Querying and storing XML

Week 4
XML Shredding
February 5-8, 2013

Storing XML data

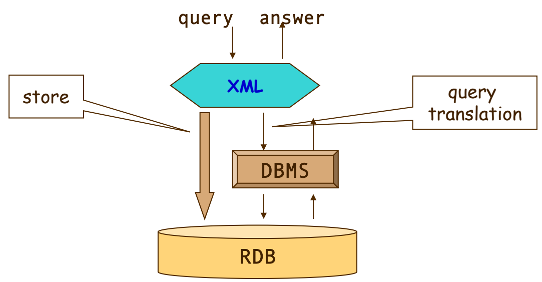
- Flat streams: store XML data **as is** in text files
  - fast for storing and retrieving whole documents
  - query support: limited; concurrency control: no
- Native XML Databases: designed **specifically** for XML
  - XML document stored in XML specific way
  - Goal: Efficient support for XML queries
- Colonial Strategies: **Re-use** existing DB storage systems
  - Leverage mature systems (DBMS)
  - Simple integration with legacy data
  - Map XML document into underlying structures
  - E.g., shred document into flat tables

Why transform XML data to relations?

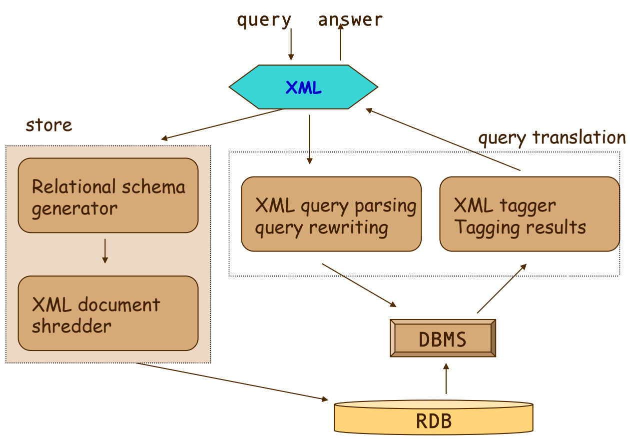
- Native XML databases need:
  - storing XML data, indexing,
  - query processing/optimization
  - concurrency control
  - updates
  - access control, . . .
  - **Nontrivial**: the study of these issues is still in its infancy – incomplete support for general data management tasks
- Haven't these already been developed for relational DBMS!?
- Why not take advantage of available DBMS techniques?

From XML (+ DTD?) to relations

- Store and query XML data using traditional DBMS
  - **Derive** a relational schema (generic or from XML DTD/schema)
  - **Shred** XML data into relational tuples
  - **Translate** XML queries to SQL queries
  - **Convert** query results back to XML



Architecture: XML Shredding



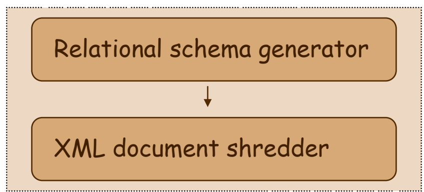
Nontrivial issues

- **Data model mismatch**
  - DTD: recursive, regular expressions/nested content
  - relational schema: tables, single-valued attributes
- **Information preservation**
  - lossless: there should be an effective method to reconstruct the **original** XML document from its relational storage
  - propagation/preservation of integrity constraints
- **Query language mismatch**
  - XQuery, XSLT: Turing-complete
  - XPath: transitive edges (descendant, ancestor)
  - SQL: first-order, limited / no recursion

Schema-conscious & selective shredding

Derivation of relational schema from DTD

- Should be lossless
  - the original document can be effectively reconstructed from its relational representation
- Should support querying
  - XML queries should be able to be rewritten to efficient relational queries



Running example – a book document

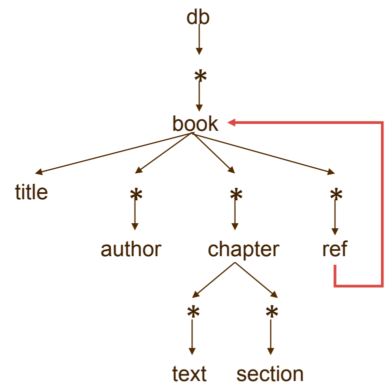
- DTD:

```
<!ELEMENT db   (book*)>
<!ELEMENT book (title,authors*,chapter*, ref*)>
<!ELEMENT chapter  (text | section)*>
<!ELEMENT ref  book>
<!ELEMENT title  #PCDATA>
<!ELEMENT author  #PCDATA>
<!ELEMENT section  #PCDATA>
<!ELEMENT text  #PCDATA>
```

- Recursive (`book`, `ref`, `book`, `ref`, ...)
- Complex regular expressions

Graph representation of the (simplified) DTD

- Each element type/attribute is represented by a unique node
- Edges represent the subelement (and attribute) relations
- *: 0 or more occurrences of subelements
- Cycles indicate recursion
  - e.g., book
- Simplification: e.g., (text | section)*
  - text* | section* -- ignore order

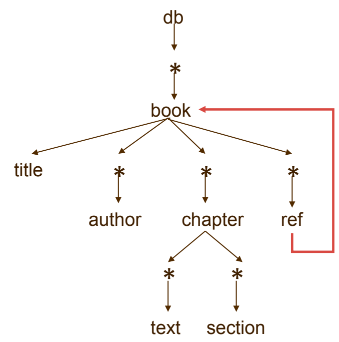


Canonical representation

- Store an XML document as a graph (tree)
  - Node relation: `node(nodeId, tag, type)`
  - e.g., `node(02, book, element)`, `node(03, author, element)`
- Edge relation: `edge(parent, child)`
  - `parent`, `child`: source and destination nodes; e.g., `edge(02, 03)`
- Pros and cons
  - Lossless: the original document can be reconstructed
  - Querying efficiency: Requires many joins
  - A simple query `/db/book[author="Bush"]/title` requires 3 joins of the edge relation!
  - `//book//title` - requires recursive SQL queries (not well supported)

Schema-conscious shredding/inlining

- Require DTD
- Represent the DTD as a graph (simplifying regular expressions)
- Traverse the DTD graph depth-first and create relations for the nodes
  - the root
  - each * node
  - each recursive node
  - each node of in-degree > 1
- Inlining: nodes with in-degree of 1 are inlined as fields
  - no relation is created



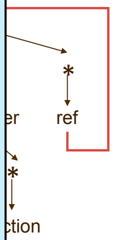
Schema-conscious shredding/inlining

- Re
- Re (si
- Tra an
  -
  -
  -
  -
- Inl are
  -

Assumption 1:
Order doesn't matter

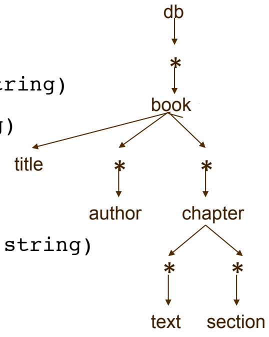
Assumption 2:
Correlations between elements don't matter
(a,b)* -> a*,b*

Resulting DTD still correct, but less precise



Relational schema

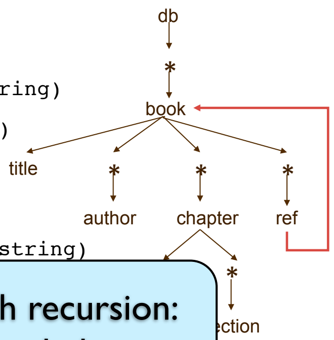
- db(<u>dbID</u>)
- book(<u>bookID</u>, parentID, title: string)
- author(<u>authorID</u>, bookID, author: string)
- chapter(<u>chapterID</u>, bookID)
- text(<u>textID</u>, chapterID, text: string)
- section(<u>sectionID</u>, chapterID, section: string)



- To preserve the semantics
  - ID: each relation has an artificial ID (key)
  - parentID: foreign key coding edge relation
  - Column naming: path in the DTD graph
- Note: `title` is inlined

Relational schema

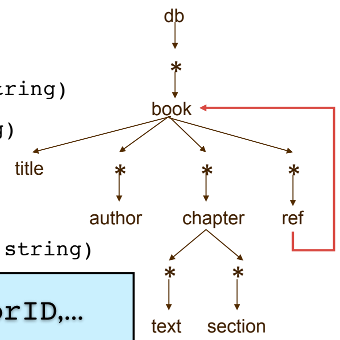
- db(<u>dbID</u>)
- book(<u>bookID</u>, parentID, **code,** title: string)
- author(<u>authorID</u>, bookID, author: string)
- chapter(<u>chapterID</u>, bookID)
- text(<u>textID</u>, chapterID, text: string)
- section(<u>sectionID</u>, chapterID, section: string)
- **ref(<u>refID</u>, bookID)**
- To preserve the semantics
  - ID: each relation has an artif
  - parentID: foreign key coding
  - Column naming: path in the
- Note: `title` is inlined



Dealing with recursion: code needed to distinguish book and `ref` parents

Relational schema

- db(<u>dbID</u>)
- book(<u>bookID</u>, parentID, **code,** title: string)
- author(<u>authorID</u>, bookID, author: string)
- chapter(<u>chapterID</u>, bookID)
- text(<u>textID</u>, chapterID, text: string)
- section(<u>sectionID</u>, chapterID, section: string)



Keys: `book.bookID`, `author.authorID`,...
Foreign keys:
`book.parentID` ⊆ `db.dbID` if code = 1
`book.parentID` ⊆ `ref.refID` if code = 0
`author.bookID` ⊆ `book.bookID`, ...

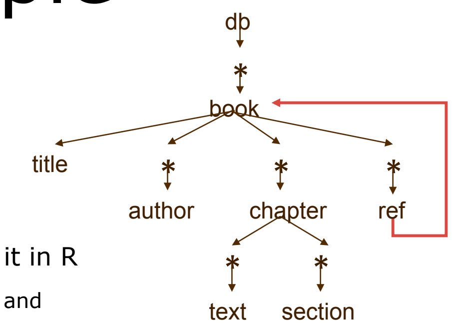
Summary of schema-driven shredding

- Use DTD/XML Schema to decompose document
- Shared inlining:
  - Rule of thumb: Inline as much as possible to minimize number of joins
  - Shared: do not inline if shared, set-valued, recursive
  - Hybrid: also inline if shared but not set-valued or recursive
- Reorganization of regular expressions:
  - `(text | section)* → text* | section*`
- Querying: Supports a large class of common XML queries
  - Fast lookup & reconstruction of inlined elements
  - Systematic translation unclear (not given in Shanmagusundaram et al.)
  - But can use **XML Publishing** techniques (next week)

Summary of schema-driven shredding (2)

- Instance mapping can be easily derived from schema mapping.
- Is it **lossless**? No
  - The order information is lost (simplification of regular expressions defining element types)
- Is there anything missing?
  - "core dumping" the entire document to a new database
  - In practice one often wants to select relevant data from the document
  - to store the selected data in an existing database of a **predefined schema**
- XML Schema: type + **constraints**
  - What happens to XML constraints? Can we achieve normal forms (BNCF, 3NF) for the relational storage?

Selective shredding example



- Existing relational database R :
  - book (id, title) ref (id1, id2)
- Select data from XML and store it in R
  - books with title containing "WMD", and
  - books cited, directly or indirectly
- Difference:
  - select only part of the data from an input document
  - store the data in an existing database with a fixed schema

Mapping specification: XML2DB mappings

- XML2DB Mapping:
  - Input: XML document T of a DTD D, and an existing database schema R
  - Output: a list of SQL inserts Δ_R, updating the database of R
- An extension of Attribute Grammars:
  - treat the DTD D as an ECFG (extended context-free grammar)
  - associate semantic attributes and actions with each production of the grammar
    - attributes: passing data top-down $book, …
    - actions: generate SQL inserts Δ_R
    - Evaluation: generate SQL inserts in parallel with XML parsing
- **[Fan, Ma DEXA 2006] --- see additional readings**

XML2DB mappings

- **Simplified DTD:** element type definitions e → r where
  - r ::= PCDATA | ε | $a_1, \ldots, a_n$ | $a_1 + \ldots + a_n$ | a*
  - Note: **subset** of full DTD regexps (e.g. (a|b)*,c not directly allowed)
- Relation variables: for each relation schema Ri, define a variable Δ_{Ri}, which holds tuples to be inserted into Ri
- Attributes: $e associated with each element type e
  - $e: tuple-valued, to pass data values top-down
- Associate "semantic actions" with each e → r
  - written rule(a -> r)

Semantic actions

rule(p) ::= *stmts*

stmts ::= ε | *stmt* ; *stmts*

stmt ::= \$a := $(x_1,\ldots,x_n)$ | $\Delta_{Ri} := \Delta_{Ri} \cup \{(x_1,\ldots,x_n)\}$ | id = **gen_id**()

| **if** C **then** *stmt* **else** *stmt*

x ::= \$b.A | **text**(b) | str | id | ⊤ | ⊥

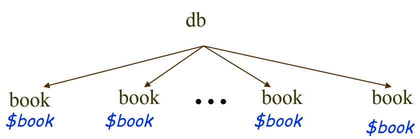
C ::= x = x' | x <> x' | x contains x' | ...

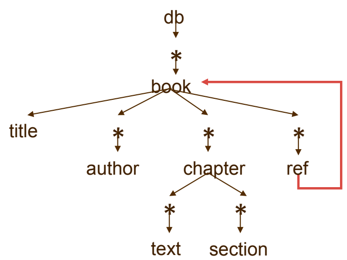
- Given (a -> r), rule(a -> r) can read from (fields of) \$a and should assign values to \$b for each element name b appearing in r
  - Can also extract values of text fields of a using **text**(b) (left to right)
  - Special values "top" and "bot", fresh IDs
  - Rules can also generate tuples to be added to relations Δ_{Ri}
- Conditional tests C can include equality, string containment, ...

Example: XML2DB mapping

```
db → book*

   $book  := top    /* children of the root */
```

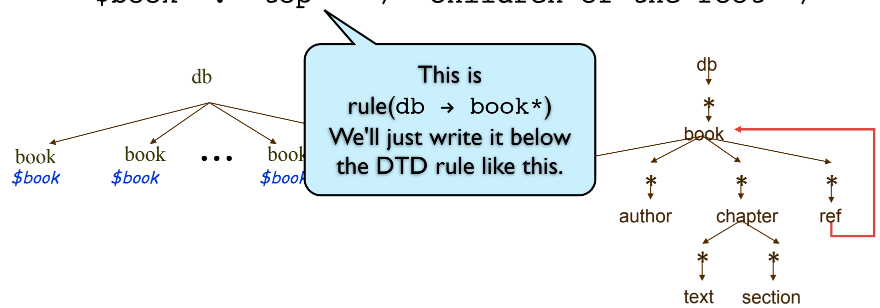




Example: XML2DB mapping

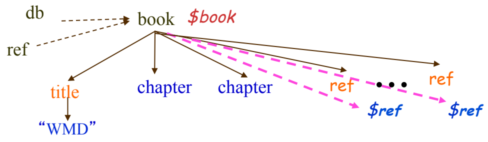
```
db → book*

   $book  := top    /* children of the root */
```



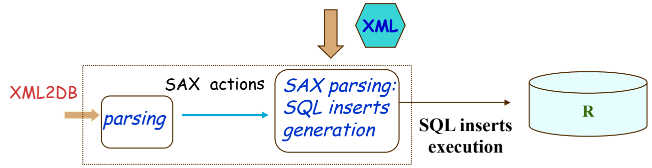
Example: Semantic action

```
book → title, author*, chapter*, ref*
    if (text(title) contains "WMD"
        or ($book <> ⊤ and $book <> ⊥))
    then id := gen_id( );
         book := Δbook ∪ { (id, text(title)) };
         if $book <> ⊤
         then ref := Δref ∪ { ($book, id) };
         $ref := id;
    else  $ref := ⊥
```



- target relation schema: `book (id, title), ref (id1, id2)`
- `gen_id( )`: a function generating a fresh unique id
- conditional: title is "WMD" or is referenced by a book of title "WMD"

Implementing XML2DB mappings



- SAX parsing extended with corresponding semantic actions
  - `startDocument( ), endDocument( )`
  - `startElement(A, eventNo), endElement(A), text(s)`
- SQL updates:

```
insert into book
select *
from   Δbook
```


Schema-oblivious shredding and indexing

Schema-oblivious storage

- Storage easier if we have a fixed schema
- But:
- Often don't have schema
- Or schema may change over time
  - schema updates require reorganizing or reloading! Not fun.
- Alternative: **schema-oblivious** XML storage

Stupid idea #1: CLOB

- Well, XML is just text, right?
- Most databases allow CLOB (Character Large Object) columns - unbounded length string
- So you just store the XML text in one of these
- Surprisingly popular
  - and can make sense for storing "document-like" parts of XML data (eg HTML snippets)
  - But not a good idea if you want to query the XML

Stupid (?) idea #2: SQL/XML

- Instead of blindly using CLOBs...
- Extend SQL with XML-friendly features
  - "XML" column type
  - Element/attribute construction primitives
  - Ability to run XPath or XQuery queries (or updates) on XML columns
- Also surprisingly popular (MS, IBM, Oracle)
  - Pro: At least DB knows it's XML, and can (theoretically) act accordingly (e.g. store DOM tree, shred, use XML DB, ...)
  - Pro?: Part of SQL 2003 (SQL/XML extensions)
  - Con: Frankenstein's query language

SQL/XML example

```
CREATE TABLE Customers(
  CustomerID int PRIMARY KEY,
  CustomerName nvarchar(100),
  PurchaseOrders XML, ...}
```

```
SELECT CustomerName,
       query(PurchaseOrders,
   'for $p in /po:purchase-order
    where $p/@date < xs:date("2002-10-31")
    return <purchaseorder date="{$p/@date}">
               {$p/*}
             </purchaseorder>')
FROM  Customers
WHERE CustomerID = 42
```


Schema-oblivious shredding/indexing

- Can we store arbitrary XML in a relational schema (even without DTD)?
- Of course we can (saw last time):
  - `node(nodeID, tag, type)`
  - `edge(parent, child)`
  - `attribute(nodeID, key, value)`
  - `text(nodeID, text)`
- What's wrong with this?

Quiz

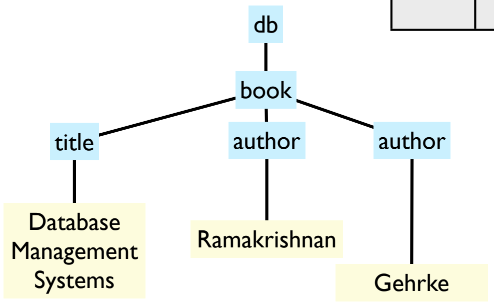
- Fill in tables
- Write SQL query for:
- /db/book/title/text()

edge

parent	child

node

nodeId	tag	type



text

nodeId	text

Quiz

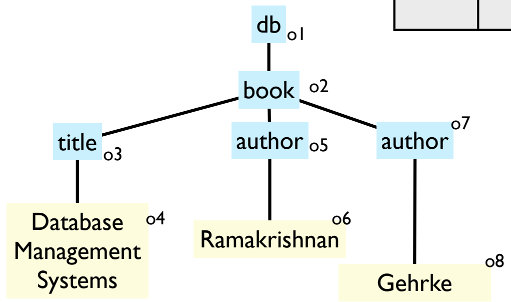
- Fill in tables
- Write SQL query for:
- /db/book/title/text()

edge

parent	child

node

nodeId	tag	type



text

nodeId	text

Quiz

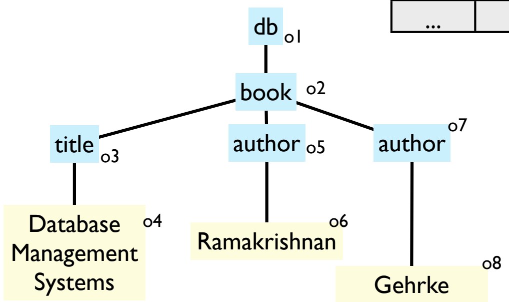
- Fill in tables
- Write SQL query for:
- /db/book/title/text()

edge

parent	child
o1	o2
o2	o3
o3	o4
...	...

node

nodeId	tag	type



text

nodeId	text

Quiz

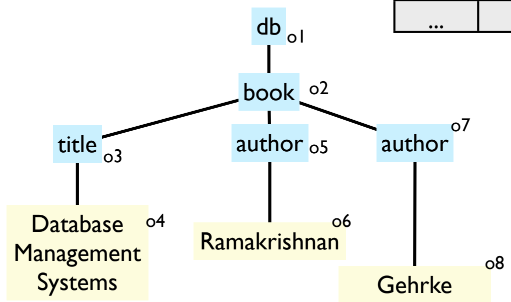
- Fill in tables
- Write SQL query for:
- /db/book/title/text()

edge

parent	child
o1	o2
o2	o3
o3	o4
...	...

node

nodeId	tag	type
o1	db	ELT
o2	book	ELT
o4		TEXT
...	...	...



text

nodeId	text

Quiz

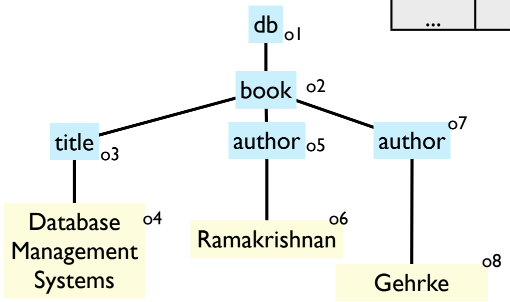
- Fill in tables
- Write SQL query for:
- /db/book/title/text()

edge

parent	child
o1	o2
o2	o3
o3	o4
...	...

node

nodeId	tag	type
o1	db	ELT
o2	book	ELT
o4		TEXT
...	...	...



text

nodeId	text
o4	Database Management Systems
o6	Ramakrishnan
o8	Gehrke

Quiz

/db/book/title/text() in SQL:

```
SELECT txt.text
FROM node w, edge e1,
     node x, edge e2,
     node y, edge e3,
     node z, text txt
WHERE w.tag = "db" AND w.type = "ELT"
  AND e1.parent = w.nodeId
  AND e1.child = x.nodeId
  AND x.tag = "book"
  AND ...
  AND z.type = "TEXT"
  AND z.nodeId = txt.nodeId
```


Problems with edge storage

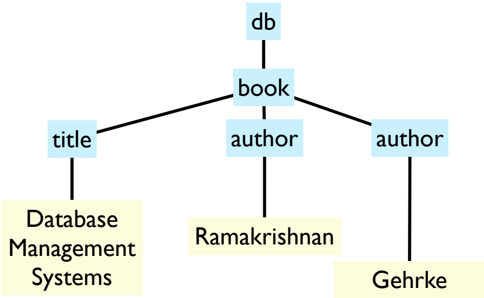
- Indexing unaware of tree structure
  - hard to find needles in haystacks
  - fragmentation - subtree might be spread across db
- Incomplete query translation
  - descendant axis steps involve recursion
  - need additional information to preserve document order
  - filters, sibling, following edges also painful
- Lots of joins
  - joins + no indexing = trouble

Node IDs and Indexing

- Idea: Embed **navigational** information in each node's **identifier**
- Then indexing the ids can improve query performance
  - and locality, provided ids are ordered (and order ~ tree distance)
- Two main approaches (with many refinements):
  - Dewey Decimal Encoding
  - Interval Encoding

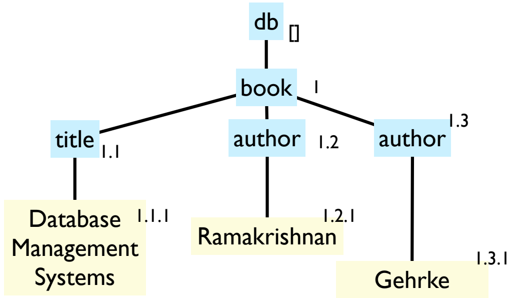
Dewey Decimal Encoding

- Each node's ID is a list of integers
  - $[i_1,i_2, \ldots ,i_n]$ (often written $i_1.i_2. \ldots .i_n$)
  - giving the "path" from root to this node



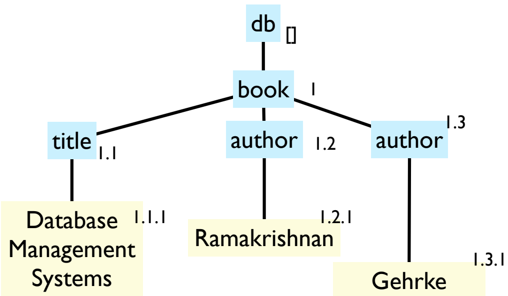
Dewey Decimal Encoding

- Each node's ID is a list of integers
  - $[i_1,i_2, \ldots ,i_n]$ (often written $i_1.i_2. \ldots .i_n$)
  - giving the "path" from root to this node



Dewey Decimal Encoding

- Each node's ID is a list of integers
  - $[i_1,i_2, \ldots ,i_n]$ (often written $i_1.i_2. \ldots .i_n$)
  - giving the "path" from root to this node



nodeID	tag	type
[]	db	ELT
1	book	ELT
1.1	title	ELT
1.1.1		TEXT
1.2	author	ELT
1.2.1		TEXT
1.3	author	ELT
1.3.1		TEXT

Querying

- Descendant (or self) = (strict) prefix
  - Descendant(p,q) $\Leftrightarrow p \prec q$
  - DescendantOrSelf(p,q) $\Leftrightarrow p \preccurlyeq q$
- Child: immediate prefix
  - Child(p,q) $\Leftrightarrow p \prec q$ and $|p| + 1 = |q|$
- Parent, ancestor : reverse p and q

Querying

- Descendant (or self) = (strict) prefix
  - Descendant(p,q) ⇔ $p \prec q$
  -
- C
  -
- Pa

> Prefix:
> $1 \prec 1.2 \prec 1.2.3 \prec 1.2.3.4.5$
> ...
> Length:
> $|1.2.3| = 3$
> $|3.2.1.2| = 4$
> ...

Querying

- Descendant (or self) = (strict) prefix
  - Descendant(p,q) ⇔ $p \prec q$
  - DescendantOrSelf(p,q) ⇔ $p \preccurlyeq q$
- Child: immediate prefix
  - Child(p,q) ⇔ $p \prec q$ and $|p| + 1 = |q|$
- Parent, ancestor : reverse p and q

Example

- Extend SQL with prefix, length UDFs
- How to solve //a//b[c]?

```
SELECT b.nodeID
FROM node a, node b
WHERE a.tag = 'a', b.tag = 'b'
  AND PREFIX(a.nodeID,b.nodeID)
  AND EXISTS(SELECT *
             FROM node c
             WHERE c.tag='c'
               AND PREFIX(b.nodeID,c.nodeID)
               AND LEN(b.nodeID) + 1 =
                   LEN(c.nodeID))
```


Example

- Extend SQL with prefix, length UDFs
- How to solve //a//b[c]?

```
SELECT b.nodeID
FROM node a, node b
WHERE a.tag = 'a', b.tag = 'b'
  AND PREFIX(a.nodeID,b.nodeID)
  AND EXISTS(SELECT *
             FROM node c
             WHERE c.tag='c'
               AND PREFIX(b.nodeID,c.nodeID)
               AND LEN(b.nodeID) + 1 =
                   LEN(c.nodeID))
```

> //a//b

Example

- Extend SQL with prefix, length UDFs
- How to solve //a//b[c]?

```
SELECT b.nodeID
FROM node a, node b
WHERE a.tag = 'a', b.tag = 'b'
  AND PREFIX(a.nodeID,b.nodeID)
  AND EXISTS(SELECT *
             FROM node c
             WHERE c.tag='c'
               AND PREFIX(b.nodeID,c.nodeID)
               AND LEN(b.nodeID) + 1 =
                   LEN(c.nodeID))
```

//a//b

[c]

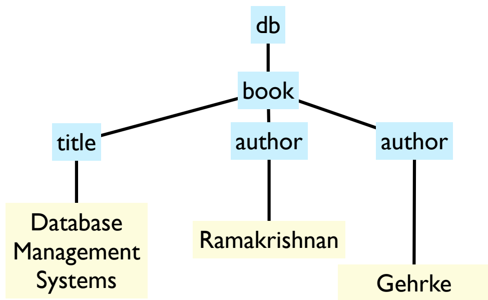
Sibling, following axis steps

- Following Sibling: same immediate prefix, with final step
  - Sibling(p,q) $\Leftrightarrow \exists r.\ p = r.i$ and $q = r.j$ and $i < j$
  - can also define this as a UDF
- Following: Definable as composition of ancestor, following-sibling, descendant
  - or: $\exists r.\ p = r.i.p'$ and $q = r.j.q'$ and $i < j$
- Preceding-sibling, preceding: dual (swap p,q)

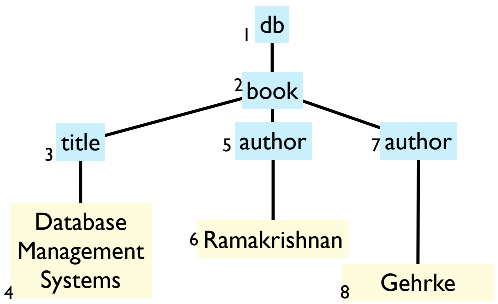
Interval encoding

- Drawback of DDE: needs strings, UDFs
  - DBMS may not know how to optimize, rewrite effectively for query optimization
- But RDBMSs generally support numerical values, indexing, rewriting
  - most business applications involve numbers after all...
- Interval encoding: alternative ID-based indexing/ shredding scheme
  - IDs are pairs of numbers
  - Several ways of doing this

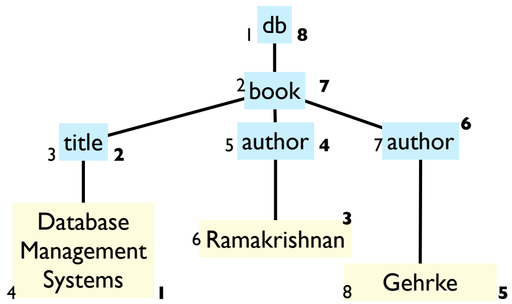
Pre/post numbering



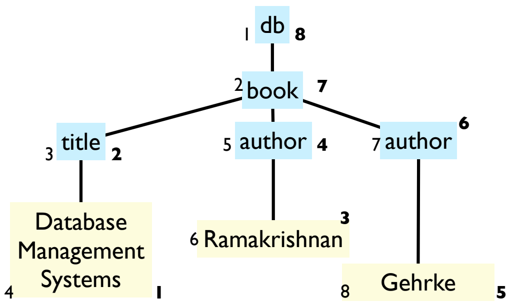
Pre/post numbering



Pre/post numbering

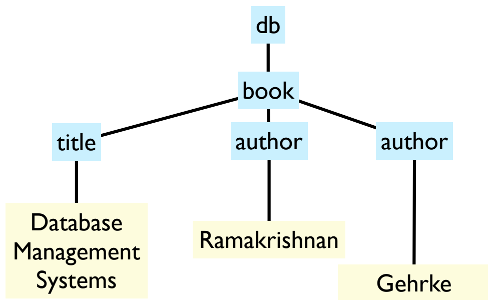


Pre/post numbering

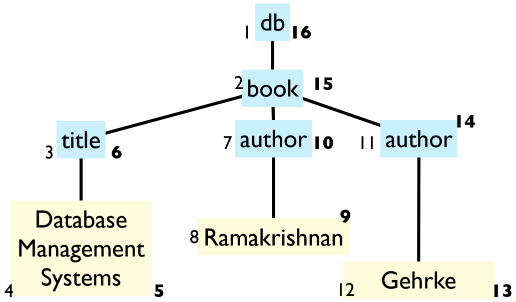


pre	post	par	tag	type
1	8		db	ELT
2	7	1	book	ELT
3	2	2	title	ELT
4	1	3		TEXT
5	4	2	author	ELT
6	3	5		TEXT
7	6	2	author	ELT
8	5	7		TEXT

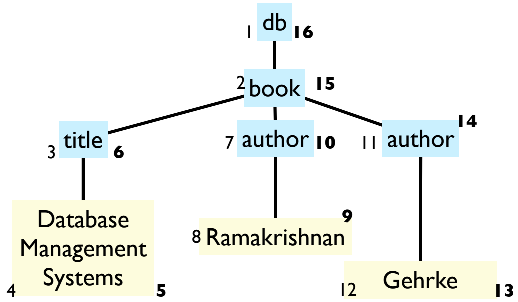
Begin/end numbering



Begin/end numbering



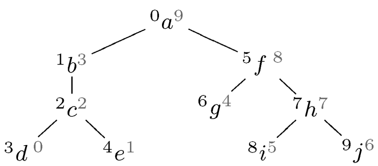
Begin/end numbering

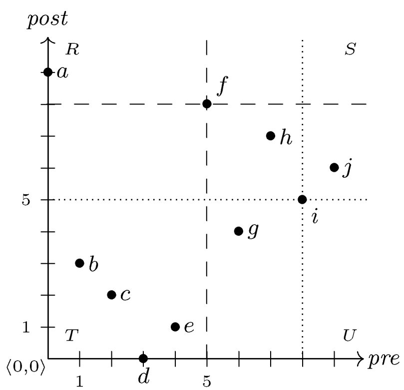


begin	end	par	tag	type
1	16		db	ELT
2	15	1	book	ELT
3	6	2	title	ELT
4	5	3		TEXT
7	10	2	author	ELT
8	9	7		TEXT
11	14	2	author	ELT
12	13	11		TEXT

Pre/post plane

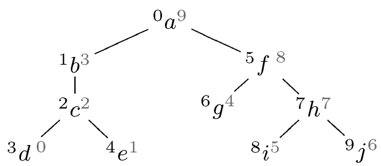
[Grust et al. 2004]

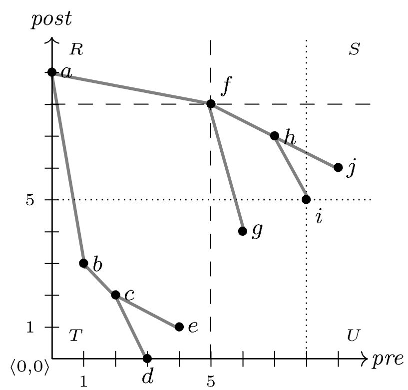




Pre/post plane

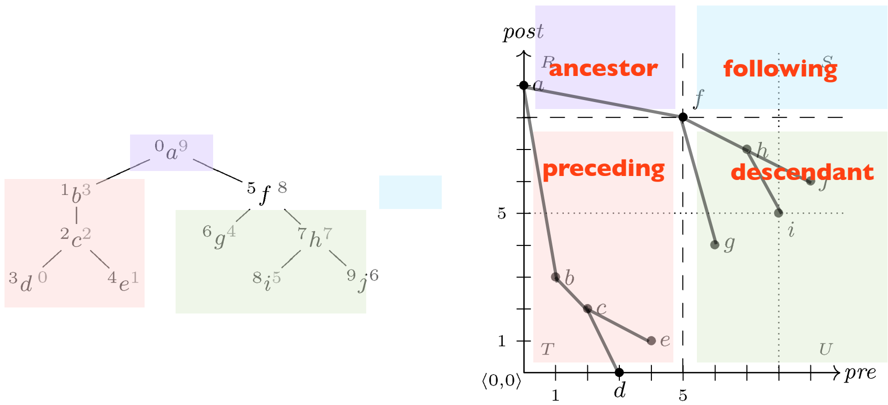
[Grust et al. 2004]



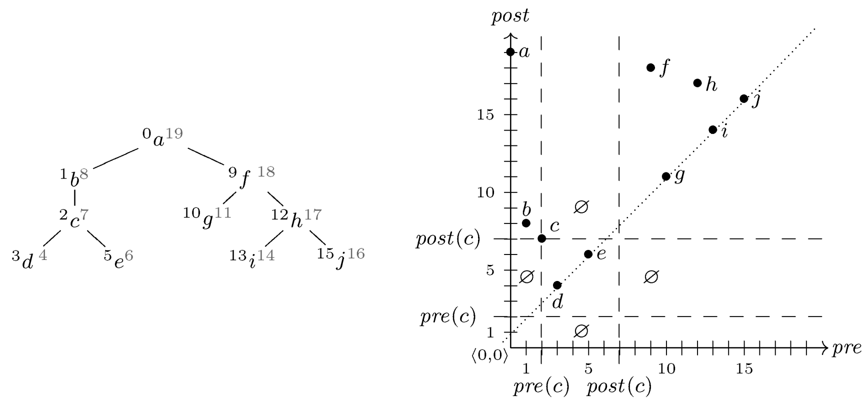


Pre/post plane

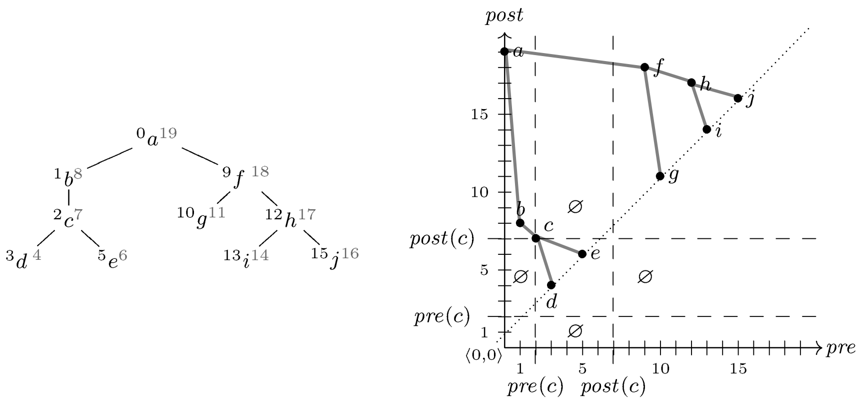
[Grust et al. 2004]



Begin/end plane



Begin/end plane



Why "Interval"?

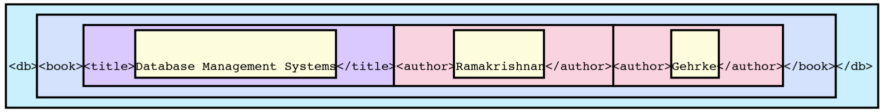
- Think of XML text as a linear string
- Begin and end are ~ positions of opening and closing tags

`<db><book><title>Database Management Systems</title><author>Ramakrishnan</author><author>Gehrke</author></book></db>`

- Each tag corresponds to an interval on line
- Interval inclusion = descendant

Why "Interval"?

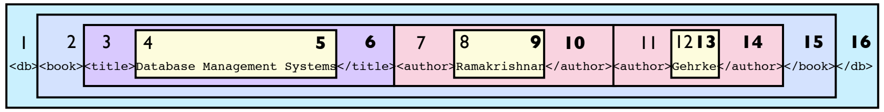
- Think of XML text as a linear string
- Begin and end are ~ positions of opening and closing tags



- Each tag corresponds to an interval on line
- Interval inclusion = descendant

Why "Interval"?

- Think of XML text as a linear string
- Begin and end are ~ positions of opening and closing tags



- Each tag corresponds to an interval on line
- Interval inclusion = descendant

Querying (begin/end)

- Child: use parent field
  - Child(p,q) $\Leftrightarrow$ $p.begin = q.par$
- Descendant: use interval inclusion
  - Descendant(p,q) $\Leftrightarrow$ $p.begin < q.begin$ and $q.end < p.end$
  - DescendantOrSelf(p,q) $\Leftrightarrow$ $p.begin \leq q.begin$ and $q.end \leq p.end$
- Ancestor, parent: just flip p,q, as before

Sibling, following (begin/end)

- Can define following as follows:
  - Following(p,q) $\Leftrightarrow$ $p.end < q.begin$
- Then following-sibling is just:
  - FollowingSibling(p,q) $\Leftrightarrow$ $p.end < q.begin$ and $p.par = q.par$

Example:

- No need for UDFs. Index on begin, end.
- How to solve //a//b[c]?

```
SELECT b.pre
FROM node a, node b
WHERE a.tag = 'a', b.tag = 'b'
  AND a.begin < b.begin
  AND b.end < a.end
  AND EXISTS(SELECT *
             FROM node c
             WHERE c.tag='c'
               AND c.par = b.begin
```


Example:

- No need for UDFs. Index on begin, end.
- How to solve //a//b[c]?

```
SELECT b.pre
FROM node a, node b
WHERE a.tag = 'a', b.tag = 'b'
  AND a.begin < b.begin
  AND b.end < a.end
  AND EXISTS(SELECT *
             FROM node c
             WHERE c.tag='c'
               AND c.par = b.begin
```

//a//b

Example:

- No need for UDFs. Index on begin, end.
- How to solve //a//b[c]?

```
SELECT b.pre
FROM node a, node b
WHERE a.tag = 'a', b.tag = 'b'
  AND a.begin < b.begin
  AND b.end < a.end
  AND EXISTS(SELECT *
             FROM node c
             WHERE c.tag='c'
               AND c.par = b.begin
```

//a//b

[c]

Node IDs and indexing: summary

- Goal: leverage existing RDBMS indexing
  - Dewey: string index, requires PREFIX, LEN UDFs
  - Interval: integer pre/post indexes, only requires arithmetic
- For both techniques: what about updates?
  - DDE: requires renumbering
    - but there are update-friendly variants
  - Interval encoding: can require re-indexing 50% of document

Next time

- XML publishing
    - Efficiently Publishing Relational Data as XML Documents
  - SilkRoute : a framework for publishing relational data in XML
  - Querying XML Views of Relational Data
- Reviews due Monday 4pm