Foundations of XML Data Manipulation

Giorgio Ghelli

5. Storage and Manipulation of SSD

Shamelessly “inspired” by

Ioana Manolescu tutorial

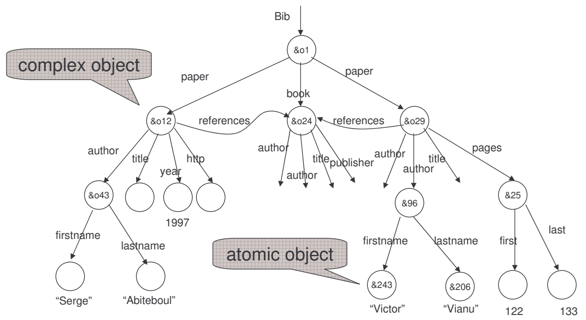
The problem

- 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

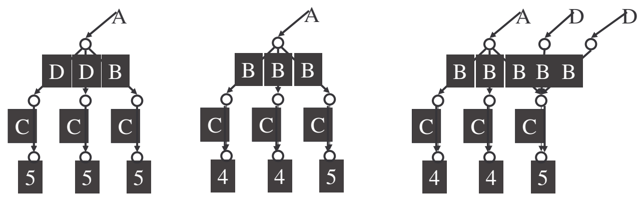
Indexing in LORE

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

Access plans

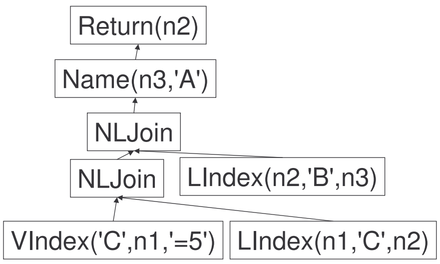
- Top down or bottom up navigation?

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



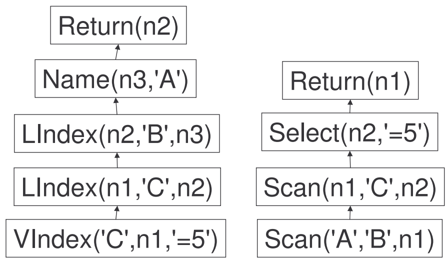
Access plans: bottom up

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



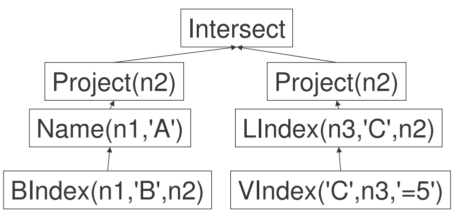
Bottom up and top down

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



Hybrid access plans

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



Path indexes

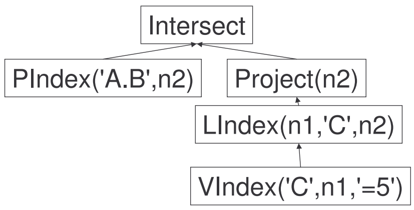
- PIndex(**p**, o): all objects reachable by the path **p**

Using a path index

select x

from A.B x

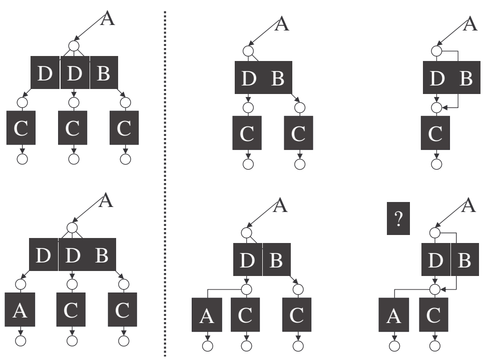
where x.C = 5



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*

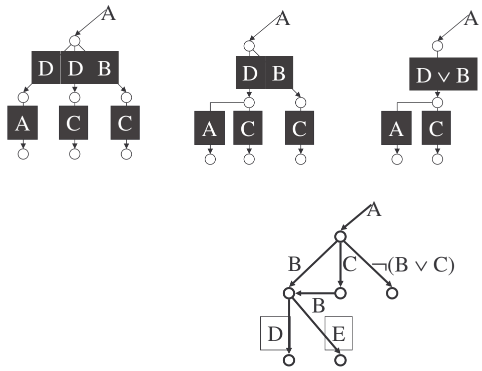
Optimization via dataguide

- 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 Schemas



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

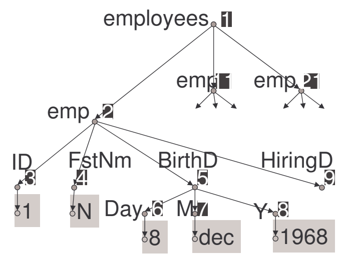
- Data may be schema-less
- Data may have a schema
- Data may change its schema over time

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

From	Pos	Tag	To	Data
-	1	employees	1	
1	1	emp	2	
2	1	ID	3	1
2	2	FN	4	Nancy
2	3	BD	5	
5	1	Day	6	8
5	2	Month	7	dec
5	3	Year	8	1968
2	4	HD	9	
9	…	…	…	
1	2	emp	11	
11	…	…	…	
1	3	emp	21	



Navigating the edge table

- //FN/text():
  select e.Data from edge e where e.Tag = 'FN'
- //emp[ID='1']/FN/text()
  select e3.Data
  srom 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

From	Pos	Tag	To	Data
-	1	employees	1	
1	1	emp	2	
2	1	ID	3	1
2	2	FN	4	Nancy
2	3	BD	5	
5	1	Day	6	8
5	2	Month	7	dec
5	3	Year	8	1968
2	4	HD	9	
9	…	…	…	
1	2	emp	11	
11	…	…	…	
1	3	emp	21	

employees

From	Pos	To	Data
-	1	1	

emp

From	Pos	To	Data
1	1	2	
1	2	11	
1	3	21	

ID

From	Pos	To	Data
2	1	3	1

Navigation

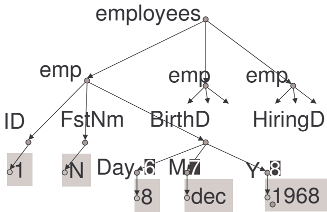
- //emp[ID='1']/FN/text()
  select e1.Data
  from edge e1, edge e2, edge e3
  where e1.Tag = 'emp' and e1.to = e2.from
  and e2.Tag = 'ID' and e2.Data = 1
  and e1.to = e3.from and e3.Tag = 'FN'

  ⇒ 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



ID	First Name	BD-D	BD-M	BD-Y

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

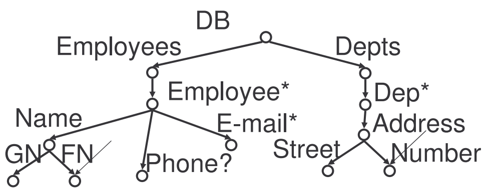
XRel approach: interval coding

- 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.start< N2.start and N2.end > N1.end
- Path expression: regexp matching with Path table, join the result with the data tables

XParent

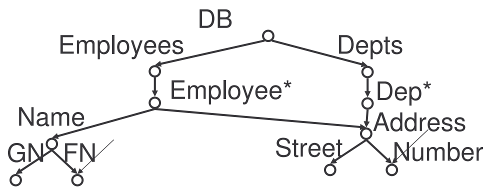
- Tables:
  - LabelPath(ID, lengh, pathExpr)
  - Data(pathID, nID, ord, value)
  - Element(pathID, ord, nID)
  - ParentChild(pID, cID)
- Use ParentChild instead of interval coding: equi-join instead of <-join

Schema-driven storage



DB(Id,Employees,Depts)
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)

Employee(PId,Id,…,Address,Addr.Street,Addr.Number)
Dep(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

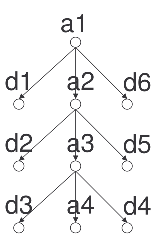
Navigation

- Simple paths:
  - Edge storage and join
  - Path partitioning and union
- Recursive paths:
  - Expansion to simple paths
  - Recursive navigation
  - Structural joins!

Structural Joins

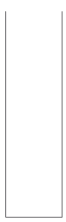
```
a=AList->firstNode; d=DList->firstNode; OutputList=NULL;
while((the input lists are not empty or the stack is not empty){
   if (a.StartPos > stack->top.EndPos
       && d.StartPos > stack->top.EndPos ) {
              stack->pop(); }
   else if (a.StartPos < d.StartPos) {
              stack->push(a)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

Stack-tree-desc



a1
a2
a3
a4

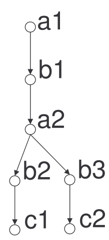
d1
d2
d3
d4
d5
d6

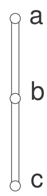


d1,a1
d2,a1
d2,a2
d3,a1
d3,a2
d3,a3
d4,a1
d4,a2
d4,a3
d5,a1
d5,a2
d6,a1

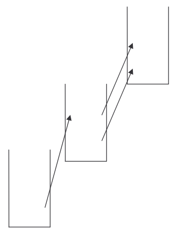
Holistic Path Joins

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





a1
b1
a2
b2
c1
b3
c2



a1, b1, c1
a1, b2, c1
a2, b2, c1
a1, b1, c2
a1, b3, c2
a2, b3, c2

Algorithm PathStack

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

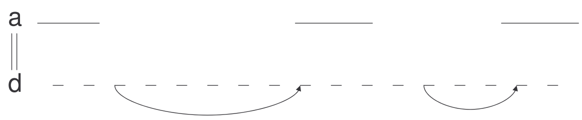
Holistic Twig Joins

```
while ¬end(q) {
   q_act = getNext(q);
   if ¬isRoot(q_act) { cleanStack(parent(q_act), nextL(q_act)); }
   if (isRoot(q_act) or notExpty(S[parent(q_act)]) {
         cleanStack(q_act, nextL(q_act));
         push( S[q_act], ( next(T[q_act]), top(S[parent(q_act)]) );
         advance(T[q_act]);
         if (isLeaf(q_act)) {
                  showSolutions(S[q_act]);
                  pop(S[q_act]); } }
   else {advance(T[q_act]); }
```

- getNext(q): next stream, skipping elements that do not participate in any solution

Skipping

- Linear time is not optimal:



- 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-vldbj99]
- XRel
- XParent
- Path partitioning in Monet [ScKeWiWa WEBDB 00]
- Holistic Path Joins [bks2002-twigjoin]