XML Publishing

CPS 296.1
Topics in Database Systems

XML publishing

• Legacy data: normalized and stored in many flat relations; managed by relational DBMS
• XML data: un-normalized, nested; wanted by next-generation applications

<table>
<thead>
<tr>
<th>Department</th>
<th>Project</th>
<th>Employee</th>
</tr>
</thead>
<tbody>
<tr>
<td>DeptId</td>
<td>ProjId</td>
<td>DeptName</td>
</tr>
<tr>
<td>10</td>
<td>888</td>
<td>Purchasing</td>
</tr>
<tr>
<td>705</td>
<td>10</td>
<td>Recycling</td>
</tr>
</tbody>
</table>

• XML data: un-normalized, nested; wanted by next-generation applications

Roadmap

• IBM’s XPERANTO project
• AT&T’s SilkRoute project

Issues in publishing XML

• Structuring and tagging
  – Flat relations ➔ nested XML elements
  – Relational schema ➔ XML tags
• Processing inside versus outside relational engine
  ➢ Space of alternatives (Shanmugasundaram et al.)

Issues in publishing XML

<table>
<thead>
<tr>
<th>Space of alternatives (Shanmugasundaram et al.)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Early Structuring</td>
</tr>
<tr>
<td>Inside Engine</td>
</tr>
<tr>
<td>Outside Engine</td>
</tr>
</tbody>
</table>

Stored procedure approach

• Early tagging, early structuring; outside engine
• Application and/or stored procedure explicitly issue queries for each element (and recursively for sub-elements)

<table>
<thead>
<tr>
<th>DBMS Engine</th>
</tr>
</thead>
<tbody>
<tr>
<td>Department</td>
</tr>
<tr>
<td>Employee</td>
</tr>
<tr>
<td>Project</td>
</tr>
<tr>
<td>((10, Purchasing), (Internet), (John), (Recycling), (Mary))</td>
</tr>
</tbody>
</table>

• Problem: many small queries

Correlated CLOB approach (slide 1)

• Early tagging, early structuring; inside engine
• Use user-defined functions for tagging
  – “XML” is a special type like CLOB

```
define XML Constructor DEPT(dname VARCHAR(20), emplist.XML, projlist.XML) as
<department name={dname}>
<emplist>{emplist}</emplist>
<projlist>{projlist}</projlist>
</department>
```
Correlated CLOB approach (slide 2)

- Use user-defined aggregate XMLAGG to concatenate XML fragments into one
- Use correlated subqueries for structuring

```
select DEPT(d.name,
    (select XMLAGG(EMP(e.name))
    from Employee e where e.deptno = d.deptno),
    (select XMLAGG(PROJ(p.name))
    from Project p where p.deptno = d.deptno)
) from Department d
```

- Problems
  - Correlated execution of sub-queries → nested loop
  - Repeated creation and copying of big CLOB’s

De-correlated CLOB approach

- Early tagging, early structuring: inside engine
- Relational engine may be smart enough to convert correlated subqueries into joins

Example
  - Compute employee lists associated with all departments
  - Compute project lists associated with all departments
  - Join results on department id

- Still a problem: creation and copying of CLOB’s

Late tagging, late structuring

- Contents for XML produced without structure (in arbitrary order)
- Tagger enforces order and inserts tags

```
Result XML Document
Tagger
Unstructured content
Relational Query Processor
```

Redundant relation approach

- Late tagging, late structuring; either inside or outside engine
- Join all source relations together
  - Use outerjoin to preserve childless parents

```
(10, Purchasing)  (10, John)  (10, Internet)
(10, Purchasing)  (10, Mary)  (10, Recycling)
(20, Accounting)  (10, Internet) (10, Recycling)
(10, Purchasing)
```

- Problem: large relational output because of redundancy

Unsorted outer union approach

- Late tagging, late structuring; either inside or outside engine
- Do not join siblings; outerjoin them separately with parent and then outer union them together

```
(Purchasing, NULL, Internet)
(Purchasing, NULL, Recycling)
(Purchasing, John, NULL)
(Purchasing, Mary, NULL)
```

- Problem: wide tuples with many NULL columns

Hash-based tagger

- For late tagging, late structuring
- Use a hash table to group tuples returned by the relational query
  - Example
    - Returned tuple: (Purchasing, NULL, Internet, 0)
    - Hash on “Purchasing” to locate the parent element
    - If not found, create a parent element
    - Group the tuple under the parent
  - After all tuples are hashed, go through the hash table to generate XML
- Problem: require memory to be large enough to hold the entire output document
Late tagging, early structuring

- Contents for XML produced with structure (in sorted order; already grouped)
- Tagger just inserts tags (constant space)

```
+-------------------+
| Tagger             |
| Structured content |
+-------------------+
```

Sorted outer union approach

- Late tagging, early structuring; tagging either inside or outside engine
- Same outerjoin/outer union query as the unsorted outer union approach, with the additional ORDER BY clause to sort/group output tuples
  - Example: ORDER BY Aid, Bid, Cid, Did, Eid

```
<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
<th>E</th>
</tr>
</thead>
<tbody>
<tr>
<td>A1</td>
<td>B1</td>
<td>NULL</td>
<td>D1</td>
<td>NULL</td>
</tr>
<tr>
<td>A1</td>
<td>B1</td>
<td>NULL</td>
<td>D2</td>
<td>NULL</td>
</tr>
<tr>
<td>A1</td>
<td>B2</td>
<td>NULL</td>
<td>D3</td>
<td>NULL</td>
</tr>
<tr>
<td>A1</td>
<td>B2</td>
<td>NULL</td>
<td>D4</td>
<td>NULL</td>
</tr>
<tr>
<td>A1</td>
<td>NULL</td>
<td>C1</td>
<td>NULL</td>
<td>E1</td>
</tr>
<tr>
<td>A1</td>
<td>NULL</td>
<td>C1</td>
<td>NULL</td>
<td>E2</td>
</tr>
</tbody>
</table>
```

Conclusion from experiments

- Inside engine is much faster than outside engine
  - Cost of binding out tuples to host variables is high
- Given sufficient main memory, unsorted outer union with hash-based tagger is the best
- With limited memory, sorted outer union with constant-space tagger is the best

Sorted outer union revisited

- One large SQL query, consisting of left outerjoins combined using outer unions followed by sorting
  - Example (in SilkRoute’s RXL syntax)

```
from Supplier $s
  construct <supplier>
    <name>$s.name</name>
  { from Nation $n
    where $s.nationkey = $n.nationkey
    construct <nation>$n.name</nation> }
  { from PartSupp $ps, Part $p
    where $ps.suppkey = $ps.suppkey and $ps.partkey = $p.partkey
    construct <part><name>$p.name</name></part> }
</supplier>
```

Other alternatives

- Multiple SQL queries, each sorted for efficient merging
- Middleware merge-outerjoins sorted results; still constant space requirement
  - Example: fully partitioned
    - SORT(Supplier)
    - SORT(Supplier JOIN Nation)
  - Example: partially partitioned
    - SORT(Supplier LEFT OUTERJOIN Nation)
    - SORT(Supplier LEFT OUTERJOIN Part Supp, Part)
- Questions
  - How to enumerate alternatives?
  - How to pick the best alternative?

View tree

- Captures nesting of output XML elements
Plan enumeration using view tree

- Each alternative plan corresponds to a partitioning of the view tree into smaller trees
  - Each edge can be either broken or unbroken in a partitioning
  - Number of alternative plans = \(2^E\) where \(E\) is the number of edges in the view tree

View tree reduction

- Edges are labeled by 1, ?, or *
  - 1: Parent has one and only one such sub-element (e.g., <supplier> – <nation>)
  - ?: Parent has zero or one such sub-element
  - *: Parent has zero or more such sub-elements
  - Requires the knowledge of database constraints (e.g., Supplier.nationkey references Nation.nationkey)
  - Heuristic: after partitioning, nodes connected by 1 edges can be merged together (e.g., <supplier> and <nation>)
    - Use JOIN instead of OUTERJOIN

Cost formula

- \(\text{cost}(q) = a \times \text{evaluation\_cost}(q) + b \times \text{data\_size}(q)\)
- \(\text{data\_size}(q) = |\text{attrs}(q)| \times \text{cardinality}(q)\)
  - Assume estimates of evaluation\_cost and cardinality can be obtained from the relational DBMS optimizer
  - \(a\) and \(b\) are parameters that depend on the middleware/database environment
    - May require some benchmarking/tuning to find out

Experiments with the plan space

- Alternatives do matter
- View tree reduction heuristics do help
- Sorted outer union (i.e., no partitioning) is significantly slower than the optimal plan (with partitioning)
  - Need a cost-based optimizer to pick the best partitioning

Greedy algorithm for plan generation

- Threshold parameters \(t_1\) and \(t_2\)
  - But who determines these parameters?
- For each edge, compute the cost of merging two partitions connected by this edge: 
  \(\text{cost}(q_{\text{combined}}) - (\text{cost}(q_1) + \text{cost}(q_2))\)
  - \(< 0\) means beneficial
  - If \(< t_1\) (e.g., \(-60000\)), mark the edge as mandatory
  - If \(< t_2\) (e.g., \(600\)), mark the edge as optional
  - Size of search space is reduced from \(2^E\) to \(2^E\) (of optional edges)
- Experiments show that the greedy algorithm is able to pick near-optimal plans

More pointers

- Fernandez et al. “SilkRoute: Trading Between Relations and XML.” WWW, 1999