XML Research at UTA
Indexing XML Data

- When searching for web pages, you use keyword-based boolean formulas in a web search engine, such as Yahoo or Google:
  - database and “Leonidas Fegaras”
- Based on text retrieval (studied by IR researchers):
  - Only content words (terms) are indexed (a, the, of, etc are not)
  - Vector space model:
    - a document $D$ is represented by $n$ terms: $(d_1, \ldots, d_n)$, where $d_i \geq 0$ is the significance of the term in $D$. When $d_i=0$, the $i$th term doesn’t occur in $D$
    - Significance of a term in $D$ is proportional to the number of occurrences of this term in $D$ but inversely proportional to the number of documents containing this term
    - A query $Q$ is a vector $(q_1, \ldots, q_n)$, where $q_i$ is either 1 or 0.
    - Similarity function between a query $Q$ and a document $D$:
      \[ Q \cdot D = \sum q_i d_i \]
      - ie, how close is vector $Q$ to the vector $D$ in the $n$-dimensional space
    - Given $m$ documents, retrieve the $k$ closest documents.
Inverted Lists

• Document retrieval engines are based on inverted lists:
  – Given a term, find all documents that contain this term (a document list)
  – If multiple terms, get the intersection of all retrieved document lists
  – Then, find the k best documents from the intersection according to the similarity function.

• What about complex boolean formulas?
  – Use intersection for ‘and’, union for ‘or’, complement for ‘not’

• What about phrases, such as “computer science”?
  – Need to store the position of each term in a document
  – Then, you search for “science” in close proximity with “computer”

• One possible implementation of inverted list:
  – An inverted index: a B⁺-tree where the index key is the term (a word) and the data is a bucket that contains references to documents.
Web Search Engines

- Use crawlers (robots) to populate the inverted indexes off-line
  - Billions of documents
  - The web is dynamic; pages must be revisited since they may have been changed
  - Usually they employ depth-first search to index all links in a web page
  - Documents may be cached in local storage (google)
  - Not restricted to plain text documents (eg, PDF, PostScript, etc)
  - Indexes are cached in memory (100’s of GBs) while crawlers update a copy on disk
  - Effectiveness (quality of retrieved documents) is of high importance
    - Very complex similarity functions and term significance
    - Web pages are ranked according to the location and how many times are referenced by other web pages
    - Use of synonyms and related words
    - Relevance feedback is related to the list of retrieved web pages that are actually visited by the user.
Indexing Web-Accessible XML Data

• Need to index both structure and content:
  
  for $b$ in document("*")//book
  where $b$/author//lastname="Smith"
  return $b$/title

• Web-accessible queries may contain many wildcard projections:
  1. Users may be unaware of the detailed structure of the requested XML documents
  2. They may want to find multiple documents with incompatible structures using just one query
  3. They may want to accommodate a future evolution of structure without changing the query.

• Need to locate only those XML documents from the web that
  • Match all the paths appearing in the query
  • Satisfy the query content restrictions.
The XML Inverse Indexes

XML inverse indexes can be coded in ODL:

```plaintext
struct word_spec { doc, level, location };

struct tag_spec
{ doc, level, ordinal, beginloc, endloc };

class XML_word ( key word extent word_index )
{ attribute string word;
  attribute set< word_spec > occurs;
};

class XML_tag ( key tag extent tag_index )
{ attribute string tag;
  attribute set< tag_spec > occurs;
};
```
XML-OQL path expressions over web-accessible XML data can now be translated into OQL code over these indexes.

The path expression e/A is mapped to:

```
select y.doc, y.level, y.begin_loc, y.end_loc
from x in e
    a in tag_index,
    y in a.occurs
where a.tag="A"
and x.doc=y.doc
and x.level+1=y.level
and x.begin_loc<y.begin_loc
and x.end_loc>y.end_loc
```

A typical query optimizer will use the primary index of tag_index (a B⁺-tree) to find the elements with tag “A”.
But …

• Each projection in a web-accessing query, such as e/A, generates one big OQL query. What about:

  /books/book/author/lastname

This will generate a 4-level nested query.

• If you apply query unnesting though, you make this query flat:

  select b4
  from a1 in tag_index, b1 in a1.occurs,
       a2 in tag_index, b2 in a2.occurs,
       a3 in tag_index, b3 in a3.occurs,
       a4 in tag_index, b4 in a1.occurs
  where a1.tag="books" and a2.tag="book" and a3.tag="author"
  and a4.tag="lastname" and b1.doc=b2.doc=b3.doc=b4.doc
  and b1.level+1=b2.level and b2.level+1=b3.level and b3.level+1=b4.level
  and b1.begin_loc<b2.begin_loc and b1.end_loc>b2.end_loc
  and …
Need for a New XML Algebra

• Translating XQuery to OQL makes sense if the data are already stored in an OODB

• If we want access XML data in their native form (web-accessible files), from relational DBs etc, we need a new algebra well-suited for handling tree-structured data
  – Must capture all XQuery features
  – Must be suitable for efficient processing using the established relational DB technology
  – Must have solid theoretical basis
  – Must be suitable for query decorrelation (important for XML stream processing)
An XML Algebra

Based on the nested-relational algebra:

\( \rho_v(T) \) the entire XML data source T is accessed by v

\( \sigma_{\text{pred}}(X) \) select fragments from X that satisfy \( \text{pred} \)

\( \pi_{v_1,\ldots,v_n}(X) \) projection

\( X \cup Y \) merging

\( X \bowtie_{\text{pred}} Y \) join

\( \mu_{\text{pred}^v,\text{path}}(X) \) unnesting (retrieve descendents of elements)

\( \Delta_{\text{pred}^\oplus,\text{head}}(X) \) apply \text{head} and reduce by \( \oplus \)

\( \Gamma_{\text{group},\text{pred}^v,\oplus,\text{head}}(X) \) group-by \text{group}, apply \text{head} to each group, reduce each group by \( \oplus \)
Semantics

\[ \rho_v(T) \{ <v = T> \} \]

\[ \sigma_{pred}(X) \{ t | t \in X, \text{pred}(t) \} \]

\[ \pi_{v_1,\ldots,v_n}(X) \{ <v_1=t.v_1,\ldots,v_n=t.v_n> | t \in X \} \]

\[ X \cup Y \]

\[ X \bowtie_{pred} Y \{ tx \circ ty | tx \in X, ty \in Y, \text{pred}(tx,ty) \} \]

\[ \mu_{pred,v,path}(X) \{ t \circ <v=w> | t \in X, w \in \text{PATH}(t,path), \text{pred}(t,w) \} \]

\[ \Delta_{pred}^{\ominus,\text{head}}(X) \ominus/\{ \text{head}(t) | t \in X, \text{pred}(t) \} \]

\[ \Gamma_{\text{group},pred}^{v,\ominus,\text{head}}(X) \ldots \]
Example #1

where $b$/publisher = “Addison-Wesley”
and $b$/@year > 1991
return <book> { $b/title } </book>

\[
\Delta \cup_{\text{elem(“book”},$b/title)}
\]

\[
\sigma_{$b$/publisher=“Addison-Wesley”
and $b$/@year > 1991}$b
\]

\[
\mu_{\$v/bib/book}$v
\]

\[
\rho_{\text{document(“http://www.bn.com”)}}
\]
Example #2

```xml
<result>
  { for $u in document(“users.xml”)//user_tuple
    return <user> { $u/name }
    { for $b in document(“bids.xml”)//bid_tuple[userid=$/u/userid]/itemno $i in document(“items.xml”)//item_tuple[itemno=$b]
      return <bid> { $i/description/text() } </bid>
      sortby(.) }
    sortby(name) }
  </user>
</result>
```

$$\Delta$$

Document structure:
- `users.xml`
- `bids.xml`
- `items.xml`

**Sort Order**:
- `user_tuple`
- `bid_tuple`
- `item_tuple`

**Conditions**:
- `userid` in `bids.xml`
- `itemno` in `items.xml`

**Expression**:
- `<result>`
- `{ for $u in document(“users.xml”)//user_tuple
  return <user> { $u/name }
  { for $b in document(“bids.xml”)//bid_tuple[userid=$/u/userid]/itemno $i in document(“items.xml”)//item_tuple[itemno=$b]
    return <bid> { $i/description/text() } </bid>
    sortby(.) }
  sortby(name) }
</user>"
Translating XQuery into XML Algebra

- First, paths with wildcard selections (e//A) are instantiated to concrete paths.
- Then, the XQuery is translated into list comprehensions:
  \{ head | v_1 \in X_1, \ldots, v_n \in X_n, \text{pred} \}
- Then, comprehensions are normalized (if the domain of a generator is another comprehension, it is flatten out).
- Then, normalized comprehensions are converted into algebraic forms according to the algebra semantics.
- Finally, nested queries are unnested using a complete query decorrelation algorithm.
Query Decorrelation (Unnesting)

Similar to OODB query decorrelation:
Processing Streamed XML Data

• Most web servers are pull-based:
  – A client submits a request, the server returns the requested data
• Doesn’t scale well for large number of clients and large query results
• Replication of data increases availability, but there is a tradeoff between Quality of Data (QoD) and Quality of Service (QoS)
  – Cached web pages
  – Materialized views
• Alternative method: pushed-based dissemination
  – The server broadcasts data in a continuous stream
  – The client connects to multiple streams and evaluates queries locally
  – No handshaking, no error-correction
  – All processing is done at the client side
  – The only task done by the server is slicing and multicasting data
    • Critical data may be repeated more often than no-critical data
    • Invalid data may be revoked
    • New updates may be broadcast as soon as they become available.
A Framework for Processing XML Streams

• The server slices an XML data source (eg, an XML document) into manageable XML fragments.
  – Each fragment is a filler that fills a hole
  – Each fragment may contain holes which can be filled by other fragments
  – Each fragment is wrapped with control information, such as its unique hole ID, the path that reaches this fragment, etc

• The client opens connections to streams and evaluates XQueries against these streams.
  – For large streams, it’s a bad idea to reconstruct the streamed data in memory before start processing the data. Need to process fragments as soon they become available at the server
  – There are blocking operators that require unbounded memory:
    • Sorting
    • Joins between streams
    • Group-by/aggregation.
Handling Unbounded Memory

- The server sends *punctuations* that indicate properties about the content of data transmitted or about to transmitted
  - “You have already seen all stock prices for stocks beginning with ‘A’”
  - “You have already seen temperature measurements for 10am to 12pm”
- Punctuations are sent through the multicast stream as special fragments
- Clients can use these punctuations to flash their local buffers.
  - Scenario: a client performs an XQuery which, for each author, displays the titles of books written by the author.
    - Needs a group-by by author name which can be evaluated by a hash table
    - The server may send punctuations of the form: “you have already seen all authors beginning with ‘A’”
    - Upon receipt of a punctuation, the client can flush all buckets in the hash table whose group-by key satisfies the punctuation predicate.