AFilter: Adaptable XML Filtering with Prefix-Caching and Suffix-Clustering

K. Selçuk Candan
Wang-Pin Hsiung
Songting Chen
Junichi Tatemura
Divyakant Agrawal

NEC Laboratories America, Inc.
Motivation: Efficient Message Filtering

Aim:
- large number of filter statements
- high throughput

Is this message relevant to any business process?
Assumptions

Expressions are of type, $P\{/,,*\}$

XML Data stream

Messages are in some XML format

Filter Statements

Path Matches
XML Path Filtering

Expressions are of type, $P\{/, //, *\}$

What is the most appropriate internal (index+runtime) representation?

Filtered Statements

XML Data stream

Messages are in some XML format
Approach I: Finite Automata

• Input (path) is a string
  – of elements from a root to a leaf
• Filter statements are
  – (path) expressions with wildcards

• So why not use DFA/NFAs?
  – YFilter [Diao et al.], XScan [Ives et al.], XQRL [Florescu et al.],
Finite Automata

• Each data node causes a state transition in the underlying FA representation of the filters

(a) XPath queries

(b) A corresponding NFA

Figure taken from YFilter
Finite Automata

• Each data node causes a state transition in the underlying representation of the filters

• Problem in
  – deep
  – recursive
data

• Number of active states can be exponentially large [Diao et al.], [Green et al.],

Figure taken from YFilter
Finite Automata

• Each data node causes a state transition in the underlying FA representation of the filters

• Use “lazy” state enumeration as opposed to an “eager” approach [Green et al.]
  – Helps, but still exponential in query depth

Figure taken from YFilter
Approach 2: Push Down Automata

- Use a stack to organize the data & states

**XPush** [Gupta et al.], **SPEX** [Olteanou et al.], **XSQ** [Peng et al.]

---

**Stack-based memory management for the states**

Figure taken from XPush
Push Down Automata

• Use a stack to organize the data&states

  • Depending on the approach used the number of active states can be
    – exponentially large in the number of predicates (XPush)
    – quadratic in the depth of the stream (SPEX)
    – query_size * depth_of_document (PathM)

Figure taken from XPush
Other Approaches

- **XTrie** [Chan et al.]
  - Uses “tries” for path string matching
  - Benefits from prefix commonalities
  - No suffix sharing across filter statements

- **FiST** [Kwon et al.]
  - Filters the entire (twig) statement holistically
  - Little sharing across filter statements

- **TurboXPath** [Josifovski et al.]
  - Avoids FAs
  - Little sharing across filter statements

- [Bar-Yossef et al.]
  - Effective use of buffers
  - Little sharing across filter statements
Observations

• Major savings in execution time can only come from simultaneous prefix and suffix sharing
  – can we actually do this?

• Active state enumeration is costly
  – can we have a compact representation and lazy (triggered) enumeration?

• We shouldn’t need too much memory for correct filtering
  – can we take the use of memory under our control?
AFilter (a modular architecture)
AFilter

Linear size indexing of filter statements

XML Data Stream

MEMORY MNGMT. (CACHING)

LAZY RESULT ENUMERATION (TRIGGERS)
AxisView (blueprint for filters)

\{ q_1 = \texttt{d/a/b}, q_2 = \texttt{a/b/a/b}, q_3 = \texttt{a/b/c}, q_4 = \texttt{a* c} \}
AxisView

One node per symbol

One “assertion” per query step

\[ \{q_1 = /d//a//b, q_2 = /a//b//a//b, q_3 = /a//b//c, q_4 = /a/*/c\} \]
AxisView

Edges from leaves to root

\[
\begin{align*}
q_1 &= /\text{d}/\text{a}/\text{b} \\
q_2 &= /\text{a}/\text{b}/\text{a}/\text{b} \\
q_3 &= /\text{a}/\text{b}/\text{c} \\
q_4 &= /\text{a}/\text{*}/\text{c}
\end{align*}
\]
AxisView

One trigger per query

\[
\{q_1 = /\star /d/ /a/ /\bar{b}, q_2 = /\star /a/ /\bar{b}/ /a/ /\bar{b}, q_3 = /\star /a/ /\bar{b}/ /c, q_4 = /a/ /\star /c\}\]

© NEC Corporation 2006
PRLabel-tree (optional, trie)

Prefix labels

\[ \{ q_1 = /a//b, q_2 = /a//b//a//b, q_3 = /a//b//c, q_4 = /a//c \} \]
SFLLabel-tree (optional, trie)

\{q_1 = / / d / / a / / b, q_2 = / / a / / b / / a / / b, q_3 = / / a / / b / / c, q_4 = / / a / * / c\}
AFilter
StackBranch (path encoding)

Conceptually similar to PathStack [Bruno et al.] for structural joins

Encodes the hierarchical information in the current path segment compactly

One stack per symbol
StackBranch (path encoding)
StackBranch (path encoding)
Triggering

Triggering query steps

© NEC Corporation 2006
Triggering & following

Can we reach the root stack?

Advantages:
- edges are never followed if no triggering
- benefits from tighter selectivity at the leaves
- edges are followed in a clustered manner
Clustered edge following

Hash Join
Clustered edge following

Hash Join

Continue follow
Clustered edge following

Hash Join

Cont.
Clustered edge following

Hash Join
Clustered edge following

- one path match found -
Clustered edge following

- one path match found -

Note:
• Memory usage linear (in the depth of the data tree)

Challenge:
• If we have more memory, can we trade it for efficiency?
AFilter
Prefix caching / PRCache

• Observation:
  – repeated evaluations of the same candidate assertion at a node will always lead to the same result.
Prefix caching / PRCache

• Observation:
  – repeated evaluations of the same candidate assertion at a node will always lead to the same result.
Prefix caching / PRCache

- Observation:
  - repeated evaluations of the same candidate assertion at a node will always lead to the same result.

Alt 1. Index and cache partial results against the prefix labels
- prevents redundant traversals
- enables prefix sharing

Alt 2. Index and cache only the failures
- prevents non-productive traversals
Prefix caching / PRCache

• Observation:
  – repeated evaluations of the same candidate assertion at a node will always lead to the same result.

Index and cache partial results against the prefix labels

Index and cache only the failures (prevent non productive traversals)

Note:
• Decouples memory/cache management from correctness.
• The system
  – will work correctly, even if the cache size is zero!
  – Will work faster if there is some cache..

Challenge:
• Can we benefit from suffix commonalities across filter statements?
Suffix compressed traversals

Large number of query steps increases the hash join cost during edge traversal

(q₁ = /a/b, q₂ = /a/b/a/b, and q₃ = /c/a/b)
Suffix compressed traversals

(q₁ = //a//b, q₂ = //a//b//a//b, and q₃ = //c//a//b)

SFLabel tree
Suffix compressed traversals

(q₁ = //a//b, q₂ = //a//b//a//b, and q₃ = //c//a//b)

SFLabel tree

Problem:
• Prefix caching and suffix clustering are not entirely compatible.
Overlaps in Prefix/Suffix labels

prefix equal assertions
Overlaps in Prefix/Suffix labels

Problem:
- Use of suffix labels (instead of individual assertions) may hide prefix caching opportunities
Overlaps in Prefix/Suffix labels

Problem:
- Use of suffix labels (instead of individual assertions) may hide prefix caching opportunities.

In the paper, we describe:
- early unfolding, and
- late unfolding
schemes to discover prefix caching opportunities during suffix compressed traversals.
AFilter
Experiment Setup

- Java (JDK 1.5) implementation
- 1.7GHZ Pentium 4
- Data
  - NITF DTD
  - Book DTD
  - ToXgene data generator [Barbosa et al.]

<table>
<thead>
<tr>
<th>Acronym</th>
<th>Filtering approach</th>
</tr>
</thead>
<tbody>
<tr>
<td>YF</td>
<td>YFilter</td>
</tr>
<tr>
<td>AF-nc-ns</td>
<td>AFilter, no cache, no suffix x compression</td>
</tr>
<tr>
<td>AF-nc-suf</td>
<td>Suffi x Compressed AFilter, no cache</td>
</tr>
<tr>
<td>AF-pre-ns</td>
<td>AFilter, prefix x caching only, no suffix x compression</td>
</tr>
<tr>
<td>AF-pre-suf-early</td>
<td>Suffi x Compressed AFilter, prefix x cache, early unfolding</td>
</tr>
<tr>
<td>AF-pre-suf-late</td>
<td>Suffi x Compressed AFilter, prefix x cache, late unfolding</td>
</tr>
</tbody>
</table>
Experiment results

![Graph showing Time vs Cache Size for 100K Queries with different cache size measurements. The graph indicates a decrease in time as cache size increases, with a specific notation for "AFilter + Suffix + Cache" showing the lowest time.](image-url)
Experiment results
Experiment results
Experiment results
Conclusions

- AFilter
  - provides tradeoff between memory and performance and can work with only linear memory (if needed)
    - decouples memory management from correctness
  - avoids unnecessarily eager result/state enumerations
    - triggering benefits lower selectivities at the leaves
  - exploits simultaneously various sharing opportunities:
    - common steps (AxisView),
    - common prefixes (PRLabel-tree), and
    - common suffixes (SFLabeltree).
- The best results are obtained when both prefix and suffix clustering are exploited simultaneously.
Empowered by Innovation

NEC

NEC Laboratories America, Inc.