Informationssysteme

Kapitel 18 – Verwaltung von XML

18.1 Speicherung von XML in relationalen DBS

18.2 Indizierung von XML

18.3 Komprimierung von XML

18.4 XML-Datenbanken

18.1 Requirements for Storing XML

- Lossless reconstruction of documents
  - order matters only for document-centric documents
  - ignoring comments, processing instructions, entities, …
- Efficient reconstruction of documents
- Efficient query evaluation on documents
  - Boolean queries with XPath
- Efficient updates of documents
  - Addition, deletion of documents
  - Modification of documents

Classification of Storage Methods

- Store complete XML documents in the file system
- Store in relational databases (RDBMS):
  - Store the structure of XML documents (i.e., the XML data graph) in generic tables
  - Derive a schema-specific database schema for storing XML documents (DTD or XML Schema required)

  Big advantage: reuse existing RDBMS infrastructure and experience from the last 20 years
- Natively store XML documents in a dedicated database

  But: re-invent all the auxiliary strucures: indexes, cache, disk organization, transaction management, log, …

XML and RDBMS

XML	Relational databases
hierarchical, arbitrary deep structure	flat, unnested tables
Elements may appear several times	Columns have one fixed value per row
Elements are ordered	Rows are unordered
Schema is optional and may be open	Schema is mandatory
Complex element content with choices	Single schema definition per row

Unclear how to map semistructured XML to well-structured RDBMS

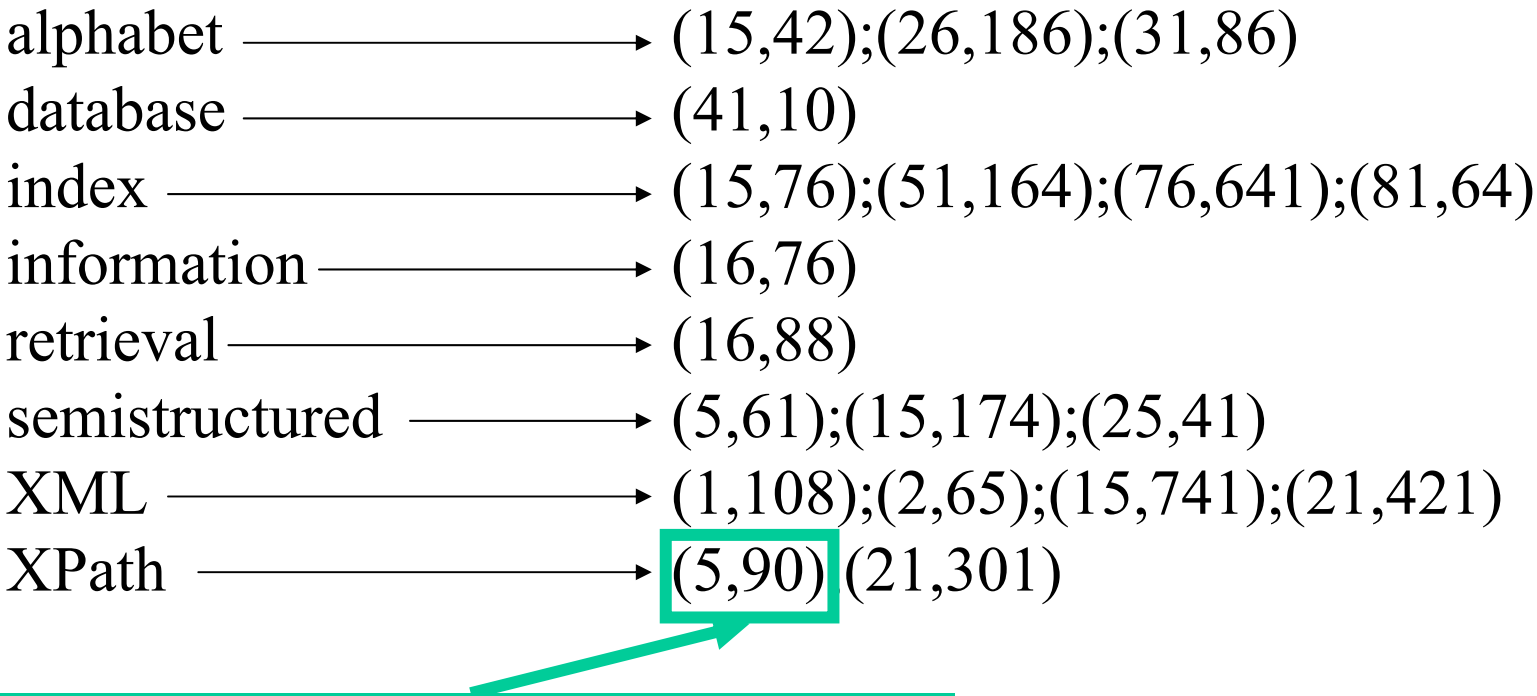
Storing Complete XML Documents

Documents are stored as files or as CLOBs (Character Large Object) in the database

- Very ineffective to answer queries (scan complete document collections for answers)
- Build additional index structures:
  - Inverted File Index (where does a term appear)
  - Inverted File plus structure index to answer structural queries

Inverted File Index

Store appearance of terms in documents (like index of a book)



Answer queries like „xml and index“, „information near retrieval“

But: not suitable for evaluating path expressions

Structure Index

- Compact representation of structural information for evaluating path expressions

Element	(DocID,Pos)	Order	Parent
dblp	(1,1)	1	
article	(1,10)	1	(1,1)
article	(1,251)	2	(1,1)
author	(1,21)	1	(1,10)
author	(1,64)	2	(1,10)

Used in combination with inverted file to answer queries like
`//article[CONTAINS(author,“Weikum“)]`

But: only for XML trees
descendant-or-self axis is hard to evaluate
result must be constructed from original document

Generic Relational Tables for XML

Store both structure and content in relational tables

Elements

DocID	ID	ElementName	Type	Value	Order	Parent
1	1	**dblp**			1	
1	2	**article**			1	1
1	3	**author**	String	P. Muth	1	2
1	4	**author**	String	G. Weikum	2	2
1	5	**article**			2	1

Attributes

ElementDocID	ElementID	AttributeName	Type	Value
1	2	**key**	ID	MuthW00

Mapping XPath to SQL

Queries (e.g. in XPath) are mapped to equivalent SQL queries on the generic tables:

XPath: `//article[author="G. Weikum"]`

```
SELECT e1.DocID,e1.ID
FROM   Elements e1, Elements e2
WHERE  e1.ElementName="article" AND
       e2.ElementName="author" AND
       e2.Value="G. Weikum" AND
       e2.parent=e1.ID AND
       e1.DocID=e2.DocID;
```

Generic Table Summary

- No DTD/Schema required to store XML documents in a relational database
- Result documents are constructed from the SQL result
- descendant-or-self axis is hard to evaluate (cannot be done directly in SQL!)
- Supports only XML trees (without links)

Generic Representation of Graphs

- Only one table for all kinds of nodes in the graph
- One additional table for edges of the graph
- Ignore order (use only for data-centric documents)

Nodes

DocID	ID	NodeName	Type	Value
1	1	`dblp`	Element	
1	2	`article`	Element	
1	3	`author`	Element	
1	4	`key`	Attribute	MuthW00
1	5	`author`	Text	P. Muth

Edges

SourceDocID	SourceID	TargetDocID	TargetID	Type
1	1	`1`	2	containment
1	3	`1`	4	attribute

...

DTD-Based Schema Generation (1)

Expoit DTD information to derive table definitions

Simple Example (just **#PCDATA** subelements):

```
<!ELEMENT article ((author)+,title,journal,year)>
<!ATTLIST article key ID #REQUIRED>
<!ELEMENT title (#PCDATA)> ...
```

1. Generate a table for the root element `article`

article					
DocID	ID	title	journal	year	key

2. Add DocumentID plus automatically maintained ID to the table

3. Add a column for each `#PCDATA` subelement that occurs at most once

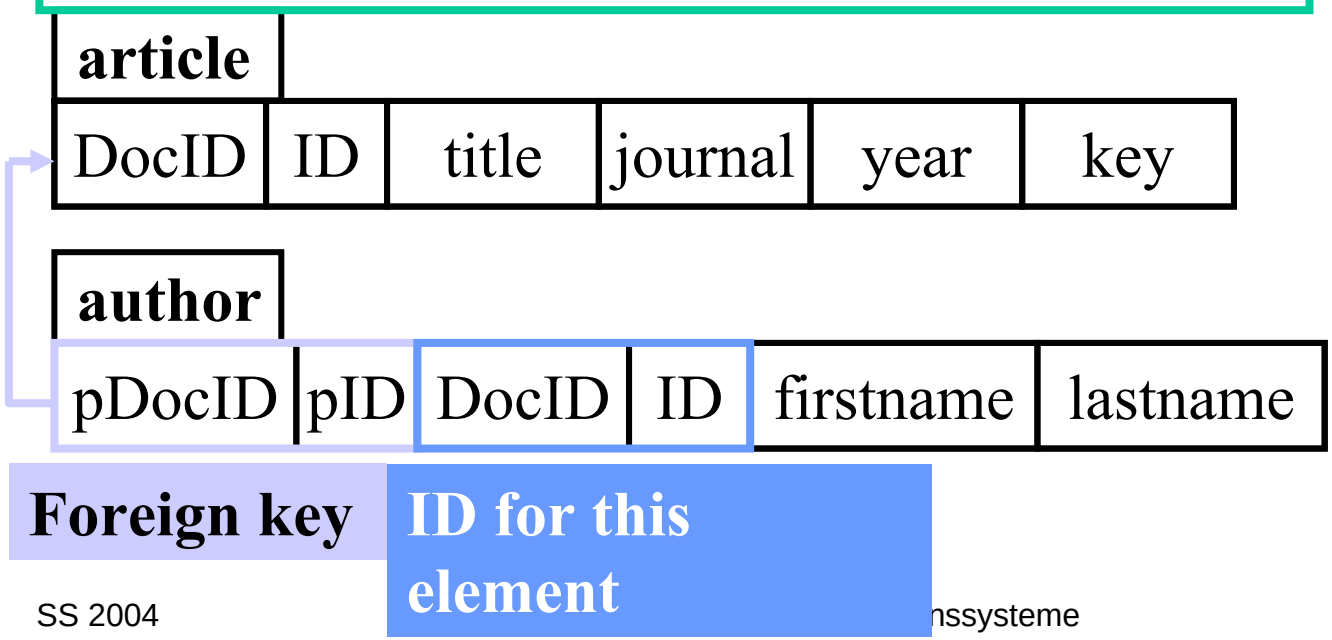
4. Add a column for each attribute

DTD-Based Schema Generation (2)

Subelements that occur multiple times are moved to separate relations linked with foreign keys

Simple Example (ctd.):

```
<!ELEMENT article ((author)+,title,journal,year)>
<!ATTLIST article key ID #REQUIRED>
<!ELEMENT title (#PCDATA)>
<!ELEMENT author (firstname, lastname)>
```



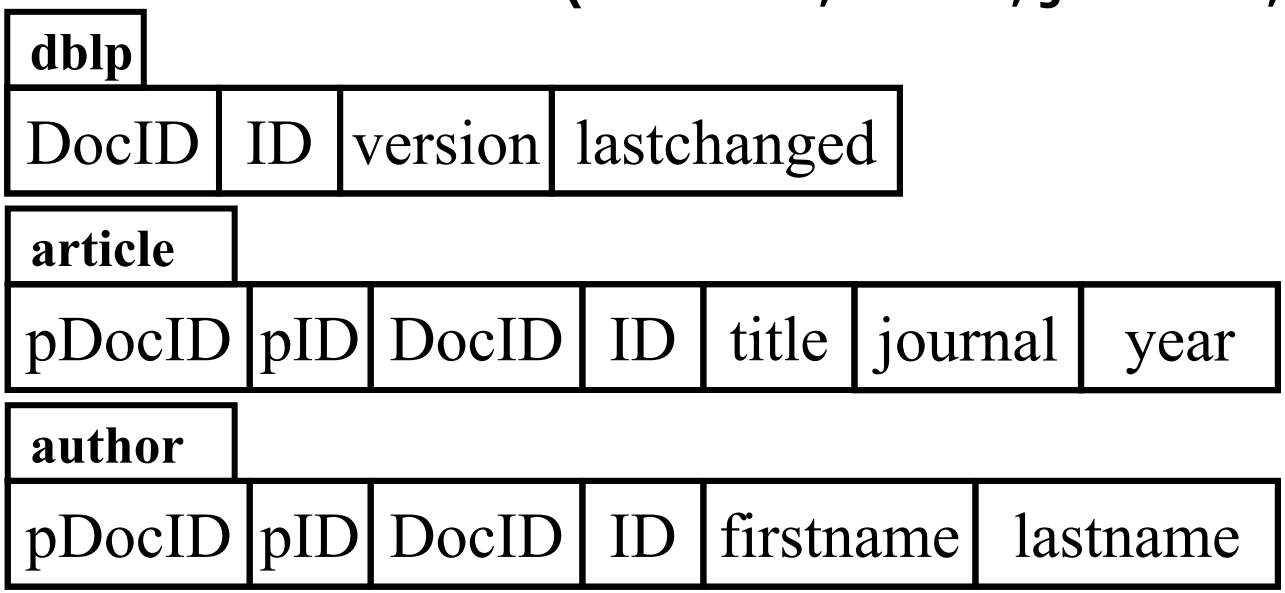
Generation for Complex Content

Complex Content is

- inlined, if it occurs at most once
- stored in an external table, if it may occur more often

Example:

```
<!ELEMENT dblp (meta,(article)*)>
<!ELEMENT meta (version,lastchange)>
<!ELEMENT article (author+,title,journal,year)>
```



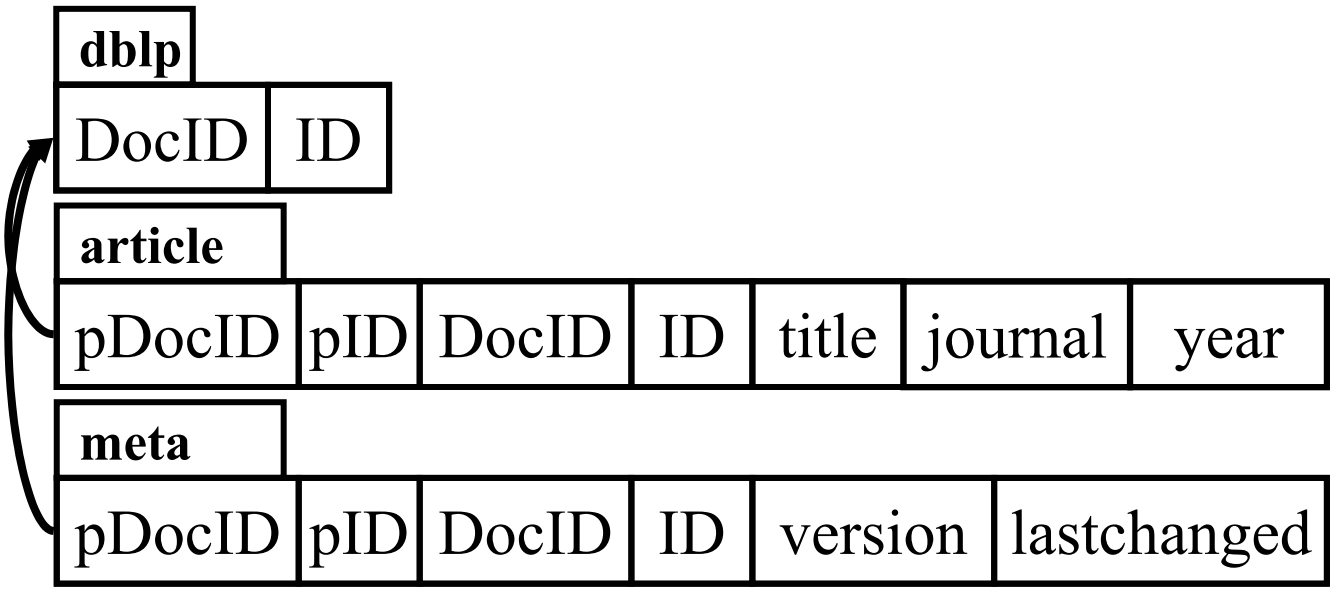
Alternative for Complex Content

Complex Content can also always be stored externally and point to its parent relation, regardless of its occurrence

(Makes creation easier, but query evaluation less efficient)

Example:

```
<!ELEMENT dblp (meta,(article)*)>
<!ELEMENT meta (version,lastchange)>
<!ELEMENT article (title,journal,year)>
```



Problem 1: Alternatives

Which schema should we generate for

```
<!ELEMENT vehicle (car|bike|truck|ship)>
```

Solution 1: encode all alternatives into one relation

$\Rightarrow$ many empty columns, not space-effective

Solution 2: store only the alternative that is used in an external table and use foreign keys for linkage

$\Rightarrow$ saves space, but requires more time to evaluate queries

Problem 2: Recursions

Problematic Scenario:

```
<!ELEMENT E1 (A,B,E2)>
<!ELEMENT E2 (C,D,E1?)>
```

Solution:

Break recursion by storing information externally and linking via foreign keys (as if **`E1`** occurred multiple times)

Problem 3: ANY

What is a good database schema for

```
<!ELEMENT description ANY>
```

⇒ Cannot be efficiently converted to a database schema

⇒ Store complete content of this element as XML (using a CLOB)

More on Schema Generation

- XPath queries can be mapped to SQL queries (typically containing many joins)
- Very (space and time) efficient if XML is well structured (data-centric)
- Not so efficient if XML is unstructured (document-centric)
- Can be extended to automatically decide which children to inline (based on query statistics: children that are often used are likely to be inlined)

18.2 Indexing XML

Problem: How to evaluate an XPath expression like
`//article/author[name="Weikum"]`

Two options:

- Traverse the complete XML graph and search for matching subgraphs (but: *very* inefficient for large graphs and small result sets)
- Maintain appropriate index structures to speed up query evaluation; two kinds:
  - Indexes on the content of elements and value of attributes
  - Indexes on the structure of the XML graph

Content Index (CI)

Retrieval methods of CI:

- Find elements and/or attributes that have a string in their content/value
- Find elements and/or attributes whose content/value satisfies a given template

Result of these methods is a list of nodes (or node IDs) in the XML graph that satisfy the search condition

Common implementation:

- **Inverted Lists** (maybe augmented with additional information like tf- and idf- values) plus efficient string search index on index entries
- Evaluation of arbitrary template expressions can be hard

Structure Index (SI)

SI should support all structural XPath axes:

`child, descendant, descendant-or-self, self, parent, ancestor, ancestor-or-self, following, preceding, following-sibling, preceding-sibling, attribute`

Most important axes: ($\Rightarrow$ Path Index PI)

- **`child (/)`**
- **`attribute (@)`**
- **`parent (..)`**
- **`descendant-or-self (//)`**

Interface Method of SI:

Given a node set **N** and an XPath axis **A** as input, compute the nodes reachable from the nodes in **N** via the axis **A** and return the set **N‘** of these nodes.

XPath Evaluation

Evaluation of an XPath location step **`axis::test[condition]`** with input node set **N** (result of the previous location step or root node for the first location step in the location path):

- Compute the result set **N1** by following **`axis`** from nodes in **N**
- Compute set **N2** of nodes in **N1** that satisfy **`test`**
- Compute set **N3** of nodes in **N2** that satisfy **`condition`** (which is again an XPath location path)
- If **`condition`** has the form **`.="string"`**
  - Compute (using CI) set **N4** of nodes that satisfy the condition
  - Compute **N3** by intersecting **N2** and **N4**

Using Pre- and Postorder as SI

Idea:

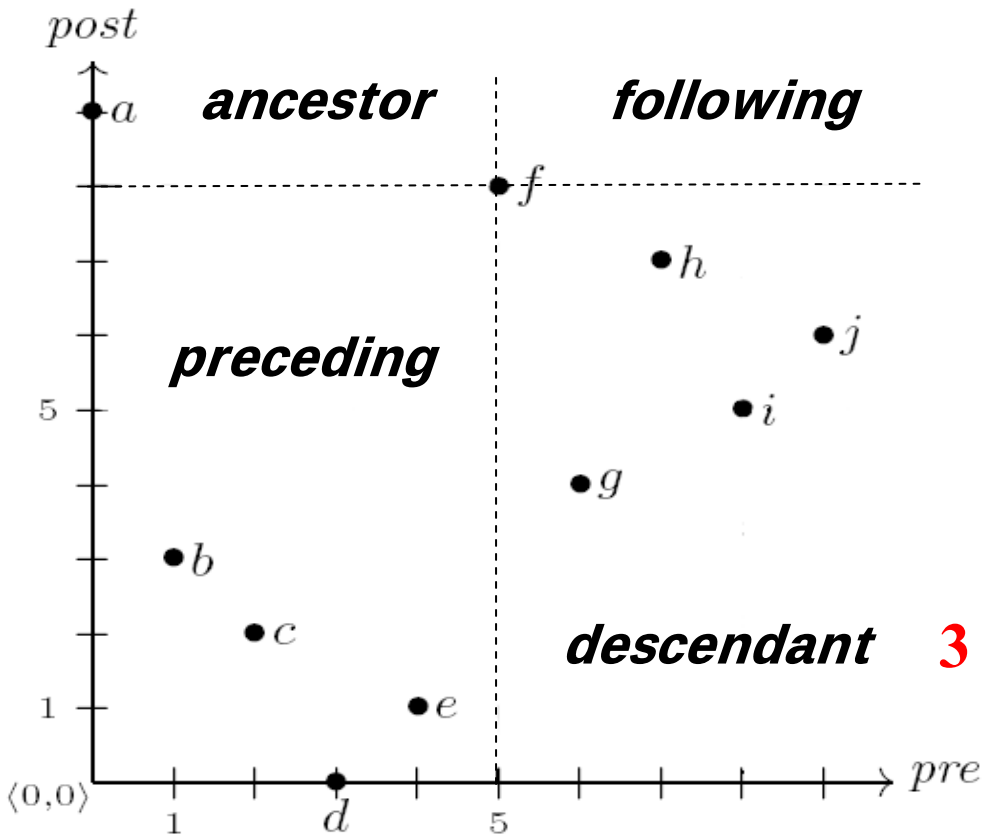
- Use two numbering schemes (pre- and postorder) for the nodes in the XML tree
- Compute result of following an XPath axis from a given node by evaluating conditions on pre- and postorder ranks

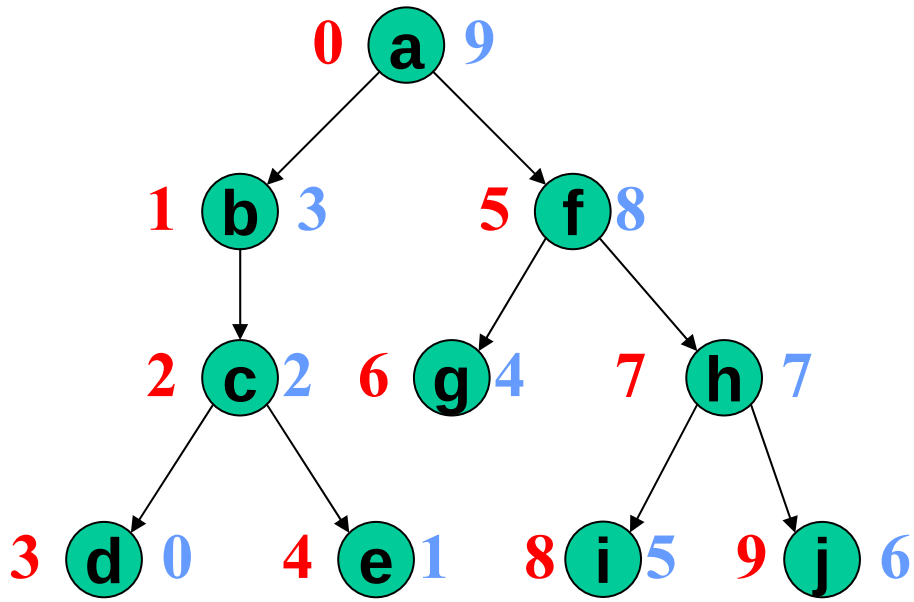
How to compute pre- and postorder:

- Compute depth-first traversal of XML tree
- Preorder rank = order in which the traversal enters nodes
- Postorder rank = order in which the traversal exits nodes

(see Slides 17-28 and 17-29)

Details on Pre- and Postorder SI





Additionally store preorder rank of the parent

Pre- and Postorder Conditions

Database Schema for storing nodes:

Pre	**Post**	**Par**	**Tag**
Preorder rank	Postorder rank	Parents preorder rank	Stores the element tag or attribute tag

SQL Query conditions for important axes:

Axis	**Pre**	**Post**	**Par**	**Tag**
`child`	(pre(v),∞)	[0,post(v))	pre(v)	*
`descendant`	(pre(v),∞)	[0,post(v))	*	*
`descendant-or-self`	[pre(v),∞)	[0,post(v)]	*	*
`parent`	[par(v),par(v)]	(post(v),∞)	*	*
`ancestor`	[0,pre(v))	(post(v),∞)	*	*
`ancestor-or-self`	[0,pre(v)]	[post(v),∞)	*	*
`following`	(pre(v),∞)	(post(v),∞)	*	*
`following-sibling`	(pre(v),∞)	(post(v),∞)	par(v)	*

Pre-/Postorder Summary

Advantages:

- Supports all XPath axes
- Efficiently handles **`ancestor-or-self`** queries (`//`)

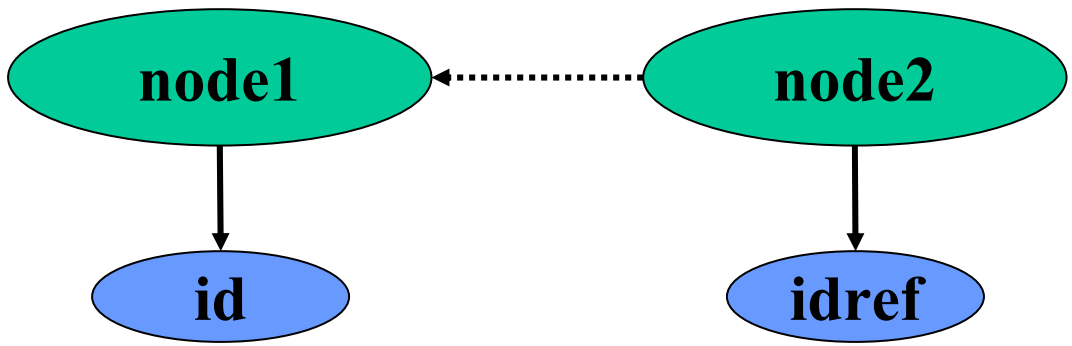
Problems:

- Does not support links
- Numbering scheme has to be recomputed upon updates
- Cannot compute the distance between nodes (would be interesting for ranking results)

Modelling Links in the XML Graph

For simplicity, we model ID/IDREF-links by one single edge (useful for information retrieval):

```
<node1 id="sample"> … </node1>
…
<node2 idref="sample"> … </node2>
```



APEX

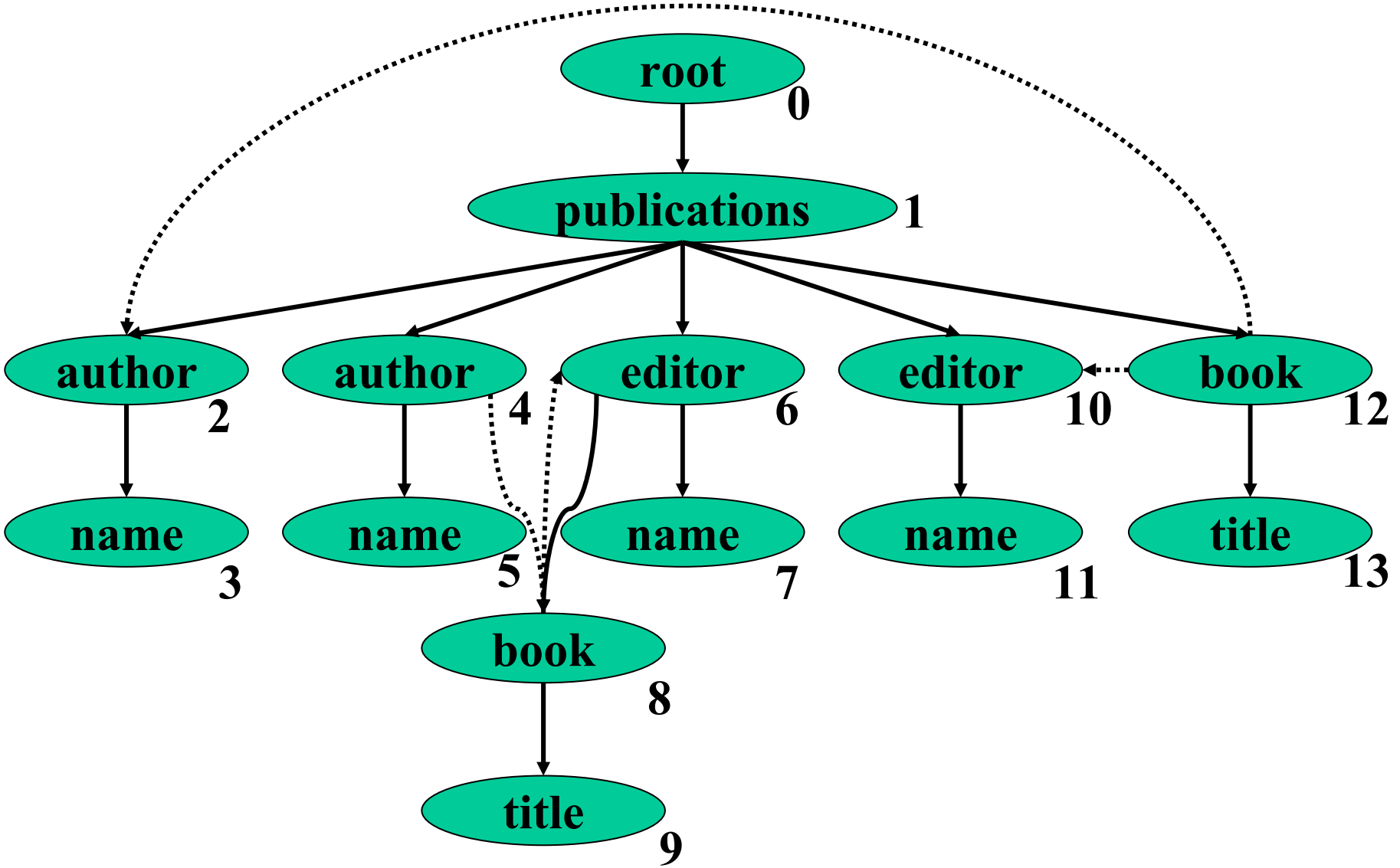
APEX: **A**daptive **P**ath Ind**E**x for **X**ML Data

Components:

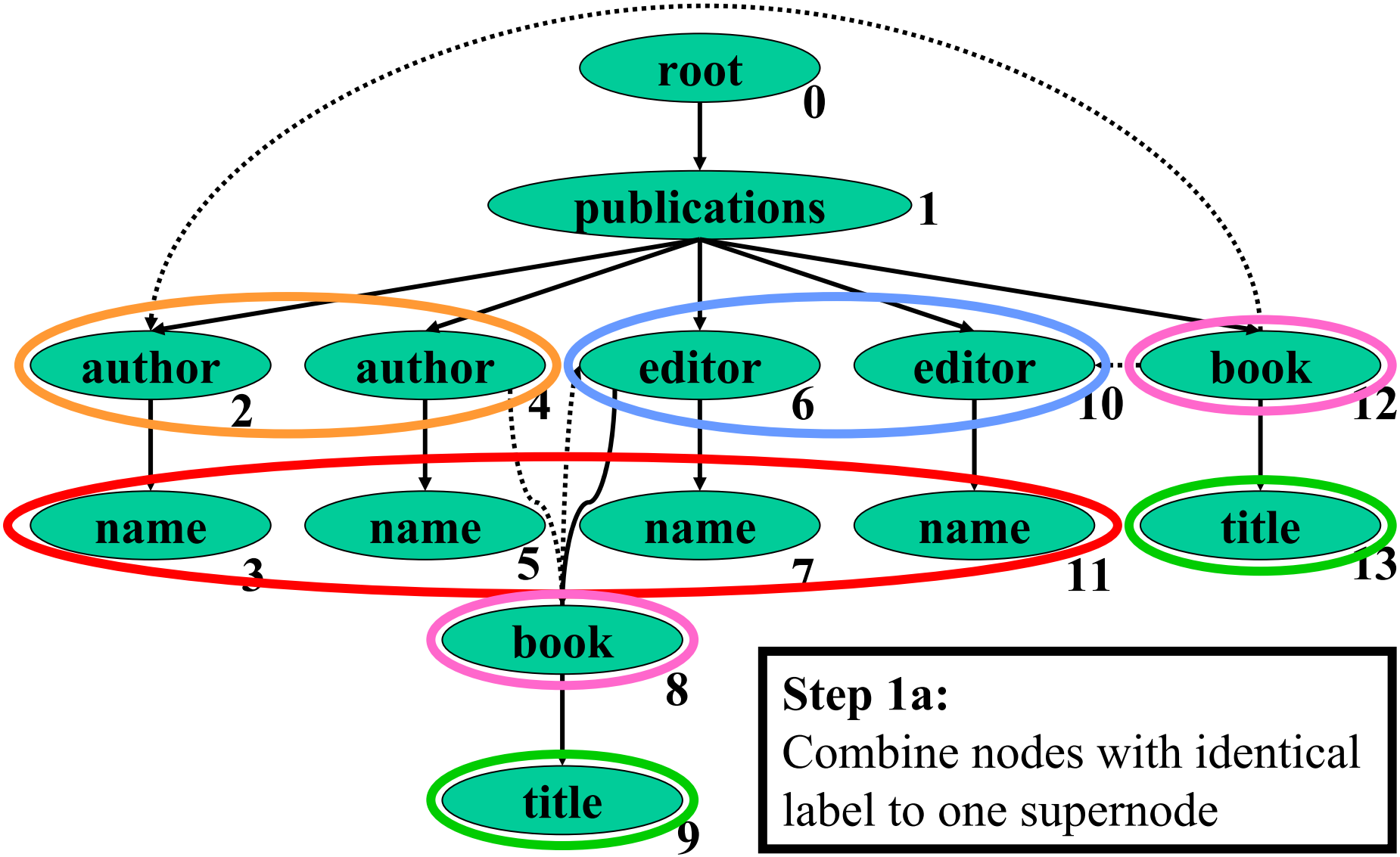
- Structural Summary: Compactly store information about connections in-between node types
- Collect information about workload
- Refine structural summary to efficiently support often used subqueries (at the price of possibly higher cost for less frequently used queries)
- Adaptive to changing workloads

We consider an adaption of APEX to our XML data model

Step 1: Create Structural Summary

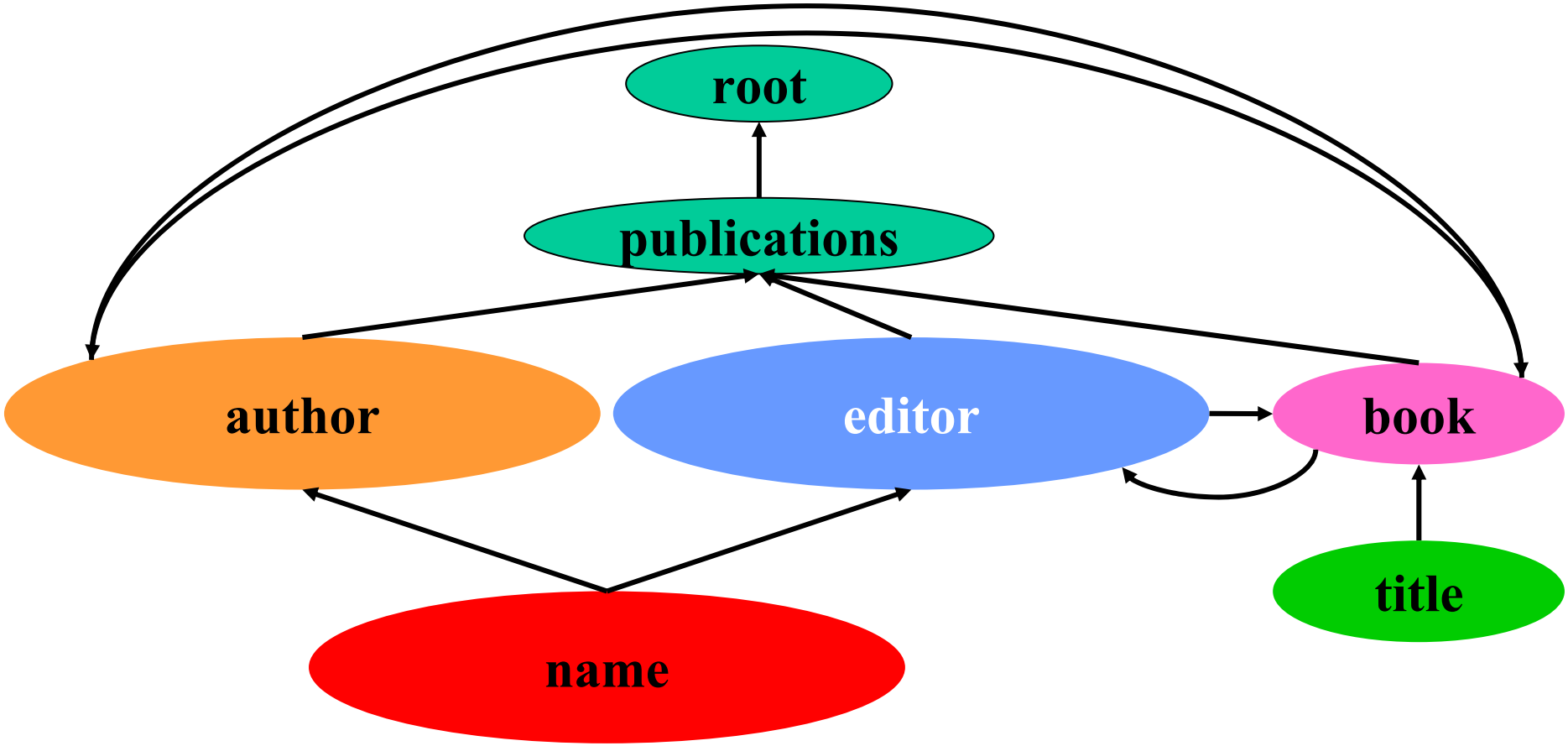


Step 1: Create Structural Summary



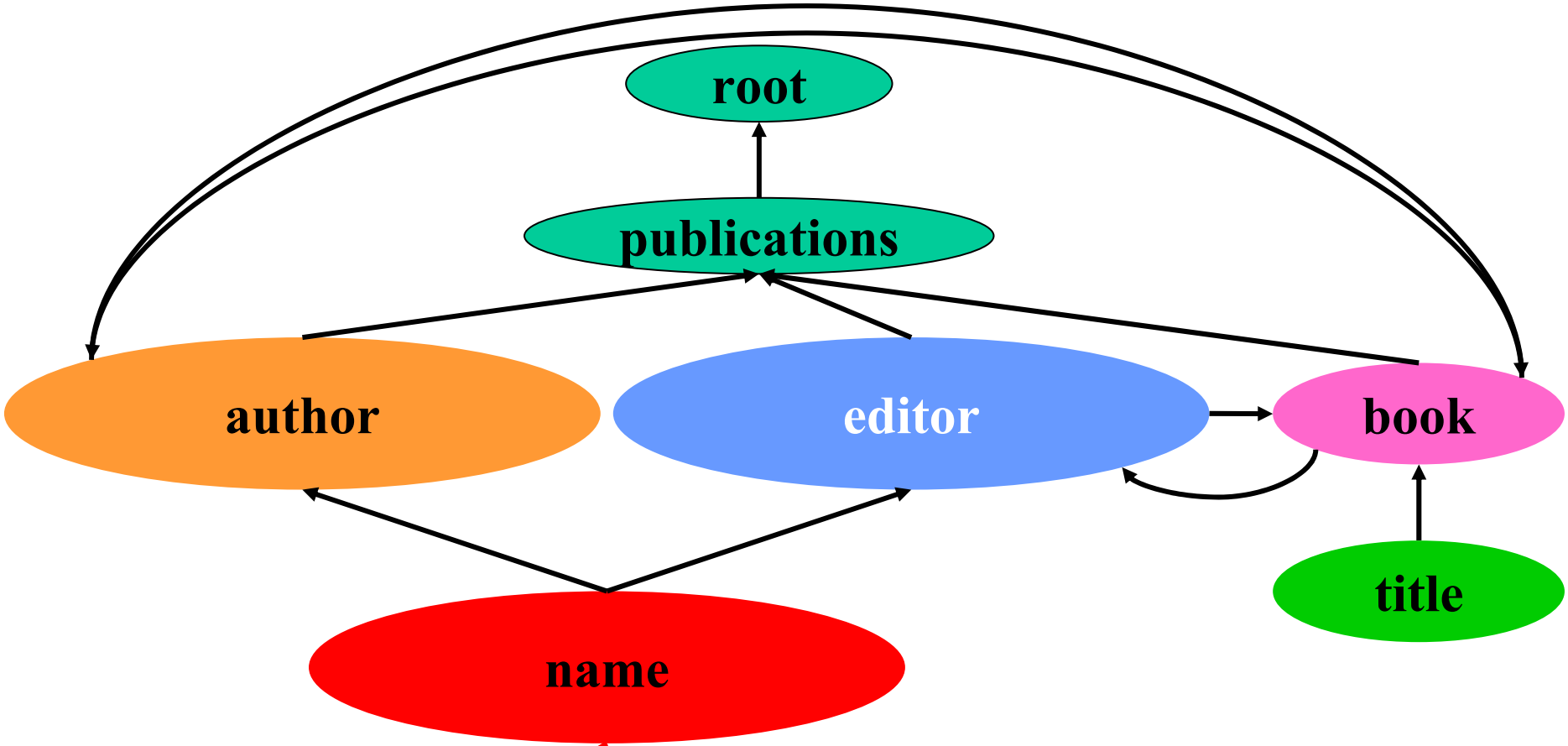
Step 1a:
Combine nodes with identical label to one supernode

Step 1: Create Structural Summary



Step 1b:
Add edges between supernodes in reverse order

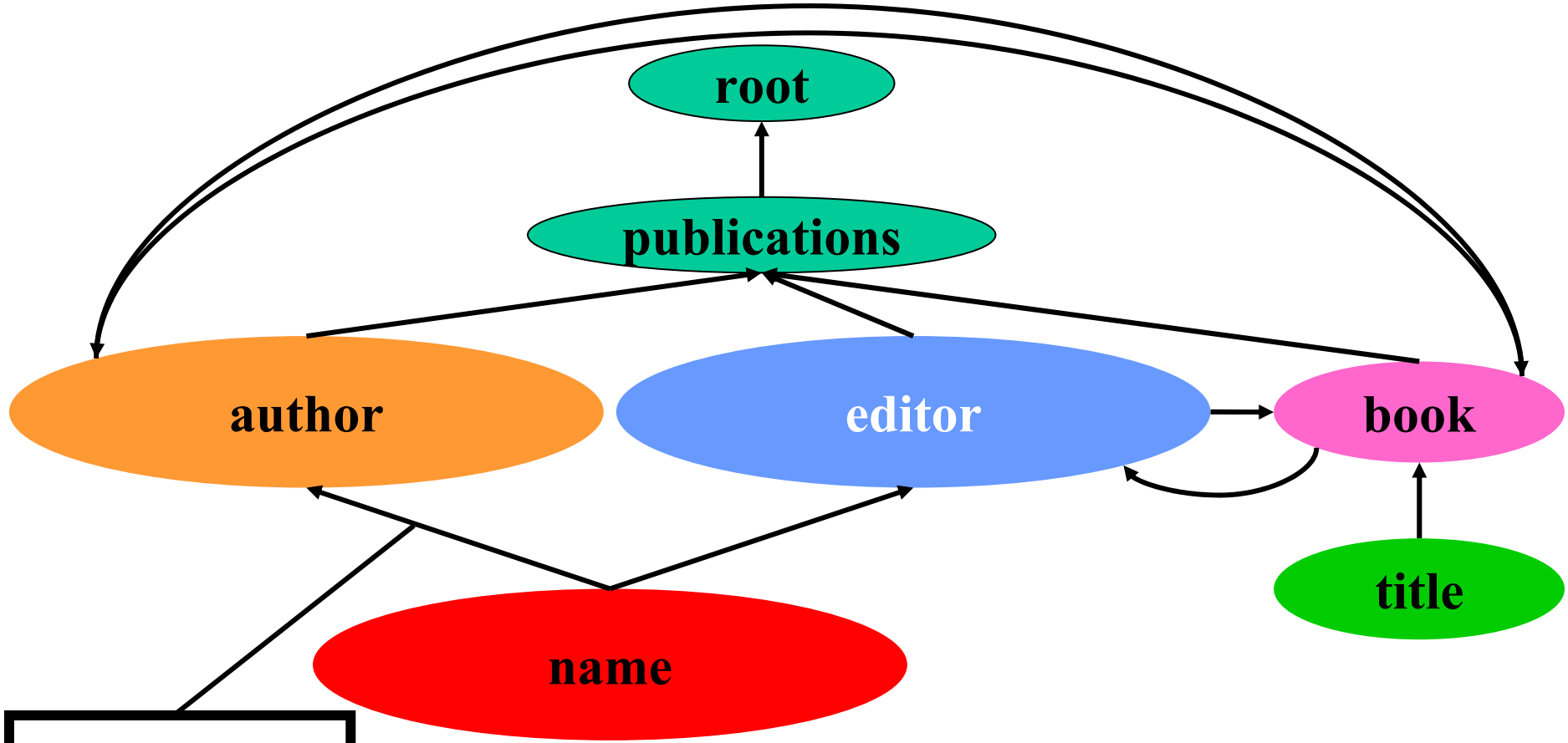
Step 1: Create Structural Summary



instances:
3,5,7,11

Step 1c:
augment supernodes with sorted instance information

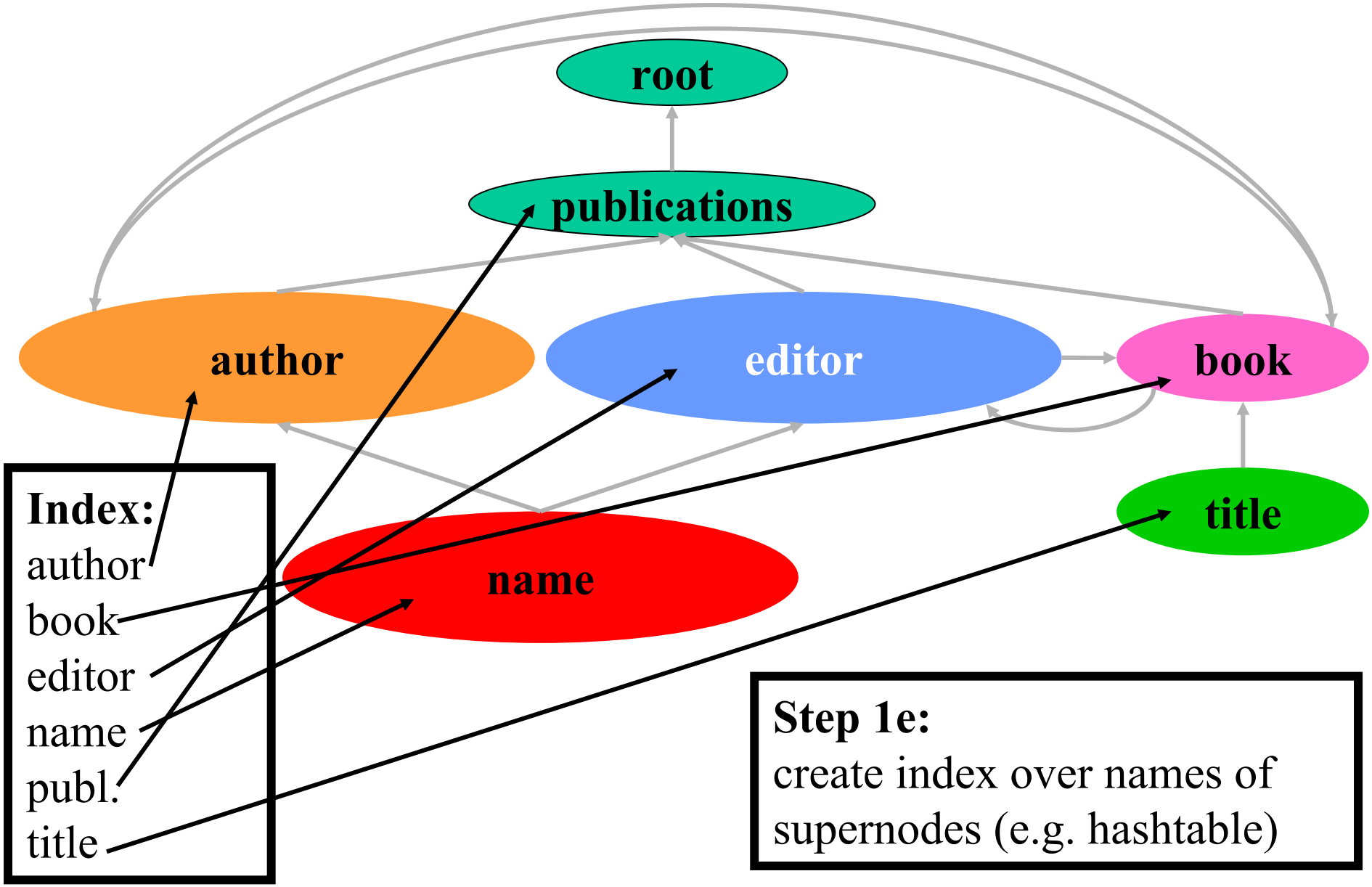
Step 1: Create Structural Summary



connections:
(2,3)
(4,5)

Step 1d:
augment edges with sorted connection information

Step 1: Create Structural Summary



Step 1e:
create index over names of supernodes (e.g. hashtable)

Evaluation with Structural Summary

- **`//name`**:
  search for “name“ in supernode name index, go to supernode for “name“, extract instance list
- **`//author/name`**:
  get supernode for “name“ as before, follow edge to “author“ supernode, get instances from edge annotation
- **`author/name`**: (starting at one specific author node)
  get supernode for “name“ as before, follow edge to “author“ supernode, get instance for specified starting node from edge annotation

Structural Summary is initial version of APEX‘

Step 2: Collect Workload Info

- Count how often paths appear in the query workload:

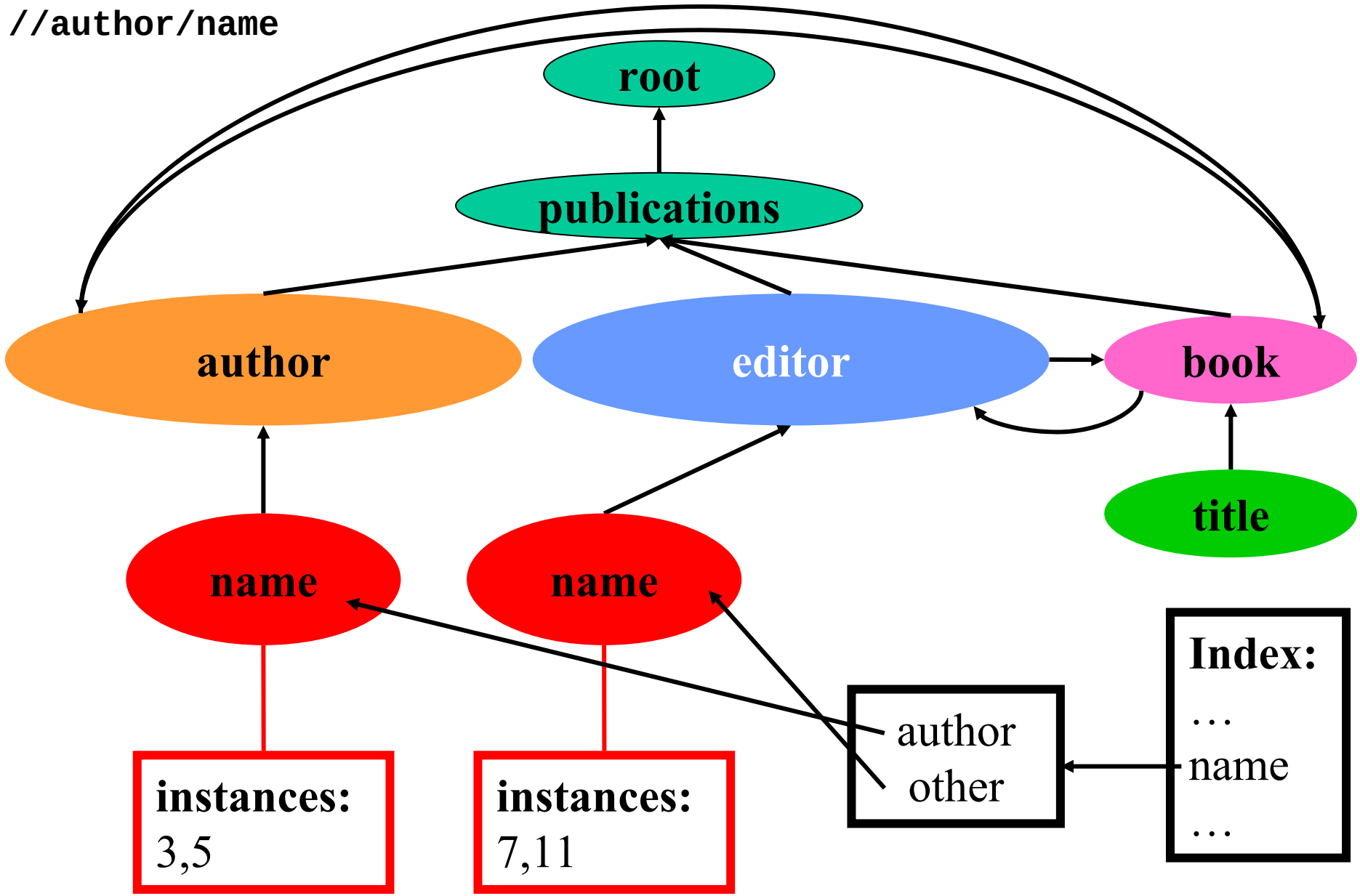
  10x `//author/name`

  3x `//director/title`

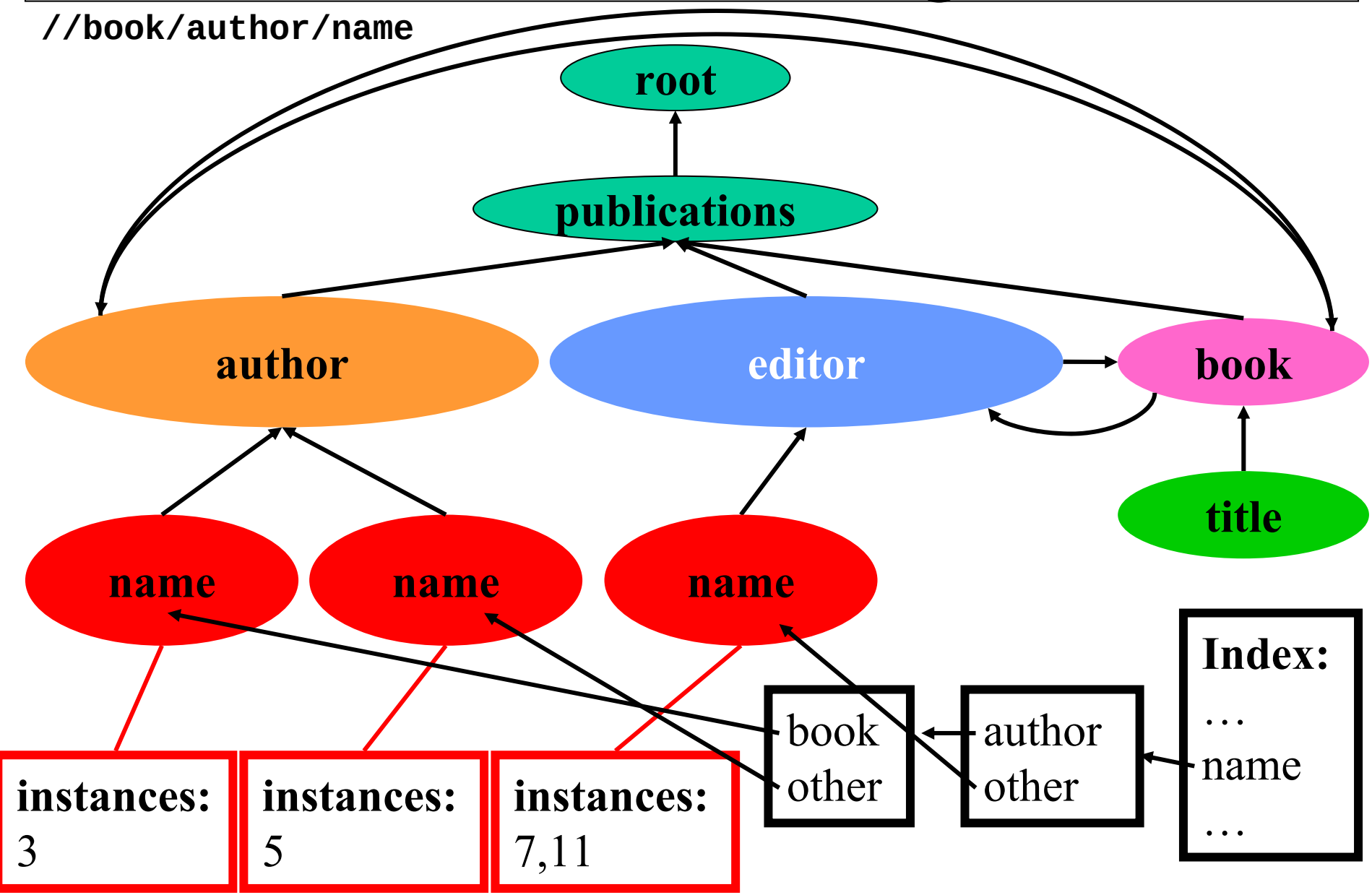
  2x `//movie/title`

- Update structural summary to efficiently support often used paths (with usage over a fixed threshold):
  Split associated supernodes such that often used paths can be evaluated without further evaluation
  in our example: `//author/name`

Step 3: Update Structural Summary



Most Effective on Longer Paths



APEX Summary

Advantages:

- Adaptive to workload changes
- Efficiently handles **`ancestor-or-self`** queries (`//`)
- Efficiently handles often used queries
- Supports links

Problems:

- How to set threshold? (Guru required!)
- Performance under structural updates unclear
- Performs less efficiently for less frequently used queries
- Performs poorly for queries like **`a//b`**

Indexing the XML-based Web

How to index the XML-based Web of the future?

- Too large for existing index structures (>10^{10} pages)
- Inter-document queries (following XLinks)
- Optimized for IR (non-boolean queries)
- Distance-based ranking for `//` expressions
- Varying structure and element naming
- First results should be returned very quickly ($\Rightarrow$ architecture should be pipelined)

"Normal" queries (like this XXL-query) are no longer sufficient:

```
SELECT $p FROM INDEX
WHERE publication AS $p AND $p.author AS $a
  AND $a LIKE "Weikum" AND $p.content LIKE "XML"
  AND $p.citation.author LIKE "Jim Gray"
```

Indexing the XML-based Web

How to index the XML-based Web of the future?

- Too large for existing index structures (>10^{10} pages)
- Inter-document queries (following XLinks)
- Optimized for IR (non-boolean queries)
- Distance-based ranking for `//` expressions
- Varying structure and element naming
- First results should be returned very quickly (⇒ architecture should be pipelined)

Automatically add structural vagueness:

```
SELECT $p FROM INDEX
WHERE #.~publication AS $p AND $p.#.~author AS $a
  AND $a LIKE “Weikum“ AND $p.#.~content LIKE “XML“
  AND $p.#.~citation.#.~author LIKE “Jim Gray“
```

Connection Index

Evaluation of path expressions with structural vagueness requires support for inner path wildcards (`//` or `#`)

Basic approach:

- Compute transitive closure **C** of XML graph (containing all the *connections* between nodes)
- Find compact representation **C‘** of **C**
- Evaluate wildcard expressions using **C‘**

Computing the Transitive Closure

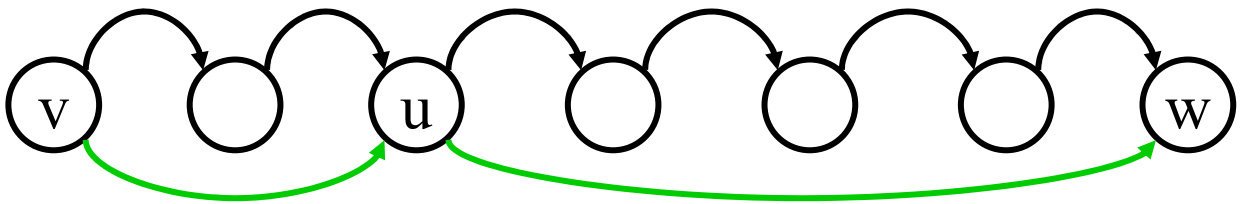
Given a directed graph G=(V,E), the transitive closure C=(V,T) can be computed in $O(|V|^3)$ using the Warshal algorithm or in O(|T|*max(outdeg)) using the following alg:

```
Set  T:=E∪{(v,v)|v∈V};
Set ΔT:=T;
while (ΔT≠∅)
{ Set T‘:=∅;
  for all p:=(v,u) in ΔT
    for all e:=(u,w) in E
      if (v,w)∉T then T‘:=T‘∪{(v,w)};
  T:=T∪T‘;
  ΔT:=T‘;
}
```

Idea:
In round i of the loop, compute all connections over paths of length i+1 by extending existing connections over paths of length i

Compact Representation

- For each node v, maintain two sets of labels (which are node names): $L_{in}(v)$ and $L_{out}(v)$
- For each connection (v,w),
  - choose a node **u** on the path from **v** to **w**
  - add **u** to $L_{out}(v)$ and to $L_{in}(w)$
- Then $(v,w) \in T \Leftrightarrow L_{out}(v) \cap L_{in}(w) \neq \emptyset$



Two-hop Cover of T

- Minimize the sum of the label sizes (NP-complete $\Rightarrow$ approximation required)

Query Evaluation with Two-Hop

- **`//author`**:
  search for “author“ among all nodes (can be done efficiently using index on node names)
- **`//author/name`**:
  get nodes for “author“ as before, go to connected nodes with distance 1 and look for “name“ nodes (distance information may be included in the index)
- **`author/name`**: (starting at one specific author node)
  as before, for one specific author node
- **`//author//editor`**:
  get “author“ nodes and the union of their L_{out} sets, get “editor“ nodes and the union of their L_{in} sets, compute intersection

Web-Scale XML Indexing

Existing indexing strategies are not usable for Web-scale data:

- Hardly support for inner-query path wildcards (necessary for expressing structural vagueness)
- Documents are very heterogeneous (no single indexing technique can be the best for all documents)
- Sometimes no support for intra-document links
- No explicit support for inter-document links (treat links as if they were ordinary containment edges) $\Rightarrow$ few, very large documents to index
- Space, build and execution time for large-scale XML data is unacceptable

FLIX: Index Framework for XML

Current Research Project: ***FLIX*** (**F**ramework for Indexing **L**arge Collections of **I**nterlinked **X**ML Documents)

Basic Principles:

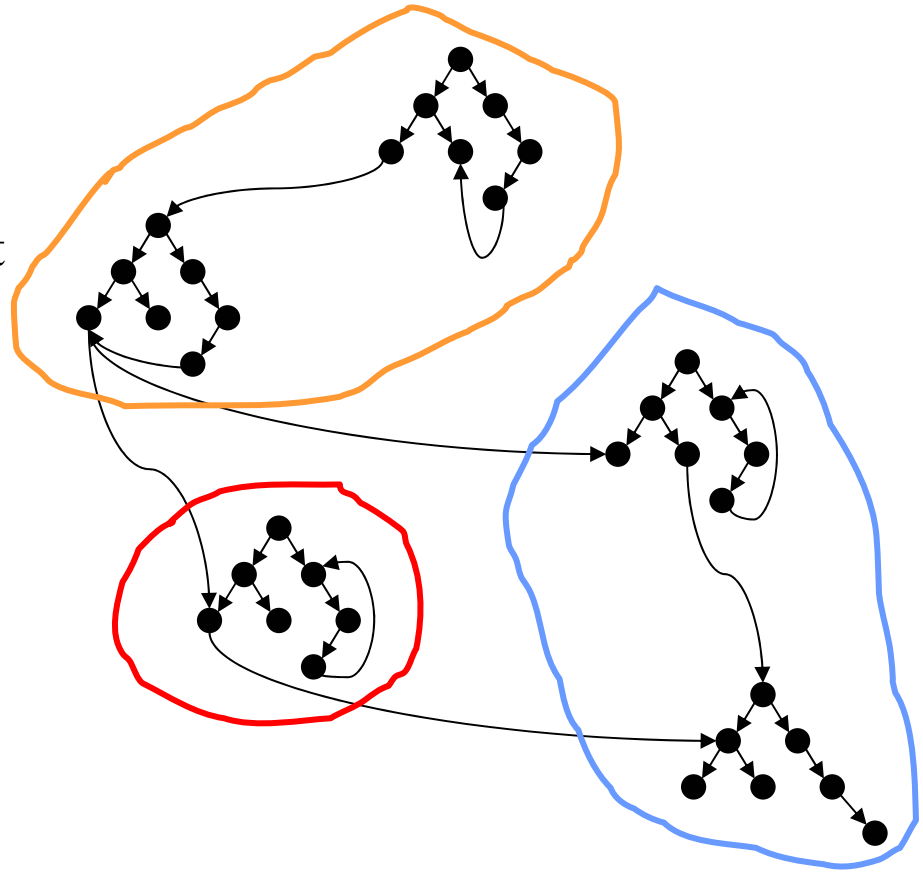
- Automatically construct fragments of the set of interlinked XML documents
- Choose an „optimal“ connection index for each meta document (depending on workload and available space)
- Incrementally build results for path queries with inner path wildcards in approximate order of distance ($\Rightarrow$ most relevant results first)

FLIX Example

Step 1:

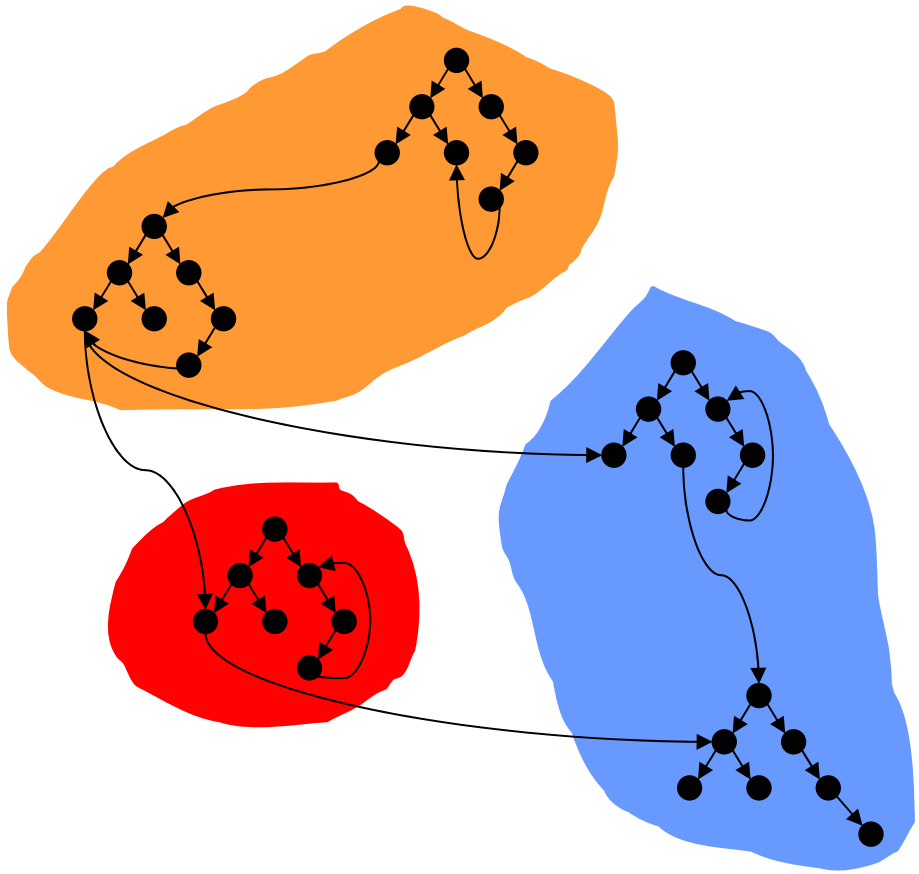
Choose meta documents

- using a graph-theoretic measure
- by identifying units with coherent information
- by analyzing query workload



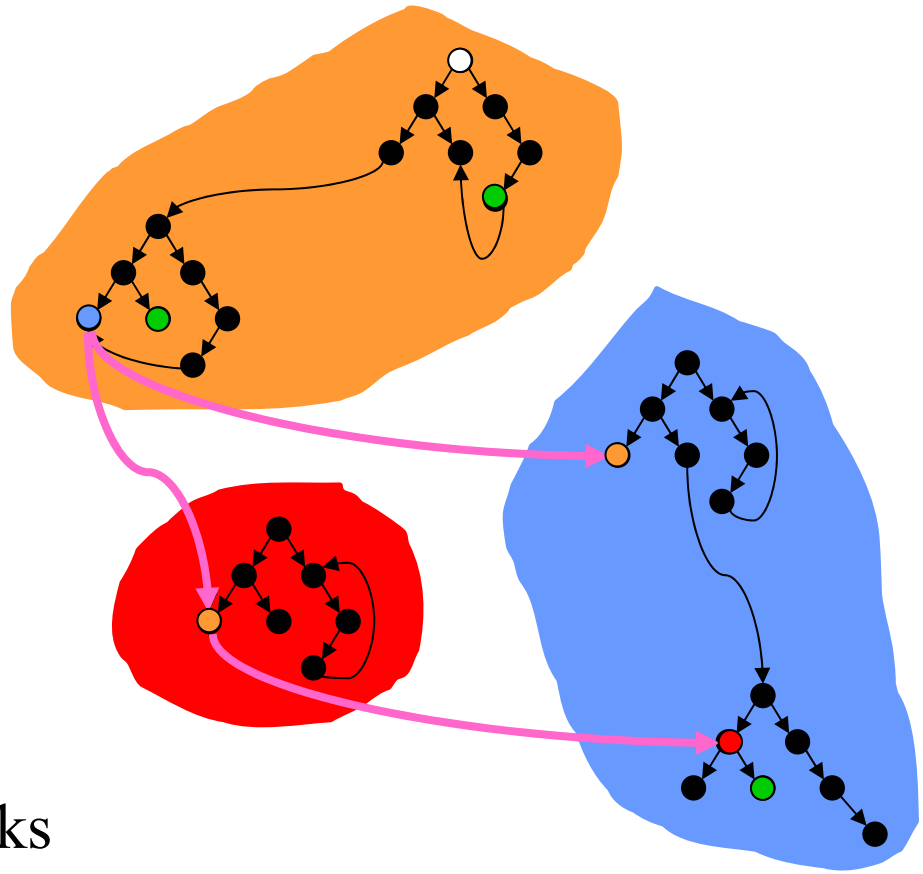
FLIX Example

Step 2:
Choose and build optimal connection index for each meta document



FLIX Example

Step 3:
Incrementally evaluate query with inner path wildcard
`//a//b`



- Find set of instances of **`a`**
- Evaluate query within resulting meta documents
- Find elements with outgoing links in resulting meta documents and follow the links in ascending distance
- Evaluate rest of query (//b) in target meta document

18.3 XML Compression

XML is not very storage efficient:

- contains a lot of redundancy (increases readability for human users):
  - opening and closing tags
  - element and attribute names
- character-based:
  - 4294967295 (10 bytes) vs. 0xFFFFFFFF (4 bytes)

⇒Compression can save much storage space

Text-based compressors (like gzip): ~80-90% compr. ratio

Using information about XML structure gives even better compression ratios

Example: Web Log Data

ASCII File 15.9 MB (gzipped 1.6MB):

```
202.239.238.16|GET / HTTP/1.0|text/html|200|1997/10/01-00:00:02|-|4478|-|-|http://www.net.jp/|Mozilla/3.1
```

XML-ized inflates to 24.2 MB (gzipped 2.1MB):

```
<apache:entry>
   <apache:host> 202.239.238.16 </apache:host>
   <apache:requestLine> GET / HTTP/1.0 </apache:requestLine>
   <apache:contentType> text/html </apache:contentType>
   <apache:statusCode> 200</apache:statusCode>
   <apache:date> 1997/10/01-00:00:02</apache:date>
   <apache:byteCount> 4478</apache:byteCount>
   <apache:referer> http://www.net.jp/ </apache:referer>
   <apache:userAgent> Mozilla/3.1$[$ja$]$(I)</apache:userAgent>
</apache:entry>
```

XMill

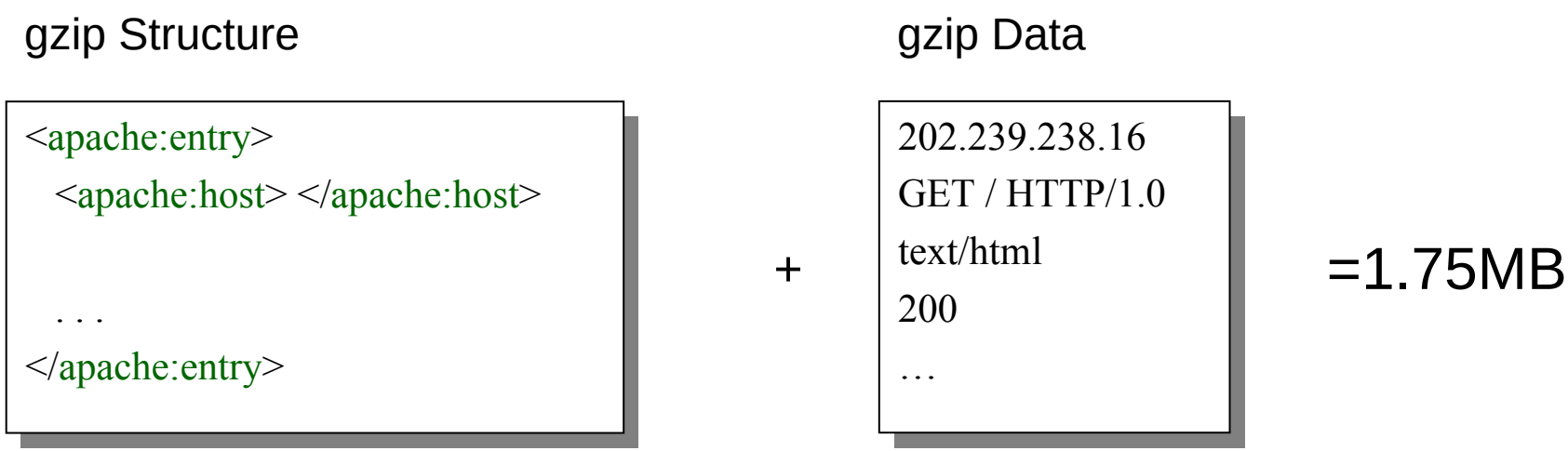
- Specialized compressor for XML data
- Utilizes three basic principles:
  - Compress the structure separately from the data
  - Group the data values per element type
  - Apply semantic (specialized) compressors

$\Rightarrow$ Even better compression ratios (optimality of compression can be proven)

(Slides for XMill partly taken from the original presentation at SIGMOD 2000)

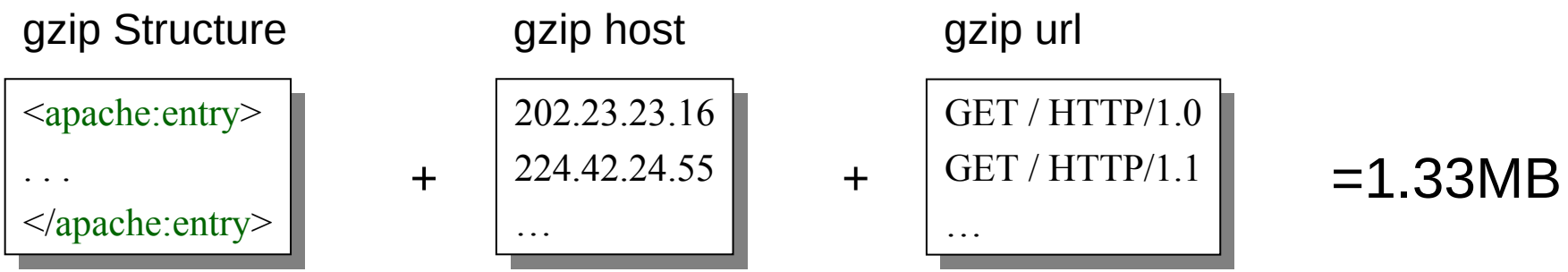
XMill Principles (1)

Compress the structure separately from the data:



XMill Principles (2)

Group the data values per element type:



XMill Principles (3)

Apply semantic (specialized) compressors for known types:

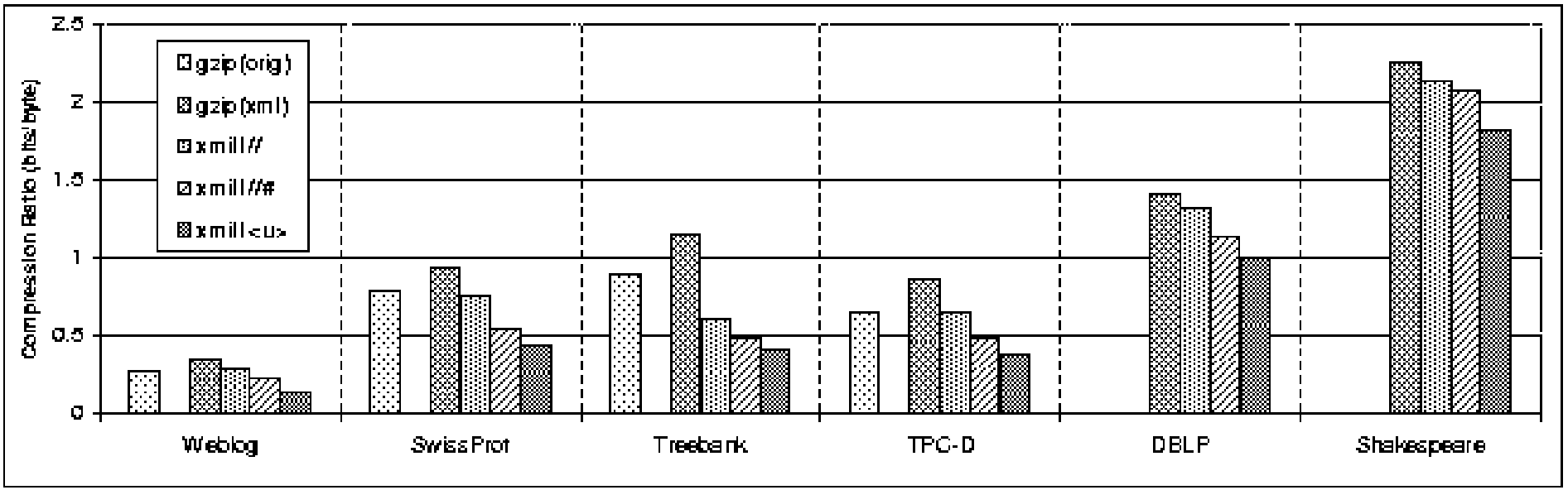
gzip Structure + gzip c1(host) + gzip c2(url) + ... =0.82MB

Examples:

- 8, 16, 32-bit integer encoding (signed/unsigned)
- differential compressing (e.g. 1999, 1995, 2001, 2000, 1995, ...)
- compress lists, records (e.g. 104.32.23.1 → 4 bytes)

semantic compressor selection by user and/or schema information

XMill Compression Factors



- gzip (orig) works on the original data
- gzip (xml) works on the XMLified data
- XMill // separates structure and data
- XMill //# additionally groups the data by element names
- XMill <u> additionally applies type-specific compressors

18.4 Native XML Databases

Key concepts of native XML Databases:

- (Logically) store XML documents without converting them
- Logical unit of information is a single XML Document
- Efficient query evaluation on the XML documents (with index structures for structure and content)
- Update, insert and delete XML documents
- All „standard“ database features:
  - Transactions
  - Distribution, replication
  - Parallel server
  - Multiuser
  - Security
  - Query optimization, Performance
  - …

Native XML vs. XML-Enabled RDMBS

XML-Enabled RDBMS:

- XML mapped to relational data model
- Optimizer tuned for relational algebra (projection, join, selection)
- Evaluation of query may require numerous SQL joins

Native XML-DB:

- XML stored natively
- Optimizer tuned for tree algebra (e.g., tree traversal)
- Evaluation of query using specific index structures

XML Schema Design

- Basic unit is one XML document
- What should go into a document:
  - Document = individual thing, event, …(about a product, report, order, measurement, treatment)
  - Document collects related information (about a project, process, career, …)
- Principle: Store exactly the information within a single document that is most often requested together
- Refer to less frequently used information via XLinks

XML Databases: Tamino

- **Tamino: T**ransactional **A**rchitecture for **M**anaging **In**ternet **O**bjects
- Available since 1999 from Software AG
- Evaluation version available (252MB!)
- Best-known native XML database
- Key concepts:
  - Extension of XPath for querying
  - Updates, Inserts, Deletes
  - Built-in Extensions for Information Retrieval
  - Support for optional validation against XML Schema
  - Transparent integration of non-XML data from other sources (automatic mapping to XML view of the data)
  - Extensible Architecture

Transaction Support in Tamino

- A transaction consists of a set of operations (queries and/or updates) that form a unit
- Transactional principles (ACID):
  - Atomicity (all-or-nothing)
  - Consistency (maintain database consistency)
  - Isolation (run transaction as if it was alone in the DB)
  - Durability (data must survive failures)
- Transaction support in Tamino:
  - A+D using logging (+restoring information from logs) plus two-phase commit for distributed transactions (Windows only)
  - I using document-level locks ($\Rightarrow$ potential performance bottleneck)
  - C by checking modified documents against constraints

Other XML Databases: Timber

- Research prototype from Uni Michigan
  - Applies existing backend system *Shore* for disk management, buffering, concurrency control
  - XML documents are stored in fragments, where each fragment roughly corresponds to a DOM node
  - Supports content indexes on elements and attributes, variant of pre/post-ordering as structure index
  - Uses tree algebra for optimizing XQuery expressions (focus of the project)
  - Updates of XML documents supported (but with problems with the numbering when too many updates occur)

Other XML Databases: Natix

- Research prototype from Uni Mannheim
  - XML documents are split into fragments in a clever way (minimizing reconstruction effort and maximizing query performance) $\Rightarrow$ one of the focusses of the project
  - Sophisticated transaction management techniques (e.g., element-level locking, timestamps)
  - Interfaces to Java, C++ (using DOM and SAX), WebDAV, HTTP and Filesystem
  - Inverted File as content index plus variant of pre/postorder scheme as structure index

XML Databases: Xindice

- Open-source project from Apache
  - Availabe from http://xml.apache.org
  - Source code available (but not well documented)
  - Optimized for many, small XML documents (max document size: 5MB)
  - Implements XPath and update operations
  - API available
  - Supports only content indexes, no structure index
    $\Rightarrow$ poor query performance
  - May be a starting point for using XML databases

Sources and Further Literature for Part 18

- M. Klettke, H. Meyer: **Speicherung von XML-Dokumenten – eine Klassifikation**. Datenbank-Spektrum 3(5), 2003.
- H. Schöning: **XML und Datenbanken**. Hanser, 2003.
- D. Florescu, D. Kossmann: **Storing and Querying XML Data using an RDBMS**. IEEE Data Engineering Bulletin 22(3), 1999.
- G. Kappel et al.: **X-Ray – Towards Integrating XML and Relational Database Systems**. Technical Report, Uni Linz, 2000.
- T. Grust: **Accelerating XPath Location Steps**. SIGMOD Conference, 2002.
- C. Chun et al.: **APEX: An Adaptive Path Index for XML data**. SIGMOD Conference, 2002.
- Q. Chen et al: **D(K)-Index: An Adaptive Structural Summary for Graph-Structured Data.** SIGMOD Conference, 2003.
- H. Liefke and D. Suciu: **XMILL: An Efficient Compressor for XML Data**. SIGMOD Conference, 2000.
- J.-K. Min et al.: **XPRESS: A Queriable Compression for XML Data**. SIGMOD Conference, 2003.

Sources and Further Literature for Part 18

- Tamino: **http://www.softwareag.com/tamino/**
- Timber: **http://www.eecs.umich.edu/db/timber/**
- **Th. Fiebig et al.** : Natix: A Technology Overview. NODe 2002, Springer LNCS 2593.
- Xindice: **http://xml.apache.org/xindice/**