STREAM PROCESSING OF XPATH QUERIES WITH PREDICATES (2003)

Presenters: Jens Emil Gydesen and Martin Bjeldbak Madsen
Paper authors: Gupta, A. and Suciu, D.

June 4, 2015

TABLE OF CONTENTS

INTRODUCTION

Introduction: scenario & motivation

- Incoming stream of XML (SAX events)
  - How to evaluate **many** XPath filters on this stream?

Defined as the *XML stream processing problem.*

Introduction: scenario & motivation

- Incoming stream of XML (SAX events)
  - How to evaluate **many** XPath filters on this stream?

Defined as the *XML stream processing problem.*

Occurs in many situations

- XML packet routing
- Selective dissemination of information
- Notification systems
- ...

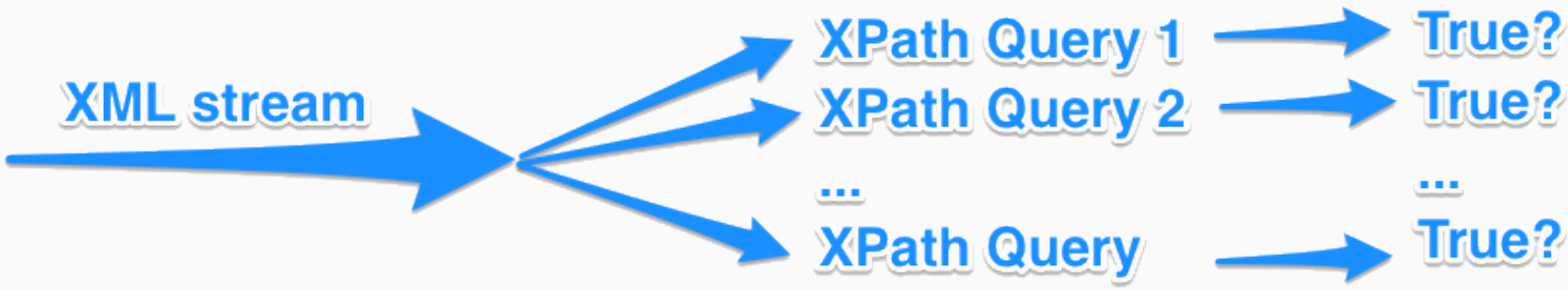
WHAT IS SAX?

SAX is a type of parser implementing the following 5 methods

```
startDocument()
  startElement(a)
    text("3")
  endElement(a)
endDocument()
```

...simply translates to XML document `<a>3</a>`

- Definition:
  - Given a set of XPath filters and a stream of XML documents, compute for each document D, the set of filters that match D
- The number of filters and predicates can be large (hundreds of thousands)
- Parallel / sequential query evaluation not scalable



- Reduce the number of XPath predicate evaluations by
  - Grouping of common predicates

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

  - Evaluating (groups of) predicates once and reuse results

METHOD

- A modified *deterministic*[1] *pushdown automaton* (PDA)
- The PDA is constructed by composing alternating finite automata (AFA)
- Each AFA is constructed by each XPath filter

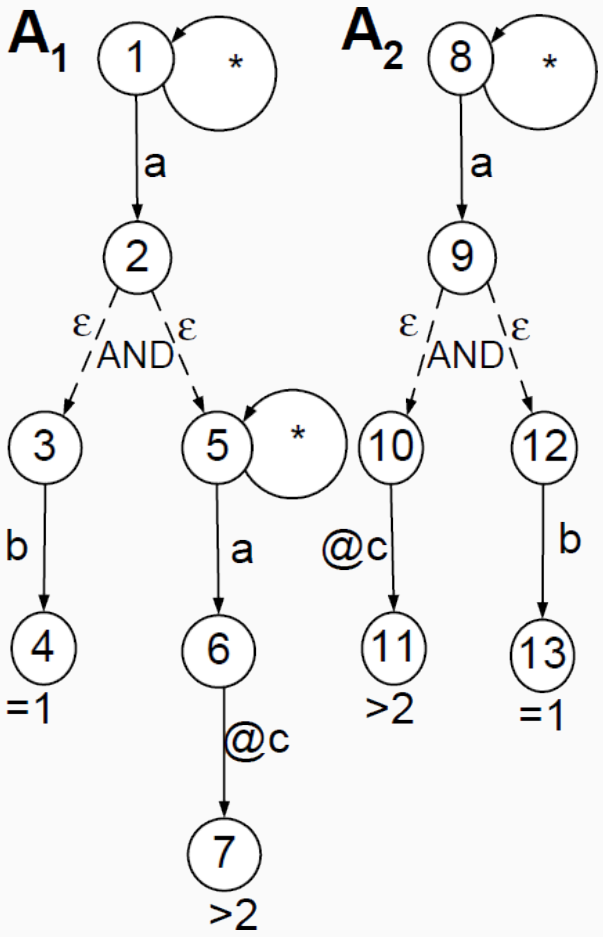
[1] Lookup in O(1) time on complete version

WHAT IS AN AFA?

XPath filters are translated to AFAs

A_1 `//a[b/text()=1 and .//a[@c>2]]`

A_2 `//a[@c>2 and b/text()=1]`



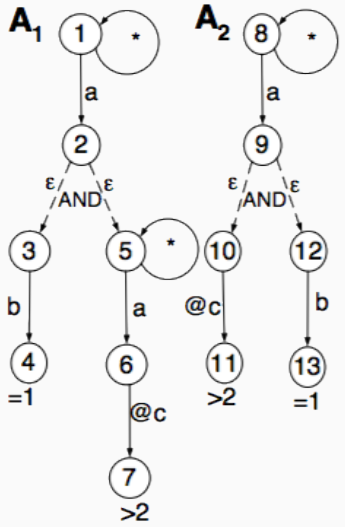
- Built by implementing SAX events

procedure `startDocument()`
  $q^t \leftarrow q_0^t \quad q^b \leftarrow q_0^b$
  $s \leftarrow$ **empty stack**;
procedure `startElement(`a`)`
  push$(s, (q^t, q^b))$;
  $q^t \leftarrow t_{push}(q^t, a)$
  $q^b \leftarrow q_0^b$
procedure `text(`str`)`
  $q^b \leftarrow t_{value}(q^t, str)$
procedure `endElement(`a`)`
  $q_{aux} \leftarrow t_{pop}(q^b, a)$
  $(q_s^t, q_s^b) \leftarrow$ **pop**(s);
  $q^b \leftarrow t_{add}^b(q_s^b, q_{aux})$
  $q^t \leftarrow t_{add}^t(q_s^t, q_{aux})$
procedure `endDocument()`
  return $t_{accept}(q^b)$;

CONSTRUCTING THE XPUSH MACHINE

- Built by implementing SAX events
- The XPush machine computed lazily at runtime
  - Computing all the states eagerly results in an exponential number of states
  - Compute only those that are met at runtime
- Each state in the XPush machine is a set of states in the AFA

- Built by implementing SAX events
- The XPush machine computed lazily at runtime
  - Computing all the states eagerly results in an exponential number of states
  - Compute only those that are met at runtime
- Each state in the XPush machine is a set of states in the AFA



q_0	$\emptyset$
q_1	$\{4, 13\}$
q_2	$\{7, 11\}$
q_3	$\{3, 12\}$
q_4	$\{6, 10\}$
q_5	$\{3, 6, 10, 12\}$
q_6	$\{5\}$
q_7	$\{5, 8\}$
q_8	$\{3, 5, 12\}$
q_9	$\{3, 5, 8, 12\}$
q_{10}	$\{5, 6, 10\}$
q_{11}	$\{5, 6, 8, 10\}$
q_{12}	$\{3, 5, 6, 10, 12\}$
q_{13}	$\{3, 5, 6, 8, 10, 12\}$
q_{14}	$\{1, 5\}$
q_{15}	$\{1, 5, 8\}$
q_{16}	$\{1, 3, 5, 12\}$
q_{17}	$\{1, 3, 5, 8, 12\}$
q_{18}	$\{1, 5, 6, 10\}$
q_{19}	$\{1, 5, 6, 8, 10\}$
q_{20}	$\{1, 3, 5, 6, 10, 12\}$
q_{21}	$\{1, 3, 5, 6, 8, 10, 12\}$

OPTIMIZATIONS

METHODS OF OPTIMIZATION

- Top-down pruning
- Order optimization
- Early notification optimization
- Training in the XPush machine

TOP-DOWN PRUNING

- Removes false positives
- For queries of the form e[c/text()="c"], remove all predicates that does not appear under an `e` element

```
<e>
  <c> "c" </c>
  …
  <c> "c" </c>
</e>
<a>
  <c> "c" </c>
</a>
```

- Expensive to perform, but reduces number of states (reducing memory needed)

ORDER OPTIMIZATION

- If there is a order of elements defined in the DTD, we can faster find false filters
- If we have a query:

```
/person[name/text()="Smith" and age/text()="33" and
phone/text()="5551234"]
```

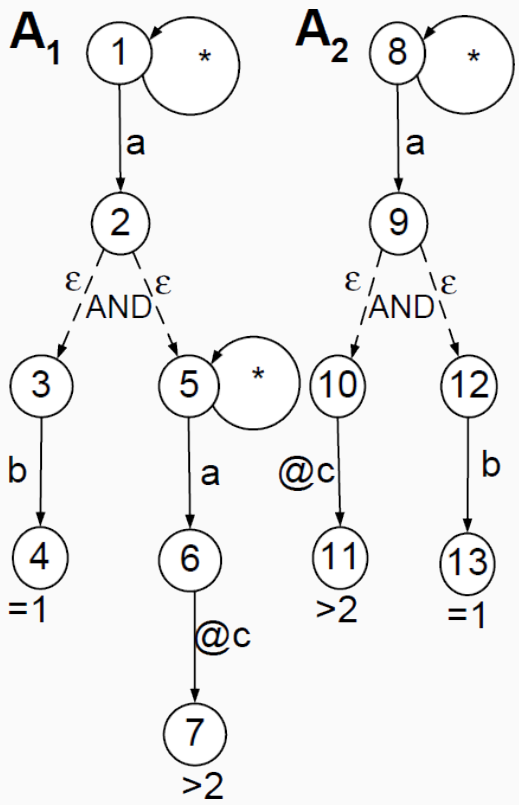
- And XML data:

```
<person><name>John</name><age>33</age>...
```

- Then we can stop after the name predicate and not "activate" the age predicate
- Reduces average length of states and thus run time

EARLY NOTIFICATION OPTIMIZATION

- In early notification, the evaluation of a AFA in the XPush machine is stopped early once the first branching state has matched some node in the XML document
- Requires top-down pruning to ensure correctness
- Generated more states but reduces average length of states

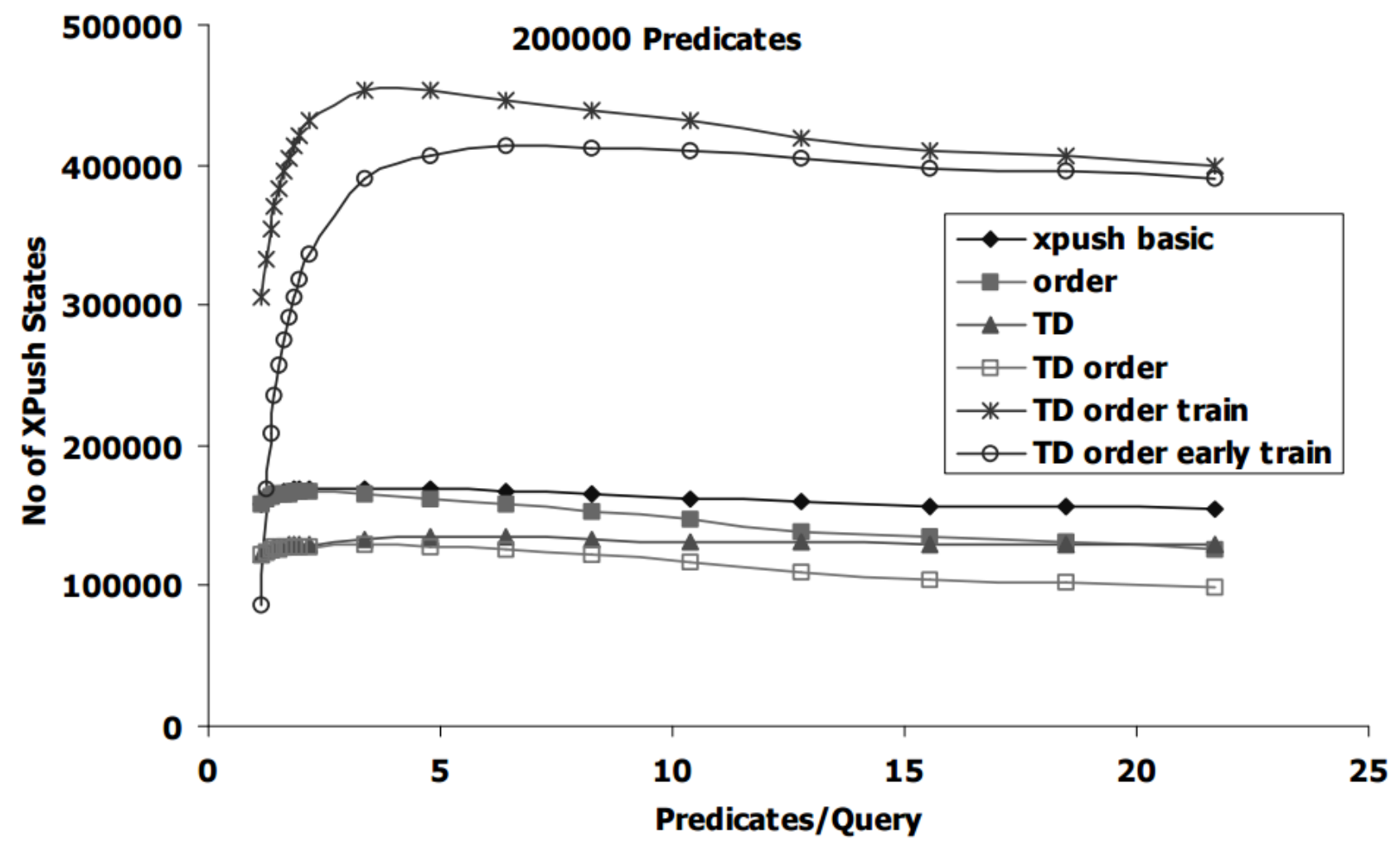


TRAINING IN THE XPUSH MACHINE

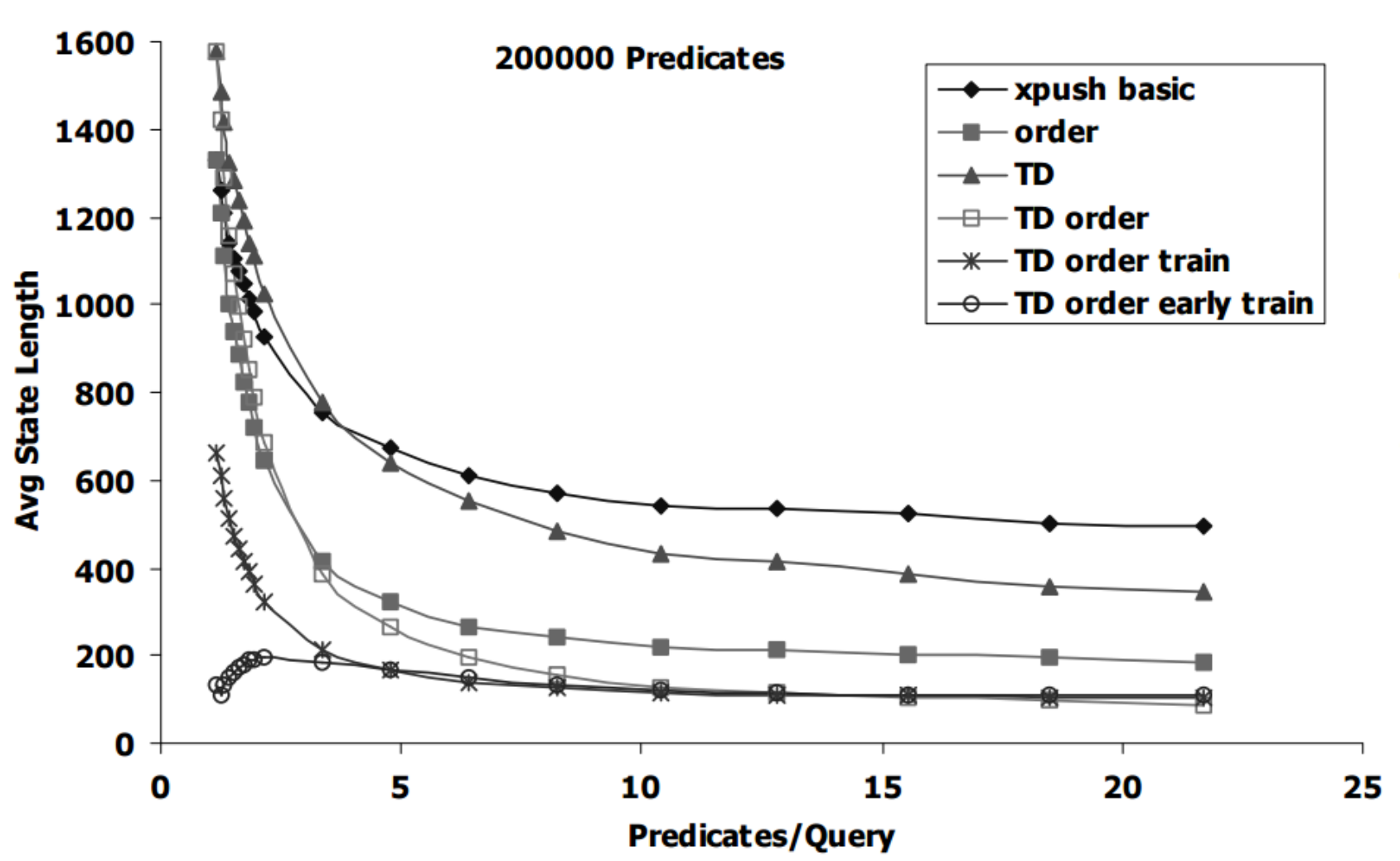
- Generate training data from workload queries
  - If we have * or // we can use the DTD
- The DTD is also used to generate elements in the right order
- Precompute states based on the training data
- Can do lookup instead of runtime evaluation
- Increases number of states but reduces average length of states

EVALUATION

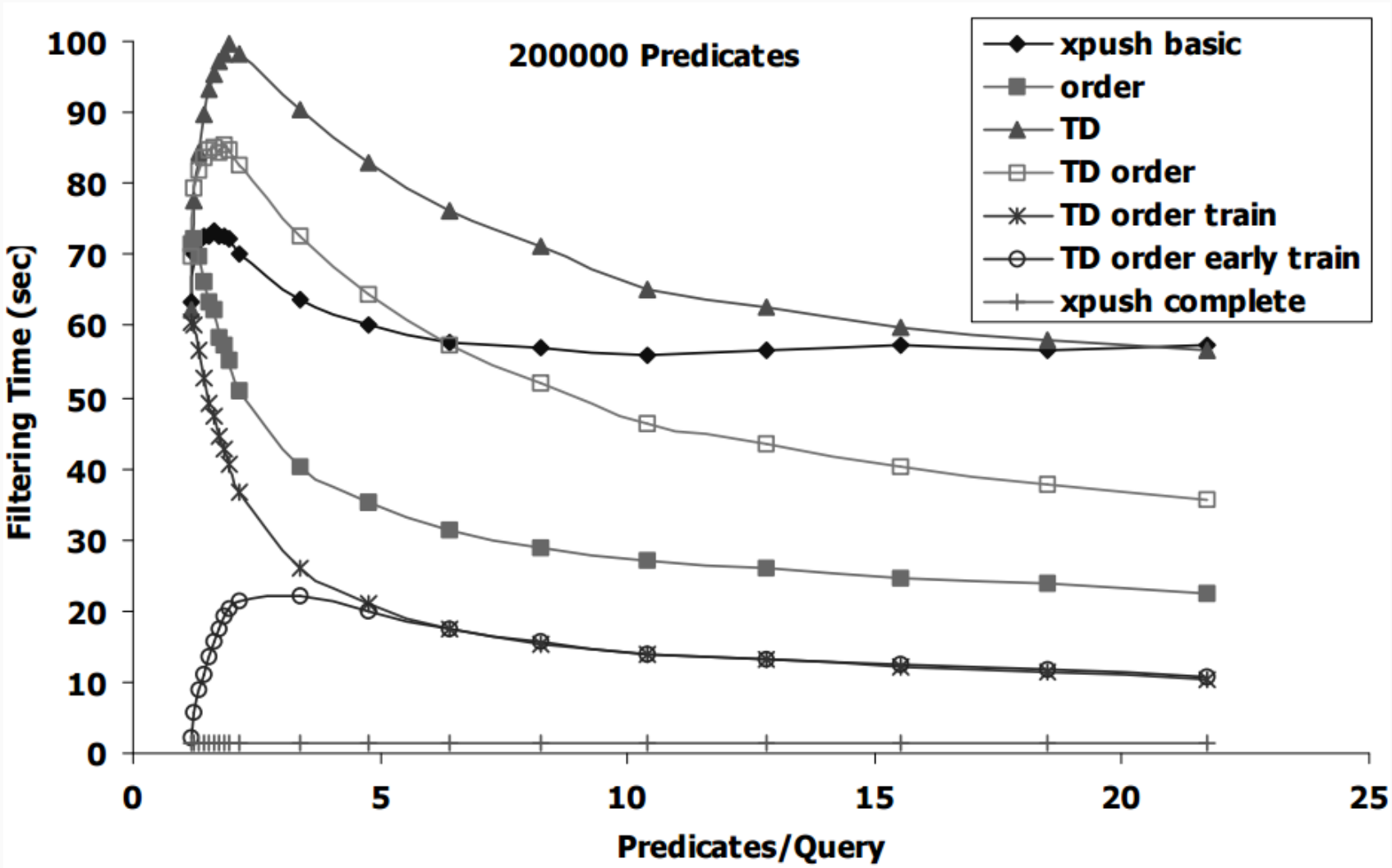
NUMBER OF STATES



AVERAGE LENGTH OF STATES



FILTER EVALUATION TIME



LIMITATIONS

LIMITATIONS

- Supports only a subset of XPath (e.g. no equal on ID)
- Can only stream a single document at a time
- Creation of states and AFAs is expensive
  - If all the predicate are unique, falls back to normal execution + creating of AFAs
  - Results in slow execution than not using this approach
- The queries cannot be evaluated in parallel
- Updates to the XPath queries requires recomputing the XPush Machine from scratch

CONCLUSION

CONCLUSION

- XPush machine is a modified PDA that processes each SAX event in $\mathcal{O}(1)$ time independent of the query workload
- The XPush machine needs to be lazily computed in most applications
- With their setup it ran almost twice as fast as the Apache parser (in 2003)