XML Storage

CPS 296.1

Topics in Database Systems

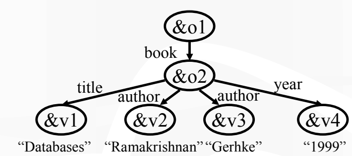
Approaches

- Text files
  - Use DOM/XSLT to parse and access XML data
- Specialized DBMS
  - Lore, Strudel, eXist, etc.
  - Still a long way to go
- Object-oriented DBMS
  - eXcelon (ObjectStore), ozone, etc.
  - Not as mature as relational DBMS
- Relational (and object-relational) DBMS
  - Middleware and/or object-relational extensions

Mapping XML to relational

- Just use a CLOB column
  - \+ Simple, compact, reasonable clustering
  - \+ Additional text indexing can help
  - – Updates are expensive
  - – Poor integration with query processing
- Use generic schema
  - Florenscu and Kossman, "A Performance Evaluation of Alternative Mapping Schemes for Storing XML Data in a Relational Database." Technical Report, INRIA, 1999
- Use DTD to derive schema
  - Shanmugasundaram et al., "Relational Databases for Querying XML Documents: Limitations and Opportunities." *VLDB*, 1999

Storing arbitrary XML



- Just a labeled directed graph
  - Internal nodes: elements with sub-elements and/or attributes; labeled by OID
  - Leaf nodes: attributes, or elements with atomic values; labeled by VID and value
  - Edges: links to sub-elements or attributes; labeled by name

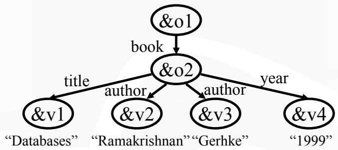
Mapping the link structure

- Edge table: edge(source, ordinal, name, target)
  - Source: parent OID
  - Target: child OID or VID
  - Name: attribute name, or tag name of the sub-element
  - Ordinal: order of the outgoing edges from source (corresponding to the order in the XML source)
- Primary key: {source, ordinal}
  - Primary index supports forward traversal
- Secondary Index: {name, target}
  - Supports backward traversal

Mapping leaf values

- Approach 1: separate value tables
  - One table for each datatype: string(VID, value), date(VID, value), etc.
  - Primary key: { VID }; secondary index: { value }
- Approach 2: inlining
  - In edge table, target stores value instead of VID
  - One column for each type, or
  - One VARCHAR column for all

Example mapping



edge

source	ordinal	name	target
&o1	1	book	&o2
&o2	1	title	&v1
&o2	2	author	&v2
&o2	3	author	&v3
&o2	4	author	&v4
…	…	…	…

value

VID	value
&v1	Databases
&v2	Ramakrishnan
&v3	Gerhke
&v4	1999
…	…

Mapping queries

- Path expression becomes joins
  - Example: book/section/title

```
select e3.target
from edge e1, edge e2, edge e3
where e1.name = 'book' and e1.target = e2.source
and e2.name = 'section' and e2.target = e3.source
and e3.name = 'title';
```

  ➢ Let relational query optimizer pick traversal (join) order!
- Wildcards require SQL3 recursion
  - Example: book//title

```
with reachable-from-book(tag, ID) as
  (select name, target from edge where name = 'book') union all
  (select name, target from reachable-from-book, edge where ID = source)
select ID from reachable-from-book where tag = 'title';
```

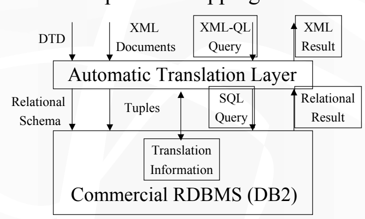
  ➢ Traditional query optimizer may not be smart enough to recognize the reverse join order

Experiments

- Joins hurt, but performance is reasonable for most queries, even complex ones
- Inlining helps a lot, even for big values
- Clustering edge table by name helps
- Certain queries, e.g., reconstruction of the original XML document, are expensive because of declustering
  - Recall that edge table is ordered by {source, ordinal}
  - Assigning OID's in DFS order helps, but edges are still not listed in DFS order

Storing XML with DTD

- Observation: more structure → more optimization
  - Much XML data conforms to pre-defined DTD's
  - Use DTD's to optimize mapping to relational schema



DTD graph

- Issues in mapping DTD to relational schema
  - Complex DTD specification involving wildcards
  - Tow-level nature of relational schema (tuples and attributes) versus arbitrary nesting of DTD
  - Recursion

Simplification of DTD

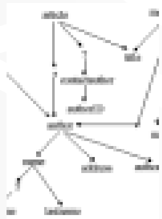
- Flattening
  - (e1, e2)* → e1*, e2*
  - (e1, e2)? → e1?, e2?
  - (e1 | e2) → e1?, e2?
- Grouping
  - …, e*, …, e*, … → e*, …
  - …, e*, …, e?, … → e*, …
  - …, e?, …, e*, … → e*, …
  - …, e?, …, e?, … → e*, …
- Simplification
  - e1** → e1*
  - e1*? → e1*
  - e1?* → e1*
  - e1?? → e1?

➢ Not equivalent transformations, but oh well…

Naïve mapping

- Each type of element becomes one relation, whose columns include
  - An ID
  - ID of the parent element (if applicable)
  - All attributes of the element
- Example
  - article(ID)
  - author(ID, parentID)
  - title(ID, parentID, value)
  - …

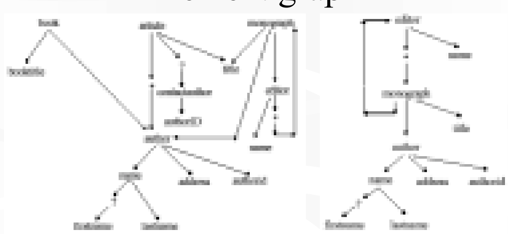
➢ Many joins!



Basic inlining

- Intuition: inline sub-elements as much as possible in order to avoid joins
- Complications
  - Cannot inline a set of sub-elements (*)
    - Resort back to join using foreign keys
  - Any element can be the root element
    - Create one relation per element
  - Recursion: inlining can go into an infinite loop
    - Detect and break cycles; again resort back to joins

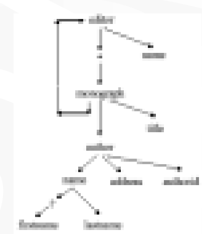
Element graph



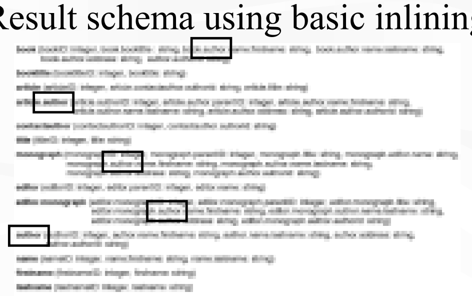
- For each element, construct an element graph using a DFS on the DTD graph
  - Shows what a valid XML document looks like if it is rooted at this element
  - Cycles are detected and treated as backlinks

Basic inlining algorithm

- For each element
  - Construct the element graph
  - Create a relation for the root
  - Inline all descendents, except
    - Subtree below *: create a new relation with parentID to the element above *
    - Node with an incoming backpointer: create a new relation with parentID to the source of the backpointer
- Example:
  - editor: inline name, plus parentID to editor.monograph
  - editor.monograph: inline everything below, plus parentID to editor



Result schema using basic inlining



- author becomes scattered (i.e., it appears once for each possible instantiation from a root)
  - Need multiple queries to find all authors

Shared inlining

- Same intuition as basic inlining: inline as many sub-elements as possible
  - Also as before, */recursion cannot be inlined
- However, to avoid scattering, do not inline an element if it is shared (i.e., appears in different contexts)
- Technique:
  - Node with in-degree 1 in the DTD graph: inline
    - Special cases: * and recursion
  - Node with in-degree 0: create a separate relation, because it cannot be inlined
  - Node with in-degree greater than 1: create a separate relation, because it is shared

Shared inlining example

- book: inline booktitle
- article: inline contactauthor
- monograph: inline editor and name, with parentID (to what?)
  - Note there is no relation for editor!
- title (shared)
- author (shared): inline everything

Result schema using shared inlining

[illegible]

- Subtlety 1: There is no relation for a non-shared, inlinable element (e.g., editor)
  - What if it is root? What if a foreign key needs to reference it?
  - Reuse the relation in which it appears (e.g., monograph)
    - Introduce isRoot column; set irrelevant columns to NULL
- Subtlety 2: A shared element appears in different contexts (e.g., /article/author, /book/author, etc.)
  - Together with parentID, we need to store parentCODE so we know in which relation to look for matching ID

Basic versus shared inlining

- Shared inlining reduces scattering and hence the number of queries
  - More efficient than basic inlining for finding all authors (anywhere in the XML document)
- Shared inlining introduces extra joins for processing path expressions
  - Less efficient than basic inlining for finding /book/author
- Best of both worlds?
  - ➢Hybrid inlining

Hybrid inlining

- Same as shared inlining, but additionally inline shared elements that are not recursive or below *
- Do not attempt to enumerate all contexts as basic inlining does

[illegible]

- author now appears twice (inlined once)
- title is now completely inlined twice

Shared versus hybrid inlining

- Hybrid inlining reduces joins through shared elements by inlining them whenever possible
  - No join needed for //book[contains(booktitle, ”database”]/author[firstname=”Jeff”] (shared inlining requires one)
- Hybrid inlining requires more queries to union together scattered information
  - Two queries to find //author[firstname=”Jeff”] (shared inlining only needs one)

➢ Shared inlining and hybrid inlining target query- and join-reduction respectively

Experiments

- 37 DTDs from real life
- Query set not from real life: all path expressions (that are valid in a given DTD) of a given length
- Metric
  - Total number of joins required for processing one path expression
  - Study trade-off of inlining
    - Number of queries per path expression
    - Number of joins per query

Results

- Basic inlining blows up with too many relations
- Shared versus hybrid
  - 35% of the DTD's: $J_{hybrid} << J_{shared}$, $Q_{hybrid} > Q_{shared}$, $TJ_{hybrid} < TJ_{shared}$
  - 5% of the DTD's: $J_{hybrid} << J_{shared}$, $Q_{hybrid} >> Q_{shared}$, $TJ_{hybrid} \sim TJ_{shared}$
  - 16% of the DTD's: $J_{hybrid} < J_{shared}$, $Q_{hybrid} >> Q_{shared}$, $TJ_{hybrid} > TJ_{shared}$
  - 43% of the DTD's: $J_{hybrid} \sim J_{shared}$, $Q_{hybrid} \sim Q_{shared}$, $TJ_{hybrid} \sim TJ_{shared}$
- Sets of sub-elements contribute to much of the fragmentation
- Number of joins per SQL query scales with the length of the path expression
- If all path expressions start from the root, one query
  - Hybrid is strictly better

Translating queries

- Translating path expressions
  - Inlined → no join required
  - Not inlined → join required
- Dealing with wildcards
  - Example: /article/child::*/lastname
    - Translation is not as simple as using the edge table
    - Traversal may go through either column (if inlined) or join (if not inlined)
    - Need to look at the schema and generate all instantiations
  - Example: /monograph//lastname
    - Recursion is required
  - Example: /book//lastname
    - No recursion is required

Structuring query results

- Simple results are fine
  - Each tuple returned by SQL query gets converted to an element
- Simple grouping is fine
  - Tuples can be returned by SQL query in sorted order; adjacent tuples are grouped into an element
- Complex results are problematic, e.g., article with multiple authors and multiple references
  - One SQL query can only return a single table, whose columns cannot store sets
  - Option 1: return one table, with all combinations of authors and references → bad
  - Option 2: return two tables, one with only authors and the other with only references → join is done outside the RDBMS

RDBMS wish list

- Support for sets
- Reference type to get rid of parentCODE
- IR indexes to facilitate full-text searches
- Flexible comparison to cast strings automatically into appropriate types
- Multiple-query optimization for processing path expressions
- Complex recursion for processing regular path expressions

Afterthoughts

- How does inlining relate to object clustering in object-oriented DBMS, or even clustering in relational DBMS?
- Instead of tweaking schema to get performance, should we implement better clustering support in DBMS?
- Starting with a schema without any inlining, how do we drive the clustering strategy? From schema, data, query workload, or query results?
- What if there is no DTD?
  - Use data mining to derive schema
  - Deutsch et al. "Storing Semistructured Data with STORED." *SIGMOD*, 1999