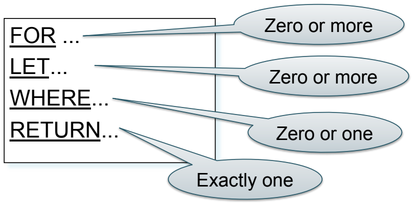
Introduction to Data Management
CSE 344

Lecture 13: XQuery and JSON

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

FLWR ("Flower") Expressions



```
FOR ...
LET...
WHERE...
RETURN...
```

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
```

Result:

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

FOR-WHERE-RETURN

Equivalently (perhaps more geekish)

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

And even shorter:

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

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>
```

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

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:

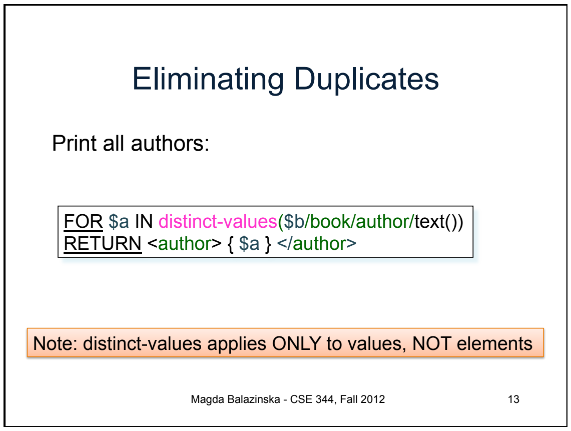
```
FOR $x IN doc("bib.xml")/bib/book
WHERE count($x/author)>3
RETURN $x
```

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

Aggregates

Same thing:

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

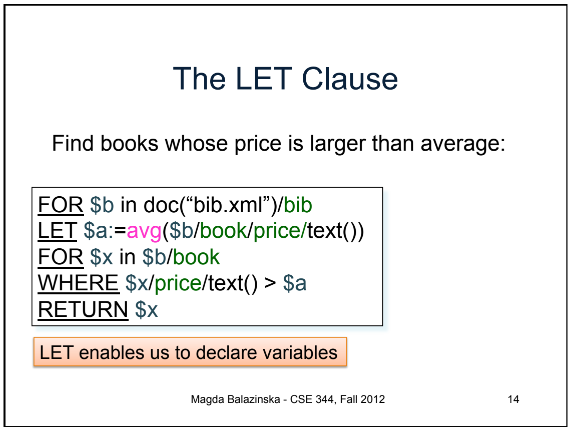


Eliminating Duplicates

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

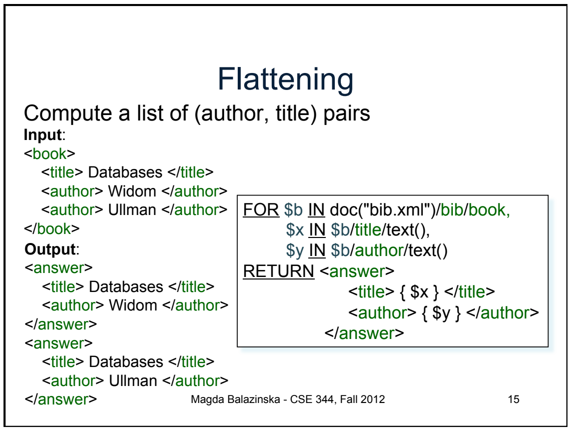


The LET Clause

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



Flattening

Compute a list of (author, title) pairs

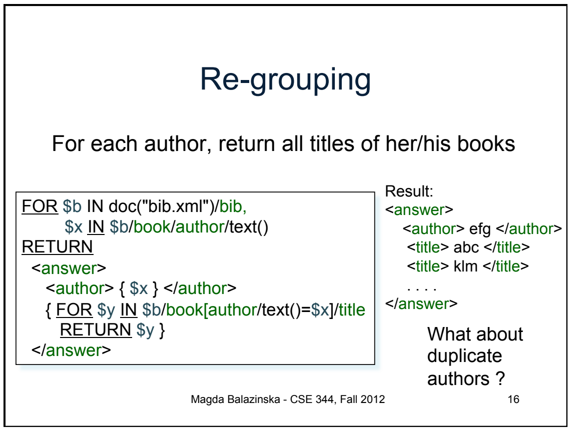
Input:

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

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

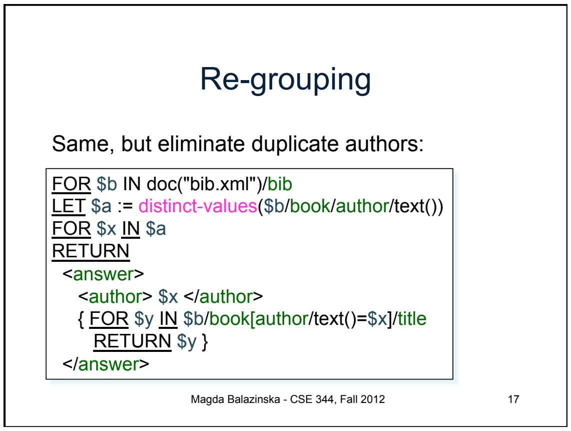
For each author, return all titles of her/his books

```
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>
```

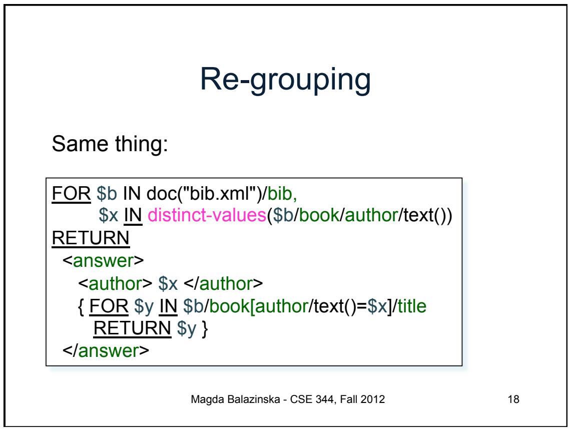
What about duplicate authors ?



Re-grouping

Same, but eliminate duplicate authors:

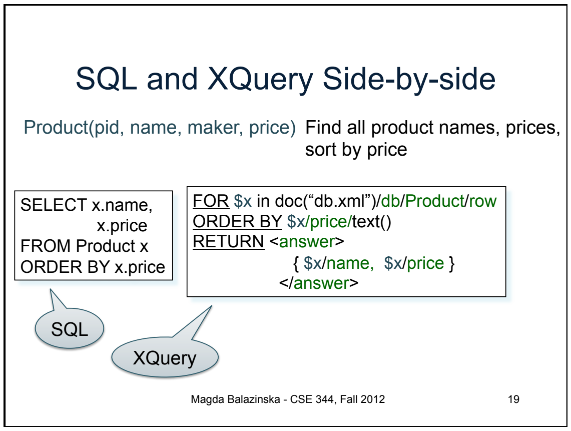
```
FOR $b IN doc("bib.xml")/bib
LET $a := distinct-values($b/book/author/text())
FOR $x IN $a
RETURN
  <answer>
     <author> $x </author>
     { FOR $y IN $b/book[author/text()=$x]/title
        RETURN $y }
  </answer>
```



Re-grouping

Same thing:

```
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

Product(pid, name, maker, price)

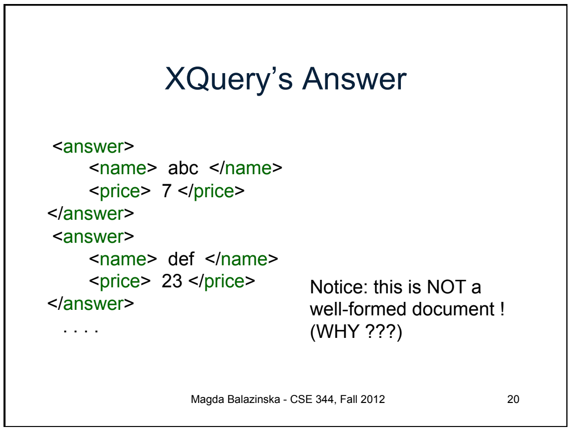
Find all product names, prices, sort by price

```
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>
```

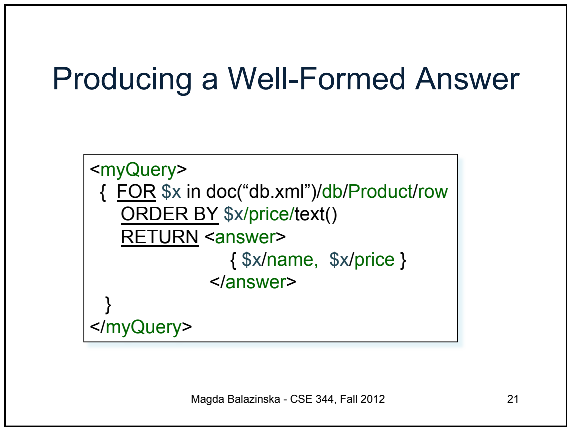
XQuery



XQuery's Answer

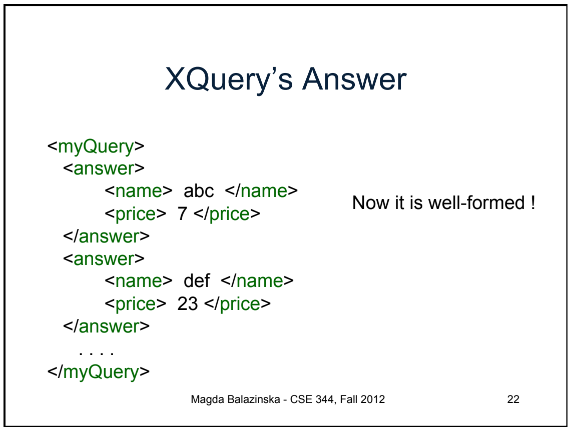
```
 <answer>
      <name>  abc  </name>
      <price>  7 </price>
</answer>
 <answer>
      <name>  def  </name>
      <price>  23 </price>
</answer>
  . . . .
```

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



Producing a Well-Formed Answer

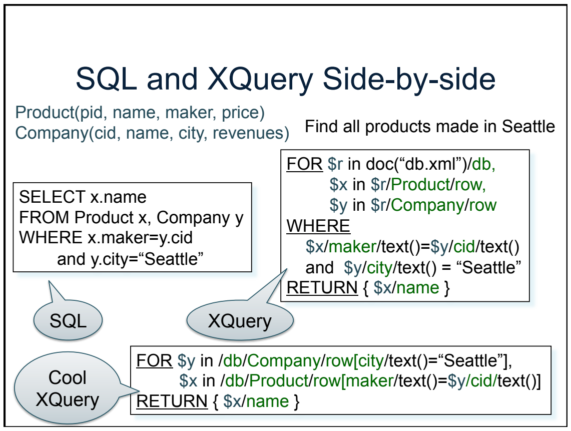
```
<myQuery>
  {  FOR $x in doc("db.xml")/db/Product/row
     ORDER BY $x/price/text()
     RETURN <answer>
                     { $x/name,  $x/price }
                 </answer>
  }
</myQuery>
```



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)

Find all products made in Seattle

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

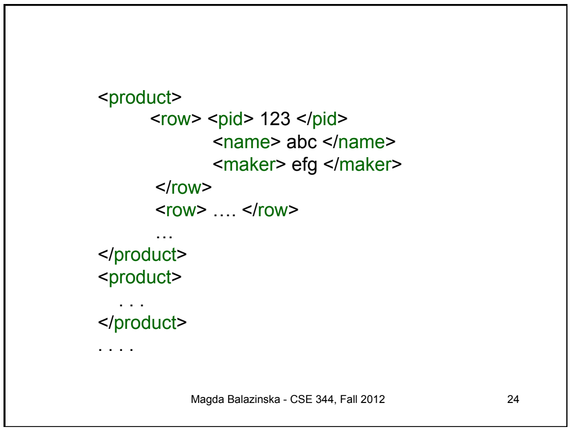
SQL

```
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 }
```

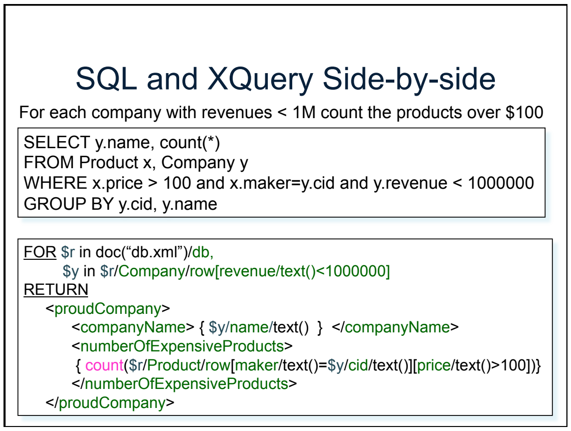
XQuery

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>
. . . .
```

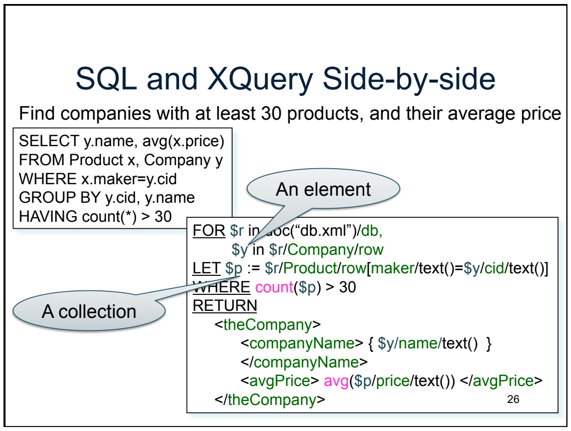


SQL and XQuery Side-by-side

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>
```



SQL and XQuery Side-by-side

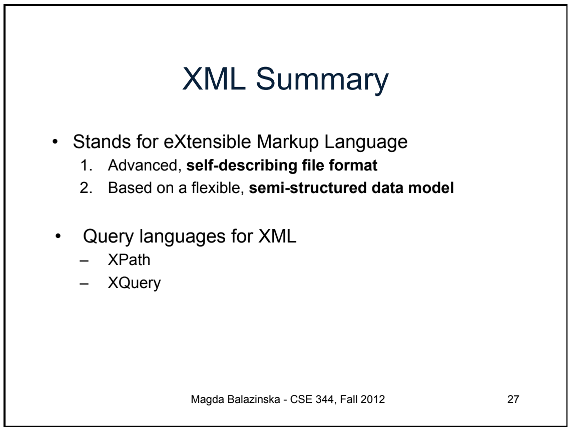
Find companies with at least 30 products, and their average price

```
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

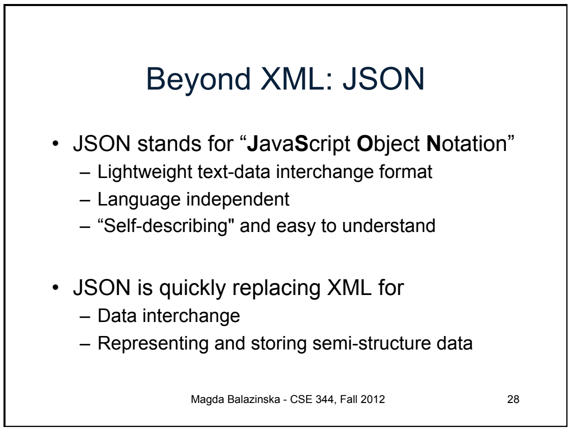
A collection

```
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>
```



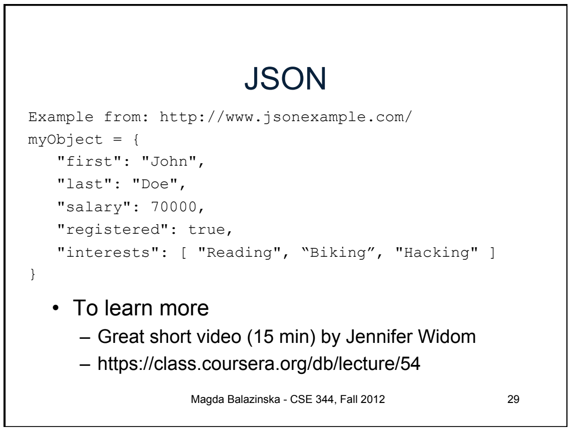
XML Summary

- 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" ]
}
```

- To learn more
  - Great short video (15 min) by Jennifer Widom
  - https://class.coursera.org/db/lecture/54