Student Presentation: Multi-Query Processing for XML

CPS 216

Advanced Database Systems

Announcements (April 20)

- Homework #3 has been graded
  - Grades posted on Blackboard; sample solution available today
- Homework #4 due today
- Final exam on Monday, April 26
  - Open book, open notes; 3 hours—no time pressure!
  - Comprehensive, but with emphasis on the second half of the course and materials exercised in homework
  - ☞Final review this Thursday
  - ☞A sample final will be available on Thursday
- Project demo period: Tues./Wed. after the final
  - Final report due before the demo
  - ☞Your schedule will be confirmed by this Thursday

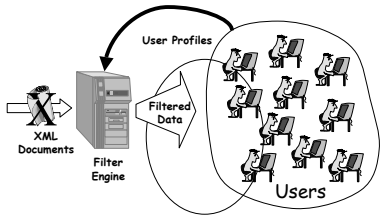
Multi-query processing for XML

- Swapna: introduction and Y-Filter (shared path processing)
  - Diao & Franklin. "Path Sharing and Predicate Evaluation for High-Performance XML Filtering." *TODS*, 2003
- Brian: IndexFilter (shared path processing using an interval-based index)
  - Bruno et al. "Navigation- vs. Index-Based XML Multi-Query Processing." *ICDE*, 2003
- Hao: shared XQuery processing
  - Diao & Franklin. "Query Processing for High-Volume XML Message Brokering." *VLDB*, 2003

XML Filtering

- In distributed computing services (Web Services, data and application integration etc.), XML is the way data to be exchanged should be encoded.
- In XML filtering system, continuously arriving streams of XML documents are parsed through a filtering engine.
- Documents are matched to query specifications and delivered.
- Queries are specified in XPath, which specifies constraints over structure (path expressions) and content (value-based predicates)

XML Filtering



The challenge is to efficiently and quickly match incoming XML documents against the potentially huge set of queries.

YFilter: shared Path Matching

Yanlei *Daio Et Al., ACM TODS, Dec. 2003*

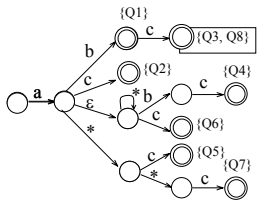
- Earlier project, XFilter, used event-based parsing and Finite State Machines (FSMs)
- A separate FSM is created for each path expression
  - Redundant work done – commonality among path expressions not exploited.
- For large-scale systems, shared processing is essential
- YFilter uses a Non-deterministic Finite Automation (NFA) based approach to share path matching work among queries

YFilter *continued…*

- Combines all queries into a single machine
- Merges common prefixes of paths
  - Common paths processed only once
- Query specifications written in XPath
  - Query path expressions expressed as sequence of location steps.
- Location step consists of
  - Axis – '/' and "//"
  - Node test – element name and wildcard operator '*'
  - Predicates

Representing XPath Queries in YFilter

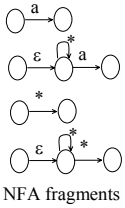
Q1 = /a/b
Q2 = /a/c
Q3 = /a/b/c
Q4 = /a//b/c
Q5 = /a/*/c
Q6 = /a//c
Q7 = /a/*/*/c
Q8 = /a/b/c



NFA Fragments for Location Steps

/a
//a
/*
//*

Location steps

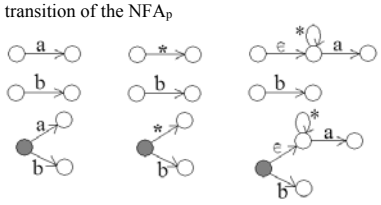


NFA fragments

- NFA fragments are directed graphs
- ε-transition moving to a state with self-loop

Constructing a Combined NFA

- Concatenate NFA fragments for location steps in a path expression
- Traverse combined NFA until
  1. NFA_p is reached.
  2. There is no transition matching the corresponding transition of the NFA_p



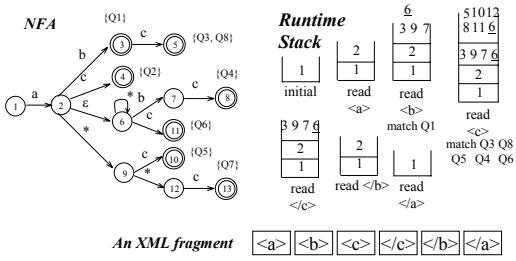
Implementing the NFA

- NFA basically uses a Hash table approach
- A data structure is created for each state having
  1. ID of the state
  2. Type information (accepting state or //-child)
  3. Hash table containing transitions from that state
  4. For accepting states, an ID list of corresponding queries
- Transition hash table contains [symbol, stateID] pairs
  - symbol (key) – label of outgoing transition
  - stateID – child state that the transition leads to

NFA Execution

- The NFA is executed in an event-driven fashion.
- As an arriving document is parsed, events raised by parser drives the NFA transitions.
- "End of element" event – NFA execution backtracks to the state it was when "start of element" was raised.
- Stack mechanism is used to enable backtracking.

NFA Execution



Predicate Evaluation

- Intuitive approach – extend NFA by including additional transitions to states representing successful evaluation of predicates.
  - Results in an explosion of number of states
  - Destroys path-sharing feature
- Inline approach – Value based predicates are processed as soon as elements in path expressions that predicates address are matched.
- Selection Postponed (SP) approach – waits until entire path expression is matched, and applies all value-based predicates for the matched path.

Inline vs SP

- Interesting observation: the delayed predicate processing of SP outperforms the eager processing of Inline by a wide margin.
- The main differences between the 2 approaches
  1. Inline: performs early predicate evaluation which does not prune future work

     SP: performs structure matching to prune the set of queries for which predicate evaluation needs to be considered
  2. Inline: evaluation of predicates in the same query happens at different independent states

     SP: failure of one predicate in a query stops evaluation of rest of the predicates immediately
  3. Inline: requires bookkeeping, maintenance cost includes setting the information and undoing it during backtracking

Performance Overview

- Sharing provides order-of-magnitude improvements
  - In the experiments, even with 100,000 concurrent queries, filtering was faster than the parser.
- No exponential blow-up of active states in NFA execution
- Robust under query workloads with "//" and '*' operators
- Efficient for query updates
  - Tens of milliseconds for inserting 1000 queries, and stabilizes at 5 ms after 50,000 queries exist in the system.
- For value-based predicates, SP approach performs better than Inline approach

Bibliography

[1] Franklin, M.J., Diao, Y. High-Performance XML Filtering: An Overview of YFilter

[2] Franklin, M. XML + Query Processing: A Foundation for Intelligent Networks. Available at http://www.cs.berkeley.edu/~franklin/Talks/XSym03.ppt

[3] Diao, Y., Altinel, M., Franklin, M. J., Zhang, H., Fischer, P. Path Sharing and predicate evaluation for high-performance XML filtering. Available at http://www.cs.berkeley.edu/~diaoyl/publications/yfilter-tods-2003-acm.pdf

Navigation vs. Index-Based XML Multi-Query Processing

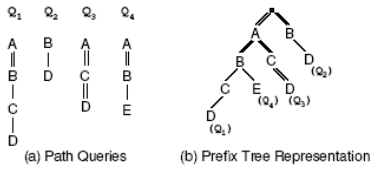
By : Brian Cook

Overview

- Problem : Multiple path queries need to be run against a stream XML documents
- Typical approach is to step through each tag one at a time to find a match
- Can indexing the XML documents speed up query processing?
  - Depends on the situation

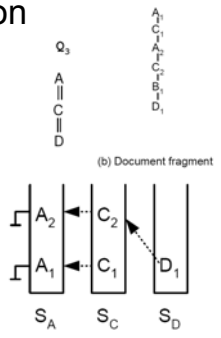
Prefix Sharing

- When processing several queries at the same time there may be some paths in common
- Reduces space needed to represent input queries



Y-Filter - Modification

- Y-Filter didn't generate actual results of matches
- Modified the algorithm by adding one stack per node in the prefix tree
  - The stack keeps track of all matches from the root to the given node
  - Each element is a pair :
    - <node from XML doc, pointer to position in the parent stack>



Index Filter - Overview

- Basic Idea – Index the document to avoid processing certain tags that cannot be part of any query.
- Requires that a pre-computed index is available on the xml document or one needs to be created on the fly.

Indexing Documents

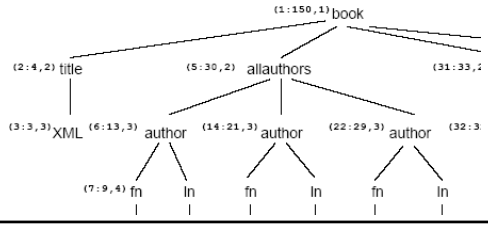
- Each XML tag has the following information associated with it (L : R , D)
  - L : Left – number of words from the beginning of the document until the **start** of the tag.
  - R : Right – number of words from the beginning until the **end** of the tag.
  - D : Depth – nesting depth of an element
- Create a B-tree for efficient lookup of these index nodes

Indexing Documents cont.

- Calculating ancestor / descendant relationships very easy
  - $Node_1$ is an ancestor of $Node_2$ if :
    - $L_1 < L_2$ and $R_1 > R_2$
  - $Node_1$ is a parent of $Node_2$ if :
    - $L_1 < L_2$ and $R_1 > R_2$ and $D_1 + 1 = D_2$
  - $Node_1$ is an descendant of $Node_2$ if :
    - $L_1 > L_2$ and $R_1 < R_2$
  - $Node_1$ is an child of $Node_2$ if :
    - $L_1 > L_2$ and $R_1 > R_2$ and $D_1 - 1 = D_2$

Example Indexed XML Doc

- *The* author *node* (6:13, 3) *is a descendant of the* book *node* (1:150, 1)
  - $L_{book} = 1 < 6 = L_{author}$ and $R_{book} = 150 > 13 = R_{author}$

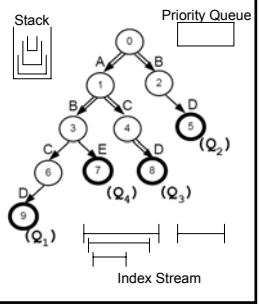


PathStack

- PathStack is a CPU and I/O optimal algorithm with these limitations :
  - Complete input has been read in
  - Can only process one query at a time
- Compared to Index-Filter by executing each query separately then aggregating the results.

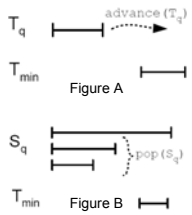
Data structures used in Index-Filter

- Associate each node (*q*) in the prefix tree :
  - An index stream *Tq*- indexed positions of document nodes that match *q*, sorted by their left values
  - A Stack – same as discussed earlier
  - A Priority Queue – access to child of the node having smallest left value in its index stream



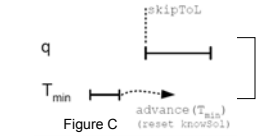
Index-Filter

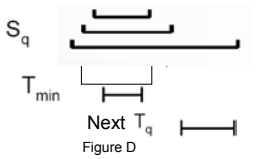
- Process the earliest child interval first (in document order)
  - Pick from the priority queue
- Pruning ancestors :
  - Skip until the node can be an ancestor (Figure A)
  - Pop all nodes on the stack that cannot be ancestors (Figure B)



Index-Filter

- Pruning descendents:
  - Skip until the node can be a descendant (Figure C)
- Output when (Figure D) :
  - *Tmin* cannot have any more ancestors
  - *Tmin* is accepting (end of a match)



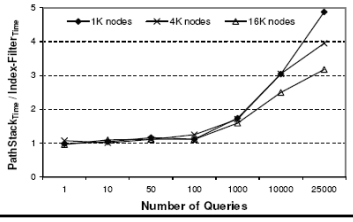


Results : Overview

- Index-Filter is 3-5 times better than PathStack for large number of queries (>1000)
- Index-Filter (pre-computed index) is better than Y-Filter when the number of queries is small (<500) or the document size is large
- On the contrary, Y-Filter outperforms Index-Filter for small documents and large number of queries
- Index-Filter (index on the fly) about the same as Y-Filter for most queries.

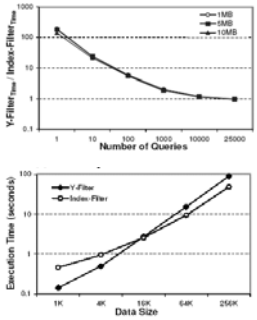
Results : Index-Filter vs. PathStack

- Small number of queries there is no difference between the two
- Index-Filter is 3 to 5 times more efficient for large number of queries,
- Why? Index-Filter uses the prefix tree to avoid processing the same portion of similar queries.



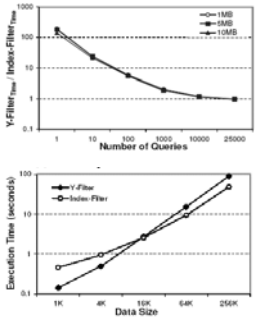
Index Filter vs. Y-Filter

- Index-Filter (pre-processed index) better :
  - When number of input queries are small (<500)
  - Document size is large
  - Why? Index-Filter only goes through a small part of the input document, since it only processes the indexes whose tags are present in the input query.



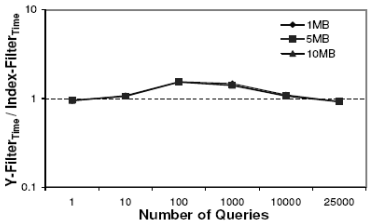
Index Filter vs. Y-Filter

- Y-Filter better :
  - When number of input queries are large (previous slide)
  - When the document size is small
  - Why? Y-Filter's hash tables scale better than priority queues in Index-Filter.



Index Filter vs. Y-Filter

- Index-Filter (on the fly index) :
  - About the same performance as Y-Filter when processing the index on the fly.



Conclusions

- "While most XML query processing techniques work off of SAX events, in some cases it pays off to parse the input document in advance and augment it with auxiliary information that can be used to evaluate the queries faster."
- However, if the index is not pre-processed then and needs to be created on the fly, the Index-Filter and Y-Filter are about the same.

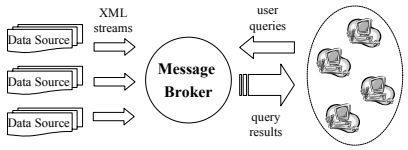
Query Processing for High-Volume XML Message Brokering

Yanlei Diao Michael Franklin (UC, Berkeley)

Presenter: Hao He

Revised slides stolen from author's homepage

XML Message Broker

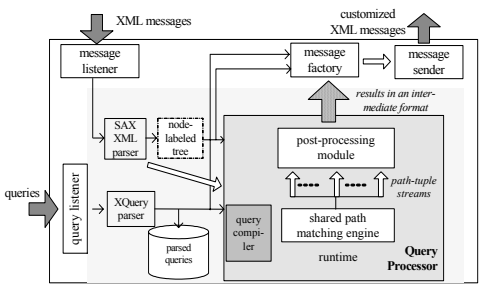


- **XML message brokers:** Central exchange points for messages sent between applications/users.
- **User subscriptions**: Specification of user interests, written in an XML query language.
- **XML streams**: Continuously arriving XML data items. The message broker matches data items to queries, transforms them, and routes the results.

Efficient Transformation

- **Goal**: customized result generation for tens of thousands of queries!
- Leverage prior work on shared path matching (i.e.,YFilter)
  - How, and to what extent can a shared path matching engine be exploited?
- Build customization functionality on top of it
  - What post-processing of path matching output is needed?
  - How can this be done most efficiently?

Message Broker Architecture



Query Specification

A query is a FLWR expression enclosed by a constant tag.

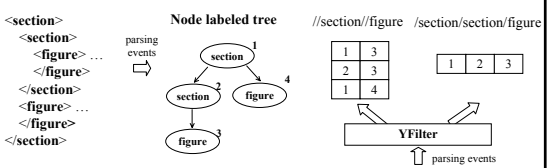
```
<sections>
{
  for    $s                in $doc//section
  where  $s/title          =  "XML"
    and  $s/figure/title =  "XML processing"
  return  <section>
            { $s//section//title }
            { $s//figure }
          </section>
}
</sections>
```

binding path

predicate paths

return paths

PathTuple Streams



A PathTuple stream for each matched path expression:

- PathTuple: A unique path match, one field per location step.
- Ordering: PathTuples in a stream are always output in increasing order of node ids in the last field.
- Path oriented shredding: query processing = operations on tuple streams.

Output of Query Processor

GroupSequence-ListSequence format for all the nodes selected from the input message.

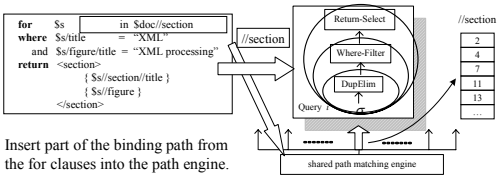
```
<sections>
{
  for    $s              in  $doc//section
  where  $s/title        =   "XML"
    and  $s/figure/title =   "XML processing"
  return <section>
           { $s//section//title }
           { $s//figure }
         </section>
}
</sections>
```

```
section_2:
    {title_6, title_8},
    {figure_5, figure_9}
section_11:
    {title_14},
    {figure_15, figure_16}
```


Basic Approaches

- Three query processing approaches exploiting shared path matching.
  - Post-process path tuple streams to generate results.
  - Plans consist of relation-style/tree-search based operators.
  - Differ in the extent they push work down to the path engine.
- Tension between shared path matching and result customization!
  - PathTuples in a stream are returned in a single, fixed order for all queries containing the path.
  - They can be used differently in post-processing of the queries.

Alternative 1: PathSharing-F



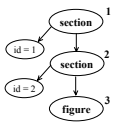
Insert part of the binding path from the for clauses into the path engine.

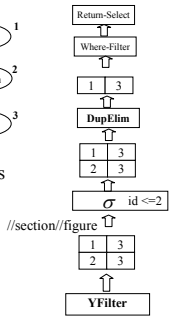
An external plan for *each* query:

- Selection: value-based comparisons in the binding path (//section[@id <= 2]).
- DupElim: when same node is bound multiple times in the stream.
- Where-Filter: tests predicate paths in the *where* clause (tree-search routine).
- Return-Select: applies the *return* clause (tree-search routine).

Duplicate Elimination

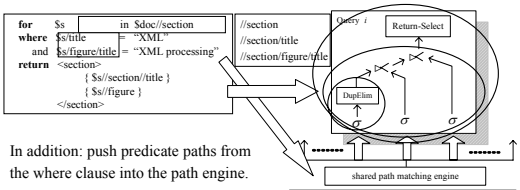
```
<figures>
{ for  $f  in
       $doc//section[@id<=2]//figure
  where  …
  return …
}
</figures>
```





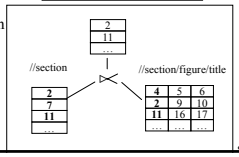
- Duplicates for the binding path: PathTuples containing the same node id in the last field.
- Cause redundant work in later operators and a duplicate result.
- DupElim ensures that the same node is emitted only once.

Alternative 2: PathSharing-FW



In addition: push predicate paths from the where clause into the path engine.

Semijoins: find query matches after paths in the for and the where clause are matched.

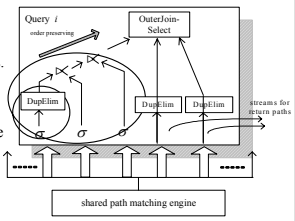


Alternative 3: PathSharing-FWR

Also push return paths from the return clause into the path engine.

OuterJoin-Select: generate results.

- create a group for each binding path tuple in the leftmost input.
- left outer join the binding path tuple with a return stream to create a list.
- order preserving
- hash vs merge based



Duplicates for a return path:

- Defined on the join field and the last field of the return path stream.
- Need DupElim on return paths before outer joins.

Optimizations

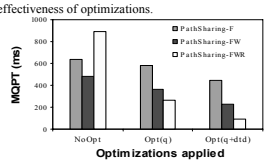
- Observation: More path sharing → more sophisticated processing plans.
- Tension between shared path streams and result customization.
  - Different notions of duplicates for binding/return paths.
  - Different stream orders for the inputs of join operators.
- Optimizations based on query / DTD inspection:
  - Removing unnecessary DupElim operators;
  - Turning hash-based operators to merge/scan-based ones.

Performance Comparison

- Three alternatives w./w.o. optimizations, non-recursive data

Summary of the results:

- PathSharing-FWR when combined with optimizations based on queries and DTD usually provides the best performance.
  - It performs rather poorly without optimizations.
- Effectiveness of optimizations:
  - Query inspection improves the performance of all alternatives;
  - Addition of DTD-based optimizations improves them further.
  - Recursive data challenges the effectiveness of optimizations.



Shared Post-processing

So far, a separate post-processing plan per query.

- The best performing approach (PathSharing-FWR) only uses relational style operators.
- Sharing techniques similar to shared *Continuous Query processing*, but highly tailored for XML message brokering.
  - Query rewriting
  - Shared group by for outer joins
  - Selection pullup over semijoins (NiagaraCQ)
  - Shared selection (TriggerMan, NiagaraCQ, TelegraphCQ)
- Shared post-processing can provide great improvement in scalability!

Conclusions

Result customization for a large set of queries:

- Sharing is key to high-performance.
- Can exploit existing path sharing technology, but need to resolve the inherent tension between path sharing and result customization.
- Results show that aggressive path sharing performs best when using optimizations.
- Relational style operators in post-processing enable use of techniques from the literature (multi-query optimization, CQ processing).