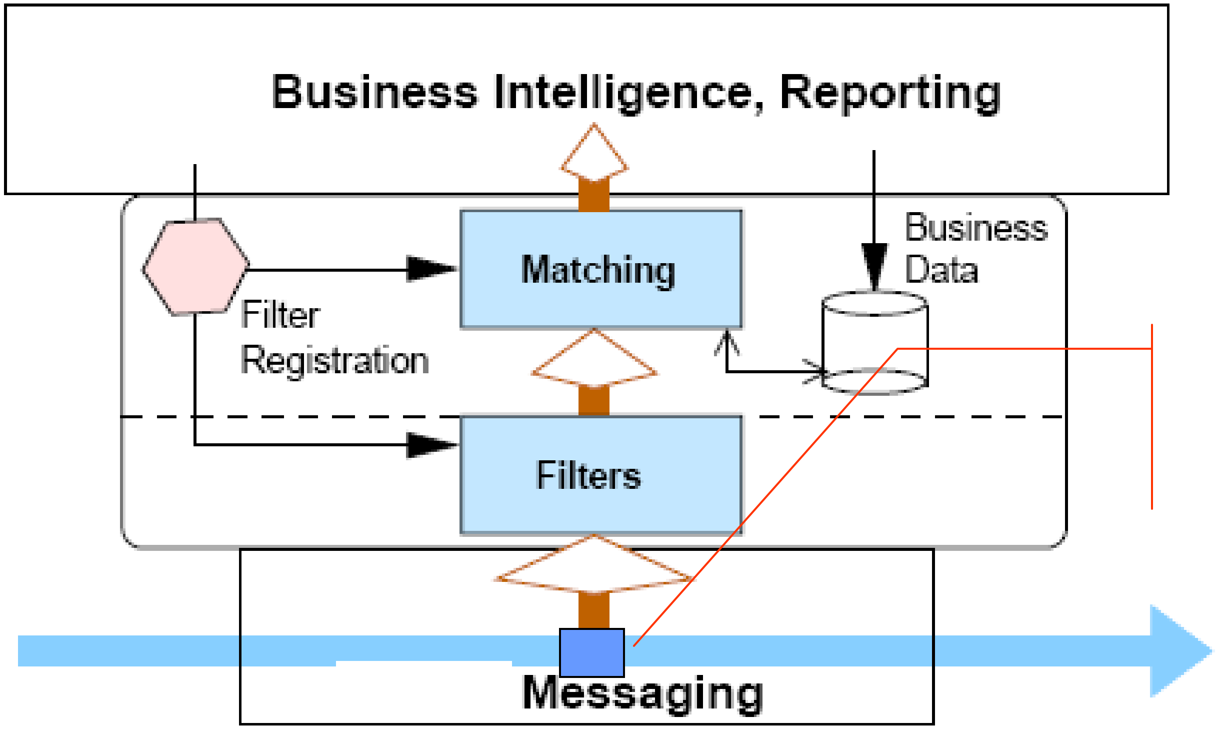
AFilter: Adaptable XML Filtering with Prefix-Caching and Suffix-Clustering

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

NEC Laboratories America, Inc.

Empowered by Innovation NEC

Motivation: Efficient Message Filtering



Is this message relevant to any business process?

Aim:

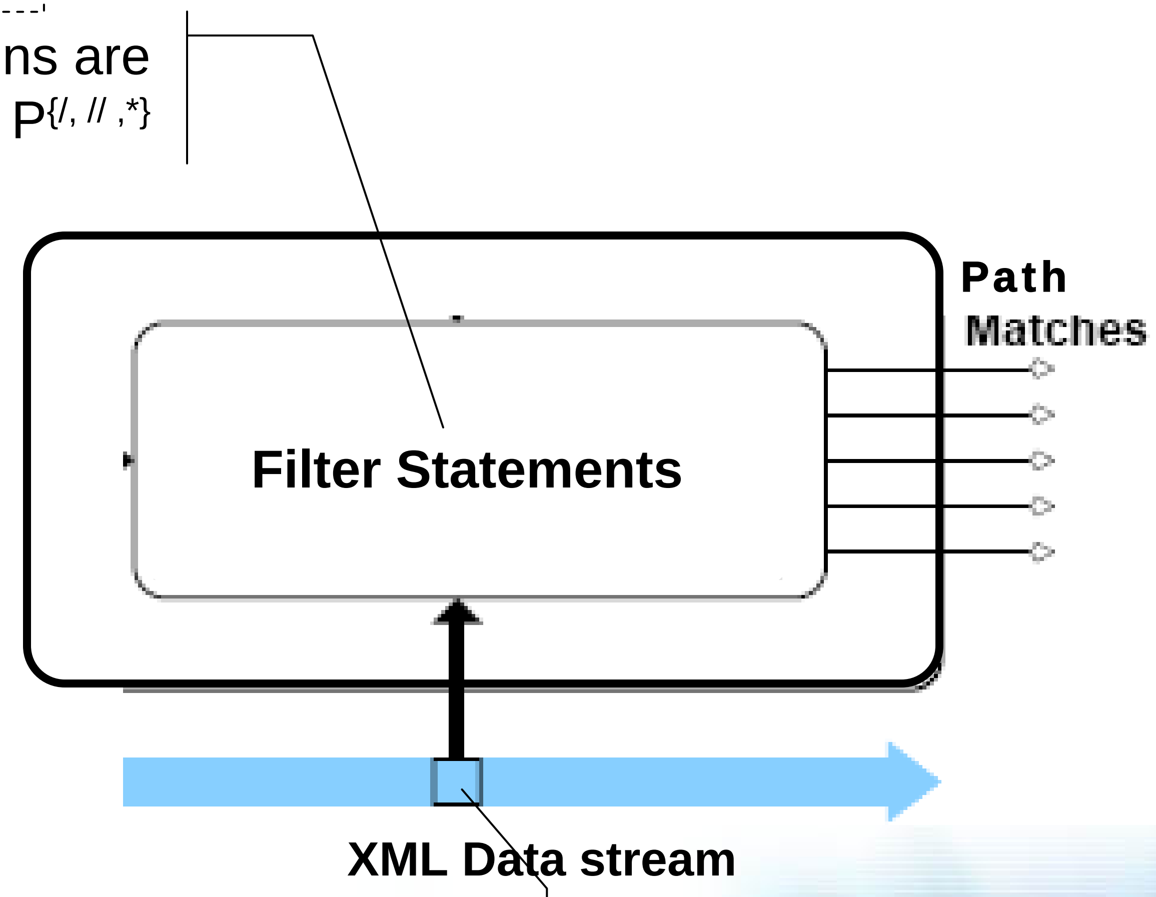
- large number of filter statements
- high throughput

Assumptions

$//a//b/c$

$/a/*/c.$

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



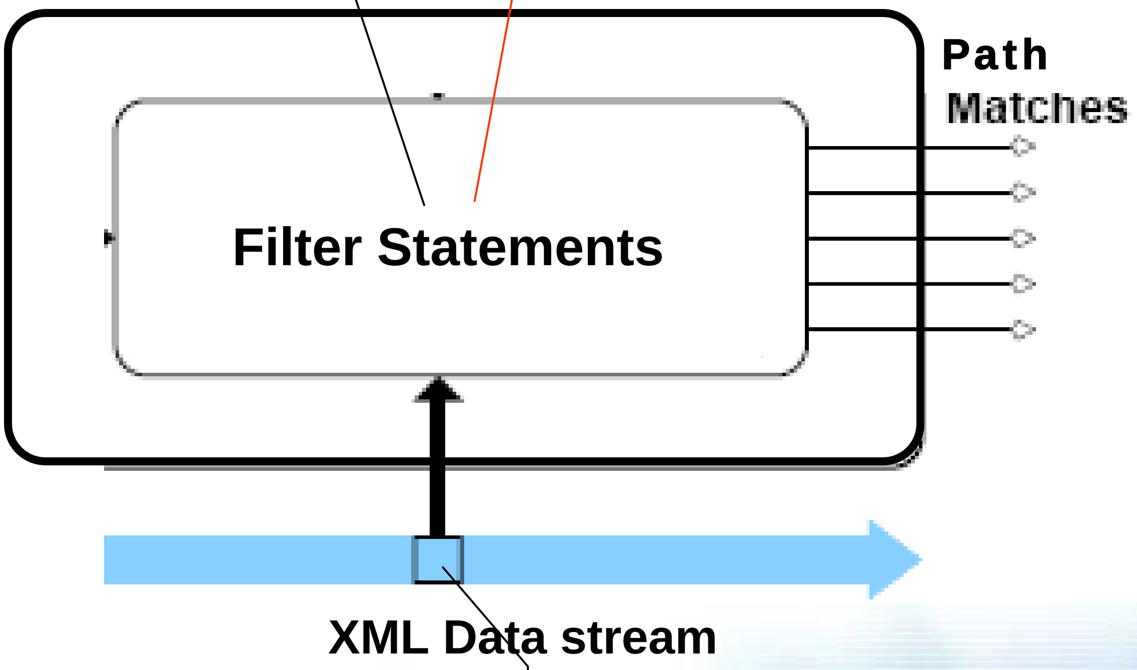
Messages are in some XML format

XML Path Filtering

$//a//b/c$
$/a/*/c.$

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

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



Messages are in some XML format

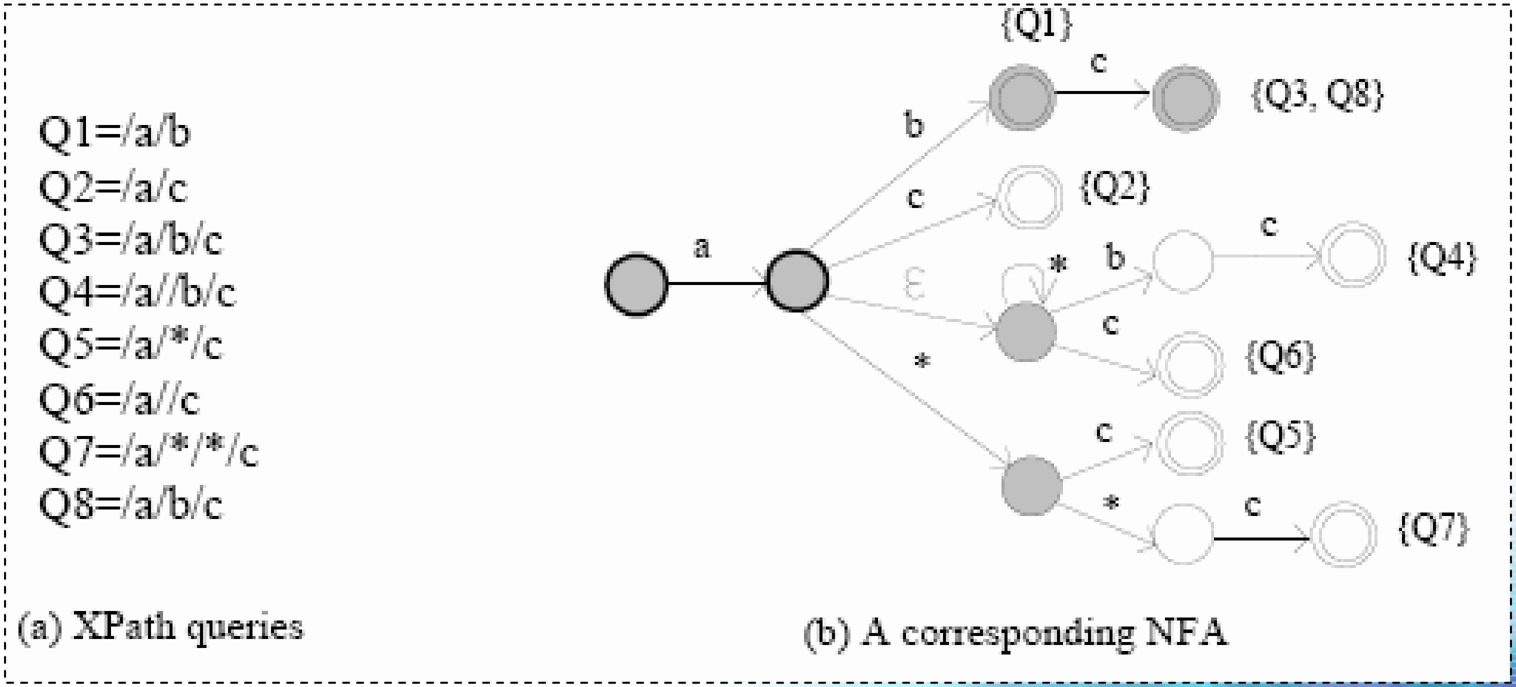
Approach I: Finite Automata

- 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



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

(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

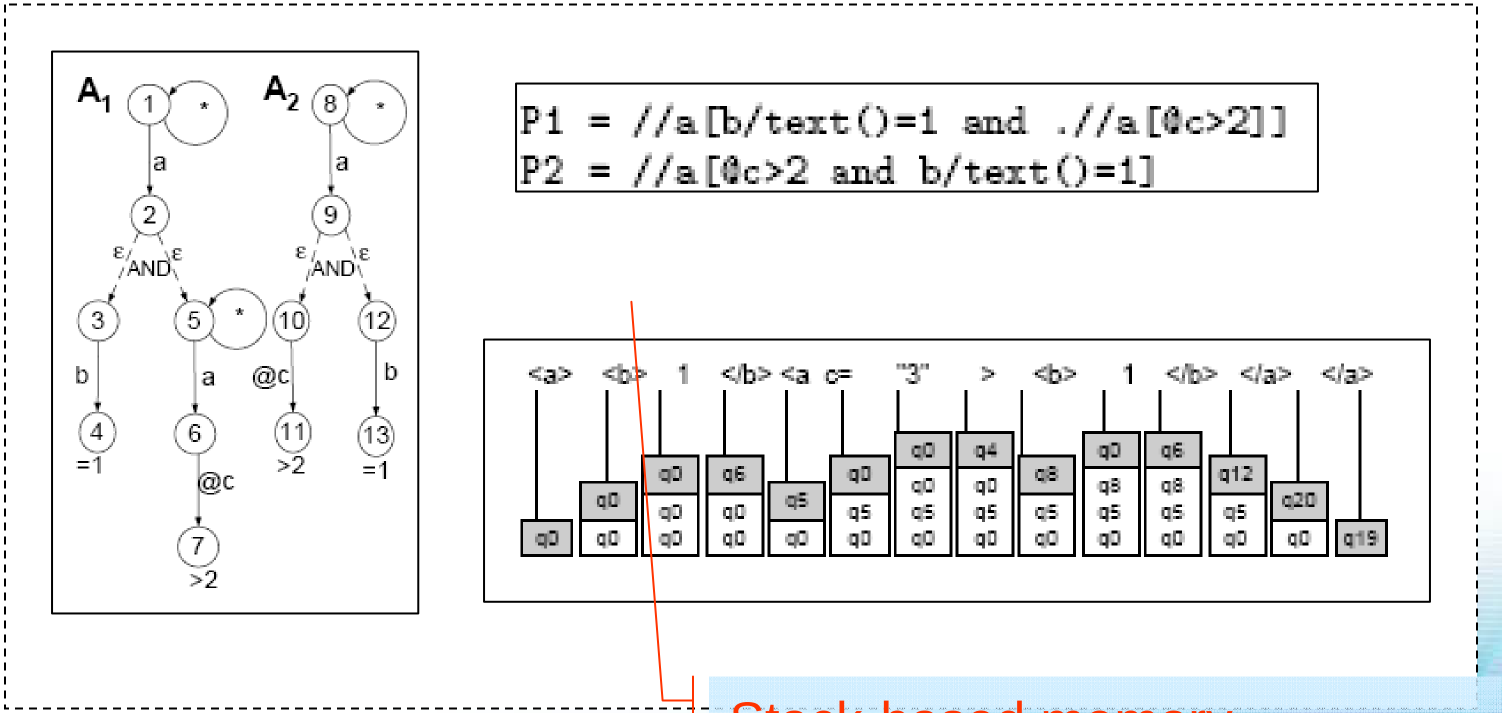
- Each data node causes a state transition in the underlying FA representation of the filters

- 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.]



```
P1 = //a[b/text()=1 and .//a[@c>2]]
P2 = //a[@c>2 and b/text()=1]
```

Figure taken from XPush

Stack-based memory management for the states

Push Down Automata

- Use a stack to organize the data&states XPush [Gupta et al.],SPEX [Olteanou et al.],XSQ [Peng et al.]

- 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

Empowered by Innovation NEC

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

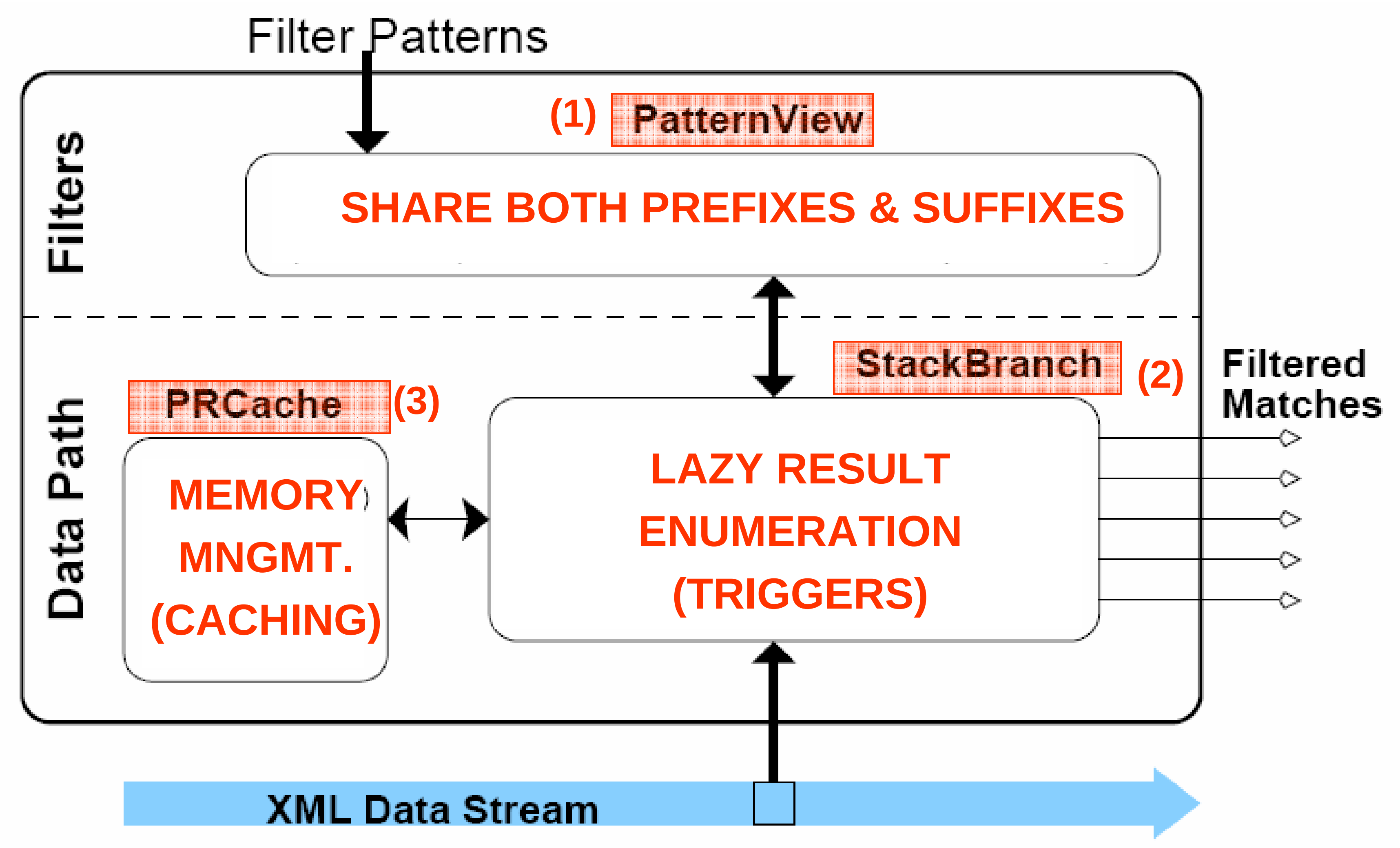
Observations

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

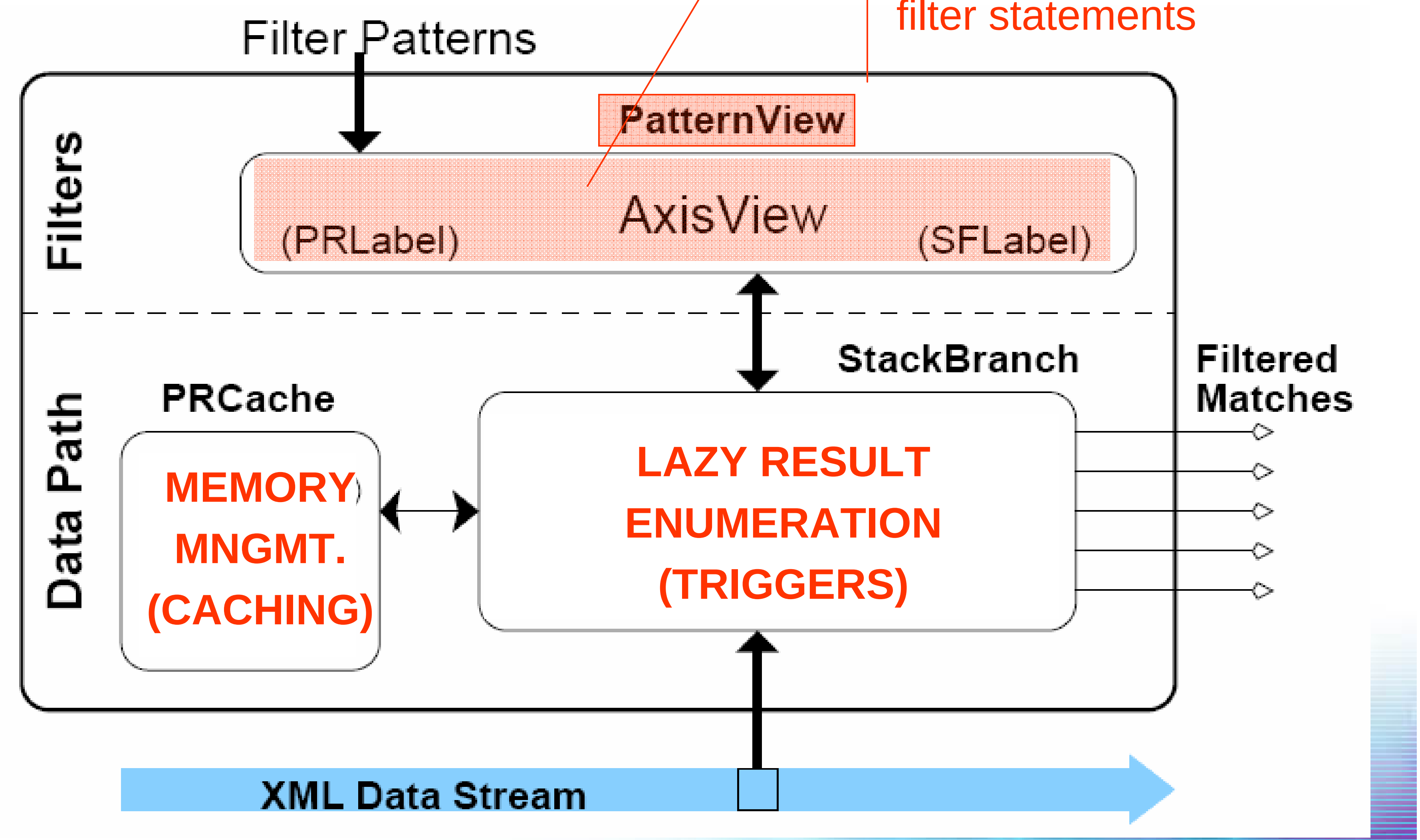
- Active state enumeration is costly
  - *can we have a compact representation and lazy (triggered) enumeration?*

- 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

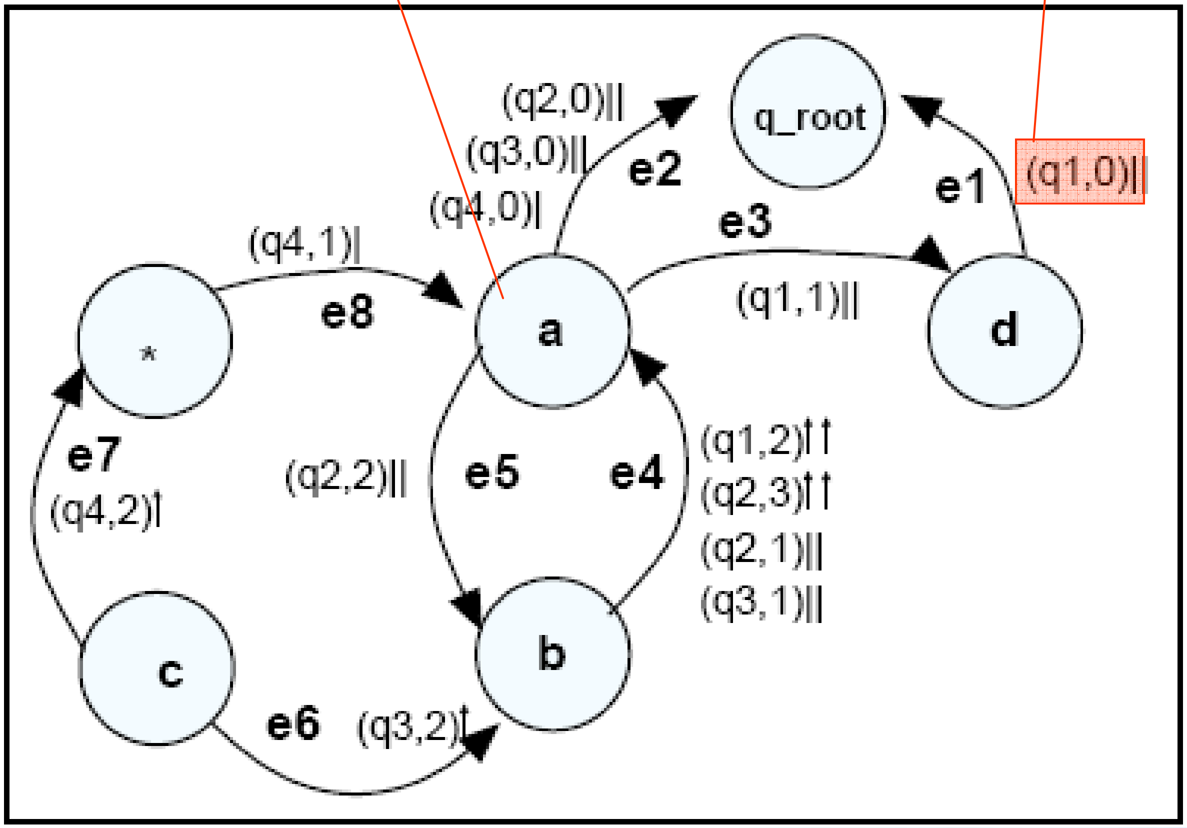
AxisView (blueprint for filters)

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

AxisView

One node per symbol

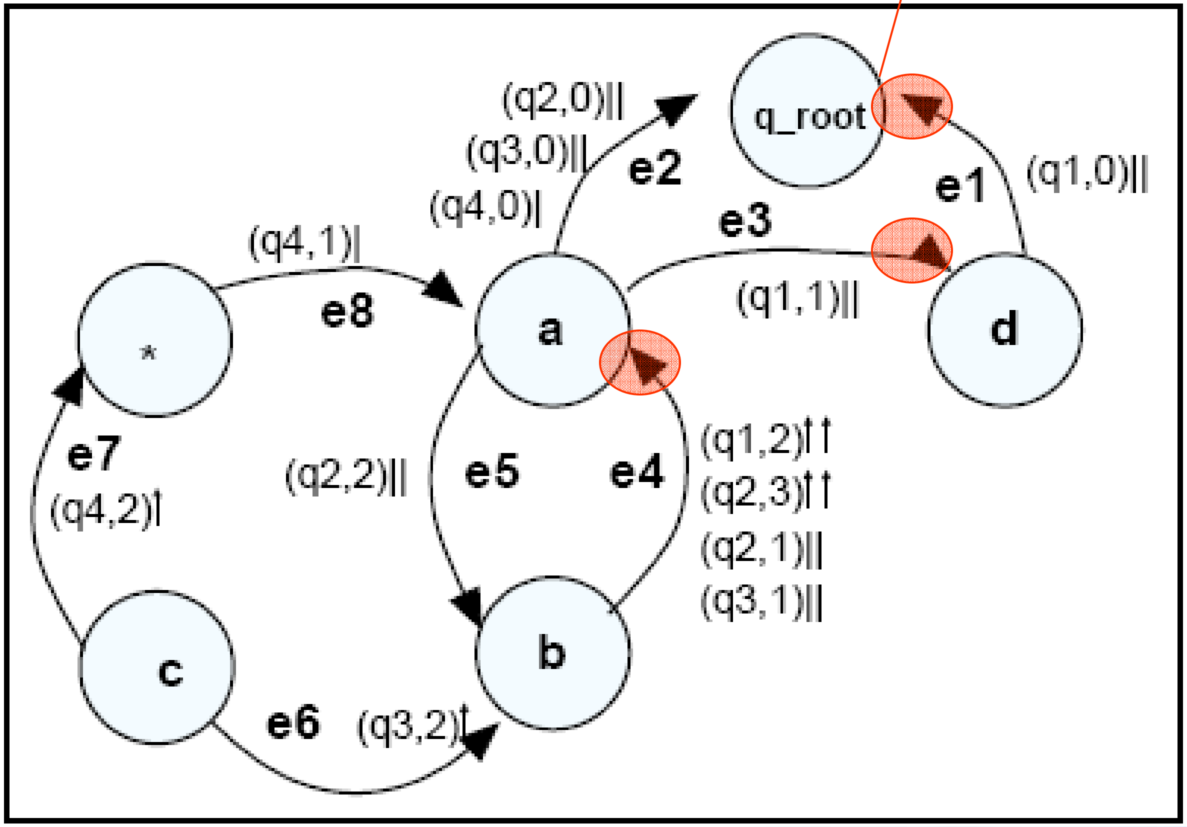
One “assertion” per query step



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

AxisView

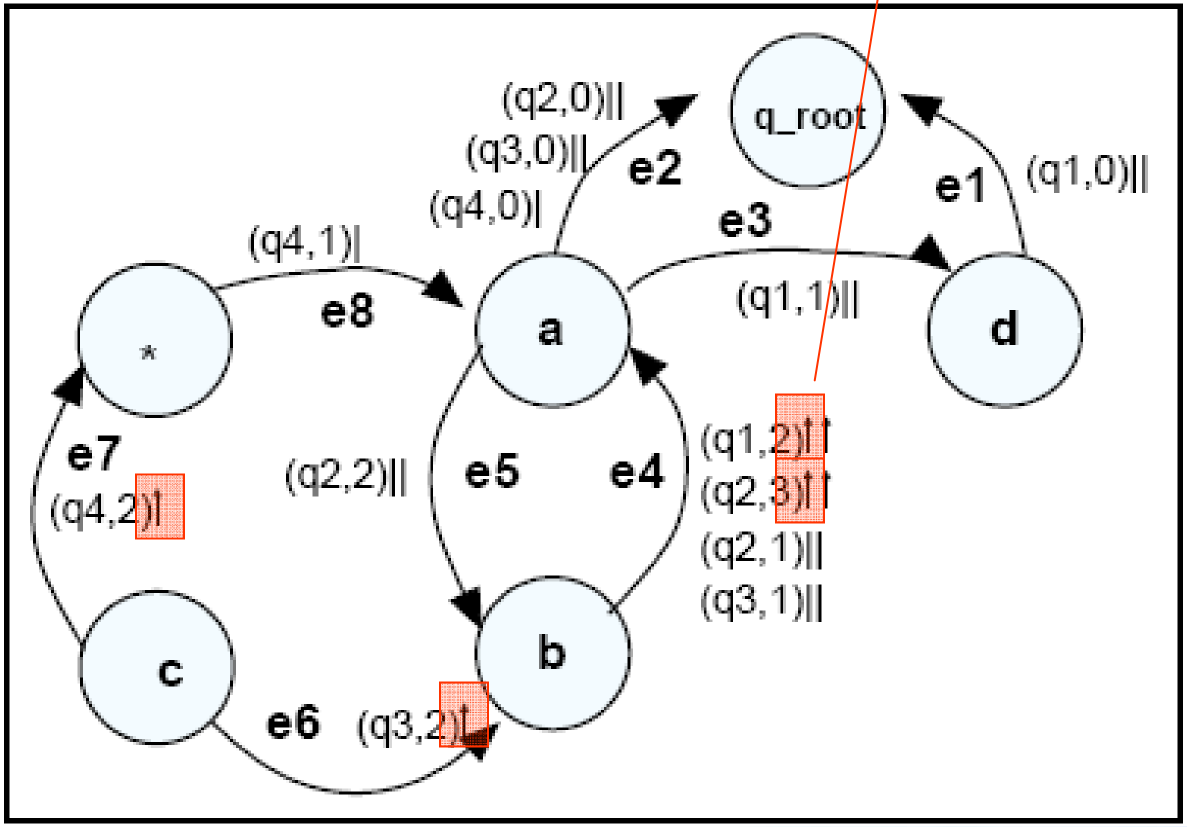
Edges from leaves to root



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

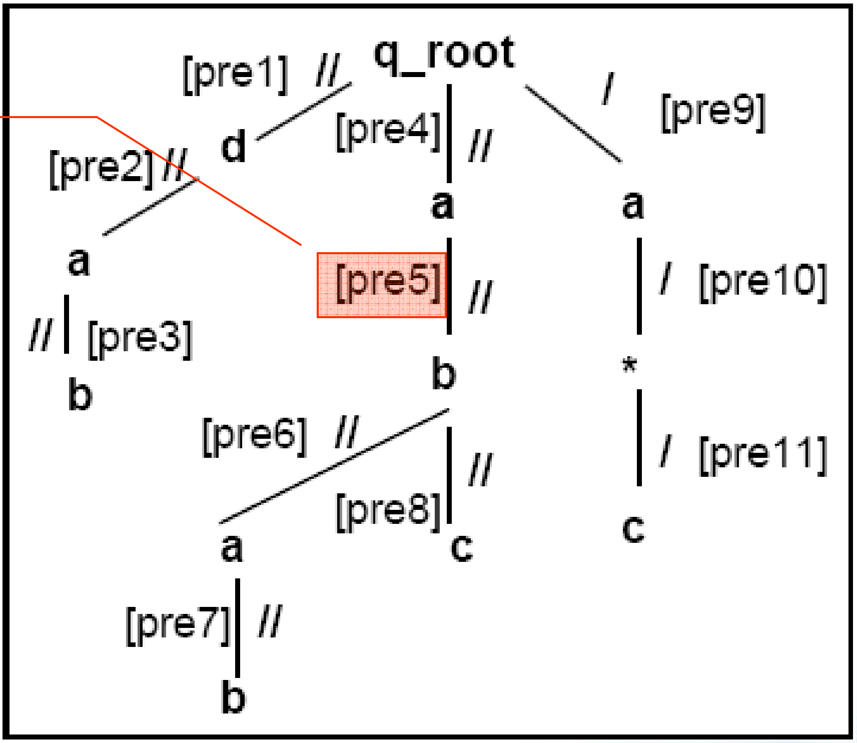
AxisView

One trigger per query



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

PRLabel-tree (optional, trie)

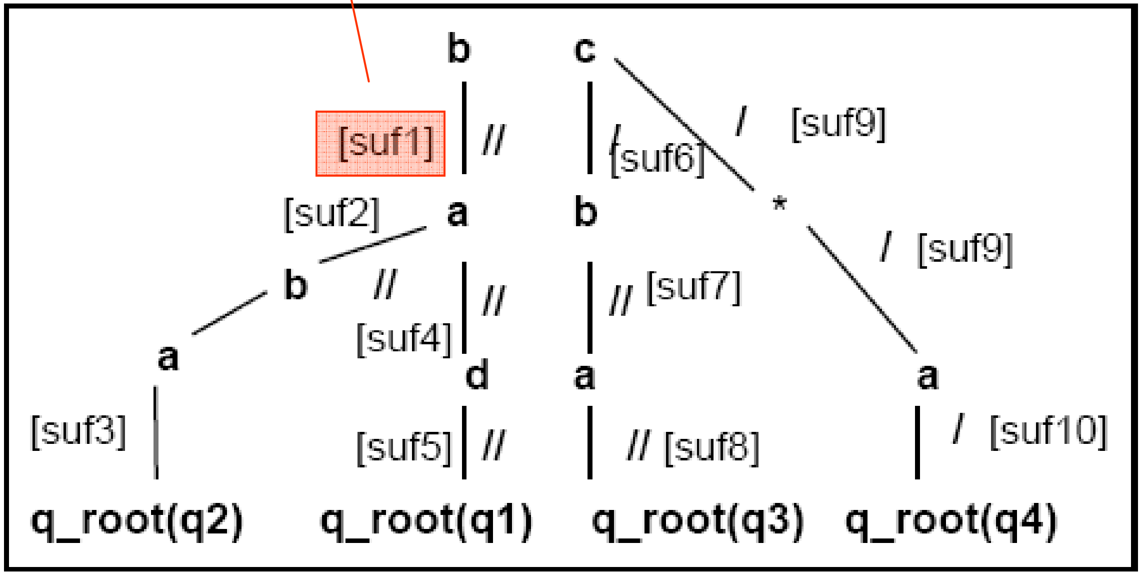


Prefix labels

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

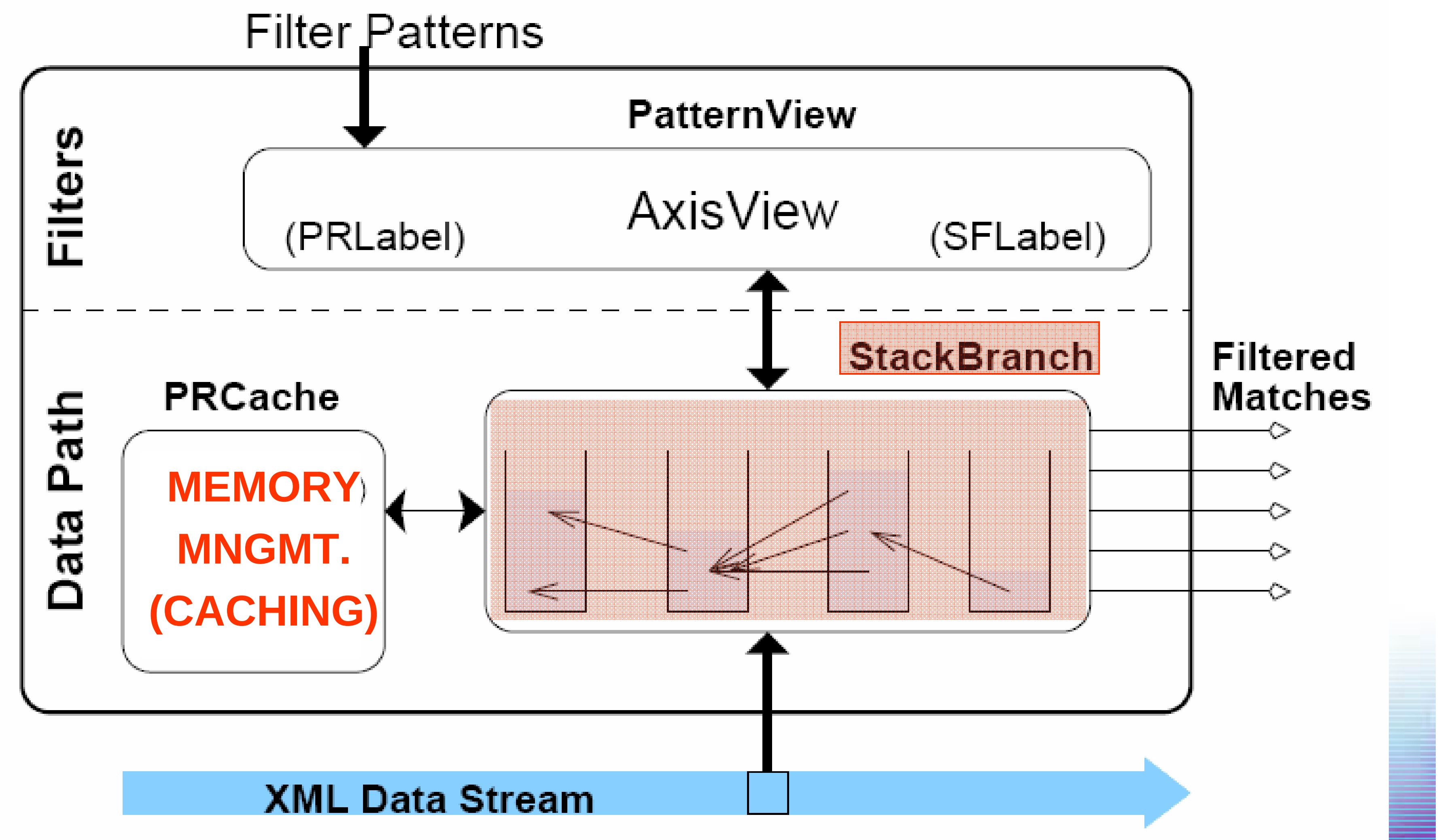
SFLabel-tree (optional, trie)

Suffix labels

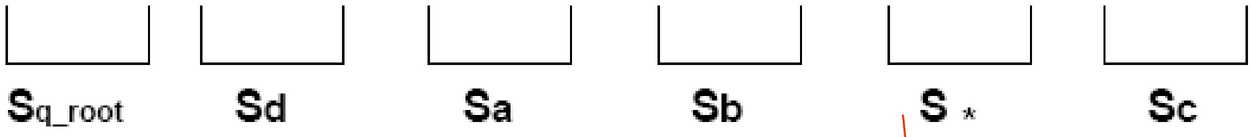


$$\{q_1 = //d//a//b, q_2 = //a//b//a//b, q_3 = //a//b/c, q_4 = /a/*/c\}$$

AFilter


Filter Patterns
PatternView
AxisView
(PRLabel)
(SFLabel)
Filters
StackBranch
Filtered Matches
PRCache
MEMORY
MNGMT.
(CACHING)
Data Path
XML Data Stream

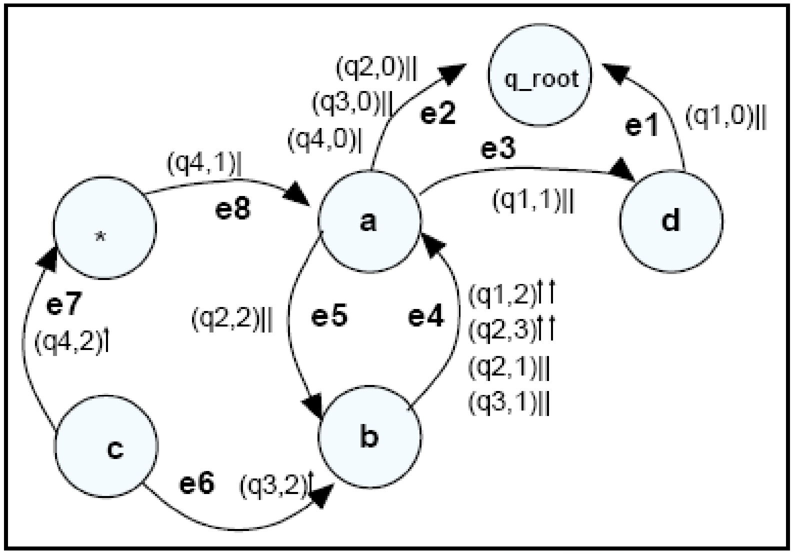
StackBranch (path encoding)



S_{q_root} S_d S_a S_b S_* S_c

Empty

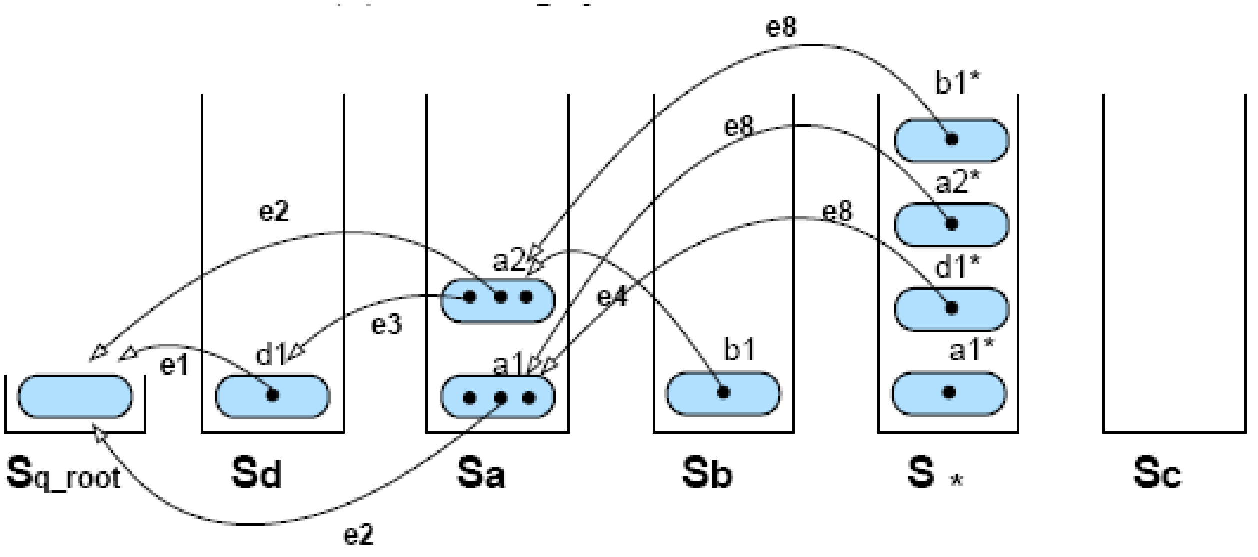
One stack per symbol



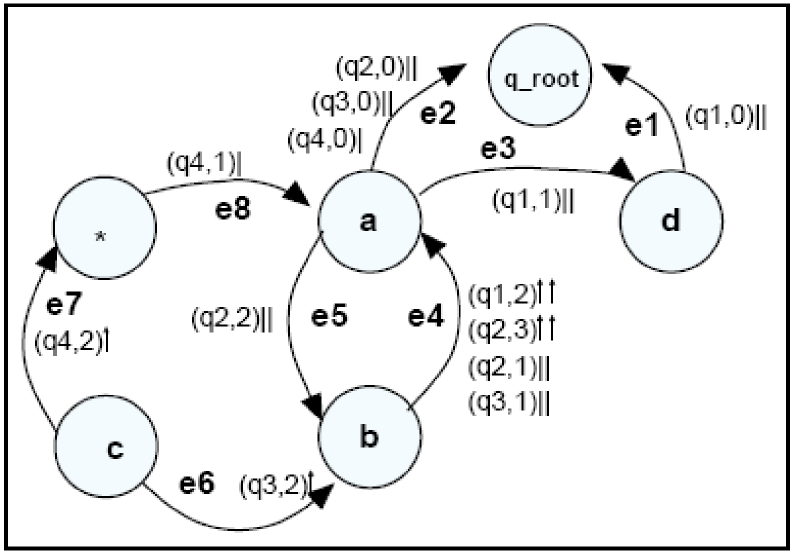
Conceptually similar to PathStack [Bruno et al.] for structural joins

Encodes the hierarchical information in the current path segment compactly

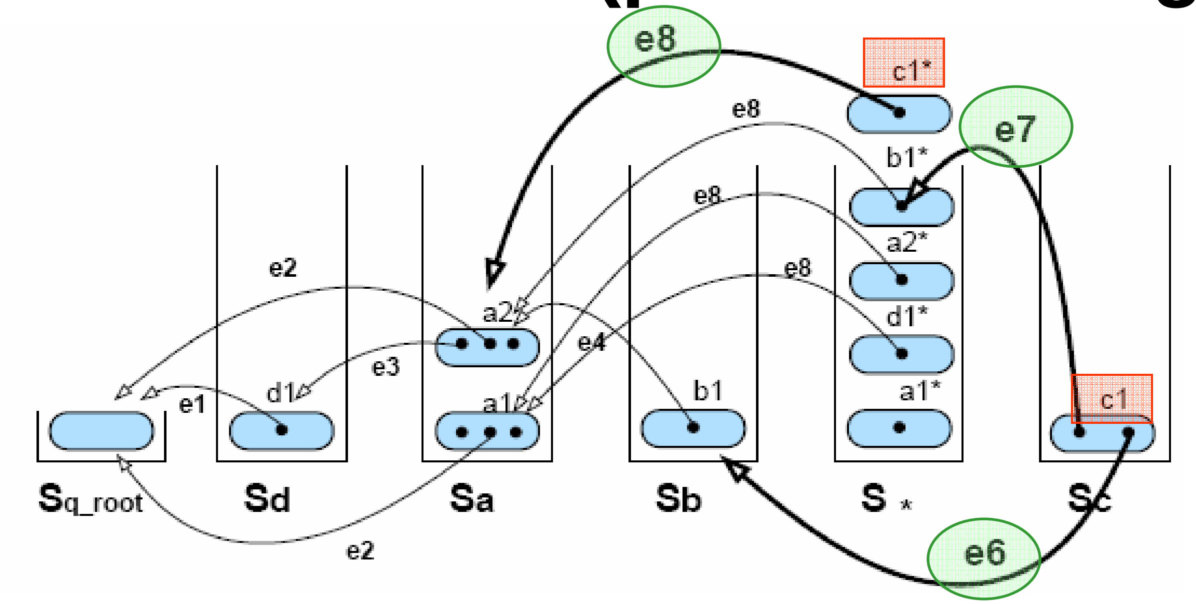
StackBranch (path encoding)

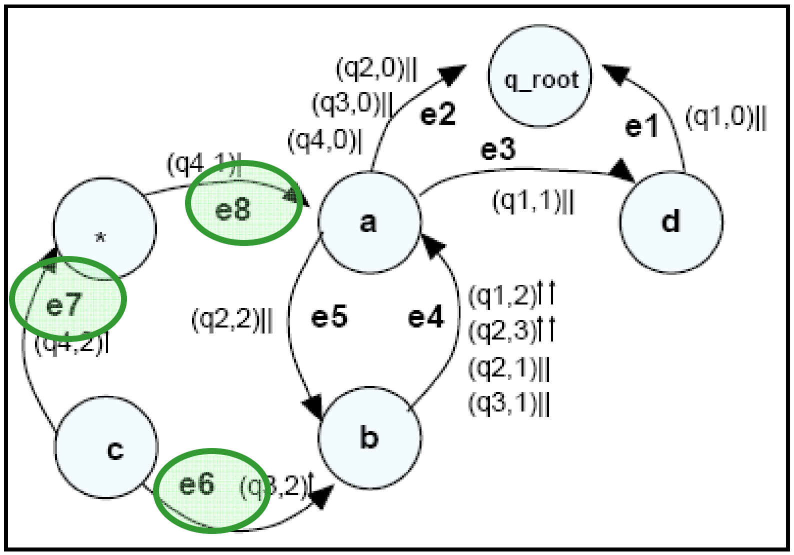


After <a><d><a><b>



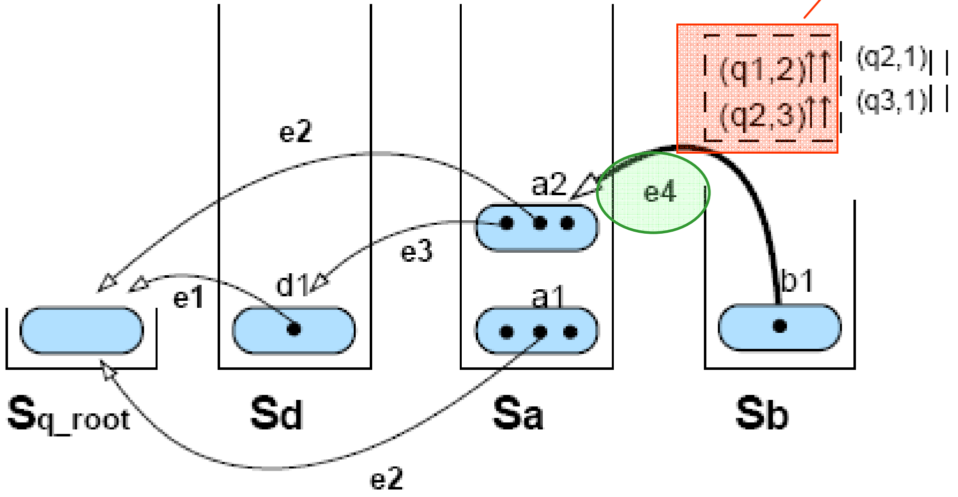
StackBranch (path encoding)



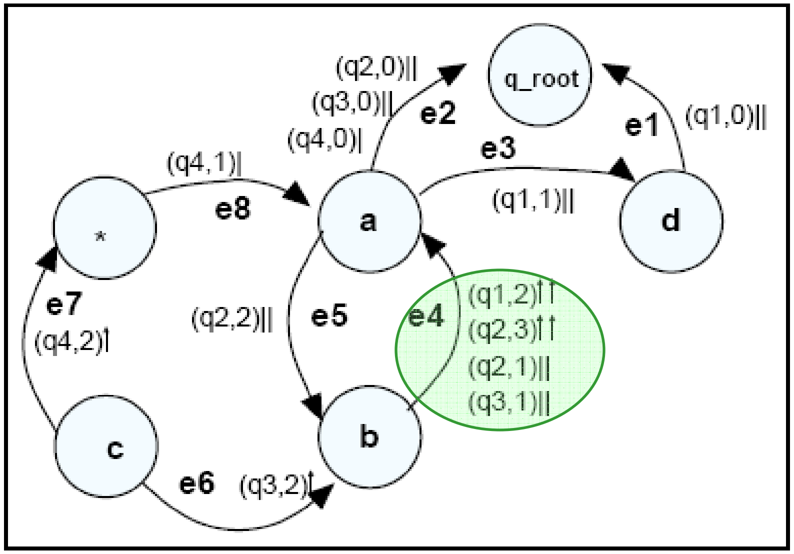


After <a><d><a><b><c>

Triggering

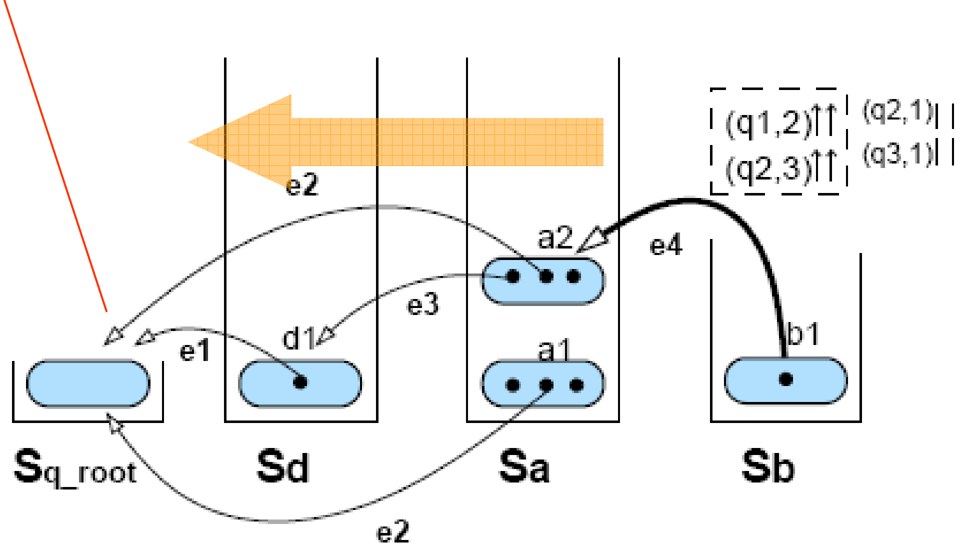


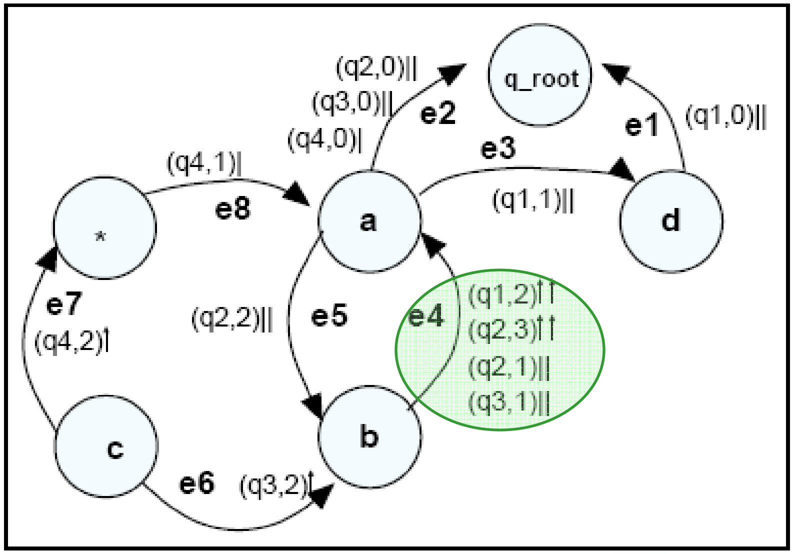
Triggering query steps



Triggering & following

Can we reach the root stack?

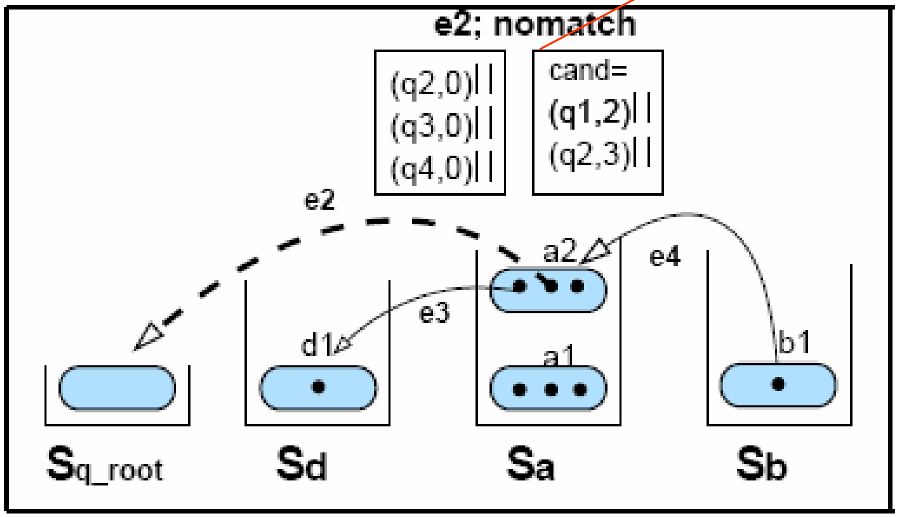




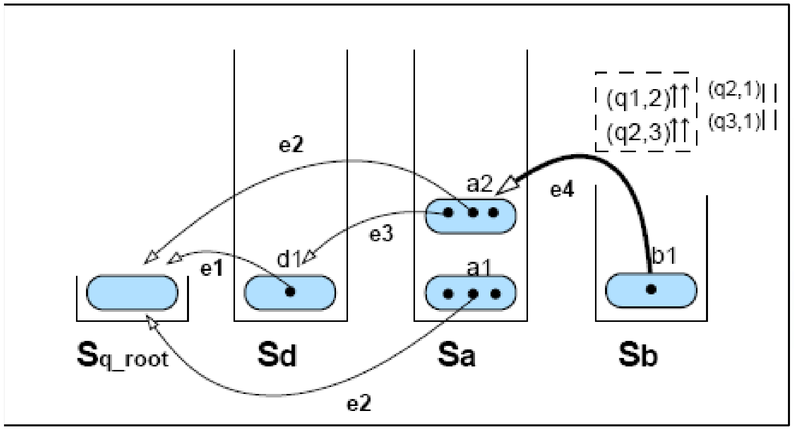
Advantages:

- edges are never followed if no triggering
- benefits from tighter selectivity at the leaves
- edges are followed in a clustered manner

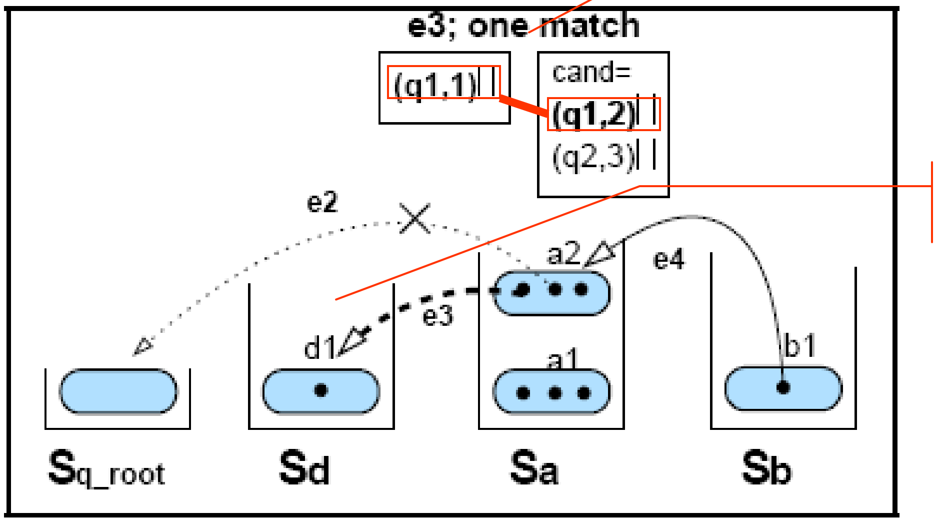
Clustered edge following



Hash Join

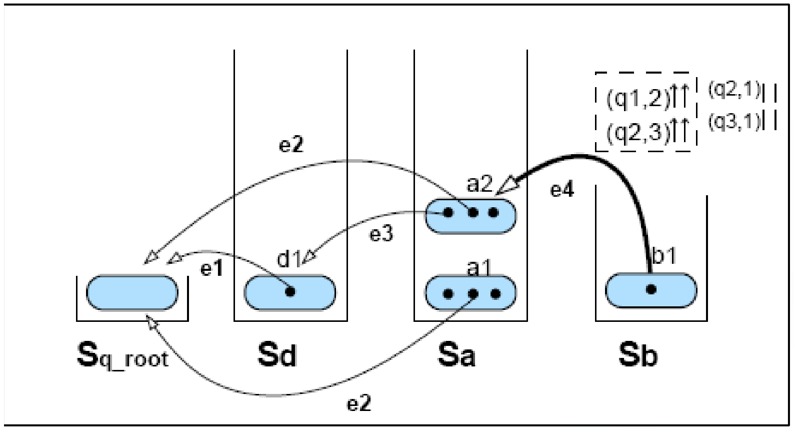


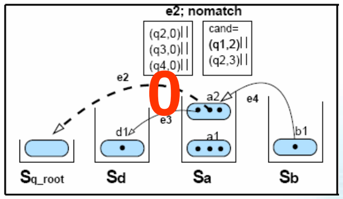
Clustered edge following



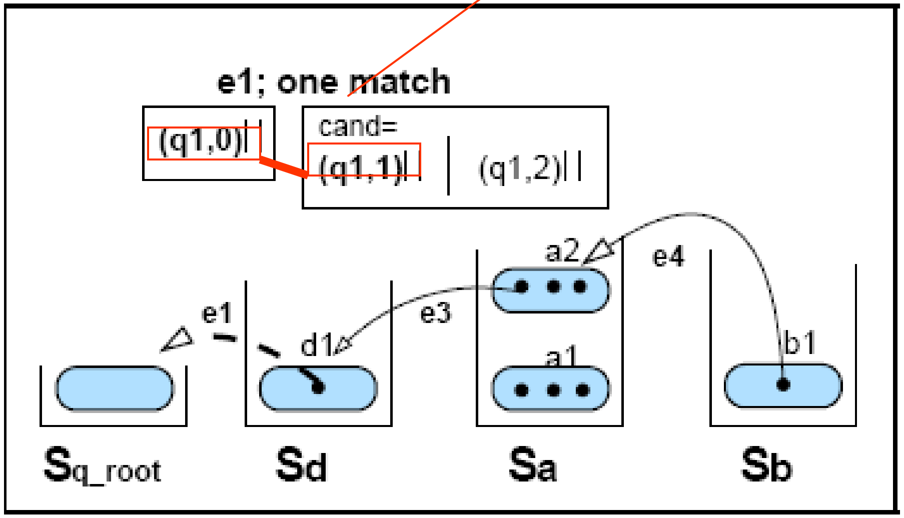
Hash Join

Continue follow

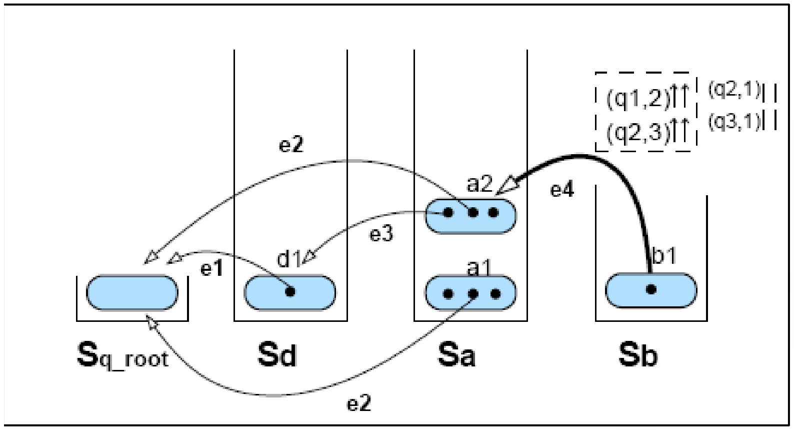


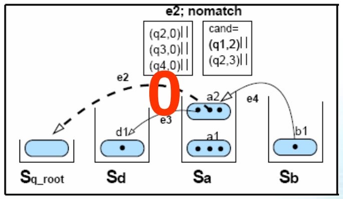


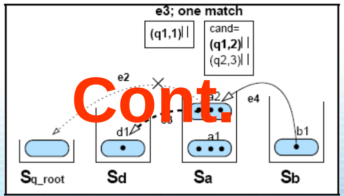
Clustered edge following



Hash Join

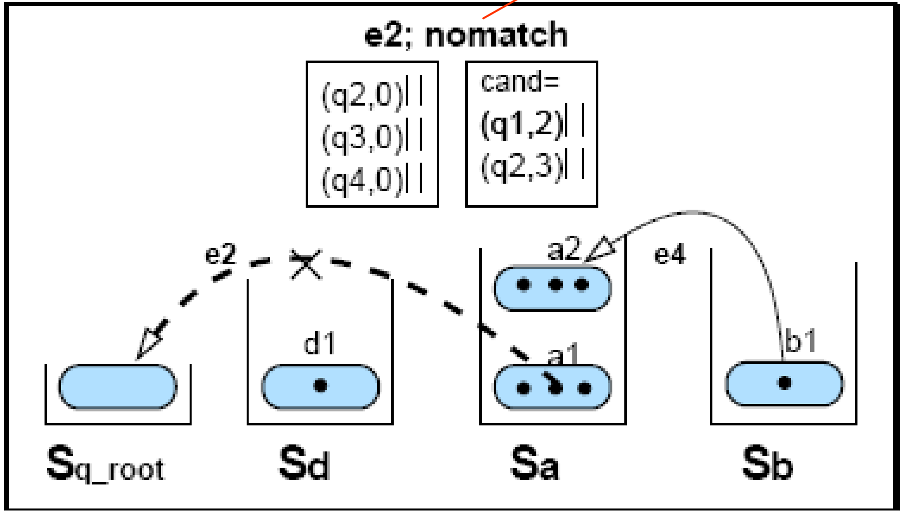


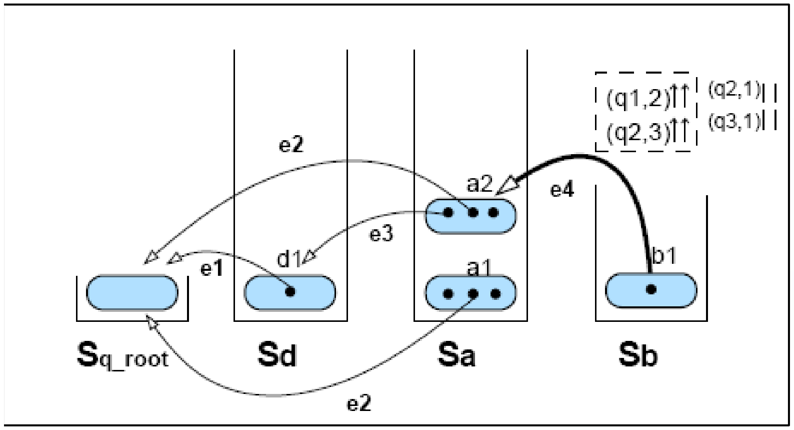


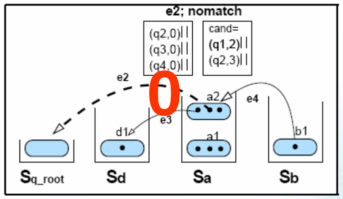


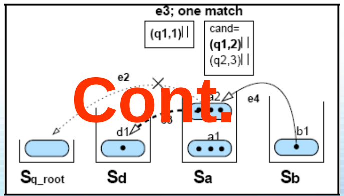
Clustered edge following

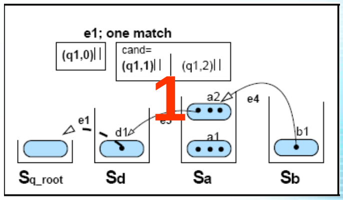
Hash Join





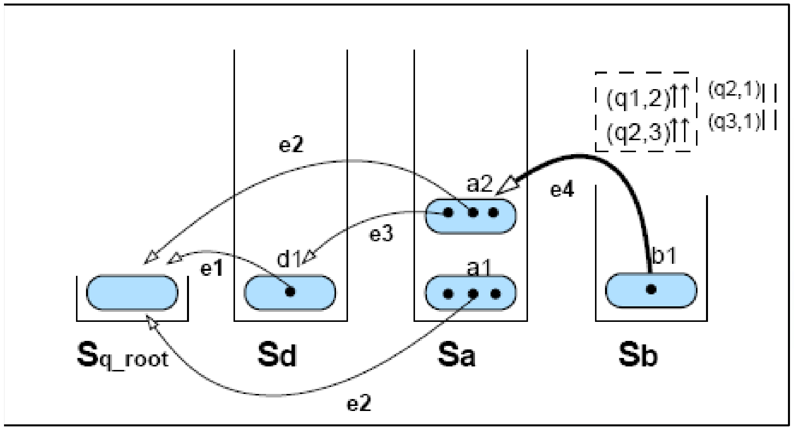


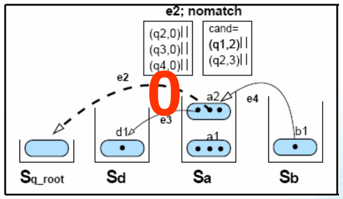


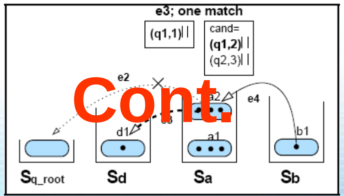


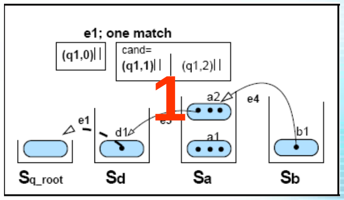
Clustered edge following

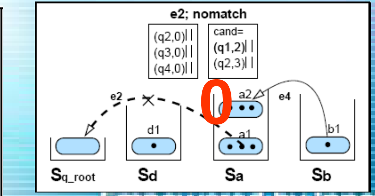
- one path match found -











Clustered edge following

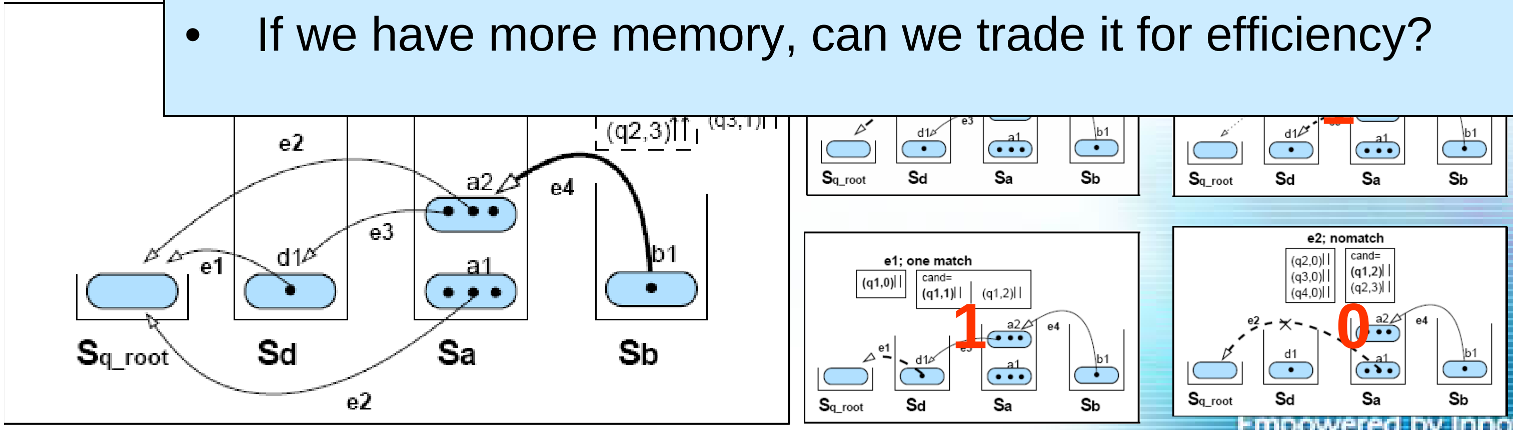
- one path match found -

Note:

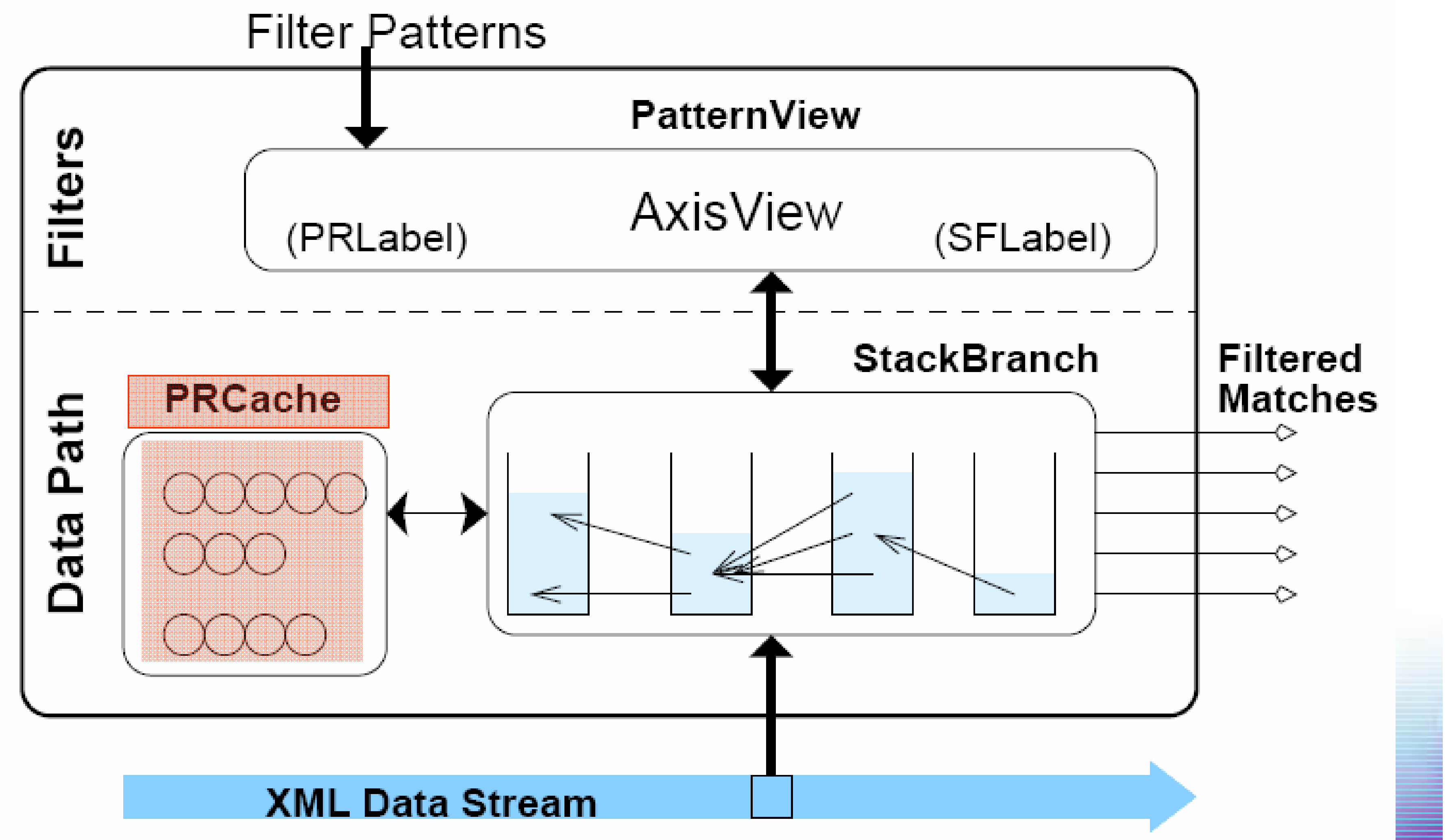
- Memory usage linear (in the depth of the data tree)

Challenge:

- If we have more memory, can we trade it for efficiency?

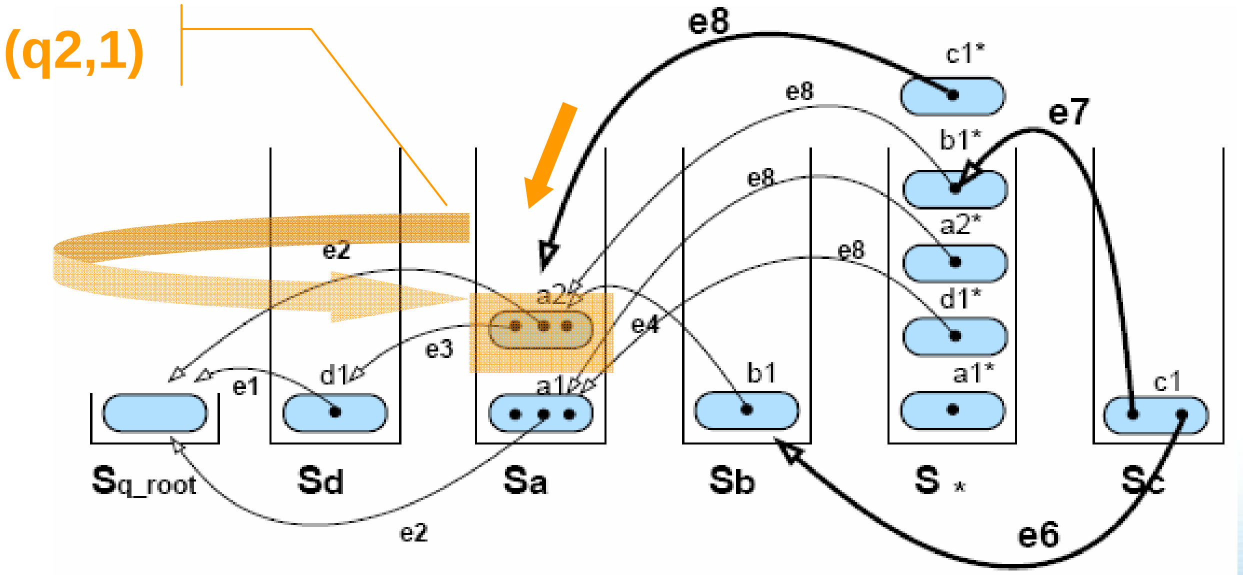


AFilter


Filter Patterns
Filters
PatternView
AxisView
(PRLabel)
(SFLabel)
Data Path
PRCache
StackBranch
Filtered Matches
XML Data Stream

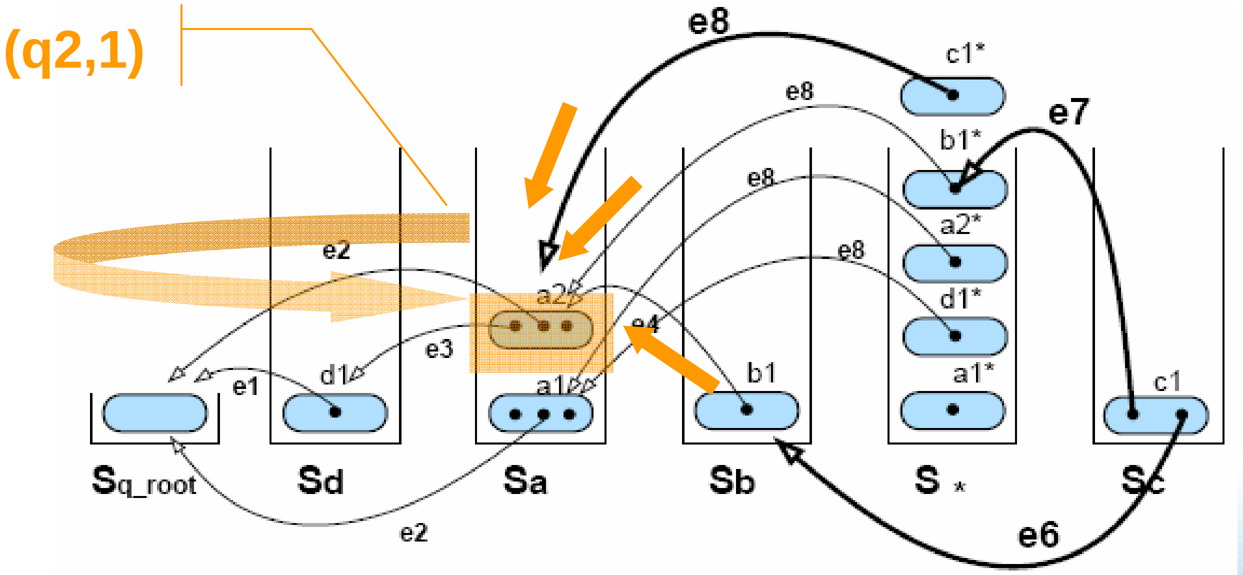
Prefix caching / PRCache

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



Prefix caching / PRCache

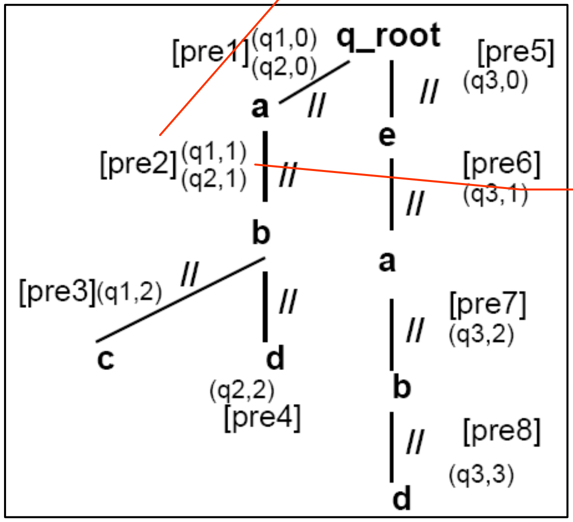
- Observation:
  - repeated evaluations of the same candidate assertion at a node will always lead to the same result.



Prefix caching / PRCache

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

PRLabel tree



Alt 1. Index and cache partial results against the prefix labels

- prevents redundant traversals
- enables prefix sharing

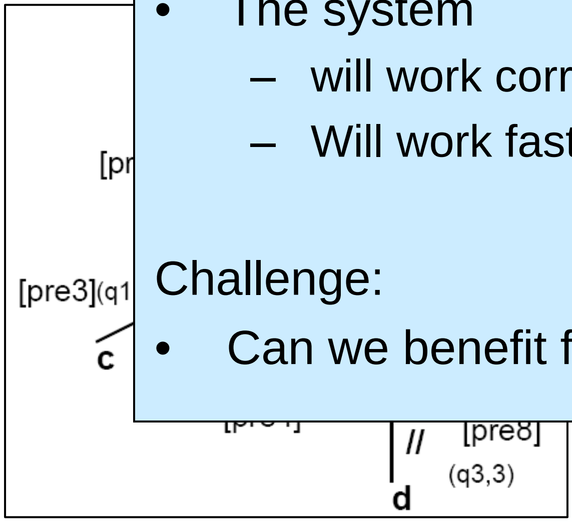
Alt 2. Index and cache only the failures

- 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



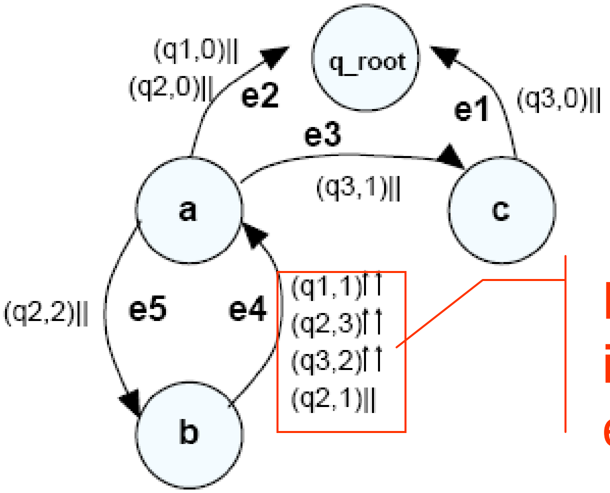
Note:

- 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?

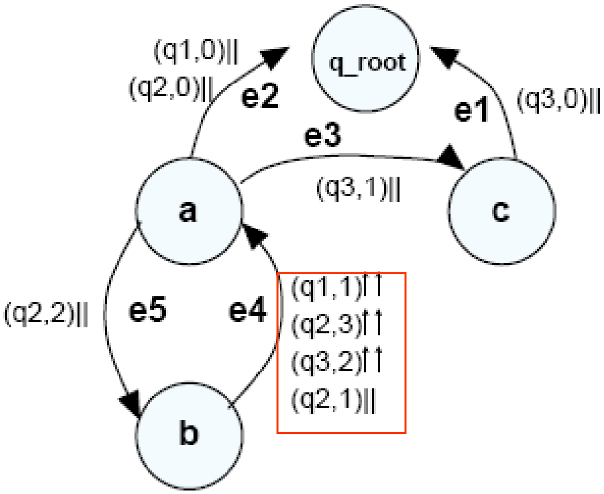
Suffix compressed traversals

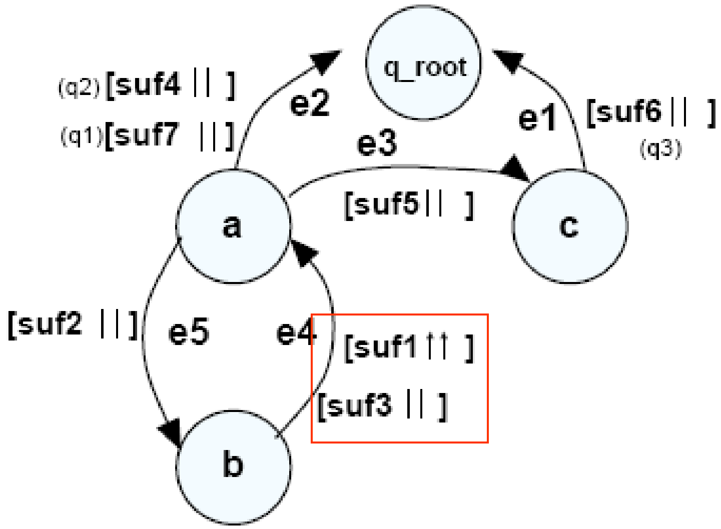


Large number of query steps **increases** the hash join cost during edge traversal

$(q_1 = //a//b, q_2 = //a//b//a//b,$ **and** $q_3 = //c//a//b)$

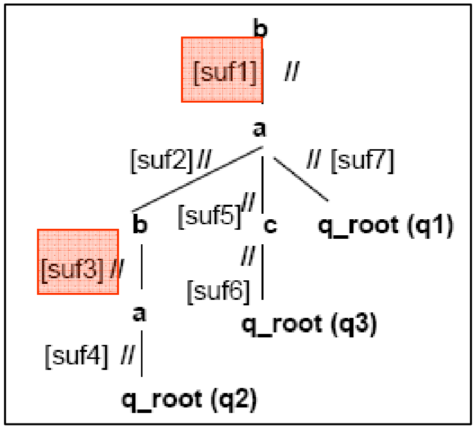
Suffix compressed traversals



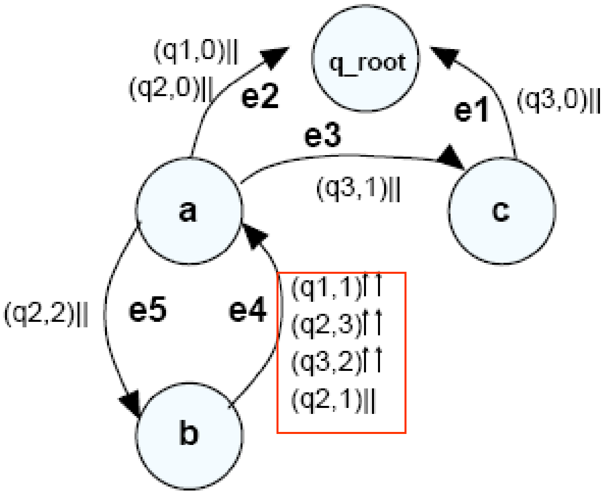


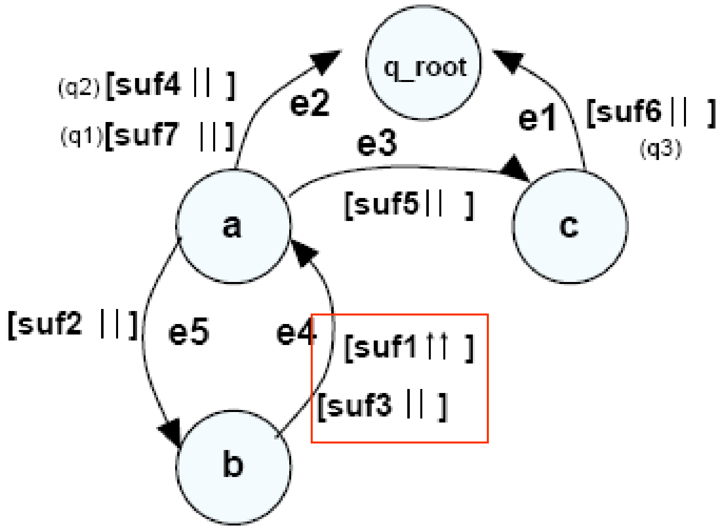
$(q_1 = //a//b, q_2 = //a//b//a//b, \textbf{and } q_3 = //c//a//b)$

SFLabel tree



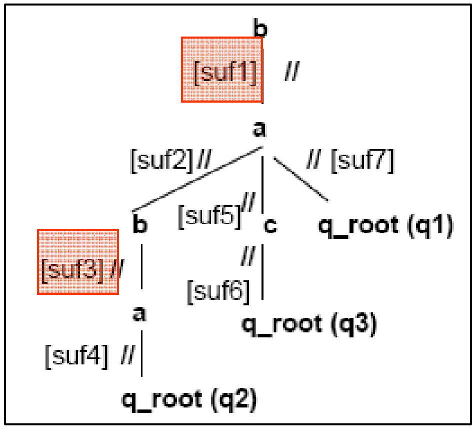
Suffix compressed traversals





$(q_1 = //a//b, q_2 = //a//b//a//b, \text{ and } q_3 = //c//a//b)$

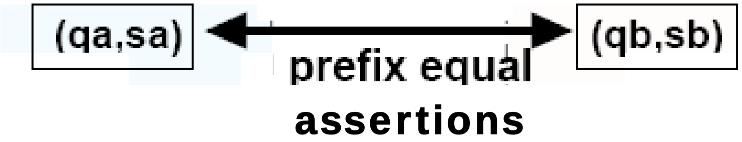
SFLabel tree



Problem:

- Prefix caching and suffix clustering are not entirely compatible.

Overlaps in Prefix/Suffix labels

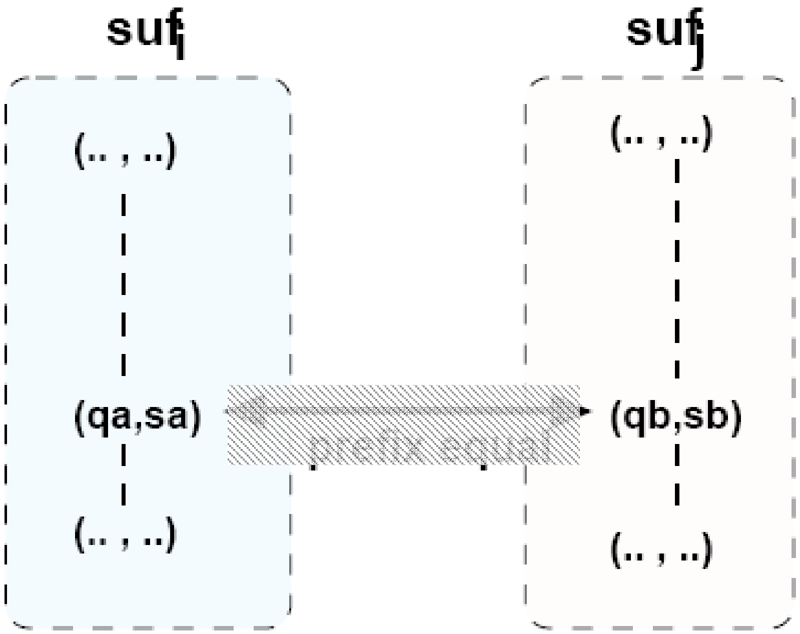


(qa,sa)

(qb,sb)

prefix equal assertions

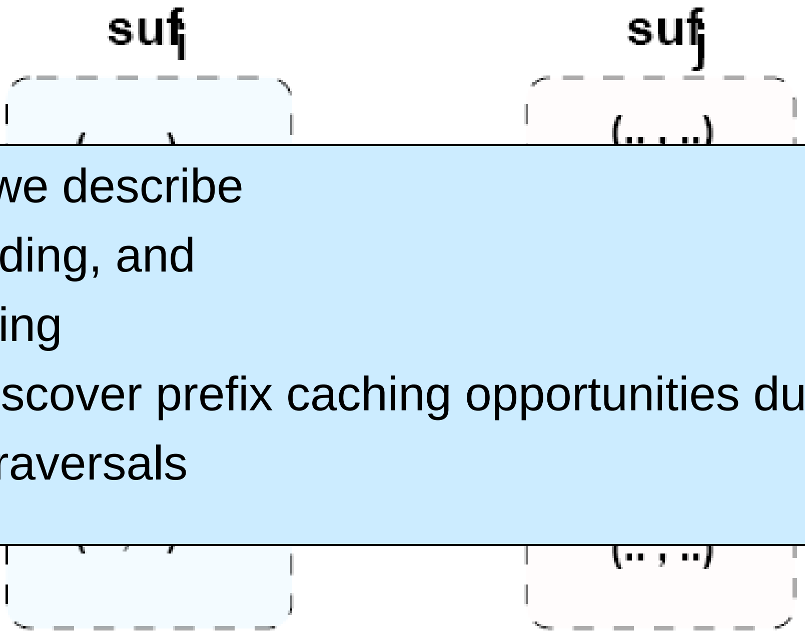
Overlaps in Prefix/Suffix labels



Problem:

- Use of suffix labels (instead of individual assertions) may **hide** prefix caching opportunities)

Overlaps in Prefix/Suffix labels



In the paper, we describe

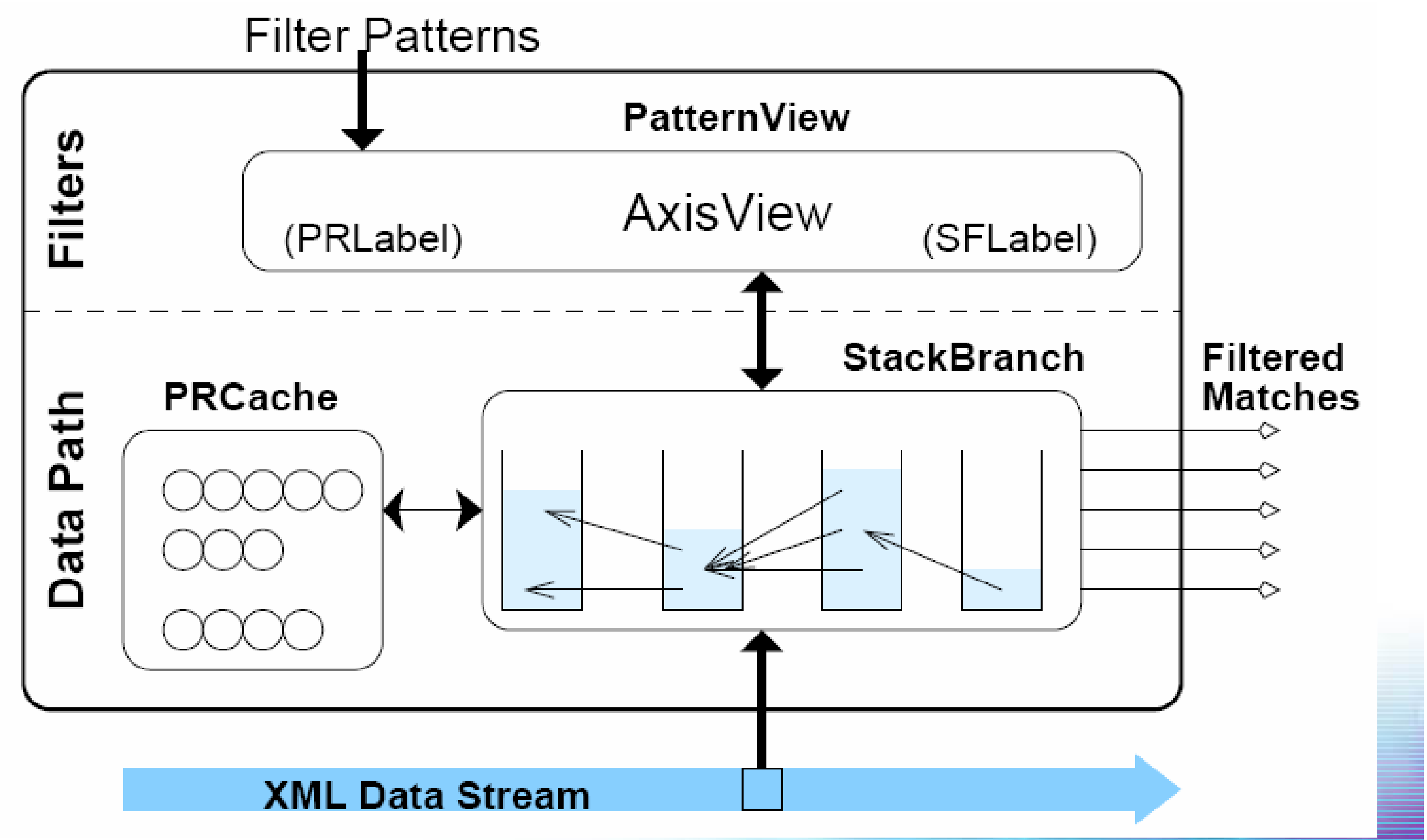
- early unfolding, and
- late unfolding

schemes to discover prefix caching opportunities during suffix compressed traversals

Problem:

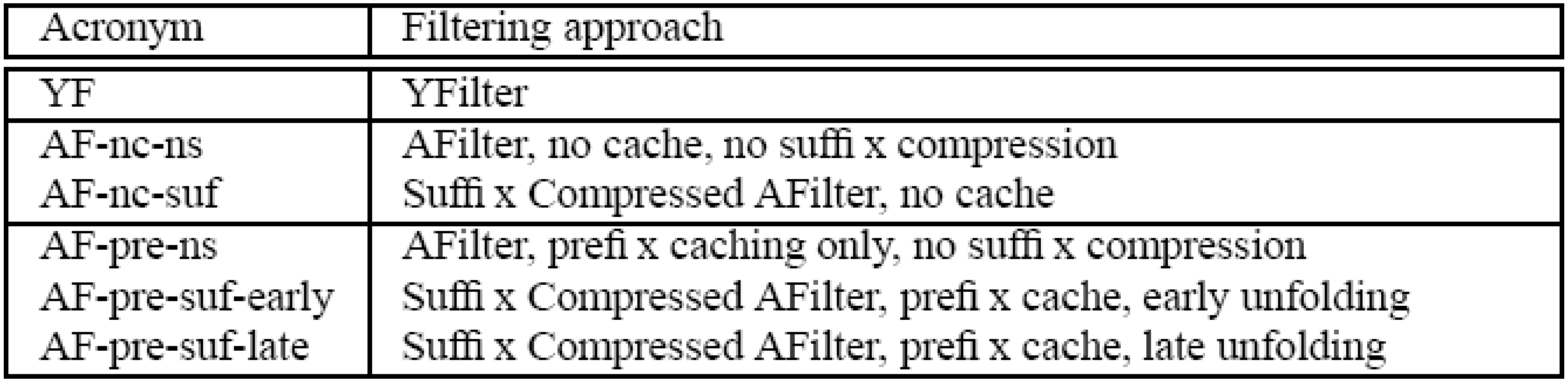
- Use of suffix labels (instead of individual assertions) may hide prefix caching opportunities)

AFilter


Filter Patterns
Filters
PatternView
AxisView
(PRLabel)
(SFLabel)
Data Path
StackBranch
Filtered Matches
PRCache
XML Data Stream

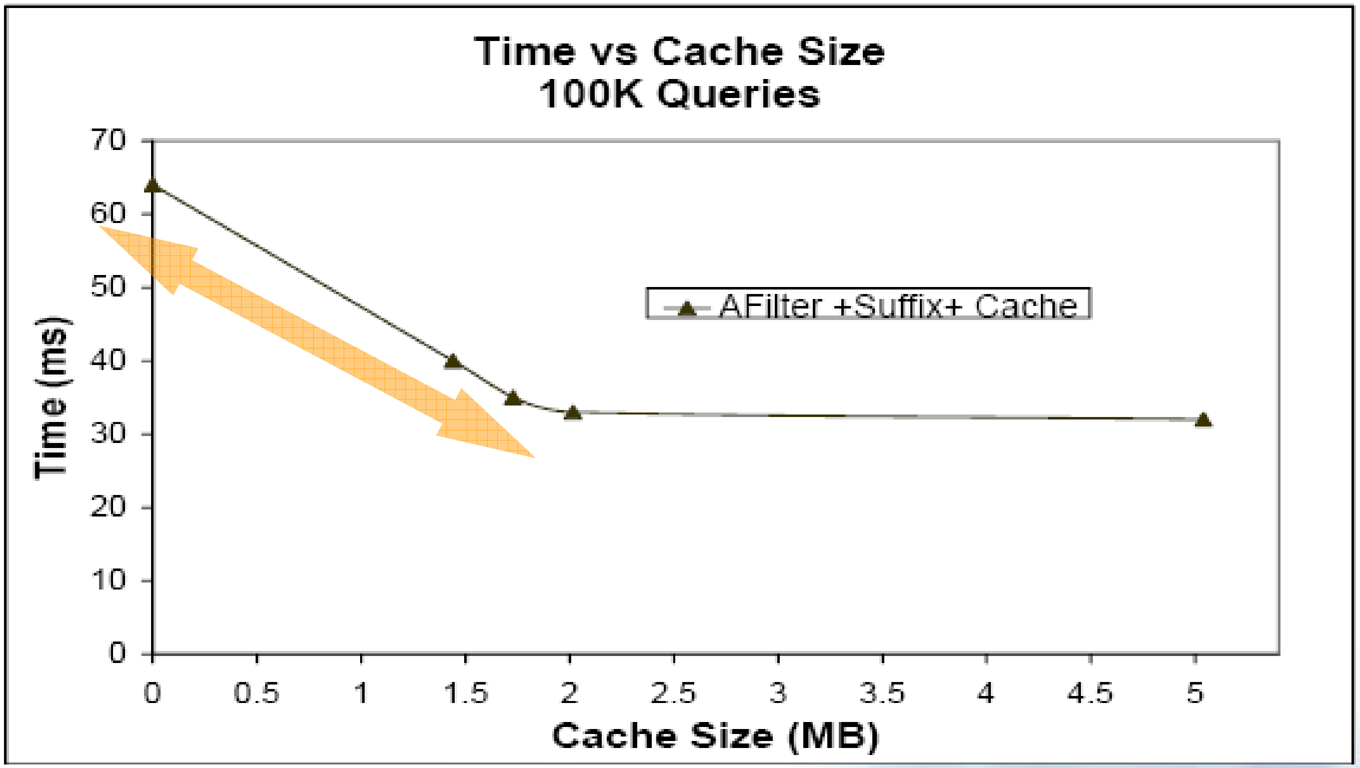
Experiment Setup

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

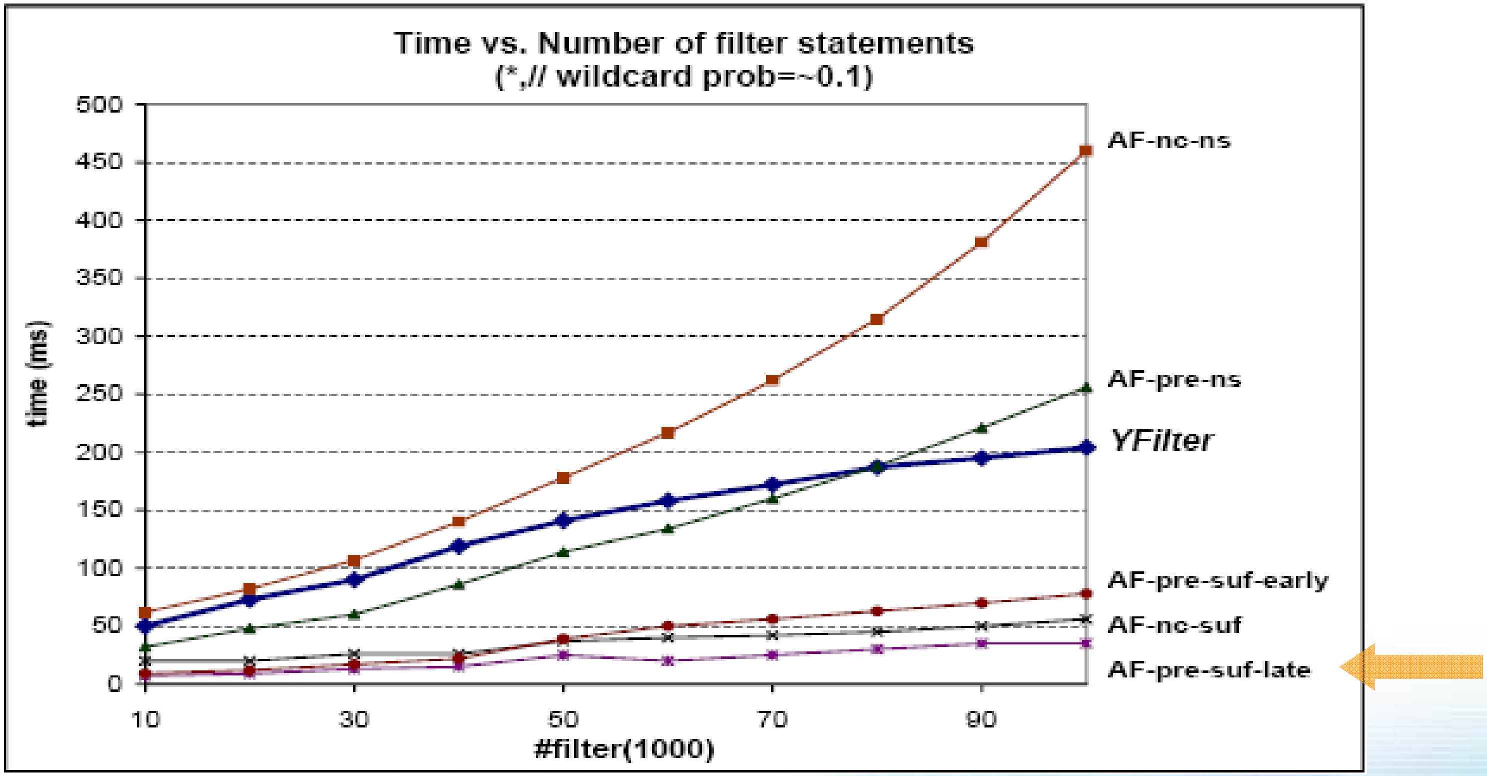


Acronym	Filtering approach
YF	YFilter
AF-nc-ns	AFilter, no cache, no suffix compression
AF-nc-suf	Suffix Compressed AFilter, no cache
AF-pre-ns	AFilter, prefix caching only, no suffix compression
AF-pre-suf-early	Suffix Compressed AFilter, prefix cache, early unfolding
AF-pre-suf-late	Suffix Compressed AFilter, prefix cache, late unfolding

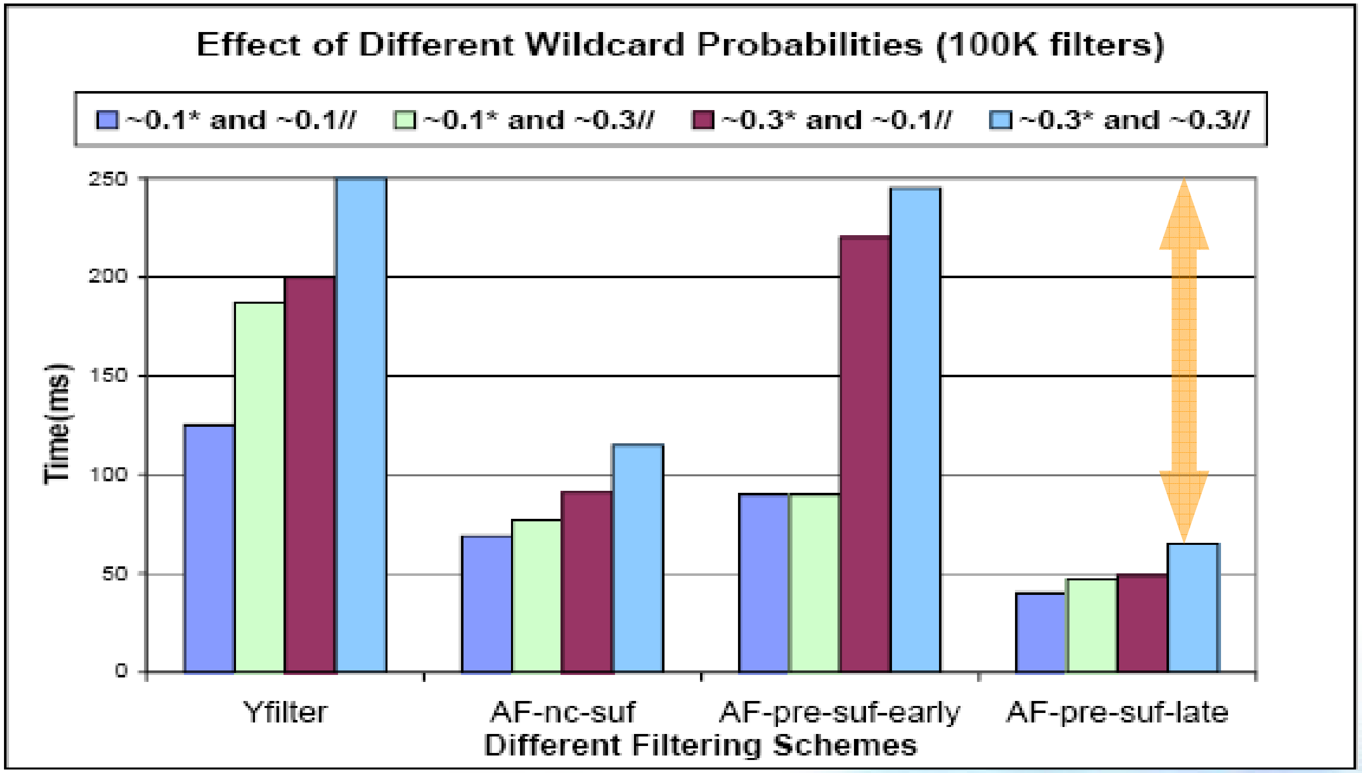
Experiment results



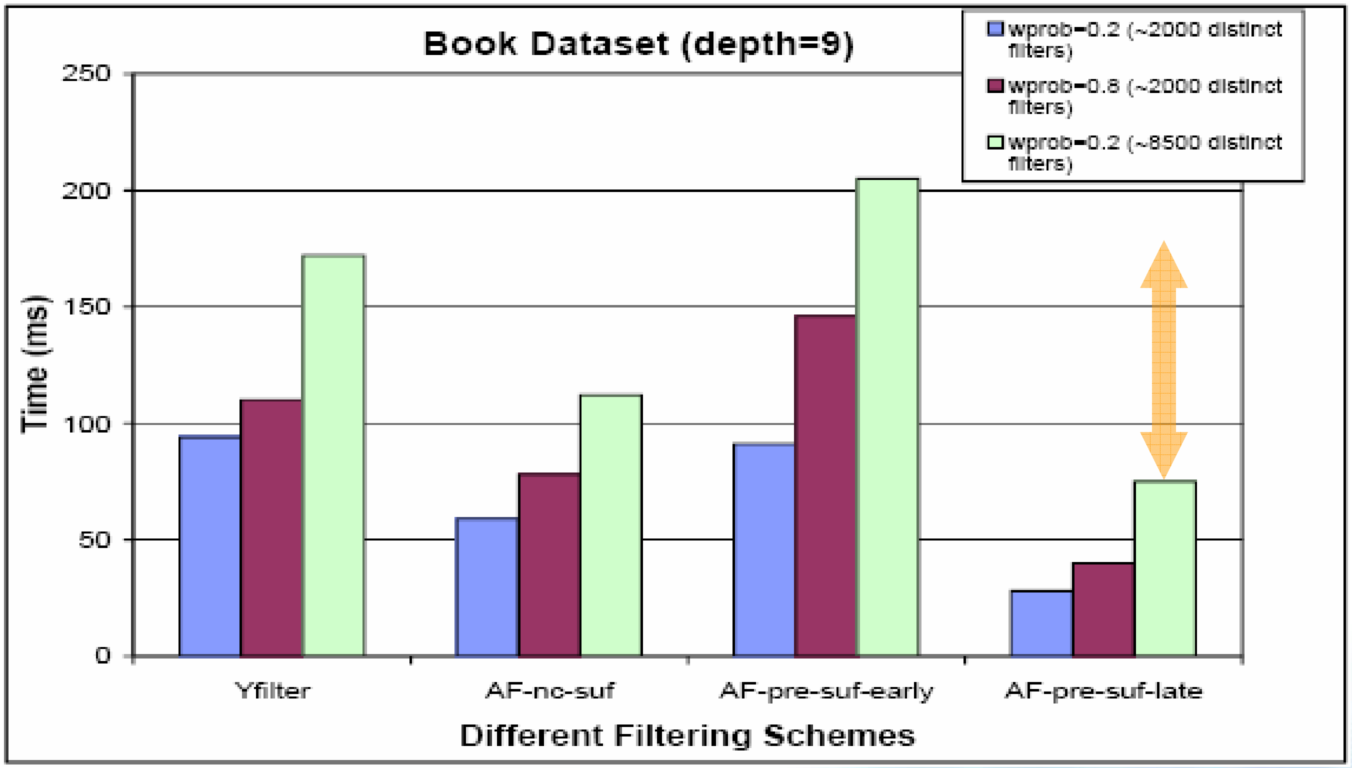
Experiment results



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.