Complexity of Earliest Query Answering with Streaming Tree Automata

Olivier Gauwin  Anne-Cécile Caron  Joachim Niehren  Sophie Tison

INRIA Lille, Mostrare

9th January 2008
Streaming for XML

Stream: `<people><external> . . . `
Query Answering in Streaming

```
<people><external> ... 

people
  
  external ... student
    
    name ... mail
      
      "Paul" "John" "j@mail.org"

query

ul
  
  li ... 
    
    b i
      
      "John" "j@mail.org"

<ul><li> ... 

output data
```
Related work on streaming

- **XQuery:**
  - Schmidt, Scherzinger & Koch (07)
  - Stark, Fernandez, Michiels & Siméon (07)

- **Transducers:**
  - Frisch & Nakano (07)
  - Nakano (04)

- **XPath:**
  - Gupta & Suciu (03)
  - Green, Miklau & Onizuka (03)
  - Bar-Yossef, Fontoura & Josifovski (04)
  - Grohe, Koch & Schweikardt (05)
XPath query: /people/*[mail]

Earliest selection:
when <mail> is read

Earliest failure:
when </external> is read
Related work on Earliest Query Answering

- **Automata:**
  - Berlea (06, 07): queries defined Pre-Order Automata

- **XPath:**
  - Olteanu (07): Forward XPath
  - Benedikt & Jeffrey (07): XPath filters with zero lookahead
What is missing?

*n*-ary queries \((n \geq 0)\)

- collect *n*-tuples of nodes, like in XQuery
- \(q(t) \subseteq \text{nodes}(t)^n\)

Select pairs (name, mail)

```
people
  ┌─┐  ┌─┐
  | |   | |
external ∖ student
  └─┘  └─┘
  ┌─┐  ┌─┐  ┌─┐  ┌─┐
  | |   | |   | |
name ∖ name ∖ mail
  └─┘  └─┘  └─┘
  ┌─┐  ┌─┐
  | |   | |
"Paul" ∖ "John" ∖ "j@mail.org"
```

```
ul
  ┌─┐  ┌─┐
  | |   | |
li ∖ ∖ ∖
  └─┘  └─┘
  ┌─┐  ┌─┐
  | |   | |
b ∖ i
  └─┘  └─┘
  ┌─┐  ┌─┐
  | |   | |
"John" ∖ "j@mail.org"
```
What is missing?

**schema:** safe for *valid* continuations

▶ example:

**XPath query:** `/people/*[mail]`

- `people`
  - `external`
    - `name`
      - “Paul”
    - `… student`
      - `name`
      - “John”
      - `mail`
      - “j@mail.org”

**DTD**

```
DTD {
  external → name
  student → name ... mail
  ...
```

**Earliest selection:**
when `<student>` is read

**Earliest failure:**
when `<external>` is read
What is missing?

**schema:** safe for *valid* continuations

- example:

  **XPath query:** /people/*[mail]

```
people
  └── external...
    └── student...
      └── name...
          └── mail
            └── “Paul” “John” “j@mail.org”
```

**DTD**

```
\{
  external → name
  student → name ... mail
...
```

**Earliest selection:**
when `<student>` is read

**Earliest failure:**
when `<external>` is read

Consequence: MSO queries defined by automata
Fundamental questions

- how to define *earliest selection* and *earliest failure*?
Fondamental questions

- how to define *earliest selection* and *earliest failure*?
- is there an algorithm to compute these earliest positions?
Fundamental questions

- How to define *earliest selection* and *earliest failure*?
- Is there an algorithm to compute these earliest positions?
- How complex is it?
1. Earliest Query Answering

2. Streaming Tree Automata

3. Streaming Algorithm
1. **Earliest Query Answering**

2. **Streaming Tree Automata**

3. **Streaming Algorithm**
Notations

Trees and events

- \( \text{events}(t) = \{ \text{start} \} \cup (\{ \text{open}, \text{close} \} \times \text{nodes}(t)) \)
**Notations**

**Trees and events**

- $\text{events}(t) = \{\text{start}\} \cup (\{\text{open, close}\} \times \text{nodes}(t))$
- $t^\leq_e$: the tree which contains all nodes of $t$ opened before $e$
Notations

Trees and events

- \text{events}(t) = \{\text{start}\} \cup (\{\text{open, close}\} \times \text{nodes}(t))
- \ t^{\leq e} \text{: the tree which contains all nodes of } t \text{ opened before } e
- \text{equal}_e(t, t') \iff \begin{cases} e \in \text{events}(t) \cap \text{events}(t') \\ \text{and} \\ t^{\leq e} = t'^{\leq e} \end{cases}

\begin{align*}
t = & \quad \pi_0 \quad \pi_1 \quad \pi_2 \\
& \quad \pi_0 \quad \pi_3 \quad \pi_1 \quad \pi_2 \\
& \quad \pi_0 \quad \pi_3 \quad \pi_4

t' = & \quad \pi_0 \quad \pi_3 \quad \pi_2
\end{align*}

\text{equal}_{\text{close}, \pi}(t, t') \quad \text{and} \quad \neg \text{equal}_{\text{open}, \pi}(t, t')
Sufficient events for selection

$$(\tau, e) \in \text{sel}_q^S(t) \iff \left\{ \begin{array}{l} \tau \in \text{nodes}(t \leq e)^n \land \\
\forall t' \in S. \text{equal}_e(t, t') \Rightarrow \tau \in q(t') \end{array} \right.$$ 

Example

$q_0 = "select \text{ nodes that don't have a next sibling}"$

$\text{sel}_{q_0}(t)$ contains:

- $(\pi_0, e)$ for events $e$ following $(\text{open}, \pi_0)$
- $(\pi_2, e)$ for events $e$ following $(\text{close}, \pi_1)$
- $(\pi_3, (\text{close}, \pi_0))$
Sufficient events for selection

\[(\tau, e) \in \text{sel}^S_q(t) \iff \begin{cases} \tau \in \text{nodes}(t^{\leq e})^n \land \\ \forall t' \in S. \text{equal}_e(t, t') \Rightarrow \tau \in q(t') \end{cases} \]

Example

\(q_0 = \) “select nodes that don’t have a next sibling”

consider the DTD: \(S_0 = \{ a \rightarrow a^* b, \ b \rightarrow \epsilon \}\)

\(\text{sel}_{q_0}(t)\) contains:

- \((\pi_0, e)\) for events \(e\) following \((\text{open, } \pi_0)\)
- \((\pi_2, e)\) for events \(e\) following \((\text{close, } \pi_1)\)
- \((\pi_3, (\text{close, } \pi_0))\)
Optimal events for selection

\[(\tau, e) \in \text{opt}_{sel_q}^S(t) \iff e = \min_{\preceq t} \{e' \mid (\tau, e') \in \text{sel}_q^S(t)\}\]

**Example**

\[q_0 = \text{“select nodes that don’t have a next sibling”}\]

\[
\text{opt}_{sel_{q_0}}(t) = \begin{cases} 
(\pi_0, (\text{open}, \pi_0)), \\
(\pi_2, (\text{close}, \pi_1)), \\
(\pi_3, (\text{close}, \pi_0))
\end{cases}
\]

\[
\begin{array}{c}
\pi_0 \\
a \\
\pi_1 \\
a \\
b \\
\pi_2 \\
b \\
\pi_3 \\
\end{array}
\]
Optimal events for selection

\[(\tau, e) \in \text{opt}_q \Rightarrow e = \min \{ e' | (\tau, e') \in \text{sel}_q \} \]

**Example**

\[q_0 = "\text{select nodes that don't have a next sibling}"\]

Consider the DTD: \[S_0 = \{ a \rightarrow a^* b, \ b \rightarrow \epsilon \}\]

\[
\begin{align*}
\text{opt}_q(t) = \begin{cases} 
(\pi_0, (\text{open}, \pi_0)), \\
(\pi_2, (\text{close}, \pi_1)), \\
(\pi_3, (\text{close}, \pi_0)) 
\end{cases} \\
\text{opt}_q(t) = \begin{cases} 
(\pi_0, (\text{open}, \pi_0)), \\
(\pi_2, (\text{open}, \pi_2)), \\
(\pi_3, (\text{open}, \pi_3)) 
\end{cases}
\end{align*}
\]
Optimal events for selection

\[(\tau, e) \in \text{opt\_sel}_q^S(t) \iff e = \min_{\leq t} \{e' \mid (\tau, e') \in \text{sel}_q^S(t)\}\]

Example

\[q_0 = "\text{select nodes that don't have a next sibling}"\]

Consider the DTD: \(S_0 = \{a \rightarrow a^* b, \ b \rightarrow \epsilon\}\)

\[
\text{opt\_sel}_{q_0}(t) = \begin{cases} 
(\pi_0, (\text{open, } \pi_0)), \\
(\pi_2, (\text{close, } \pi_1)), \\
(\pi_3, (\text{close, } \pi_0))
\end{cases}
\]

Similarly, we define sufficient and optimal events for failure.
Complexity of Sufficiency: for MSO queries

**Sufficiency problem**: Decide whether $(\tau, e) \in \text{sel}_q^S(t)$

**Queries defined by MSO formula**

The sufficiency problem for MSO-defined queries is decidable but non-elementary.

**Queries defined by tree automata**

Sufficiency and optimality for queries defined by non-deterministic tree automata are DEXPTIME-hard.
Complexity of Sufficiency: for MSO queries

Sufficiency problem: Decide whether $(\tau, e) \in \text{sel}^S_q(t)$

Queries defined by MSO formula

The sufficiency problem for MSO-defined queries is decidable but non-elementary.

Queries defined by tree automata

Sufficiency and optimality for queries defined by non-deterministic tree automata are DEXPTIME-hard.

- Sufficiency for failure: same complexity (not obvious).
Complexity of Sufficiency: for MSO queries

Sufficiency problem: Decide whether \((\tau, e) \in \text{sel}^S_q(t)\)

Queries defined by MSO formula

The sufficiency problem for MSO-defined queries is decidable but non-elementary.

Queries defined by tree automata

Sufficiency and optimality for queries defined by non-deterministic tree automata are DEXPTIME-hard.

- Sufficiency for failure: same complexity (not obvious).
- We proved hardness results for other classes of queries.
Determinism

Wanted: deterministic automata when evaluated in streaming order
Determinism

Wanted: deterministic automata when evaluated in streaming order

- bottom-up / top-down automata: does not fit
Determinism

Wanted: deterministic automata when evaluated in streaming order

- bottom-up / top-down automata: does not fit
- document order: Nested Word Automata
  - problem: queries on events, not nodes
Determinism

Wanted: deterministic automata when evaluated in streaming order

- bottom-up / top-down automata: does not fit
- document order: Nested Word Automata
  - problem: queries on events, not nodes
- we define Streaming Tree Automata
1 Earliest Query Answering

2 Streaming Tree Automata

3 Streaming Algorithm
Streaming Tree Automata

A: STA on $T_{\Sigma}$ with $\Sigma = \{a, b\}$ and $\Gamma = \{\gamma_1, \gamma_2\}$

states = \{p_0, p_1, p_2\} \quad \text{init} = \{p_0\} \quad \text{final} = \{p_1, p_2\}

rules = \{
\begin{align*}
\text{open } a & \ p_0 \rightarrow p_0 \ \gamma_1 \\
\text{open } b & \ p_0 \rightarrow p_0 \ \gamma_2 \\
\text{open } a & \ p_2 \rightarrow p_2 \ \gamma_1 \\
\text{open } b & \ p_2 \rightarrow p_2 \ \gamma_2 \\
\text{close } a & \ p_0 \ \gamma_1 \rightarrow p_0 \\
\text{close } b & \ p_0 \ \gamma_2 \rightarrow p_1 \\
\text{close } a & \ p_2 \ \gamma_1 \rightarrow p_2 \\
\text{close } b & \ p_2 \ \gamma_2 \rightarrow p_2
\end{align*}
\}

Stack:

\[
\begin{array}{c}
\end{array}
\]
Streaming Tree Automata

\( A: \text{STA on } T_\Sigma \text{ with } \Sigma = \{a, b\} \text{ and } \Gamma = \{\gamma_1, \gamma_2\} \)

\( \text{states} = \{p_0, p_1, p_2\} \quad \text{init} = \{p_0\} \quad \text{final} = \{p_1, p_2\} \)

\( \text{rules} = \{ \)

\begin{align*}
\text{open } a & \; p_0 \rightarrow p_0 \; \gamma_1 \\
\text{open } b & \; p_0 \rightarrow p_0 \; \gamma_2 \\
\text{open } a & \; p_2 \rightarrow p_2 \; \gamma_1 \\
\text{open } b & \; p_2 \rightarrow p_2 \; \gamma_2 \\
\text{close } a & \; p_0 \; \gamma_1 \rightarrow p_0 \\
\text{close } b & \; p_0 \; \gamma_2 \rightarrow p_1 \\
\text{close } a & \; p_2 \; \gamma_1 \rightarrow p_2 \\
\text{close } b & \; p_2 \; \gamma_2 \rightarrow p_2 \\
\end{align*} \)

Stack:

\[
\begin{array}{c}
\gamma_1 \\
\end{array}
\]
Streaming Tree Automata

A: STA on $T_\Sigma$ with $\Sigma = \{a, b\}$ and $\Gamma = \{\gamma_1, \gamma_2\}$

states = \{\(p_0, p_1, p_2\}\} \quad \text{init} = \{\(p_0\)\} \quad \text{final} = \{\(p_1, p_2\)\}

rules = \{
  \text{open } a \quad p_0 \rightarrow p_0 \quad \gamma_1
  \text{open } b \quad p_0 \rightarrow p_0 \quad \gamma_2
  \text{open } a \quad p_2 \rightarrow p_2 \quad \gamma_1
  \text{open } b \quad p_2 \rightarrow p_2 \quad \gamma_2
  \text{close } a \quad p_0 \quad \gamma_1 \rightarrow p_0
  \text{close } b \quad p_0 \quad \gamma_2 \rightarrow p_1
  \text{close } a \quad p_2 \quad \gamma_1 \rightarrow p_2
  \text{close } b \quad p_2 \quad \gamma_2 \rightarrow p_2
\}

Stack: \[
\begin{array}{c}
\gamma_1 \\
\gamma_1
\end{array}
\]
Streaming Tree Automata

A: STA on $T_\Sigma$ with $\Sigma = \{a, b\}$ and $\Gamma = \{\gamma_1, \gamma_2\}$
states = $\{p_0, p_1, p_2\}$  init = $\{p_0\}$  final = $\{p_1, p_2\}$

rules = {
  open $a$ $p_0 \rightarrow p_0$ $\gamma_1$
  open $b$ $p_0 \rightarrow p_0$ $\gamma_2$
  open $a$ $p_2 \rightarrow p_2$ $\gamma_1$
  open $b$ $p_2 \rightarrow p_2$ $\gamma_2$
  close $a$ $p_0$ $\gamma_1 \rightarrow p_0$
  close $b$ $p_0$ $\gamma_2 \rightarrow p_1$
  close $a$ $p_2$ $\gamma_1 \rightarrow p_2$
  close $b$ $p_2$ $\gamma_2 \rightarrow p_2$
}

Stack: $\begin{pmatrix} \gamma_2 \\ \gamma_1 \\ \gamma_1 \end{pmatrix}$
Streaming Tree Automata

A: STA on $T_\Sigma$ with $\Sigma = \{a, b\}$ and $\Gamma = \{\gamma_1, \gamma_2\}$

states = $\{p_0, p_1, p_2\}$  init = $\{p_0\}$  final = $\{p_1, p_2\}$

rules = 

open $a$ $p_0 \rightarrow p_0 \gamma_1$
open $b$ $p_0 \rightarrow p_0 \gamma_2$
open $a$ $p_2 \rightarrow p_2 \gamma_1$
open $b$ $p_2 \rightarrow p_2 \gamma_2$
close $a$ $p_0 \gamma_1 \rightarrow p_0$
close $b$ $p_0 \gamma_2 \rightarrow p_1$
close $a$ $p_2 \gamma_1 \rightarrow p_2$
close $b$ $p_2 \gamma_2 \rightarrow p_2$

Stack: 

$\gamma_1$

$\gamma_1$

$\gamma_1$
Streaming Tree Automata

$A$: STA on $T_\Sigma$ with $\Sigma = \{a, b\}$ and $\Gamma = \{\gamma_1, \gamma_2\}$

states = $\{p_0, p_1, p_2\}$ init = $\{p_0\}$ final = $\{p_1, p_2\}$

rules = $\{$
- open $a$ $p_0 \rightarrow p_0 \gamma_1$
- open $b$ $p_0 \rightarrow p_0 \gamma_2$
- open $a$ $p_2 \rightarrow p_2 \gamma_1$
- open $b$ $p_2 \rightarrow p_2 \gamma_2$
- close $a$ $p_0 \gamma_1 \rightarrow p_0$
- close $b$ $p_0 \gamma_2 \rightarrow p_1$
- close $a$ $p_2 \gamma_1 \rightarrow p_2$
- close $b$ $p_2 \gamma_2 \rightarrow p_2$\}

Stack:
$\gamma_1$
Streaming Tree Automata

A: STA on $T_{\Sigma}$ with $\Sigma = \{a, b\}$ and $\Gamma = \{\gamma_1, \gamma_2\}$
states = $\{p_0, p_1, p_2\}$ init = $\{p_0\}$ final = $\{p_1, p_2\}$

rules = 
- open $a$ $p_0 \rightarrow p_0 \gamma_1$
- open $b$ $p_0 \rightarrow p_0 \gamma_2$
- open $a$ $p_2 \rightarrow p_2 \gamma_1$
- open $b$ $p_2 \rightarrow p_2 \gamma_2$
- close $a$ $p_0 \gamma_1 \rightarrow p_0$
- close $b$ $p_0 \gamma_2 \rightarrow p_1$
- close $a$ $p_2 \gamma_1 \rightarrow p_2$
- close $b$ $p_2 \gamma_2 \rightarrow p_2$

Stack: 
\[
\begin{array}{c}
\gamma_2 \\
\gamma_1
\end{array}
\]
Streaming Tree Automata

$A$: STA on $T_\Sigma$ with $\Sigma = \{a, b\}$ and $\Gamma = \{\gamma_1, \gamma_2\}$
states = $\{p_0, p_1, p_2\}$ init = $\{p_0\}$ final = $\{p_1, p_2\}$

rules = {$$
\text{open } a \ p_0 \rightarrow p_0 \ \gamma_1 \\
\text{open } b \ p_0 \rightarrow p_0 \ \gamma_2 \\
\text{open } a \ p_2 \rightarrow p_2 \ \gamma_1 \\
\text{open } b \ p_2 \rightarrow p_2 \ \gamma_2 \\
\text{close } a \ p_0 \ \gamma_1 \rightarrow p_0 \\
\text{close } b \ p_0 \ \gamma_2 \rightarrow p_1 \\
\text{close } a \ p_2 \ \gamma_1 \rightarrow p_2 \\
\text{close } b \ p_2 \ \gamma_2 \rightarrow p_2$$

Stack: $\gamma_1$
Streaming Tree Automata

$A$: STA on $T_\Sigma$ with $\Sigma = \{a, b\}$ and $\Gamma = \{\gamma_1, \gamma_2\}$

states = \{p_0, p_1, p_2\} \quad \text{init} = \{p_0\} \quad \text{final} = \{p_1, p_2\}$

rules = \{
    \text{open } a \ p_0 \rightarrow p_0 \gamma_1 \\
    \text{open } b \ p_0 \rightarrow p_0 \gamma_2 \\
    \text{open } a \ p_2 \rightarrow p_2 \gamma_1 \\
    \text{open } b \ p_2 \rightarrow p_2 \gamma_2 \\
    \text{close } a \ p_0 \gamma_1 \rightarrow p_0 \\
    \text{close } b \ p_0 \gamma_2 \rightarrow p_1 \\
    \text{close } a \ p_2 \gamma_1 \rightarrow p_2 \\
    \text{close } b \ p_2 \gamma_2 \rightarrow p_2 \}

Stack: $\begin{bmatrix} \end{bmatrix}$
Implications on Complexity

Deterministic STAs

The sufficiency problem of queries defined by deterministic STAs is in PTIME combined complexity.

Non-deterministic STAs

The sufficiency problem of queries represented by non-deterministic STAs is DEXPTIME-complete.

hardness is known, membership is obtained by determinization
1 Earliest Query Answering

2 Streaming Tree Automata

3 Streaming Algorithm
A basic algorithm

Signature

- **Input:**
  - a query $q$ and a schema $S$
  - the stream of events of a tree $t \in T_{\Sigma}$

- **Output:**
  - all the tuples of $q(t)$ at their earliest event w.r.t. $S$
A basic algorithm

**Signature**

- **Input:**
  - a query $q$ and a schema $S$
  - the stream of events of a tree $t \in T_\Sigma$

- **Output:**
  - all the tuples of $q(t)$ at their earliest event w.r.t. $S$

**Computation**

- generate all candidate tuples on the fly
- if a candidate is sufficient for selection, output it and remove it

This algorithm is clearly earliest.
Reducing space consumption

Concurrency

- \( \tau \) is *alive* at event \( e \) if it is neither selected nor failed at \( e \)
- the *concurrency* is the maximal number of candidates that are alive at a same event

introduced by Bar-Yossef, Fontoura, Josifovski (without schema)

Complexity

- by testing for *failures*, we can discard some candidates
- we only keep alive candidates
- space complexity bounded by concurrency
An efficient algorithm for STAs

Main ideas

- input: deterministic STA $A$ recognizing the canonical language of $q$

At each event and for each alive candidate:
- memoize the state reached in $A$
- compute safe states for selection and failure

2 steps

1. Logic: build the STA $E(A)$ that recognizes sufficiency
2. Algorithm: compute safe states
E(A): an STA for detecting sufficiency

E(A) = A + additional information on sufficiency

- states of E(A) are of the form \((p, S, F)\) where:
  - \(p \in \text{states}^A\)
  - \(S \subseteq \text{states}^A\) are safe states for selection
  - \(F \subseteq \text{states}^A\) are safe states for failure
$E(A)$: an STA for detecting sufficiency

$E(A) = A +$ additional information on sufficiency

- states of $E(A)$ are of the form $(p, S, F)$ where:
  - $p \in \text{states}^A$
  - $S \subseteq \text{states}^A$ are safe states for selection
  - $F \subseteq \text{states}^A$ are safe states for failure
- let $r(e) = (p, S, F)$, $r$ being a run on $t$ for one candidate $\tau$
  - $e$ is sufficient for selection iff $p \in S$
  - $e$ is sufficient for failure iff $p \in F$
E(A): an STA for detecting sufficiency

E(A) = A + additional information on sufficiency

• states of E(A) are of the form (p, S, F) where:
  ▶ p ∈ states
  ▶ S ⊆ states are safe states for selection
  ▶ F ⊆ states are safe states for failure

• let r(e) = (p, S, F), r being a run on t for one candidate τ
  ▶ e is sufficient for selection iff p ∈ S
  ▶ e is sufficient for failure iff p ∈ F

⇒ Sufficiency is MSO-definable!
Efficient algorithm

Overview

- $A$ is deterministic $\Rightarrow E(A)$ is deterministic
- we don’t build $E(A)$, but its runs on the fly
- to do so, we compute the safe states in 2 steps
  - precomputation of an accessibility relation (in PTIME)
  - at reception of an event, use this relation to compute the safe states (in PTIME)
use this formalization on some decision problems
earliest query answering problem of Forward XPath with schemas
find other fragments suitable for streaming
improve data structure for candidates (Meuss, Schultz and Bry, 2001)
queries defined by selection automata
Thank you!