CS157A:
Introduction to Database Management Systems

Chapter 12: XQuery

Suneuy Kim

XQuery

- A standard for high-level querying of databases containing data in XML form.
- Uses the same data model for XPath. That is, all values produced by XQuery are sequence of items.

XQuery: Sequences

- Sequences are created using parenthesis with strings inside quotes or double quotes and numbers as such. XML elements can also be used as the items of a sequence.
- A sequence is ordered; their items have ordinal position, starting at 1, and may include duplicates.

Example: Sequences

- You can use single or double quotes, but for most character strings a single quote is used.

('a', 'b', 'c', 'd', 'e', 'f')

("apple", 'banana', "carrot", 'dog', "egg", 'fig')

- You can also intermix data types.

('a', 'b', 'c', 1, 2, 3)

- You can also store XML elements in a sequence.

('apple', <banana/>, <fruit type="carrot"/>, <animal type='dog'/>, <vehicle>car</vehicle>)

XQuery Basic Syntax Rules (from w3schools.com)

- XQuery is case-sensitive
- XQuery elements, attributes, and variables must follow the XML naming rules.
- An XQuery string value can be in single or double quotes
- An XQuery variable is defined with a $ followed by a name, e.g. $bookstore
- XQuery comments are delimited by (: and :), e.g. (: XQuery Comment :)

XQuery: FLWR

1. A combination of at least one for or let
2. Optional where clause

(Note: where clause works together with for, not with let.)

3. Exactly one return clause

FLWR: for clause

for $x **in** xquery expression

- At each iteration, the variable is assigned to each item in the sequence denoted by the expression.
- What follows for clause will be executed once for each value of the variable.

FLWR: let clause

- `let variable := expression`
  - The sequence of items defined by the expression becomes the value of the variable.
  - Example

```
let $stars:=doc("stars.xml")
```

FLWR: where clause

- where clause works together with for, not with let.
- At each iteration of the nested loops, evaluate where clause if any.
- If the where clause returns TRUE, invoke the return clause, and append its value to the output.

FLWR: return clause

- The sequence of items produced by the expression is appended to the sequence of items produced so far.
- Do not be confused with return statement in Java.
- It is illegal to return an attribute. Do return data(attribute).

Example:StarMovieData.xml

```
<StarMovieData>
  <Star starID = "cf" starredIn = "sw">
    <Name>Carrie Fisher</Name>
    <Address><Street>123 Maple St.</Street><City>Holly wood</City></Address>
    <Address><Street>5 Locust Ln.</Street> <City>Malibu</City></Address>
  </Star>
  <Star starID = "mh" starredID = "sw">
    <Name>Mark Hamil </Name>
    <Street>456 Oak Rd.</Street>
    <City>Brentwood</City>
  </Star>
  <Movie movieID="sw" starsOf = "cf">
     <Title>Star Wars</Title><Year>1977</Year>
 </Movie>
</StarMovieData>
```

Example

```
declare base-uri "file:///Users/skim/xquery/";
let $smd := doc("StarMovieData.xml")
for $s in $smd/StarMovieData/Star
where $s/(data(@starID)= "mh")
return $s/Name
```

```
Output:
<Name>Mark Hamil</Name>
```

Replacement of variables by their values

```
declare base-uri " file:///Users/skim/xquery/";
let $movies := doc("movies.xml")
for $m in $movies/Movies/Movie
return <Movie title =
"$m/@title">$m/Version/Star</Movie>
```

What would be the output ?

```
<Movie title="$m/@title">$m/Version/Star</Movie>
<Movie title="$m/@title">$m/Version/Star</Movie>
```

Why ?

Replacement of variables by their values

```
let $movies := doc("/Users/skim/xquery/movies.xml")
for $m in $movies/Movies/Movie
return <Movie title =
"{$m/@title}">{$m/Version/Star}</Movie>
```

With **curly braces**, $m/@title and $m/Version/Star are interpreted as XPath expressions, not literals.

Result:

```
<?xml version="1.0" encoding="UTF-8"?>
<Movie title="King Kong">
  <Star>Jeff Bridges</Star>
  <Star>Jessica Lange</Star>
  <Star>Carrie Fisher</Star>
  <Star>Carrie Fisher</Star>
</Movie>
<Movie title="Footloose">
  <Star> Kevin Bacon</Star>
  <Star>John Lithgow</Star>
  <Star>Sarah Jessica Parker</Star>
</Movie>
<Movie title="Amadeus">
  <Star>F. Murray Abraham</Star>
</Movie>
```

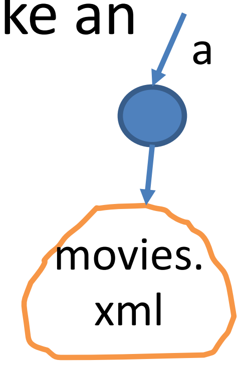
Replacement of variables by their values

```
let $starSeq := (
  let $movies := doc("Users/skim/xquery/movies.xml")
  for $m in $movies/Movies/Movie
  return $m/Version/Star
)
return <Stars>{$starSeq}</Stars>
```

```
<?xml version="1.0" encoding="UTF-8"?>
<Stars>
   <Star>Jeff Bridges</Star>
   <Star>Jessica Lange</Star>
   <Star>Carrie Fisher</Star>
   <Star>Carrie Fisher</Star>
   <Star> Kevin Bacon</Star>
   <Star>John Lithgow</Star>
   <Star>Sarah Jessica Parker</Star>
   <Star>F. Murray Abraham</Star>
</Stars>
```

Comparisons in XQuery

- Comparisons imply "**there exists**" sense.
- A xml element comes with an identity so that you can make an identity comparison.



```
let $movies := doc("/Users/skim/xquery/movies.xml")
return <a>{$movies}</a>/* = $movies
true (contents are compared.)
```

```
let $movies := doc("/Users/skim/xquery/movies.xml")
return <a>{$movies}</a>/* is $movies
false (identities are compared.)
```

- Sequences resulting from the same xquery expression are identical

```
let $movies1 := doc("/Users/skim/xquery/movies.xml")/Movies
let $movies2 := doc("/Users/skim/xquery/movies.xml")/Movies
return $movies1 is $movies2
true
```

- Element whose value is a string is coerced to that string

<a>test</a> = "test" → will be true

Comparisons in XQuery

- Comparing values
  - `=, !=, <, <=, >, >=` implied existential semantics
  - `eq, ne, lt, le, gt, ge` compares single atomic values
- Comparing nodes (sequences or XML elements)
  - **`is`** compare two nodes based on identity
  - `<<` compare two nodes based on document order
  - `deep-equal` if they have all the same attributes and have children in the same order (structure)

=, !=, <, <=, >, >=

- Existential comparison: They compare values of two sequences and return true if any pair of elements from the two sequences satisfy the relation.

  (1,2,3) = (3,4) → true

  (1,2,3) >= (3,4) → true
- The string comparisons will be done lexicographically.

eq, ne, lt, le, gt, ge

- To compare single values or sequences of single or no items.
- Fail if either operand is a sequence of multiple items.
- String does not promote to a number type automatically. If you want to compare values as numbers, you must convert it to number.

e.g.)

```
for $i in ("1","3","5"), $j in (2,4,6)
where xs:integer($i) lt $j
return <pair>{$i},{$j}</pair>
```

Example: Existential nature of comparison

To find the name(s) of Star(s) who live at 123 Maple St., Malibu from Stars.xml.

```
declare option saxon:output
"indent=yes";

let $stars :=
doc("/Users/skim/xquery/stars.xm
l")
for $s in $stars/Stars/Star
where $s/Address/Street = "123
Maple St." and $s/Address/City=
"Malibu"
return $s/Name
```

```
<?xml version="1.0" encoding="UTF-8"?>
<Name>Carrie Fisher</Name> ← wrong !
<Name>Tom Hanks </Name>
```

```
<Stars>
  <Star>
   <Name>Carrie Fisher</Name>
   <Address>
        <Street>123 Maple St.</Street>
        <City>Hollywood</City>
   </Address>
   <Address>
       <Street>5 Locust Ln.</Street>
       <City>Malibu</City>
   </Address>
   </Star>
  <Star>
  <Name>Tom Hanks </Name>
  <Address>
       <Street>123 Maple St.</Street>
       <City>Malibu</City>
   </Address>
   </Star>
</Stars>
```

Another attempt

```
declare option saxon:output "indent=yes";

let $stars :=
doc("/Users/skim/xquery/stars.xml")
for $s in $stars/Stars/Star
where $s/Address/Street eq "123 Maple St."
and $s/Address/City eq  "Malibu"
return $s/Name

Runtime error !
```

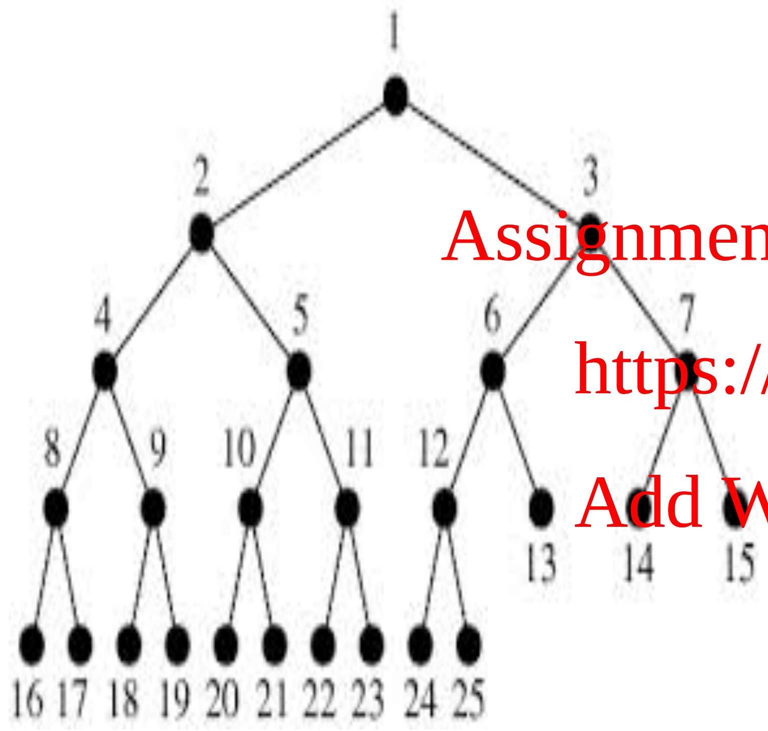
Example: = vs. eq

Suppose $s/Address/Street produces a sequence "123 Maple St." and "5 Locust Ln.",

- `$s/Address/Street = "123 Maple St."` is true (however based on the existential comparison)

- `$s/Address/Street eq "123 Maple St."` is error !

Axis Example



Axis	results
self	5
ancestor	1,2
ancestor-or-self	1,2,5
parent	2
child	10,11
descendant	10,11,20,21,22, 23
descendant-or-self	5,10,11,20,21,22,23
following	3,6,7,12,13,14,15,24,25
preceding	4,8,9,16,17,18,19
following-sibling	6,7
preceding-sibling	4

```xml
<?xml version="1.0" encoding="UTF-8"?>
<x1>
 <x2>
  <x4>
     <x8><x16></x16><x17></x17></x8>
     <x9><x18></x18><x19></x19></x9>
  </x4>
  <x5>
     <x10><x20></x20><x21></x21></x10>
     <x11><x22></x22><x23></x23></x11>
  </x5>
 </x2>
 <x3>
  <x6>
     <x12><x24></x24><x25></x25></x12><x13></x13>
  </x6>
  <x7>
     <x14></x14><x15></x15>
  </x7>
 </x3>
</x1>
```

XML data corresponding to the tree structure of the data in pp.82.

A solution using axis

Find the star(s) who lives at 123 Maple St., Malibu from Stars.xml.

```
declare option saxon:output "indent=yes";
let $stars := doc("/Users/skim/xquery/stars.xml")
for $star in $stars/Stars/Star
  for $street in $star//Street
  where ($street = "123 Maple St." and $street/following-sibling::City ="Malibu")
  return <Star>{$star/Name} </Star>
```

Node Comparison: is

- To compare single node for identity

```
let $a:= <a>test</a>
let $b:= <a>test</a>
return $a is $b → false
vs.
let $a:= <a>test</a>
let $b:= $a
return $a is $b → true
```

Node comparison: deep-equal

- To traverse the tree structure of nodes (XML elements or sequences) to see if they are identical in structure and value.

Examples returning false:

deep-equal ((1,2), (2,1))

deep-equal(<t a="1">z</t>, <t **b**="1">z</t>)

Examples returning true:

deep-equal(doc('movies.xml'),doc('movies.xml'))

deep-equal(<a>123</a>,<a>123</a>)

deep-equal(<t a="1">z</t>, <t a="1">z</t>)

Order Comparison Operators

To compare the positions of two XML elements in an XML document

- op1 << op2 returns true if op1 precedes op2 in a document order.

- op1 >> op2 returns true if op1 follows op2 in a document order.

Example: Order Comparison

Produce a XML document that contains every combination of two students from grades.xml, in which the order does not matter.

```
declare option saxon:output "indent=yes";
let $g:=
doc("/Users/skim/xquery/grades.xml")/Grades
for $s1 in $g/Student, $s2 in $g/Student
where $s1 << $s2
return <pair>{($s1/Name,$s2/Name)}</pair>
```

```
<Grades>
<Student>
   <Name>Ming Scout </Name>
   <Exams>
     <Exam>99</Exam>
     <Exam>79</Exam>
   </Exams>
 </Student>
<Student>
   <Name>John Lee</Name>
   <Exams>
     <Exam>50</Exam>
     <Exam>69</Exam>
   </Exams>
 </Student>
</Grades>
```

Finding ordinal positions in Sequence

Within a FLWR expression, the `for` clause has a mechanism to track the ordinal position of currently iterated item using the `at`-clause.

```
declare option saxon:output "indent=yes";
<result>
{for $fruit at $index in("apple", "banana", "grape")
 return <fruit position="{$index}">{$fruit} </fruit>
}
</result>
```

```
<?xml version="1.0" encoding="UTF-8"?>
<result>
  <fruit position="1">apple</fruit>
  <fruit position="2">banana</fruit>
  <fruit position="3">grape</fruit>
</result>
```

Example

Find all students who scored below 70 in any exam. Show their names, the scores, and which exam it is.

```
declare option saxon:output"indent=yes";

let $roster := doc("/Users/skim/xquery/grades.xml")/Grades
for $s in $roster/Student
for $exam at $index in $s/Exams/Exam
 where xs:integer($exam)lt 70
   return  <concerned>
           <name>{data($s/Name)}</name>
           <exam>{$index}</exam>
           <score>{data($exam)}</score>
         </concerned>
```

Nested loop in XQuery

- Doubly nested loop

```
for
$s1 in $movies/Movies/Movie/Version/Star,
$s2 in $stars/Stars/Star

=

for $s1 in $movies/Movies/Movie/Version/Star
for $s2 in $stars/Stars/Star
```

Joins: Example

```
declare option saxon:output "indent=yes";
let
    $movies :=
doc("/Users/skim/xquery/movies.xml"),
    $stars :=
doc("/Users/skim/xquery/stars.xml")
for $s1 in $movies/Movies/Movie/Version/Star,
    $s2 in $stars/Stars/Star
where data($s1) = data($s2/Name)
return $s1
```

Elimination of Duplicates

- The `distinct-values($args)` function returns a sequence of unique atomic values from $arg.
- The $arg sequence can contain atomic values or nodes, or a combination of the two
- The nodes in the sequence have their typed values extracted. This means that only the *contents* of the nodes are compared, not their names.
- Example:

distinct-values(("apple", <a>apple</a>, ("apple","apple")))

→ apple

distinct-values:Example

```
declare option saxon:output "indent=yes";
let $starSeq := distinct-values (
 let $movies :=
doc("/Users/skim/xquery/movies.xml")
 for $m in $movies/Movies/Movie
 return $m/Version/Star
)
return <Stars>{$starSeq}</Stars>
```

Note: A space counts in comparison.

Without distict-values(), Carry Fisher appears 2 times.

Exercise

Write a XQuery program that produces unique stars from movies.xml in the format of

```
<Stars>
  <Star> </Star>
   ….
  <Star> </Star>
</Stars>
```

Universal Quantifier: `every`

declare option saxon:output "indent=yes";

let $stars := doc("/Users/skim/xquery/stars1.xml")

for $s in $stars/Stars/Star

where every $c in $s/Address/City satisfies $c = "Hollywood"

return $s/Name

[Q1] $s//City

[Q2] What if a star’s resident consists of Street and City without Address? every $c in () is always true.

Existential Quantifier: `some`

```
declare option saxon:output "indent=yes";

let $stars := doc("/Users/skim/xquery/stars1.xml")
for $s in $stars/Stars/Star
where some $c in $s/Address/City satisfies $c =
"Hollywood"
return $s/Name
```

The where cause is Identical to

```
where $s/Address/City = "Hollywood"
```

Aggregations: sum, count, max/min

Find the sum, count, average, and max of the first exams.

```
declare option saxon:output "indent=yes";

let $roster :=
doc("/Users/skim/xquery/grades.xml")/Grades
let $ex1:=
    $roster/Student/Exams/Exam[1]
return (<sum>{sum($ex1)}</sum>,
        <count>{count($ex1)}</count>,
        <avg>{avg($ex1)}</avg>,
        <max>{max($ex1)}</max>)
```

Effective Boolean Value

The *EBV* of an expression is:

1. The actual value if the expression is of type boolean.
2. FALSE if the expression evaluates to 0, "" [the empty string], or () [the empty sequence].
3. TRUE otherwise.

Example:

- @year ="1976" is true if the value of year attribute is 1976.
- /Movies/Movie/Version[@year ="1976"] is true if some move version is made at 1976.

Boolean Operators

- and, or, not
- Take boolean values of the expressions first

e.g.) not (3 eq 5 or 0) is true

- Functions true()/false() returns true/false

if then else

- if (E_1) then E_2 else E_3 is evaluated by:
  - Compute the EBV of E_1.
  - If true, the result is E_2; else the result is E_3.

Example: if-then-else

- Find the students who scored below 70 on exam2. Show their names and scores.

```
declare option saxon:output "indent=yes";
let $g:=
doc("/Users/skim/xquery/grades.xml")/Grades
for $s in  $g/Student
let $ex2 := $s/Exams/Exam[2]
return
   if (data($ex2) < 70) then
      (data($s/Name), data($ex2))
   else ()
```

Example: if-then-else

- Find the names and scores of all students who scored higher on exam 2 than on exam 1.

```
let $g:=
doc("/Users/skm/xquery/grades.xml")//Grades
for $s in $g/Student
let $ex1:= $s/Exams/Exam[1]
let $ex2:= $s/Exams/Exam[2]
return  if (data($ex2) > data($ex1)) then
             <Result>{$s/Name}{$ex1}{$ex2}</Result>
            else ()
```

FLOWR: order by

- The optional `order by` clause is used in FLOWR expression to specify the sort order of the result.
- It takes expressions that specify the sorting properties.
- The default order is ascending, and the explicit use of keyword `descending` will reverses the order.

Example: order by

Consider all versions of all movies, order them by year, and produce a sequence of Movie elements with the title and year as attributes.

```
let $movies :=
doc("/Users/skim/xquery/movies.xml")
for $m in $movies/Movies/Movie, $v in
$m/Version
order by $v/@year, $m/@title
return
  <Movie title = "{$m/@title}" year =
"{$v/@year}"/>
```

Example: order by

Find the average score on the exams for each student. Produce a sequence of students consisting of name and average score. Sort the sequence by descending order of average score

```
for $s in
doc("/Users/skim/xquery/grades.xml")/Grades/Student
let $avg := avg($s/Exams/Exam)
order by $avg descending
return
<Student>
   {$s/Name}
   <average>{$avg}</average>
</Student>
```

Example: Match Pattern

pattern	meaning
match="section//title"	Matches any <title> elements contained within <section> elements.
match="section/title[@short-name]"	Matches <title> elements that are children of <section> elements, and that have a short-name attribute.
match="appendix//section[@type='reference']/title"	Matches <title> elements that are children of <section> elements. The section must also have a type attribute with the value "reference", and have an <appendix> ancestor.
match="appendix[.//section[@type='reference']/title]"	Matches <appendix> elements that contain <section> descendants. These <section> elements, in turn, must have both a type attribute with the value "reference" and a <title> child.