ΕΛΛΗΝΙΚΗ ΔΗΜΟΚΡΑΤΙΑ
ΠΑΝΕΠΙΣΤΗΜΙΟ ΚΡΗΤΗΣ

Εισαγωγή στην Επιστήμη και Τεχνολογία των Υπηρεσιών

Ενότητα 11: XQuery

Χρήστος Νικολάου
Τμήμα Επιστήμης Υπολογιστών

Με τη συγχρηματοδότηση της Ελλάδας και της Ευρωπαϊκής Ένωσης

Άδειες Χρήσης

- Το παρόν εκπαιδευτικό υλικό υπόκειται στην άδεια χρήσης **Creative Commons** και ειδικότερα

Αναφορά – Μη εμπορική Χρήση – Όχι Παράγωγο Έργο v. 3.0

(Attribution – Non Commercial – Non-derivatives)

- Εξαιρείται από την ως άνω άδεια υλικό που περιλαμβάνεται στις διαφάνειες του μαθήματος, και υπόκειται σε άλλου τύπου άδεια χρήσης. Η άδεια χρήσης στην οποία υπόκειται το υλικό αυτό αναφέρεται ρητώς.

Χρηματοδότηση

- Το παρόν εκπαιδευτικό υλικό έχει αναπτυχθεί στα πλαίσια του εκπαιδευτικού έργου του διδάσκοντα.
- Το έργο «**Ανοικτά Ακαδημαϊκά Μαθήματα στο Πανεπιστήμιο Κρήτης**» έχει χρηματοδοτήσει μόνο τη αναδιαμόρφωση του εκπαιδευτικού υλικού.
- Το έργο υλοποιείται στο πλαίσιο του Επιχειρησιακού Προγράμματος «Εκπαίδευση και Δια Βίου Μάθηση» και συγχρηματοδοτείται από την Ευρωπαϊκή Ένωση (Ευρωπαϊκό Κοινωνικό Ταμείο) και από εθνικούς πόρους.

Με τη συγχρηματοδότηση της Ελλάδας και της Ευρωπαϊκής Ένωσης

XML
XQuery
635.444

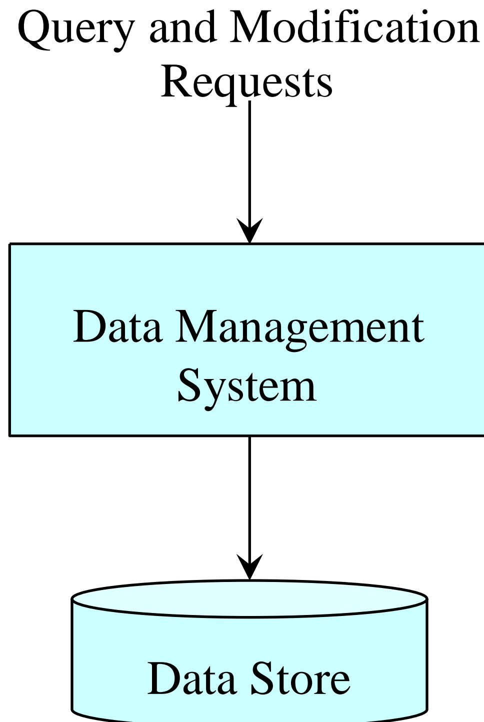
David Silberberg

Lecture 23

Databases

- A database system consists of
  - A data store
    - A place to store the data for inserts, updates, deletes, and queries
    - Data must be represented in a manner conducive to enabling straightforward updates and retrievals
  - A data management system
    - A system that organizes and manages stored data
    - Enables the modification and extraction of the data
  - A query access language
    - A language with a syntax and semantics that enables dynamic access to the data
    - A language that enables modification of the data

Database Schematic (Simplified)



- Queries and modification requests retrieve and update data
  - SQL examples
    - SELECT statements are queries
    - INSERT, UPDATE, and DELETE statements are modification requests
- Data management system
  - Manages the structure of the data store
  - Manages external query and modification requests
    - Transforms stored data into query results
    - Reflects update requests in the data store
- Data Store
  - Structured representation of the data

Relational Database Model

- Simple data structures
  - Rows, columns
  - Simple data items
- Solid model
  - Foundation for consistency
  - Normalization eliminates data anomalies
  - Database can be maintained with integrity rules
- Set-oriented data manipulation
  - Non-procedural
  - Relational algebra - set theory (SQL & QUEL)
  - Relational calculus - logic (QBE)

Definitions and Concepts

- *Database* models some real life concept and captures actual data that represents some state of real life
- *Relational database* is a database modeled by relations
- *Relation* R defined over n sets D_1, D_2, … D_n where D_i represents some domain.
- *n-tuple* (tuple) is a set $< d_1, d_2, \ldots, d_n >$ where $d_1 \sum D_1$, $d_2 \sum D_2$, …

Example Relation and Domains

- R is a set of 0 or more n-tuples (not necessarily distinct)
- MFG(manufacturer, state)
  - D_1 is set of all potential manufacturers
  - D_2 is a set of all 50 states
  - d_1 is any member of D_1 (i.e., Proctor & Gamble, McCormick Spice, etc.)
  - d_2 is any member of D_2 (i.e., OH, MD, etc.)

Keys

- *key* - minimum nonempty subset of relation's attributes that uniquely defines each tuple
- Examples:
  - CUST(cust_no, cust_name, cust_address)
    - key is (cust_no)
  - ORDER(cust_no, part_no, quantity)
    - key is (cust_no, part_no)
  - PART: (part_no, part_name, manufacturer, cost)
    - key is (part_no)
  - MFG: (manufacturer, owner)
    - key is (manufacturer)

Normalization Addresses Anomalies in RDBMS

- Repetition anomaly - state repeated
- Update anomaly - state updated twice
- Insertion anomaly
  - Cannot add new manufacturer until we have part
  - Visa versa
- Deletion anomaly - deleting 'doodad' deletes XYZ from DB

part_no	part_name	cost	manufacturer	state
1	widgit	$3	Acme Inc.	MD
2	thing-a-ma-bob	$5	Acme Inc.	MD
3	doodad	$4	XYZ Ent.	NJ

Normalization is Achieved Through Decomposition

- Start with *universal relation* - 1NF
- Iterated decomposition -> 2NF & 3NF (really BCNF)
- There are also 4NF and 5NF, which are not covered
- 5NF □ 4NF □ 3NF □ 2NF □ 1NF (□ is subset)
- Seek loss-less decomposition in normalization process
  - Joins reconstruct tables (defined later)
  - Dependency preservation

Dependency Structures

- If for each X in R, there is only one Y value: X->Y (determines)
- Key <u>determines</u> non-keys
- Example:

CUST: cust_no -> (cust_name, cust_address)

PART: part_no -> (part_name, manufacturer, cost)

ORDER: (cust_no, part_no) -> (quantity)

MFG: (manufacturer) -> (owner)

Relational Algebra

- Fundamental operations
  - Selection
  - Projection
  - Union
  - Set difference
  - Cartesian product
- Derived operations
  - Intersection
  - $\cup$-join
  - natural join
  - semi-join
  - quotient
- Others
  - Outer join
  - Transitive closure

Selection

- $\int_{F}(R)$
  - F is formula
  - F is first order logic - won't get into it
  - part_no -> (part-name, manufacturer, cost)

- $\int_{cost = 3}(R)$

part_no	part_name	cost	manufacturer
1	widgit	$3	Acme Inc.
2	thing-a-ma-bob	$5	Acme Inc.
3	doodad	$4	XYZ Ent.

Selection in SQL

```
SELECT    *
FROM      part
WHERE     cost = 3
```

Projection

- $\angle_{A,B}(R)$
  - R is relation
  - A, B are columns

- $\angle_{part\text{-}name,\ cost}(R)$

part_no	part_name	cost	manufacturer
1	widgit	$3	Acme Inc.
2	thing-a-ma-bob	$5	Acme Inc.
3	doodad	$4	XYZ Ent.

Projection in SQL

```
SELECT    part_name,
          cost
FROM      part
```

∪-Join

- R $\bowtie_F$ S
  - R and S are relations
  - F is some formula (=, <>, >, <, δ, ε , like)

R:

part_no	part_name	cost	man_no
1	widgit	$3	1
2	thing-a-ma-bob	$5	1
3	doodad	$4	2

S:

man_no	manufacturer
1	Acme Inc.
2	XYZ Ent.

∪-Join in SQL

```
SELECT part_no, part_name, cost, R.man_no, S.man_no, manufacturer
FROM     R, S
WHERE  R.man_no < S.man_no
```

part_no	part_name	cost	R.man_no	S.man_no	manufacturer
1	widgit	$3	1	2	XYZ Ent.
2	thing-a-ma-bob	$5	1	2	XYZ Ent.

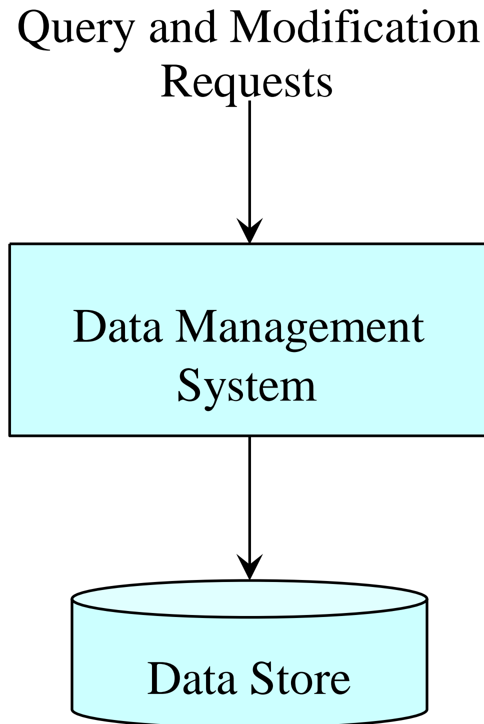
Natural Join

- $R \bowtie_F S$
  - R and S are relations
  - F is the formula '='

```
SELECT part_no, part_name, cost, R.man_no, S.man_no, manufacturer
FROM    R, S
WHERE  R.man_no = S.man_no
```

part_no	part_name	cost	R.man_no	S.man_no	manufacturer
1	widgit	$3	1	1	Acme Inc.
2	thing-a-ma-bob	$5	1	1	Acme Inc.
3	doodad	$4	2	2	XYZ Ent.

Database Schematic for XML



- Queries and modification requests retrieve and update data
  - Query languages have been XPath, XSLT, SAX, DOM, JDOM, and JAXP
  - Update languages have been editors, XSLT, SAX, DOM, JDOM, and JAXP
- Data management system
  - For the most part, this is XSLT, SAX, DOM, JDOM, and JAXP runtime environment
  - None of them really perform data management
- Data Store
  - XML document (or set of documents)

Problems with Current Methodologies

- We will not address the data management and modification issue
  - There are research and a few commercial systems that are starting to address these issues
  - Some commercial systems (e.g., Oracle) puts an XML-ish interface onto an RDBMS
- Data Retrieval
  - Until now, every new request for information required writing a new program or stylesheet
    - Approach difficult to maintain
    - New requests require new programming logic
    - Relating data through joins is difficult to manage
  - Declarative query syntax
    - Simple and flexible
    - Does not require new programming
    - XML query languages provide declarative access to XML data

XML Query Languages

- Many XML Query languages have been proposed
- XQuery is the language recommended by the W3C
  - http://www.w3.org/TR/xquery/
  - Designed to be a small and easy to implement language
  - Queries are concise and easily understood
  - Flexible enough to query broad spectrum of XML source (databases and documents)
- XQuery is derived from
  - Quilt (influenced by XML query languages Lorel and YATL)
  - XPath and XQL (path expression syntax for hierarchical documents)
  - XML-QL (binding variables to create new structures)
  - SQL (SELECT-FROM-WHERE pattern)
  - OQL (object-oriented query language that returns structures)
- Plenty of information at: **http://www.w3.org/XML/Query**

XQuery Language

- Functional language - query is represented as an expression (like OQL)
  - Supports several kinds of expressions
  - Structure and appearance of a queries may differ significantly depending on the kinds of expressions are used.
  - The various forms of expressions can be nested with full generality
- The input and output of a query are instances of an XPath data model
  - Document is modeled as a tree of nodes
  - Data model is capable of modeling not only an XML document but also a well-formed fragment of a document, a sequence of documents, or a sequence of document fragments
  - An instance of the data model is an ordered sequence of nodes, each of which may contain nested sequences of nodes.

XQuery Expressions

- The principal forms of XQuery expressions
  - Path expressions
  - Element constructors
  - FLWOR expressions
  - Expressions involving operators and functions
  - Conditional expressions
  - Quantified expressions
  - Expressions that test or modify datatypes

Predefined Namespaces

- Certain namespace prefixes are predeclared by XQuery
- Bound to fixed namespace URIs

xml = http://www.w3.org/XML/1998/namespace

xs = http://www.w3.org/2001/XMLSchema

xsi = http://www.w3.org/2001/XMLSchema-instance

fn = http://www.w3.org/2005/xpath-functions

local = http://www.w3.org/2005/xquery-local-functions

Path Expressions

- Can begin with an expression that identifies a specific node or sequence of nodes in a document.
  - `fn:doc(string)` returns the root node of a named document.
- Can also begin with "/" or "//" which represents an implicit root node
- The execution environment defines a "context node"
  - Referenced by dot (".") inside the path expression
- Consists of a series of "steps"
  - Each step represents movement through a document along a specified "axis"
  - Each step can apply one or more predicates to eliminate nodes that fail to satisfy a given condition
  - The result of each step is a sequence of nodes that serves as a starting point for the next step

Path Expression Queries

- Simple query
  - Retrieve the figure(s) with caption "Tree Frogs" in the second chapter of the document named **zoo.xml**
  - `fn:doc("zoo.xml")//chapter[2]//figure[caption = "Tree Frogs"]`
- Queries may specify a sequence of nodes by a specifying its ordinal number in the sequence (as in chapter[2], chapter[fn:position() = (1,3 5, 7)], and chapter[fn:position() = (2 to 5)])
  - *Retrieve all the figures in chapters 2 through 5 of the document named "zoo.xml."*
  - `fn:doc("zoo.xml")//chapter[fn:position() = (2 to 5)]//figure`

Path Expressions with Operators

- Path expressions can contain operators that are defined over simple datatypes.
  - If operand is a node, contents are extracted and converted to a typed value (e.g., <grade>89</grade> -> 89)
  - If no argument, its implicit argument is the current (context) node

- Retrieve the annual salary of a single employee named "Fred"

```
fn:doc("emp.xml")//emp[name="Fred"]/salary * 12
```

- Retrieve the annual salaries of employees named "Fred"

```
for $a in fn:doc("emp.xml")//emp[name="Fred"]/salary
return
<result>
    {$a * 12}
</result>
```

Element Constructors

- Path expressions search for elements in existing documents
- However, a query often needs to generate new elements
- XQuery expressions allow embedding XML elements that represent themselves
  - Called an *element constructor*
  - Allows literal XML fragments to be "pasted" into queries
- Trivial example:
  - Retrieve (generate) an <emp> element that has an "empid" attribute and nested <name> and <job> elements

```
<emp empid = "12345">
   <name>John Smith</name>
   <job>Anthropologist</job>
</emp>
```

Computed Constructors – Simple Case

- Constructors can be created with various keywords: `element`, `attribute`, `document`, `text`, `processing-instruction`, `comment`

- The following is an alternate way of using computed constructors:

```
element emp {
  attribute empid {"12345"}
  element name {"John Smith"}
  element job {"Anthropologist"}
}
```

Element Constructor with Binding

- Often, contents of elements or attribute values need to be computed by some expression
- XQuery expressions to be computed that are inside element constructors are enclosed in curly braces (much like XSLT)
- In the following example, we assume that variables $id, $name, and $job are bound to strings and nodes elsewhere in the query
- Query - Retrieve (generate) an <emp> element that has an "empid" attribute.

```
<emp empid = "{$id}">
    {$name}
    {$job}
</emp>
```

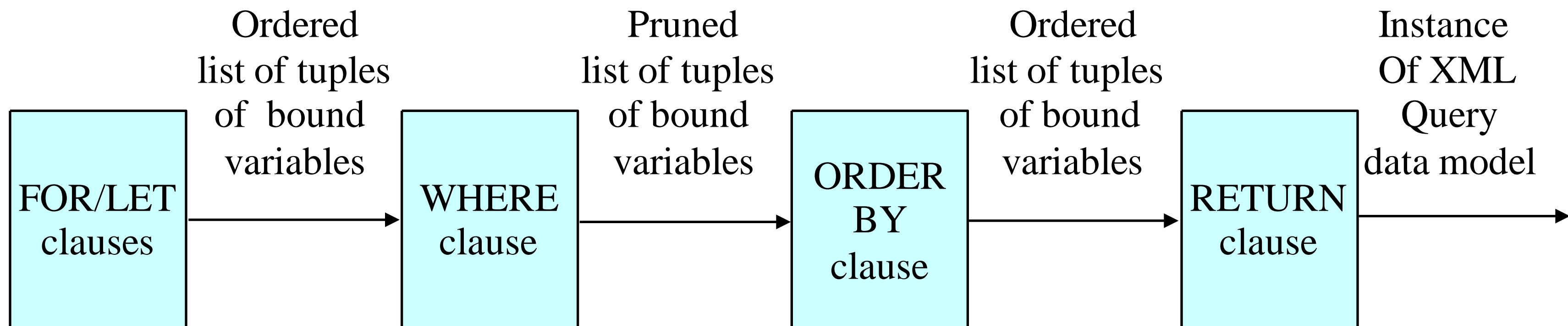
Constructing Elements and Attributes

- Names of elements and attributes may be computed by an expression
- For this purpose, XQuery allows the name of an element or attribute to be XQuery expressions enclosed in curly braces
  - If the name in a start-tag is an expression, the name must be omitted from the corresponding end-tag
  - however, when a start-tag contains a constant name, the same name must be specified in the matching end-tag
- Example (assume $e is bound to some element with numeric content)
  - Retrieve (generate) a new element having the same name and attributes as $e, and with numeric content equal to twice the content of $e.

```
LET $elt := $e/text()
RETURN
  element {$elt}     # replicates the name of $e
   {$e/@*}                # replicates the attributes of $e
   {2 * number($e)}   # doubles the content of $e
```

FLWOR Expressions

- A FLWOR (pronounced "flower") expression is constructed from FOR, LET, WHERE, ORDER BY, and RETURN clauses
  - Must appear in a specific order
  - Binds values to one or more variables and then uses these variables to construct a result



FOR Clause

- Used whenever iteration is needed
- Introduces one or more variables, associating each variable with an expression
  - FOR-clause might contain a path expression that returns a sequence of nodes
- Result of the FOR-clause is a sequence of tuples
  - Each sequence of tuples contains a binding for each of the variables in the FOR-clause.
  - The variables are bound to individual values returned by their respective expressions.
- Each variable in a FOR-clause can be thought of as iterating over the values (sequence of tuples) returned by its respective expression, in order.

FOR Examples

- Example 1
  - `for $b in fn:doc("bib.xml")//book`
  - Assigns to $b the sequence of tuples, each of which represents a <book> node in the document "bib.xml"
  - The query will iterate over the tuples bound to $b
- Example 2
  - ```
    for $b in fn:doc(“bib.xml”)//book,
         $c in $b/author
    ```
  - Sets up a double loop over books and authors of books
    ```

LET Clause

- Used to bind one or more variables to one or more expressions
- LET clause binds each variable to the value of its respective expression without iteration
  - Results in a single binding for each variable
  - Unlike FOR clause
- Example of difference between FOR and LET clauses
  - `for $b in fn:doc("bib.xml")//book`
    - Results in many bindings in an iteration
    - Each binding binds the variable $b to one book in the library
  - `let $b := fn:doc("bib.xml")//book`
    - Results in a single binding
    - Binds the variable $b to a sequence containing all the books in the library
  - `let $b := for … where … return …`
    - $b is the set of returned items

FOR/LET Combination

- A FLWOR expression may contain several FOR and LET clauses
- Expressions used in FOR and LET-clauses may reference variables bound earlier in the FLWOR expression
- Result of the FOR and LET clauses is an ordered sequence of tuples of bound variables
  - If all FOR-clause expressions are independent, the number of tuples generated is the product of the cardinalities of all the FOR clause expressions
  - A FLWOR expression that contains no FOR clauses generates exactly one binding-tuple
- Order of the tuples generated by the FOR and LET clauses is determined by the order in which values are returned by the FOR-clause expressions
  - The order in which the variables are bound determines the order of nested iteration of the FLWOR expression.

WHERE Clause

- Bindings generated by the FOR and LET clauses are filtered by optional WHERE clause
- Only tuples for which the condition in the WHERE clause is true are used to invoke the RETURN clause
- The WHERE clause predicates
  - Connected by AND and OR
  - Usually reference bound variables
- Variables bound by a FOR clause usually represent individual nodes
  - Typically used in scalar predicates such as $p/color = "Red"
- Variables bound by a LET clause usually represent a sequences of nodes
  - Typically used in set-oriented predicates such as avg($p/price) > 100
- Order of the binding-tuples generated by the FOR and LET clauses is preserved

Comparison Expressions

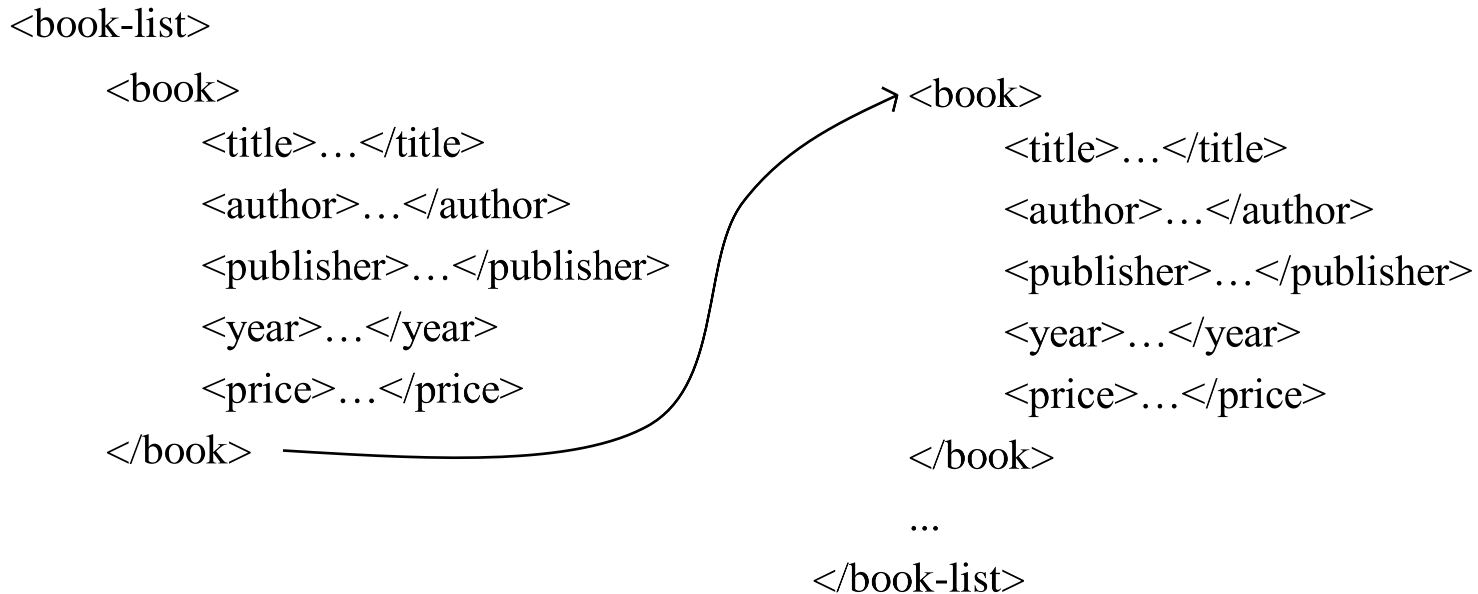
- Value comparisons – compare single values
  - Comparators: eq, ne, lt, le, gt, ge
  - `$book/author eq "Smith"` is true only if the text value is "Smith"
  - `//book[year gt 2005]` is true only if <year> is a child of <book> and its value is greater than 2006
  - `<year>2008</year> eq <year>2008</year>` is true since their *atomized* values are equal
- General comparisons – compare operands of sequences of any length
  - Comparators: =, !=, <, <=, >, >=
  - `$book/author = "Smith"` is true if any subelement's value is "Smith"
  - `(1,2) = (2,3)` is true since one or more of the elements overlap
  - `(1,2) = (3,4)` is not true because no elements overlap
- Node comparisons – compare node elements by their order or document identity
  - Comparators: is, <<, >>
  - `//book[year eq 1998] is //book[publisher eq "Morgan Kaufmann"]` is true if they refer to the same book node
  - `<year>2008</year> is <year>2008</year>` is false since they refer to different nodes
  - `//book[author eq "Smith"] >> //book[author eq "Jones"]` is true if the book written by Smith occurs later in the document than the book written by Jones
    `Kaufmann"]`

RETURN Clause

- Generates the output of the FLWOR expression
  - May be any sequence of nodes or primitive values
- The RETURN clause is executed once for each tuple of bindings
  - That is generated by the FOR and LET clauses
  - Satisfies the condition in the WHERE clause
  - Preserves the order of these tuples
- Contains an expression that contains
  - Element constructors
  - References to bound variables
  - Nested sub-expressions
- Results generated by individual executions of the RETURN clause are concatenated together
  - Preserves order
  - May contain duplicate nodes

Examples

- Example "bib.xml" document



ORDER BY

- **Make an alphabetic list of publishers. Within each publisher, make a list of books, each containing a title and a price, in descending order by price**

```
<publisher_list>
   {for $p in fn:distinct-values(fn:doc("bib.xml")//publisher)
    order by name
    return
       <publisher>
         <name> {$p} </name>
         {for $b in fn:doc("bib.xml")//book[publisher= $p]
           order by price descending
           return
             <book>
               {$b/title}
               {$b/price}
             </book>
         }
       </publisher>
   }
</publisher_list>
```

Query Examples

- List the titles of books published by Morgan Kaufmann in 1998

```
for    $b in fn:doc("bib.xml")//book
where  $b/publisher = "Morgan Kaufmann"
       and $b/year = "1998"
return $b/title
```

- If order is not important, use the unordered function

```
for    $b in fn:unordered(fn:doc("bib.xml")//book)
where  $b/publisher = "Morgan Kaufmann"
       and $b/year = "1998"
return $b/title
```

DISTINCT Function

- Distinct function eliminates duplicate values from a set of tuples
  - Two elements are considered to have duplicate values if their names, attributes, and normalized content are equal
  - Distinct function retains one node FOR $p IN distinct-values(doc("bib.xml")//publisher)
- List each publisher and the average price of its books

```
for $p in fn:distinct-values(fn:doc("bib.xml")//publisher)
let $a := fn:avg(fn:doc("bib.xml")//book[publisher = $p]/price)
return
   <publisher>
      <name> {$p} </name>
      <avgprice> {$a} </avgprice>
   </publisher>
```

Aggregate Functions

- List the publishers who have published more than 100 books

```
<big_publishers>
   {
    for     $p in fn:distinct-values(fn:doc("bib.xml")//publisher)
    let     $b := fn:doc("bib.xml")//book[publisher = $p]
    where  count($b) > 100
    return $p
   }
</big_publishers>
```

- SQL does not allow count() in the WHERE clause

Structural Transformation Using Embedded FLWOR Expressions

- Invert the structure of the input document so that, instead of each book element containing a sequence of authors, each distinct author element contains a sequence of book-titles

```
<author_list>
   {
    for $a in fn:distinct-values(fn:doc("bib.xml")//author)
    return
      <author>
         <name> {$a} </name>
         {
          for $b in fn:doc("bib.xml")//book[author = $a]
          return $b/title
         }
      </author>
   }
</author_list>
```

LET Simplifies Query Expressions

- For each book whose price is greater than the average price, return the title of the book and the amount by which the book's price exceeds the average price.

```
<result>
   {
    let $a := fn:avg(fn:doc("bib.xml")//book/price)
    for $b in fn:doc("bib.xml")//book
    where $b/price > $a
    return
       <expensive_book>
          {$b/title}
          <price_difference>
             {$b/price - $a}
          </price_difference>
       </expensive_book>
   }
</result>
```

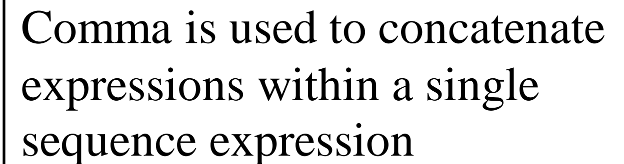
Complex Transformations

- Computed element names and attribute names are used to perform structural transformations
- Construct a new element having the same name as the element bound to $e. Transform all the attributes of $e into subelements, and all the subelements of $e into attributes

```
element {name($e)}
    {
     for $c in $e/*
     return attribute {name($c)} {string($c)}
    }
    {
     for $a in $e/@*
     return
         element {name($a)} {string($a)}
    }
```

Conditional Expressions

- Make a list of holdings, ordered by title. For journals, include the editor, and for all other holdings, include the author.




```
for $h in //holding
order by title
return
   <holding>
      {$h/title,
       if ($h/@type = "Journal")
       then  $h/editor
       else $h/author
      }
   </holding>
```

Alternate Conditional Expression

- Make a list of holdings, ordered by title. For journals, include the editor, and for all other holdings, include the author.

```
for $h in //holding
order by title
return
   <holding>
      {$h/title}
      {if ($h/@type = "Journal")
       then $h/editor
       else $h/author
      }
   </holding>
```

Quantified Expressions (SOME)

- SOME Expression
  - Generates multiple bindings for a variable, using values returned by the expression in the IN clause
  - For each of these bindings, the expression in the SATISFIES expression is executed
  - If at least one execution of the SATISFIES expression returns the Boolean value True, then the result is True
  - Otherwise the result is False
  - If the expression in the IN clause does not return any nodes, the result is False.
- Find titles of books in which both sailing and windsurfing are mentioned in the same paragraph

```
for $b in //book
where some $p in $b//para satisfies
  (contains($p, "sailing") and
   contains($p, "windsurfing"))
return $b/title
```

Quantified Expressions (EVERY)

- EVERY Expression
  - Generates multiple bindings for a variable, using values returned by the expression in the IN clause
  - For each of these bindings, the expression in the SATISFIES expression is executed
  - If every execution of the SATISFIES expression returns the Boolean value True, then the result is True
  - Otherwise the result is False
  - If the expression in the IN clause does not return any nodes, the result is True.
- Find titles of books in which sailing is mentioned in every paragraph

```
for $b IN //book
where every $p in $b//para satisfies
                contains($p, "sailing")
return $b/title
```

- This query also returns books that contain no paragraphs

Core Functions

- XQuery provides a core library of built-in functions
  - Example: fn:root() - returns the root node of a named document
  - All the functions of the XPath core function library
  - Aggregation functions – fn:avg(), fn:sum(), fn:count(), fn:max(), and fn:min()
  - fn:distinct-values() function eliminates duplicate nodes from a sequence,
  - fn:empty() function returns True if and only if its argument is an empty sequence

User Defined Functions

- XQuery allows users to define functions of their own
- Function definition specifies
  - Name of the function
  - Names and datatypes of the parameters
  - Datatype of the result
  - Datatypes are specified by their qualified names
- A function definition also provides an expression (called the "function body") that defines how the result of the function is computed from its parameters
- When called, function arguments must be valid instances of the declared parameter types
- The function results must be a valid instance of the declared result type

Example Function

- Find the maximum depth of the document named "partlist.xml"

```
declare function local:depth($e as node()) as
   xs:integer
{
   (: A node with no children has depth 1 :)
   (: Otherwise, add 1 to max depth of children :)

   if (fn:empty($e/*))
       then 1
       else fn:max(for $c in $e/* return
   local:depth($c)) + 1
};

local:depth(fn:doc("partlist.xml"))
```

Another Example

- Prepare a summary of employees that are located in Denver.

```
declare function local:summary($emps as element(employee)*)
    as element(dept)*
{
    for $d in fn:distinct-values($emps/deptno)
    let $e := $emps[deptno = $d]
    return
          <dept>
                  <deptno>{$d}</deptno>
                  <headcount> {fn:count($e)} </headcount>
                  <payroll> {fn:sum($e/salary)} </payroll>
          </dept>
};

local:summary(fn:doc("acme_corp.xml")//employee[location = "Denver"])
```

Joins

- Combine data from multiple sources into a single result
- Example documents
  - Document "parts.xml" contains many <part> elements
    - Each <part> element contains <partno> and <description> subelements
  - Document "suppliers.xml" contains many <supplier> elements
    - Each <supplier> element contains <suppno> and <suppname> subelements
  - Document "catalog.xml"
    - Contains information about the relationships between suppliers and parts
    - Contains many <item> elements, each of which in turn contains <partno>, <suppno>, and <price> subelements

Inner Join Example

- Generate a "descriptive catalog" derived from the catalog document, but containing part descriptions instead of part numbers and supplier names instead of supplier numbers. Order the new catalog alphabetically by part description and secondarily by supplier name.

```
<descriptive-catalog>
   {
     for $i in fn:doc("catalog.xml")//item,
         $p in fn:doc("parts.xml")//part[partno = $i/partno],
         $s in fn:doc("suppliers.xml")//supplier[suppno =
   $i/suppno]
     order by description, suppname
     return
        <item>
           {
           $p/description,
           $s/suppname,
           $i/price
           }
        </item>
}
</descriptive-catalog>
```

Outer Join

- Return names of all the suppliers in alphabetic order, including those that supply no parts; inside each supplier element, list the descriptions of all the parts it supplies, in alphabetic order.

```
for $s in fn:doc("suppliers.xml")//supplier
order by suppname
return
   <supplier>
      {
       $s/suppname,
       for $i in fn:doc("catalog.xml")//item
                  [suppno = $s/suppno],
            $p in fn:doc("parts.xml")//part
                  [partno = $i/pno]
       order by .
       return $p/description
      }
   </supplier>
```

Future of XQuery

- With the emergence of XML
  - Distinctions among various forms of information, such as documents and databases, are quickly disappearing
  - XQuery is designed to support queries against a broad spectrum of information sources
  - The versatility of XQuery will help XML to realize its potential as a universal medium for data interchange.
- Future versions of XQuery may include:
  - Data definition facilities for persistent views
  - Function overloading and polymorphic functions
  - Facilities for updating XML data
  - An extensibility mechanism whereby function libraries can be created, containing functions implemented in various programming languages

Τέλος Ενότητας

Με τη συγχρηματοδότηση της Ελλάδας και της Ευρωπαϊκής Ένωσης