COMP 9314

XML – Part 2

Agenda

 getLast week, we discuss mainly XML basics & XML storage
  This week:
    - Some basic XML indexing
    - XPath containment
    - From distributed XML data to XML data synchronization

Path indexing

- Traversing graph almost = query processing for semistructured / XML databases (the whole point)
- Normally, it requires to traverse the data from the root and return all nodes X reachable by a path matching the given regular path expression
- Motivation: allows the system to answer regular path expressions without traversing the whole graph

Major Criteria for indexing

- Speed up the search (by cutting the search space down)
- Relatively smaller size than the original data graph
- Easy to maintain (during data loading during updates)

Two more techniques

- Based on the idea of language-equivalence
- Idea like Data Guide in Lore

An Example of DAG Data

This week:

- Support
- Staff
- Person

Motivation: allows the system to answer regular path expressions without traversing the whole graph
Index graph based on language-equivalence

- A reduced graph that summarizes all paths from the root in the data graph
- The paths from root to o12
  - staff
  - dept/member
  - support/member

Language-equivalent nodes

- Let \( L(x) := \{ w \mid \exists \text{ a path from the root to } x \text{ labeled } w \} \)
- The set \( L(x) \) may be infinite when there are cycles
- Nodes \( x, y \) are language-equivalent \( (x \equiv y) \) if \( L(x) = L(y) \)
- We construct index I by taking the nodes to be the equivalent classes for \( \equiv \)

Language-equivalent

- The paths from root to o3
  - staff
  - dept/member
- Paths to o4 happen to be exactly the same 2 sequences
- Same for o8 and o12
- \( o3 \equiv o4 \equiv o8 \equiv o12 \)

Equivalence classes

\[ o3 \equiv o4 \equiv o8 \equiv o12 \]
\[ o1 \equiv o2 \equiv o7 \]
\[ o12 \equiv o13 \]
\[ o5 \equiv o6 \equiv o9 \]
\[ o10 \equiv o11 \]

The index graph

Query processing based on the index graph

```
dept/member(name, phone)
-> dept/member/name  UNION  dept/member(phone)
-> {o5, o6, o9}  UNION  {o10}
-> {o5, o6, o9, o10}
```
**About this indexing scheme**

- The index graph is never > the data
- In practice, the index graph is small enough to fit in memory
- Construct the index is however a problem
  - check two nodes are language-equivalent is very expensive (are PSPACE)
  - approximation based on bisimulation exists

---

**About Data Guide**

- unique labels at each node
- (hence) extents are no longer disjoint
- query processing proceeds as before
- size of the index may ≥ data size
- not desirable when the data is irregular & has many cycles

---

**Intro to distributed query evaluation**

- Web data is inherently distributed
- Reuse some techniques from distributed RDBMS if some schema info is known
- **New techniques required if no schema info is known**
- In XML, these links are denoted in XLinks and XPointers.

---

**Example query without schema knowledge**

- Assume data are distributed in 3 sites
- Assume the RPE: a.b*.c
- The query starts from site 1

---

**The database**
**Naïve approach**

- A naïve approach takes too many communication steps
- We have to do more work locally
- A better approach needs to:
  1. identify all external references
  2. identify targets of external references

**Query Processing**

- Given a query, we compute its automaton
- Send it to each site
- Start an identical process at each site
- Compute two sets Stop(n, s) and Result(n, s)
- Transmits the relations to a central location and get their union

**Input and output nodes**

- Site 1
  - Inputs: x1 (root), x4
  - Outputs: y1, y3
- Site 2
  - Inputs: y1, y3
  - Outputs: z2
- Site 3
  - Inputs: z2
  - Outputs: x4

**Stop and Result at site 2**

<table>
<thead>
<tr>
<th>Start</th>
<th>Stop</th>
</tr>
</thead>
<tbody>
<tr>
<td>(y1, s2)</td>
<td>(z2, s2)</td>
</tr>
<tr>
<td>(y3, s2)</td>
<td>(z2, s2)</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Start</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>(y1, s2)</td>
<td>y3</td>
</tr>
<tr>
<td>(y1, s3)</td>
<td>y1</td>
</tr>
<tr>
<td>(y3, s3)</td>
<td>y3</td>
</tr>
</tbody>
</table>

**Union of the relations**

<table>
<thead>
<tr>
<th>Start</th>
<th>Stop</th>
</tr>
</thead>
<tbody>
<tr>
<td>(x1, s1)</td>
<td>(y1, s2)</td>
</tr>
<tr>
<td>(x4, s2)</td>
<td>(y3, s3)</td>
</tr>
<tr>
<td>(y1, s2)</td>
<td>(z2, s2)</td>
</tr>
<tr>
<td>(y3, s2)</td>
<td>(z2, s2)</td>
</tr>
<tr>
<td>(z2, s2)</td>
<td>(z4, s2)</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Start</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>(x1, s3)</td>
<td>x1</td>
</tr>
<tr>
<td>(x4, s2)</td>
<td>x3</td>
</tr>
<tr>
<td>(x4, s3)</td>
<td>x4</td>
</tr>
<tr>
<td>(y1, s2)</td>
<td>y3</td>
</tr>
<tr>
<td>(y1, s3)</td>
<td>y1</td>
</tr>
<tr>
<td>(y3, s3)</td>
<td>y3</td>
</tr>
<tr>
<td>(z2, s1)</td>
<td>x3</td>
</tr>
<tr>
<td>(z2, s2)</td>
<td>z2</td>
</tr>
<tr>
<td>(z2, s3)</td>
<td>z2</td>
</tr>
</tbody>
</table>

The result of the query is \( \{y3, z2, x3\} \)

**XPath Containment**

*Credit: slides in this topic authored by Dan Suciu*
**Assumptions**

Additional things in XPath, which we ignore:
- 13 axes:
  - child (/), descendant (/), parent (.), etc
- Order:
  - second child, following sibling, etc
- Complex predicates:
  - @age>25 AND @age<35
- Functions
- Boolean operations
  - AND, OR, NOT

**Remark 1: Branches May Overlap**

**Remark 2: Query Types**

Query written by human:

Query generated automatically:

**Equivalence, Containment**

- E = E' if they return the same result
- E ⊆ E' if the result returned by E is a subset of that returned by E'
**Prior Work**

- Define XPath*\// to be:
  \[ E ::= \text{nodeType} \mid * \mid E/E \mid E//E \mid E[E] \]
- Similarly, XPath* and XPath\//=

<table>
<thead>
<tr>
<th>XPath*</th>
<th>PTIME</th>
<th>[Yannakakis'81]</th>
</tr>
</thead>
<tbody>
<tr>
<td>XPath//</td>
<td>PTIME</td>
<td>[Amer-Yahya et al'01]</td>
</tr>
<tr>
<td>XPath**//</td>
<td>Decidable</td>
<td>[Wood'00]</td>
</tr>
</tbody>
</table>

**Examples of Containment**

- Example of containment:

  \[
  \begin{array}{c}
  \text{person} \\
  \text{name}
  \end{array} \subseteq \begin{array}{c}
  \text{person} \\
  \text{name}
  \end{array}
  \]

**Examples of Containment**

- A homomorphism from E' to E is always sufficient
- For XPath* and XPath** it is also necessary

**Containment for XPath*\//=**

- Interaction between * and // turns out to be hard
- Study linear XPath*\//= first
- Then full XPath*\//=

**Linear XPath*\//=**

- Define a block in E'=
  - Starts with a symbol (not *)
  - Ends with a symbol (not *)
  - Does not have any //
- Define a rubber band in E' =
  - Has only * nodes, at least one // edge

**Practical Algorithm for Linear XPath*\//=**
**Practical Algorithm for Linear XPath**

**Fact** $E \subseteq E'$ iff there exists a homomorphism from the blocks/rubber-bands of $E'$ to $E$

**Algorithm** Match greedily blocks in $E'$ to $E$, skipping nodes for rubber bands

Worst case: $O(mn)$

[Milo&Suciu’99]

---

**Example 1**

```
&person/*/name <= &person/*/name
```

---

**Example 2**

---

**Example 3**

---

**Branching XPath**

- Single homomorphism doesn’t suffices

```
&person/*/name <= &person/*/name
```

---

**Practical Algorithms for Branching XPath**

- Will be EXPTIME in general
- Should run in PTIME for:
  - Linear XPath+
  - XPath*
Practical Algorithms for Branching XPath*/*/j

- Better: should be parametric PTIME:
  - Linear XPath*/*/j plus small number of branches
  - XPath* plus small number of //’s
  - XPath// plus small number of *’s

- Reason: users may use branches, //’s, *’s occasionally

Practical Algorithms for Branching XPath*/*/j

Running time is exponential in general, but...

Let \( m \) = number of //’s in \( E \)
Let \( n \) = number of *’s in \( E’ \)

Theorem The algorithm runs in time:
\[ O(|E| \times |E'| \times n^m) \]

Parametric PTIME: XPath* + some //’s
PTIME: XPath//

From distributed XML to mobile XML

- Other than query processing
- Mobile users care about Data Sync

A Mobile DB System

Traditional Approach

Assumptions

- Data stored in XML documents/databases
- Data retrieved using XQL or XPath
- Data updated using some extended update operations, e.g.,
  - author/namelupdate("smith")
  - book/remarkestdelte()
  - /db/book!moveAfter(/db/magazine)
**Update Scenario 1**

c/d update(“23”)

1 2 3

\[/ab \quad /ab/c \quad /c\]

**Update Scenario 2**

d/e/f update(“10”)

1 2 3

\[/ab \quad /ab/c \quad /c\]

**Update Scenario 3a**

d/e/f update(“15”)

1 2 3

\[/ab \quad /ab/c \quad /c\]

**Update Scenario 3b**

d/e/f update(“15”)

1 2 3

\[/ab \quad /ab/c \quad /c\]

---

**Example snapshot**

**Containment Index**

<table>
<thead>
<tr>
<th>cid</th>
<th>path</th>
<th>len</th>
<th>ptr</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>/</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>12</td>
<td>/a</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>15</td>
<td>/a</td>
<td>6</td>
<td></td>
</tr>
<tr>
<td>16</td>
<td>/ab</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>/a</td>
<td>4</td>
<td></td>
</tr>
<tr>
<td>9</td>
<td>/ab</td>
<td>3</td>
<td></td>
</tr>
<tr>
<td>13</td>
<td>/ab/c</td>
<td>4</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>/ab/b</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>/ab/d</td>
<td>6</td>
<td></td>
</tr>
<tr>
<td>-1</td>
<td>/c</td>
<td>2</td>
<td></td>
</tr>
</tbody>
</table>
Outline of the Algorithms

- **Insertion**
  - Sort the PEs according to
    - #tokens, //, /*, literal
  - For each PE:
    - Search for a match from the root node
    - Insert the CID into the appropriate place
    - Special attention to // & * (details in the paper)

Synchronization Algorithm

- Assume CID = c and its local db = path1 and its update = path2/upd(x), do the following using the constructed containment index:
  - Notify CIDs in all c's ancestors
  - Perform either:
    - (A) Notify CIDs in all c's descendants
    - (B) Perform the containment test on each descendant node against path1/path2. If contained, notify all CIDs of that node.
  - (A) or (B) or their mixture depending on the ratio bet'n server-load and network-load

Update merging

- Idea: merging several update statements to minimize the mobile device's connection time
- Solution: conflict detection & resolution
- Conflict detection: similar to the previous containment index except now dealing with operations instead of cids

Conflicts

- Direct conflict (DC)
  - e.g., a/insert(d); a/delete( )
- Syntax conflict (SC)

<table>
<thead>
<tr>
<th>op</th>
<th>opAfter</th>
<th>opBefore</th>
</tr>
</thead>
<tbody>
<tr>
<td>DC</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>DC</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>SC</td>
<td>No</td>
<td>Yes</td>
</tr>
</tbody>
</table>

Operation Index

<table>
<thead>
<tr>
<th>opid</th>
<th>cid</th>
<th>path</th>
<th>op</th>
<th>len</th>
<th>ptr</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>11</td>
<td>/</td>
<td>ins</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>12</td>
<td>/a</td>
<td>del</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>13</td>
<td>/b</td>
<td>insb</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>14</td>
<td>/b</td>
<td>insb</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>15</td>
<td>/ac</td>
<td>ins</td>
<td>4</td>
<td></td>
</tr>
</tbody>
</table>

Performance
Results

- Update notification based on containment relationship
- “False negative” is acceptable when the server-load is more critical than the network-load
- Efficient index & its algorithm to support the containment checking
- Mechanism for merging updates to further enhance the performance