Foundations of XML Data Manipulation

Giorgio Ghelli

5. Storage and Manipulation of SSD

Shamelessly “inspired” by Ioana Manolescu tutorial
The problem

- Consider the queries
  - $doc // e-mail
  - $doc ../../name = 'ghelli']/e-mail
- We do not want to bring the whole $doc in main memory
- Set manipulation rather than tuple manipulation

Classifying stores

- Essential criteria:
  - Clustering
  - Encoding of parent/child relationship
OEM data model

Storing OEM

- No schema!
- Storing objects in LORE:
  - Store the graph, clustered in depth-first order
  - Operator: Scan(document,path), returns a set of objects
Indexing in LORE

- \( \text{VIndex}(l, o, \text{pred}) \): all objects \( o \) with an incoming \( l \)-edge, satisfying the predicate
- \( \text{LIndex}(o, l, p) \): all parents of \( o \) via an \( l \)-edge
- \( \text{BIndex}(x, l, y) \): all edges labeled \( l \)

Access plans

- Top down or bottom up navigation?
  
  select \( x \)
  from A.B x
  where \( x.C = 5 \)
Access plans: bottom up

```
select x
from A.B x
where x.C = 5
```

```
\begin{align*}
\text{Return}(n2) \\
\text{Name}(n3,'A') \\
\text{NLJoin} \\
\text{NLJoin} \\
\text{LIndex}(n2,'B',n3) \\
\text{VIndex('C',n1,'=5')} \\
\text{LIndex}(n1,'C',n2) \\
\end{align*}
```

Bottom up and top down

```
select x
from A.B x
where x.C = 5
```

```
\begin{align*}
\text{Return}(n2) \\
\text{Name}(n3,'A') \\
\text{LIndex}(n2,'B',n3) \\
\text{LIndex}(n1,'C',n2) \\
\text{VIndex('C',n1,'=5')} \\
\text{Return}(n1) \\
\text{Select}(n2,'=5') \\
\text{Scan}(n1,'C',n2) \\
\text{Scan('A','B',n1)} \\
\end{align*}
```
Hybrid access plans

```
select x
from A.B x
where x.C = 5
```

Path indexes

- PIndex(p, o): all objects reachable by the path p
Using a path index

```
select x
from A.B x
where x.C = 5
```

Intersect

PIndex('A.B',n2)
Project(n2)

LIndex(n1,'C',n2)
VIndex('C',n1,'=5')

DataGuides

- Introduced in Lore: a compact representation of all paths in the graph
- A DG for $s$ is an OEM object $d$ such that:
  - Every path of $s$ reaches exactly one object in $d$
  - Every path in $d$ is a path for $s$
- DG: a schema *a posteriori*
- Used to:
  - Inform the user
  - Expand wildcards in paths
  - Inform the optimizer
DataGuides for trees

Building a DataGuide

- Similar to converting a NFA to a DFA
- Linear time for trees
- Exponential in time and space for graphs
- In practice, works well for regular structures
**Which dataguide is better?**

- Minimal dataguide
- Strong dataguide: if p1 and p2 both reach the same node in d, then p1 and p2 have the same target set in s
  - Each target-set in the source has its own node and in the guide
  - Easy to build
  - Easy to maintain, by keeping track of the many-to-many node correspondence between s and d

**Optimization via dataguide**

- Expanding paths: a//z => a/b/z | a/c/d/z
- Deleting paths that are not in the data
- Contracting paths: a/c/e/d/z => //d/z
  - May be more efficient for a bottom-up evaluation
- Keeping statistic information
  - However, statistic are needed for every k-length path, not just for rooted paths
Graph schemas

- Each edge in the scheme is labeled by a label predicate (a set of labels)
- Predicates are deterministic
- Conformance is defined by a simulation between s and d:
  - Root of data in root of schema
  - For every n1 in d1 with n1-l-n2, we have d1-l-d2 with n2 in d2
- No request for surjectivity, or injectivity
Graph indexing

• Group nodes in sets, possibly disjoint
• Store the extent of each set
• Grouping criteria:
  – Reachable by exactly the same Forward paths: 1-index
  – Indistinguishable by any F&B path: FB-index
  – Indistinguishable by the paths in a set Q: covering indexes
  – Indistinguishable by any path longer than k: A(k) index

XML Storage in RDBMS
Using RDBMS for XML

- Advantages
  - Transactions
  - Optimization
- Issues
  - Data storage
  - Query translation

Storing data

- Data may be schema-less
- Data may have a schema
- Data may change its schema over time
Approaches

• Based on schema:
  – Based only on schema
  – Based on schema + cost informations

• Unknown schema:
  – Derive schema from data
  – Generic approach

• User defined

Unknown schema: the edge table

<table>
<thead>
<tr>
<th>From Pos</th>
<th>Tag</th>
<th>To Data</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>1</td>
<td>employees 1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>emp 2</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>ID 3</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td>FN 4 Nancy</td>
</tr>
<tr>
<td>2</td>
<td>3</td>
<td>BD 5</td>
</tr>
<tr>
<td>5</td>
<td>1</td>
<td>Day 6 8</td>
</tr>
<tr>
<td>5</td>
<td>2</td>
<td>Month 7 dec</td>
</tr>
<tr>
<td>5</td>
<td>3</td>
<td>Year 8 1968</td>
</tr>
<tr>
<td>2</td>
<td>4</td>
<td>HD 9</td>
</tr>
<tr>
<td>9</td>
<td>…</td>
<td>…</td>
</tr>
<tr>
<td>1</td>
<td>2</td>
<td>emp 11</td>
</tr>
<tr>
<td>11</td>
<td>…</td>
<td>…</td>
</tr>
<tr>
<td>1</td>
<td>3</td>
<td>emp 21</td>
</tr>
</tbody>
</table>
Navigating the edge table

- //FN/text():
  select e.Data from edge e where e.Tag = ‘FN’

- //emp[ID='1']/FN/text()
  select e3.Data
  from edge e1, edge e2, edge e3
  where e1.Tag = ‘emp’ and e1.to = e2.from
  and e2.Tag = ‘ID’ and e2.Data = 1
  and e1.to = e3.from and e3.Tag = ‘FN’

- Navigation through multi-way join (XPath to FO translation)

Partitioned edge table

<table>
<thead>
<tr>
<th>From Pos</th>
<th>Tag</th>
<th>To</th>
<th>Data</th>
</tr>
</thead>
<tbody>
<tr>
<td>-</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>ID</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>FN</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>BD</td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>Day</td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>Month</td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>Year</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>HD</td>
<td></td>
<td></td>
</tr>
<tr>
<td>9</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>emp</td>
<td></td>
<td></td>
</tr>
<tr>
<td>11</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>emp</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>From Pos</th>
<th>To</th>
<th>Data</th>
</tr>
</thead>
<tbody>
<tr>
<td>-</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td></td>
<td>2</td>
</tr>
<tr>
<td>3</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td></td>
<td>8</td>
</tr>
<tr>
<td>7</td>
<td></td>
<td>dec</td>
</tr>
<tr>
<td>8</td>
<td></td>
<td>1968</td>
</tr>
<tr>
<td>9</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
<td>11</td>
</tr>
<tr>
<td>11</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td></td>
<td>21</td>
</tr>
</tbody>
</table>

employees

<table>
<thead>
<tr>
<th>From Pos</th>
<th>To</th>
<th>Data</th>
</tr>
</thead>
<tbody>
<tr>
<td>-</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

emp

<table>
<thead>
<tr>
<th>From Pos</th>
<th>To</th>
<th>Data</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2</td>
<td>11</td>
</tr>
<tr>
<td>1</td>
<td>3</td>
<td>21</td>
</tr>
</tbody>
</table>

ID

<table>
<thead>
<tr>
<th>From Pos</th>
<th>To</th>
<th>Data</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>1</td>
<td>3</td>
</tr>
</tbody>
</table>
Navigation

• //emp[ID='1']/FN/text()
  
  select e1.Data
  from edge e1, edge e2, edge e3
  where e1.Tag = ‘emp’ and e1.to = e2.from
  and e2.Tag = ‘ID’ and e2.Data = 1
  and e1.to = e3.from and e3.Tag = ‘FN’

⇒ select FN.Data
  from emp, ID, FN
  where emp.to = ID.from and ID.Data = 1
  and emp.to = FN.from

• Joining smaller tables

Related storage schemes

• The universal relation:
  – employees ◦◦ emp ◦◦ ID ◦◦ FN ◦◦ ...

• Materialized views over edges:
  – emp ◦ ID ◦ FN ◦ HD ...

• The STORED approach:
  – Materialized views based on pattern frequencies in the database
  – Overflow tables for the rest
Flat storage

<table>
<thead>
<tr>
<th>ID</th>
<th>First Name</th>
<th>BD-D</th>
<th>BD-M</th>
<th>BD-Y</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Nancy</td>
<td>8</td>
<td>dec</td>
<td>1968</td>
</tr>
<tr>
<td>2</td>
<td>Andrew</td>
<td>19</td>
<td>feb</td>
<td>1952</td>
</tr>
<tr>
<td>3</td>
<td>Janet</td>
<td>30</td>
<td>aug</td>
<td>1963</td>
</tr>
<tr>
<td>4</td>
<td>Margaret</td>
<td>19</td>
<td>sep</td>
<td>1958</td>
</tr>
</tbody>
</table>

Path partitioning in Monet

- For each root-to-inner-node path:
  - Path(n1,n2,ord):
    - employees.emp{(1,2,1);(1,11,2);(1,21,3)}
    - employees.emp.ID{(2,3,1);…}

- For each root-to-leaf path:
  - Path(n1,val)
    - employees.emp.ID.text{(3,'1');…}

- Path summary
- No join for linear path expressions
### XRel approach: interval coding

- **Tables:**
  - Path(PathID, PathExpr)
  - Element(DocID, PathID, Start, End, Ordinal)
  - Text(DocID, PathID, Start, End, Value)
  - Attribute(DocID, PathID, Start, End, Value)
- **Ancestor relation:**
  - \( N1.\text{start} < N2.\text{start} \) and \( N2.\text{end} > N1.\text{end} \)
- **Path expression:** regexp matching with Path table, join the result with the data tables

### XParent

- **Tables:**
  - LabelPath(ID, length, pathExpr)
  - Data(pathID, nID, ord, value)
  - Element(pathID, ord, nID)
  - ParentChild(plID, cID)
- **Use ParentChild instead of interval coding:** equi-join instead of \(<\)-join
Schema-driven storage

Employee(PId, Id, Name, Name.GN, Name.FN, Phone)
E-Mail(PId, Id, E-mail)
Depts(PId, Id, Address, Addr.Street, Addr.Number)

Sharing the address

Employee(PId, Id, Name, Name.GN, Name.FN)
Dep(PId, Id)
Address(PId, Id, Address, Addr.Street, Addr.Number)
Cost based approach

- Valuate a query load against one possible representation of the DTD
- Schema transformations:
  - type A=[b [Integer], C, d*], type C=e [String]
  - equivalent to type A=[b [Integer], e [String], d*]
  - a[t1|t2] equivalent to [t1] | a[t2]

User defined mapping

- Express (relational) storage by custom expressions over the XML document
  - Relation = materialized view over the XML document
- Rewrite XQuery to SQL
- R(y,z) :- Auctions.item x, x.@id.text() y, x.price.text() z
- S(u,v) :- Auctions.item t, t.@id.text() u, t.description.text() v
- for $x$ in //item
  - return <res> {$x/price}, {$x/description} </res>
  - select z, v from R, S where R.y=S.u
  - Reasoning about: XPath containment, functional dependencies, cardinality constraints
Navigation

• Simple paths:
  – Edge storage and join
  – Path partitioning and union

• Recursive paths:
  – Expansion to simple paths
  – Recursive navigation
  – Structural joins!

Structural Joins

```c
a=AList->firstNode; d=DList->firstNode; OutputList=NULL;
while((the input lists are not empty or the stack is not empty){
  if (a.StartPos > stack->top.EndPos
      && d.StartPos > stack->top.EndPos ) {
    stack->pop(); }
  else if (a.StartPos < d.StartPos) {
    stack->push(a)
    a = a->nextNode }
  else {
    for (a1=stack->bottom; a1 != NULL; a1 = a1->up) {
      append (a1,d) to OutputList; }
    d = d->nextNode; }
}
```
Stack-tree-desc

Holistic Path Joins

- PathStack: All the joins of a path with one scan
- Compact encoding of solutions
Algorithm PathStack

while \( \neg\text{end}(q) \) {
    \( q_{\text{min}} = \text{getMinSource}(q); \)
    for \( q_i \) in subtreeNodes(q)
        while (\( \neg\text{empty}(S[q]) \) and
            \( \text{topR}(S[q]) < \text{nextL}(T[q_{\text{min}}]) \)) pop(S[q]);
    push( S[q_{\text{min}}], (\text{next}(T[q_{\text{min}}]), \text{top}(S[\text{parent}(q_{\text{min}})]));
    advance(T[q_{\text{min}}]);
    if (isLeaf(q_{\text{min}})) {
        \text{showSolutions}(S[q_{\text{min}}]);
        pop(S[q_{\text{min}}]);
    }
}

Holistic Twig Joins

• Consider:
  – for \( x \) in //b, \( y \) in \( x//e, z \) in \( x//d... \)
• Avoid constructing \( (x, y) \) pairs for \( x \) which have e descendant but no d descendant
Holistic Twig Joins

while ¬end(q) {
    qₜₐₓ = getNext(q);
    if ¬isRoot(qₜₐₓ) { cleanStack(parent(qₜₐₓ), nextL(qₜₐₓ)); }
    if isRoot(qₜₐₓ) or notEmpty(S[parent(qₜₐₓ)]) {
        cleanStack(qₜₐₓ, nextL(qₜₐₓ));
        push{ S[qₜₐₓ], next(T[qₜₐₓ]), top(S[parent(qₜₐₓ)]) );
        advance(T[qₜₐₓ]);
        if isLeaf(qₜₐₓ) {
            showSolutions(S[qₜₐₓ]);
            pop(S[qₜₐₓ]); } } } } } } else {advance(T[qₜₐₓ]); } } } } } { getNext(q): next stream, skipping elements that do not participate in any solution

Skipping

• Linear time is not optimal:

a ——— ——— ———

\|\n
d ——— ——— ——— ——— ——— ——— ———

• Needs the ability to search d on the basis of start: d is an XBTree
Summary

- Linear complexity join algorithms based on region identifiers
- Sub-linear variants exist, based on skipping
- Holistic twig joins reduce intermediary results
- All the factors to keep into account:
  - Data access cost
  - Join cost
  - Sort cost

Some references

- Graph schemas [Fernandez Suciu 98]
- LORE
- The STORED approach [DeutchFernandezSuciu99]
- Schema-driven storage [shanmugasundaram-etal-vldb-99]
- XRel
- XParent
- Path partitioning in Monet [ScKeWiWa WEBDB 00]
- Holistic Path Joins [bks2002-twigjoin]