Web Databases and Applications

XML Indexing

Why is Indexing Needed?

- Allows fast access to data by replicating portions of the data in special purpose structures.
- Despite the additional cost (storage, maintenance and complexity) they have shown to be useful in evaluating queries.

Index Types

- Structural index
  – Accessing all elements of given name
  – Ancestor-descendant and parent-child relationship between elements
- Content index
  – Accessing elements containing given keywords
  – Supporting most text search functionalities

Classical Content Index

- Classically based on inverted lists
  - For each term, gives the doc.ID + localization
- Several variations allows different search types
  - Offset, Relative, Proximity
- Generally stored in a B+-Tree to optimize search for a given word
- Size is an important issue
  - Memory and Disk

Words **Localization**

- *t1* : doc1-100, doc1-300, doc3-200, …

- *t2* : doc2-30, doc4-70, …

- *t3* : doc4-87, doc5-754, …

- (word, localization)
  - Fixed entry (word repeated)
- (word, Frequency, (localization)*)
  - Variable length entry

Problem with XML

- Support of element addressing
  - Doc.ID should include NodeId (Xpath) + Offset
- Index size becomes very large
  - XPath are long
- Support of typed data
  - Integer, float, simple types of XML schema
  - Requires classical indexes for certain elements
- Query processing
  - Structural joins
  - Text search
  - Exact search
- Support of updates
  - Incremental updates would be a plus

Path-based approach

- Represent XML document into tree or graph structure
- Index XML document directly
  - Without the support of DTD
- Mainly use the memory as the index storage
- Properties
  - Keep the structural information to improve query performance
  - Easy to support query with regular path expression

Examples

- Patricia Trie
  - Cooper et al. 2001
- A(*k*)-Index
  - Raghav Kaushik et al., 2002
- DataGuides
  - J. McHugh et al., 1997
- APEX (Adaptive Path Index for XML Data)
  - C. W. Chung et al., 2002

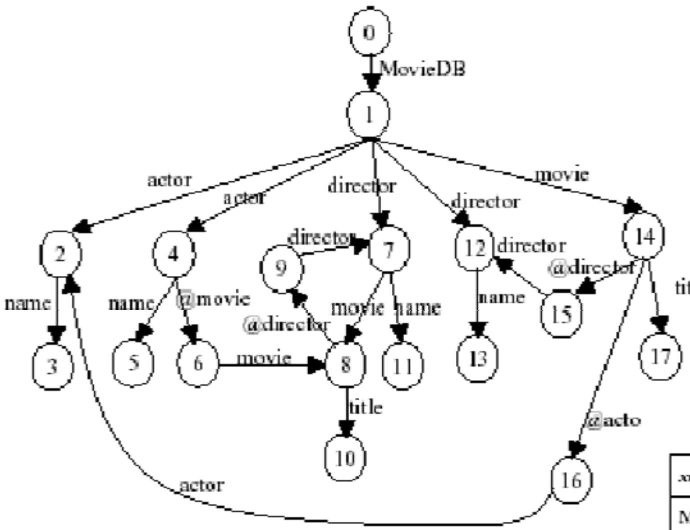
Partricia Tries

- Cooper et al. 2001
- Idea:
  - Partitioned Partricia Tries to index strings
  - Encode XPath expressions as strings
    (encode names, encode atomic values)

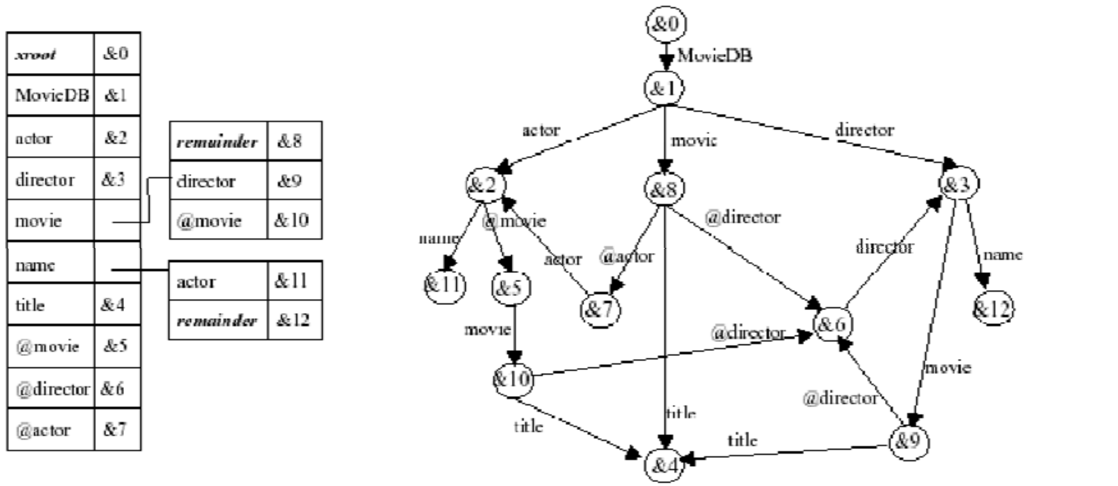
<book>
<author>Whoever</author>
<author>Not me</author>
<title>No Kidding</title>
</book>

B A 1 Whoever
B A 2 Not me
B T No Kidding

APEX Example



XML data structure



APEX Hash tree and Graph

Node Labeling

- Labeling the locations of elements in XML document
- Use two types of labeling schemes:
  - Interval labeling Schemes
  - Prefix labeling Schemes
- Properties
  - Locating the elements in XML document effectively
  - Determine the parent-child relationship quickly
    - BUS, D Shin, H Jang, H Jin, ACM DL'98
    - RRC (Relative Region Coordinate) Kha, D. D . Et al., 2001
    - XISS (XML Indexing and Storage System) Quanzhong Li et al., 2001

RRC Example

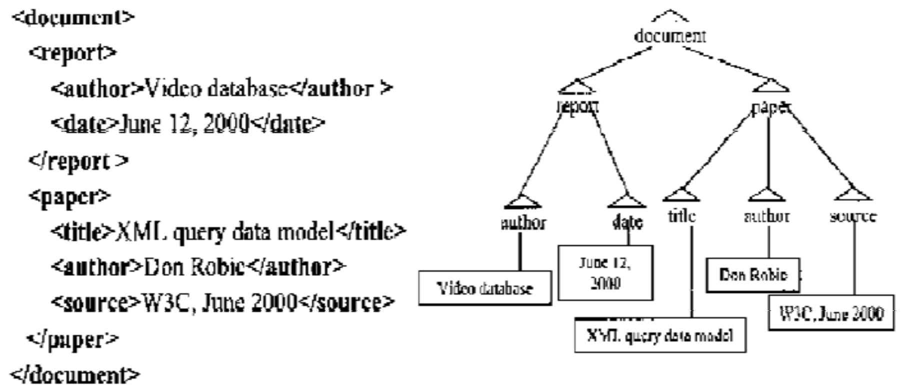
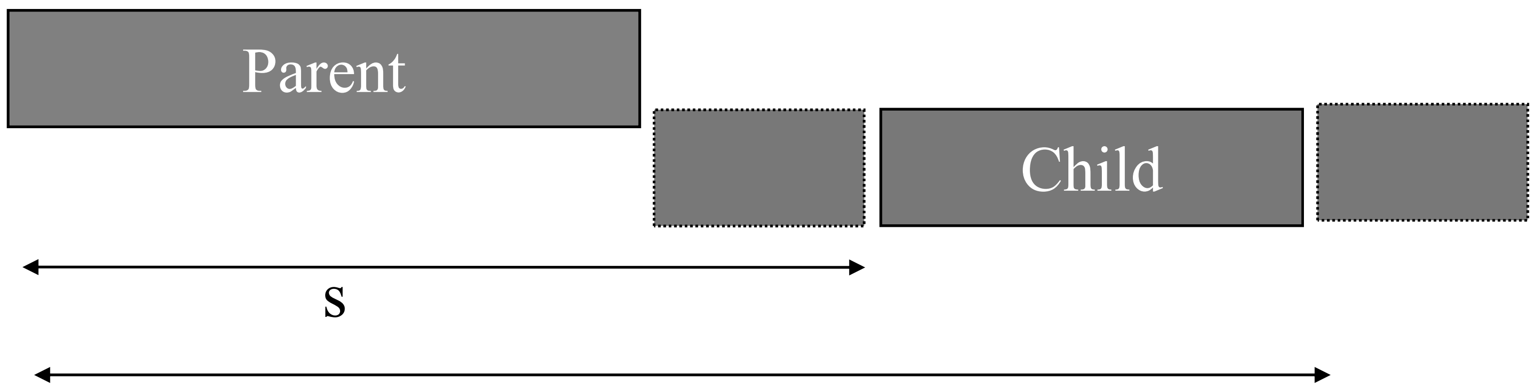


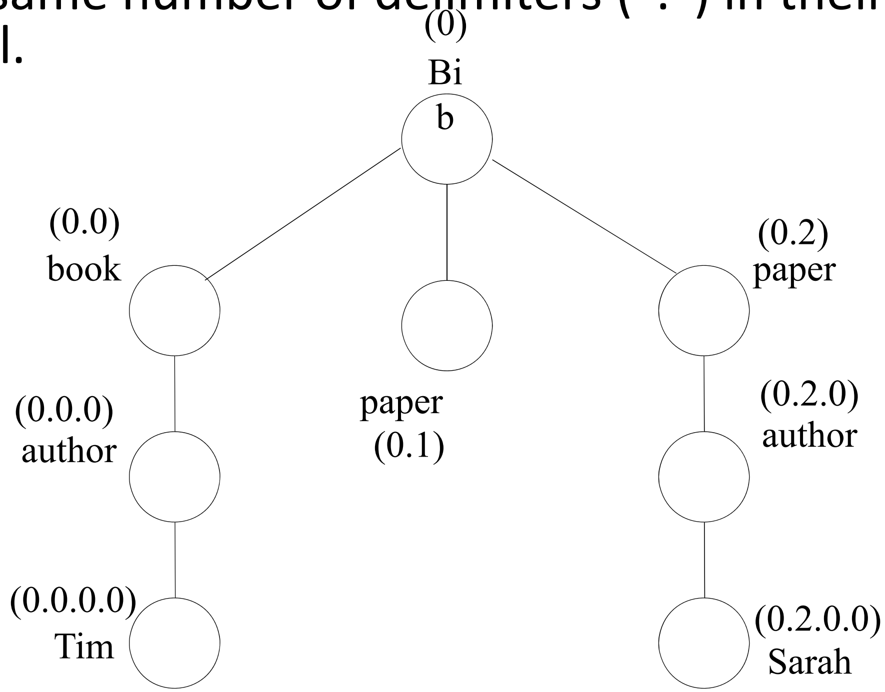
Figure 1. A simple XML document



Dewey - Structure

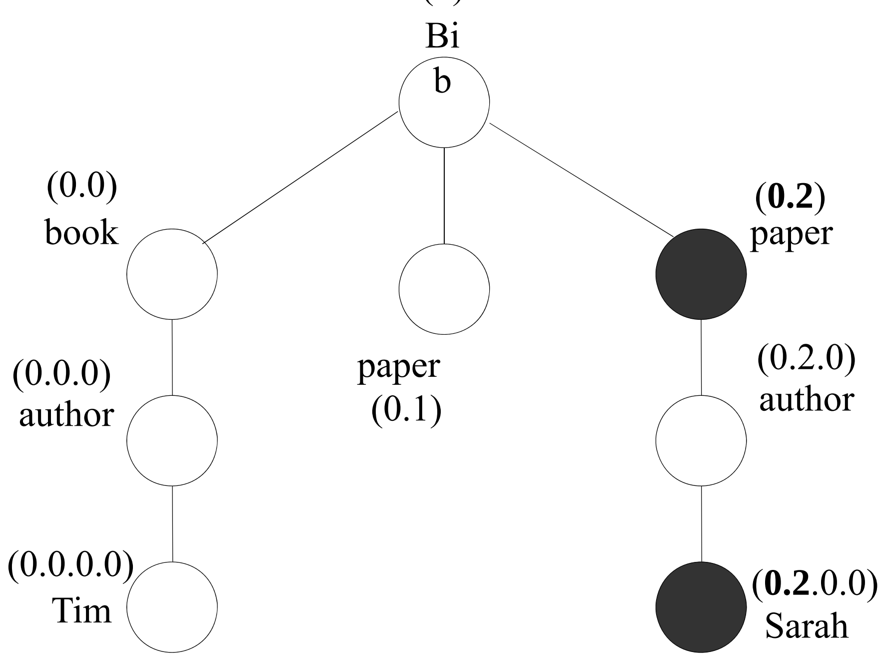
Tatarinov et al., 2002

- Each node is assigned a label that represents the path from the document’s root to the node.
- Each component of the label represents the local order of an ancestor node.
- Nodes with the same number of delimiters (“.”) in their label are in the same level.



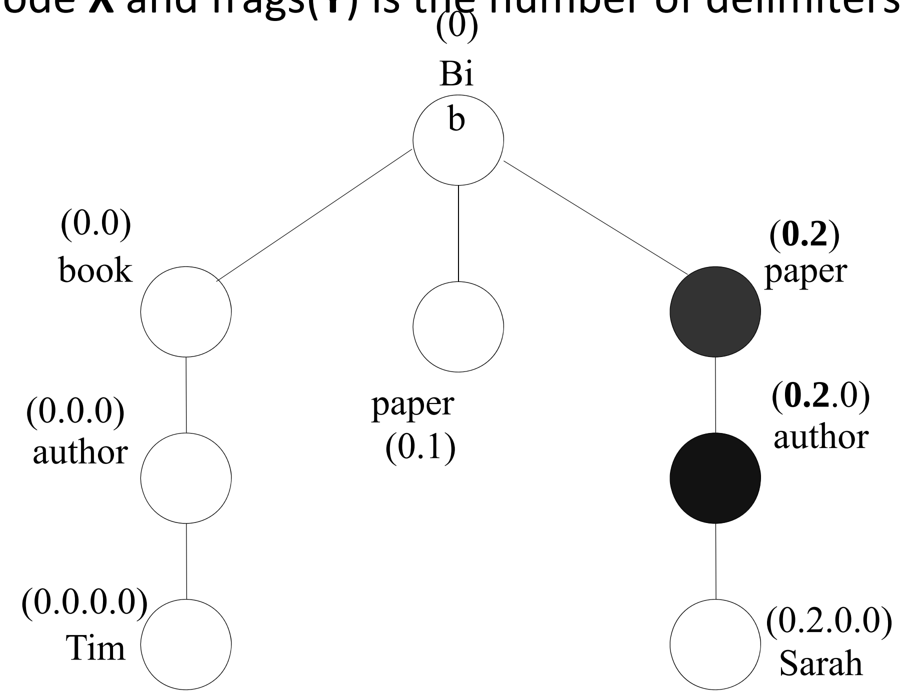
Dewey – Supported Queries (1/3)

- Ancestors / Descendants
  - Node “**X**” is an ancestor of node “**Y**” if the label of node “**X**” is a substring of the label of node “**Y**”.



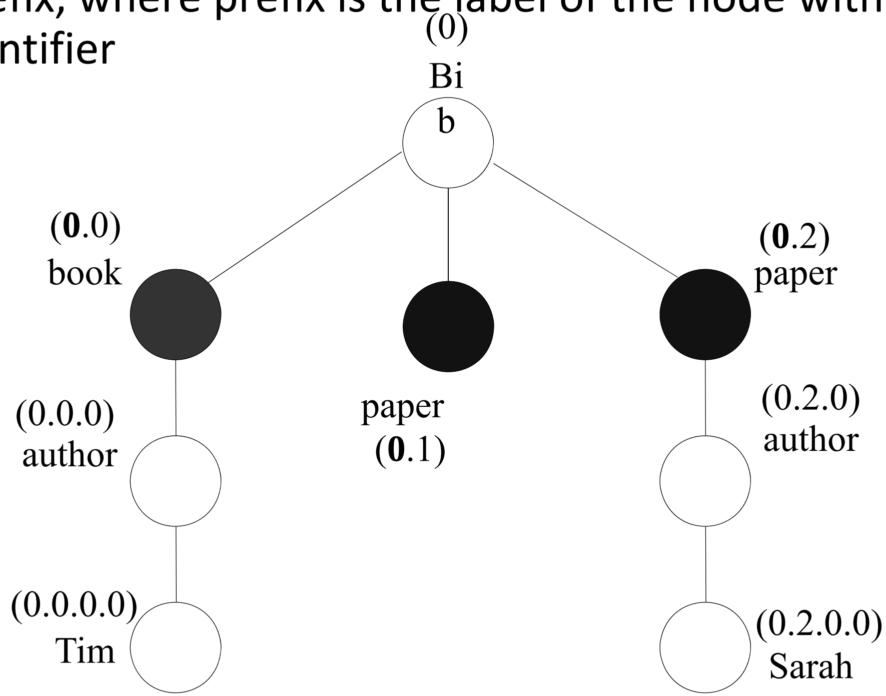
Dewey – Supported Queries (2/3)

- Parent / Child
  - Node “**X**” is parent of node “**Y**” if:
    - The label of node “**X**” is a substring of the label of node “**Y**” **and**
    - frags(**X**) = frags(**Y**) – 1, where frags(**X**) is the number of delimiters of the label of node **X** and frags(**Y**) is the number of delimiters of label of node **Y**.



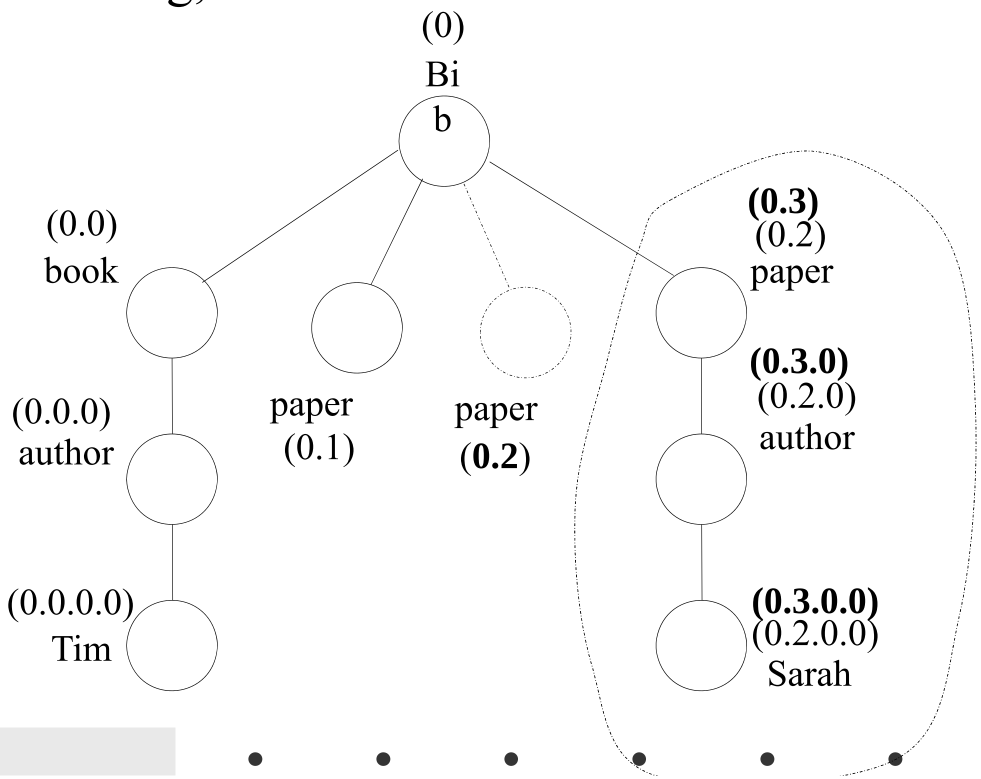
Dewey – Supported Queries (3/3)

- Siblings
  - Nodes “**X**” and “**Y**” are siblings if:
    - They have the same number of delimiters in their labels **and**
    - X.prefix = Y.prefix, where prefix is the label of the node without its positional identifier



Dewey – Updates

- Insertion of new node
  - The label of the nodes in the subtree rooted at the following sibling need to be updated
  - *O(n)* nodes need relabeling, where *n* is the number of nodes of the XML file



Dewey

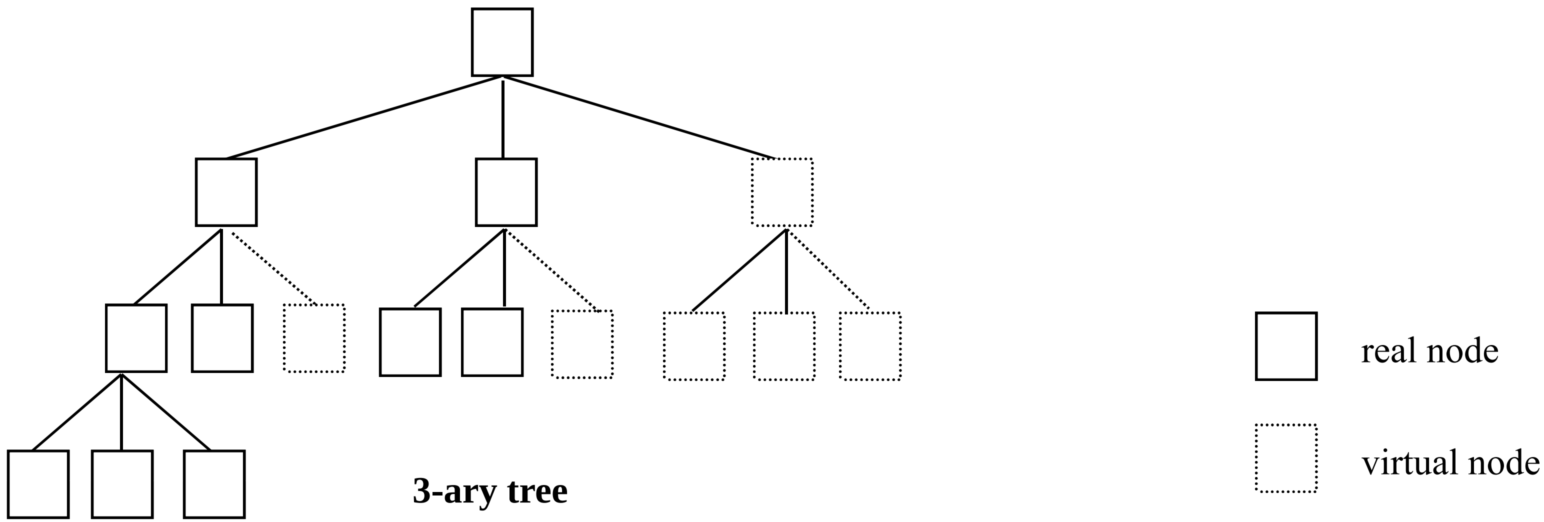
- Not efficient for dynamic XML files with many updates
  - Need to re-label many nodes
- As the depth of the tree increases:
  - Label size of a node increases rapidly
    - Storage size increases rapidly
  - It becomes more costly to infer the supported queries between any two nodes (the string prefix matching becomes longer)
- Overflow problem
  - *The original fixed length of bits assigned to store the size of the label is not enough*.

BUS

- BUS: An Effective Indexing and Retrieval Scheme in Structured Documents
  - D Shin, H Jang, H Jin, ACM DL'98
  - Xpath expression queries

Document Tree

- Lee et al, ACM DL 1996.
- Represent each document as a *k-ary* complete tree and assign a UID to each node



3-ary tree

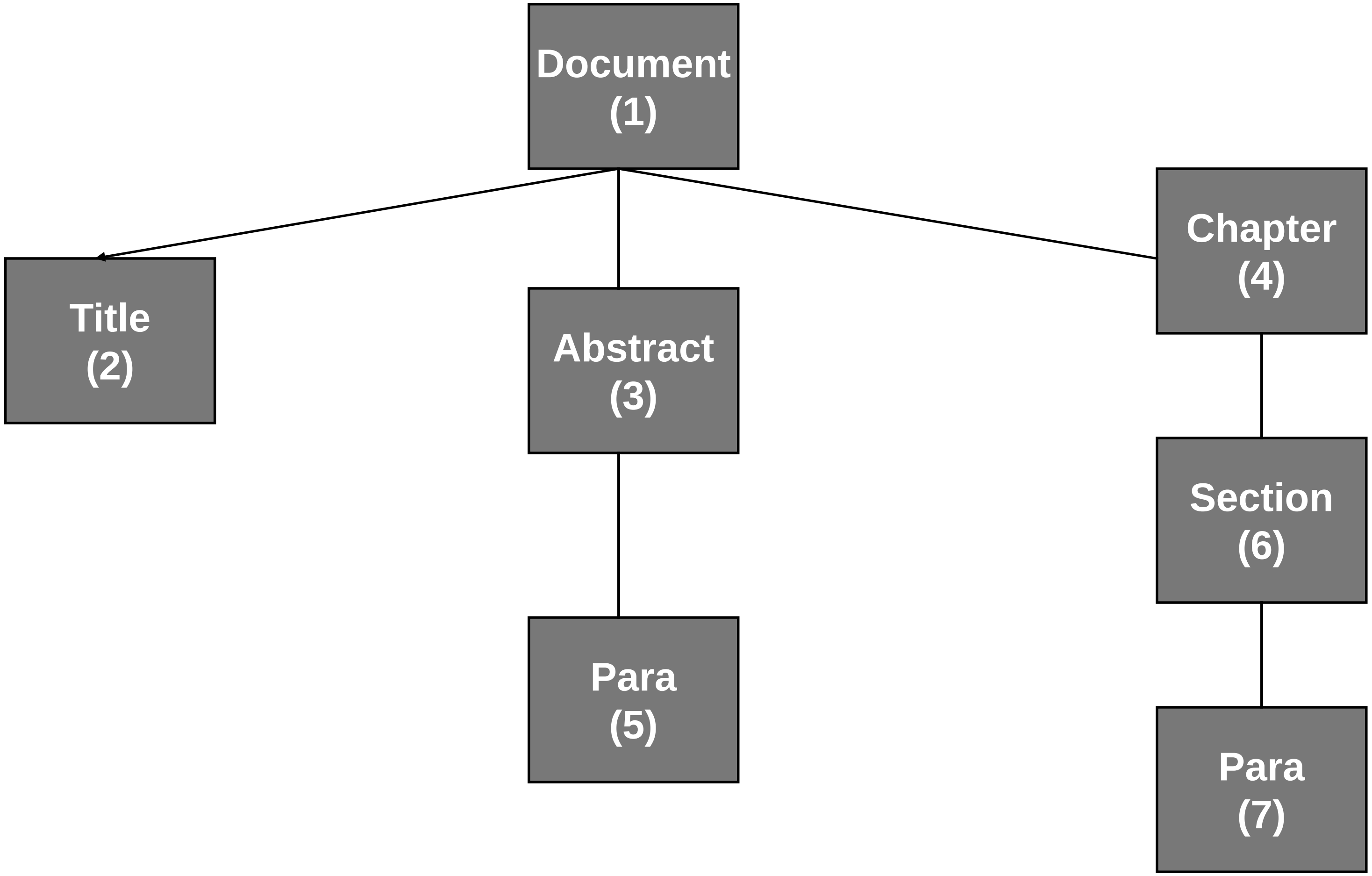
K-ary table

- Each document is assigned *k*, which is the maximum number of siblings in the document tree.
- Each element has an entry (row) in the *K-ary* table
- When a query is issued, the result set has pointers to the *K-ary* table.

Level and Element Type Number

- Level
  - Level means the level in the document tree
  - It gives a clue how many parent function is applied to get to a target element
- Element type number
  - A unique number is assigned to each element type in DTD It enables to filter out unnecessary elements and accumulate the correct frequencies
- Element location
  - The unique position of an element instance in a document tree

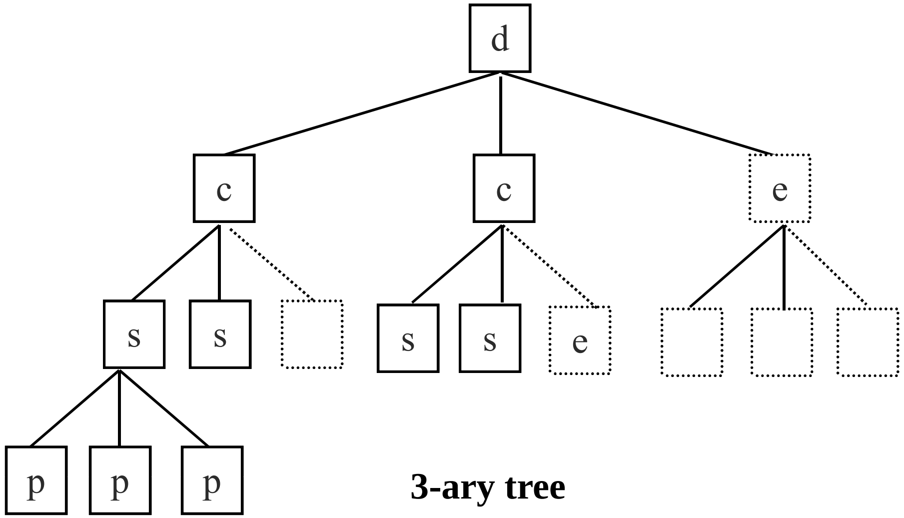
Element Labeling



UID

- Unique element identifier

parent(i) = [(i-2)/k+1]

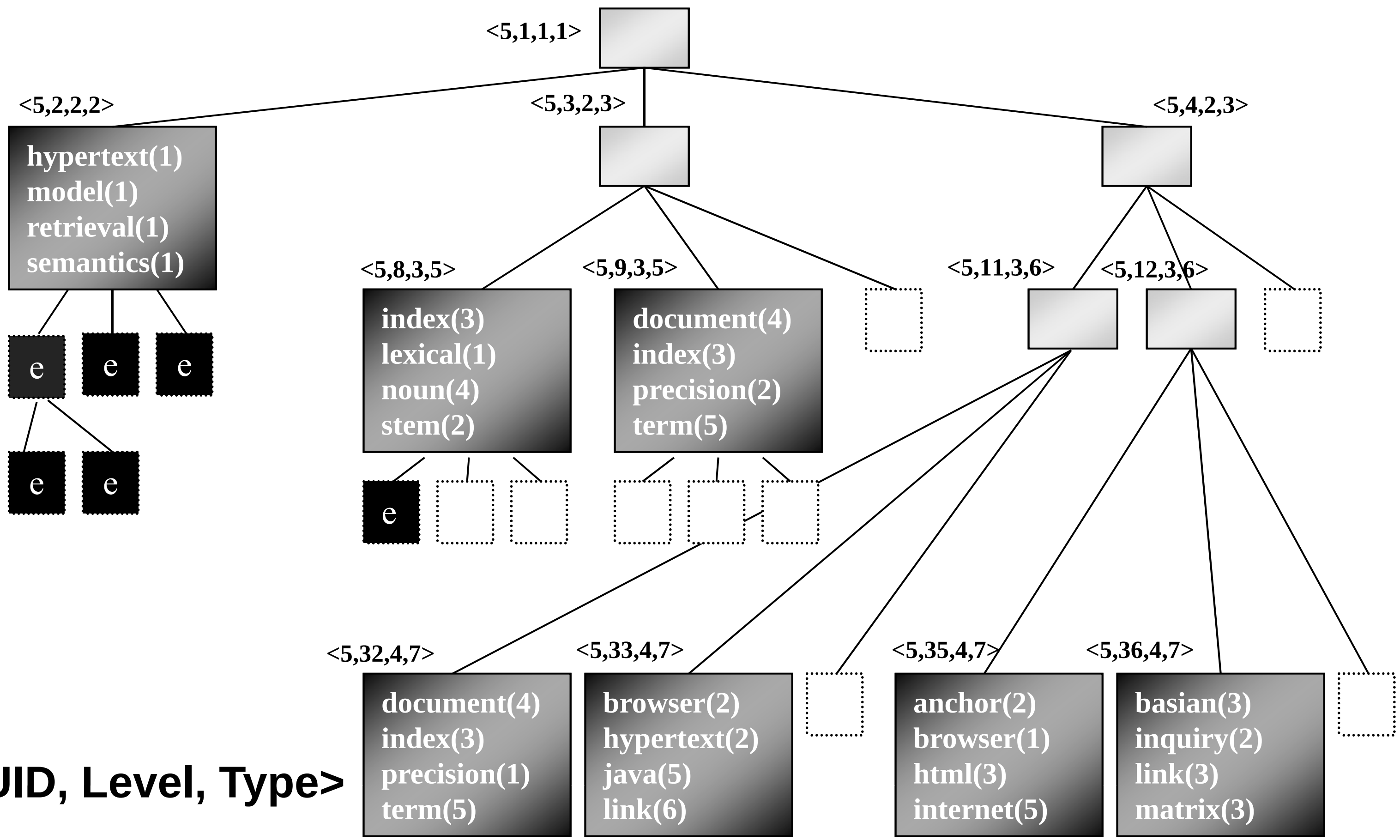


3-ary tree

Result of assigning UIDs

element	UID	element	UID
D1	1	S3	8
C1	2	S4	9
C2	3	P1	14
S1	4	P2	15
S2	5	P3	16

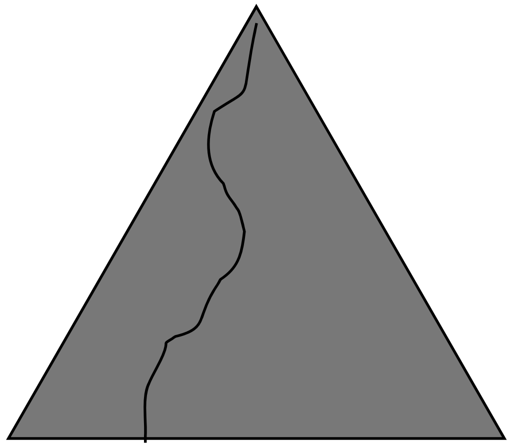
General Element Identifier



<docid, UID, Level, Type>

Indexing with GIDs

- Entries are of the form
  - (term, frequency, GID)
- Examples
  - (hypertext,1,<5,2,2,2>)
  - (model,1, <5,2,2,2>)
  - (index,3, <5,8,3,5>)
- Storing these information in a posting file
- Storing the keywords and index them with a B-tree

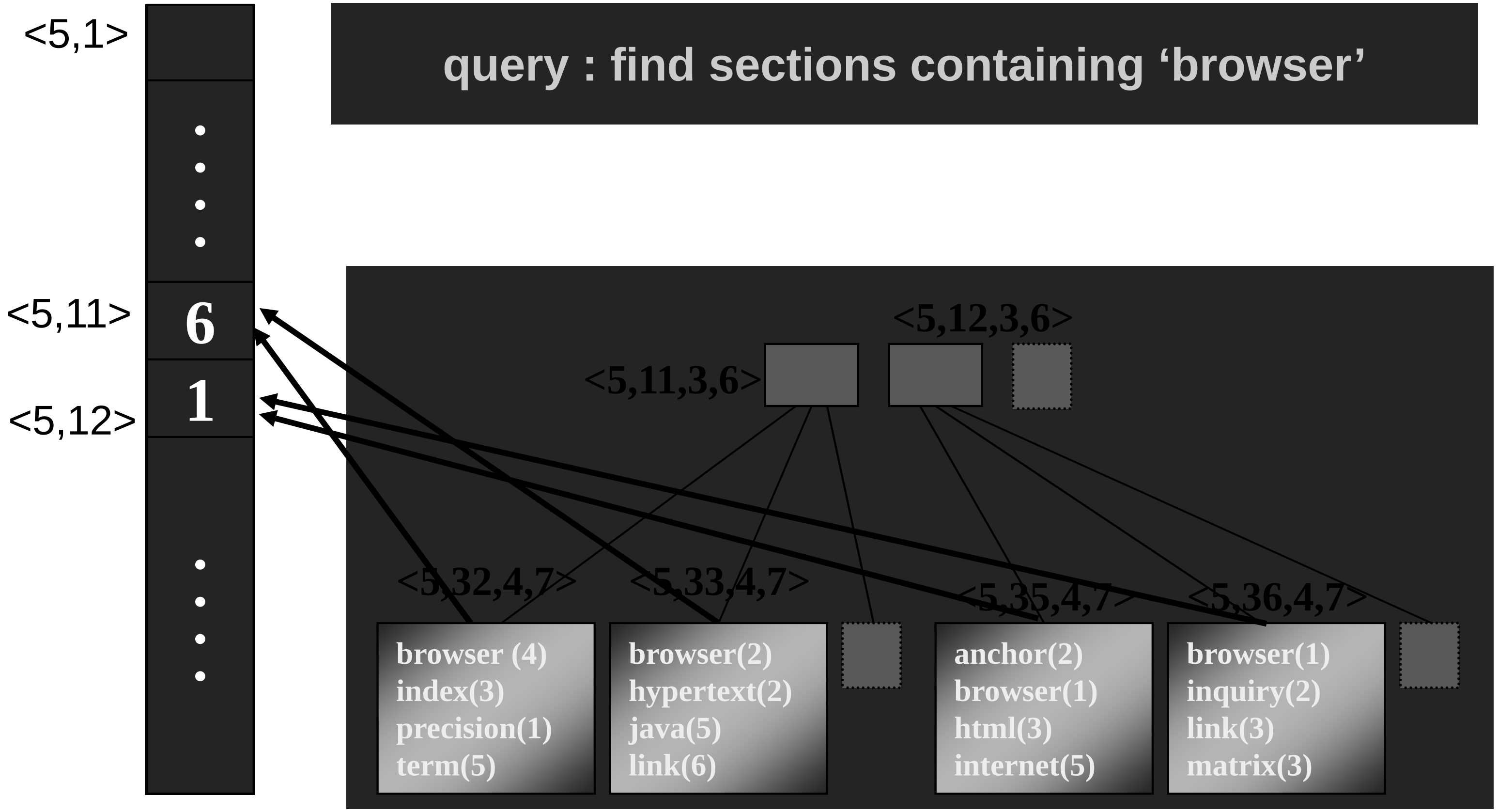


model

Query Evaluation

- Find out the section containing “browser”?
  - Level 3, element type is “section” or “para in section”
  - Access the posting file and extract the postings
  - A posting (browser,2,<5,33,4,7>) found
  - User level – text level = 1
  - Find the UID of the parent => 11
  - Can be used to sum up all frequencies from the sub-elements
    - For counting, document ranking, …
    - Need to use accumulators

Accumulators in a Hash Table



Oracle's XML Index

- Universal index for XML document collections
  - Indexes paths within documents
  - Indexes hierarchical information using dewey-style order keys
  - Indexes values as strings, numbers, dates
  - Stores base table rowid and fragment "locator"
- No dependence on Schema
  - Any data that can be converted to number or date is indexed as such regardless of Schema
- Option to index only subset of XPaths
- Allows Text (Contains) search embedded within XPath

XML Index Path Table (Oracle)

```
<po>
  <data>
    <item>foo</item>
    <pkg>123</pkg>
    <item>bar</item>
  </data>
</po>
```

BaseRid	Path	OrderKey	Value	Locator	NumValue
Rid1	po				
Rid1	po.data	1		7	
Rid1	po.data.item	1.1	“foo”	18	
Rid1	po.data.pkg	1.2	“123”	39	123
Rid1	po.data.item	1.3	“bar”	58	

OrdPath

- **ORDPATHs: Insert-Friendly XML Node Labels**
  - Patrick O'Neil, Elizabeth O'Neil1, Shankar Pal, Istvan Cseri, Gideon Schaller, Nigel Westbury
  - SIGMOD 2004
  - SQL Server 2005 implementation

OrdPath

- Aims to provide efficient insertion at any position of an XML tree, and also supports extremely high performance query plans for native XML queries.
- Tree modifications
  - new may be inserted
  - sub-trees be deleted
  - sub-trees may be moved around within the tree

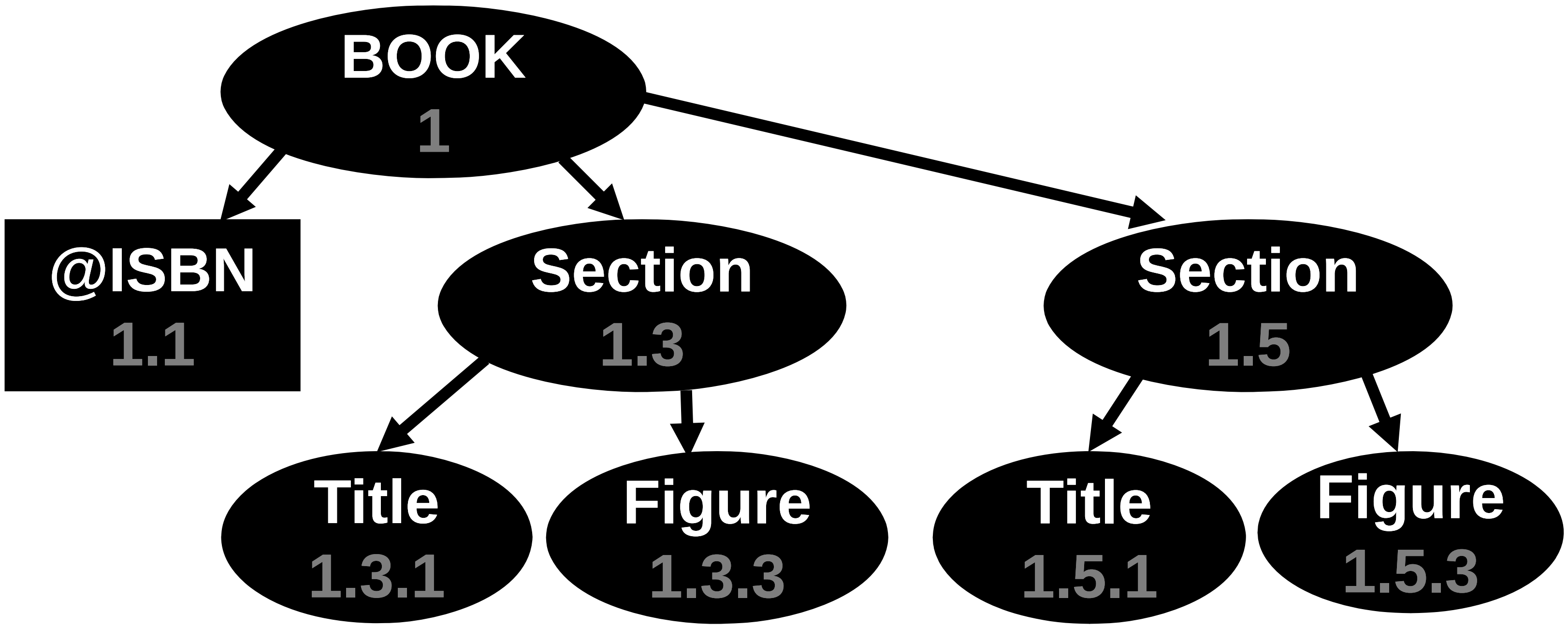
OrdPath

- Encodes the parent-child relationship by extending the parent's ORDPATH label with a component for the child.
  - E.g.: **1.5.3.9** might be the parent ORDPATH, **1.5.3.9.1** the child.
- The various child components reflect the children's relative sibling order, so that byte-by-byte comparison of the ORDPATH labels of two nodes yields the proper document order.
- A new node (possibly a root node of a sub-tree) can be inserted under any designated parent node in an existing tree.
  - Its label is generated using an additional intermediate "careting" component that falls between the components of its left and right siblings.

OrdPath

- At the beginning
  - Only *positive*, *odd* integers are assigned during an initial load; even-numbered and negative integer component values are reserved for later insertions into an existing tree
- Inserting in the middle
  - Even numbers are used as carets only. Do not count as components that increase the depth of the nodes.
  - E.g. new nodes in between 3.5.5 and 3.5.7
    - New siblings: 3.5.6.1, 3.5.6.2, …
    - A subtree: 3.5.6.1, 3.5.6.1.1, 3.5.6.3, 3.5.6.3.1, 3.5.6.3.3, 3.5.6.3.3.1, 3.5.6.3.3.3, 3.5.6.3.5, 3.5.6.5, 3.5.6.5.1

ORDPATH Label of Nodes



Node Table

ORDPATH	TAG	NODETYPE	VALUE
1	1 (Book)	10 (Element)	NULL
1.1	2 (ISBN)	2 (Attribute)	'1-55860-...'
1.3	3 (Section)	11 (Element)	NULL
1.3.1	4 (Title)	13 (Element)	'Bad Bugs'
1.3.3	5 (Figure)	12 (Element)	NULL
1.5	3 (Section)	11 (Element)	NULL
1.5.1	4 (Title)	13 (Element)	'Tree frogs'
1.5.3	5 (Figure)	12 (Element)	NULL

Primary Index

- A primary key (with a clustered index) on the NODE table provides efficient query access
- For each XML instance in base table, the index creates several rows of data
  - The number of rows in the index is approximately equal to the number of nodes in the XML binary large object.
- Primary key = (primary key ID, ORDPATH)

Indexing on Node Table

- A primary key (with a clustered index) on the NODE table provides efficient query access to XML data.
  – A query that retrieves all the descendents of X will find them clustered on disk just after X, in ORDPATH order

Compressed ORDPATH Format

- Successive variable-length Li/Oi bitstrings
- Each Li bitstring specifies the length in bits of the succeeding Oi bitstring.
  - Li bitstrings provide a number of important properties
  - Given that we know where an Li bitstring starts (as we do with L0), we can identify where it stops;
  - Each Li bitstring specifies the length in bits of the succeeding Oi bitstring;
  - Li bitstrings are generated to maintain document order;
  - Li/Oi components can specify *negative* ordinals Oi as well as positive ones; negative ordinals support multiple inserts of nodes to the left of a set of existing siblings.

Compressed ORDPATH Format

- A prefix encoding schemes for Li bitstrings
  - Li bitstring 01 identifies a component Li/Oi encoding with assigned length Li = 3, indicating a 3-bit Oi bitstring.
  - The following Oi bitstrings (000, 001, 010, . . ., 111) represent Oi values of the first eight integers, (0, 1, 2, . . ., 7).
    - 01101 is the bitstring for ORDPATH “5”.
  - Bitstring 100 identifies an encoding with Li = 4 and the 4-bit Oi bitstrings that follow represent the range [8, 23]
- Using the scheme,
  - ORDPATH = "1.5.3.-9.11"
    - 01 001 01 101 01 011 00011 1111 100 0011
    - L0=3 O0=1 L1=3 O1=5 L2=3 O2=3 L3=4 O3=-9 L4=4 O4=11

Query Execution

- An XQuery expression is translated into relational operations Consider the evaluation of the path expression
  - ‘Retrieve section titles in the book with the specified ISBN’:
  - /BOOK[@ISBN=‘1-55860-438-3”]/SECTION
  - SELECT SerializeXML (N2.ID, N2.ORDPATH)
    FROM NODE N1 JOIN NODE N2 ON (N1.ID = N2.ID)
    WHERE N1.PATH_ID = PATH_ID(/BOOK/@ISBN) AND
    N1.VALUE = '1-55860-438-3‘ AND
    N2.PATH_ID = PATH_ID(/BOOK/SECTION) AND
    Parent (N1.ORDPATH) = Parent (N2.ORDPATH)
- Note that the primary XML index is not used when retrieving a full XML instance.

Secondary Index

- Primary index may not provide the best performance for queries based on path expressions

- Performance slows down for large XML values.
  - all rows in the primary XML index corresponding to an XML BLOB are searched sequentially for large XML instances – slow!

- Having a secondary index built on the path values and node values in the primary index can significantly speed up the index search
  - PATH(PATH_VALUE), PROPERTY, VALUE, Content indexing

Secondary Index

- Secondary XML indexes help with bottom-up evaluation
  - After the qualifying XML nodes have been found in the secondary XML indexes, a back join with the primary XML index enables continuation of query execution with those nodes.

Indexing on Structure + Contents

- Indexing and Searching XML Documents based on Content and Structure Synopses
  - Weimin He, Leonidas Fegaras, David Levine, Bncod 2007
- Keyword queries are NOT adequate for XML search

An example query beyond Google:

Find the price of the book whose author's lastname is "Smith" and whose title contains "XML" and "SAX"

Semantic search using an XPath Query:

//book[author/lastname ~ "Smith"][title ~ "XML" and "SAX"]/price

- Simpler query formats cannot express complex containment relationships:

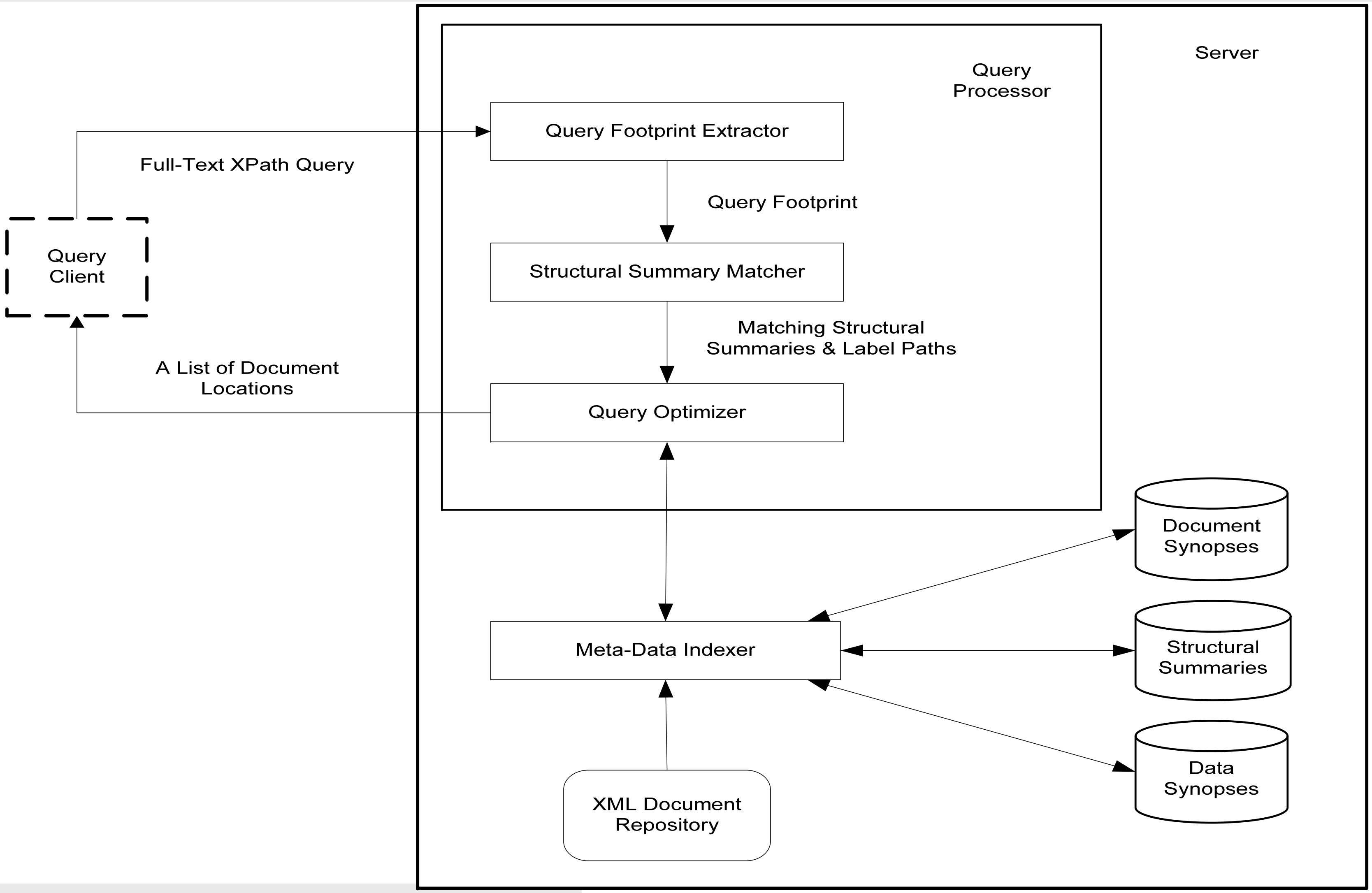
  [(lastname, Smith), (title, XML & SAX), price]

- Fully indexing XML data is neither efficient nor scalable

Key ideas

- A framework for indexing and searching schema-less XML documents based on data synopses extracted from documents
- Two novel data synopsis structures that can achieve higher query precision and scalability
- A hash-based processing algorithm to speed up searching

System Architecture



Specification of Search Queries

- XPath is extended with a simple IR syntax:

  Queries may contain predicates of the form: **e ~ S**

  - **e** is an XPath expression
  - **S** is a search predicate that takes the form:

    "term" | S1 and S2 | S1 or S2 | (S)

- A running query example:

  //auction//item[location ~ "Dallas"][description ~ "mountain" and "bicycle"]/price

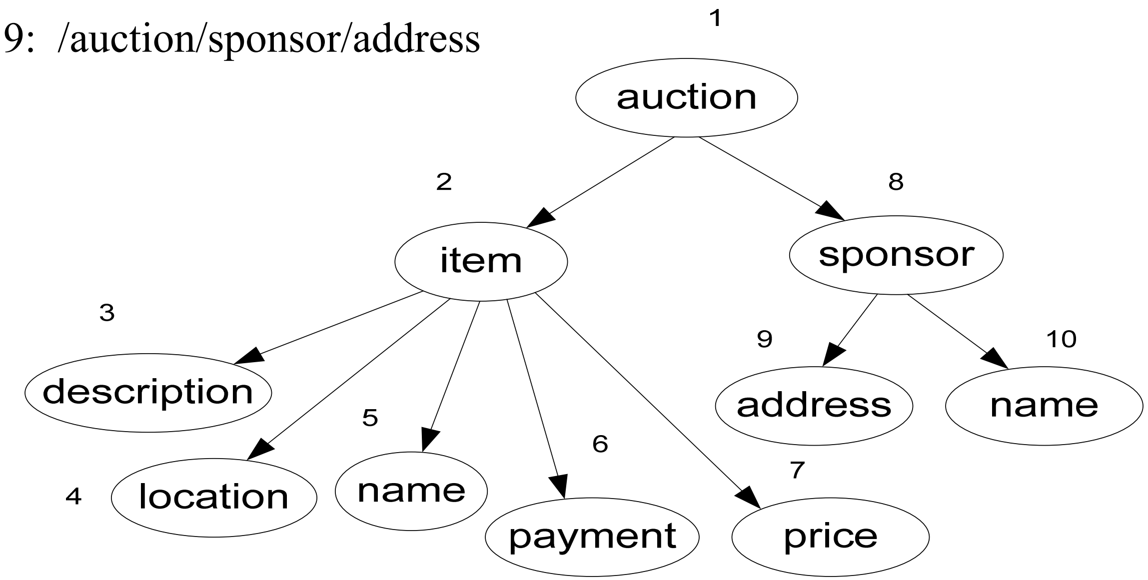
- Query result:

  A list of document locations (path names) that satisfy the query

Data Indexing

- **Structural Summary (SS)**
  - A tree that captures all unique paths in an XML document
  - It is constructed from XML data incrementally
  - Each SSnode# corresponds to a unique full label path:

9: /auction/sponsor/address



Data Indexing

- **Content Synopsis (CS)**
  - Summarizes the text associated with an SS node in an XML document
  - Approximated as a bit matrix of size W×L
    - W is the number of term buckets
    - L is the document positional ranges of elements that directly contain terms associated with node k
    - L is fixed but W may depend on the document size
  - Stored as a B^+-tree that implements the mapping

    $$(\text{SSnode\#}, \text{doc\#}) \rightarrow \text{bit-matrix}$$
  - Used in evaluating search predicates in the query

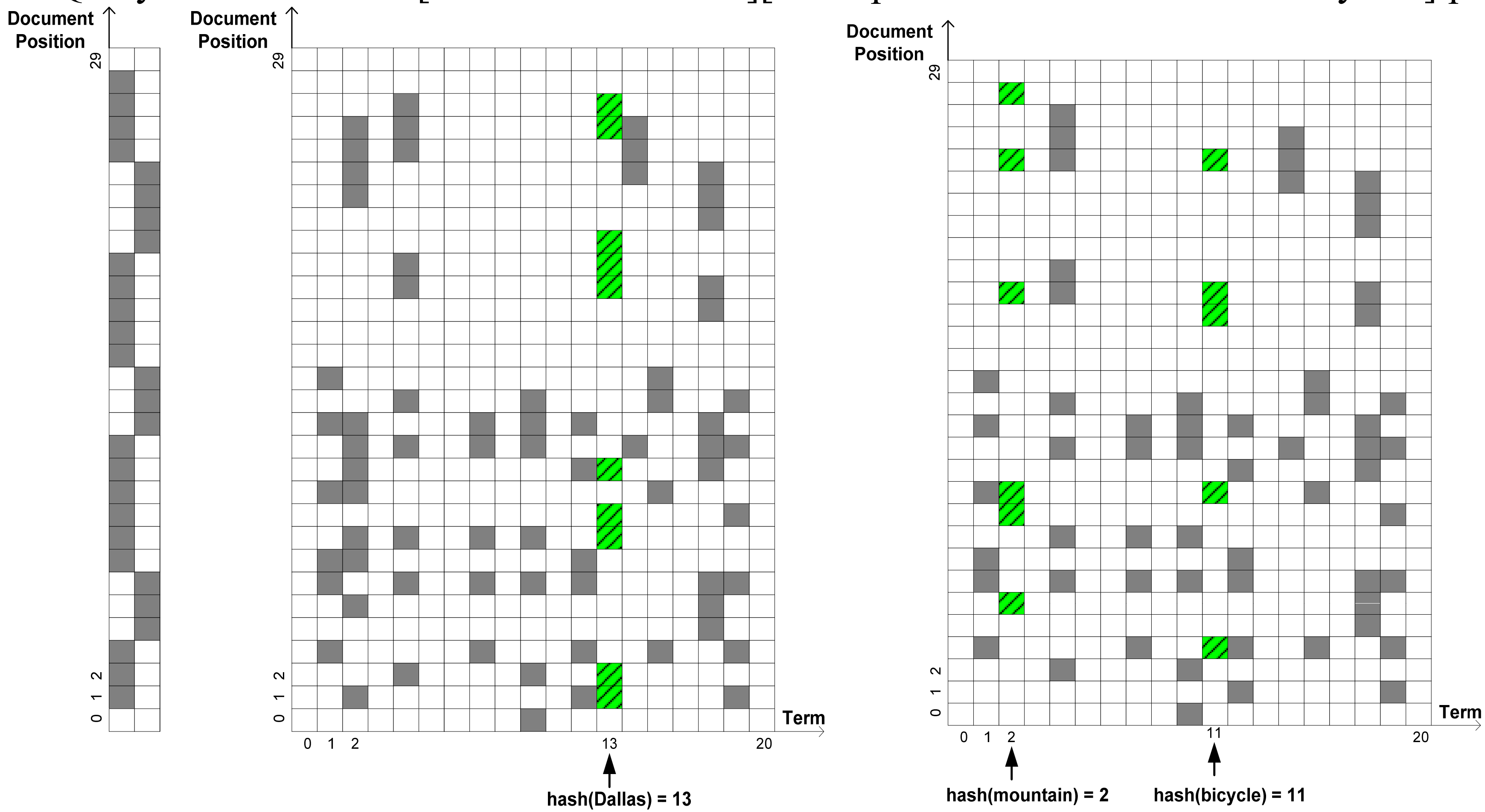
Data Indexing

- **Positional Filter (PF)**
  - Captures the position spans of all XML elements associated with an SS node in an XML document
  - Represented as a bit matrix of size M×L, where $M \geq 2$
  - Stored as a B^+-tree that implements the mapping

    $$(\text{SSnode\#}, \text{doc\#}) \rightarrow \text{bit-matrix}$$
  - Used in enforcing containment constraints *among* query predicates

Content Synopsis Example

Query: //auction//item[location ~ “Dallas”][description ~ “mountain” and “bicycle”]/price



Positional Filter for /auction/item

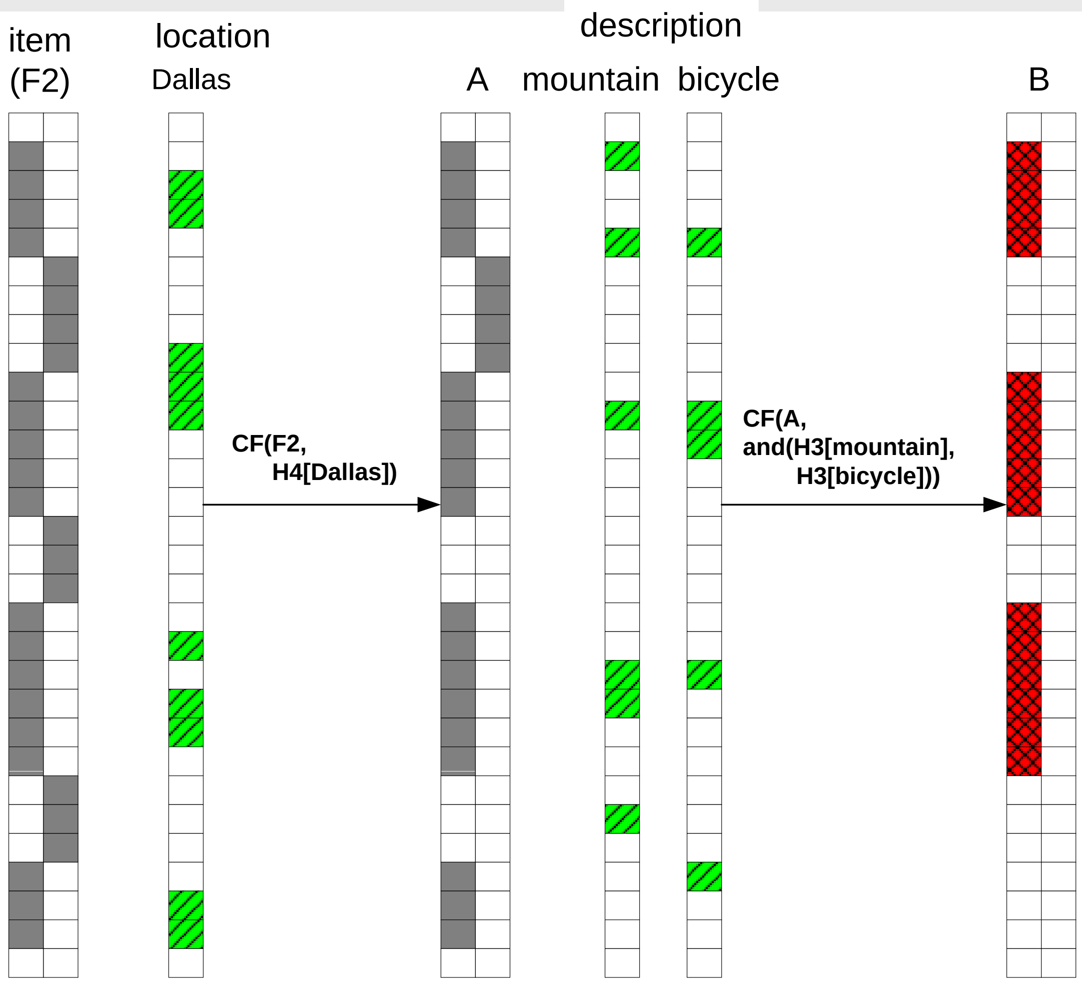
Content Synopsis for /auction/item/location

Content Synopsis for /auction/item/description

Containment Filtering

Query:

//auction//item[location ~ "Dallas"][description ~ "mountain" and "bicycle"]/price



Query Processing Overview

- Query Footprint (QF) Extraction
  - Query: //auction//item[location ~ “Dallas”][description ~ “mountain” and “bicycle”]/price
  - QF: //auction//item:0[location: 1][description: 2]/price
- Structural Summary Matching
  - Retrieve all structural summaries that match the QF
    - The standard preorder numbering scheme is used to represent an SS
    - An SS is stored as a B^+-tree that implements the mapping:

      tag → {(SS#, SSnode#, begin_word, end_word, level)}
    - We use containment joins to retrieve the qualified *full label paths* that match the entry points in the QF

  [/auction/item, /auction/item/location, /auction/item/description]
- Containment Filtering
  - Qualified document locations are collected and returned
    - The unit of query processing is a mapping from a doc# to a bit matrix of size M×L (positions)
    - An empty bit matrix means an unqualified document

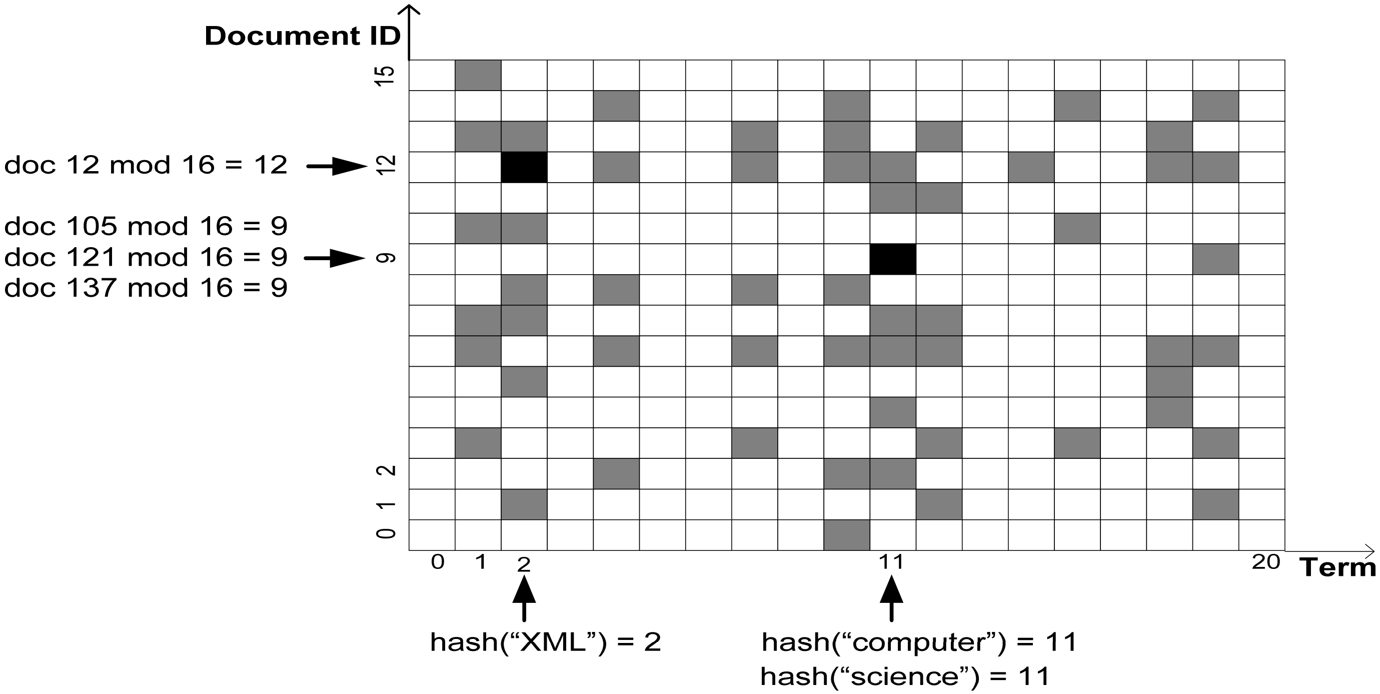
Two-Phase Containment Filtering

- **Many sources of inefficiency:**
  - A large number of full label path may match a single generic XPath query
  - A long list of data synopses has to be retrieved for each label path in a QF
  - The retrieved lists of data synopses have to be correlated at each step during containment filtering
- **Solution:**
  - Aggregate data synopses lists from multiple documents into a single bit matrix, called *Document Synopsis*, of size W×D

    path → bit-matrix

    so that, given a term t and a full label path p, the document doc# is a candidate if the document synopsis for p is set at [hash(t),hash(doc#)]
  - Need a two-phase containment filtering algorithm to prune unqualified document locations before the actual containment filtering

Document Synopsis



The document synopsis for /biblio/book/paragraph