# XML and Databases

**Lecture 8**
*Streaming Evaluation: how much memory do you need?*

Sebastian Maneth
NICTA and UNSW

*CSE@UNSW -- Semester 1, 2009*

Small XPath Quiz

Can you give an expression that returns the last / first occurrence of each distinct price element?

```
<b>
<price>3</price>
<price>1</price>
<price>3</price>
<price>1</price>
<price>3</price>
<price>4</price>
<price>1</price>
<price>7</price>
</b>
```

| Should return (last) | Should return (first) |
| --- | --- |
| <price>3</price> | <price>3</price> |
| <price>4</price> | <price>1</price> |
| <price>1</price> | <price>4</price> |
| <price>7</price> | <price>7</price> |

Small XPath Quiz

Can you give an expression that returns the last / first occurrence of each distinct price element?

```
<b>
<price>3.0</price>
<price>1</price>
<price>3.00</price>
<price>1</price>
<price>3</price>
<price>4</price>
<price>1.000</price>
<price>7</price>
</b>
```

Should return

```
<price>3</price>
<price>4</price>
<price>1.000</price>
<price>7</price>
```

Should return

```
<price>3.0</price>
<price>1</price>
<price>4</price>
<price>7</price>
```

What if we mean *number-distinctness* (not strings)?

0. Recall

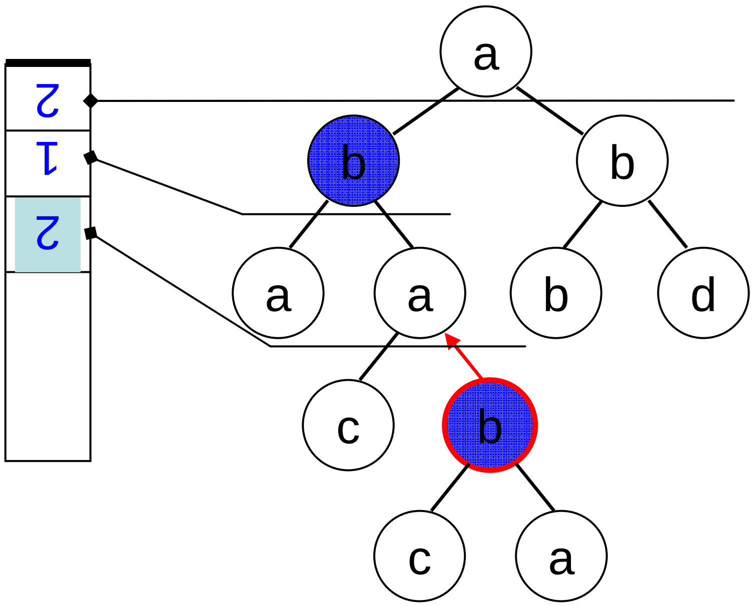
→ Evaluation of Simple Paths //a/b/c
→ Arbitrary Queries over //, /, *

# Outline

1. Automaton Approach
2. Parallel Evaluation of Multiple Queries
3. Sizes of Automata
4. How to deal with Filters
5. Existing Systems for Streaming XPath Evaluation

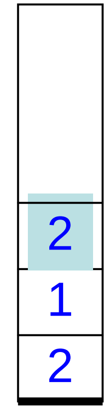
# Recall: Top-Down Evaluation of *Simple Paths*

→ evaluate in *one single pre-order traversal* (using a **stack)**




```
//a/b    =Q
```

query match position: p = 2

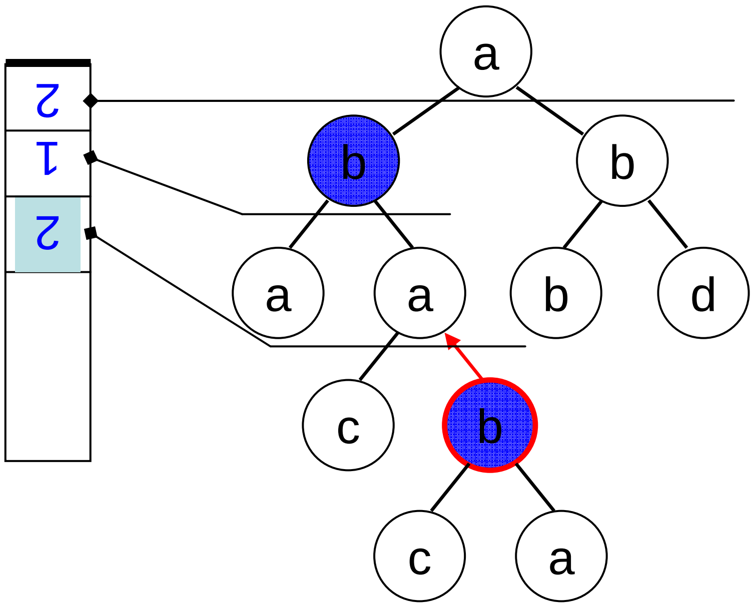



[endElement( b )] p = pop() = 2

[startElement( a )]
[startElement( b )] *result node*
[startElement( a )]
[endElement( a )]
[startElement( a )]
[startElement( c )]
[endElement( c )]
[startElement( b )] *result node*
[startElement( c )] push(1)
[endElement( c )] p = pop() = 1
[startElement( a )] push(1)
[endElement( a )] p = pop() = 1

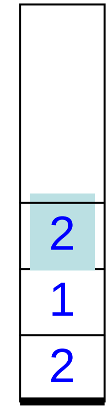
# Recall: Top-Down Evaluation of *Simple Paths*

→ evaluate in *one single pre-order traversal* (using a **stack)**




`//a/b    =Q`

query match position: p = 2




Streaming Algorithm!

➔ No need to store the document!!
Can evaluate on SAX event stream.

Simple Path `//a_1/a_2/a_3/ . . . /a_n`

**TIME** **one pass** through document tree.
**SPACE** *stack of query positions*.
height is bounded by depth of document tree.

BUT

Need **output buffers**, if subtrees of match nodes should be printed!

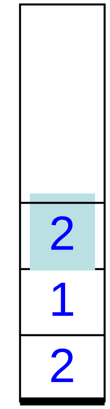
# Recall: Top-Down Evaluation of *Simple Paths*

→ evaluate in *one single pre-order traversal* (using a **stack)**

If we print **node-IDs**, then no output buffers are needed!

`//a/b =Q`

query match position: p = 2




➔ True Streaming, with memory need proportional to height.

Streaming Algorithm!

➔ No need to store the document!! Can evaluate on SAX event stream.

Simple Path `//a_1/a_2/a_3/ . . . /a_n`

**TIME** **one pass** through document tree.
**SPACE** *stack of query positions*.
height is bounded by depth of document tree.

BUT

Need **output buffers**, if subtrees of match nodes should be printed!

# Recall: Top-Down Evaluation of *Simple Paths*

→ evaluate in *one single pre-order traversal* (using a **stack)**

If we print **node-IDs**, then no output buffers are needed!

`//a/b =Q`

2
1
2

query match position: p = 2

➔ any good implementation of this algorithm should work for documents with *depth up to a couple of millions*, and
**NO restriction on document size**!

Streaming Algorithm!

➔ No need to store the document!!
Can evaluate on SAX event stream.

Simple Path `//a_1/a_2/a_3/ . . . /a_n`

**TIME** **one pass** through document tree.
**SPACE** *stack of query positions*.
height is bounded by depth of document tree.

1 Byte is enough for small queries!

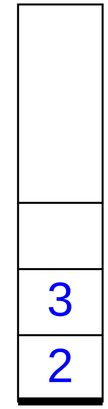
# Arbitrary *Slash+Slashslash*

→ evaluate in *one single pre-order traversal* (using a **stack)**

Arbitrary queries with `/,//,*`

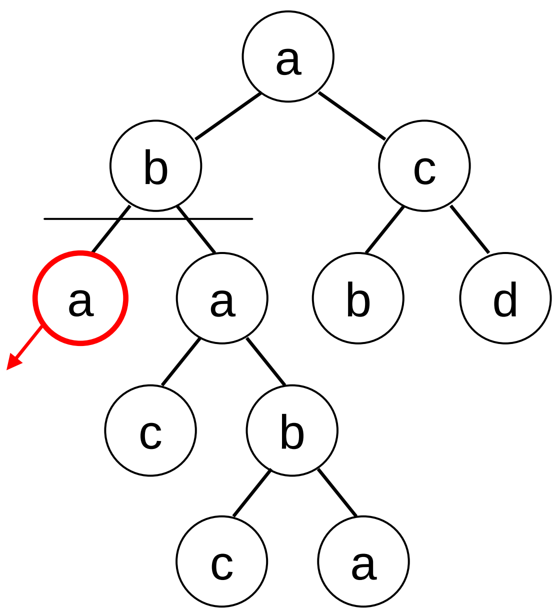
multiple `//`'s

`//a/b//c`



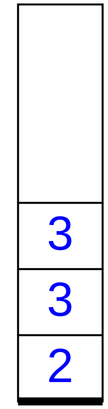

query match position: p = 3

no match
stay in p=3!




…
[startElement( a )] push(3)
[endElement( a )] p = pop() = 3

# Arbitrary *Slash+Slashslash*




→ evaluate in *one single pre-order traversal* (using a **stack)**

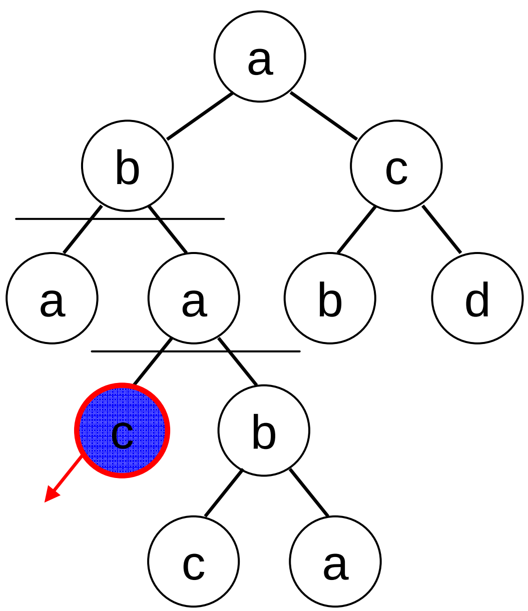
Arbitrary queries with `/,//,*`

multiple `//`’s

`//a/b//c`

query match position: p = 3

no match
stay in p=3!

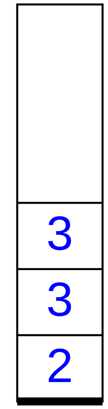



…
[startElement( a )] push(3)
[endElement( a )] p = pop() = 3
[startElement( a )] push(3)
[startElement( c )] push(3)

Result node!
Mark it, and stay in p=3.

# Arbitrary *Slash+Slashslash*

→ evaluate in *one single pre-order traversal* (using a **stack)**



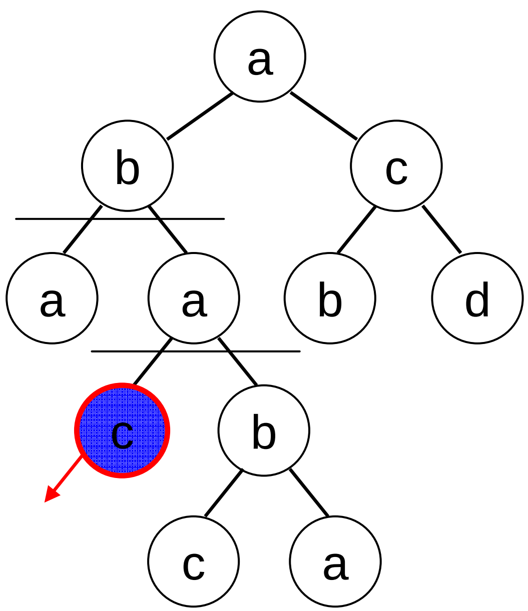

Arbitrary queries with `/,//,*`

multiple `//`'s

`//a/b//c`

query match position: p = 3

no match
stay in p=3!



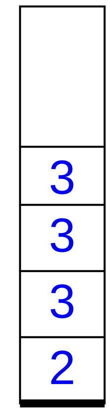

…
[startElement( a )] push(3)
[endElement( a )] p = pop() = 3
[startElement( a )] push(3)
[startElement( c )] push(3)

Result node!
Mark it, and stay in p=3.

*Output Node-ID* *Start copying to Output Buffer*

# Arbitrary *Slash+Slashslash*

→ evaluate in *one single pre-order traversal* (using a **stack)**



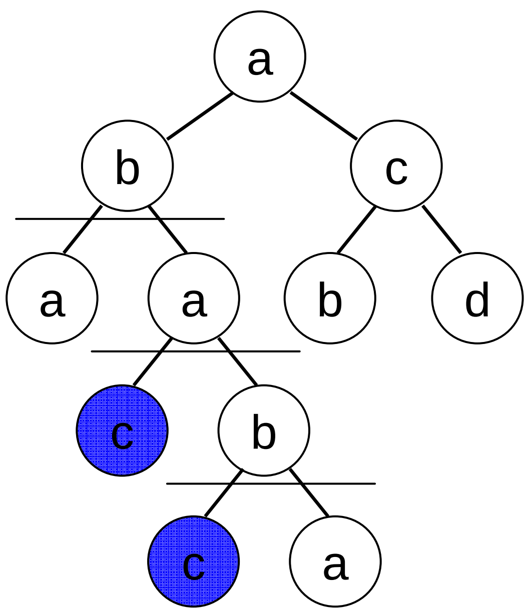

Arbitrary queries with `/,//,*`

multiple `//`'s

`//a/b//c`

query match position: p = 3

no match
stay in p=3!




…
[startElement( a )] push(3)
[endElement( a )] p = pop() = 3
[startElement( a )] push(3)
[startElement( c )] push(3)
[endElement( c )] p = pop() = 3
[startElement( b )] push(3)
[startElement( c )] push(3)
…

# Arbitrary *Slash+Slashslash*

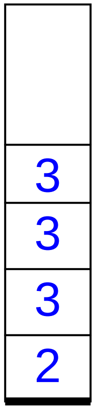
→ evaluate in *one single pre-order traversal* (using a **stack)**

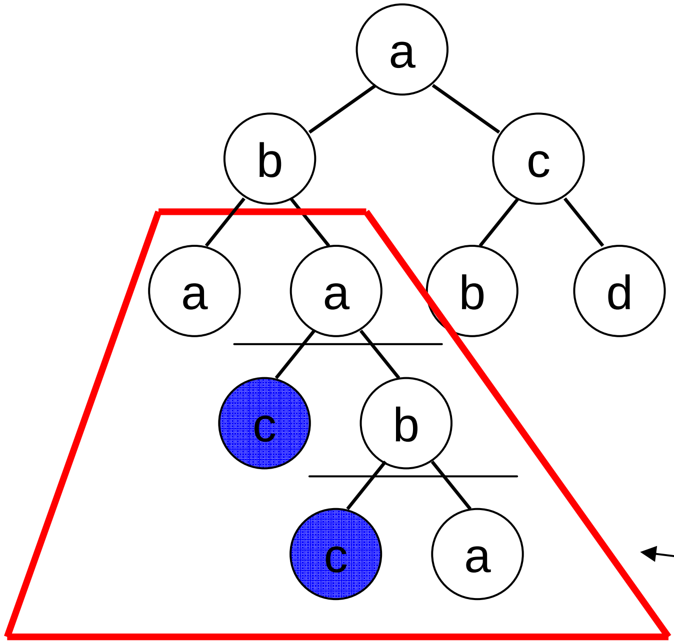
Arbitrary queries with `/,//,*`

multiple `//`'s

`//a/b//c`

query match position: p = 3



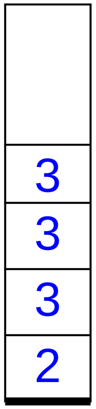




Stay at position 3,
for the *complete subtree*!

Never go back to pos. 1 or pos. 2!

# Arbitrary *Slash+Slashslash*

→ evaluate in *one single pre-order traversal* (using a **stack)**

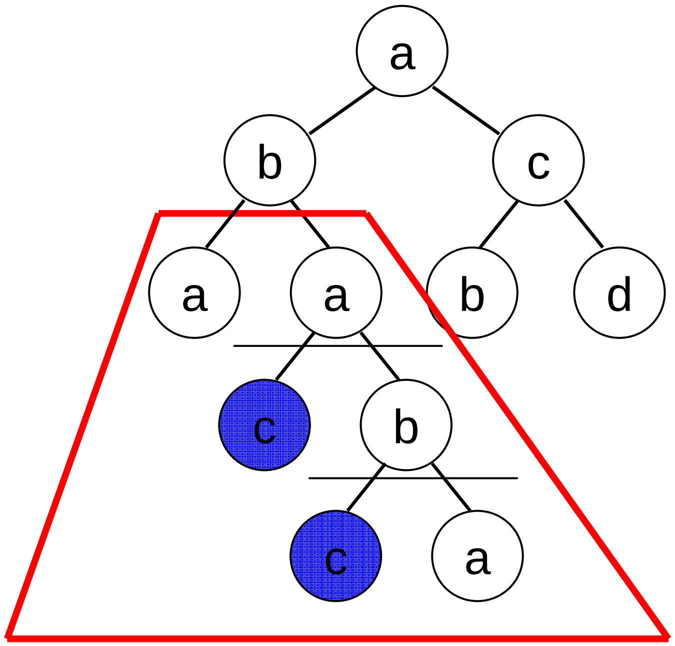



Arbitrary queries with `/,//,*`

multiple `//`'s

`//a/b//c`

query match position: p = 3




Optimizations (for Output Buffers)

(1) If *inside a matched subtree*, record position (or range within buffer), instead of creating a new output buffer.

(2) If *subtree is finished* (we are not inside a match), then we can write its buffer out and can start with empty buffer again.
[ Worst Case:
root node selected. size of doc. Needed. ]

# Arbitrary *Slash+Slashslash*

→ evaluate in *one single pre-order traversal* (using a **stack)**

Arbitrary queries with `/,//,*`

multiple `//`'s

`//a/b//c`

query match position: p = 3

| |
|---|
| |
| 3 |
| 3 |
| 3 |
| 2 |

`//a/b//c/d/*/e//f/g//h`

➔ **Same as before**

jump back within `/`-sequence.
AT MOST to the beginning of the last `//`.

Use KMP within `/`-sequence.

For `*`'s: build several KMP-tables.

# Arbitrary *Slash+Slashslash*

→ evaluate in *one single pre-order traversal* (using a **stack)**

Arbitrary queries with `/,//,*`

multiple `//`'s

`//a/b//c`

3
3
3
2

query match position: p = 3

`//a/b//c/d/*/e//f/g//h`

If Node-IDs are printed, then no output buffers are needed.

Then:
Memory proportional to height.
Should run for arbitrary large docs!

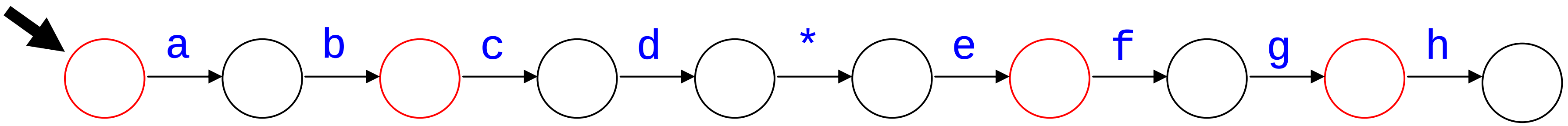
*Query Problem* is solved!

Leave optimizations of

`>cat file.xml [1.2.7,1.3,1.3.1.1, …]`

To OS/UNIX hackers.. ☺

# 1. Automaton Approach




`//a/b//c/d/*/e//f/g//h`

➔ **Same as before**

jump back within `/`-sequence.
AT MOST to the beginning of the last `//`.

Use KMP within `/`-sequence.

For `*`'s: build several KMP-tables.

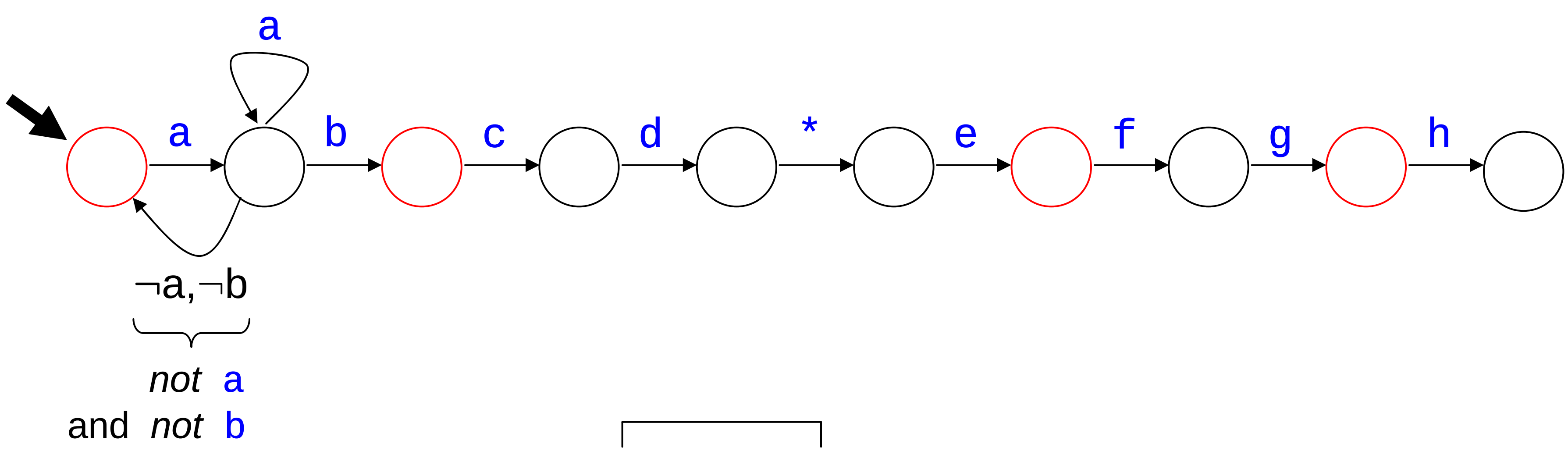
**Recall**

Deterministic Automaton runs in

→ ***linear time*** and
→ ***constant space***

(plus *stack of states*, if we run on paths of a tree)

# 1. Automaton Approach




`//a/b//c/d/*/e//f/g//h`

➔ **Same as before**

jump back within `/`-sequence.
AT MOST to the beginning of the last `//`.

Use KMP within `/`-sequence.

For `*`'s: build several KMP-tables.

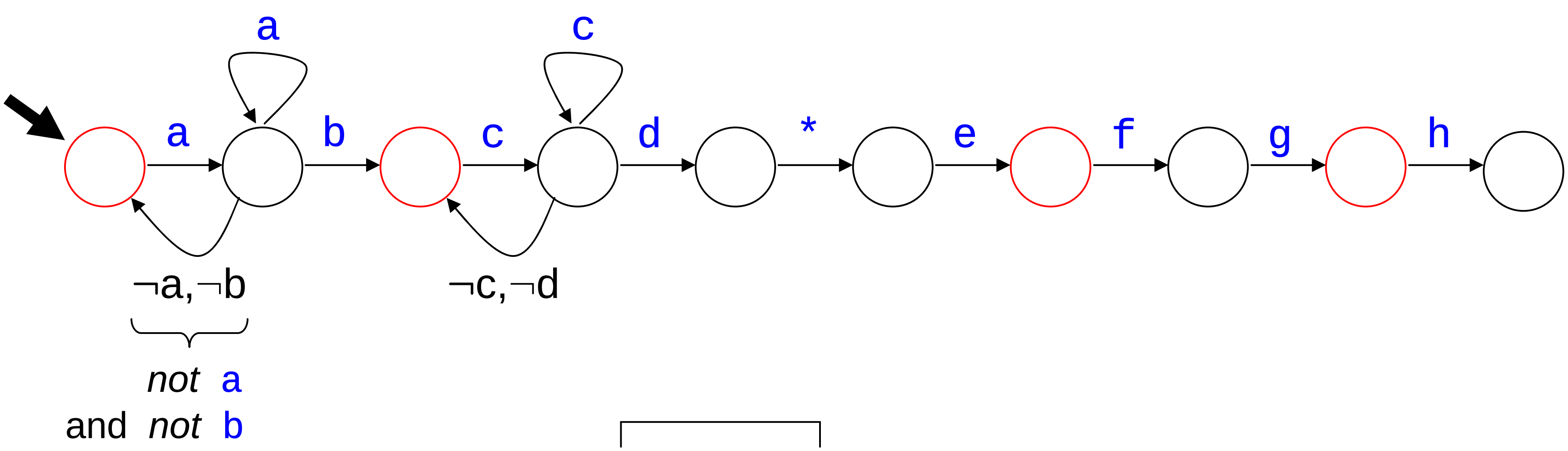
**Recall**

Deterministic Automaton runs in

→ ***linear time*** and
→ ***constant space***

(plus *stack of states*, if we run on paths of a tree)

# 1. Automaton Approach




`//a/b//c/d/*/e//f/g//h`

➔ **Same as before**

jump back within `/`-sequence.
AT MOST to the beginning of the last `//`.

Use KMP within `/`-sequence.

For `*`'s: build several KMP-tables.

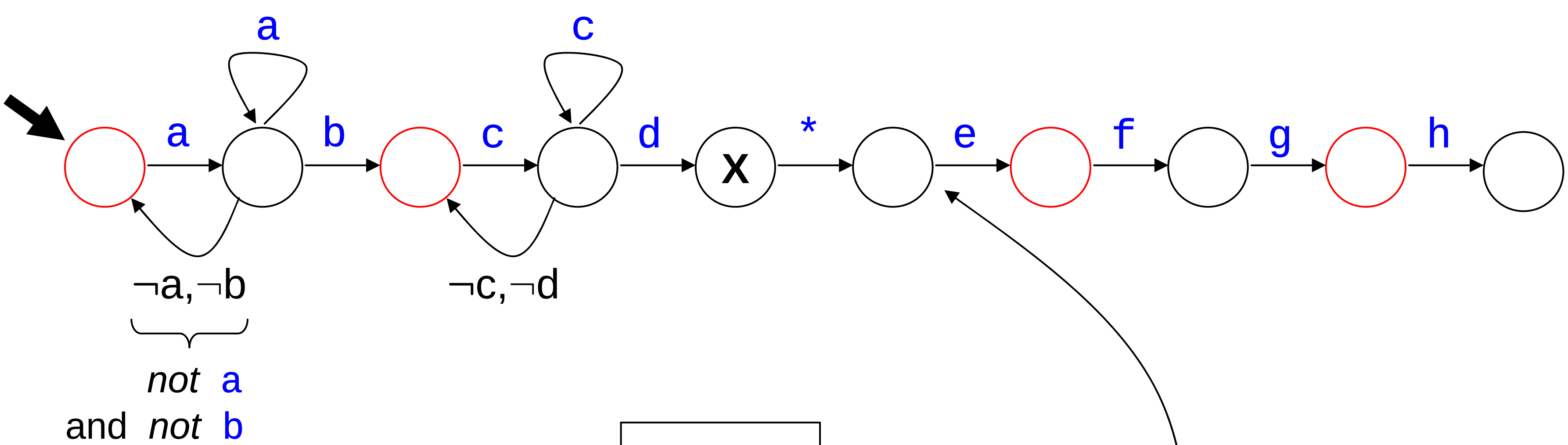
**Recall**

Deterministic Automaton runs in

→ ***linear time*** and
→ ***constant space***

(plus *stack of states*, if we run on paths of a tree)

# 1. Automaton Approach



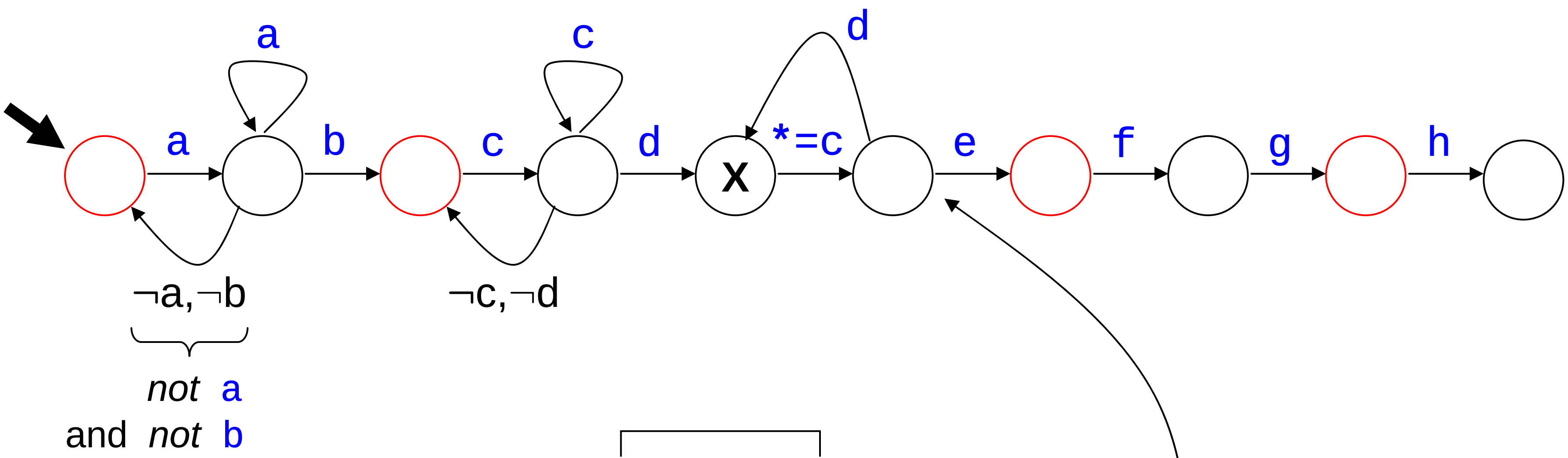

//a/b//c/d/*/e//f/g//h

Problem
If it is NOT an e here, then what to do??

E.g.,
a b c d c d

We should be in state **X**!

# 1. Automaton Approach




`//a/b//c/d/*/e//f/g//h`

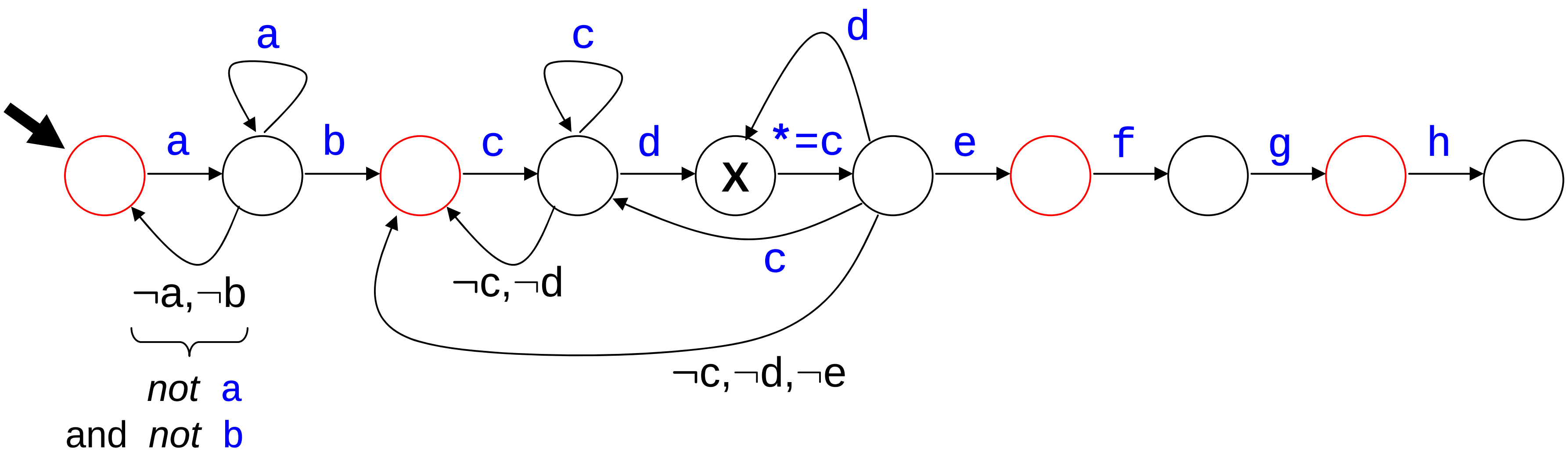
Problem
If it is NOT an e here, then what to do??

➔ Need to know what the * was!!

E.g.,
a b c d c d

We should be in state **X**!

# 1. Automaton Approach



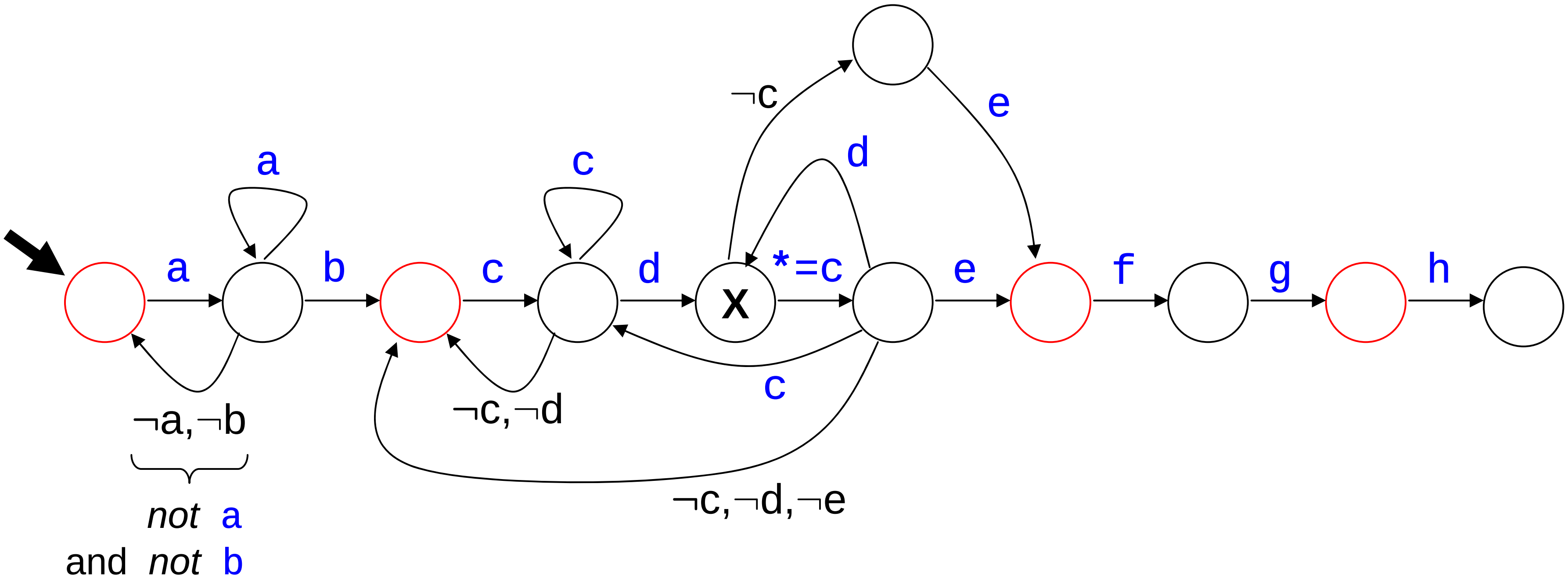

`//a/b//c/d/*/e//f/g//h`

***=?** Which other letters need to be considered?

c d x y

≠c

≠e




`//a/b//c/d/*/e//f/g//h`

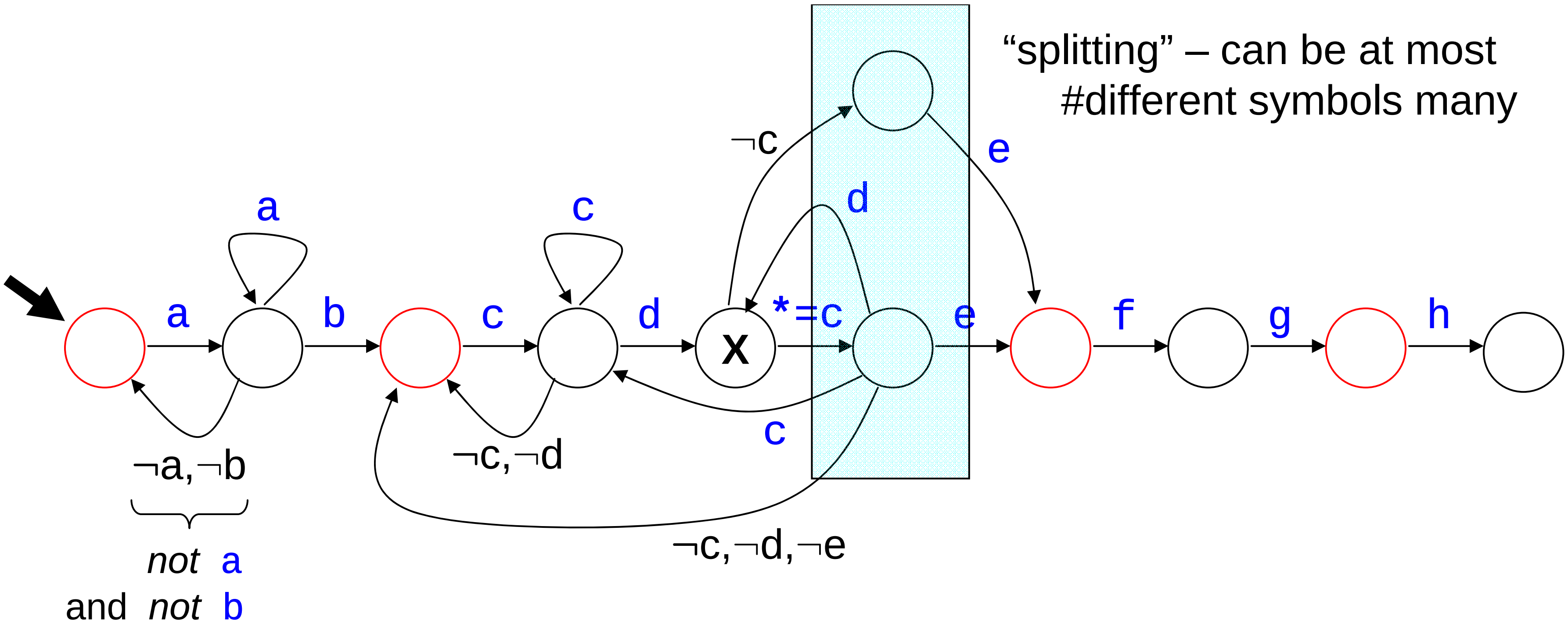
***=?** Which other letters need to be considered?

c d x y

≠c ≠e

→ for x≠c, not important what x is

→ only x=c / x≠ c matters




//a/b//c/d/*/e//f/g//h

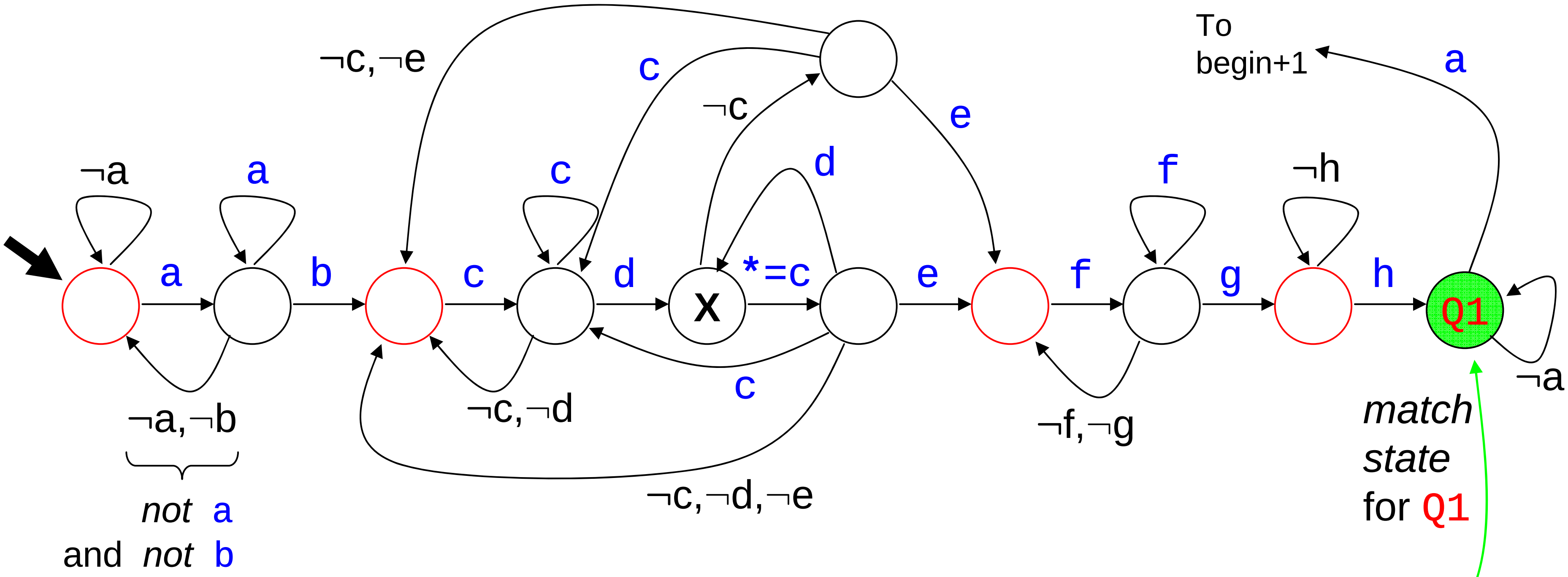
***=?** Which other letters need to be considered?

c d x y

≠c ≠e

→ for x≠c, not important what x is

→ only x=c / x≠ c matters




**Q1 = //a/b//c/d/*/e//f/g//h**

***=?** Which other letters need to be considered?

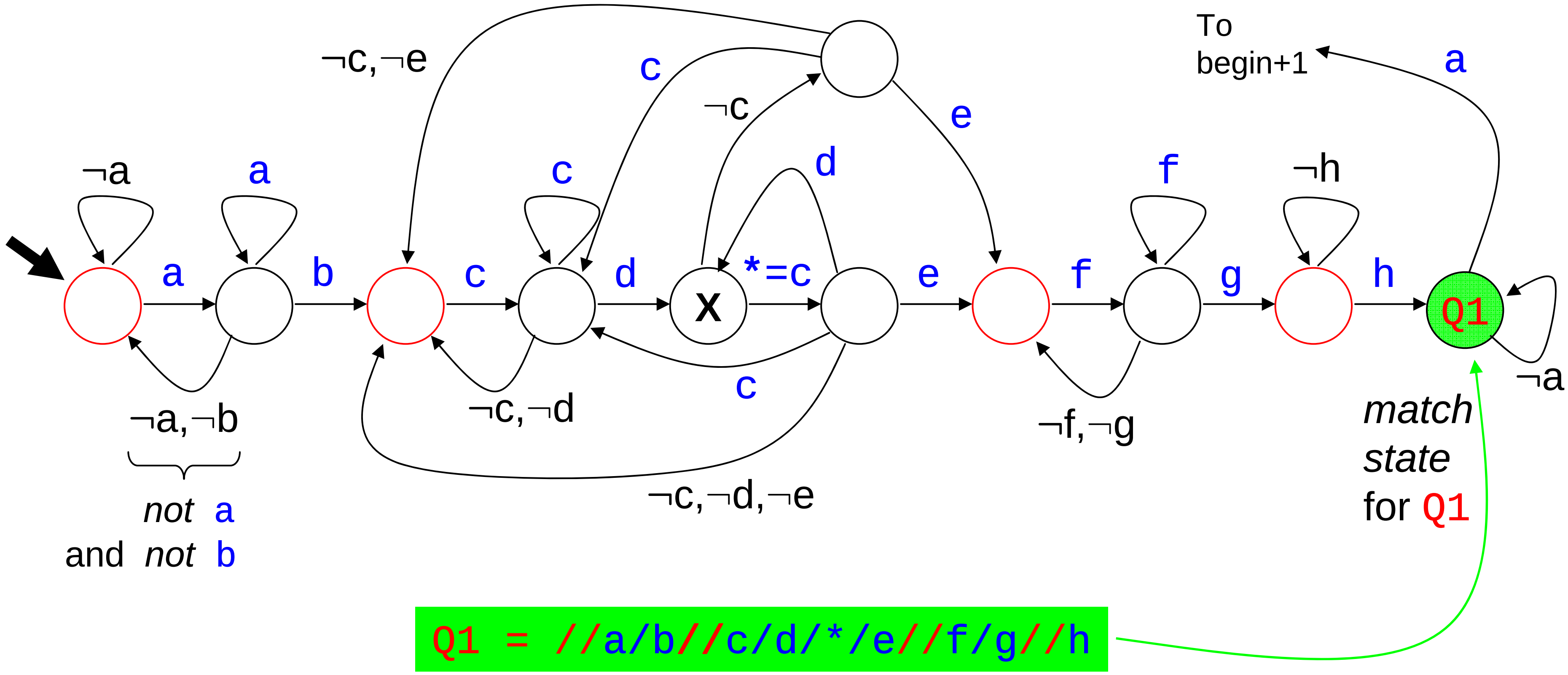
c d x y

x: ≠c

y: ≠e

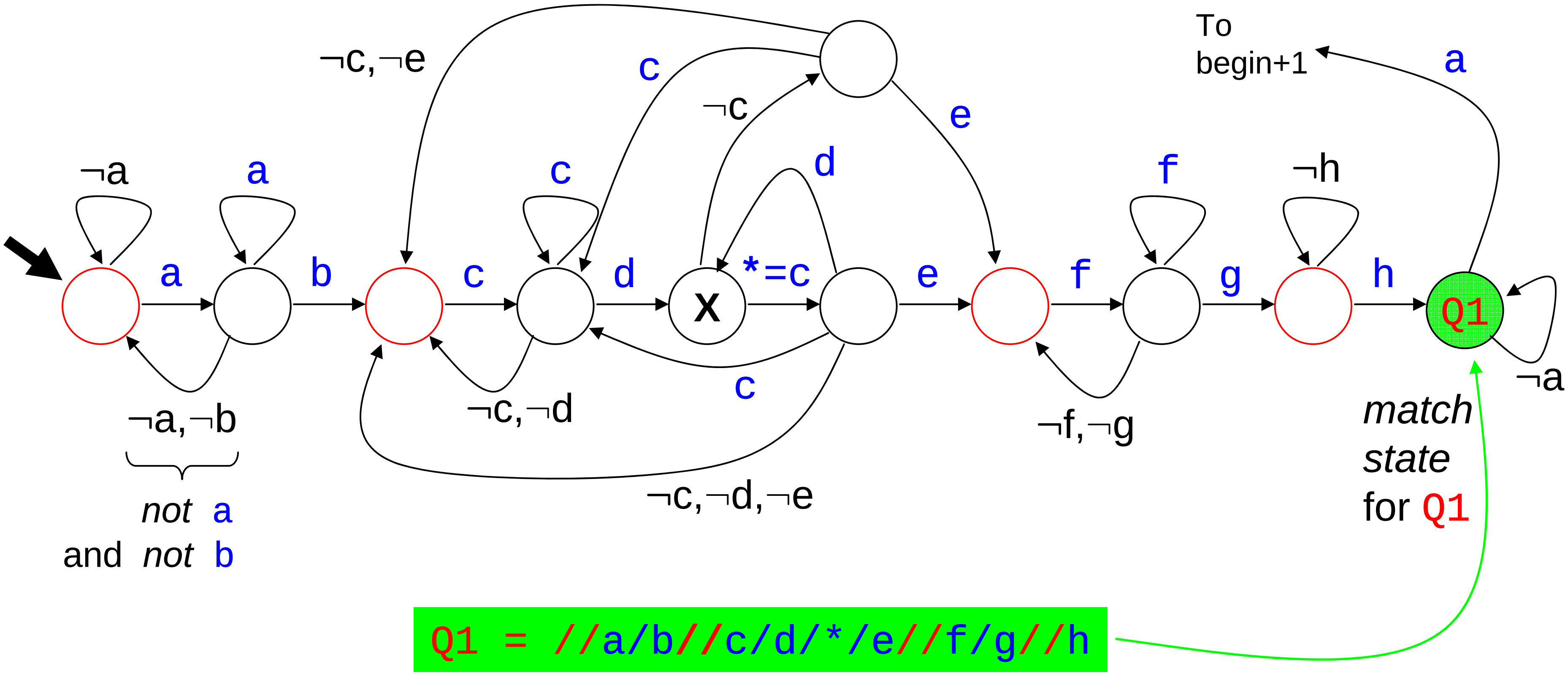
→ for x≠c, not important what x is

→ only x=c / x≠ c matters




Advantage of automata:

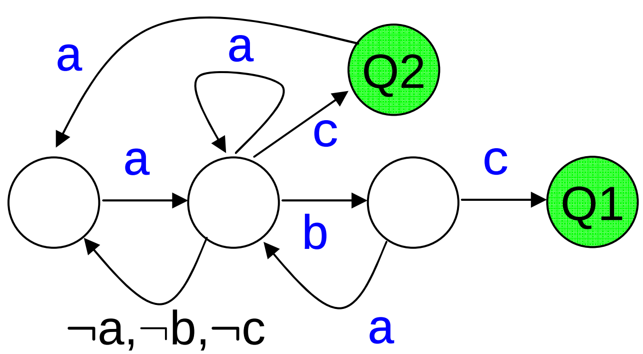
→ can be *combined* to evaluate MANY queries "in parallel".

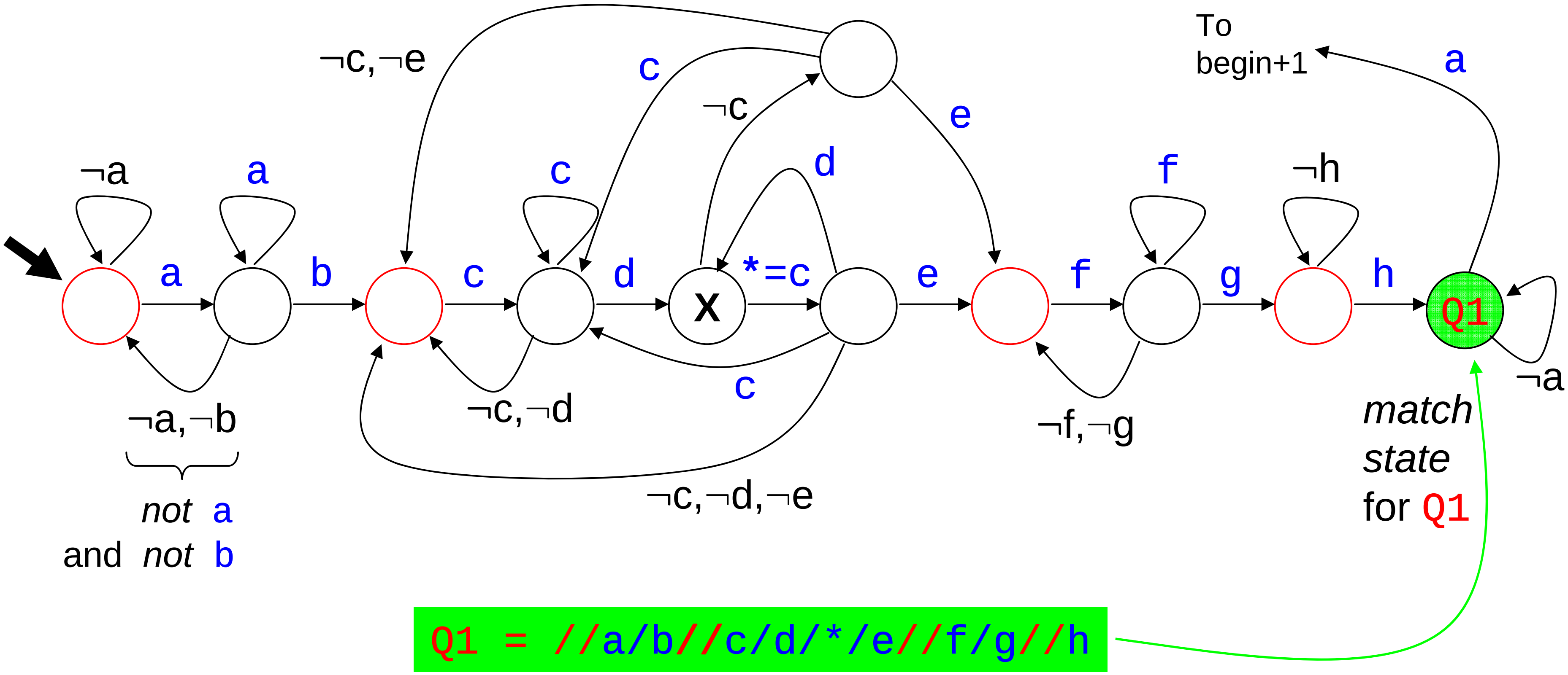



Advantage of automata:

→ can be *combined* to evaluate MANY queries "in parallel".

Q1=//a/b/c
Q2=//a/c

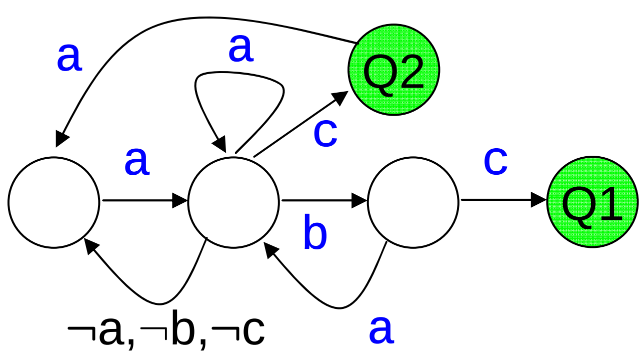





Advantage of automata:

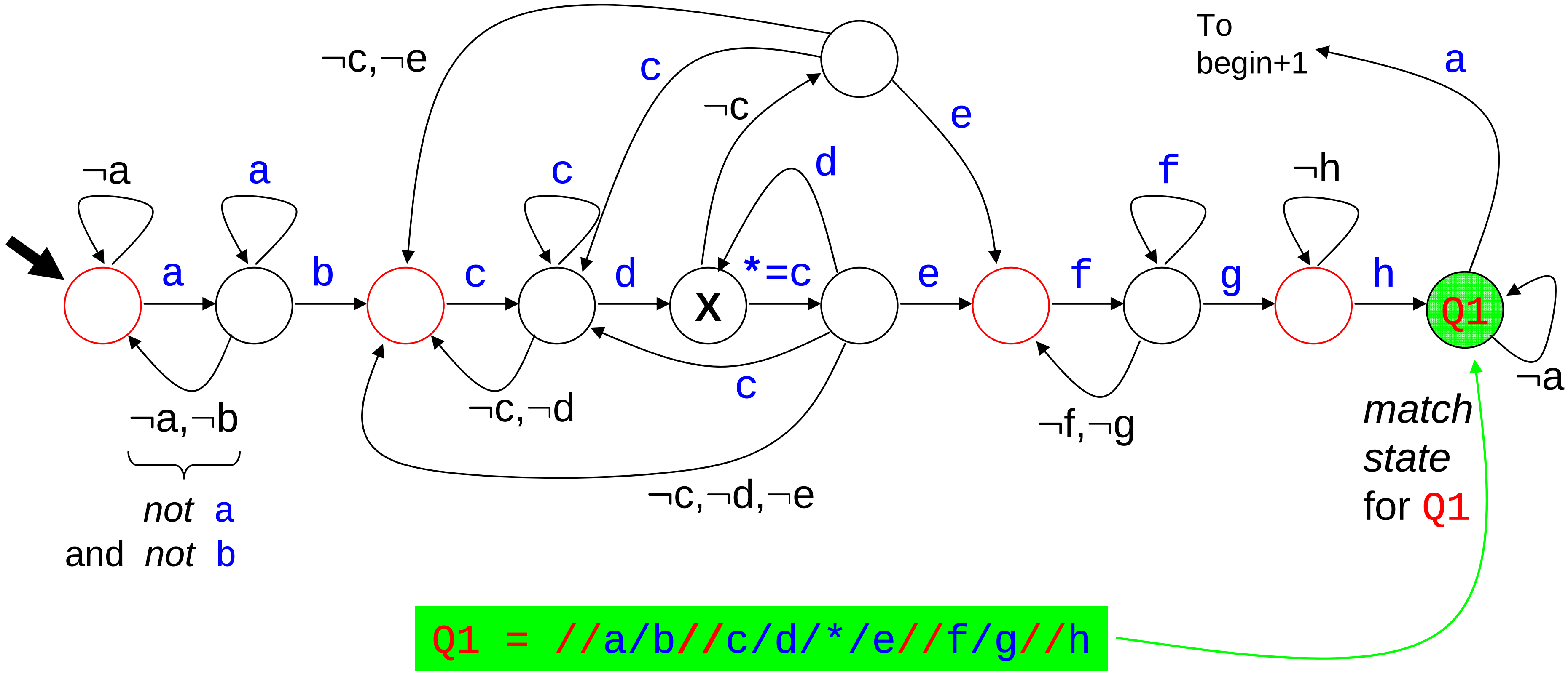
→ can be *combined* to evaluate MANY queries "in parallel".

Q1=//a/b/c
Q2=//a/c




## Questions

1. Which transition is WRONG?

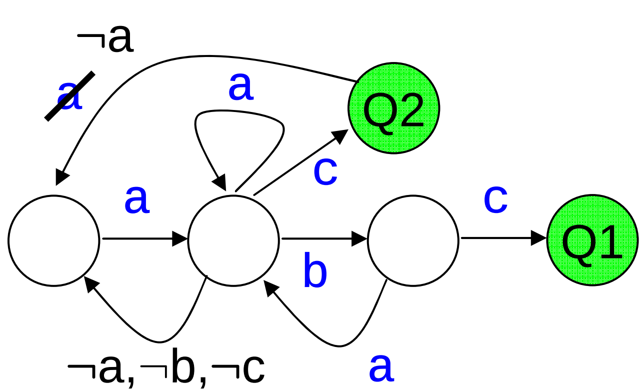



Advantage of automata:

→ can be *combined* to evaluate MANY queries "in parallel".

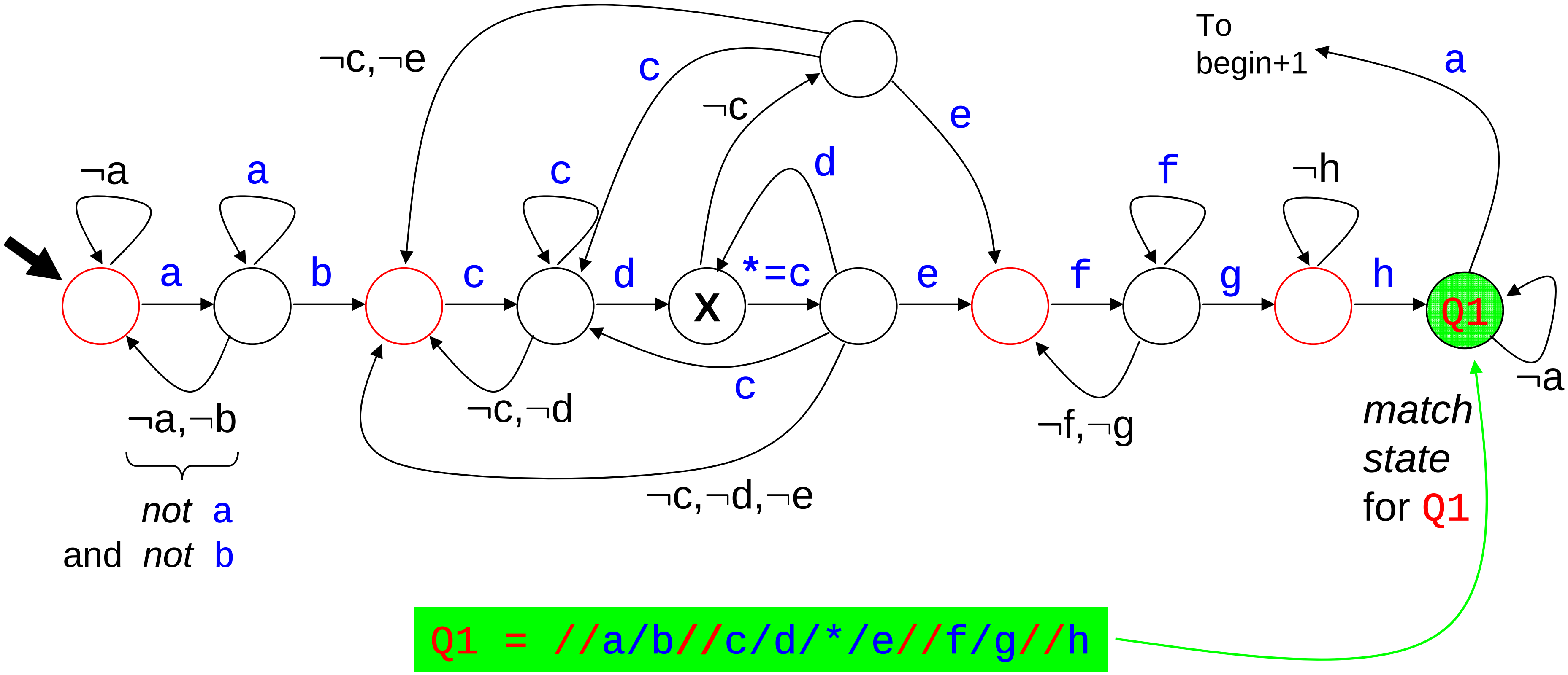
Q1=//a/b/c

Q2=//a/c




**Questions**

1. Which transition is WRONG?
2. How many transitions are missing?

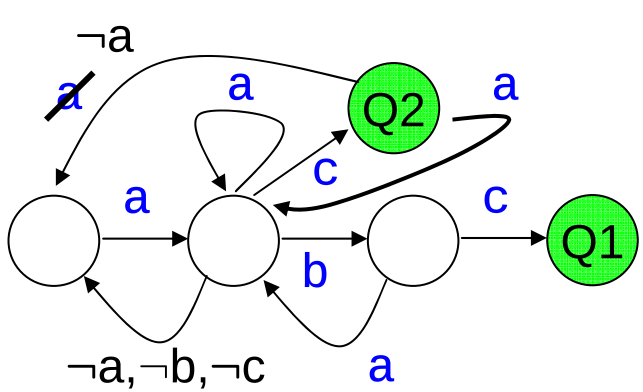



Advantage of automata:

→ can be *combined* to evaluate MANY queries "in parallel".

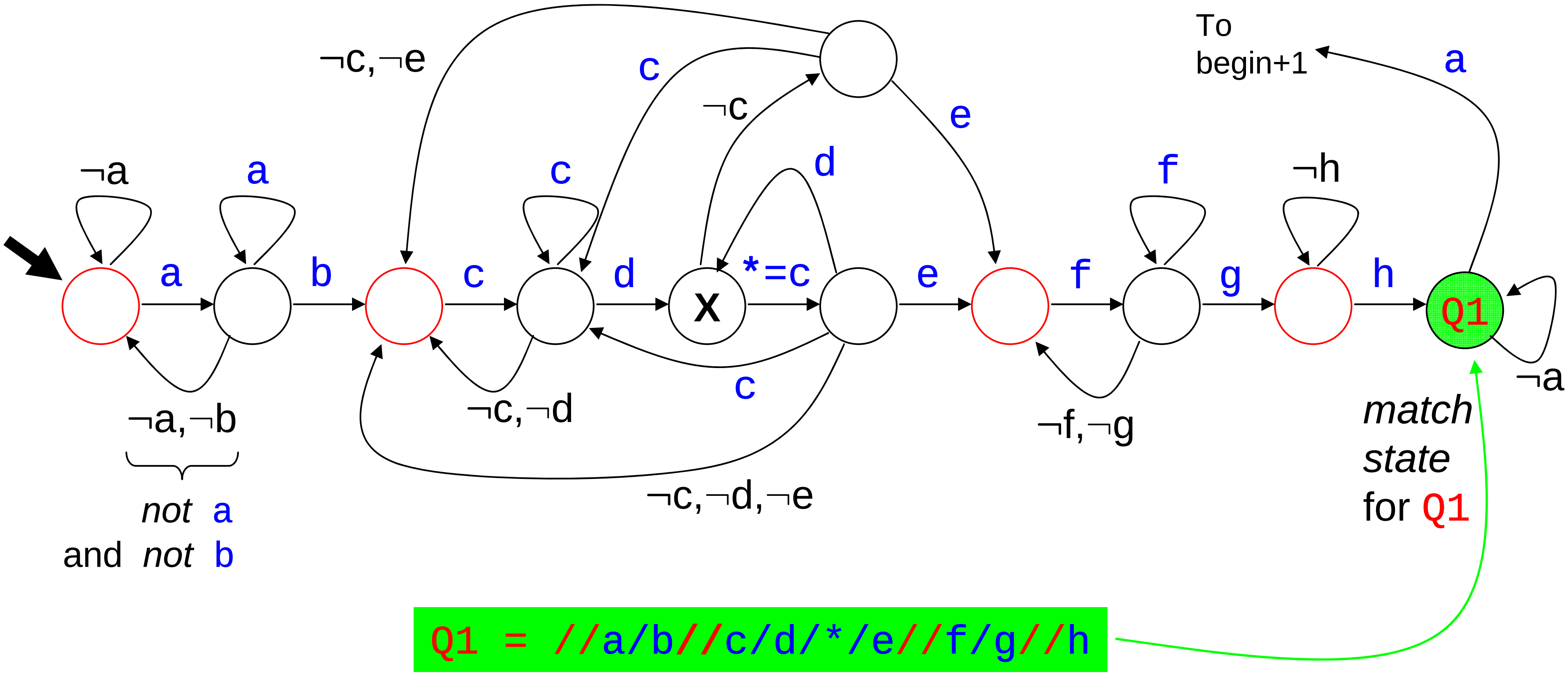
Q1=//a/b/c

Q2=//a/c




## Questions

1. Which transition is WRONG?
2. How many transitions are missing?

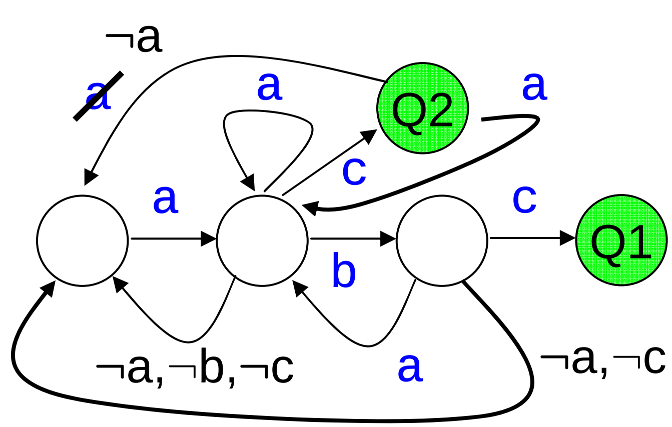



Advantage of automata:

→ can be *combined* to evaluate MANY queries "in parallel".

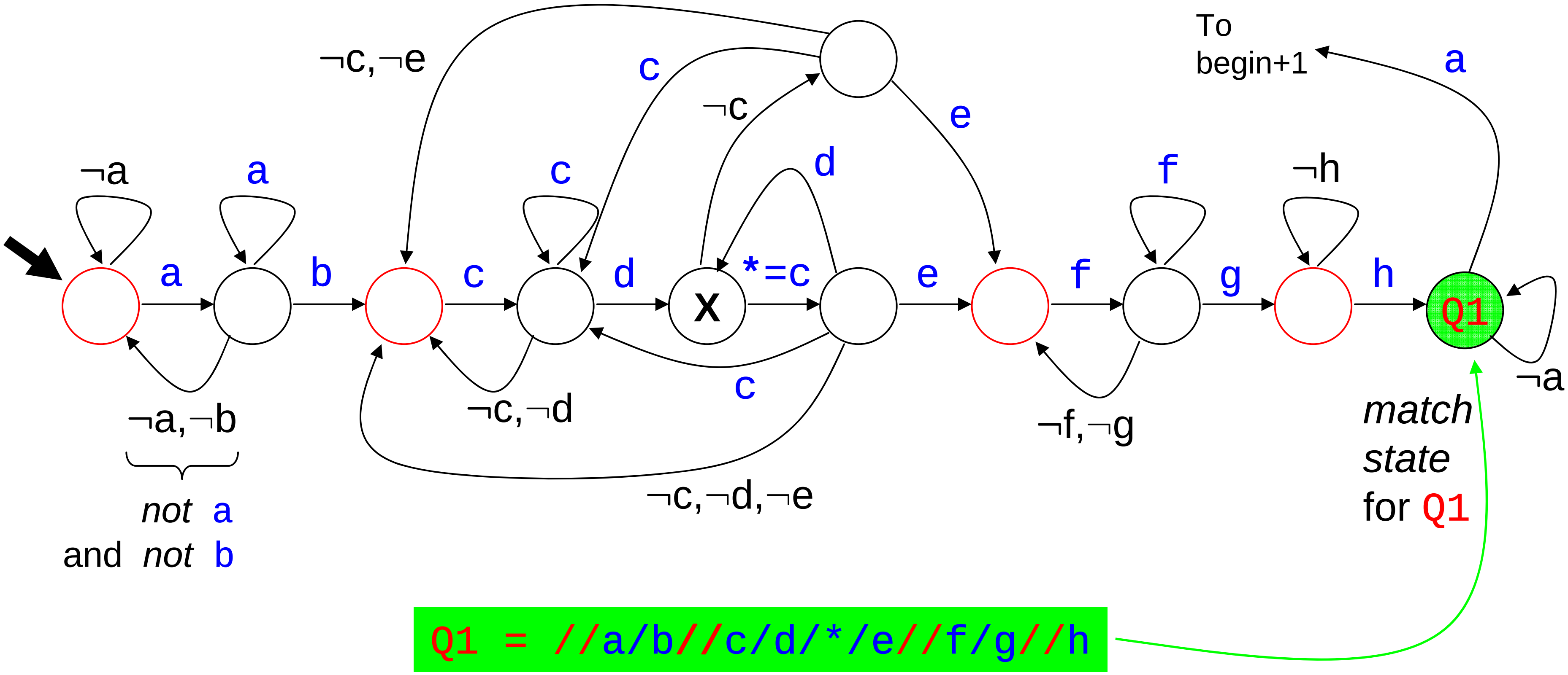
Q1=//a/b/c

Q2=//a/c




## Questions

1. Which transition is WRONG?
2. How many transitions are missing?

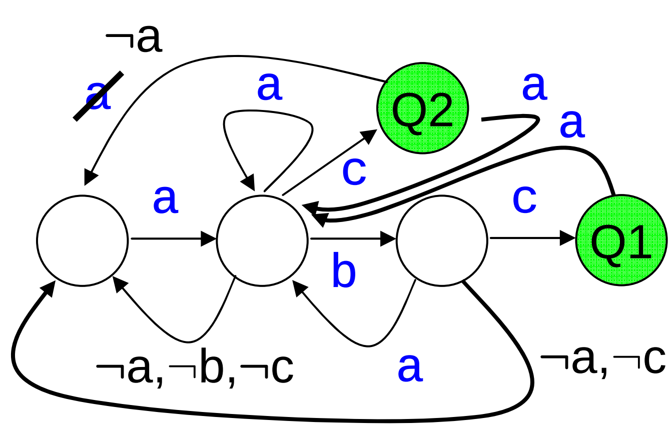



Advantage of automata:

→ can be *combined* to evaluate MANY queries "in parallel".

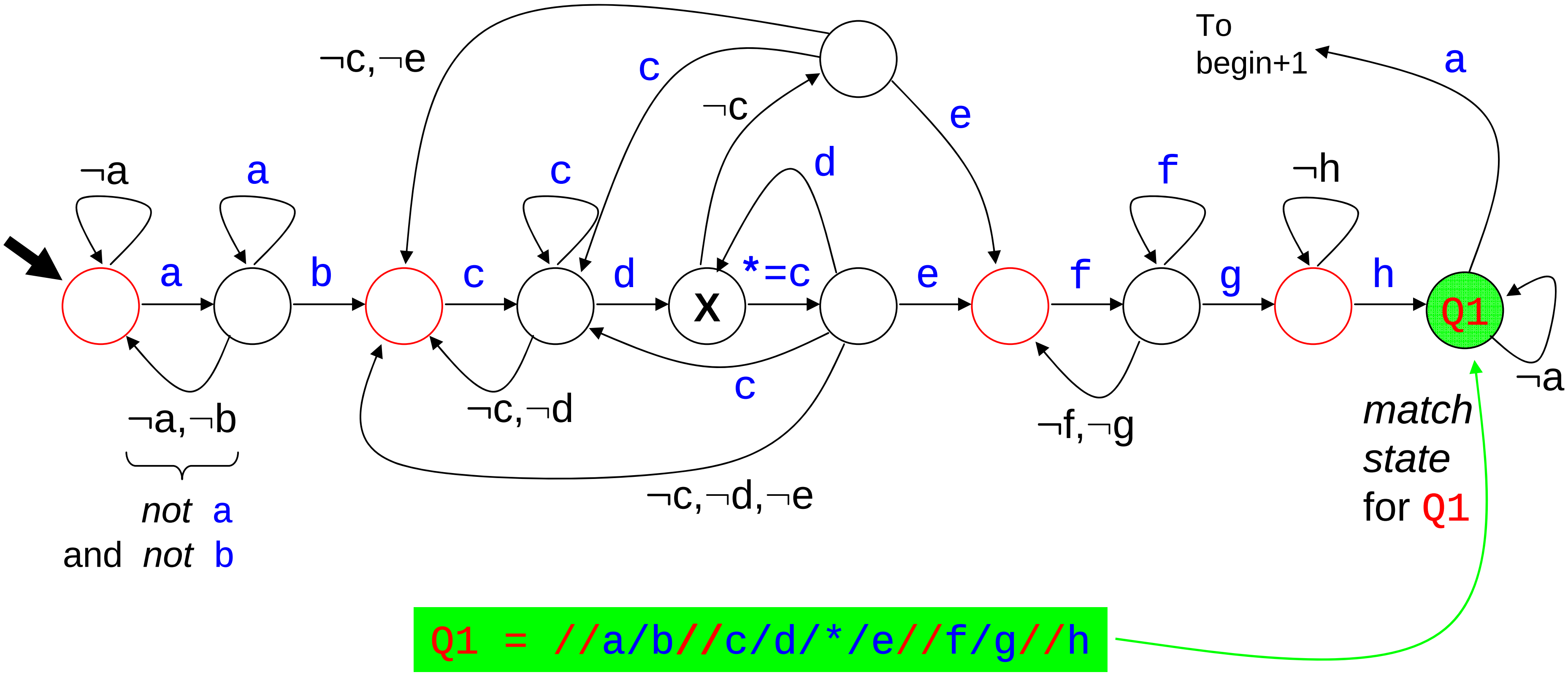
Q1=//a/b/c

Q2=//a/c




**Questions**

1. Which transition is WRONG?
2. How many transitions are missing?

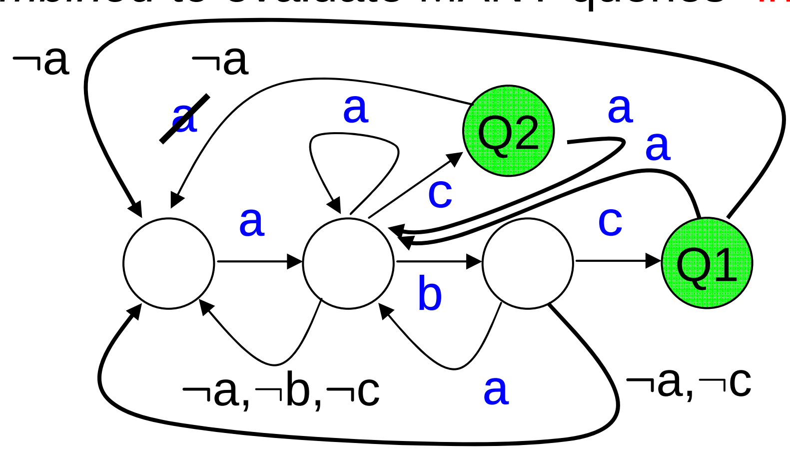



Advantage of automata:

→ can be *combined* to evaluate MANY queries "in parallel".

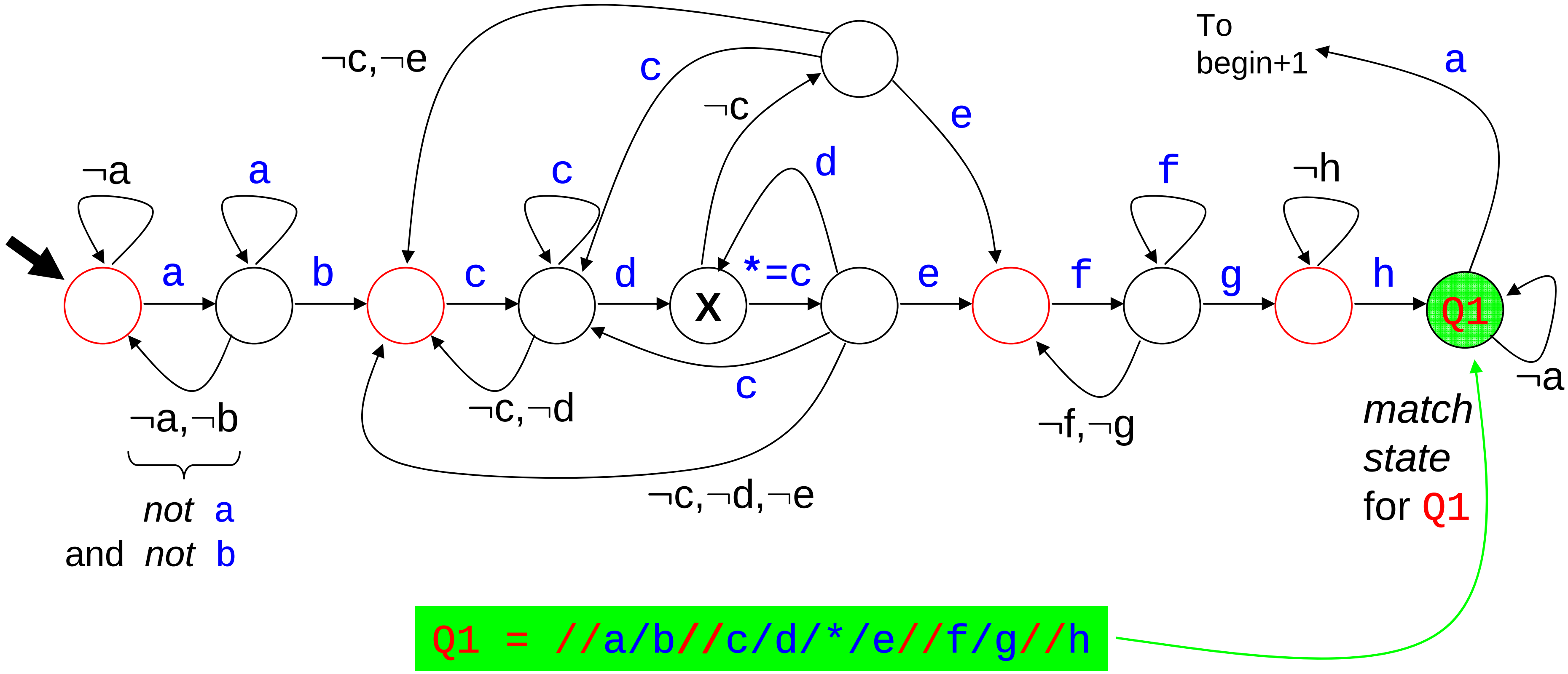
Q1=//a/b/c

Q2=//a/c



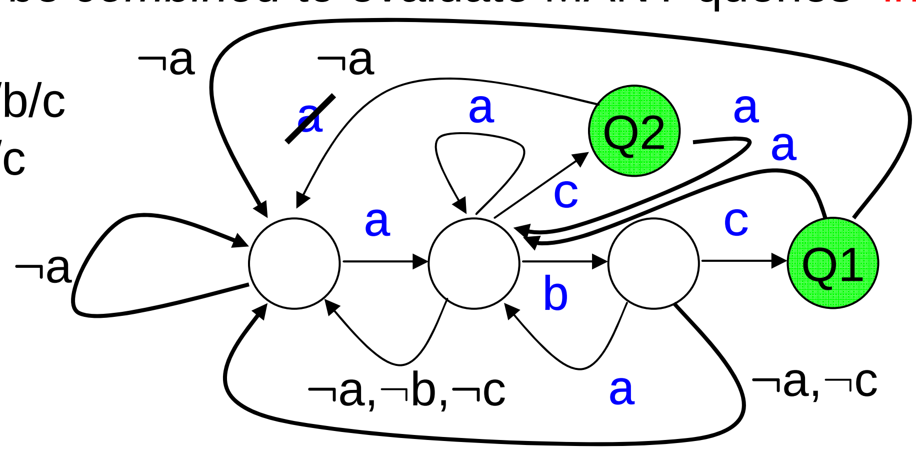

**Questions**

1. Which transition is WRONG?
2. How many transitions are missing?



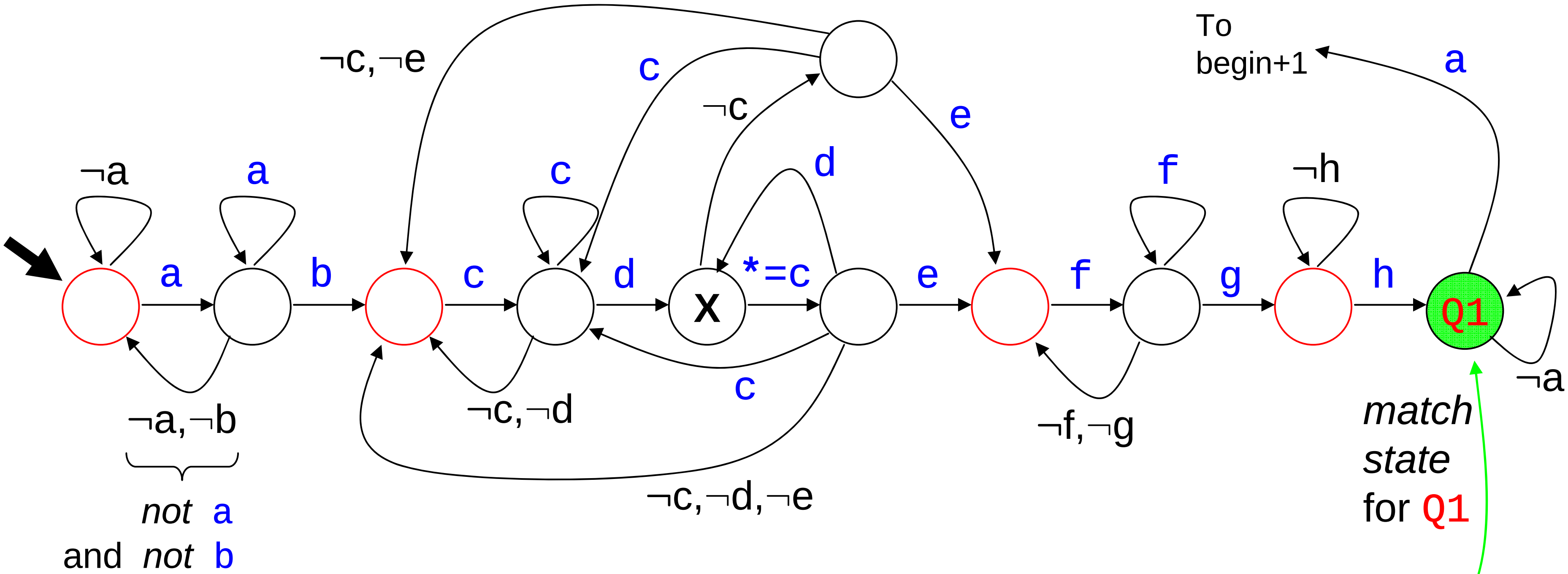

Advantage of automata:
→ can be *combined* to evaluate MANY queries "in parallel".

Q1=//a/b/c
Q2=//a/c




**Questions**

1. Which transition is WRONG?
2. How many transitions are missing?



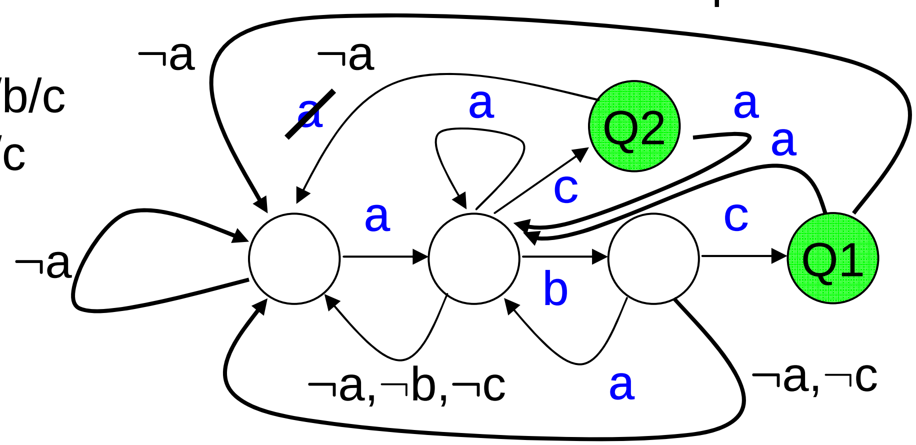

Q1 = //a/b//c/d/*/e//f/g//h

Advantage of automata:

→ can be *combined* to evaluate MANY queries "in parallel".

Q1=//a/b/c

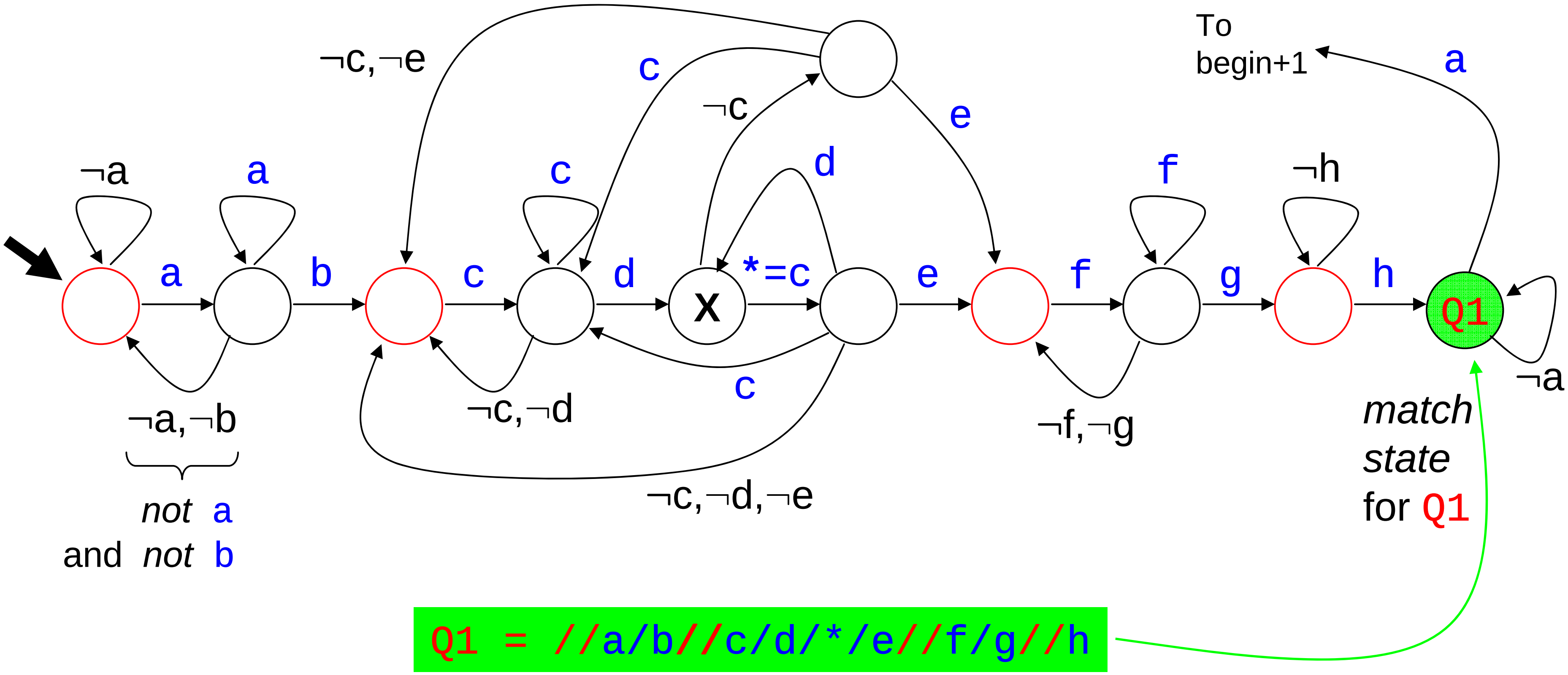
Q2=//a/c




**Questions**

1. Which transition is WRONG?
2. How many transitions are missing?

→ **5**

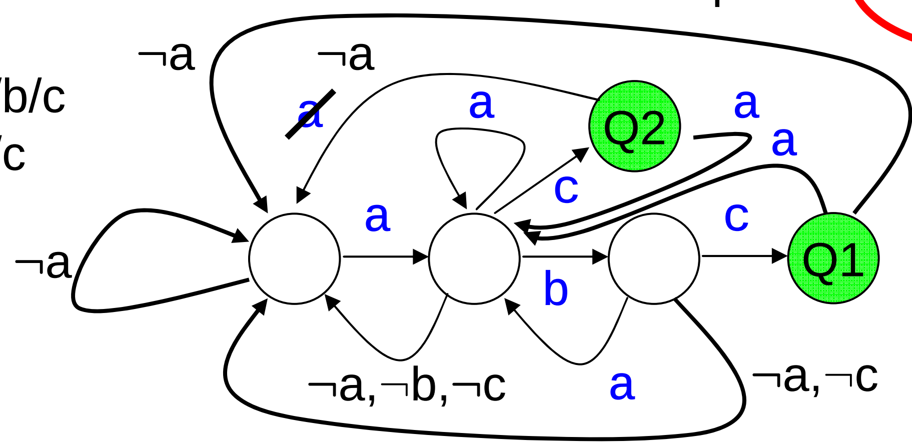



Advantage of automata:

→ can be *combined* to evaluate MANY queries "in parallel".

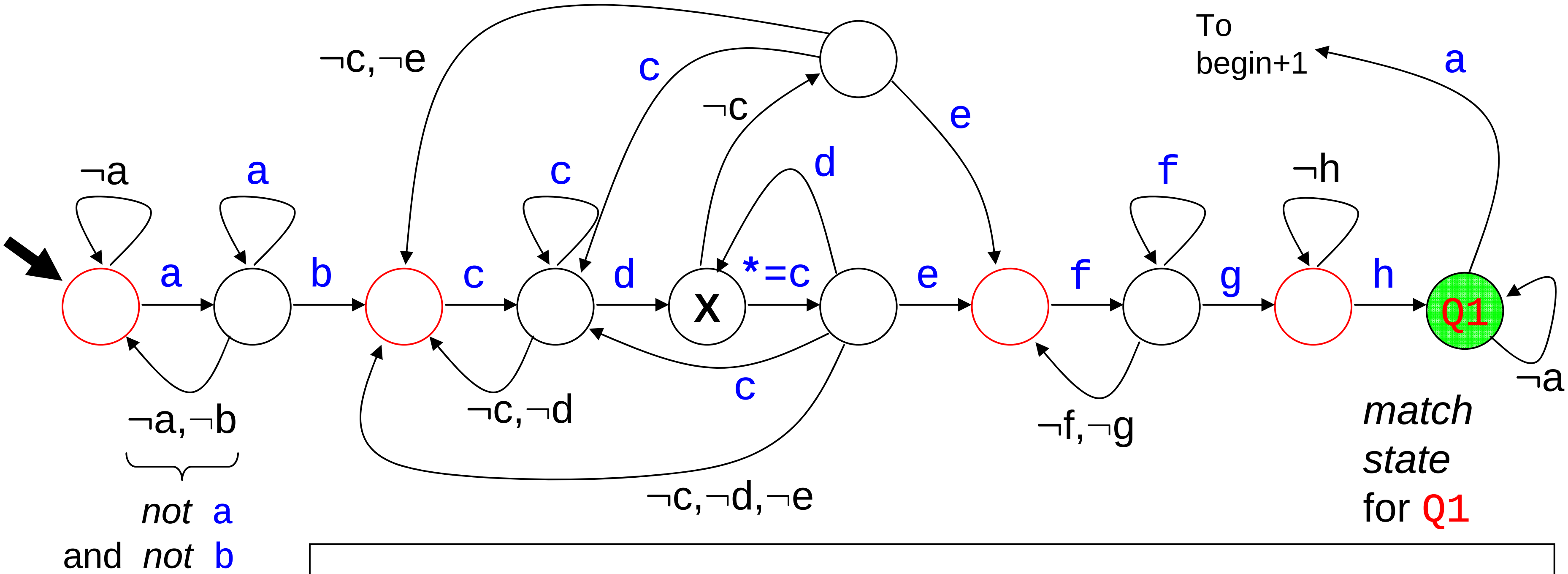
Q1=//a/b/c

Q2=//a/c




ONE look-up per node!

Combined automaton:
SIZE ≤ SIZE(A1) x SIZE(A2)




**Question**

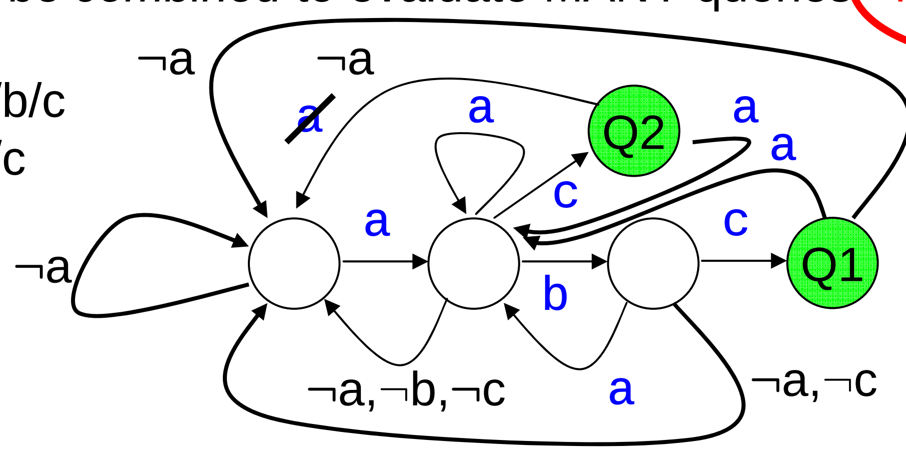
What is SIZE(A1) wrt size of Q1?

Take
(1) SIZE(A) = #states
(2) SIZE(A) = #transitions

Advantage of automata:
→ can be *combined* to evaluate MANY queries "in parallel".

ONE look-up per node!

Q1=//a/b/c
Q2=//a/c



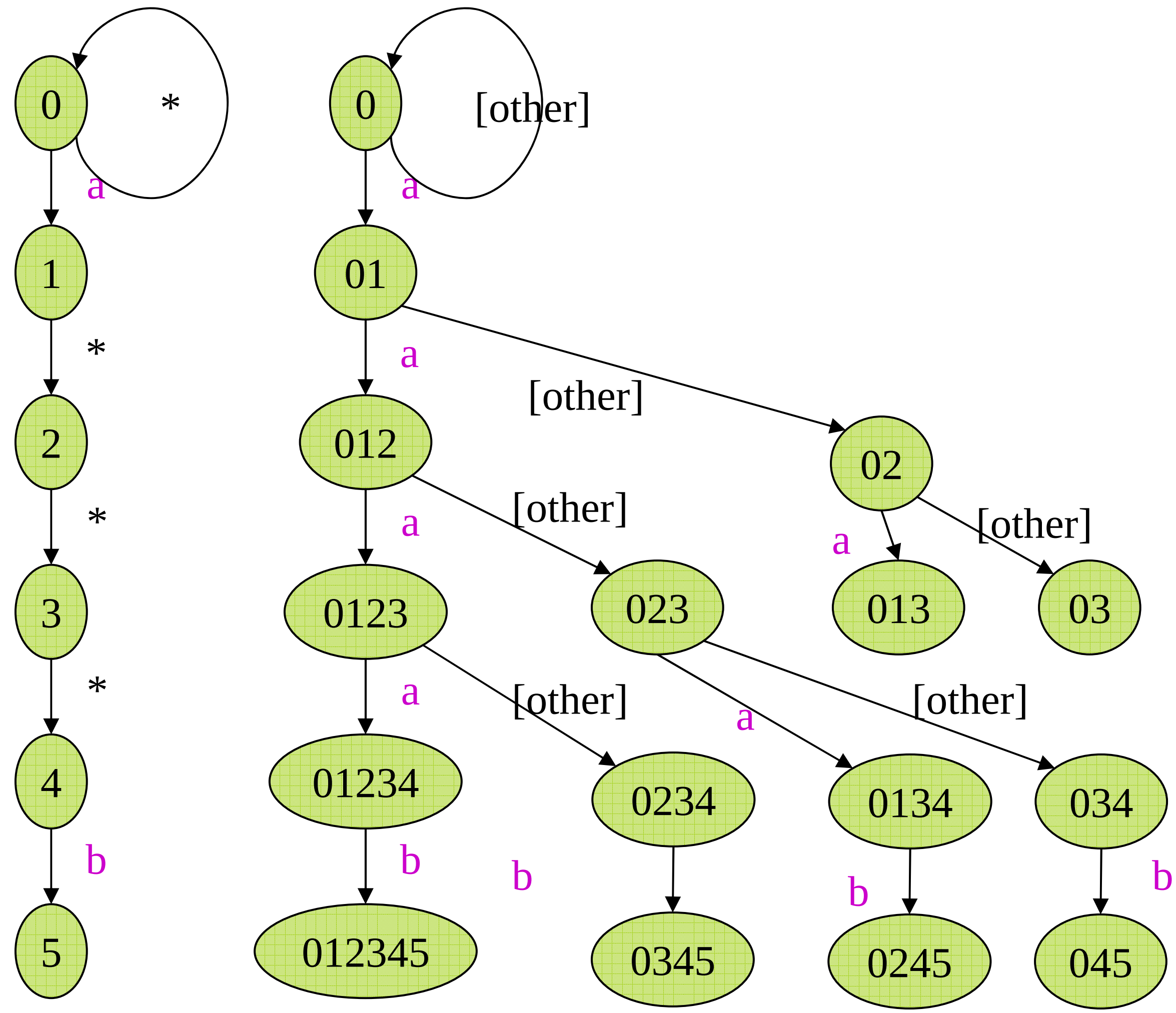

Combined automaton:
SIZE ≤ SIZE(A1) x SIZE(A2)

# 3. The Size of the DFA

//a/*/*/*/b

Size of DFA = exponential in *'s

(not a real concern)




**NFA**

**DFA** (fragment, and without back edges)

# 3. The Size of the DFA

**Theorem** [GMOS'02] The number of states in the DFA for one linear XPath expression P is at most:

$$\boxed{k+|P|\ k\ s^m}$$

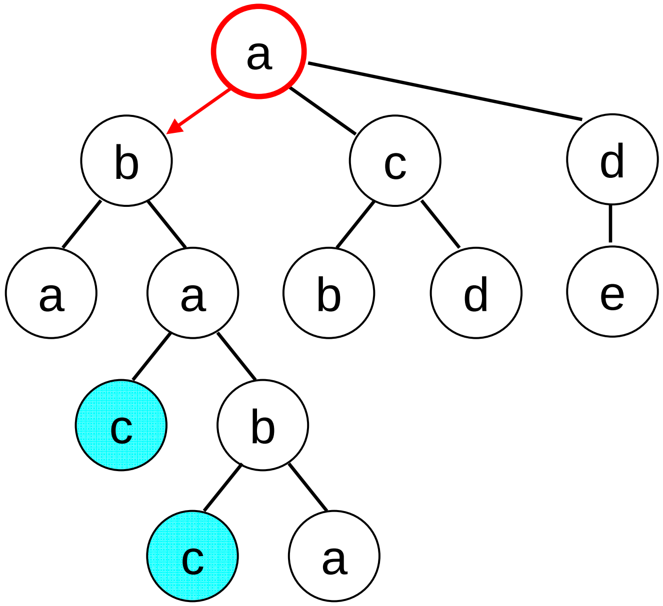
k = number of //

s = size of the alphabet (number of tags)

m = max number of * between two consecutive //

How to deal with filters?

`//a[./d/e]/b//c`

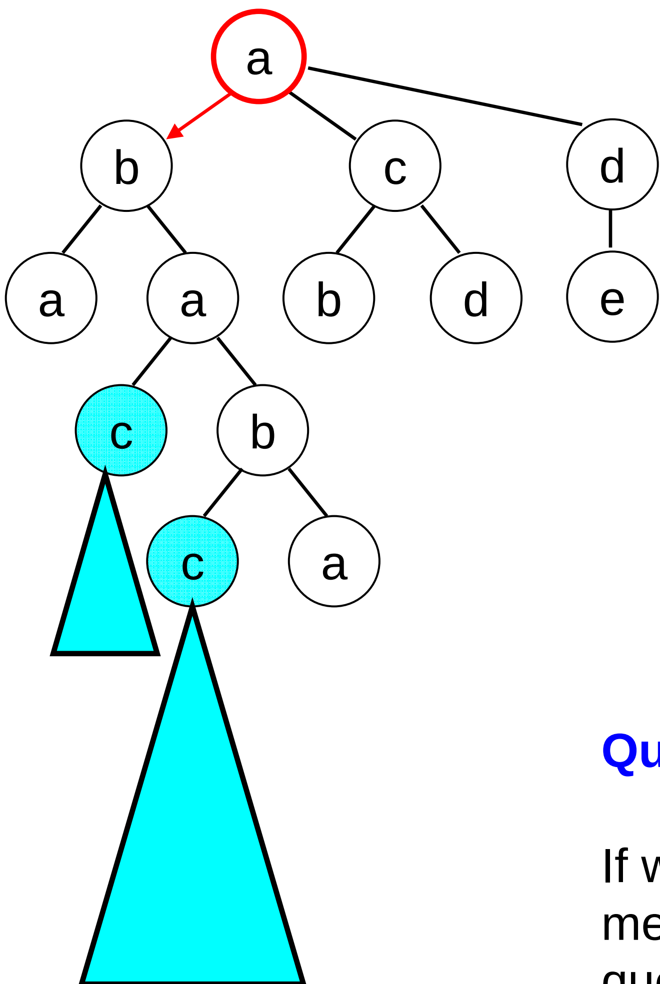



When we meet the c-nodes
(in pre order traversal)
we do not know yet if the
**filter** will evaluate to true.

How to deal with filters?

`//a[./d/e]/b//c`

When we meet the c-nodes (in pre order traversal) we do not know yet if the **filter** will evaluate to true.




Must be stored in memory

➔ We have to use *buffers,* as before.

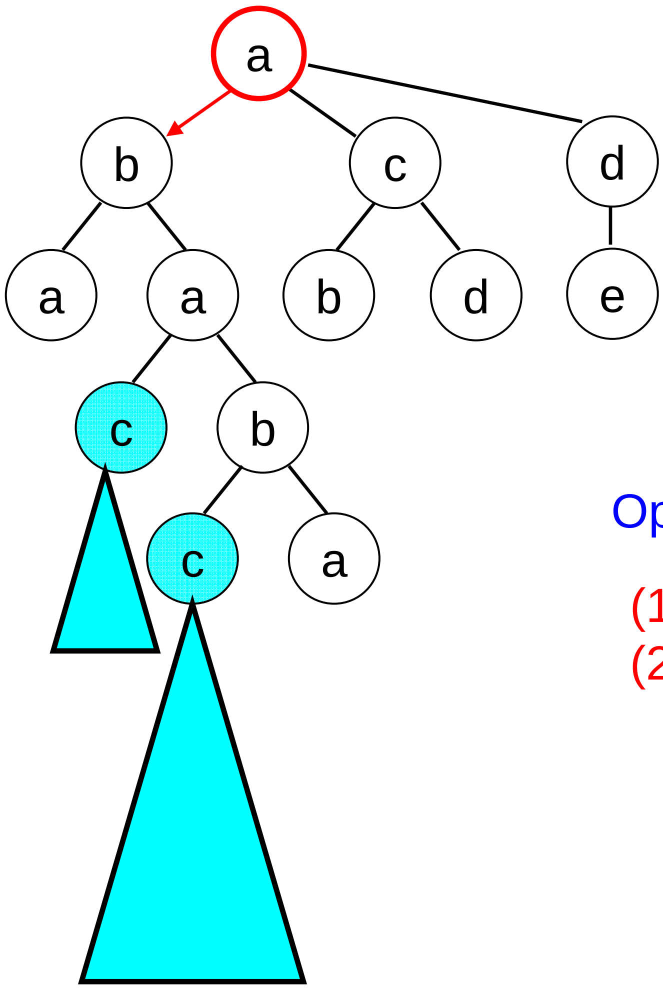
However, now buffers may be **deleted** without being used.

**Question**

If we output node ID's, then how much memory is needed in the worst case for queries with filters?

How to deal with filters?

`//a[./d/e]/b//c`




Must be stored in memory

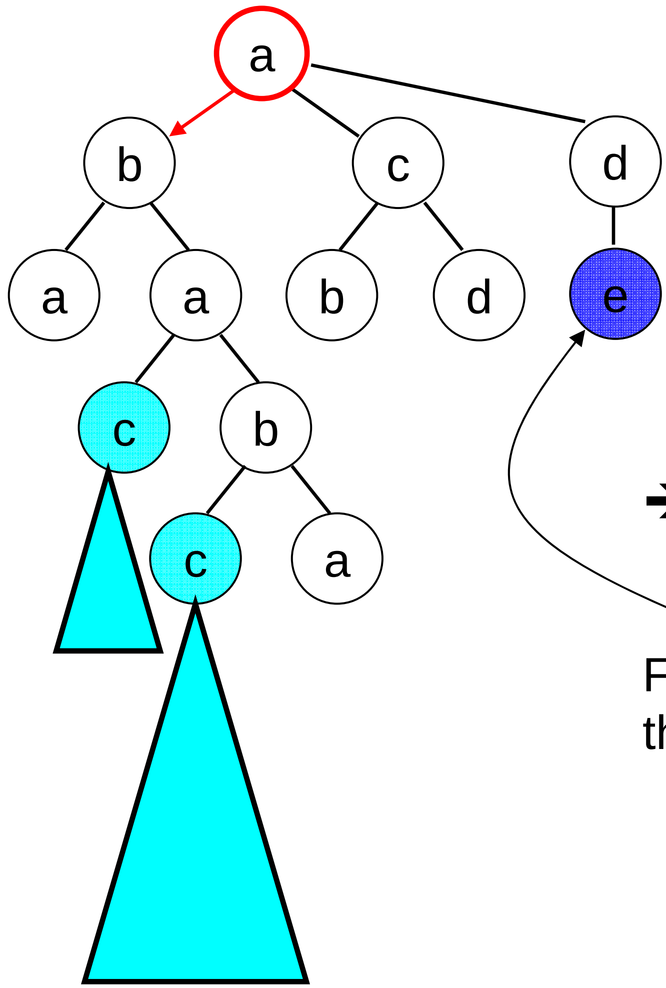
➔ Size of largest documents that can be streamed in this way depends on
- #filters,
- sizes of (pre) selected trees,
- quality of (1), (2), etc..

Optimizations

(1) Store potential match trees as DAGs
(2) Release potential match trees as early as possible!

How to deal with filters?

`//a[./d/e]/b//c`




Must be stored in memory

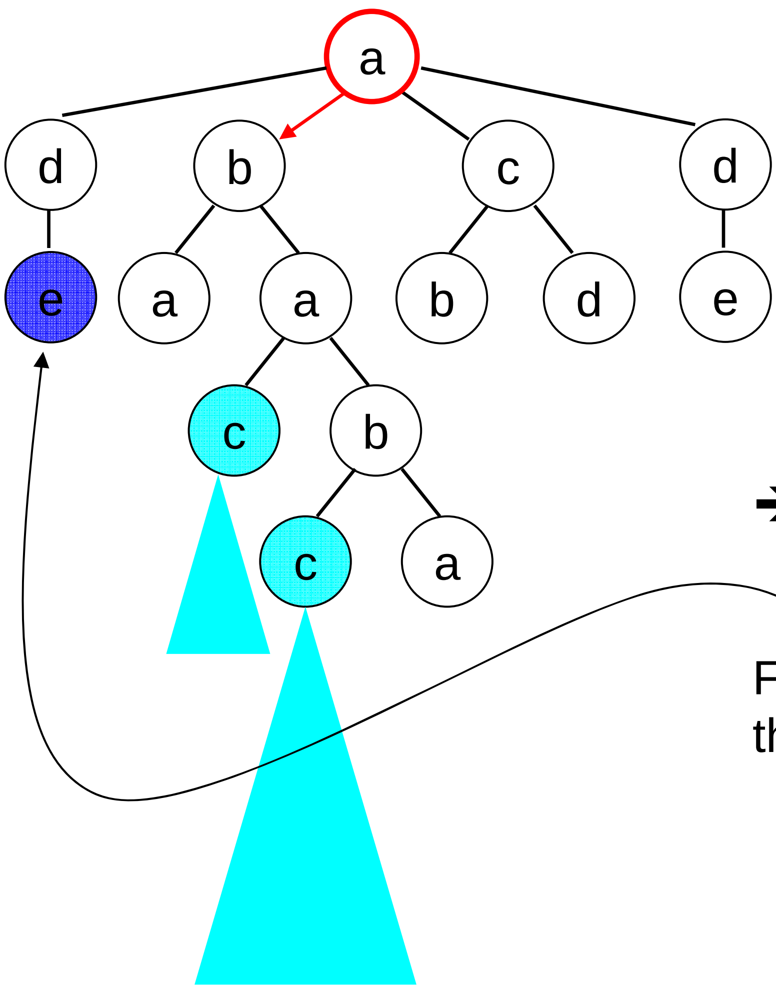
➔ Size of largest documents that can be streamed in this way depends on
- #filters,
- sizes of (pre) selected trees,
- quality of (1), (2), etc..

➔ Release potential match trees *as early as possible*!

Find earliest point at which we know the filter is true.

How to deal with filters?

`//a[./d/e]/b//c`




➔ Size of largest documents that can be streamed in this way depends on
- #filters,
- sizes of (pre) selected trees,
- quality of (1), (2), etc..

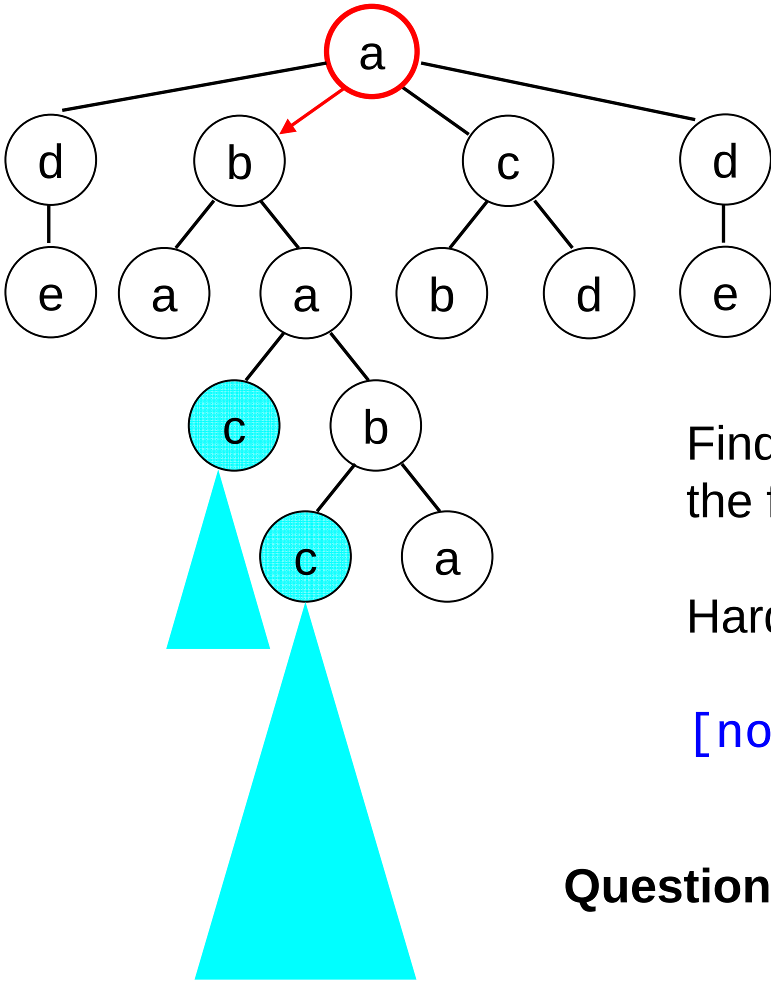
➔ Release potential match trees *as early as possible*!

Find earliest point at which we know the filter is true.

No need to store. Stream! ☺

How to deal with filters?

`//a[./d/e]/b//c`




➔ Size of largest documents that can be streamed in this way depends on

- #filters,
- sizes of (pre) selected trees,
- quality of (1), (2), etc..

Find earliest point at which we know the filter is true.

Harder for *Boolean combinations*:

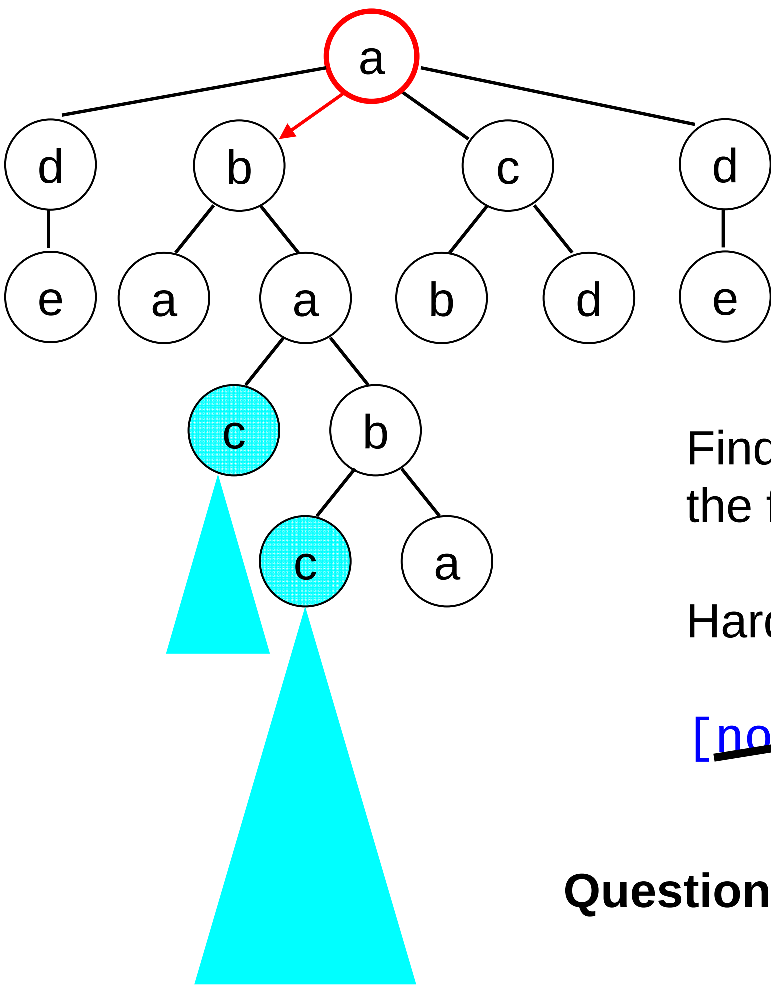
`[not(./d/e) and (./c/d or //b/c)]`

**Question** where is the earliest point for this filter?

No need to store. Stream! ☺

How to deal with filters?

//a[./d/e]/b//c




No need to store. Stream! ☺

➔ Size of largest documents that can be streamed in this way depends on
- #filters,
- sizes of (pre) selected trees,
- quality of (1), (2), etc..

Find earliest point at which we know the filter is true.

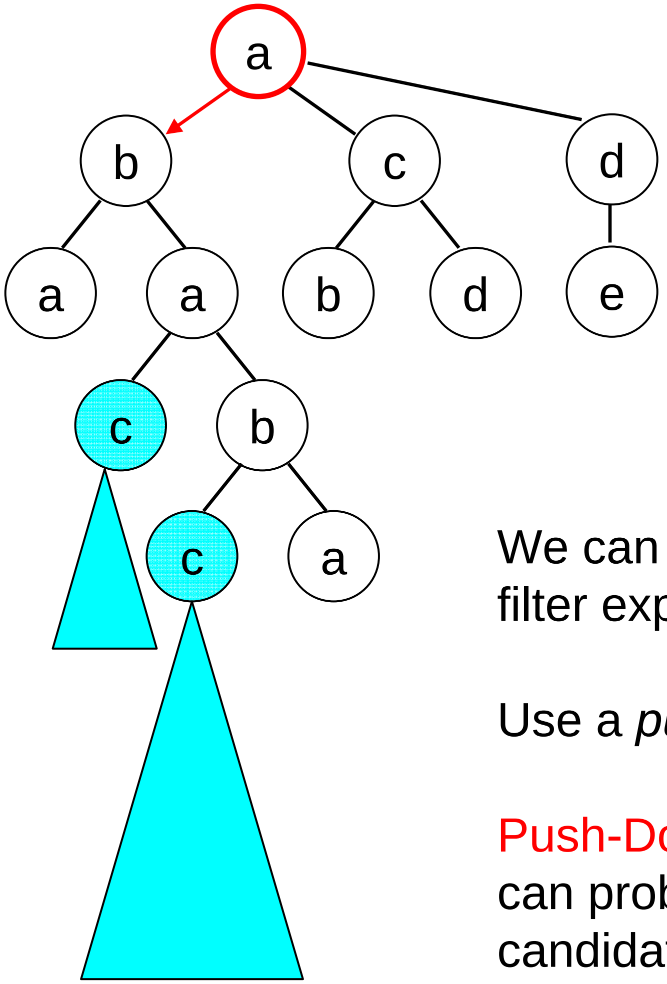
Harder for *Boolean combinations*:

[~~not(./d/e)~~ **and** (./c/d **or** //b/c)]

**Question** where is the earliest point for this filter?
→ and now?

How to deal with filters?

`//a[./d/e]/b//c`




➔ Size of largest documents that can be streamed in this way depends on
- #filters,
- sizes of (pre) selected trees,
- quality of (1), (2), etc..

We can also construct automata for filter expressions!

Use a *push-down* for potential candidates.

Push-Down Automaton
can probably be designed so that it pops/outputs candidates *as early as possible*.

How to deal with filters?

`//a[./d/e]/b//c`

Another Idea

Use **2-pass algorithm**: first (bottom-up) phase to mark subtrees with filter information.
Second (top-down) phase to determine match nodes.

Why is this interesting?

→ Fast main memory evaluation
→ Use disk as intermediate store  (stream twice)

# 5. Streaming XPath Algorithms

- XFilter and YFilter [Altinel and Franklin 00] [Diao et al 02]
- X-scan [Ives, Levy, and Weld 00]
- XMLTK [Avila-Campillo et al 02]
- XTrie [Chan et al 02]
- SPEX [Olteanu, Kiesling, and Bry 03]
- Lazy DFAs [Green et al 03]
- The XPush Machine [Gupta and Suciu 03]
- XSQ [Peng and Chawathe 03]
- TurboXPath [Josifovski, Fontoura, and Barta 04]
- …

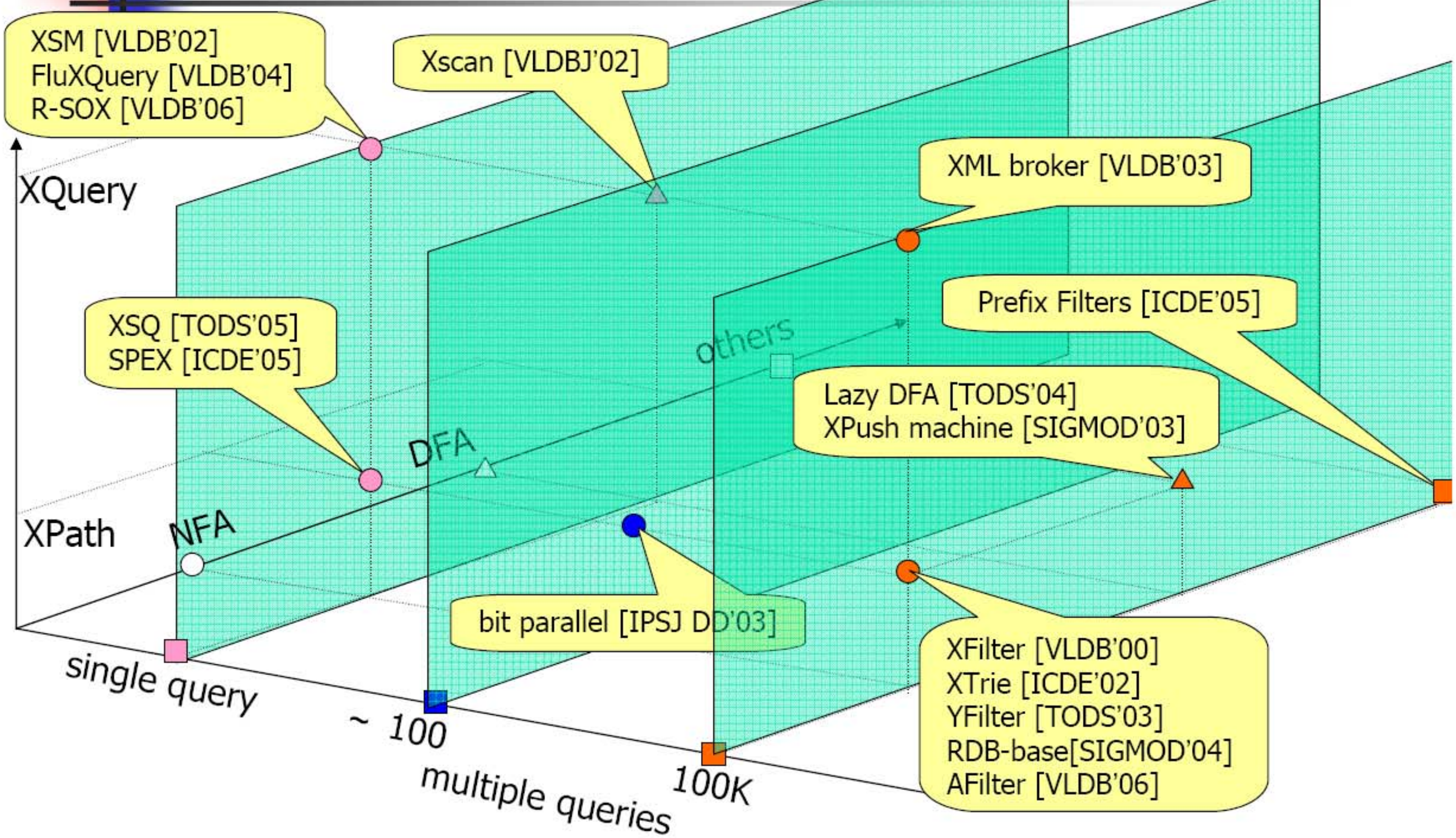
# 5. Streaming XPath Algorithms

Some following slides are by T. Amagasa and M Onizuka (Japan)
See http://www.dasfaa07.ait.ac.th/DASFAA2007_tutorial3_1.pdf

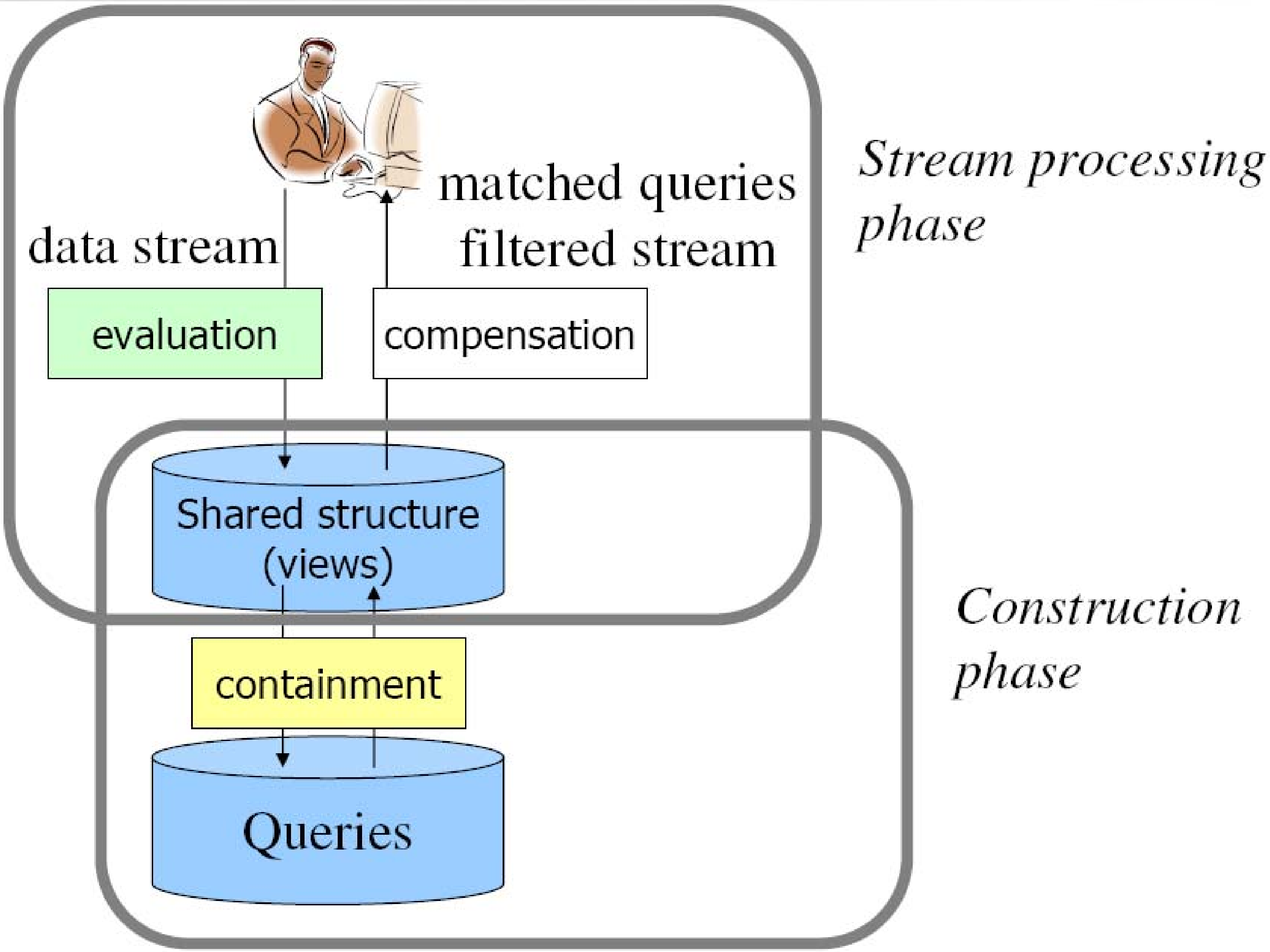
Most of the following slides are by Dan Suciu (the above slides are
Actually also based on Suciu's slides ☺ )
See
http://www.cs.washington.edu/homes/suciu/talk-spire2002.ppt

**Duality -> XML databases -> XML streams**

# Overviews (cont.)

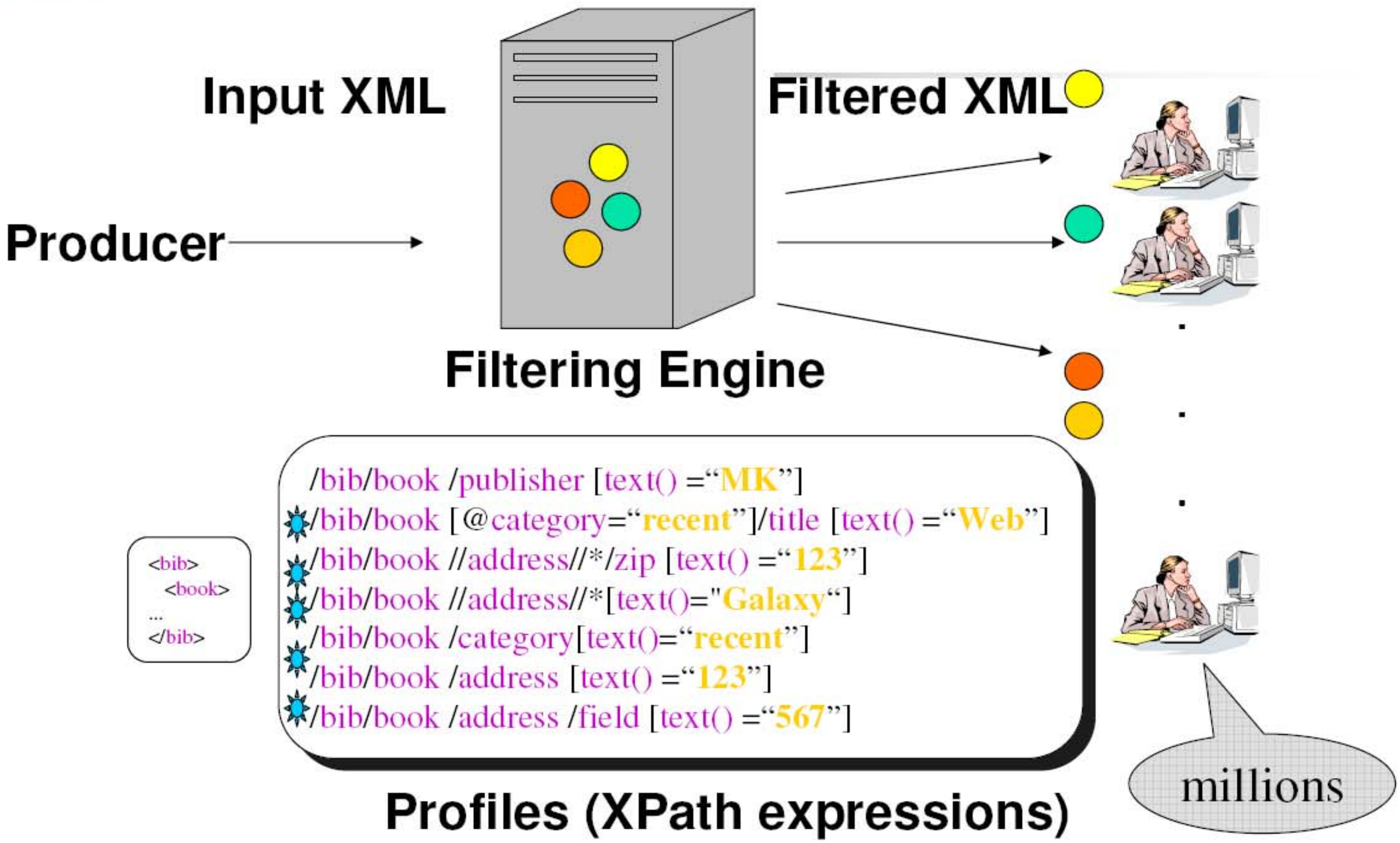


# Overview of XML stream



Duality -> XML databases -> XML streams

# SDI: Selective Dissemination of Information

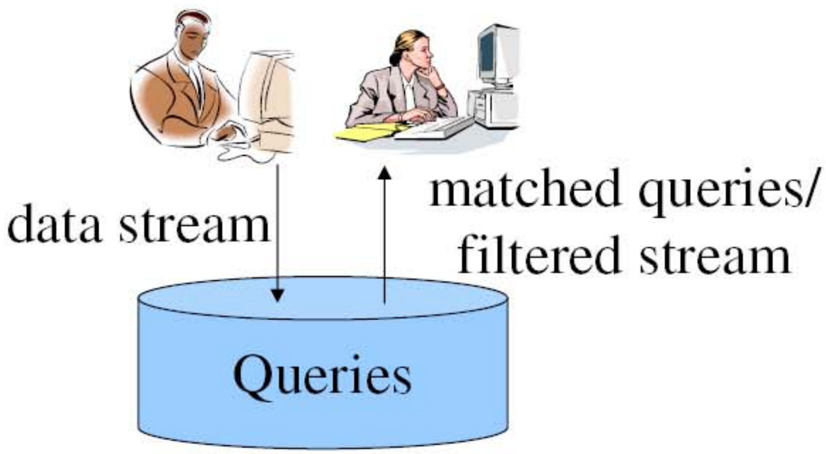


# XML stream applications

- SDI system/alert system

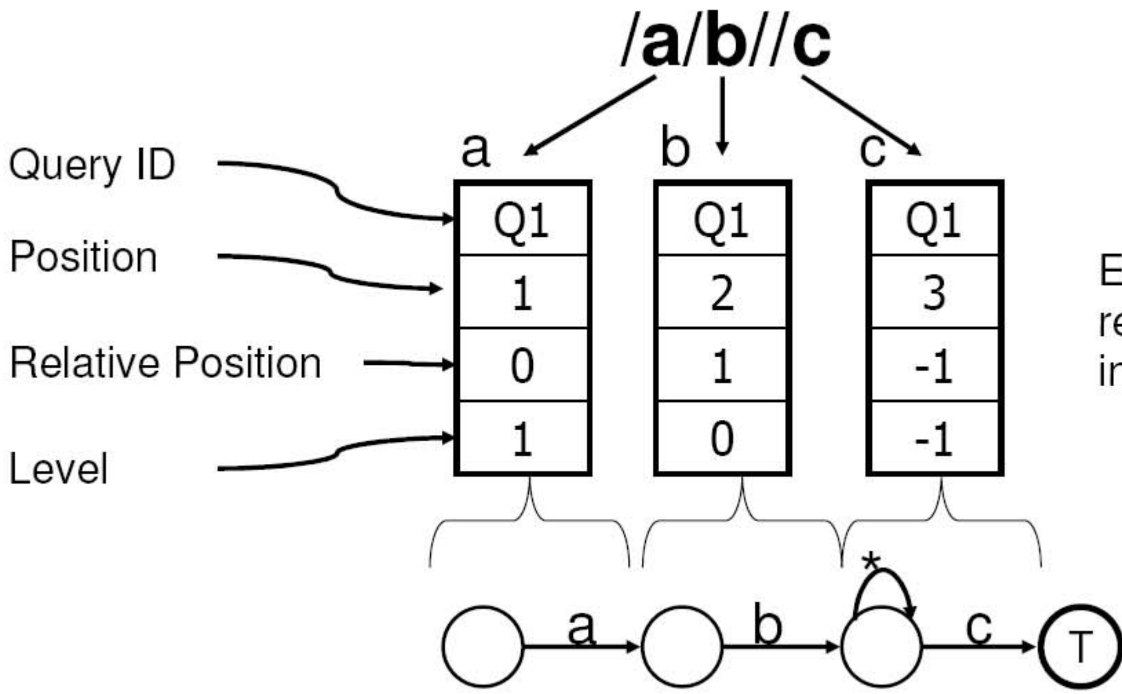
  stock, real estates, news feeds, flight departure/arrival
- Incremental transformation

  XTim [WWW'05], XPath maintenance [SIGMOD'05]



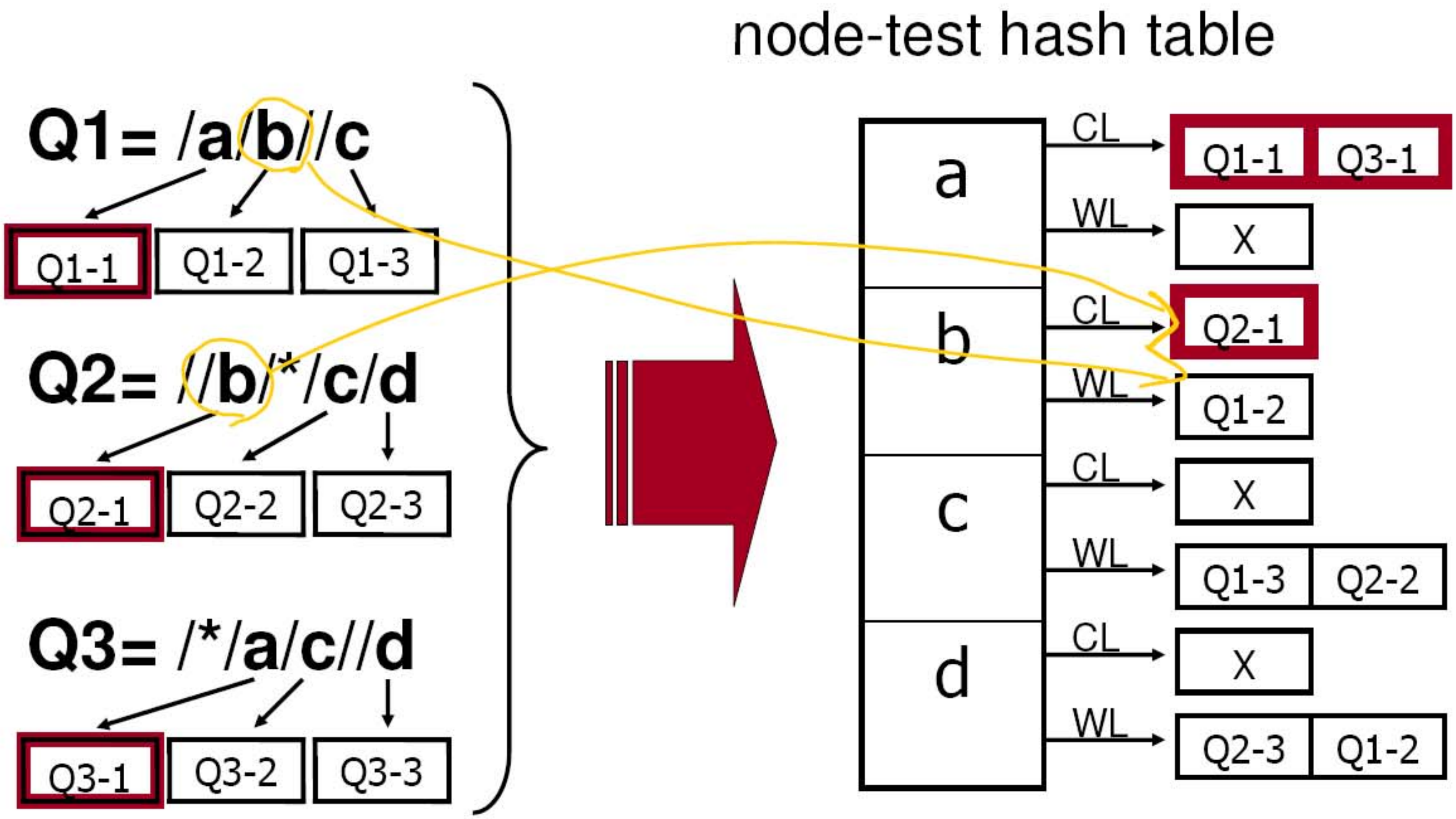
# XFilter (cont.)
# NFA, view class: //tag

## Decomposing XPath Query



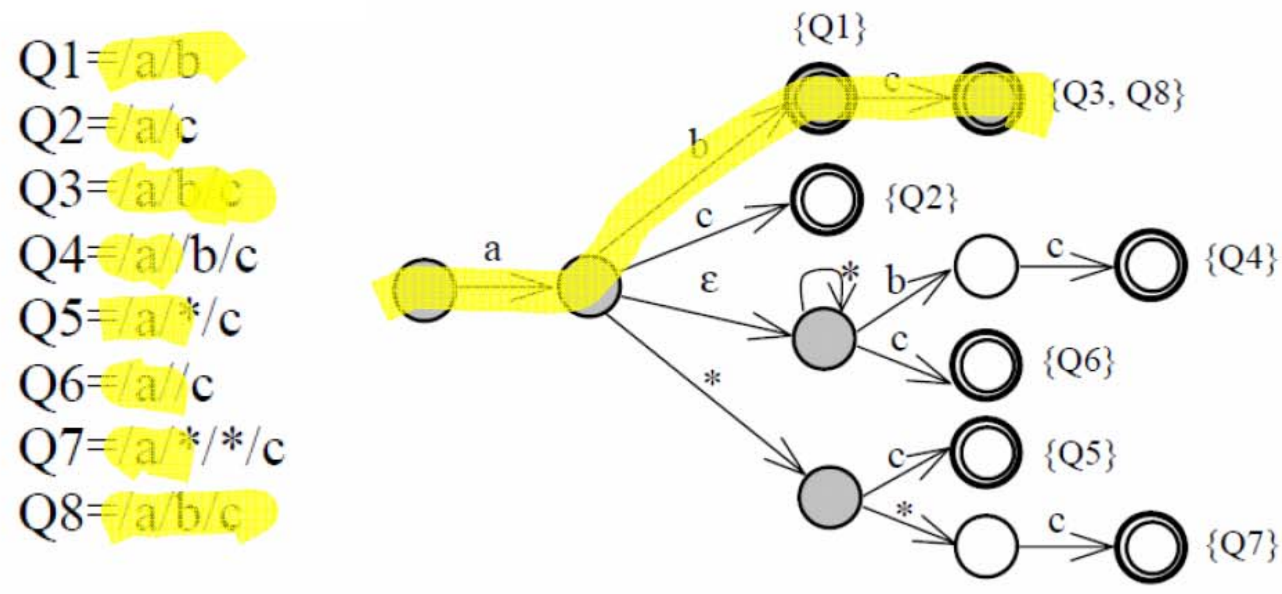

Every Path Node represents a state in the FSM

# XFilter (cont.)
## NFA, view class: //tag



# YFilter
## NFA, view class: XP{/,//,*}

- prefix sharing
- Predicates are processed by labels




(a) XPath queries

(b) A corresponding NFA (YFilter)

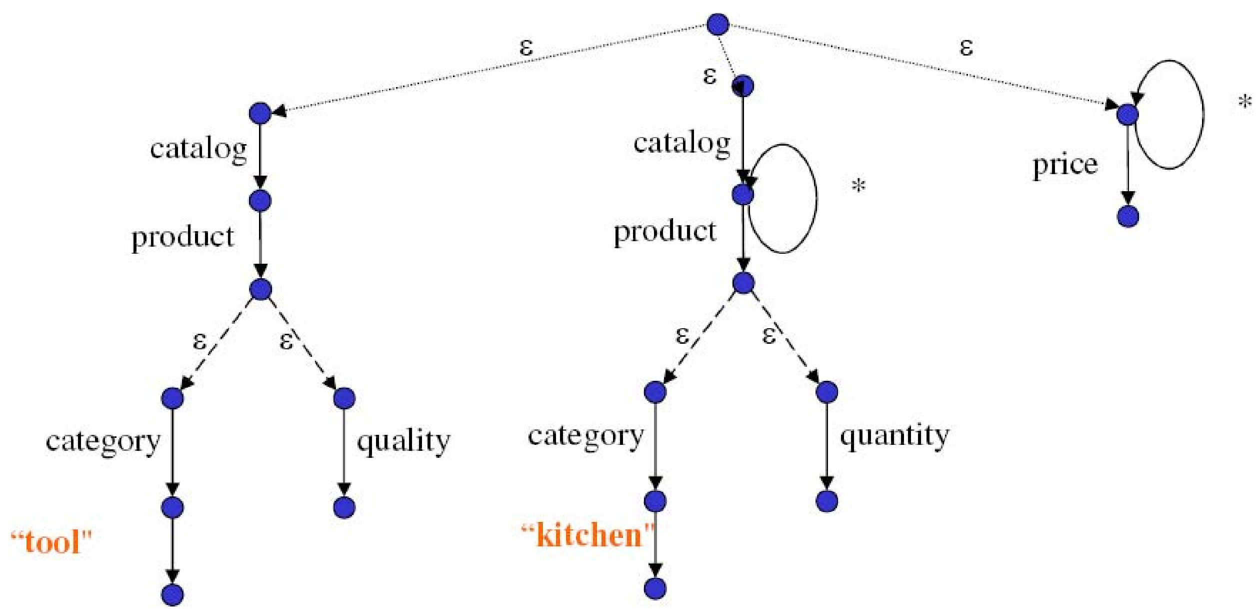
# Shared data structure

- Sharing identical structures among query trees
  What to share? node-test, simple path, branch, etc.

| What to share? | View class | Algorithms |
|---|---|---|
| node-test | //tag | XFilter [VLDB'00] |
| simple sub-path | //tag1/…/tagN | XTrie [ICDE'02] |
| simple path | XP{/,//,*} | YFilter [TODS'03], Lazy DFA [TODS'04], Prefix Filters [ICDE'05], AFilter [VLDB'06] |
| branch | XP{[],/,//,*} | XPush machine [SIGMOD'03] |
| … | … | … |

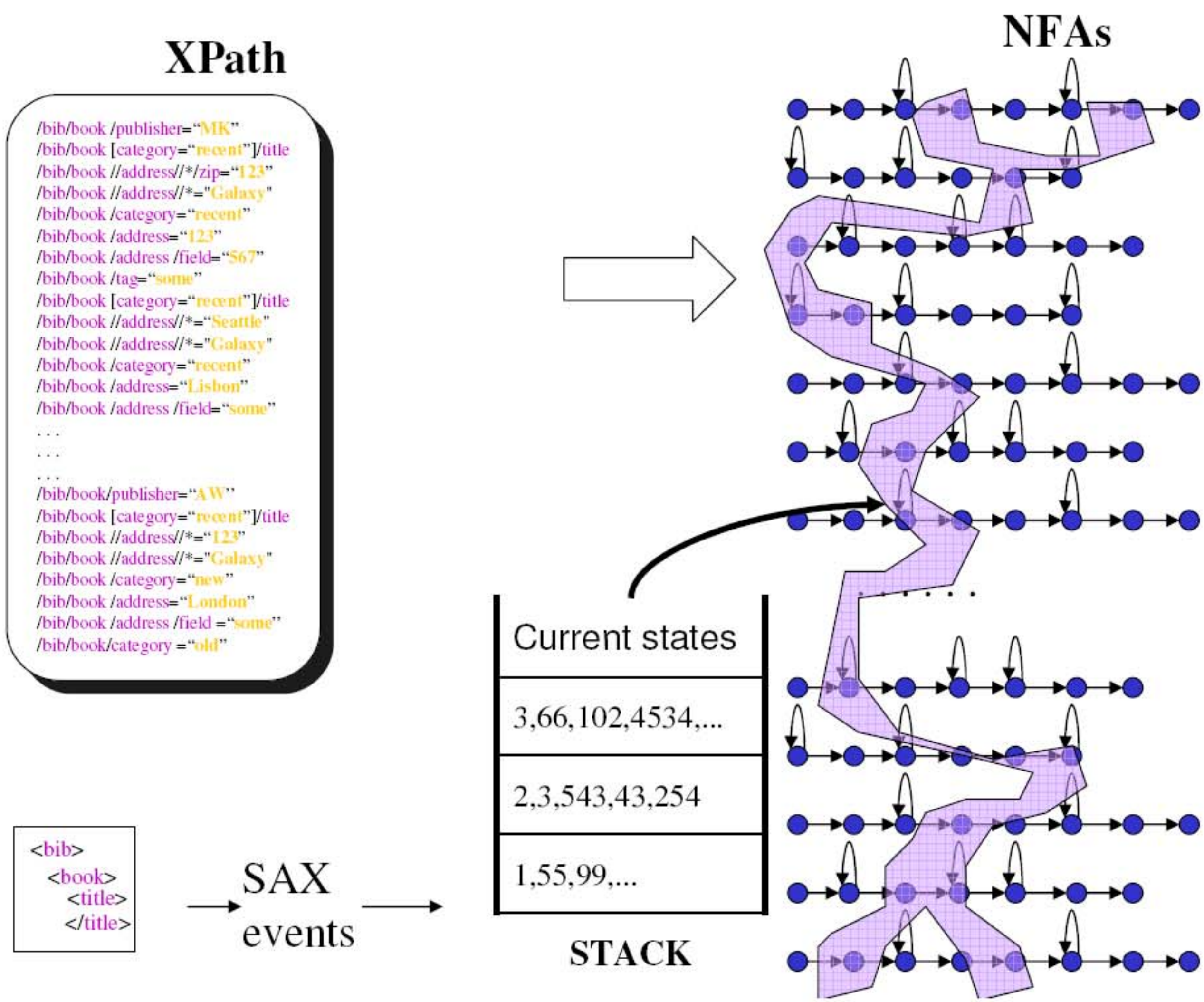
# XPath Processing with FA
## -- From XPath (XP{[],/,//,*) to NFA --

/catalog/product[category="**tools**"]/quantity
/catalog//product[category="**kitchen**"]/quality
//price



**Duality -> XML databases -> XML streams**

# NFA-based XPE Processing



# Basic NFA Evaluation

Properties:

☺ Space = linear

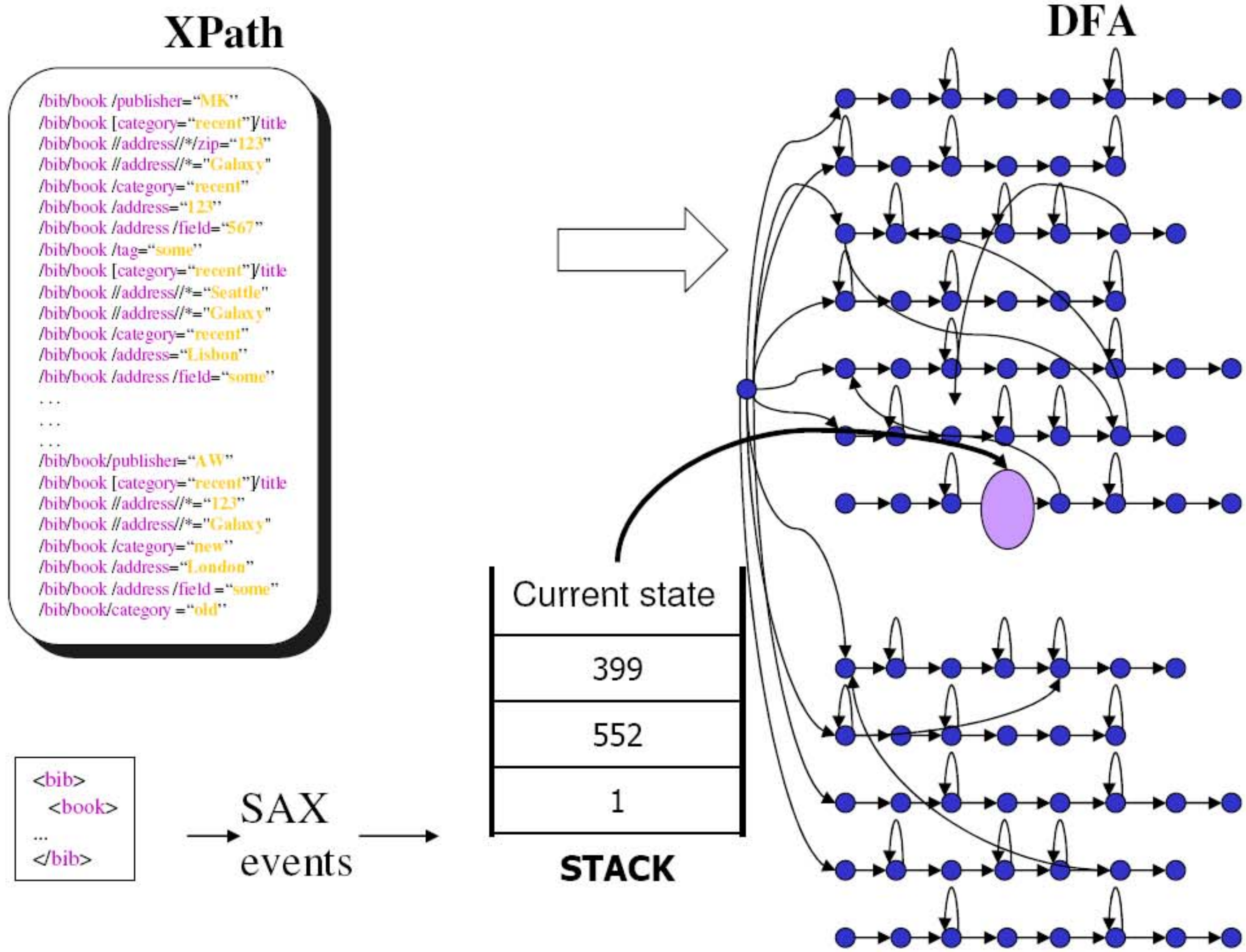
☹ Throughput = decreases linearly

Systems:

- XFilter [Altinel&Franklin'99], YFilter.
- XTrie [Chan et al.'02]

Duality -> XML databases -> XML streams

# DFA-based XPE Processing



# Basic DFA Evaluation

Properties:

☺ Throughput = constant !

☹ Space = GOOD QUESTION

System:

- XML Toolkit [University of Washington]
  **http://xmltk.sourceforge.net**

# The Size of the DFA

**Theorem** [GMOS'02] The number of states in the DFA for one linear XPath expression P is at most:

$$\boxed{k+|P|\, k\, s^m}$$

k = number of //

s = size of the alphabet (number of tags)

m = max number of * between two consecutive //

# Size of DFA: Multiple Expressions

//section//footnote
//table//footnote
//figure//footnote
. . . . .
//abstract//footnote

100 expressions

$2^{100}$ states !!

There is a theorem here too, but it's not useful…

# Solution:
# Compute the DFA Lazily

- Also used in text searching
- But will it work for $10^6$ XPath expressions ?
- YES !
- For XPath it is *provably* effective, for two reasons:
  - XML data is not very deep
  - The nesting structure in XML data tends to be predictable

# Lazy DFA
# DFA, view class: XP{/,//,*}

## Features

- Sharing the process of / and //, * and tag
- DFA-based
- Compute DFA lazily (on demand)
- # of DFA states
  - Independent from # of XPath exprs.
  - Depends on DataGuide size (schema)

## Issue

- Predicates: XPush machine [SIGMOD'03]

# Lazy DFA and “Simple” DTDs

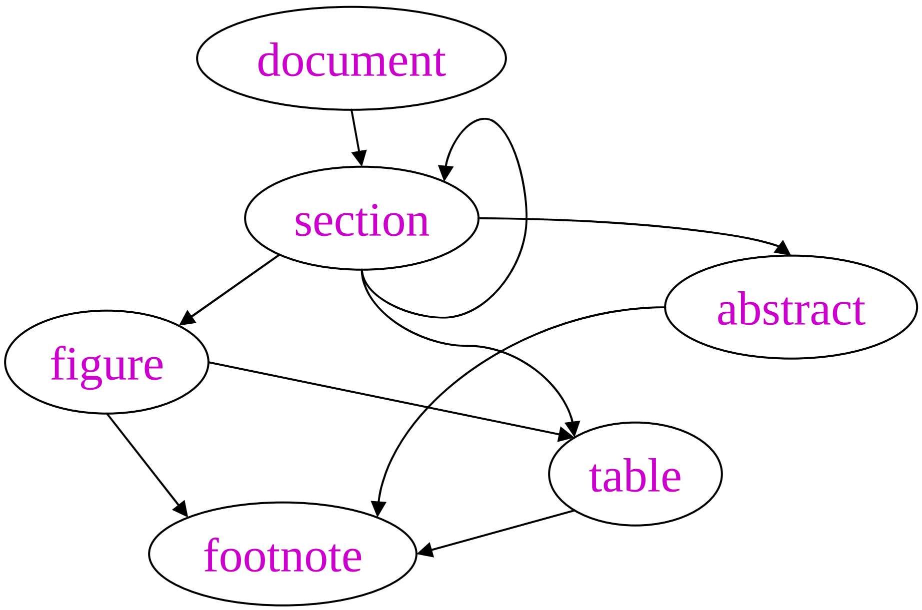
- Document Type Definition (DTD)
  - Part of the XML standard
  - Will be replaced by XML Schema
- Example DTD:

```
<!ELEMENT document (section*)>
<!ELEMENT section ((section|abstract|table|figure)*)>
<!ELEMENT figure (table?,footnote*)>
. . . . .
```

**Definition** A DTD is simple if all cycles are loops

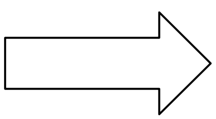
# Lazy DFA and "Simple" DTDs

Simple DTD:




XPath expressions

//section//footnote
//table//footnote
//figure//footnote
//abstract//footnote



Eager DFA "remembers" $2^4$ sets

Lazy DFA "remembers" only 4 sets

# Lazy DFA and "Simple" DTDs

**Theorem** [GMOS'02] If the XML data has a "simple" DTD, then lazy DFA has at most:

$$\boxed{1+D(1+n)^d}$$

states.

n = max depths of XPath expressions

D = size of the "unfolded" DTD

d = max depths of self-loops in the DTD

**Fact of life**: "Data-like" XML has simple DTDs

# Lazy DFA and Data Guides

- “Non-simple” DTDs are useless for the lazy DFA
- “Everything may contain everything”

```
<!ELEMENT document (section*)>
<!ELEMENT section  ((section|table|figure|abstract|footnote)*)>
<!ELEMENT table    ((section|table|figure|abstract|footnote)*)>
<!ELEMENT figure   ((section|table|figure|abstract|footnote)*)>
<!ELEMENT abstract ((section|table|figure|abstract|footnote)*)>
```

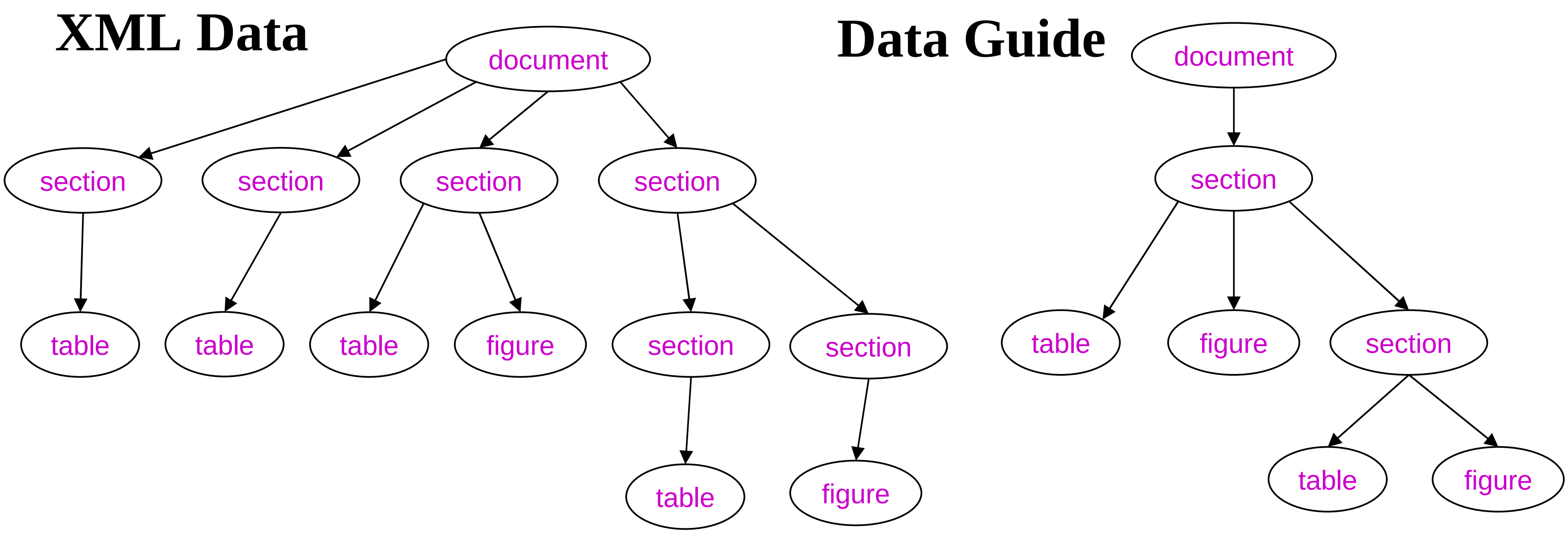
Fact of life: “Text”-like XML has non-simple DTDs

# Lazy DFA and Data Guides

**Definition** [Goldman&Widom'97]

The data guide for an XML data instance is the Trie of all its root-to-leaf paths

# Lazy DFA and Data Guides



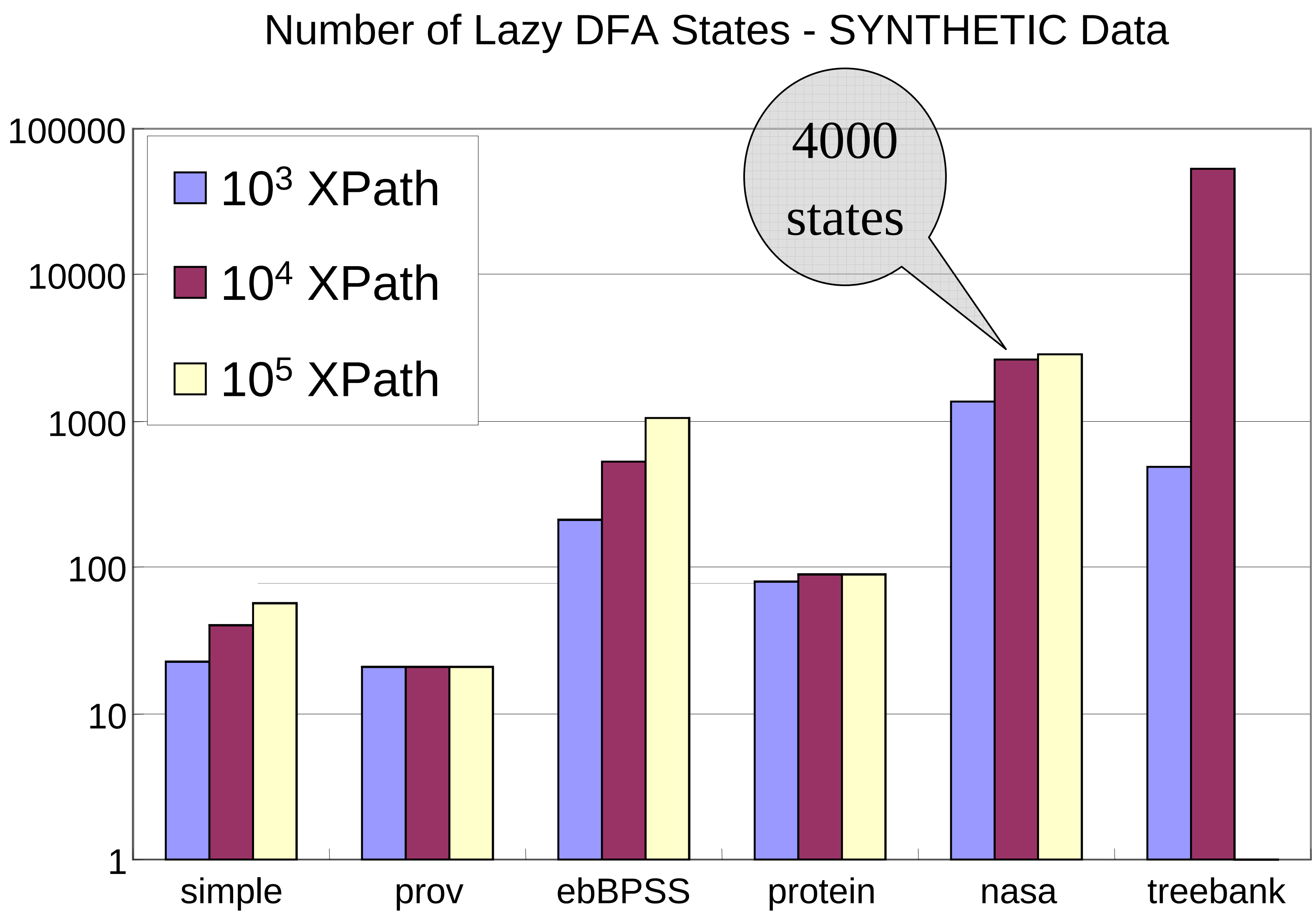

Fact of life: real XML data has "small" data guide [Liefke&S.'00]

# Lazy DFA and “Simple” DTDs

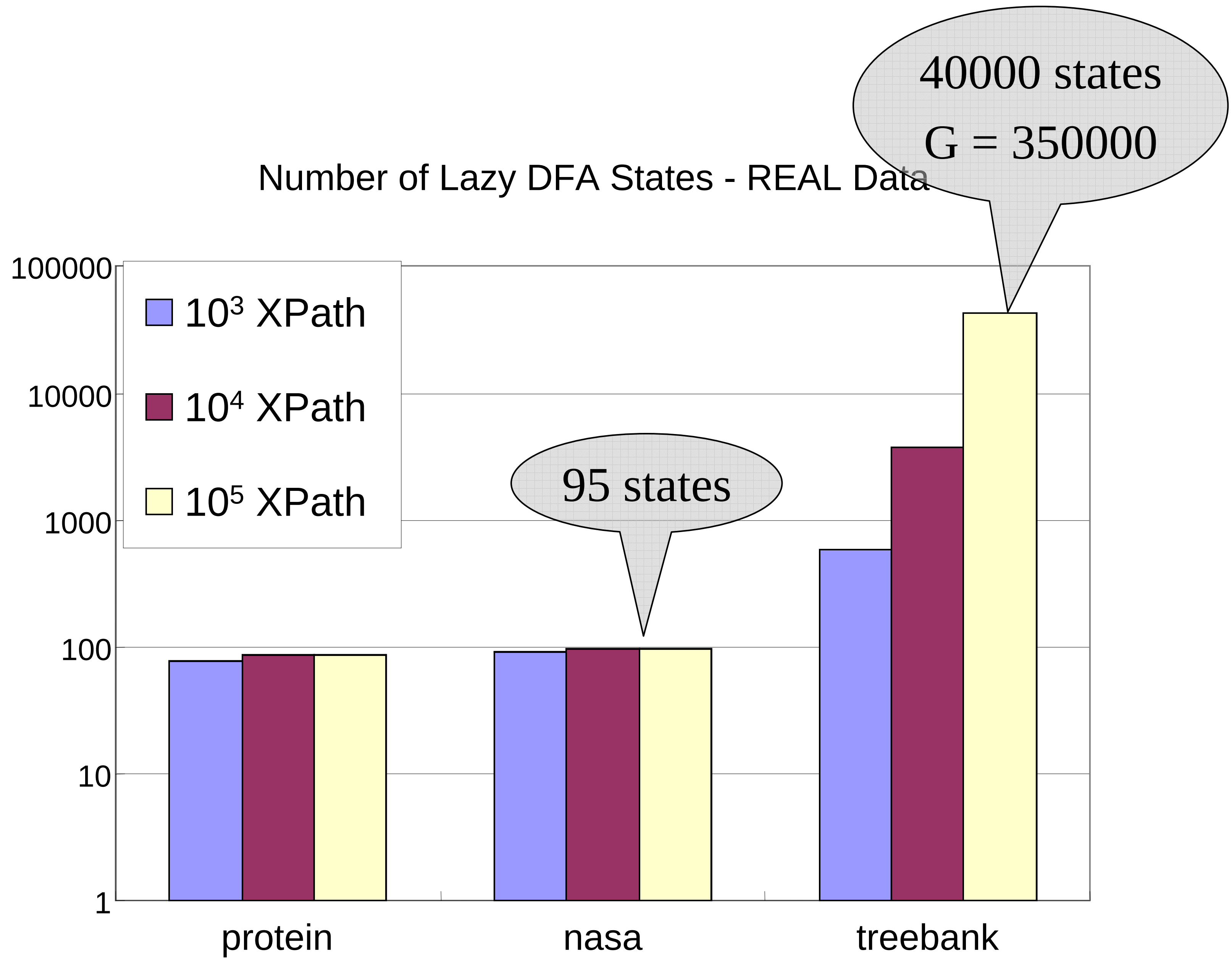
**Theorem** [GMOS’02] If the XML data has a data guide with G nodes, then the number of states in the lazy DFA is at most:

$$\boxed{1+G}$$

G = number of nodes in the data guide

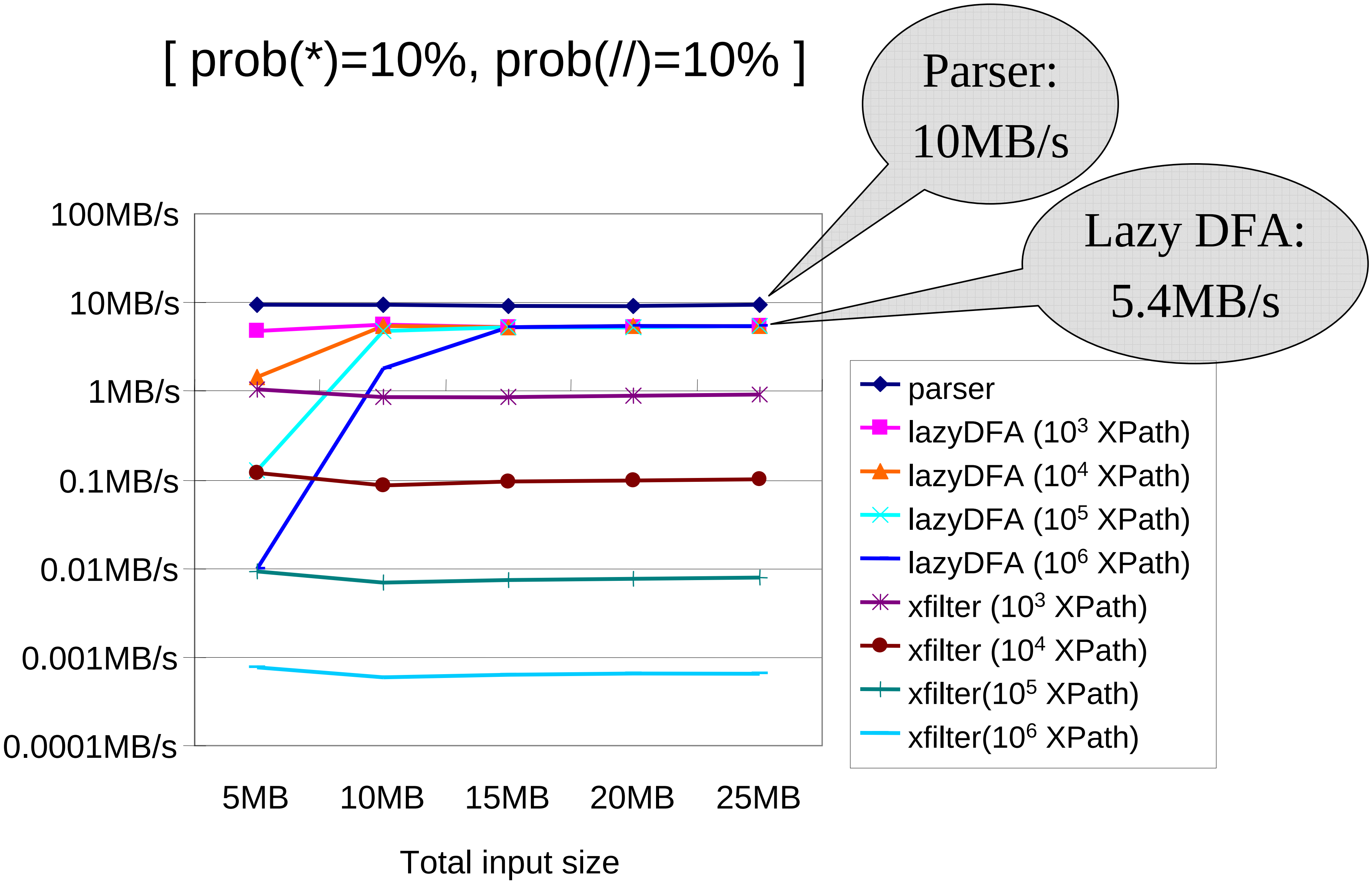


Number of Lazy DFA States - SYNTHETIC Data
4000 states
$10^3$ XPath
$10^4$ XPath
$10^5$ XPath
100000
10000
1000
100
10
1
simple
prov
ebBPSS
protein
nasa
treebank


40000 states
G = 350000
Number of Lazy DFA States - REAL Data
100000
10000
1000
100
10
1
10³ XPath
10⁴ XPath
10⁵ XPath
95 states
protein
nasa
treebank

# Throughput for $10^3$, $10^4$, $10^5$, $10^6$ XPath expressions

[ prob(*)=10%, prob(//)=10% ]



END

Lecture 9