their names and values are the same)
- Atomic values do not have identity.

from: XQuery: An XML query language. Donald D. Chamberlin. IBM Systems Journal 41(4): 597-615 (2002)

Data Model [5]

- Among all the nodes in a hierarchy there is a total ordering called *document order*, in which each node appears before its children.
- Document order corresponds to the order in which the nodes would appear if the node hierarchy were represented in XML format.

from: XQuery: An XML query language. Donald D. Chamberlin. IBM Systems Journal 41(4): 597-615 (2002)

Data Model [6]

- The node in any document that appears first in that document's document order is the document node, which contains the entire document.
- The document node does not correspond to anything visible in the document, rather it represents the document itself.
- Then, element, comment and processing instruction nodes appear in the order they occur in the document.

from: XQuery: A Guided Tour. Jonathan Robie. Ch. 1 in (D. Chamberlin, D. Draper, M. Fernández et al., 2004)

Data Model [7]

- Element nodes appear before their children nodes.
- Attributes are not considered children of the element they pertain to, but they appear after that element and before the element's children.
- The relative order of attribute nodes is implementation-dependent.
- In document order, each node appears exactly once, so that sorting nodes in document order causes duplicate removal.

from: XQuery: A Guided Tour. Jonathan Robie. Ch. 1 in (D. Chamberlin, D. Draper, M. Fernández et al., 2004)

Data Model [8]

- To see the document order in the printed form of an XML document, one marks the first character of each element's start tag, or attribute name, or comment or processing instruction, or text node.
- If the first character of one node N occurs before the first character of another node N′, N will appear before N′ in document order.
- In the example below, the relevant first characters are in red and underlined.

```
<?xml version="1.0"?>
<!-- books.xml -->
<book year="1999">
        <title>The Economics of Technology and Content for Digital TV</title>
        <editor>
           <last>Gerbarg</last>
           <first>Darcy</first>
           <affiliation>CITI</affiliation>
        </editor>
        <publisher>Kluwer Academic Publishers</publisher>
        <price>129.95</price>
    </book>
```

from: XQuery: A Guided Tour. Jonathan Robie. Ch. 1 in (D. Chamberlin, D. Draper, M. Fernández et al., 2004)

Data Model [9]

- Document order between nodes in different hierarchies is implementation-defined but must be consistent, i.e., all the nodes in one hierarchy must be ordered either before or after all the nodes in another hierarchy.
- Sequences may be heterogeneous, i.e., they may contain mixtures of various types of nodes and atomic values.
- However, a sequence never appears as an item in another sequence, i.e., sequences are flat.

from: XQuery: An XML query language. Donald D. Chamberlin. IBM Systems Journal 41(4): 597-615 (2002)

Data Model [10]

- All operations that create sequences are defined to flatten their operands so that the result of the operation is a single-level (i.e., flat) sequence.
- There is no distinction between an item and a sequence of length one, i.e., a node or atomic value is considered to be identical to a sequence of length one containing that node or atomic value.

from: XQuery: An XML query language. Donald D. Chamberlin. IBM Systems Journal 41(4): 597-615 (2002)

Data Model [11]

- Sequences of length zero are valid and are sometimes used to represent missing or unknown information, in much the same way that null values are used in relational.
- In addition to sequences, the query data model defines a special value called the *error value*, which is the result of evaluating an expression that contains an error.

from: XQuery: An XML query language. Donald D. Chamberlin. IBM Systems Journal 41(4): 597-615 (2002)

Data Model [12]

- An error value may not be combined in a sequence with any other value.
- Input XML documents can be transformed into the query data model by a process called *schema validation*.
- This process parses the document, validates it against a possibly particular schema, and represents it as a hierarchy of nodes and atomic values, labeled with possibly particular type information derived from the schema.

from: XQuery: An XML query language. Donald D. Chamberlin. IBM Systems Journal 41(4): 597-615 (2002)

Data Model [13]

- If an input document does not have a particular schema, it is validated against a permissive default schema that assigns generic types — nodes are labeled **`anyType`** and atomic values are labeled **`anySimpleType`**.
- The result of a query may be transformed from the query data model into an XML representation by a process called *serialization*.

from: XQuery: An XML query language. Donald D. Chamberlin. IBM Systems Journal 41(4): 597-615 (2002)

Data Model [14]

- It is worth noting that the result of a query is not always a well-formed XML document.

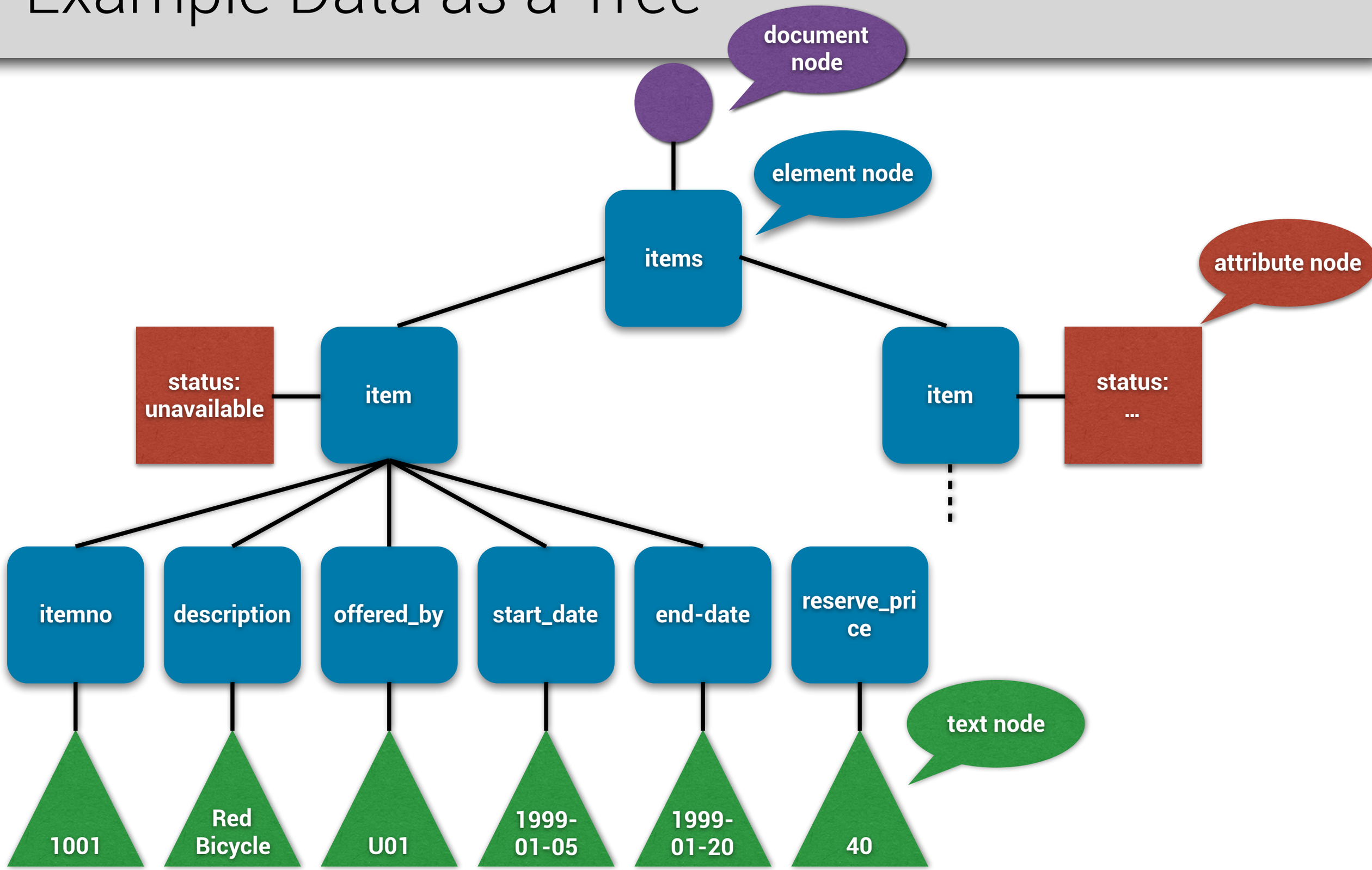
- For example, a query might return an atomic value such as the number **47**, or a sequence of elements with no common parent.

from: XQuery: An XML query language. Donald D. Chamberlin. IBM Systems Journal 41(4): 597-615 (2002)

Example Data as a Text: `items.xml`

```
<items>
  <item status="unavailable">
    <itemno>1001</itemno>
    <description>Red Bicycle</description>
    <offered_by>U01</offered_by>
    <start_date>1999-01-05</start_date>
    <end_date>1999-01-20</end_date>
    <reserve_price>40</reserve_price>
  </item>

<item status="available">
  <itemno>1002</itemno>
  <description>Motorcycle</description>
  <offered_by>U02</offered_by>
  <start_date>1999-02-1</start_date>
  <end_date>1999-03-15</end_date>
  <reserve_price>500</reserve_price>
</item>

<item status="available">
  <itemno>1003</itemno>
  <description>Old Bicycle</description>
  <offered_by>U02</offered_by>
  <start_date>1999-01-10</start_date>
  <end_date>1999-02-20</end_date>
  <reserve_price>25</reserve_price>
</item>

  <item status="unavailable">
    <itemno>1004</itemno>
    <description>Tricycle</description>
    <offered_by>U01</offered_by>
    <start_date>1999-02-25</start_date>
    <end_date>1999-03-08</end_date>
    <reserve_price>15</reserve_price>
  </item>

<!-- 8< ... snip ... 8< -->
</items>
```

Example Data as a Tree



from: XQuery: An XML query language. Donald D. Chamberlin. IBM Systems Journal 41(4): 597-615 (2002)

More Example Data: `bids.xml`

```
<bids>
  <bid>
    <userid>U02</userid>
    <itemno>1001</itemno>
    <bid_amount>35</bid_amount>
    <bid_date>1999-01-07</bid_date>
  </bid>

  <bid>
    <userid>U04</userid>
    <itemno>1001</itemno>
    <bid_amount>40</bid_amount>
    <bid_date>1999-01-08</bid_date>
</bid>

<bid>
  <userid>U02</userid>
  <itemno>1001</itemno>
  <bid_amount>45</bid_amount>
  <bid_date>2000-01-11</bid_date>
</bid>

<bid>
  <userid>U04</userid>
  <itemno>1001</itemno>
  <bid_amount>50</bid_amount>
  <bid_date>1999-01-13</bid_date>
</bid>

<bid>
  <userid>U02</userid>
  <itemno>1001</itemno>
  <bid_amount>55</bid_amount>
  <bid_date>1999-01-15</bid_date>
</bid>

<!-- 8< ... snip ... 8< -->
</bids>
```

MANCHESTER 1824

The University of Manchester

XQuery Basics

XQuery: Basics [1]

- XQuery is case-sensitive and keywords are in lowercase.

```
[wrong] FOR $x in 1 to 10 …

[wrong] for $x IN 1 to 10 …

[correct] for $x in 1 to 10 …

[different] for $X in 1 to 10 …
```

from: XQuery: An XML query language. Donald D. Chamberlin. IBM Systems Journal 41(4): 597-615 (2002)

XQuery: Basics [2]

- Characters enclosed between "`(:`" and "`:)`" are considered to be comments and are ignored during query processing (except, of course, inside a quoted string, where they are considered to be part of the string).
- Of course, XQuery comments in a query are not to be confounded with an XML comment in a document.

```
for $x      (: iterate over deciles :)
 in 1 to 10
…
```

XQuery: Basics [3]

- XQuery is a functional language, which means that it is made up of expressions that return values and do not have side effects.
- XQuery has several kinds of expressions, most of which are composed from lower-level expressions, combined by operators or keywords.

from: XQuery: An XML query language. Donald D. Chamberlin. IBM Systems Journal 41(4): 597-615 (2002)

XQuery: Basics [4]

- XQuery expressions are fully composable, i.e., where an expression is expected, any suitably typed expression may be used.

- The value of an expression, in general, is a heterogeneous sequence of nodes and atomic values.

from: XQuery: An XML query language. Donald D. Chamberlin. IBM Systems Journal 41(4): 597-615 (2002)

XQuery Literals

Expressions: Literals

- The simplest kind of XQuery expression is literal, which represents an atomic value.

- The following are several examples of literals:
  - `47` is a literal of type **`integer`**
  - `4.7` is a literal of type **`decimal`** because it contains a decimal point
  - `4.7E3` is a literal of type **`double`** because it contains an exponent
  - `"47"` is a literal of type **`string`** (single quotes are allowed inside double-quoted strings)
  - `'47'` is a literal of type **`string`** (double quotes are allowed inside single-quoted strings)

from: XQuery: An XML query language. Donald D. Chamberlin. IBM Systems Journal 41(4): 597-615 (2002)

Expressions: Complex Literals

- Atomic values of other types may be created by calling constructors.
- A *constructor* is a function that creates a value of a particular type from a string containing a lexical representation of the desired type.
- In general, a constructor has the same name as the type it constructs.
- The following example uses a constructor to create a value of type date:

```
xs:date("1999-12-31")
```

from: XQuery: An XML query language. Donald D. Chamberlin. IBM Systems Journal 41(4): 597-615 (2002)

Expressions: Complex Literals [1]

- Any XQuery expression may be enclosed in parentheses.
- Parentheses are useful (and sometimes mandatory) for making explicit the order in which an expression should be evaluated.
- The following examples of arithmetic expressions show how parentheses can be used to control the precedence of operators.
- `(2 + 4) * 5` has the value `30` because the sub-expression `(2 + 4)` is evaluated first
- `2 + 4 * 5` has the value `22` because `*` has a higher precedence than `+`

from: XQuery: An XML query language. Donald D. Chamberlin. IBM Systems Journal 41(4): 597-615 (2002)

Expressions: Complex Literals [2]

- The comma operator concatenates two values to form a sequence.
- Sequences are often enclosed in parentheses as explicit delimiters, although this is not required.
- An empty pair of parentheses denotes the empty sequence.
- Since sequences cannot be nested, the comma operator constructs a sequence consisting of all the items in its left operand, followed by all the items in its right operand.

from: XQuery: An XML query language. Donald D. Chamberlin. IBM Systems Journal 41(4): 597-615 (2002)

Expressions: Complex Literals [3]

- A sequence can also be constructed by the **`to`** operator, which returns a sequence consisting of all the integers between its left operand and its right operand, inclusive.
- The following examples illustrate construction of sequences:

- `1,2,3` is a sequence of three values: `1` followed by `2` followed by `3`
- `(1,2,3)` is identical to the sequence `1,2,3`
- `((1,2),(),3)` is identical to the sequence `1,2,3` because sequences are flattened
- `1 to 3` is identical to the sequence `1,2,3`

from: XQuery: An XML query language. Donald D. Chamberlin. IBM Systems Journal 41(4): 597-615 (2002)

XQuery Variables

Expressions: Variables [1]

- A variable in XQuery is a name that begins with a *dollar sign*.
- A variable may be bound to a value and used in an expression to represent that value.
- One way to bind a variable is by means of a `let` expression, which binds one or more variables and then evaluates an inner expression.

from: XQuery: An XML query language. Donald D. Chamberlin. IBM Systems Journal 41(4): 597-615 (2002)

Expressions: Variables [2]

- The value of the `let` expression is the result of evaluating the inner expression with the variables bound.
- The following example illustrates a `let` expression that returns the sequence `1,2,3`:

```
let   $start:=1, $stop:= 3
return $start to $stop
```

- A `let` expression is a special case of a FLWOR (`for`, `let`, `where`, `order by`, `return`) expression, which provides additional ways to bind variables.

MANCHESTER 1824

The University of Manchester

XQuery Function Calls

Expressions: Function Calls [1]

- Another simple form of XQuery expression is a *function call*.
- XQuery provides a core function library and a mechanism whereby users can define additional functions.
- Function calls in XQuery employ the usual notation in which the arguments of the function are enclosed in parentheses.

from: XQuery: An XML query language. Donald D. Chamberlin. IBM Systems Journal 41(4): 597-615 (2002)

Expressions: Function Calls [2]

- The following example calls the core library function **`substring`** to extract the first six characters from a string:

```
substring("Martha Washington", 1, 6)
```

MANCHESTER 1824

The University of Manchester

XQuery Path Expressions

Path Expressions [1]

- Path expressions in XQuery are based on the syntax of XPath.
- A path expression consists of a series of *steps*, separated by the slash character (/).
- The result of each step is a sequence of nodes.
- The value of the path expression is the node sequence that results from the last step in the path.
- Each step is evaluated in the context of a particular node, called the *context node*.

from: XQuery: An XML query language. Donald D. Chamberlin. IBM Systems Journal 41(4): 597-615 (2002)

Path Expressions [2]

- In general, a step can be any expression that returns a sequence of nodes.
- One important kind of step, called an *axis step*.
- It can be thought of as beginning at the context node and moving through the node hierarchy in a particular direction, called an *axis*.
- As the axis step moves along the designated axis, it selects nodes that satisfy a selection criterion.

from: XQuery: An XML query language. Donald D. Chamberlin. IBM Systems Journal 41(4): 597-615 (2002)

Path Expressions [3]

- The selection criterion can select nodes based on their names, their positions with respect to the context node, or a predicate based on the value of a node.

- XPath defines 13 axes, but XQuery does not support the namespace XPath axis.

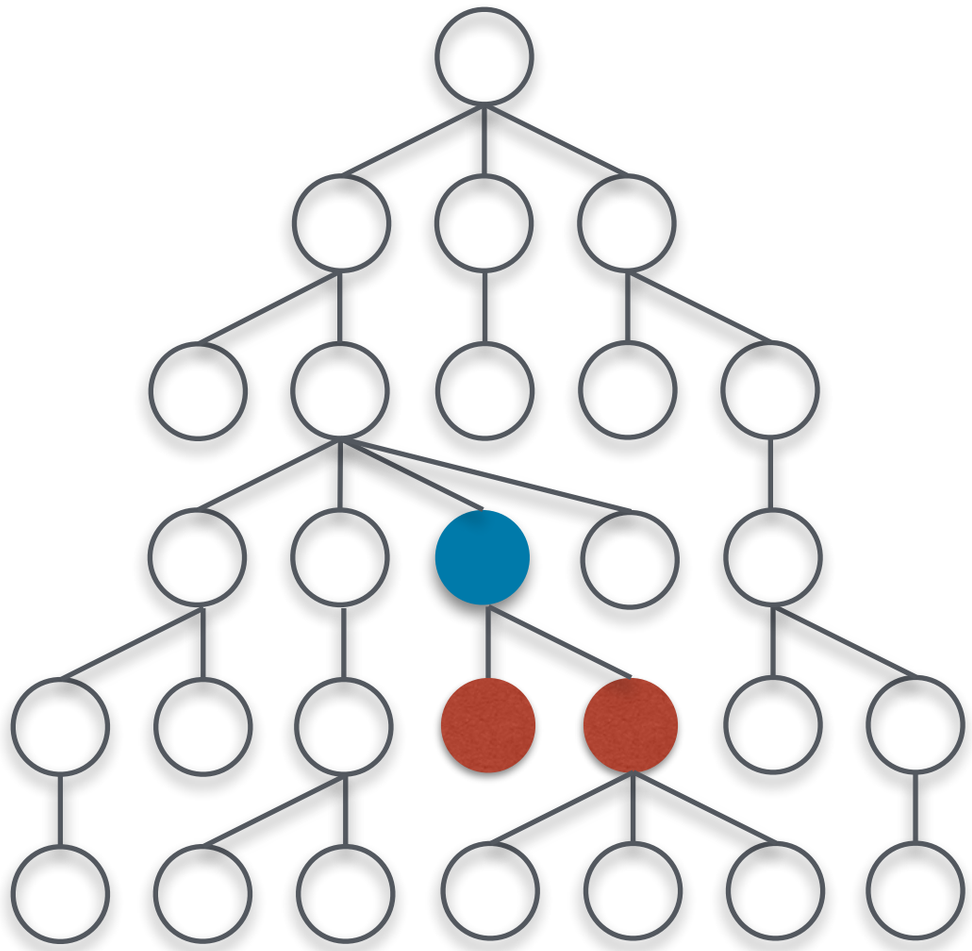
from: XQuery: An XML query language. Donald D. Chamberlin. IBM Systems Journal 41(4): 597-615 (2002)

Path Expressions [4]

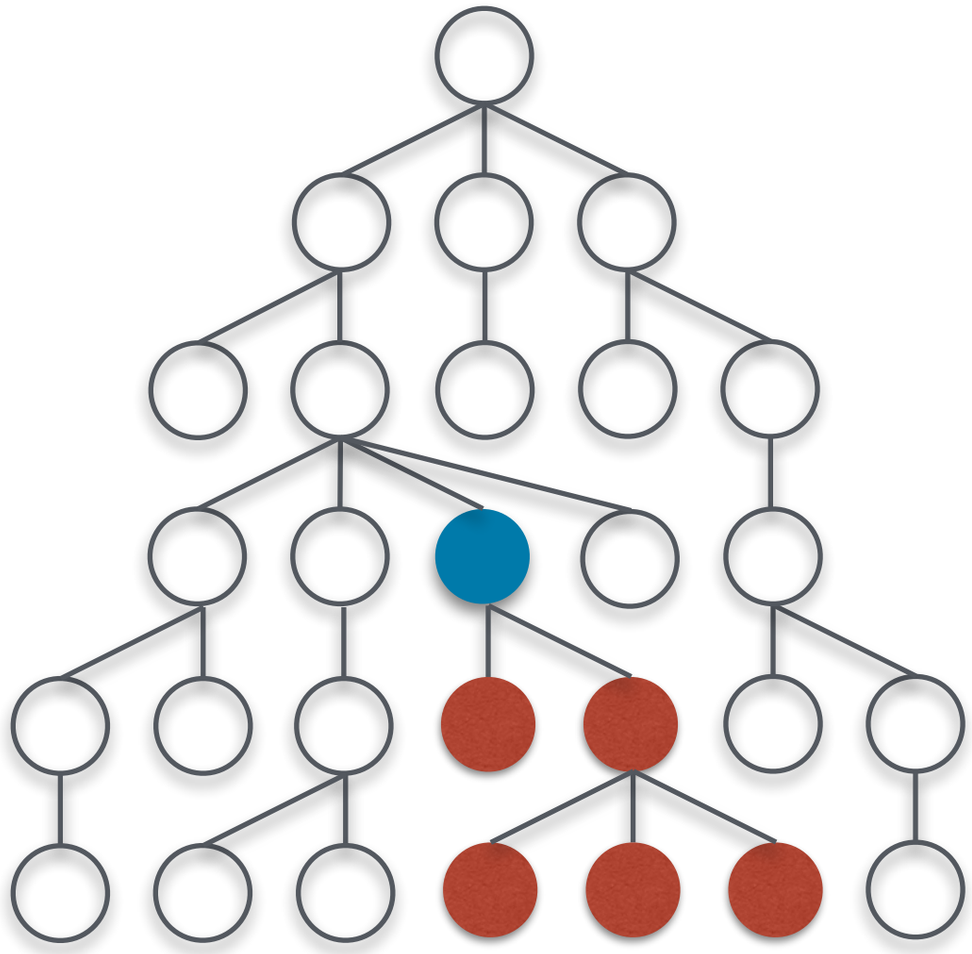
- XQuery supports the following *forward* axes:
  - `child`
  - `descendant`
  - `attribute`
  - `self`
  - `descendant-or-self`
  - `following-sibling`
  - `following`

from: XQuery: An XML query language. Donald D. Chamberlin. IBM Systems Journal 41(4): 597-615 (2002)

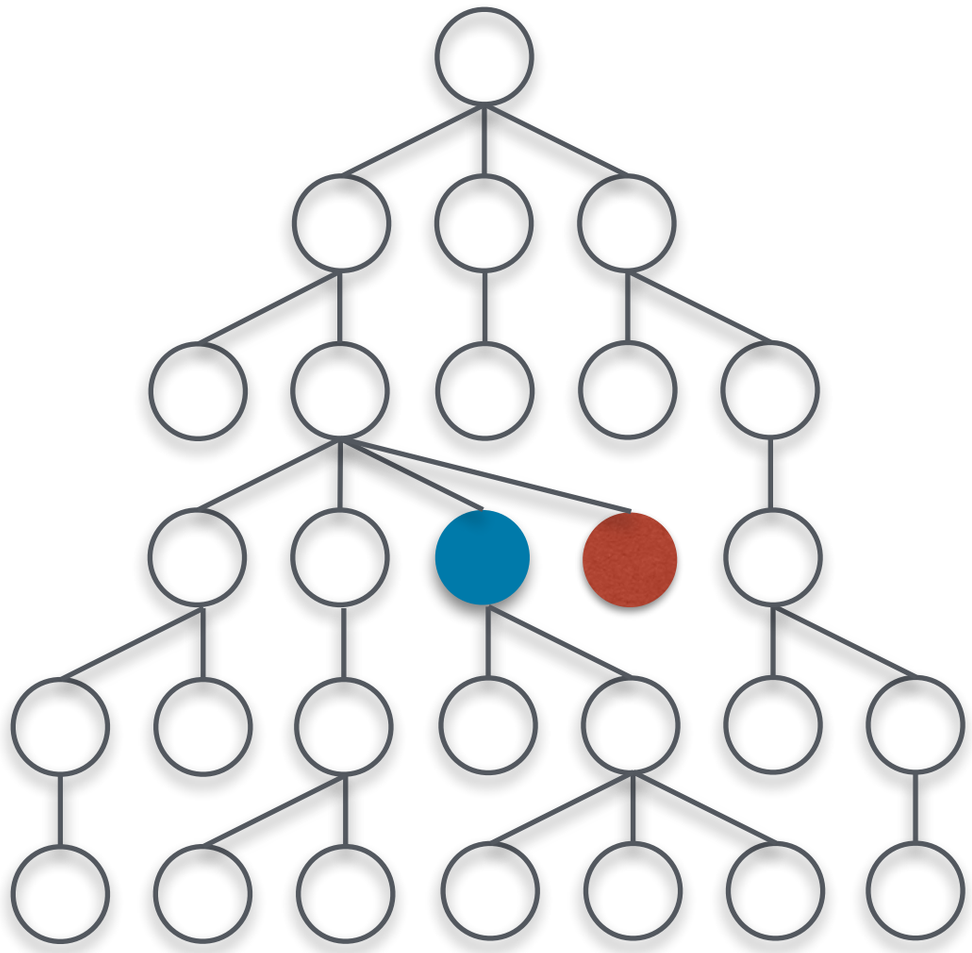
The `child` Axis



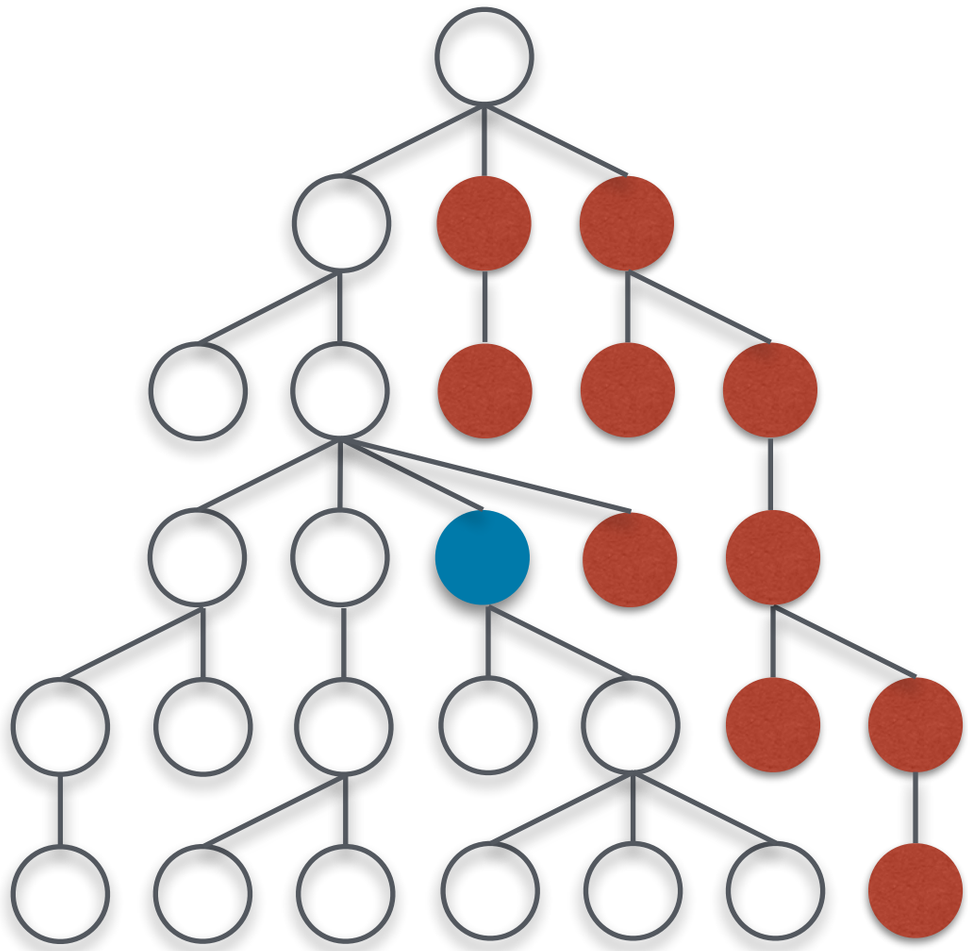
The `descendant` Axis



The `following-sibling` Axis



The `following` Axis

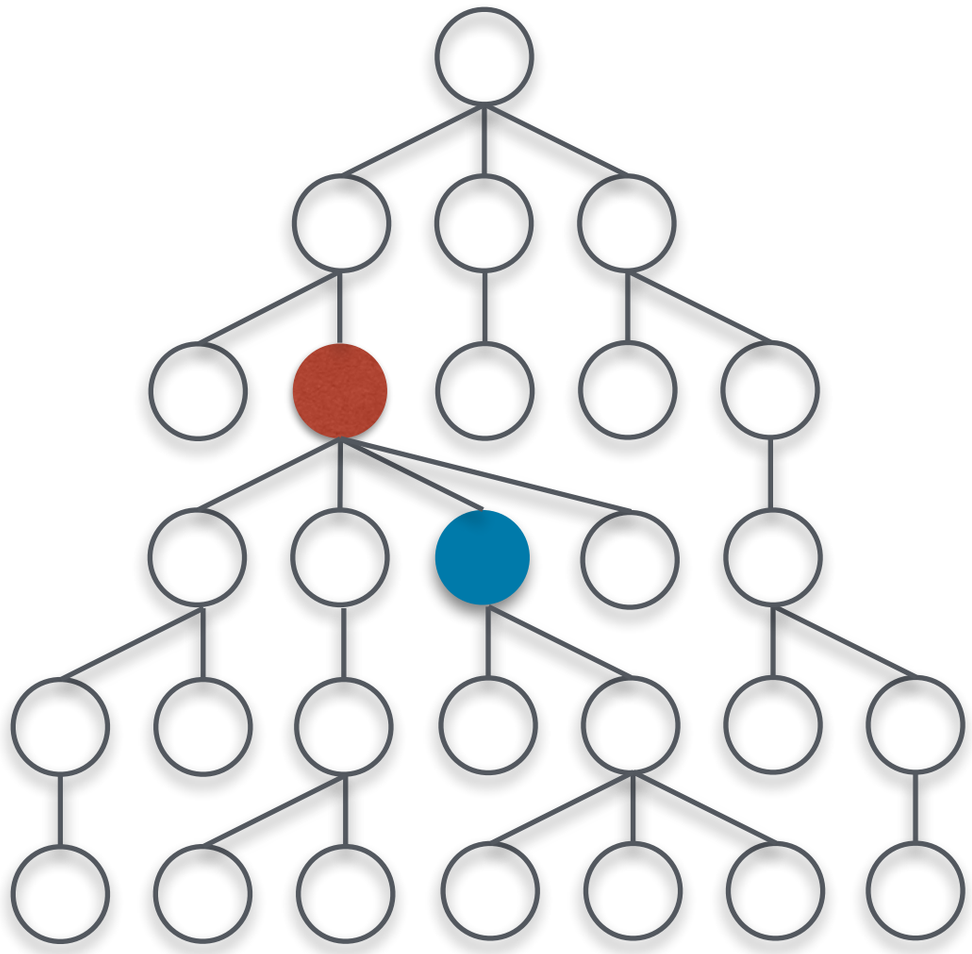


Path Expressions [5]

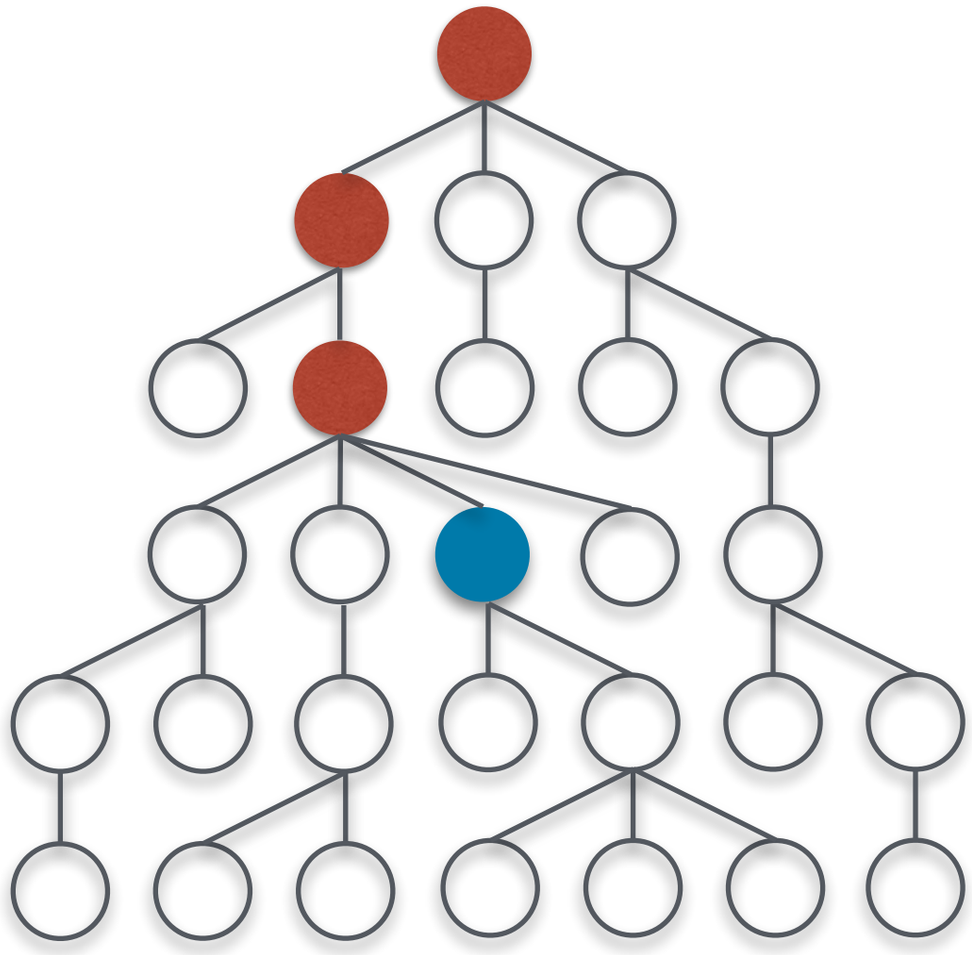
- XQuery supports the following *reverse* axes:
  - `parent`
  - `ancestor`
  - `preceding-sibling`
  - `preceding`
  - `ancestor-or-self`

from: XQuery: An XML query language. Donald D. Chamberlin. IBM Systems Journal 41(4): 597-615 (2002)

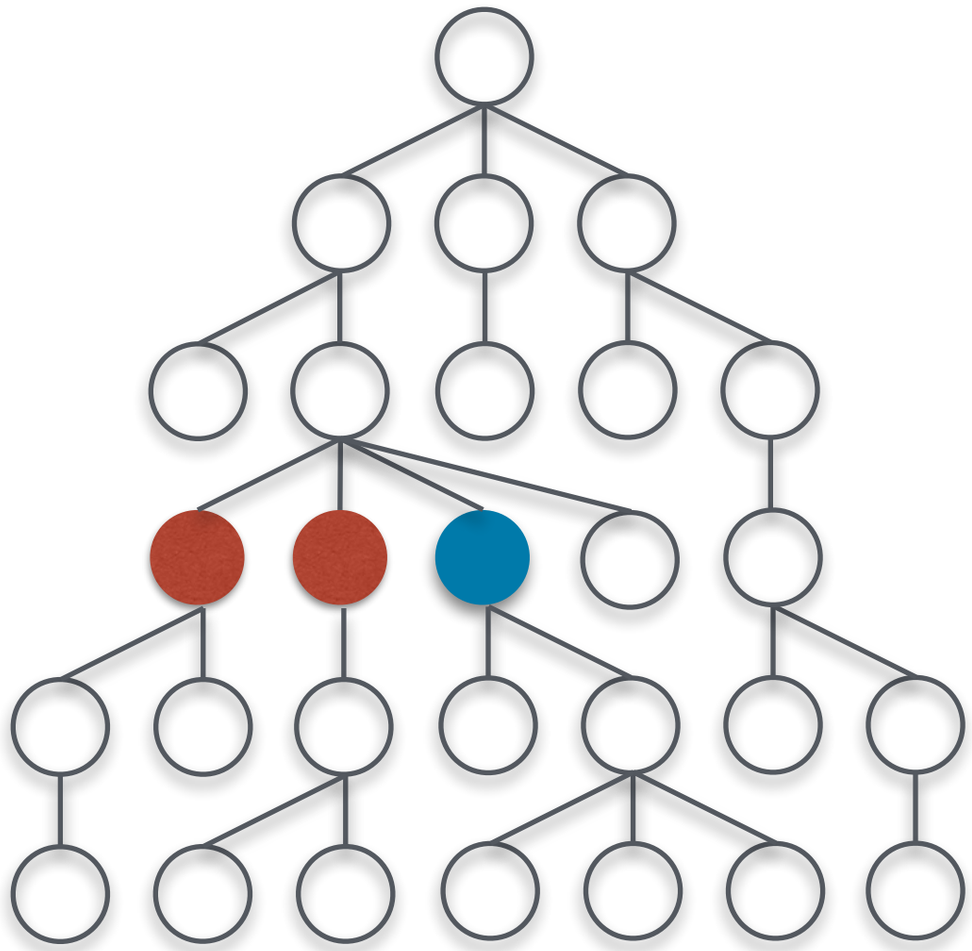
The `parent` Axis



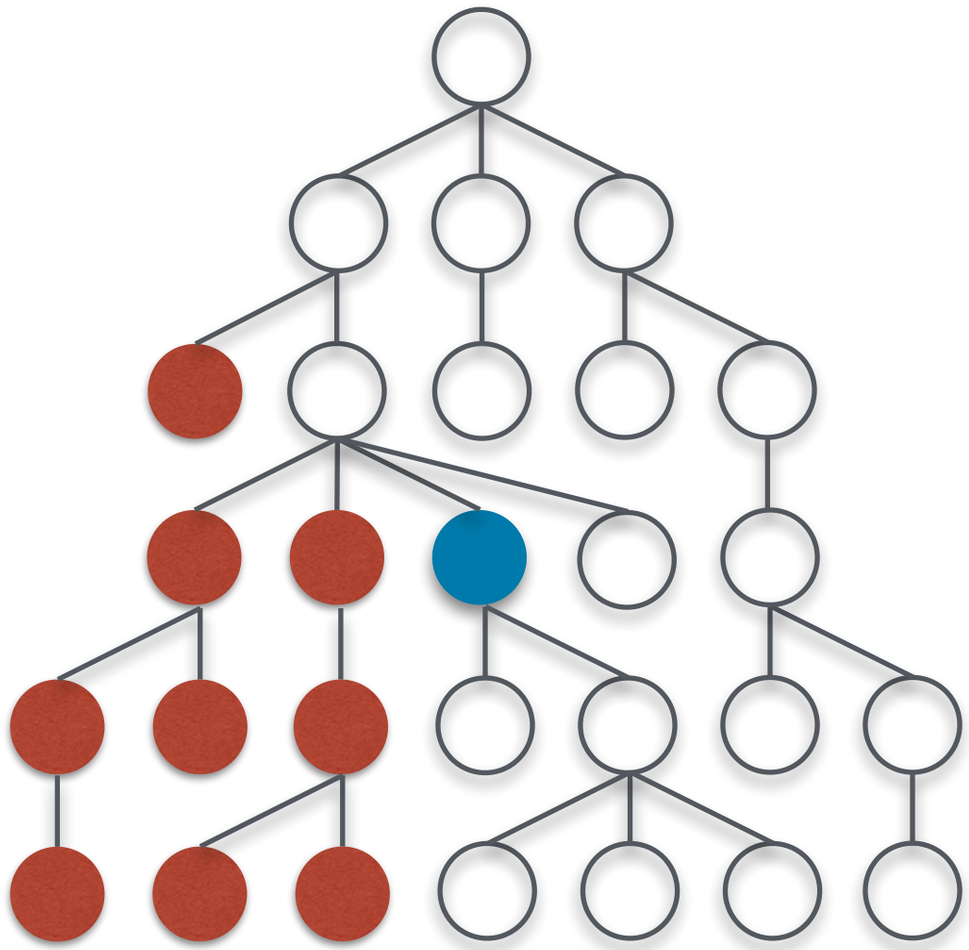
The `ancestor` Axis



The `preceding-sibling` Axis



The `preceding` Axis



Path Expressions [6]

- As a path expression is evaluated, the nodes selected by each step serve in turn as context nodes for the following step.
- If a step has several context nodes, it is evaluated for each of the context nodes in turn.
- The resulting node sequences are combined by the union operator to form the result of the step.
- The result of a step is always a sequence of distinct nodes (without duplicates based on node identity), in document order.

from: XQuery: An XML query language. Donald D. Chamberlin. IBM Systems Journal 41(4): 597-615 (2002)

Path Expression Examples [1]

- Path expressions may be written in either unabbreviated syntax or abbreviated syntax.
- The unabbreviated syntax for an axis step consists of an axis and a selection criterion, separated by two colons.
- Q1, in the next slide, illustrates a four-step path expression using unabbreviated syntax.

from: XQuery: An XML query language. Donald D. Chamberlin. IBM Systems Journal 41(4): 597-615 (2002)

Path Expression Examples [2]

Q1
(unabbreviated)

(Q1) List the descriptions of all items offered for sale by seller U03.

```
doc("items.xml")/child::*/child::item[child::offered_by="U03"]/child::description
```

Path Expression Examples [3]

- The first step invokes the built-in `doc` function, which returns the document node for the document named `items.xml`.
- The second step is an axis step that finds all children of the document node (`*` selects all nodes on the given axis, which in this case is only a single element node named `items`).
- The third step follows the `child` axis again to find all the child elements at the next level that are named `item` and that have a child named `offered_by` with value `U03` (i.e., the user named `"Dee Linquent"`).

from: XQuery: An XML query language. Donald D. Chamberlin. IBM Systems Journal 41(4): 597-615 (2002)

Path Expression Examples [4]

- The result of the third step is a sequence of **`item`** element nodes.
- Each of these **`item`** nodes is used in turn as the context node for the fourth step, which follows the **`child`** axis again to find the **`description`** elements that are children of the given **`item`**.
- The final result of the path expression is the result of the fourth step: a sequence of **`description`** element nodes, in document order.

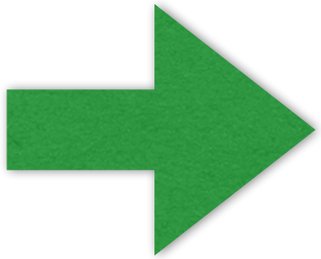
Path Expression Examples [5]

- In practice, path expressions are usually written using abbreviated syntax.
- Several kinds of abbreviations are provided.
- Perhaps the most important of these is that the axis specifier may be omitted when the child axis is used.
- Since child is the most commonly used axis, this abbreviation is helpful in reducing the length of many path expressions.
- The next slides describe some valid abbreviations.

Abbreviations [1]

'child::' is the default axis

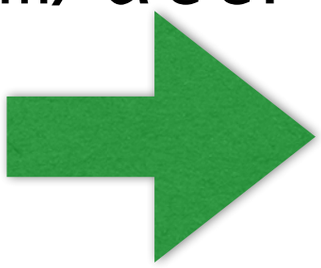
```
$doc/child::items/child::item/child::description
```



```
$doc/items/item/description
```

'attribute::' reduces to '@'

```
$doc/items/item/attribute::status
```



```
$doc/items/item/@status
```

Abbreviations [2]

'descendant-or-self::node()' reduces to the empty string

```
$doc/descendant-or-self::node()/*   ⇒   $doc//*
```

'self::node()' reduces to '.'

```
$doc/self::node()   ⇒   $doc/.
```

'parent::node()' reduces to '..'

```
$doc/items/item/parent::node()   ⇒   $doc/items/item/..
```

Path Expression Examples [6]

(Q1) List the descriptions of all items offered for sale by seller U03.

Q1 (abbreviated)

```
doc("items.xml")/*/item[offered_by="U03"]/description
```

Path Expression Examples [7]

- When two steps are separated by a double slash rather than by a single slash, it means that the second step may traverse multiple levels of the hierarchy, using the **`descendant`** axis rather than the single-level **`child`** axis.
- The result of Q2, shown in the next slide, is a sequence of element nodes that could, in principle, have been found at various levels of the node hierarchy (though, in our sample document **`items.xml`**, all description nodes are found at the same level).

from: XQuery: An XML query language. Donald D. Chamberlin. IBM Systems Journal 41(4): 597-615 (2002)

Path Expression Examples [8]

(Q2) List all description elements found in the document items.xml.

```
doc("items.xml")//description
```

Path Expression Examples [9]

- Within a path expression, a single dot (`.`) refers to the context node, and two consecutive dots (`..`) refer to the parent of the context node.
- These notations are abbreviated invocations of the **`self`** and **`parent`** axes, respectively.
- Names found in path expressions are usually interpreted as names of element nodes.

from: XQuery: An XML query language. Donald D. Chamberlin. IBM Systems Journal 41(4): 597-615 (2002)

Path Expression Examples [10]

- However, a name prefixed by the `@` character is interpreted as the name of an attribute node.
- This is an abbreviation for a step that traverses the **`attribute`** axis.
- These abbreviations are illustrated by Q3 (in the next slide).
- It iterates over the **`description`** nodes returned by Q2, and binds them to the variable **`$description`**, then traverses the **`parent`** axis to the parent **`item`** node, and then traverses the **`attribute`** axis to find an attribute named **`status`**.
- The result of Q3 is a sequence of attribute nodes.

from: XQuery: An XML query language. Donald D. Chamberlin. IBM Systems Journal 41(4): 597-615 (2002)

Path Expression Examples [11]

(Q3) Find the `status` attribute of the item that is the parent of each description.

```
for    $description in doc("items.xml")//description
return $description/../@status
```

XQuery Predicates

Predicates [1]

- In XQuery, a *predicate* is an expression, enclosed in square brackets, that is used to filter a sequence of values.

- Predicates are often used in the steps of a path expression.

- For example, in the step `item[offered_by="U03"]`, the phrase `offered_by="U03"` is a predicate that is used to select certain item nodes and discard others.

- We will refer to the items in the sequence being filtered by a predicate as *candidate items*.

from: XQuery: An XML query language. Donald D. Chamberlin. IBM Systems Journal 41(4): 597-615 (2002)

Predicates [2]

- The predicate is evaluated for each candidate item, using the candidate item as the context item for evaluating the predicate expression.
- The term *context item* is a generalization of the term *context node* and may indicate either a node or an atomic value.
- Within a predicate expression, a single dot (`.`) indicates the context item.

from: XQuery: An XML query language. Donald D. Chamberlin. IBM Systems Journal 41(4): 597-615 (2002)

Predicates [3]

- Each candidate item is selected or discarded according to the following rules:
- **[R1]** If the predicate expression evaluates to a Boolean value, the candidate item is selected if the value of the predicate expression is true.
- This type of predicate is illustrated by the following example, which selects `item` nodes that have a `reserve-price` child node whose value is greater than 1000:

```
item[reserve_price>1000]
```

Predicates [4]

- **[R2]** If the predicate expression evaluates to a number, the candidate item is selected if its ordinal position in the list of candidate items is equal to the number.

- This type of predicate is illustrated by the following example, which selects the fifth item node on the child axis:

```
item[5]
```

from: XQuery: An XML query language. Donald D. Chamberlin. IBM Systems Journal 41(4): 597-615 (2002)

Predicates [5]

- **[R3]** If the predicate expression evaluates to an empty sequence, the candidate item is discarded, but if the predicate expression evaluates to a sequence containing at least one node, the candidate item is selected.

- This form of predicate can be used to test for the existence of a child node that satisfies some condition.

from: XQuery: An XML query language. Donald D. Chamberlin. IBM Systems Journal 41(4): 597-615 (2002)

Predicates [6]

- This is illustrated by the following example, which selects item nodes that have a **`reserve_price`** child node, regardless of its value:

```
item[reserve_price]
```

- Several different kinds of operators and functions are often used inside predicates.
- In the following few slides, some of the most common and most useful of these operators and functions are briefly described.

XQuery Comparison Operators

Comparison Operators: Value Comparison [1]

- The value comparison operators are:
  - eq
  - ne
  - lt
  - le
  - gt
  - ge

from: XQuery: An XML query language. Donald D. Chamberlin. IBM Systems Journal 41(4): 597-615 (2002)

Comparison Operators: Value Comparison [2]

- These operators can compare two scalar values, but they raise an error if either operand is a sequence of length greater than one.

- If either operand is a node, the value comparison operator extracts its value before performing the comparison.

- For example, `item[reserve_price gt 1000]` selects an `item` node if it has exactly one `reserve_price` child node whose value is greater than 1000.

Comparison Operators: General Comparison [1]

- The general comparison operators are:
  - =
  - !=
  - <
  - <=
  - >
  - >=
- These operators can deal with operands that are sequences, providing implicit existential semantics for both operands.

from: XQuery: An XML query language. Donald D. Chamberlin. IBM Systems Journal 41(4): 597-615 (2002)

Comparison Operators: General Comparison [2]

- Like the value comparison operators, the general comparison operators automatically extract values from nodes.
- For example, `item[reserve_price >1000]` selects an `item` node if it has at least one `reserve_price` child node whose value is greater than 1000.
- The existential semantics is expressed by the phrase at least one.

from: XQuery: An XML query language. Donald D. Chamberlin. IBM Systems Journal 41(4): 597-615 (2002)

Comparison Operators: Node Comparison

- The node comparison operators are:
  - `is`
  - `isnot`
- These operators can deal with operands that are sequences, providing implicit existential semantics for both operands.
- These operators compare the identities of two nodes.
- For example, **`$node1`** `is` **`$node2`** is true if the variables **`$node1`** and **`$node2`** are bound to the same node (that is, the node identity is the same for both variables).

from: XQuery: An XML query language. Donald D. Chamberlin. IBM Systems Journal 41(4): 597-615 (2002)

Comparison Operators: Node Order Comparison

- The node order comparison operators are:
  - <<
  - >>
- These operators compare the positions (before, after) of two nodes.
- For example, **$node1<< $node2** is true if the node bound to **$node1** occurs earlier in document order than the node bound to **$node2**.

from: XQuery: An XML query language. Donald D. Chamberlin. IBM Systems Journal 41(4): 597-615 (2002)

Comparison Operators: Logical Comparison [1]

- The logical comparison operators are:
  - `and`
  - `or`
- These operators can be used to combine logical conditions inside a predicate.
- For example, the following predicate selects `item` nodes that have exactly one `offered_by` child element with the value `"U04"` and also have at least one `reserve_price` child element with any value:

```
item[offered_by eq "U04"
and reserve_price]
```

from: XQuery: An XML query language. Donald D. Chamberlin. IBM Systems Journal 41(4): 597-615 (2002)

Comparison Operators: Logical Comparison [2]

- Note that if the label of a node (e.g., **`reserve_price`**) appears where a Boolean value is expected, then it is interpreted as a test of the existence of a node with such a label.
- Note further that in XQuery there are no Boolean literals *true* and *false*, rather one has to use a function to construct then, i.e., a call to **`true()`** returns the Boolean value *true*, and a call to **`false()`** returns the Boolean value *false*.

Comparison Operators: Logical Comparison [3]

- Note finally that **`not`** is also a function rather than an operator.
- It serves to invert a Boolean value, turning *true* into *false* and *false* into *true*.
- The following step uses the **`not`** function with an existence test to find **`item`** nodes that have no **`reserve_price`** child element:

```
item[not(reserve_price)]
```

XQuery Qualified Names

Qualified Names [1]

- In all of the above examples, element and attribute names have been simple identifiers.

- However, the XML Namespace recommendation allows elements and attributes to have two-part names in which the first part is a namespace prefix, followed by a colon.

- A name qualified by a namespace prefix is called a *QName*.

from: XQuery: An XML query language. Donald D. Chamberlin. IBM Systems Journal 41(4): 597-615 (2002)

Qualified Names [2]

- Each namespace prefix must be bound to a URI (Uniform Resource Identifier) that uniquely identifies a namespace.
- This convention allows each application to define names in its own namespace without danger of colliding with names defined by other applications, and it allows a query to unambiguously refer to names defined by various applications.

from: XQuery: An XML query language. Donald D. Chamberlin. IBM Systems Journal 41(4): 597-615 (2002)

Qualified Names [3]

- If the prefix **`auction`** were bound to the namespace URI of an on-line auction application, the step **`item[reserve-price>1000]`** might be written using QNames as follows:

```
auction:item[auction:reserve-price>1000]
```

- The process of binding a prefix to a namespace URI is described later.

- In most of our examples, we use one-part names rather than QNames.

- This use is realistic because XQuery provides a way to specify a default namespace for a query.

- Such use makes it unnecessary for a query to use QNames unless it needs to refer to names from multiple namespaces.

XQuery Element Constructors

Element Constructors [1]

- Path expressions are powerful, but they have an important limitation: they can only select existing nodes.
- A full query language needs a facility to construct new elements and attributes and to specify their contents and relationships.
- This facility is provided in XQuery by a kind of expression called an element constructor.
- The simplest kind of element constructor looks exactly like the XML syntax for the element to be created.

from: XQuery: An XML query language. Donald D. Chamberlin. IBM Systems Journal 41(4): 597-615 (2002)

Element Constructors [2]

- For example, the following expression constructs an element named **highbid** containing one attribute named **status** and two child elements named **itemno** and **bid_amount**:

```
<highbid status="pending">
   <itemno>4871</itemno>
   <bid_amount>250</bid_amount>
</highbid>
```

- In the example above, the values of the elements and attributes are constants.
- However, in many cases it is necessary to create an element or an attribute whose value is computed by some expression.

from: XQuery: An XML query language. Donald D. Chamberlin. IBM Systems Journal 41(4): 597-615 (2002)

Element Constructors [3]

- In this case, the expression is enclosed in curly braces to indicate that it is to be evaluated rather than treated as literal text.

- The expression is evaluated and replaced by its value in the element constructor.

Element Constructors [4]

- In the following example, the values of the elements and attributes are computed by expressions.

- The variables **`$s`**, **`$i`**, and **`$bids`** used in these expressions must be bound by some enclosing expression.

```
<highbid status="{$s}">
   <itemno>
      {$i}
   </itemno>
   <bid_amount>
      {max($bids[itemno=$i]/bid_amount)}
   </bid_amount>
</highbid>
```

from: XQuery: An XML query language. Donald D. Chamberlin. IBM Systems Journal 41(4): 597-615 (2002)

Element Constructors [5]

- The content of an element constructor may be any expression.
- In general, the expression used in an element constructor may generate a sequence of items, including atomic values, elements, and attributes.
- Attributes that are generated inside an element constructor become attached to the constructed element.
- Elements and atomic values that are generated inside an element constructor become the content of the constructed element.

from: XQuery: An XML query language. Donald D. Chamberlin. IBM Systems Journal 41(4): 597-615 (2002)

Element Constructors [6]

- In the following example, an element constructor contains an expression, enclosed in curly braces, that generates one attribute and two subelements.

- The variable **$b** must be bound by some enclosing expression.

```
<highbid>
  {$b/status,
   $b/itemno,
   $b/bid_amount
  }
</highbid>
```

from: XQuery: An XML query language. Donald D. Chamberlin. IBM Systems Journal 41(4): 597-615 (2002)

Element Constructors [7]

- The element node produced by an element constructor is a new node with its own node identity.
- If the newly constructed element has child nodes and attributes that are derived from existing nodes, as in the above example, the new child nodes and attributes are copies of the nodes from which they were derived, with their own node identities.

Element Constructors [8]

- In the above examples of element constructors, even though the content of the element may be computed, the name of the constructed element is a known constant.
- However, it is sometimes necessary to construct an element whose name as well as its content is computed.
- For this purpose, XQuery provides a special kind of constructor called a *computed element constructor*.

Element Constructors [9]

- A computed element constructor consists of the keyword element, followed by two expressions in curly braces:
  - the first expression computes the name of the element
  - the second expression computes the content of the element.
- For an example of the use of a computed constructor, suppose that the variable **$e** is bound to an element with a numeric value.
- We need to construct a new element that has the same name as **$e** and the same attributes as **$e**, but we want its value to be twice the value of **$e**.

Element Constructors [10]

- This construction can be accomplished by the following expression, which uses the data function to extract the numeric value of the original node:

```
element
   {name($e)}
   {$e/@*, data($e)*2}
```

- Similar to a computed element constructor, XQuery provides a *computed attribute constructor*, which consists of the keyword attribute, followed by two expressions in curly braces: the first expression computes the name of the attribute and the second expression computes its value.

from: XQuery: An XML query language. Donald D. Chamberlin. IBM Systems Journal 41(4): 597-615 (2002)

Element Constructors [11]

- An attribute constructor can be used anywhere an attribute is valid, for example, inside an element constructor.

```
attribute
   {"range"}
   {if (data($p)>100)
    then "expensive"
    else "cheap"}
```

- The following attribute constructor, based on binding `$p`, to `reserve_price` elements might generate an attribute that looks like `range="expensive"` or `range="cheap"`.
- This example uses a conditional (`if-then-else`) expression.

MANCHESTER 1824

The University of Manchester

XQuery Iteration and Sorting

Iteration and Sorting [1]

- Iteration is an important part of a query language.
- XQuery provides a way to iterate over a sequence of values, binding a variable to each of the values in turn and evaluating an expression for each binding of the variable.
- The simplest form of iteration in XQuery consists of a **`for`** clause that names a variable and provides a sequence of values over which the variable is to iterate, followed by a **`return`** clause that contains the expression to be evaluated for each variable binding.

from: XQuery: An XML query language. Donald D. Chamberlin. IBM Systems Journal 41(4): 597-615 (2002)

Iteration and Sorting [2]

- The following example illustrates this simple form of iteration:

```
for     $n in (2,3)
return $n+1
```

- The result of this simple iterative expression is the sequence `(3, 4)`.

- A `for` clause may specify more than one variable, with an iteration sequence for each variable.

- Such a `for` clause produces tuples of variable bindings that form the Cartesian product of the iteration sequences.

from: XQuery: An XML query language. Donald D. Chamberlin. IBM Systems Journal 41(4): 597-615 (2002)

Iteration and Sorting [3]

- Unless otherwise specified, the binding tuples are generated in an order that preserves the order of the iteration sequences, using the leftmost variable as the outer loop and the rightmost variable as the inner loop.
- The following example illustrates a `for` clause that contains two variables and two iteration sequences:

```
for $m in (2,3),
    $n in (5,10)
return
  <fact>
   {$m} times {$n} is {$m*$n}
  </fact>
```

from: XQuery: An XML query language. Donald D. Chamberlin. IBM Systems Journal 41(4): 597-615 (2002)

Iteration and Sorting [4]

- The result of this expression is the following sequence of four elements:

```
<fact>2 times 5 is 10</fact>
<fact>2 times 10 is 20</fact>
<fact>3 times 5 is 15</fact>
<fact>3 times 10 is 30</fact>
```

- The `for` clauses illustrated above and the `let` clause illustrated earlier are both special cases of a FLWOR expression.

Iteration and Sorting [5]

- In its most general form, a FLWOR expression may have multiple **`for`** clauses, multiple optional **`let`** clauses, an optional **`where`** clause, an optional **`order by`** clause and a mandatory **`return`** clause.

- As we have already seen, the function of the **`for`** clause and **`let`** clause is to bind variables.
- Each of these clauses contains one or more variables and an expression associated with each variable.

from: XQuery: An XML query language. Donald D. Chamberlin. IBM Systems Journal 41(4): 597-615 (2002)

Iteration and Sorting [6]

- The expressions evaluate to sequences and may contain references to variables bound in previous clauses.
- A `for` clause iterates each variable over the associated sequence, binding the variable in turn to each item in the sequence
- A `let` clause binds each variable to the associated sequence as a whole.
- This difference is illustrated in the next slide.

from: XQuery: An XML query language. Donald D. Chamberlin. IBM Systems Journal 41(4): 597-615 (2002)

Iteration and Sorting [7]

- The pair of clauses (right top) is not a full FLWOR expression because it does not have a **`return`** clause.

- The **`for`** clause and **`let`** clause simply produce a sequence of binding tuples.

- This can be seen if we add a **`return`** clause (right middle).

- The clauses in the above example produce the sequence of binding pairs (right bottom).

```
for $i in (1 to 3)
let $j := (1 to $i)
```

```
for $i in (1 to 3)
let $j := (1 to $i)
return
  <ij>
    ({$i},{$j})
  </ij>
```

```
<ij>(1,1)</ij>
<ij>(2,1 2)</ij>
<ij>(3,1 2 3)</ij>
```

Iteration and Sorting [8]

- In general, the number of binding tuples produced by a series of **`for`** clauses and **`let`** clauses is equal to the product of the cardinalities of the iteration expressions in the **`for`** clauses.
- A **`let`** clause without any for clause, of course, produces only a single binding tuple.
- The binding tuples produced by the **`for`** clauses and **`let`** clauses in a FLWOR expression are filtered by the optional **`where`** clause.
- The **`where`** clause contains an expression that is evaluated for each binding tuple.

Iteration and Sorting [9]

- If the value of the **`where`** expression is the Boolean value *true* or a sequence containing at least one node (i.e., an existence test evaluates to *success*), the binding tuple is retained; otherwise the binding tuple is discarded.
- The **`return`** clause of the FLWOR expression is then executed once for each binding tuple retained by the **`where`** clause, in order.
- The results of these executions are concatenated into a sequence that serves as the result of the FLWOR expression.

Iteration and Sorting [10]

- The power of FLWOR is illustrated by Q4, a query over our auction database.

- It constructs a sequence of popular items, based on the number of bids per item.

Iteration and Sorting [11]

(Q4) For each item that has at least five bids, generate a popular item element containing the item number, description, and bid count.

```
for $i in doc("items.xml")/*/item
let $b := doc("bids.xml")/*/bid[itemno=$i/itemno]
where count ($b)>=5
return
  <popular_item>
{ $i/itemno,
  $i/description,
  <bid_count>{count($b)}</bid_count>
}</popular_item>
```

Iteration and Sorting [12]

- The `for` clause and `let` clause produce a binding pair for each item in `items.xml`.
- In each binding pair, `$i` is bound to the item and `$b` is bound to a sequence containing all the bids for that item.
- The `where` clause retains only those binding tuples in which `$b` contains at least five bids.
- The `return` clause then generates an output element for each of these bindings, containing the item number, description, and bid count.

from: XQuery: An XML query language. Donald D. Chamberlin. IBM Systems Journal 41(4): 597-615 (2002)

Iteration and Sorting [13]

- By default, the order of the output sequence of a FLWOR expression preserves the order of the iteration sequences.
- The prefix operator **`unordered`** can be used before any expression to indicate that the order of the result is not significant.
- This gives the implementation greater flexibility to optimize the execution of the expression (for example, by iterating in a different order).

Iteration and Sorting [14]

- In a FLOWR expression, a sequence can be ordered by an **`order by`** clause that contains one or more ordering expressions.
- For each item in the original sequence, the ordering expressions are evaluated using the given item as the context item.
- The items in the original expression are then reordered into ascending or descending order based on the values of their ordering expressions.
- Of course, each ordering expression must return a single result, and these results must be comparable by the **`gt`** operator.

Iteration and Sorting [15]

- For the purpose of an **`order by`** clause, an empty sequence requires special consideration as described in the standard.
- A **`order by`** clause is used to reorder the results of a FLWOR expression.
- For example, if it is desired for the popular-item elements generated by Q4 to be sorted into ascending order by description, the following clause could be added after the **`where`** clause:

```
for $i in doc("items.xml")/*/item
let $b := doc("bids.xml")/*/bid[itemno=$i/itemno]
where count ($b)>=5
order by $i/description
return
  <popular_item>{
     $i/itemno, $i/description,<bid_count>{count($b)}</bid_count>
  }</popular_item>
```

(Q4) with ordered result

Iteration and Sorting [16]

- Q4 illustrates how a FLWOR expression can have some of the same characteristics as a join query in a relational database system and also some of the same characteristics as a grouping query.

- Q4 is like a join query because it correlates elements found in two different XML files, named `items.xml` and `bids.xml`.

- It is also like a grouping query because it groups bids together by item number and computes the number of bids in each group (in SQL, this might have been expressed as `GROUP BY itemno`).

XQuery Arithmetic Operators

Arithmetic Operators [1]

- We have already seen examples of the use of arithmetic operators.
- XQuery provides the usual arithmetic operators: **+**, **-**, *****, **`div`**, and **`mod`**.
- The division operator in XQuery is called **`div`** to distinguish it from the slash that is used in path expressions.
- When the subtraction operator follows a name, it must have a preceding blank to distinguish it from a hyphen, since a hyphen is a valid name character in XML.
- XQuery also provides the aggregating functions **`sum`**, **`avg`**, **`count`**, **`max`**, and **`min`**, which operate on a sequence of numbers and return a numeric result.

from: XQuery: An XML query language. Donald D. Chamberlin. IBM Systems Journal 41(4): 597-615 (2002)

Arithmetic Operators [2]

- Arithmetic operators are defined on numeric values (or, in the case of the aggregating functions, sequences of numeric values).
- Numeric values include values of type `integer`, `decimal`, `float`, `double`, or types derived from these types.
- When the operands of an arithmetic operator are mixed, they are promoted to the nearest common type using the promotion hierarchy: `integer` → `decimal` → `float` → `double`.

from: XQuery: An XML query language. Donald D. Chamberlin. IBM Systems Journal 41(4): 597-615 (2002)

Arithmetic Operators [3]

- If an operand of an arithmetic operator is a node, its typed value is automatically extracted.
- The behaviour of arithmetic operators on empty sequences is an important special case.
- In XQuery, an empty sequence is sometimes used to represent missing or unknown information, in much the same way that a null value is used in relational systems.
- For this reason, the `+`, `-`, `*`, `div`, and `mod` operators are defined to return an empty sequence if either of their operands is an empty sequence.

from: XQuery: An XML query language. Donald D. Chamberlin. IBM Systems Journal 41(4): 597-615 (2002)

Arithmetic Operators [3]

- To illustrate the application of this rule, suppose that the variable `$emps` is bound to a sequence of **`emp`** elements, each of which represents an employee and contains a **`name`** element, a **`salary`** element, an optional **`commission`** element, and an optional **`bonus`** element.

- Given the employee data in the next slide, the expression in Q5 (in the subsequent slide) transforms this sequence into a new sequence of **`emp`** elements, each of which contains a **`name`** element and a **`pay`** element whose value is the employee's total pay.

- For those employees whose commission or bonus is missing (`$e/commission` or `$e/bonus` evaluates to an empty sequence), the generated **`pay`** element will be empty.

Example Data: `employees-complete.xml`

```
<employees>
  <emp>
    <name>John Smith</name>
    <salary>100</salary>
    <commission>10</commission>
    <bonus>22</bonus>
  </emp>
  <emp>
    <name>Jane Silver</name>
    <salary>120</salary>
    <commission>18</commission>
    <bonus>22</bonus>
  </emp>
  <emp>
    <name>Jim Soames</name>
    <salary>80</salary>
    <commission>18</commission>
    <bonus>22</bonus>
  </emp>
  <emp>
    <name>Julia Saunders</name>
    <salary>130</salary>
    <commission>26</commission>
    <bonus>34</bonus>
  </emp>
  <emp>
    <name>James Swift</name>
    <salary>95</salary>
    <commission>26</commission>
    <bonus>34</bonus>
  </emp>
</employees>
```

from: XQuery: An XML query language. Donald D. Chamberlin. IBM Systems Journal 41(4): 597-615 (2002)

Arithmetic Operators [4]

```
for $e in doc("employees-complete.xml")/employees/*
let $p := ($e/salary + $e/commission + $e/bonus)
order by $p
return
  <emp>{
     $e/name,
     <pay>{$p}</pay>
  }</emp>
```

(Q5) Replace the salary, commission, and bonus subelements of an emp element with a new pay element containing the sum of the values of the original elements, and return ordered by pay.

```
<emp>
  <name>Jim Soames</name>
  <pay>120</pay>
</emp>
<emp>
  <name>John Smith</name>
  <pay>132</pay>
</emp>
<emp>
  <name>James Swift</name>
  <pay>155</pay>
</emp>
<emp>
  <name>Jane Silver</name>
  <pay>160</pay>
</emp>
<emp>
  <name>Julia Saunders</name>
  <pay>190</pay>
</emp>
```

from: XQuery: An XML query language. Donald D. Chamberlin. IBM Systems Journal 41(4): 597-615 (2002)

Arithmetic Operators [5]

- Note, however, that Q5, above, only works when every employee has a salary, a commission and a bonus.
- If both the commission and bonus are optional, then it may be desirable to provide a default value that can be substituted for missing operands in the arithmetic expression that computes the pay.
- A later slide illustrates how a user-defined function can be written for this purpose.

XQuery Operators on Sequences

Operators on Sequences [1]

- In a sense, all XQuery operations are operations on sequences, since every value in XQuery is either a sequence or an error.

- However, XQuery provides three operators that are specifically designed for combining sequences of nodes: **`union`**, **`intersect`**, and **`except`**.

- A **`union`** of two node sequences is a sequence containing all the nodes that occur in either of the operands.

- The **`intersect`** operator produces a sequence containing all the nodes that occur in both of its operands.

- The **`except`** operator produces a sequence containing all the nodes that occur in its first operand but not in its second operand.

- The **`union`**, **`intersect`**, and **`except`** operators return node sequences in document order and eliminate duplicates from their result sequences, based on node identity.

Operators on Sequences [2]

- Query Q6 (below) provides an example of the use of the `intersect` operator.

(Q6) Construct a new element, named recent-large-bids, containing copies of all the bid elements in the document `bids.xml` that have a bid_amount of at least 200 and a bid_date after January 1, 1999.

```
<recent_large_bids>
  {doc("bids.xml")/*/bid[bid_amount >= 200]
   intersect
   doc("bids.xml")/*/bid[bid_date > xs:date("1999-01-01")]}
</recent_large_bids>
```

Operators on Sequences [3]

- Expressions that apply the **`union`**, **`intersect`**, and **`except`** operators can often be expressed in another way.

- For example, the query (below) is equivalent to Q6.

(Q6) without a sequence operation

```
<recent_large_bids>
  {doc("bids.xml")/*/bid[bid_amount >= 200 and
                         bid_date > xs:date("1999-01-01")]}
</recent_large_bids>
```

from: XQuery: An XML query language. Donald D. Chamberlin. IBM Systems Journal 41(4): 597-615 (2002)

Operators on Sequences [4]

- It is important to remember that **`intersect`** and **`except`** are not useful in combining sequences of nodes from different documents, since there is no possibility that two nodes in different documents could have the same node identity.

- For example, consider the following expression:

```
doc("items.xml")//itemno
 except
doc("bids.xml")//itemno
```

from: XQuery: An XML query language. Donald D. Chamberlin. IBM Systems Journal 41(4): 597-615 (2002)

Operators on Sequences [5]

- The expression in the previous slide applies the **`except`** operator to two sequences of **`itemno`** nodes.
- Since the node sequences are selected from different documents, there is no possibility that any node in the second sequence could be identical to a node in the first sequence.
- Therefore, this query returns all the **`itemno`** nodes in items.xml.
- If the intent of the query had been to make a list of **`itemno`** elements for items that have no bids, this can be accomplished as shown in the next slide.

Operators on Sequences [6]

- We use the library function **`empty`**, which returns true if its operand is an empty sequence.

- The predicate **`itemno eq $i/`** **`itemno`** extracts and compares the content of **`itemno`** nodes rather than their identity.

List all items for which there have been no bids.

```
for    $i in doc("items.xml")//item
where  empty(doc("bids.xml")//bid[itemno eq $i/itemno])
return $i/itemno
```

from: XQuery: An XML query language. Donald D. Chamberlin. IBM Systems Journal 41(4): 597-615 (2002)

Operators on Sequences [7]

- The operator | (i.e., the vertical bar) has the same semantics as the **union** operator and is more concise in path expressions.

- For example, the path expression to the right first finds the union of all **b** children and all **c** children of nodes in the sequence bound to **$a**. then the **d** children of the members of that union.

```
$a/(b|c)/d
```

from: XQuery: An XML query language. Donald D. Chamberlin. IBM Systems Journal 41(4): 597-615 (2002)

XQuery Conditional Expressions

Conditional Expression [1]

- A conditional expression provides a way of executing one of two expressions, depending on the value of a third expression.
- It is written in the familiar, **`if-then-else`** tripartite format that is common to many languages.
- In XQuery, all three parts are required, and the expression in the **`if`** clause must be enclosed in parentheses.
- The result of a conditional expression depends on the value of the expression in the **`if`** clause, called the *test expression*.

from: XQuery: An XML query language. Donald D. Chamberlin. IBM Systems Journal 41(4): 597-615 (2002)

Conditional Expression [2]

- The rules are as follows:
  - If the value of the test expression is the Boolean value *true*, or a sequence containing at least one node (serving as an existence test), the **`then`** clause is executed.
  - If the value of the test expression is the Boolean value *false* or an empty sequence, the **`else`** clause is executed.
  - Otherwise, the conditional expression returns the error value.

from: XQuery: An XML query language. Donald D. Chamberlin. IBM Systems Journal 41(4): 597-615 (2002)

Conditional Expression [3]

- The following simple conditional expression might be used to return the **`price`** of a **`part`**, depending on the existence of an attribute named **`discounted`** (independently of the value of the attribute):

```
if ($part/@discounted)
  then $part/wholesale
  else $part/retail
```

- Q7 in the next slide is an example of a more complex query that contains a conditional expression.

- That query also illustrates several levels of nesting of FLWOR expressions and element constructors.

Conditional Expression [4]

```
<bid_status_report>{
  for $item in doc("items.xml")/*/item
  return
    <item>{
     $item/itemno,
     for $bid in doc("bids.xml")/*/bid[itemno = $item/itemno]
     return
      <bid>{
        $bid/bidder,
        $bid/bid_amount,
        <status>{
           if ($bid/bid_date > $item/end_date)
           then "too late"
           else if ($bid/bid_amount < $item/reserve_price)
                then "too small"
                else "OK"}
        </status>}
      </bid>}
  </item>}
</bid_status_report>
```

(Q7) Generate a report containing the status of the bids for various items. Label each bid with a status "OK", "too small", or "too late". Enclose the report in an element called bid-status-report.

MANCHESTER 1824

The University of Manchester

XQuery Quantified Expressions

Quantified Expression [1]

- Quantified expressions allow testing of some condition to see whether it is true
  - for *some* value in a sequence (called an *existential quantifier*), or
  - for *every* value in a sequence (called a *universal quantifier*).
- The value of a quantified expression is always either *true* or *false*.
- Like a FLWOR expression, a quantified expression allows a variable to iterate over the items in a sequence, being bound in turn to each item in the sequence.

from: XQuery: An XML query language. Donald D. Chamberlin. IBM Systems Journal 41(4): 597-615 (2002)

Quantified Expression [2]

- A quantified expression that begins with **`some`** returns the value *true* if the test expression is true for *at least one* variable binding, as in the first example below.
- For example, the second quantified expression below returns the value *false* because the test expression is true for some but not all bindings.

- A quantified expression that begins with **`every`**, in contrast, returns the value *true* if the test expression is true *for every one* of the variable bindings.

```
some $n in (5, 7, 9, 11) satisfies $n > 10
```

```
every $n in (5, 7, 9, 11) satisfies $n > 10
```

Quantified Expression [3]

```
<underpriced_items>{
  for $i in  doc("items.xml")/*/item
  where every $b in doc("bids.xml")/*/bid[itemno = $i/itemno]
        satisfies $b/bid_amount > 2*$i/reserve_price
  return $i
}</underpriced_items>
```

(Q8) Find the items in items.xml for which all the bids received were more than twice the reserve price. Return copies of all these item elements, enclosed in a new element called underpriced_items.

from: XQuery: An XML query language. Donald D. Chamberlin. IBM Systems Journal 41(4): 597-615 (2002)

MANCHESTER 1824

The University of Manchester

XQuery Functions

Functions [1]

- We have already seen several examples of functions, including the **doc** function and aggregating functions such as **avg**.
- XQuery provides a library of predefined functions, and also allows users to define functions of their own.
- A function may take zero or more parameters.
- A function definition must specify the name of the function and the names of its parameters.
- It may optionally specify types for the parameters and the result of the function.
- It must also provide the body of the function, which is an expression enclosed in curly braces.

from: XQuery: An XML query language. Donald D. Chamberlin. IBM Systems Journal 41(4): 597-615 (2002)

Functions [2]

- When the function is called, the arguments of the function call are bound to the parameters of the function and the body is executed, producing the result of the function call.
- If no type is specified for a function parameter, that parameter accepts values of any type.
- If no type is specified for the result of the function, the function may return a value of any type.

from: XQuery: An XML query language. Donald D. Chamberlin. IBM Systems Journal 41(4): 597-615 (2002)

Functions [3]

- Functions must be assigned a namespace.
- The predefined `local` namespace can be used if no specific namespace is intended
- The following example defines a function named **`local:highbid`** that takes an element node as its parameter and returns a double value.
- This function interprets its parameter as an item element and extracts its item number.
- It then finds and returns the largest bid amount that has been recorded for that item number.
- The example illustrates an invocation of the **`local:highbid`** on item **`1001`**.

from: XQuery: An XML query language. Donald D. Chamberlin. IBM Systems Journal 41(4): 597-615 (2002)

Functions [4]

```
declare function local:highbid($item as element())
        as xs:double?
   {
    max(doc("bids.xml")//bid[itemno = $item/itemno]/bid_amount)
   };
local:highbid(doc("items.xml")//item[itemno = "1001"])
```

from: XQuery: An XML query language. Donald D. Chamberlin. IBM Systems Journal 41(4): 597-615 (2002)

Functions [5]

- The types used as the argument types and result type of a function definition may be simple types such as `double`, or more complex types such as `element()` and `attribute()`.

- The rules for declaring types in function definitions are described in more detail later.

- The arguments of a function call must match the declared types of the function parameters.

- For this purpose, a function argument of a numeric type may be promoted to the declared parameter type, using the promotion hierarchy `integer` → `decimal` → `float` → `double`.

from: XQuery: An XML query language. Donald D. Chamberlin. IBM Systems Journal 41(4): 597-615 (2002)

Functions [6]

- An argument is also considered to be a match if the type of the argument is derived from (i.e., is a subtype of) the declared parameter type.
- If a function that expects an atomic value is called with an argument that is an element, the typed value of the element is extracted and checked for compatibility with the expected parameter type before it is passed to the function.
- The value produced by the body of a function must also match the return type declared in the function definition, using the same rules that are used for parameter matching.

Functions [7]

- Recall the employee data seen above.
- The example function in the next slide illustrates how a user might write a function to provide a default value for missing data.
- The function named **`defaulted`** takes two parameters: a (possibly missing) element node, and a default value.
- If the element is present and has a nonempty value, the function returns that value; but if the element is absent, the function returns the default value.

from: XQuery: An XML query language. Donald D. Chamberlin. IBM Systems Journal 41(4): 597-615 (2002)

Functions [8]

```
declare function local:defaulted
   ($e as element()?, $d as xs:anyAtomicType)
   as xs:anyAtomicType
   {
    if (not(exists($e))) then $d else data($e)
   }
;
let    $employees := doc("employees.xml")
return for $emp in $employees//emp return
<emp>
{
  $emp/name,
  <pay>
  {
    $emp/salary
    +local:defaulted($emp/commission,0)
    +local:defaulted($emp/bonus,0)
  }</pay>
}</emp>
```

Note that this expression in the return clause is a rewrite of Q5 above (unordered).

Functions [9]

- A function that invokes itself in its own body is called a *recursive function*, and two functions whose bodies invoke each other are called *mutually recursive functions*.
- Recursion is a powerful feature in function definitions, particularly in functions that are defined over a hierarchical data model such as XML.

Functions [10]

- As an illustration of a recursive function, the **`depth`** function in the following example can be invoked on an element and returns the depth of the element hierarchy beginning with its argument.

- If the argument element has no descendants, the depth of the hierarchy is one.

- Otherwise, the depth of the hierarchy is one more than the maximum depth of any hierarchy rooted in a child of the argument element; this value is computed by a recursive call to the depth function.

- The example also illustrates a function call that invokes the depth function to find the depth of the document named **`bids.xml`**.

Functions [11]

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

from: XQuery: An XML query language. Donald D. Chamberlin. IBM Systems Journal 41(4): 597-615 (2002)

MANCHESTER 1824

The University of Manchester

XQuery Types

Types [1]

- In writing a query, it is sometimes necessary to refer to a particular type.
- For example, function definitions may describe the types of the function parameters and result, as noted in the previous section.
- Other types of XQuery expressions, described later in these slides, also may wish to refer to specific types.
- One way to refer to a type is by its qualified name, or QName.

Types [2]

- A QName may refer to a built-in type such as **`xs:integer`** or to a type that is defined in some schema, such as `abc:address`.
- If the QName has a namespace prefix (the part to the left of the colon), that prefix must be bound to a specific namespace URI.
- This binding is accomplished by a namespace declaration in the query prolog, described later.

from: XQuery: An XML query language. Donald D. Chamberlin. IBM Systems Journal 41(4): 597-615 (2002)

Types [3]

- Another way to refer to a type is by a generic form such as **`element()`** or **`attribute()`**.

- These forms may optionally take a QName as argument that further restricts the name or type of the node.

- For example, **`element()`** denotes any element; **`element(shipto)`** denotes an element whose name is **`shipto`**; and **`element(abc:address)`** denotes an element whose type is **`address`** as declared in the namespace **`abc`**.

- The keyword **`attribute()`** denotes any attribute, **`node()`** denotes any node, and **`item()`** denotes any item (node or atomic value).

from: XQuery: An XML query language. Donald D. Chamberlin. IBM Systems Journal 41(4): 597-615 (2002)

Types [4]

- A reference to a type may optionally be followed by one of three occurrence indicators:
  - * means zero or more
  - + means one or more
  - ? means zero or one
- The absence of an occurrence indicator denotes *exactly one occurrence* of the indicated type.
- The use of occurrence indicators is illustrated by the examples in the next slide.

from: XQuery: An XML query language. Donald D. Chamberlin. IBM Systems Journal 41(4): 597-615 (2002)

Types [5]

- `element(memo)?` denotes an optional occurrence of an element with the name **`memo`**
- `element(person, surgeon)+` denotes one or more elements with name **`person`** annotated with the type **`surgeon`**
- `element()*` denotes zero or more unrestricted elements
- `attribute()?` denotes an optional attribute of any name or type

from: XQuery: An XML query language. Donald D. Chamberlin. IBM Systems Journal 41(4): 597-615 (2002)

Types [6]

- Type references occur not only in function definitions but also in several other places in XQuery.
- One of these places is as the second operand of `instance of`, a binary operator that returns *true* if its first operand is an instance of the type named in its second operand.
- The examples in the next slide illustrate usage of the `instance of` operator, presuming that the prefix `xs` is bound to the standard XML schema namespace.

from: XQuery: An XML query language. Donald D. Chamberlin. IBM Systems Journal 41(4): 597-615 (2002)

Types [7]

- `49 instance of xs:integer` returns true
- `"Hello" instance of xs:integer` returns false
- `$a instance of element(shipto)` returns true if `$a` is bound to an element whose name is `shipto`
- `<partno>369</partno> instance of element()` returns true

Types [8]

- Occasionally it may be necessary for a query to process an expression in a way that depends on the dynamic (run-time) type of the expression.

- For example, a query might be preparing mailing labels and might need to extract geographical information from various types of addresses.

- For such applications, XQuery provides an expression called **`typeswitch`**, which is loosely modelled on the switch statement of the C or Java languages, among others.

Types [9]

- The first part of a **`typeswitch`** consists of the expression whose type is being tested, called the *operand expression*, and optionally a variable that is bound to the value of the operand expression.
- This is followed by one or more *case clauses*, each of which contains a type and an expression.
- The operand expression is tested against the type named in each of the case clauses in turn.

from: XQuery: An XML query language. Donald D. Chamberlin. IBM Systems Journal 41(4): 597-615 (2002)

Types [10]

- The use of a **`typeswitch`** expression is illustrated by the example in the next slide.
- Such an expression could occur inside a loop in which the variable **`$customer`** iterates over a set of customer elements, each of which has a subelement named **`billing_address`**.
- The **`billing_address`** subelements are of several different types, each of which needs to be handled in its own way.

Types [11]

- In the example, **$a** is bound to a billing address and then one of several expressions is evaluated, depending on the dynamic type of **$a**.
- Within each case clause, **$a** has a specific type, for example, in the first case clause, the type of **$a** is known to be element of type **USAddress**.
- If a billing address element is encountered that does not conform to one of the expected types, the result of this example expression is **"unknown"**.

Types [12]

```
typeswitch($customer/billing-address)
    case $a as element(*, USAddress)
         return $a/state
    case $a as element(*, CanadaAddress)
         return $a/province
case $a as element(*, JapanAddress)
         return $a/prefecture
default return "unknown"
```

The types of the elements would have been defined in an XMLSchema specification.

from: XQuery: An XML query language. Donald D. Chamberlin. IBM Systems Journal 41(4): 597-615 (2002)

Types [13]

- Type names are also used in XQuery expressions called **`cast as`**, **`castable as`**, and **`treat as`**.
- Each of these expressions consists of an expression (often enclosed in parentheses), a keyword, and a reference to a type.
- A cast expression is used to convert the result of an expression into one of the built-in types of XML Schema.
- A predefined set of casts is supported.

from: XQuery: An XML query language. Donald D. Chamberlin. IBM Systems Journal 41(4): 597-615 (2002)

Types [14]

- For example, the result of the expression `$x div 5` could be cast to the `xs:double` type by the expression `$(x div 5) cast as xs:double`.
- A cast may return an error value if it is unsuccessful.
- For example, `($mystring) cast as xs:integer` will be successful if `$mystring` is a string representation of an integer, but will return an error if `$mystring` has the value `"Hello"` .

Types [15]

- A **`treat`** expression is used to ensure that the dynamic (run-time) type of an expression conforms to an expected type.

- For example, suppose that the static (compile-time) type of the expression **`$customer/shipping-address`** is **`Address`**.

Types [16]

- A certain subexpression may be meaningful only for values that conform to a subtype of **`Address`**, such as `USAddress`.
- The writer of the subexpression may use a **`treat`** expression to declare the expected type of the subexpression, as in the following example:

```
($customer/shipping-address) treat as element(*, USAddress)
```

Types [17]

- Unlike a `cast` expression, a `treat` expression does not actually change the type of its operand.
- Instead, its effect is twofold:
  - it assigns a specific static type to its operand, which can be used for type-checking when the query is compiled; and
  - at execution time, if the actual value of the expression does not conform to the named type, it returns an error value.

from: XQuery: An XML query language. Donald D. Chamberlin. IBM Systems Journal 41(4): 597-615 (2002)

Types [18]

- To see how a query processor might make use of the information provided by a **`treat`** expression, consider the following example:

```
$customer/billing-address/zipcode
```

- A type-checking XQuery compiler might consider the above example to be a type error, since the static type of **`$customer/billing-address`** is **`Address`**, and the **`Address`** type does not in general have a **`zipcode`** subelement.

Types [19]

- However, in the following reformulation of the example, the static type of the expression is changed to **`USAddress`**, which has a **`zipcode`** subelement, and the type error is removed:

```
($customer/billing-address treat as element(*, USAddress))/zipcode
```

from: XQuery: An XML query language. Donald D. Chamberlin. IBM Systems Journal 41(4): 597-615 (2002)

MANCHESTER 1824

The University of Manchester

Validation in XQuery

Validation [1]

- The process of schema validation is defined in by the XML Schema standard.
- Schema validation may be applied to an XML document or to a part of a document such as an individual element.
- The material being validated is compared with the definitions in a given schema, which describes a particular kind of document.

from: XQuery: An XML query language. Donald D. Chamberlin. IBM Systems Journal 41(4): 597-615 (2002)

Validation [2]

- The validation process may label an element as valid or invalid; it may also assign a specific type to an element and provide additional information such as default values for certain attributes.
- For example, validation of an element named **`shipto`** might assign it the specific type **`USAddress`** and might provide a default value for its carrier attribute.

from: XQuery: An XML query language. Donald D. Chamberlin. IBM Systems Journal 41(4): 597-615 (2002)

Validation [3]

- Schema validation is applied to input documents as part of the process of representing them in the query data model.
- In addition, schema validation can be invoked explicitly on a query result or on some intermediate expression within a query.
- The query data model associates a type annotation with each element node.

from: XQuery: An XML query language. Donald D. Chamberlin. IBM Systems Journal 41(4): 597-615 (2002)

Validation [4]

- A type annotation indicates that an element has been validated as conforming to a specific named type.
- Elements that have not been validated or that do not conform to a named type have the generic type annotation **anyType**.
- For example, an element that is created by an element constructor has the type annotation **anyType** until it is given a more specific type by a validate expression.

from: XQuery: An XML query language. Donald D. Chamberlin. IBM Systems Journal 41(4): 597-615 (2002)

Validation [5]

- The following example constructs an element and validates it against the schema(s) that are named in the query prolog:

```
validate {
   <shipto>
      <street>123 Elm St.</street>
      <city>Elko, NV</city>
      <zipcode>85039</zipcode>
   </shipto>
}
(: Note: not always supported :)
```

- Type annotations are used by expressions such as instance of and typeswitch that test the type of an element, and by expressions such as function calls that require an element of a particular type.
- For example, validation of the shipto element above might assign it a type annotation of `USAddress`, which might enable it to be used as an argument to a function whose parameter type is element of type `USAddress`.

Structure of a Query in XQuery

Query Structure [1]

- In XQuery, a query consists of two parts called the *query prolog* and the *query body*.

- The query prolog contains a series of declarations that define the environment for processing the query body.

- The query body is simply an expression whose value defines the result of the query.

from: XQuery: An XML query language. Donald D. Chamberlin. IBM Systems Journal 41(4): 597-615 (2002)

Query Structure [2]

- The query prolog is needed only if the query body depends on one or more namespaces, schemas, or functions.
- We will discuss declarations for namespaces, schemas, and functions separately.
- If such a dependency exists, the object(s) that the query body depends on must be declared in the query prolog.

Query Structure [3]

- A namespace declaration defines a namespace prefix and associates it with a namespace URI.
- The prefix can be any identifier.
- For example, the following namespace declaration defines the prefix **xyz** and associates it with a specific URI:

```
declare namespace xyz
  = "http://www.xyz.com/example/names";
```

Query Structure [4]

- The declaration in the previous slide enables the prefix **`xyz`** to be used in QNames in the query body.
- It associates the prefix with the URI of a specific namespace and serves as a unique qualifier for names of elements, attributes, and types.
- For example, **`xyz:billing-address`** might uniquely identify the billing-address element defined in the namespace http://www.xyz.com/example/names.

Query Structure [5]

- If desired, multiple namespace prefixes can be associated with the same namespace.
- The query prolog can declare a default namespace that applies to all unqualified element and type names and another default namespace that applies to all unqualified function names.
- The syntax for declaring default namespaces is illustrated in the following example:

```
declare default element namespace
  = "http://www.xyz.com/example/names";
declare default function namespace
  = "http://www.xyz.com/example/functions";
```

from: XQuery: An XML query language. Donald D. Chamberlin. IBM Systems Journal 41(4): 597-615 (2002)

Query Structure [6]

- If no default namespaces are provided, unqualified names of elements, types, and functions are considered to be in no namespace.
- Unqualified attribute names are always considered to be in no namespace.
- In addition to namespace declarations, a query prolog can contain one or more schema imports.

Query Structure [7]

- A schema import identifies a schema by its URI and optionally provides a second URI that specifies the location where the schema file can be found.
- The purpose of the schema import is to make available to the query processor the definitions of elements, attributes, and types that are declared in the named schema.
- The query processor can use these definitions for validating newly constructed elements, for optimization, and for doing static type analysis on a query.

Query Structure [8]

- A schema usually defines a set of elements, attributes, and types in a particular namespace, called its target namespace, but it does not define a namespace prefix.
- Therefore, a schema import may specify a namespace prefix to be bound to the target namespace of the given schema.

from: XQuery: An XML query language. Donald D. Chamberlin. IBM Systems Journal 41(4): 597-615 (2002)

Query Structure [9]

- For example, the following schema import binds the namespace prefix **`xhtml`** to the target namespace of a particular schema and also provides the system with a nonbinding hint for where this schema can be found:

```
import schema namespace xhtml
  = "http://www.w3.org/1999/xhtml"
at "http://www.w3.org/1999/xhtml/xhtml.xsd";
```

Query Structure [10]

- In addition to namespace declarations and schema imports, a query prolog may contain one or more function definitions.
- We have seen examples of function definitions in an earlier section.
- The functions defined in the query prolog may be used in the query body or in the bodies of other functions.
- The query prolog also provides a means of importing function definitions from an external function library, as in the example at the foot of this slide.

```
module namespace gis = "http://example.org/gis-functions";
```

Conclusions [1]

- XQuery is a functional language consisting of several types of expressions that can be composed with full generality.
- XQuery expression-types include path expressions, element constructors, function calls, arithmetic and logical expressions, conditional expressions, quantified expressions, expressions on sequences, and expressions on types.

Conclusions [2]

- XQuery is defined in terms of a data model based on heterogeneous sequences of nodes and atomic values.
- An instance of this data model may contain one or more XML documents or fragments of documents.
- A query provides a mapping from one instance of the data model to another instance of the data model.
- A query consists of a prolog that establishes the processing environment, and an expression that generates the result of the query.

from: XQuery: An XML query language. Donald D. Chamberlin. IBM Systems Journal 41(4): 597-615 (2002)

Conclusions [3]

- There have been two major versions of the language: XQuery 1.0 and XQuery 3.0.
- These slides have not covered features in the latter versions, such as the full-text and update facilities.
- Just as XML has established itself as an application-independent format for exchange of information on the Internet, XQuery is designed to serve as an application-independent format for exchange of queries.
- XQuery provides a standard way to retrieve information from XML data sources, it will help XML to realize its potential as a universal information representation.
- XQuery has been closely integrated with SQL-based relational databases by all the major DBMS vendors.

from: XQuery: An XML query language. Donald D. Chamberlin. IBM Systems Journal 41(4): 597-615 (2002)

MANCHESTER 1824

The University of Manchester

Bibliography

References

1. **XQuery: An XML query language**. Donald D. Chamberlin. *IBM Systems Journal* 41(4): 597-615 (2002) http://dx.doi.org/10.1147/sj.414.0597

2. **XQuery from the Experts**. D. Chamberlin, D. Draper, M. Fernández et al. *Addison-Wesley, 2004*. ISBN 0-321-18060-7. © Pearson Education Limited 2004

3. **An Introduction to XML and Web Technologies**. Anders Møller and Michael Schwartzbach. *Addison-Wesley, 2006*. ISBN 978-0-321-26966-9. http://www.brics.dk/ixwt/ © Pearson Education Limited 2006

4. **XQuery**. Priscilla Walmsley. *O'Reilly, 2007*. ISBN 978-0-596-00634-1. http://www.datypic.com/books/xquery/ © Priscilla Walmsley 2007

W3C Documents

I. **XML Path Language (XPath) 2.0 (Second Edition)**. Anders Berglund, Scott Boag, Don Chamberlin, Mary F. Fernández, Michael Kay, Jonathan Robie, Jérôme Siméon. http://www.w3.org/TR/xpath20/

II. **XQuery 1.0: An XML Query Language (Second Edition)**. Scott Boag, Don Chamberlin, Mary F. Fernández, Daniela Florescu, Jonathan Robie, Jérôme Siméon. http://www.w3.org/TR/xquery/

III. **XML Query Use Cases**. Don Chamberlin, Peter Fankhauser, Daniela Florescu, Massimo Marchiori, Jonathan Robie. http://www.w3.org/TR/xquery-use-cases/

IV. **XQuery 1.0 and XPath 2.0 Formal Semantics (Second Edition)**. Denise Draper, Michael Dyck, Peter Fankhauser, Mary Fernández, Ashok Malhotra, Kristoffer Rose, Michael Rys, Jérôme Siméon, Philip Wadler. http://www.w3.org/TR/query-semantics/

V. **The XML Query Algebra**. Peter Fankhauser, Mary Fernández, Ashok Malhotra, Michael Rys, Jérôme Siméon, Philip Wadler. http://www.w3.org/TR/2001/WD-query-algebra-20010215/

VI. **XQuery 1.0 and XPath 2.0 Functions and Operators (Second Edition)**. Ashok Malhotra, Jim Melton, Norman Walsh, Michael Kay. http://www.w3.org/TR/xpath-functions/

VII. **XQuery 3.0: An XML Query Language**. Jonathan Robie, Don Chamberlin, John Snelson. http://www.w3.org/TR/xquery-30/