# ECS 165B: Database System Implementation
# Lecture 19

UC Davis

May 10, 2010

Based on slides due to Zack Ives

# Class Agenda

- Last time:
  - Results of mid-course evaluation
  - Querying XML
- Today:
  - Views and XML Views of Relations
- Reading:
  - none

# Views and Relational Encodings of XML

# Recall XQuery

- "FLWOR" statement:

  `for` {iterators that bind variables}

  `let` {collections}

  `where` {conditions}

  `order by` {order-conditions}

  `return` {output constructor}

- XQuery is ~SQL-like, but cleaner and more orthogonal
- Based on paths and binding tuples, with *collections* and *trees* as its first-class objects
- See www.w3.org/TR/xquery/ for more details on the language

# XQuery Works Well with Schema, and Validates Against It (Incl. Keys)

```
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">
<xsd:complexType name="ThesisType">
    <xsd:attribute name="key" type="xsd:string"/>
    …
    <xsd:sequence>
        <xsd:element name="author" type="xsd:string"/>
        ...
        <xsd:element name="school" type="xsd:string"/>
        …
    </xsd:sequence>
</xsd:complexType>
<xsd:element name="mastersthesis" type="ThesisType">
      <xsd:key name="mtId">
            <xsd:selector xpath="."/>  <xsd:field xpath="@key"/>
      </xsd:key>
      <xsd:keyref name="schoolRef" refer="schoolId">
            <xsd:selector xpath="./school"/>  <xsd:field
      xpath="./text()"/>
      </xsd:keyref>
    </xsd:element>
</xsd:schema>
```

# Alternate Data Representations

XQuery enables us to go from XML ➔ XML

… or XML ➔ XHTML, or XML ➔ text

… In SQL, we go relations ➔ relations

**What about relations ➔ XML and XML ➔ relations?**

Let's start with XML ➔ XML, relations ➔ relations

# Views in SQL and XQuery

- Recall: a view is essentially a named query
- We use the name of the view instead of the query (treating it as if it were the relation it returns)

SQL:

```
CREATE VIEW V(A,B,C) AS
SELECT A,B,C FROM R
WHERE R.A = "123"
```

XQuery:

```
declare function V() as element(content)* {
   for $r in doc("R")/root/tree,
      $a in $r/a, $b in $r/b, $c in $r/c
   where $a = "123"
   return <content>{$a, $b, $c}</content>
}
```

*Using the views:*

```
SELECT *
FROM V, R
WHERE V.B = 5
   AND V.C = R.C
```

```
for $v in V()/content,
 $r in doc("r")/root/tree
where $v/b = $r/b
return $v
```

# What's Useful about Views

Providing security/access control

- We can assign users permissions on different views
- Can select or project so we only reveal what we want!

Can be used as relations in other queries

- Allows the user to query things that make more sense

Describe *transformations* from one schema (the base relations) to another (the output of the view)

- The basis of converting from XML to relations or vice versa
- This will be incredibly useful in data integration, discussed soon…

Allow us to define *recursive* queries

# Materialized vs. Virtual Views

- A virtual view is a named query that is actually re-computed every time – view predicate replaced by its definition in query:

```
CREATE VIEW V(A,B,C) AS
SELECT A,B,C FROM R
WHERE R.A = "123"
```

```
SELECT *
FROM V, R
WHERE V.B = 5 AND V.C = R.C
```

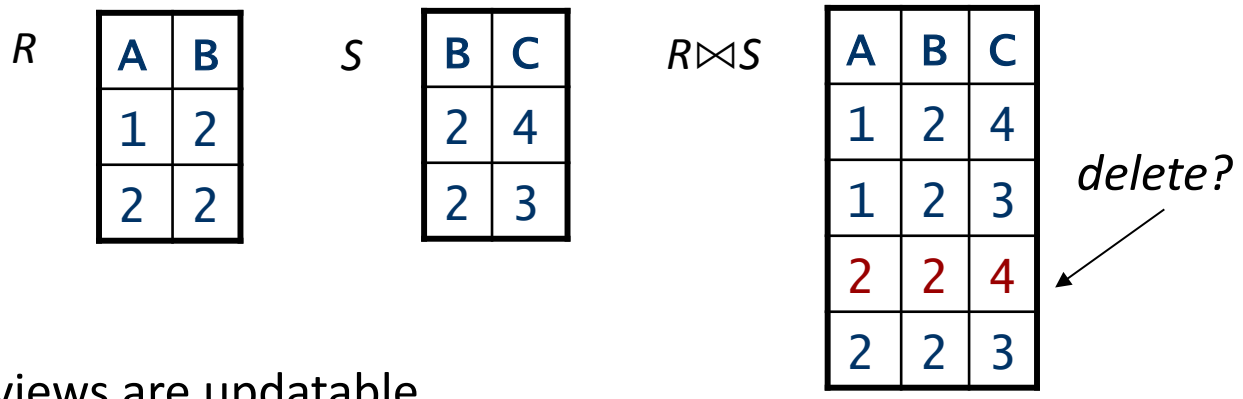
- A materialized view is one that is computed once and its results are stored as a table
  - Think of this as a cached answer
  - These are incredibly useful!
  - Techniques exist for using materialized views to answer other queries
  - Materialized views are the basis of relating tables in different schemas

# Views Should Stay Fresh

- Views should behave, from the user's perspective, exactly like source relations
- For materialized views, there's an association that must be maintained:
  - If tuples change in the base relation, they should change in the view
  - If tuples change in the view, that should reflect in the base relation(s)

# View Maintenance and the View Update Problem

- There exist algorithms to incrementally recompute a materialized view when the base relations change
- We can try to propagate view changes to the base relations
  - However, there are lots of views that aren't easily updatable:




  - We can ensure views are updatable by enforcing certain constraints (e.g., no aggregation), but this limits the kinds of views we can have

# Views as a Bridge between Data Models

Despite XML's apparent flexibility, the following claim holds:

> "XML can't represent anything that can't be expressed in in the relational model"

If this is true, then we must be able to represent XML in relations

> Store a relational view of XML
> (or create an XML view of relations)

# A View as a Translation between XML and Relations

- We'll discuss techniques from the most-cited paper in this area (Shanmugasundaram et al), but there are many more (Fernandez et al., …)
- Technology already transferred into commercial systems
  - XPERANTO at IBM Research was incoporated into DB2 v9
  - Current versions of SQL Server and Oracle have XML support too

# Issues in Mapping Relational ⇔ XML

- We know the following:
  - XML is a tree
  - XML is **semi**-structured
    - There's some structured "stuff"
    - There is some unstructured "stuff"
- Issues relate to describing XML structure, particularly parent/child in a relational encoding
  - Relations are flat
  - Tuples can be "connected" via foreign-key/primary-key links

# The Simplest Way to Encode a Tree: Edge Tables

- Suppose we had:
-

```
<tree id="0">
  <content id="1">
    <sub-
content>XYZ
    </sub-content>
    <i-content>14
    </i-content>
  </content>
</tree>
```

- Where we have no IDs, invent values

Edge

| key | label | type | value | parent |
|---|---|---|---|---|
| 0 | tree | ref | - | - |
| 1 | content | ref | - | 0 |
| 2 | sub-content | ref | - | 1 |
| 3 | i-content | ref | - | 1 |
| 4 | - | str | XYZ | 2 |
| 5 | - | int | 14 | 3 |

*What are the shortcomings of this approach?*

# Florescu/Kossmann Improved Edge Approach

- Consider order, typing; separate the values

Edge

| parent | ord | label | flag | target |
|---|---|---|---|---|
| - | 1 | tree | ref | 0 |
| 0 | 1 | content | ref | 1 |
| 1 | 1 | sub-content | str | v2 |
| 1 | 1 | i-content | int | v3 |

Vint

| vid | value |
|---|---|
| v3 | 14 |

Vstring

| vid | value |
|---|---|
| v2 | XYZ |

# How Do You Reconstruct the XML Using SQL?

- Assume we know the structure of the XML tree (we'll see how to avoid this later)

- We can compute an "XML-like" relation using "outer unions" – a technique pioneered in XPERANTO system @ IBM

  – Idea: if we take two non-union-compatible expressions, pad each with `NULL`s, we can `UNION` them together

  – Let's see how this works…

# A Relation that Mirrors the XML Hierarchy

```
<tree id="0">
    <content id="1">
        <sub-content>XYZ</sub-content>
        <i-content>14</i-content>
    </content>
</tree>
```

- Output relation, encoding this tree, would look like:

| rLabel | rid | rOrd | clabel | cid | cOrd | sLabel | sid | sOrd | str | int |
|---|---|---|---|---|---|---|---|---|---|---|
| tree | 0 | 1 | - | - | - | - | - | - | - | - |
| - | 0 | 1 | content | 1 | 1 | - | - | - | - | - |
| - | 0 | 1 | - | 1 | 1 | sub-content | 2 | 1 | - | - |
| - | 0 | 1 | - | 1 | 1 | - | 2 | 1 | XYZ | - |
| - | 0 | 1 | - | 1 | 2 | i-content | 3 | 1 | - | - |
| - | 0 | 1 | - | 1 | 2 | - | 3 | 1 | - | 14 |

# A Relation that Mirrors the XML Hierarchy

```
<tree id="0">
    <content id="1">
        <sub-content>XYZ</sub-content>
        <i-content>14</i-content>
    </content>
</tree>
```

- Output relation, encoding this tree, would look like:

| rLabel | rid | rOrd | cLabel | cid | cOrd | sLabel | sid | sOrd | str | int |
|---|---|---|---|---|---|---|---|---|---|---|
| tree | 0 | 1 | - | - | - | - | - | - | - | - |
| - | 0 | 1 | content | 1 | 1 | - | - | - | - | - |
| - | 0 | 1 | - | 1 | 1 | sub-content | 2 | 1 | - | - |
| - | 0 | 1 | - | 1 | 1 | - | 2 | 1 | XYZ | - |
| - | 0 | 1 | - | 1 | 2 | i-content | 3 | 1 | - | - |
| - | 0 | 1 | - | 1 | 2 | - | 3 | 1 | - | 14 |

# A Relation that Mirrors the XML Hierarchy

```
<tree id="0">
    <content id="1">
        <sub-content>XYZ</sub-content>
        <i-content>14</i-content>
    </content>
</tree>
```

- *Colors are representative of separate SQL queries…*

| rLabel | rid | rOrd | cLabel | cid | cOrd | sLabel | sid | sOrd | str | int |
|---|---|---|---|---|---|---|---|---|---|---|
| tree | 0 | 1 | - | - | - | - | - | - | - | - |
| - | 0 | 1 | content | 1 | 1 | - | - | - | - | - |
| - | 0 | 1 | - | 1 | 1 | sub-content | 2 | 1 | - | - |
| - | 0 | 1 | - | 1 | 1 | - | 2 | 1 | XYZ | - |
| - | 0 | 1 | - | 1 | 2 | i-content | 3 | 1 | - | - |
| - | 0 | 1 | - | 1 | 2 | - | 3 | 1 | - | 14 |

# SQL for Computing The Relation

Edge

| parent | ord | label | flag | target |
|---|---|---|---|---|
| - | 1 | tree | ref | 0 |
| 0 | 1 | content | ref | 1 |
| 1 | 1 | sub-content | str | v2 |
| 1 | 1 | i-content | int | v3 |

Vint

| vid | value |
|---|---|
| v3 | 14 |

Vstring

| vid | value |
|---|---|
| v2 | XYZ |

- For each sub-portion we preserve the keys (target, ord) of the ancestors
- Root:

```
select E.label AS rLabel, E.target AS rid, E.ord AS rOrd,
null AS cLabel, null AS cid, null AS cOrd, null AS subOrd,
null AS sid, null AS str, null AS int
from Edge E
where parent IS NULL
```

| rLabel | rid | rOrd | clabel | cid | cOrd | sLabel | sid | sOrd | str | int |
|---|---|---|---|---|---|---|---|---|---|---|
| tree | 0 | 1 | - | - | - | - | - | - | - | - |

# SQL for Computing This Relation

- For each sub-portion we preserve the keys (target, ord) of the ancestors
- Root:

```
select E.label AS rLabel, E.target AS rid, E.ord AS rOrd,
null AS cLabel, null AS cid, null AS cOrd, null AS subOrd,
null AS sid, null AS str, null AS int
from Edge E
where parent IS NULL
```

- First-level children:

```
select null AS rLabel, E.target AS rid, E.ord AS rOrd,
E1.label AS cLabel, E1.target AS cid, E1.ord AS cOrd, null
AS …
from Edge E, Edge E1
where E.parent IS NULL AND E.target = E1.parent
```

# The Rest of the Queries

- Grandchild:

```
select null as rLabel, E.target AS rid, E.ord AS rOrd, null
AS cLabel, E1.target AS cid, E1.ord AS cOrd, E2.label as
sLabel, E2.target as sid, E2.ord AS sOrd, null as …
from Edge E, Edge E1, Edge E2
where E.parent IS NULL AND E.target = E1.parent AND
E1.target = E2.parent
```

- Strings:

```
select null as rLabel, E.target AS rid, E.ord AS rOrd, null
AS cLabel, E1.target AS cid, E1.ord AS cOrd, null as
sLabel, E2.target as sid, E2.ord AS sOrd, Vi.val AS str,
null as int
from Edge E, Edge E1, Edge E2, Vint Vi
where E.parent IS NULL AND E.target = E1.parent AND
E1.target = E2.parent AND Vi.vid = E2.target
```

- Integers: similar to above

# Finally…

- Union them all together:

```
( select E.label as rLabel, E.target AS rid, E.ord AS rOrd,
…
  from Edge E
  where parent IS NULL)
UNION (
  select null as rLabel, E.target AS rid, E.ord AS rOrd,
E1.label AS cLabel,
  E1.target AS cid, E1.ord AS cOrd, null as …
  from Edge E, Edge E1
  where E.parent IS NULL AND E.target = E1.parent
) UNION (

  .

  :
) UNION (
  .
  :

)
```

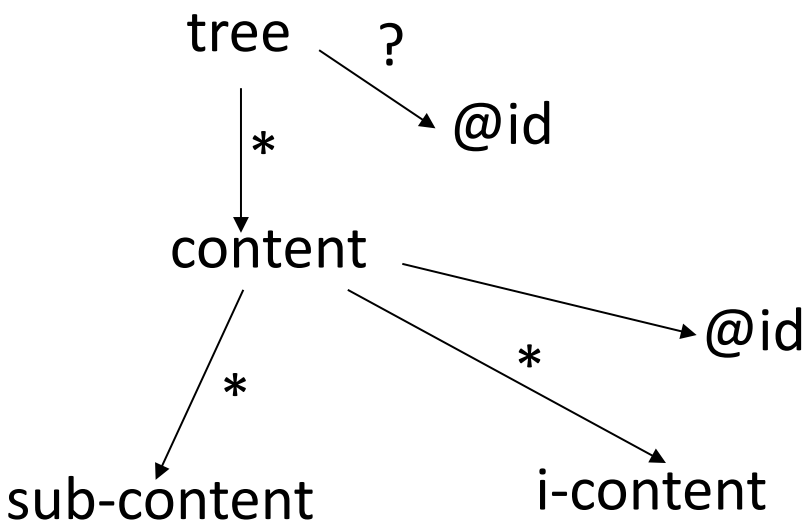
- Then another module will add the XML tags, and we're done!

# "Inlining" Techniques

- Folks at Wisconsin noted we can exploit the "structured" aspects of semi-structured XML
  - If we're given a DTD, often the DTD has a lot of required (and often singleton) child elements
    - Book(title, author*, publisher)
  - Recall how normalization works in a traditional DBMS:
    - Decompose until we have everything in a relation determined by the keys
    - … But don't decompose any further than that
  - Shanmugasundaram et al. try not to decompose XML beyond the point of singleton children

# Inlining Techniques

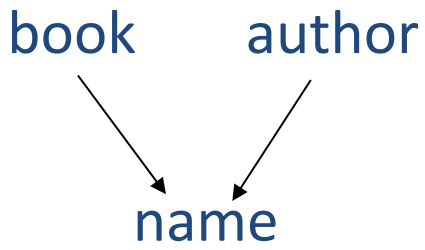
- Start with DTD, build a graph representing structure




- The edges are annotated with ? ("optional"), * ("zero or more")

# Building Schemas

- Now, they tried several alternatives that differ in how they handle *elements w/multiple ancestors*
  - Can create a separate relation for each path
  - Can create a single relation for each element
  - Can try to inline these




- For tree examples, these are basically the same
  - Combine non-set-valued things with parent
  - Add separate relation for set-valued child elements
  - Create new keys as needed

# Schema for Our Example

```
TheRoot(rootID)
Content(parentID, id, @id)
Sub-content(parentID, varchar)
I-content(parentID, int)
```

- *If we suddenly changed DTD to <!ELEMENT content(sub-content*, i-content?) what would happen?*

# XQuery to SQL

- Inlining method needs external knowledge about the schema
  - Needs to supply the tags and info not stored in the tables
- We can actually directly translate simple XQuery into SQL over the relations – not simply reconstruct the XML

# An Example

```
for $X in document("mydoc")/tree/content
where $X/sub-content = "XYZ"
return $X
```

- The steps of the path expression are generally joins
  - … Except that some joins are eliminated by the fact we've inlined subelements
- Let's try it over the schema:

```
TheRoot(rootID)
Content(parentID, id, @id)
Sub-content(parentID, varchar)
I-content(parentID, int)
```

# Summary: XML Views of Relations

- We've seen that views are useful things
- Allow us to store and refer to the results of a query
- We've seen an example of a view that changes from XML to relations – and we've even seen how such a view can be posed in XQuery and "unfolded" into SQL
  - Current versions of the major DBMSs support XML and (fragments of) XQuery this way