# Module 4

# Implementation of XQuery

## Part 2: Data Storage

# Aspects of XQuery Implementation

- Compile Time + Optimizations
  - Operator Models
  - Query Rewrite
  - Runtime + Query Execution
- XML Data Representation
  - XML Storage
  - XML Indexes
  - Compression + Binary XML

# Questions to ask for XML storage

- ***What*** actions are done with XML data?
- ***Where*** does the XML data live?
- ***How*** is the XML data processed?
- ***In which*** granuluarity is XML data processed?

- **There is no one fits all solution !?!**
  **(This is an open research question.)**

# What?

- Possible uses of XML data
  - ship (serialize)
  - validate
  - query
  - transform (create new XML data)
  - update
- Example:
  - UNICODE reasonably good to ship XML data
  - UNICODE terrible to query XML data

# Where?

- Possible locations for XML data
  - wire (XML messages)
  - main-memory (intermediate query results)
  - disk (database)
  - (mobile devices)
- Example
  - Compression great for wire and mobile devices
  - Compression not good for main-memory (?)

# How?

- Alternative ways to process XML data
  - materialized, all or nothing
  - streaming (on demand)
  - anything in between
- Examples
  - trees good for materialization
  - trees bad for stream-based processing

# Granularity?

- Possible granularities for data processing:
  - documents
  - items (nodes and atomic values)
  - tokens (events)
  - bytes
- Example
  - tokens good for fine granularity (items)
  - tokens bad for whole documents

# Scenario I: XML Cache

- Cache XHTML pages or results of Web Service calls

| | | | | | |
|---|---|---|---|---|---|
| ship | yes | wire | yes | materialize | yes |
| validate | maybe | m.-m. | yes | stream | maybe |
| query | no | disk | yes | granularity | docs/ items |
| transform | maybe | | | | |
| update | no | | | | |

# Scenario II: Message Broker

- Route messages according to simple XPath rules
- Do simple transformations

| | | | | | |
|---|---|---|---|---|---|
| ship | yes | wire | yes | materialize | no |
| validate | yes | m.-m. | yes | stream | yes |
| query | yes | disk | no | granularity | docs |
| transform | yes | | | | |
| update | no | | | | |

# Scenario III: XQuery Processor

- apply complex functions
- construct query results

| | | | | | |
|---|---|---|---|---|---|
| ship | no | wire | yes | materialize | yes |
| validate | yes | m.-m. | yes | stream | yes |
| query | yes | disk | maybe | granularity | item |
| transform | yes | | | | |
| update | no | | | | |

# Scenario IV: XML Database

- Store and archive XML data

| | | | | | |
|---|---|---|---|---|---|
| ship | yes | wire | no | materialize | yes |
| validate | yes | m.-m. | yes | stream | yes |
| query | yes | disk | yes | granularity | collection ? |
| transform | yes | | | | |
| update | yes | | | | |

# Object Stores vs. XML Stores

- Similarities
  - nodes are like objects
  - identifiers to access data
  - support for updates
- Differences
  - XML: tree not graph
  - XML: everything is ordered
  - XML: streaming is essential
  - XML: dual representation (lexical + binary)
  - XML: data is context-sensitive

# XML Data Representation Issues

- Data Model Issues
  - InfoSet vs. PSVI vs. **XQuery data model**
- Storage Structures basic Issues
  1. Lexical-based vs. typed-based vs. both
  2. Node indentifiers support
  3. Context-sensitive data (namespaces, base-uri)

  ...
- Storage alternatives:
  - Plain text
  - Trees
  - Arrays of Tokens
  - Binary XML
  - Items
  - (Relational) tables

# Lexical vs. Type-based

- Data model requires both properties, but allows only one to be stored and compute the other
- Functional dependencies
  - string + type annotation -> value-based
  - value + type annotation -> schema-norm. string
- Example

  "0001" + xs:integer -> 1

  1 + xs:integer -> "1"
- Tradeoffs:
  - Space vs. Accuracy
  - Redundancy:  cost of updates
  - indexing: restricted applicability

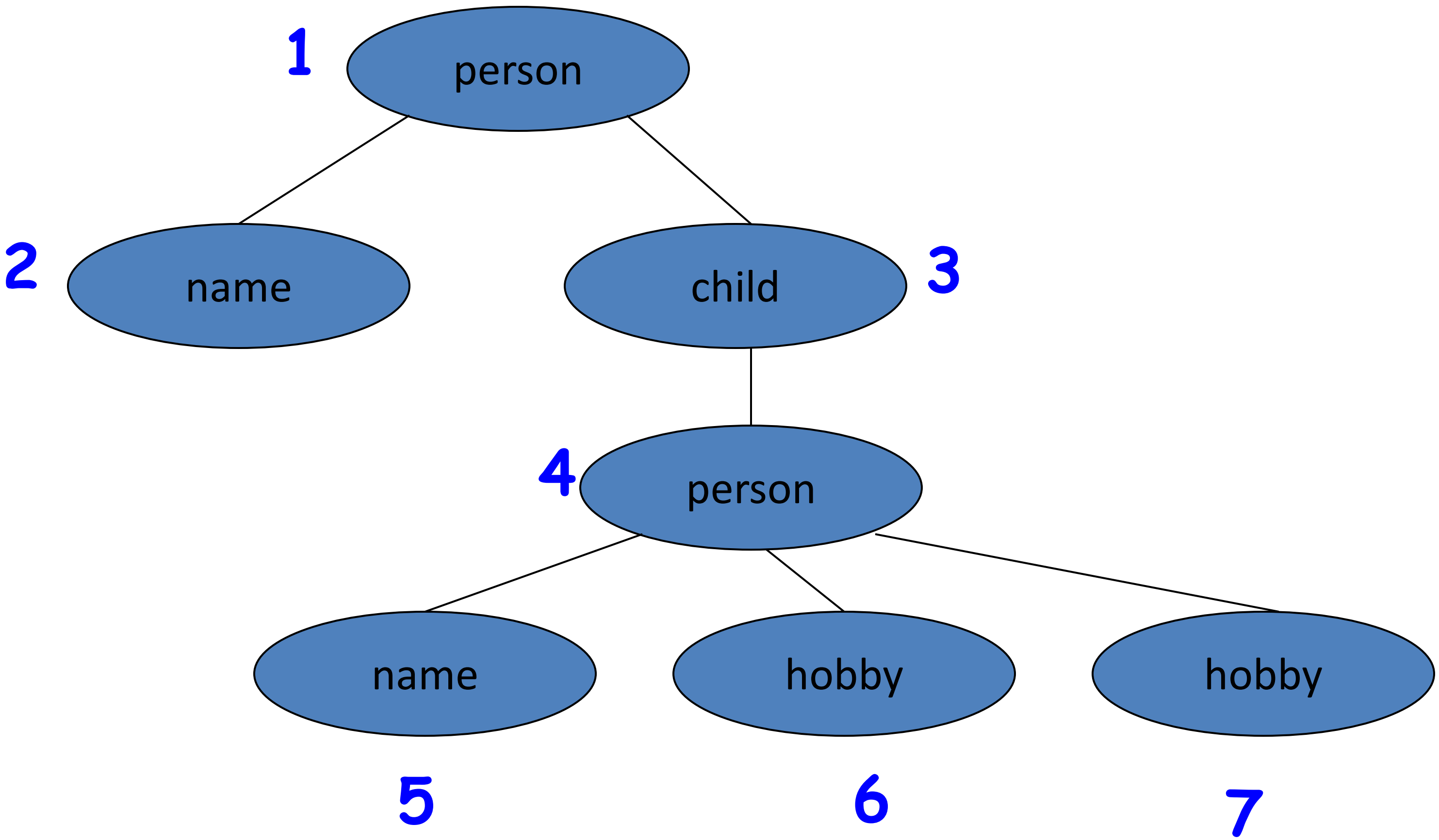
# Node Identifiers Considerations

- XQuery Data Model Requirements
  - identify a node uniquely (implements identity)
  - lives as long as node lives
  - robust to updates
- Identifiers might include additional information
  - Schema/type information
  - Document order
  - Parent/child relationship
  - Ancestor/descendent relationship
  - Document information
- Required for indexes

# Simple Node Identifiers

- Examples:
  - Alternative 1 (data: trees)
    - id of document (integer)
    - pre-order number of node in document (integer, possibly float)
  - Alternative 2 (data: plain text)
    - file name
    - offset in file
- Encode document ordering (Alternative 1)
  - identity: doc1 = doc2 AND pre1 = pre2
  - order: doc1 < doc2
    OR (doc1 = doc2 AND pre1 < pre2)
- Not robust to updates
- Not able to answer more complex queries

# Example: Pre-Order Integer Identifiers



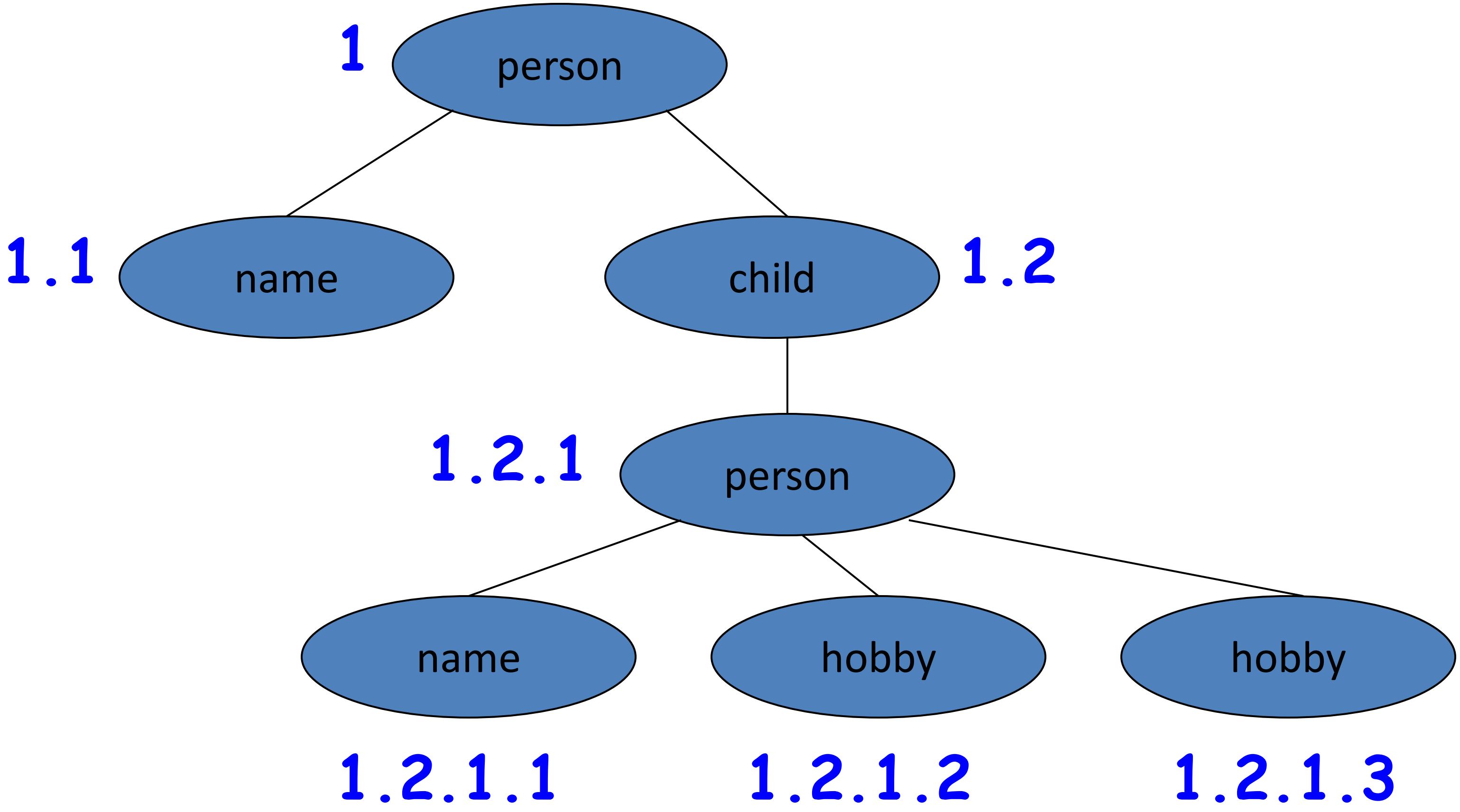
# Dewey Order

Tatrinov et al. 2002

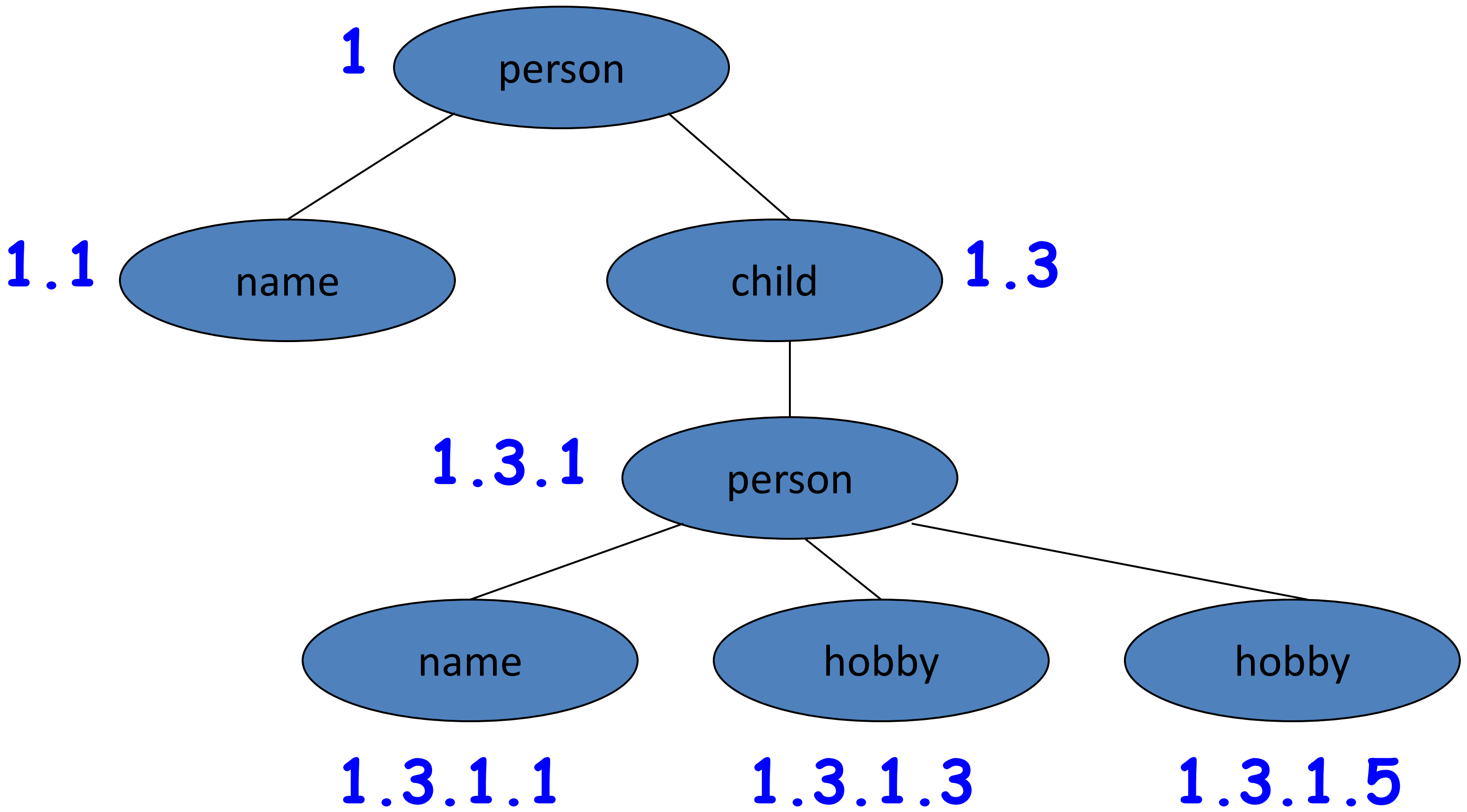
- Idea:
  - Generate surrogates for each path
  - 1.2.3 identifies the third child of the second child of the first child of the given root
- Assessment;
  - **good:** order comparison, ancestor/descendent easy
  - **bad:** updates expensive, space overhead
- Improvement: ORDPath Bit Encoding

  O'Neil et al. 2004 (Microsoft SQL Server)

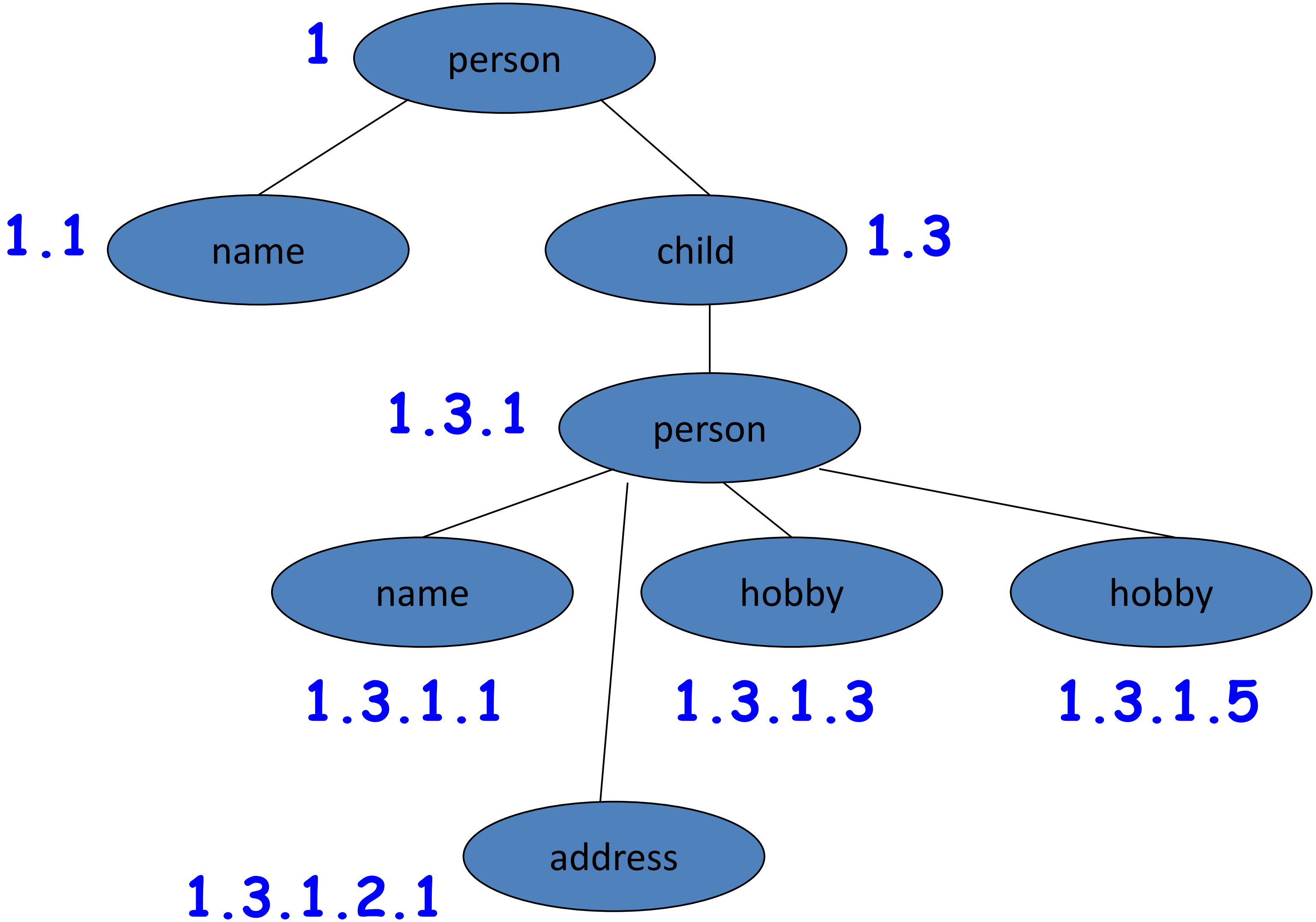
# Example: Dewey Order (without ORDPATH)



# Example: Dewey Order (ORDPATH)



# ORDPATH and updates



# ORDPath Details

- Use odd numbers for „regular“ numbering
- Gaps for even numbers are used for inserting
  - Fill in even value
  - Extend path depth by one with regular, odd values
  - Count height by looking only at odd components
  - Essentially infinite inserting
- Efficient representation possible: Bitstrings!
- Effective ORDPath lengths fairly short

# XML Storage Alternatives

- Plain Text (UNICODE)
- Trees with Random Access
- Binary XML / arrays of events (tokens)
- Item Model
- Tuples  (e.g., mapping to RDBMS)

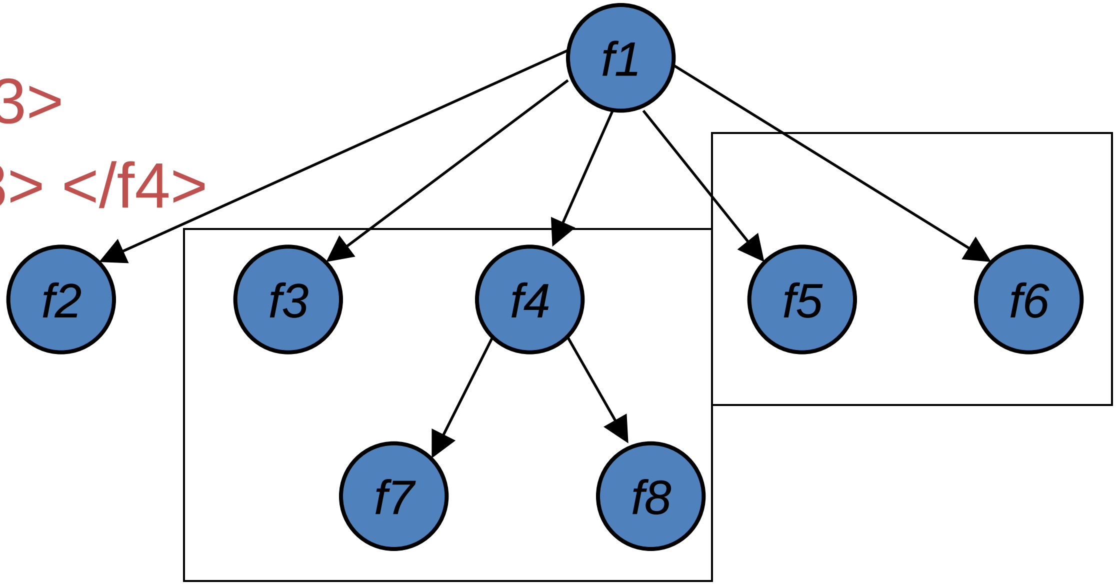
# Plain Text

- Use XML standards to encode data
- Advantages:
  - simple, universal
  - indexing possible
- Disadvantages:
  - need to re-parse (re-validate) all the time
  - no compliance with XQuery data model (collections)
  - not an option for XQuery processing

# Trees

- XML data model uses tree semantics
  - use Trees/Forests to represent XML instances
  - annotate nodes of tree with data model info
- Example

<f1>
<f2>..</f2> <f3>..</f3>
<f4> <f7/> <f8>..</f8> </f4>
<f5/> <f6>..</f6>
</f1>



# Trees

- Advantages
  - natural representation of XML data
  - good support for navigation, updates index built into the data structure
  - compliance with DOM standard interface
- Disadvantages
  - difficult to use in streaming environment
  - difficult to partition
  - high overhead: *mixes indexes and data*
  - index everything
- Example: DOM, *others*
- Lazy trees possible: minimize IOs, able to handle large volumes of data

# Trees on disk: Natix

Moerkotte 2000ff

- Each sub-tree is stored in a *record*
- Store records in blocks as in any database
- If record grows beyond size of block: *split*
- Split: establish *proxy nodes* for subtrees
- Technical details:
  - use B-trees to organize space
  - use special concurrency & recovery techniques

# Natix

<bib>

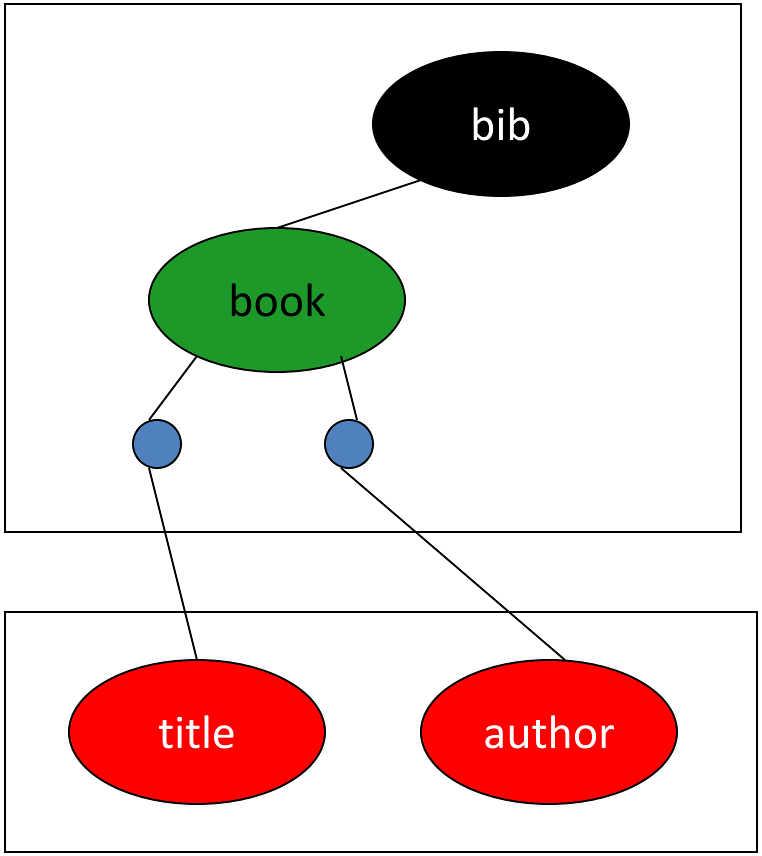
<book>

<title>...</title>

<author>...</author>

</book>

</bib>



# Binary XML as a flat array of „events“

- Linear representation of XML data
  - pre-order traversal of XML tree
- Node -> array of events  (or tokens)
  - tokens carry the data model information
- Advantages
  - good support for stream-based processing
  - low overhead: separate indexes from data
  - logical compliance with SAX standard interface
- Disadvantages
  - difficult to debug, difficult programming model
  - possibly high runtime overhead (very fine-grained)
- Current standardization efforts: EXI

# Example Binary XML as tokens array

```
<?xml version="1.0">
<order id="4711" >
   <date>2003-08-19</date>
   <lineitem xmlns = "www.boo.com" >
   </lineitem>
</order>
```

# Translating XML into Event Stream (I)

```
<?xml version="1.0">
<order id="4711" >
    <date>2003-08-19</date>
```

```
BeginDocument()
BeginElement(„order“, „xs:untypedAny“, 1)
BeginAttribute(„id“, „xs:untypedAtomic“, 2)
CharData(„4711“)
EndAttribute()
BeginElement(„date“, „xs:untypedAny“, 3)
Text(„2003-08-19“, 4)
EndElement()
```

# Translating XML into Event Stream (II)

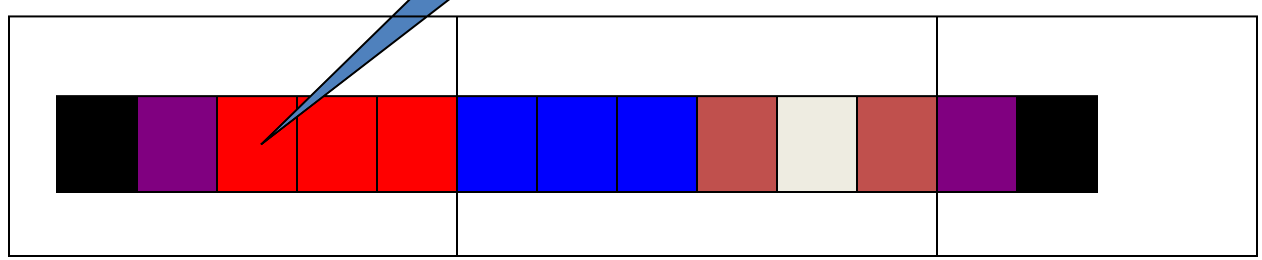
<lineitem xmlns = „www.boo.com“ >
</lineitem>
</order>

```
BeginElement(„www.boo.com:lineitem“
  , „xs:untypedAny“, 5)
NameSpace(„www.boo.com“, 6)
EndElement()
EndElement()
EndDocument()
```

# Storing Event Stream as Token Sequence

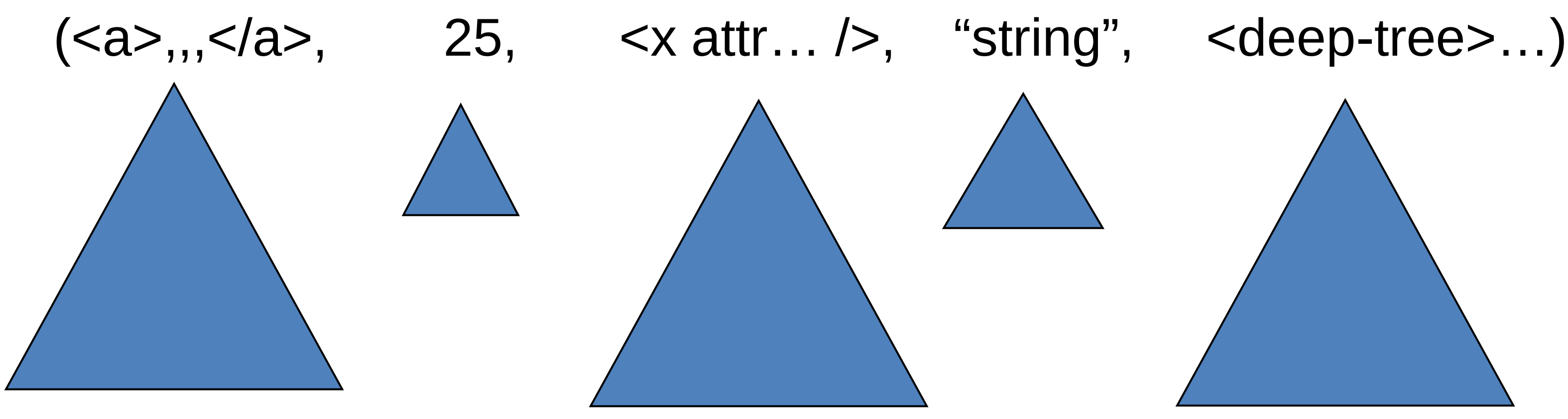
BeginDocument()
BeginElement(„order“, „xs:untypedAny“, 1)
BeginAttribute(„id“, „xs:untypedAtomic“, 2)
CharData(„4711“)
EndAttribute()
BeginElement(„date“, „xs:untypedAny“, 3)
Text(„2003-08-19“, 4)
EndElement()
BeginElement(„www.boo.com:lineitem“, „xs:untypedAny“, 5)
NameSpace(„www.boo.com“, 6)
EndElement()
EndElement()
EndDocument()

**AttributeToken**
(attributeName = "id", dataType = xs:untypedAtomic, nodeID = 2)



# Item Model (Zorba)

- Use XDM item model:
  - Atomic Values / Nodes
  - Item Accessors: attributes, children, node-kind, namespace-nodes, string-value, …
  - Internal implementation opaque



# Item Model - Advantages

- Good match to formal specifications
- Many iterators simpler to implement
  - For-bindings get easier
    (just one item - doesn't need materialization)
- Lazy evaluation/materialization possible
- Coarse granularity => fewer next() calls => Faster
- Smaller memory consumption memory
  - (start-token + end-token = just one item)
  - Coarse grained / lazy materialization inside item

# Item Model - Disadvantages

- Laziness is harder to achieve:
  an item has to take an iterator
- Node construction harder
  (e.g. attributes need special consideration)
- Interaction with Indexes:
  - Item = Tree = primary index?
  - Indexes into the item?
- Memory management troubles:
  - Streaming in coarse granularity, possible degradation into non-streaming query processing (item = root node of 1 GB file)
  - Explicit analysis of free/used inside item necessary

# Item Model vs Trees vs Tokens

- Better alignment with XDM, operator semantics
- Higher abstraction level, many physical implementations possible
  - Unparsed Text, Tokens, Integrated indexes,...
- Fine-Grained operations more difficult
- Complexity in the item implementation

=> This is still research work,
many open issues

# XML Data represented as tuples

- Motivation: Use an RDBMS infrastructure to store and process the XML data
  - transactions
  - scalability
  - richness and maturity of RDBMS
- Alternative relational storage approaches:
  - Store XML as Blob (text, binary)
  - *Generic shredding of the data (edge, binary, …)*
  - *Map XML schema to relational schema*
  - Binary (new) XML storage integrated tightly with the relational processor (=> see before, maybe also in a later talk)

# Mapping XML to tuples

- External to the relational engine
  - Use when :
    - The structure of the data is relatively simple and fixed
    - The set of queries is known in advance
  - Processing involves hand written SQL queries + procedural logic
  - Frequently used, but not advantageous
    - Very expensive (performance and productivity)
    - Server communication for every single data fetch
    - Very limited solution
- Internally by the relational engine
  - See next slides

# XML Example

```
<person id = 4711>
   <name>  Lilly Potter </name>
   <child>  <person id = 314>
          <name> Harry Potter </name>
          <hobby> Quidditch </hobby>
  </child>
</person>
<person id = 666>
   <name> James Potter </name>
   <child>  314 </child>
</person>
```

<person id = 4711>

<name> Lilly Potter </name>

<child> <person id = 314>

<name> Harry Potter </name>

</child>

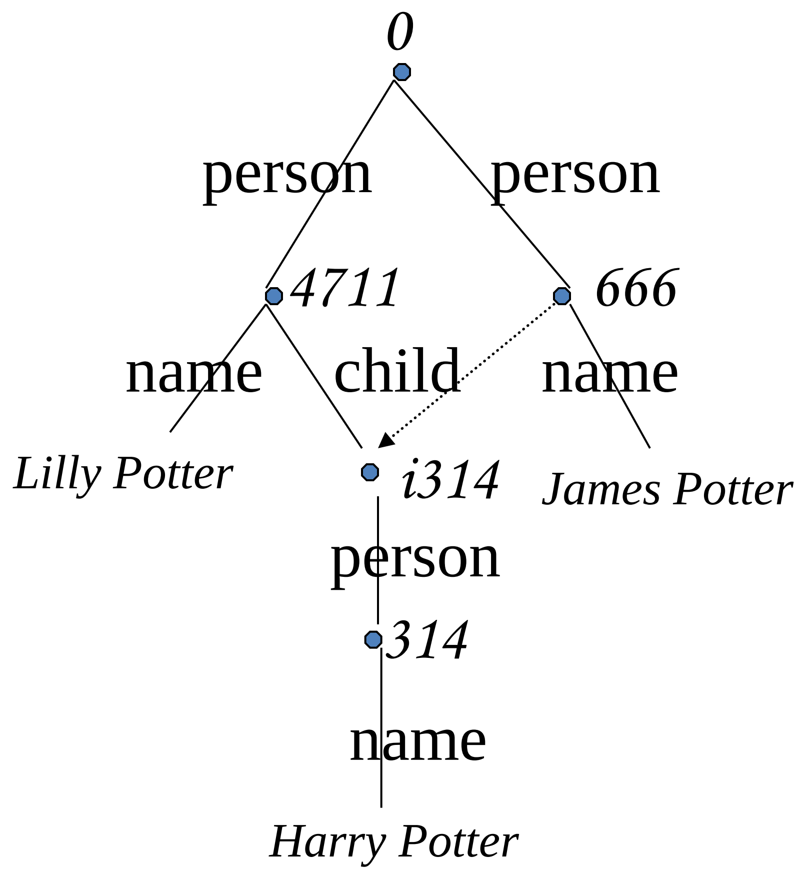
</person>

<person id = 666>

<name> James Potter </name>

<child> 314 </child>

</person>



# Edge Approach

(Florescu & Kossmann 99)

*Edge Table*

| Source | Label | Target |
|---|---|---|
| 0 | person | 4711 |
| 0 | person | 666 |
| 4711 | name | v1 |
| 4711 | child | i314 |
| 666 | name | v2 |
| 666 | child | i314 |

*Value Table (String)*

| Id | Value |
|---|---|
| v1 | Lilly Potter |
| v2 | James Potter |
| v3 | Harry Potter |

*Value Table (Integer)*

| Id | Value |
|---|---|
| v4 | 12 |

# Binary Approach

Partition *Edge* Table by *Label*

*Person Table*

| Source | Target |
|---|---|
| 0 | 4711 |
| 0 | 666 |
| i314 | 314 |

*Name Table*

| Source | Target |
|---|---|
| 4711 | v1 |
| 666 | v2 |
| 314 | v3 |

*Child Table*

| Source | Target |
|---|---|
| 4711 | i314 |
| 666 | i314 |

*Age Table*

| Source | Target |
|---|---|
| 314 | v4 |

# Axis navigation

- How to compute a/b:
  - Follow the edge, match type/label

- How to compute a//b:
  - Recursively follow edge from a

  => High cost!

# Tree Encoding

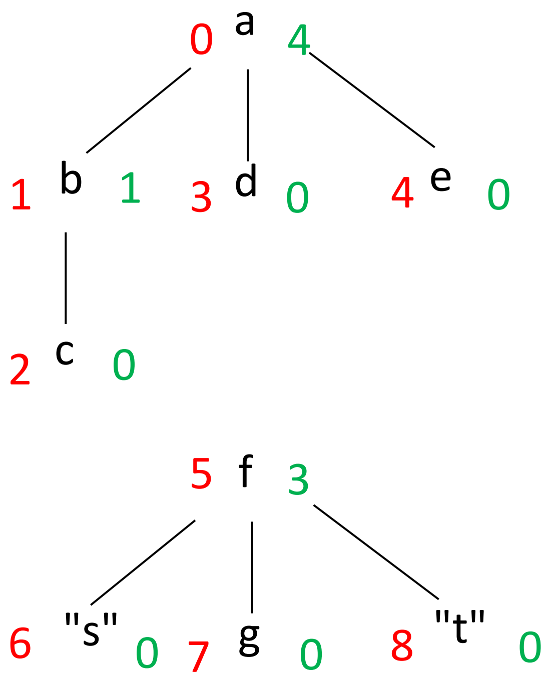
(Grust 2004)

- For every node of tree, keep info
  - ***pre:*** pre-order number
  - ***size:*** number of descendants
  - ***level:*** depth of node in tree
  - ***kind:*** element, attribute, name space, …
  - ***prop:*** name and type
  - ***frag:*** document id (forests)

# Tree Encoding – Labeling of Tree

```
<a>
  <b><c/></b>
  <d/>
  <e>
</a>

<f>
  s<g/>t
<f>
```

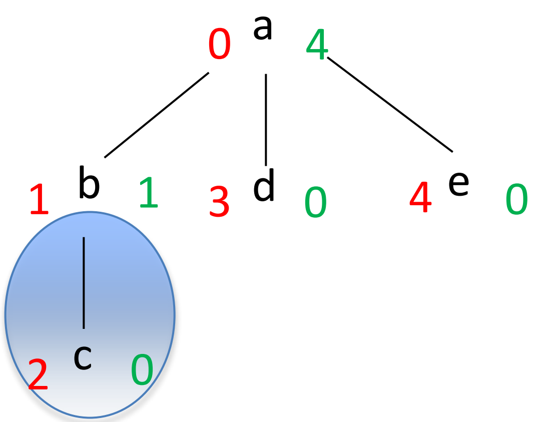


# Example: Tree Encoding

| *pre* | *size* | *level* | *kind* | *prop* | *frag* |
|---|---|---|---|---|---|
| 0 | 4 | 0 | elem | a | 0 |
| 1 | 1 | 1 | elem | b | 0 |
| 2 | 0 | 2 | elem | c | 0 |
| 3 | 0 | 1 | elem | d | 0 |
| 4 | 0 | 1 | elem | e | 0 |
| 5 | 3 | 0 | elem | f | 1 |
| 6 | 0 | 1 | text | s | 1 |
| 7 | 0 | 1 | elem | g | 1 |
| 8 | 0 | 1 | text | t | 1 |

# Descendant Axis

- Descendant
- v: Context Node
- v‘: match candidates
- pre(v) < pre(v‘) && pre(v‘) <= pre(v)+size(v)

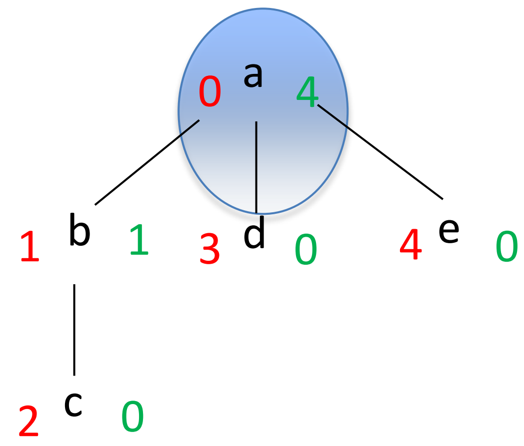



- What about the child axis?
- pre(v) < pre(v‘) && pre(v‘) <= pre(v)+size(v) && level(v)+1 = level(v‘)

# Some other axes

Ancestor:

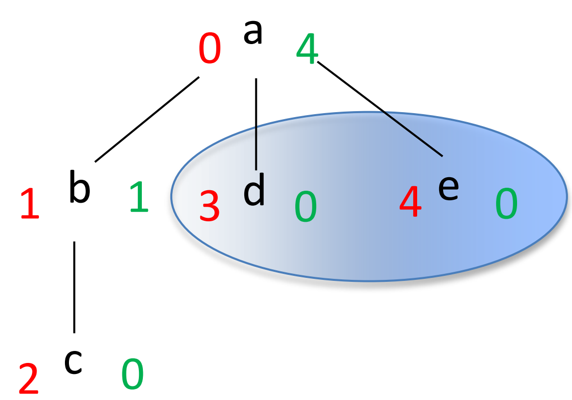
- pre(v‘) < pre(v) && pre(v‘)+size(v‘) > pre(v)+size(v)




Following:

- pre(v‘) > pre(v)+size(v)

....



# Query Plans and Index Support

- Why is pre(v) < pre(v‘) <= pre(v)+size(v) useful?
- Think about typical indexes: B-Tree
  - Scan Index to find (v): log (Data Size)
  - Assume that we use a row store, size(v) is there
  - Linear Scan from v until size(v) items have been read
  - Nodes are in document order (pre!)
- Child axis:
  use a clustered index on level, pre:
- [level (v) + 1, pre(v)] < [level (v‘), pre(v‘)] <=
  [level (v) + 1, pre(v) + size(v)]
- Yet again a index scan!

# Schema-driven Mapping: DTD -> RDB

Shanmugasundaram et al. 1999

- Idea: Translate DTDs into Relations
  - Element Types -> Tables
  - Attributes -> Columns
  - Nesting (= relationships) -> Tables
  - „Inlining“ reduces fragmentation
- Special treatment for recursive DTDs
- Surrogates as keys of tables
- (Adaptions for XML Schema possible)

# Example

```
<!ELEMENT book (title, author)>
<!ELEMENT article (title, author*)>
<!ATTLIST book price CDATA>
<!ELEMENT title (#PCDATA)>
<!ELEMENT author (firstname, lastname)>
<!ELEMENT firstname (#PCDATA)>
<!ELEMENT lastname (#PCDATA)>
<!ATTLIST author age CDATA>
```

# Example: Relation „book“

<!ELEMENT book (title, author)>
<!ELEMENT article (title, author*)>
<!ATTLIST book price CDATA>
<!ELEMENT title (#PCDATA)>
<!ELEMENT author (firstname, lastname)>
<!ELEMENT firstname (#PCDATA)>
<!ELEMENT lastname (#PCDATA)>
<!ATTLIST author age CDATA>

- **book(<u>bookID</u>, book.price, book.title, book.author.fname, book.author.lname, book.author.age)**
- **Inlining possible, since 1:1 relation between book and author**

# Example: Relation „article“

<!ELEMENT book (title, author)>
<!ELEMENT article (title, author*)>
<!ATTLIST book price CDATA>
<!ELEMENT title (#PCDATA)>
<!ELEMENT author (fname, lname)>
<!ELEMENT firstname (#PCDATA)>
<!ELEMENT lastname (#PCDATA)>
<!ATTLIST author age CDATA>

- **article (artID, art.title)**
- **artAuthor (<u>artAuthorID</u>, *artID*, art.author.fname, art.author.lname, art.author.age)**
- **1:N article:author => separate tables, embed foreign key**

# Example (continued)

- Represent each element as a relation
  - element might be the root of a document

**title(<u>titleId</u>, title)**
**author(<u>authorId</u>, author.age, author.fname, author.lname)**
**fname(<u>fnameId</u>, fname)**
**lname(<u>lnameId</u>, lname)**

# Recursive DTDs

<!ELEMENT book (author)>

<!ATTLIST   book title CDATA>

<!ELEMENT author (book*)>

<!ATTLIST   author name CDATA>

- **book(bookId, book.title, book.author.name)**
- **author(authorId, author.name)**
- **author.book(author.bookId, *authorId*, author.book.title)**

# Tradeoffs of tuple mapping variants (I)

- Store XML as a BLOB in relational database
  - index by materializing indexed expressions in separate columns
  - Plus: store XML in parsed and validated form
  - Minus: proprietary solution (blob is a black box)
  - Minus: replicate data for indexing
- Model-driven Shredding
  - Edge, binary approach + alternatives
  - Corresponds to generic APIs for Java
  - Plus: very general; integrates well with relational data
  - Minus: lower performance than specific mappings

# Tradeoffs of tuple mapping variants (II)

- Schema-based Shredding
  - Map XML Schema / DTD to SQL DDL
  - Plus: integrates well with relational data
  - Minus:  missing tools, complicated
- SQL / XML
  - Not discussed here
  - Extend SQL with XML data type
  - Plus: integrates well with relational data
  - Minus: not clear how it integrates with application, odd „marriage“