XML Stream Processing

Yanlei Diao
University of Massachusetts Amherst

XML Data Streams

- XML is the “wire format” for data exchanged online.
  - Purchase orders
  - News feeds
    [http://blogs.law.harvard.edu/tech/rss](http://blogs.law.harvard.edu/tech/rss)
  - Stock tickers
  - Transactions of online auctions
    [http://bigblog.com/online_auctions.xml](http://bigblog.com/online_auctions.xml)
  - Network monitoring data

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

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 finite automaton (FA) for each path: mapping location steps to machine states.

Query Compilation

Map location steps to automaton fragments

/\( a \)
/\( * \)
/\( a \)

Concatenate automaton fragments for location steps in a query

Query "a/b" for multi-path processing o.w. not needed

Is this Automaton deterministic or non-deterministic?
Event-Driven Query Execution

- Query execution retrieves all matches of a path expression.
- Event-driven execution of FA:
  - Parsing events (esp. start of elements) drive the FA execution.
  - Elements that trigger transitions to the accepting states are returned.
- Run FA over XML with nested structure, retrieve all results
  - Approach 1: shred XML into a set of linear paths
  - Approach 2: augment FA 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 executed against the data. Indexes of data enable selective searches of the data.
  - 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 FA for all paths.
- Complete prefix sharing among paths.
- Moore machine with output: accepting states \( \rightarrow \) partition of query ids.
- Nondeterministic Finite Automaton (NFA)-based implementation: a small machine size, flexible, easy to maintain, etc.

\[
\begin{align*}
Q_1 &= /a/b \\
Q_2 &= /a/c \\
Q_3 &= /a/b/c \\
Q_4 &= /a/b/c \\
Q_5 &= /a/*/b \\
Q_6 &= /a//c \\
Q_7 &= /a/*/c \\
Q_8 &= /a/b//c
\end{align*}
\]
YFilter uses a stack mechanism to handle XML.

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

### Execution Algorithm

![Diagram of NFA and Runtime Stack]

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

- **Restricted path expressions**

- **Eager DFA**

  - Green et al. studied the 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 'A's may yield different results.

The DFA size is $O(2^w)$ 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...

Lazy DFA (contd.)

- A DTD graph is simple, if the only loops are self-loops.
- **Theorem**: the size of a 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 a lazy DFA grows large...
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.

YFilter: Mapping XML to Relational

- XQuery is much more complex than regular expressions.
- Leverage efficient relational processing
- XQuery stream processing = relational operations on path-tuple streams!
Example Query Plan for FWR

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.

Buffering in XML Stream Processing

- Buffering in XQuery stream processing
  - Whether an element belongs to the result set is uncertain as it depends on predicates that have not been evaluated
- Goal: avoid materialization and minimize buffering
- Issues to address:
  - What queries require buffering?
  - What elements to buffer?
  - When and how to prune dynamic data structures?
    - Buffered elements
    - State in any dynamic data structures

Buffering in XML Stream Processing

//section[title="XML"]
- Requires buffering?
  - Yes
- What element to buffer?
  - section
- When to prune?
  - until the end of section is encountered, or
  - when no more title can occur in this section. (Use DTD!)
Buffering in XML Stream Processing

//section[figure="XML"]//title

- Requires buffering?
  - Yes
- What element to buffer?
  - title
- When to prune?
  - until a figure matches the predicate, or
  - when no more figure can occur in this section (use DTD), or
  - the end of section is encountered.

Yanlei Diao, University of Massachusetts Amherst 5/3/07