Query languages 1 (NDBI001)

XML

Jaroslav Pokorný

MFF UK, Praha

pokorny@ksi.mff.cuni.cz

Content

Part I: Introduction

Documents vs. databases

The world of documents

> a lot of small documents

> usually static

> implicit structure

section, paragraph, sentence,

> tagging

> adapted to a person

> content

format/annotation

> paradigms

“store as”, wysiwyg

> metadata

author’s name, date, subject

The world of databases

> a number of large databases

> usually dynamic

> explicit structure (schema)

> records

> adapted to a machine

> content

schema, data, methods

> paradigms

atomicity, parallelism, isolation, durability

> metadata

schema description

What to do with them?

Documents

- editing
- printing
- lexical control
- counting words
- information retrieval (IR)
- search

Databases

- actualization
- data cleaning
- querying
- transformations

Boundaries between documents and db

- Boundaries between the world of documents and the world of databases is not clear.
- In some proposals both approaches are legal.
- Somewhere in the middle are
  - formatting languages
  - semistructured data

research
in the second half of 90ties

Semistructured data is defined as data, which is unordered or incomplete, its structure may change, even unpredictably.

Ex.: data in web sources, HTML pages, Bibtex files, biological data.

Markup languages: HTML, XML, XHTML, …

HTML

- Lingua franca for publishing hypertext on the WWW
- Designed for presentation, how the web browser should display the text, pictures and buttons on a web page.
- Fixed set of tags, attributes, nesting elements, …, but allowing some irregularities, simple to learning, …
- HTML $\in$ XML $\subset$ SGML

Text (PCDATA)

start tag

end tag

```
<HTML>
<HEAD><TITLE>Sejdeme se Na Vlachovce</TITLE></HEAD>
<BODY>

<H1>Introduction</H1>
<IMG SRC="http://.../logo.jpeg" WIDTH="200" HEIGHT="150" >

</BODY>
</HTML>
```

attribute name

attribute name

Part II: XML language

- XML structure
- XML text
- XML attributes
- Tree structure of XML

XML structure

- XML is a content markup language
- XML data is an instance of semistructured data.
- XML consists of tags and a text
- tags occur in pairs <date> ...</date>
- must be properly nested

<date> <day> ... </day> ... </date> --- well

<date> <day> ... </date>... </day> --- wrong

(not possible: <i> ... ... </i> ...)

XML text

- XML has only one "basic" type -- text.
- Text is bounded by tags, e.g.,

  <title> Database alphabet </title>

  <year> 1999 </year> --- 1999 is a text
- XML text is of type PCDATA (Parsed Character DATA). It uses 16-bit encoding (Unicode).
- Later we will see, how with XML data can be specified new types.

XML structure

Nested tags can be used to express different structures. For example, n-tuple (row):

```
<person>
  <name> Jane Smith </name>
  <phone> 2191 4264 </phone>
  <email> smith@ksi.ms.mff.cuni.cz </email>
</person>
```

XML structure (cont.)

A list can be represented using repeatedly the *same* tag:

```
<addresses>
  <person> ... </person>
  <person> ... </person>
  <person> ... </person>
  ...
</addresses>
```

Terminology

Segment of XML document with start and corresponding end tag is called element. The text between tags is element content.

element

<person>

<name> Jane Smith </name>

<phone> 2191 4264 </phone>

<phone> 2191 4363 </phone>

<email> smith@ksi.mff.cuni.cz <email>

/person>

is not element

subelement

Element can be empty (it has no content – except of attributes)

<name> </name> or shortly <*name/*>

Terminology

Start tag of an element can contain attributes. They are used typically to description of element content.

```
<item>
  <word language = "A"> cheese </word>
  <word language = "F"> fromage </word>
  <word language = "N"> Käse </word>
  <meaning> Food created… </meaning>
</item>
```

Attributes

Further use - expressing dimensions or types

```
<picture>
    <height dim = “cm”> 2400 </height>
    <width dim = “cm”> 96 </ width>
    <data coding = “gif” compression = “zip”>
        M05-.+C$@02!G96YE<FEC ...
    </data>
</picture>
```

Mixed content

Element can contain mix of elements and data of type PCDATA

```
<washing>
  <name> Persil 1.2 </name>
    <motto>
        The world <dubious> favorite </dubious> of
        washing powder
    </motto>
</washing>
```

Remark: data of this form is not typically generated from (relational) databases.

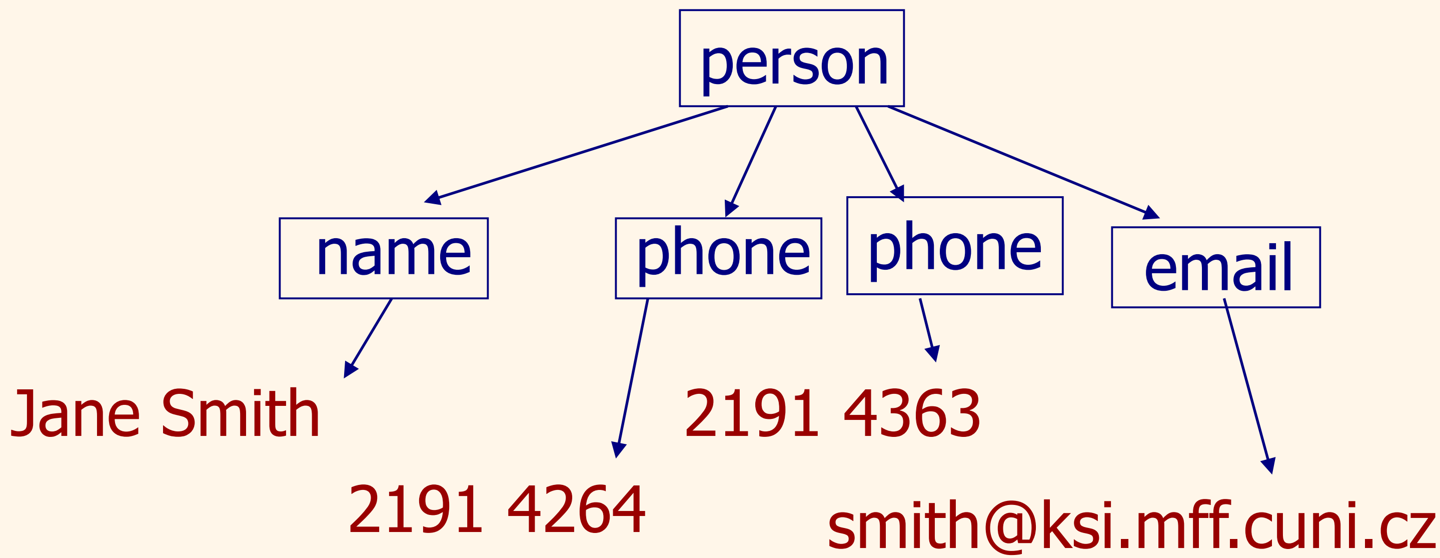
Complete XML Document

```
<?xml version="1.0"?>
<person>
  <name> Jane Smith </name>
  <phone> 2191 4264 </phone>
  <email> smith@ksi.mff.cuni.cz </email>
</person>
```

XML declaration

XML has a tree structure

- Figure contains a model of an XML text
- differences w.r.t. models of semistructured data, which use typically edge labeling



Example: relational DB representation

projects:

p_name	budget	controlled

employees:

name	PIN	age

Relations projects and employees in XML

projects and employees are mixed

```
<db>
  <project>
    <p_name> Searching </p_name>
    <budget> 100000 </budget>
    <controlled> Kopecký, M. </controlled>
  </project>
  <employee>
    <name> Dvorský, J. </name>
    <PIN> 700321/1423 </PIN>
    <age> 29 < /age>
  </employee>
  <employee>
    <name> Mikulová,L. </name>
    <PIN> 715512/0132 </PIN>
    <age> 38 </age>
  </employee>
  <project>
    <p_name> Sorting </p_name>
    <budget> 700000 </budget>
    <controlled> Mikulová,L.
</controlled>
  </project>
  :
</db>
```

Relations projects and employees in XML

employees are "behind" projects

```
<db>
  <projects>
    <project>
        <p_name> Searching </p_name>
        <budget> 100000 </budget>
        <controlled> Kopecký, M. </controlled>
    </project>
    <project>
        <p_name> Sorting </p_name>
        <budget> 700000 </budget>
        <controlled> Mikulová, L. </controlled>
    </project>
   :
  </projects>
  <employees>
    <employee>
        <name> Kopecký, M. </name>
        <PIN> 640802/3200</PIN>
        <age> 35 </age>
    </employee>
    <employee>
        <name> Mikulová,L. </name>
        <PIN> 715512/0132 </PIN>
        <age>38 </age>
    </employee>
   :
  </employees>
</db>
```

Relations projects and employees in XML

or wihout “separator” tag …

```
<db>
  <projects>
    <p_name> Searching </p_name>
    <budget> 100000 </budget>
    <controlled> Kopecký, M. </controlled>
    <p_name> Sorting </p_name>
    <budget> 700000 </budget>
    <controlled> Mikulová,L </controlled>
    :
  </projects>
  <employees>
    <name> Kopecký, M. </name>
    <PIN> 640802/3200 </PIN>
    <age> 35 </age>
    <name> Mikulová, L</name>
    <PIN> 715512/0132 </PIN>
    <age> 38 </age>
    :
  </employees>
</db>
```

More about attributes

```
<db>
  <film id="f1">
    <title>Turbína</title>
    <director>Novak A.</director>
    <cast idrefs="h1 h2"></cast>
    <budget>100000</budget>
  </film>
  <film id="f2">
    <title>Batalion</title>
    <director>Buřita S.</director>
    <cast idrefs="h2 h9 h21"></cast>
    <budget>110000</budget>
  </film>
  <film id="f3">
    <title>Gabriela</title>
    <director>Vrchota J.</director>
    <cast idrefs="h1 h8"></cast>
    <budget>90000</budget>
  </film>
  …
  <actor id="h1">
    <name>M. Glázrová</name>
    <playing_in idrefs="f1 f3 f78" >
    </playing_in>
  </actor>
  <actor id="h2">
    <name>K. Höger</name>
    <playing_in idrefs="f1 f2 f11">
    </playing_in>
    <age>38</age>
  </actor>
  <actor id="h3">
    <name>H. Vítová</name>
    <playing_in idrefs="f2 f35">
    </playing_in>
  </actor>
  :
</db>
```

When to use attributes

It is not always clear, when to use attributes.

Attributes are not "seen".

```
<person PIN= “780730/0013”>
  <name> J. Black </name>
  <email>
    black@ksi.mff.cuni.cz
  </email>
  ...
</person>
```

```
<person>
  <PIN> 780730/0013 </PIN>
  <name> J. Black </name>
  <email>
      black@ksi.mff.cuni.cz
  </email>
  ...
</person>
```

Document conforming to the rule “nesting tags” and not having same attributes in its start tag, is called well-formed.

Part III: XML data model

- Description of document type with DTD
- Association to the object data model

Document type description via DTD

- Document Type Descriptors (DTDs) assign to XML documents a structure.
- there is *certain* relationship between DTD and a database schema,
- DTD is a *syntactic* specification.

Example: Personal address book

<person>

<name> Říha Antonín </name>

<with_title> Dr. A. Říha </ with_title>

<address>Malostranské 25 </address>

<address> Praha, 100 00 </address>

<phone> 2191 4268 </phone>

<fax> 2191 4323 </fax>

<phone> 2191 4323 </phone>

<email> riha@ksi.mff.cuni.cz</email>

</person>

exactly one name

max. 1

as many rows for addresses as needed

mixed phones and faxes

as many as needed

Structure specification

- name — specifies a name element
- with_title? — specifies optional (0 or 1) elements with_title
- name, with_title? — specifies name followed optionally by with_title
- address* — specifies 0 or more address elements
- phone | fax — phone *or* fax element
- (phone | fax)* — 0 or more phone or fax elements
- email* — 0 or more email elements

Structure specification (cont.)

The whole structure of the person element is specified as

name, with_title?, address*, (phone | fax)*, email*

It's a regular expression. Why is it important?

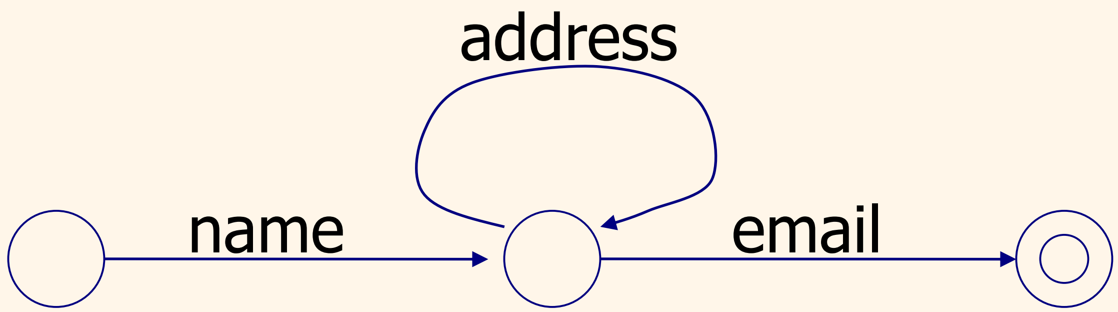
Regular expressions

Each regular expression determines a corresponding *finite automaton.*

Ex.:

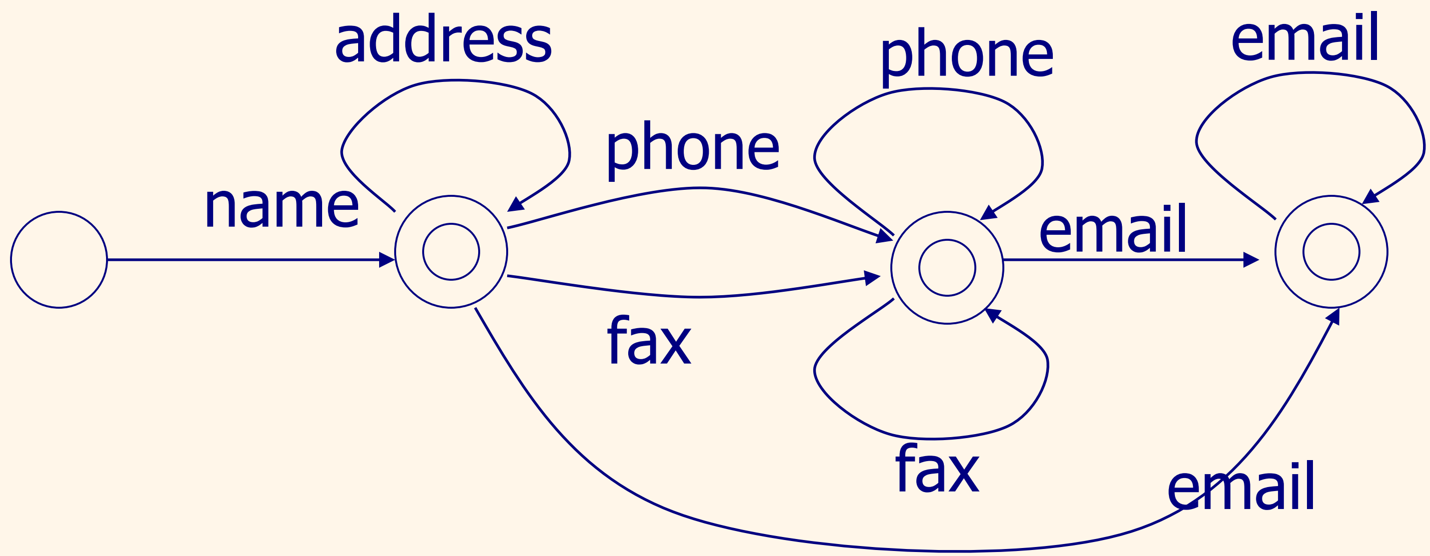
name, address*, email

Corresponding simple program (parser)



Another example

name, address*, (phone | fax)*, email*



Adding optional with_title leads to complications with the size of the automaton

DTD for address book

```
<!DOCTYPE address book [
  <!ELEMENT address book (person*)>
  <!ELEMENT person
      (name, with_title?, address*, (fax | phone)*, email*)>
  <!ELEMENT name       (#PCDATA)>
  <!ELEMENT with_title    (#PCDATA)>
  <!ELEMENT address (#PCDATA)>
  <!ELEMENT phone      (#PCDATA)>
  <!ELEMENT fax        (#PCDATA)>
  <!ELEMENT email      (#PCDATA)>
]>
```

Revised relational DB

projects:

p_name	budget	controlled

employees:

name	PIN	age

Two DTDs for relational DB

```
<!DOCTYPE db [
  <!ELEMENT db          (projects,employees)>
  <!ELEMENT projects    (project*)>
  <!ELEMENT employees   (employee*)>
  <!ELEMENT project     (p_name, budget, controlled)>
  <!ELEMENT employee    (name, PIN, age)>
  ...
]>
```

```
<!DOCTYPE db [
  <!ELEMENT db          (project | employee)*>
  <!ELEMENT project     (p_name, budget, controlled)>
  <!ELEMENT employee    (name, PIN, age)>
  ...
]>
```

Recursive DTD

```
<!DOCTYPE genealogy [
    <!ELEMENT genealogy (person*)>
    <!ELEMENT person (
                 name,
                 birthday,
                 person,                    -- mother
                 person  )>                 -- father
    ...
]>
```

Where is a problem? Parents are mandatory. Order.

Recursive DTD (cont.)

```
<DOCTYPE genealogy [
    <!ELEMENT genealogy (person*)>
    <!ELEMENT person (
                    name,
                    birthday,
                    person?,                -- mother
                    person? )>              -- father
    ...
]>
```

Where is the problem now? Order.

Better solution: with ID, IDREF, IDREFS

Some things are difficult to specify

Each employee element contains elements name, age and PIN in any order.

```
<!ELEMENT employee
    ( (name, age, PIN) | (age, PIN, name) |
     (PIN, name, age) | ...
    )>
```

Suppose a situation, when there are more attributes of employees!

Regular expressions in XML

▪ A	tag A occurs
▪ e1,e2	expression e1 followed by e2
▪ e*	0 or more e occurrences
▪ e?	optional -- 0 or 1 occurrences
▪ e+	1 or more occurrences
▪ e1 | e2	either e1 or e2
▪ (e)	grouping

Specification of attributes in DTD

```
<!ELEMENT height (#PCDATA)>
<!ATTLIST height
      dimension CDATA #REQUIRED
      accuracy CDATA #IMPLIED >
```

Attribute dimension is required; attribute accuracy is optional.

CDATA is character data, not usually parsed.

Specification of attributes ID and IDREF

```
<!DOCTYPE family [
  <!ELEMENT family   (person)*>
  <!ELEMENT person  (name)>
  <!ELEMENT name     (#PCDATA)>
  <!ATTLIST person
                id          ID          #REQUIRED
                mother IDREF    #IMPLIED
                father    IDREF    #IMPLIED
                children IDREFS  #IMPLIED>
]>
```

Well-formed document having DTD and conforms to the DTD is called valid.

Some valid data

```
<family>
    <person  id=„jane"  mother="marie" father="josef">
        <name> Jane Novak </name>
    </person>
    <person id="josef" childern=„jane vít">
        <name> Josef Novak </name>
    </person>
    <person id="marie" childern=„jane  vít">
        <name> Marie Novak </name>
    </person>
        <person  id="vít"  mother=”marie" father="josef">
        <name> Vít Novak </name>
    </person>
</family>
```

Consistency of values of ID and IDREF(S) attributes

- If attribute is declared as ID
  - associated values must all be different in the document
- If attribute is declared as IDREF
  - associated value must exist as a value of an ID attribute (no „hanging pointers“) in the given document
- similarly for all values of IDREFS attribute
- ID, IDREF and IDREFS attributes *are not typed*

DTD vs. db schemes (or types)

- Comparing to db standards (or programming languages), DTDs are rather weak specifications.
  - only one basic type -- PCDATA
  - no useful “abstractions” (e.g., sets)
  - IDREF are not typed. They point to something, but they don’t know what!
  - no IC, e.g., child is inverse to parents
  - no methods
- Proposals how to extend XML: schemes, IC
  - XML Schema (proposal of W3C)
  - Microsoft in Explorer 5.
- Today’s popular JSON: „lightweight “ and more simple alternative to XML

Part IV: XPath - overview

XML data model in XPath

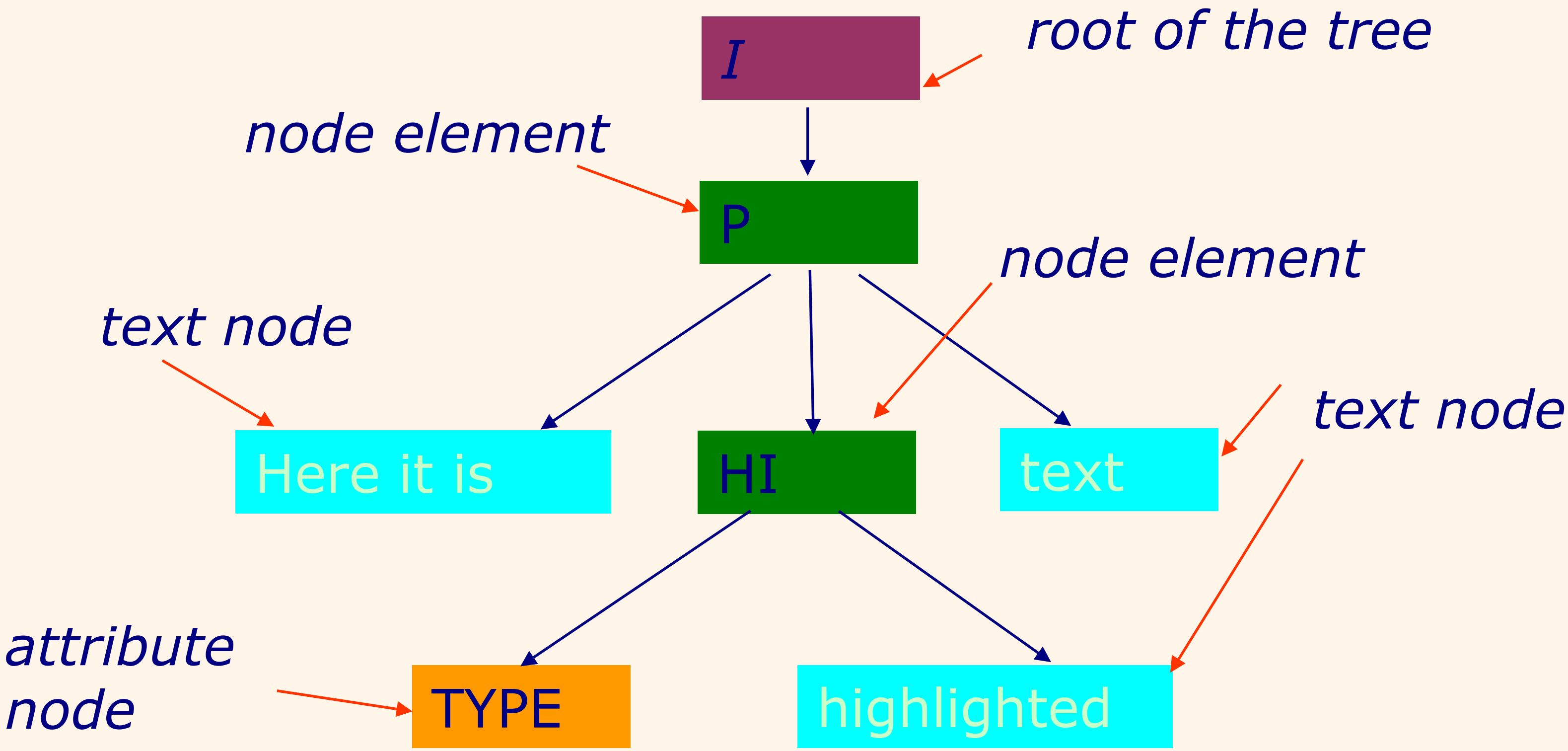
Node types in the model

- root node
- nodes elements
- text nodes,
- attribute nodes,
- nodes for comments
- nodes of processing instructions
- nodes name spaces

What is no there: section CDATA, references to entities and DTD

XML data model - example

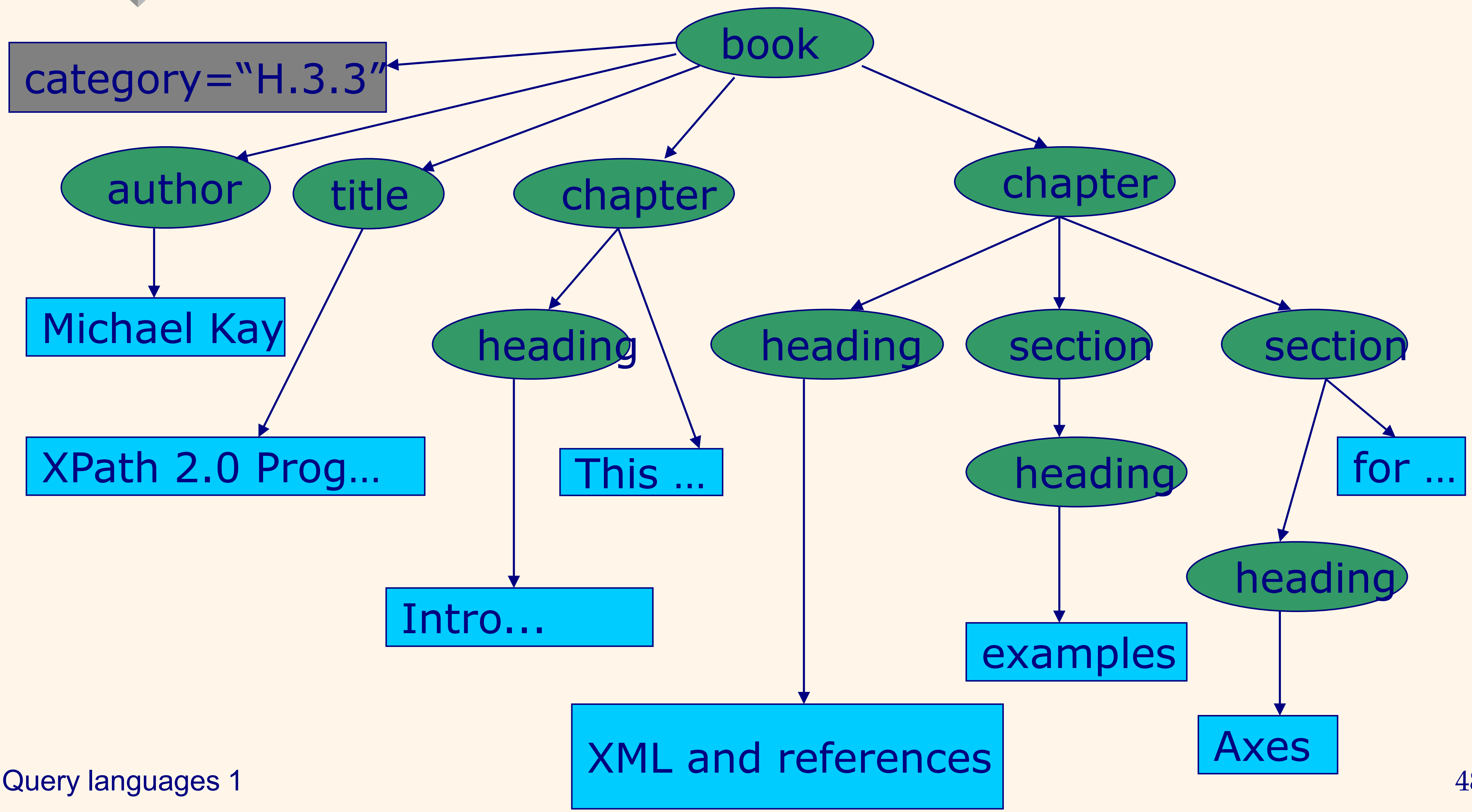
<P>Here it is a<HI TYPE=„ital‟>highlighted</HI>text.<P>



XPath expressions and their evaluation

- XPath expressions denote queries.
- the result of expression evaluation - possibilities:
  - a node set,
  - number,
  - Boolean value,
  - string

XML document



XPath – basic constructs

simple path specifies one step in navigation in db.

x/l

key notion: regular path expression

Ex.:	biblio/(report l article)	-- alternative
	author/first_name?	-- partial information
	report/reference*/author	-- Kleene closure

Remark: R+, where R is a regular expression, is equivalent to R/R*

Paths in XPath

- path, which starts with / represents absolute path, starting from the root of XML data
  - Ex.: /book
  - Remark: absolute path can select *more than one* element.
  - Remark: query: / selects “the whole document”.
- path, which does not start with / represents relative path starting from the current (context) element
  - Ex.: chapter/heading

    Remark: the result are all headings of chapters, that are descendants of the current node
- path starting // can start *anywhere* in document
  - Ex.: //heading selects each element heading, which occurs in document
  - Remark: expensive query

XPath axes

- Queries use various relations between nodes (axes in XPath):

X::Y means "select Y from axis X"

self – set of the nodes

  self::node() is the current node

ancestor – nodes lying on the path from *u* to the root,

ancestor-or-self – *u* and nodes lying on the path from *u* to the root,

parent – the first node lying on path from *u* to the root,

child – immediate descendants of the node *u*,

  /child::X is the same as /X

XPath axes

descendent - all nodes, for that is node *u* an ancestor,

descendent-or-self - *u* and all nodes, for that is node *u* an ancestor,

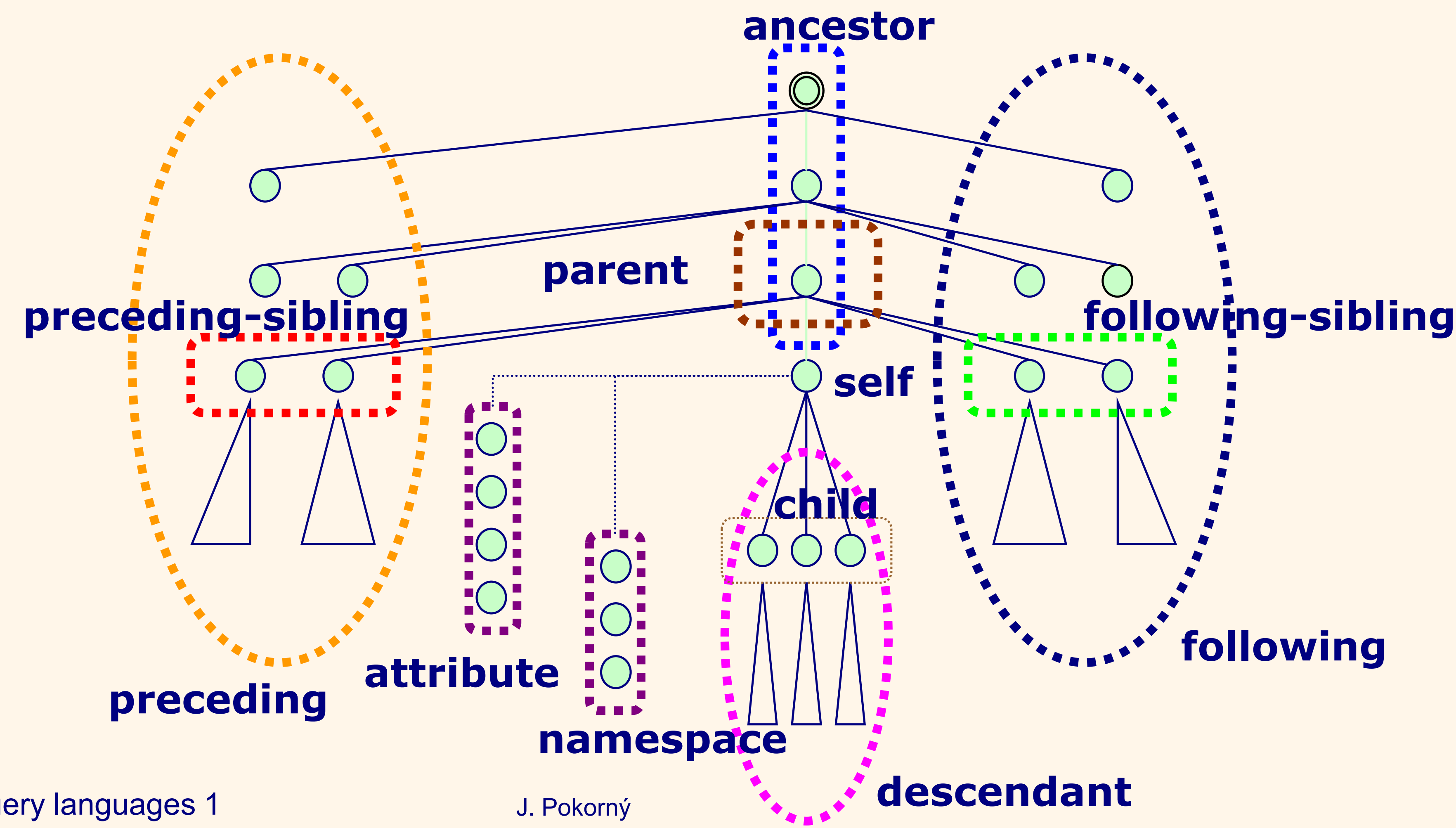
preceding-siblings – siblings of node *u* preceding *u* in preorder tree traversal,

following-siblings – siblings node *u* following *u* in preorder tree traversal,

preceding – nodes preceding *u* (except for its ancestors) in preorder tree traversal,

following – nodes following *u* (except for its descendants) in preorder tree traversal.

XPath axes


ancestor
parent
preceding-sibling
following-sibling
self
child
attribute
namespace
preceding
following
descendant

Axes - examples

- //book/descendant::* returns all descendants of every element book
- //book/descendant::chapter returns all chapter descendants of very element book
- //parent::* returns all elements, that are a parent of a node, i.e. tree leafs will not be in result
- //section/parent::* every parent of a section element
- //parent::chapter is each chapter element, which is a parent (i.e. has children)
- /library/book[3]/following::* everything, what is after the third book of the library

Abbreviations (syntactic suger) for axes

(nothing) corresponds to child::

@ corresponds to attribute::

. corresponds to self::node()

.//X corresponds to self::node()/descendant-or-self::node()/child::X

.. corresponds to parent::node()

../X corresponds to parent::node()/child::X

// corresponds to /descendant-or-self::node()/

//X corresponds to /descendant-or-self::node()/child::X

XPath – query examples

- In the most of queries their path is based on the children axis
- Examples of queries:

/article/*/paragraph

article//figure

//article[author='Michael Kay']

More complex:

//article[title = 'XPath 2.0 Programming']/author

article[author]//name -- requires a sibling

Which query does this expression express?

//figure/ancestor::chapter/following-sibling::chapter

?

XPath – query examples

//figure/ancestor::chapter/following-sibling::chapter

The answer:

the chapters, following (with the same superelement) any chapter containing a figure

XPath – more about semantics

- Simple path (step) is evaluated w.r.t. a context.

context consists of:

  - context node,
  - position of the node in context and the context scope (the number of nodes),
  - bind variables, library functions, name space declarations

- simple path has a form: axis::node-test[predicate]
  - axis selects a set of nodes-candidates (e.g. children),
  - node-test filters candidates, based on the node type and the name (name elements,…),
  - predicate (Boolean expression) further filters nodes,
  - the rest goes into the result.

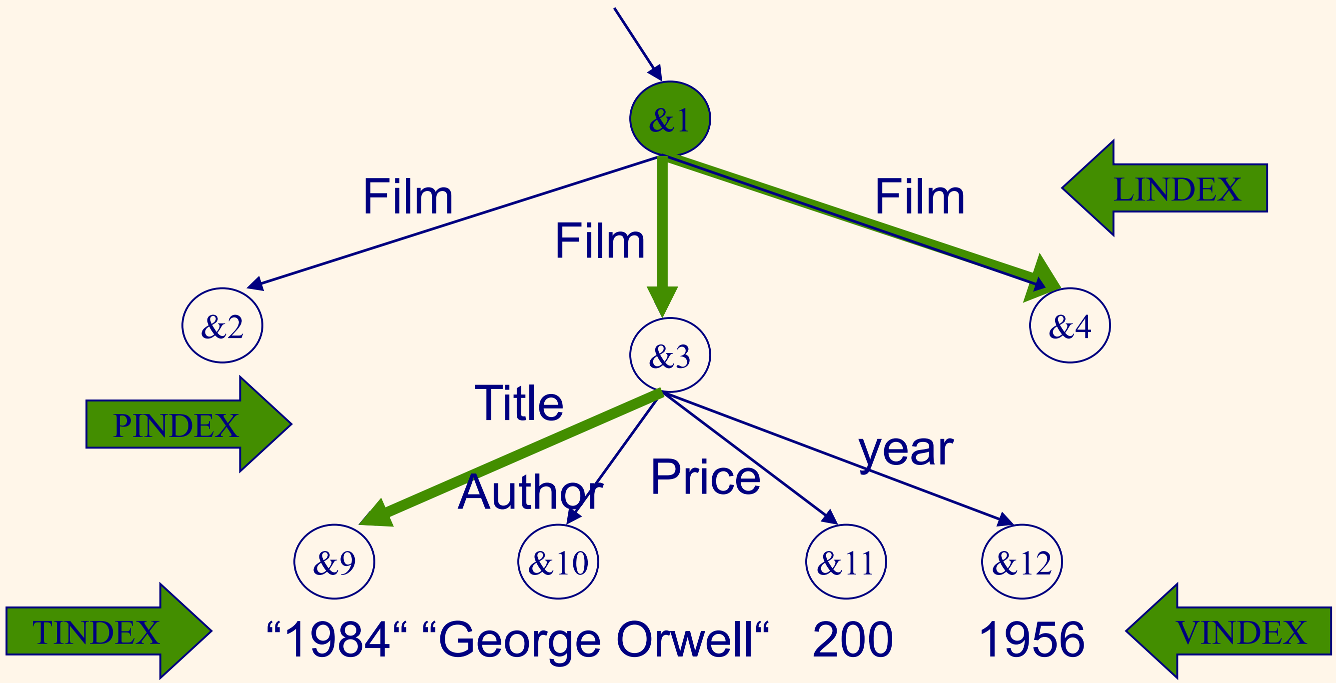
Part V: Indexing XML data

Methods for indexing XML data

- indexing as a fulltext

  disadvantage: querying by structure is not possible
- indexing relations in a classical way (Lore)
- indexing based on positions
  - using absolute or relative addresses for representation of words and tags positions in XML document
- indexing based on paths
  - paths are encoded according to a tree-traversal order
  - all paths leading to all words are encoded
  - It is possible to query both a content and structure

Indexing in Lore



See: http://infolab.stanford.edu/lore/

Value Index (VINDEX)

- Input: tag T, comparison Θ, value v
- Output: all atomic objects having incoming edge T and value v' satisfying Θ and value v.

  Ex.: (Price, >, 150)

  Result is {&11, &15}
- Vindex can be implemented, e.g., with B+-trees;

Link Index (LINDEX)

- Input: oid o and tag T
- Output: all parent objects having edge T incoming to o.
  - If T is omitted, all parents and their tags are returned.

  Ex.: retrieve a parent with Lindex for object &4 via edge labeled Film;

  It returns parent object &1
- Lindex can be implemented, e.g., with linear hashing

Link Index (LINDEX)

Ex.: db/A/B[C=5]

Uses Vindex and Lindex:

- Find C component via Vindex and (C, =, 5)
- Try, whether there is a path A/B from db to this object via two calling Lindex.
- Return evaluation.

Text Index (TINDEX)

- Input: TINDEX provides searching using a keyword w in form (w, T), where T is a tag.
- Output : list of postings <o, n>
- Can be implemented via inverted lists, mapping word w and tag T to a list of atomic values v with input edge T, where v contains w on position n.
- Tag can be omitted.

Ex.: Look up with TINDEX for all objects containing word “Ford“ and having incoming edge Name.

Result:{<&17, 1><&21, 2>}

Path Index (PINDEX)

- Input: object o and expression p denoting a path
- Output : all objects reachable from o via the path p
- Restriction: usually only simple paths, i.e. those starting in named objects and containing no regular expressions

Ex.: db/Film/Title

Pindex for retrieving all objects reachable via db/Film/Title

Result: {&5, &9, &14}

Evaluating top-down directly

Ex.: db.Film[Price < 200]

- All subelements of the Film element in db are searched and for each look up, the content of subelement Price is tested if its value is less than 200.
- This leads to depth-first traversal of the tree matching edges, which occur in path expressions.

Evaluating bottom-up with indexes

Ex.: db.Film[Price < 200]

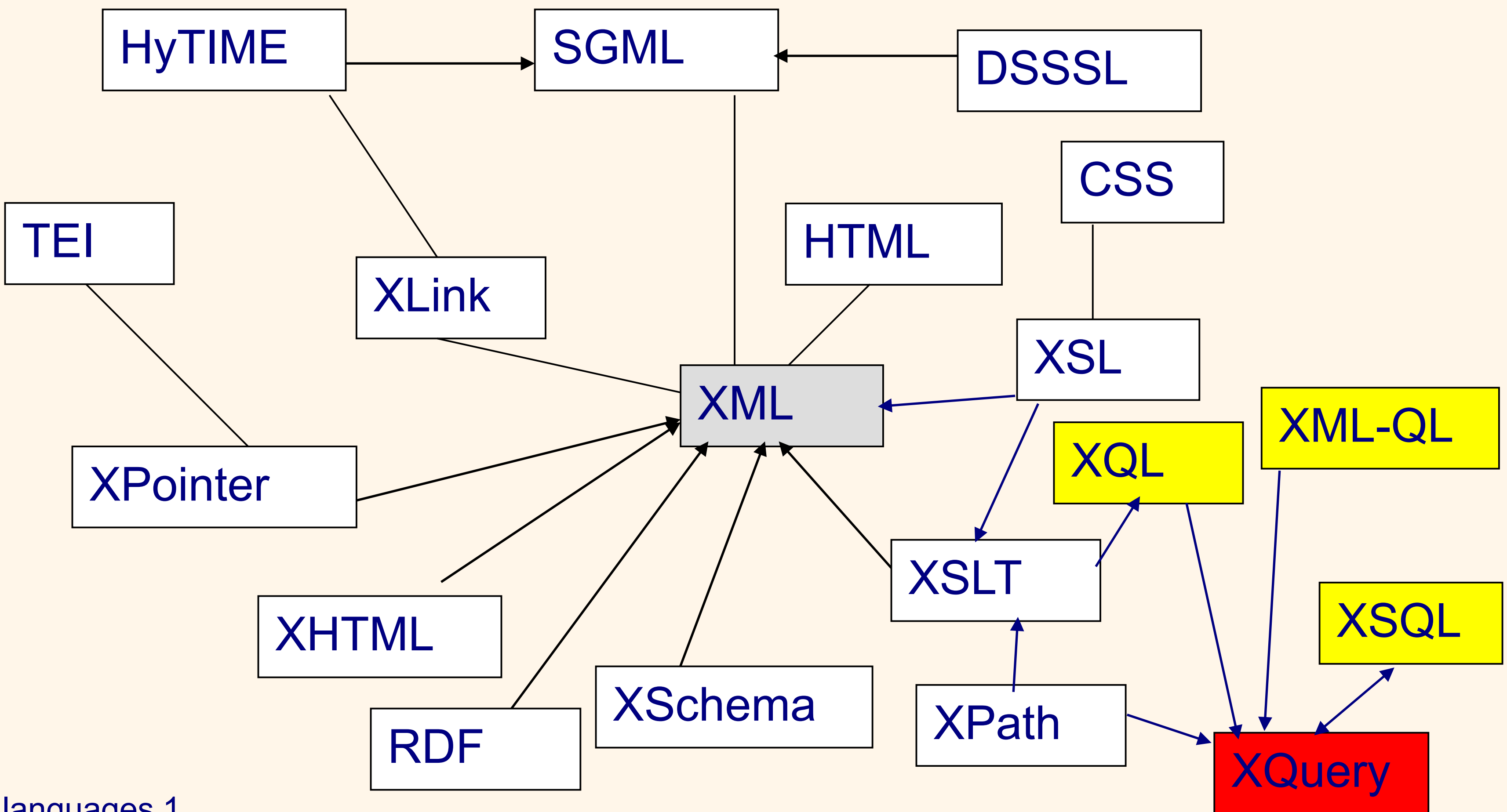
- First find all objects meeting the condition using an appropriate Vindex.
- For each from these objects traverse backward in the tree to their parents using Lindex.
- Advantage: avoids the paths, which do not meet the condition.

Hybrid evaluating with indexes

Ex.: db.Film[Price < 200]

- Something is evaluated (not necessarily everything) what concerns the condition by top-down approach.
- Then there are found directly the objects meeting the condition with Videx. Then it continues by traversing via Lindex to the same point as with top-down approach.
- Query result is found as an intersection set of the objects set and combination of traversing paths.

XML – standards family



Conclusion

- Indexes occur in native XML databases
- Proposed various types of indexes on XML data to execute efficiently XPath queries.
- XPath 2.0 is also a subset of XQuery 1.0.
- Indexes provide efficient support for processing queries in these languages.