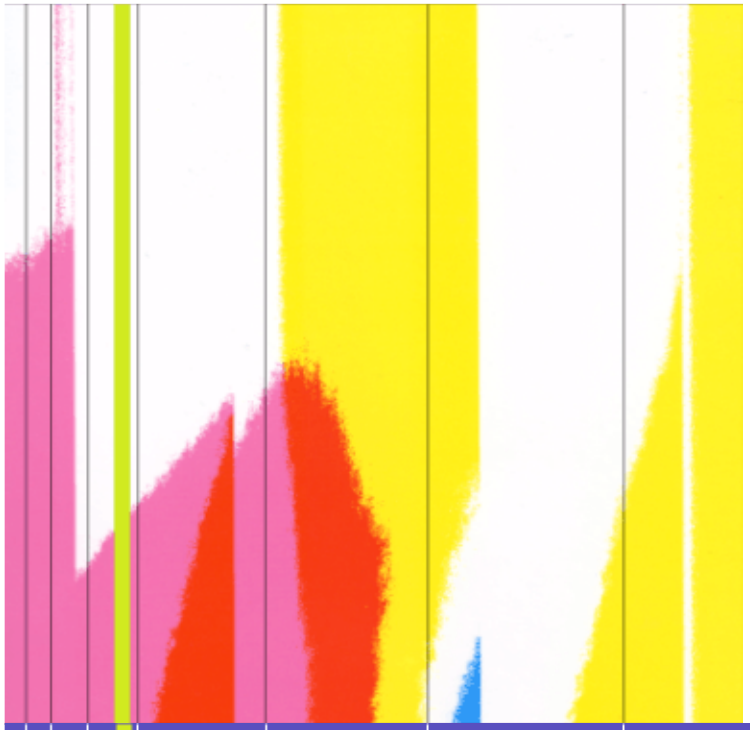


XML query processing: storage and query model interplay

INSTITUT NATIONAL
DE RECHERCHE
EN INFORMATIQUE
ET EN AUTOMATIQUE

INRIA

Ioana Manolescu-Goujot

INRIA Futurs - LRI - PCRI, France

Gemo group

Context: XML query processing

XML assumed known

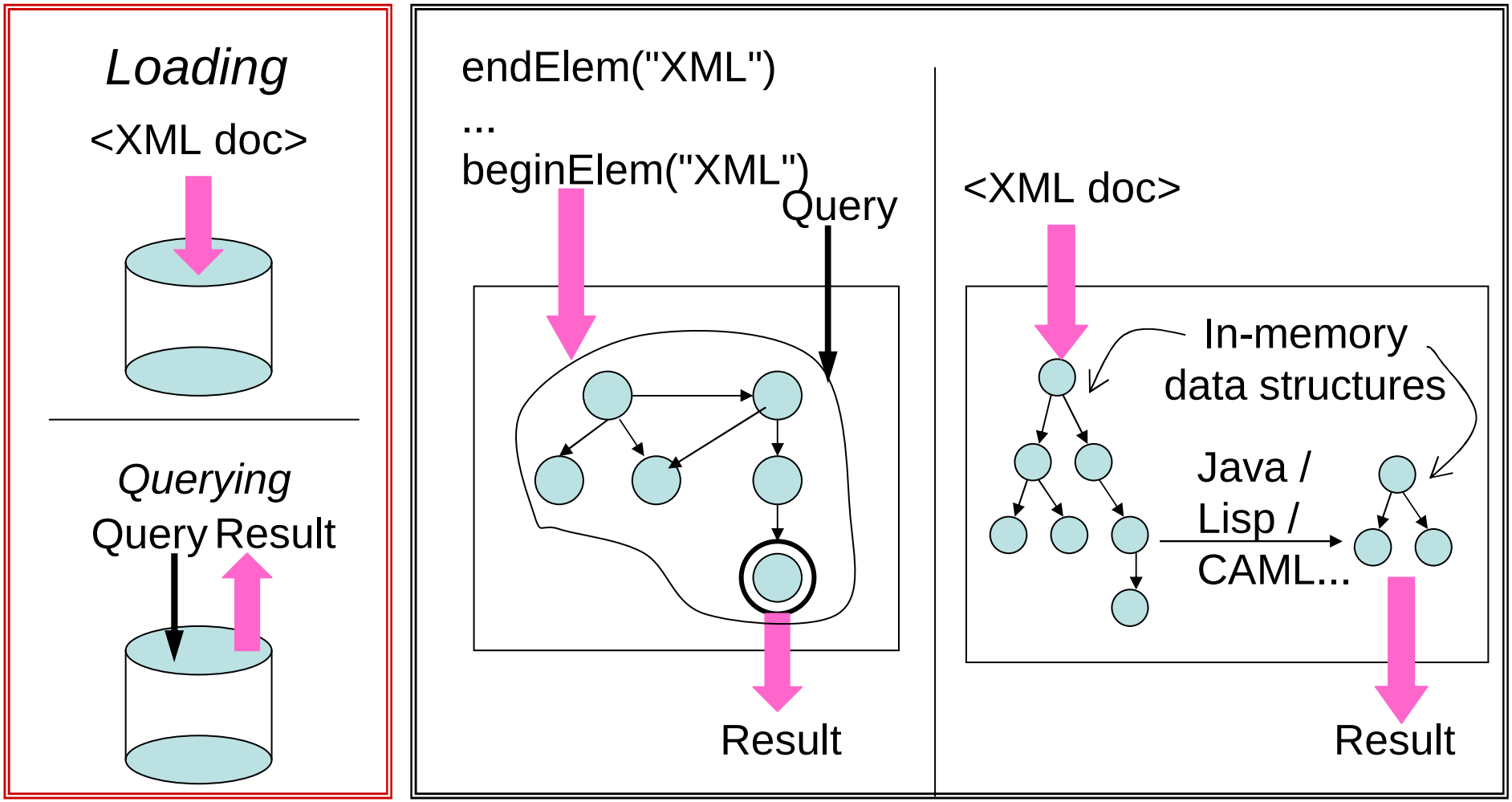
XML querying: not as simple as it seems

- XQuery, XQuery processing model

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



XML query processing scenarios (1/2)

"Persistent store"

Logging / archiving an ongoing activity

- Clients, orders, products... Dana used purchase orders :-)
- 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

- Data marshalling is a pain
- In real life, there are not just databases

Our topic

Disprove:

"XML research is magmatic"

Disprove:

"Nothing has been/can be done; despair and die"

Disprove:

"Everybody has his own storage scheme": not true, some people have others'

Classify what has been done to take care of the details under the *very*, *very*, *very*, high level

With a database view

Our topic (really): XML query processing on persistent stores

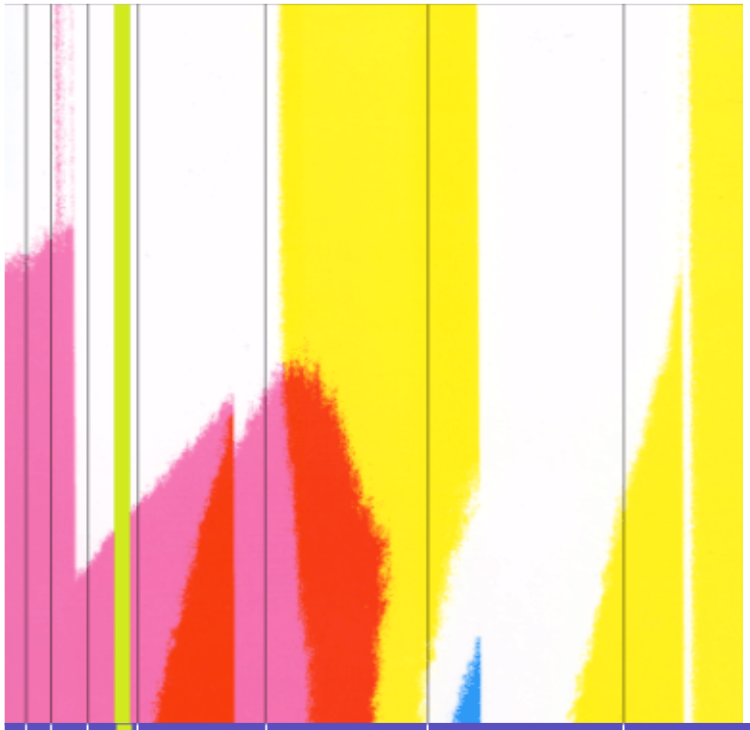
Preliminaries

- What is there to store ?
- What do queries ask for ?
- The long and winding road between the two

Storage schemes for XML

- What is stored where ?
- What is not stored ?
- How would queries be answered ?

Wrap-up and missing things



Preliminaries

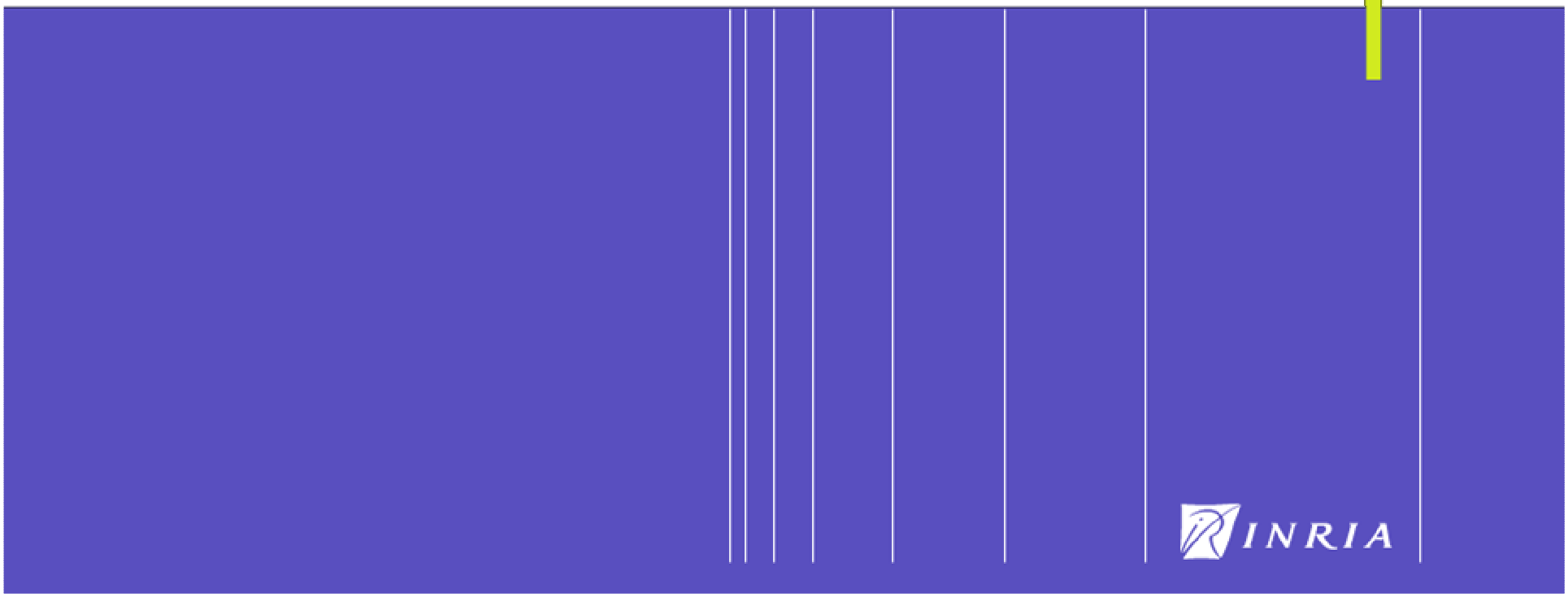
What is there to store in XML data ?
What do queries ask for ?
Where do storage systems stand ?

INSTITUT NATIONAL
DE RECHERCHE
EN INFORMATIQUE
ET EN AUTOMATIQUE

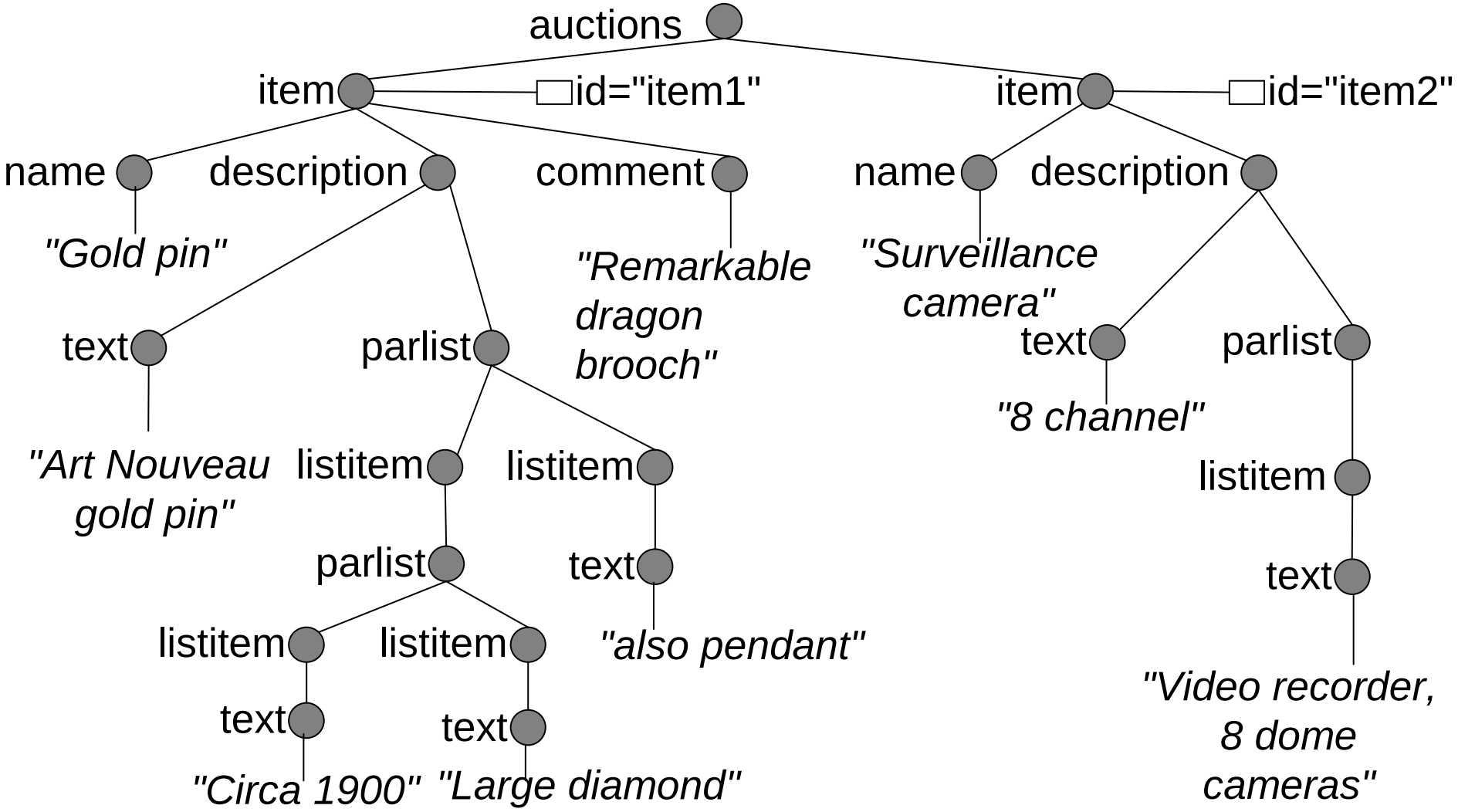
INRIA

Preliminaries:

What is there to store in XML data ?



Sample XML document



Nodes and node identity

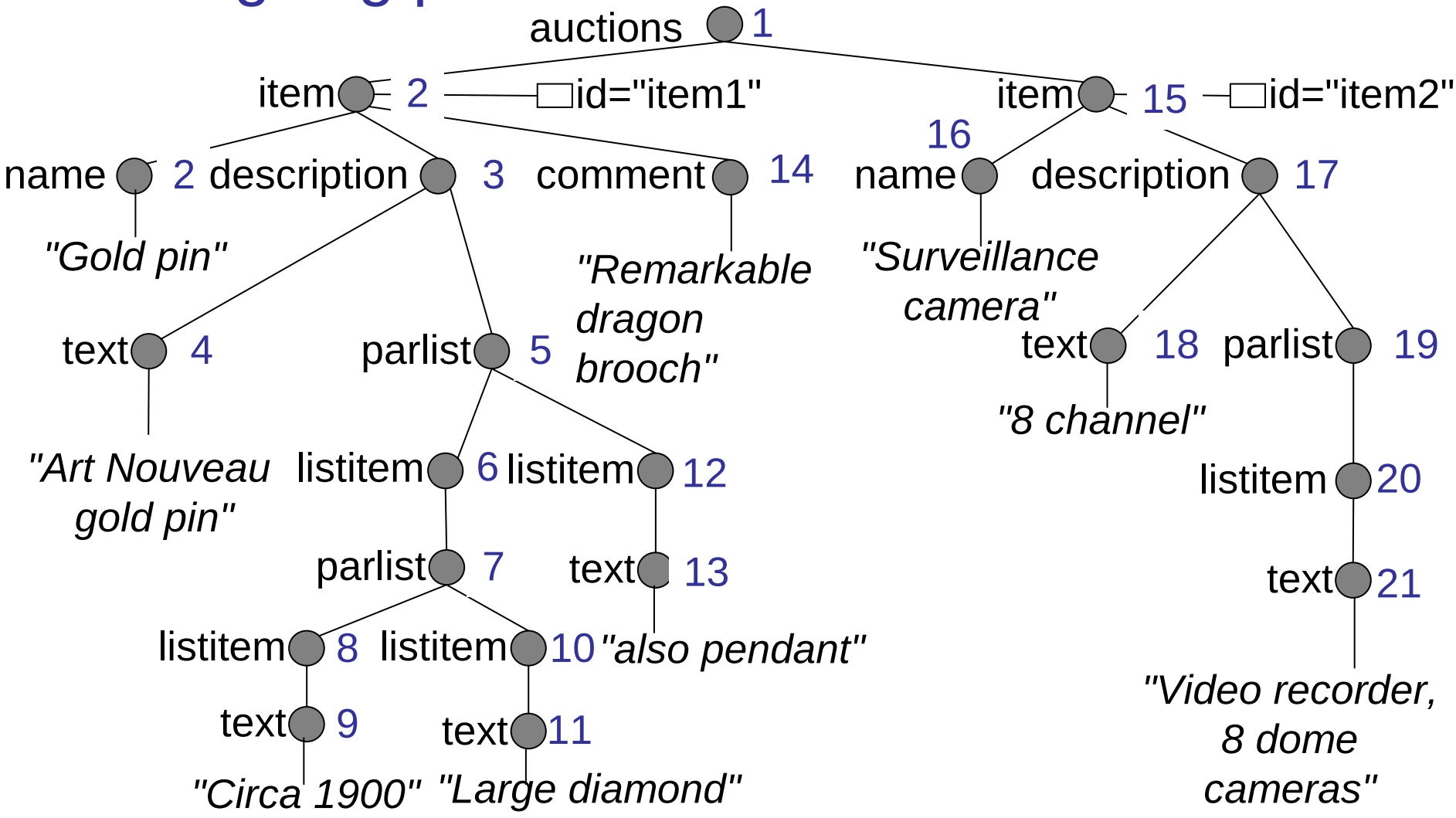
7 types of nodes: document, **element**, **attribute**, **text**, *namespace, processing instruction, comment [XQDM]*

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

(ElemID, attr name) determine attr. value ⇒
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

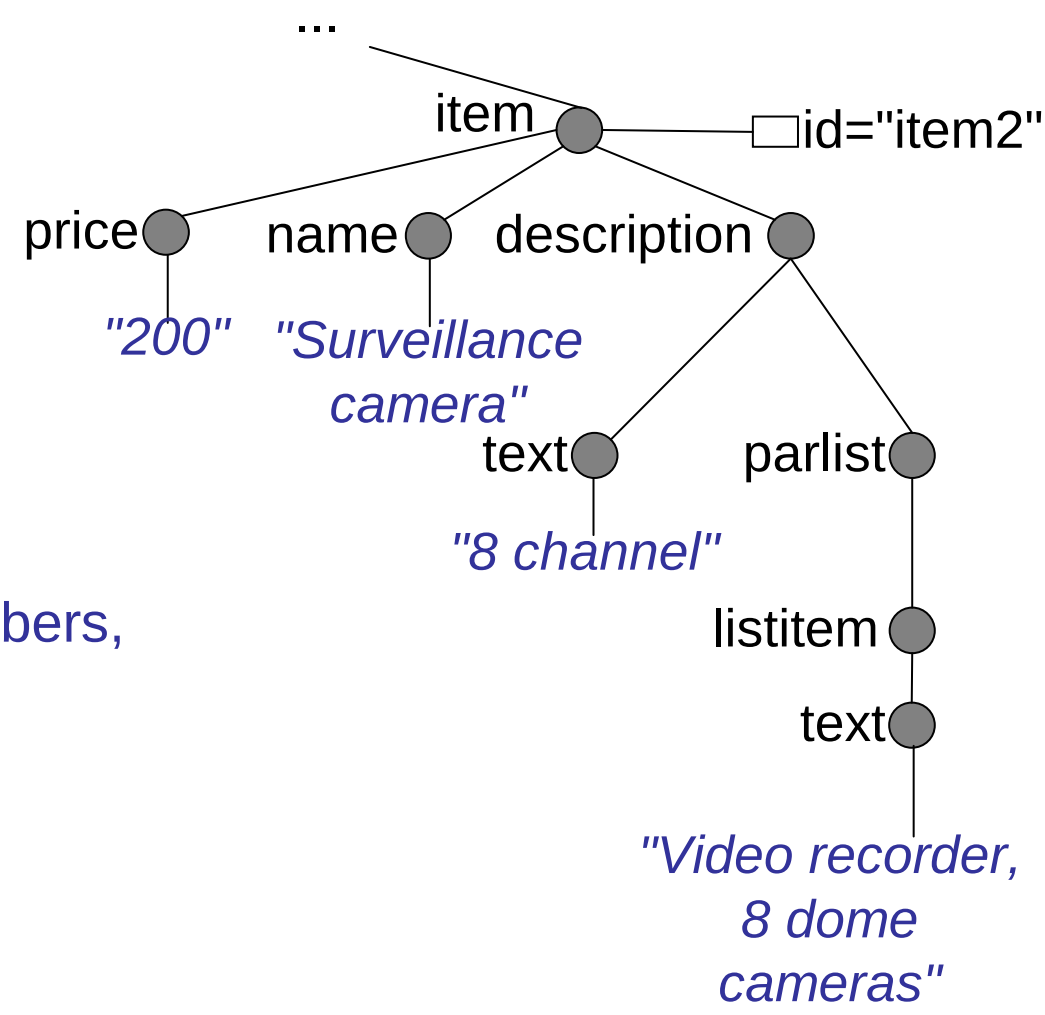
Text nodes

Level 0: bunch of strings

Level 1: strings, numbers, booleans

Level 2: bags of words, numbers, boolean

This is still a simplification

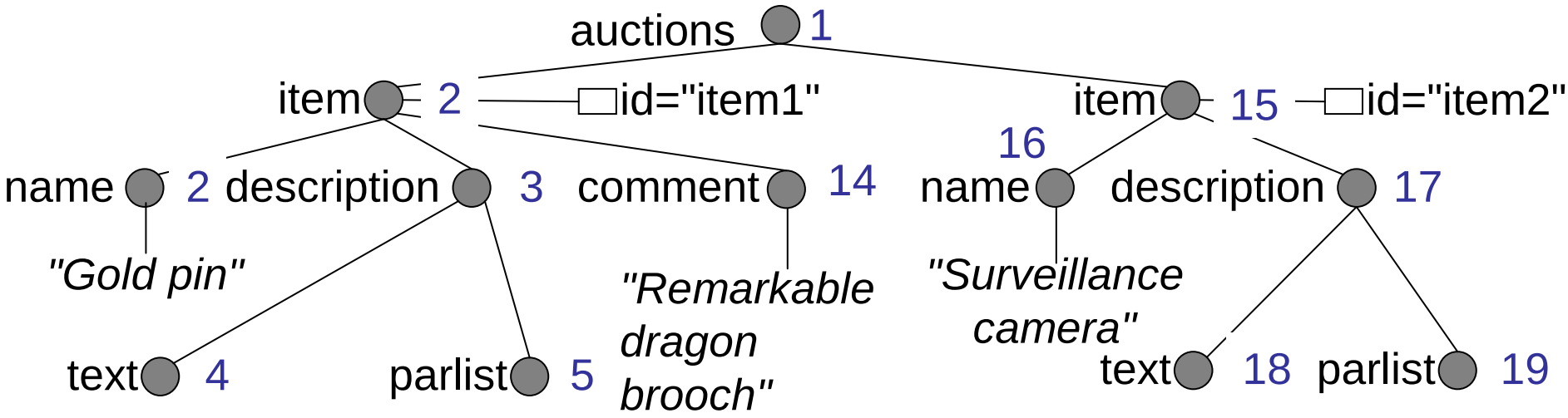


Document structure: relationships among nodes

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 15

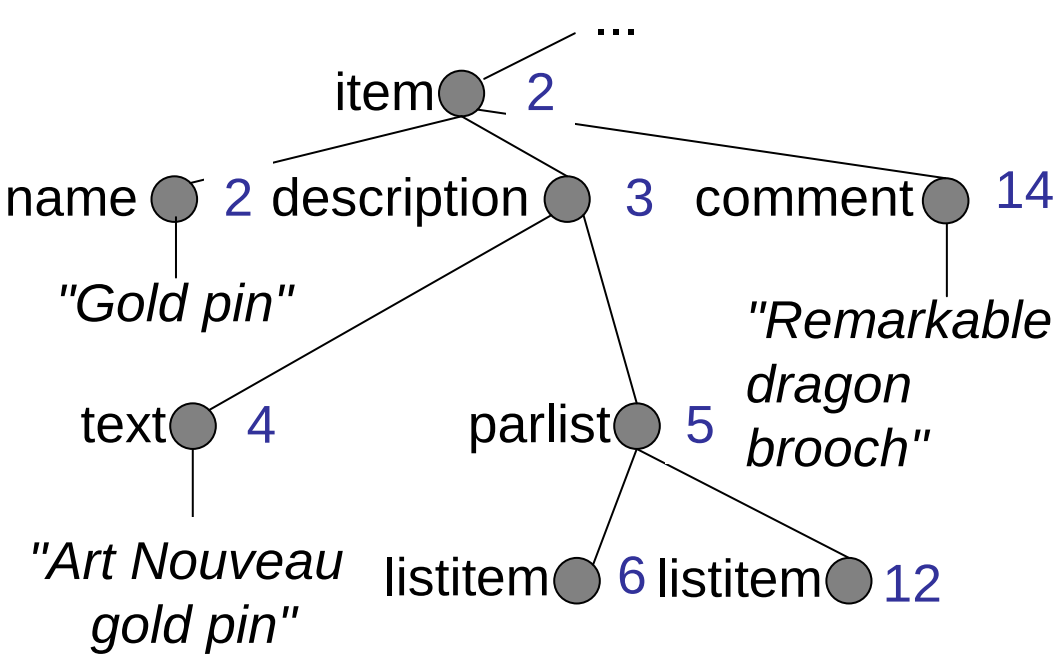
Element 2 has the attribute id="item1"

Element 2 has the text child "Gold pin"

Document order

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

Document structure: node names

Element and attribute names must be stored

Can be considered "document schema"...

"Semistructured data contains its own schema"
"XML is semistructured data"

"XML contains its own schema"

Not any longer [XSch,XFS]

Document structure, revisited

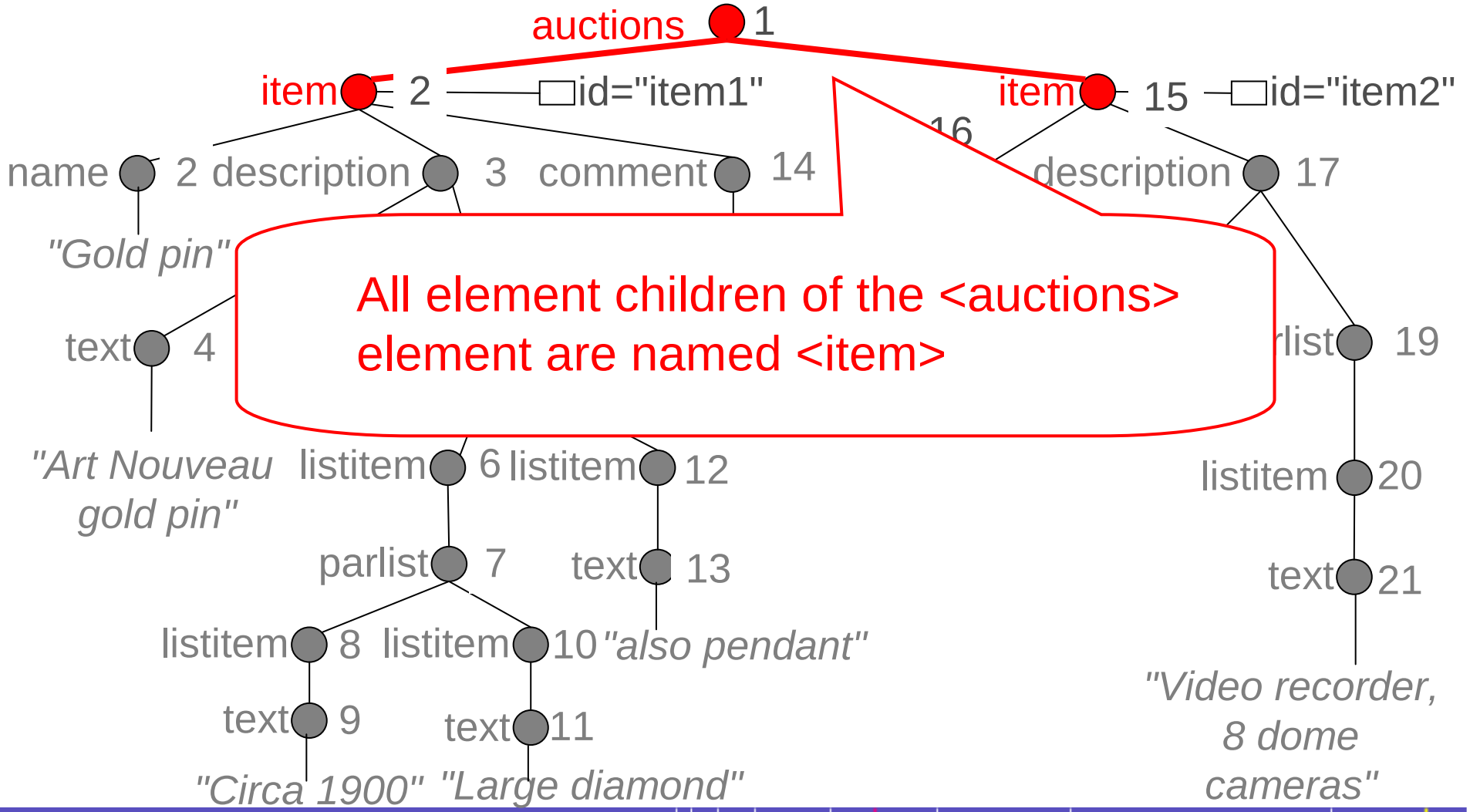
Structure =

- *Invariants* ("constraints", "schema", "DTD") +
- *Instances* (particular instantiations of a degree of freedom left by the invariants)

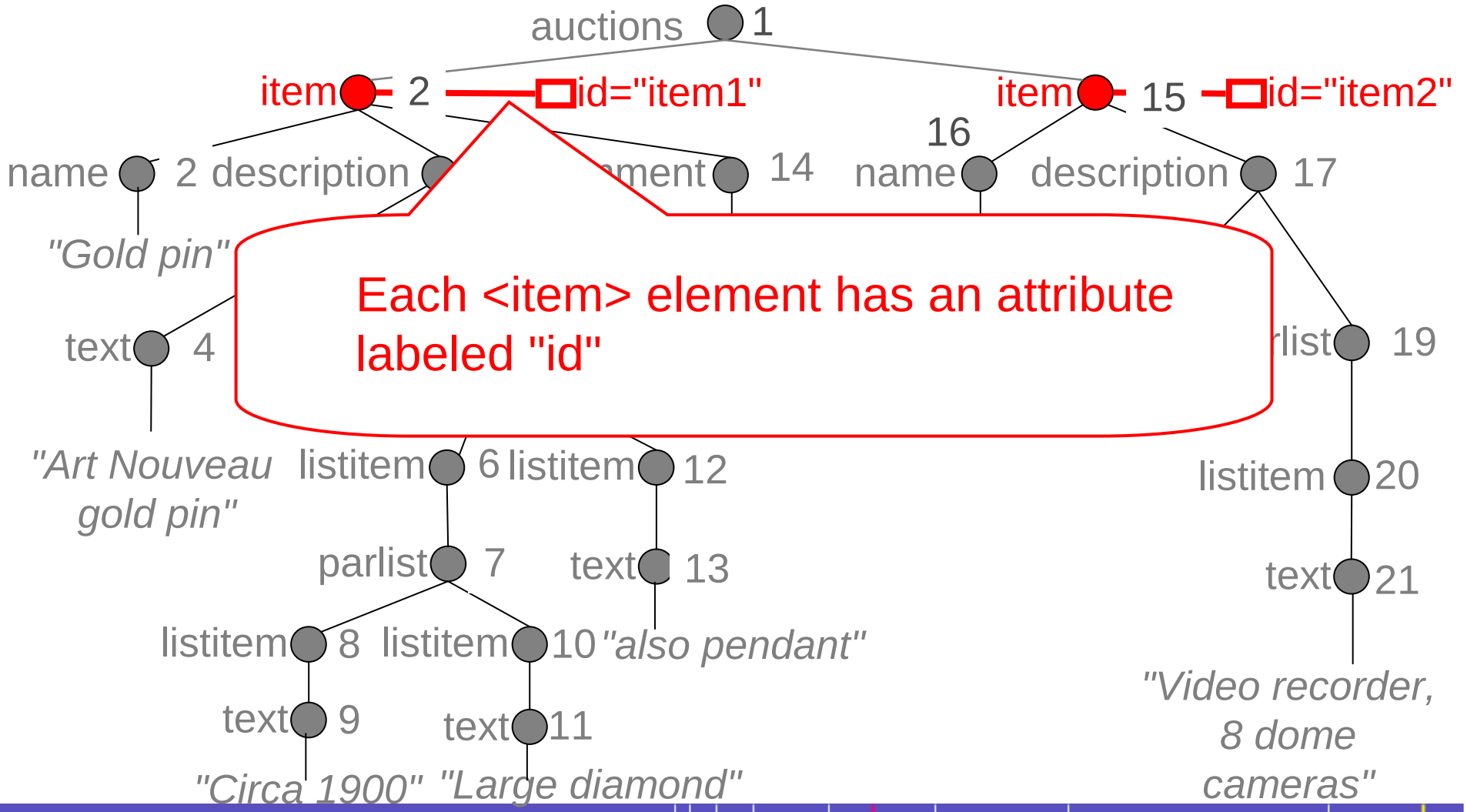
Many formalisms for specifying the invariants

- DTDs, XML Schemas
- Structural summaries [GW97]
- Graph schemas [FS98]
- Richer constraint languages

Invariants in XML document structure

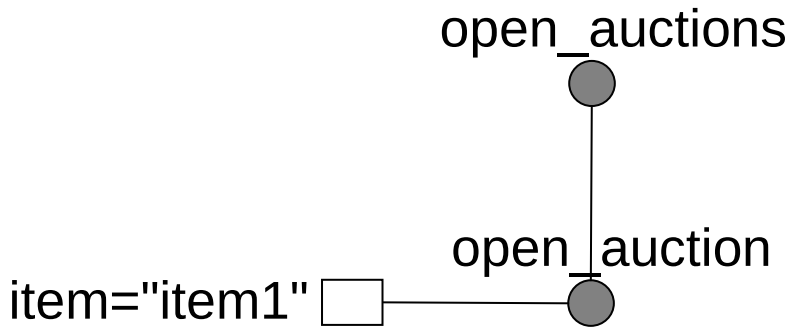


Invariants in XML document structure

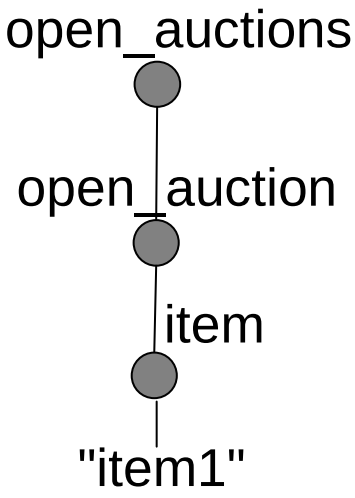


Invariants in document structure

"All itemID values in a
document are unique."

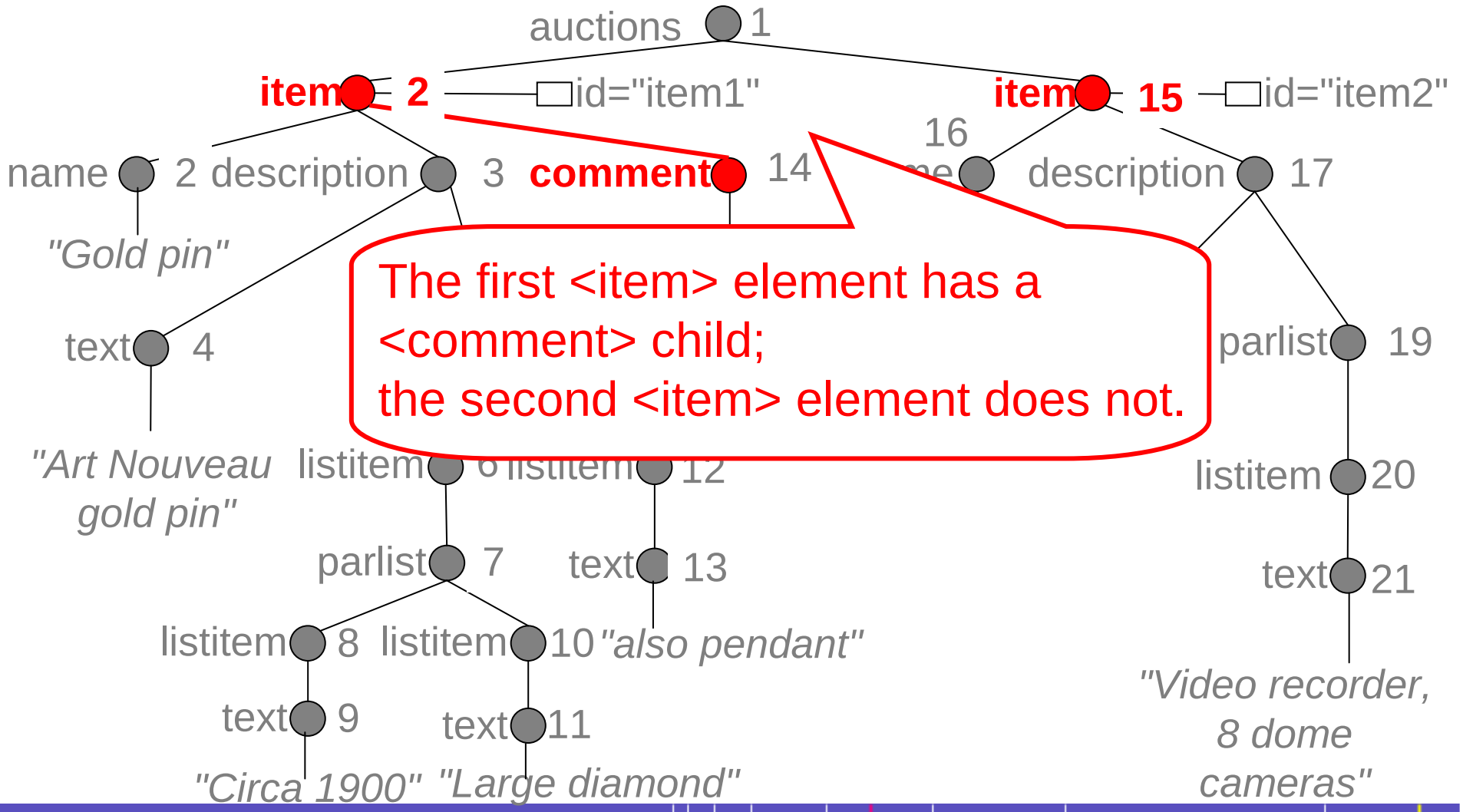


"For each value of the
item attribute of an<open_auction> element,
there exists an <item> element
having the itemID attribute with the same value"

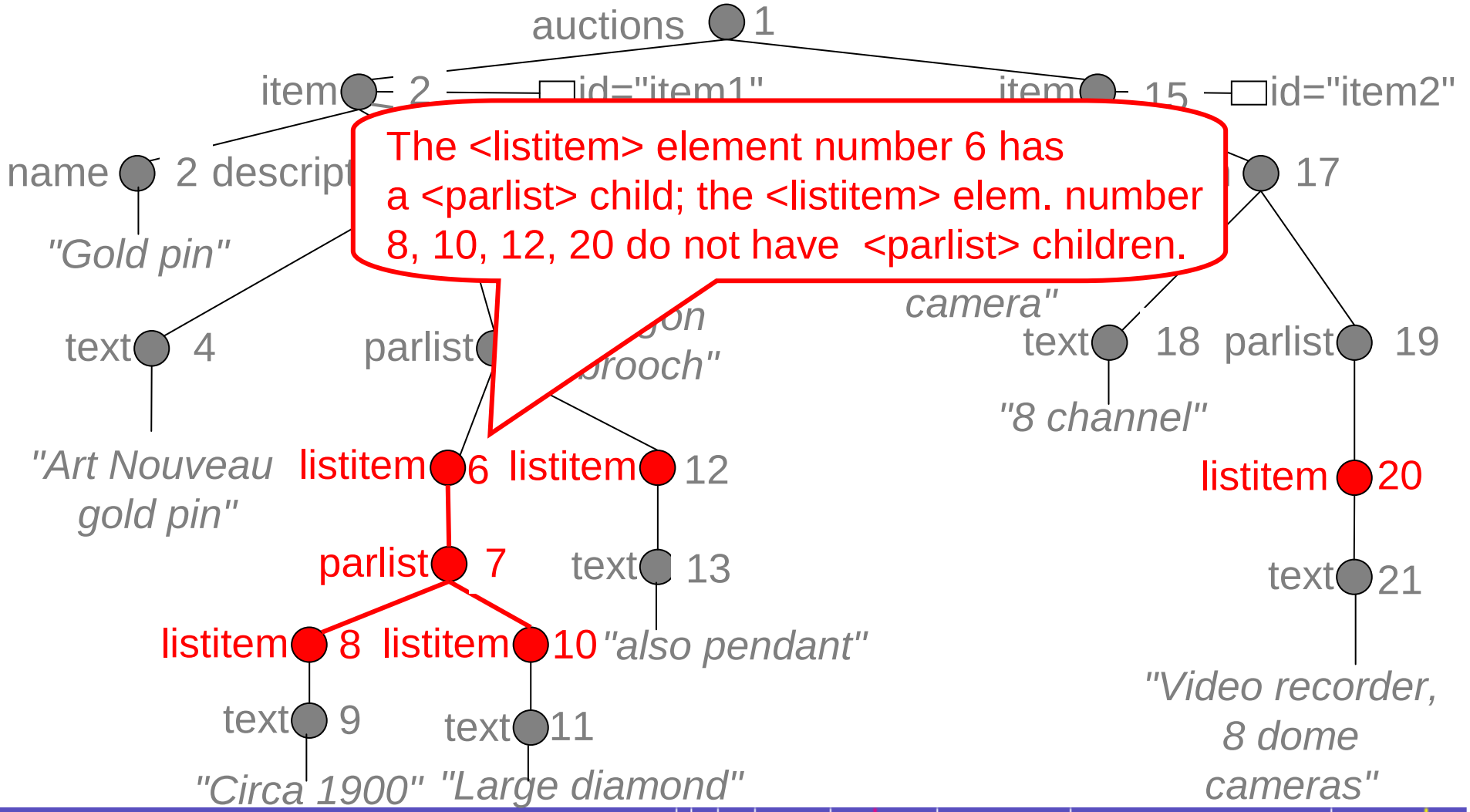


Change the "item" attribute into a subelement...

Instance structure information



Instance structure information



Wrap-up: what should we store

Node identity and order

(Typed) data values

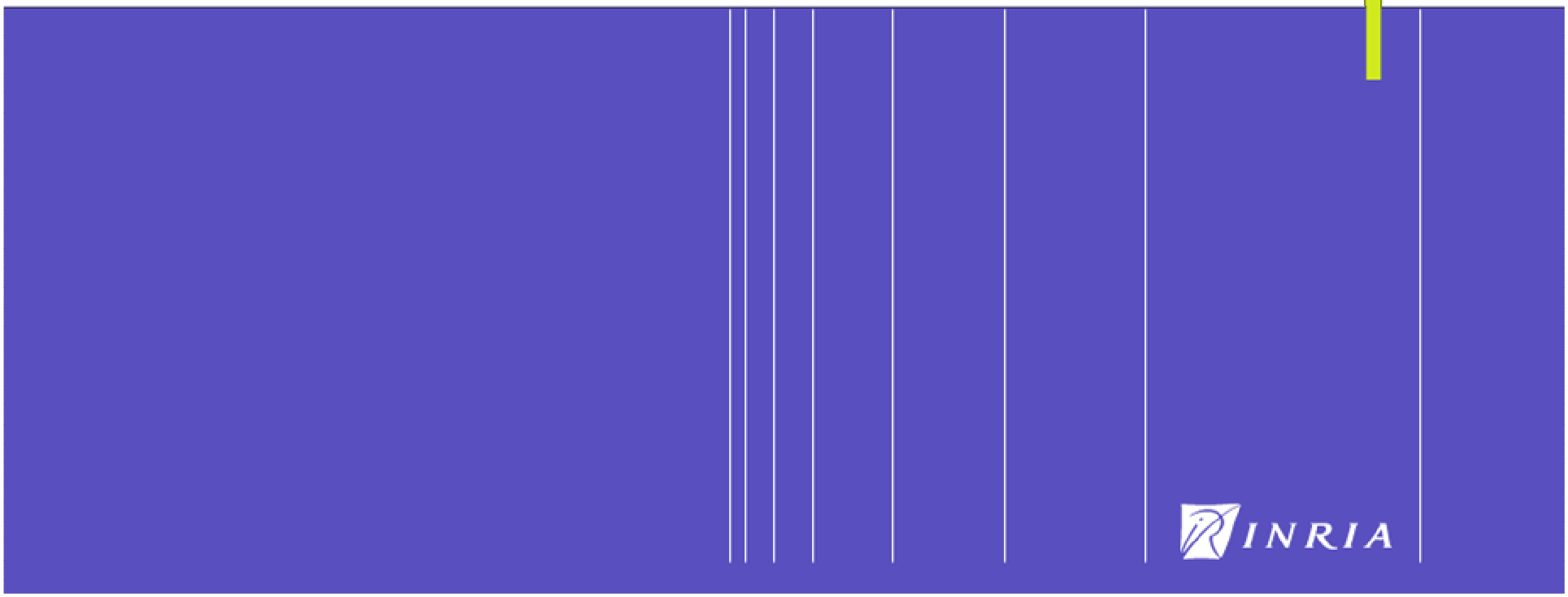
Document structure = invariants + particular instances

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

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

Complex constraints require special care for updates

Preliminaries:
what do queries ask for ?



One (popular) example: XQuery

Several W3C documents help define this

XQuery Data Model [XQDM]:

- what information does the XML document contain, with respect to the XQuery processor

XQuery Function and Operators [XQFO]:

- which operations are permitted in XQuery on various data type, and what do they mean

XQuery Formal Semantics [XQFS]:

- what should be the answer to a given query
  (what it means)

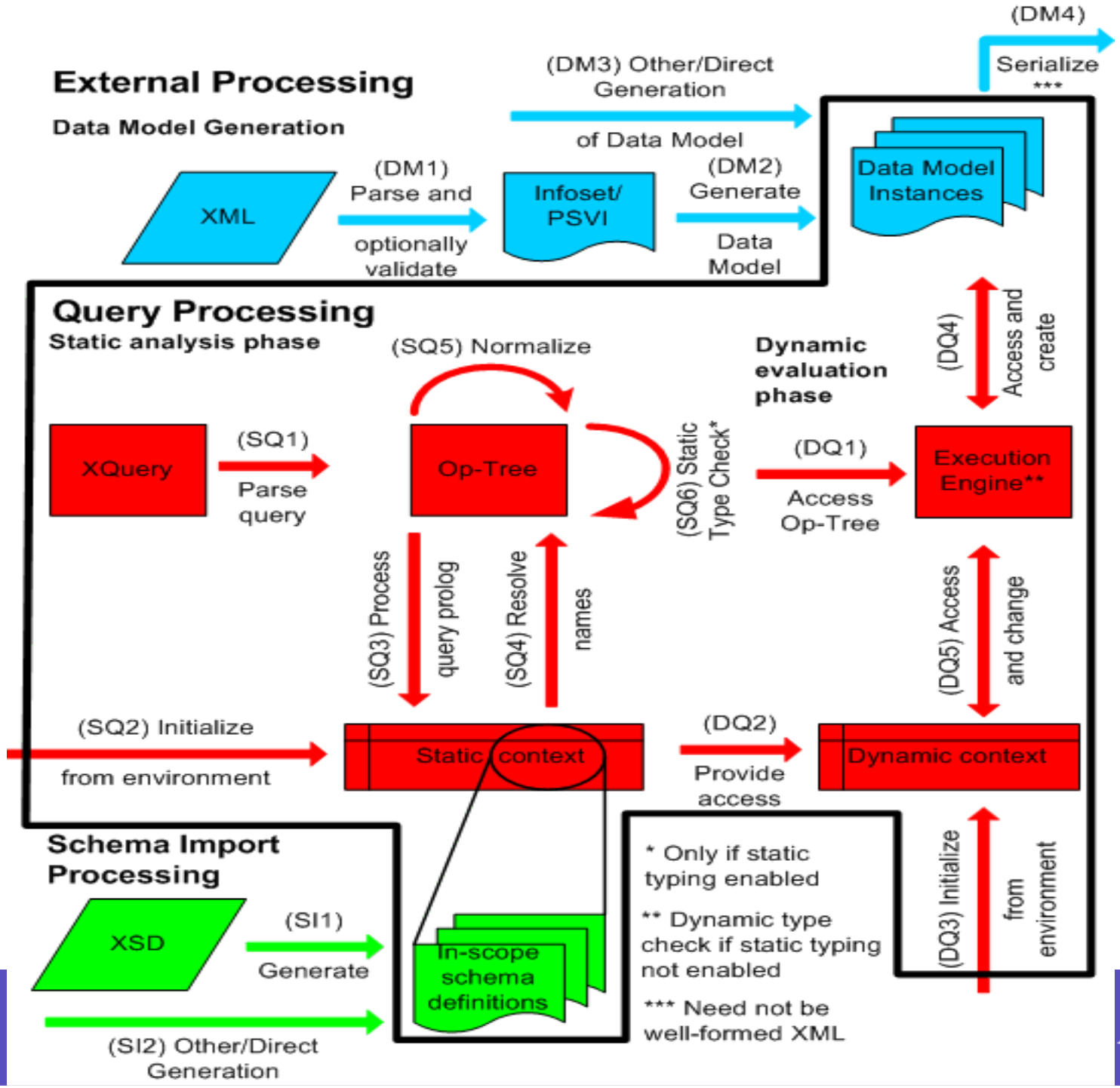
XQuery Formal Semantics

Every XQuery expression can be equivalently reduced to an expression in a simpler language, XQuery Core

To an XQuery Core expression is associated

- A type
  - More or less precise, depending on the availability of type information on the input
- A value

The whole picture: XQuery Processing Model


External Processing
Data Model Generation
XML
(DM1) Parse and optionally validate
Infoset/ PSVI
(DM2) Generate Data Model
(DM3) Other/Direct Generation of Data Model
Data Model Instances
(DM4) Serialize ***
Query Processing
Static analysis phase
XQuery
(SQ1) Parse query
Op-Tree
(SQ5) Normalize
(SQ6) Static Type Check*
Dynamic evaluation phase
(DQ1) Access Op-Tree
Execution Engine**
(DQ4) Access and create
(SQ3) Process query prolog
(SQ4) Resolve names
(DQ5) Access and change
(SQ2) Initialize from environment
Static context
(DQ2) Provide access
Dynamic context
Schema Import Processing
XSD
(SI1) Generate
In-scope schema definitions
(SI2) Other/Direct Generation
* Only if static typing enabled
** Dynamic type check if static typing not enabled
*** Need not be well-formed XML
(DQ3) Initialize from environment

The XQuery processing model

"This processing model is not intended to describe an *actual implementation*, although a naive implementation *might* be based upon it.

It does not prescribe an implementation technique, but *any implementation should produce the same results as obtained by following this processing model* and applying the rest of the Formal Semantics specification."

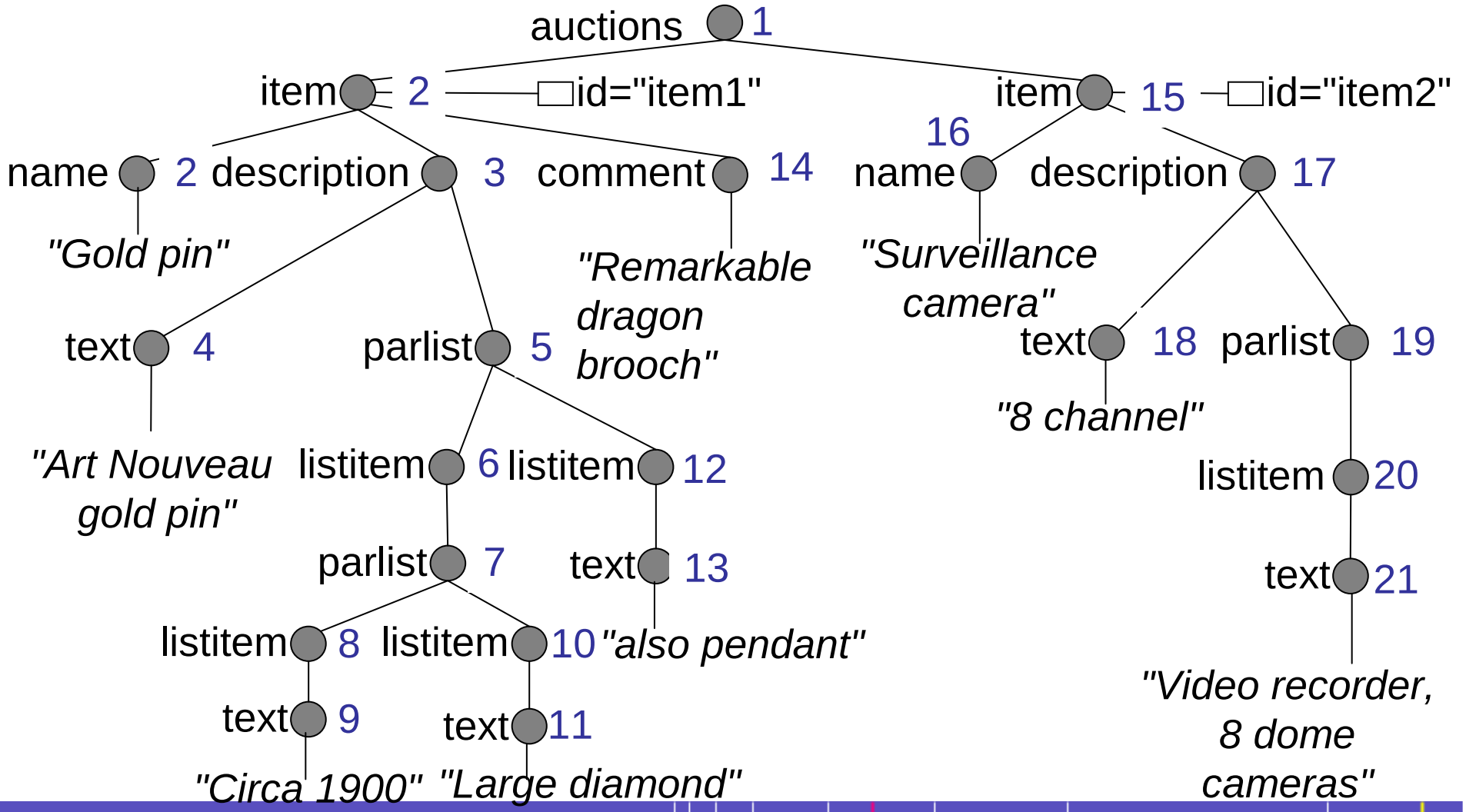
Essential part of XQuery: path expressions

Navigation axes:

- child, parent
- descendent, ancestor,
- descendent-or-self, ancestor-or-self
- attribute,
- self,
- following, preceding,
- following-sibling, preceding-sibling.

Path predicates: order-related, or path branches

Sample path expressions:
//item[@id="item1"]//parlist//listitem/text/text()
//item/description[parlist/listitem/text]//listitem[3]



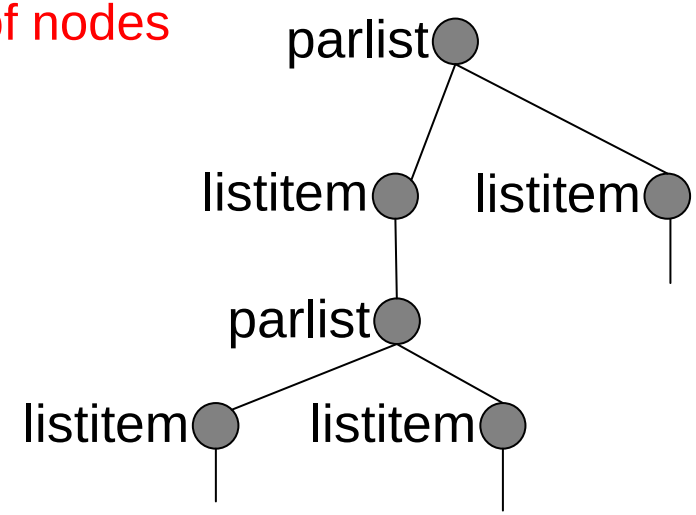
A few things you should know about XQuery

Path expressions result in duplicate-free lists of nodes
E.g. //parlist//listitem

Lots of existential semantics

- [1, 2] = [2, 3] true
- [2, 3] = [3, 4] true
- [1, 2] = [3, 4] false
- [1, 2] != [3, 4] true

Constructed elements must have fresh identity



A few things you should know about XQuery

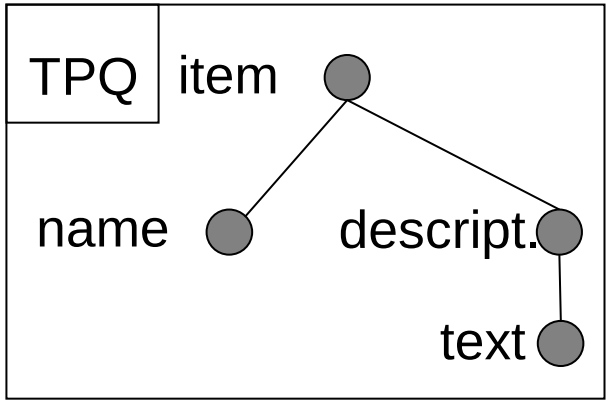
It is more than tree pattern queries

How do we write TPQ in XQuery ?

```
for $i in //item
    $n in $i/name
    $d in $i/description, $t in $i/text
return <res> {$t} </res>
```

```
for $i in //item, $t in $i/description/text
    return $i/name
```

```
for $i in //item[name]
return <res> for $d in $i//description
                return <a> $d/text </a> </a>
```



A few other things about XQuery

The semantics of *x op y* is defined by a large switch on the types associated to x and y

There are many atomic types for time durations, moments in time (with timezone) etc.

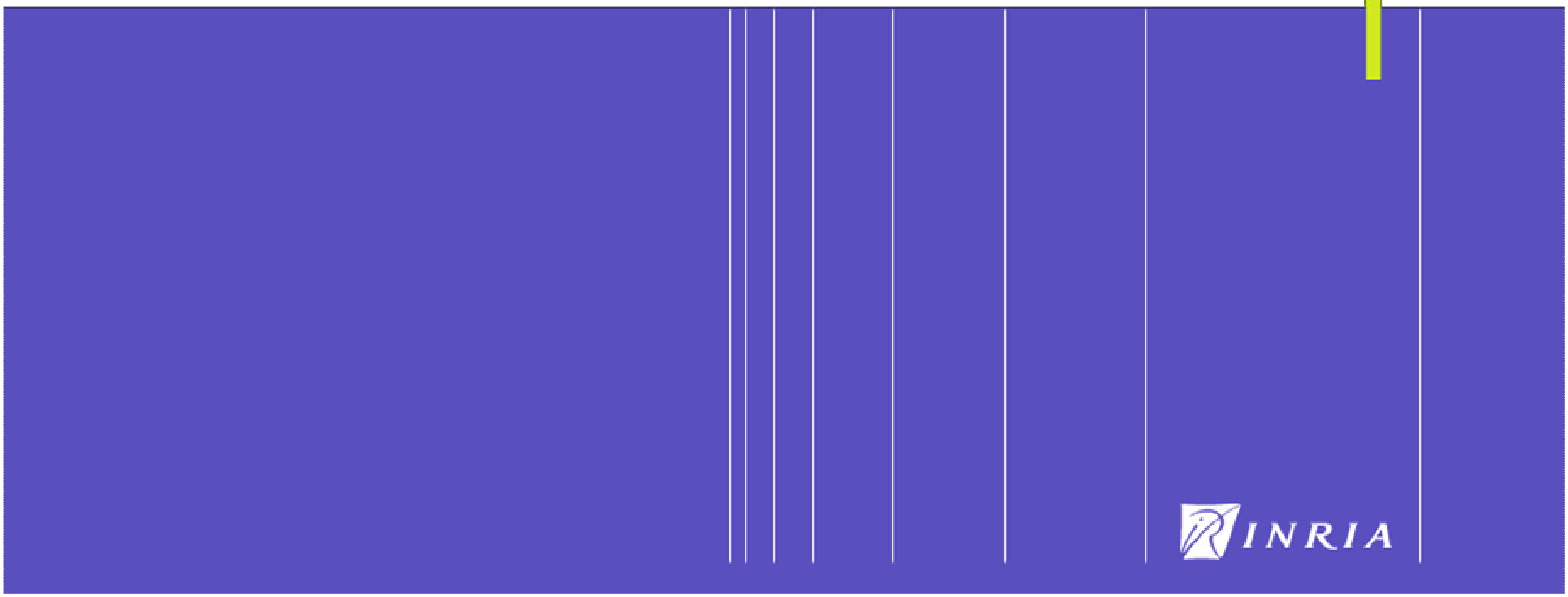
Equality on atomic numeric types may not be transitive...

These should not stop anyone from doing XQuery research

Good to have a viewpoint on *what is supported* from the processing model, and *what to do with the rest*

Preliminaries:

Where do storage systems stand wrt the XQuery processing model



Where (most) persistent storage systems stand

Ignore comments, processing instructions, entities

Simplify the set of atomic types

- "Error ! We do not support timezones yet."

Some focus on the *child* and *descendent* axes only

Keep little trace of schemas and complex types

- Common assumption: support for these can be plugged in the front-end

Focus on: storing *simple values*, and *ordered data trees*.

Where (most) persistent storage systems stand

Provide selective data access

Provide set-at-a-time execution primitives,
for a language whose formal semantics
is defined tuple-at-a-time (recall: this is allowed by FS !)

Advantage: performance

- Selective data access
- Set-at-a-time processing allows for better disk access locality
  - Hash join vs. nested loops join
- Naive navigation-based tuple-at-a-time path query processors have complexity *exponential in the size of the query* [GKP02]; set-at-a-time scales better.

Where (most) persistent storage systems stand

Provide set-at-a-time execution primitives,
for a language whose formal semantics
is defined tuple-at-a-time (recall: this is allowed by FS !)

Disadvantages:

- Must make explicit effort to *prove correctness*
- Some features may not be doable this way
- But they do not seem to be the essential ones

Towards the end of the religious war

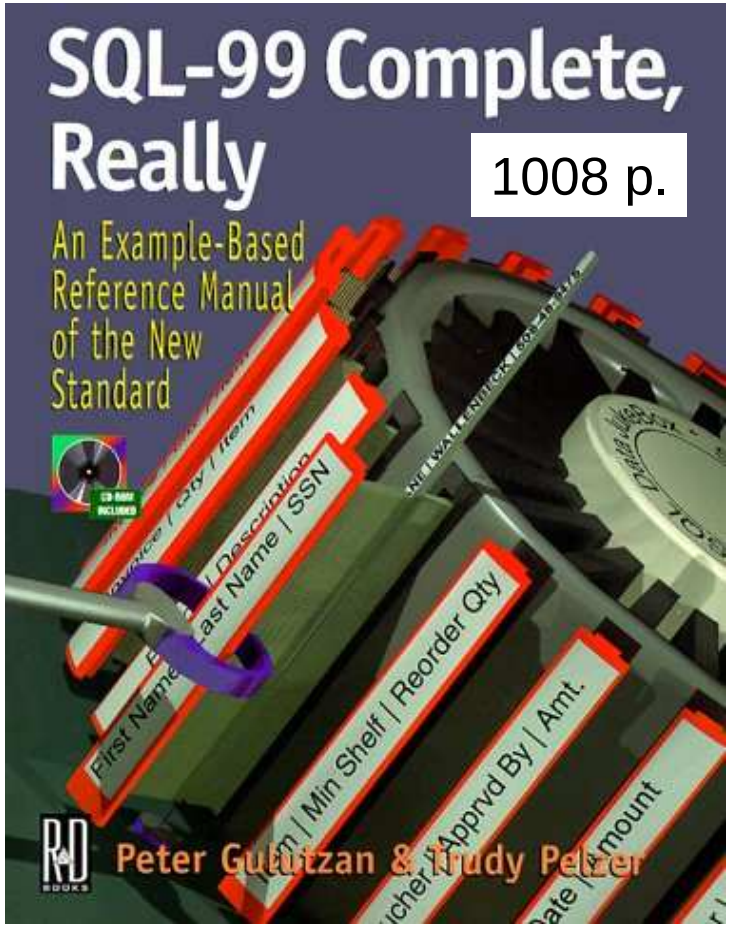
Completeness vs. efficiency

Work started at opposite ends grows towards convergence

Incompleteness is fine for academic research, as long as it's clear

- what is left out
- how it could be added

Towards the end of the religion war



From www.w3.org/XML/Query#specs

Specifications:

XML Query Requirements

XML Query Use Cases

XQuery 1.0 and XPath 2.0 Data Model

XSLT 2.0 and XQuery 1.0 Serialization

XQuery 1.0 and XPath 2.0 Formal Semantics

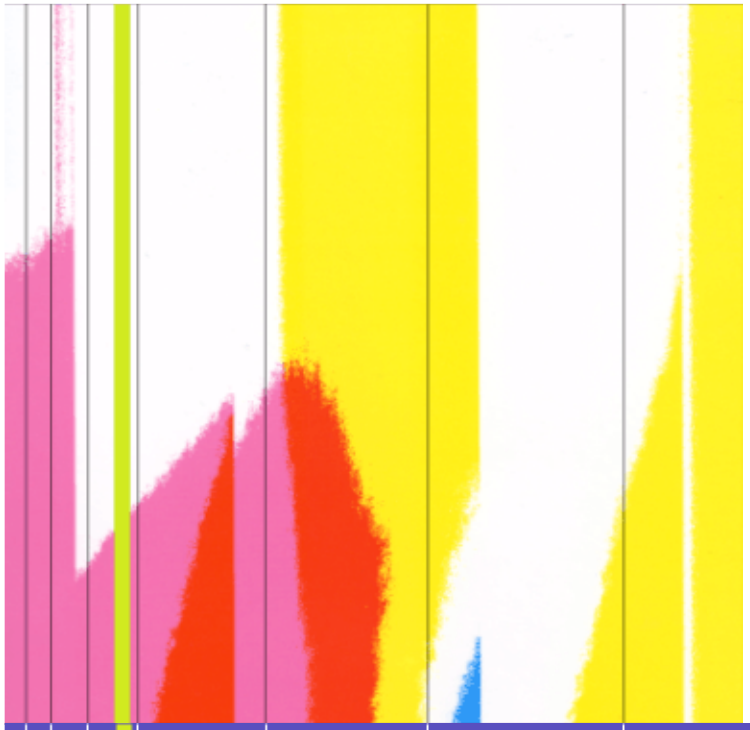
XQuery 1.0: An XML Query Language

XML Syntax for XQuery 1.0 (XQueryX)

XQuery 1.0 adn XPath 2.0 Functions and Operators

XPath Requirements Version 2.0

XML Path Language (XPath) 2.0



2.
XML storage and indexing models

INSTITUT NATIONAL
DE RECHERCHE
EN INFORMATIQUE
ET EN AUTOMATIQUE

INRIA

Some well-known relevant problems

Access path selection: choosing the best way to retrieve from disk a given data set

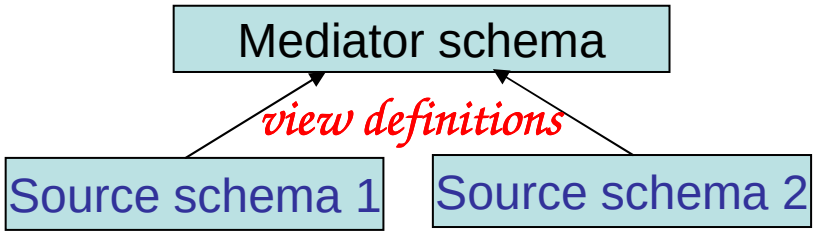
Parameters: data model, available storage structures

- Relational scan vs. index-based access
- Object navigation vs. scanning class extents

View-based query rewriting: reformulating a query based on a set of (materialized) views

Parameters: data model, view and query languages

- Local-as-view data integration



Some well-known relevant problems

Materialized view selection: choosing a set of views to materialize to speed up certain queries (supposes view-based query rewriting)

Parameters: query language, query workload, space

- Materialized view selection in a data warehouse: all sales of beer, or sales of beer in May, or sales of beer in France by brand...

Automatic index and view selection: choosing a set of views and indexes to speed up certain queries

Parameters: query language, query workload, space

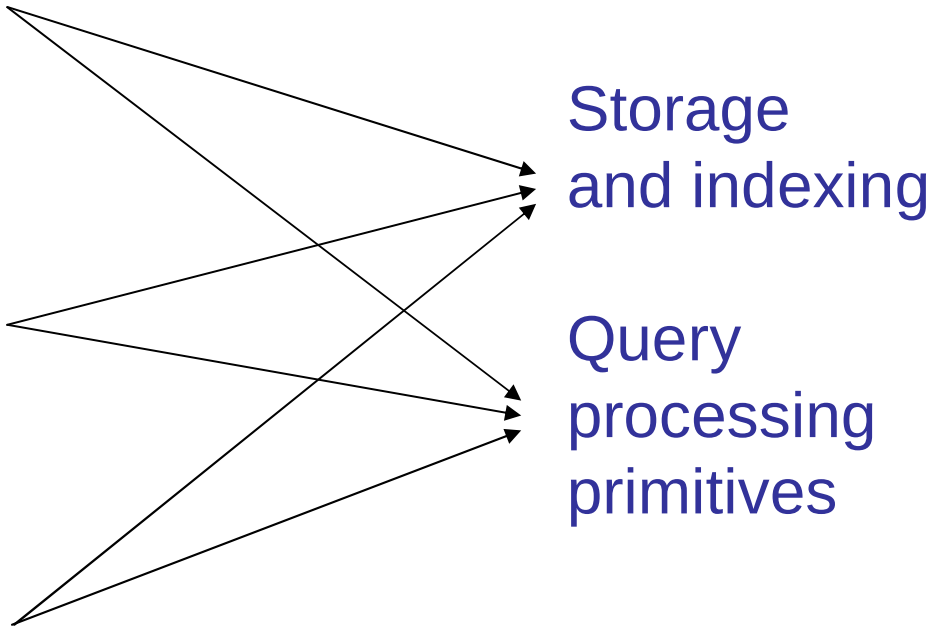
- Index tuning wizards (SQL Server, DB2)

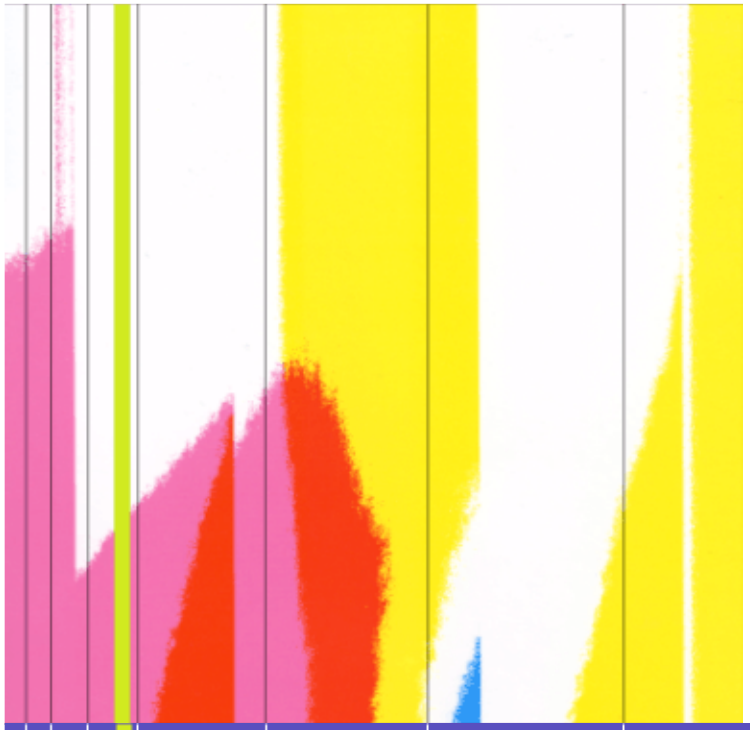
Plan

2.1 The precursors: OEM and Lore

2. 2 Relational stores schemes

2.3 Native stores





2.1 Storage, indexing, and query processing for OEM data

INSTITUT NATIONAL
DE RECHERCHE
EN INFORMATIQUE
ET EN AUTOMATIQUE

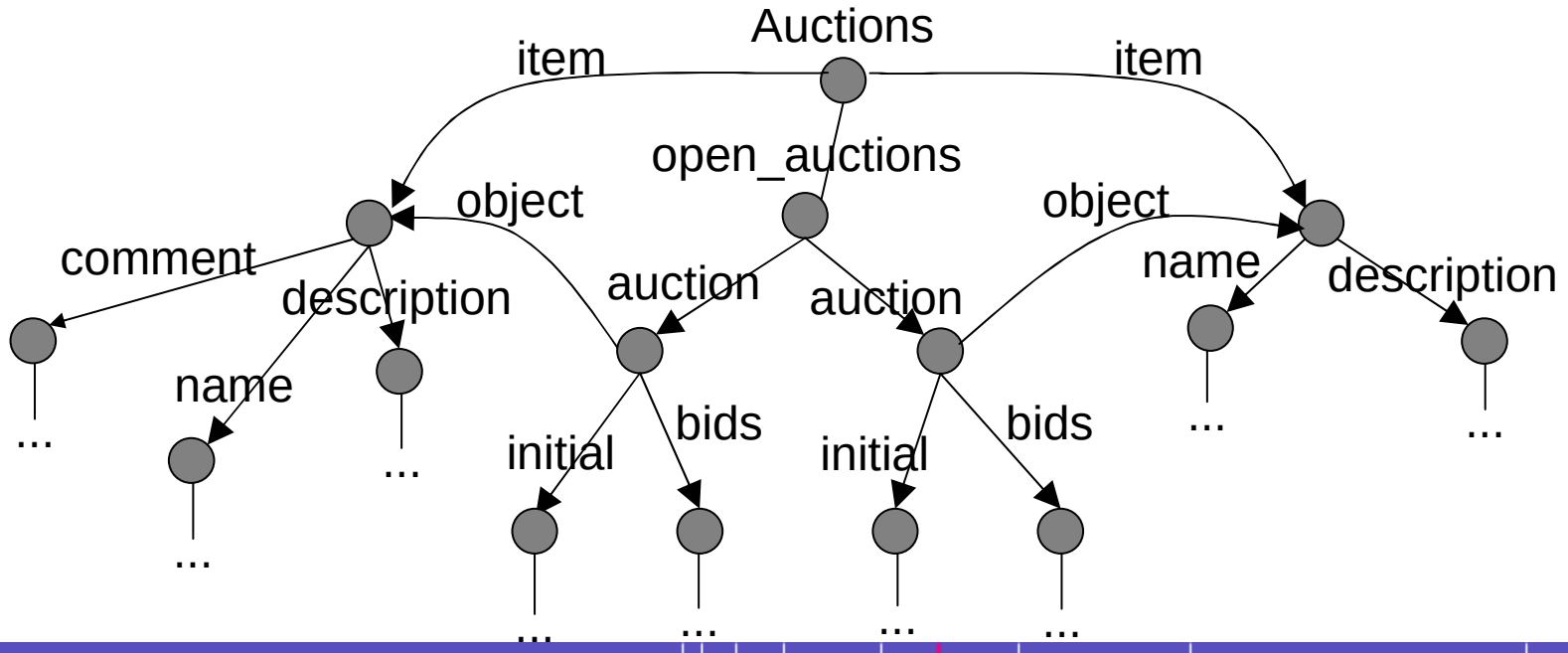
INRIA

OEM data model [PGW95]

Labeled, directed, unordered graph of objects

Objects have unique identity

Atomic objects = values (simple atomic types)



The interest of OEM today

Many interesting ideas for XML indexing

- Tree / graph data model for XML

Graph-structured data has interesting applications

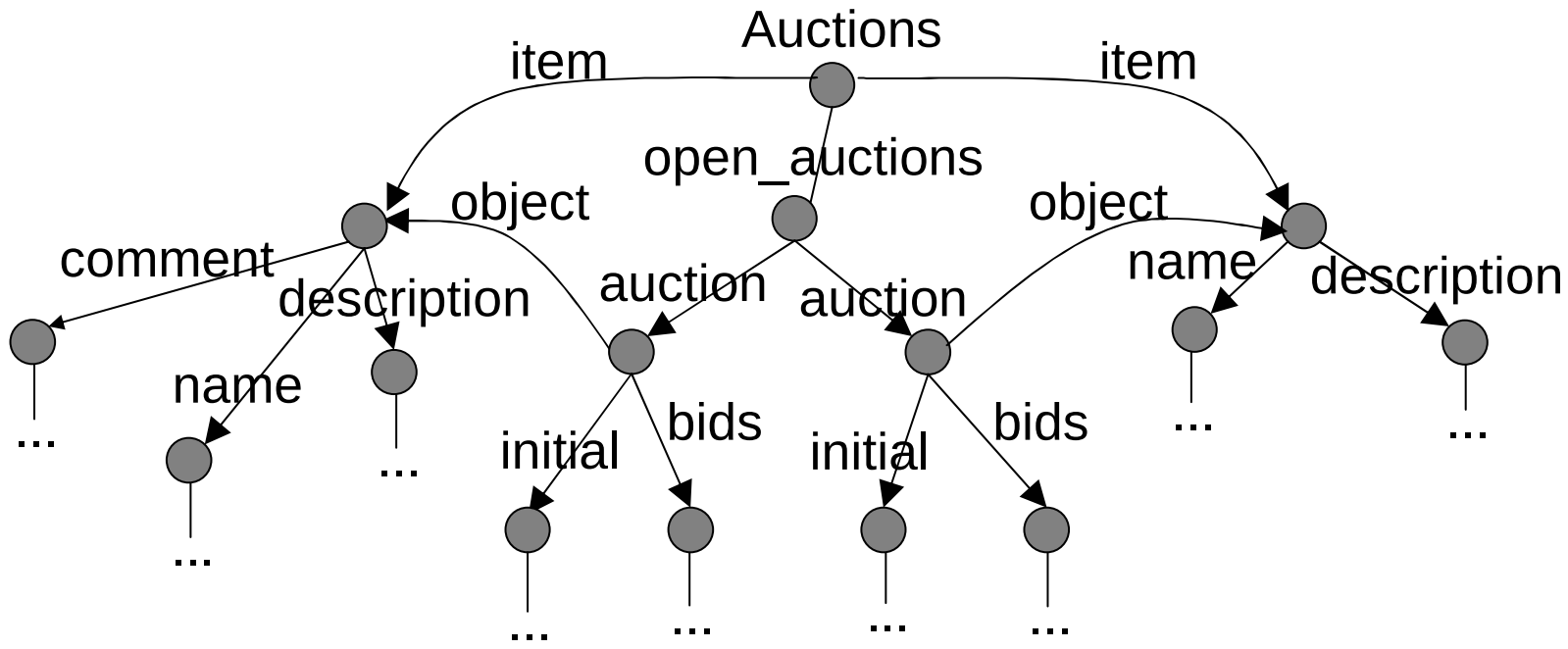
- Category hierarchies
- This document is a *Presentation* and an *EDBT2004Document* and an *XMLStorageDocument*
- (Heard of database file system ?)

OEM featured cycles; today's applications more focused on DAGs

Storing OEM objects

No invariants ! Need to store

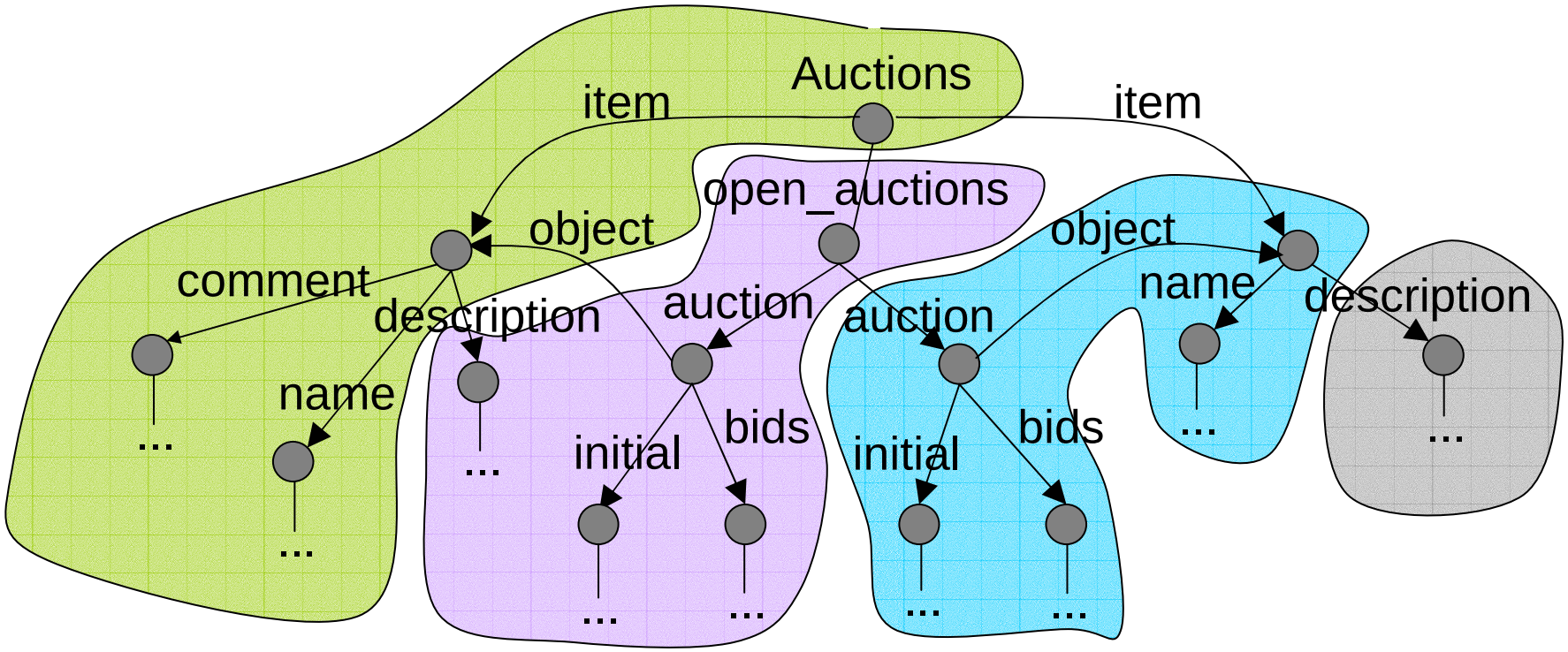
- Object identity, simple values
- Named edges



Storing OEM objects in LORE [MAG+97]

Objects clustered in pages in depth-first order, including simple value leaves

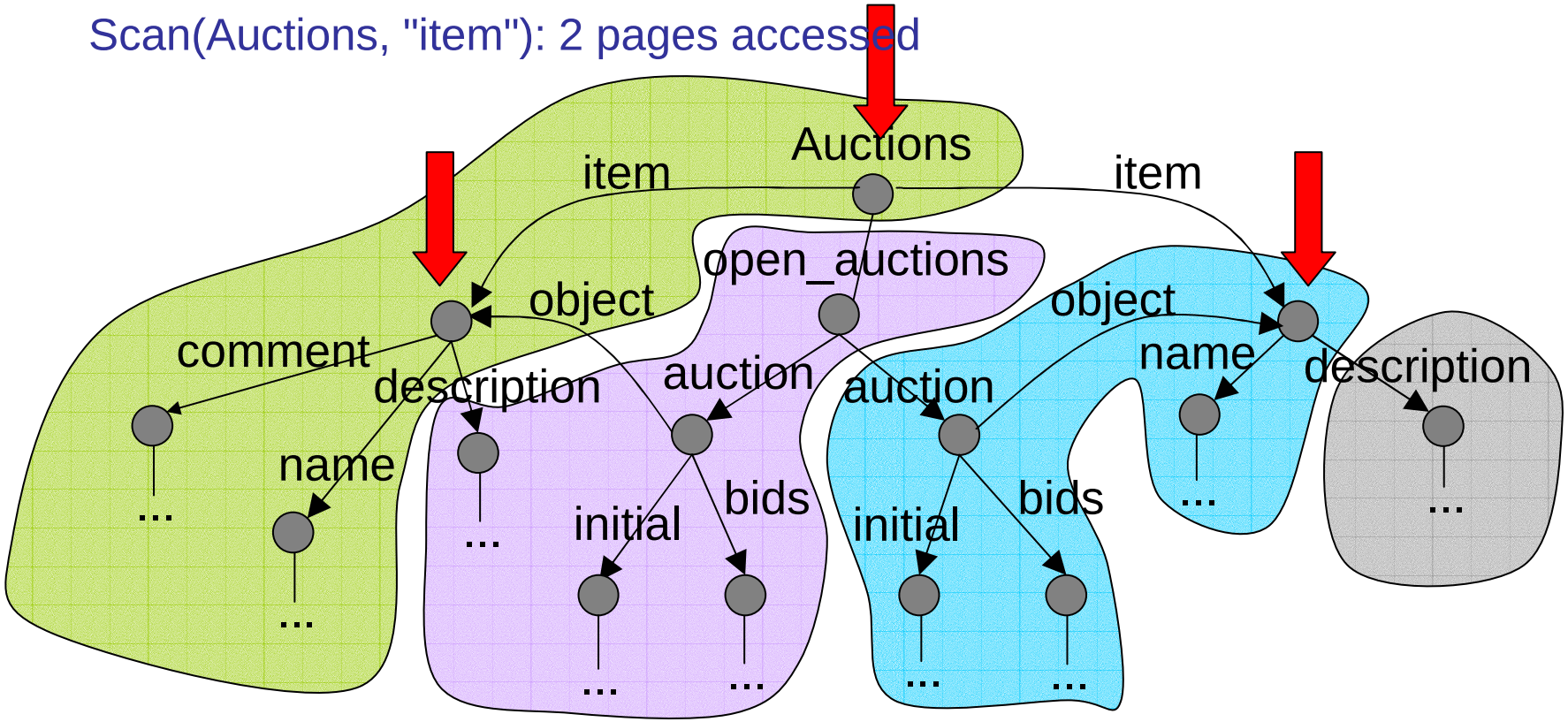
Basic physical operator: Scan(obj, path)



Scanning objects in Lore

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

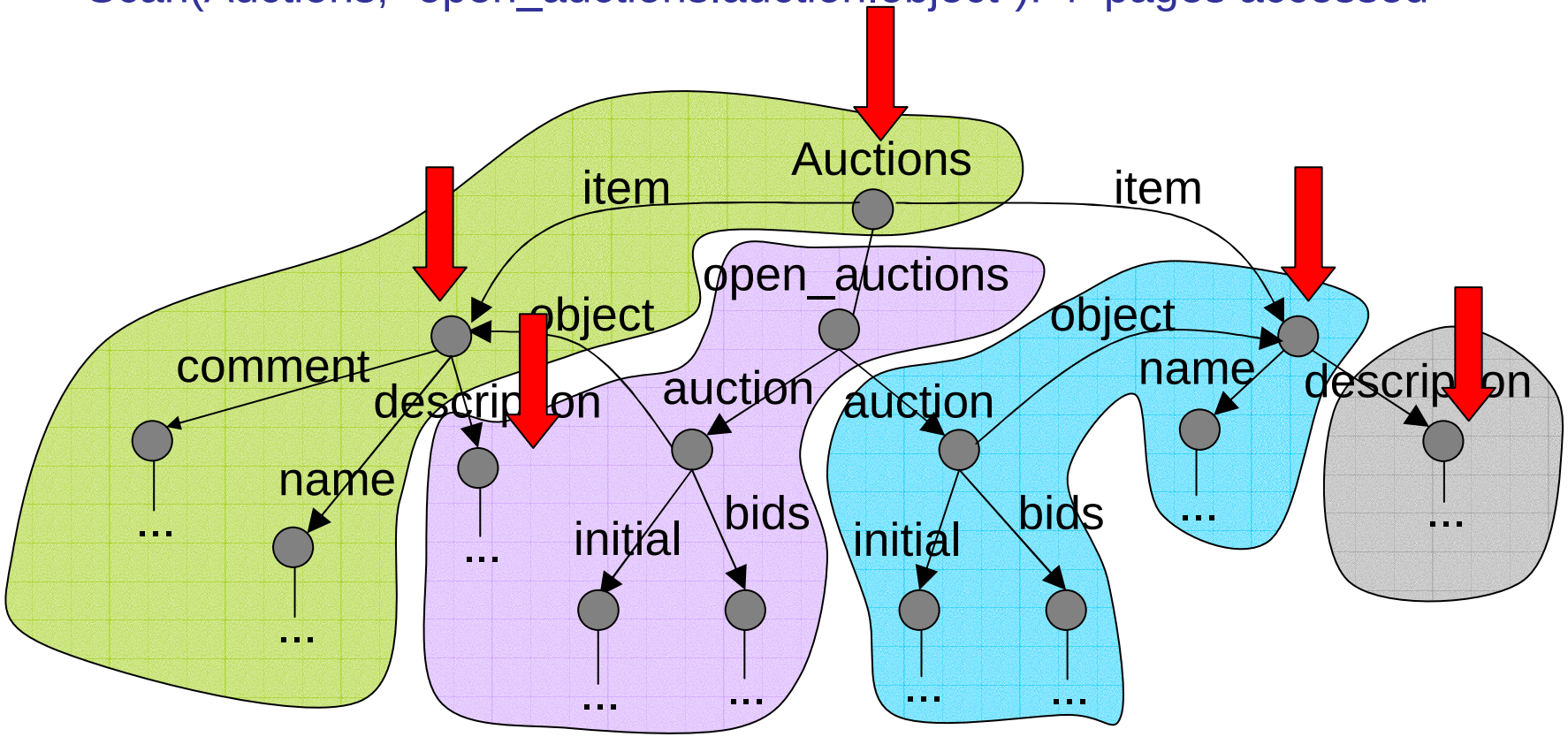
Scan(Auctions, "item"): 2 pages accessed



Scanning objects in Lore

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

Scan(Auctions, "open_auctions.auction.object"): 4 pages accessed



Indexing OEM objects in Lore

[MW97,MWA+98,MW99a,MW99b]

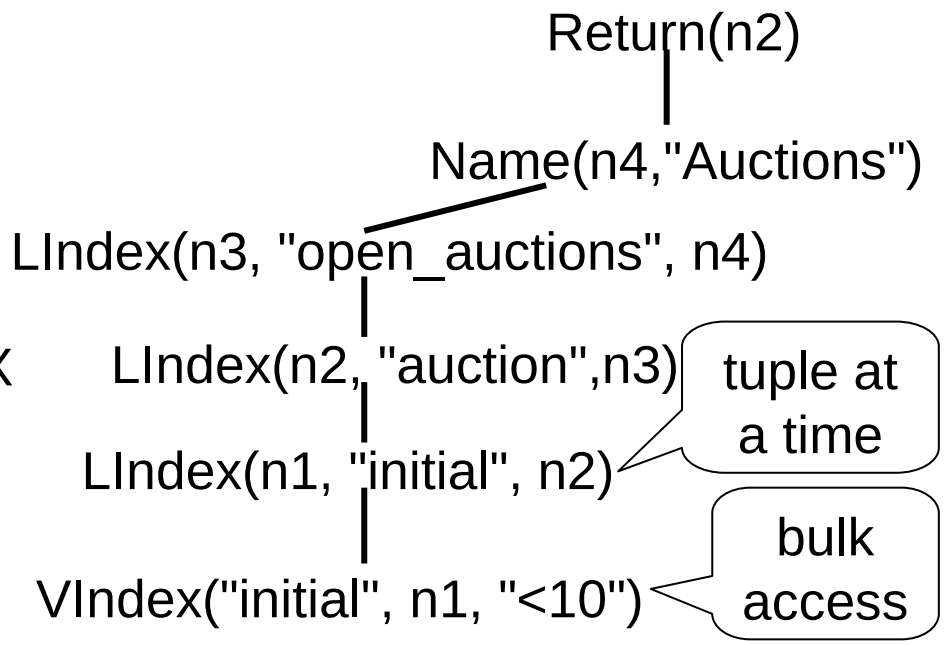
VIndex(**l**, o, **pred**): all objects o with an incoming l-edge, satisfying the predicate

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

- "Reverse pointer chasing"

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

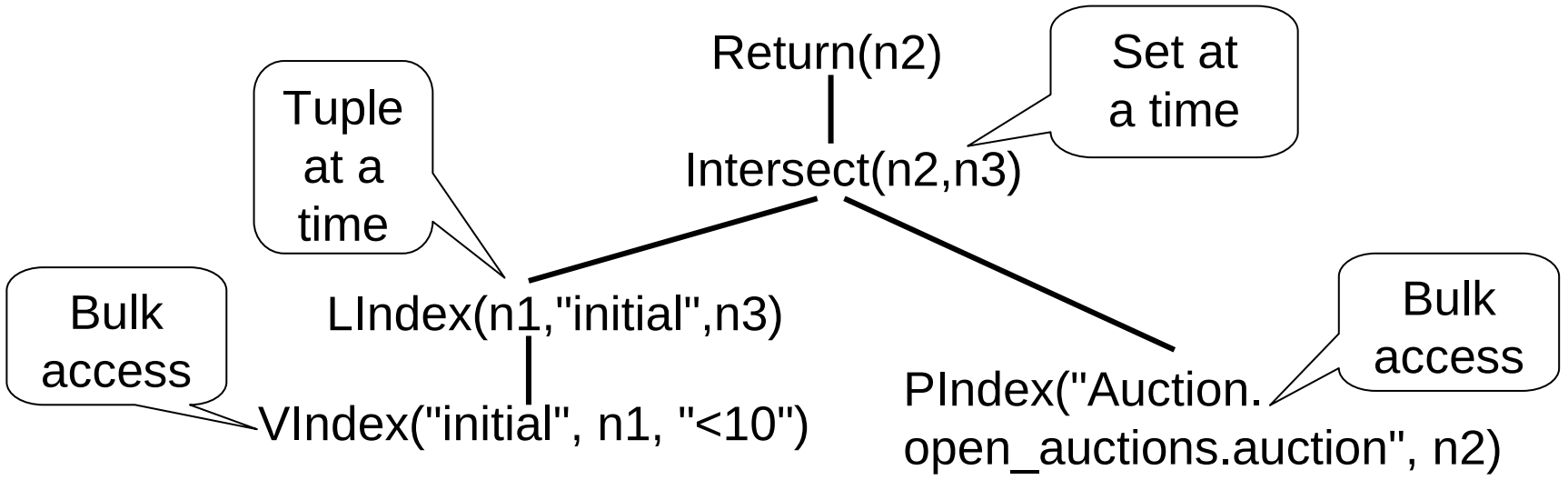
```
select X
from Auction.open_auctions.auction X
where X.initial < 10
```



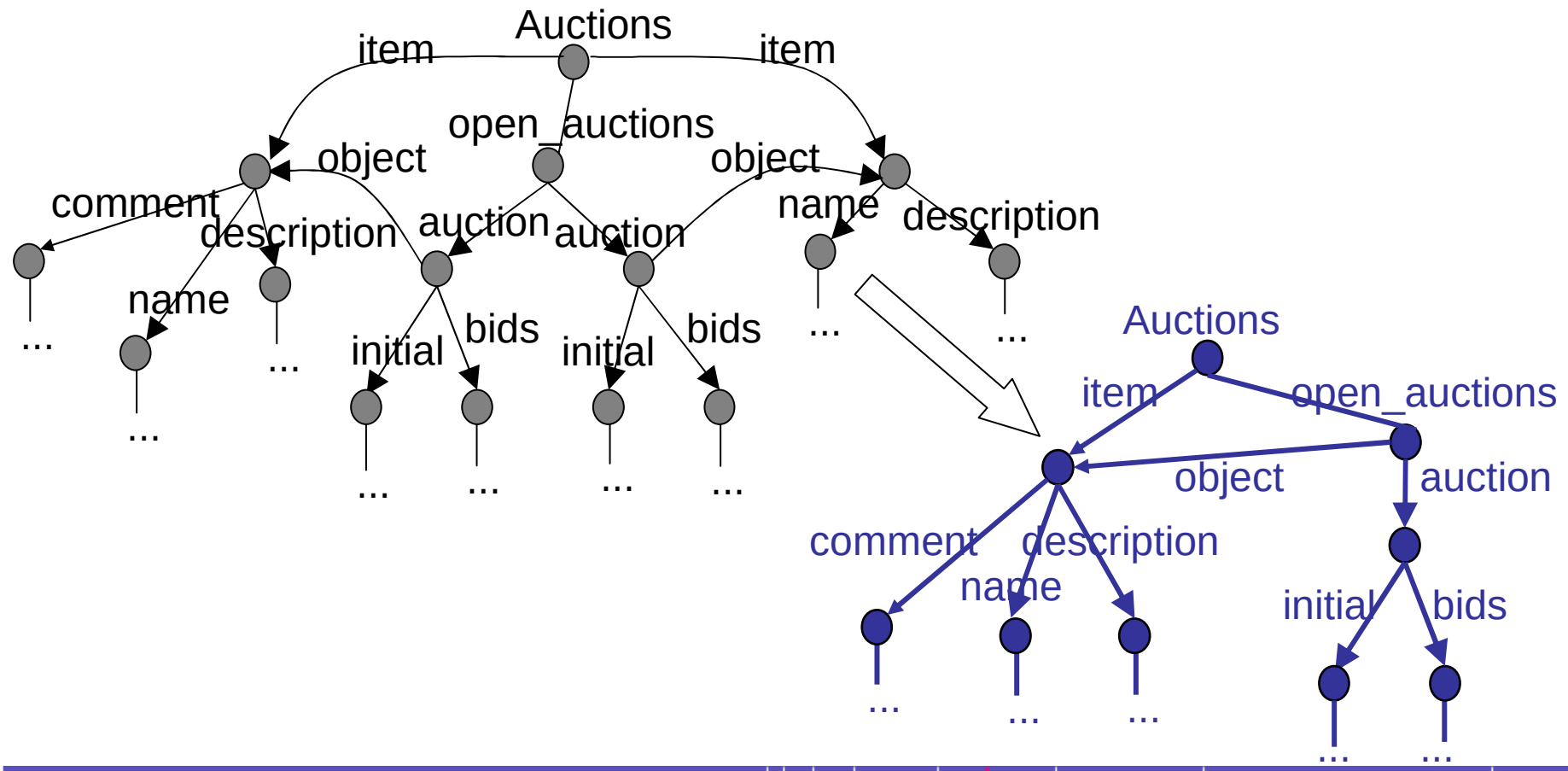
Indexing OEM objects in Lore [MW97]

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

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



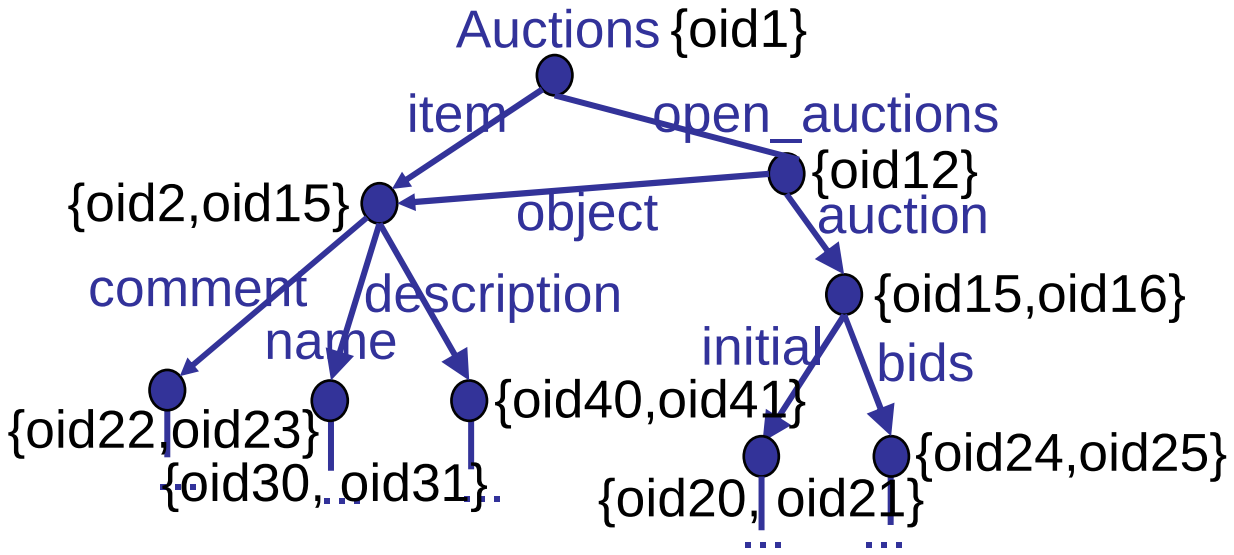
The idea behind path indexes: DataGuides [GW97]



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*



Another OEM indexing scheme: Template-index [MS99]

A T-index is meant for queries of the form

$P_1\ x_1\ P_2\ x_2\ \ldots\ P_k\ x_k$, returning $x_1, x_2, \ldots, x_k$

where each P_is is a general path expression, and may include wildcards (Example: *.item x P y)

Partition nodes in equivalence classes *dictated by the template*.

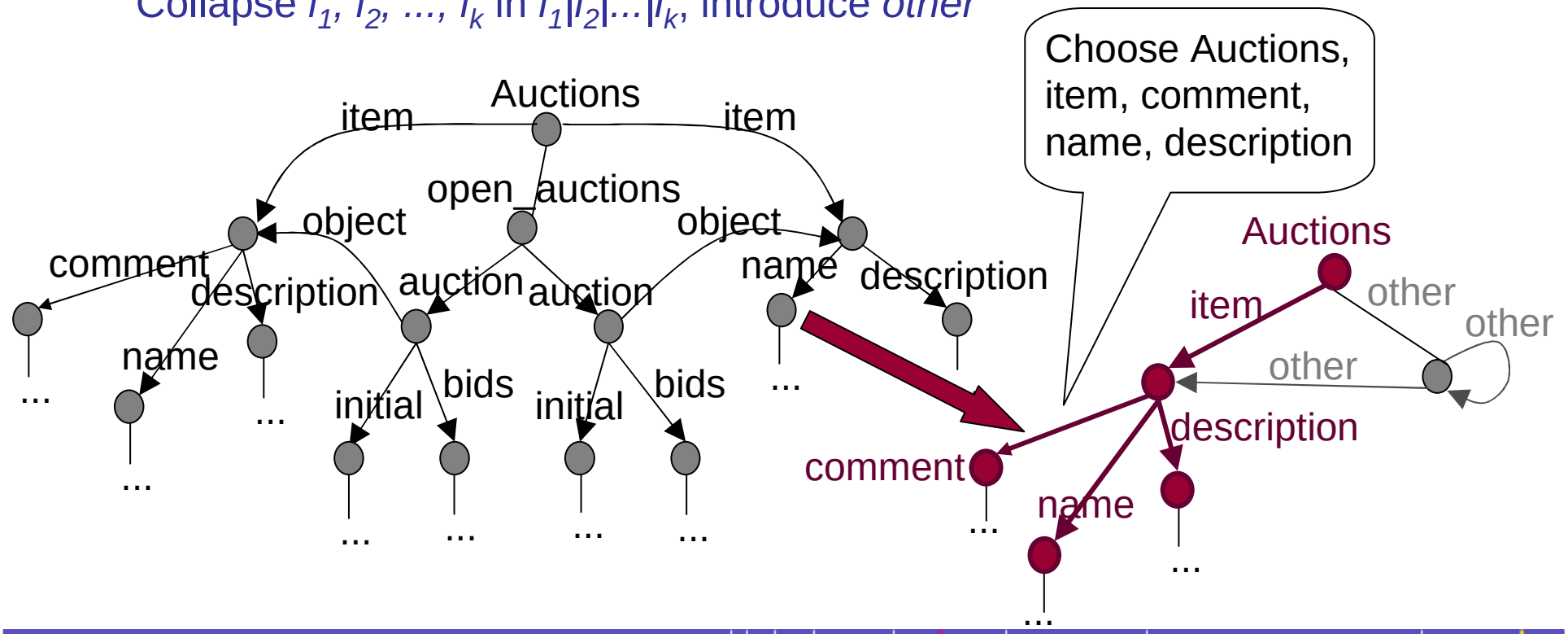
Simple case: 1-Index

- Groups nodes having *the same sets of incoming paths*
- DataGuides group nodes *under each of their incoming paths*

Another OEM indexing scheme: graph schemas [FS98]

Simplifications of the data instance (less labels)

Collapse l_1, l_2, ..., l_k in $l_1|l_2|...|l_k$; introduce *other*



Path query optimization based on graph schemas [FS98]

Prune path queries:

Auctions.*.description.*.listitem

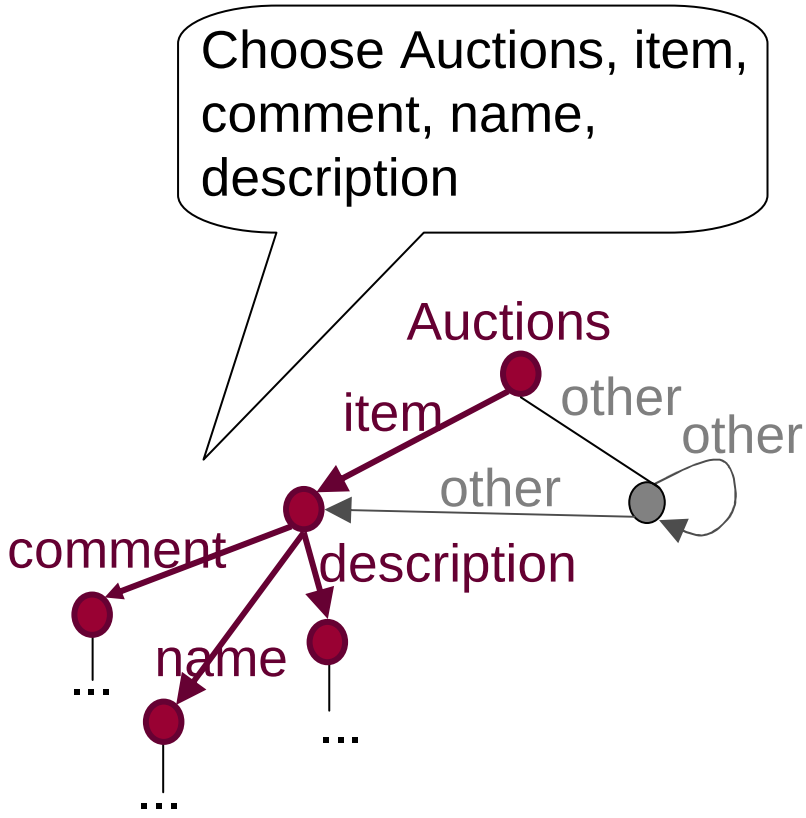
becomes

Auctions.item.description.*listitem

~ schema-based simplification

Associate extents to graph states, and use the extents to answer

~ indexing



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

Equivalence notions:

1. Reachable by some common paths: *DataGuide* [MW97]
2. Reachable by exactly the same paths: *1-index* [MS99] or, equivalently, indistinguishable by *any* forward path expression
3. Indistinguishable by any (forward and backward) path expression: *F&B Index* [ABS99,KBN+02]
4. Indistinguishable by the (forward and backward) path expressions in the set Q: *covering index* [KBN+02]
5. Indistinguishable by any path expression of length < k: *A(k) index* [KSB+02]

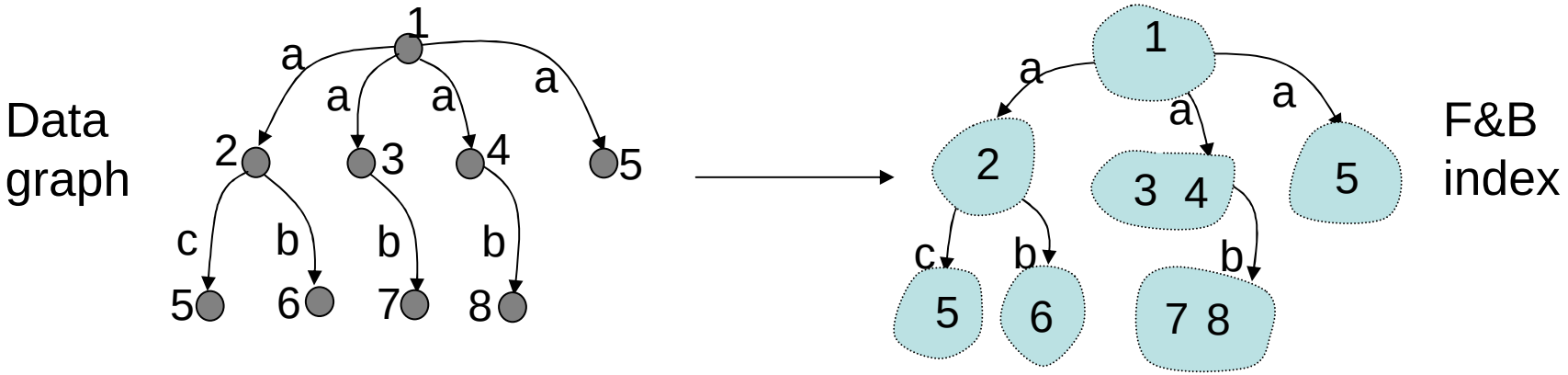
F&B index

Group together nodes reachable by exactly the same paths

Path language:

- Navigate along one edge in both directions
- Navigate along any number of edges, in both directions

n1 ~ n2: *for any path expression p, either n1 and n2 are in the answer of p, or neither are in the answer of p.*



Computing the F&B i

Group of nodes A is *succ-stable* wrt B
A ⊂ succ(B) or A an succ(re disjoint

A partition is *succ-stable* if all node g

Index built on succ-stable partition:

- if there is an edge from A to B
- then from every node in ext(A)

A partition is *pred-stable* if all node grou pairwise pred-stable.

The F&B index is the smallest graph index both pred-stable and succ-stable.

All nodes in any group have parents in exactly the same groups

All nodes in any group have children in exactly the same groups

Pred- and succ-stable partition leads to an index covering all path expressions

Computing the F&B index

P = label partition on the nodes (XML style)

Repeat

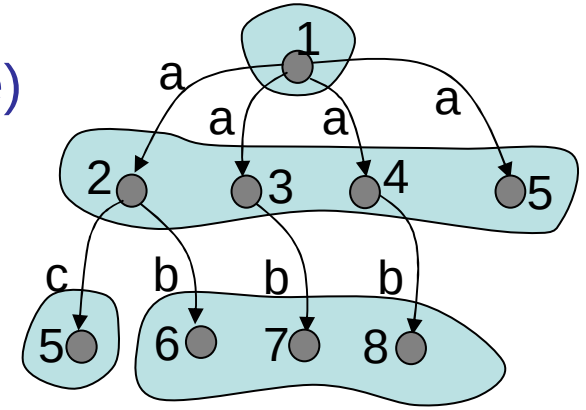
Reverse edges in the graph.

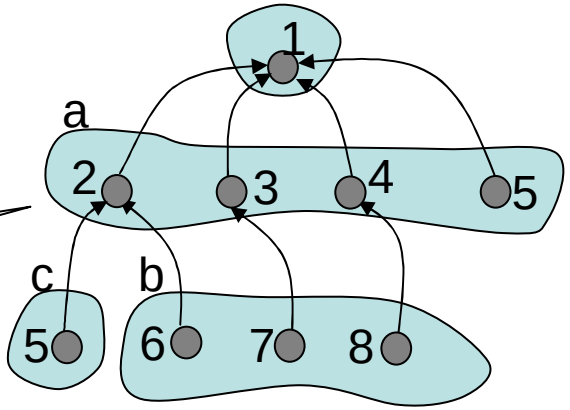
Refine P to make it succ-stable.

Reverse back edges in the graph.

Refine P to make it pred-stable.

Until P does no longer change.





group a is not included in succ(b)
but it intersects succ(b)

Computing th

P = label partition on the nodes (XML s

Repeat

Reverse ed

Refine P to make it succ-stable.

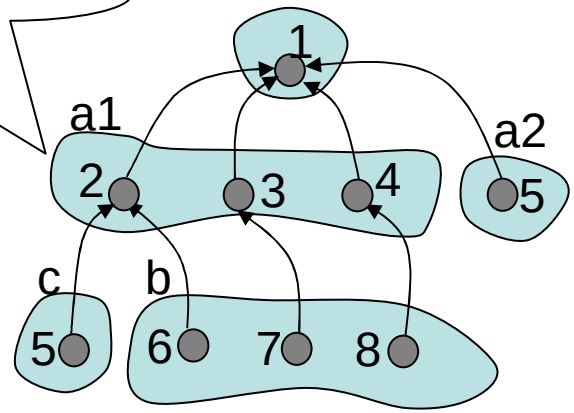
Reverse back edges in the graph.

Refine P to make it pred-stable.

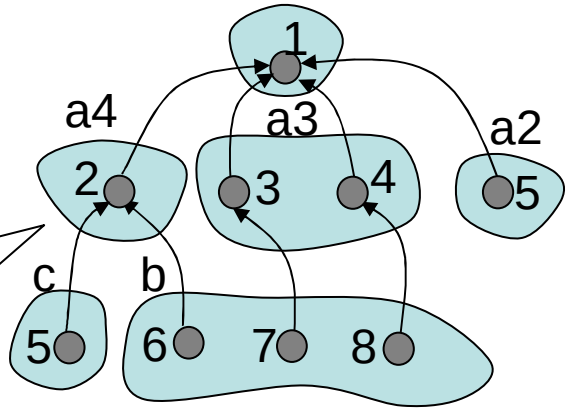
Until P does no longer change.

group a1 is included in succ(b),
group a2 is disjoint from succ(b)

group a1 intersects succ(c),
and is not included in succ(c)



group 1 included in succ(a2, a3, a4)
group a4 included in succ(a3)
group a3, a4 included in succ(a2)
succ-stable partition



Computing the F&B index

P = label partition on the nodes (XML style)

Repeat

Reve

Refin

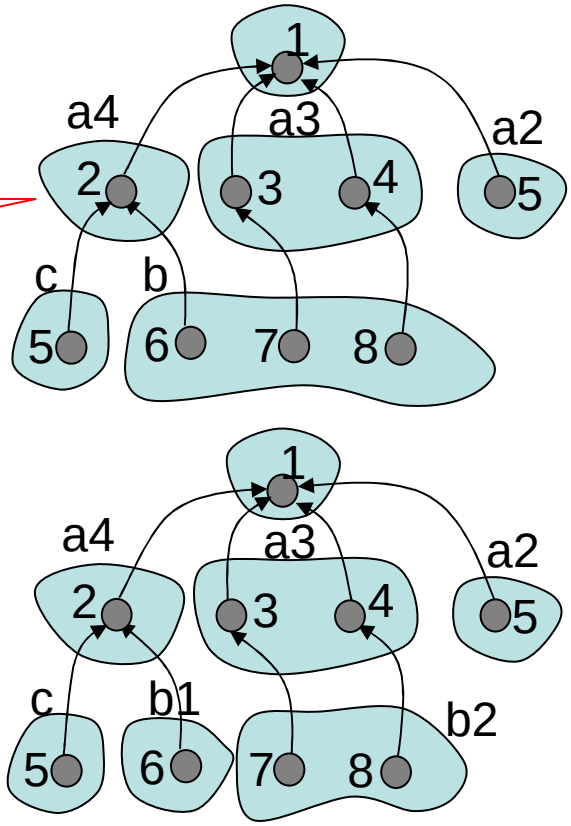
Reverse back edges in the graph.

Refine P to make it pred-stable.

Until P does

Pred(a4) intersects group b
and is not included in group b

Now P is pred-stable
and succ-stable



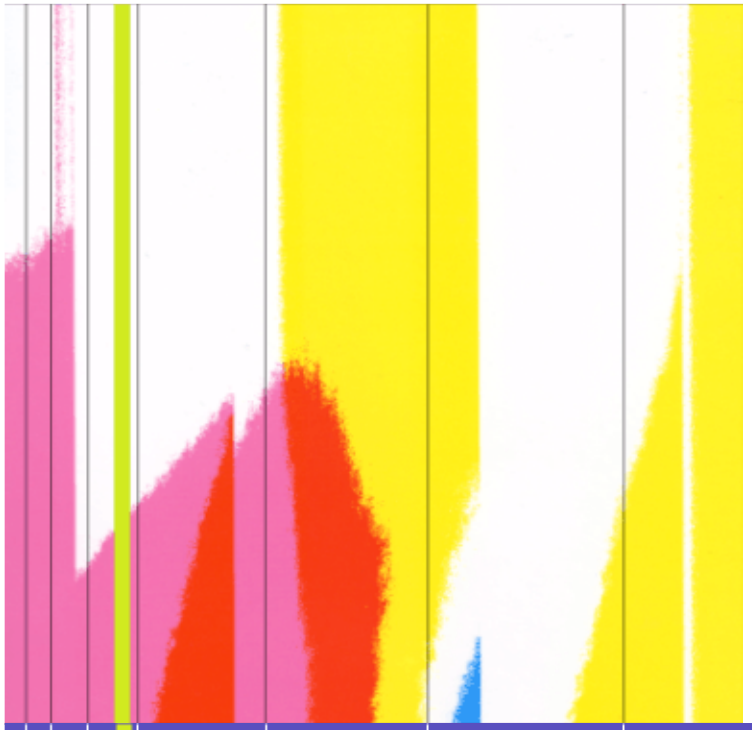
Summary: OEM storage and indexing

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).*
    - Formalism behind it: bisimulation [ABS99]

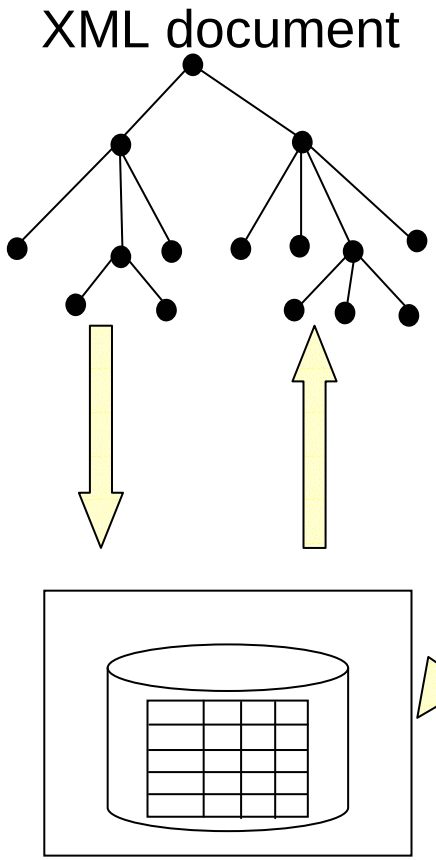


2.2
XML storage, indexing, and query processing based on relational databases

INSTITUT NATIONAL
DE RECHERCHE
EN INFORMATIQUE
ET EN AUTOMATIQUE

INRIA

Managing XML documents in RDBMSs



Load in RDBMS

Recompose document

Query processing:

- Translate XQuery in SQL
- Run SQL query
- Serialize XML result

RDBMS:

- *ACID properties*
- *Performance*

Choosing a relational storage schema for XML (1/2)

It must be possible to store *any* document

Documents *may* have an XML schema

Relational schemas seek to:

- *Exploit / identify structural invariants*
  - *"Tables for fixed or frequent structures"*
- *Factorize or avoid storage of labels*
- Factorize storage of values

Compared to OEM storage, relational XML storage is "already" indexing the data

Translation / mapping information sometimes hard to pinpoint

Choosing a relational storage schema for XML (2/2)

It must support efficient query processing (workload)

This depends on:

- The data (XML document shape and size, data values etc.)
- Physical data layout in the RDBMS
- XQuery-to-SQL translation; RDBMS optimizer, ...

It must support data and schema updates

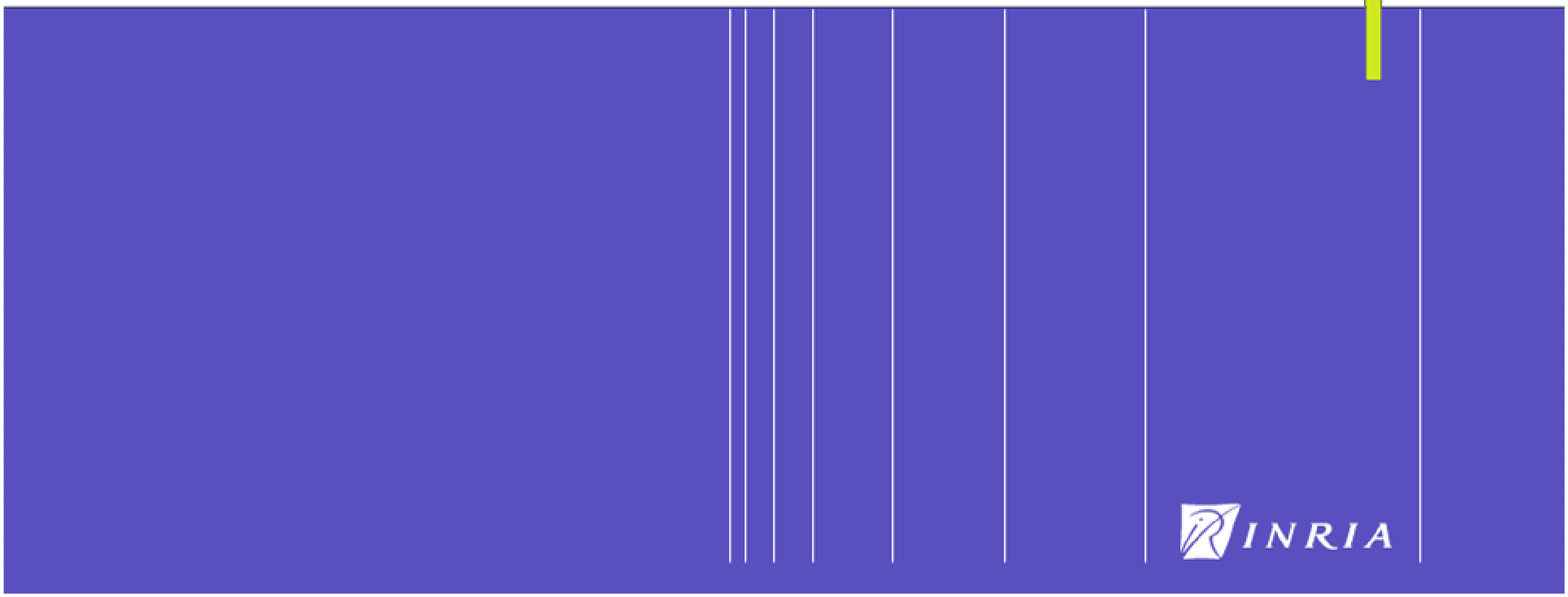
It may adapt to data and workload changes

It should be compact

Classification of relational storage schemes for XML documents

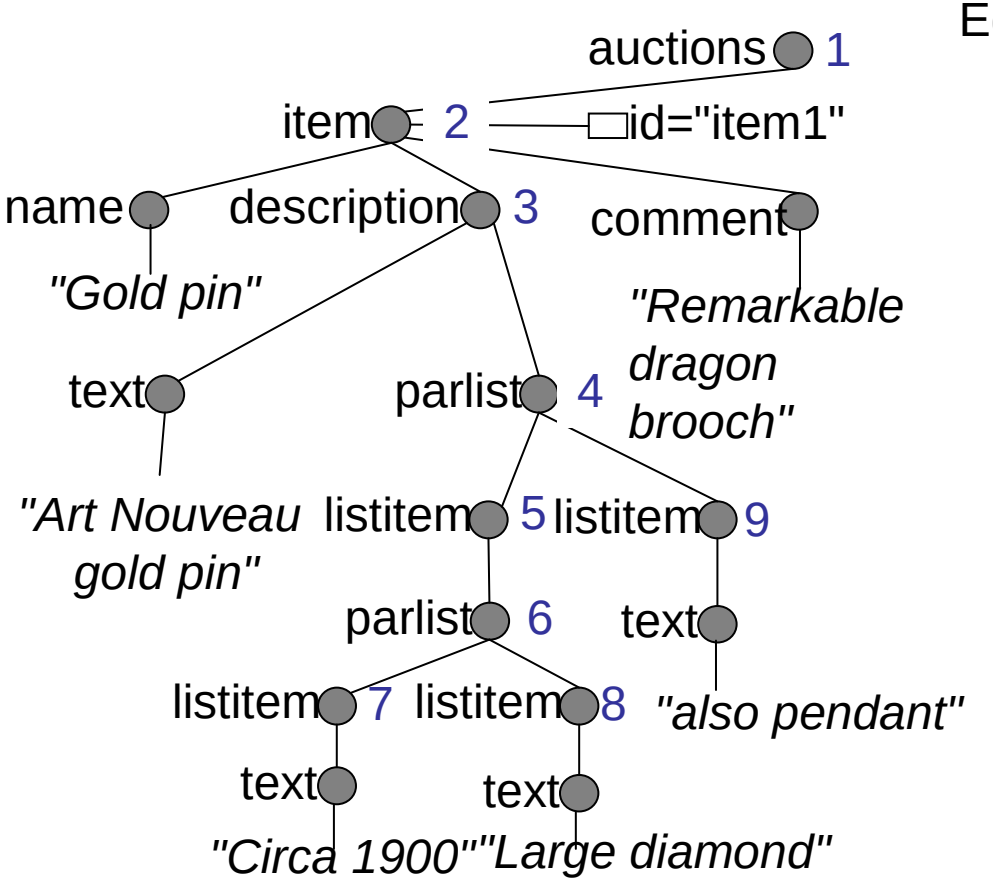
- No XML schema assumption (~ "Tree OEM")
  - Generic: same relational schema for all
  - Derive relational schema from the data
- Based on an XML schema
  - Based only on an XML schema
  - Based on schema and cost information
- User-defined

2.2.1
Relational storage for XML in the absence of an XML schema



The "Edge" storage scheme [FK99]

Simplest possible baseline: ordered edges



Edge(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	4
.....	.....	.....	.....

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

Index on pID, and (name,target)

Query processing on the "Edge" storage

Edge(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	4
.....			

//item

//comment

//item
[@id="
description

select target from Edge
where name="item"

Actual query answering also requires reconstructing the element... will discuss that later

e2,

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

3-way join on the Edge table

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

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 Lore's BIndex,
but *this is the storage*

Interesting groups of nodes: *those with*

Store tags in schema, not in data

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

Other storage schemes derived from "Edge"

Universal relation [FK99]:

EdgeAuction ⋈ EdgeItem ⋈ EdgeID ⋈ ...

Query-dictated materialized views over the partitioned Edge tables

EdgeAuction ⋈ EdgeItem

EdgeAuction ⋈ EdgeID

~ Materialized view selection problem

The STORED approach [DFS99]

Derive relational schema *from the XML data*

- Analyze elements to find *frequent patterns*
- Create one table for each type with enough *support* in the database
  E.g. one relation for articles with up to 3 authors
- Stores the remaining elements in *overflow tables*
  E.g. the remaining article authors

~ Materialized views over partitioned Edge, based on pattern frequency

Performance benefits: less joins for many queries

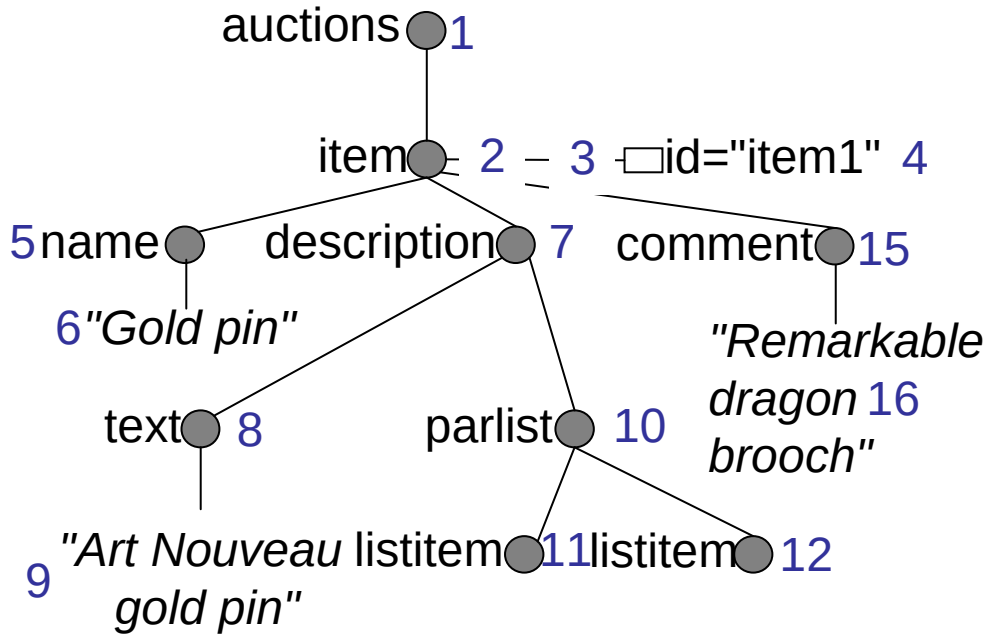
"Interesting node groups" are frequent ones

Most tags stored in the schema, not in the data

Path partitioning in Monet [SKV+00]

1 relation per root-to-leaf data path: Path(id1, id2, ord)

1 relation per root-to-inner node data path: Val(id, val)



auctions (id1, id2, ord)
auctions.item(id1, id2, ord)
auctions.item.id(id1, id2, ord)
auctions.item.name(id1, id2, ord)
...
auctions.item.id.val (id, val)
auction.item.name.val (id, val)
...

Path partitioning in Monet [SKV+00]

(Very) similar to Lore's PIndex, DataGuides and 1-Index

But this is the storage

IDs are stored once per descendent

Values stored once

No join required for linear path expressions + wildcards

Path summary is needed to find elementary paths for a given path expression

auctions (id1, id2, ord)
auctions.item(id1, id2, ord)
auctions.item.id(id1, id2, ord)
auctions.item.name(id1, id2, ord)
...
auctions.item.id.val (id, val)
auction.item.name.val (id, val)
...

Path partitioning in Monet [SKV+00]

(Very) similar to Lore's PIndex, DataGuides

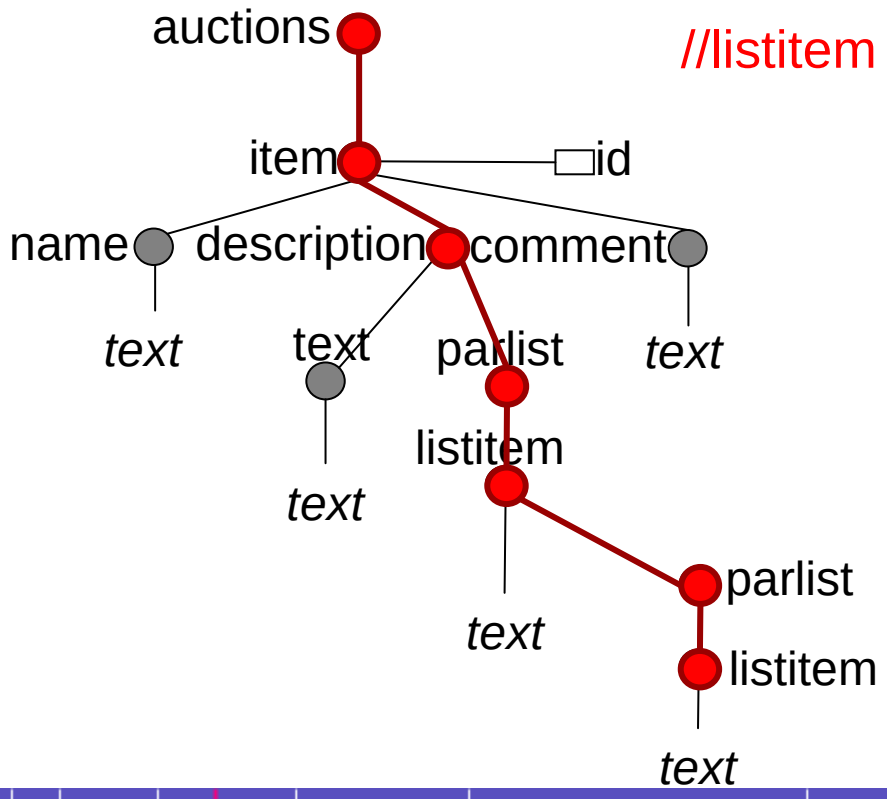
But this is the storage

IDs are stored once per descendent

Values stored once

No join required for linear path expressions + wildcards

Path summary is needed to find elementary paths for a given path expression



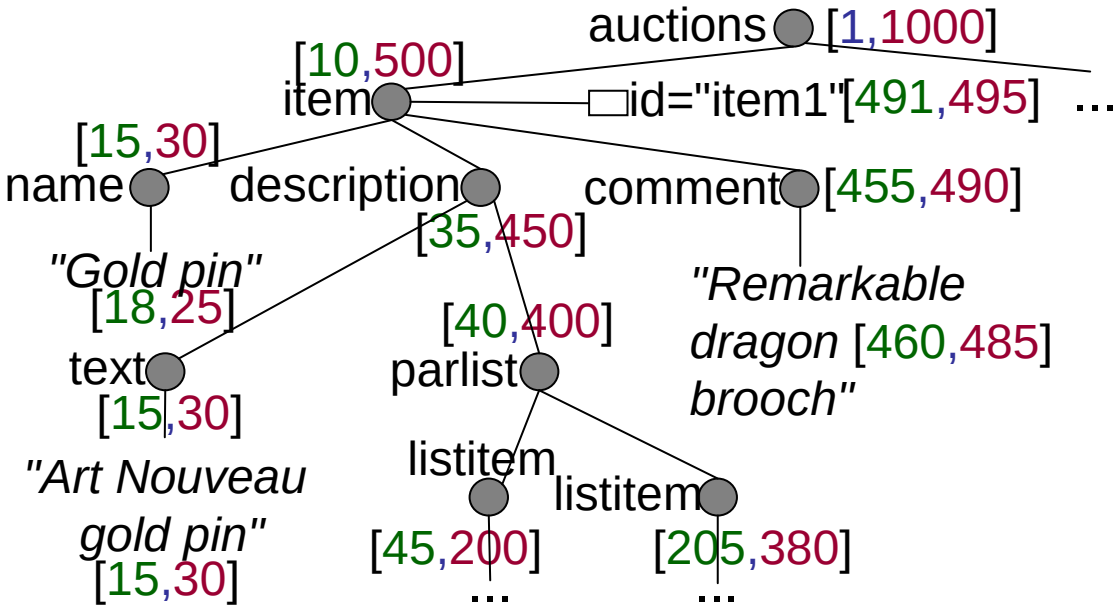
The XRel approach [YAT+01]

Path(PathID,PathExpr)

Element(DocID, PathID, Start, End, Ordinal)

Text(DocID, PathID, Start, End, Value)

Attribute(DocID, PathID, Start, End, Value)



Factorize path information in Path table
- (1, "/auctions")
- (2, "/auctions/item")
- (3, /auctions/item/name")

Assign region IDs [Start, End] to nodes

N1 ancestor of N2 iff N1.Start < N2.Start and N1.End > N2.End

Children relationship: extra condition on paths

The XRel approach [YAT+01]

Path(PathID,PathExpr)

Element(DocID, PathID, Start, End, Ordinal)

Text(DocID, PathID, Start, End, Value)

Attribute(DocID, PathID, Start, End, Value)

Processing regular path expressions: match *PE* against Path.PathExpr (string pattern matching)

Thanks to region IDs, *no joins required for linear path expressions*

Theta-joins still needed for branching path expressions

- match paths, then use indexes on Element.pathID

The XParent approach [JLW+01]

LabelPath(ID, lengh, pathExpr)

Data(pathID, nID, ord, value)

Element(pathID, ord, nID)

ParentChild(pID, cID)

Same string-based path encoding scheme as XRel

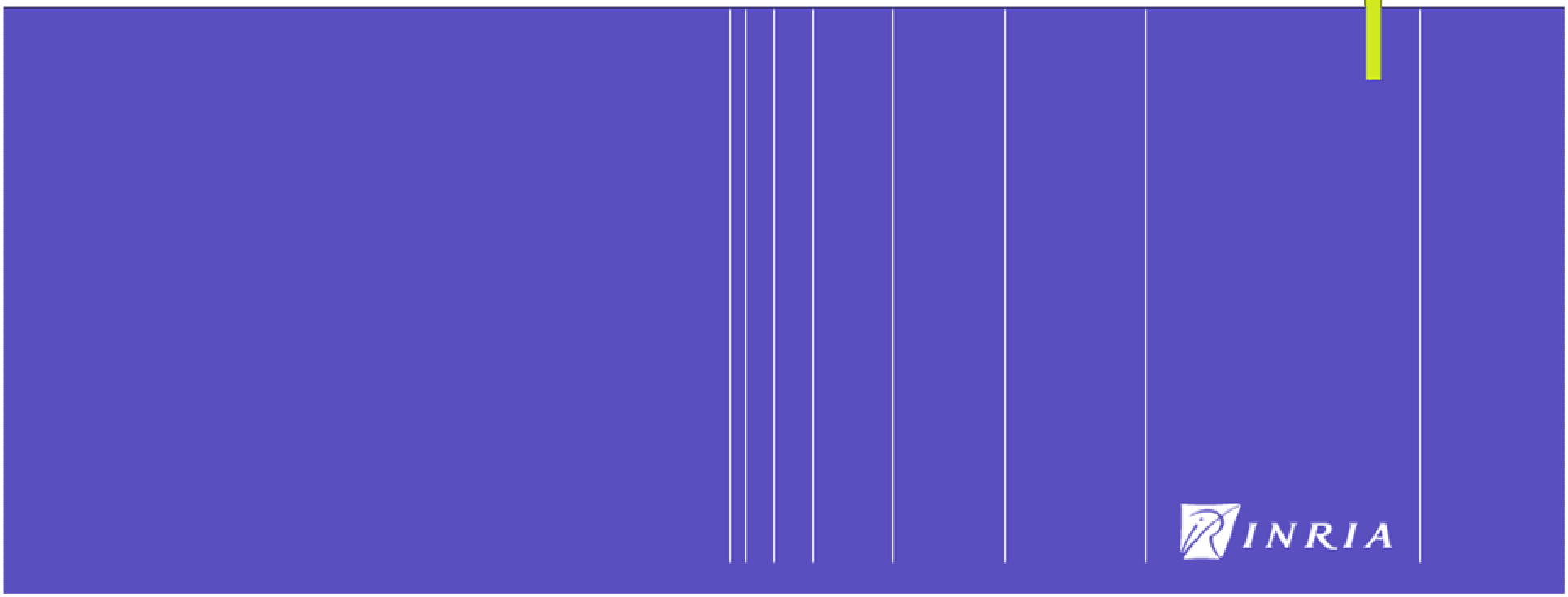
Drops region-based IDs, to avoid theta-joins

Parent-child relationships stored as such

Ancestor-descendent relationships established based on paths

XRel, XParent: still single large tables, only viable with indices

2.2.2
Relational storage for XML using an XML schema



DTD-derived relational storage schemes [STH+99]

Use a DTD graph derived from the DTD

<!ELEMENT a (b*, f)>

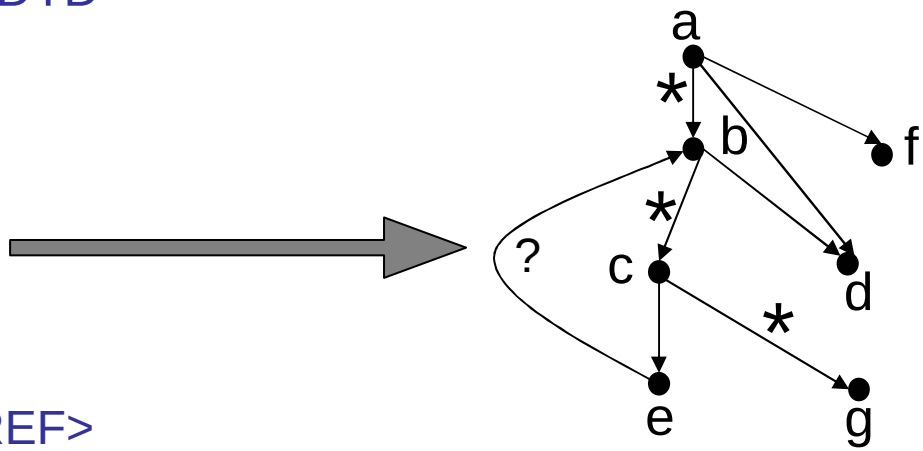
<!ELEMENT b (c+,d)>

<!ATTLIST b bID REQUIRED #ID>

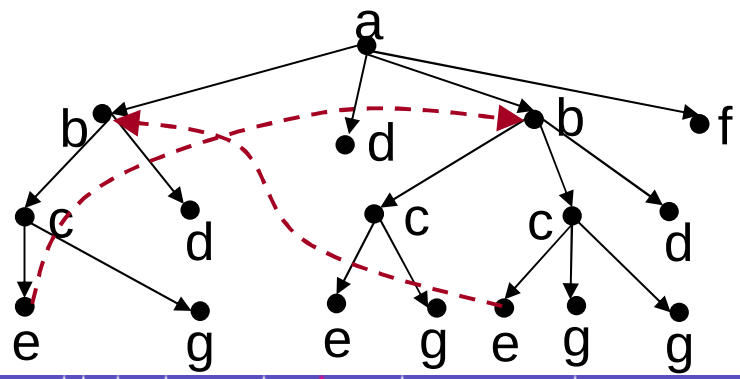
<!ELEMENT b (e, g*)>

<!ATTLIST e eREF REQUIRED #IDREF>

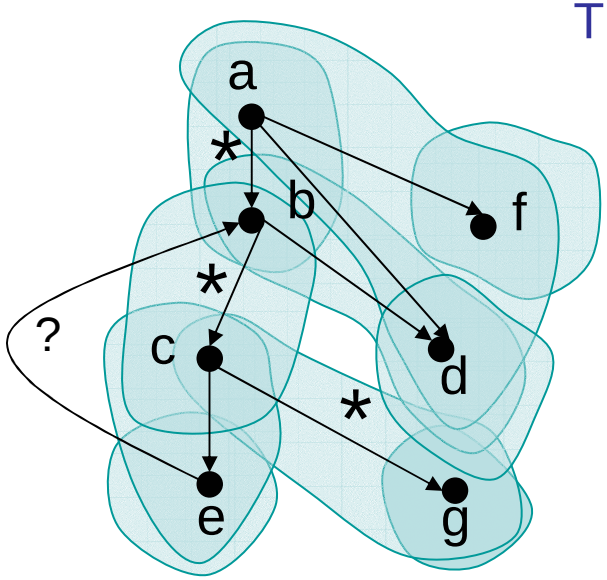
<!ELEMENT d|e|f|g #PCDATA>



XML document:



"Basic" relational storage [STH+99]



1 relation for every DTD graph node

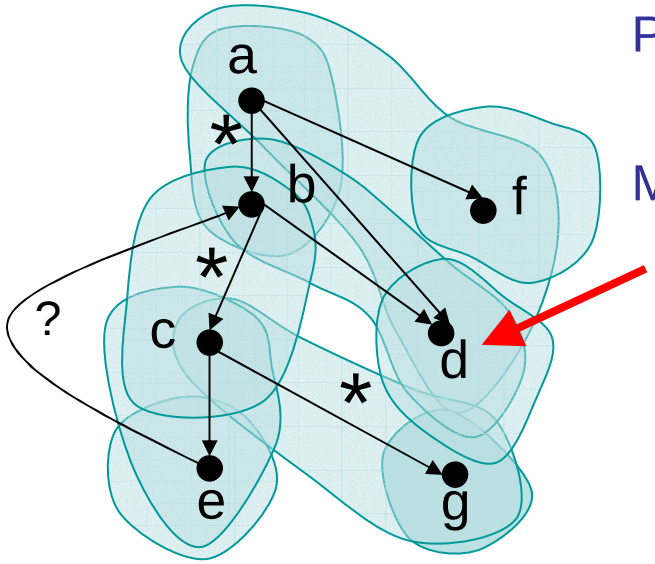
- XML documents can be rooted at any element

This relation contains all data contents accessible from the DTD graph node, except:

- * children (create separate relation)
- nodes with backpointers (create separate relation)

A(a.ID, a.f.val, a.d.val)
B(b.ID, *b.parentID*, b.ID.val, b.d.val)
A.B(a.b.ID, a.b.parentID, a.b.ID.val, a.b.d.val)
C(c.ID, *c.parentID*, c.e.val, c.e.attrE.val)
B.C(b.c.ID, *b.c.parentID*)
C.G(c.g.ID, *c.g.parentID*, c.g.val)
D(d.ID, *d.parentID*, d.val)
E(e.ID, *e.parentID*, e.val, e.attrE.val)
F(f.ID, *f.parentID*, f.val)
G(g.ID, *g.parentID*, g.val)

"Basic" relational storage [STH+99]



Node labels stored in the relational schema

Parent-child links materialized; // requires joins

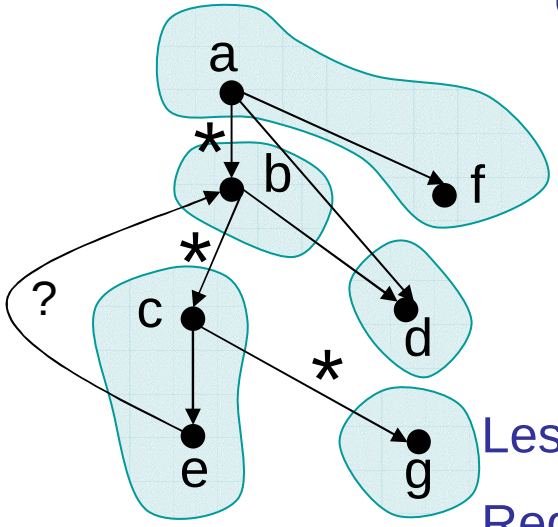
Path information split between relational schema and DTD (graph?)

Many tables, unions required, redundancy

A(a.ID, a.f.val, a.d.val)
B(b.ID, b.parentID, b.ID.val, b.d.val)
A.B(a.b.ID, a.b.parentID, a.b.ID.val, a.b.d.val)
C(c.ID, c.parentID, c.e.val, c.e.attrE.val)
B.C(b.c.ID, b.c.parentID)
C.G(c.g.ID, c.g.parentID, c.g.val)
D(d.ID, d.parentID, d.val)
E(e.ID, e.parentID, e.val, e.attrE.val)
F(f.ID, f.parentID, f.val)
G(g.ID, g.parentID, g.val)

"Shared" relational storage [STH+99]

1 relation for every DTD graph node with in-degree 0 or >1, or under a * or backpointer



A(a.ID, a.f.val, a.f.isRoot)
B(b.ID, b.parentID, b.ID.val)
C(c.ID, c.parentID, c.e.val, c.e.attrE.val, c.e.isRoot)
D(d.ID, d.parentID, d.parentCode, d.val)
G(g.ID, g.parentID, g.val)

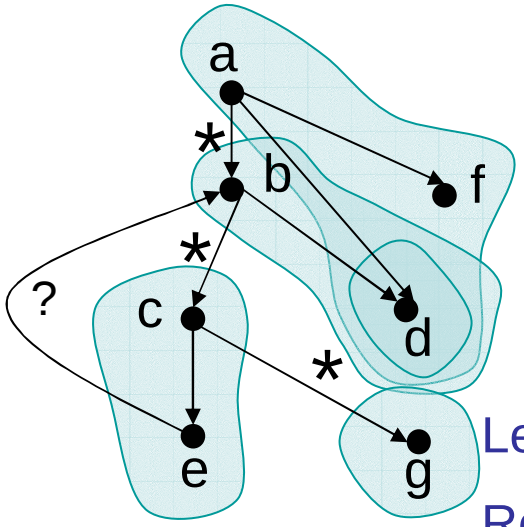
Less relations, no redundancy

Requires joins for reconstruction

Order support is complicated by ID absence

"Hybrid" relational storage [STH+99]

1 relation for every DTD graph node with indegree 0, under a *, or under a backpointer



A(a.ID, a.f.val, a.f.isRoot, a.d.val)
B(b.ID, b.parentID, b.ID.val, b.d.val)
C(c.ID, c.parentID, c.e.val, c.e.attrE.val, c.e.isRoot)
D(d.ID, d.parentID, d.val)
G(g.ID, g.parentID, g.val)

Less relations, no redundancy

Requires joins for reconstruction

Order support is complicated by ID absence

DTD-derived relational storage schemes

Labels stored in the schema; only some IDs stored

~ E-R analysis of data in XML:

- *"If every <a> has 1 <d>, and all <d>s have <a> parents, then <d> is an attributes of <a>s; otherwise, <d> is an entity"*
- DTD (graph) must be kept and exploited to translate queries, handle inlined elements, ...

Long path queries may be simplified, then matched accessing few tables

To bind many branches, (several) joins still needed

Choosing a relational schema based on an XML Schema and a query workload [BFR+02]

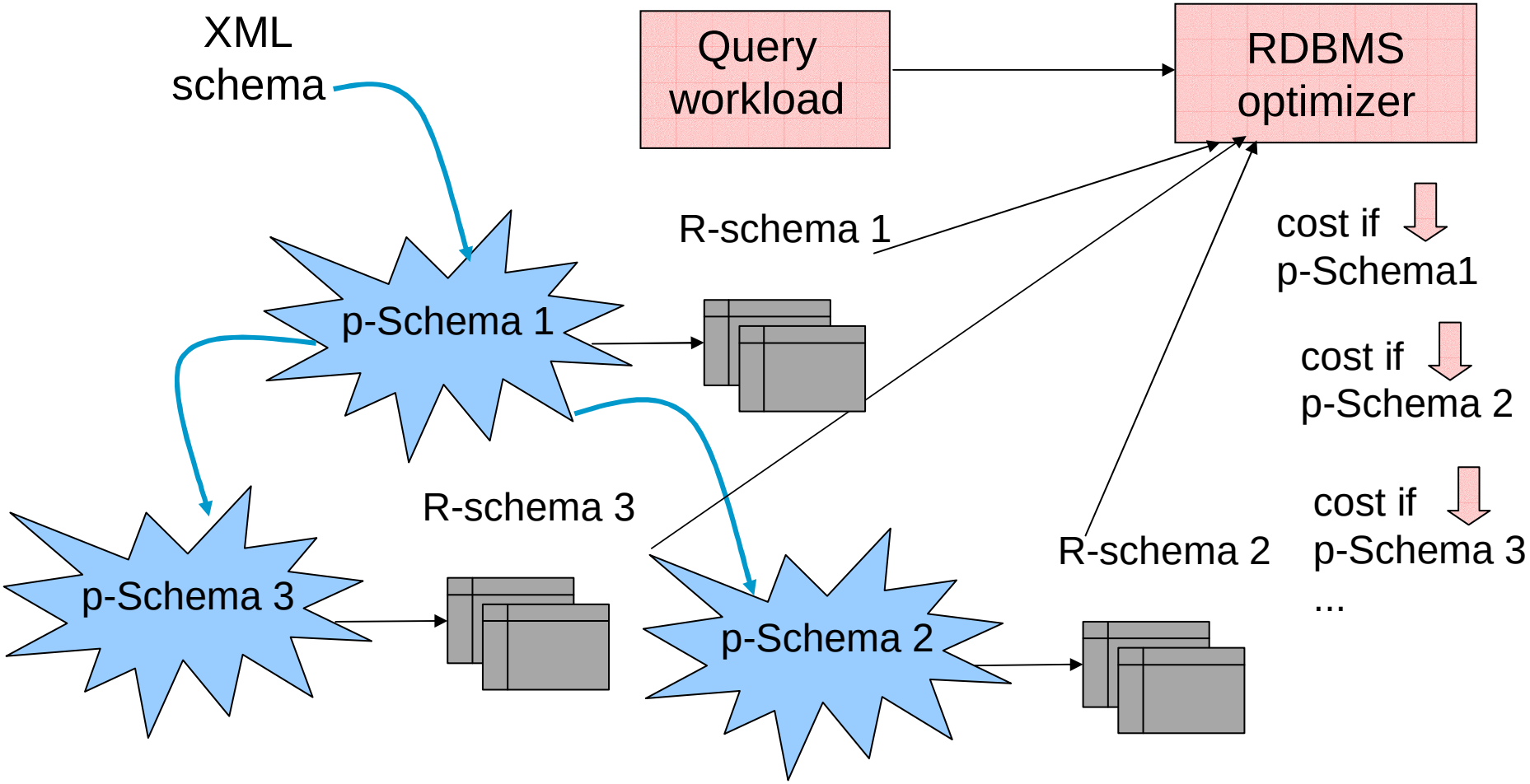
The idea:

- Different relational schemas work best for different workloads
- Fixed storage for a given *physical XML Schema* (~shared)
- Transformation rules for moving between p-Schemas
  - Equivalent p-Schemas: the sets of valid documents are the same
  - Differences due to XML Schema syntax

Cost-based optimization in the space of possible transformations

- Impact of a given p-Schema transformation estimated by the RDBMS's optimizer

Cost-based choice of a relational schema



Schema transformation rules based on queries

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 [t1] | a[t2]
- *Useful to separate if a[t1] often queried together, a[t2] rarely or never queried together*

Schema transformation rules based on queries

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 choice of relational schema

~ Materialized view selection for a dataset and workload

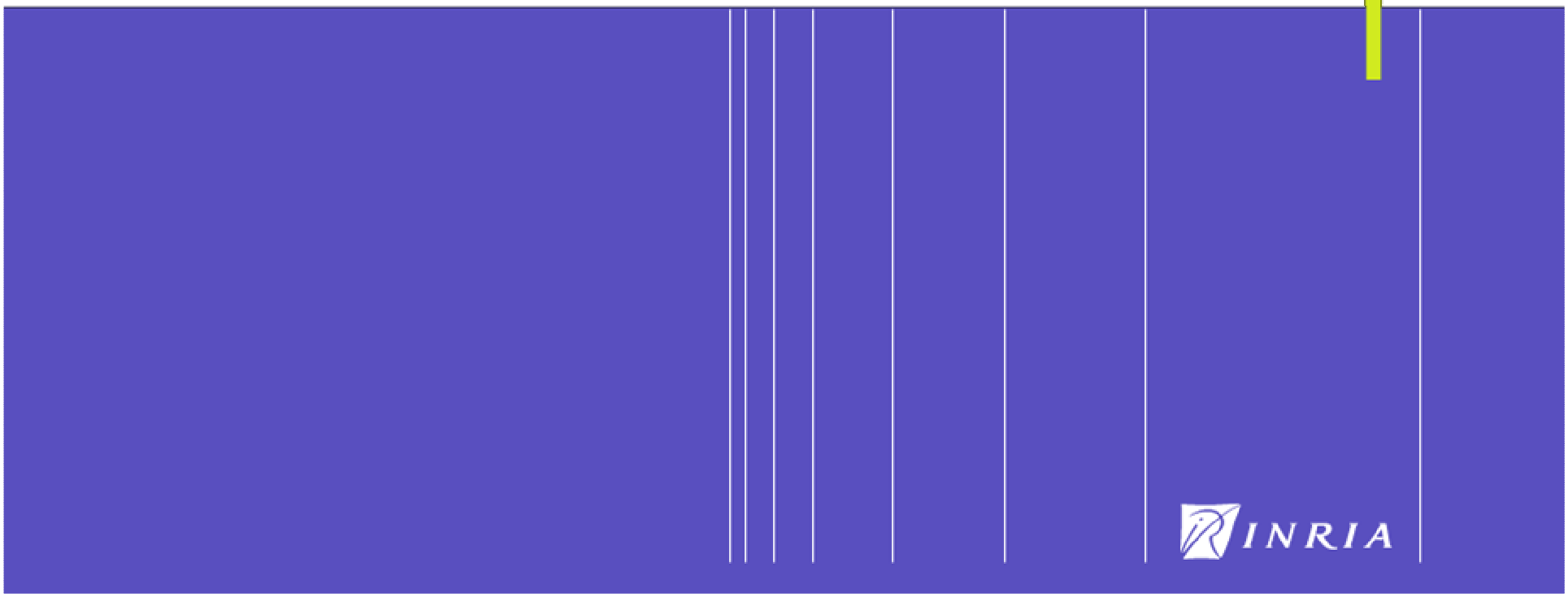
This is the storage

Optimizer estimates can be wrong, but the optimizer will make *the same mistake* when choosing the best plan

Search space explored in [BFR+02]:

- 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

2.2.3
User-defined relational mappings for XML documents



User-defined XML-to-relational mappings

[MFK01,DS03]

Express (relational) st… expressio…
document

- Relation = materia… XML docum…

Finding useful tables r… se…

R(y,z) :- Auctions.item x, x.@id.text() y, x.price

S(u,v) :- Auctions.item t, t.@id.text() u, t.description.text() v

for $x in //item

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

Does each item have exactly one price ?

Does each item have exactly one description ?

Is Auc… item … the sam… //item ?

Is @id a key for item ?

Not so fast.

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

?

User-defined XML-to-relational mappings
[MFK01,DS03]

Express (relational) storag pressions
document
- Relation = materialized document

Finding useful tables requ qu

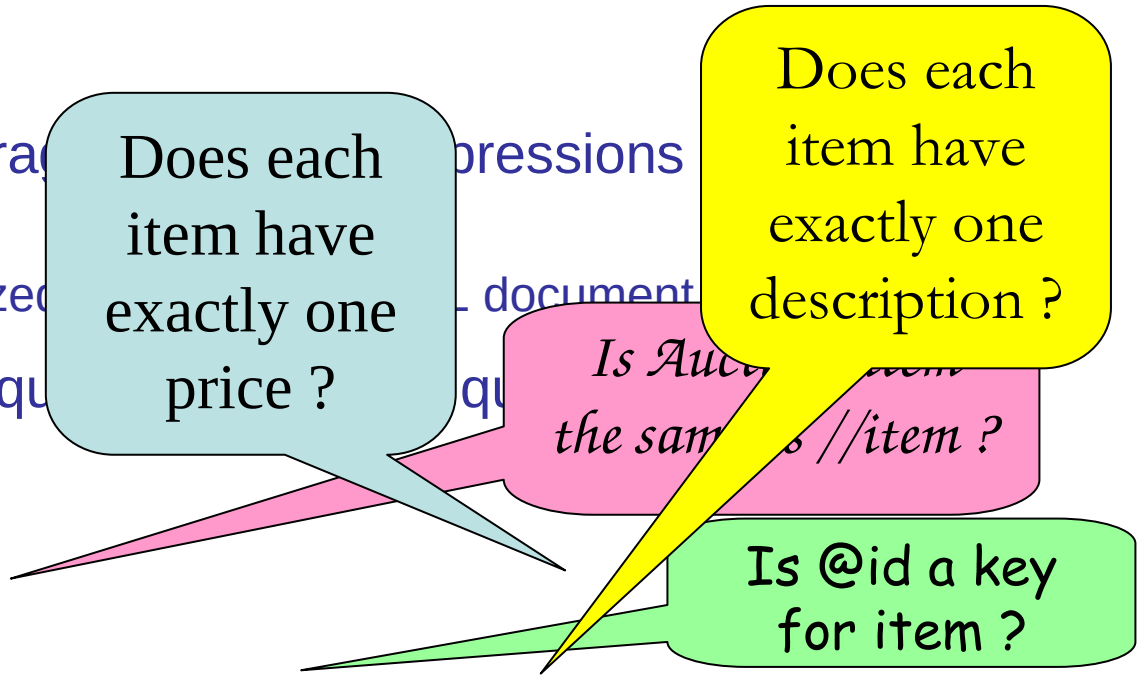
XPath containment

Functional dependency

Cardinality constraints

Query containment /rewriting under constraints

Techniques based on the chase [DS03]



User-defined XML-to-relational mappings [MFK01,DS03]

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

- Relation = materialized view over the XML document

Other issue: we would want to write

R(**XID**, y,z) :- Auctions.item x, **x.id XID**, x.@id.text() y, x.price.text() z

S(**XID**, u,v) :- Auctions.item t, **x.id XID** t.@id.text() u, t.description.text() v

x.id not in the document ! But, in the data model [XQDM]

Summary: user-defined XML-to-relational 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...

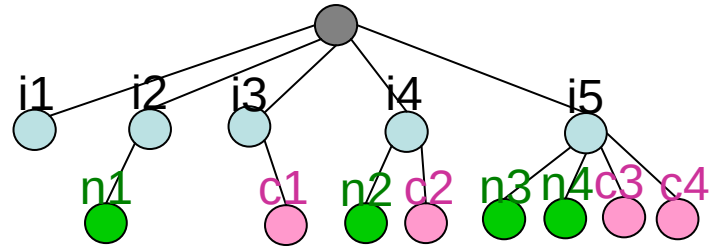
The reconstruction issue

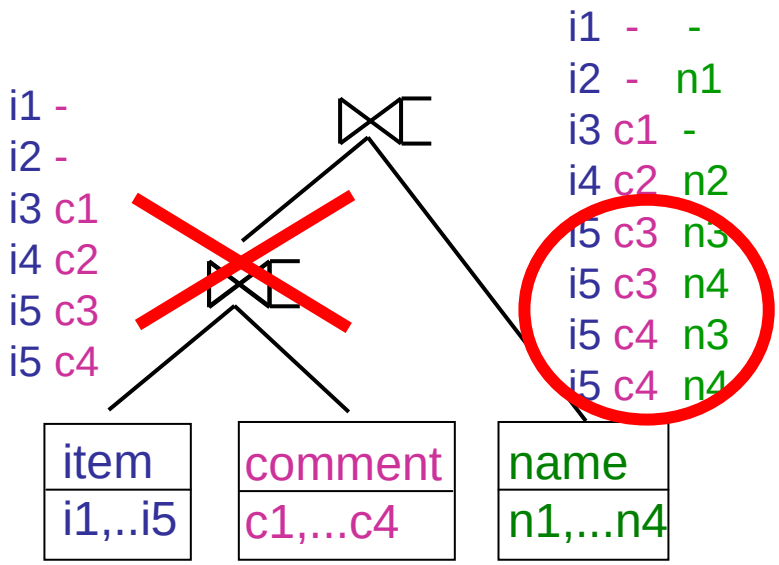
Query: /Auctions/item

This concerns:

- /Auctions/item
- /Auctions/item/name
- /Auctions/item/@id
- /Auctions/item/comment
- /Auctions/item/description
- /Auctions/item/description/text
- /Auctions/item/description/parlist
- /Auctions/item/description/parlist/listitem
- .....

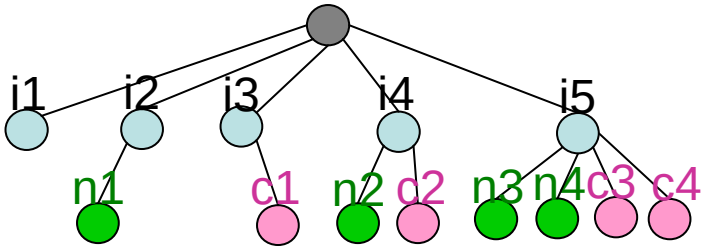
Simplification: item, comment, name

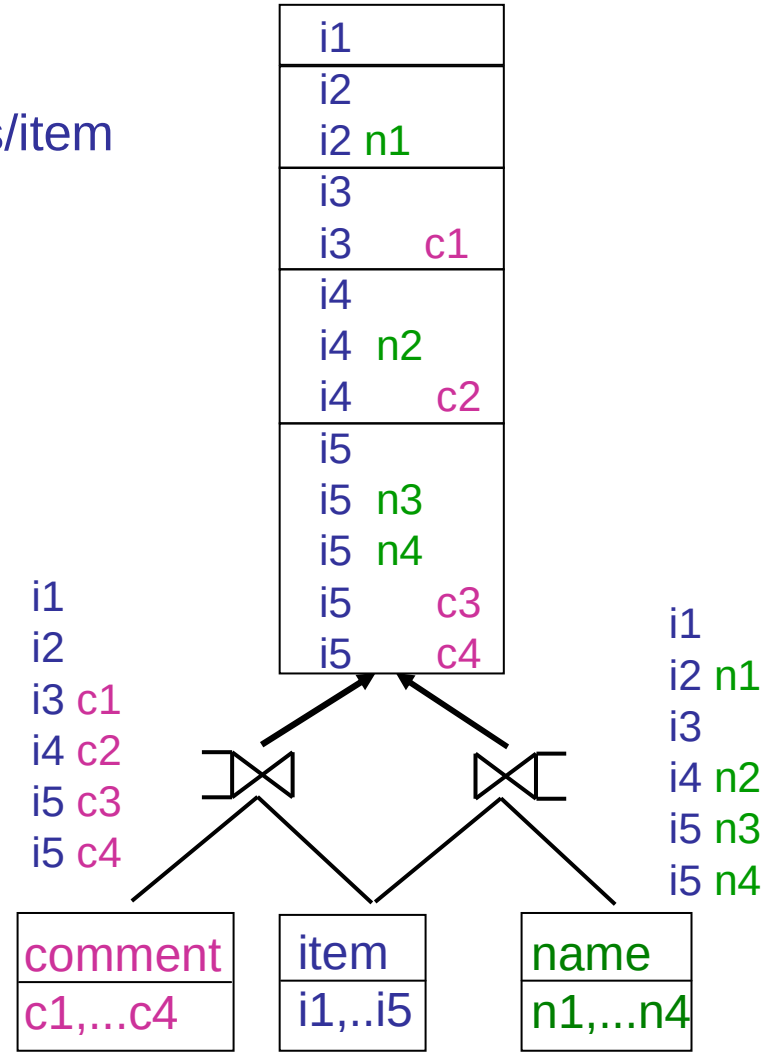




The reconstruction issue

/Auctions/item

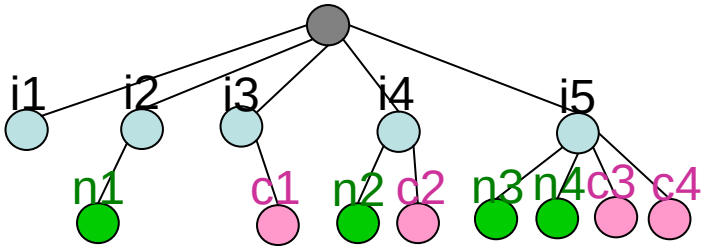




The reconstruction issue

/Auctions/item

i1
i2
i2 n1
i3
i3 c1
i4
i4 n2
i4 c2
i5
i5 n3
i5 n4
i5 c3
i5 c4



i1
i2
i3 c1
i4 c2
i5 c3
i5 c4

i1
i2 n1
i3
i4 n2
i5 n3
i5 n4

comment
c1,...c4

Plan $\mathcal{P}$

name
n1,...n4

Two alternatives:

- Run $\mathcal{P}$ twice
- Materialize intermediary results

Result of fragmentation

Same problem for complex returned results

Summary of relational storage models for XML

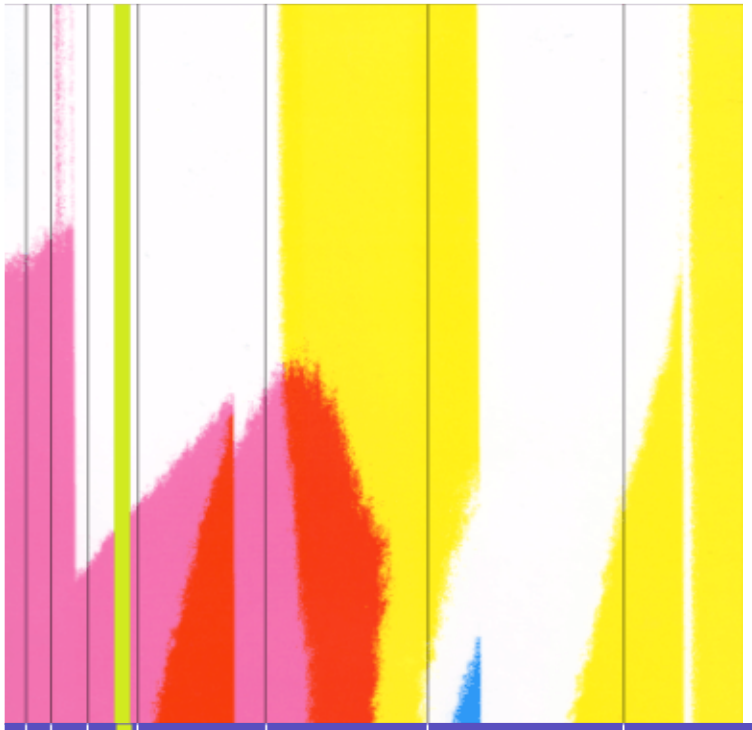
Relations *alone* only go that far

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*



2.3
Native stores for XML: storage, indexing, and query processing primitives

INSTITUT NATIONAL
DE RECHERCHE
EN INFORMATIQUE
ET EN AUTOMATIQUE

INRIA

Plan

Labeling schemes

Query primitives: structural joins

Native storage models for XML

Indexing models for XML

Conclusions

Why start with labeling schemes ?

Idea: assign labels to XML elements

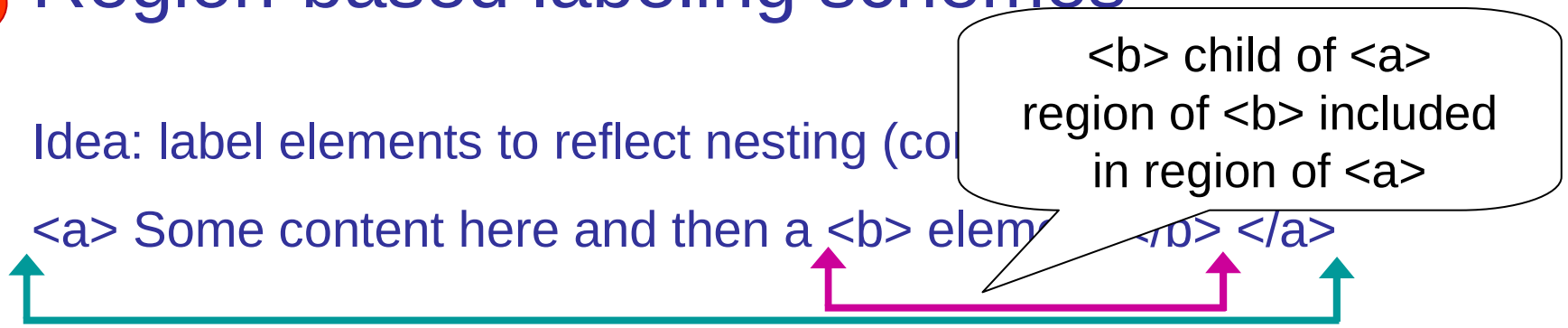
- unique identifiers +
- useful information for query processing

Source of big performance improvements over relational storage + traditional joins

Many labeling schemes

- trade-off between space occupancy, information contents, and suitability to updates [CKM02]
- most frequent one: region-based ("pre-post")
  - shortcomings and alternatives
- new ones still being produced

Region-based labeling schemes



Idea: label elements to reflect nesting (co

<a> Some content here and then a elem

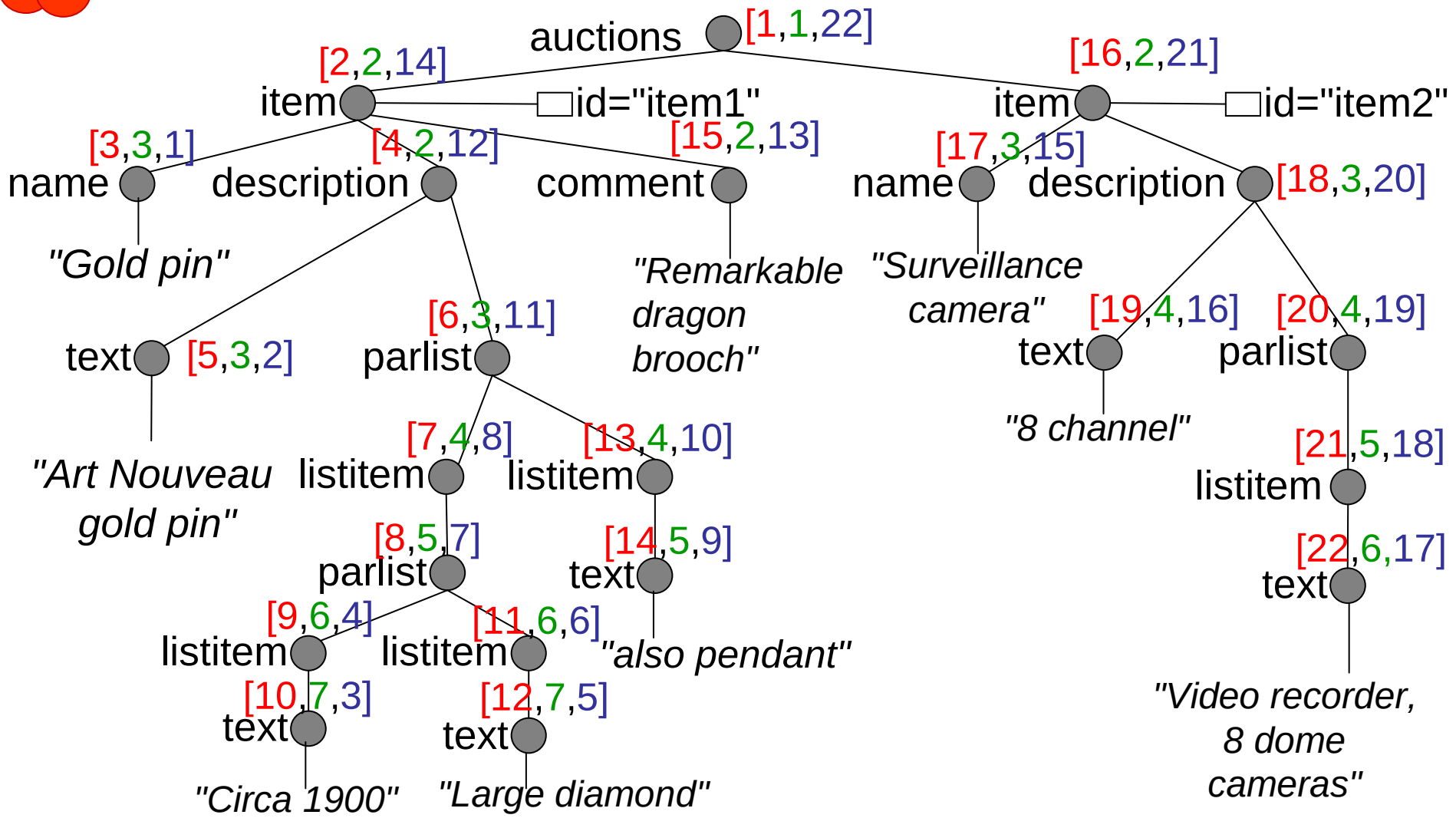
Label <a> with [1, 2], and with [2, 1]

Add also nesting level

- label <a> with [1, 2, 1]
- label with [2, 1, 2]

Variant: *start* and *end* counting in characters in the file (recycled IR indexing technique)

Region-based structural identifiers



The interest of region-based identifiers

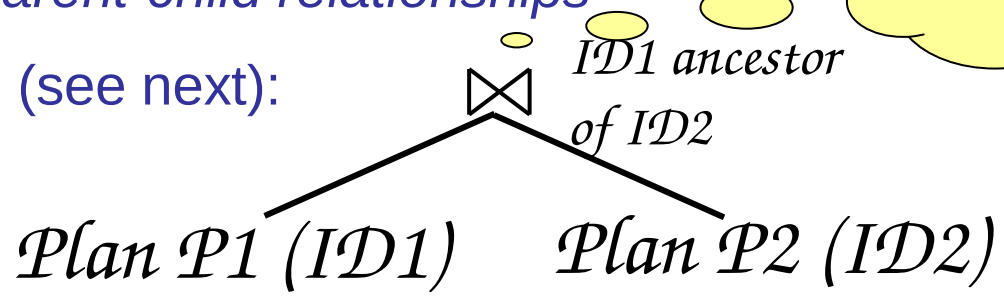
Element e1 is an ancestor of element e2 iff
e1.pre < e2.pre and e1.post > e2.post

Element e1 is a parent of element e2 iff
e1.pre < e2.pre and e1.post > e2.post and e1.depth+1=e2.depth

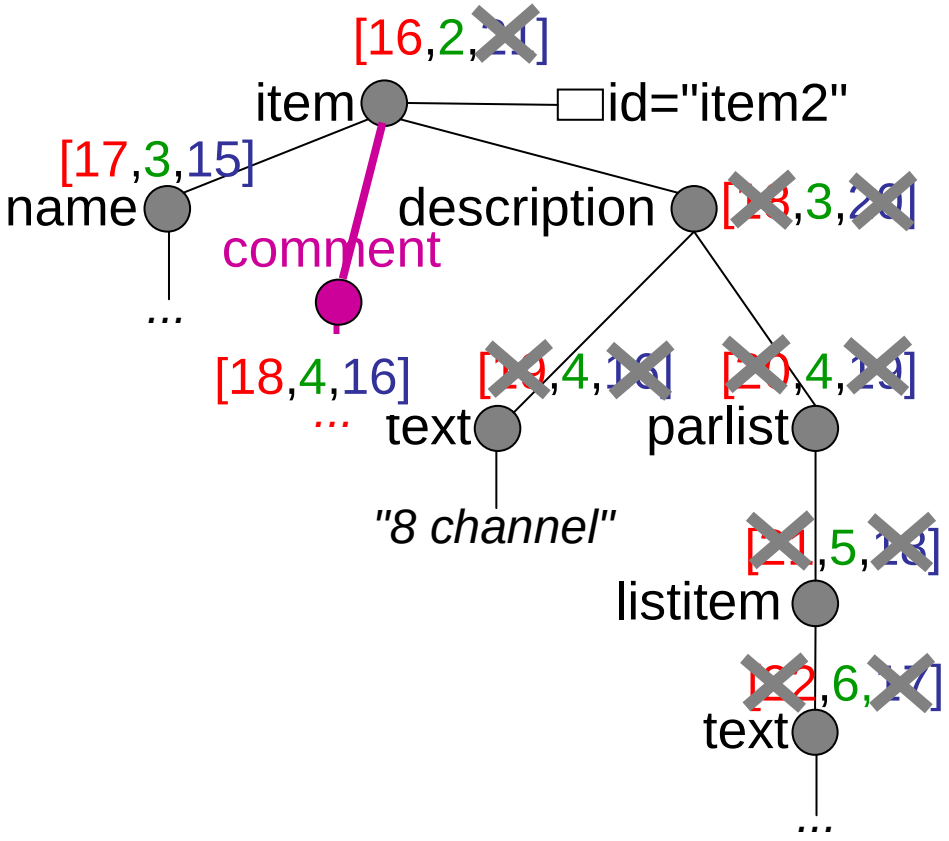
Establishing ancestor-descendent relationships b
establishing parent-child relationships

The end
of navigation !

Structural join (see next):



Region-based identifiers and updates



Possible solutions:

- Leave empty intervals [LM+01]
- Use real numbers [JKC+02]
- Multi-versioning [CTZ+02]
- Append another "discriminating" label to [pre, depth, post]

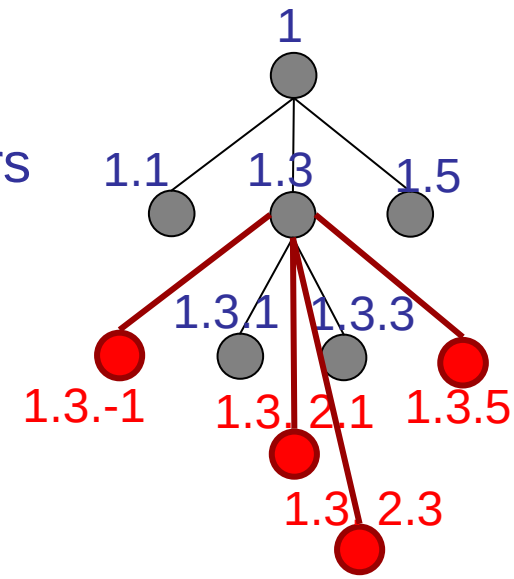
Other labeling scheme: ORDPATHS [NNP+04]

Implemented in MS SQL Server

Label each node by a sequence of integer numbers

- Initial loading: all odd numbers

This scheme records: order; depth; parent; ancestor-descendent relationships.



Insert first sibling: decrease leftmost code by 2

Insert last sibling: increase rightmost code by 2

Insert between siblings: use even numbers (does not affect node depth)

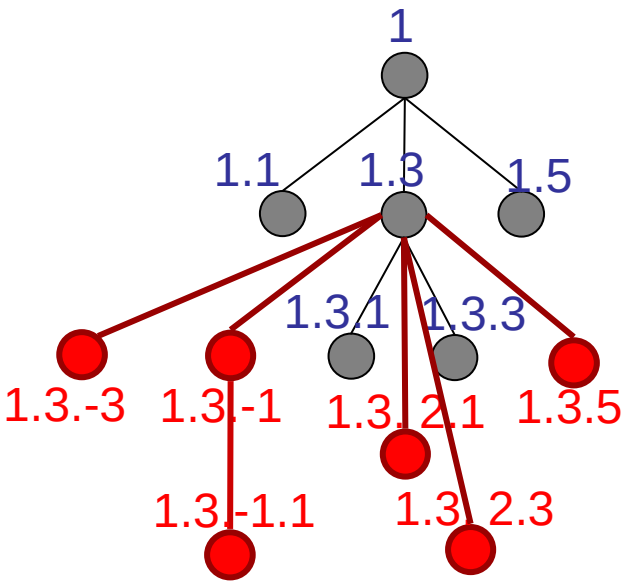
Internal representation of ORDPATHs

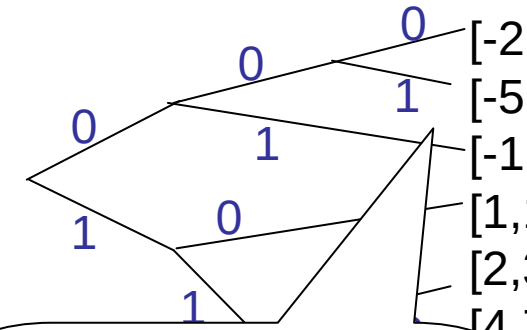
L0 O0 L1 O1 ... Li Oi ... where:

- Oi is the i-th number
- Li encodes Oi's length
- Li's must satisfy prefix property

Partition [-max, max] in intervals of length 2^i

Build prefix tree on the partitioning



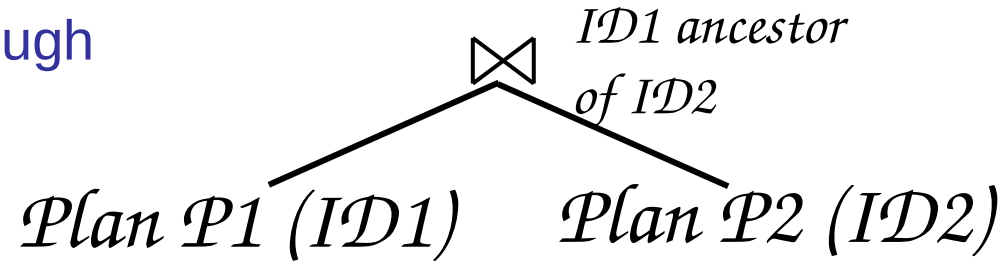


Moderate length on real-life documents [MBF03]

[-21,-6]	If Oi∈[-21,-6], Li=000, -21 is 0000, ..., -6 is 1111
[-5,-2]	If Oi∈[-5,-2], Li=001, -5 is 00, ..., -2 is 11
[-1,0]	If Oi∈[-1,0], Li=01, -1 is 0, 0 is 1
[1,1]	If Oi∈[1,1], Li=10, Oi is not represented
[2,3]	If Oi∈[2,3], Li=110, 2 is 0, 3 is 1
[4,7]	If Oi∈[4,7], Li=1110, 4 is 00, 7 is 11
,23]	If Oi∈[8,23], Li=1111, 8 is 0000, 23 is 1111

Query primitive: structural join

Relationship established through simple comparisons




```
select *
from Element e1, Element e2
where e1.doc = e2.doc and e1.pre < e2.pre and e1.post > e2.post
```

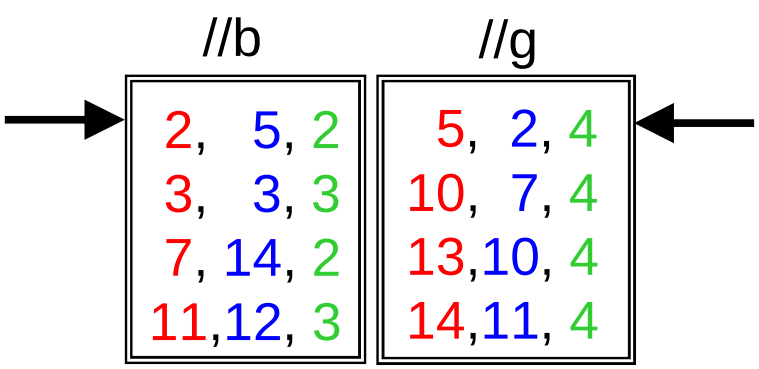
Non-equality join !

Relational implementations perform poorly [ZND+01]

Alternative efficient *stack-based algorithms* [AJK+02]

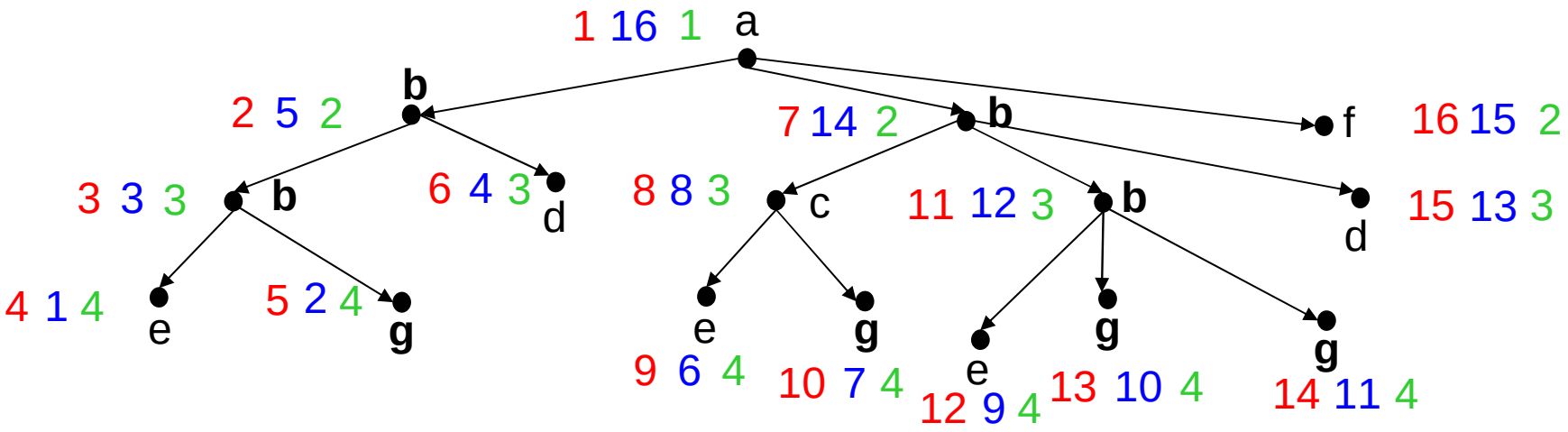
- Algorithm StackTreeDesc [AJK+02], holistic twig joins [BKS02]
- Supporting indexes [JLW+03], join ordering [WPJ03]

Algorithm StackTreeDesc

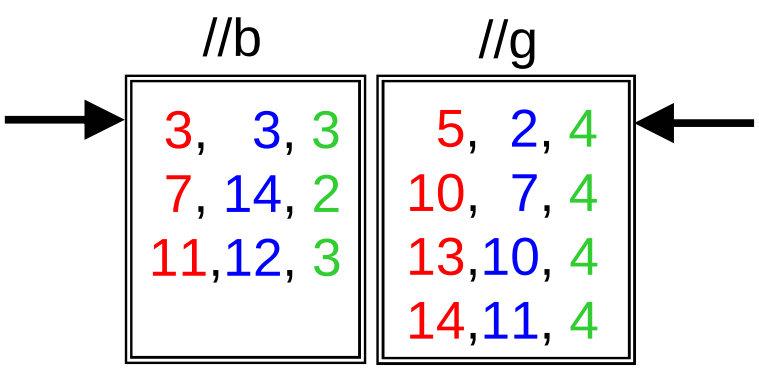


Stack

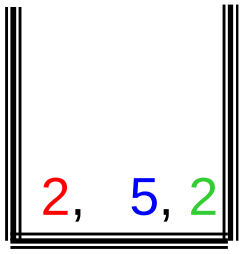
Result



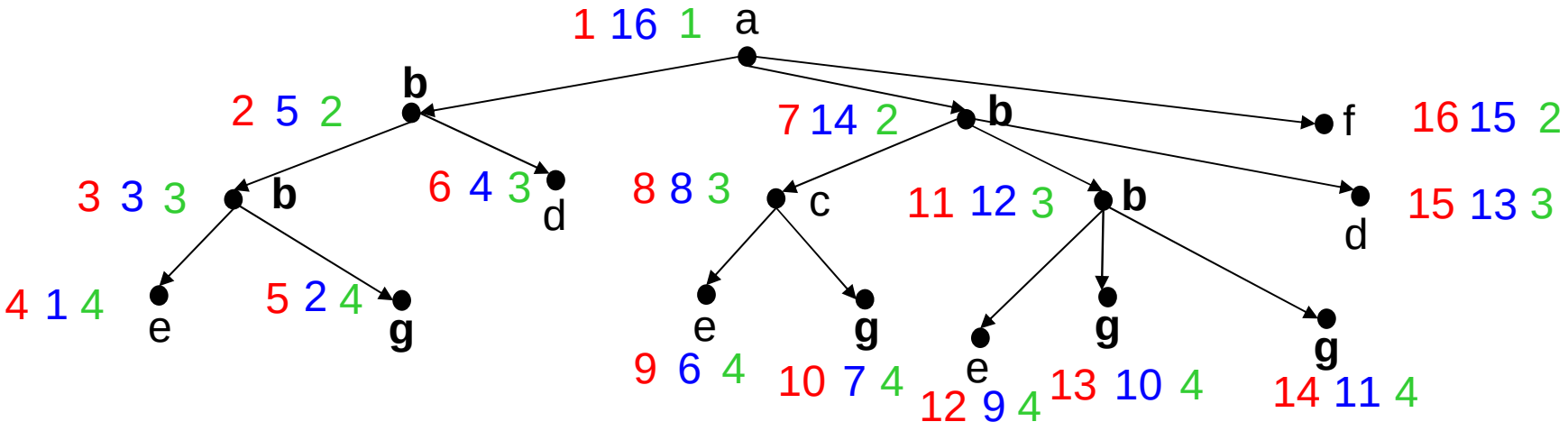
Algorithm StackTreeDesc



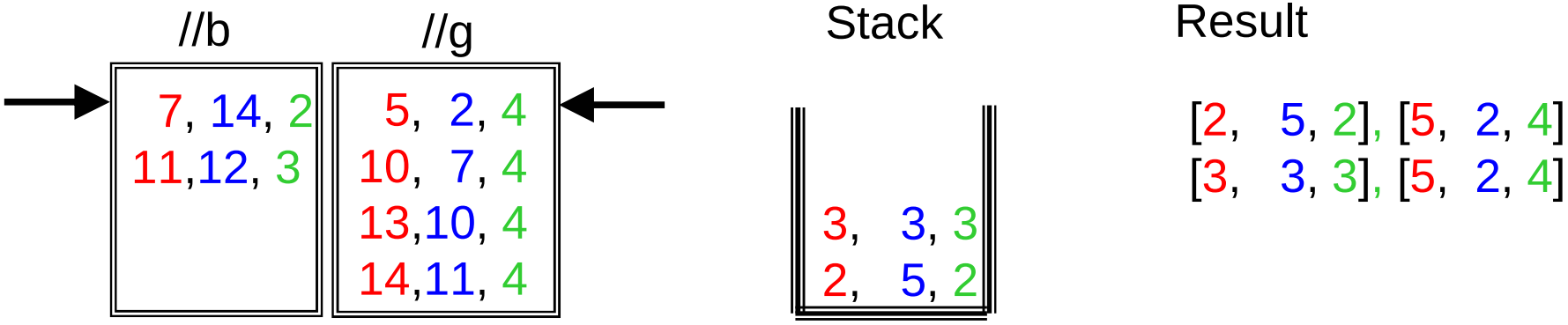
Stack

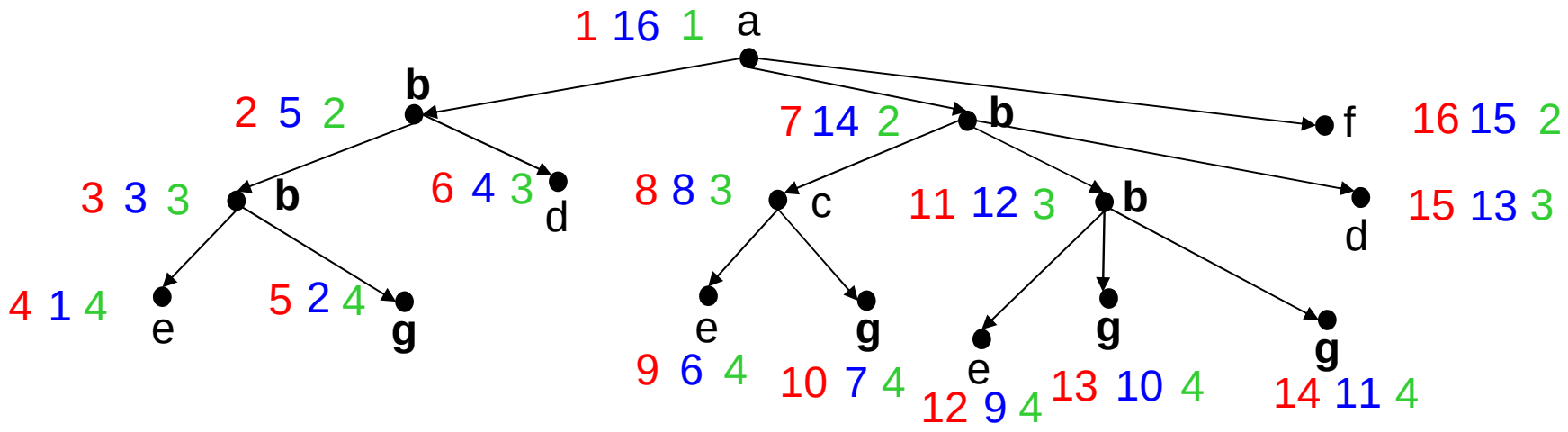


Result

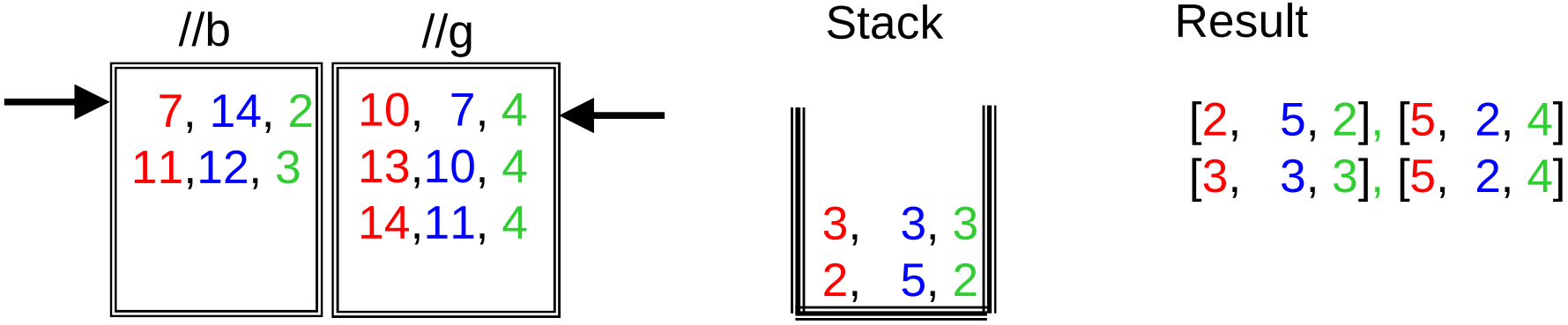


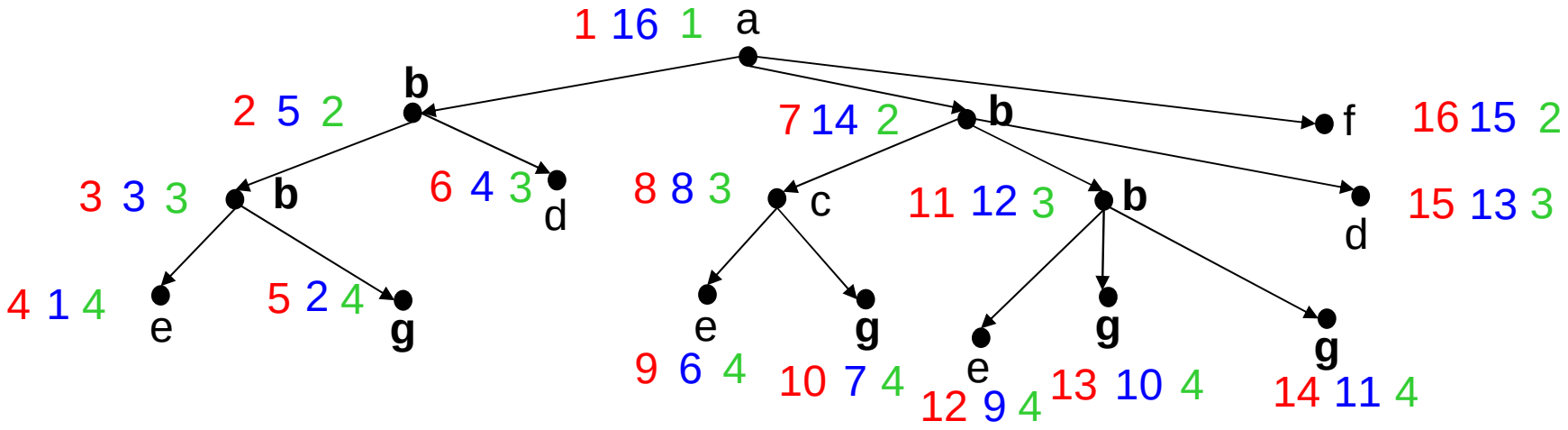
Algorithm StackTreeDesc



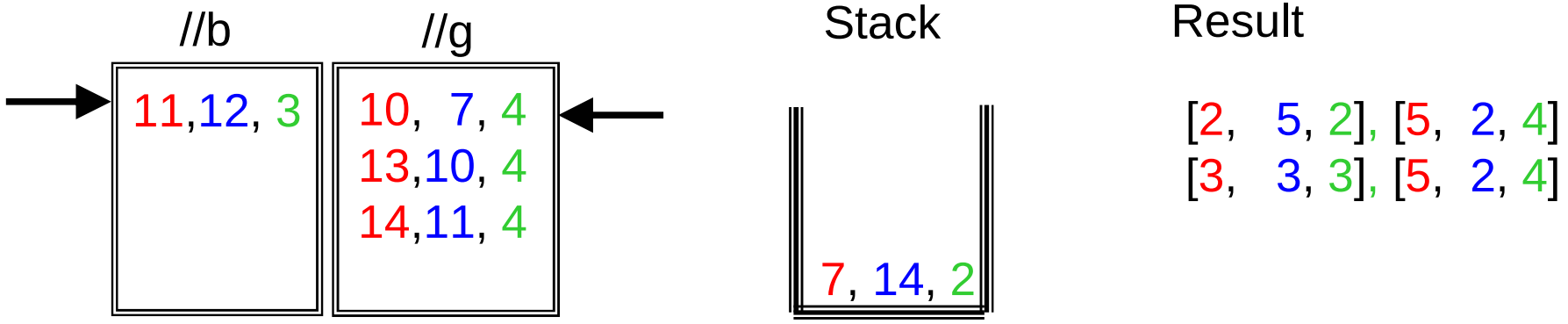


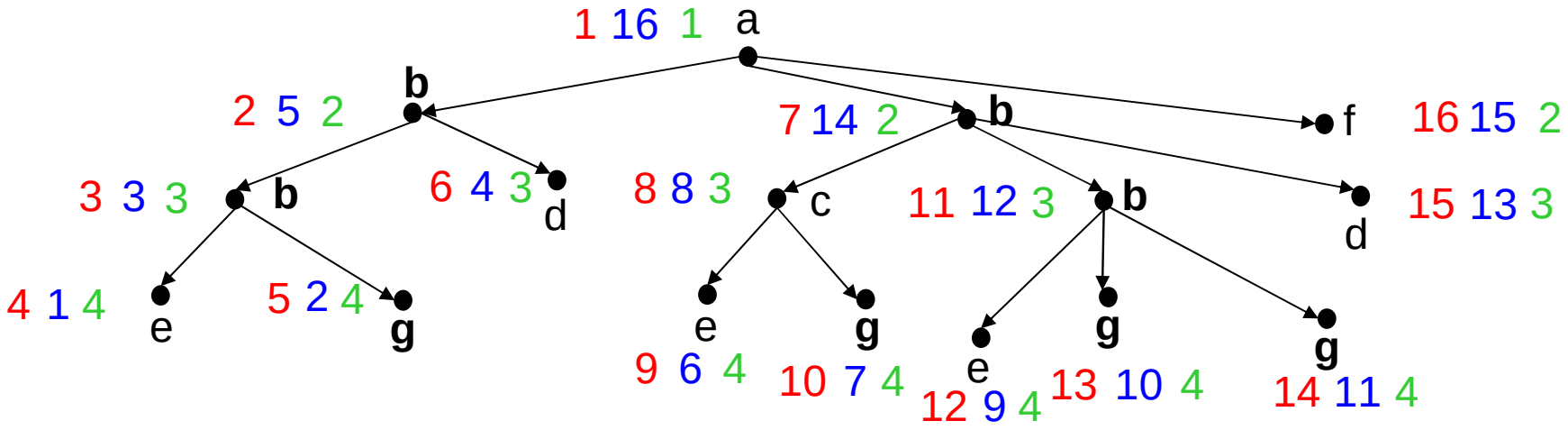
Algorithm StackTreeDesc



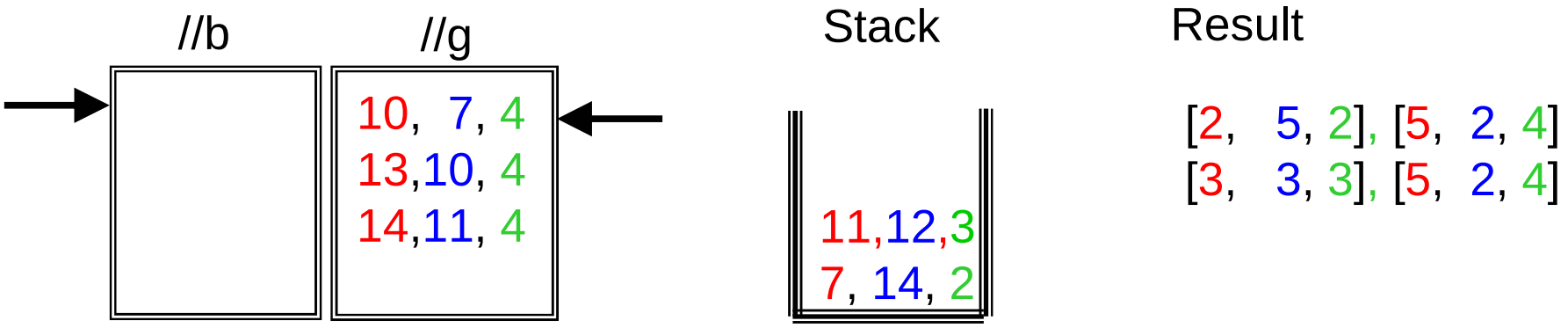


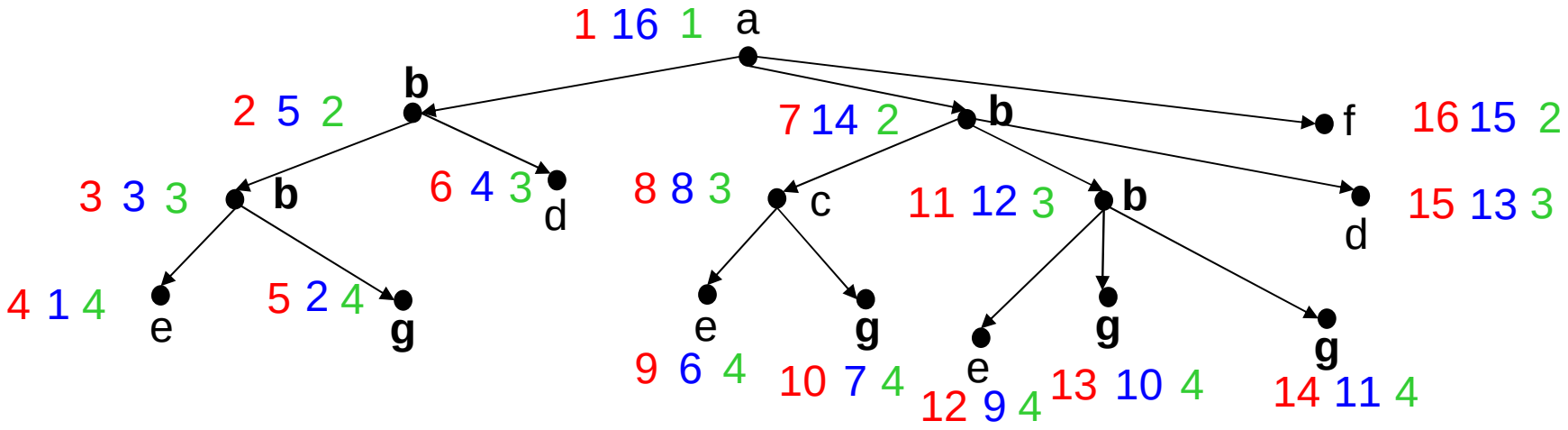
Algorithm StackTreeDesc



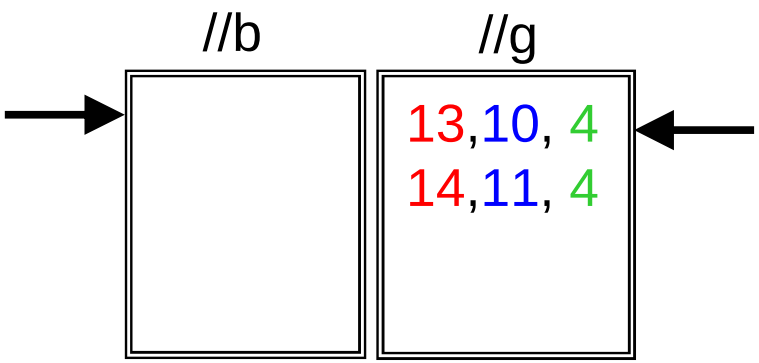


Algorithm StackTreeDesc

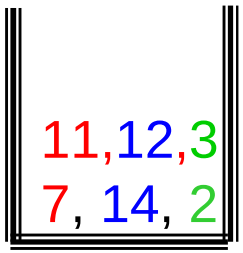




Algorithm StackTreeDesc

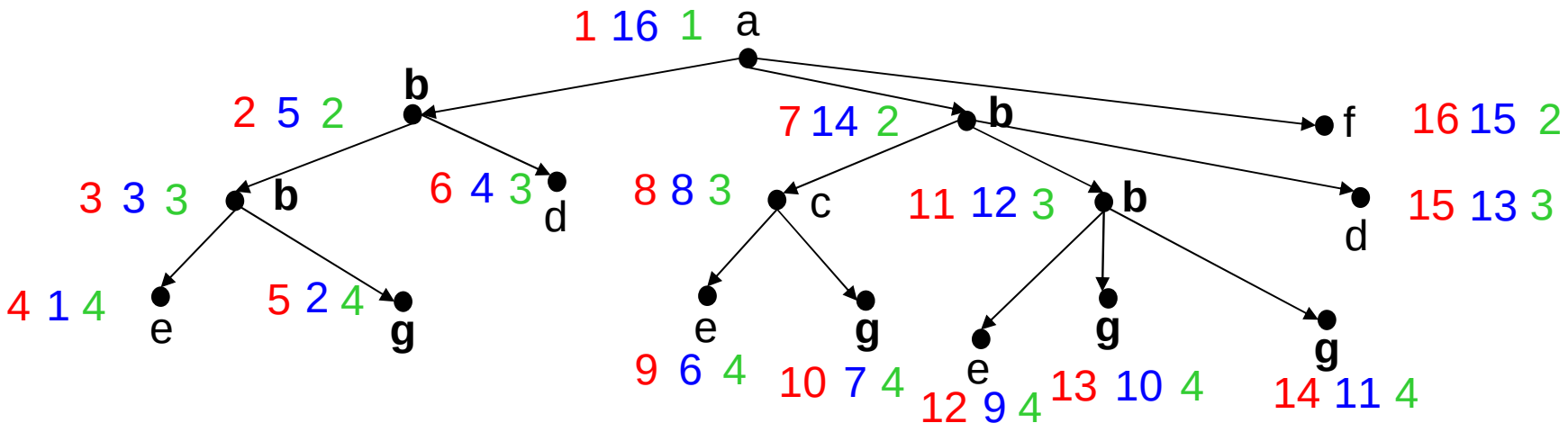


Stack

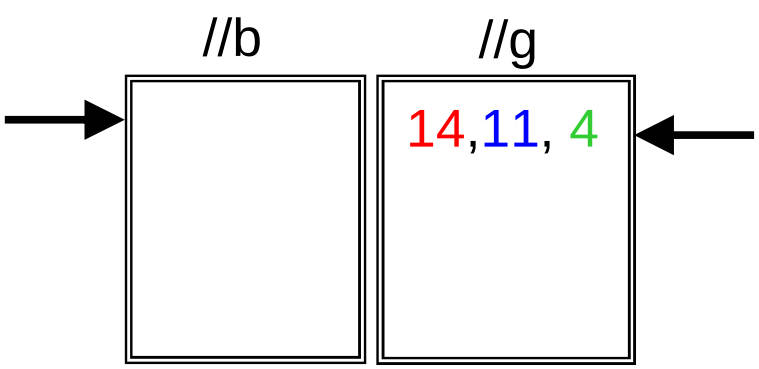


Result

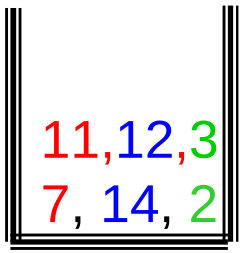
[2, 5, 2], [5, 2, 4]
[3, 3, 3], [5, 2, 4]
[7, 14, 2], [10, 7, 4]
[7, 4, 2], [13,10,4]
[11,12,3], [13,10,4]



Algorithm StackTreeDesc

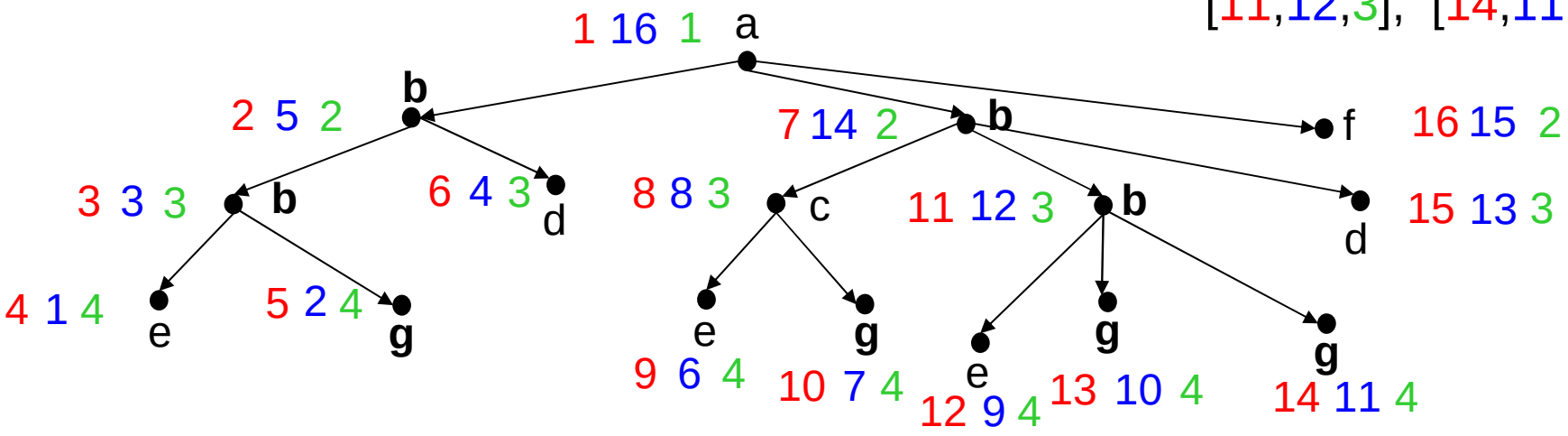


Stack

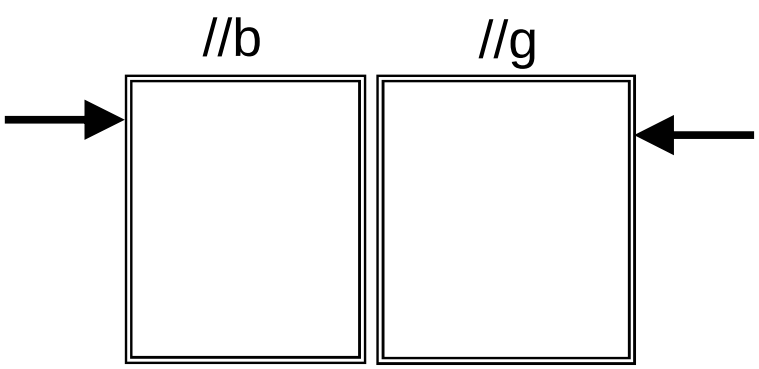


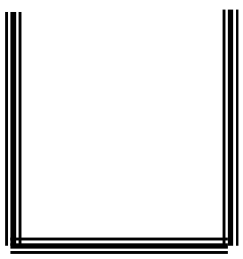
Result

[2, 5, 2], [5, 2, 4]
[3, 3, 3], [5, 2, 4]
[7, 14, 2], [10, 7, 4]
[7, 4, 2], [13,10,4]
[11,12,3], [13,10,4]
[7, 4, 2], [14,11,4]
[11,12,3], [14,11,4]



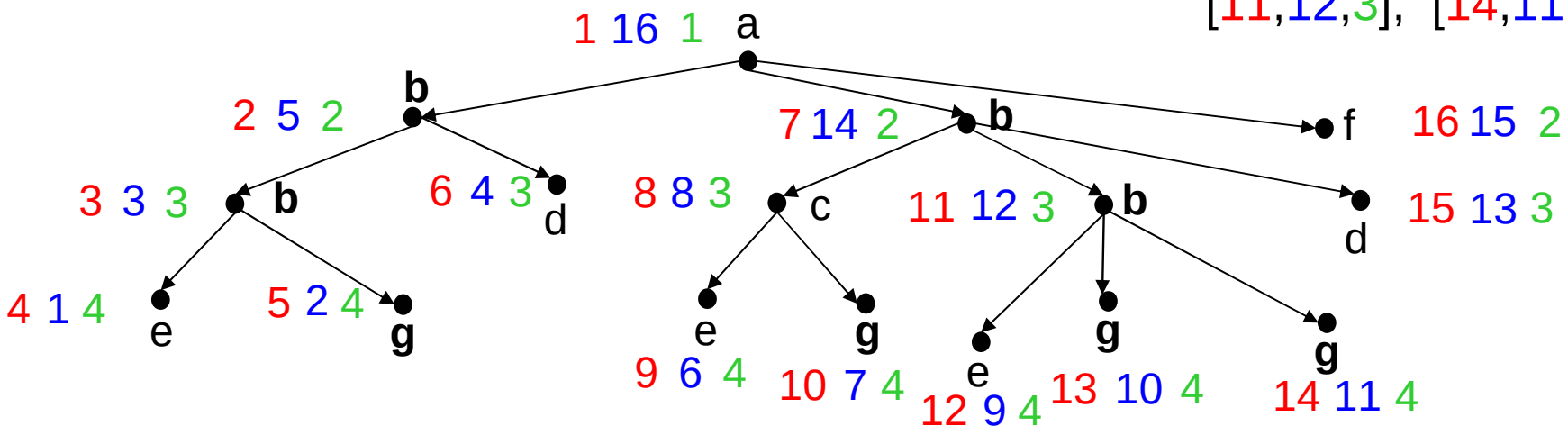
Algorithm StackTreeDesc





Result

[2, 5, 2], [5, 2, 4]
[3, 3, 3], [5, 2, 4]
[7, 14, 2], [10, 7, 4]
[7, 4, 2], [13,10,4]
[11,12,3], [13,10,4]
[7, 4, 2], [14,11,4]
[11,12,3], [14,11,4]



Remarks on StackTreeDesc

I/O cost in O(m+n) + stack

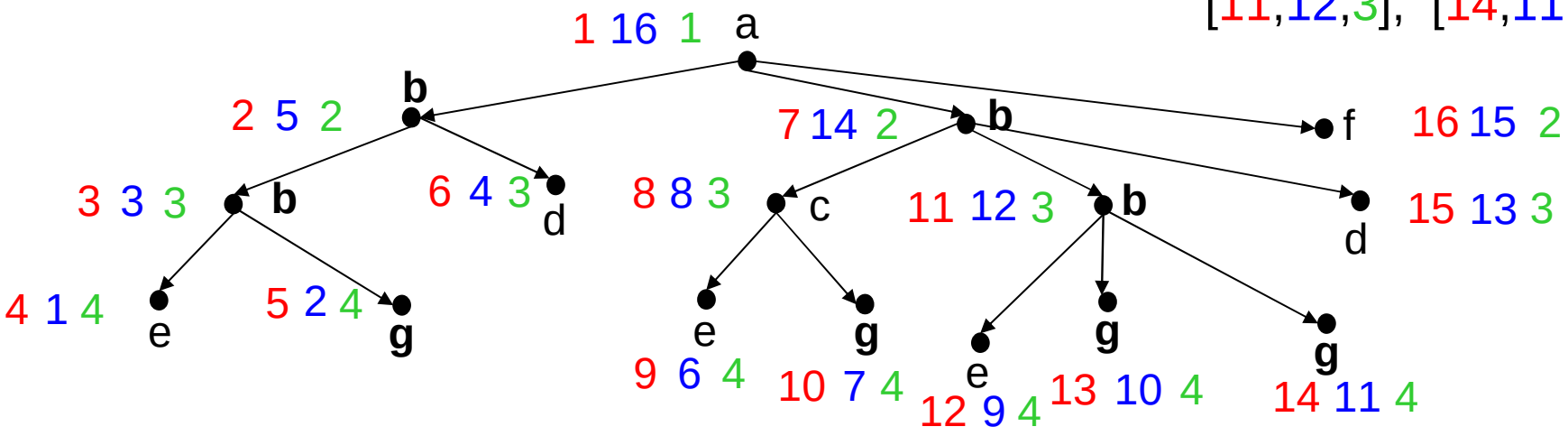
Does *not* compute //b//g (duplicates !)

Returns result in *descendent* order

Similar algorithm for result in *ancestor* order

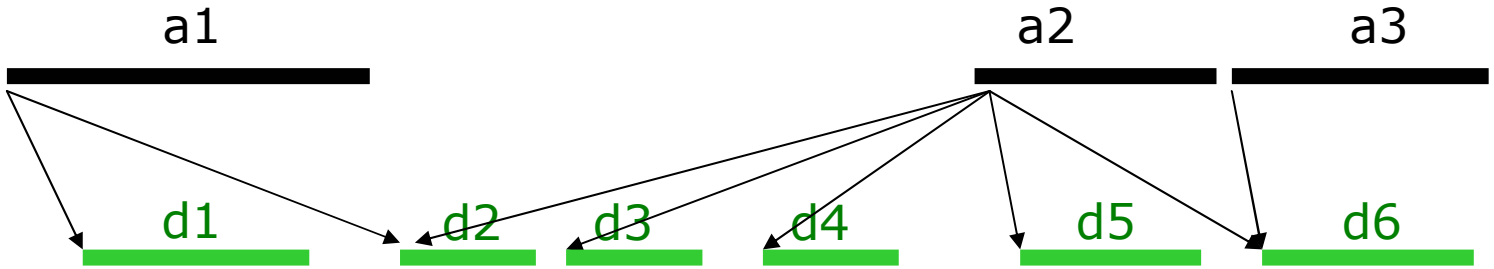
Result //b join //g

[2, 5, 2], [5, 2, 4]
[3, 3, 3], [5, 2, 4]
[7, 14, 2], [10, 7, 4]
[7, 4, 2], [13,10,4]
[11,12,3], [13,10,4]
[7, 4, 2], [14,11,4]
[11,12,3], [14,11,4]



Skipping structural joins

Avoid useless comparisons: avoid scanning all the input



Can be replaced with:

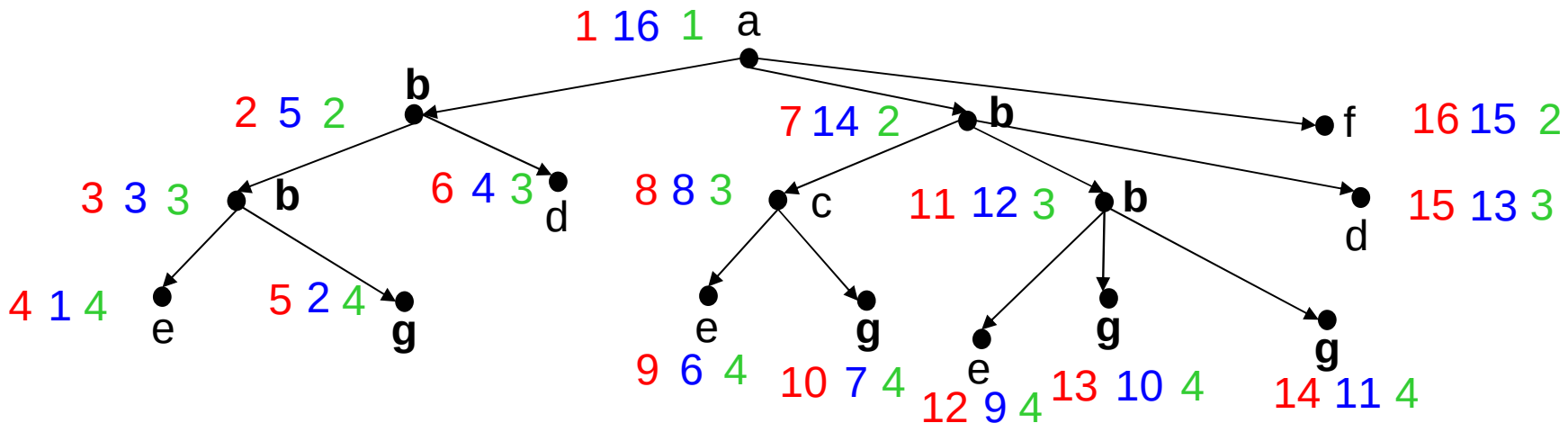
This requires searching the descendent by pre

Holistic twig joins [BKS02]

Idea: avoid constructing intermediary results when matching twig patterns

E.g. for $x in //b, $y in $x//e, $z in $x//d...
Avoid constructing ($x, $y) pairs for $x without $z bindings

1 holistic twig join operator with 3 inputs, 1 stack per input



Structural joins: summary [BKS02]

Linear complexity join algorithms based on region identifiers

Sub-linear variants exist, based on *skipping*

- Algorithmic vs. disk I/O reduction
- Based on ordered storage/ind

Holistic twig joins reduce intermedia esults

Cost of matching a twig patt n =

- Data access cost +
- Join cost (including intermediary results) +
- [Sort cost] + [Duplicate elimination cost]

Depends on storage/index

Depends on algorithm

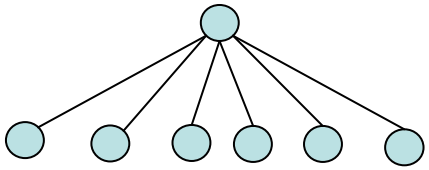
Depends on join order

Native storage models for XML

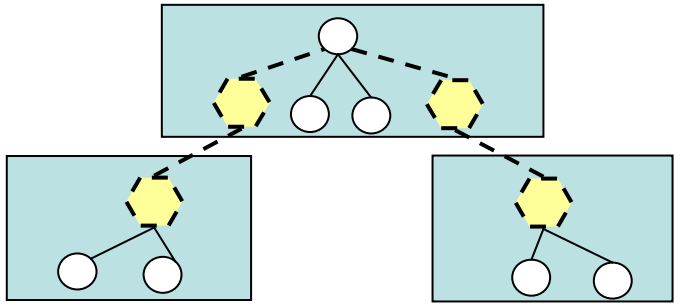
Native storage = persistent trees [JKC+02,FHK+02]

XML nodes split among disk pages

Logical XML document



Physical representation



Tags encoded with a dictionary

Node representation optimized based on fixed page size

Storage evolution on updates ~ B-trees [FHK+02]

Native XML storage and indexing models

Persistent trees only allow navigation... remember Lore ?

Approach taken in Timber and Natix:

- Attach region-based labels to XML elements
- Twig pattern matching = structural join or region labels
- *Index labels by tag.* This largely outperforms navigation [HBG+03].

Another approach: partition region labels by incoming path (as *storage*) [MAB+04]

- More concise (tags stored in schema)
- Less duplicate elimination, joins; more unions required

More native encoding & storage & indexing schemes

[WKF+03], [K03] based on binary tree encoding, [RM04],[ZKO04]

Path-partitioned storage with structural identifiers [MAB+04]

XQueC: started out as an XML compression + querying project

Assign region identifiers to all elements

Group together IDs found under the same exact path

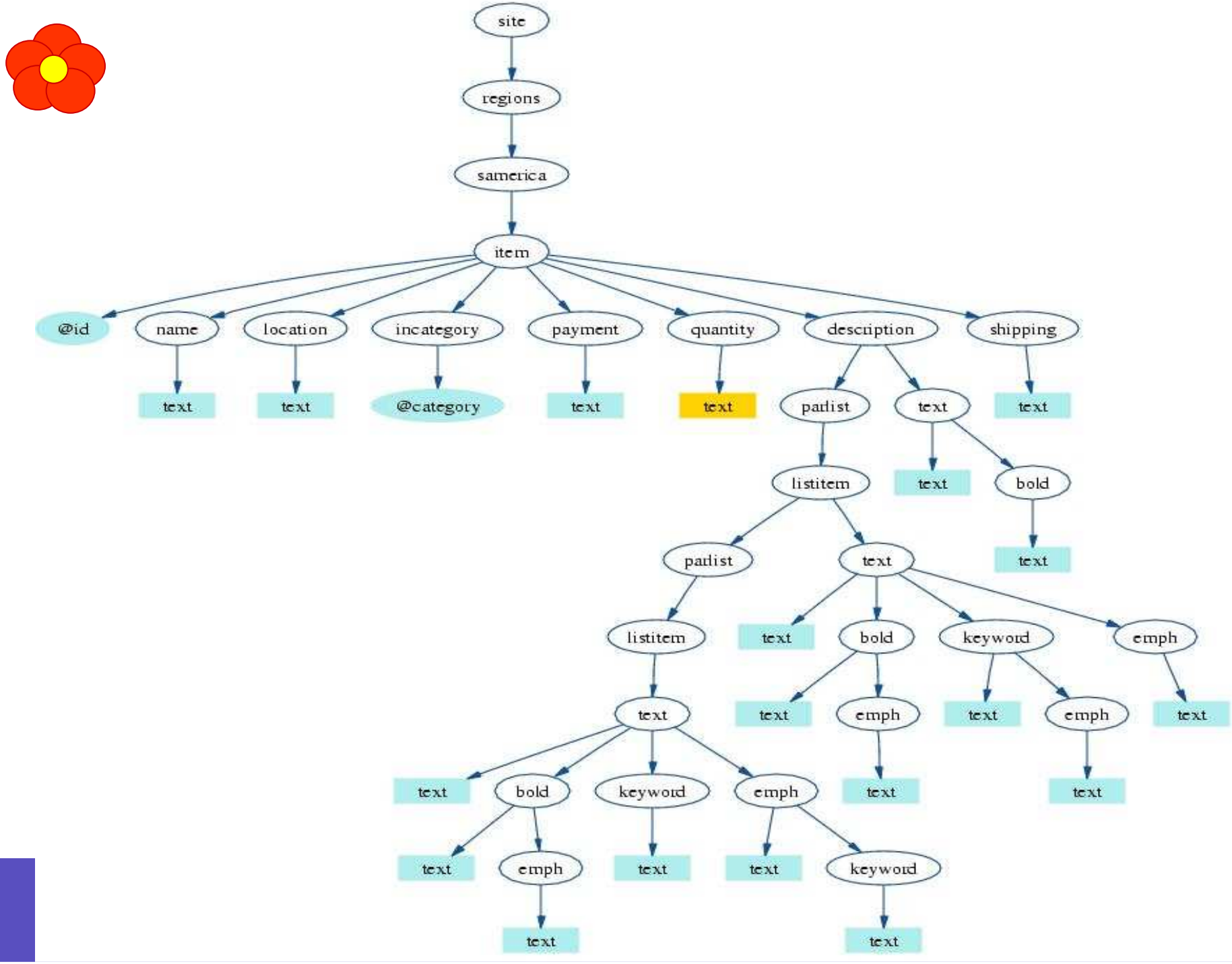
- Equivalence: incoming path (1-Index coincides with DataGuide)
- Store an ID sequence per path

Group together values found under the same exact path

- Store a (pre, value) *sequence* per path

Keep path summary (dataguide) with

- *Structure information*
- *Cardinality constraints: min(a,b), max(a,b) for parent-child a,b*


site
regions
samerica
item
@id
name
location
incategory
payment
quantity
description
shipping
text
text
@category
text
text
parlist
text
text
listitem
text
bold
text
parlist
text
listitem
text
bold
keyword
emph
text
text
emph
text
emph
text
text
bold
keyword
emph
text
emph
text
text
text
text
text

Query processing on path-partitioned storage

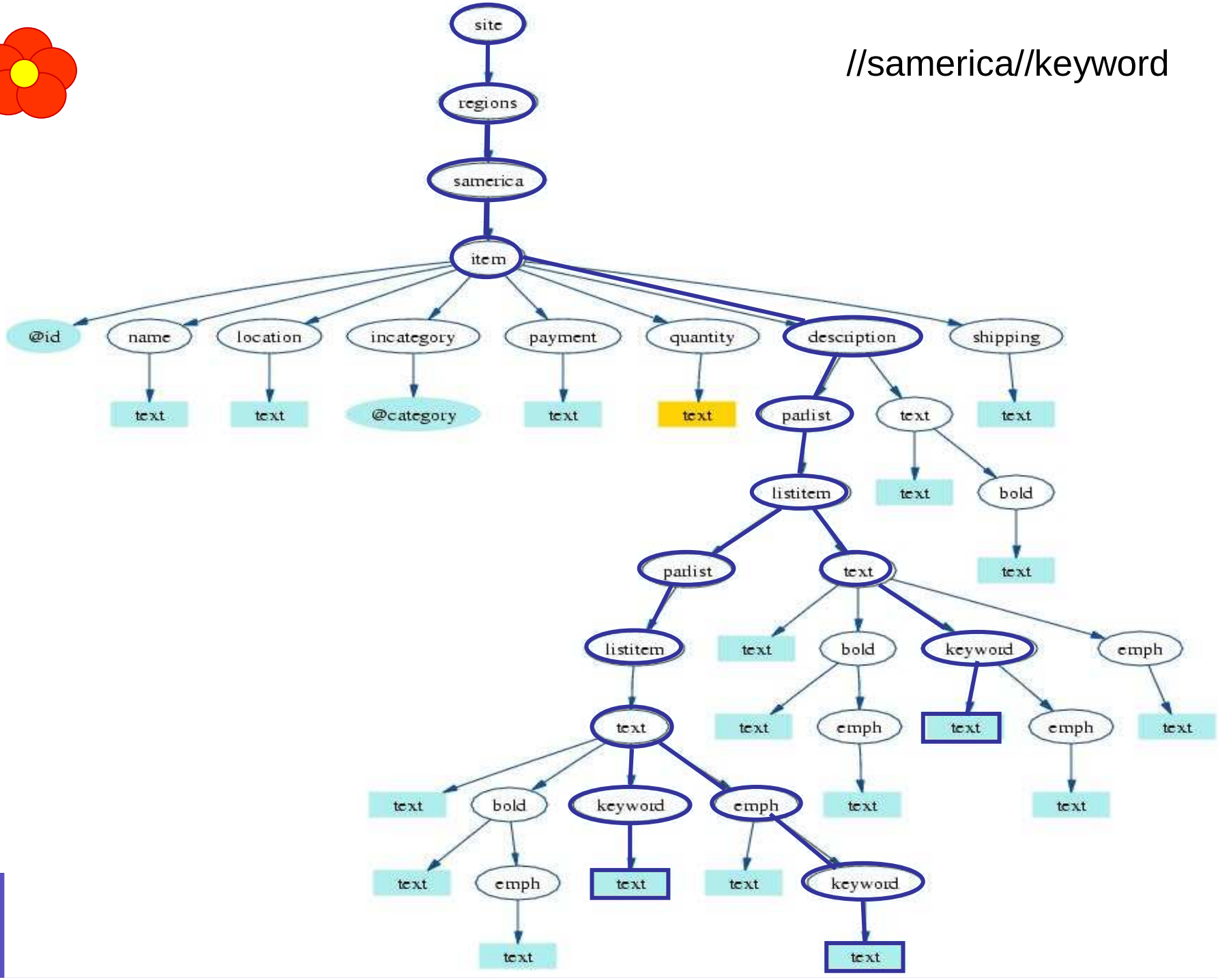
Linear path queries: / or //-separated labels

- Match the query on the path summary -> summary nodes
- Merge the ID sequences of the corresponding summary nodes
- No sort, duplicate-elimination needed

More complex XQuery queries

- Infer possible paths for every variable
- Prune some paths based on path summary
  ~ constraint-based minimization
  ~ query simplification
- Scan and join just IDs on the proper paths

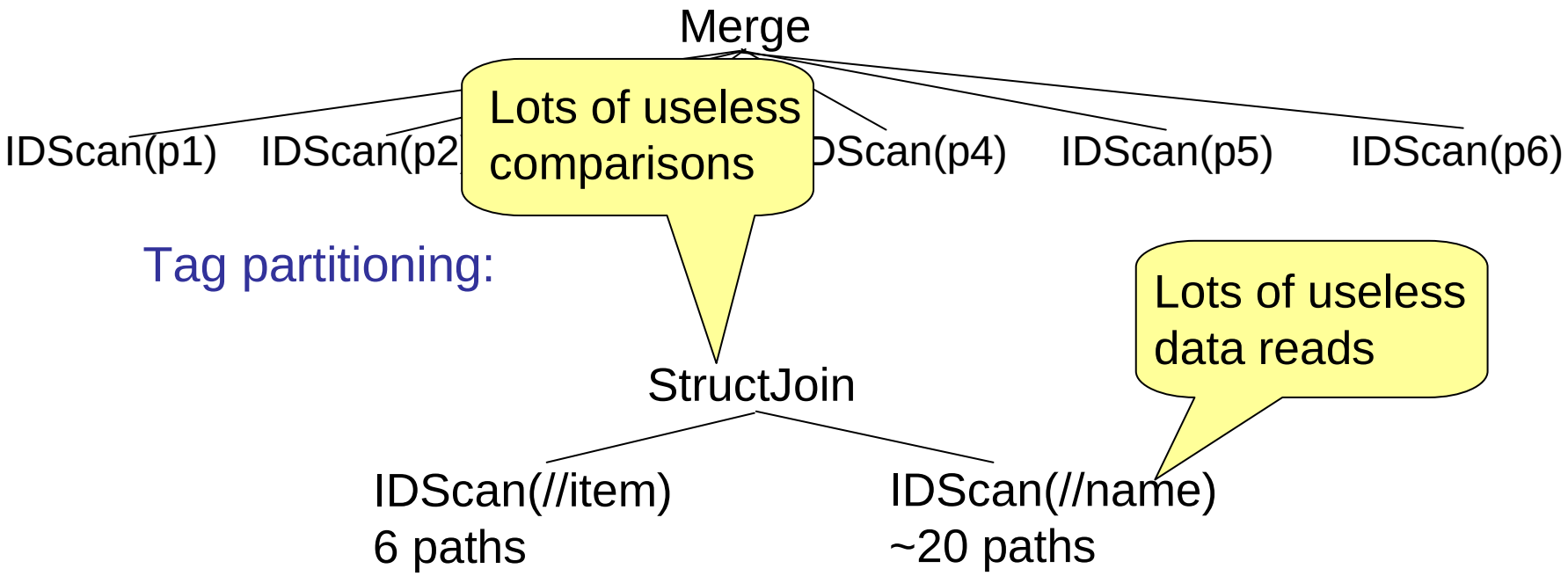
//samerica//keyword



Query processing based on a path summary

for $n in //item/name...

Paths for $n: {/site/regions/africa/item/name,
/site/regions/asia/item/name,...
/site/regions/samerica/item/name}



Query processing based on a path summary

for $v in (/|//) t1 (/|//) t2 (/|//) t3 ... (/|//) tk...

- Path partitioning: 1 merge
- Tag partitioning: extra data reads + (k-1) way join

for $v1 in p1, $v2 in $v1/p2, ..., vk in (v1|v2|...vk-1)/pk...

- Path partitioning: k-way join *(need to bind all vi's !)*
- Tag partitioning: m-way join, m=Σ(length(pi))

Minimization

- Eliminate existential branches: //item[name]
- Reduce path sets: //listitem//parlist

Reasoning about duplicates and order

- //anc//desc: *iff anc is not recursive, anc and desc order coincide*

Native storage and indexing models: summary

Overall transformation:

- Document -- logical encoding/labeling -- mapping to physical store

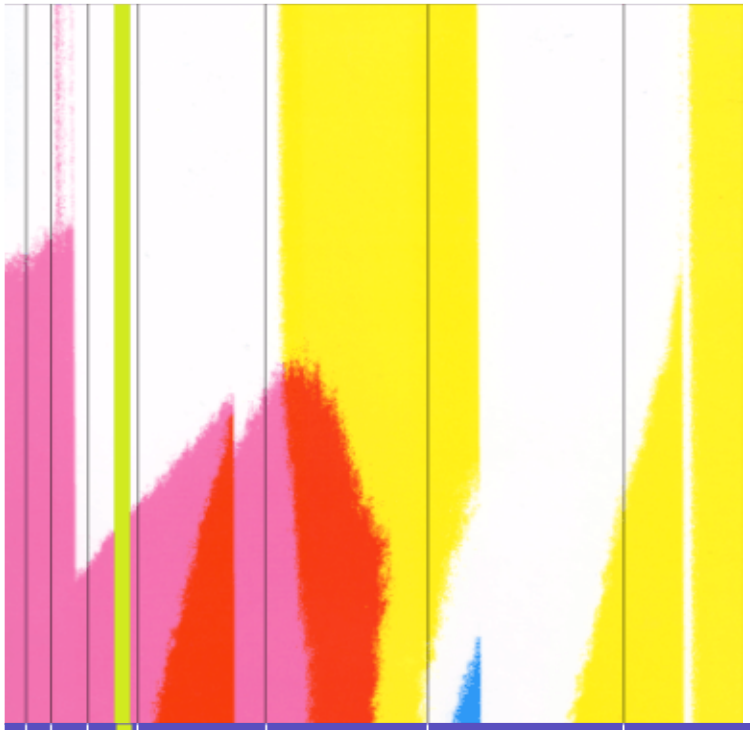
Some labeling schemes reduce the number of joins (ORDPATHS)

Navigation outperformed by index-based access

Persistent trees provide for fast serialization

Construction of new, complex result remains complex

Likely to yield interesting encoding and storage solutions



Summing up

INSTITUT NATIONAL
DE RECHERCHE
EN INFORMATIQUE
ET EN AUTOMATIQUE

INRIA

Summing up

XML databases: if they ever exist, they should have:

1. storage model
2. indexes,
3. materialized views

Right now, work in these directions is well mixed up

The problem is difficult because XML is so far from what should go on disk

- *Verbose* (labels); *type-heterogeneous*
- Difficult to reconciliate *order, conciseness, and updates*
- *Fragmentation favors "for" clause and hurts "return" clause* (selective access vs. fast serialization)

Maybe it's not good for disk (data exchange...) But it raises neat problems.

Still needed in the area of XML storage / indexing / mat. view selection

General methods for identifying useful storage structures

- Combine Shared and XRel and make them work with the IndexFabric (automatically)

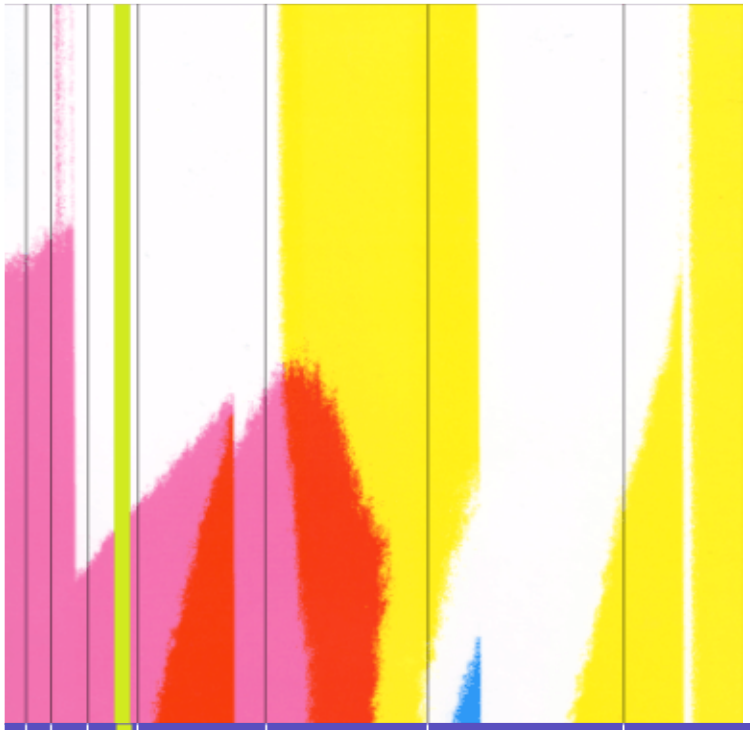
Good will for testing actual queries on actual data

- Not "we picked 5 path expressions of length 3, because we considered them to be representative" (and return node IDs)
- "XMark without presentation tags"
- Many trade-offs exist; do not fill a hole by digging another one

Consensus as to the usefulness of various language schemas

- [MBV03]: DTD 40%, XMLSchema < 1%

Practical, general approach for handling XML constraints as storage metadata (many documents...)



Merci

INSTITUT NATIONAL
DE RECHERCHE
EN INFORMATIQUE
ET EN AUTOMATIQUE

INRIA

References

[ABC+04] A.Arion, A.Bonifati, G.Costa, S.D'Aguanno, I.Manolescu and A.Pugliese. "Efficient Query Evaluation over Compressed XML Data", EDBT 2004

[ABS99] S.Abiteboul, P.Buneman and D.Suciu. "Data on the web: from relations to semistructured data and XML", Morgan Kauffmann, 1999.

[AJK+02] S.Al-Khalifa, H.V.Jagadish, N.Koudas, J.Patel, D.Srivastava and Y.Wu. "Structural Joins: A Primitive for Efficient XML Query Pattern Matching", ICDE 2002

[BCF03] V.Benzaken, G.Castagna and A.Frisch. "CDuce: an XML-centric general-purpose language", ICFP 2003

References

[BFR+02] P.Bohannon, J.Freire, P.Roy and J.Simeon. "From XML Schema to Relations: A Cost-Based Approach to XML Storage", ICDE 2002

[BKS02] N.Bruno, N.Koudas and D.Srivastava. "Holistic Twig Joins: Optimal XML Pattern Matching", SIGMOD 2002

[CKM02] E.Cohen, H.Kaplan and T.Milo. "Labeling Dynamic XML", PODS 2002

[CSF+01] B.Cooper, N.Sample, M.Franklin, G.Hjaltason and M.Shadmon. "A Fast Index for Semistructured Data", VLDB 2001

[CTZ+01] S.Chien, V.Tsotras, C.Zaniolo and D.Zhang. "Efficient complex query support for multiversion XML documents", EDBT 2002

[DFS99] A.Deutsch, M.Fernandez and D.Suciu. "Storing Semistructured Data with STORED", SIGMOD 1999

[DT03] A.Deutsch and V.Tannen. "MARS: A System for Publishing XML from Mixed and Redundant Storage", VLDB 2003

References

[FHK+02] T.Fiebig, S.Helmer, C-C. Kanne, G.Moerkotte, J.Neumann, R.Schiele and T.Westman. "Anatomy of a native XML base management system", VLDB Journal 2002.

[FHK+03] D.Florescu, C.Hillery, D.Kossmann, P.Lucas, F.Riccardi, T.Westmann, M.Carey, A.Sundararajan and G.Agrawal. "The BEA/XQRL Streaming XQuery Processor", VLDB 2003

[FK99] D.Florescu and D.Kossmann. "Storing and Querying XML Using an RDBMS. IEEE Data Engineering Bulletin, 1999.

[FS98] M.Fernandez and D.Suciu. "Optimizing Regular Path Expressions Using Graph Schemas", ICDE 1998

[FSC+03] M.Fernandez and J.Simeon and B.Choi and A.Marian and G.Sur. "Implementing XQuery 1.0: The Galax Experience", VLDB 2003

References

[G02] T.Grust. "Accelerating XPath location steps", VLDB 2002

[GKP02] G.Gottlob, C.Koch and R.Pichler. "Efficient Algorithms for Processing XPath Queries", VLDB 2002

[GW97] R.Goldman and J.Widom. "DataGuides: Enabling Query Formulation and Optimization in Semistructured Databases", VLDB 1997

[HBG+03] A.Halverson, J.Burger, L.Galanis, A.Kini, R.Krishnamurthy, A.Rao, F.Tian, S.Viglas, Y.Wang, J. Naughton and D.DeWitt. "Mixed Mode XML Querey Processing", VLDB 2003

[JLW+01] H.Jiang, H.Lu, W.Wang and J.Yu. "Path Materialization Revisited: An Efficient Storage Model for XML Data", AICE 2001

References

[JKC+02] H.V.Jagadish, S.Al-Khalifa, A.Chapman, L.Lakshmanan, A.Nierman, S.Paparizos, J.Patel, D.Srivastava, N.Wiwatwattana, Y.Wu and C.Yu. "TIMBER: a Native XML database", VLDB Journal 2002

[JLW+03] H.Jiang, H.Lu, W.Wang and B.Ooi. "XR-Tree: Indexing XML Data for Efficient Structural Joins", ICDE 2003.

[K03] C.Koch. "Efficient Processing of Expressive Node-Selecting Queries on XML Data in Secondary Storage: A Tree Automata-based Approach", VLDB 2003

[KBN+02] R.Kaushik, P.Bohannon, J.Naughton and H.Korth. "Covering Indexes for Branching Path Queries", SIGMOD 2002

[LM01] Q. Li and B.Moon. "Indexing and querying XML data for regular path expressions", VLDB 2001

[LMP02] B. Ludascher, P.Mukhopadhyay and Y.Papakonstantinou. "A Transducer-Based XML Query Processor", VLDB 2002

References

[MAB+04] I.Manolescu, A.Arion, A.Bonifati and A.Pugliese. "Path Sequence-Based XML Query Processing", tech. report (BDA) 2004

[MAG+97] J.McHugh, S.Abiteboul, R.Goldman, D.Quass and J.Widom. "Lore: A Database Management System for Semistructured Data", SIGMOD Record 1997

[MBV03] L.Mignet and D.Barbosa and P.Veltri. "The XML Web: a First Study", WWW 2003.

[MFK01] I.Manolescu, D.Florescu and D.Kossmann. "Anwering XML Queries over Heterogeneous Data Sources", VLDB 2001

[MW97] J.McHugh and J.Widom. "Query Optimization for Semistructured Data", tech. report, 1997

[MWA+98] J.McHugh, J.Widom, S.Abiteboul, Q.Luo and A.Rajaraman. "Indexing Semistructured Data", tech. report, 1998

References

[MW99a] J.McHugh and J.Widom. "Query Optimization for XML", VLDB 1999

[MW99b] J.McHugh and J.Widom. "Optimizing Branching Path Expressions", tech. report, 1999

[MS99] T.Milo and D.Suciu. "Index Structures for Path Expressions", ICDT 1999

[OOP+04] P.O'Neill, E.O'Neill, S.Pal, I.Cseri, G.Schaller and N.Westbury. "ORDPATHs: Insert-Friendly XML Node Labels", SIGMOD 2004

[PGW95] Y.Papakonstantinou, H.Garcia-Molina and J.Widom. "Object Exchange Across Heterogeneous Information Sources", ICDE 1995

References

[RM04] P.Rao and B.Moon. "PRIX: Indexing and Querying XML Using Prufer Sequences", ICDE 2004

[SKW+00] A. Schmidt, M.Kersten, M.Windhouwer and F.Waas. "Efficient Relational Storage and Retrieval of XML Documents", WebDB 2000

[STH+99] J.Shanmugasundaram, K.Tufte, G.He, C.Zhang, D.DeWitt and J.Naughton. "Relational Databases for Querying XML Documents: Limitations and Opportunities", VLDB 1999

[WPF+03] H.Wang, S.Park, W.Fan and P.Yu. "ViST: A Dynamic Index Method for Querying XML Data by Tree Structures", SIGMOD 2003

[WPJ03] Y.Wu, J.Patel and H.V.Jagadish. "Structural Join Order Selection for XML Query Optimization", ICDE 2003

References

[YAT+01] M.Yoshikawa, T.Amagasa, T.Shimura and S.Uemura. "XRel: A Path-Based Approach to Storage and Retrieval of XML Documents Using Relational Databases", ACM TOIT 2001

[ZND+01] C.Zhang, J.Naughton, D.DeWitt, Q.Luo and G.Lohman. "On Supporting Containment Queries in Relational Database Management Systems", SIGMOD 2001