Streaming Tree Automata and XPath

Olivier Gauwin

Mostrare Project

Ph.D. Defense
September 28th, 2009

supervisors: Joachim Niehren and Sophie Tison

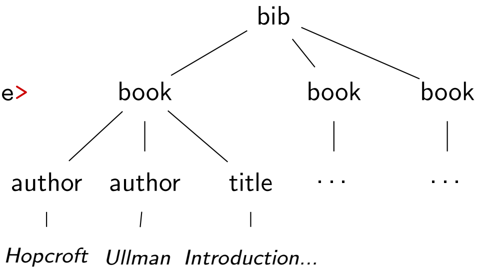
XML

A Format for Semi-Structured Data

XML Document

```
<bib>
  <book>
    <author> Hopcroft </author>
    <author> Ullman </author>
    <title> Introduction... </title>
  </book>
  <book>
    ⋮
  </book>
  <book>
    ⋮
  </book>
</bib>
```

Corresponding Tree



XML

A Format for Semi-Structured Data

XML Document | Corresponding Tree

```
<bib>

</bib>
```

bib

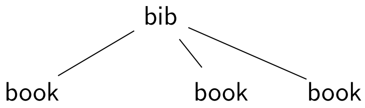
XML

A Format for Semi-Structured Data

XML Document

```
<bib>
  <book>



  </book>
  <book>

  </book>
  <book>

  </book>
</bib>
```

Corresponding Tree



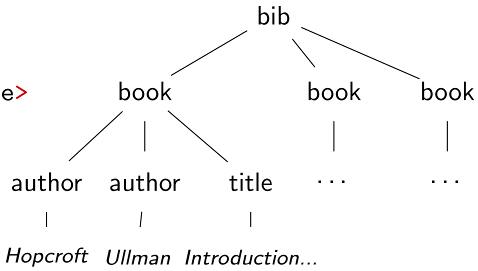
XML

A Format for Semi-Structured Data

XML Document

```
<bib>
  <book>
    <author> Hopcroft </author>
    <author> Ullman </author>
    <title> Introduction... </title>
  </book>
  <book>
    ⋮
  </book>
  <book>
    ⋮
  </book>
</bib>
```

Corresponding Tree



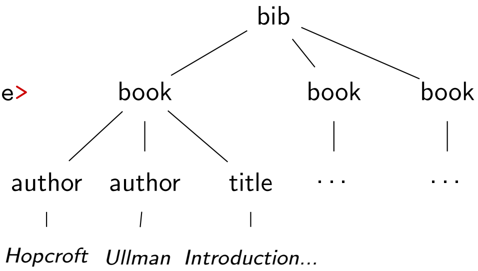
XML

A Format for Semi-Structured Data

XML Document

```
<bib>
  <book>
    <author> Hopcroft </author>
    <author> Ullman </author>
    <title> Introduction... </title>
  </book>
  <book>
    ⋮
  </book>
  <book>
    ⋮
  </book>
</bib>
```

Corresponding Tree



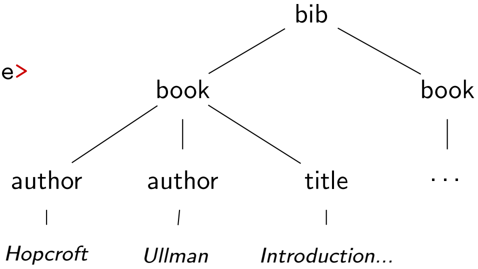
finite labeled ordered unranked trees

Streaming

- process data on-the-fly
- objective: low memory consumption (buffering)
- use cases:
  - huge data (larger than main memory)
  - natural stream sources:
    - network sockets
    - sensors
    - subscribed feeds
    - etc.

XML Streams

```
<bib>
  <book>
    <author> Hopcroft </author>
    <author> Ullman </author>
    <title> Introduction... </title>
  </book>
  <book>
    ⋮
  </book>
</bib>
```

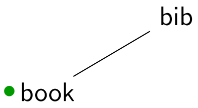


XML Streams

```
<bib>
```

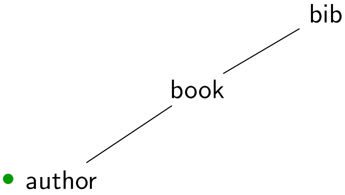
XML Streams

```
<bib>
  <book>
```



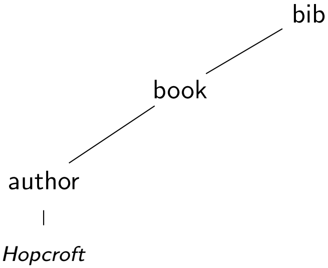
XML Streams

```
<bib>
  <book>
    <author>
```



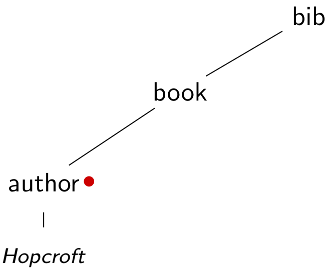
XML Streams

```
<bib>
  <book>
    <author> Hopcroft
```



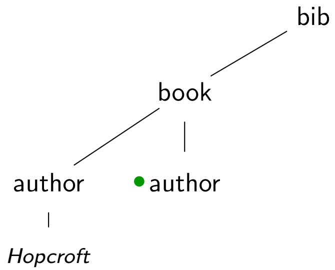
XML Streams

<bib>
  <book>
    <author> *Hopcroft* </author>



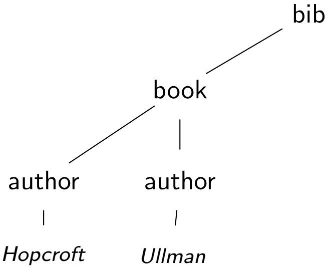
XML Streams

```
<bib>
  <book>
    <author> Hopcroft </author>
    <author>
```



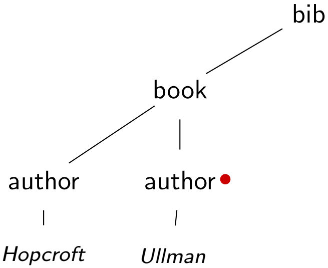
XML Streams

```
<bib>
  <book>
    <author> Hopcroft </author>
    <author> Ullman
```



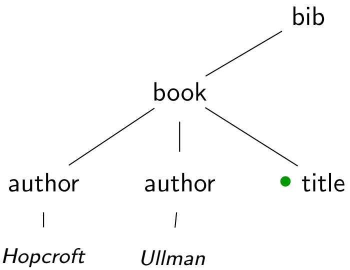
XML Streams

```
<bib>
  <book>
    <author> Hopcroft </author>
    <author> Ullman </author>
```



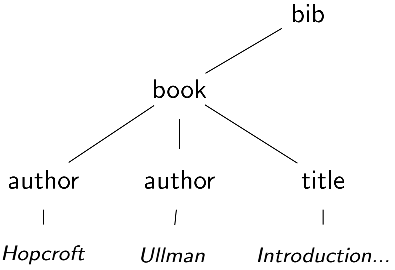
XML Streams

```
<bib>
  <book>
    <author> Hopcroft </author>
    <author> Ullman </author>
    <title>
```



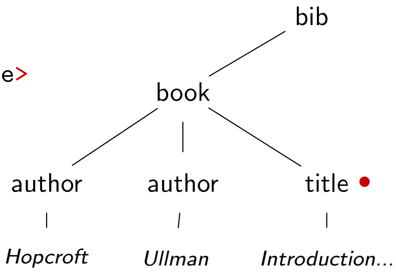
XML Streams

```
<bib>
  <book>
    <author> Hopcroft </author>
    <author> Ullman </author>
    <title> Introduction...
```



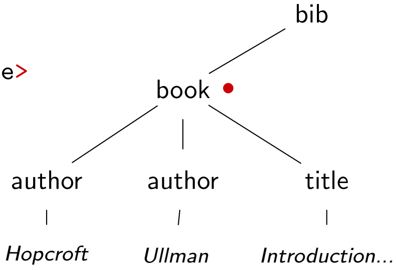
XML Streams

```
<bib>
  <book>
    <author> Hopcroft </author>
    <author> Ullman </author>
    <title> Introduction... </title>
```



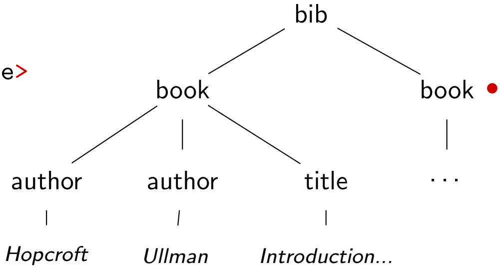
XML Streams

```
<bib>
  <book>
    <author> Hopcroft </author>
    <author> Ullman </author>
    <title> Introduction... </title>
  </book>
```



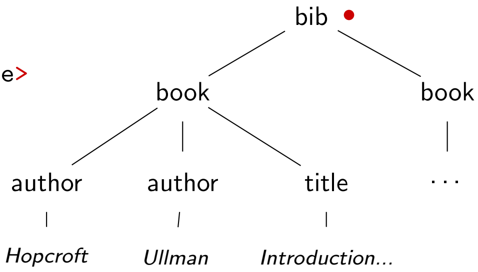
XML Streams

```
<bib>
  <book>
    <author> Hopcroft </author>
    <author> Ullman </author>
    <title> Introduction... </title>
  </book>
  <book>
    ⋮
  </book>
```

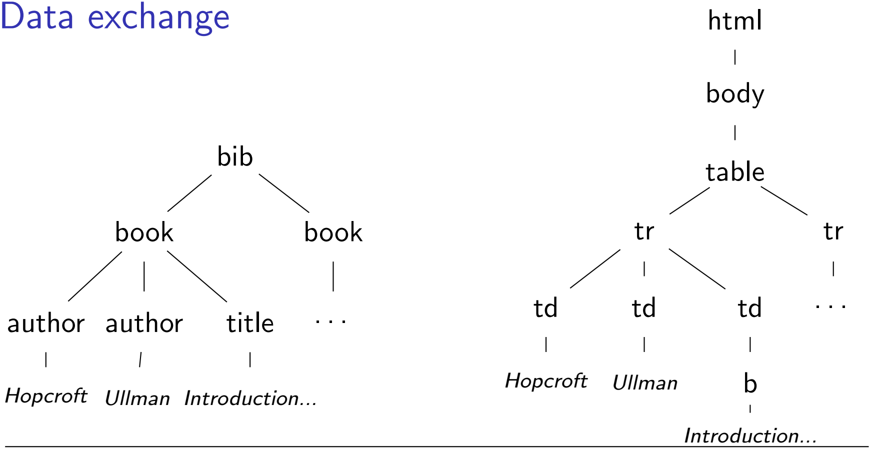


XML Streams

```
<bib>
  <book>
    <author> Hopcroft </author>
    <author> Ullman </author>
    <title> Introduction... </title>
  </book>
  <book>
    ⋮
  </book>
</bib>
```



Data exchange



conform to schema 1 → conform to schema 2

- requires transformations
- usually based on selection of tuples of nodes
- via queries

Queries

Monadic Queries

we only deal with monadic queries ($n = 1$) in this talk, i.e.:

$$Q(t) \subseteq nod(t)$$

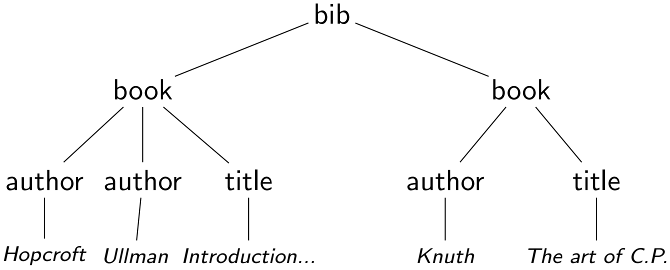
For clarity, we ignore schemas.

Studied Query Classes

- XPath
- Queries by Automata

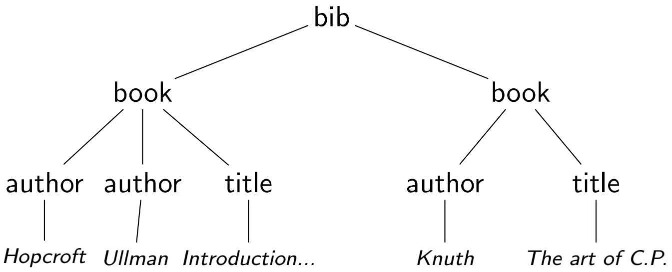
XPath

- W3C query language for XML documents
- navigational language



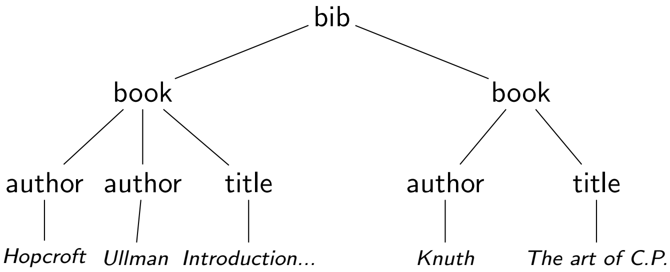
XPath

- W3C query language for XML documents
- navigational language
  - axis: *ch* (child), ch^* (descendant-or-self), etc.



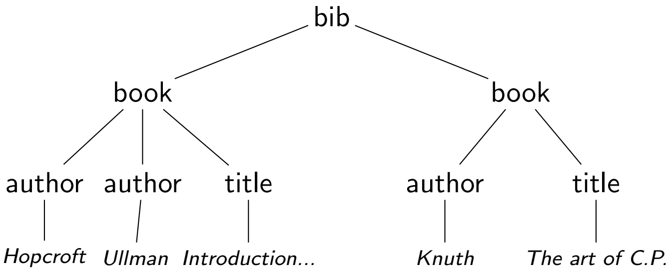
XPath

- W3C query language for XML documents
- navigational language
  - axis: *ch* (child), ch^* (descendant-or-self), etc.
  - step: ch^*::book   (or with wildcard: ch^*::$*$)



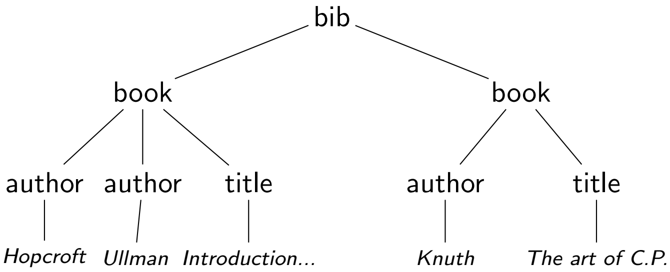
XPath

- W3C query language for XML documents
- navigational language
  - axis: ch (child), ch^* (descendant-or-self), etc.
  - step: ch^*::book (or with wildcard: ch^*::$*$)
  - path: ch^*::book/ch::author



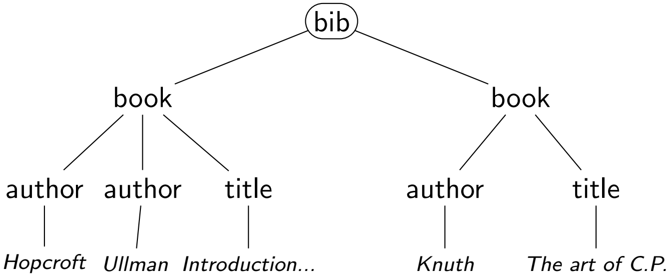
XPath

- W3C query language for XML documents
- navigational language
  - axis: ch (child), ch^* (descendant-or-self), etc.
  - step: ch^*::book (or with wildcard: $ch^*::*$)
  - path: ch^*::book/ch::author
  - filter: [ch::author="*Hopcroft*"] (and also: [F *and* F], [*not*(F)])



XPath

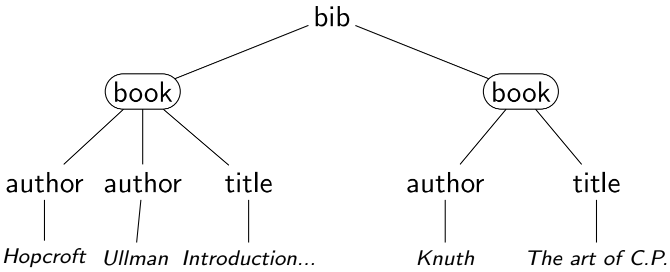
- W3C query language for XML documents
- navigational language
  - axis: ch (child), ch^* (descendant-or-self), etc.
  - step: ch^*::book (or with wildcard: $ch^*::*$)
  - path: ch^*::book/ch::author
  - filter: [ch::author=''*Hopcroft*''] (and also: [*F and F*], [*not*(*F*)])



- for instance: /ch^*::book[ch::author=''*Hopcroft*'']/ch::author
  selects all co-authors of Hopcroft

XPath

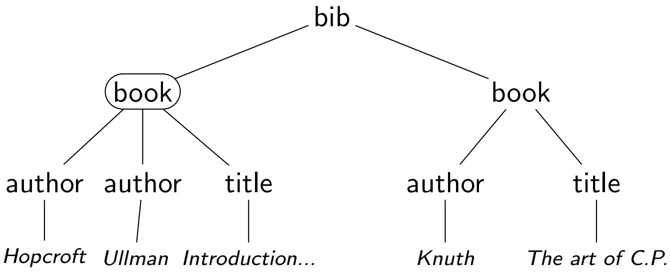
- W3C query language for XML documents
- navigational language
  - axis: *ch* (child), ch^* (descendant-or-self), etc.
  - step: ch^*::book (or with wildcard: ch^*::$*$)
  - path: ch^*::book/*ch*::author
  - filter: [*ch*::author=″*Hopcroft*″] (and also: [*F and F*], [*not*(*F*)])



- for instance: /ch^*::book[*ch*::author=″*Hopcroft*″]/*ch*::author
  selects all co-authors of Hopcroft

XPath

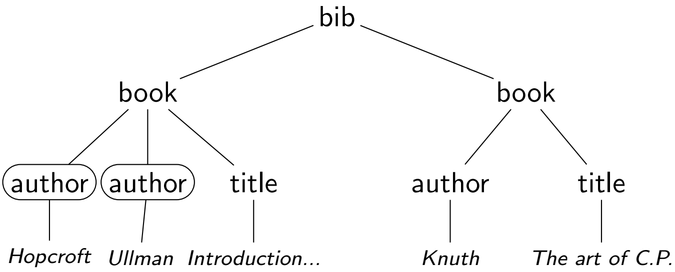
- W3C query language for XML documents
- navigational language
  - axis: ch (child), ch^* (descendant-or-self), etc.
  - step: ch^*::book (or with wildcard: ch^*::$*$)
  - path: ch^*::book/ch::author
  - filter: [ch::author="*Hopcroft*"] (and also: [F *and* F], [*not*(F)])



- for instance: /ch^*::book[ch::author="*Hopcroft*"]/ch::author
  selects all co-authors of Hopcroft

XPath

- W3C query language for XML documents
- navigational language
  - axis: *ch* (child), ch^* (descendant-or-self), etc.
  - step: ch^*::book (or with wildcard: ch^*::$*$)
  - path: ch^*::book/*ch*::author
  - filter: [*ch*::author="*Hopcroft*"] (and also: [*F and F*], [*not*(*F*)])



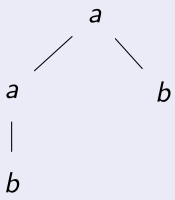
- for instance: /ch^*::book[*ch*::author="*Hopcroft*"]/*ch*::author
  selects all co-authors of Hopcroft

XPath fragments

- Core XPath : navigational core (no data values)
- Forward XPath : Core XPath restricted to forward axes
- Downward XPath : Core XPath restricted to axes *ch*, ch^*

Queries by Automata

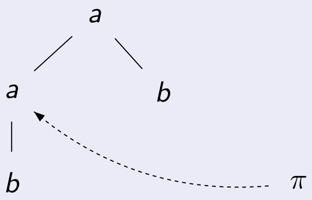
Canonical trees

$t =$ 

Queries by Automata

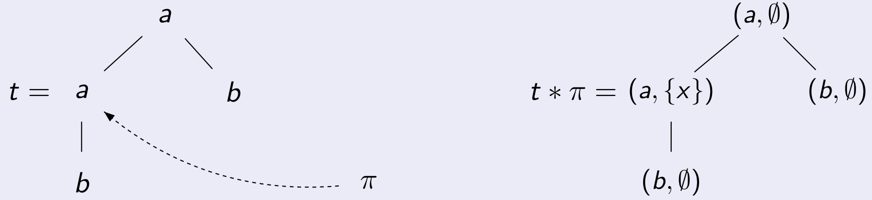
Canonical trees

$t =$



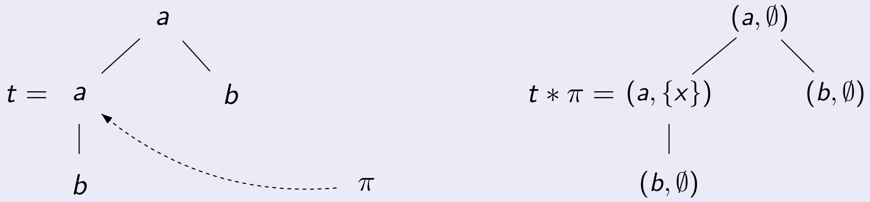
Queries by Automata

Canonical trees



Queries by Automata

Canonical trees



Canonical language

- A monadic query Q defines the tree language $L_Q = \{t * \pi \mid \pi \in Q(t)\}$
- A language L of canonical trees defines the query $Q(t)$ such that $\pi \in Q(t)$ iff $t * \pi \in L$

A tree automaton over $\Sigma \times 2^{\{x\}}$ recognizing canonical trees defines a query over Σ.

Querying XML Streams

Example: co-authors of Hopcroft

XML stream	Buffers	Actions
<bib>$_1$		

Querying XML Streams

Example: co-authors of Hopcroft

XML stream	Buffers	Actions
<bib>$_1$		
<book>$_2$		

Querying XML Streams

Example: co-authors of Hopcroft

XML stream	Buffers	Actions
<bib>$_1$		
<book>$_2$		
<author>$_3$	3	

Querying XML Streams

Example: co-authors of Hopcroft

XML stream	Buffers	Actions
`<bib>`$_1$		
`<book>`$_2$		
`<author>`$_3$	3	
Ullman	3	

Querying XML Streams

Example: co-authors of Hopcroft

XML stream	Buffers	Actions
<bib>$_1$		
<book>$_2$		
<author>$_3$	3	
Ullman	3	
</author>	3	

Querying XML Streams

Example: co-authors of Hopcroft

XML stream	Buffers	Actions
<bib>$_1$		
<book>$_2$		
<author>$_3$	3	
Ullman	3	
</author>	3	
<author>$_4$	3,4	

Querying XML Streams

Example: co-authors of Hopcroft

XML stream	Buffers	Actions
`<bib>`$_1$		
`<book>`$_2$		
`<author>`$_3$	3	
Ullman	3	
`</author>`	3	
`<author>`$_4$	3,4	
Hopcroft		output $\{3, 4\}$

Querying XML Streams

Example: co-authors of Hopcroft

XML stream	Buffers	Actions
<bib>$_1$		
<book>$_2$		
<author>$_3$	3	
Ullman	3	
</author>	3	
<author>$_4$	3,4	
Hopcroft		output $\{3, 4\}$
</author>		

Querying XML Streams

Example: co-authors of Hopcroft

XML stream	Buffers	Actions
<bib>$_1$		
<book>$_2$		
<author>$_3$	3	
Ullman	3	
</author>	3	
<author>$_4$	3,4	
Hopcroft		output $\{3, 4\}$
</author>		
<author>$_5$		output $\{5\}$

Querying XML Streams

Example: co-authors of Hopcroft

XML stream	Buffers	Actions
<bib>$_1$		
<book>$_2$		
<author>$_3$	3	
Ullman	3	
</author>	3	
<author>$_4$	3,4	
Hopcroft		output $\{3, 4\}$
</author>		
<author>$_5$		output $\{5\}$
Vianu		

Querying XML Streams

Example: co-authors of Hopcroft

XML stream	Buffers	Actions
<bib>$_1$		
<book>$_2$		
<author>$_3$	3	
Ullman	3	
</author>	3	
<author>$_4$	3,4	
Hopcroft		output $\{3, 4\}$
</author>		
<author>$_5$		output $\{5\}$
Vianu		
</author>		

Querying XML Streams

Example: co-authors of Hopcroft

XML stream	Buffers	Actions
<bib>$_1$		
<book>$_2$		
<author>$_3$	3	
Ullman	3	
</author>	3	
<author>$_4$	3,4	
Hopcroft		output $\{3, 4\}$
</author>		
<author>$_5$		output $\{5\}$
Vianu		
</author>		
</book>		

Querying XML Streams

Example: co-authors of Hopcroft

XML stream	Buffers	Actions
<bib>$_1$		
<book>$_2$		
<author>$_3$	3	
Ullman	3	
</author>	3	
<author>$_4$	3,4	
Hopcroft		output $\{3, 4\}$
</author>		
<author>$_5$		output $\{5\}$
Vianu		
</author>		
</book>		
<book>		

Querying XML Streams

Example: co-authors of Hopcroft

XML stream	Buffers	Actions
<bib>$_1$		
<book>$_2$		
<author>$_3$	3	
Ullman	3	
</author>	3	
<author>$_4$	3,4	
Hopcroft		output $\{3, 4\}$
</author>		
<author>$_5$		output $\{5\}$
Vianu		
</author>		
</book>		
<book>		
<author>$_6$	6	

Querying XML Streams

Example: co-authors of Hopcroft

XML stream	Buffers	Actions
<bib>$_1$		
<book>$_2$		
<author>$_3$	3	
Ullman	3	
</author>	3	
<author>$_4$	3,4	
Hopcroft		output $\{3, 4\}$
</author>		
<author>$_5$		output $\{5\}$
Vianu		
</author>		
</book>		
<book>		
<author>$_6$	6	
Knuth	6	

Querying XML Streams

Example: co-authors of Hopcroft

XML stream	Buffers	Actions
<bib>$_1$		
<book>$_2$		
<author>$_3$	3	
Ullman	3	
</author>	3	
<author>$_4$	3,4	
Hopcroft		output $\{3, 4\}$
</author>		
<author>$_5$		output $\{5\}$
Vianu		
</author>		
</book>		
<book>		
<author>$_6$	6	
Knuth	6	
</author>	6	

Querying XML Streams

Example: co-authors of Hopcroft

XML stream	Buffers	Actions
<bib>$_1$		
<book>$_2$		
<author>$_3$	3	
Ullman	3	
</author>	3	
<author>$_4$	3,4	
Hopcroft		output $\{3, 4\}$
</author>		
<author>$_5$		output $\{5\}$
Vianu		
</author>		
</book>		
<book>		
<author>$_6$	6	
Knuth	6	
</author>	6	
</book>		discard $\{6\}$

Querying XML Streams

Example: co-authors of Hopcroft

XML stream	Buffers	Actions
<bib>$_1$		
<book>$_2$		
<author>$_3$	3	
Ullman	3	
</author>	3	
<author>$_4$	3,4	
Hopcroft		output $\{3,4\}$
</author>		
<author>$_5$		output $\{5\}$
Vianu		
</author>		
</book>		
<book>		
<author>$_6$	6	
Knuth	6	
</author>	6	
</book>		discard $\{6\}$
</bib>		

Questions and Contributions

Questions and Contributions

- How much memory is needed?
  - new model: *Streaming Random Access Machines*, for query answering algorithms over streams
  - we derive a lower space bound

Questions and Contributions

- How much memory is needed?
  - new model: *Streaming Random Access Machines*, for query answering algorithms over streams
  - we derive a lower space bound
- How to characterize query languages suitable to streaming?
  - new measure: *m-Streamability*
  - hardness results for queries by automata and XPath

Questions and Contributions

- How much memory is needed?
  - new model: *Streaming Random Access Machines*, for query answering algorithms over streams
  - we derive a lower space bound
- How to characterize query languages suitable to streaming?
  - new measure: *m-Streamability*
  - hardness results for queries by automata and XPath
  - *Bounded Concurrency and Delay* (not presented here)

Questions and Contributions

- How much memory is needed?
  - new model: *Streaming Random Access Machines*, for query answering algorithms over streams
  - we derive a lower space bound
- How to characterize query languages suitable to streaming?
  - new measure: *m-Streamability*
  - hardness results for queries by automata and XPath
  - *Bounded Concurrency and Delay* (not presented here)
- Are there tractable fragments?

Questions and Contributions

- How much memory is needed?
  - new model: *Streaming Random Access Machines*,
    for query answering algorithms over streams
  - we derive a lower space bound
- How to characterize query languages suitable to streaming?
  - new measure: *m-Streamability*
  - hardness results for queries by automata and XPath
  - *Bounded Concurrency and Delay* (not presented here)
- Are there tractable fragments?
  - queries defined by deterministic *Streaming Tree Automata*
    - ⋆ *Earliest Query Answering* algorithm

Questions and Contributions

- How much memory is needed?
  - new model: *Streaming Random Access Machines*,
    for query answering algorithms over streams
  - we derive a lower space bound
- How to characterize query languages suitable to streaming?
  - new measure: *m-Streamability*
  - hardness results for queries by automata and XPath
  - *Bounded Concurrency and Delay* (not presented here)
- Are there tractable fragments?
  - queries defined by deterministic *Streaming Tree Automata*
    - ⋆ *Earliest Query Answering* algorithm
  - *k-Downward XPath*: a streamable fragment of XPath

Buffering Requirements

of query answering algorithms over XML streams

- $O(|t|)$ is equivalent to in-memory algorithms
  - too much space

Buffering Requirements

of query answering algorithms over XML streams

- $O(|t|)$ is equivalent to in-memory algorithms
  - too much space
- $O(1)$ = bounded buffering (independent from t)

Buffering Requirements

of query answering algorithms over XML streams

- $O(|t|)$ is equivalent to in-memory algorithms
  - too much space
- $O(1)$ = bounded buffering (independent from t)
  - Boolean queries (tree acceptors)

Buffering Requirements

of query answering algorithms over XML streams

- $O(|t|)$ is equivalent to in-memory algorithms
  - too much space

- $O(1)$ = bounded buffering (independent from t)
  - Boolean queries (tree acceptors)
    - ⋆ validation wrt a DTD in $O(1)$ is only known for restricted forms of DTDs (Segoufin, Vianu 02)

Buffering Requirements

of query answering algorithms over XML streams

- $O(|t|)$ is equivalent to in-memory algorithms
  - too much space

- $O(1)$ = bounded buffering (independent from t)
  - Boolean queries (tree acceptors)
    - validation wrt a DTD in $O(1)$ is only known for restricted forms of DTDs (Segoufin, Vianu 02)
    - for Positive Core XPath, filtering non-recursive documents requires space at least exponential in the size of the expression (Benedikt, Jeffrey 07)

Buffering Requirements

of query answering algorithms over XML streams

- $O(|t|)$ is equivalent to in-memory algorithms
  - too much space

- $O(1)$ = bounded buffering (independent from t)
  - Boolean queries (tree acceptors)
    - validation wrt a DTD in $O(1)$ is only known for restricted forms of DTDs (Segoufin, Vianu 02)
    - for Positive Core XPath, filtering non-recursive documents requires space at least exponential in the size of the expression (Benedikt, Jeffrey 07)
  - monadic queries

Buffering Requirements

of query answering algorithms over XML streams

- $O(|t|)$ is equivalent to in-memory algorithms
  - too much space
- $O(1)$ = bounded buffering (independent from t)
  - Boolean queries (tree acceptors)
    - ★ validation wrt a DTD in $O(1)$ is only known for restricted forms of DTDs (Segoufin, Vianu 02)
    - ★ for Positive Core XPath, filtering non-recursive documents requires space at least exponential in the size of the expression (Benedikt, Jeffrey 07)
  - monadic queries
    - ★ for Positive Core XPath, there is no streaming algorithm using bounded buffering, even on non-recursive documents (Benedikt, Jeffrey 07)

Buffering Requirements

of query answering algorithms over XML streams

- $O(|t|)$ is equivalent to in-memory algorithms
  - too much space
- $O(1)$ = bounded buffering (independent from t)
  - Boolean queries (tree acceptors)
    - validation wrt a DTD in $O(1)$ is only known for restricted forms of DTDs (Segoufin, Vianu 02)
    - for Positive Core XPath, filtering non-recursive documents requires space at least exponential in the size of the expression (Benedikt, Jeffrey 07)
  - monadic queries
    - for Positive Core XPath, there is no streaming algorithm using bounded buffering, even on non-recursive documents (Benedikt, Jeffrey 07)
    - $O(1)$ is impossible for co-authors of Hopcroft

Concurrency

(Bar-Yossef, Fontoura, Josifovski 05)

Alive nodes

A node π of t is alive for Q at event η if:

- there is a continuation t' of t after η s.t. $\pi \in Q(t')$
- there is a continuation t'' of t after η s.t. $\pi \notin Q(t'')$

Concurrency

The concurrency of Q wrt t is the maximal number of simultaneous alive nodes.

Concurrency

Example

Query Q: co-authors of Hopcroft

	XML stream	Alive nodes
	<bib>$_1$	
	<book>$_2$	
	<author>$_3$	3
	Ullman	3
	</author>	3
η	<author>$_4$	3,4

Concurrency

Example

Query Q: co-authors of Hopcroft

	XML stream	Alive nodes
	<bib>$_1$	
	<book>$_2$	
	<author>$_3$	3
	Ullman	3
	</author>	3
η	<author>$_4$	3,4

Nodes 1 and 2 are not alive at η because:

- there is no continuation for which they are selected

Concurrency

Example

Query Q: co-authors of Hopcroft

	XML stream	Alive nodes
	<bib>$_1$	
	<book>$_2$	
	<author>$_3$	3
	Ullman	3
	</author>	3
η	<author>$_4$	3,4

Nodes 1 and 2 are not alive at η because:

- there is no continuation for which they are selected

Nodes 3 and 4 are alive at η because:

Concurrency

Example

Query Q: co-authors of Hopcroft

	XML stream	Alive nodes
	<bib>$_1$	
	<book>$_2$	
	<author>$_3$	3
	Ullman	3
	</author>	3
η	<author>$_4$	3,4
	Hopcroft	output $\{3, 4\}$
	</author>	
	</book>	
	</bib>	

Nodes 1 and 2 are not alive at η because:

- there is no continuation for which they are selected

Nodes 3 and 4 are alive at η because:

- there is one continuation for which they are selected and

Concurrency

Example

Query Q: co-authors of Hopcroft

	XML stream	Alive nodes
	<bib>$_1$	
	<book>$_2$	
	<author>$_3$	3
	Ullman	3
	</author>	3
η	<author>$_4$	3,4
	Vianu	3,4
	</author>	3,4
	</book>	discard $\{3, 4\}$
	</bib>	

Nodes 1 and 2 are not alive at η because:

- there is no continuation for which they are selected

Nodes 3 and 4 are alive at η because:

- there is one continuation for which they are selected and
- there is one continuation for which they are rejected

Concurrency

Example

Query Q: co-authors of Hopcroft

	XML stream	Alive nodes
	<bib>$_1$	
	<book>$_2$	
	<author>$_3$	3
	Ullman	3
	</author>	3
η	<author>$_4$	3,4
	Vianu	3,4
	</author>	3,4
	</book>	discard $\{3,4\}$
	</bib>	

The concurrency of Q wrt t is 2.

Nodes 1 and 2 are not alive at η because:

- there is no continuation for which they are selected

Nodes 3 and 4 are alive at η because:

- there is one continuation for which they are selected and
- there is one continuation for which they are rejected

Concurrency

A space lower bound?

The concurrency was known to be a lower bound on a very special case.

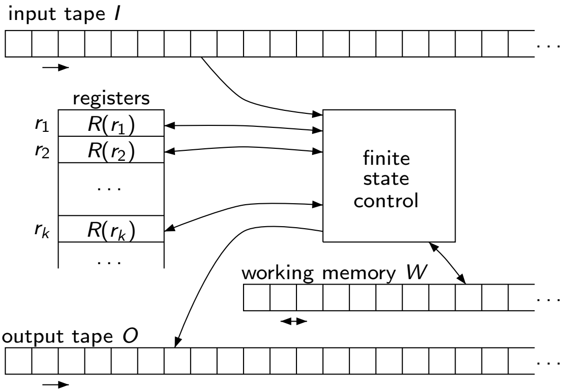
Theorem (Bar-Yossef, Fontoura, Josifovski 05)

Let t be a non-recursive tree, Q a query in Downward XPath without wildcard, and c the close-concurrency of Q wrt t. Then there is a tree t' similar to t for which evaluating Q requires space at least c.

Is concurrency a lower bound for all query answering algorithms on XML streams?

- in general no, due to possible compaction of buffered candidates
- in known algorithms yes

Streaming Random Access Machines (SRAMs)



- node identifiers are stored in registers, and unknown from controller
  - this avoids compaction tricks
- space used = number of registers + used working memory

Streaming Random Access Machines (SRAMs)

Theorem

Concurrency is a space lower bound for queries computed by SRAMs.

Deciding Bounded Concurrency

Hardness results

$$\underbrace{\exists k.\ \forall t.\ \text{concurrency of } Q \text{ on } t \leq k}_{\text{bounded concurrency}} \quad \Rightarrow \quad \text{bounded buffering is possible}$$

Deciding Bounded Concurrency

Hardness results

$$\underbrace{\exists k.\ \forall t.\ \text{concurrency of } Q \text{ on } t \leq k}_{\text{bounded concurrency}} \quad \Rightarrow \quad \text{bounded buffering is possible}$$

Hard queries

$$all(Q) = /self::*[lastchild::*[Q]]/ch::*$$

Given a query class $\mathcal{Q}$ for which *all* and *not* can be defined in polynomial time, deciding whether a query has bounded concurrency is harder than universality of the corresponding Boolean query.

Consequences

- coNP-hard for Downward XPath
- EXPTIME-hard for queries by non-deterministic automata

Deciding Bounded Concurrency

Positive results

Theorem

For queries defined by deterministic Streaming Tree Automata:

- *deciding bounded concurrency is in* PTIME
- *deciding k-bounded concurrency is in* PTIME *when k is fixed*

Similar results for bounded delay.

This result is obtained through properties of recognizable relations over unranked trees, and a reduction to bounded valuedness of transducers (SEIDL 92).

Bounded vs Unbounded Concurrency

We also want to deal with queries with unbounded concurrency:

- on real documents, concurrency may be bounded, even though not specified in schemas (e.g. co-authors)
- concurrency may be large for some trees, and small for others

Towards a Measure of Streamability

- Space and time restrictions
  - time also has to be considered:
  - deciding aliveness of a node at a given event is often computationally hard
    - ⋆ coNP-hard for Downward XPath,
    - ⋆ EXPTIME-hard for queries by automata
- Streamability concerns query classes, not queries
  - a query class $\mathcal{Q}$ is a set of query definitions $e \in \mathcal{Q}$ with size $|e| \geq 1$ and defining queries Q_e
  - for instance: XPath expressions, automata, etc.

Streamability

Definition

Let $m \in \mathbb{N} \cup \{\infty\}$. A query class $\mathcal{Q}$ is m-streamable iff

there exists a polynomial p such that for all $e \in \mathcal{Q}$:

- an SRAM $\mathcal{M}_e$ computing Q_e can be built in time $p(|e|)$
- for all trees t with $concur_{Q_e}(t) \leq m$:

  $\mathcal{M}_e$ uses per event space and time in $p(|e|)$

Streamability

Definition

Let $m \in \mathbb{N} \cup \{\infty\}$. A query class $\mathcal{Q}$ is m-streamable iff

there exists a polynomial p such that for all $e \in \mathcal{Q}$:

- an SRAM $\mathcal{M}_e$ computing Q_e can be built in time $p(|e|)$
- for all trees t with $concur_{Q_e}(t) \leq m$:

  $\mathcal{M}_e$ uses per event space and time in $p(|e|)$

Hierarchy

0-streamable $\supseteq$ 1-streamable $\supseteq$ 2-streamable $\supseteq \ldots \supseteq$ ∞-streamable

Streamability

Definition

Let $m \in \mathbb{N} \cup \{\infty\}$. A query class $\mathcal{Q}$ is m-streamable iff

there exists a polynomial p such that for all $e \in \mathcal{Q}$:

- an SRAM $\mathcal{M}_e$ computing Q_e can be built in time $p(|e|)$
- for all trees t with $concur_{Q_e}(t) \leq m$:

  $\mathcal{M}_e$ uses per event space and time in $p(|e|)$

Hierarchy

0-streamable $\supseteq$ 1-streamable $\supseteq$ 2-streamable $\supseteq \ldots \supseteq$ ∞-streamable

∞-streamability vs finite streamability

$\mathcal{Q}$ is ∞-streamable iff:

- $\mathcal{Q}$ is m-streamable for all $m \in \mathbb{N}$ (with the same polynomial p) and
- $\mathcal{Q}$ has polynomially bounded concurrency, i.e., there is a polynomial p' s.t. $\forall e \in \mathcal{Q}$, $\forall t$, $concur_{Q_e}(t) \leq p'(|e|)$

Hardness of Streamability

Theorem

If $\mathcal{Q}$ is a query class such that:

1. *queries $all(Q_e)$ can be defined in* $\text{P}\small{\text{TIME}}$ *in* $|e|$
2. *membership $a \in L_{[Q_e]}$ can be tested in* $\text{P}\small{\text{TIME}}$ *in* $|e|$
3. *$\mathcal{Q}$ is 0-streamable*

then universality of Boolean queries $\{[Q_e] \mid e \in \mathcal{Q}$ descending$\}$ can be solved in $\text{P}\small{\text{TIME}}$.

$all(Q) = /self::*[lastchild::*[Q]]/ch::*$

Hardness of Streamability

Theorem

If $\mathcal{Q}$ is a query class such that:

1. *queries $all(Q_e)$ can be defined in* PTIME *in* $|e|$
2. *membership $a \in L_{[Q_e]}$ can be tested in* PTIME *in* $|e|$
3. *$\mathcal{Q}$ is 0-streamable*

then universality of Boolean queries $\{[Q_e] \mid e \in \mathcal{Q} \text{ descending}\}$ can be solved in PTIME.

$all(Q) = /self::*[lastchild::*[Q]]/ch::*$

Consequences

- Forward XPath is not 0-streamable except if P=NP.
- queries by automata are not 0-streamable.

Positive results?

Question

Are there streamable and expressive fragments of XPath and automata?

Streamability of Queries by Automata

- non-deterministic automata are not 0-streamable

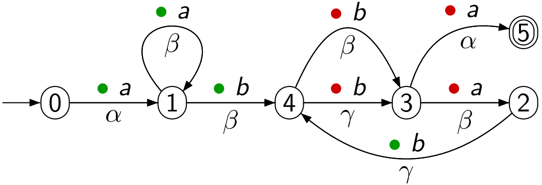
Streamability of Queries by Automata

- non-deterministic automata are not 0-streamable
- automata will be evaluated according to pre-order traversal of trees
  - we use the corresponding notion of determinism

Streamability of Queries by Automata

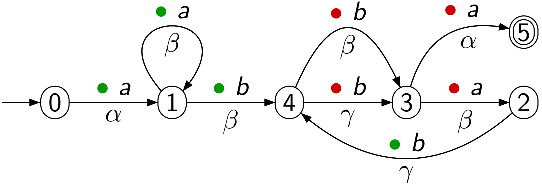
- non-deterministic automata are not 0-streamable
- automata will be evaluated according to pre-order traversal of trees
  - we use the corresponding notion of determinism
  - we define Streaming Tree Automata, a variant of:
    - Pushdown Forest Automata (Neumann, Seidl 98)
    - Visibly Pushdown Automata (Alur, Madhusudan 04)
    - Nested Word Automata (Alur 07)
    - etc.

Streaming Tree Automata (STAs)

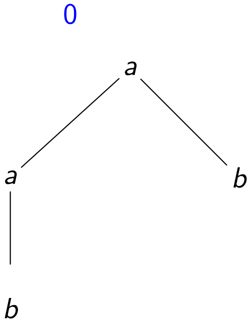


A: STA on $\mathcal{T}_\Sigma$ with $\Sigma = \{a, b\}$
and $stat_e = \{0, 1, 2, 3, 4, 5\}$
and $stat_n = \{\alpha, \beta, \gamma\}$

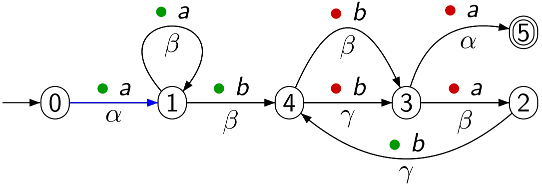
Streaming Tree Automata (STAs)



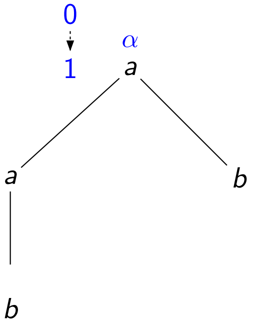
A: STA on $\mathcal{T}_\Sigma$ with $\Sigma = \{a, b\}$
and $stat_e = \{0, 1, 2, 3, 4, 5\}$
and $stat_n = \{\alpha, \beta, \gamma\}$



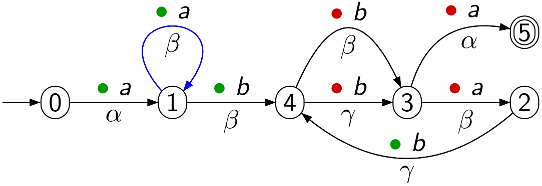
Streaming Tree Automata (STAs)



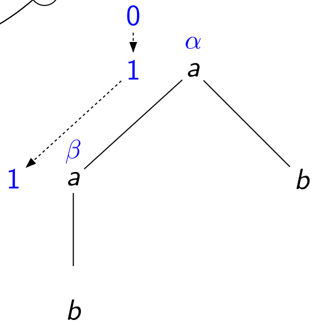
A: STA on $\mathcal{T}_\Sigma$ with $\Sigma = \{a, b\}$
and $stat_e = \{0, 1, 2, 3, 4, 5\}$
and $stat_n = \{\alpha, \beta, \gamma\}$



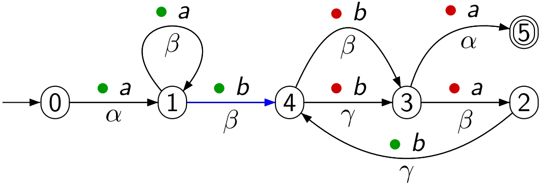
Streaming Tree Automata (STAs)



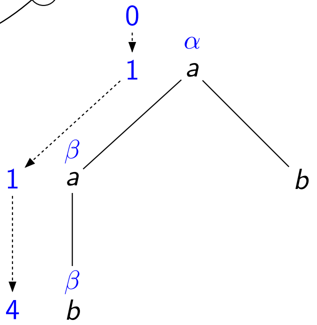
A: STA on $\mathcal{T}_\Sigma$ with $\Sigma = \{a, b\}$
and $stat_e = \{0, 1, 2, 3, 4, 5\}$
and $stat_n = \{\alpha, \beta, \gamma\}$



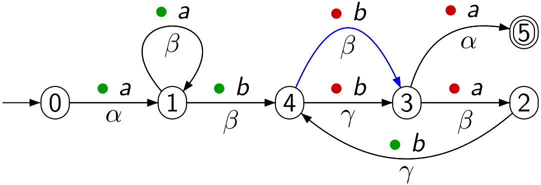
Streaming Tree Automata (STAs)



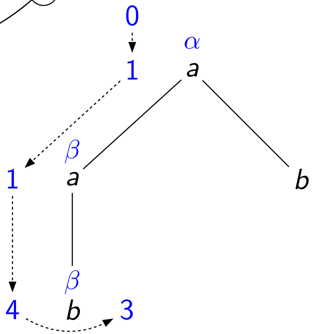
A: STA on $\mathcal{T}_\Sigma$ with $\Sigma = \{a, b\}$
and $stat_e = \{0, 1, 2, 3, 4, 5\}$
and $stat_n = \{\alpha, \beta, \gamma\}$



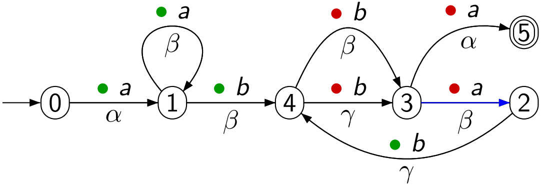
Streaming Tree Automata (STAs)



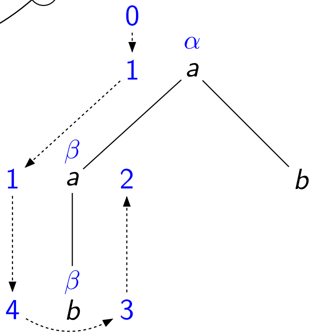
A: STA on $\mathcal{T}_\Sigma$ with $\Sigma = \{a, b\}$
and $stat_e = \{0, 1, 2, 3, 4, 5\}$
and $stat_n = \{\alpha, \beta, \gamma\}$



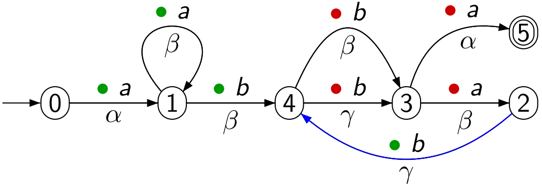
Streaming Tree Automata (STAs)



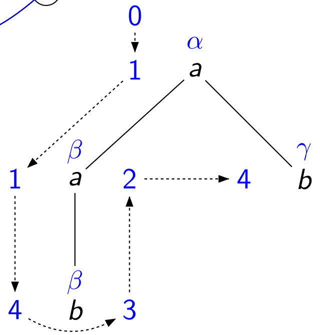
A: STA on $\mathcal{T}_\Sigma$ with $\Sigma = \{a, b\}$
and $stat_e = \{0, 1, 2, 3, 4, 5\}$
and $stat_n = \{\alpha, \beta, \gamma\}$



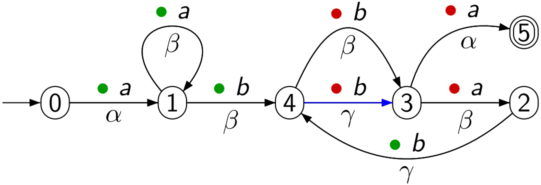
Streaming Tree Automata (STAs)

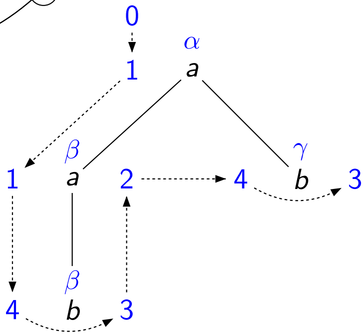


A: STA on $\mathcal{T}_\Sigma$ with $\Sigma = \{a, b\}$
and $stat_e = \{0, 1, 2, 3, 4, 5\}$
and $stat_n = \{\alpha, \beta, \gamma\}$



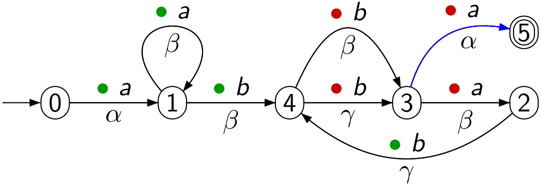
Streaming Tree Automata (STAs)



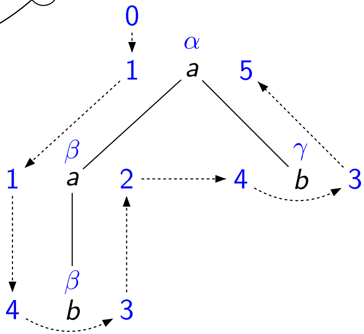


A: STA on $\mathcal{T}_\Sigma$ with $\Sigma = \{a, b\}$
and $stat_\mathrm{e} = \{0, 1, 2, 3, 4, 5\}$
and $stat_\mathrm{n} = \{\alpha, \beta, \gamma\}$

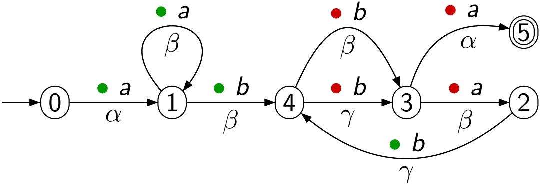
Streaming Tree Automata (STAs)



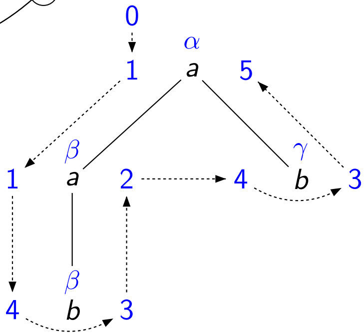
A: STA on $\mathcal{T}_\Sigma$ with $\Sigma = \{a, b\}$
and $stat_e = \{0, 1, 2, 3, 4, 5\}$
and $stat_n = \{\alpha, \beta, \gamma\}$



Streaming Tree Automata (STAs)



A: STA on $\mathcal{T}_\Sigma$ with $\Sigma = \{a, b\}$
and $stat_{\mathsf{e}} = \{0, 1, 2, 3, 4, 5\}$
and $stat_{\mathsf{n}} = \{\alpha, \beta, \gamma\}$



Deterministic STAs (dSTAs) respect the streaming order.

Streamability of Queries by dSTAs

Theorem

The class $\mathcal{Q}^{\delta}_{\text{dSTAs}}$ of queries defined by dSTAs on trees of depth at most δ is m-streamable for all $m \geq 0$.

- proved using an Earliest Query Answering algorithm
- $\mathcal{Q}^{\delta}_{\text{dSTAs}}$ is not ∞-streamable

Earliest Query Answering (EQA)

EQA algorithms:

- output selected nodes as soon as possible
- reject nodes that are not selected as soon as possible

In other words: only keep alive nodes in memory.

E_A: an STA for detecting sufficiency

A	$\rightarrow$	E_A
defines the query Q		defines the query Q
		detects earliest selection/rejection

E_A: an STA for detecting sufficiency

A	$\rightarrow$	E_A
defines the query Q		defines the query Q
		detects earliest selection/rejection

E_A: an STA for detecting sufficiency

A	$\rightarrow$	E_A
defines the query Q		defines the query Q
		detects earliest selection/rejection

- remark: for words, all states are already safe or unsafe...

E_A: an STA for detecting sufficiency

A	$\rightarrow$	E_A
defines the query Q		defines the query Q
		detects earliest selection/rejection

- remark: for words, all states are already safe or unsafe...
- ...but not for STAs: it depends on the context (i.e. the stack)

E_A: an STA for detecting sufficiency

A	$\rightarrow$	E_A
defines the query Q		defines the query Q detects earliest selection/rejection

- remark: for words, all states are already safe or unsafe...
- ...but not for STAs: it depends on the context (i.e. the stack)
- dynamic computation of safe states for selection and rejection

EQA for Queries by dSTAs

Problem: E_A has size exponential in $|A|$

- and we want a $\textsc{Ptime}$ algorithm

EQA for Queries by dSTAs

Problem: E_A has size exponential in $|A|$

- and we want a $\textsc{Ptime}$ algorithm

Solution: we build parts of E_A on the fly for the input tree t

- safe states are updated at every event in $\textsc{Ptime}$
- E_A is deterministic: we compute one run per alive node

EQA for Queries by dSTAs

Problem: E_A has size exponential in $|A|$

- and we want a PTIME algorithm

Solution: we build parts of E_A on the fly for the input tree t

- safe states are updated at every event in PTIME
- E_A is deterministic: we compute one run per alive node

Complexity

- PTIME precomputation
- PTIME per event and per alive node
- space = concurrency (alive nodes) + depth (stack)

$\rightarrow \mathcal{Q}^{\delta}_{\text{dSTAs}}$ is m-streamable for all $m \geq 0$.

XPath Streamability

- Forward XPath is not 0-streamable.

XPath Streamability

- Forward XPath is not 0-streamable.
- PTIME translation of a fragment of XPath to dSTAs implies its streamability.

XPath Streamability

- Forward XPath is not 0-streamable.
- PTIME translation of a fragment of XPath to dSTAs implies its streamability.
- the usual XPath $\rightarrow$ deterministic automata translation is doubly exponential (VARDI, WOLPER 94), (LIBKIN, SIRANGELO 08)

k-Downward XPath

=Downward XPath with the additional restrictions:

1. the total number of filters $[\ldots]$ is bounded by $k \geq 0$
2. all steps with ch^* have a label test (i.e. no $ch^*::*$)
3. if $ch^*::a$ appears, then no a-descendant of an a-node
4. bound on the depth of valid trees

PTIME Translation of k-Downward XPath to dSTAs

by induction on the structure of k-Downward XPath expressions

Example: $/ch^*::a[not(ch::c)]/ch::b$

- A_b checks whether the root is labeled by $(b, \{x\})$

PTIME Translation of k-Downward XPath to dSTAs

by induction on the structure of k-Downward XPath expressions

Example: $/ch^{*}::a[not(ch::c)]/ch::b$

- A_b checks whether the root is labeled by $(b, \{x\})$
- $A_{ch::b}$ runs A_b on every child of the root, and succeeds iff A_b succeeds at least once

PTIME Translation of k-Downward XPath to dSTAs

by induction on the structure of k-Downward XPath expressions

Example: $/ch^*::a[not(ch::c)]/ch::b$

- A_b checks whether the root is labeled by $(b, \{x\})$
- $A_{ch::b}$ runs A_b on every child of the root, and succeeds iff A_b succeeds at least once
- similarly for $A_{ch::c}$ (the label must be $(c, \emptyset)$ instead of $(b, \{x\})$)

PTIME Translation of k-Downward XPath to dSTAs

by induction on the structure of k-Downward XPath expressions

Example: $/ch^{*}::a[not(ch::c)]/ch::b$

- A_b checks whether the root is labeled by $(b, \{x\})$
- $A_{ch::b}$ runs A_b on every child of the root, and succeeds iff A_b succeeds at least once
- similarly for $A_{ch::c}$ (the label must be $(c, \emptyset)$ instead of $(b, \{x\})$)
- $A_{not(ch::c)}$ is the complement of $A_{ch::c}$

PTIME Translation of k-Downward XPath to dSTAs

by induction on the structure of k-Downward XPath expressions

Example: $/ch^*::a[not(ch::c)]/ch::b$

- A_b checks whether the root is labeled by $(b, \{x\})$
- $A_{ch::b}$ runs A_b on every child of the root, and succeeds iff A_b succeeds at least once
- similarly for $A_{ch::c}$ (the label must be $(c, \emptyset)$ instead of $(b, \{x\})$)
- $A_{not(ch::c)}$ is the complement of $A_{ch::c}$
- $A_{[not(ch::c)]/ch::b}$ is the intersection of $A_{not(ch::c)}$ and $A_{ch::b}$

PTIME Translation of k-Downward XPath to dSTAs

by induction on the structure of k-Downward XPath expressions

Example: $/ch^*::a[not(ch::c)]/ch::b$

- A_b checks whether the root is labeled by $(b, \{x\})$
- $A_{ch::b}$ runs A_b on every child of the root, and succeeds iff A_b succeeds at least once
- similarly for $A_{ch::c}$ (the label must be $(c, \emptyset)$ instead of $(b, \{x\})$)
- $A_{not(ch::c)}$ is the complement of $A_{ch::c}$
- $A_{[not(ch::c)]/ch::b}$ is the intersection of $A_{not(ch::c)}$ and $A_{ch::b}$
- $A_{/ch^*::a[not(ch::c)]/ch::b}$ looks for a-nodes, runs $A_{[not(ch::c)]/ch::b}$ for each of them, and succeeds iff one of them succeeded

PTIME Translation of k-Downward XPath to dSTAs

by induction on the structure of k-Downward XPath expressions

Example: $/ch^*::a[not(ch::c)]/ch::b$

- A_b checks whether the root is labeled by $(b, \{x\})$
- $A_{ch::b}$ runs A_b on every child of the root, and succeeds iff A_b succeeds at least once
- similarly for $A_{ch::c}$ (the label must be $(c, \emptyset)$ instead of $(b, \{x\})$)
- $A_{not(ch::c)}$ is the complement of $A_{ch::c}$
- $A_{[not(ch::c)]/ch::b}$ is the intersection of $A_{not(ch::c)}$ and $A_{ch::b}$
- $A_{/ch^*::a[not(ch::c)]/ch::b}$ looks for a-nodes, runs $A_{[not(ch::c)]/ch::b}$ for each of them, and succeeds iff one of them succeeded

Thanks to the restrictions, all steps preserve determinism, and the construction is in PTIME.

Known algorithms for streaming XPath

Fragment	0-str.	m-str. $\forall m$	∞-str.	look-ahead

Known algorithms for streaming XPath

Fragment	0-str.	m-str. $\forall m$	∞-str.	look-ahead
Downward XPath (Ramanan 05) (Bar-Yossef, F., J. 05) (Gou, Chirkova 07)	×	×	×	✓

Known algorithms for streaming XPath

Fragment		0-str.	m-str. $\forall m$	∞-str.	look-ahead
Downward XPath	(Ramanan 05) (Bar-Yossef, F., J. 05) (Gou, Chirkova 07)	×	×	×	✓
Forward XPath	(Olteanu 07)	×	×	×	✓

Known algorithms for streaming XPath

Fragment	0-str.	m-str. $\forall m$	∞-str.	look-ahead
Downward XPath (Ramanan 05) (Bar-Yossef, F., J. 05) (Gou, Chirkova 07)	×	×	×	✓
Forward XPath (Olteanu 07)	×	×	×	✓
k-Downward XPath (G., Niehren 09)	✓	✓	×	✓

Known algorithms for streaming XPath

Fragment	0-str.	*m*-str. $\forall m$	∞-str.	look-ahead
Downward XPath (Ramanan 05) (Bar-Yossef, F., J. 05) (Gou, Chirkova 07)	×	×	×	✓
Forward XPath (Olteanu 07)	×	×	×	✓
k-Downward XPath (G., Niehren 09)	✓	✓	×	✓
Strict Backward XUntil (Benedikt, Jeffrey 07)	✓	✓	✓	×

etc.

Conclusion

Main contributions

Streamability

- SRAMs model for query answering algorithms on streams
- Streamability measure
  - Hardness results
- Testing bounded concurrency
  - Hardness results
  - PTIME procedure for queries by deterministic STAs (LATA'09)

Main contributions

Streamability

- SRAMs model for query answering algorithms on streams
- Streamability measure
  - Hardness results
- Testing bounded concurrency
  - Hardness results
  - PTIME procedure for queries by deterministic STAs (LATA'09)

Streamable fragments

- Queries by deterministic STAs (IPL'08)
  - Earliest Query Answering algorithm (FCT'09)
- *k*-Downward XPath

Perspectives

Future Work

- implementations
  - our algorithms focus on low memory consumption
  - this requires additional time

Perspectives

Future Work

- implementations
  - our algorithms focus on low memory consumption
  - this requires additional time
- XProc

Perspectives

Future Work

- implementations
  - our algorithms focus on low memory consumption
  - this requires additional time
- XProc

Open Questions

- How to relax (approximate?) the earliest condition?

Perspectives

Future Work

- implementations
  - our algorithms focus on low memory consumption
  - this requires additional time
- XProc

Open Questions

- How to relax (approximate?) the earliest condition?
- Can we extend the fragments (more XPath axes, etc.)?

Perspectives

Future Work

- implementations
  - our algorithms focus on low memory consumption
  - this requires additional time
- XProc

Open Questions

- How to relax (approximate?) the earliest condition?
- Can we extend the fragments (more XPath axes, etc.)?
- Are there logical characterizations of streamable query classes?
  - bounded concurrency, bounded delay, etc.

Perspectives

Future Work

- implementations
  - our algorithms focus on low memory consumption
  - this requires additional time
- XProc

Open Questions

- How to relax (approximate?) the earliest condition?
- Can we extend the fragments (more XPath axes, etc.)?
- Are there logical characterizations of streamable query classes?
  - bounded concurrency, bounded delay, etc.
- Can we extend these results to transformations?

Thank you