XML Stream Processing

Yanlei Diao
University of Massachusetts Amherst
XML Data Streams

- XML is the “wire format” for data exchanged online.
  - Purchase orders
    http://www.oasis-open.org/committees/tc_home.php?wg_abbrev=ubl
  - News feeds
    http://blogs.law.harvard.edu/tech/rss
  - Stock tickers
    http://www.tickertech.com/products/xml/
  - Transactions of online auctions
    http://bigblog.com/online_auctions.xml
  - Network monitoring data
    http://ganglia.sourceforge.net/
XQuery Usage Scenarios

- XML message brokers
  - Simple path expressions, for authentication, authorization, routing, etc.
  - Single input message, relatively small
  - Transient and streaming data (no indexes)

- XML transformation language in Web Services
  - Large and complex queries, mostly for transformation
  - Input message + external data sources, small/medium sized data sets ($xK \rightarrow xM$)
  - Transient and streaming data (no indexes)

- Semantic data verification
  - Mostly messages
  - Potentially complex (but small) queries
  - Streaming and multi-query optimization required
XML Stream Processing

- XML data continuously arrives from external sources, queries are evaluated every time a new data item is received.

- A key feature of stream processing is the ability to process as it arrives.
  - Natural fit for XML message brokering and web services where messages need to be filtered and transformed on-the-fly.

- XML stream processing allows query execution to start before a message is completely received.
  - Short delay in producing results.
  - No need to buffer the entire message for query processing.
  - Both are crucial when input messages are large, e.g. the equivalent of a database’s worth of data.
Event-Based Parsing

- XML stream processing is often performed on the granularity of an XML parsing event produced by an event-based API.

```xml
<?xml version="1.0" ?>
<report>
  <section id="intro" difficulty="easy">
    <title>Pub-Sub</title>
  </section>
  <section difficulty="easy">
    <figure source="g1.jpg">
      <title>XML Processing</title>
    </figure>
  </section>
  <section>
    <figure source="g2.jpg">
      <title>Scalability</title>
    </figure>
  </section>
</report>
```

```xml
<Start Document>
  <Start Element: report>
    <Start Element: section>
      <Start Element: title>
        Characters: Pub/Sub
      </title>
      <End Element: title>
      </section>
      <Start Element: title>
        Characters: XML Processing
      </title>
      <End Element: title>
    </section>
    <End Element: section>
    </section>
  </End Element: section>
  <End Element: report>
  </End Document>
```
Matching a Single Path Expression

XFilter [Altinel&Franklin’00], YFilter [Diao et al.’02], Tukwila [Ives et al.’02]

- Simple paths: ( (“/” | “//”) (ElementName | “*”) )+
  - A simple path can be transformed to a regular expression
  - Let $\Sigma =$ set of element names:
    - ‘/’ is translated to the ‘·’ concatenation operator
    - “//” is translated to $\Sigma^*$
    - ‘*’ is translated to $\Sigma$
    - “/a//b” can be translated to “a $\Sigma^*$ b”

- A finite state machine (FSM) for each path: mapping steps to machine states.
Query Compilation

Map location steps to FSM fragments
- Location steps
  - \(/a\)
  - \(/\ast\)
  - //a
- FSM fragments

Concatenate FSM fragments for location steps in a query
- Query “/a//b”

Is the FSM deterministic or non-deterministic?
Event-Driven Query Execution

- Query execution retrieves all matches of a path expression.
- Event-driven execution of FSM:
  - Parsing events (esp. start of elements) drive the FSM execution.
  - Elements that trigger transitions to the accepting states are returned.
- Execute an FSM over XML data with nested structure:
  - Approach 1: shred XML into a set of linear paths
  - Approach 2: augment FSM execution with backtracking
Multi-Query Processing

- Problem: evaluate a set $Q = Q_1, \ldots, Q_n$ of path queries against each incoming XML document.
- Brute force: iterate the query set, one query at a time
- Indexing of queries: inverse problem of traditional query processing
  - **Traditional DB:** Data is stored persistently; queries search the data for results. **Indexes of data** enable it to be searched without having to sequentially scan it.
  - **XML stream processing:** Queries are persistently stored; documents or their parsing events drive the matching of queries. **Indexes of queries** enable selective matching of documents to queries.
- **Sharing of processing:** commonalities exist among queries. Shared processing avoids redundant work.
Constructing the Combined FSM

YFilter [Diao et al.'03] builds a combined FSM for all paths.

- Complete prefix sharing among paths.
- Moore machine with an output function: accepting states → partition of query ids.
- Nondeterministic Finite Automaton (NFA)-based implementation: a small machine size, flexible, easy to maintain, etc.

\[
\begin{align*}
Q1 &= /a/b \quad Q5 = /a/*/b \\
Q2 &= /a/c \quad Q6 = /a//c \\
Q3 &= /a/b/c \quad Q7 = /a/*//*/c \\
Q4 &= /a//b/c \quad Q8 = /a/b/c
\end{align*}
\]

Yanlei Diao, University of Massachusetts Amherst
YFilter uses a stack mechanism to handle XML.

- Backtracking in the NFA.
- No repeated work for the same element.
Implementation Choices

- **Non-Deterministic Automata (NFA) versus Deterministic Automata (DFA)**
  - NFA: small machine, large numbers of transitions per element
  - DFA: potentially large machine, one transition per element

- **Worse-case comparison for regular expressions**

<table>
<thead>
<tr>
<th></th>
<th>Single regular expression of length $n$</th>
<th>$m$ regular expressions compiled together</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Processing complexity</td>
<td>Machine size</td>
</tr>
<tr>
<td>NFA</td>
<td>$O(n^2)$</td>
<td>$O(2n)$</td>
</tr>
<tr>
<td>DFA</td>
<td>$O(1)$</td>
<td>$O(\Sigma^n)$</td>
</tr>
</tbody>
</table>

- **Restricted path expressions**

  - YFilter specific
  - Possible in practice?

---

Yanlei Diao, University of Massachusetts Amherst  
5/1/2006
Eager DFA

- Green et al. studied size of DFA for the restricted set of path expressions [Green et al.’03]
- Eager DFA:
  - Query compile time, translates from NFA to DFA
  - Creates a state for every possible situation that runtime may require
- Single path:
  - Linear for “//e1/e2/e3/e4/e5”
  - Exponential for “//e1/*//*/e5”
- Multiple paths:
  - Exponential for “//e1//b”, “//e2//b”, …, “//e5//b”
Example of an Eager DFA

//a/*/*/c/d

Need to remember all the A’s in three consecutive characters, as different combinations of As may yield different results.
“a a b b c d”
“a b a b c d”

The DFA size is $O(2^{w+1})$ where $w$ is the number of ‘*’. 
Lazy DFA

- Lazy DFA is constructed at run time, on demand
  - Initially, it has a single state
  - Whenever it attempts to make a transition into a missing state, it computes it and updates the transition
  - Hope: only a small set of the DFA states is needed.

- Exploits DTD to derive upper bounds
  - A DTD graph is simple, if the only loops are self-loops.
    - Theorem: size of lazy DFA is exponential only in the maximal number of simple cycles a path can intersect. (not exponential in # of paths!)
  - More complex recursive DTDs: e.g. “table” contains “list” and “list” contains “table”
    - Even lazy DFA grows large…

DTD graph
Predicates in Path Expressions

- Predicates can address attributes, text data, or positions of elements.
  - **Value of an attribute** in an element, e.g.,
    //section[@difficulty = “easy”].
  - **Text data** of an element, e.g.,
    //section/title[text()=“XPath”].
  - **Position** of an element, e.g.,
    //section/figure[text()=“XPath”][1].
Predicate Evaluation

- **Extend the NFA**
  - including additional states representing successful evaluation of predicates and transitions to them

- **Potential problems**
  - A potentially huge increase in the machine size
  - Destroy sharing of path expressions

- **Recent work [Gupta & Suciu 2003]**: Possible to build an efficient pushdown automaton using lazy construction if
  - No mixed content, such as “<a> 1 <b> 2 </b> </a>”.
  - Can afford to periodically rebuild the automaton from scratch.
  - Can afford to train the automaton in each construction.
XQuery is much more complex than regular expressions.

Leverage efficient relational processing for transformation

- Complex XML stream processing = relational query operations on path-tuple streams!

YFilter: Mapping XML to Relational

P1: //section//figure  P2: //section/section/figure

\[
\begin{align*}
\text{report} & \quad 1 \\
\text{section} & \quad 2 \\
\text{title} & \quad 3 \\
\text{section} & \quad 4 \\
\text{figure} & \quad 5 \\
\text{title} & \quad 6 \\
\text{figure} & \quad 7 \\
\text{title} & \quad 8 \\
\end{align*}
\]

\[
\begin{align*}
\{P1\} & \quad \text{figure} \\
\{P2\} & \quad \text{figure} \\
\end{align*}
\]
Example Query Plan for FWR

Q1:
for $s$ in $doc//section[@difficulty="easy"]
where $s/title = "Pub/Sub" 
and $s/figure/title = "XML processing"
return <section>
    { $s//section//title }
    { $s//figure }
</section>

Push all paths into the path engine.

An external (post-processing) plan for each query:
- Selection: evaluates value-based predicates.
- Projection: projects onto specific fields and removes duplicates.
- Semijoin: handles correlations between for and where paths, finds query matches.
- Outerjoin-Select: handles correlations between for and return paths, generates query results.

Yanlei Diao, University of Massachusetts Amherst
Buffering in XML Stream Processing

- Buffering in XQuery stream processing
  - Whether an element belongs to the result set depends on other predicates
    - //section[title="XML"]
      - Buffer: section
      - Until the end of section is encountered or when no more title can occur (use DTD!)
    - //section[figure="XML"]/title
      - Buffer: title
      - Until a figure matches the predicate or the end of section is encountered
    - //section[.//title="XML"]//figure
      - Buffer: e.g. figure(5) for section(2)-figure(5) and section(4)-figure(5)
      - Until both paths are satisfied (with the predicates) or none of them can be satisfied any more

Yanlei Diao, University of Massachusetts Amherst
5/1/2006
Full XQuery Stream Processor

- Stream processing for the entire XQuery language [Daniela et al.’04]
  - General algebra for XQuery
  - Pull-based token-at-a-time execution model
  - Lazy evaluation (like other functional languages)
  - Some preliminary work on sharing [Diao et al.’04]

- Buffering is a big concern [Barton et al.’03, Peng&Chawathe’03, Koch et al.’04]

- Sharing is crucial for performance and scalability for large numbers of queries