Introduction to Data Management CSE 344

Lecture 14: XQuery, JSON

Announcements

- WQ5 (XML) due Tuesday night
- HW4 due Thursday night
  - From now on only one week for homeworks
  - Start early!
  - Attend sections!
    - To prepare for the HW of the following week
    - To avoid any last minute login issues
    - To ask questions on the HW due on that day
  - Check out discussion board regularly or subscribe!
    - Your question might already have been answered there
    - Otherwise, post your question
- Update: At most one late day allowed on HW8

Review: XML Data

```
<bib>
    <book>  <publisher> Addison-Wesley </publisher>
            <author> Serge Abiteboul </author>
            <author> <first-name> Rick </first-name>
                     <last-name> Hull </last-name>
            </author>
            <author> Victor Vianu </author>
            <title> Foundations of Databases </title>
            <year> 1995 </year>
    </book>
    <book price="55">
            <publisher> Freeman </publisher>
            <author> Jeffrey D. Ullman </author>
            <title> Principles of Database and Knowledge Base Systems </title>
            <year> 1998 </year>
    </book>
</bib>
```

Querying XML Data

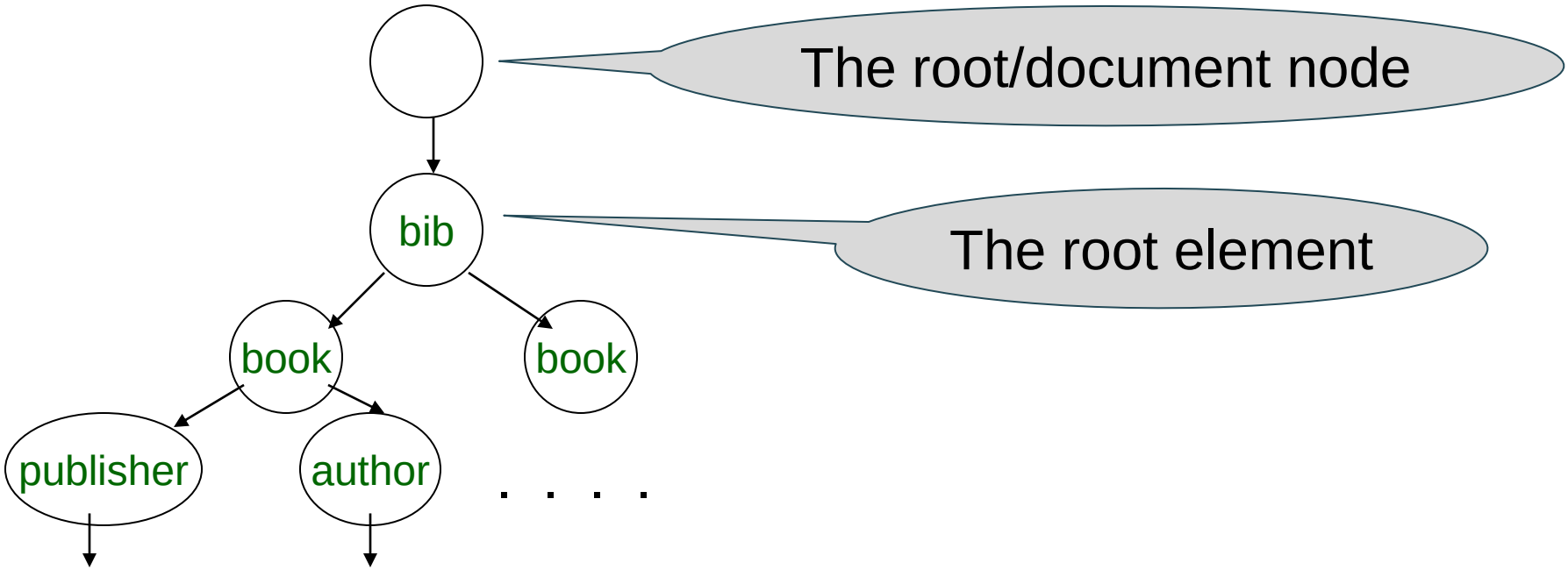
- XPath = simple navigation (last lecture)
- XQuery = the SQL of XML (this lecture)
- XSLT = recursive traversal (will not discuss in class)
  - Extensible Stylesheet Language

- Think of XML/XQuery as one of several data exchange solutions.
  - Another solution: Json/Jsoniq http://www.jsoniq.org/

Data Model for XPath

XPath returns a sequence of items. An item is either:

- A value of primitive type, or
- A node (doc, element, or attribute)



XPath: Simple Expressions

Returns a sequence of elements

```
/bib/book/year
```

Result: <year> 1995 </year>
<year> 1998 </year>

```
/bib/paper/year
```

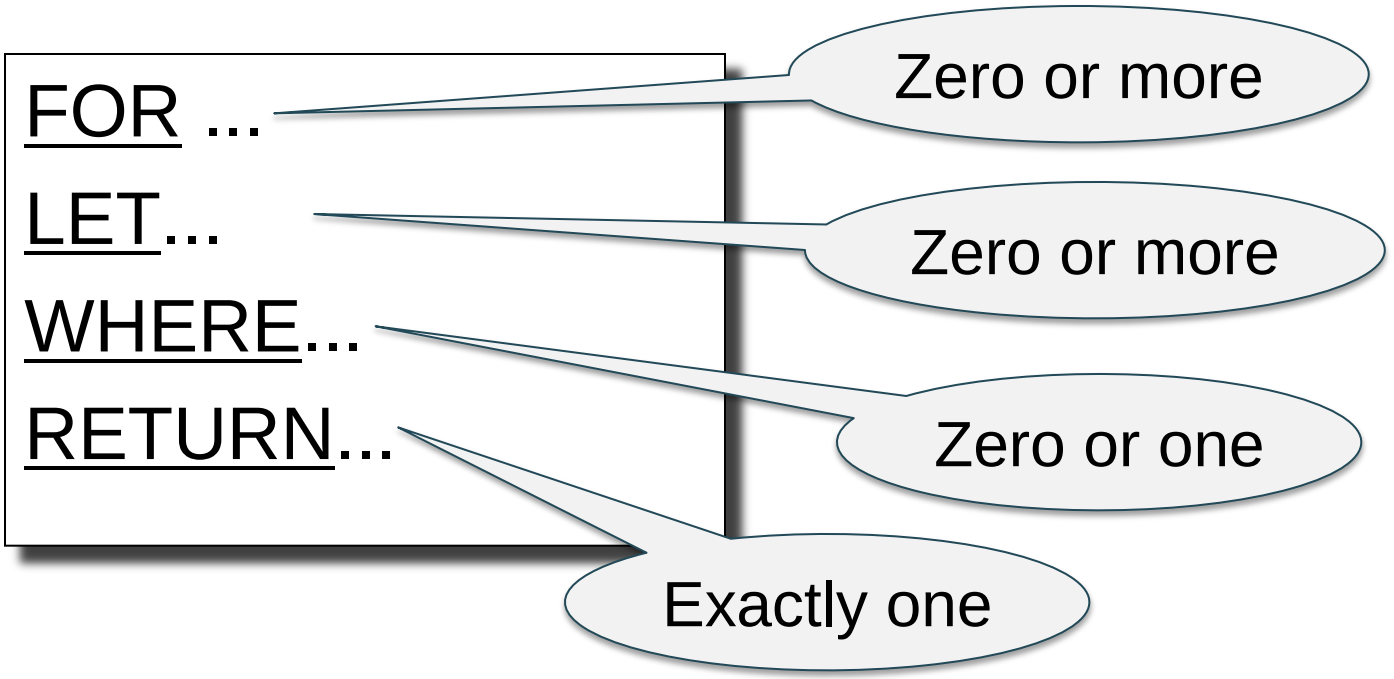
Result: empty (there were no papers)

XQuery

- Standard for high-level querying of databases containing data in XML form
- Based on Quilt, which is based on XML-QL
- Uses XPath to express more complex queries
- Chapter 12.2

FLWR ("Flower") Expressions

Like Select-From-Where in SQL…



FOR-WHERE-RETURN

Find all book titles published after 1995:

```
FOR $x IN doc("bib.xml")/bib/book
WHERE $x/year/text() > 1995
RETURN $x/title
```

XQuery is case sensitive,
write in lowercase
"let, for" etc.

Result:

```
<title> abc </title>
<title> def </title>
<title> ghi </title>
```

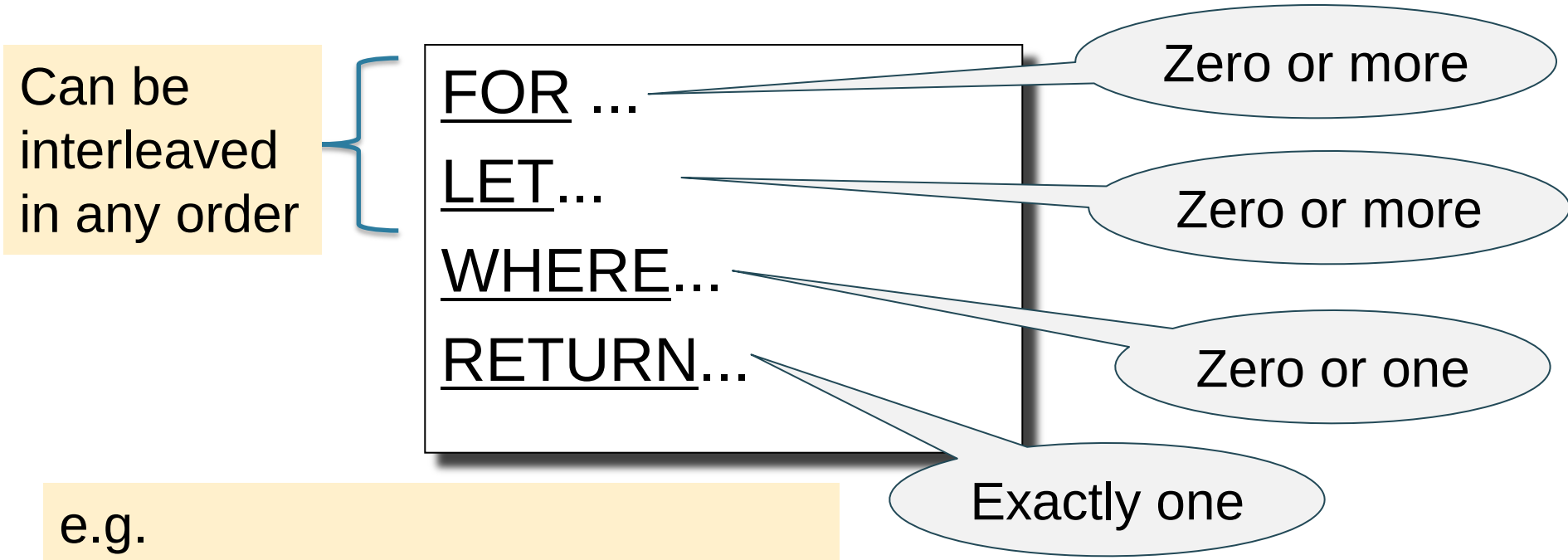
XQuery vs. XPath

- Same model for values produced (sequence of items)
- Items are primitive or nodes (doc, element, attributes)
- Every XPath expression is also an XQuery expression (with a RETURN)
- But, XQuery allows much more (FLWR) like SQL (Select-From-Where)

XQuery vs. SQL

- (They operate on different data models)
- XQuery is a “functional language”, an XQuery expression can be used wherever an expression is expected
- Like SQL subqueries, but more expressive
  - e.g. We cannot use <subquery> <= <subquery>
    in SQL WHERE clause
- Has Pros and Cons
  - More general
  - But every operator must make sense when applied to a sequence of multiple items

FLWR ("Flower") Expressions



e.g.

for… for… let…. for…. let ….
(optional) where ….
(exactly one) return …

FOR-WHERE-RETURN

Simple example:

RETURN <greeting>Hello World!</greeting>

Find all book titles published after 1995:

```
FOR $x IN doc("bib.xml")/bib/book
WHERE $x/year/text() > 1995
RETURN $x/title
```

Result:

<title> abc </title>
<title> def </title>
<title> ghi </title>

FOR: evaluates <expression>, var is assigned to each item in the expression
WHERE: <condition> is evaluated to true or false

Let's try to write shorter equivalent XQueries

Find all book titles published after 1995
(hint: use XPath)

```
FOR $x IN doc("bib.xml")/bib/book
WHERE $x/year/text() > 1995
RETURN $x/title
```

```
FOR $x IN doc("bib.xml")…
RETURN $x
```

```
RETURN ....
```

FOR-WHERE-RETURN

Equivalently (perhaps more geekish)

```
FOR $x IN doc("bib.xml")/bib/book[year/text() > 1995] /title
RETURN $x
```

And even shorter:

```
RETURN  doc("bib.xml")/bib/book[year/text() > 1995] /title
```

RETURN is not the “return statement”,
can be executed multiple times inside FOR

COERCION

The query:

```
FOR $x IN doc("bib.xml")/bib/book[year > 1995] /title
RETURN $x
```

Is rewritten by the system into:

```
FOR $x IN doc("bib.xml")/bib/book[year/text() > 1995] /title
RETURN $x
```

FOR-WHERE-RETURN

- Find all book titles and the year when they were published:

```
FOR $x IN doc("bib.xml")/ bib/book
RETURN <answer>
                <title>{ $x/title/text() } </title>
                <year>{ $x/year/text() } </year>
            </answer>
```

Note the curly braces, otherwise treated as text inside tags (next slide)

Result:

```
<answer> <title> abc </title>  <year> 1995 </ year > </answer>
<answer> <title> def </title>  < year > 2002 </ year > </answer>
<answer> <title> ghk </title>  < year > 1980 </ year > </answer>
```

FOR-WHERE-RETURN

- Notice the use of “{“ and “}”
- What is the result without them ?

```
FOR $x IN doc("bib.xml")/ bib/book
RETURN <answer>
                <title>  $x/title/text() </title>
                <year> $x/year/text() </year>
              </answer>
```

<answer> <title> $x/title/text() </title> <year> $x/year/text() </year> </answer>

<answer> <title> $x/title/text() </title> <year> $x/year/text() </year> </answer>

<answer> <title> $x/title/text() </title> <year> $x/year/text() </year> </answer>

Nesting

- For each author of a book by Morgan Kaufmann, list all books he/she published:

```
FOR $b IN doc("bib.xml")/bib,
        $a IN $b/book[publisher /text()="Morgan Kaufmann"]/author
RETURN <result>
                  { $a,
                     FOR $t IN $b/book[author/text()=$a/text()]/title
                     RETURN $t
                  }
               </result>
```

In the RETURN clause comma concatenates XML fragments

FOR clause works like FROM clause, joins docs/elements

Result

```
<result>
        <author>Jones</author>
        <title> abc </title>
        <title> def </title>
  </result>
  <result>
        <author> Smith </author>
        <title> ghi </title>
  </result>
```

Aggregates

Find all books with more than 3 authors:

```
FOR $x IN doc("bib.xml")/bib/book
WHERE count($x/author)>3
RETURN $x
```

count = a function that counts
avg = computes the average
sum = computes the sum
distinct-values = eliminates duplicates

Aggregates

```
FOR $x IN doc("bib.xml")/bib/book
WHERE count($x/author)>3
RETURN $x
```

Same thing:

```
FOR $x IN doc("bib.xml")/bib/book[count(author)>3]
RETURN  $x
```

Eliminating Duplicates

Print all authors:

```
FOR $a IN distinct-values($b/book/author/text())
RETURN <author> { $a } </author>
```

Note: distinct-values applies ONLY to values, NOT elements
If applied to elements, it discards the tags and does not put them back.

The LET Clause

Find books whose price is larger than average:

```
FOR $b in doc("bib.xml")/bib
LET $a:=avg($b/book/price/text())
FOR $x in $b/book
WHERE $x/price/text() > $a
RETURN $x
```

LET enables us to declare variables
Note: := and not =

Flattening

Compute a list of (author, title) pairs

Input:

```
<bib>
  <book>
     <title> Databases </title>
     <author> Widom </author>
     <author> Ullman </author>
  </book>
</bib>
```

Output:

```
<answer>
   <title> Databases </title>
   <author> Widom </author>
</answer>
<answer>
   <title> Databases </title>
   <author> Ullman </author>
</answer>
```

```
FOR $b IN doc("bib.xml")/bib/book,
        $x IN $b/title/text(),
        $y IN $b/author/text()
RETURN <answer>
                <title> { $x } </title>
                <author> { $y } </author>
            </answer>
```

Re-grouping

For each author, return all titles of her/his books

```
FOR $b IN doc("bib.xml")/bib,
        $x IN $b/book/author/text()
RETURN
  <answer>
     <author> { $x } </author>
     { FOR $y IN $b/book[author/text()=$x]/title
        RETURN $y}
  </answer>
```

Result:

```
<answer>
    <author> efg </author>
     <title> abc </title>
     <title> klm </title>
     . . . .
</answer>
```

Note:

- $x denotes a **value** text(), <author> {$x} </author> is needed
- $y denotes an **element**, <title> $y </title> not needed

Re-grouping

What about duplicate authors ?

Eliminate duplicate authors:

```
FOR $b IN doc("bib.xml")/bib
LET $a := distinct-values($b/book/author/text())
FOR $x IN $a
RETURN
  <answer>
     <author> $x </author>
     { FOR $y IN $b/book[author/text()=$x]/title
        RETURN $y}
  </answer>
```

Re-grouping

Same thing (eliminate redundant “let, for”)

```
FOR $b IN doc("bib.xml")/bib,
        $x IN distinct-values($b/book/author/text())
RETURN
  <answer>
     <author> $x </author>
     { FOR $y IN $b/book[author/text()=$x]/title
        RETURN $y}
  </answer>
```

SQL and XQuery Side-by-side (FLWOR)

Product(pid, name, maker, price)
assume tuple == <row> .. </row>

Find all product names, prices, sort by price

```
SELECT x.name,
       x.price
FROM Product x
ORDER BY x.price
```

SQL

```
FOR $x in doc("db.xml")/db/Product/row
ORDER BY $x/price/text()
RETURN

<answer>

{ $x/name,  $x/price }
  </answer>
```

XQuery

Optional "order by" clause
"ascending" or "descending"

XQuery's Answer

```
<answer>
      <name>  abc  </name>
      <price>  7 </price>
</answer>
<answer>
      <name>  def  </name>
      <price>  23 </price>
</answer>
  . . . .
```

Notice: this is NOT a well-formed document ! (WHY ???)

Producing a Well-Formed Answer

```
<myQuery>
  {  FOR $x in doc("db.xml")/db/Product/row
     ORDER BY $x/price/text()
     RETURN <answer>
                    { $x/name,  $x/price }
                 </answer>
   }
</myQuery>
```

XQuery's Answer

```
<myQuery>
  <answer>
        <name>  abc  </name>
        <price>  7 </price>
  </answer>
  <answer>
        <name>  def  </name>
        <price>  23 </price>
  </answer>
    . . . .
</myQuery>
```

Now it is well-formed !

SQL and XQuery Side-by-side

Product(pid, name, maker, price)
Company(cid, name, city, revenues)
in XML, each tuple = "row" element

Find all products made in Seattle

```
SELECT x.name
FROM Product x, Company y
WHERE x.maker=y.cid
      and y.city="Seattle"
```

SQL

```
FOR $r in doc("db.xml")/db,
      $x in $r/Product/row,
      $y in $r/Company/row
WHERE
   $x/maker/text()=$y/cid/text()
   and  $y/city/text() = "Seattle"
RETURN { $x/name }
```

XQuery

Cool XQuery

```
FOR $y in /db/Company/row[city/text()="Seattle"],
      $x in /db/Product/row[maker/text()=$y/cid/text()]
RETURN { $x/name }
```

```
<product>
        <row> <pid> 123 </pid>
                <name> abc </name>
                <maker> efg </maker>
         </row>
         <row> …. </row>
         …
</product>
<product>
   . . .
</product>
. . . .
```

Product(pid, name, maker, price)
Company(cid, name, city, revenues)
in XML, each tuple = “row” element

SQL and XQuery Side-by-side

For each company with revenues < 1M count the products over $100

```
SELECT y.name, count(*)
FROM Product x, Company y
WHERE x.price > 100 and x.maker=y.cid and y.revenue < 1000000
GROUP BY y.cid, y.name
```

```
FOR $r in doc(“db.xml”)/db,
        $y in $r/Company/row[revenue/text()<1000000]
RETURN
    <proudCompany>
        <companyName> { $y/name/text()  }  </companyName>
        <numberOfExpensiveProducts>
         { count($r/Product/row[maker/text()=$y/cid/text()][price/text()>100])}
        </numberOfExpensiveProducts>
    </proudCompany>
```

Product(pid, name, maker, price)
Company(cid, name, city, revenues)
in XML, each tuple = “row” element

SQL and XQuery Side-by-side

Find companies with at least 30 products, and their average price

```
SELECT y.name, avg(x.price)
FROM Product x, Company y
WHERE x.maker=y.cid
GROUP BY y.cid, y.name
HAVING count(*) > 30
```

An element

A collection

```
FOR $r in doc(“db.xml”)/db,
    $y in $r/Company/row
LET $p := $r/Product/row[maker/text()=$y/cid/text()]
WHERE count($p) > 30
RETURN
  <theCompany>
    <companyName> { $y/name/text() }
    </companyName>
    <avgPrice> avg($p/price/text()) </avgPrice>
  </theCompany>
```

Other Constructs

- Quantification
  - every variable in expression1 satisfies expression2
  - some variable in expression1 satisfies expression2
  - some is rarely used

- Branching
  - if (expression1) then expression2 else expression3
  - Note that this is an expression and not an if-then-else statement, else part is necessary, can be () though

XML Summary

- Stands for eXtensible Markup Language
  1. Advanced, **self-describing file format**
  2. Based on a flexible, **semi-structured data model**

- Query languages for XML
  - XPath
  - XQuery

Beyond XML: JSON

- JSON stands for “**J**ava**S**cript **O**bject **N**otation”
  - Lightweight text-data interchange format
  - Language independent
  - “Self-describing" and easy to understand

- JSON is quickly replacing XML for
  - Data interchange
  - Representing and storing semi-structure data

JSON

```
Example from: http://www.jsonexample.com/
myObject = {
    "first": "John",
    "last": "Doe",
    "salary": 70000,
    "registered": true,
    "interests": [ "Reading", “Biking”, "Hacking" ]
}
```

Query language: Jsoniq http://www.jsoniq.org/