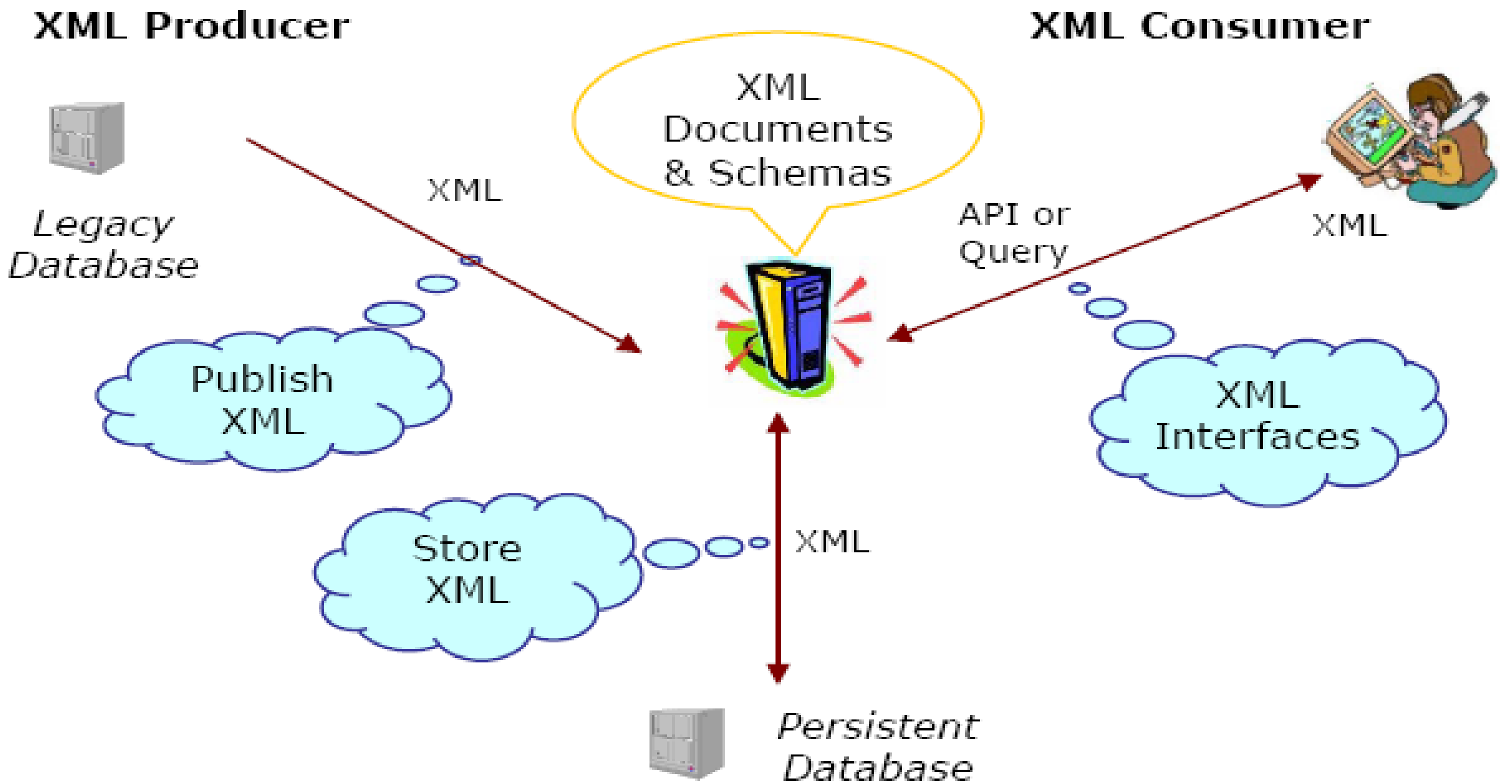
Web Data Management

Storing and Querying XML

XML Data Management

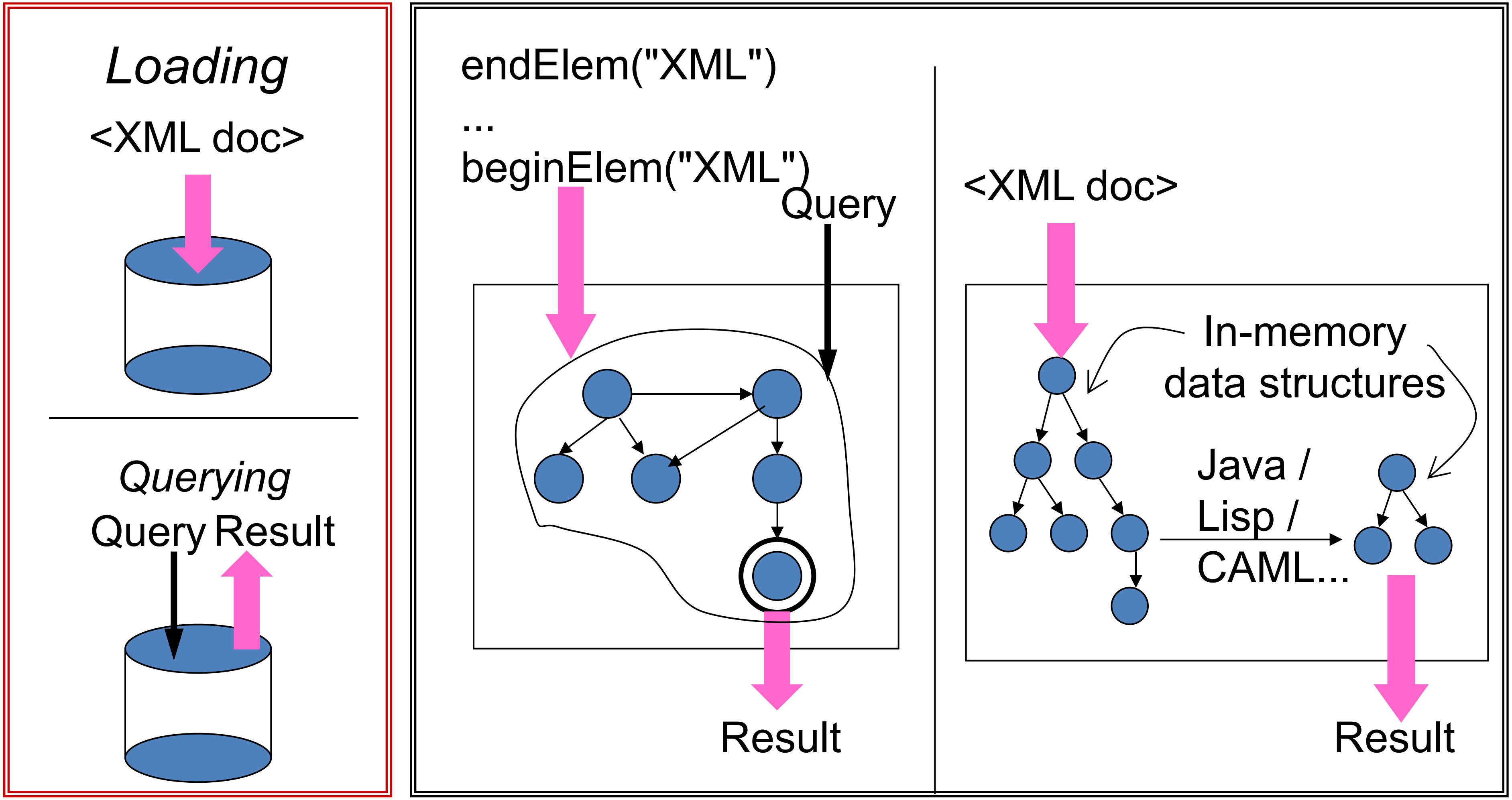


Context: XML query processing

Several possible flows

- *Data comes from persistent (disk-based) storage*
  - *First load, then query*
- Query processing "at first sight"
  - Data is queried when it is first seen
  - Not our topic [LMP02], [FSC+03], [BCF03], [FHK+03]

XML query processing scenarios



4

XML query processing scenarios (1/2)

"Persistent store"

Logging / archiving an ongoing activity

- Clients, orders, products...
- Structured text (documentation, news, image annotations, scientific data...)

Warehousing XML

"At first sight"

Fast processing of incoming documents

- Web service messages
- Workflow coordination

Many small documents to process

- In-memory, programming language approach feasible

XML query processing scenarios (2/2)

"Persistent store"

Heavier
- Needs loading

All DBMS goodies
- Set-at-a-time processing
- Query optimization
- Persistence
- Transactions
- Concurrence control
- View-based management...

"At first sight"

Lighter

May blend easily into a programming framework
- In real life, there are not just databases

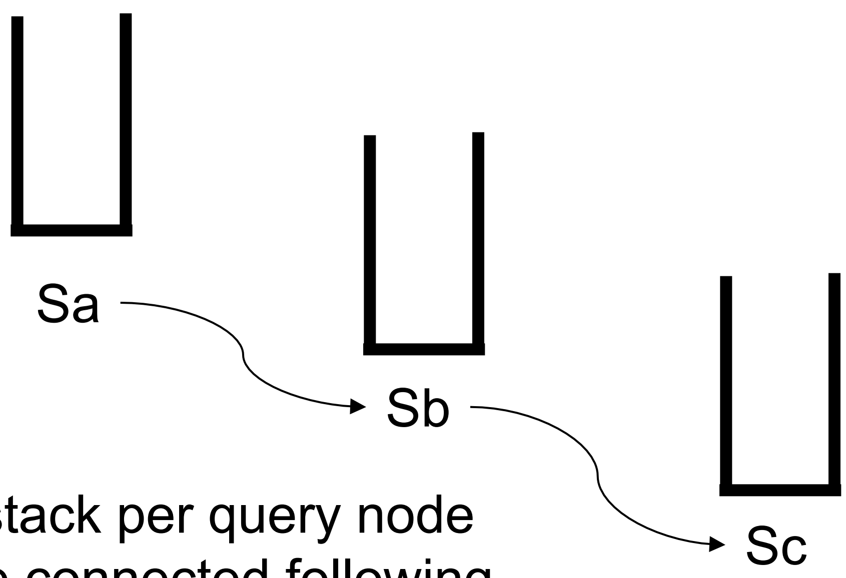
Streaming (stack-based) evaluation of tree pattern queries

Streaming processing of tree pattern queries

XML document:

<r>
<a1><b1>
<c1/>
<c2/>
</b1>
<b2>
<c3/>
</b2>
</a1>
<a2>
<c4/>
<b3/>
</a2>
</r>

Query: //a/b/c



Create 1 stack per query node
Stacks are connected following the query structure

Streaming processing of tree pattern queries

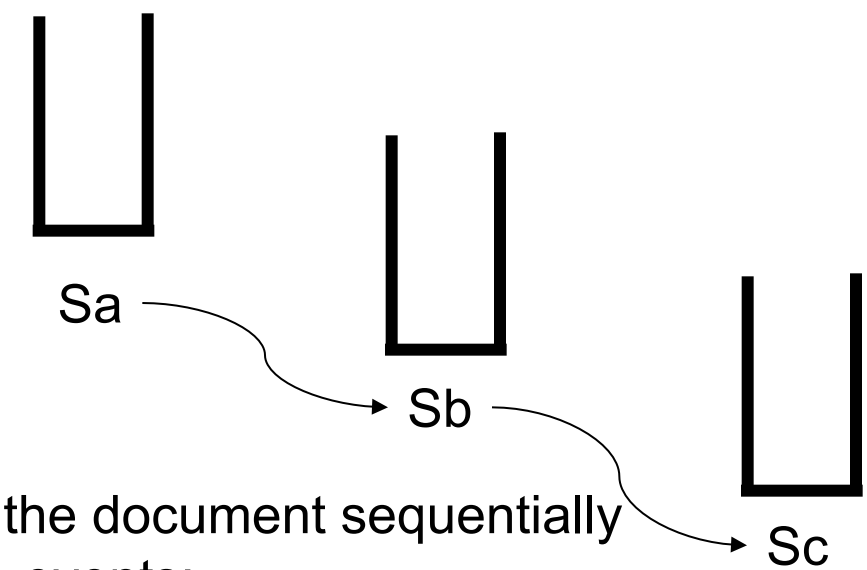
```
<r><a1><b1><c1/><c2/></b1><b2><c3/></b2></a1><a2><c4/><b3/></a2></r>
1   2     3     4    5 6    7  8     9      10 11  12    13     14     15 16  17 18  19     20
```

XML document:

```
<r>
  <a1><b1>
        <c1/>
        <c2/>
    </b1>
    <b2>
        <c3/>
    </b2>
  </a1>
  <a2>
    <c4/>
    <b3/>
  </a2>
</r>
```

SAX traversal

Query: //a/b/c



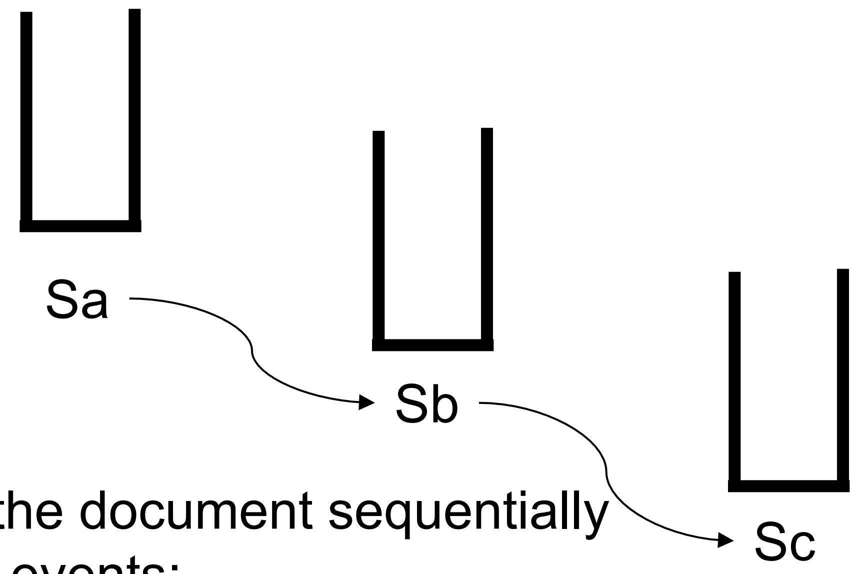
Traverse the document sequentially and issue <u>events</u>:
<u>Start element x</u> / <u>end element x</u> / <u>text s</u>

Streaming processing of tree pattern queries

<r><a1><b1><c1/><c2/></b1><b2><c3/></b2></a1><a2><c4/><b3/></a2></r>

1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22

Query: //a/b/c



Traverse the document sequentially and issue <u>events</u>:
<u>Start element x</u> / <u>end element x</u> / <u>text s</u>

Streaming processing of tree pattern queries

<r><a1><b1><c1/><c2/></b1><b2><c3/></b2></a1><a2><c4/><b3/></a2></r>

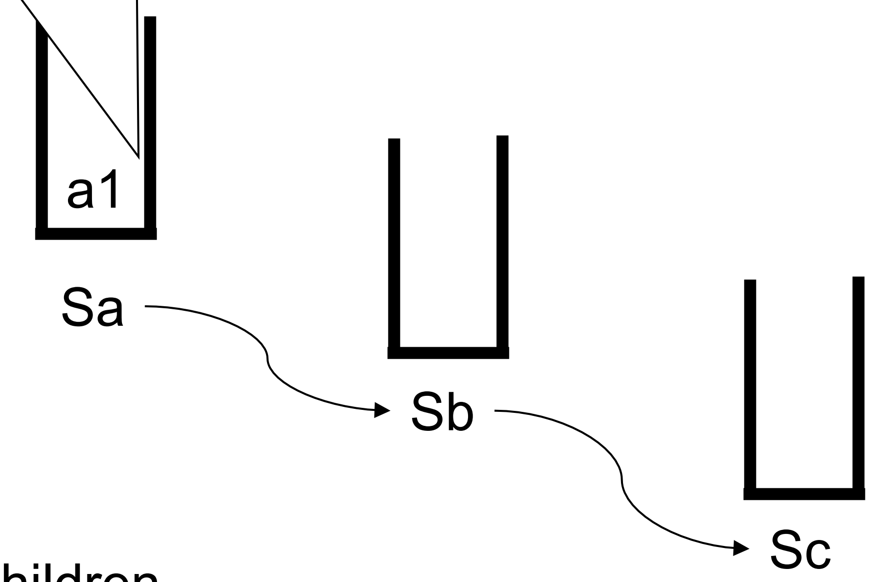
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20

When pushed, matches are open

Query: //a/b/c

On <u>begin element x</u>:
If there is a stack for x
Then if the element appears
in the right context
then push it on the stack;
connect it to the
parent match

On <u>end element x</u>:
If there is a stack for x
Then if x is on top of the stack
then if x lacks some required children
then pop x, possibly some desc



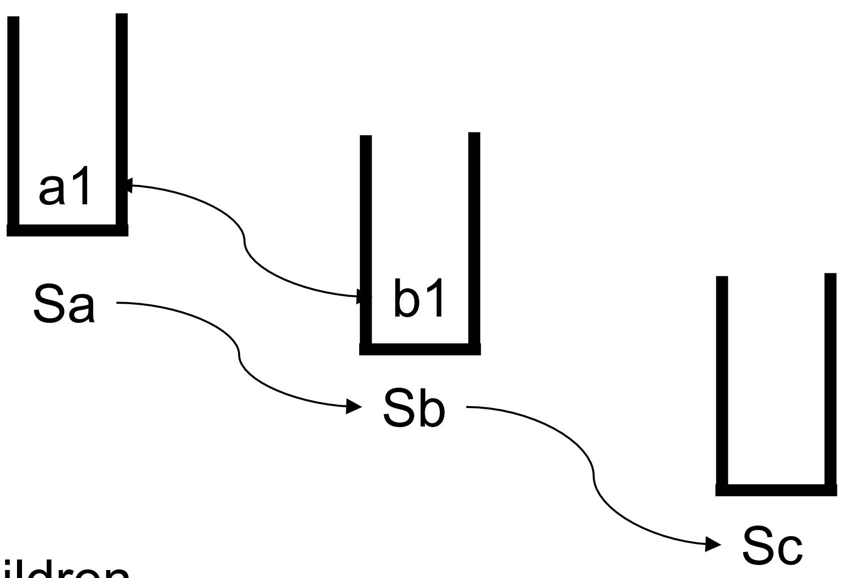
Streaming processing of tree pattern queries

<r><a1><b1><c1/><c2/></b1><b2><c3/></b2></a1><a2><c4/><b3/></a2></r>

1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20

On <u>begin element x</u>:
If there is a stack for x
Then if the element appears
in the right context
then push it on the stack;
connect it to the
parent match
On <u>end element x</u>:
If there is a stack for x
Then if x is on top of the stack
then if x lacks some required children
then pop x, possibly some desc

Query: //a/b/c



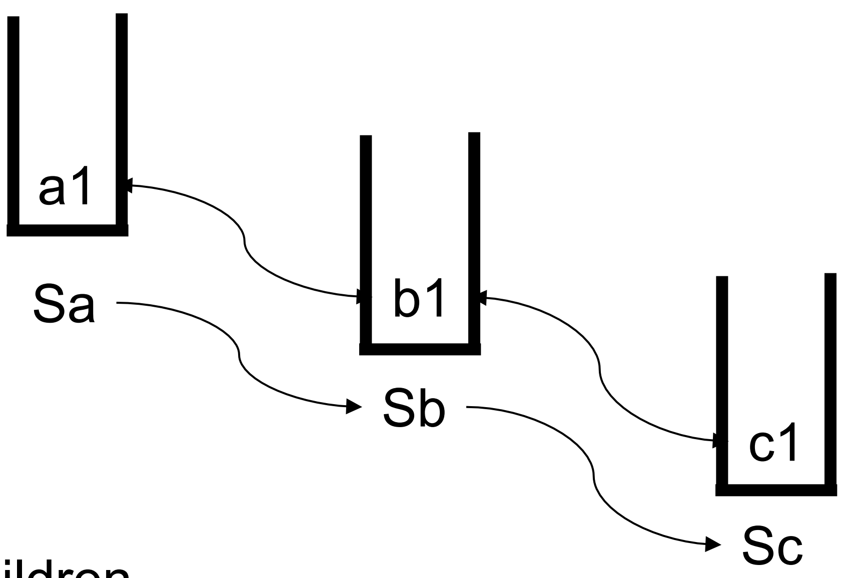
Streaming processing of tree pattern queries

<r><a1><b1><c1/><c2/></b1><b2><c3/></b2></a1><a2><c4/><b3/></a2></r>

1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20

On <u>begin element x</u>:
If there is a stack for x
Then if the element appears
in the right context
then push it on the stack;
connect it to the
parent match
On <u>end element x</u>:
If there is a stack for x
Then if x is on top of the stack
then if x lacks some required children
then pop x, possibly some desc

Query: //a/b/c



Streaming processing of tree pattern queries

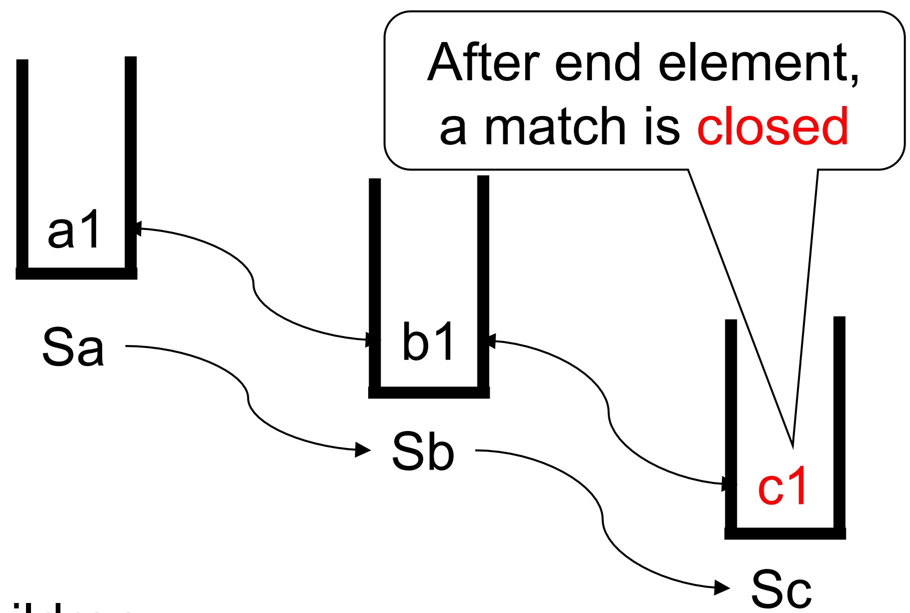
<r><a1><b1><c1/><c2/></b1><b2><c3/></b2></a1><a2><c4/><b3/></a2></r>

1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20

On <u>begin element x</u>:
If there is a stack for x
Then if the element appears
    in the right context
  then push it on the stack;
    connect it to the
    parent match

On <u>end element x</u>:
If there is a stack for x
Then if x is on top of the stack
  then if x lacks some required children
    then pop x, possibly some desc

Query: //a/b/c



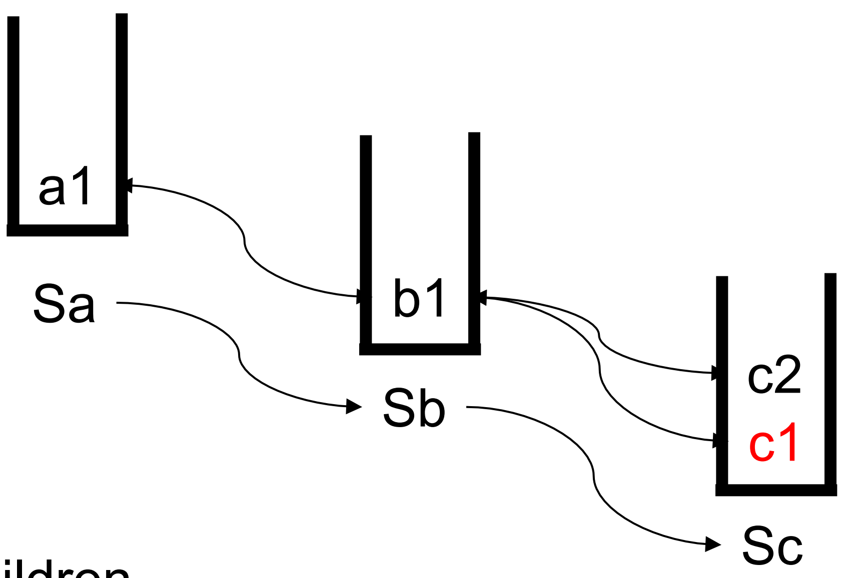
Streaming processing of tree pattern queries

<r><a1><b1><c1/><c2/></b1><b2><c3/></b2></a1><a2><c4/><b3/></a2></r>

1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20

On <u>begin element x</u>:
If there is a stack for x
Then if the element appears
in the right context
then push it on the stack;
connect it to the
parent match
On <u>end element x</u>:
If there is a stack for x
Then if x is on top of the stack
then if x lacks some required children
then pop x, possibly some desc

Query: //a/b/c



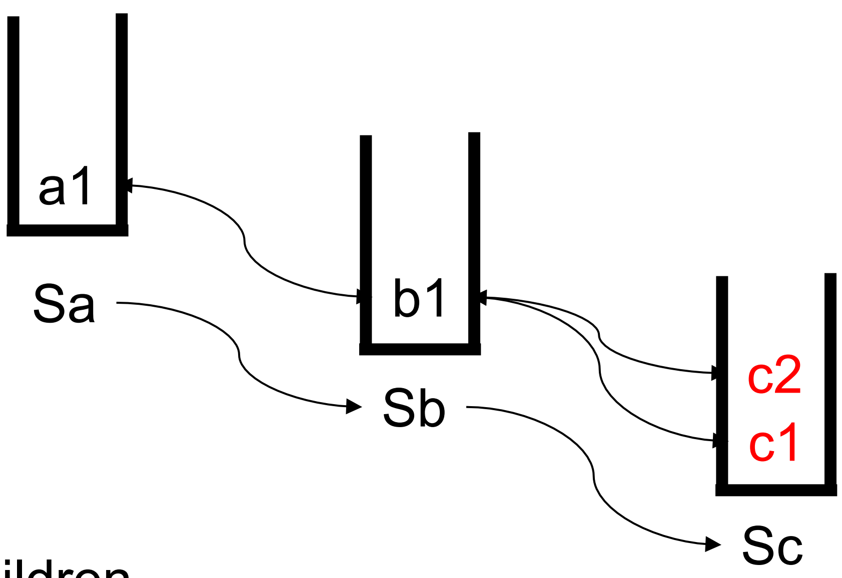
Streaming processing of tree pattern queries

<r><a1><b1><c1/><c2/></b1><b2><c3/></b2></a1><a2><c4/><b3/></a2></r>

1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20

Query: //a/b/c

On <u>begin element x</u>:
If there is a stack for x
Then if the element appears
in the right context
then push it on the stack;
connect it to the
parent match
On <u>end element x</u>:
If there is a stack for x
Then if x is on top of the stack
then if x lacks some required children
then pop x, possibly some desc



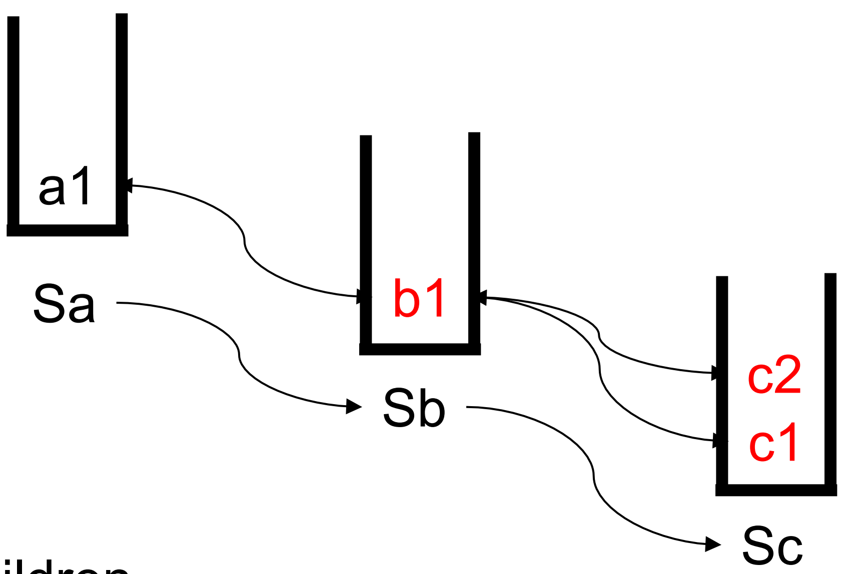
Streaming processing of tree pattern queries

<r><a1><b1><c1/><c2/></b1><b2><c3/></b2></a1><a2><c4/><b3/></a2></r>

1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20

Query: //a/b/c

On <u>begin element x</u>:
If there is a stack for x
Then if the element appears
in the right context
then push it on the stack;
connect it to the
parent match
On <u>end element x</u>:
If there is a stack for x
Then if x is on top of the stack
then if x lacks some required children
then pop x, possibly some desc



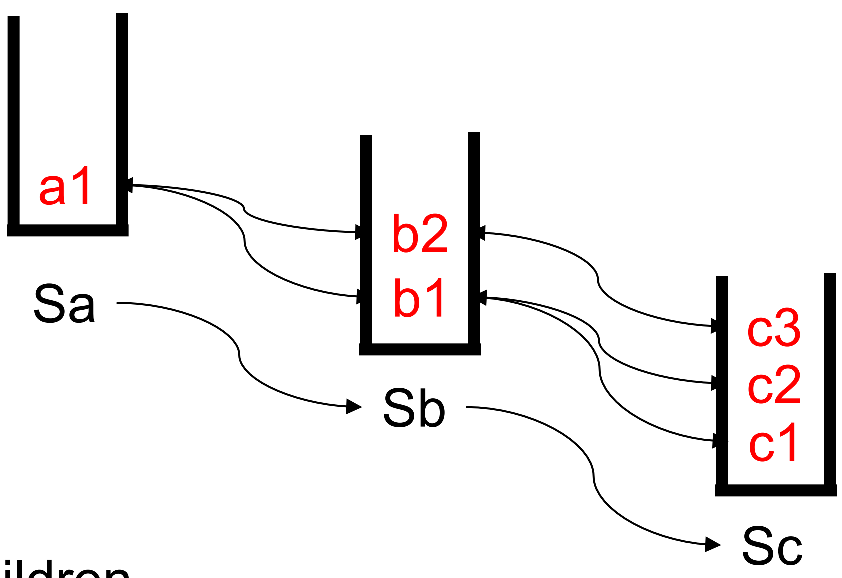
Streaming processing of tree pattern queries

<r><a1><b1><c1/><c2/></b1><b2><c3/></b2></a1><a2><c4/><b3/></a2></r>

1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20

Query: //a/b/c

On <u>begin element x</u>:
If there is a stack for x
Then if the element appears
in the right context
then push it on the stack;
connect it to the
parent match
On <u>end element x</u>:
If there is a stack for x
Then if x is on top of the stack
then if x lacks some required children
then pop x, possibly some desc



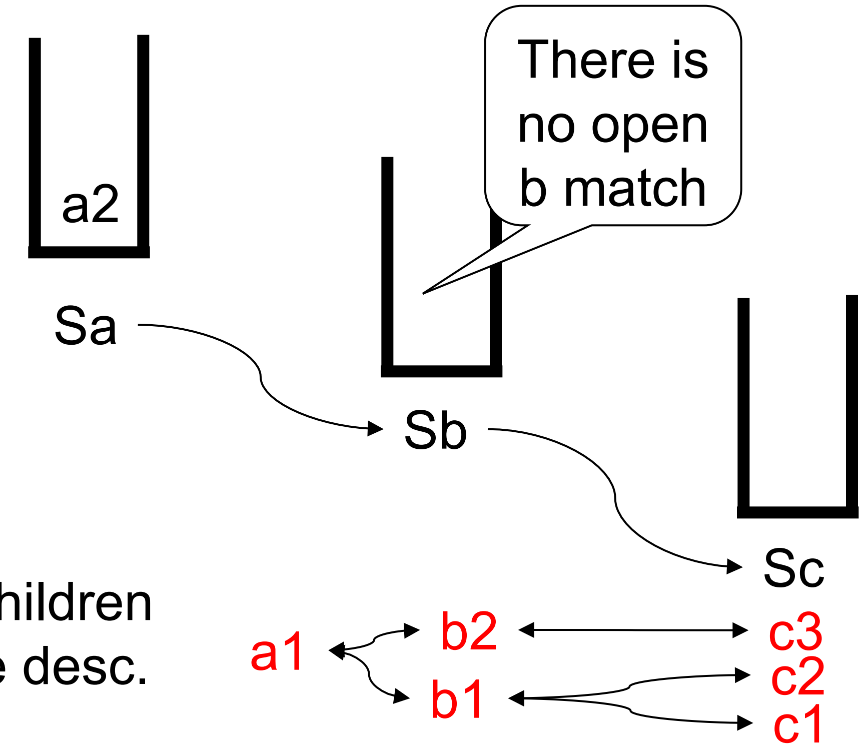
Streaming processing of tree pattern queries

<r><a1><b1><c1/><c2/></b1><b2><c3/></b2></a1><a2><c4/><b3/></a2></r>

1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20

Query: //a/b/c

On begin element x:
If there is a stack for x
Then if the element appears
in the right context
then push it on the stack;
connect it to the
parent match
On end element x:
If there is a stack for x
Then if x is on top of the stack
then if x lacks some required children
then pop x, possibly some desc.



Streaming processing of tree pattern queries

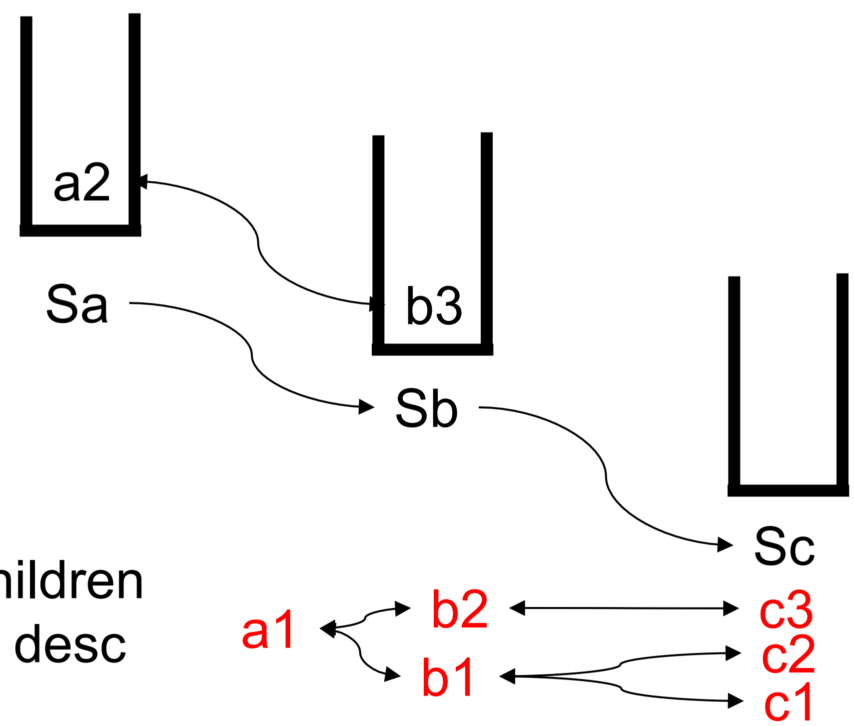
<r><a1><b1><c1/><c2/></b1><b2><c3/></b2></a1><a2><c4/><b3/></a2></r>

1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20

Query: //a/b/c

On <u>begin element x</u>:
If there is a stack for x
Then if the element appears
in the right context
then push it on the stack;
connect it to the
parent match

On <u>end element x</u>:
If there is a stack for x
Then if x is on top of the stack
then if x lacks some required children
then pop x, possibly some desc



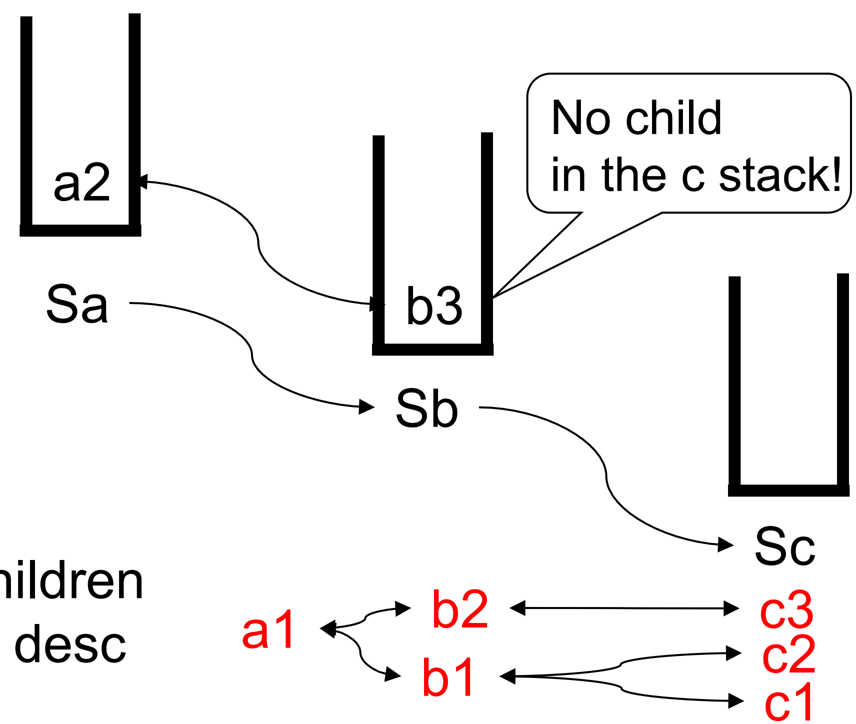
Streaming processing of tree pattern queries

<r><a1><b1><c1/><c2/></b1><b2><c3/></b2></a1><a2><c4/><b3/></a2></r>

1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20

Query: //a/b/c

On <u>begin element x</u>:
If there is a stack for x
Then if the element appears
in the right context
then push it on the stack;
connect it to the
parent match
On <u>end element x</u>:
If there is a stack for x
Then if x is on top of the stack
then if x lacks some required children
then pop x, possibly some desc



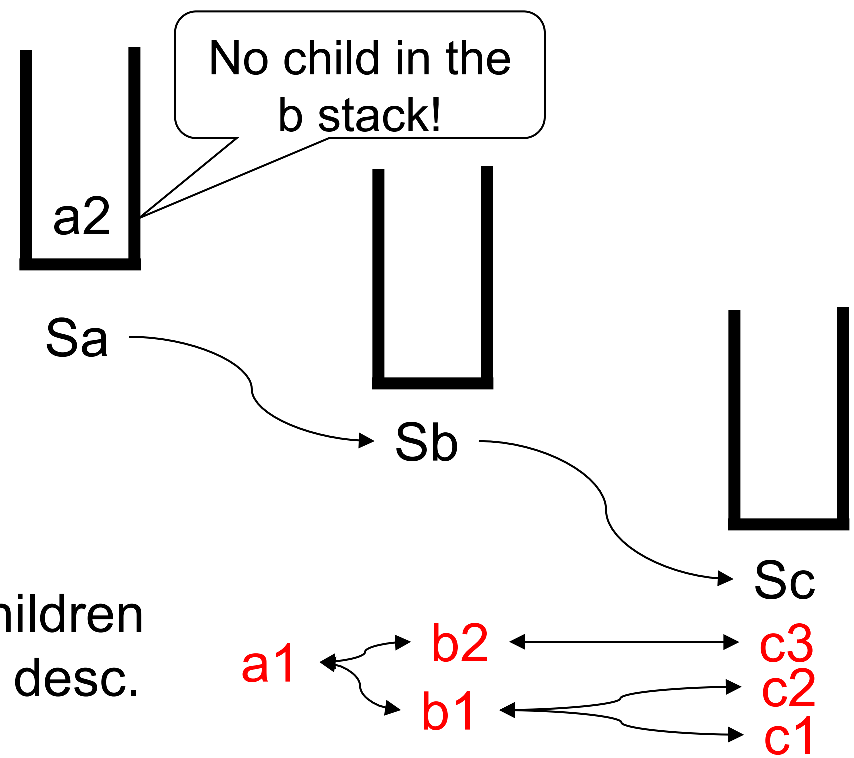
Streaming processing of tree pattern queries

<r><a1><b1><c1/><c2/></b1><b2><c3/></b2></a1><a2><c4/><b3/></a2></r>

1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20

Query: //a/b/c

On <u>begin element x</u>:
If there is a stack for x
Then if the element appears
in the right context
then push it on the stack;
connect it to the
parent match
On <u>end element x</u>:
If there is a stack for x
Then if x is on top of the stack
then if x lacks some required children
then pop x, possibly some desc.



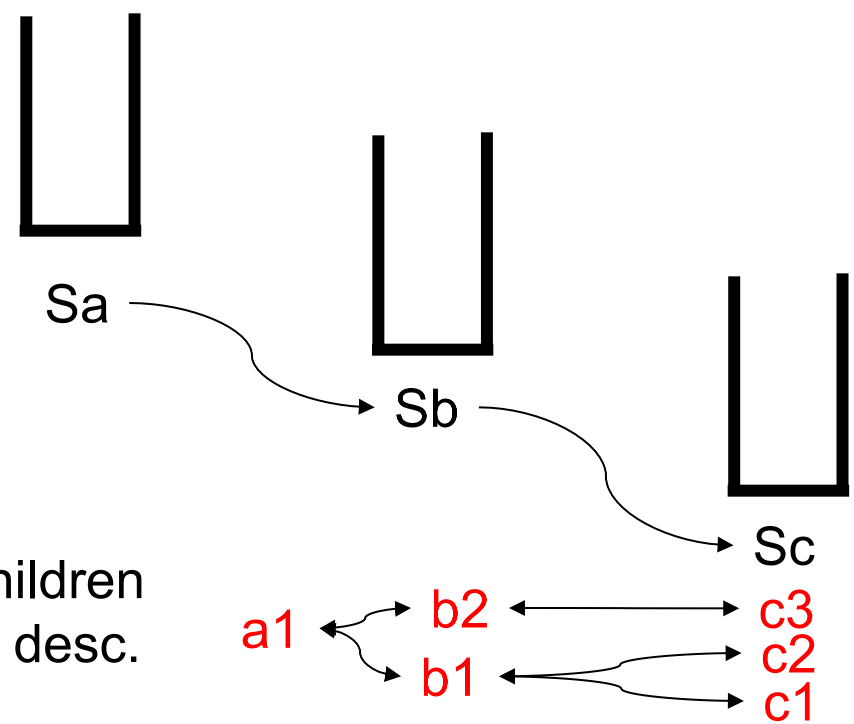
Streaming processing of tree pattern queries

<r><a1><b1><c1/><c2/></b1><b2><c3/></b2></a1><a2><c4/><b3/></a2></r>

1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20

Query: //a/b/c

On <u>begin element x</u>:
If there is a stack for x
Then if the element appears
in the right context
then push it on the stack;
connect it to the
parent match
On <u>end element x</u>:
If there is a stack for x
Then if x is on top of the stack
then if x lacks some required children
then pop x, possibly some desc.



Complexity

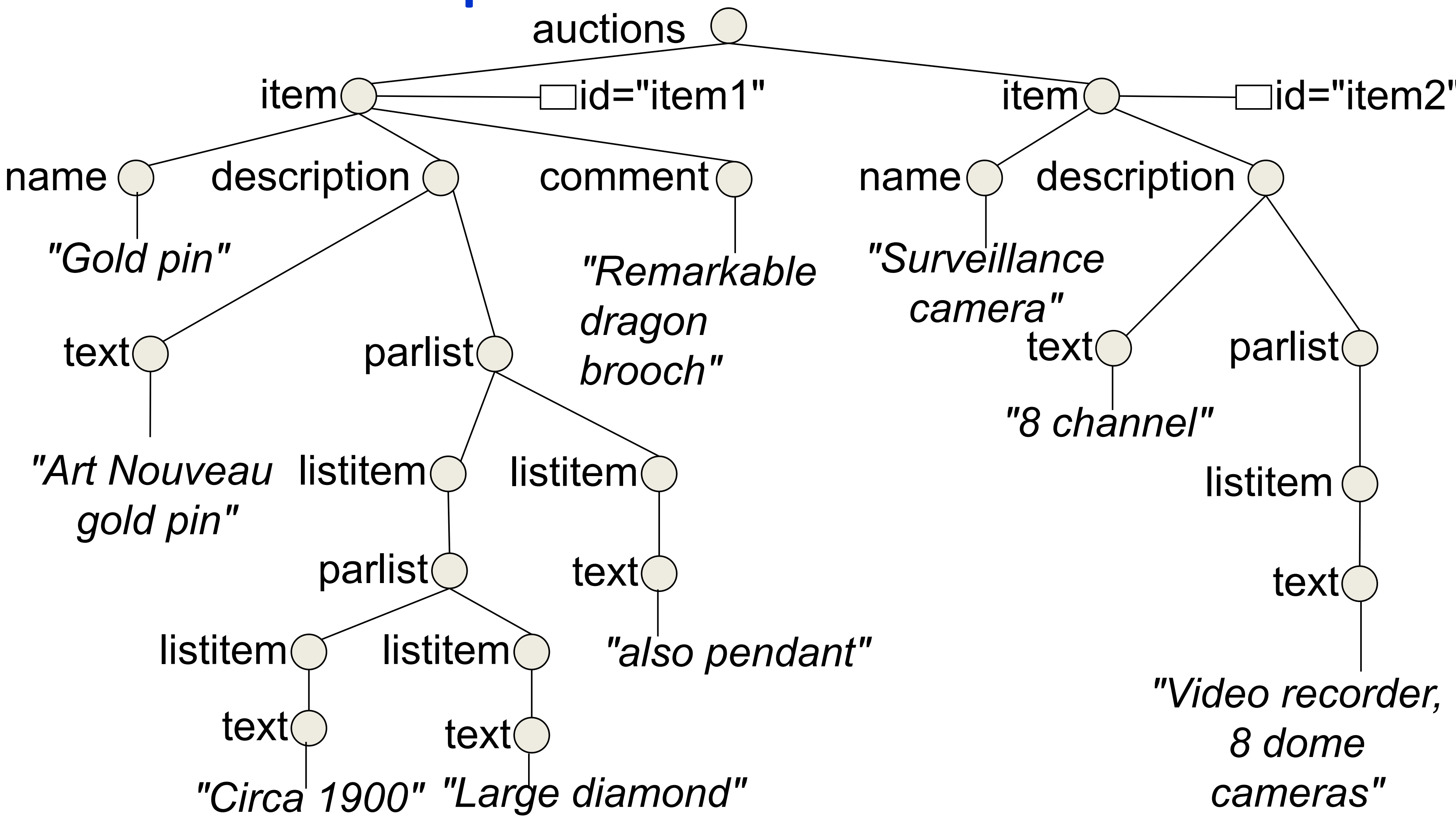
- Time: linear in the size of the document
- Space:
  - Number of stacks = number of query nodes
  - Maximal stack height = maximal depth of matches which are ancestors of one another < document depth
  - If string results are returned, string buffers may be large!

- Store, load, and query.

Requirements for an XML Storage Method

- Completeness
  - Must preserve all information content of the document
- Amenable to efficient processing
  - Navigation queries benefit from fragmentation
  - Reconstruction queries suffer from fragmentation
- Must not require precise schema information

Sample XML document



Nodes and node identity

7 types of nodes: document, **element**, **attribute**, **text**, *namespace, processing instruction, comment [XQDM]*

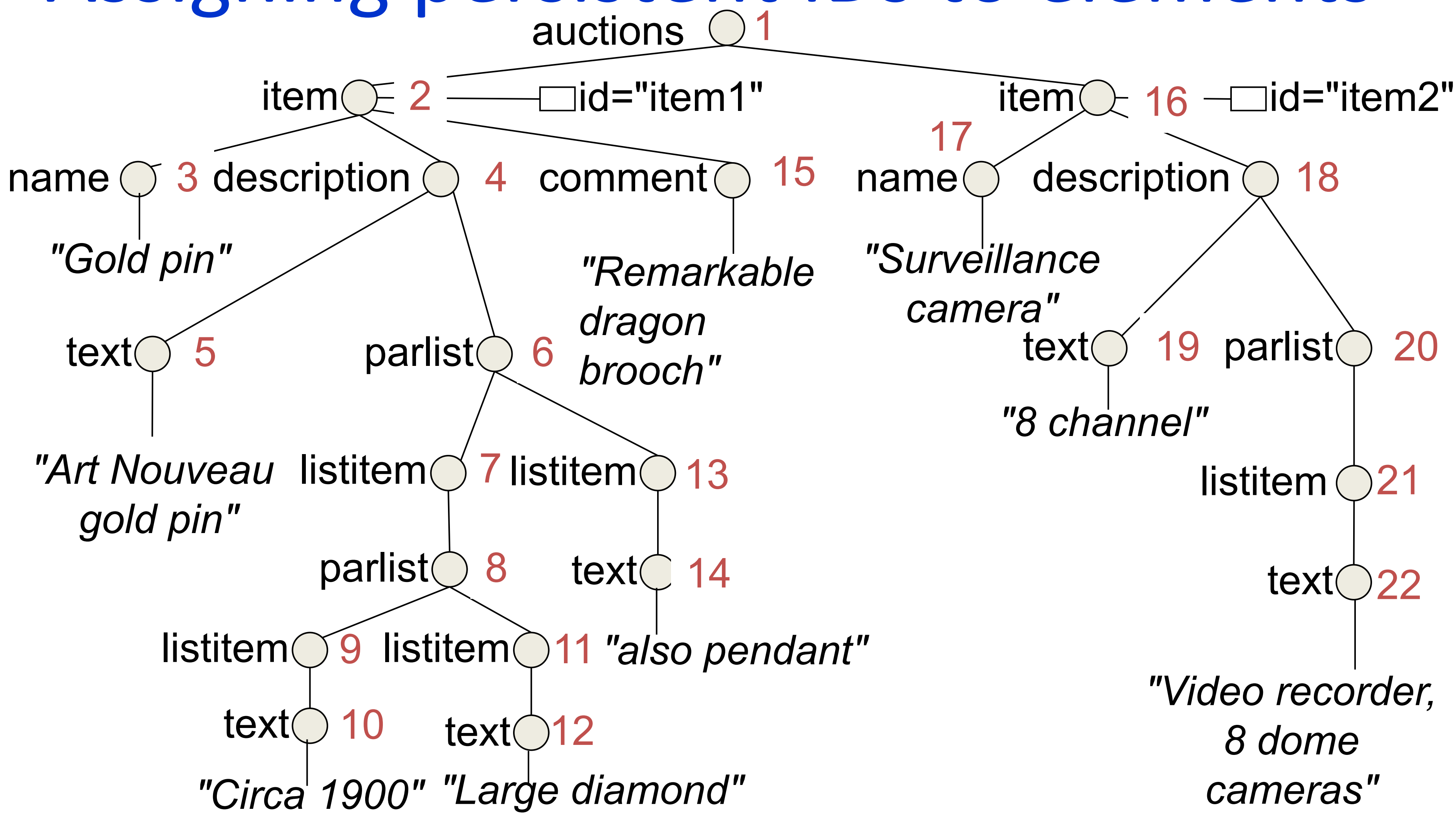
Element, attribute, namespace, PI nodes have a unique identity

(ElemID, attr name) determine attr. value $\Rightarrow$ *key issue is element identity*

– In-memory processing: "the pointer is the ID"

– *Persistent stores: must materialize some persistent IDs (not necessarily for all elements)*

Assigning persistent IDs to elements



Data values

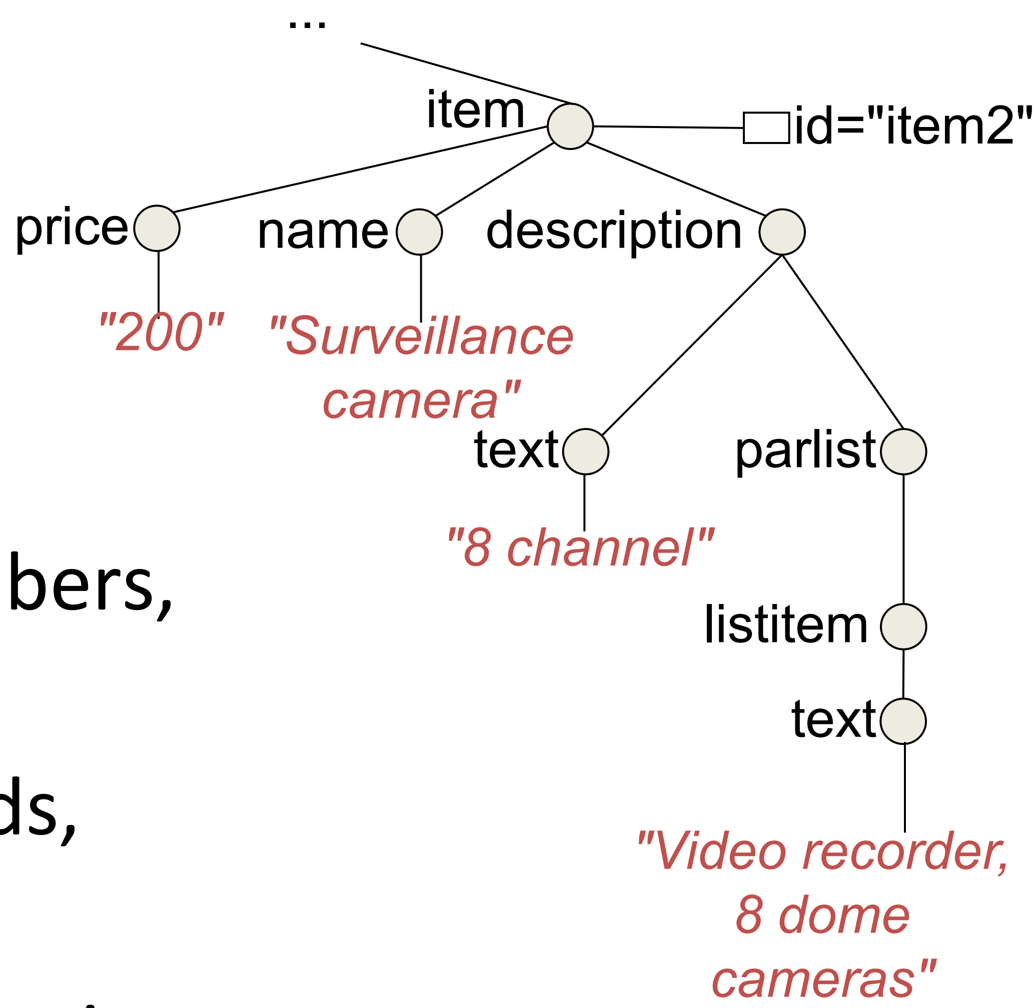
Text nodes

Level 0: bunch of strings

Level 1: strings, numbers, booleans

Level 2: bags of words, numbers, boolean

This is still a simplification

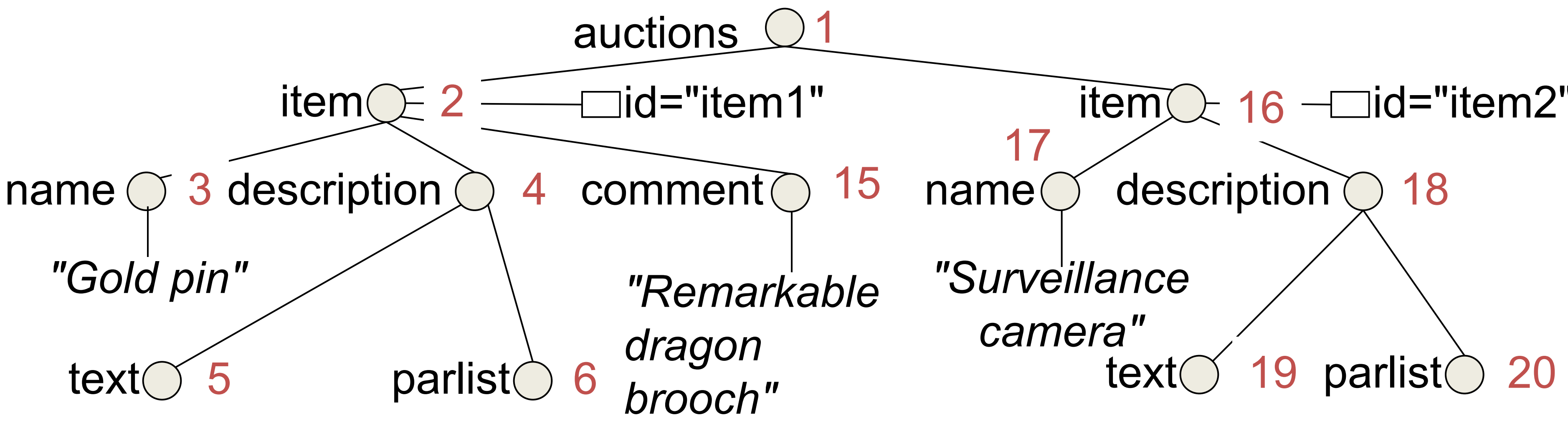


Document structure: relationships among nodes

Level 0: store parent-child relationships

- Given a node, it must be possible to find
  - Its children
  - Its parent
- Parent-child relationships between elements
- "Ownership" relationships between an element and an attribute
- "Text value" relationships between elements and text
- Elements may have several text children

Document structure: relationships among nodes



Element 1 is parent of elements 2 and 16

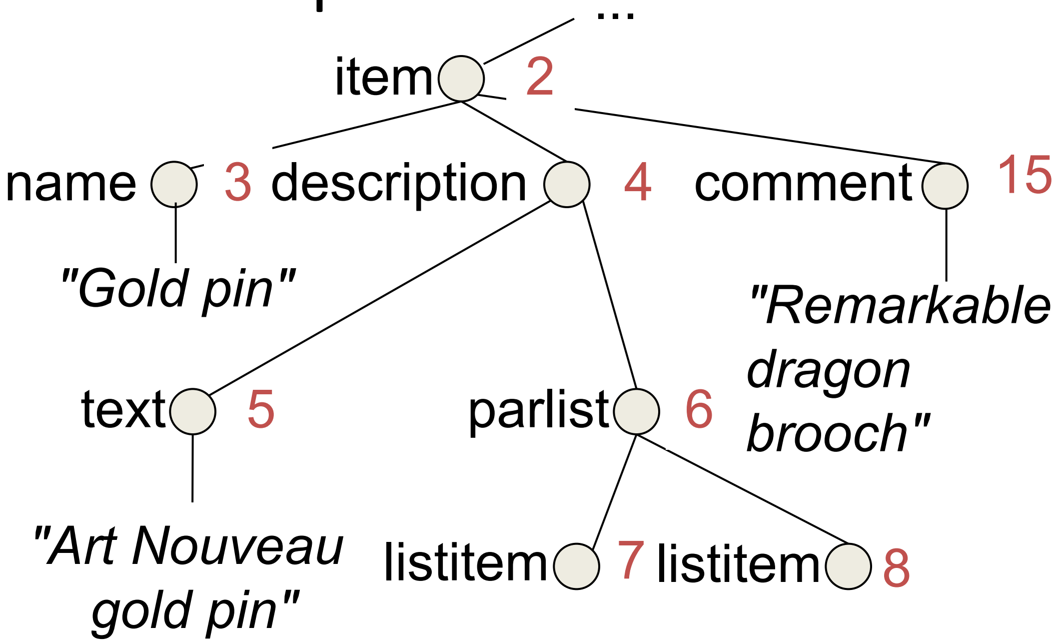
Element 2 has the attribute id="item1"

Element 3 has the text child "Gold pin"

Document structure: order and names

Nodes in an XML document appear in a well-defined total order

It must be possible to retrieve this order



Item name before item description

Storage completeness: summary

Need to store

- Node identity and order
- (Typed) data values
- Document structure = invariants + particular instances

Many invariants is good (regular data)... but they should remain small (to handle easily)

DTDs, XML Schemas are there, but do not express all desirable constraints

Complex constraints require special care for updates

Storage models for XML

They are determined by:

- data model: tuples or trees
- fragmentation strategy = choice of invariant
  - **Choose some property: node name, node path,…**
  - **Group together all tuples/trees that have the same value for the same property**

    E.g. table A contains all *A* elements

    E.g. collection C1 has all trees on path */A/B*
  - **Store each group in a separate structure**

Storage Strategies

- Flat streams:
  - store XML data as is in text files
- Native XML Databases:
  - designed specifically for XML
- Colonial Strategies:
  - re-use existing storage systems

XML Storage: Flat Streams

- Store XML documents as is in text files or CLOBs
- + Fast for storing and retrieving whole documents
- - Query support: limited
  - Navigational queries require parsing
  - Full-text queries require indexes
  - No localized updates

XML Storage: Native Storage

- New databases designed specifically for XML
- + XML documents stored as is
- + Efficient support for XML queries
- – May need to build new systems from the ground up or adapt existing systems
  - *Re-design features for XML (isolation, recovery, etc)*
  - *May have incomplete support for some general data management tasks*

Native Issues: Data Layout

- Requirements
  - Concise representation of documents
  - Efficient support for XML APIs and query languages
  - Ability to update values and structure
- Map trees into physical disk pages
  - Lots of choices: cluster sub-trees vs. cluster similar elements

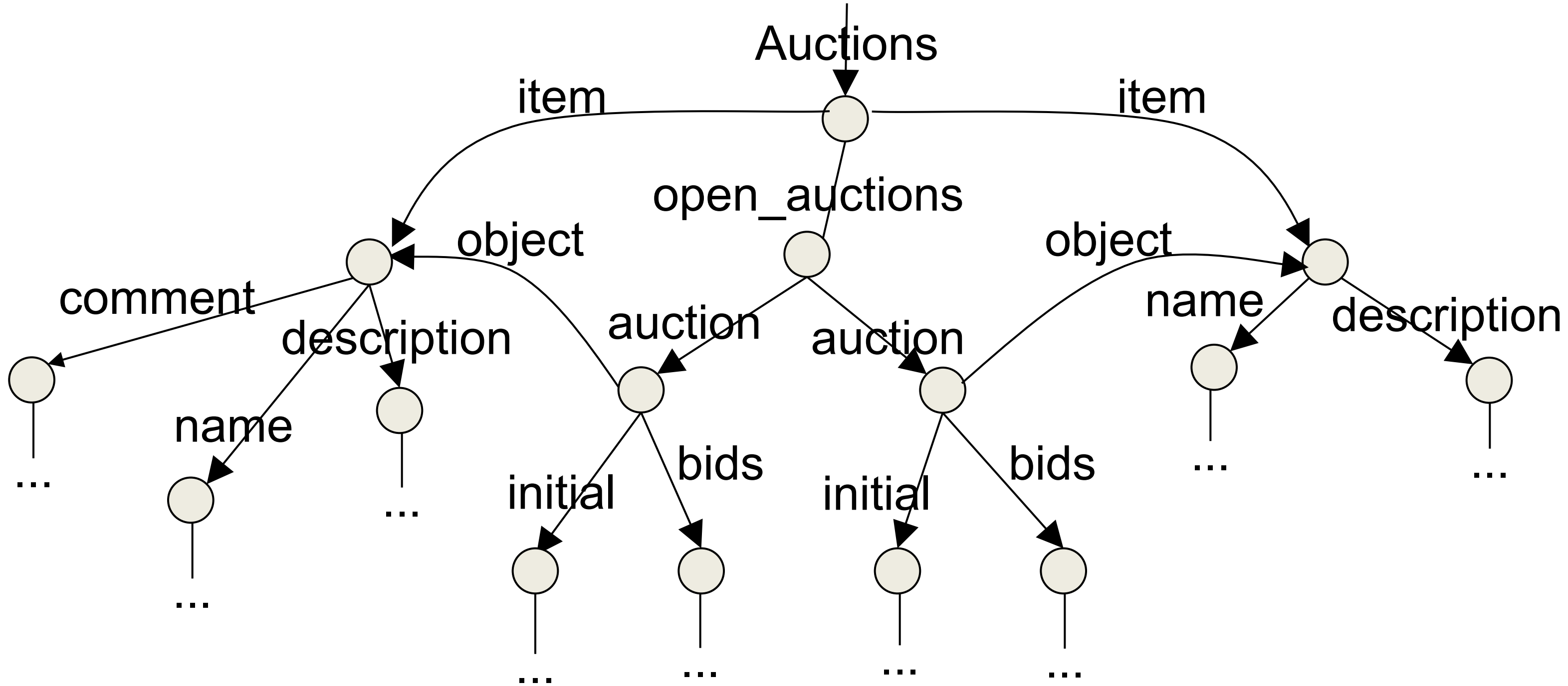
The simplest store: no fragmentation (introduced for OEM [PGW95])

OEM: Object exchange model

Labeled, directed, unordered graph of objects

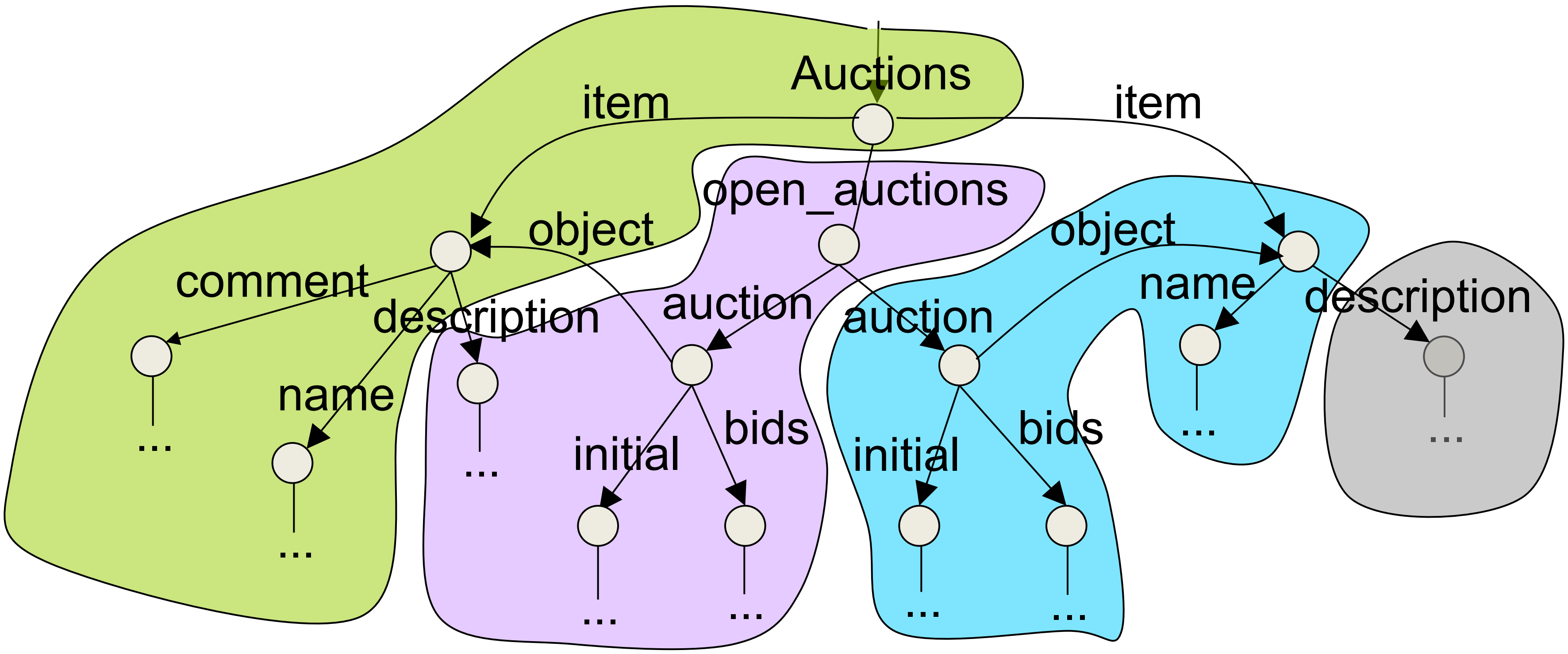
Objects have unique identity

Atomic objects = values (simple atomic types)



Storing OEM objects in LORE [MAG+97]

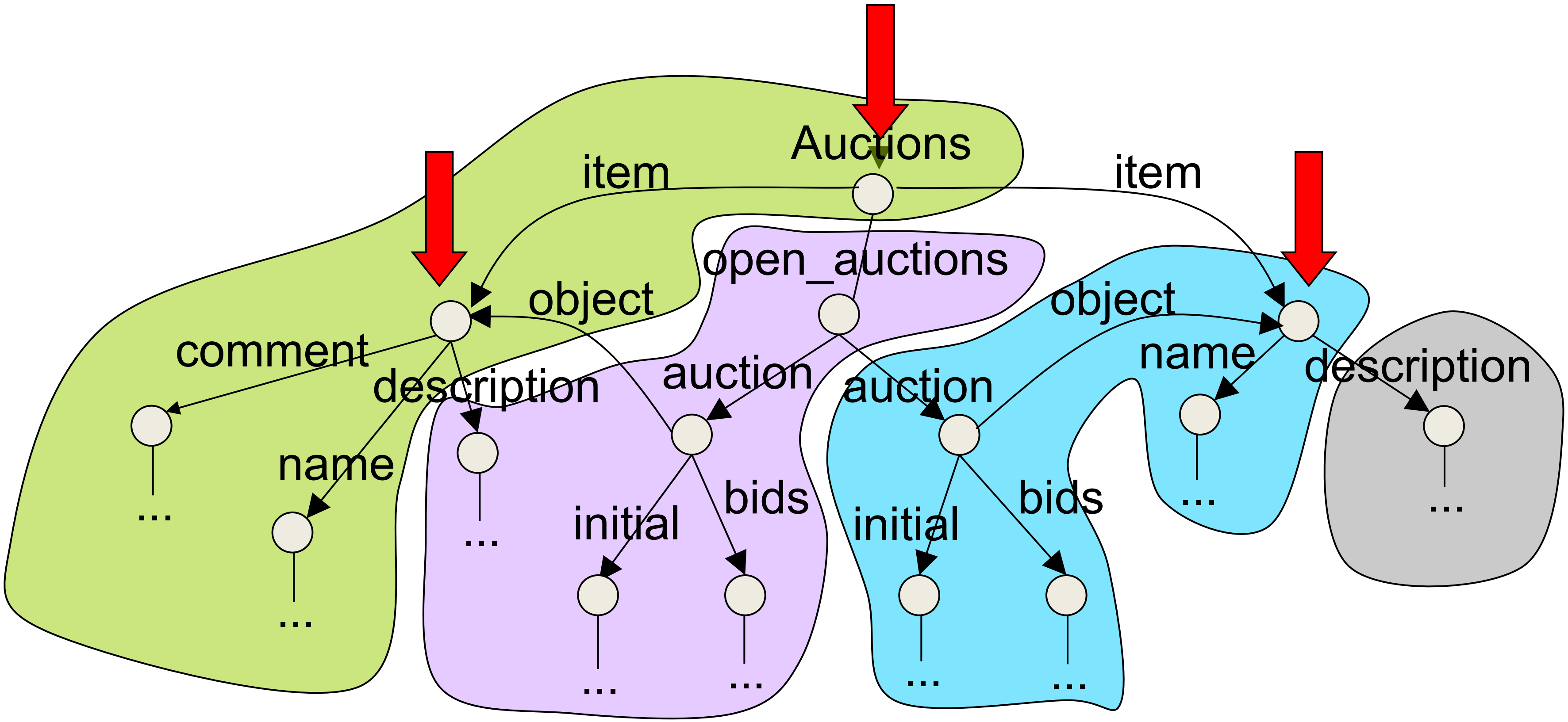
- Objects clustered in pages in depth-first order, including simple value leaves
- Basic physical operator: Scan(obj, path)



Navigation in a persistent graph

Navigation-based, tuple-at-a-time, pointer-chasing

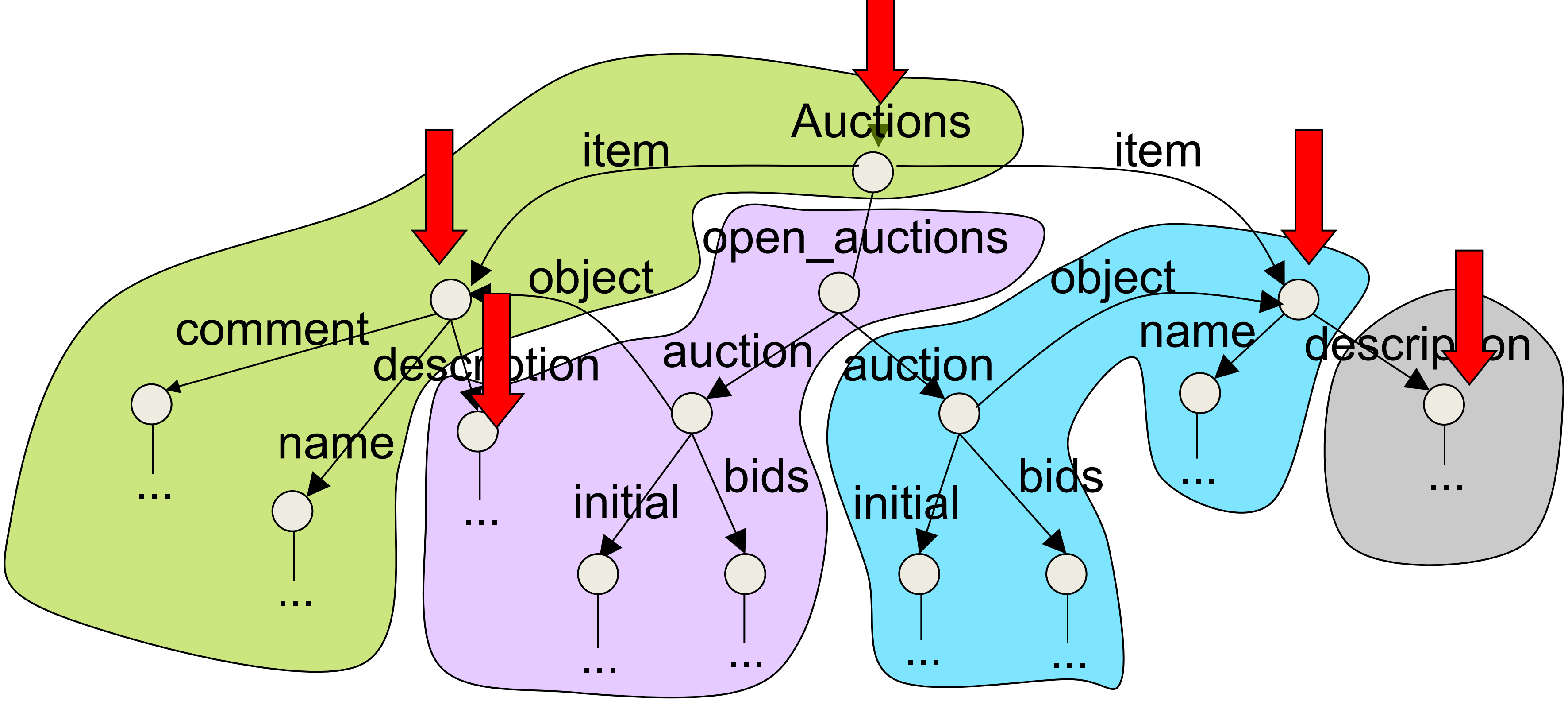
Scan(Auctions, "item"): **2 pages accessed**



Navigation in a persistent graph

Scan(Auctions, "item.description"): **4 pages accessed**

Scan(Auctions, "open_auctions.auction.object"): **3 pages accessed**



Indexing objects in a graph

[MW97,MWA+98,MW99a,MW99b]

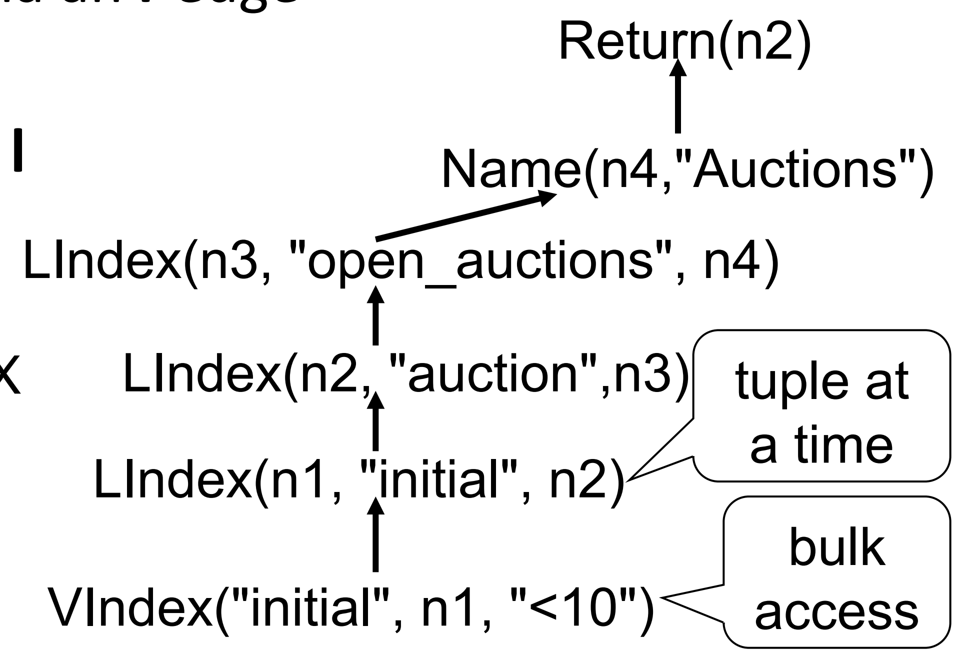
VIndex(**l**, o, **pred**): all objects o with an incoming **l**-edge, satisfying **pred**

LIndex(**o**, **l**, p): all parents of **o** via an **l**-edge

– "Reverse pointers"

BIndex(x, **l**, y): all edges labeled **l**

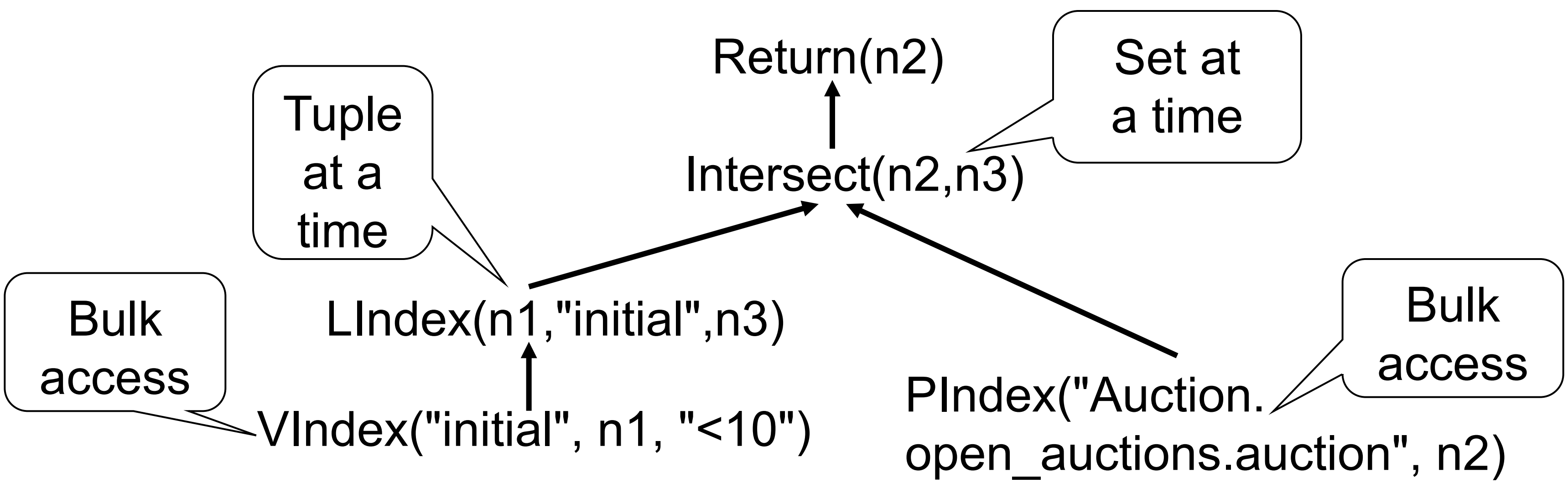
```
select X
from Auction.open_auctions.auction X
where X.initial < 10
```



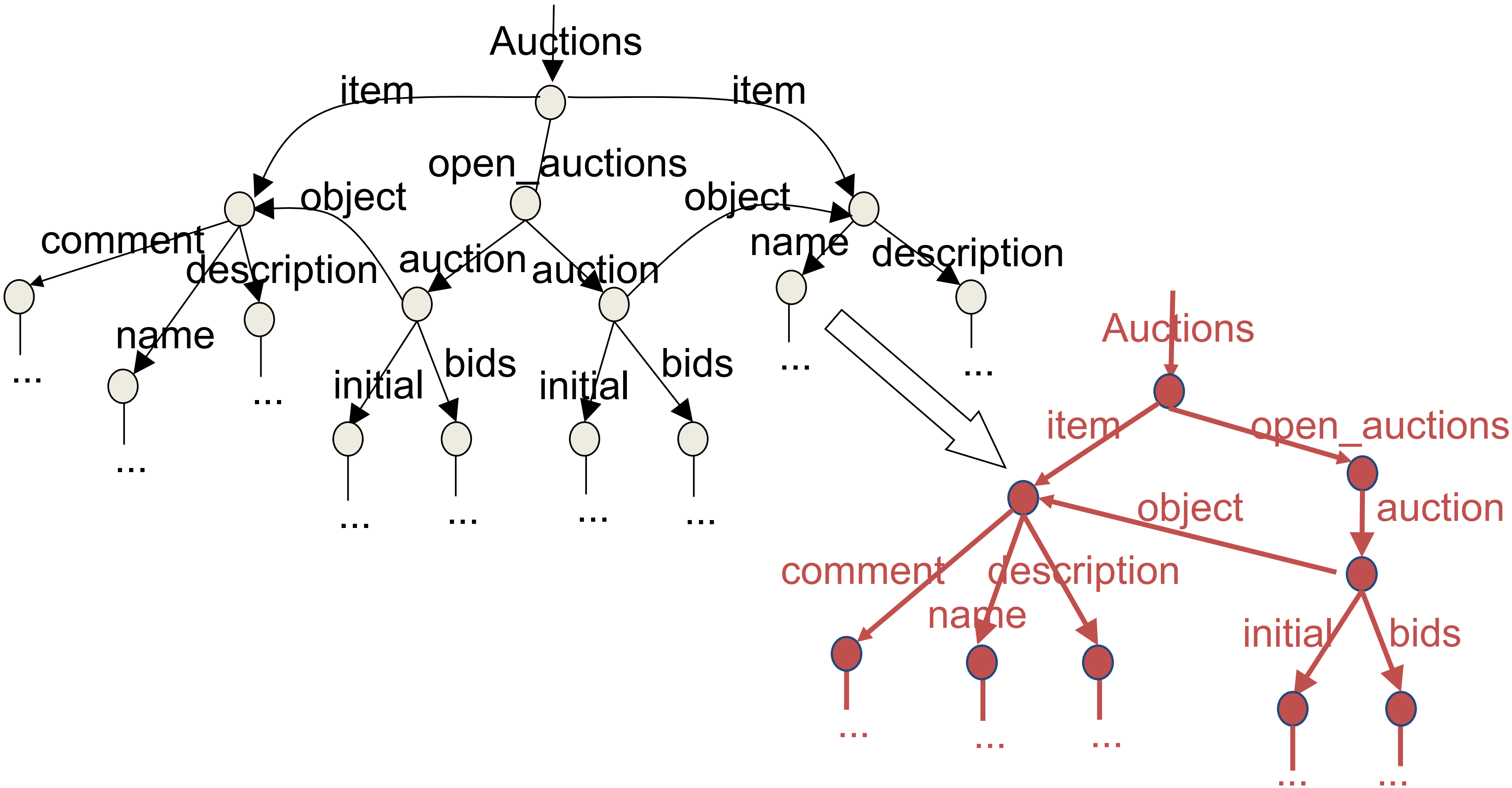
Indexing objects in a graph [MW97]

PIndex(**p**, o): all objects o reachable by the path p

select X
from Auction.open_auctions.auction.initial X
where X.initial < 10



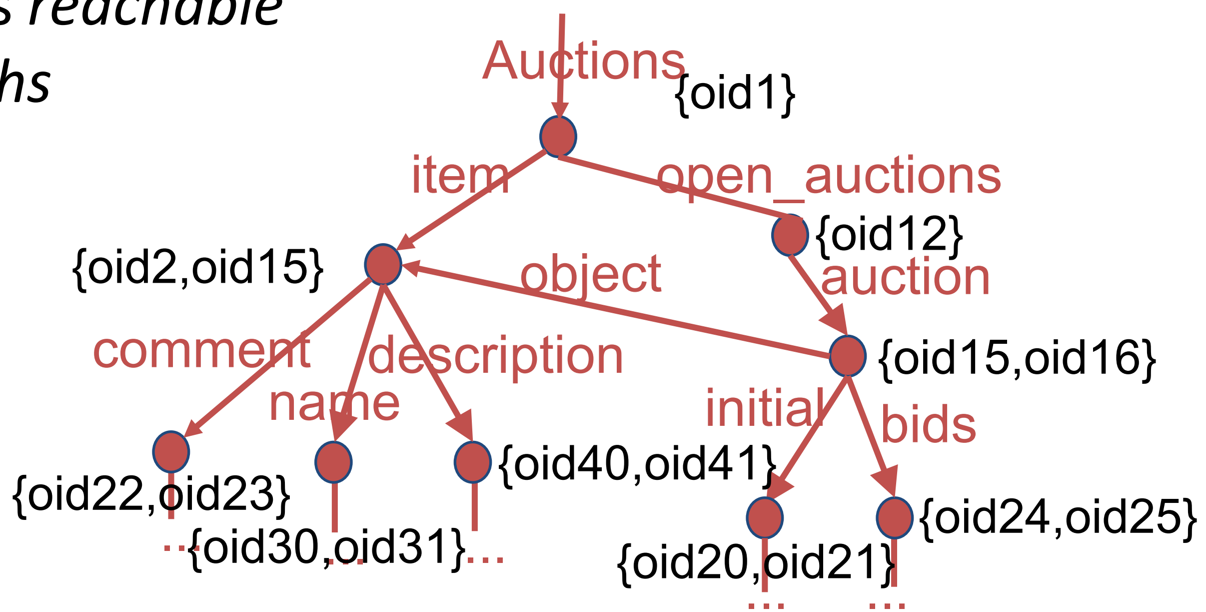
The idea behind path indexes: DataGuides [GW97]



The idea behind path indexes: DataGuides [GW97]

Graph-shaped summaries of graph data

- Invariants extracted from the data ("a posteriori schema")
- Groups all nodes *reachable by the same paths*



More on graph indexing

Graph indexing:

1. **Partition nodes into equivalence classes**
2. **Store the extent of each equivalence class, use it as "pre-cooked" answer to some queries**

Summary: persistent graph / tree storage and indexing

Very simple storage models

Quite simple value indexing [MWA+98]

Multiple graph schema/index structures

- **Identify invariants / regularity / interesting node groups**
- **Use interesting node groups:**
  - **Simplify path queries**
  - **Basis for indexing:**
    - ***Store IDs of all nodes in an interesting group. Access them directly (avoid navigation).***

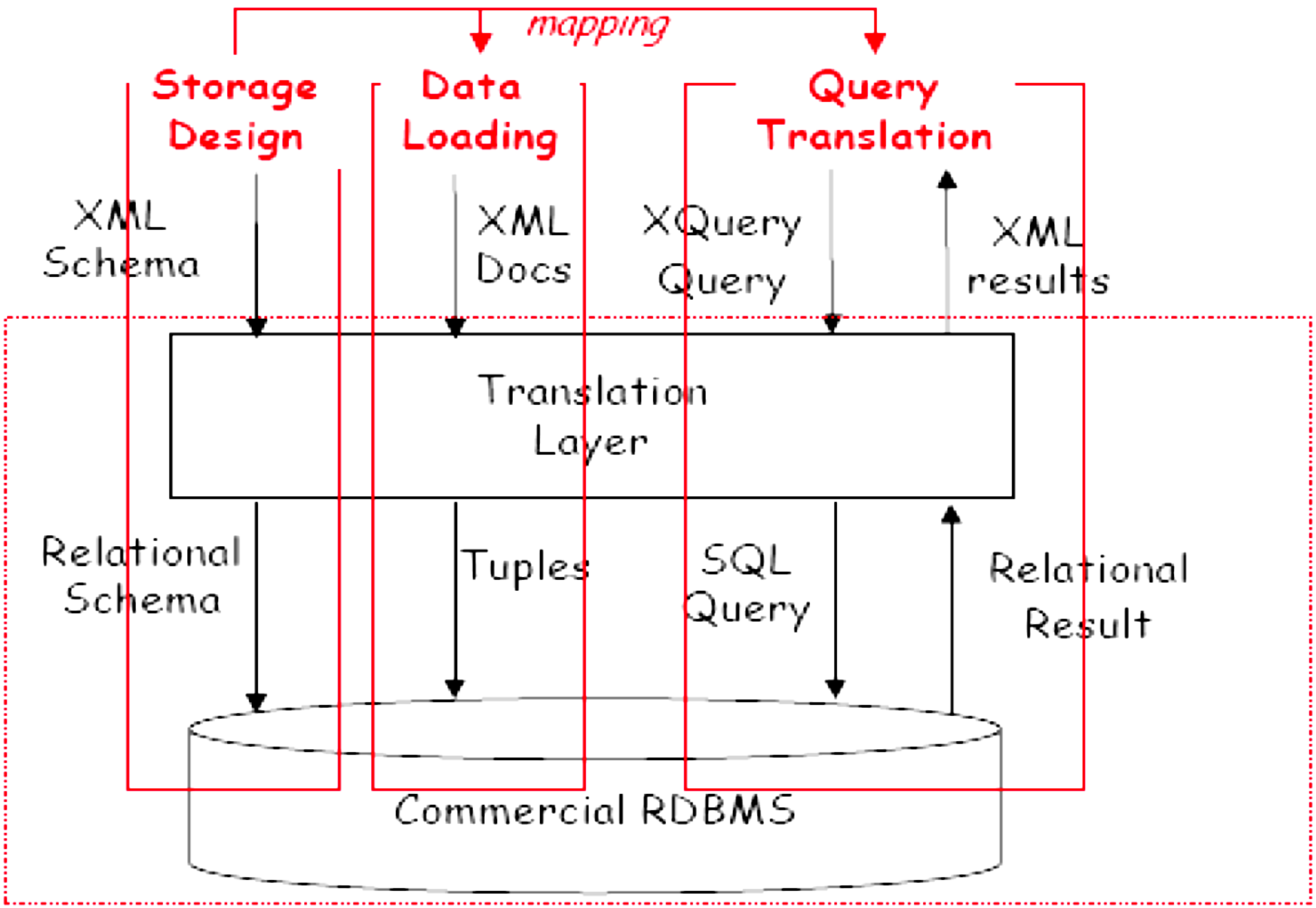
XML Storage: Colonial Storage

- Re-use existing storage systems, map XML document into underlying structures
  – E.g., shred document into flat tables
- + Leverage mature systems
- + Simple integration with legacy data
- – Slow reconstruction of textual representation
- – Query language mismatch
- – Mapping overheads

Colonial Issues

- Storage design: map XML data model onto storage model
  - XML data model → relations, objects
- Data loading: load XML document into mapped structure
  - XML document → tuples, objects
- Query translation: queries over XML document into queries over mapped document
  - XQuery, XPath → SQL, OQL
- Result translation: results into XML

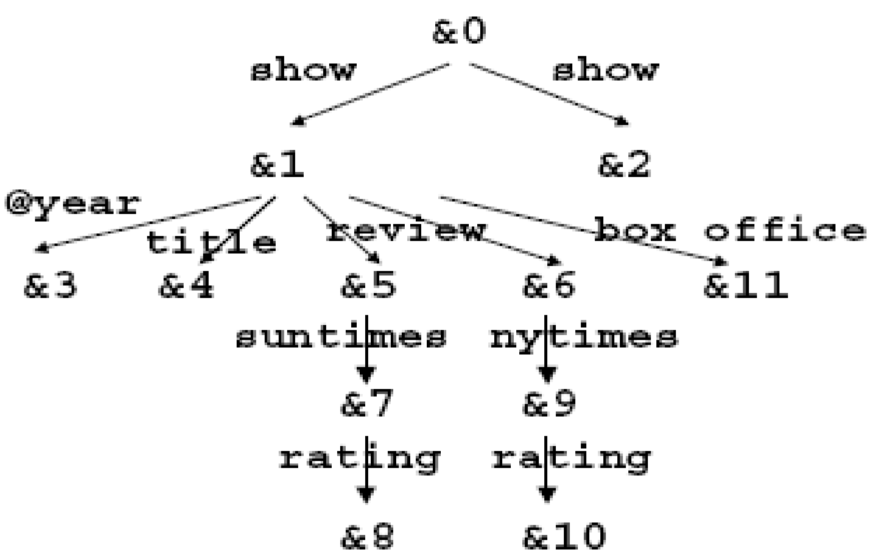
Storing XML in RDBMSs



Relational Storage Design

- There are different classes of mappings
  - Generic: fixed
  - Schema-driven: mapping inferred from DTD or schema
  - Data-driven: mapping inferred from data
  - Cost-based: mapping inferred from schema, query workload and data
  - User-defined: user specifies mapping

Generic Mapping: Edge



Edge Table

source	Child no.	tag	target
&0	1	show	&1
&0	2	show	&2
&1	1	year	&3
&1	2	title	&4
&1	3	review	&5
&1	4	review	&6
&5	1	sun-times	&7

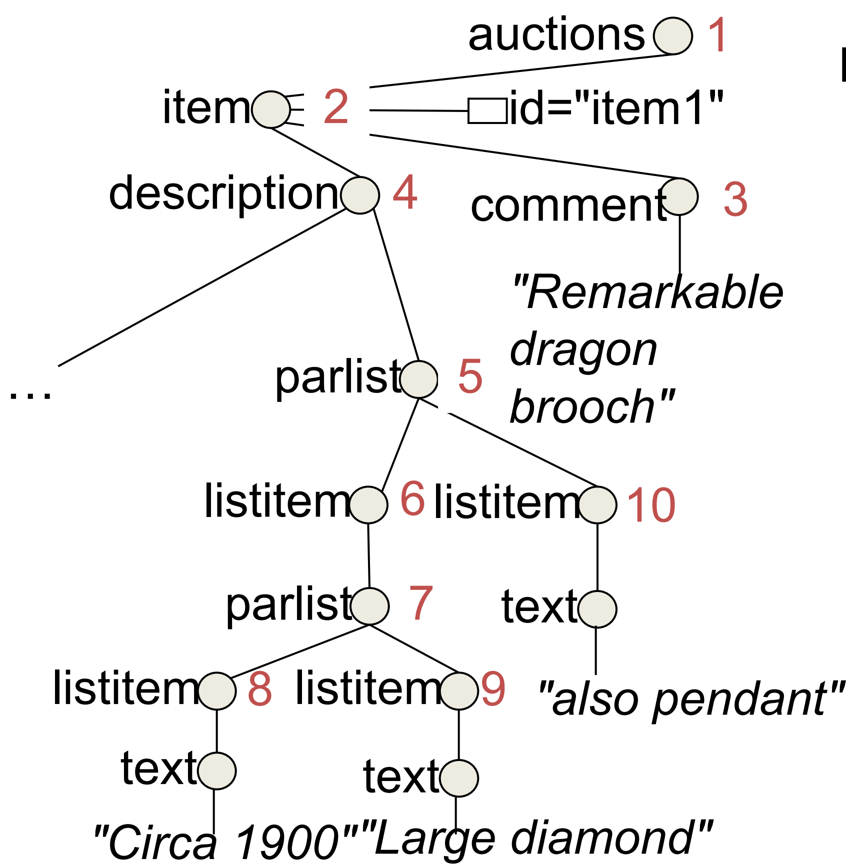
Find titles for all shows

```
SELECT Value.value
FROM Value, Edge as E1, Edge as E2
WHERE E1.tag="show",
E1.target=E2.source,
E2.tag="title",E2.target=Value.node
```

Value Table

node	value
&3	1994
&4	Fugitive, The

Generic Mapping: Edge



Edge(pID, ord, name, target)

pID	ord	name	target
-	1	auctions	1
1	1	item	2
2	1	id	"item1"
2	2	name	"Gold pin"
2	3	comment	"Remark.."
3	1	text	"Art..."
3	2	parlist	5
......	......	......	......

General (no schema or queries used)
No regularity assumed
ID may reflect document order

Index on pID, and (name,target)

Path query processing on "Edge"

Edge(pID, ord, name, target)

pID	ord	name	target
-	1	auctions	1
1	1	item	2
2	1	id	"item1"
2	2	name	"Gold pin"
2	3	comment	"Remark.."
3	1	text	"Art..."
3	2	parlist	5
...	...	...	...

//item

select target from Edge
where name="item"

//commen

e
nt"

//item
[@id="i
description

e e2,

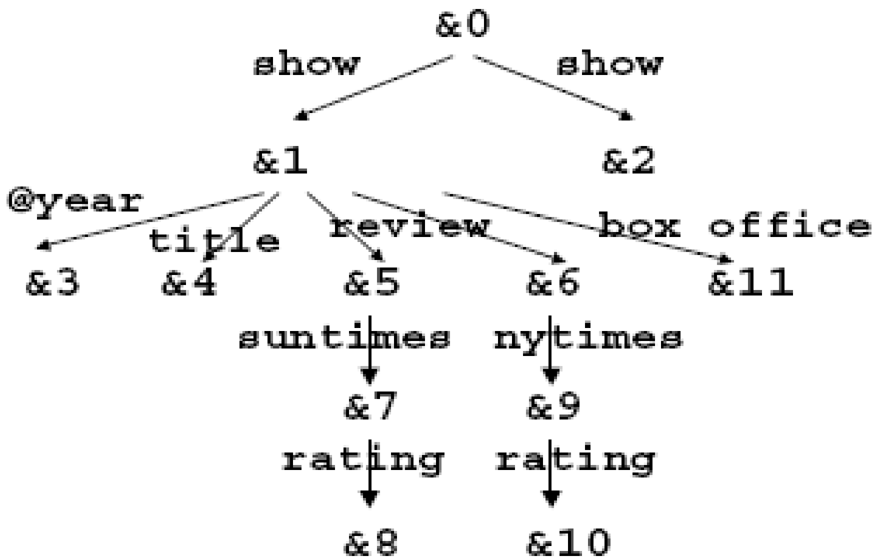
whe .name="item" and
e1.target=e2.pID and
e2.name="id" and
e2.target="item1" and
e1.target=e3.pID and
e3.name="descr"

Actual query answering also requires reconstructing the element...

3-way join on the Edge table

(Index-) join algorithms better than navigation
Still, too much data to read

Generic Mapping: Attribute



Show Table

source	ordinal	target
&0	1	&1
&0	2	&2

Title Table

source	ordinal	target
&1	2	Fugitive, The

Review Table

source	ordinal	target
&1	3	&5
&1	4	&6

Find titles for all shows

```
SELECT Title.target
FROM Title, Show
WHERE Show.target=Title.source
```

Partitioned "Edge"

EdgeAuction(pID,ord,target)

EdgeItem(pID,ord,target)

EdgeID(pID,ord,target)

EdgeDescription(pID,ord,target)

//item[@id="item1"]/description

```
select e3.target
from EdgeItem e1, EdgeID
        e2, EdgeDescription e3
where e1.target=e2.pID and
      e2.target="item1" and
      e1.target=e3.pID and
```

3-way join on (much) smaller tables

Similar to graph index, but *this is the storage*

Interesting groups of nodes: *thos* ***e label***

Store tags in schema, not in data

Some code on the side keeps the mapping between tags and table names

Generic Mappings: Summary

- Ignore regularity in structure
- Canonical relational schema
  - **Edge: store all edges in one table**
  - **Attribute: horizontal partition of Edge relation on element tag**
- Querying:
  - **Requires multi-table joins or self joins for element reconstruction**
  - **Transitive closure for answering descendant queries**

Schema-Driven Mapping

- J. Shanmugasundaram, K. Tufte, G. He, et al., "Relational Databases for Querying XML Documents: Limitations and Opportunities", VLDB 1999
- Idea: Translate DTDs into Relations
  - Element Types -> Tables
  - Attributes -> Columns
  - Nesting(= relationships) -> Tables
  - „Inlining“ reduces fragmentation
- Special treatment for recursive DTDs
- (Adaptations for XML Schema possible)

DTD Normalization

- DTDs can be very complex
  - **<!ELEMENT a ((b|c|e)?,(e?|(f?,(b,b)*))*)>**
- Simplify the DTD before translating a DTD to a relational schema,
- Property of the Simplification: If D_2 is a simplification of D_1, then every document that conforms to D_1 also ***almost*** conforms to D_2
  - almost means that it conforms, if the ordering of sub-elements is ignored

Simplification Rules

$(e_1, e_2)^* \rightarrow e_1^*, e_2^*$

$(e_1, e_2)? \rightarrow e_1?, e_2?$

$(e_1|e_2) \rightarrow e_1?, e_2?$

$e_1^{**} \rightarrow e_1^*$

$e_1^*? \rightarrow e_1^*$

$e_1?^* \rightarrow e_1^*$

$e_1?? \rightarrow e_1?$

$e_1+ \rightarrow e_1^*$

…, a*, …, a*, … → a*, …

…, a*, …, a?, … → a*, …

…, a?, …, a*, … → a*, …

…, a?, …, a?, … → a*, …

…, …a, …, a, … → a*, …

$(b|c|e)?,(e?|f+)$

$(e_1, e_2)^* \rightarrow e_1^*, e_2^*$

$(e_1, e_2)? \rightarrow e_1?, e_2?$

$(e_1|e_2) \rightarrow e_1?, e_2?$

$e_1^{**} \rightarrow e_1^*$

$e_1^*? \rightarrow e_1^*$

$e_1?^* \rightarrow e_1^*$

$e_1?? \rightarrow e_1?$

$e_1+ \rightarrow e_1^*$

…, a*, …, a*, … → a*, …

…, a*, …, a?, … → a*, …

…, a?, …, a*, … → a*, …

…, a?, …, a?, … → a*, …

…, …a, …, a, … → a*, …

$(e_1, e_2)^* \rightarrow e_1^*, e_2^*$

$(e_1, e_2)? \rightarrow e_1?, e_2?$

$(e_1|e_2) \rightarrow e_1?, e_2?$

$e_1^{**} \rightarrow e_1^*$

$e_1^*? \rightarrow e_1^*$

$e_1?^* \rightarrow e_1^*$

$e_1?? \rightarrow e_1?$

$e_1+ \rightarrow e_1^*$

$\ldots, a^*, \ldots, a^*, \ldots \rightarrow a^*, \ldots$

$\ldots, a^*, \ldots, a?, \ldots \rightarrow a^*, \ldots$

$\ldots, a?, \ldots, a^*, \ldots \rightarrow a^*, \ldots$

$\ldots, a?, \ldots, a?, \ldots \rightarrow a^*, \ldots$

$\ldots, \ldots a, \ldots, a, \ldots \rightarrow a^*, \ldots$

(b|c|e)?,(e?|f+)

(b?,c?,e?)?,e??,f+?

$(e_1, e_2)^* \rightarrow e_1^*, e_2^*$

$(e_1, e_2)? \rightarrow e_1?, e_2?$

$(e_1|e_2) \rightarrow e_1?, e_2?$

$e_1^{**} \rightarrow e_1^*$

$e_1^*? \rightarrow e_1^*$

$e_1?^* \rightarrow e_1^*$

$e_1?? \rightarrow e_1?$

$e_1+ \rightarrow e_1^*$

..., a*, ..., a*, ... → a*, ...

..., a*, ..., a?, ... → a*, ...

..., a?, ..., a*, ... → a*, ...

..., a?, ..., a?, ... → a*, ...

..., ...a, ..., a, ... → a*, ...

(b|c|e)?,(e?|f+)

(b?,c?,e?)?,e??,f+?

b??,c??,e??,e??,f+?

$(e_1, e_2)^* \rightarrow e_1^*, e_2^*$

$(e_1, e_2)? \rightarrow e_1?, e_2?$

$(e_1|e_2) \rightarrow e_1?, e_2?$

$e_1^{**} \rightarrow e_1^*$

$e_1^*? \rightarrow e_1^*$

$e_1?^* \rightarrow e_1^*$

$e_1?? \rightarrow e_1?$

$e_1+ \rightarrow e_1^*$

…, a*, …, a*, … → a*, …

…, a*, …, a?, … → a*, …

…, a?, …, a*, … → a*, …

…, a?, …, a?, … → a*, …

…, …a, …, a, … → a*, …

(b|c|e)?,(e?|f+)

⇓

(b?,c?,e?)?,e??,f+?

⇓

b??,c??,e??,e??,f+?

⇓

b??,c??,e??,e??,f*?

$(e_1, e_2)^* \rightarrow e_1^*, e_2^*$

$(e_1, e_2)? \rightarrow e_1?, e_2?$

$(e_1|e_2) \rightarrow e_1?, e_2?$

$e_1^{**} \rightarrow e_1^*$

$e_1^*? \rightarrow e_1^*$

$e_1?^* \rightarrow e_1^*$

$e_1?? \rightarrow e_1?$

$e_1+ \rightarrow e_1^*$

…, a*, …, a*, … → a*, …

…, a*, …, a?, … → a*, …

…, a?, …, a*, … → a*, …

…, a?, …, a?, … → a*, …

…, …a, …, a, … → a*, …

(b|c|e)?,(e?|f+)

⇓

(b?,c?,e?)?,e??,f+?

⇓

b??,c??,e??,e??,f+?

⇓

b??,c??,e??,e??,f*?

⇓

b?,c?,e?,e?,f*

$(e_1, e_2)^* \rightarrow e_1^*, e_2^*$

$(e_1, e_2)? \rightarrow e_1?, e_2?$

$(e_1|e_2) \rightarrow e_1?, e_2?$

$e_1^{**} \rightarrow e_1^*$

$e_1^*? \rightarrow e_1^*$

$e_1?^* \rightarrow e_1^*$

$e_1?? \rightarrow e_1?$

$e_1+ \rightarrow e_1^*$

…, a*, …, a*, … → a*, …

…, a*, …, a?, … → a*, …

…, a?, …, a*, … → a*, …

…, a?, …, a?, … → a*, …

…, …a, …, a, … → a*, …

(b|c|e)?,(e?|f+)

↓

(b?,c?,e?)?,e??,f+?

↓

b??,c??,e??,e??,f+?

↓

b??,c??,e??,e??,f*?

↓

b?,c?,e?,e?,f*

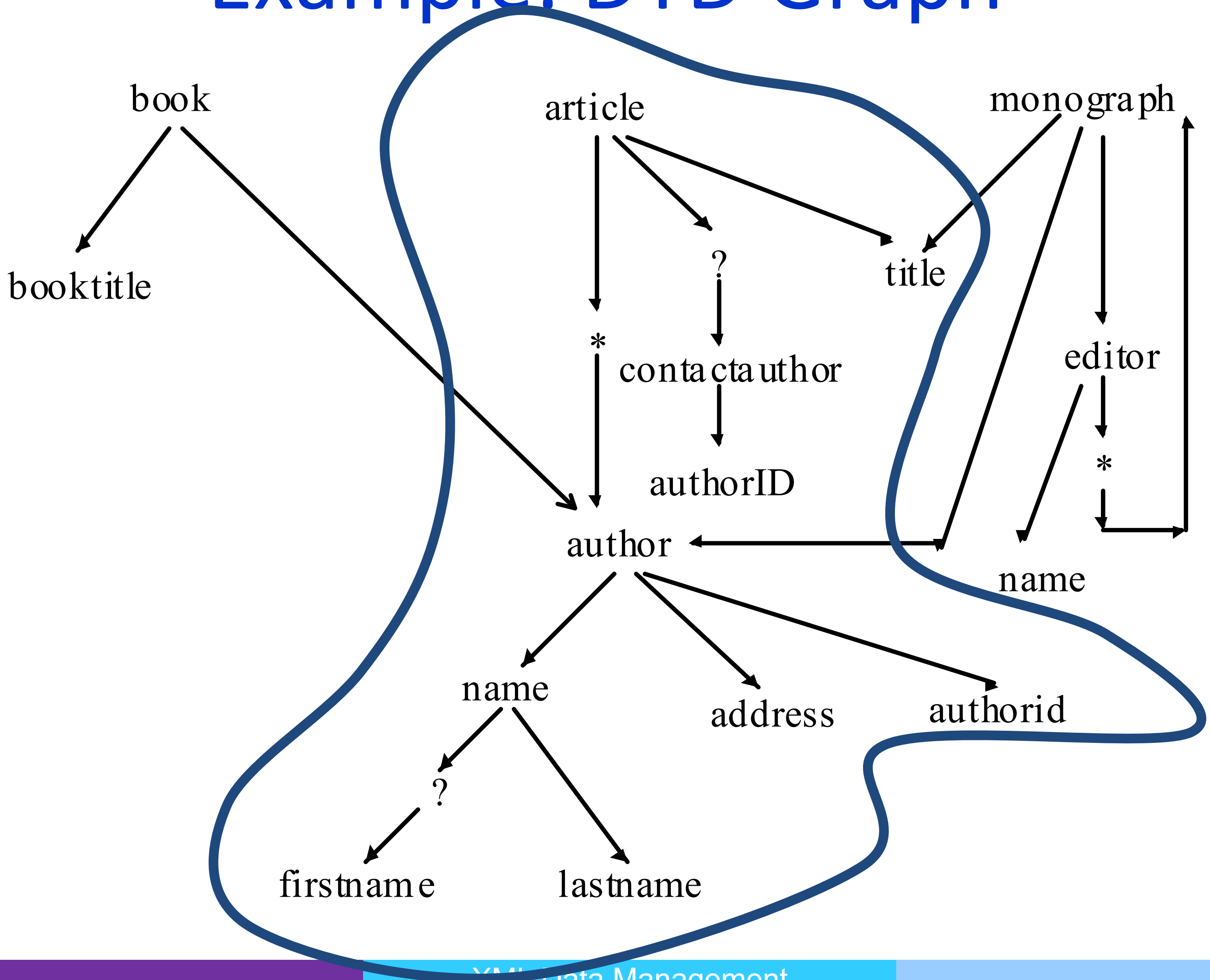
↓

b?,c?,e*,f*

DTD Graphs

- In order to describe a technique for converting a DTD to a schema it is convenient to describe DTDs (or rather simplified DTDs) as graphs
- Its nodes are elements, attributes and operators in the DTD
- Each element appears exactly once in the graph
- Attributes and operators appear as many times as they are in the DTD
- Cycles indicate recursion

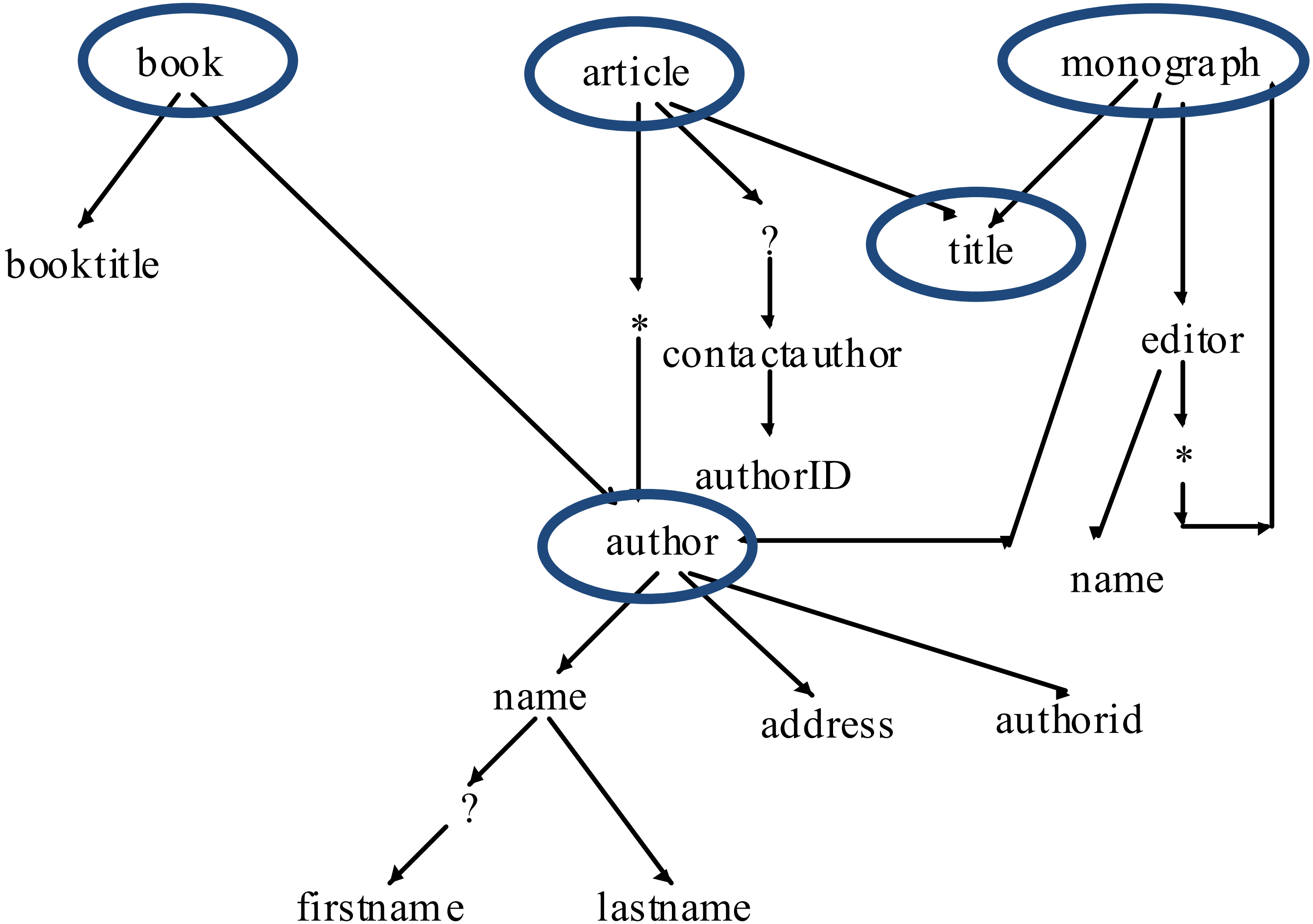
Example: DTD Graph



Creating the Schema: Shared-Inline Technique

- When creating the schema for a DTD, we create a relation for:
  - each element with in-degree greater than 1
  - each element with in-degree 0
  - each element below a *
  - one element from each set of mutually recursive elements, having in-degree 1
- All other elements are "inlined" into their parent's relation (i.e., added into their parents relations)

Relations for which elements?



book (bookID: integer, book.booktitle : string)

article (articleID: integer, article.contactauthor.authorid: string)

monograph (monographID: integer,

monograph.parentID: integer,

monograph.parentCODE: integer,

monograph.editor.name: string)

title (titleID: integer, title: string ,

title.parentID: integer, title.parentCODE: integer)

author (author.parentID: integer, author.parentCODE: integer,

authorID: integer, author.authorid: string

author.address: string, author.name.firstname: string,

author.name.lastname: string,)

What are these for?

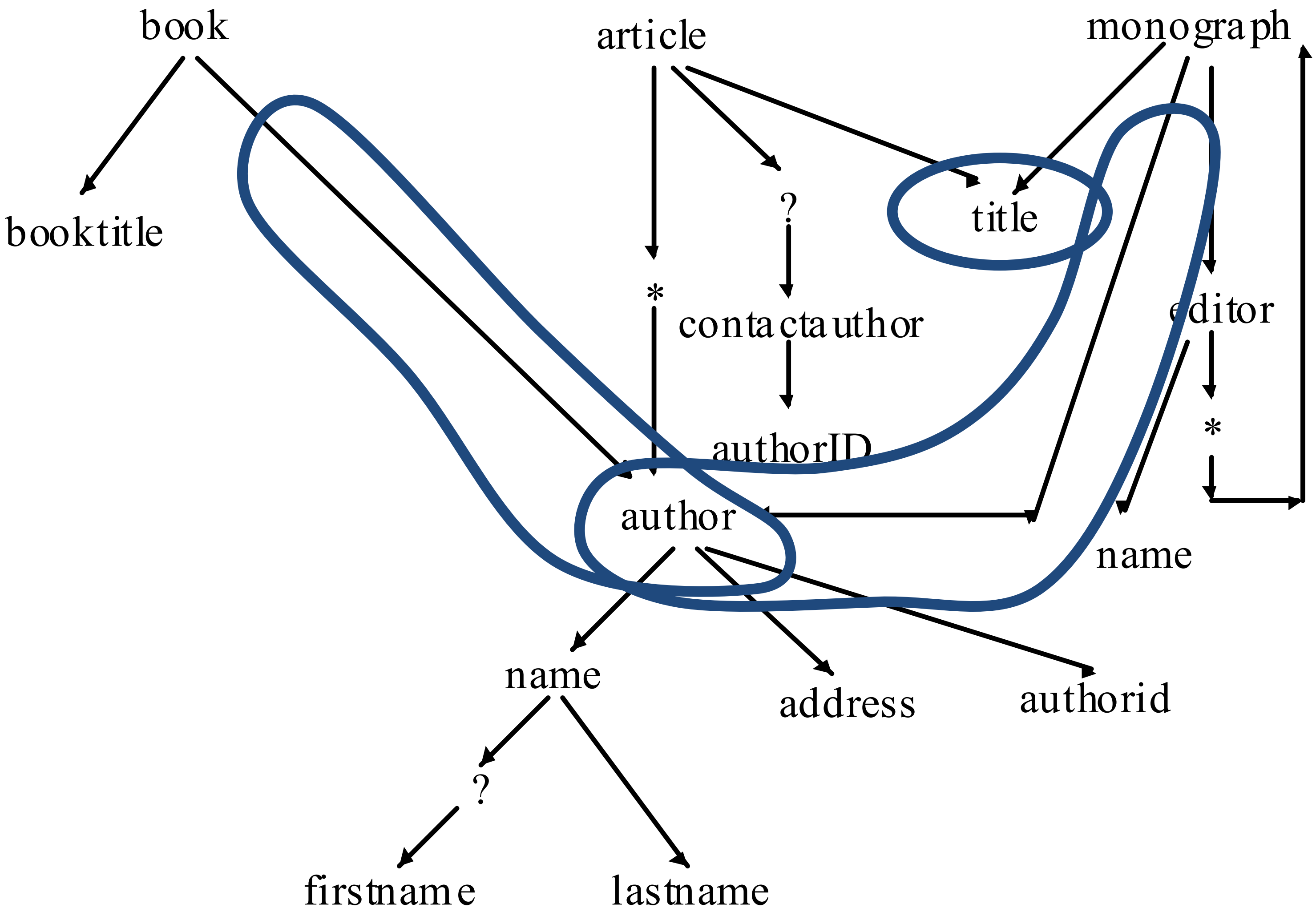
Advantages/Disadvantages

- Advantages:
  - Reduces number of joins for queries like “get the first and last names of an author”
  - Efficient for queries such as “list all authors with name Jack”
- Disadvantages:
  - Extra join needed for “Article with a given title name”

Hybrid-Inline Technique

- Same as Shared, except also inline elements with in-degree greater than one for the places in which they are not recursive or reached through a * node

What, in addition, will be inline?



book (bookID: integer, book.booktitle : string,

author.name.firstname: string, author.name.lastname: string,

author.address: string, author.authorid: string)

article (articleID: integer, article.contactauthor.authorid: string,

article.title: string)

monograph (monographID: integer, monograph.parentID: integer,

monograph.parentCODE: integer, monograph.title: string,

author.name.firstname: string, author.name.lastname: string,

author.address: string, author.authorid: string,

monograph.editor.name: string,)

author (authorID: integer, author.parentID: integer,

author.parentCODE: integer, author.name.firstname: string,

author.name.lastname: string, author.address: string,

author.authorid: string)

Why do we still have an author relation?

Advantages/Disadvantages

- Advantages:
  - Reduces joins through shared elements (that are not set or recursive elements)
  - Reduces joins for queries like “get first and last names of a book author” (like Shared)
- Disadvantages:
  - Requires more SQL sub-queries to retrieve all authors with first name Jack (i.e., unions)
- Tradeoff between reducing number of queries and reducing number of joins
  - Shared and Hybrid target query- and join-reduction, respectively

Schema-Driven: Summary

- Use DTD/XML Schema to decompose document
- Shared/Hybrid
  - Rule of thumb: inline as much as possible to minimize number of joins
  - Shared: do not inline if shared, set-valued, recursive
  - Hybrid: also inline if shared but not set-valued or recursive
- Querying:
  - + Fast lookup & reconstruction of inlined elements
  - - Reconstruction may require multi-table joins and unions

Preserve Constraints

- D. Lee, W.W. Chu, "Constraints-Preserving Transformation from XML Document Type Definition to Relational Schema", ER 2000.
- DTDs encapsulate certain types of constraints
  - Domain: <!ATTLIST author gender (male|female) >
  - Cardinality: <!ELEMENT article (title, author+, ref*, price?)>
  - Inclusion: <!ATTLIST contact aid IDREF #REQUIRED>
- Hybrid-inline approach can be modified to preserve these constraints, and to generate SQL constraint statements: "create domain", "NOT NULL", "UNIQUE", id and foreign key.
  - The key is assumed to be the attribute of type ID, whenever it exists.

Data-Driven: STORED

- Schemaless data
- Analyze data, try to infer schema graph: “mine” data for common (regular) patterns with high-support
- Example:
  - Discover from IMDB data that every show has year and title
  - Create a table for show that contains year and title
  - Use generic mapping for irregular parts of data
- Querying: use derived mapping definition to automatically translate queries

More Mappings…

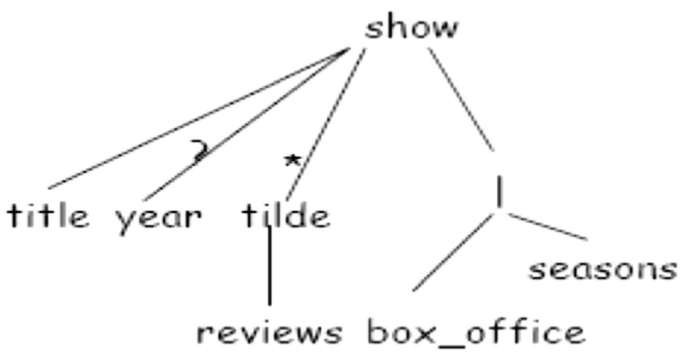


TABLE Show
(show_id INT,
title STRING,
year INT,
box_office INT,
seasons INT)

TABLE Review
(review_id INT,
tilde STRING,
review STRING,
parent_Show INT)

(I) Inline as many elements as possible

TABLE Show
(show_id INT,
title STRING,
year INT,
box_office INT,
seasons INT)

TABLE NYTReview
(review_id INT,
review STRING,
parent_Show INT)

TABLE Review
(review_id INT,
tilde STRING,
review STRING,
parent_Show INT)

(II)Partition reviews table-one for NYT,one for rest

TABLE Show1
(show1_id INT,
title STRING,
year INT,
box_office INT)

TABLE Show2
(show2_id INT,
title STRING,
year INT,
seasons INT)

TABLE Review
(review_id INT,
tilde STRING,
review STRING,
parent_Show INT)

(III)Split Show table into TV and Movies

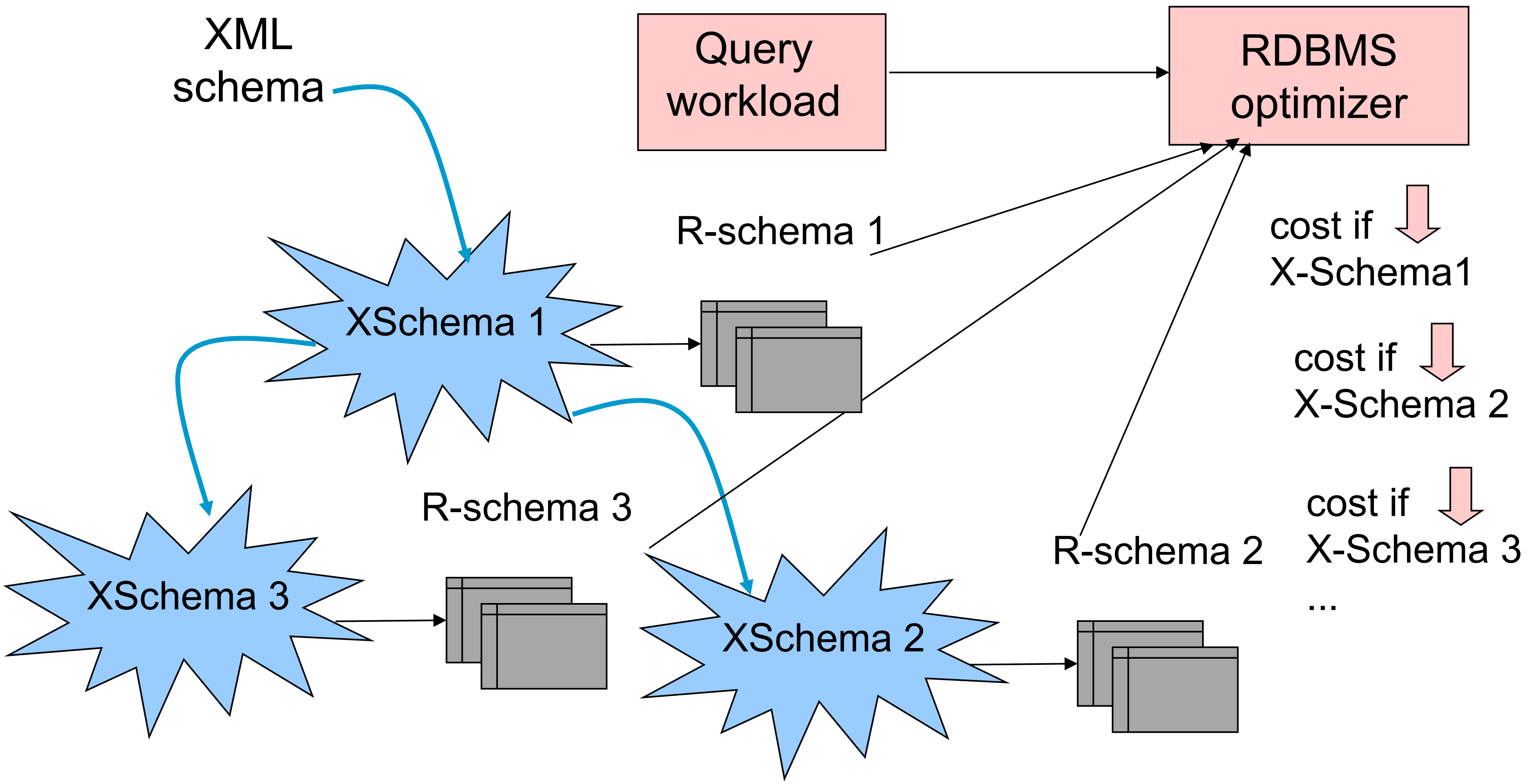
There are many alternative mappings!

- Performance depends on data, schema and query workload
- A fixed mapping is unlikely to be the best for all applications

Cost-Based: LegoDB

- Application-driven shredding
- Automatically generates and explores a space of possible mappings
  - Uses information from *schema*, *data statistics* and *query workload*
- Uses a standard relational optimizer to evaluate cost of mappings
  - Selects the mapping which has the lowest cost for a given application
- XQuery is automatically translated at runtime

Cost-Based


XML schema
Query workload
RDBMS optimizer
R-schema 1
XSchema 1
cost if X-Schema1
cost if X-Schema 2
R-schema 3
XSchema 3
XSchema 2
R-schema 2
cost if X-Schema 3
...

Schema Transformation Rules

- Inlining / Outlining
  - type A=[b [Integer], C, d*], type C=e [String] equivalent to type A=[b [Integer], e [String], d*]
  - *Inlining useful if C is always queried through ancestor A*
- Union Factorization / Distribution
  - (a, (b|c)) equivalent to (a, b) | (a, c)
  - a[t1|t2] equivalent to a[t1] | a[t2]
  - *Useful to separate if a[t1] often queried together, a[t2] rarely or never queried together*

Schema Transformation Rules

- Repetitions merge / split
  - a+ equivalent to (a, a*)
  - *If the first <a> is isolated, it can be inlined with parent*
- Wildcard rewritings
  - A[b ~[String]*] equivalent to a[b[(c|d)*]], where c=tag1[String] and d=(~! tag1)[String]
  - *If a/b/tag1 often queried, a/b/other never queried, separate them.*

Cost-Based: Summary

- ~ Materialized view selection for a dataset and workload
- Optimizer estimates can be wrong, but the optimizer will make *the same mistake* when choosing the best plan
- Search space explored:
  - Node labels factorized in the schema
  - Schema management module needed to identify pertinent relations
  - Various points in the search space vary the number of *unions* and *joins* required by a query

User-Defined Mappings

- Supported by most commercial RDBMS
  - User specifies how to map elements to tables
- Flexible mapping but…
- There are drawbacks:
  - Requires knowledge of XML and relational technology
  - Many different mappings
    - Hard to choose the best for an application
  - Data changes → need to update mapping

User-Defined Mappings

Express (relational) sto[…] (algebrai[…]
over the XML docu[…]

– Relation = materiali[…] XM[…]

Finding useful tables re[…]se

R(y,z) :- Auctions.item x, x.@id.text() y, x.pri[…]e

S(u,v) :- Auctions.item t, t.@id.text() u, t.d[…]t() v

for $x in //item

return <res> {$x/price}, {$x/description} </res>

Does each item have exactly one price ?

Is Auc[…]item the sam[…]s //item ?

Does each item have exactly one description ?

Is @id a key for item ?

Not so fast.

select z, v
from R, S
where R.y=S.u

?

User-Defined Mappings

Express (relational) storage by custom expressions over the XML document

- Relation = mate ... ver the X

Finding useful tables ... based qu

Does each item have exactly one price ?

Does each item have exactly one description ?

XPath containment

Is Auction/item the same as //item ?

Functional dependency

Is @id a key for item ?

Cardinality constraints

Query containment/rewriting under constraints

Techniques based on the chase

User-Defined Mappings

- Express (relational) storage by custom expressions over the XML
- Must check storage completeness
- Most generic; potential for good performance (materialized views!)
- Can also express non-relational storage models
- *Rewriting is complex.*

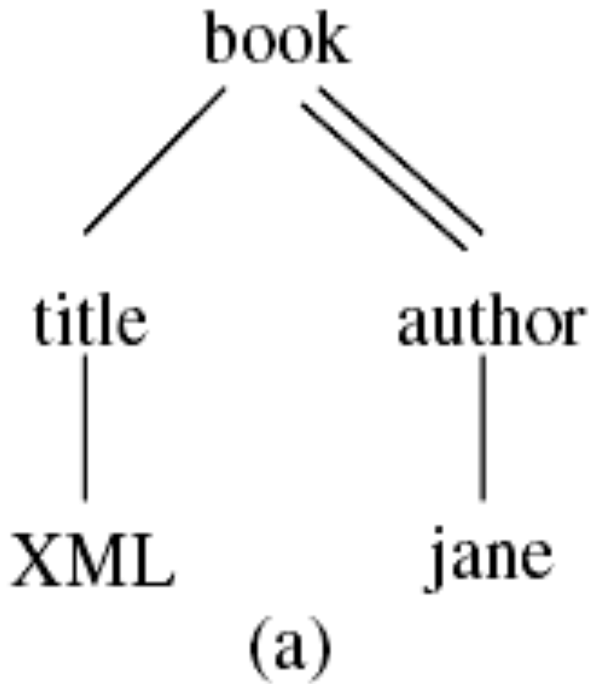
- Poor man's solution: cut in flexibility (and performance)
- Less freedom in the mappings
  - Assign IDs to all elements
  - Map each element to a table...

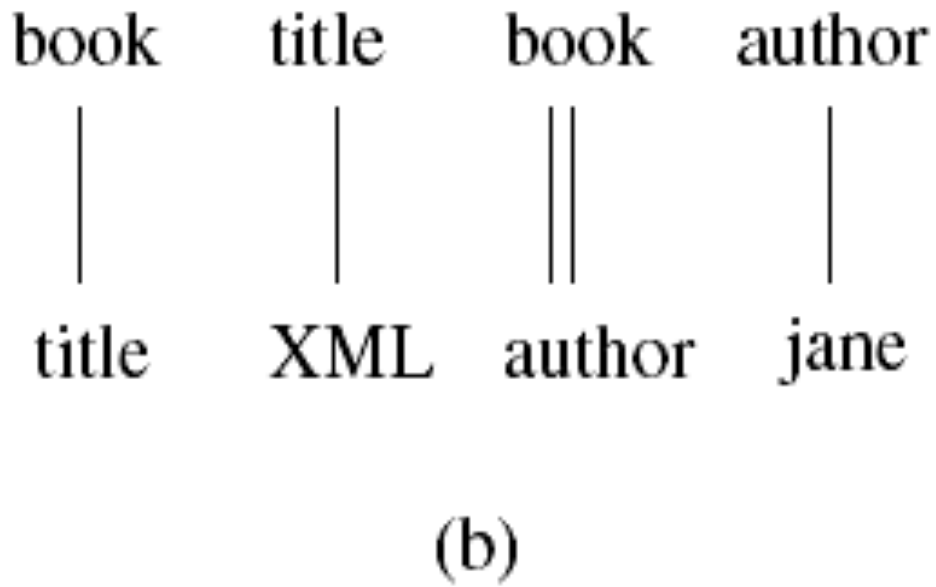
Summary for Relational Storage for XML

- Relations *alone* only go that far
- Many solutions around *S-P-J materialized view selection over partitioned Edge table*
- Flexible (or generic) storage requires *view-based query rewriting*
- Interesting performance advantages stem from various *encodings: path, ID, ...*
- Fragmentation (horizontal/vertical) *facilitates navigation* and *complicates reconstruction*

Structural Join Algorithms

Tree Pattern Query – Structural Relationships



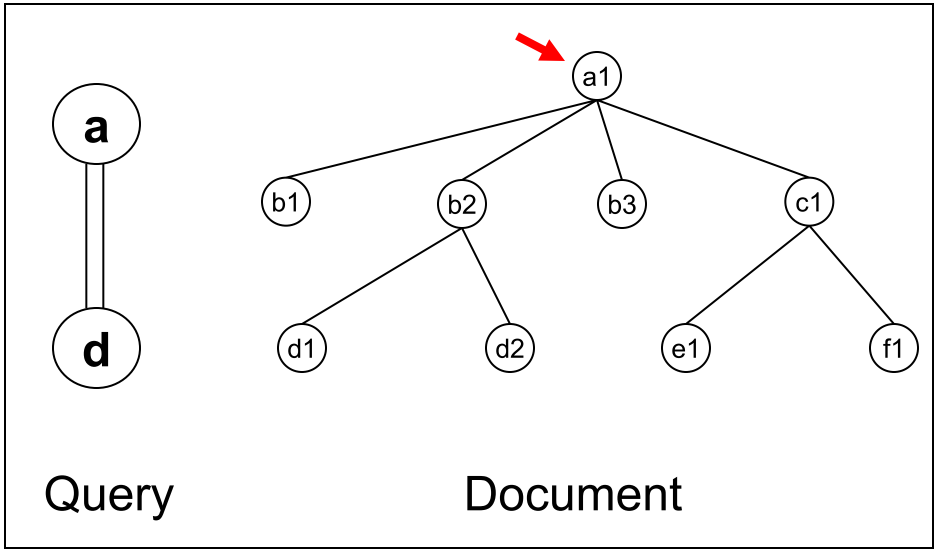


Query Evaluation Methods

- Iterator model of execution
  - Navigation
  - Streaming
- Set-based execution model
  - Structural join

Navigation

- Navigation



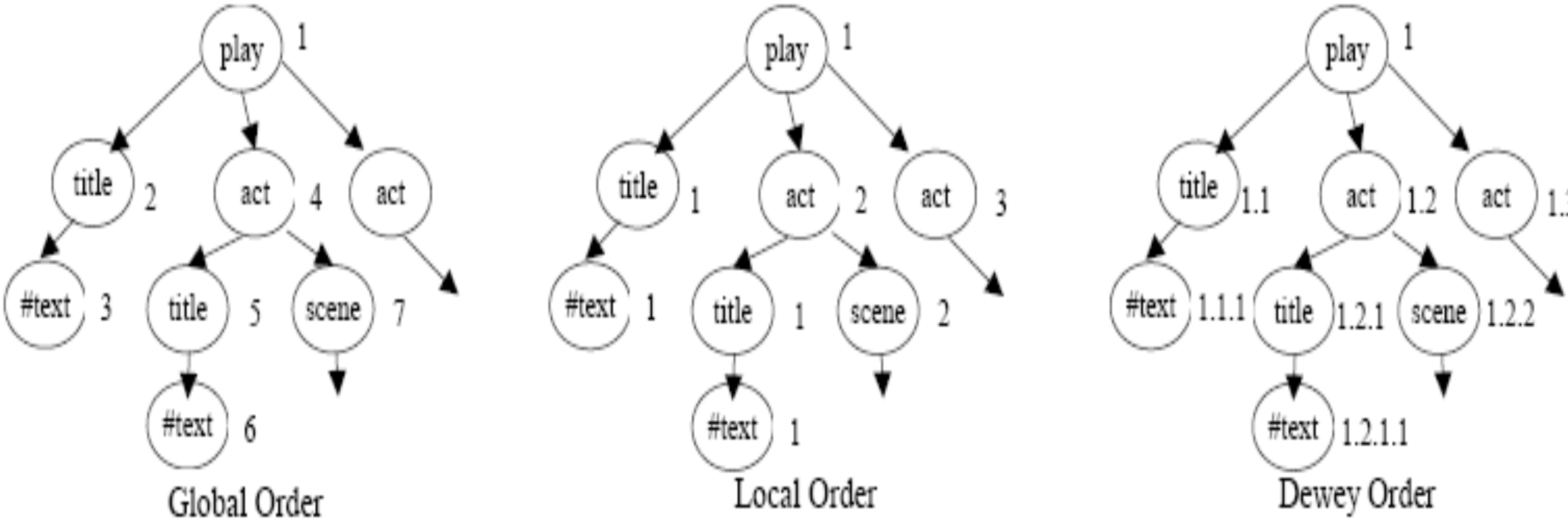
Why do we need XML Node Label?

- We want to store XML documents in RDB in order to utilize the RDB legacy
- XML data model is *ordered*, but relational data model is *unordered*
- How can we support *ordered* XML data model in *unordered* relational model?
- Encode the *order* as data value
- => XML node labeling

Why Start With Labeling Schemes ?

- Idea: assign labels to XML elements
  - unique identifiers +
  - useful information for query processing
- Source of big performance improvements over relational storage + traditional joins
- Many labeling schemes
  - trade-off between space occupancy, information contents, and suitability to updates
  - most frequent one: region-based (“pre-post")
    - shortcomings and alternatives
  - new ones still being produced

Traditional XML Node Labeling



Problem of Traditional Labeling

- Global Order

  - Poor insertion performance

    (could require whole renumbering)

- Local Order

  - Still require local renumbering for insertion

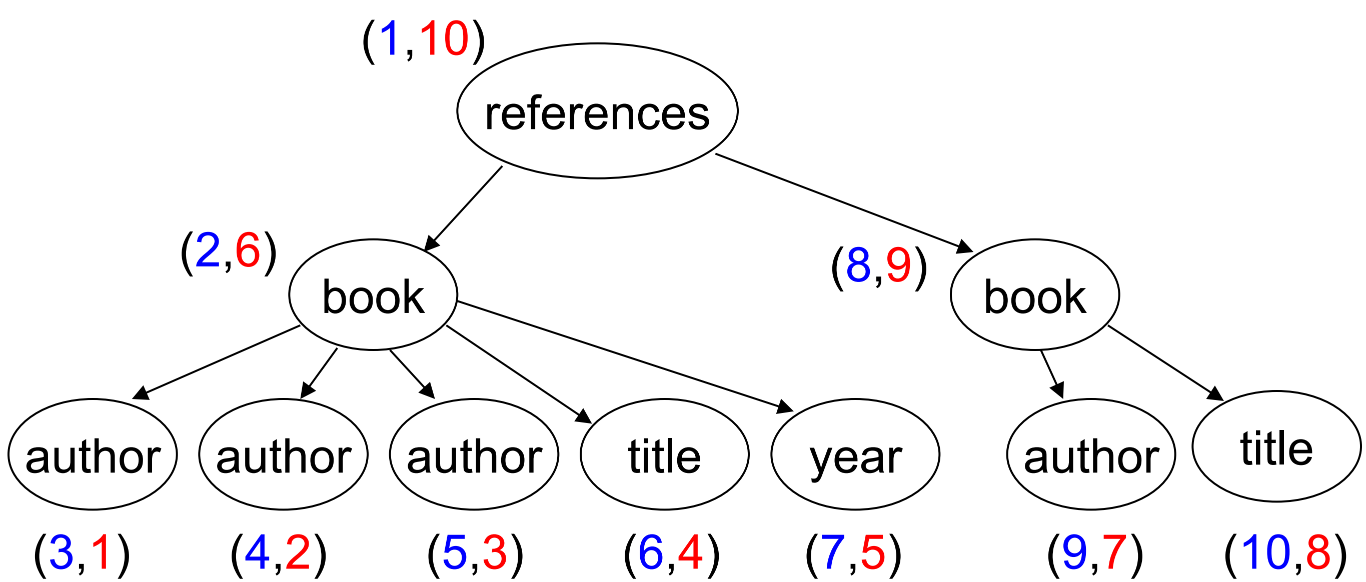
  - The tree semantics is not represented very well

- Dewey Order

  - Still require local renumbering for insertion

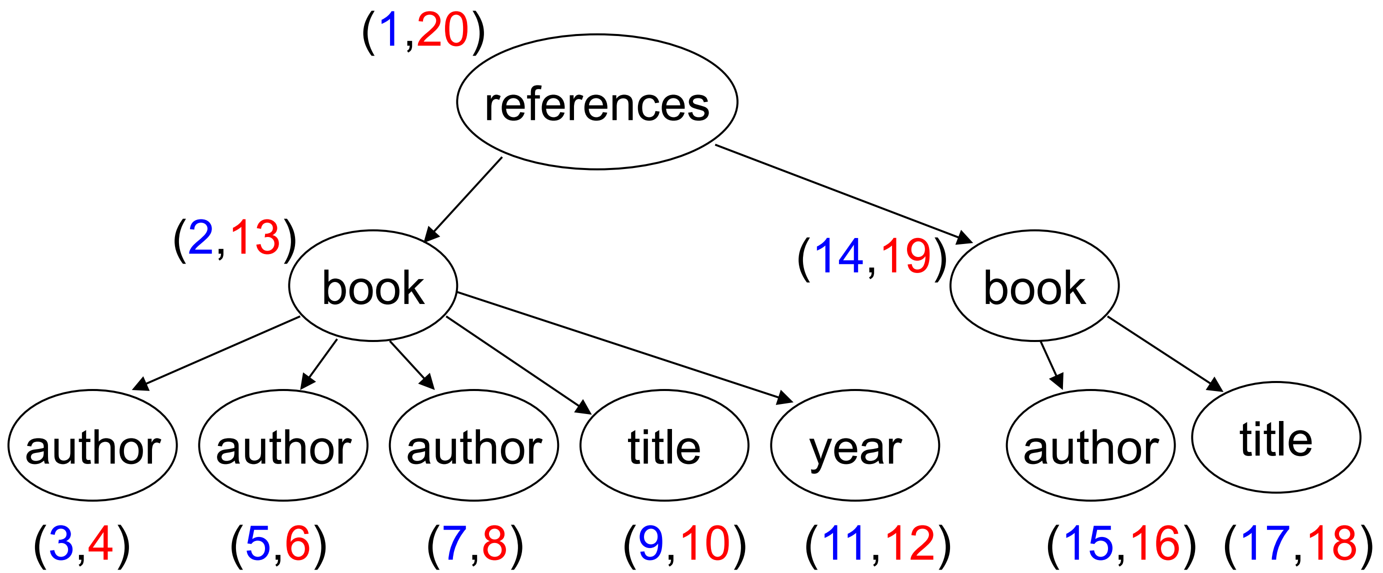
Node Labeling Scheme (1)

- (preorder, postorder) [Dietz82]
  - x is an ancestor of y iff x occurs before y in the preorder traversal and after y in the postorder traversal.



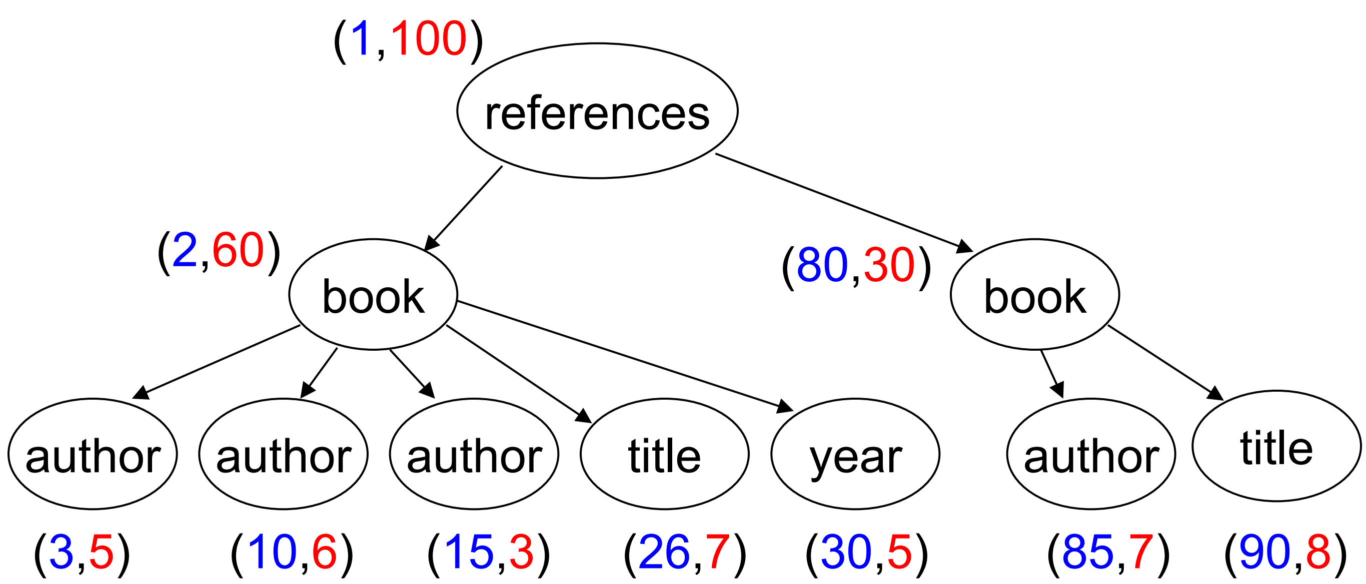
Node Labeling Scheme (2)

- (begin, end) [Zhang01]
  - The begin and end positions can be generated by doing a depth-first traversal of the tree and sequentially assigned a number at each visit.



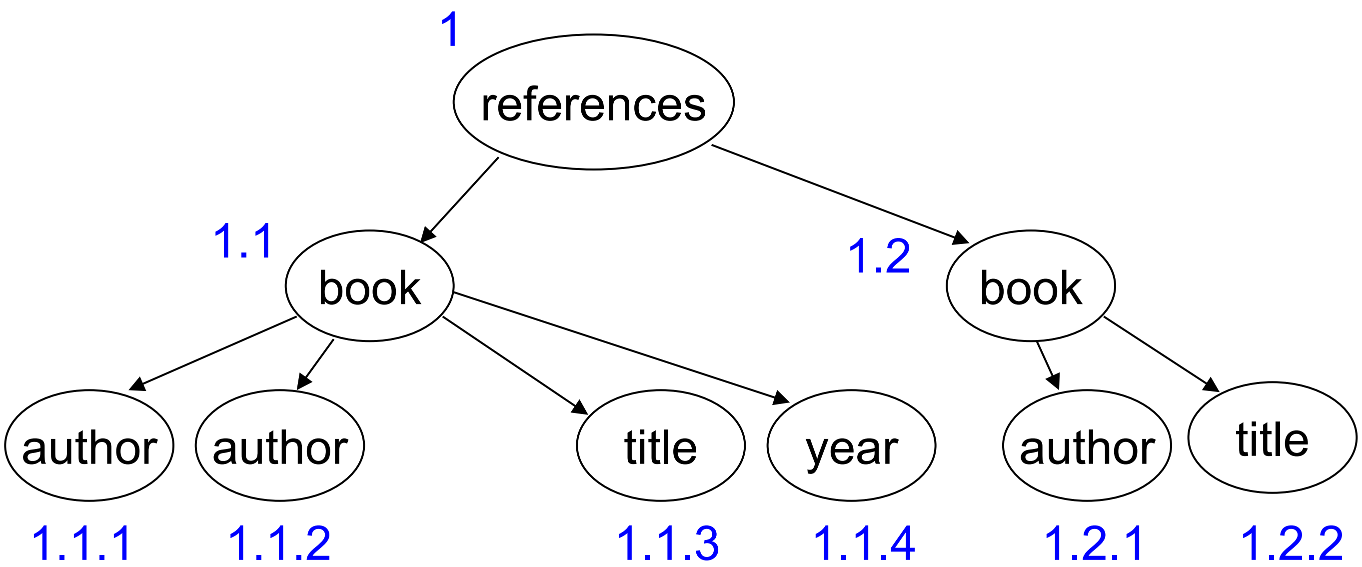
Node Labeling Scheme (3)

- (order, size) [Li01]
  - order(x)<order(y) & order(y)+size(y)<=order(x)+size(x)
  - order(x)+size(x) < order(y)



Node Labeling Scheme (4)

- Dewey Decimal Coding [Tatarinov02]

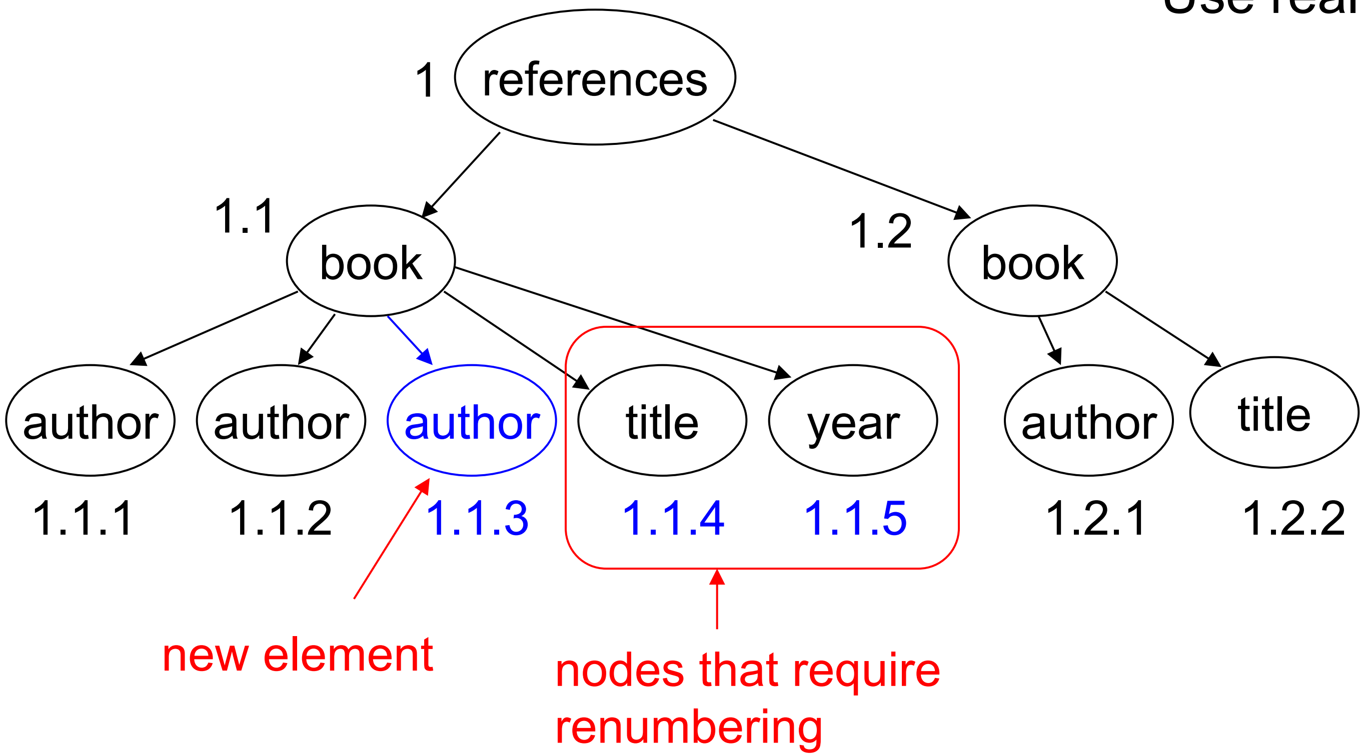


Node Labeling and Updates

- Inserting new elements

Possible solutions:

- Leave empty intervals [Li01]
- Use real numbers [JKC+02]

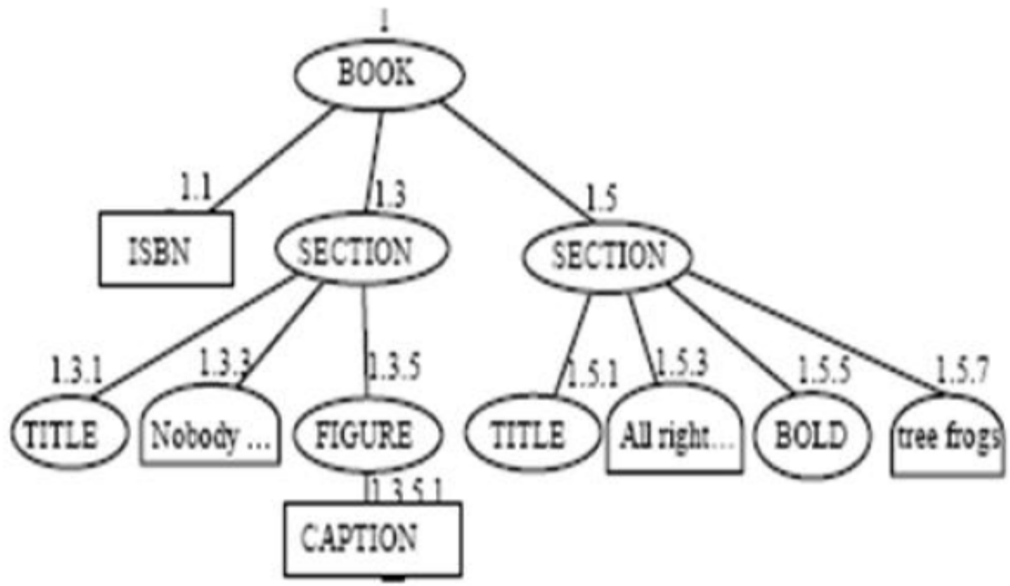


Introduction to ORDPATH?

- ORDPATH is an insertion-friendly XML node labeling similar to the Dewey Ordering
- ORDPATH provides efficient insertion at any position of an XML tree
- Byte-by-byte comparison of ORDPATH yields the proper document order
- ORDPATH keeps the semantics of XML tree
- ORDPATH supports a high performance query plan

Example of ORDPATH

```
<BOOK ISBN="1-55860-438-3">
    <SECTION>
        <TITLE> Bad Bugs</TITLE>
            Nobody loves bad bugs.
        <FIGURE CAPTION="Sample bug"/>
    </SECTION>
    <SECTION>
        <TITLE> Tree Frogs </TITLE>
            All right-thinking people
            <BOLD> love </BOLD> tree frogs.
    </SECTION>
</BOOK>
```



- Only positive odd integers are assigned for the initial load
- Even and negative integers are reserved for later insertions
- Stored as compressed binary representation

XML shredding with ORDPATH

ORDPATH	TAG	NODE TYPE	VALUE
1.	1 (BOOK)	1 (Element)	null
1.1	2 (ISBN)	2 (Attribute)	'1-55860-438-3'
1.3	3 (SECTION)	1 (Element)	null
1.3.1	4 (TITLE)	1 (Element)	'Bad Bugs'
1.3.3	--	4 (Value)	'Nobody loves bad bugs.'
1.3.5	5 (FIGURE)	1 (Element)	null
1.3.5.1	6 (CAPTION)	2 (Attribute)	'Sample bug'
1.5	3 (SECTION)	1 (Element)	null
1.5.1	4 (TITLE)	1 (Element)	'Tree frogs'
1.5.3	--	4 (Value)	'All right-thinking people'
1.5.5	7 (BOLD)	1 (Element)	'love '
1.5.7	--	4 (Value)	'tree frogs'

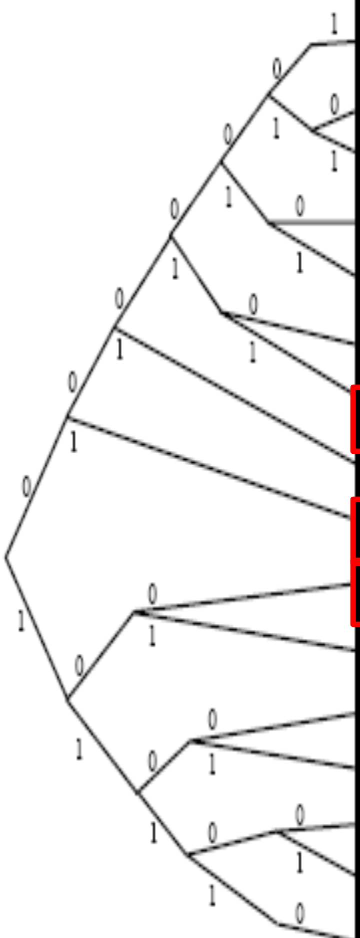
- XML document is shredded into a node table
- Another table of tag name to ID will be needed

Compressed ORDPATH Format

L_0	O_0	L_1	O_1	. . .	L_k	O_k

- ORDPATH is represented as pairs of variable-length Li/Oi bitstrings
- Li represents the length of Oi bitstring
- Oi represents each integer of ORDPATH
- Li bitstrings are encoded using prefix-free encoding scheme with binary tree table
- This format tells the way to parse all the ORDPATH bitstrings from left to right

Example



Bitstring	L_i	O_i value range
0000001	48	$[-2.8\times10^{14}, -4.3\times10^{9}]$
0000010	32	$[-4.3\times10^{9}, -69977]$
0000011	16	[-69976, -4441]
000010	12	[-4440, -345]
000011	8	[-344, -89]
00010	6	[-88, -25]
00011	4	[-24, -9]
001	3	[-8, -1]
01	3	[0, 7]
100	4	[8, 23]
101	6	[24, 87]
1100	8	[88, 343]
1101	12	[344, 4439]
11100	16	[4440, 69975]
11101	32	$[69976, 4.3\times10^{9}]$
11110	48	$[4.3\times10^{9}, 2.8\times10^{14}]$

A table of Li with Oi value range

- Using the Li encoding table on the left, the bitstring of ORDPATH="1.5.3.-9.11" can be represented as

01	001	01	101	01	011	00011	1111	100	0011
$L_0=3$	$O_0=1$	$L_1=3$	$O_1=5$	$L_2=3$	$O_2=3$	$L_3=4$	$O_3=-9$	$L_4=4$	$O_4=11$

- If X is a prefix of Y, X is a parent of Y
- This keeps the semantics of original XML tree

ORDPATH Length

Bitstring	L_i	O_i value range
0000001	48	[-2.8×10^{14}, -4.3×10^{9}]
0000010	32	[-4.3×10^{9}, -69977]
0000011	16	[-69976, -4441]
000010	12	[-4440, -345]
000011	8	[-344, -89]
00010	6	[-88, -25]
00011	4	[-24, -9]
001	3	[-8, -1]
01	3	[0, 7]
100	4	[8, 23]
101	6	[24, 87]
1100	8	[88, 343]
1101	12	[344, 4439]
11100	16	[4440, 69975]
11101	32	[69976, 4.3×10^{9}]
11110	48	[4.3×10^{9}, 2.8×10^{14}]

(a)

Bitstring	L_i	O_i value range
000000001	20	[-1118485, -69910]
00000001	16	[-69909, -4374]
0000001	12	[-4373, -278]
000001	8	[-277, -22]
00001	4	[-21, -6]
0001	2	[-5, -2]
001	1	[-1, 0]
01	0	[1, 1]
10	1	[2, 3]
110	2	[4, 7]
1110	4	[8, 23]
11110	8	[24, 279]
111110	12	[280, 4375]
1111110	16	[4376, 69911]
11111110	20	[69912, 1118487]

(b)

When fan-out is 2:

(a) needs $\boxed{\text{5 bits}}$

(b) needs $\boxed{\text{3 bits}}$

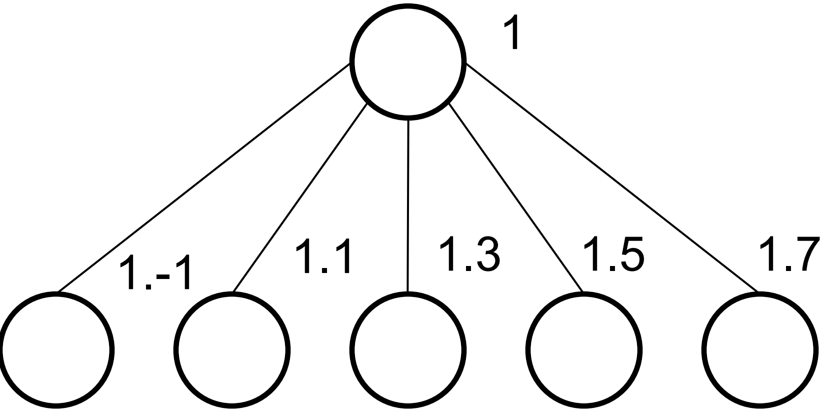
When fan-out is 50:

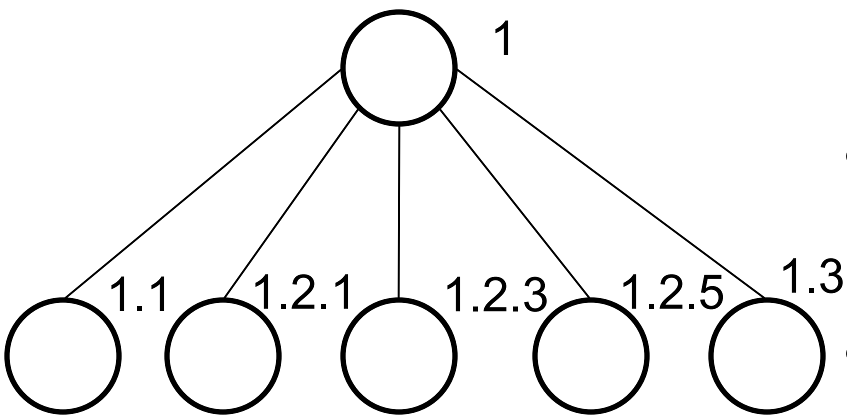
(a) needs $\boxed{\text{9 bits}}$

(b) needs $\boxed{\text{13 bits}}$

- Depending on the statistics (e.g. fan-out) of the XML tree, efficient encoding of Li differs

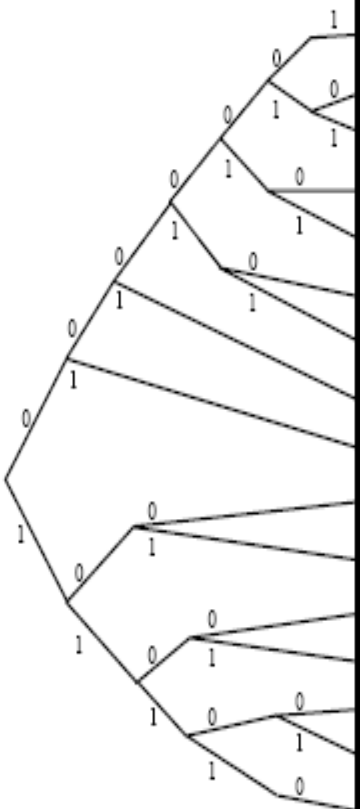
Node Insertions with ORDPATH





- When inserting to the right of all existing children, add 2 to the last child
- When to the left, -2
- When in-between, **caret in**
- Use even integer & one more component
- Enables multiple nodes insertion in-between
- Maintains the proper document order

ORDPATH order after Caret-in



Bitstring	L_i	O_i value range
0000001	48	$[-2.8 \times 10^{14}, -4.3 \times 10^{9}]$
0000010	32	$[-4.3 \times 10^{9}, -69977]$
0000011	16	[-69976, -4441]
000010	12	[-4440, -345]
000011	8	[-344, -89]
00010	6	[-88, -25]
00011	4	[-24, -9]
001	3	[-8, -1]
01	3	[0, 7]
100	4	[8, 23]
101	6	[24, 87]
1100	8	[88, 343]
1101	12	[344, 4439]
11100	16	[4440, 69975]
11101	32	$[69976, 4.3 \times 10^{9}]$
11110	48	$[4.3 \times 10^{9}, 2.8 \times 10^{14}]$

- 1.1: 01 001 01 001
- 1.2.1: 01 001 01 010 01 001
- 1.2.3: 01 001 01 010 01 011
- 1.2.5: 01 001 01 010 01 101
- 1.3: 01 001 01 011

1) Simply compare the bitstring

2) 0 padding & bit-bit comparison

Indexing with ORDPATH

- ORDPATH as the primary index:
  - XML nodes can be sequentially stored on disk in the ORDPATH order
  - Provides an efficient retrieval
  - Ex. A query that retrieves all descendants of X
  - All descendants can be found clustered just after X
- Secondary index:
  - TAG column index provides fast look up by name
  - VALUE column index provides search by text
  - LEVEL of nodes index is useful for Xpath query

Summary

- ORDPATH provides flexible and efficient XML node labeling
- ORDPATH can be represented as compressed binary format
- Li encoding table plays an important role in real application

References

- [Dietz82] P.F. Dietz, “Maintaining order in a linked list”, ACM Symposium on Theory of Computing, May 1982.
- [Zhang01] C. Zhang et al., “On supporting containment queries in relational database management systems”, SIGMOD 2001.
- [Li01] Q. Li and B. Moon, “Indexing and querying XML data for regular path expressions”, VLDB 2001.
- [Tatarinov02] I. Tatarinov et al., “Storing and querying ordered XML using a relational database system”, SIGMOD 2002.
- [ONeil04] P. O’Neil et al., “ORDPATHs: insert-friendly XML node labels”, SIGMOD 2004.
- [JKC+02] H.V.Jagadish, S.Al-Khalifa, A.Chapman, et al. "TIMBER: a Native XML database", VLDB Journal 2002.
- [CTZ+02] S.Chien, V.Tsotras, C.Zaniolo, et al. "Efficient complex query support for multiversion XML documents", EDBT 2002.
- [MBV03] L.Mignet and D.Barbosa and P.Veltri. "The XML Web: a First Study", WWW 2003.

Structural Joins

Relationship established through simple comparisons:

	x // y	x / y
Dewey	c(x) is prefix of c(y)	c(y)=c(x).n
(pre, post, par_pre)	x.pre<y.pre & y.post<x.post	x.pre=y.par_pre
(begin, end, level)	x.begin<y.begin & y.end<x.end	x.begin<y.begin & y.end<x.end & x.level=y.level-1
(pre, size, depth)	x.pre<y.pre & y.pre+y.size<x.pre+x.size	x.pre<y.pre & y.pre+y.size<x.pre+x.size & x.depth=y.depth-1

Structural Joins: A Primitive for Efficient XML Query Pattern Matching

S. Al-Khalifa et al., ICDE 2002

Structural Join Algorithms

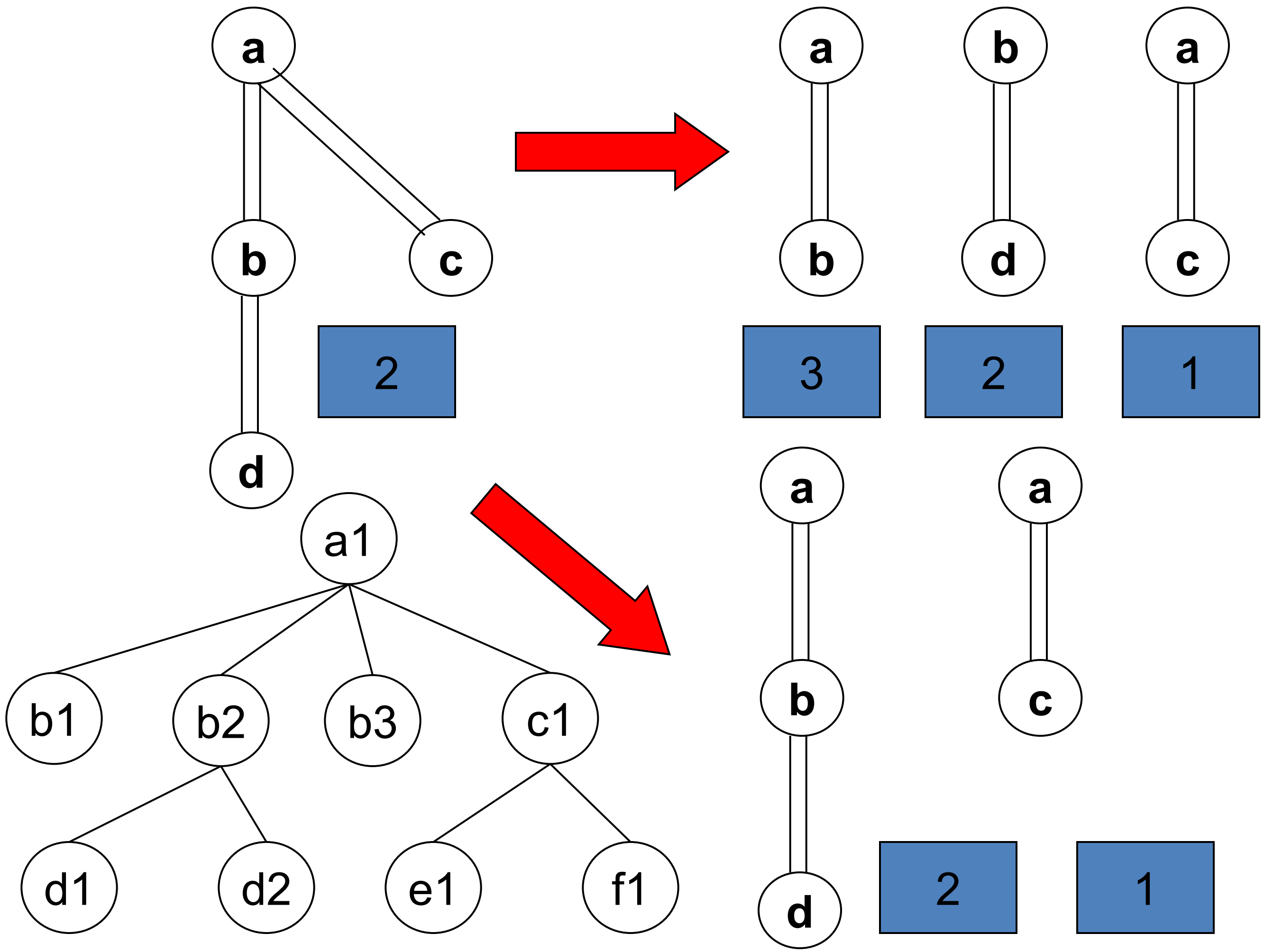
- Two input lists
  - Ancestor (or parent) and descendant (or child)
  - Both sorted by start position
- One output list
  - Pairs of ancestor/descendant or parent/child
  - Sorted by first or second element
- Two families of algorithms presented
  - *With* and *without* stacks
  - Output ordered by *ancestor* and by *descendant*

Tree Merge Join Algorithms

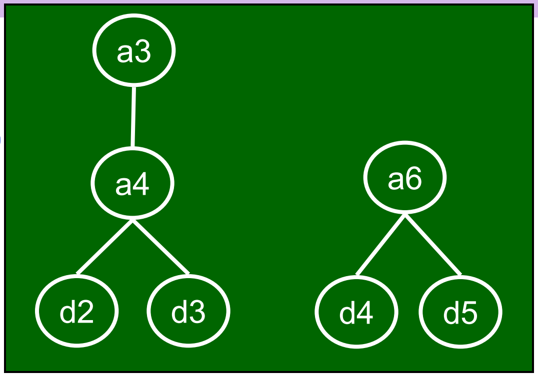
- Natural extension of traditional relational merge joins to deal with multiple inequality conditions
  – E.g. MPMGJN [Zhang01]
- Time complexity may be quadratic in the worst cases

Twig Join

Massive temporary results

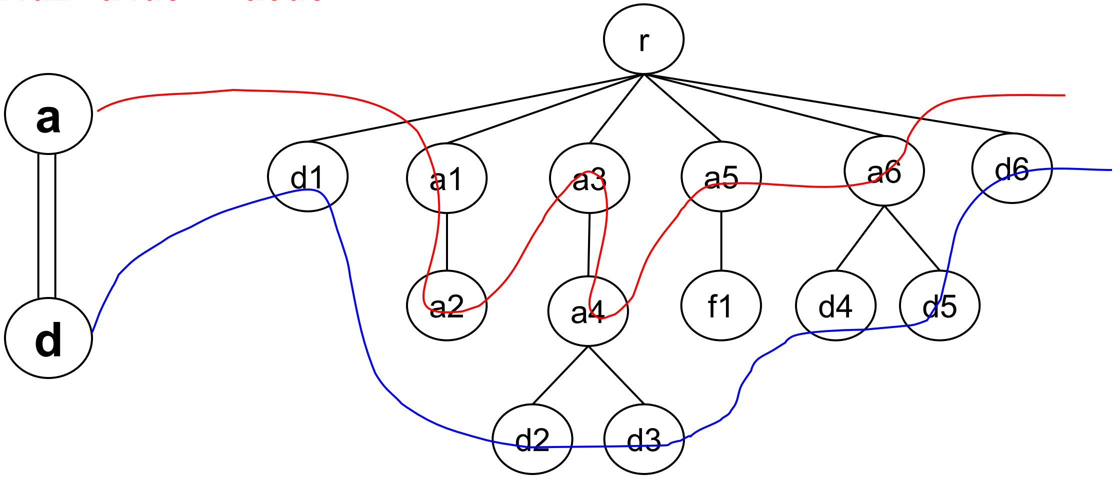


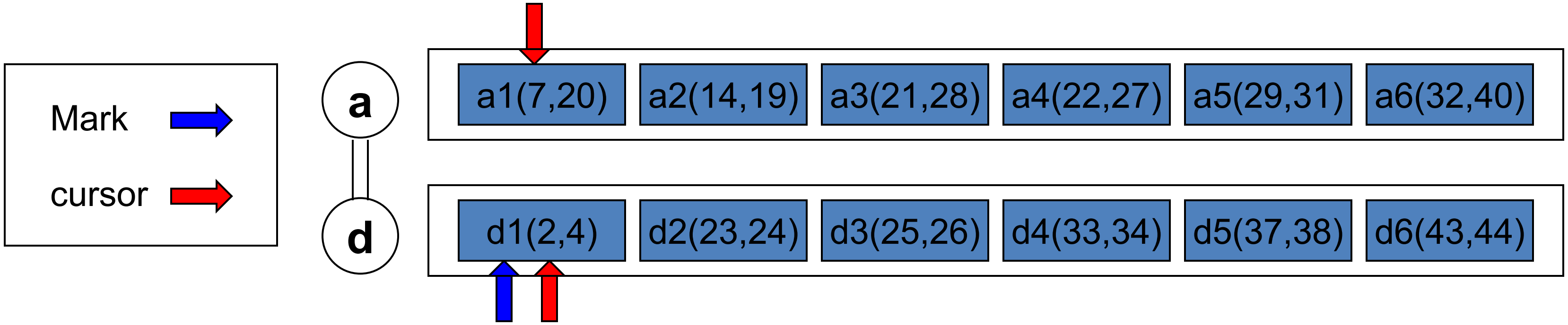
Twig: Tree mer



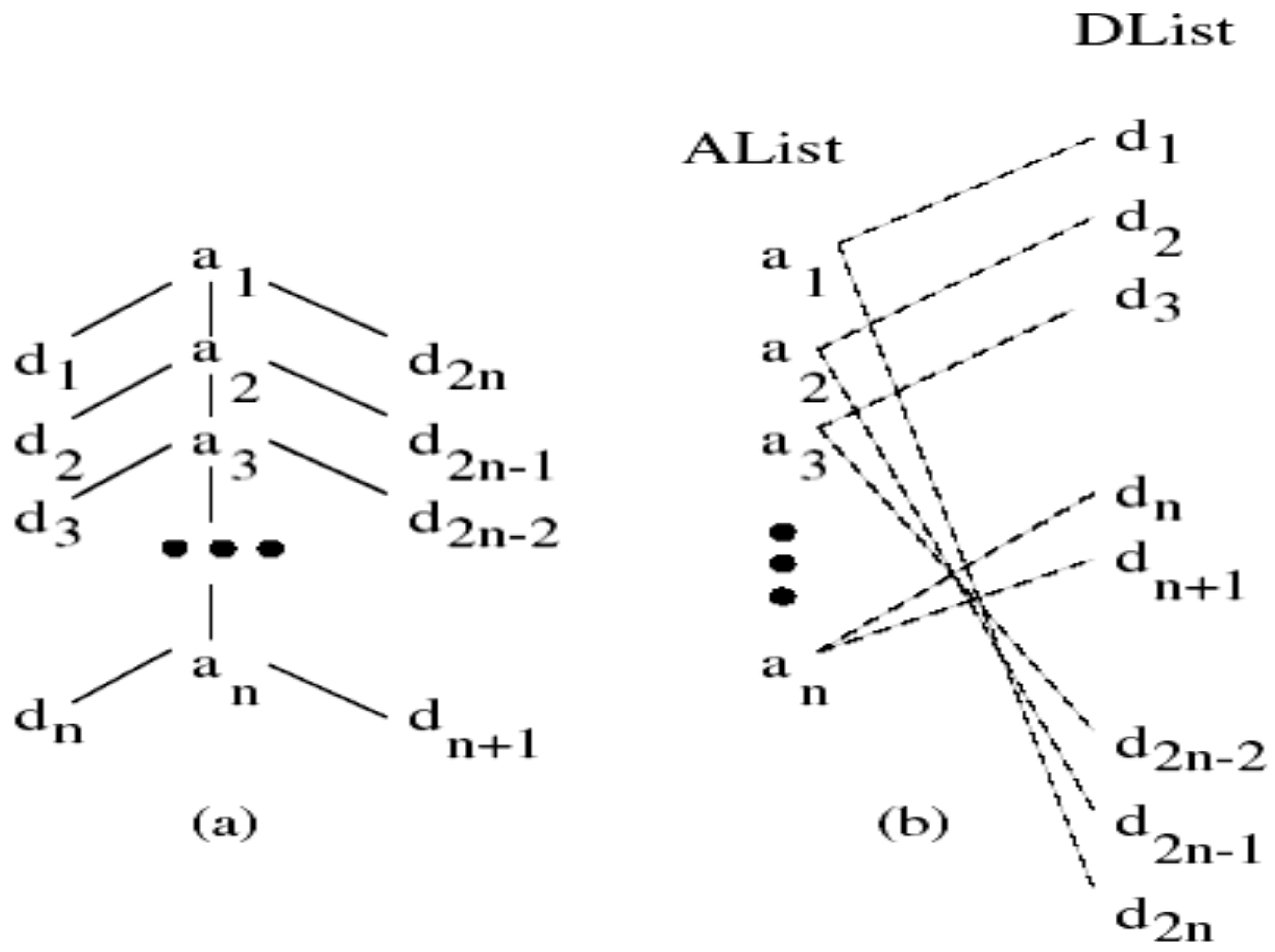
a3d2 a3d3 a6d4

a4d2 a4d3 a6d5

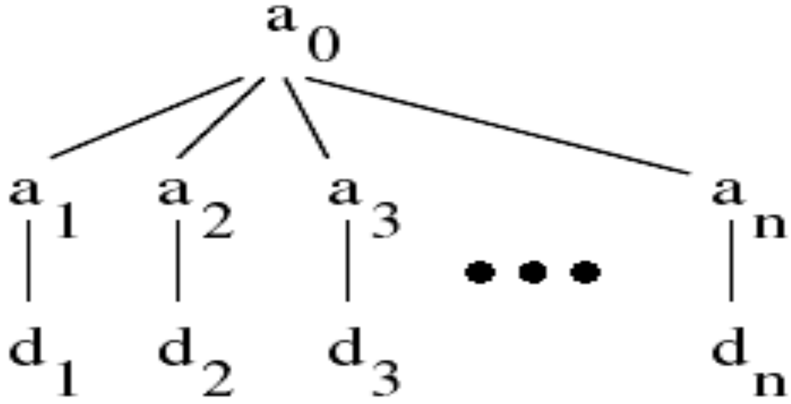


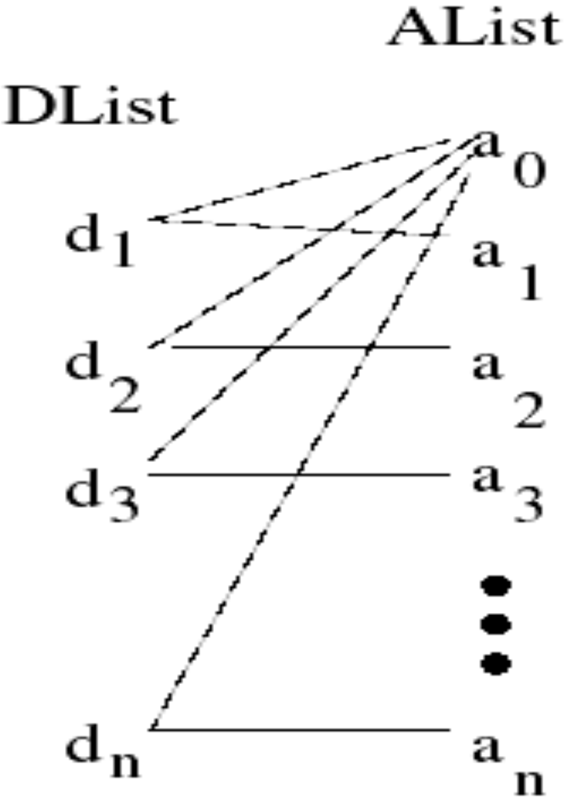


Worst Case for Tree-Merge-Anc



Worst Case for Tree-Merge-Desc

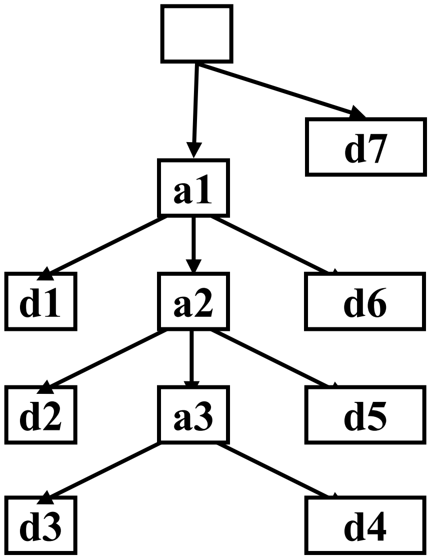




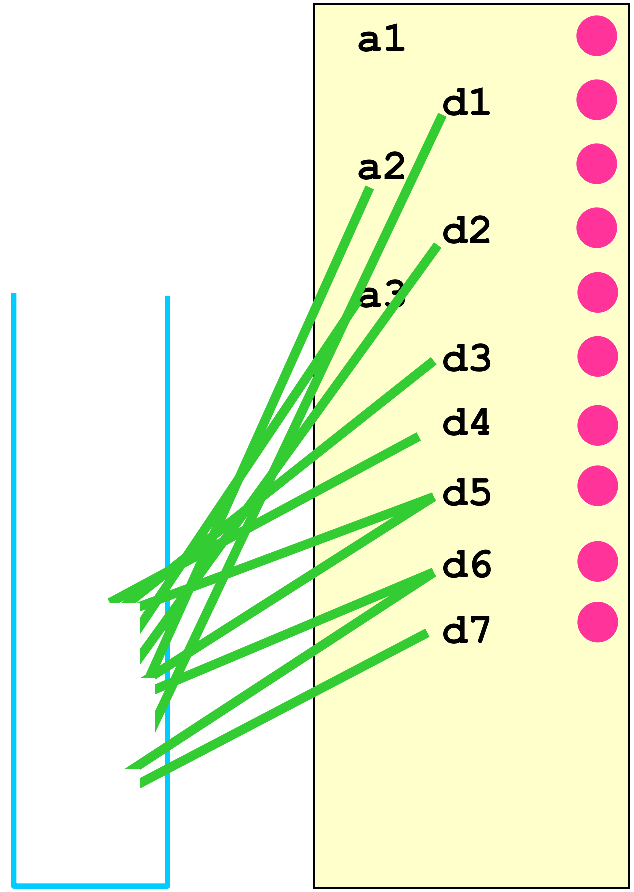
Stack Tree Join Algorithms

- Basic idea: depth first traversal of XML tree
  - *Linear* time with stack size = depth of tree
  - All ancestor-descendant relationships appear on stack during traversal
  - Traverse the lists only once
- Main problem: do not want to traverse the whole database, just nodes in AList/DList

Stack-Tree-Desc



Stack:



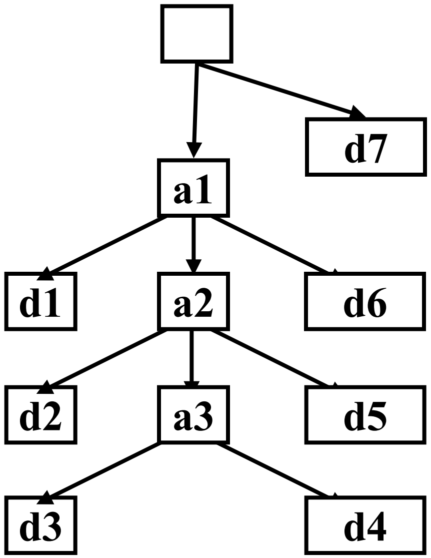
Output:

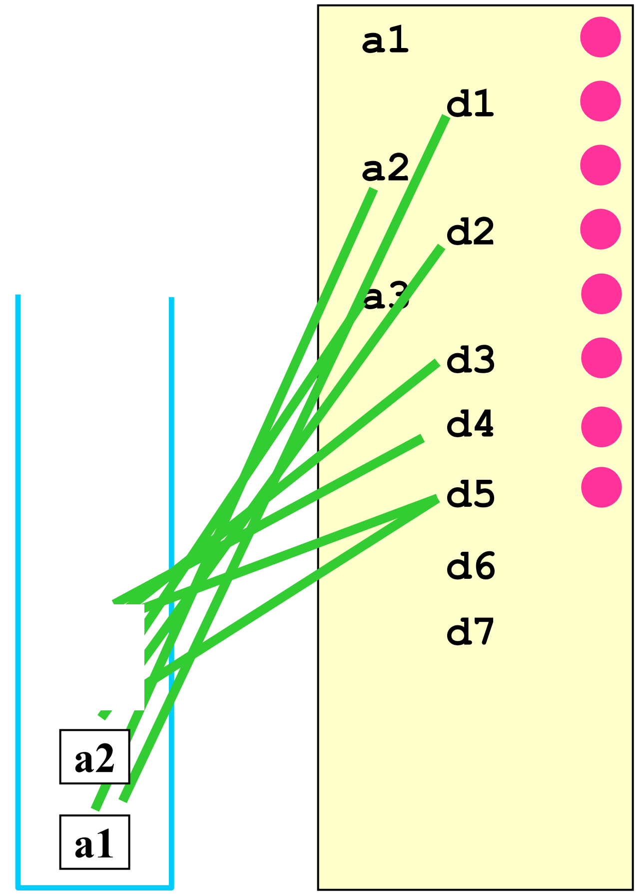
(a1,d1)	(a1,d2),(a2,d2)	(a1,d3),(a2,d3),(a3,d3)
(a1,d4),(a2,d4),(a3,d4)	(a1,d5),(a2,d5)	(a1,d6)

Stack-Tree-Anc

- Goal: Return pairs sorted by (anc, desc)
- Basic Idea: Instead of printing output immediately, store output of each level.
- When node is popped, append its output to the node below it
- Each node has 2 lists:
  - Pairs that it is part of
  - Pairs that it "inherited"

Stack-Tree-Anc

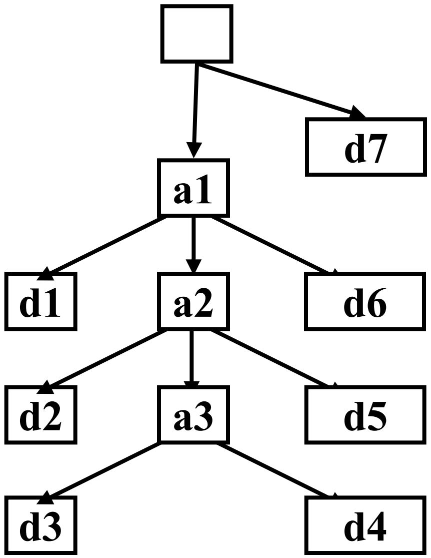




(a3,d3),(a3,d4) (a2,d2),(a2,d3),(a2,d4)

(a1,d1),(a1,d2),(a1,d3),(a1,d4)

Stack-Tree-Anc



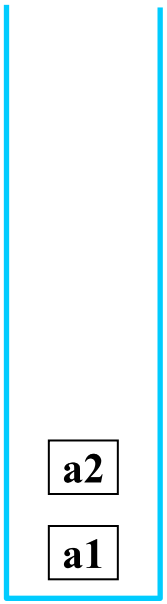
What happens next?

a1
d1
a2
d2
a3
d3
d4
d5
d6
d7

(a3,d3),(a3,d4)

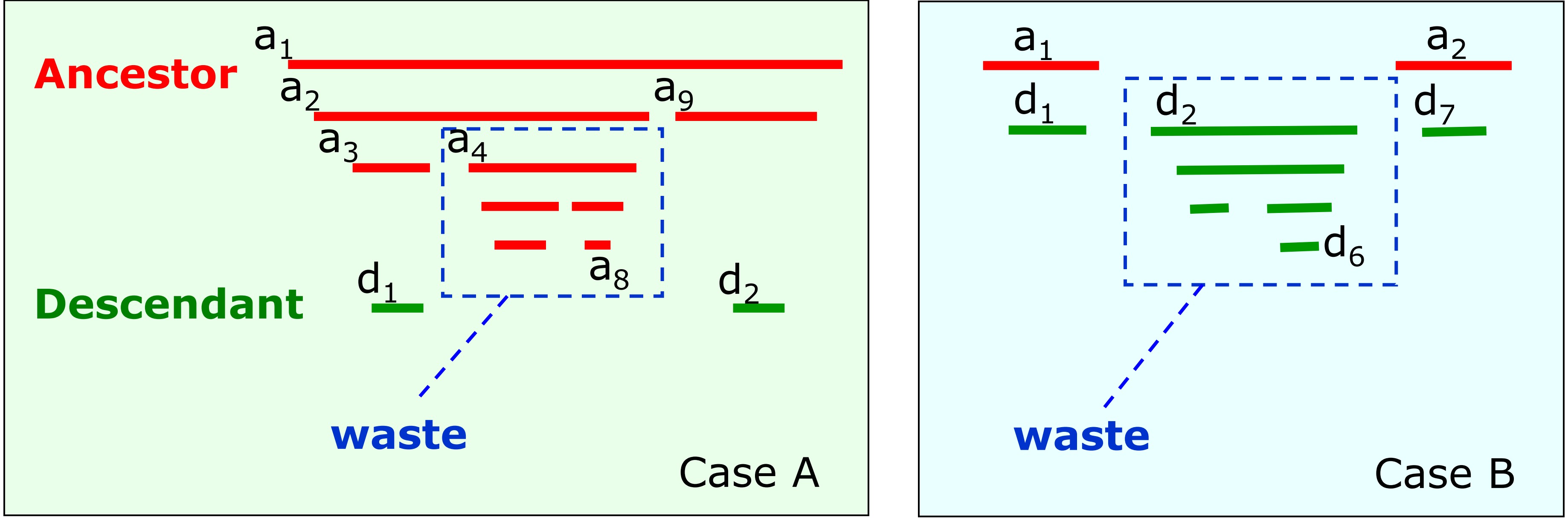
(a2,d2),(a2,d3),(a2,d4),(a2,d5)

(a1,d1),(a1,d2),(a1,d3),(a1,d4),(a1,d5)



Indexed Structural Join

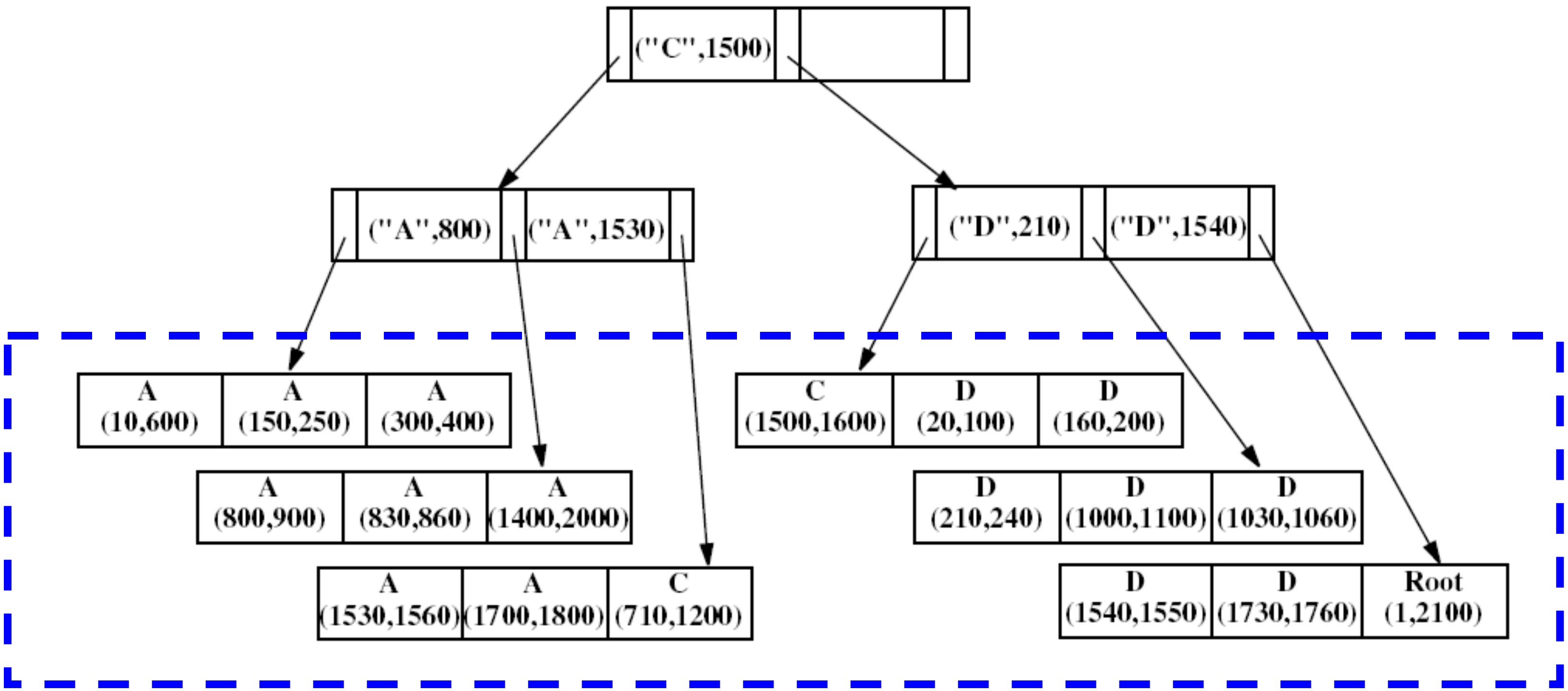
- Utilizing the existing indices (e.g. B+ tree, R*-Tree) to skip elements that do not participate in the join.
  - Shu-Yao Chien, et al., *Efficient Structural Joins on Indexed XML Documents*, VLDB 2002
- XR-tree (XML Region Tree)
  - Haifeng Jiang, et al., *XR-Tree: Indexing XML Data for Efficient Structural Joins*, ICDE 2003.
  - supports efficient retrieval of elements by structure relationship.

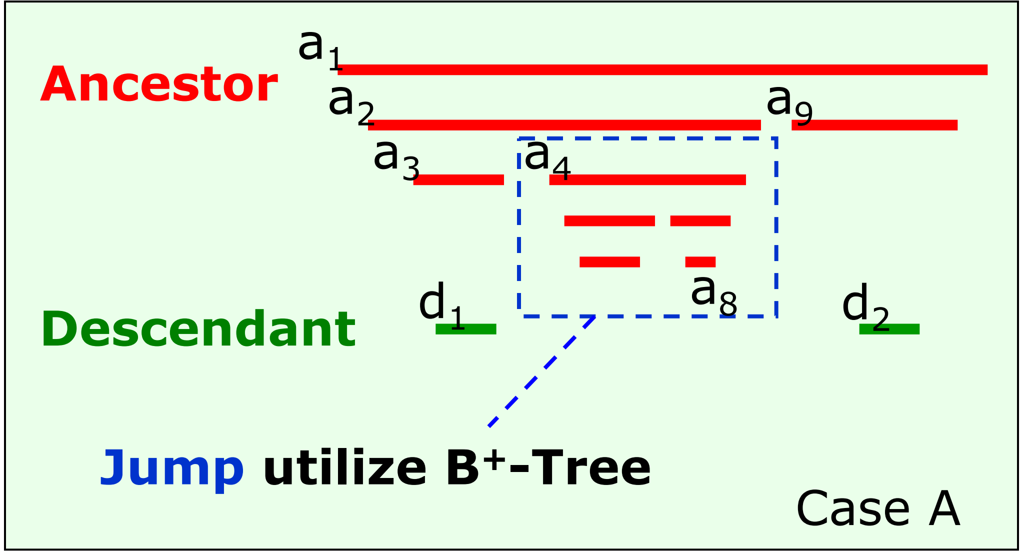


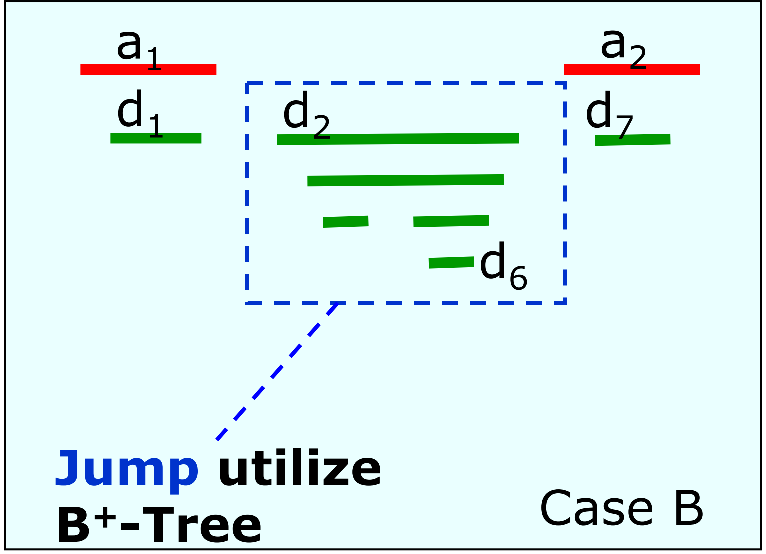
- Case A
  - (1) Push a1, a2 and a3 into the stack and join them with d1;
  - (2) Pop a3,a2 from stack
  - (3) examine (push into and pop from the stack) elements a4 ~ a8 from Ancestor-List
  - (4) Push a9 into the stack and then join a1 and a9 with d2
- Case B
  - Same the Case A
  - After a is joined with d1, the algorithm will sequentially scan the descendant elements d2~d6.

Structural Join using B$^+$-tree

- Cluster elements from the same tag.
- Multiple elements can be combined into a single index.







If (a is an ancestor of d) then

Push into stack all elements in *A* that are ancestors of d, a point to the last pushed;

Output d as a descendant of all elements in stack, d point next;

Else if (a.end < d.start)then

Pop all stack elements which are before d; (Let l be the last element popped)

Let a be the element in A having the smallest start that is larger than l.end;

Else

Output d as a descendant of all elements in stack;

If (ancestor stack is empty) then

Let d be the element in D having the smallest start that is larger than a.start;

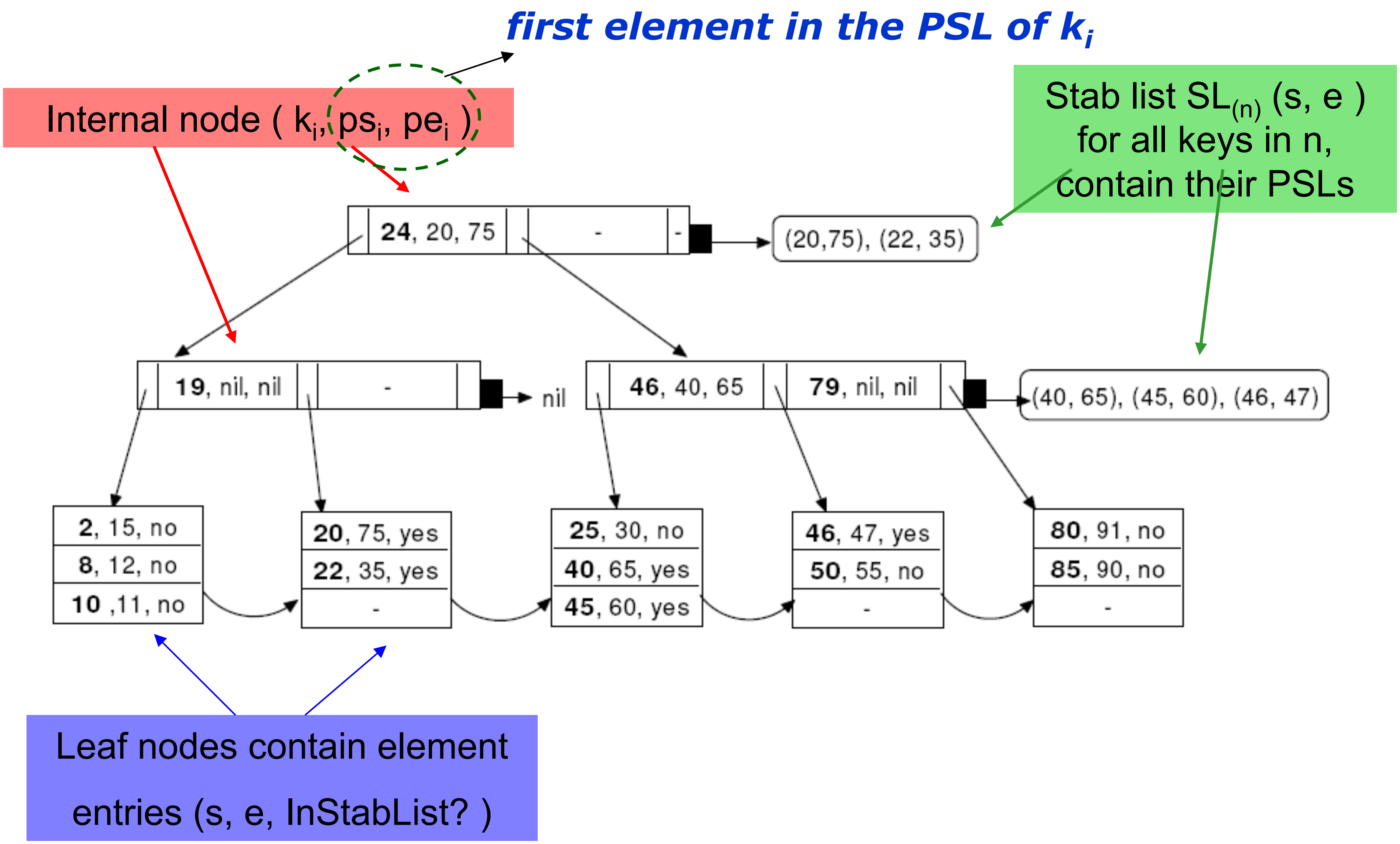
Else

Let d be the next element in D;

XR-Tree

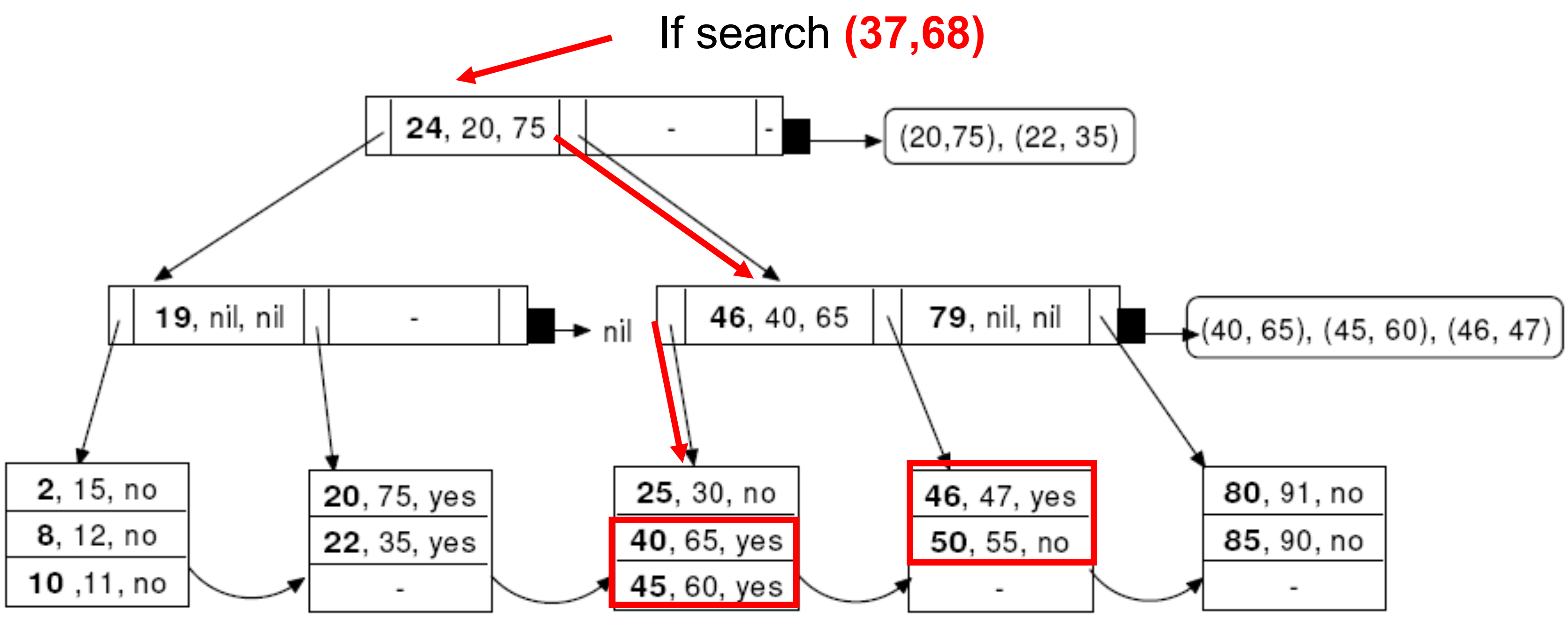
- Based on B+-tree
- Given an element E and an element set R, all E's ancestors (or descendants) in R can be efficiently identified
  - Used to skip ancestors and descendants effectively in a structural join
- Stab List
  - Given a key k and an element $E(s,e)$, k stabs E if $s \leq k \leq e$; *k primarily stabs E*, if k is the smallest key that stabs E.

An Example XR-Tree



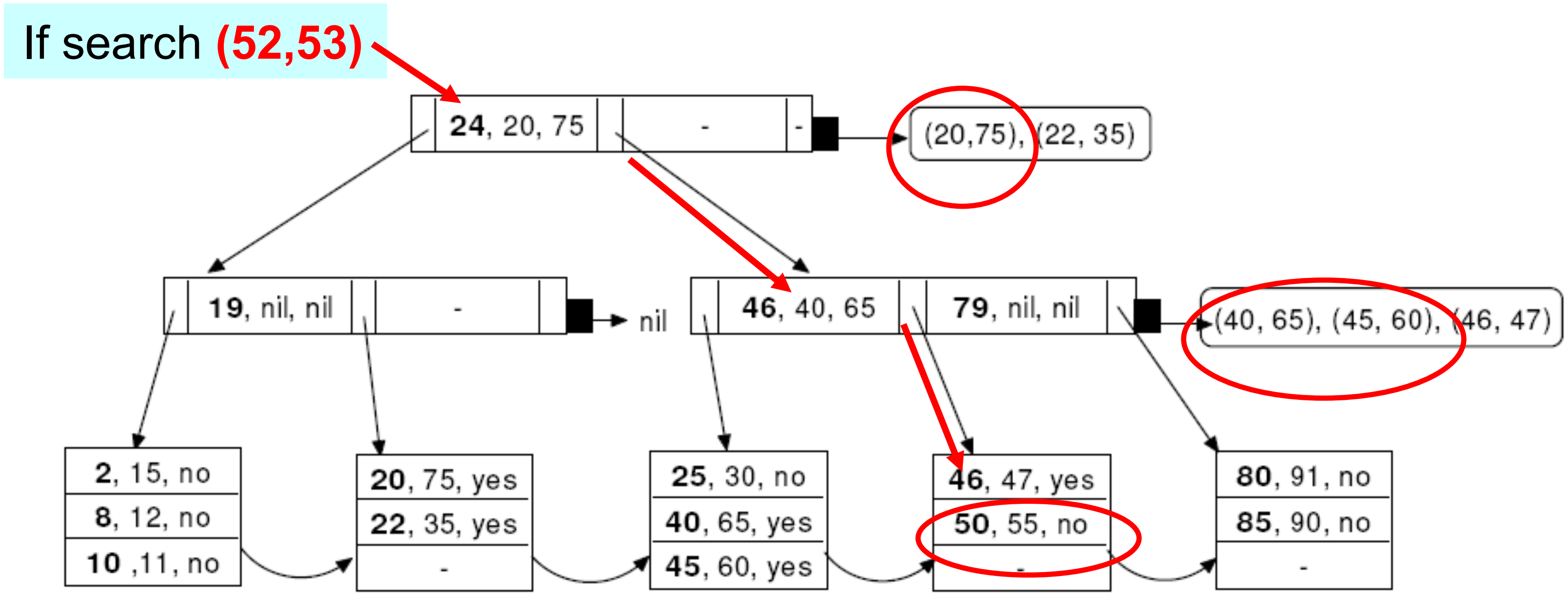
Searching for Descendants

- Given an element E_a (s_a, e_a), find all its descendants.
  - Search like B^+-tree



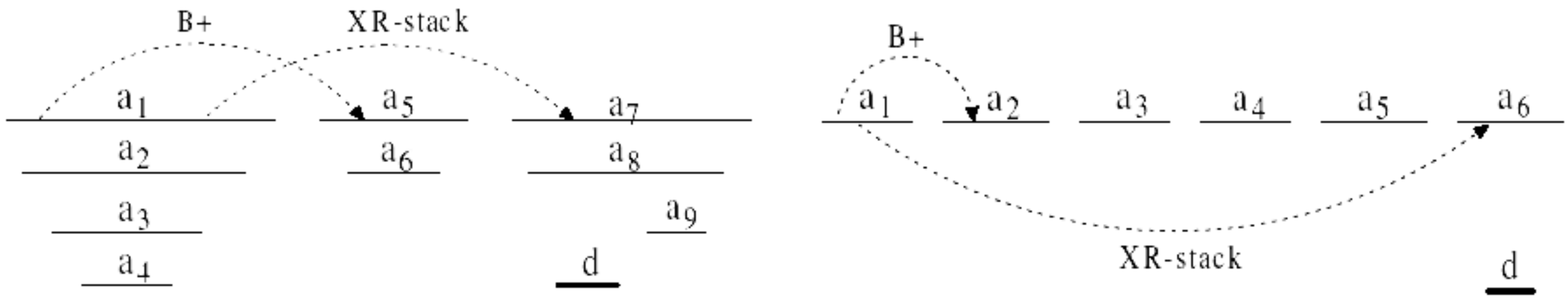
Searching for Ancestors

- During the navigation from the root to the leaf page, we search the stab lists of internal nodes to collect elements stabbed by s_d.
  - If $k_i \le s_d < k_{i+1}$, for c = i+1 to 0 do if $ps_c < s_d < pe_c$, then scan the PSL_c
  - Find the largest key k_i such that $k_i \le s_d$, traverse k_i.rightChild
  - If can't find in PSL then find in leaf page



Stack-based SJ with XR-trees

- Assume that input lists A and D.
- Both sets are indexed by XR-tree.
- The algorithm proceeds like Merge-Join but it effectively skips elements that do not participate in the join



(a) Highly nested (b) Less nested

Tree Pattern Query

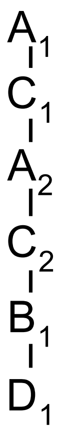
- Binary join approach: If a query is complex and contains many binary relationships, intermediate results can be very large.
- Join ordering:
  - Y.Wu, J.Patel and H.V.Jagadish. "Structural Join Order Selection for XML Query Optimization", ICDE 2003
- Holistic twig join:
  - N.Bruno, N.Koudas and D.Srivastava. "Holistic Twig Joins: Optimal XML Pattern Matching", SIGMOD 2002

Holistic Twig Joins

- Solve the entire twig query in two phases
  1. Produce “guaranteed” partial results using one pass.
  2. Merge join partial results.
- Contributions
  - PathStack and TwigStack algorithms
  - Exploiting XB-tree index

Data Structures

- Each node q in query has associated:
  - A **stream** T_q, with the labels of the elements corresponding to node q, in increasing "begin" order.
  - A **stack** S_q, with a compact encoding of partial solutions (stacks are chained).

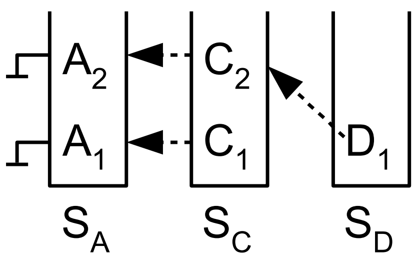


XML fragment

Query

$[A_1, C_1, D_1]$
$[A_1, C_2, D_1]$
$[A_2, C_2, D_1]$

Matches



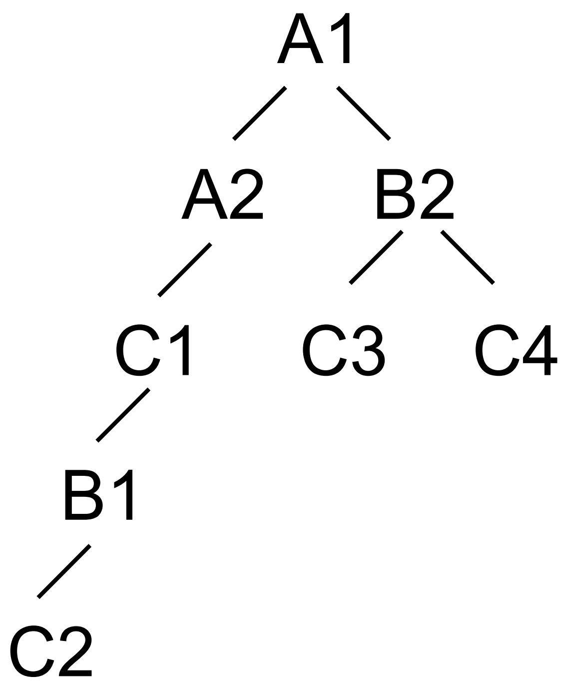
Stacks

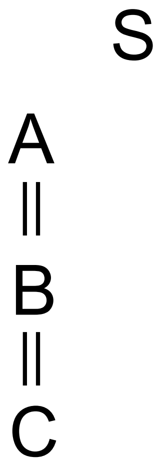
PathStack

- Handles twigs with no branches *q1//q2//…//qn*
- Input lists T_{q1}, T_{q2}, …, T_{qn} and stacks S_{q1}, S_{q2}, …, S_{qn}
- While T_{qn} is not empty:
  - Let T_{qmin} be the list whose head has smallest *begin*;
  - Clean all stacks: pop while top's *end* < *head*(T_{qmin}).*begin*;
  - Push *head*(T_{qmin}) on S_{qmin}, with pointer to *top*($S_{parent(\mathrm{qmin})}$);
  - If *q*min is the leaf (*qn*), output results and pop S_{qmin};

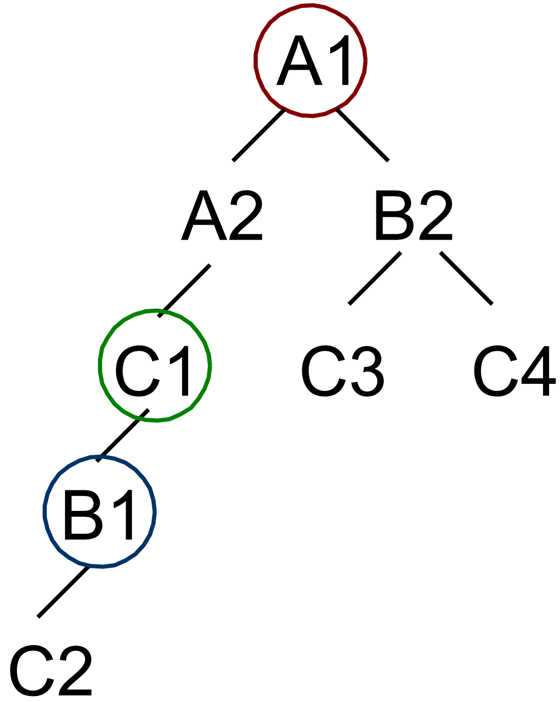
- Check properties
  - Elements in a stack form a containment chain
  - Each stack element points to the top one in the parent stack that contains it

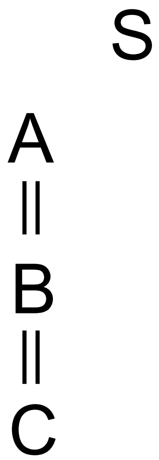
PathStack Example (1)



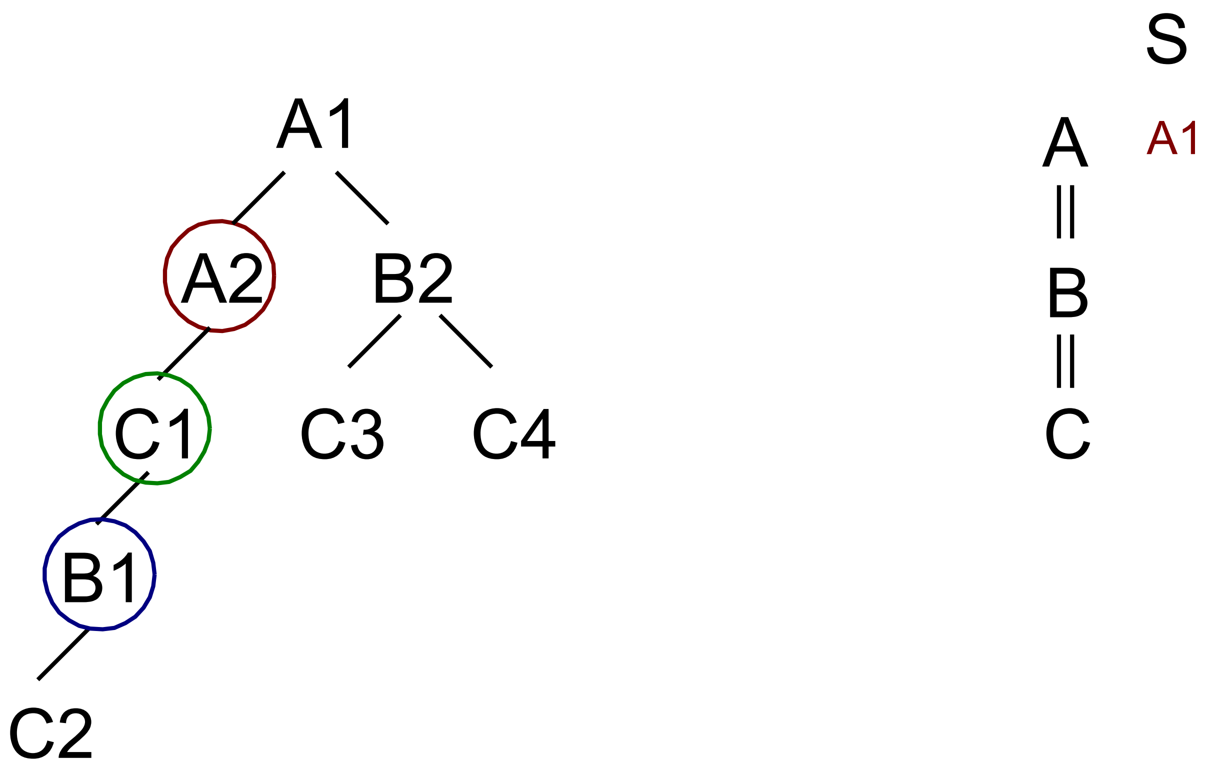


PathStack Example (2)

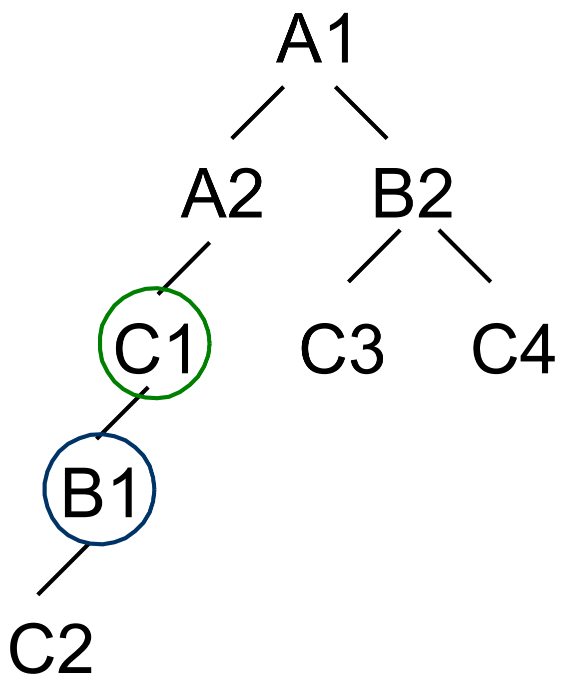


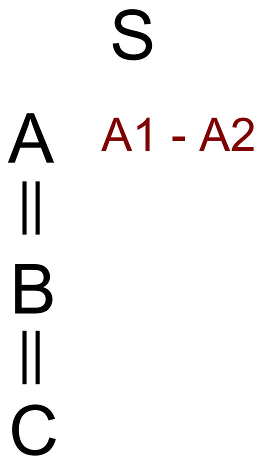


PathStack Example (3)

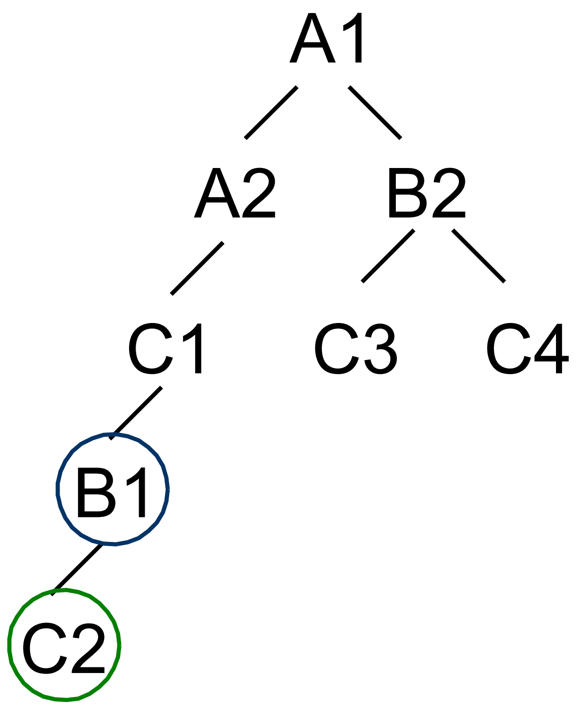


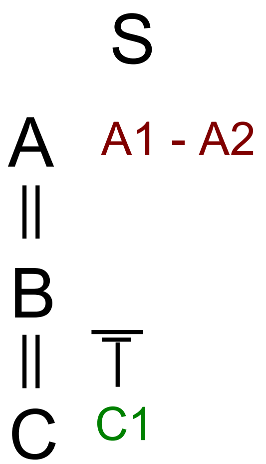
PathStack Example (4)



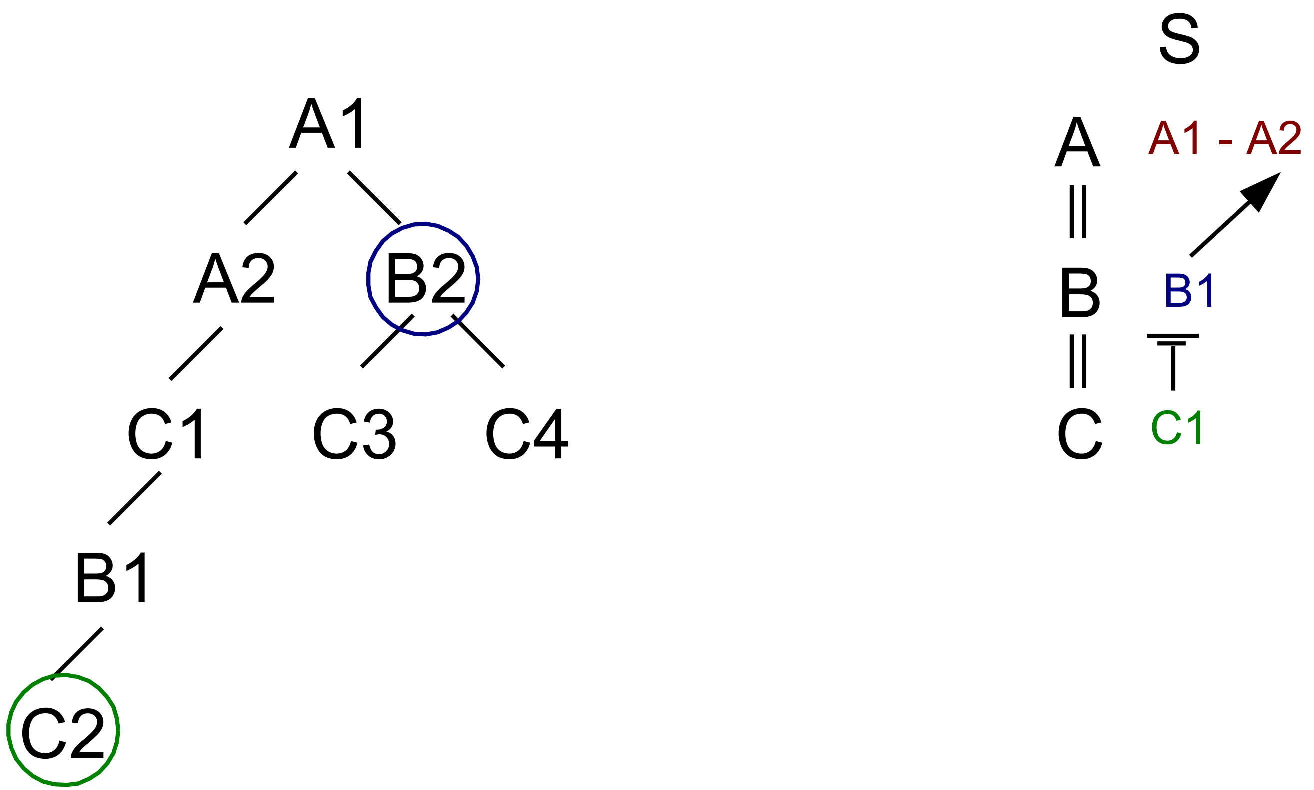


PathStack Example (5)

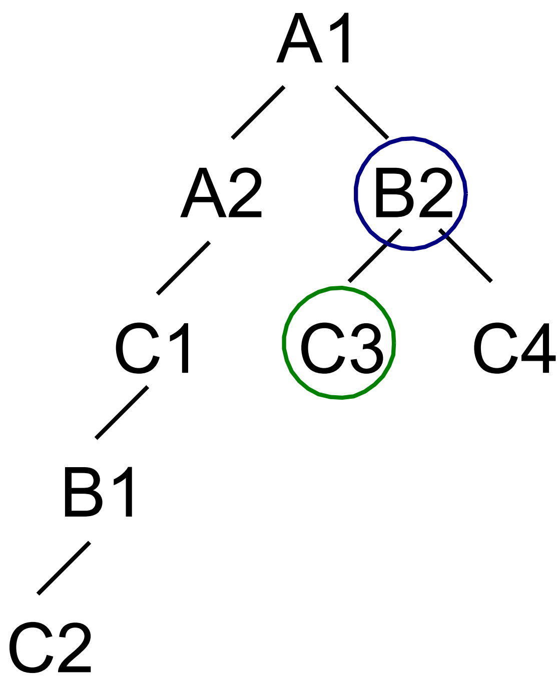


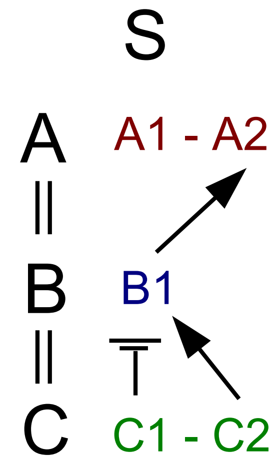


PathStack Example (6)



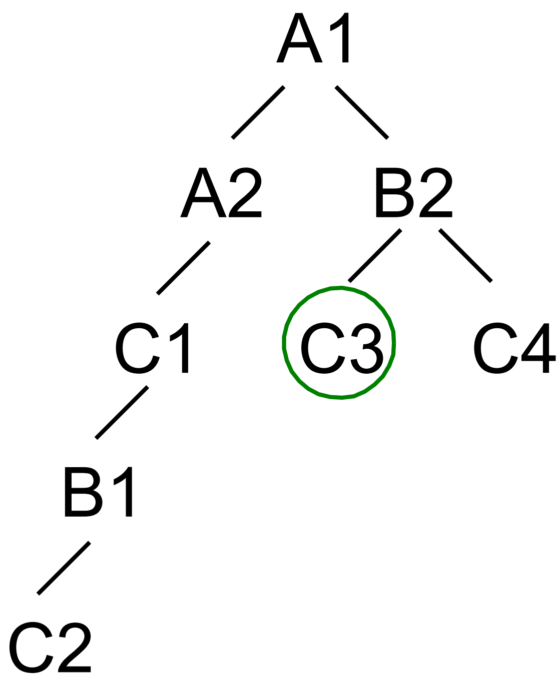
PathStack Example (7)

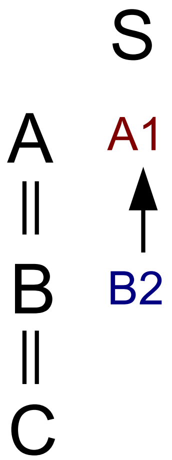




A1,B1,C2
A2,B1,C2

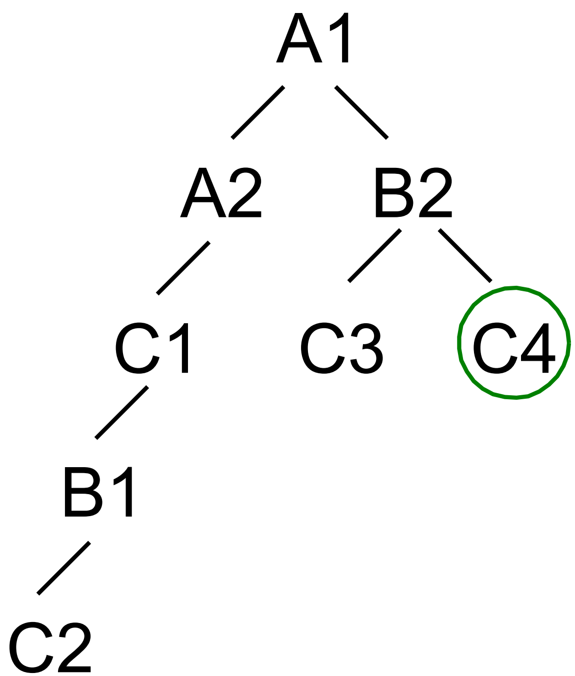
PathStack Example (8)

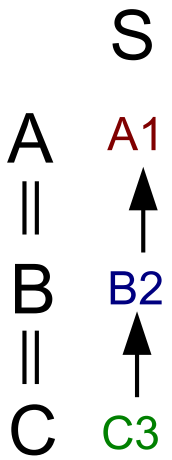




A1,B1,C2
A2,B1,C2

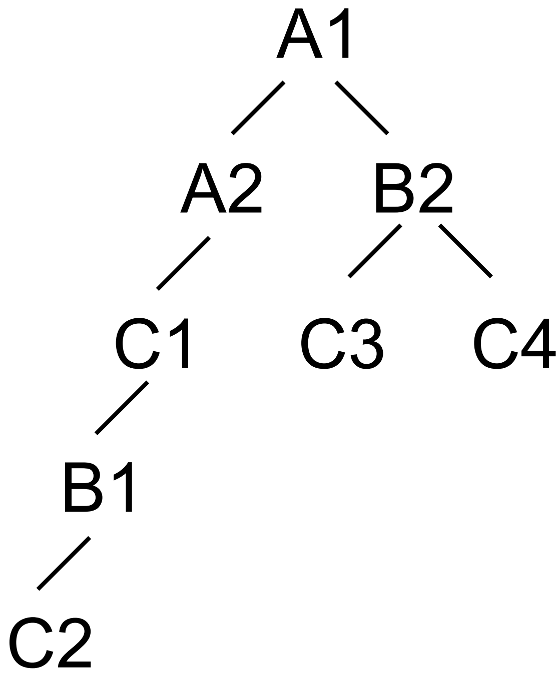
PathStack Example (9)

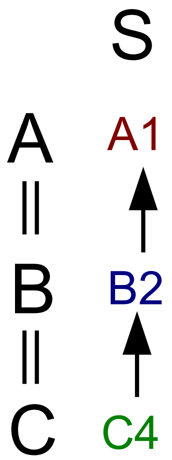




A1,B1,C2
A2,B1,C2
A1,B2,C3

PathStack Example (10)

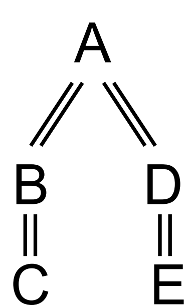


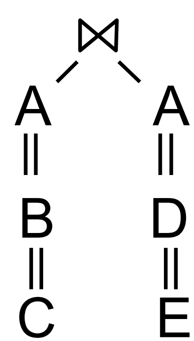


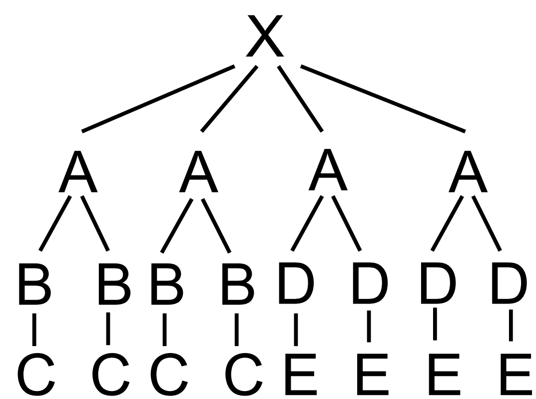
A1,B1,C2
A2,B1,C2
A1,B2,C3
A1,B2,C4

Twig Queries

- Naïve adaptation of PathStack.
  - Solve each root-to-leaf path independently.
  - Merge-join each intermediate result.
- Problem: Many intermediate results might not be part of the final answer.



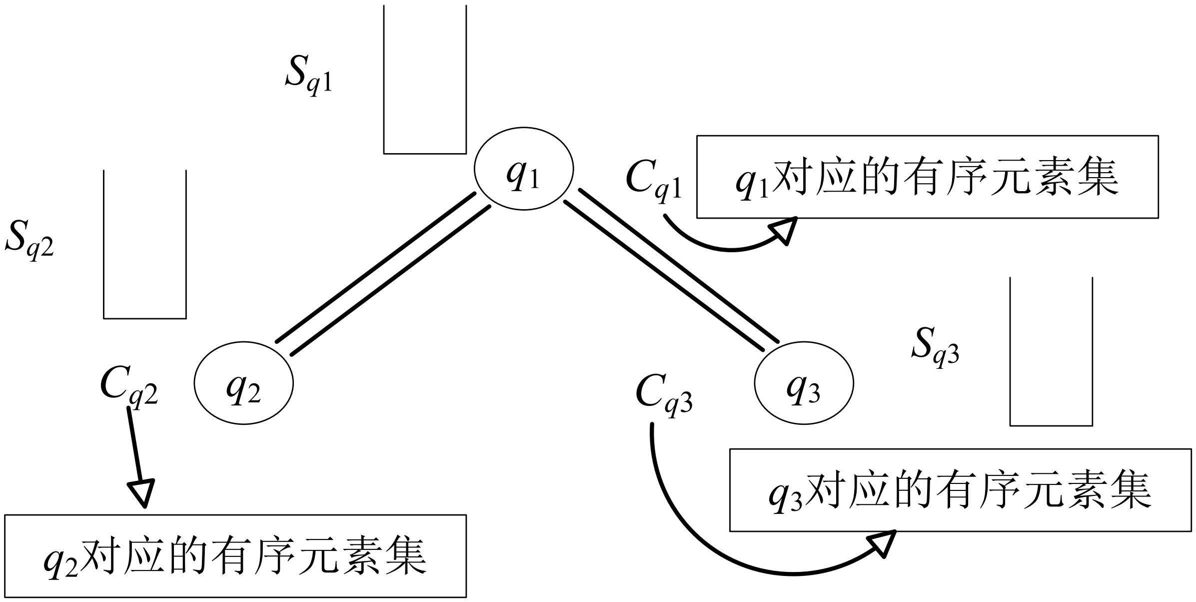




TwigStack

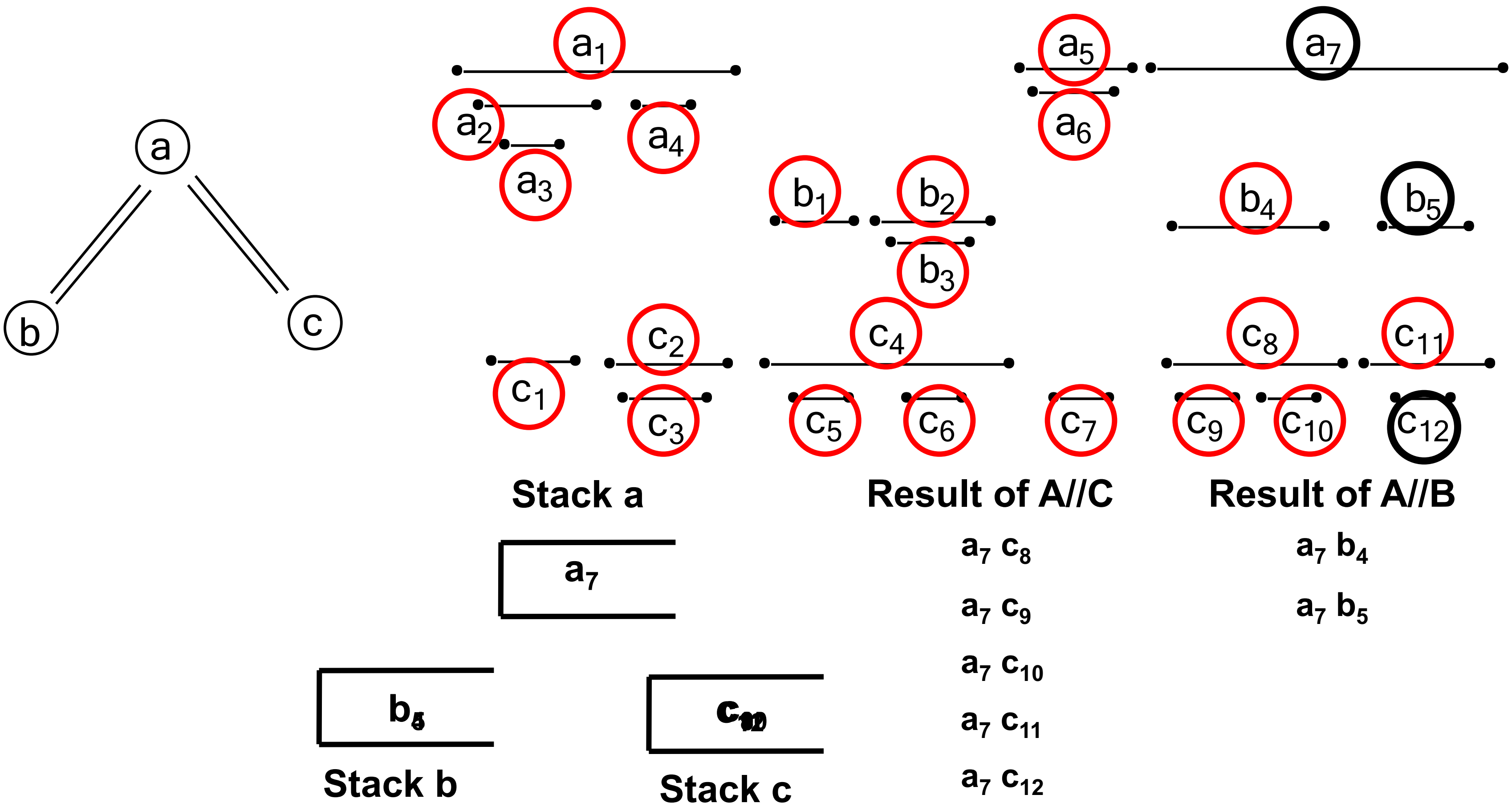
- Compute only partial solutions that are guaranteed to extend to a final solution (possible if twig contains only //). Specifically, when pushing e_q onto stack S_q, ensure that
  – e_q has a descendent $e_{q'}$ in each input list $T_{q'}$ where q' is a child of q
  – Each $e_{q'}$ recursively satisfies the above property
- Merge partial solutions to obtain all matches.

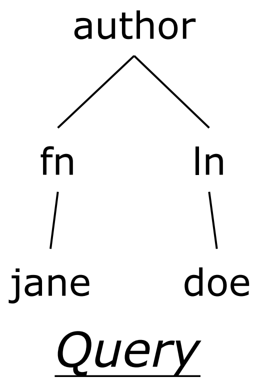
TwigStack



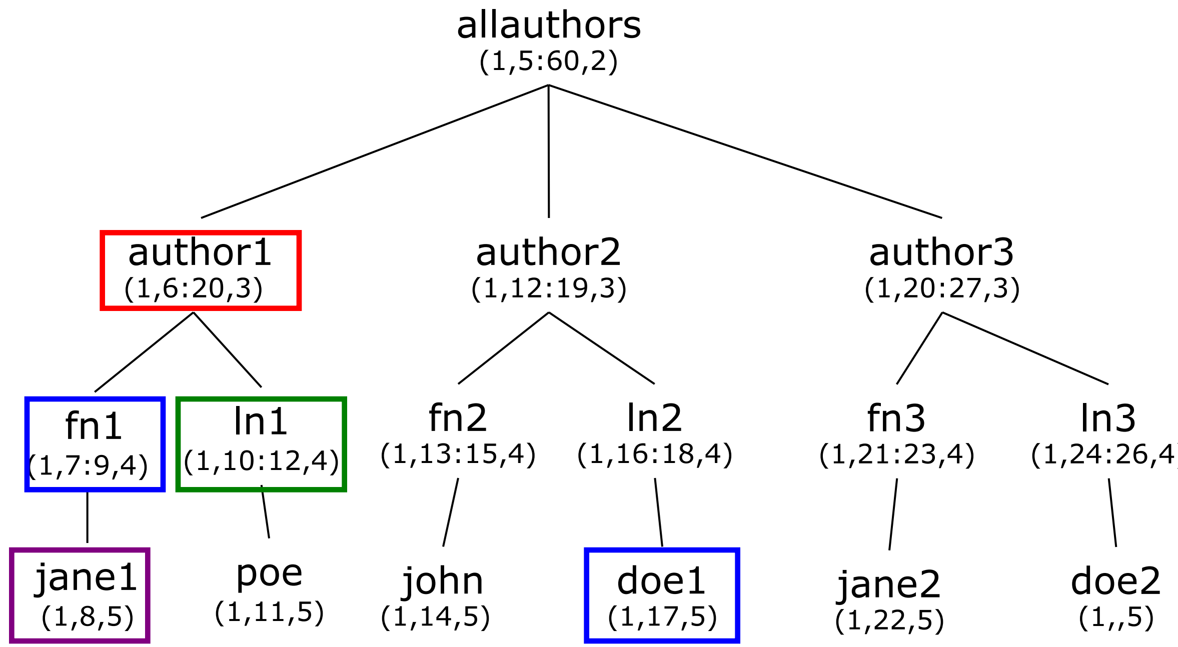
TwigStack

TwigStack


a
b
c
a1
a2
a3
a4
a5
a6
a7
b1
b2
b3
b4
b5
c1
c2
c3
c4
c5
c6
c7
c8
c9
c10
c11
c12
Stack a
Result of A//C
Result of A//B
a7
a7 c8
a7 c9
a7 c10
a7 c11
a7 c12
a7 b4
a7 b5
Stack b
Stack c



Query



Document

Streams

T_a : a1, a2, a3

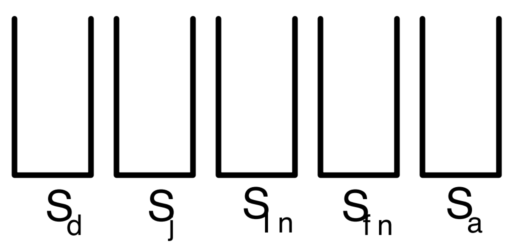
T_{fn}: fn1, fn2, fn3

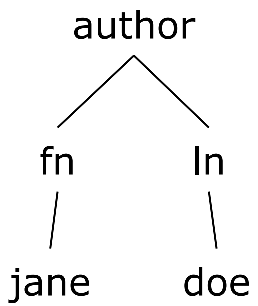
T_{ln}: ln1, ln2, ln3

T_j : j1, j2

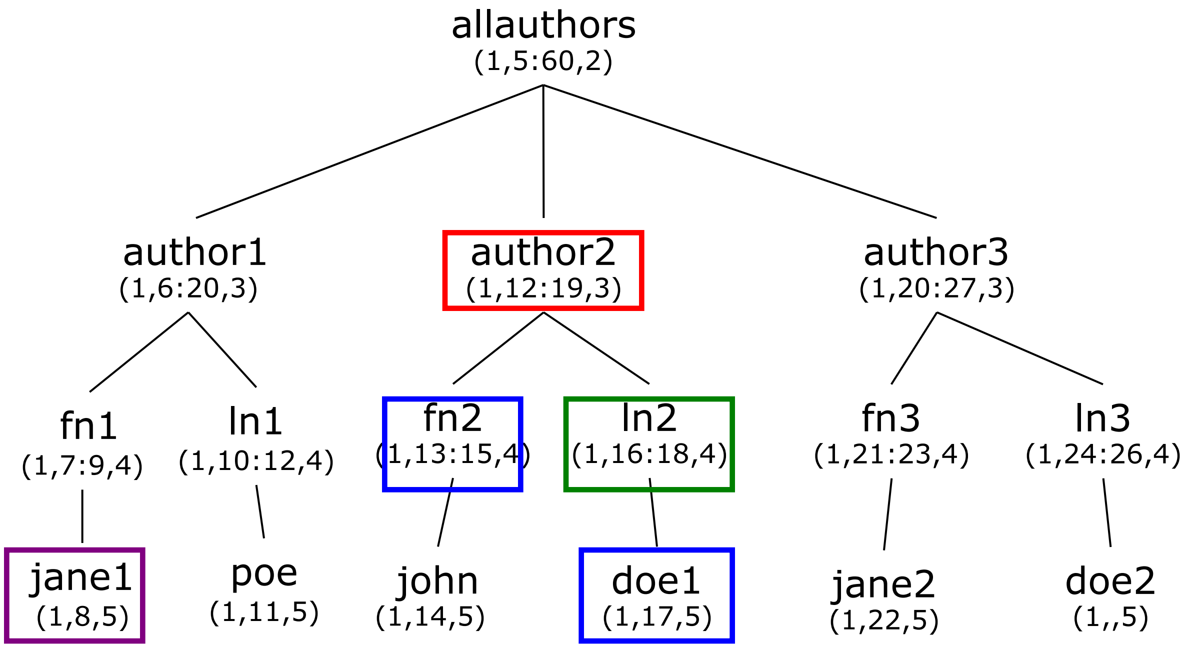
T_d : d1, d2

Stacks





Query



Document

Streams

T_a : a1, a2, a3

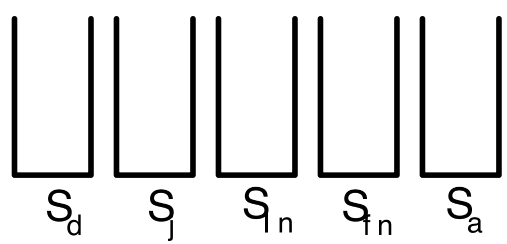
T_{fn}: fn1, fn2, fn3

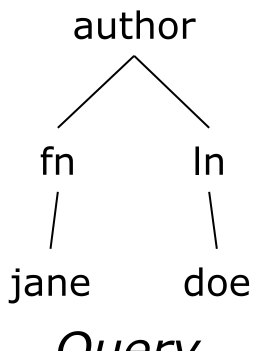
T_{ln}: ln1, ln2, ln3

T_j : j1, j2

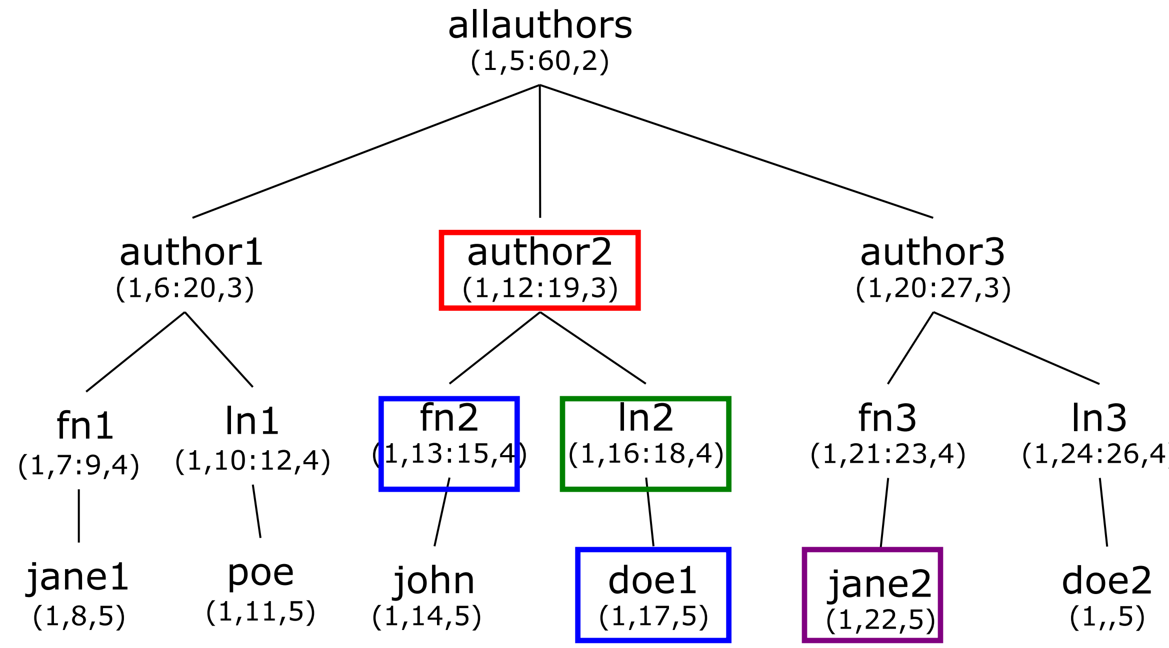
T_d : d1, d2

Stacks





Query



Document

Streams

T_a : a1, a2, a3

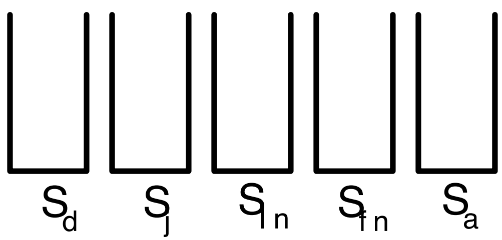
T_{fn}: fn1, fn2, fn3

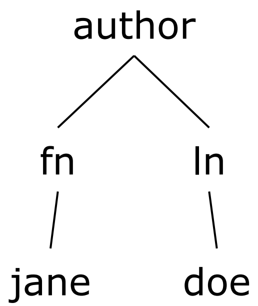
T_{ln}: ln1, ln2, ln3

T_j : j1, j2

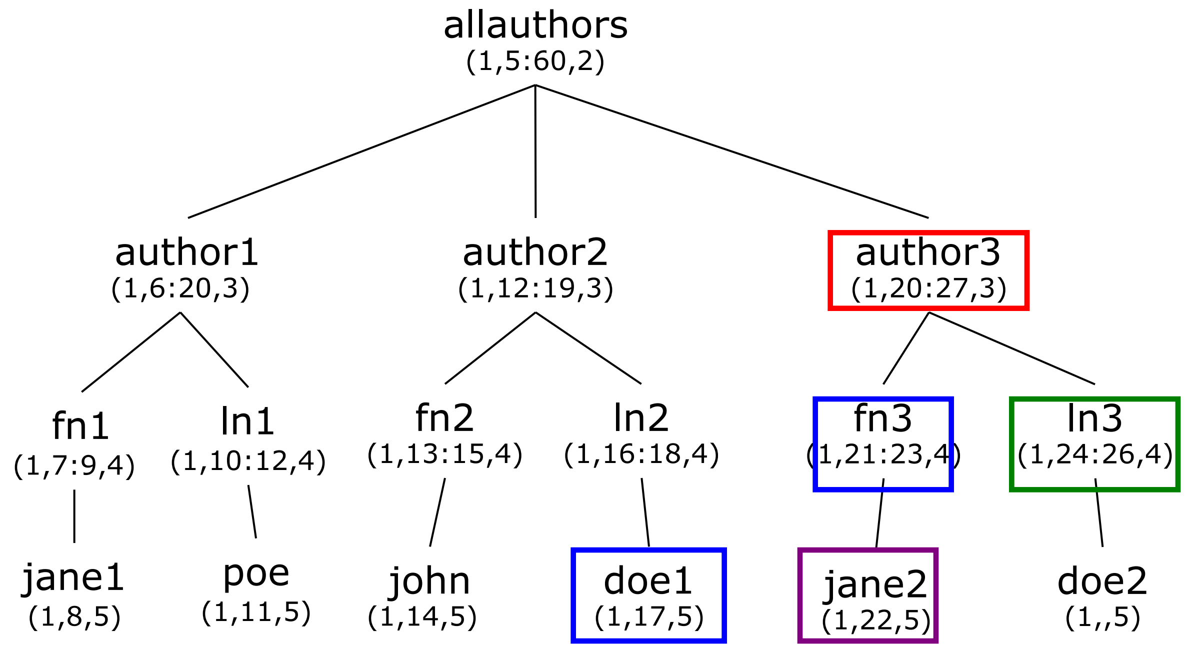
T_d : d1, d2

Stacks





Query



Document

Streams

T_a : a1, a2, $\boxed{\text{a3}}$

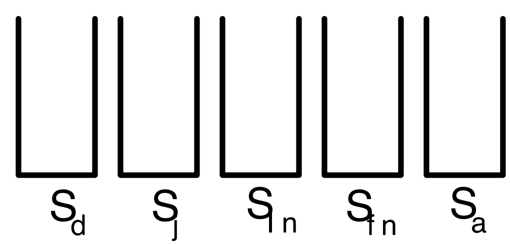
T_{fn}: fn1, fn2, $\boxed{\text{fn3}}$

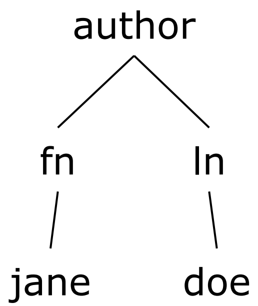
T_{ln}: ln1, ln2, $\boxed{\text{ln3}}$

T_j : j1, $\boxed{\text{j2}}$

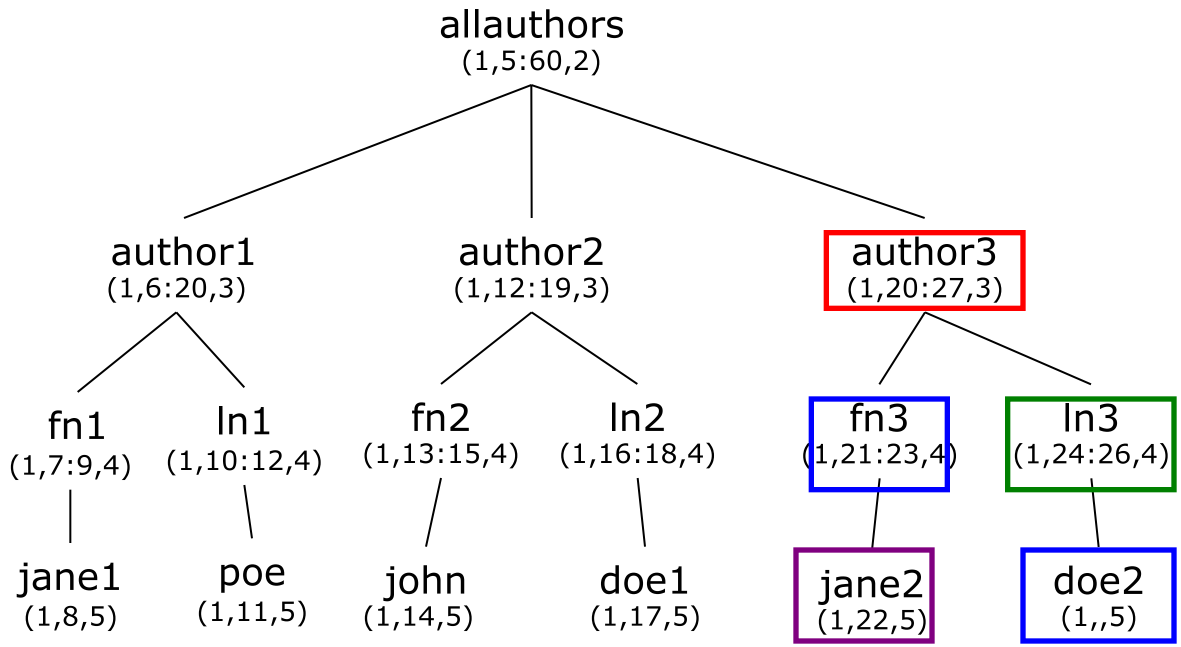
T_d : $\boxed{\text{d1,}}$ d2

Stacks





Query



Document

Streams

T_a : a1, a2, a3

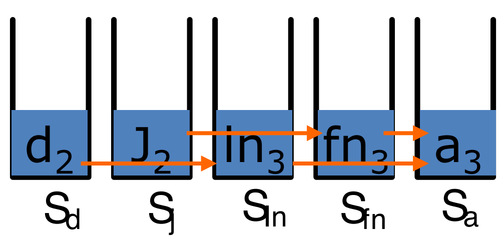
T_{fn}: fn1, fn2, fn3

T_{ln}: ln1, ln2, ln3

T_j : j1, j2

T_d : d1, d2

Stacks



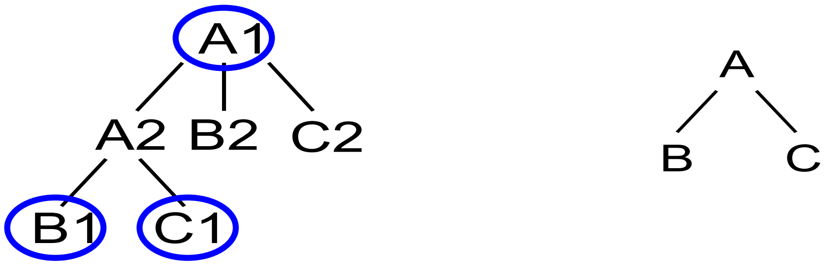
Path1: a3-fn3-j2

Path2: a3-ln3-d2

Merge (j2, fn3, d2, ln3, a3)

TwigStack Still Suboptimal for /

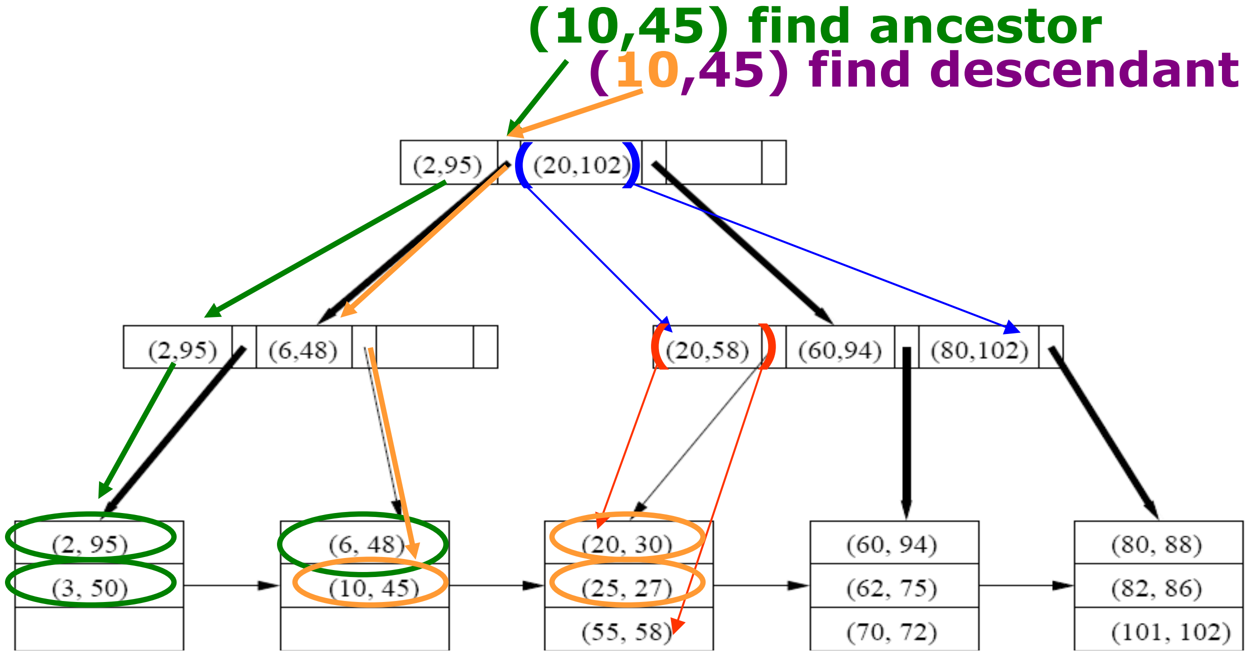
- Example:



- Desired result: (*A*1, *B*2, *C*2), (*A*2, *B*1, *C*1)
- Initial state: all three stacks empty; ready to push one of *A*1, *B*1, *C*1 onto a stack
- If we want to ensure that non-contributing nodes are never pushed onto the stack, then
  - Cannot decide on *A*1 unless we see *B*2 and *C*2
  - Cannot decide on *B*1 or *C*1 unless we see A2

XB-Tree

- XB-Trees like R-tree and B$^+$-trees
  - Parent node interval includes child node intervals
- TwigStack can be adapted to use XB-Trees with minimal changes.



More Twig Join Algorithms

- TSGeneric [Jiang, VLDB'03]
  - Indexing each stream and use them for skipping
- Prefix Path Streaming [Chen, DEXA'04]
  - Elements with the same root-to-node path are grouped together
- TwigStackList [Lu, CIKM'04]
  - Prefetching elements in cache (list) attached to twig query
- iTwigJoin [Chen, SIGMOD'05]
  - Exploiting Tag+Level and Prefix Path Streaming
- TJFast [Lu, VLDB'05]
  - Exploitation of extended Dewey labeling to identify element by node label
- Twig^{2}Stack [Chen, VLDB'06]
  - Hierarchical stack encoding for generalized tree pattern queries