Storing XML Data

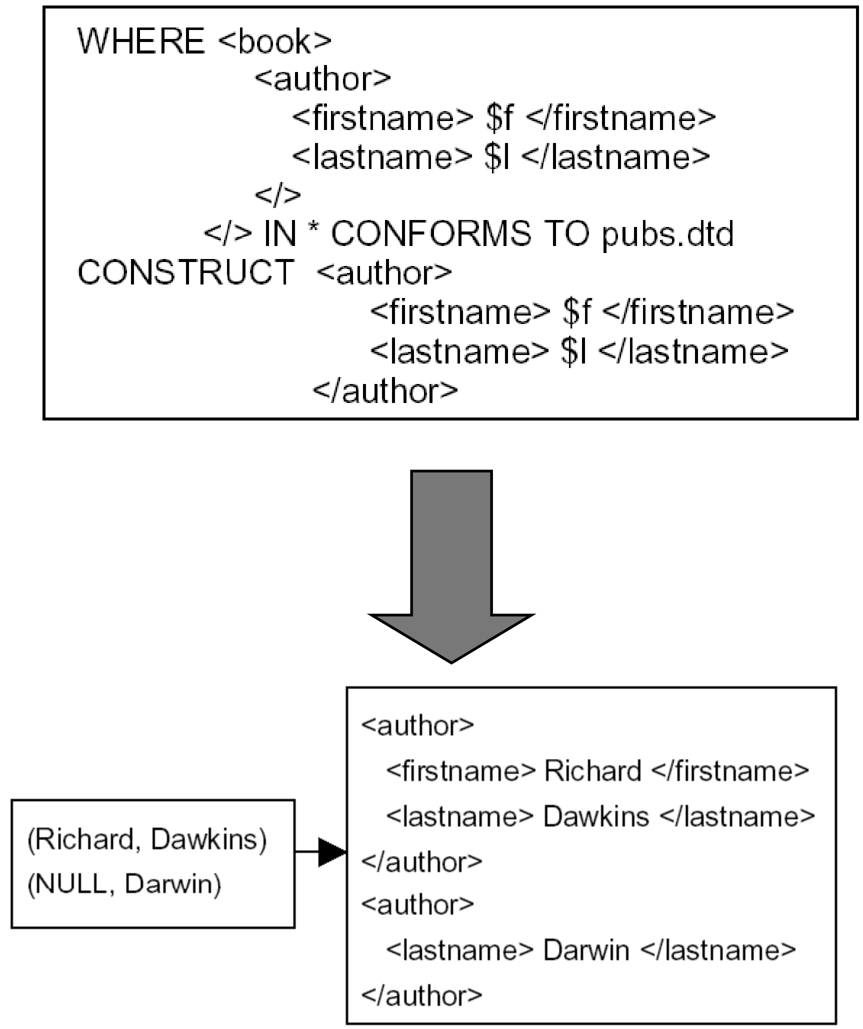
Vincent Ng
Department of Computing
Hong Kong Polytechnic University

XML to Relational

- Sophisticated query processing, storage and concurrency control techniques have been developed for relational DBMS
  - Thus, why not take advantage of available DBMS techniques?
- Store and query XML data using relational DBMS
  - Derive a relational schema from an XML DTD (schema)
  - Shred XML data into relational tuples – store XML data
  - Translate XML queries to SQL queries
- Convert query results back to XML

Converting Relational Results to XML

- Simple Structuring
  - Constructing such results from a RDBMS is natural and efficient
  - Requires attaching appropriate tags for each tuple

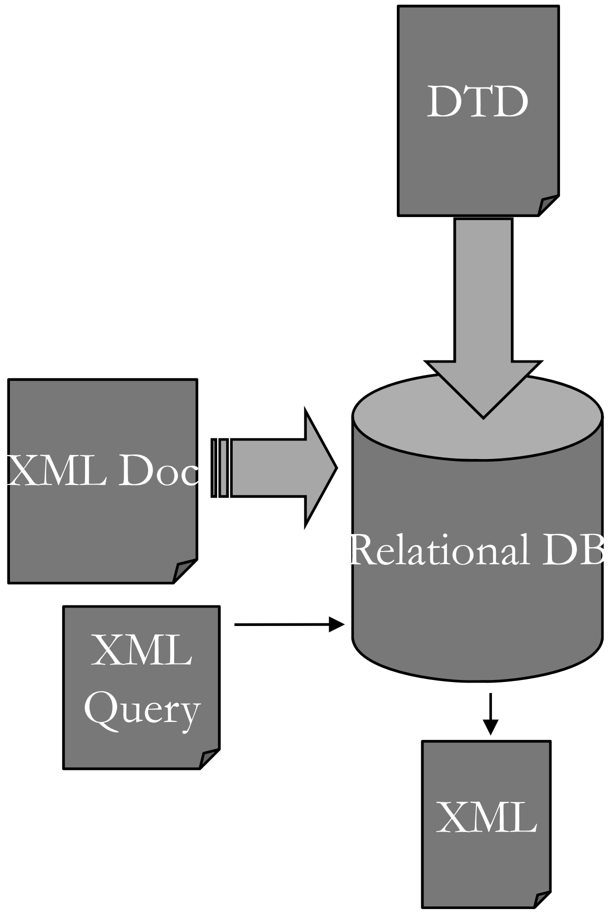


XML Data

Two Mapping Approaches

- Structure-mapping approach
  - static DTD
  - well-defined semantic constraints
- Model-mapping approach
  - dynamic DTD
  - ambiguous semantic constraints

Structure-mapping approach



- Design database schema based on DTD
- Different schemas for different DTDs
- Slides partially from a VLDB 2002 Tutorial

A Sample 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>
```

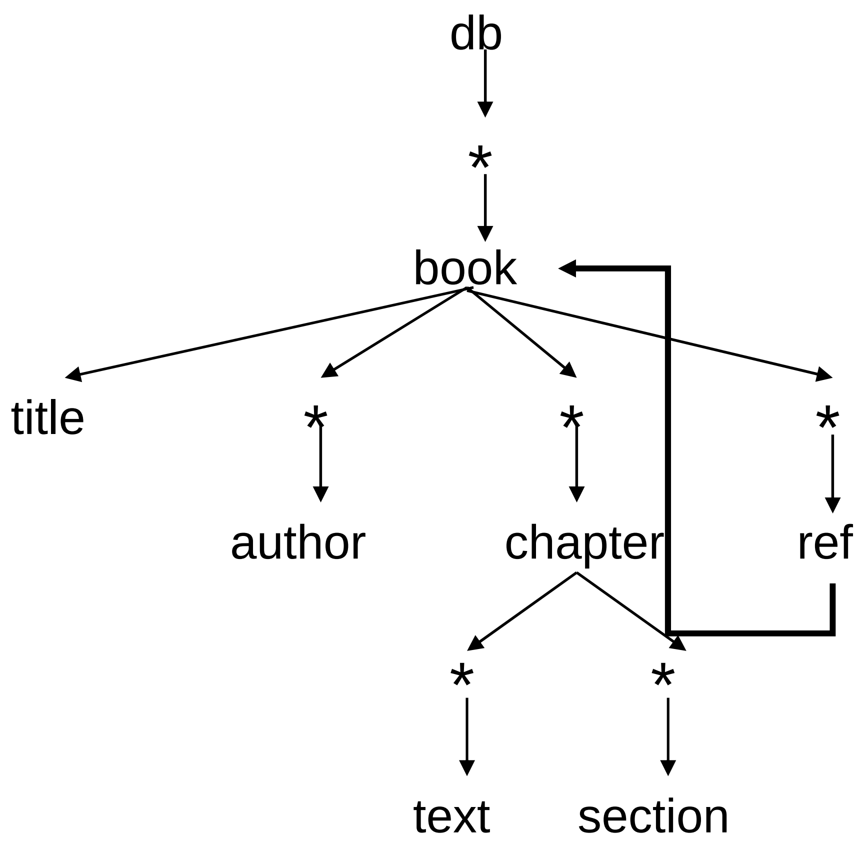
Recursive
Complex regular expressions

A DTD Graph

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

- Simplification: e.g., (text | section)*
  - text* | section* -- ignore order
  - XML document conforming to the DTD are those trees that unfold the graph
  - (special treatment: * empty at leaf)

A DTD Graph



One Possible Mapping

- 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(sid, did), sid, did: source and destination nodes; e.g., edge(02,03)
- Pros and cons
  - Lossless: the original document can be reconstructed; order preserving: one can add “order” information to the edge relation
  - Flexible schema: easy schema evolution but ignores regularity in structure; Supports heterogeneous elements with optional values
  - Querying: requires multi-table joins or self joins for element reconstruction; A simple query /db/book[author=“Bush”]/title requires 3 joins of the edge relation!

Basic Inlining

- 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 – no relation is created

db(dbID)
book(bookID,parentID,code,title:string)
author(authorID,bookID,author:string)
chapter(chapterID,bookID)
ref(refID,bookID)
text(textID,chapterID,text:string)
section(sectionID,chapterID,section:string)

Selective Mapping from XML to Relations

- Existing relational database R :
  - book(id,title)
  - ref(id1,id2)
- Select from XML and store in R
  - books with title containing “WebDB”, 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

XML2DB Mappings

XML2DB Mapping:

- Input: an XML document T of a DTD D, and an existing database schema R
- Output: a list of SQL inserts ΔR, updating the database of R

An extension of Attribute Grammars:

- treat the DTD D as an ECFG (extended context-free)
- associate semantic attributes and actions with each production of the grammar
  - attributes: passing data top-down
  - actions: generate SQL inserts ΔR
- Evaluation: generate SQL inserts in parallel with XML parsing

XML2DB Mappings

- DTD: normalized; element type definitions $e \to \alpha$

α ::= PCDATA | ε | e1, …, en | e1 + … + en | e*

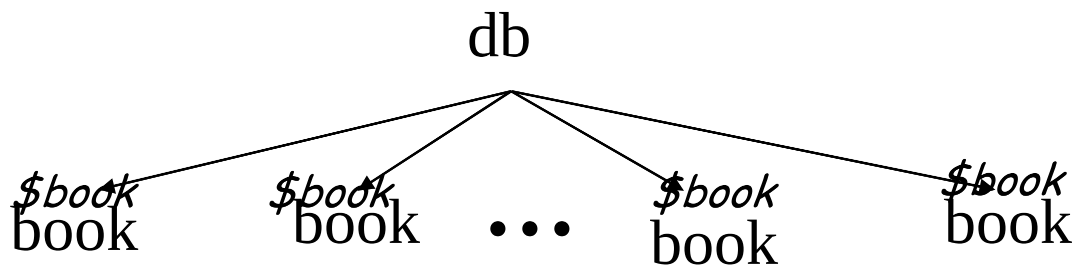
- Relation variables: for each relation schema Ri, define a variable ΔRi, which holds tuples to be inserted into Ri
- Attributes: $e associated with each element type e

  $e: tuple-valued, to pass data value top-down
- Rules: associated with each $e \to \alpha$; conditional statements
  - for each e' in α, define $e' using the parent attribute $e
  - Insert with relation variables: $\Delta Ri := \Delta Ri \cup \{tuple\}$

Example

- db $\rightarrow$ book*

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



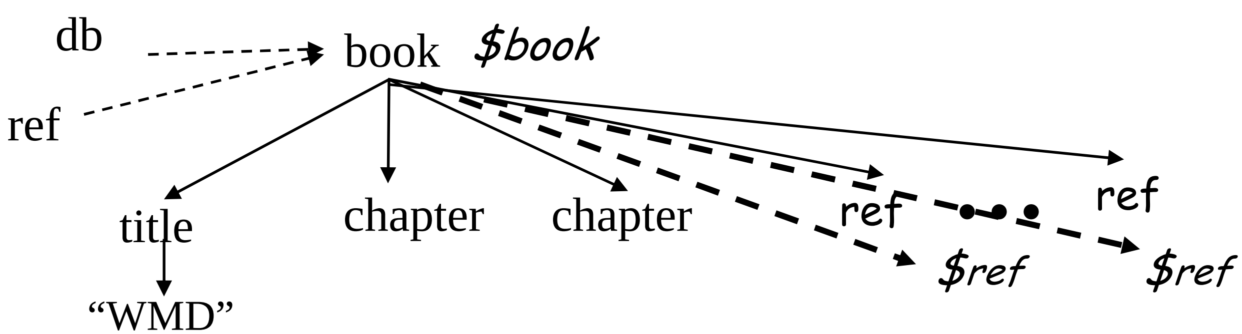
Semantic Actions

- book → title, author*, chapter*, ref*

```
if text(title) contains "WebDB" or ($book <> top and $book <> bot)
then    id := gen_id( );                              /* generate a new id */
         Δbook := Δbook ∪ { (id, text(title)) };      /* insert into book */
         if $book <> top                              /* cited by another book */
        then Δref := Δref ∪ { ($book, id) };          /* insert into ref */
        $ref := id;                                   /* passing information downward */
else  $ref := bot
```

recall relation schema: book (<u>id</u>, title), ref (id1, id2)

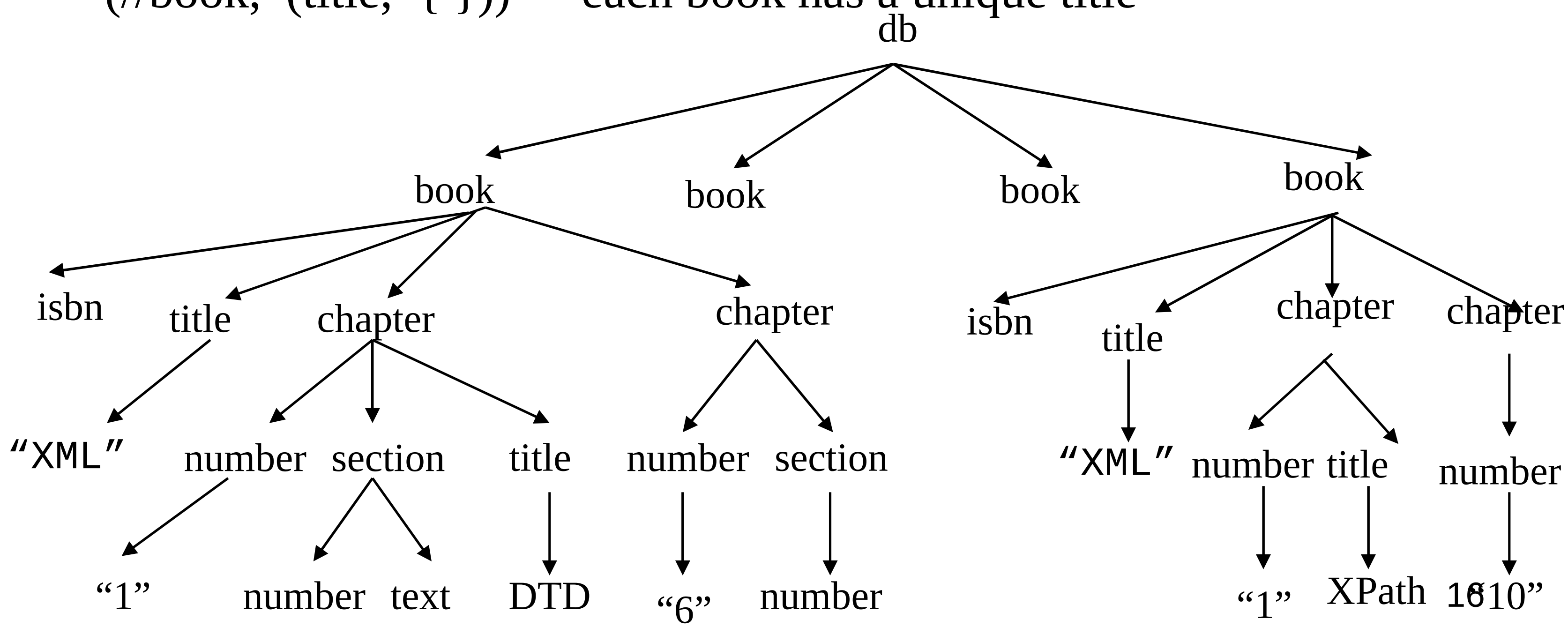
- gen_id( ): a function generating a fresh unique id
- conditional: either has "WebDB" or is referenced by a book of WebDB



XML Constraints

An XML schema consists of both types and constraints

- (//book, {isbn}) -- isbn is an (absolute) key of book
- (//book, (chapter, {number}) -- number is a key of chapter relative to book
- (//book, (title, { })) -- each book has a unique title



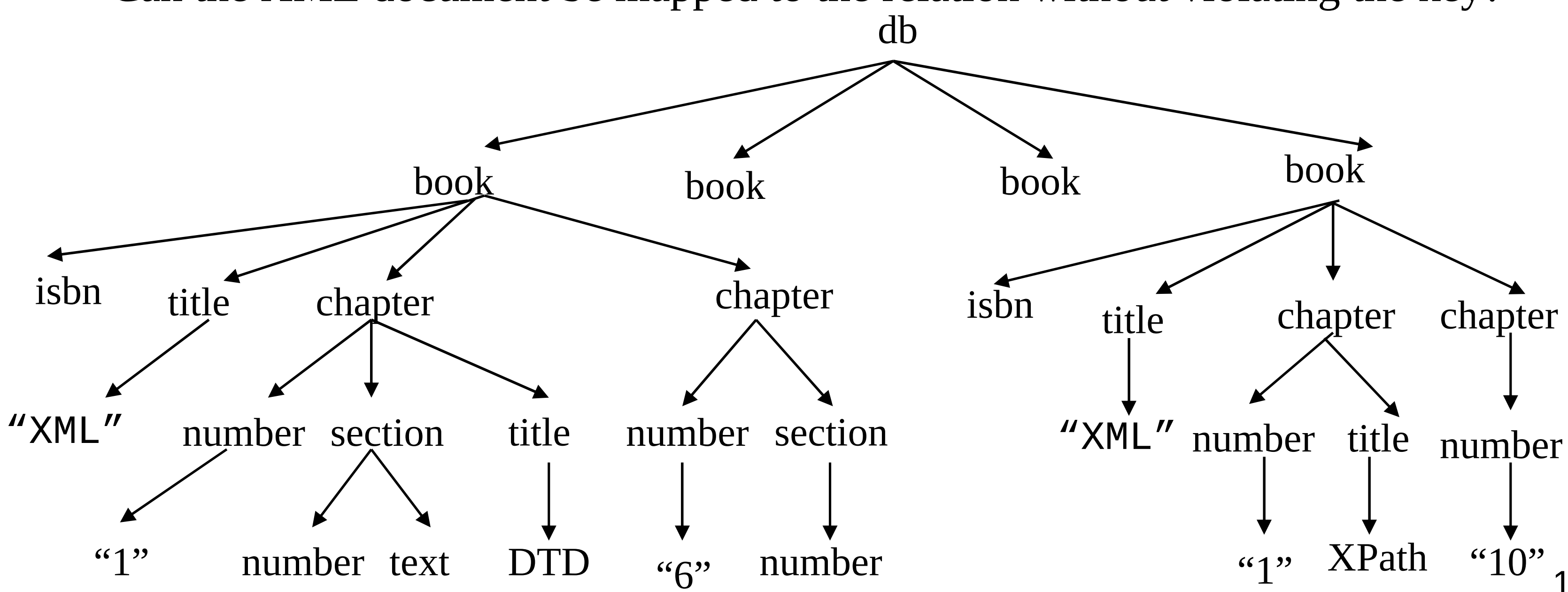
Mapping from XML to a Predefined Relation

One wants to store certain information from the XML document in:

chapter(<u>bookTitle, chapterNum</u>, chapterTitle)

- Mapping: for each book, extract its title, and the numbers and titles of all its chapters
- Predefined relational key: (bookTitle, chapterNum)

Can the XML document be mapped to the relation without violating the key?



A Safe Mapping

Now change the relational schema to

chapter(<u>isbn, chapterNum</u>, chapterTitle)

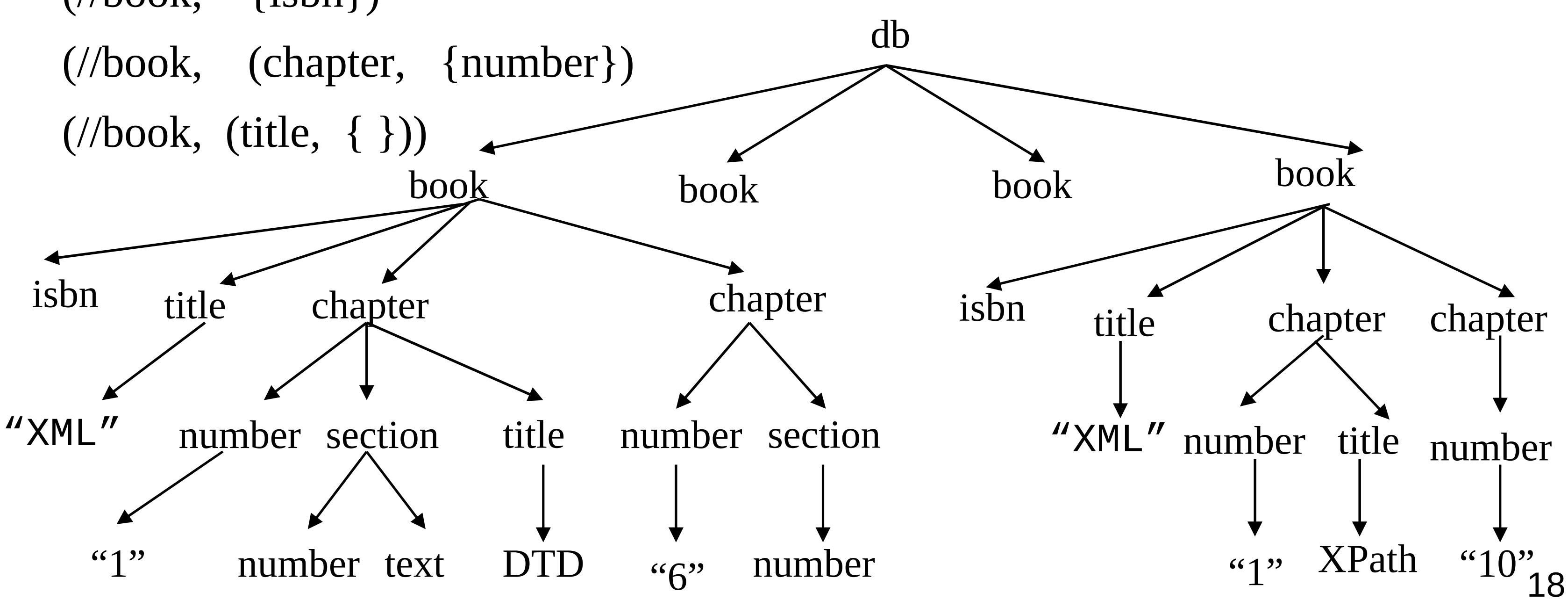
The relation can be populated without any violation. Why?

The relational key (<u>isbn, chapterNum</u>) for chapter is implied (entailed) by the keys on the original XML data:

(//book, {isbn})

(//book, (chapter, {number})

(//book, (title, { }))



Why do we care about constraints?

- Constraints are a fundamental part of the semantics of the data – mapping from XML to relations should not lose the information
- Relational constraints are important for query optimization, data cleaning, and consistency/integrity maintenance, . . .
- Constraints help us determine whether a relational schema for storing XML data makes sense or not

Problem statement: Constraint Propagation

- Input: a set K of XML keys, a predefined relational schema S, a mapping f from XML to S, and a functional dependency FD over S
- Output: is FD *implied* by K via f? I.e., does FD hold over f(T) for any XML document T that satisfies K?

Note: XML schema/DTD is not required – K is the only semantics

Model-mapping approach

- Fixed database schema for different XML documents without the support of DTD
  - Edge (Florescu and Kossmann, 1999)
  - XRel (YoshiKawa, M. and Amagasa, T., 2001)
  - XParent (H. Jiang et al., 2002)
  - iNode (Lau & Ng, 2002)
- Support well-formed but non-DTD XML documents

Edge

- One-table schema
  - **Edge(Source, Ordinal, Target, Label, Flag, Value)**
- Each row corresponds to one edge in data graph
- Require table joins for edge connections
- Cannot determine the parent-child relationship directly

A Sample File

```
<SigmodRecord>
        <issue>
        <volume>11</volume>
        <number>1</number>
        <articles>
                <article>
                <title>Annotated Bibliography on Data Design.</title>
                <initPage>45</initPage>
                <endPage>77</endPage>
                <authors>
                        <author position="00">Anthony I. Wasserman</author>
                        <author position="01">Karen Botnich</author>
                </authors>
                </article>
        </articles>
</issue>
</SigmodRecord>
```

Edge

Src	Ord	Tgt	Label	Flag	Value
0	1	1	SigmodRecord	ref	
1	1	2	issue	ref	
2	1	3	volume	val	11
2	2	4	number	val	1
2	3	5	articles	ref	
5	1	6	article	ref	
6	1	7	title	val	Annotated ….
6	2	8	initPage	val	45
6	3	9	endPage	val	77
6	4	10	authors	ref	
10	1	11	author	val	Anthony I …
11	1	12	@position	val	00
10	2	13	author	val	Karen Botnich
13	1	14	@position	val	01

XParent

- XParent
  - uses LabelPath and DataPath tables to maintain the structural information of the XML documents.
  - The DataPath table is also used to keep the parent-child relationship instead of using region as in XRel.

XParent

- Four-table schema
  - **LabelPath(PathID, Len, Path)**
  - **DataPath(Pid, Cid)**
  - **Element(PathID, Ordinal, Did)**
  - **Data(PathID, Did, Ordinal, Value)**
- Determine the parent-child relationship by using equijoin

XParent

LabelPath Table

PID	PathLen	PathExp
1	1	./SigmodRecord
2	2	./SigmodRecord./issue
3	3	./SigmodRecord./issue./volume
4	3	./SigmodRecord./issue./number
5	3	./SigmodRecord./issue./articles
6	4	./SigmodRecord./issue./articles./article
7	5	./SigmodRecord./issue./articles./article./title
8	5	./SigmodRecord./issue./articles./article./initPage
9	5	./SigmodRecord./issue./articles./article./endPage
10	5	./SigmodRecord./issue./articles./article./authors
11	6	./SigmodRecord./issue./articles./article./authors./author
12	7	./SigmodRecord./issue./articles./article./authors./author./@position

DataPath Table

PID	CID
1	2
2	3
2	4
2	5
5	6
~~5~~	~~15~~
6	7
6	8
6	9
6	10
10	11
10	12
~~10~~	~~13~~
~~10~~	~~14~~

XParent

PathID	Ordinal	Did
1	1	1
2	1	2
3	1	3
4	1	4
5	1	5
6	1	6
7	1	7
8	1	8
9	1	9
10	1	10
11	1	11
12	1	12
11	2	13
12	~~1~~	14

Element Table

PathID	Did	Ordinal	Value
3	3	1	11
4	4	1	1
7	7	1	Annotated Bibliography on Data Design.
8	8	1	45
9	9	1	77
12	12	1	00
11	11	1	Anthony I. Wasserman
12	14	~~1~~	01
11	13	2	Karen Botnich

Data Table

INode

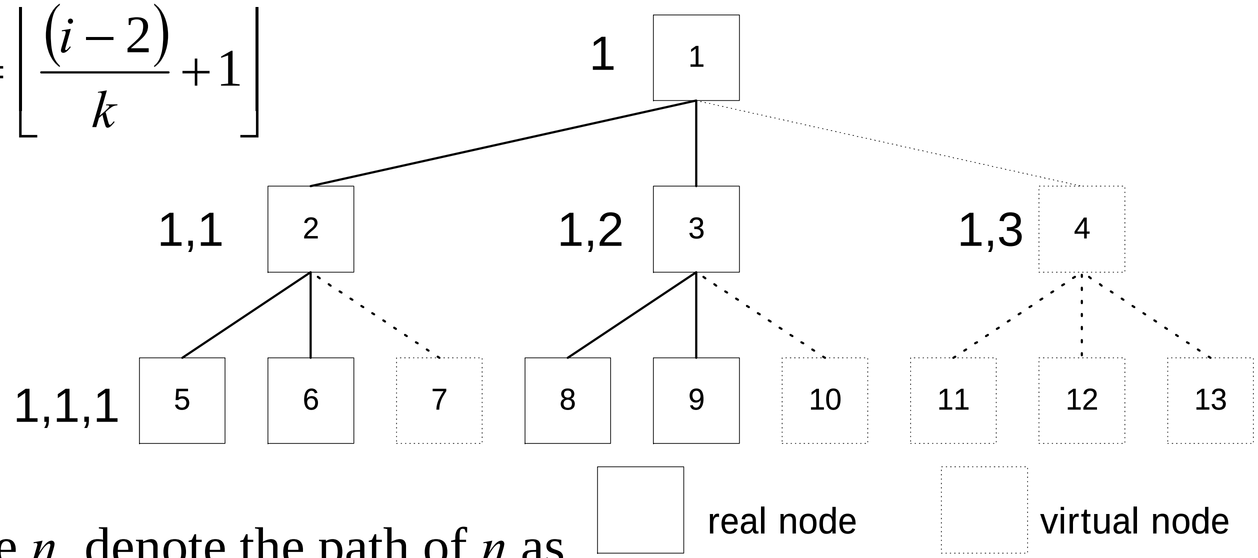
- Model-mapping approach
- Node-oriented approach
- Based on a node numbering scheme
- Using the virtual node concept to calculate the id for each node (Lee et al., 1996)

Key Features

1. Does not need to concatenate the edges to form a simple path for query processing
2. Reduces the database size by using a table to store all simple path expressions
3. The parent-child relationship is embedded in the NodeID
4. Retrieve the parent-child relationship by calculation instead of using equijoins or θ–joins

INode Numbering Scheme (1/4)

$$Parent(i) = \left\lfloor \frac{(i-2)}{k} + 1 \right\rfloor$$



- Given a node n, denote the path of n as

  $a_1, a_2, a_3, \ldots, a_k$

  where k is the depth of node n

  and $1 \leq a_i \leq n_c$

INode Numbering Scheme (2/4)

- Based on the Parent(i) function, we have a path function as follow:

$$f(path) = \begin{cases} \sum_{i=1}^{2} a_i & , k \leq 2 \\ n_c^{k-2} \sum_{i=1}^{2} a_i - n_c^{k-2} + \sum_{i=0}^{k-3} n_c^i a_{k-i} + 1 & , k > 2 \end{cases}$$

Where n_c = maximum number of child nodes for a node

k = path length of the node

a_i = value in the path at the position i

INode Numbering Scheme (3/4)

- With the path function, we embed the document-id in the node-id with the following function, where *i* is the number of decimal places for document-id

$$nodeid = f(path) + \frac{docid}{10^i}$$

INode Numbering Scheme (4/4)

- Given the node-ids of *i* and *j*, where *i* is the ancestor of *j*, and the depth difference between two nodes as *d*, we can confirm the ancestor-descendent relationship with the following function

$$nodeid_i = \left\lfloor \frac{nodeid_j - 2 - \sum_{i=1}^{d-1} n_c^i}{n_c^d} + 1 \right\rfloor + nodeid_j - \lfloor nodeid_j \rfloor$$

Table Mapping

- Three-table schema
  - **Path(PathID, PathExp)**
  - **Element(NodeID, PathID, Ordinal, Value)**
  - **Attribute(NodeID, PathID, Value)**
- Determine the parent-child relationship by calculation

Querying - Edge

Q1: /SigmodRecord/issue/articles/article[endPage=77]/authors/author

```
select authors.value
from edge sigmodrecord, edge issue, edge articles, edge article,
        edge authors, edge author, edge endpage
where sigmodreacord.label = ‘SigmodRecord’
        and issue.label = ‘issue’
        and articles.label = ‘articles’
        and article.label = ‘article’
        and authors.label = ‘authors’
        and author.label = ‘author’
        and endpage.label = ‘endPage’
        and sigmodrecord.source = ‘0’
        and sigmodrecord.target = issue.source
        and issue.target = articles.source
        and articles.target = article.source
        and article.target = authors.source
        and endpages.source = authors.source
        and author.source = authors.target
        and endpage.value = ‘77’
```

Querying - XParent

Q1: /SigmodRecord/issue/articles/article[endPage=77]/authors/author

```
select d1.value
from labelpath lp1, labelpath lp2, datapath dp1, datapath dp2, data d1, data d2
        labelpath lp3, datapath dp3
where lp1.path = ‘./SigmodRecord./issue./articles./article./authors/author'
        and lp2.path = ‘./SigmodRecord./issue./articles./article./endPage’
        lp3.path = ‘./SigmodRecord./issue./articles./article./authors'
        and d1.pathid = lp1.pathid
        and d2.pathid = lp2.pathid
        and d1.did = dp1.cid
        and d2.did = dp2.cid
        and dp3.pid = dp2.pid
        and dp1.pid = dp3.cid
        and d2.value = ‘77’
```

Query Processing - iNode

Q1: /SigmodRecord/issue/articles/article[endPage=77]/authors

```
select e1.value
from path p1, path p2, element e1, element e2
where p1.pathexp = '#/SigmodRecord#/issue#/articles#/article#/authors'
      and p2.pathexp = '#/SigmodRecord#/issue#/articles#/article#/endPage'
      and e1.pathid = p1.pathid
      and e2.pathid = p2.pathid
      and mod(e1.nodeid, 1) = mod(e2.nodeid, 1)
      and floor(((e1.nodeid – 2) / 5) + 1) = floor(((e2.nodeid - 2) / 5) + 1)
      and e2.value = '77'
```

Single Table To XML

Students Database

Student

StudId	*Name*
007	*James*
131	*Susan*
555	*Susan*

$\Rightarrow$

```
<Students>
        < Student >
                <StudId>007</StudId>
                <Name>James</Name>
        </Student >
        <Student>
                <StudId>131</StudId>
                <Name>Susan</Name>
        </Student>
        <Student>
                <StudId>555</StudId>
                <Name>Susan</Name>
        </Student>
</Students>
```

Converting XML Queries to SQL

- Converting queries with simple path expressions
  - Relation(s) corresponding to start of root path expression(s) are identified and added to from clause of SQL query
  - If necessary, path expressions are translated to joins among relations

```
WHERE <book>
            <booktitle> The Selfish Gene </booktitle>
            <author>
                <name>
                    <firstname> $f </firstname>
                    <lastname> $l </lastname>
                </name>
            </author>
        </book> IN * CONFORMING TO pubs.dtd
CONSTRUCT <result> $f $l </result>
```

```
Select Y.name.firstname,
       Y.name.lastname
From   book X, X.author Y
Where X.booktitle = “Databases”
```

```
Select A.”author.name.firstname”,
       A.”author.name.lastname”
From   author A, book B
Where B.bookID = A.parentID
       AND A.parentCODE = 0
       AND B.”book.booktitle” = “The Selfish Gene”
```