

NPRG036

XML Technologies



Lectures 11 and 12

XML Databases

4. and 11. 5. 2020

Author: **Irena Holubová**

Lecturer: **Martin Svoboda**

<http://www.ksi.mff.cuni.cz/~svoboda/courses/192-NPRG036/>

Lecture Outline

- **XML persistence**
 - Introduction
 - XML databases
 - Numbering schemes
 - Mapping techniques
-

Why XML Database?

- Motivation: requirements of applications
 - Processing of external data
 - Web pages, other textual data, structured data
 - E-commerce
 - Lists of goods, personalized views of the lists, orders, invoices, ...
 - Integration of heterogeneous information resources
 - Integrated processing of data from Web pages and from relational databases
 - Main reason: storing XML data into databases means management of huge volumes of XML data in an efficient way
-

Documents vs. Databases

World of documents

- many small documents
- usually static
- implicit structure
 - tagging
- suitable for humans

World of databases

- several huge databases
 - usually dynamic
 - explicit structure
 - schema
 - suitable for machines
-

Documents vs. Databases

Documents

- editing
- printing
- lexical checking
- word count
- information retrieval
- searching

Databases

- updating
 - data cleaning
 - querying
 - storing/transforming
-

Documents and Structured Data

- The border between the world of documents and world of databases is not exact
 - In some proposals both kinds of access are possible
 - Somewhere in the middle we can find formatting languages and semi-structured data
 - **Semi-structured data** are defined as data which are not sorted (have arbitrary order), which are not complete (have optional parts) and whose structure can "unpredictably" change
 - Web data, HTML pages, Bibtex files, biological and chemical data
 - XML data are a kind of semi-structured data
-

Classification of XML Documents

- The basic classification of XML documents results from their origin and the way they were created
 - data-oriented
 - document-oriented
 - hybrid
 - For the particular classes different ways of implementations are suitable
-

Data-oriented XML Documents

- Usually created and processed by machines
 - Regular, deep structure
 - Fully structured data
 - They do not contain
 - Mixed-content elements
 - CDATA sections
 - Comments
 - Processing instructions
 - The order of sibling elements is often unimportant
 - Example: database exports, catalogues, ...
-

Data-oriented XML Documents

```
<book id="12345">
  <title>All I Really Need To Know I Learned in
Kindergarten</title>
  <author>
    <name>Robert</name>
    <surname>Fulghum</surname>
  </author>
  <edition title="Argo">
    <year>2003</year>
    <ISBN>80-7203-538-X</ISBN>
  </edition>
  <edition title="Argo">
    <year>1996</year>
    <ISBN>80-7203-028-0</ISBN>
  </edition>
</book>
```

Document-oriented XML Documents

- ❑ Usually created and processed by humans
 - ❑ Irregular, less structured
 - Semi-structured data
 - ❑ Often contain
 - Mixed-content elements
 - CDATA sections
 - Comments
 - Processing instructions
 - ❑ The order of sibling elements is crucial
 - ❑ Example: XHTML web pages
-

Document-oriented XML Documents

```
<book id="12345">
  <title>All I Really Need To Know I Learned in
  Kindergarten</title>
  <author>Robert Fulghum</author>
  <description>A new, edited and extended publication
  published on the occasion of the fifteen anniversary of
  the first edition</description>
  <Text>
    <p>Fifteen years after publishing of <q>his</q>
    <i>Kindergarten</i> Robert Fulghum has decided to read it
    once again, now in <i>2003</i>.</p>
    <p>He wanted to find out whether and, if so, to what
    extent his opinions have changed and why. Finally, he
    modified and extended his book to...</p>
  <Text>
</book>
```

Implementation Approaches

- Differ according to the type of documents
 - Exploit typical features
 - Problem: hybrid documents
 - Ambiguous classification
 - Document-oriented techniques
- vs.
- Data-oriented techniques
-

Document-oriented Techniques (1)

- We need to preserve the document as whole
 - Order of sibling elements
 - Comments, CDATA sections, ...
 - Even whitespaces
 - For legal documents
 - **Round tripping** – storing a document into a database and its retrieval
 - The level of round tripping says to what extent the documents are similar
 - The higher level, the higher similarity
 - In the optimal case they are equivalent
-

Document-oriented Techniques (2)

□ LOB

- Storing of the whole document into a BLOB / CLOB column

- Possible in all known database systems

(+) The highest level of round tripping, fast retrieval of the whole document, extending of XML data with database features

(−) No XML operations

- The data need to be extracted from the DB and pre-processed

□ XML data type

- Like a LOB with the support for XML operations

- XML querying, XML full-text search

- Requires special indices (**numbering schemas**)

- SQL/XML
-

Document-oriented Techniques (3)

- Native XML databases (NXD)
 - Natural support for XML operations
 - XML query languages, XML update operations, DOM/SAX interfaces, ...
 - Focus on document-oriented aspects
 - Comments, CDATA sections, ...
 - The logical model is based on XML
 - i.e. we work with trees
 - The physical model can be, e.g., relational
 - i.e. we can physically store the trees, e.g., into relations
 - (+) Good level of round tripping
 - (−) The index (numbering schema) is (used to be) several times bigger than the data, necessity to start from scratch (transactions, replication, multi-user access, query optimization, ...)
-

Data-oriented Techniques (1)

- Idea: The data are stored in a relational database management system (RDBMS)
 - **Mapping method** – transforms the data into relations (and back)
 - XML queries over XML data → SQL queries over relations
 - The result of SQL query → XML document
 - Exploit data-oriented aspects (low level of round tripping)
 - It is not necessary to preserve the document as a whole
 - Order of sibling elements is ignored, document-oriented constructs (comments, whitespaces, ...) are ignored, ...
 - No (little) support for mixed-content elements
-

Data-oriented Techniques (2)

- **Middleware**
 - A separate software which ensures transformation of XML data between XML documents and relations
 - **XML-enabled database**
 - RDBMS with functions and extensions for XML data support
 - **Special related approach: XML data binding**
 - Methods for binding of XML data and objects
 - For each element type a separate class
 - Its attributes and subelements form properties of the class
 - I.e. it is not a DOM tree of objects!
-

Numbering Schemas

A **numbering schema** of a tree model of a document is a function which assigns each node a unique identifier that serves as a reference to that node for indexing and query evaluation

- Enable fast evaluation of selected relationships among nodes of XML document
 - Ancestor-descendant
 - Parent-child
 - Element-attribute
 - ...
 - Depth of the node
 - Order among siblings
 - ...
-

Numbering Schemas

- Sequential numbering schema
 - The identifiers are assigned to the nodes as soon as they are added to the system sequentially, starting from 1

 - Structural numbering schema
 - Enables to preserve and evaluate a selected relationship among any two nodes of the document
 - Often it is expected to enable fast searching for all occurrences of such a relationship in the document
-

Numbering Schemas

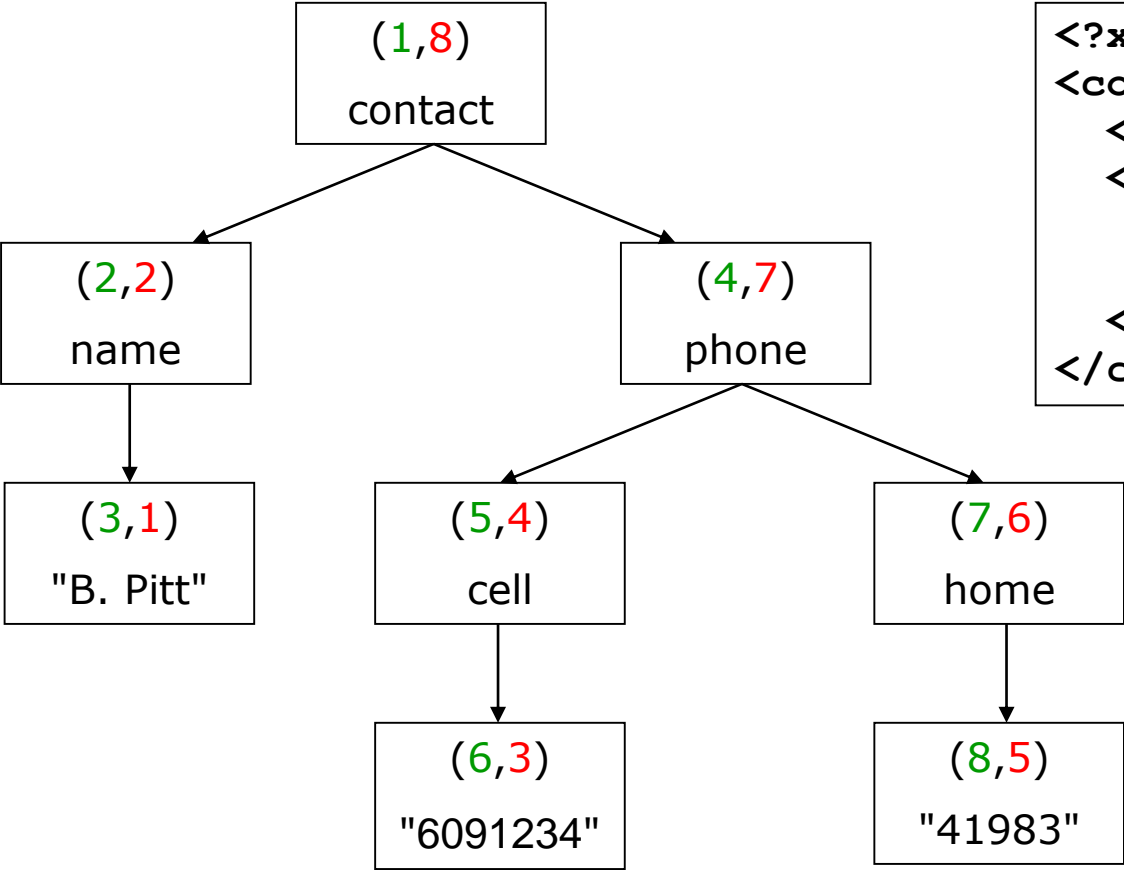
□ Stable numbering schema

- A schema which does not have to be modified (except for preserving its local features) when the structure of the respective data changes
 - i.e., on insertion/deletion of nodes

□ A schema of a structural numbering schema

- Is an ordered pair (p, L) , where p is a binary predicate and L is an invertible function which for the given XML tree model $T = (N, E)$ assigns each node $v \in N$ a binary sequence $L(v)$.
 - For each pair of nodes $u, v \in N$ predicate $p(L(u), L(v))$ is satisfied if v is in a particular relationship with u .
 - e.g. v is a descendant of u
 - Particular numbering schema: particular p and L
-

Dietz Numbering

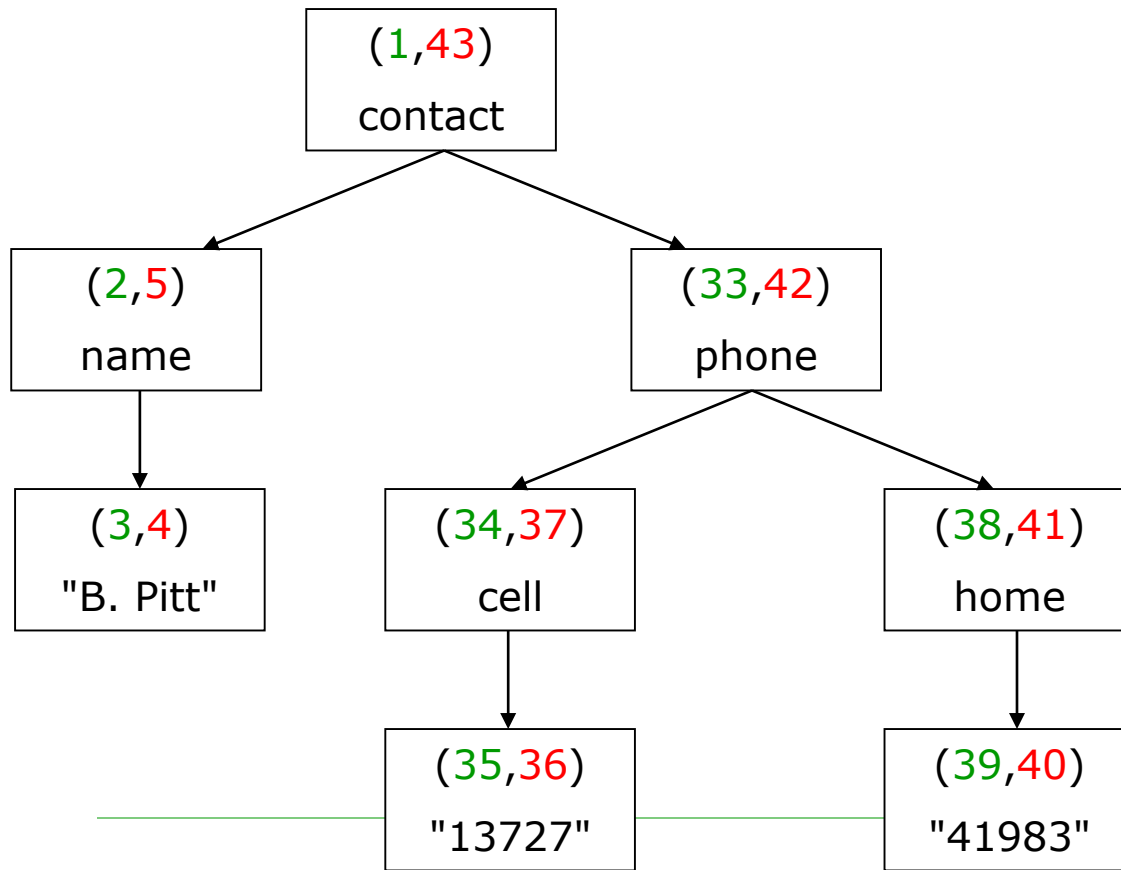


```
<?xml version="1.0"?>
<contact>
  <name>B. Pitt</name>
  <phone>
    <cell>6091234</cell>
    <home>41983</home>
  </phone>
</contact>
```

Dietz Numbering

- Preorder traversal
 - Child nodes of a node follow their parent node
 - Postorder traversal
 - Parent node follows its child nodes
 - Construction of a numbering schema
 - Each node $v \in N$ is assigned with a pair (x,y) denoting preorder and postorder order
 - Node $v \in N$ having $L(v) = (x,y)$ is a descendant node of node u having $L(u) = (x',y')$ if $x' < x$ & $y' > y$
-

Depth-first (DF) Numbering



preorder traversal +

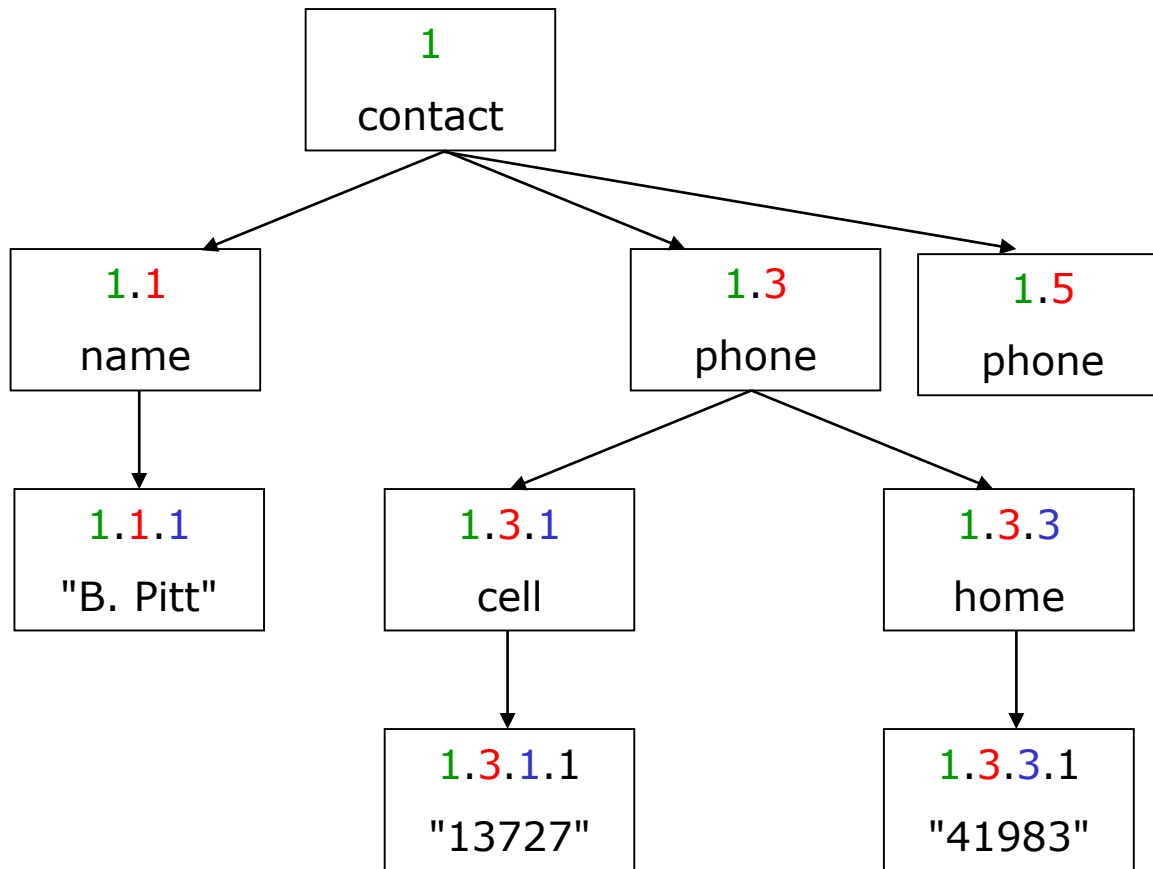
■ assigning (u_{\min}, u_{\max}) ,
where

■ u_{\min} is the time of
visiting a node

■ u_{\max} is the time of
leaving a node

■ Predicate is the same
as in the previous case

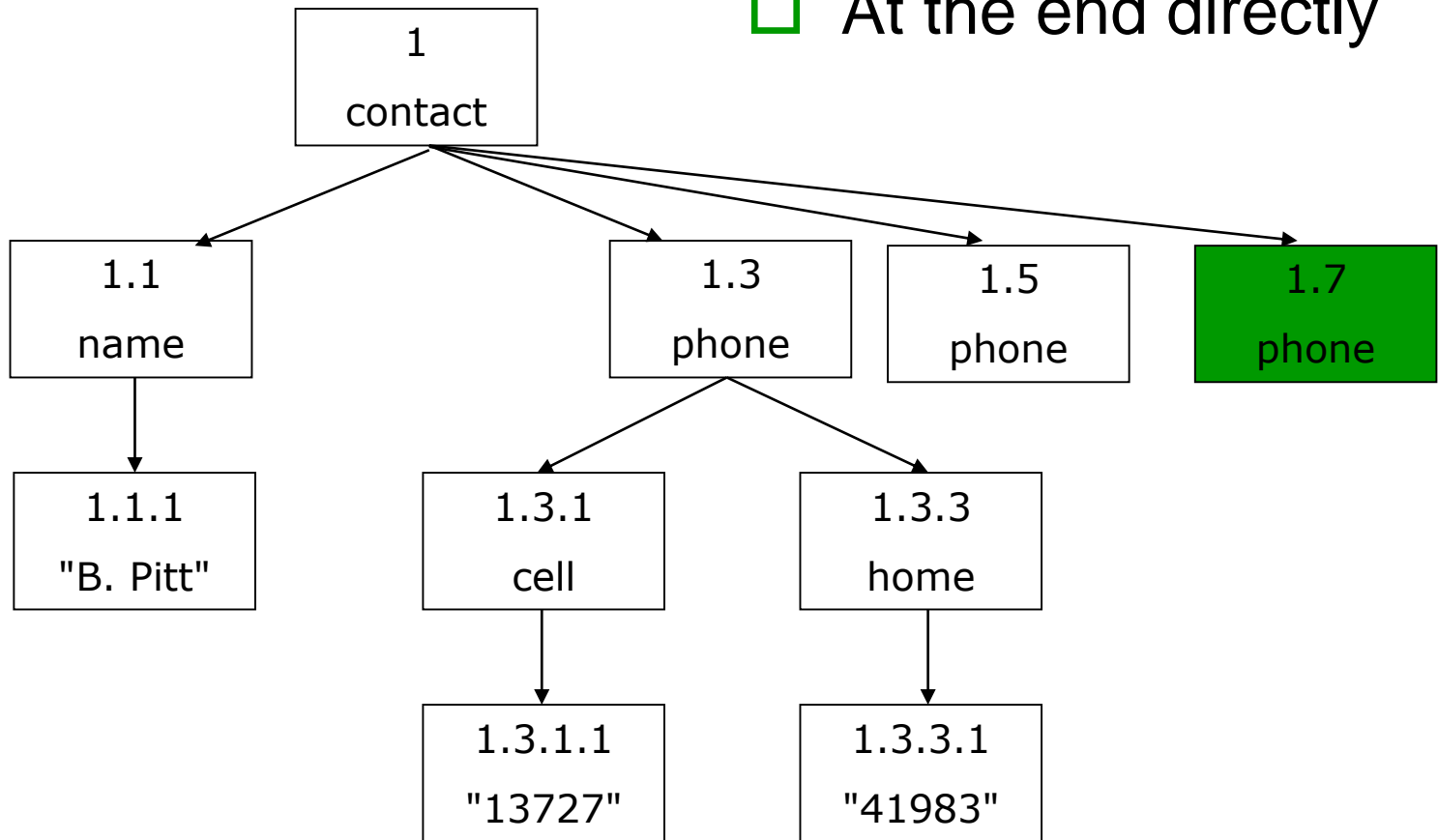
ORDPATH



- New level of tree = new level of numbering
- We use only odd numbers
- The predicate corresponds to searching a substring

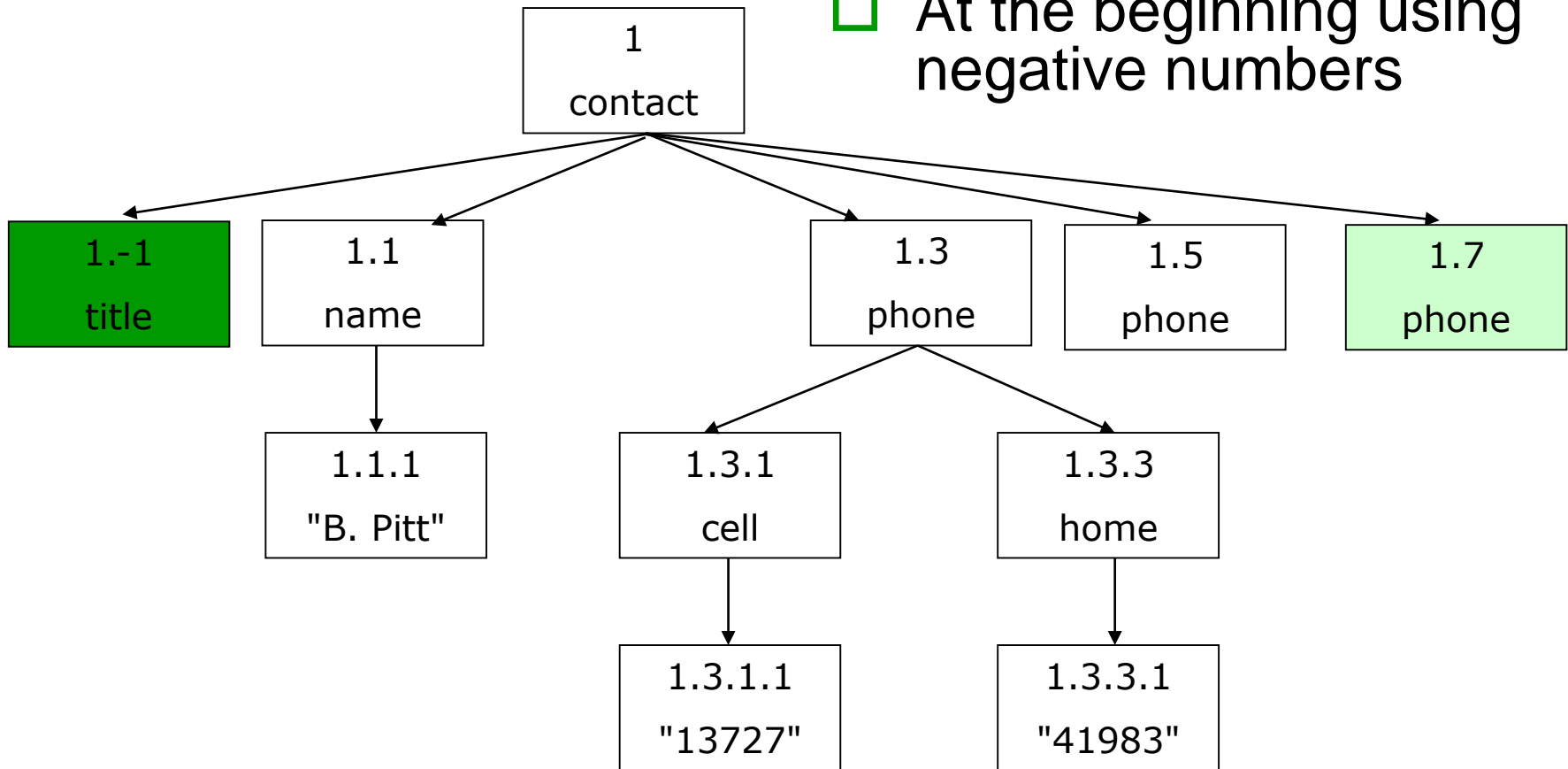
ORDPATH – Insert

- At the end directly



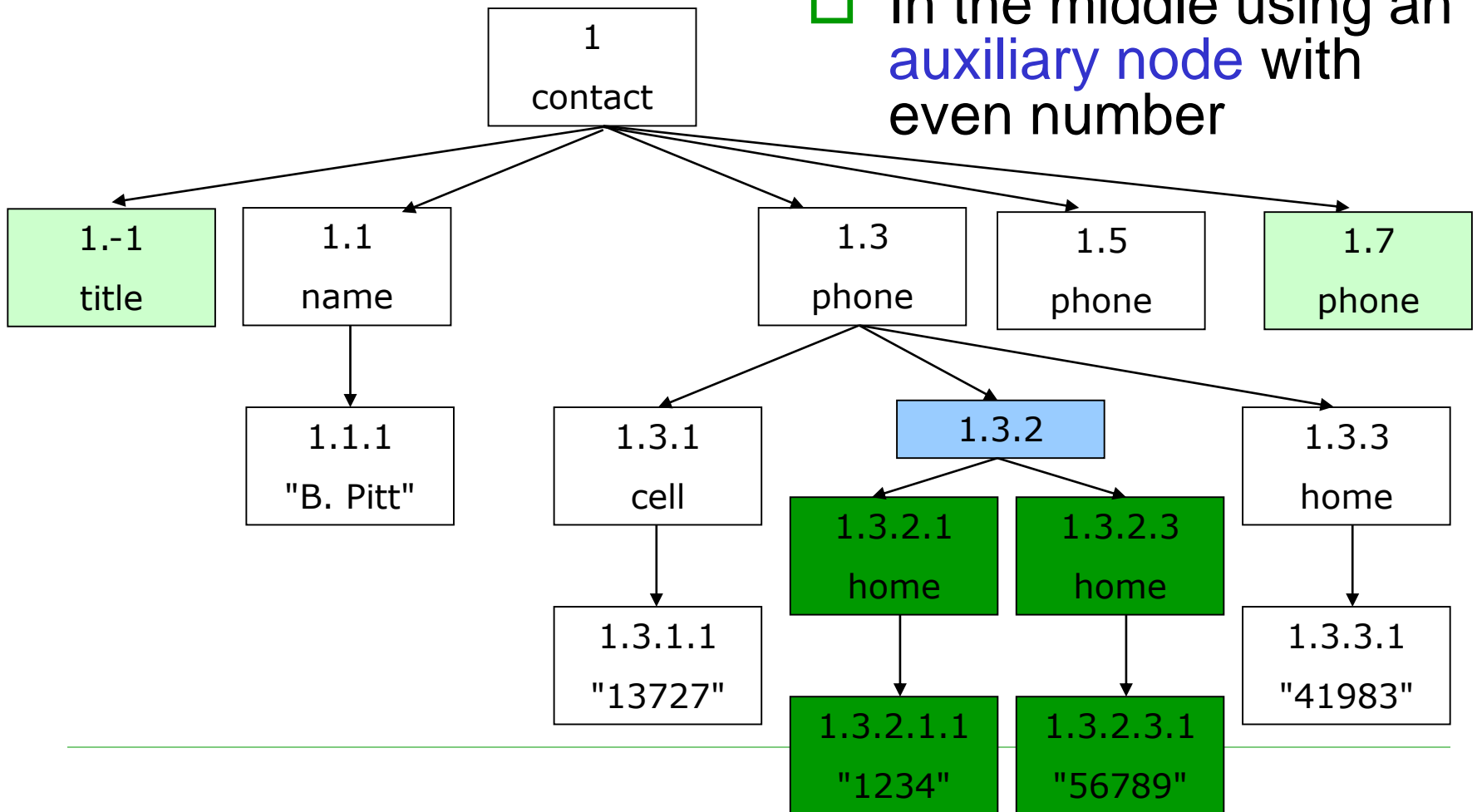
ORDPATH – Insert

- At the beginning using negative numbers



ORDPATH – Insert

- In the middle using an **auxiliary node** with even number



XML Databases

- What we want: persistent storage of XML data
 - General classification:
 - Based on a file system
 - Based on an object model
 - Based on (object-)relational databases
 - XML-enabled databases
 - Exploit a mapping method between XML data and relations
 - Native XML databases
 - Exploit a suitable data structure for hierarchical tree data
 - Usually a set of numbering schemas
-

XML Databases

- The most efficient approaches are the native ones
 - Reason: From the beginning they target the XML data structure
 - They are based on it
 - Disadvantage: We need to start from scratch
 - The databases are not only about storing the data, but also transactions, versioning, multi-user access, replication, ...
 - An alternative intuitive idea: Exploitation of a mature and verified technology of (object-) relational databases
-

Mapping Methods

- Methods for transformation between XML data and relations
 - Further classification:
 - A. **Generic** – mapping regardless XML schema of the stored XML data
 - B. **Schema-driven** – mapping based on XML schema of the stored XML data
 - DTD, XML Schema
 - C. **User-defined** – mapping provided by the user
-

A. Generic Methods

- Do not exploit XML schema of the stored data
 - Idea: Not all data have a schema
 - Approaches:
 1. A relational schema for a particular type of (collection of) XML data
 - e.g. Table-based mapping
 2. A general relational schema for any type of (collection of) XML data
 - View XML data as a general tree
 - We store the tree
 - e.g. Generic-tree mapping, Structure-centred mapping, Simple-path mapping
-

Table-based Mapping (1)

```
<Tables>
  <Table_1>
    <Row>
      <Column_1>...</Column_1>
      ...
      <Column_n>...</Column_n>
    </Row>
    ...
  </Table_1>
  ...
  <Table_n>
    <Row>
      <Column_1>...</Column_1>
      ...
      <Column_m>...</Column_m>
    </Row>
    ...
  </Table_n>
</Tables>
```


Table-based Mapping (2)

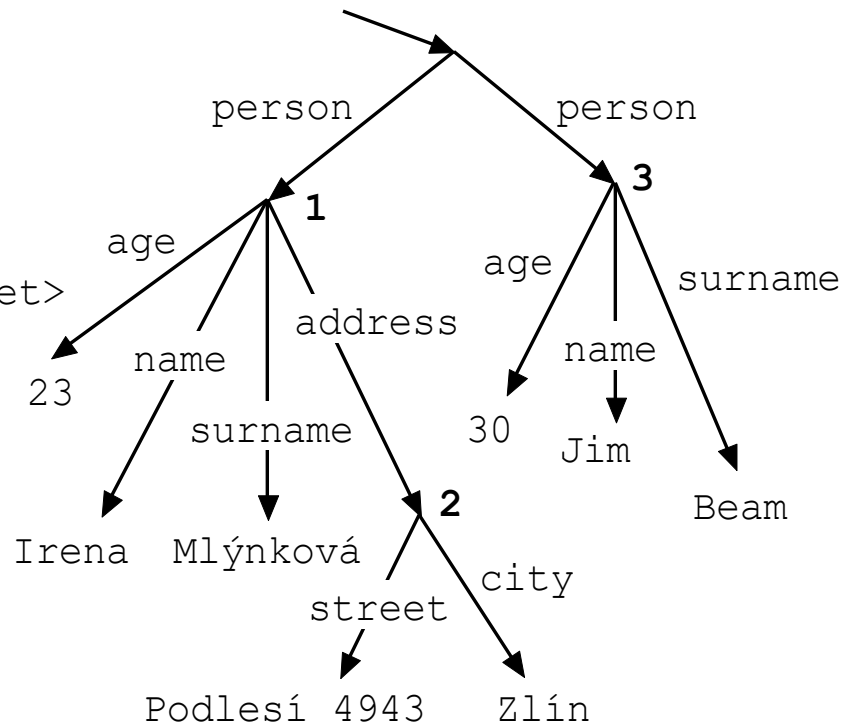
- Trivial case
 - The schema is an implicit part of the data
 - Only a limited set of documents can be stored
 - Typical usage: data transfer among multiple databases
 - There exist also more complex schemas, but the idea is the same
 - Basically again usage of (an implicit) schema
-

Generic-tree Mapping (1)

- The target relational schema enables to store any kind of XML data
 - Regardless their XML schema
 - XML document \leftrightarrow directed tree
 - Inner nodes have an ID
 - Leaves carry values of attributes or text nodes
 - Outgoing edges of a node represent subelements/attributes of the element represented by ingoing edge of the same node
 - Edges are labeled with element/attribute names
-

Generic-tree Mapping (2)

```
...  
<person id=1 age=23>  
  <name>Irena</name>  
  <surname>Mlýnková</surname>  
  <address id=2>  
    <street>Podlesí 4943</street>  
    <city>Zlín</city>  
  </address>  
</person>  
<person id=3 age=30>  
  <name>Jim</name>  
  <surname>Beam</surname>  
</person>  
...
```



Generic-tree Mapping (3)

□ Edge mapping

■ **Edge** (**sourceID**, **order**, **label**, **type**, **targetID**)

□ Type: inner edge, element/attribute edge, ...

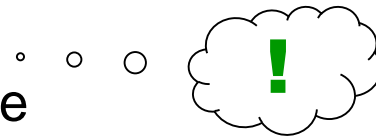
```
Edge (... , (1, 2, "name", element, -1), ...
           (1, 4, "address", inner, 2), ...)
```

□ Attribute mapping

■ Attribute = name of the edge

■ **Edge_{attribute}** (**sourceID**, **order**, **type**, **targetID**)

```
Edgename (... , (1, 2, element, -1), ...
               (3, 2, element, -1), ...)
```



Generic-tree Mapping (4)

- Universal mapping
 - `Uni (sourceID, ordera1, typea1, targetIDa1, ... orderak, typeak, targetIDak)`
 - Outer join of tables from attribute mapping
 - a_1, \dots, a_k are all the attribute names in the XML document
 - Too many null values
 - Normalized universal mapping
 - The universal table contains for each name just one record
 - Others (i.e. multi-value attributes) are stored in **overflow tables**
 - From edge mapping
-

Generic-tree Mapping (5)

- How do we store the leaf values?
 1. Special **value tables**, each for each data type used
 2. **Value columns** in the previous tables
 - Many null values (for each data type an extra column)
 - Or we ignore data types
 - Other options
 - Combination of previous approaches
 - E.g. attribute mapping for frequent attributes and edge mapping for other
-

Structure-centred Mapping (1)

- XML document \leftrightarrow directed tree
 - All nodes have the same structure:
 $N = (t, l, c, n)$, where
 - t is the type of node (i.e. ELEM, ATTR, TXT, ...)
 - l is the label of node (if exists)
 - c is text content of node (if exists)
 - $n = \{N_1, \dots, N_m\}$ is (possibly empty) list of child nodes
 - Variants of the algorithm = variants of storing the list of child nodes
 - Aim: efficient operations
-

Structure-centred Mapping (2)

1. Keys and foreign keys

- Each node is assigned with an ID (key) and ID of its parent node (foreign key)

(+) Simple, efficient updates

(−) Inefficient queries (joins of many tables)

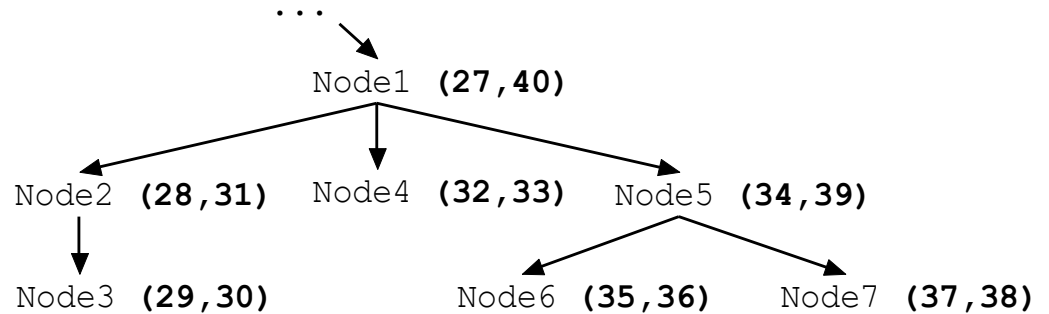
2. DF values

- Node ID = pair (DF_{min}, DF_{max})

- DF_{min} = the time of visiting a node

- DF_{max} = the time of leaving a node

Structure-centred Mapping (3)



(+) Efficient querying and reconstruction of a node

□ E.g. v is a descendant of u , if $u_{\min} < v_{\min}$ and $v_{\max} < u_{\max}$

□ The nodes can be ordered totally

(-) Inefficient updates

□ In the worst case we need to re-number the whole tree

Structure-centred Mapping (4)

$$\sigma = \frac{1}{q_k + \frac{1}{\dots \frac{1}{q_2 + \frac{1}{q_1}}}}$$

3. SICF (simple continued fraction) values

- SICF node identifier = σ , where $q_i \in \mathbf{N}$ ($i = 1, \dots, k$)

- Sequence $\langle q_1, \dots, q_k \rangle$ identifies the node

- For root node: SICF ID $\sigma = \langle s \rangle$, $s > 1$

- For all other nodes:

If node u has SICF ID = $\langle q_1, \dots, q_m \rangle$ and n child nodes u_1, \dots, u_n , then SICF ID of i -th child node is $\langle q_1, \dots, q_m, i \rangle$

- Resembles to ORDPATH

- Does not have its advantages

- We do not use the “trick” with odd and even numbers

(+) we have a more precise structural information

(–) like in the previous case

Simple-path Mapping (1)

- Assumption: XPath queries
- Idea: We can store all paths to all nodes in the documents
 - So-called **simple paths**

```
<SimpleAbsolutePathUnit> ::= <PathOp> <SimplePathUnit> |  
                           <PathOp> <SimplePathUnit> '@' <AttName>  
<PathOp>                  ::= '/'  
<SimplePathUnit>         ::= <ElementType> |  
                           <ElementType> <PathOp> <SimplePathUnit>
```

- Just a simple path is not sufficient information
 - It does not contain information about position/order of node in the document
-

Simple-path Mapping (2)

- Relational schema:
 - **Element** (IDdoc, IDpath, Order, Position)
 - **Attribute** (IDdoc, IDpath, Value, Position)
 - **Text** (IDdoc, IDpath, Value, Position)
 - **Path** (IDpath, Value)
 - **Order** of an element within its sibling nodes
 - **Position** of a word in a text is an integer value
 - **Position** of a tag is a real number
 - integral part = position of the closest preceding word
 - decimal fraction = position within tags following the closest preceding word
 - (+) Efficient processing of XPath queries
 - Implementation of ‘//’ using SQL LIKE
-

B. Schema-driven Mapping (1)

- Based on existence of an XML schema
 - Usually DTD or XML Schema
 - Algorithm:
 1. XML schema is mapped to relational schema
 2. XML data valid against the XML schema are stored into relations
 - i.e., for data with different structure (XML schema) we have a different relational schema
 - Aim: We want to create an optimal schema with "reasonable" amount of tables and null values and which corresponds to the source XML schema
-

B. Schema-driven Mapping (2)

- General characteristics of the algorithms:
 1. For each element we create a relation consisting of its attributes
 2. Subelements with maximum occurrence of one are (instead of to separate tables) mapped to tables of parent elements
 - so-called **inlining**
 3. Elements with optional occurrence → nullable columns
 4. Subelements with multiple-occurrence → separate tables
 - Element-subelement relationships are mapped using keys and foreign keys
 5. Alternative subelements →
 - separate tables (analogous to the previous case) or
 - one universal table (with many nullable fields)
-

B. Schema-driven Mapping (3)

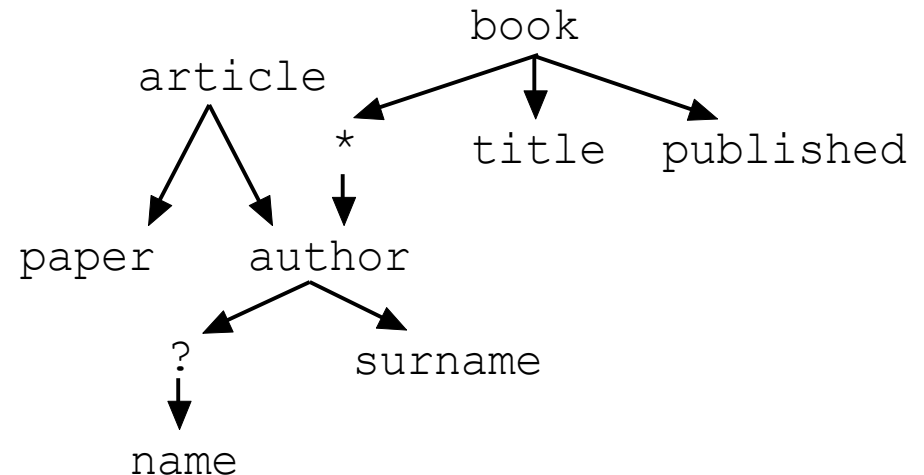
5. Order of sibling elements (if necessary) → special column
 6. Mixed-content elements usually not supported
 - Would require many columns with nullable fields
 7. Despite the previous optimizations a reconstruction of an element requires joining several tables.
- Most of the techniques use an auxiliary graph
 - Classification:
 - **Fixed methods** – exploit information only from schema
 - Basic, Shared and Hybrid
 - **Flexible methods** – exploit other information
 - LegoDB mapping, Hybrid object-relational mapping
-

Algorithms Basic, Shared and Hybrid (1)

- Continuous improvements of mapping a DTD to relational schema
 - One of the first approaches
 - **DTD graph** – auxiliary structure for creation of a relational schema
 - Nodes = elements (occur 1x) / attributes / operators
 - Directed edges = relationships element-subelement / element-attribute / element-operator / operator-element
 - Note: DTD is first "flattened" and simplified
 - Contains only operators * and ? ($+ \rightarrow *$, $a|b \rightarrow a?,b?$)
 - A classical trick
-

Algorithms Basic, Shared and Hybrid (2)

```
<!ELEMENT author (name?, surname) >
<!ELEMENT name (#PCDATA) >
<!ELEMENT surname (#PCDATA) >
<!ELEMENT book (author*, title) >
<!ATTLIST book published CDATA>
<!ELEMENT title (#PCDATA) >
<!ELEMENT article (author) >
<!ATTLIST article paper CDATA>
```



Algorithm Basic

□ Naïve approach

□ Rules:



1. For each element in the document create a separate relation

□ Motivation: The root element can be any element in the DTD

2. For each element inline as many child nodes as possible

□ We do not inline only child nodes of operator '*' and recursive subelements – they are stored in separate relations

(–) Too many relations

■ E.g. for our sample element **author** we would create two relations corresponding to two places of its usage within **book** and **article**

Algorithm Shared

- Idea: We want to map each element only once
 - Rules:
 1. Nodes with an in-degree of one are inlined to parent relations.
 2. Nodes with an in-degree of zero are stored in separate relations
 - They are not reachable from any other node
 3. Repeated elements are stored in separate relations.
 4. Of all mutually recursive elements having an in-degree one, one of them is stored in a separate relation.
 5. The problem of inlined elements, which can become roots of an instance XML document, is solved using a flag for each element that indicates this situation.
 - E.g. For our sample DTD graph we would create 3 relations **author**, **book**, **article**
- (–) The number of relations can be further reduced in some cases
-

Algorithm Hybrid

- Combination of maximum inlining of Basic and sharing in Shared
 - Rules:
 1. - 5. Same as in Shared
 6. In addition, we inline elements with an in-degree greater than one, that are neither recursive nor reached through a "*" node.
 - E.g. in our sample DTD graph it does not have any effect, but if **book** has only one **author**, it does
 - Further extension:
 - Storing of order of elements
 - Into special columns
 - Mapping of integrity constraints
 - ?, list of values, ID, IDREF, IDREFS, ...
 - [NOT] NULL, CHECK, UNIQUE, PRIMARY/FOREIGN KEY, ...
-

LegoDB Mapping (1)

- Idea: For the given XML schema we create a space of possible mappings and we select the optimal one for the given application
 - Application:
 - Sample XML documents
 - Sample XML queries + their significance
 - One step:
 1. We apply a selected transformation on the given XML schema S_{old}
 - We get a new XML schema S_{new}
 2. XML schema S_{new} is mapped (using a fixed method) to relational schema S_{rel}
 3. Sample queries are evaluated with regard to S_{rel}
 4. $S_{old} = S_{new}$
-

LegoDB Mapping (2)

- The space of possible XML transformations is infinite
 - Heuristics, greedy search strategies, ...
 - XML transformations
 - Inlining / outlining
 - $(a,(b|c)) = (a,b|a,c)$
 - $(a^+) = (a,a^*)$
 - $(a|b) \subseteq (a?,b?)$
 - $\sim = (a|(\sim!a))$, where \sim means any element and $\sim!a$ any element except for a
 - The static mapping is similar to Hybrid algorithm
-

LegoDB Mapping (3)

- (+) The most efficient mapping for the specified application
 - (−) If the application changes (the user starts to specify different queries)
 - Efficiency can be worse than in case of a fixed mapping
 - Modification of a schema is not an easy task
-

Hybrid Object-relational Mapping (1)

- Motivation: Data in XML documents are semi-structured → classical decomposition of unstructured parts leads to inefficient queries
 - i.e., we create many tables which we have to join to retrieve the data
 - Solution
 - Structured parts of the data are mapped into relations
 - Unstructured parts are stored into special **XML data types**
 - Data type for XML fragments
 - Support for XML operations
 - Motivation for SQL/XML data type XML
 - or BLOB if we do not need XML operations
 - Core problem of the algorithm: Which parts of the document are unstructured?
-

Hybrid Object-relational Mapping (2)

□ Approach:

1. Creating of DTD graph G_1
 2. For each node we evaluate the **measure of significance** ϖ
 3. Subgraphs denoted with unstructured nodes are replaced with an auxiliary attribute for XML type \rightarrow DTD graph G_2
 1. The node is not a leaf
 2. The node and its descendants have $\varpi < \text{LOD}$
 - Level of detail
 3. The node does not have a parent node that would satisfy the conditions
 4. Graph G_2 is statically mapped to a relational schema
-

Hybrid Object-relational Mapping (3)

$$w = \frac{1}{2}w_s + \frac{1}{4}w_D + \frac{1}{4}w_Q$$

- Meaning of the variables:
 - w_S (weight derived from the DTD structure)
 - The combination of values expressing the position of the element/attribute in the graph
 - w_D (weight derived from the existing XML data)
 - The ratio of the number of documents containing the element/attribute and the absolute number of documents
 - w_Q (weight derived from the queries)
 - The ratio of the number of queries containing the element/attribute and the absolute number of queries

(+) and (-) like in the previous case

C. User-defined Mapping

- The whole mapping process is defined by the user
 - Algorithm:
 1. The user creates the target relational schema
 2. The user specifies the required mapping (using a system-dependent interface)
 - Usually a declarative interface, annotations in XML schemas, special query languages, ...
 - (+) The most flexible approach
 - The user knows what (s)he wants
 - (−) The user must know several advanced technologies, the definition of an optimal relational schema is not an easy task
-

User-driven Mapping (1)

- An attempt to solve the disadvantages of user-defined mapping
 - Idea: an implicit method + user-defined local changes
 - **Annotation of schema** = user denotes fragments (subtrees) whose storage strategy should be modified
 - Pre-defined set of allowed changes of mapping
 - Usually a set of attributes and their values
 - Example – system XCacheDB
-

User-driven Mapping – XCacheDB (2)

- ❑ **INLINE** – inline the fragment into parent table
 - ❑ **TABLE** – store the fragment into a separate table
 - ❑ **BLOB_ONLY** – store the fragment into a BLOB column
 - ❑ **STORE_BLOB** – store the fragment implicitly + into a BLOB column
 - ❑ **RENAME** – change the name of table of column
 - ❑ **DATATYPE** – change the data type of the column
-

Current State of the Art of XML Databases

- Native databases vs. XML-enabled databases
 - The difference is fading away
 - Oracle DB, IBM DB2, MS SQL Server – the storage is defined by the user
 - BLOB
 - Native XML storage (typically parsed XML data + ORDPATH numbering schema)
 - Decomposition into relations – fixed schema-driven or user-driven
 - Currently user-driven annotations often denoted as obsolete
 - Standard bridge between XML and relational world: SQL/XML
-