DMKM ADB - Advanced Databases
XML Databases

Bernd Amann – 2014/2015

UPMC

September 18, 2015
Part I

XML Models and Languages
Outline

Introduction

XML Syntax

XML Programming

XML Namespaces

XML Schema Languages

Exercises
All About Shakespeare...

There exist all kind of resources about Shakespeare and his work:

- complete works,
- bibliographic references,
- critics,
- biographical data,
- theatres and festivals,
- historical background, ...
Hamlet

(c) Nicholas Ling and John Trundell, First (Bad) Quarto, 1605
Hamlet on Wikipedia (HTML)

Hamlet

From Wikipedia, the free encyclopedia

This article is about the Shakespeare play. For other uses, see Hamlet (disambiguation).

The Tragical History of Hamlet, Prince of Denmark, or more simply Hamlet, is a tragedy by William Shakespeare, believed to have been written between 1599 and 1601. The play, set in the Kingdom of Denmark, recounts how Prince Hamlet exacts revenge on his uncle Claudius, firstly for murdering the old King Hamlet (Claudius's brother and Prince Hamlet's father) and secondly for then succeeding to the throne and marrying Gertrude (the King Hamlet's widow and mother of Prince Hamlet). The play vividly portrays real and feigned madness – from overwhelming grief to seething rage – and explores themes of treachery, revenge, incest, and moral corruption.

Three different early versions of the play have survived; these are known as the First Quarto (Q1), the Second Quarto (Q2) and the First Folio (F1). Each has lines, and even scenes, that are missing from the others. Shakespeare based Hamlet on the legend of Amleth, preserved by 13th-century chronicler Saxo Grammaticus in his Gesta Danorum as subsequently retold by 16th-century scholar François de Belleforest. He may have also drawn on, or perhaps written, an earlier (hypothetical) Elizabethan play known today as the Ur-Hamlet.

The play's structure and depth of characterisation have inspired much critical scrutiny, of which one example is the centuries-old debate about Hamlet's hesitation to kill his uncle. Some see it as a plot device to prolong the action, and others see it as the result of pressure exerted by the complex philosophical and ethical issues that surround cold-blooded murder, calculated revenge and thwarted
Hamlet on DBPedia (RDF/Turtle)

@prefix dbpedia-owl: <http://dbpedia.org/ontology/> .
@prefix dbpedia: <http://dbpedia.org/resource/> .
@prefix rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#> .
@prefix rdfs: <http://www.w3.org/2000/01/rdf-schema#> .
@prefix yago: <http://dbpedia.org/class/yago/> .

dbpedia:Prince_Hamlet
    rdf:type yago:BadPerson109831962 ,
              yago:Prince110472799 ,
              yago:Criminal109977660 ,
              yago:Murderer110338707 ,
              yago:Leader109623038 ;

dcterms:subject category:Male_Shakespearean_characters ,
        category:Fictional_princes ;

rdfs:comment "Prince Hamlet is the title character ..."@en ;

dbpprop:family <http://dbpedia.org/resource/Ghost_(Hamlet)> ,
    dbpedia:King_Claudius ,
    dbpedia:Gertrude_(Hamlet) ;

dbpprop:play "Hamlet"@en ;

dbpprop:role "Prince of Denmark"@en ;

dbpprop:creator dbpedia:William_Shakespeare ;

dbpprop:quote "\"To be, or not to be, that is the question\""@en .
Some Observations

Rich information sources

- Shakespeare’s work: comedies, tragedies, poems
- Complementary documents: critics, biographies, ...
- Facts and knowledge: classification, facts, statistics, ...

Rich queries over contents and facts

- **Who** said "to be or not to be" in which play?
- **When** did Shakespeare write his first tragedy?
- Find all **tragedies** and **comedies** written by Shakespeare
- **Where** can I watch a Shakespeare play next week?

How to collect, store, integrate and query all this information?
XML

Why XML?

- universal format for information exchange
- expressive models and languages for information integrating and querying
- efficient data structures and algorithms for information processing

Why not JSON (a popular noSQL format)?

- JSON is more “light-weight”:
  - parsing: less complex type system,
  - processing: “natural” integration in python
  - size: file (not necessary memory)
- but:
  - XML includes expressive declarative languages for defining schemas, queries and transformations
- both apply similar (noSQL) concepts:
  - flexible structure
  - self-describing (markup, keys)
  - data exchange
- direct mapping between XML and JSON
Hamlet in XML

<PLAY>
<TITLE>The Tragedy of Hamlet, Prince of Denmark</TITLE>
<FM>
<P>This work may be freely copied and distributed worldwide.</P>
</FM>
<PERS
<TITLE>Dramatis Personae</TITLE>
<PERS CLAUDIUS, king of Denmark. </PERS HAMLET, son to the late, and nephew to the present king. </PERS POLONIUS, lord chamberlain. </PERS>
<PERS GROUP>
<PERS VOLTIMAND</PERS CORNELIUS</PERS ROSENCRANTZ</PERS GUILDENSTERN</PERS
<GRPDESC courtiers.</GRPDESC>
</PERS GROUP>
<PERS FRANCISCO, a soldier.</PERS>
</PERS
<SCNDESCR>SCENE Denmark.</SCNDESCR>
<PLAYSUBT>HAMLET</PLAYSUBT>
Hamlet in XML

<ACT>
    <TITLE>ACT I</TITLE>
    <SCENE>
        <TITLE>SCENE I. Elsinore. A platform before the castle.</TITLE>
        <STAGEDIR>FRANCISCO at his post. Enter to him BERNARDO</STAGEDIR>
        <SPEECH>
            <SPEAKER>BERNARDO</SPEAKER>
            <LINE>Who’s there?</LINE>
        </SPEECH>
        <SPEECH>
            <SPEAKER>FRANCISCO</SPEAKER>
            <LINE>Nay, answer me: stand, and unfold yourself.</LINE>
        </SPEECH>
        <SPEECH>
            <SPEAKER>BERNARDO</SPEAKER>
            <LINE>Long live the king!</LINE>
        </SPEECH>
        <SPEECH>
            <SPEAKER>FRANCISCO</SPEAKER>
            <LINE>Bernardo?</LINE>
        </SPEECH>
        <SPEECH>
            <SPEAKER>BERNARDO</SPEAKER>
        </SPEECH>
    </SCENE>
</ACT>
</PLAY>
Hamlet (JSON)

```json
{
    "PLAY": {
        "TITLE": "The Tragedy of Hamlet, Prince of Denmark",
        "FM": { "P": "This work may be freely copied and distributed worldwide." },
        "PERSONAE": {
            "TITLE": "Dramatis Personae",
            "PERSONA": [
                "CLAUDIUS, king of Denmark. ",
                "HAMLET, son to the late, and nephew to the present king. ",
                "POLONIUS, lord chamberlain. ",
                "FRANCISCO, a soldier."
            ],
            "PGROUP": {
                "PERSONA": [
                    "VOLTIMAND",
                    "CORNELIUS",
                    "ROSENCRANTZ",
                    "GUILDENSTERN"
                ],
                "GRPDESCR": "courtiers."
            }
        },
        "SCNDESCR": "SCENE Denmark."
    }
}
```

"PLAYSUBT": "HAMLET",
"PLAYSUBT": "HAMLET",
Hamlet (JSON)

"ACT": {
  "TITLE": "ACT I",
  "SCENE": {
    "TITLE": "SCENE I. Elsinore. A platform before the castle."
    "STAGEDIR": "FRANCISCO at his post. Enter to him BERNARDO",
    "SPEECH": [
      {
        "SPEAKER": "BERNARDO",
        "LINE": "Who’s there?"
      },
      {
        "SPEAKER": "FRANCISCO",
        "LINE": "Nay, answer me: stand, and unfold yourself."
      },
      {
        "SPEAKER": "BERNARDO",
        "LINE": "Long live the king!"
      },
      {
        "SPEAKER": "FRANCISCO",
        "LINE": "Bernardo?"
      }
    ]
  }
}
XML and Databases

- XML is a set of technology independent standards for representing and accessing structured information mixing text and data.
- XML does not define how data is stored and processed.
- Databases provide the technology for processing large collections of structured data.

It must be an answer of most monstrous size that must fit all demands. (Countess in "All's Well That Ends Well")
Course Objective

- Understand the role of XML for modelling and processing (querying) structured information mixing text and data.
- Understand how standard database technology can deal with XML data.
- Understand the more fundamental problem of querying and storing heterogeneous semi-structured data.
Course Outline

1. Course 1: Syntax and Schemas
   - XML Syntax
   - XML Programming
   - XML Schema Languages

2. Course 2: Query Languages
   - XML Query Languages
   - XPath 1.0 / 2.0 and XQuery 1.0
   - XQuery Update

3. Course 3: Storage and Query Evaluation
   - XML Storage
   - XPath Evaluation
   - XQuery Evaluation
Outline

Introduction

XML Syntax

XML Programming

XML Namespaces

XML Schema Languages

Exercises
XML is Syntax

- Ancient Greek:
  - συν (syn): "together"
  - τάξις (taxis): "an ordering"

- Linguistics:
  - study of the principles and rules for constructing phrases and sentences in natural languages.

- XML:
  - Set of rules for representing structured information.
Structuring markup

From character sequences to structured documents

An XML document is a text contents structured and annotated by markup symbols.

Structuring markup

- Start element: <A>, <a>, <b>, ...
- End element: </A>, </b>, ...
- Attributes a='v', B="12456", c3="12456", ...

Structural Constraints

1. Element markup must be nested:
   - <A><b></b></A> : well-formed
   - <A><b></A></b> : not well-formed

2. Attribute / value markup must appear inside start element markup: <A a='v'>, <B a='v' c3="12456">, ...

3. Text can appear anywhere between element symbols
Example (XML)

```xml
<?xml version='1.0' encoding='iso-8859-1' ?>
<program xmlns="http://dmkm.org/">
  <!-- the first cinema -->
  <cinema>
    <name>St. André des Arts</name>
    <address>
      <city>Paris</city>
      <street>rue St. André des Arts</street>
      <number>13</number>
    </address>
    <showtime hour='18:00' movie='#13'/>
    <showtime hour='20:00' movie='#14'/>
  </cinema>
  <movie movie_id='13'>
    <title>Brazil</title>
    <year>1986</year>
  </movie>
  <movie movie_id='14'>
    <title>The Godfather</title>
    <year>1972</year>
  </movie>
</program>
```
XML and JSON

```json
{ "program": {
    "#text": [ "Cinema information", "Information about 'Brazil'" ],
    "cinema": { "name": "St. André des Arts",
                  "address": {
                      "city": "Paris",
                      "street": "street St. André des Arts",
                      "number": "13"
                  },
    "showtime": [
        { "-hour": "18:00", "-ref_movie": "13" },
        { "-hour": "20:00", "-ref_movie": "14" }
    ],
    "movie": {
        "-movie_id": "13",
        "title": "Brazil", "year": "1986"
    }
}
```

```json
{ "program": {
    "#text": [ "Cinema information", "Information about 'Brazil'" ],
    "cinema": { "name": "St. André des Arts",
                  "address": {
                      "city": "Paris",
                      "street": "street St. André des Arts",
                      "number": "13"
                  },
    "showtime": [
        { "-hour": "18:00", "-ref_movie": "13" },
        { "-hour": "20:00", "-ref_movie": "14" }
    ],
    "movie": {
        "-movie_id": "14",
        "title": "The Godfather", "year": "1972"
    }
}
```
Annotation markup
Additional markup for facilitating the processing of XML documents.

Comments
► <!- - comment - ->

Processing instructions
Useful information for processing the document:
► <?instr arg ?>
  ► instr : processing instruction / context
  ► arguments : sequence of arg='value' expressions

Other
► entities: &lt;, &amp;, &quot;, ...
► CDATA sections : <![CDATA[ May use <, >, and & ]]>
Document Object Model (DOM)

DOM (Document Object Model)

- main memory representation of the document tree
- *global processing with random access* on the whole document

DOM tree
A DOM tree is a collection of encapsulated complex objects organized in an ordered tree.

DOM parser
A DOM parser transforms a set of binary XML files into a DOM tree.
XML Document : DOM Tree
DOM Object Types

- Node
  - Attribute
  - TreeNode
  - Leaf
  - Notation
  - Character Data
  - Processing Instruction
  - Entity
  - Document
  - Entity Reference
  - Element
  - Document Type
  - Comment
  - Text
  - CData Section
DOM programming in Python

```
cd PROGRAMMES
py exedom.py
```
XML Document : SAX events

SAX *(Simple API for XML)*

- sequence of events generated by the *preorder traversal* of the document (DOM) tree
- *local processing* with sequential access to document fragments

SAX Event Stream
Stream of events generated by the markup symbols in document order.

SAX parser
decodes a set of binary XML files and generates a SAX event
- at the beginning and the end of the document,
- before and after each markup symbol.
SAX programming

Associate callback functions to markup events:

- Callback function are executed independently and can share memory.
SAX programming in Python

cd PROGRAMMES
py exesax.py
DOM and SAX: Pros and Cons

<table>
<thead>
<tr>
<th></th>
<th>Advantage</th>
<th>Drawback</th>
</tr>
</thead>
<tbody>
<tr>
<td>DOM</td>
<td>random access</td>
<td>memory bottleneck</td>
</tr>
<tr>
<td></td>
<td>complex manipulations</td>
<td>swapping, out-of-memory</td>
</tr>
<tr>
<td>SAX</td>
<td>constant memory usage</td>
<td>sequential access</td>
</tr>
<tr>
<td></td>
<td>process large documents</td>
<td></td>
</tr>
</tbody>
</table>

Both solutions are inappropriate for providing random access to large XML documents (document collections). ⇒ need for Database Technology
Outline

Introduction

XML Syntax

XML Programming

XML Namespaces

XML Schema Languages

Exercises
Namespaces

Objective
Avoid naming conflicts when merging documents (modularity).

Definition
A namespace is an abstract collection (vocabulary) of element and attribute names identified by a URI.

Example
NY Times and Washington Post use element name rating for rating movies. Timemout includes these rating in a new document. We have three vocabularies (namespaces):

```xml
<reviews xmlns='http://www.timeout.com/
    xmlns:nyt='http://www.nytimes.com/
    xmlns:wp='http://www.washingtonpost.com'/>
    <movie>Drive</movie>
    <nyt:rating>3.5</nyt:rating>
    <wp:rating>4.0</wp:rating>
</reviews>
```
Qualified and Universal Names

The previous example defines

- a default namespace http://www.timeout.com/
- a prefix `nyt` for namespace http://www.nytimes.com/ and

<table>
<thead>
<tr>
<th>Qualified name</th>
<th>Universal name</th>
</tr>
</thead>
<tbody>
<tr>
<td>reviews</td>
<td><a href="http://www.timeout.com/reviews">http://www.timeout.com/reviews</a></td>
</tr>
<tr>
<td>nyt:rating</td>
<td><a href="http://www.nytimes.com/rating">http://www.nytimes.com/rating</a></td>
</tr>
<tr>
<td>wp:rating</td>
<td><a href="http://www.washingtonpost.com/rating">http://www.washingtonpost.com/rating</a></td>
</tr>
</tbody>
</table>
Namespace identification rules

Given an

- element name A:
  - find the nearest ancestor element with some attribute xmlns='uri' → A is in namespace uri.
  - if this attribute does not exist → A is in no namespace.
- attribute name A → A is in no namespace.
- element or attribute name pre:A:
  - find the nearest ancestor element with an attribute xmlns:pre='uri' → A is in namespace uri.
  - if this attribute does not exist → raise an error
Example

```xml
<?xml version="1.0"?>
<A>
  <B/>
  <C xmlns="http://a.b.c"/>
    <D/>
  </C>
</A>
```
Example

```xml
<?xml version="1.0"?>
<p:A xmlns:p="http://a.b.c/">
  <B attr1="val1" p:attr2="val2"/>
  <p:C/>
</p:A>
```

```
Document
  Element A
    http://a.b.c/
    Element B
      Attr attr1
        val1
      Attr attr2
        http://a.b.c/
          val2
    Element C
      http://a.b.c/

Exercises
```
Example

```xml
<?xml version="1.0"?>
<A xmlns="http://a.b.c/"
    xmlns:"">
  <B/>
  <B xmlns=""/>
  <C/>
</B>
</A>
```
Example

```xml
<?xml version="1.0" ?>
<A xmlns:p="http://a.b.c/"
    xmlns:q="http://a.b.c/">
    <p:B/>
    <q:B/>
</A>
```
Example

```xml
<?xml version="1.0" ?>
<A xmlns="http://a.b.c/
    xmlns:vide=""
    >
    <B/>
    <vide:B>
      <C/>
      </vide:B>
    </A>
```
5 - XML Schema Languages

Motivation and application examples

DTD : Document Type Definition
From DTDs to Tree Grammars
Databases and Documents

Databases
Separation between database schema and database instance:

- Database schema describes the data structures and constraints
- Database instance “contains” the data conform to a schema

XML documents
The structure is part of the instance (markup):

- self-descriptive contents
- we can extract a “schema” from each document a-posteriori

Then, why define schemas for XML documents?
What are schemas for XML documents useful for?

- **Storage optimisation:**
  - What is the best way to store a document $d$ with schema $S$?
  - Use $S$ to define adapted (optimal) storage for $d$.

- **Query optimisation:**
  - What is the best execution plan of some query $Q$ for some document $d$ with schema $S$?
  - Use $S$ to rewrite $Q$.

- **Data exchange:**
  - Can some service (application, program) with input schema $S'$ handle all document generated by service with output schema $S$?
  - Check if $\text{sat}(S) \subseteq \text{sat}(S')$ where $\text{sat}(S)$ be the set of documents *validated* by some schema $S$. 
Motivation and application examples

**DTD : Document Type Definition**

From DTDs to Tree Grammars
Example

```xml
<?xml version='1.0' encoding='iso-8859-1' ?>
<program> Cinema information
  <cinema>
    <name> St. André des Arts </name>
    <address city='Paris'>
      <street> street St. André des Arts </street>
      <number> 13 </number>
    </address>
    <showtime hour = '18:00' ref_movie = '13' />
    <showtime hour = '20:00' ref_movie = '14' />
  </cinema>
  Information about 'Brazil' :
  <movie movie_id = '13' actors = '156 158' >
    <title> Brazil </title>
    <year> 1986 </year>
  </movie>
</program>
```
Element types

An element type is defined by a name and a contents model.

Contents model

- A regular expression over the alphabet of element names;
- EMPTY : the empty element;
- ANY : any combination of elements;
- #PCDATA : any text
- mixed content: (#PCDATA | A | B ...)*

When mixing elements and text, we cannot put any restriction on the number and order of the elements.
DTD

Elements

<!ELEMENT program (#PCDATA | cinema | movie)*>
<!ELEMENT cinema (name, address, showtime*)>
<!ELEMENT name (#PCDATA) >
<!ELEMENT address (street, number?)>
<!ELEMENT showtime EMPTY>
<!ELEMENT movie (title, year) >
<!ELEMENT title (#PCDATA) >
<!ELEMENT year (#PCDATA) >
<!ELEMENT number (#PCDATA) >
<!ELEMENT street (#PCDATA) >
Regular expressions (a reminder)

Definition
Given some alphabet \( N \):

- Any symbol \( n \in N \) is a regular expression;
- If \( e \) is a regular expression, then \( (e)^* \), \( (e)^+ \) and \( (e)^? \) are regular expressions;
- If \( e_1 \) and \( e_2 \) are regular expressions, then \( (e_1, e_2) \) and \( (e_1 | e_2) \) are regular expressions;

Example

- \( (a,b,c) \)
- \( (a|b|c) \)
- \( (a,b^*,a) \)
- \( ((a,b^*),c)^+ | (a,b^*,a) \)
Regular Languages

Definition
Each regular expression $e$ over an alphabet $N$ defines a regular language $L(e)$ over $N$:

- $L(a) = \{ a \}$ for all $a$ in $N$;
- $L(e?) = L(e) \cup \{ \epsilon \}$: $\epsilon$ denotes the empty word;
- $L(e_1, e_2) = \{ m_1 m_2 \mid m_1 \in L(e_1) \land m_2 \in L(e_2) \} = L(e_1) \times L(e_1)$: all concatenations of a word in $L(e_1)$ and a word in $L(e_2)$
- $L(e_1 | e_2) = L(e_1) \cup L(e_2)$: all words in $L(e_1)$ and in $L(e_2)$ (union)
- $L(e+) = \{ m_0 m_1 \ldots m_n \mid m_i \in L(e) \}$: all concatenations of words in $L(e)$ (infinite set);
- $L(e*) = L(e+) \cup \{ \epsilon \}$;
Regular Languages

Example

- \( L(a, b) = \{ab\} \), \( L(a \mid b) = \{a, b\} \)
- \( L(a*, b) = \{b, ab, aab, aaab, \ldots\} \)
- \( L((a, b)*) = \{\epsilon, ab, abab, ababab, \ldots\} \)
- \( L((a\mid b)*) = \{\epsilon, a, b, ab, ba, aaa, aab, aba, \ldots\} \)

DTD

```xml
<!ELEMENT cinema (name, address, showtime*)>
<!ELEMENT movie (title, year) >
```

The element name sequence name address showtime showtime of the children of element cinema is a word in the language of the regular expression (name, address, showtime*).
Example

```xml
<?xml version='1.0' encoding='iso-8859-1' ?>
<program> Cinema information
    <cinema>
        <name> St. André des Arts </name>
        <address city='Paris'>
            <street> street St. André des Arts </street>
            <number> 13 </number>
        </address>
        <showtime hour = '18:00' ref_movie = '13' />
        <showtime hour = '20:00' ref_movie = '14' />
    </cinema>
    Information about 'Brazil' :
    <movie movie_id = '13' actors = '156 158' >
        <title> Brazil </title>
        <year> 1986 </year>
    </movie>
</program>
```
Attribute types and modes

Syntax

<!ATTLIST element name type mode [default]>

Attribute types

- **CDATA** : character data (no markup)
- | : enumeration, sequence of values separated by |
- **ID, IDREF, IDREFS** : identifiers and references
- **NMTOKEN/NMTOKENS** : strings without space
- **ENTITY/ENTITIES/NOTATION** : for referencing external data (multimedia)

Attribute modes

- **#REQUIRED** : attribute is mandatory
- **#IMPLIED** : attribute is optional
- **#FIXED** : the attribute value is fixed
DTD Attributes

Example

```xml
<!ATTLIST showtime hour NMTOKEN #REQUIRED
  ref_movie IDREF #REQUIRED>
<!ATTLIST movie movie_id ID #REQUIRED
  actors IDREFS #IMPLIED
  language (AN|FR|AL|ES|IT) #IMPLIED>
<!ATTLIST address city CDATA #IMPLIED 'Paris'>
```
5 - XML Schema Languages

Motivation and application examples

DTD : Document Type Definition

From DTDs to Tree Grammars
DTD: Validation and Languages

```xml
<!ELEMENT a (b*cd*) >
<!ELEMENT c (e*) >
<!ELEMENT d (f(g*)) >
```

Validation

- `L(b*cd*)`
- `L(f(g*))`
- `L(e*)`

DTD (grammaire)

```
<!ELEMENT a (b*cd*) >
<!ELEMENT c (e*) >
<!ELEMENT d (f(g*)) >
```

Document (arbre)

```
a
  b b c d d
  |   
  v   v
  e e f f g
```

Exercises
DTD revisited

DTD = tree grammar

A DTD is a grammar \((E, S, R)\) where

- \(E\) is a set of element names.
- \(S\) is a subset of \(E\) called the root elements.
- \(P = \{ e \rightarrow r \}\) contains for each element \(e \in E\) a production rule \(e \rightarrow r\) where
  - \(r \subseteq E^*\) is a regular expression over \(E\)
  - \(r = \text{EMPTY}\)
  - \(r = \text{ANY}\)
  - \(r = \#\text{PCDATA}\)
  - \(r = (\#\text{PCDATA} | e_1 | e_2 | \ldots | e_n)^*\) where \(e_i \in E\).

Exemple

```xml
<!ELEMENT program (#PCDATA | cinema | movie)*>
<!ELEMENT cinema (name, address, (showtime)*)>
```

\(\text{program} \rightarrow (\#\text{PCDATA} | \text{cinema} | \text{movie})^*\)
\(\text{cinema} \rightarrow \text{name}, \text{address}, \text{showtime}^*\)
Tree Grammars

A tree grammar (or extended DTD) is a triple \((T, E, S, P)\) where

- \(E\) is a set of element names (terminals)
- \(T\) is a set of type names (non-terminals)
- \(S\) is a subset of \(T\) (root types),
- \(P = \{ t \rightarrow e\, r \}\) is a set of production rules where
  - \(t \in T\),
  - \(e \in E\) and
  - \(r \subseteq T^*\) is a regular expression over type names (contents model).
Extended DTD

DTD

```xml
<!ELEMENT program (#PCDATA | cinema | movie)*>  
<!ELEMENT cinema (name, address, (showtime)*)>
```

Regular Tree Grammar

\[ T = \{ TProg, TCine, TMovie, TName, PCDATA, EMPTY, \ldots \} \]
\[ E = \{ \text{program, cinema, movie, name, address, showtime} \} \]
\[ S = \{ TProg \} \]
\[ P = \{ \]

\[ TProg \rightarrow \text{program} (TCine|TMovie|Pdata) * \]
\[ TCine \rightarrow \text{cinema} (TName, TAddress, TShowtime*) \]
\[ TMovie \rightarrow \text{movie} (Title, Pays, Realisateur, Acteurs*) \]
\[ TName \rightarrow \text{name} (Pdata) \]

\[ ... \} \]
Regular tree languages

Regular tree language

Each regular tree grammar $G$ defines a regular tree language $sat(G)$ containing all instances of $G$.

Validation

A document $d$ is validated by a tree grammar $G$ if $d \in sat(G)$. 
DTD: regular expressions of element names

Document

```
section
title
para*
```

```
doc

section
title
para
para
```

```
section <- title para*
```

```
DTD
```

Motivation and application examples

From DTDs to Tree Grammars
Extended DTD: element/type mapping and regular expressions over type names
Document Validation

An XML document is a tree \( A = (N, child, \lambda, root) \) where

- \( N \) is a set of nodes
- \( child(n) \) returns for each node \( n \in N \) the list of its children
- \( \lambda(n) \) returns the name of \( n \)
- \( root \in N \) is the document root.

Example:

\[
\begin{align*}
\text{\(<A\>}
\text{\(<B\>}
\text{\(<C/>\)}
\text{\(<D/>\)}
\text{\(<B/>\)}
\text{\(<A/>\)}
\end{align*}
\]

\[
\begin{align*}
\text{\(N = \{a, b1, c, d, b2\}\)}}
\text{\(child = \{(a, b1), (a, b2), (b1, c), (b1, d)\}\)}}
\text{\(\lambda = \{(a, A), (b1, B), (b2, B), (c, C), (d, D)\}\)}}
\text{\(root = a\)}
\end{align*}
\]
Type Mappings and Instances

Given an XML document $A = (N, child, \lambda, root)$ and a tree grammar $G = (T, E, S, P)$:

**Type Mapping**

An type mapping $m : N \rightarrow T$ is a function returning for each node $n \in N$ a type $t \in T$.

**Grammar Instance**

A document $A = (N, child, \lambda, root)$ is an instance of $G = (T, E, S, P)$ if there exists an type mapping $m : N \rightarrow T$ such that:

- $m(root) \in S$,
- for all children $child(n) = n_0...n_k$ of a node $n$, there exists a rule $(t \rightarrow e r) \in P$ such that
  1. $m(n) = t$,
  2. $\lambda(n) = e$, and
  3. $m(n_0)...m(n_k)$ is in $L(r)$. 

1. $m(n) = t$,
2. $\lambda(n) = e$, and
3. $m(n_0)...m(n_k)$ is in $L(r)$. 

Validation: find a type mapping from the nodes in the document tree to the types in the tree grammar.

*There might exist several mappings.*
Regular Tree Grammar

Example

- \( T = \{ \text{Doc, Title, Section, Para1, Para2} \} \)
- \( E = \{ \text{doc, title, section, para} \} \)
- \( S = \{ \text{Doc} \} \)

\[
P = \{ \begin{align*}
\text{Section} & \rightarrow \ \text{section}(\text{Title, Para1, Para2}^*), \\
\text{Para1} & \rightarrow \ \text{para}(\text{Auteur, Pcdata}), \\
\text{Para2} & \rightarrow \ \text{para}(\text{Pcdata})
\end{align*} \}
\]

\[
P' = \{ \begin{align*}
\text{Section} & \rightarrow \ \text{section}(\text{Title, Para}^+), \\
\text{Para} & \rightarrow \ \text{para}(\text{Auteur, Pcdata}|(\text{Pcdata})
\end{align*} \}
\]

\( P \equiv P' \ ? \)

Observe that element \text{para} has two types!
Local Tree Grammars

Normalisation

- replace $t \rightarrow e\ r$ and $t \rightarrow e\ r'$ by $t \rightarrow e(r|r')$.
- if $t \rightarrow e\ r$ and $t' \rightarrow e\ r$, then remove $t' \rightarrow e\ r$ and replace $t'$ by $t$ in all other rules.

Concurrent Types

Two types types $A$ and $B$ are concurrent iff there exist two rules $A \rightarrow c\ r$ and $B \rightarrow c\ r'$ for the same element $c$ ($r \neq r'$).

Local Grammar

A local grammar is a grammar without concurrent types.

Validation

In a local grammar the type of an element is identified by the element name (proof is obvious).

- DTDs are local grammars.
Unique Type Grammars

Unique type Grammars

- the types in $S$ are not concurrent
- for each production rule $r$, the types in its content model are not concurrent.

Validation

In a unique type grammar the type of an element is identified by its name and the type of its parent (proof is obvious).

XML Schemas are unique type grammars
Unique Type Grammar

Example

\[
\begin{align*}
A & \rightarrow a(B1, C) \\
B1 & \rightarrow b(Pcdata) \\
C & \rightarrow c(B2^*) \\
B2 & \rightarrow b(D^*) \\
D & \rightarrow d(Pcdata)
\end{align*}
\]

- Unique type grammar, but not a local grammar.
- Cannot be expressed by a DTD.
XML Schema et Relax NG

- XML Schema: EXAMPLES/cinema.xsd
- Relax NG: EXAMPLES/cinema.rng
- Relax NG Compact Syntax: EXAMPLES/cinema.rnc
We can define three families of tree grammars with different expressive power.

Each family corresponds to one or several schema languages for XML.

- regular tree grammars
- unique type grammars
- local grammars

DSD, XDuce, Relax core
XML schema
DTD
Unranked Tree Automata

Theory “behind” XML document processing algorithms:

- Schema languages and validation
- Query languages: monadic Datalog, query automata
- Static type checking of XML programs
Unranked Tree Automata

An *unranked tree automaton* $A$ consists in:

- a finite alphabet of element symbols $\text{alphabet}(A)$,
- a finite set of states (types) $\text{states}(A)$,
- a finite set of transition rules $\text{rules}(A)$,
- a finite set of final states $\text{final}(a) \subseteq \text{states}(A)$.

where:

- $\text{rules}(A)$ are of the form $L \xrightarrow{a} q$ where $L$ is a regular language over $\text{states}(A)$, $a \in \text{alphabet}(A)$ and $q_i, q \in \text{states}(A)$ (bottom-up tree automata).
Unranked Tree Automata

Grammar:

\[ A \rightarrow a(B1, C) \]
\[ B1 \rightarrow b(Pcdata) \]
\[ C \rightarrow c(B2^*) \]
\[ B2 \rightarrow b(D^*) \]
\[ D \rightarrow d(PCDATA) \]

Rules:

\[ (q_{B1}, q_C) \xrightarrow{a} q_A \]
\[ q_{Pcdata} \xrightarrow{b} q_{B1} \]
\[ (q_{B2}^*) \xrightarrow{c} q_C \]
\[ (q_D^*) \xrightarrow{b} q_{B2} \]
\[ Pcdata \xrightarrow{d} q_D \]
Unranked Tree Automata

Classes of Tree Automata
Tree automata $A^*$ can be classified by their:

- evaluation: bottom-up $A^*_b$ and top-down $A^*_t$
- rules: deterministic $A^*_d$ and non-deterministic $A^*_n$

Expressive power and complexity

- deterministic top-down $A^*_d$ are less expressive than deterministic bottom-up $A^*_b$.
- all non-deterministic $A^*_n$ can be transformed into deterministic $A^*_d$ with an exponential explosion of the number of states: $\text{size}(A^*_d) = \text{size}(A^*_n)^k$
Closure Property

Definition

A schema language $\mathcal{L}$ is closed under a set operation $\phi \in \{\cap, \cup, -\}$, iff for all tree grammars $G_1 \in \mathcal{L}$ and $G_2 \in \mathcal{L}$, there exists a tree grammar $G_3 \in \mathcal{L}_S$ where $L(G_3) = L(G_1)\phi L(G_2)$.

- The family of schema languages corresponding to regular tree grammars (XDuce, Relax core) are closed under union, intersection and difference.
- The family of schema languages corresponding to local (DTD) and unique type (XML schema) grammars are closed under intersection but not under difference and union.
Closure Property

Given two DTD/XML Schemas $dtd_1$ and $dtd_2$, it is not always possible to define a DTD/XML Schema:

- $dtd_1 \cup dtd_2$ which accepts exactly the instances of $dtd_1$ and $dtd_2$:
  \[
  sat(dtd_1 \cup dtd_2) = sat(dtd_1) \cup sat(dtd_2)
  \]

- $dtd_1 - dtd_2$ which accepts all instances of $dtd_1$ except those which are also instance of $dtd_1$:
  \[
  sat(dtd_1 - dtd_2) = sat(dtd_1) - sat(dtd_2)
  \]
Computing Difference, Intersection and Union

Intersection and union

- Compute the product automata $DTD_A \times DTD_B$.
- Final states of $DTD_A \cap DTD_B = \text{couple of final states in } DTD_A$ and $DTD_B$.
- Final states of $DTD_A \cup DTD_B = \text{couple of states where one state is a final state in } DTD_A$ or $DTD_B$.

Difference

- Compute the complementary automata $\neg DTD_B$ of $DTD_B$ by exchanging final states and non-final states in the corresponding deterministic tree automata.
- Compute the intersection between $DTD_A$ and $\neg DTD_B$. 
Closure under Intersection

Grammar $DTD_A$:

\[
D \rightarrow \text{doc ( } T, A?, S^* \text{)} \\
T \rightarrow \text{title (PCDATA)} \\
A \rightarrow \text{abstract (PCDATA)} \\
S \rightarrow \text{section ( } T, R?, P^* \text{)} \\
R \rightarrow \text{resume (PCDATA)} \\
P \rightarrow \text{para (PCDATA)}
\]

Grammar $DTD_B$:

\[
D \rightarrow \text{doc ( } T, S^+ \text{)} \\
T \rightarrow \text{title (PCDATA)} \\
S \rightarrow \text{section ( } T, P^* \text{)} \\
P \rightarrow \text{para (PCDATA)}
\]

$DTD_A \cap DTD_B$:

\[
D \rightarrow \text{doc ( } T, S^+ \text{)} \\
T \rightarrow \text{title (PCDATA)} \\
S \rightarrow \text{section ( } T, P^* \text{)} \\
P \rightarrow \text{para (PCDATA)}
\]
Closure under Union

$DTD_A \cup DTD_B$:

$D \rightarrow \:\text{doc} \ (T, (A?, S1*)\mid(S2*))$

$T \rightarrow \:\text{title} \ (PCDATA)$

$A \rightarrow \:\text{abstract} \ (PCDATA)$

$S1 \rightarrow \:\text{section} \ (T, R, P*)$

$S2 \rightarrow \:\text{section} \ (T, P+)$

$R \rightarrow \:\text{resume} \ (PCDATA)$

$P \rightarrow \:\text{para} \ (PCDATA)$

Compare with:

$D \rightarrow \:\text{doc} \ (T, (A?, S*) )$

$T \rightarrow \:\text{title} \ (PCDATA)$

$A \rightarrow \:\text{abstract} \ (PCDATA)$

$S \rightarrow \:\text{section} \ (T, R?, P*)$

$R \rightarrow \:\text{resume} \ (PCDATA)$

$P \rightarrow \:\text{para} \ (PCDATA)$

Compute union $DTD_A \cup DTD_B$?

- normalize $DTD_A$ and $DTD_B$
- rename concurrent type names in $DTD_A$ and $DTD_B$.
- compute the union $U$ of all rules
- normalize $U$
Closure under Difference

$\text{DTD}_A - \text{DTD}_B$:

$$
D \rightarrow \text{doc}(T, ((S1^*, S2, S1^*)|(A, S1^*)))
$$

$$
T \rightarrow \text{title}(\text{PCDATA})
$$

$$
A \rightarrow \text{abstract}(\text{PCDATA})
$$

$$
S1 \rightarrow \text{section}(T, R?, P^*)
$$

$$
S2 \rightarrow \text{section}(T, R, P^*)
$$

$$
R \rightarrow \text{resume}(\text{PCDATA})
$$

$$
P \rightarrow \text{para}(\text{PCDATA})
$$
Tree Grammars and Validation

All tree grammars can be validated in one step using a SAX parser.

Example
SAX validation of a unique type grammar (MXL Schema).

Enter node $n$ of type $a$

Find the rule $X \rightarrow a \ r$:

- $a \in S$: there exists at most one such rule
- $a \notin S$:
  - find the rule $X' \rightarrow b \ r'$ where $X$ exists in $r'$ (unique type grammar: there exists at most one such rule)

Leave node $n$ of type $a$

Compare the sequence $N$ of child types of $n$ with $r$:

- if $N \notin L(r)$ then stop with an error.
Outline

Introduction

XML Syntax

XML Programming

XML Namespaces

XML Schema Languages

Exercises
Exercises

Definitions
XML Syntax, DOM and SAX, Namespace, DTD, Tree grammar

True or False?

1. It is possible to build a DOM parser on top of a SAX parser.
2. It is possible to build a SAX parser on top of a DOM parser.
3. For each well-formed document \( d \) there exists a DTD \( T \) such that \( d \) is valid with respect to \( D \).
4. XML Schema is more expressive than DTD.
5. For any pair of DTDs \( A \) and \( B \) there exists a DTD \( C \) such that \( C \) validates only documents which are also validated by \( A \) or by \( B \).
6. For any pair of DTDs \( A \) and \( B \) there exists a DTD \( C \) such that \( C \) validates all documents which are validated by \( A \) or by \( B \).
Introduction

XPath 1.0: From Trees to Nodesets

XPath 2.0 and XQuery 1.0

Exercises
Query Languages for XML

Some history before starting


► 1998: Workshop Query Languages’98 (QL’98)

► after 1998: first XML query languages,: XOQL, XML-QL, XQL, Lore, ...

► 2003: W3C Working Draft XQuery

► 2007: W3C Recommendation XQuery 1.0 and XPath 2.0

► 2010: XQuery 1.0 and XPath 2.0 Formal Semantics

► 2011: XQuery and XPath Full Text 1.0, XQuery Update Facility 1.0

► 2012: XPath/XQuery 3.0 in preparation
XQuery and XPath

Modèles et Langages:

Groupes de travail W3C:

(Don Chamberlin)
Introduction

XPath 1.0: From Trees to Nodesets

XPath 2.0 and XQuery 1.0

Exercises
XPath 1.0

Goal

Extract nodesets from XML documents.

Usage of XPath in

- **XQuery**: extract node sequences from XML trees
- **XSLT**: choose nodes to transform and rules to apply
- **XML Schema**: define key and reference nodes
- **XLink**: define links between nodes
Conjunctive Tree Patterns

XPath Syntax and Semantics
Movies.xml

<?xml version='1.0' encoding='iso-8859-1' ?>
<timeout>
  <cinema>
    <name>Action Christine</name>
    <showtime movie="f76">6pm</showtime>
    <showtime movie="f98">8pm</showtime>
  </cinema>
  <movie id="f76">
    <title lang='fr'>Le Parrain</title>
    <title lang='en'>The Godfather</title>
    <actor><name>Brando</name></actor>
  </movie>
  <movie id="f98">
    <title>Brazil</title>
  </movie>
</timeout>
Movies.dtd

```xml
<!ELEMENT timeout (cinema*, movie*) >
<!ELEMENT cinema (name, showtime*) >
<!ELEMENT name #PCDATA>
<!ELEMENT showtime #PCDATA>
<!ATTLIST showtime movie IDREF>
<!ELEMENT movie title *, actor*> 
<!ATTLIST movie id ID >
<!ELEMENT title #PCDATA>
<!ATTLIST title lang CDATA>
<!ELEMENT actor name>
```
Movies.xml

Find
- the names of all actors of the movie “The Godfather”.
- the french title of the movie “The Godfather”
- the names of all cinemas showing a movie with M. Brando.
Conjunctive tree pattern

Definition

A **conjunctive tree pattern** is a labeled tree:

- Nodes are labeled by element/attribute names, values or symbol `*` (any element)
- Arcs connect a parent to its children (simple arc) or to its descendants (double arc).
- Exactly one (surrounded) node corresponds to the pattern result.
Example

Names of all actors of the movie “The Godfather”:

//movie[title='The Godfather']/actor/name
Conjunctive tree pattern

French title of the movie “The Godfather”:

```
//movie[title='The Godfather']/title[@lang='fr']
```
Example

All cinemas showing a movie with M. Brando:
8 - XPath 1.0: From Trees to Nodesets

Conjunctive Tree Patterns

XPath Syntax and Semantics
XPath Document Model

XPath uses the DOM representation for XML documents:

- **Ordered labeled trees**
- Each node has a `type`: `Document`, `Element`, `Attribute`, `Comment`, `Text`, `ProcessingInstruction`
- Some nodes have a `name`: elements, attributes, processing instructions
- Some nodes have a `value`: attributes, text nodes, comments
XPath Expressions

XPath expressions

A XPath Expression is a sequence of steps:

\[ \text{step}_1 / \text{step}_2 / \ldots / \text{step}_n \]

Steps

A step is composed of an axis, a filter and a sequence of predicates: \( \text{axis} :: \text{filter} [\text{predicate}_1] \ldots [\text{predicate}_n] \)

- **axis**: a structural constraint relative to the context node
- **filter**: constraints on the type and the name of the nodes
- **predicate**: value and other (complex) constraints

Example

\[
\text{child} :: \text{movie} [\text{child} :: \text{actor} / \text{child} :: \text{name} = '\text{Brando}'] [\text{position}() = 1]
\]
Examples

Example

All movies:

▶ /child::timeout/child::movie
▶ /descendant::movie

All elements:

▶ /descendant::*

All attributes:

▶ /descendant::*/*attribute::*

All comments:

▶ /descendant::comment()

All attributes of movie elements:

▶ /descendant::movie/attribute::*

```xml
<?xml version='1.0' encoding='iso-8859-1' ?>
<timeout>
  <cinema>
    <name>Action Christine</name>
    <showtime movie="f76">6pm</showtime>
    <showtime movie="f98">8pm</showtime>
  </cinema>
  <movie id="f76">
    <title lang='fr'>Le Parrain</title>
    <title lang='en'>The Godfather</title>
    <actor><name>Brando</name></actor>
  </movie>
  <movie id="f98">
    <title>Brazil</title>
  </movie>
</timeout>
```
Semantics: Nodesets and context values

Given a document \( d \):

- A **nodeset** \( L \) is a list of nodes (with no duplicates) sorted in document order.
- A **context value** is a couple \((L_C, N_C)\) where
  - \( L_C \) is a nodeset
  - \( N_C \in L_C : \text{context node} \)

Given a context value \((L_C, N_C)\), a **step**

\[
\text{axis}::\text{filter}[\text{pred}_1] \ldots [\text{pred}_n]
\]

returns the nodeset \( L_i \) accessible from node \( N_C \) by (1) following \( \text{axis} \), (2) applying filter \( \text{filter} \) and (3) validating the sequence of predicates \( \text{pred}_i \):

- \( \text{pred}_1 \) is applied on the context value obtained by step \( \text{axis}::\text{filter} \)
- all other predicates \( \text{pred}_{i+1} \) are applied on the context value obtained after the previous predicate \( \text{pred}_i \).
Expressions: Semantics

An expression \textit{expression} \( \texttt{step}_1/\ldots/\texttt{step}_n \) returns the union \( L_n \) of all node sets obtained by applying \( \texttt{step}_n \) to all context values \((L_{n-1}, N_j)\) where

- \( L_{n-1} \) is the nodeset returned by expression \( \texttt{step}_1/\ldots/\texttt{step}_{n-1} \) and
- \( N_j \) is a node in \( L_{n-1} \).

\( / \) denotes the singleton nodeset containing the document root.
Examples

- `/`: document root
- `/child::*`: all elements that can be reached from the document root by following the child axes
- `/child::*//child::*`: the children elements of the children elements of the document root
- `/descendant::*`: all element nodes
- `/descendant::*//descendant::*`: all descendants of descendants
- `/child::*//descendant::*`: same as previous expression?
XPath Axis

/child::*timeout/descendant::*movie

Ordered tree navigation

- self
- parent
- child
- attribute
- descendant
- descendant-or-self
- ancestor
- ancestor-or-self
- preceding
- following
- preceding-sibling
- following-sibling
Attribute is **not** a tree axis.
Filters

/child::timeout/descendant::*

Type filters

<table>
<thead>
<tr>
<th>Filter</th>
<th>DOM type</th>
</tr>
</thead>
<tbody>
<tr>
<td>*</td>
<td>Element or Attribute</td>
</tr>
<tr>
<td>text()</td>
<td>Text</td>
</tr>
<tr>
<td>comment()</td>
<td>Comment</td>
</tr>
<tr>
<td>processing-instruction()</td>
<td>ProcessingInstruction</td>
</tr>
<tr>
<td>node()</td>
<td>no constraint</td>
</tr>
</tbody>
</table>

Name filters

- Can be applied to nodes with a name: Element, ProcessingInstruction, Attribute
- Filter A filters all nodes with name A
Exemples

- `/descendant::D/parent::*`
- `/descendant::C/parent::*/preceding-sibling::*`
- `/descendant::B/child::C/following-sibling::G`
- `/descendant::*/preceding::*C`
- `/descendant::*text()`
A **predicate** is an expression $[\text{test}]$ where *test* is a (boolean) **predicate expression**:

- any XPath value/expression
- comparisons: $=, <, >, <=, >=, !=$
- any function call
- any boolean combination of tests using **and**, **or**, **not**.

Any value can be transformed into a boolean value.
XPath Value Types

Atomic values

- **numeric**: 5, 1.3, 100.354
  - comparisons: =, <, >, ! =, +, −, *, div, mod
- **string**: ’abc’, “abc”, “ty56”, ’3.145’
  - comparisons: =, !=
- **boolean**: true() and false()
Type conversions

Any value can be converted into to string and into boolean (for predicates)

Conversion into string

<table>
<thead>
<tr>
<th>Type</th>
<th>&quot;NaN&quot;, &quot;Infinity&quot;, &quot;0.3&quot;, &quot;143&quot;, &quot;-3.12&quot;</th>
</tr>
</thead>
<tbody>
<tr>
<td>numeric</td>
<td></td>
</tr>
<tr>
<td>boolean</td>
<td>&quot;true&quot;, &quot;false&quot;</td>
</tr>
<tr>
<td>nodeset</td>
<td>concatenation of all text descendants</td>
</tr>
<tr>
<td>numeric</td>
<td>0 or NaN</td>
</tr>
<tr>
<td>string</td>
<td>empty string</td>
</tr>
<tr>
<td>nodeset</td>
<td>empty set</td>
</tr>
</tbody>
</table>

Conversion into boolean

<table>
<thead>
<tr>
<th>Type</th>
<th>false</th>
<th>true</th>
</tr>
</thead>
<tbody>
<tr>
<td>numeric</td>
<td></td>
<td>all other values</td>
</tr>
<tr>
<td>string</td>
<td></td>
<td>all other values</td>
</tr>
<tr>
<td>nodeset</td>
<td></td>
<td>all other values</td>
</tr>
</tbody>
</table>
Exemples

- /descendant::*[child::D='ddd']
- /descendant::*[parent::B]
- /descendant::*[following-sibling::*='ddd']/child::*
- /descendant::*[child::text()]
XPath Functions

- **last()**: length of current nodeset $L_C$
- **position()**: position of current context node $N_C$ in $L_C$
- **name()**: node name
- **count(expr)**: size of nodeset generated by $expr$
- **not(expr)**: boolean negation
- **contains(str1, str2)**
- **concat(str1, str2, ...)**
- **string-length(str)**
## Abbreviated Syntax

<table>
<thead>
<tr>
<th>Expression</th>
<th>Abbreviated Syntax</th>
</tr>
</thead>
<tbody>
<tr>
<td>child::</td>
<td>default axes</td>
</tr>
<tr>
<td>descendant-or-self::node()</td>
<td>default step</td>
</tr>
<tr>
<td>parent::node()</td>
<td>..</td>
</tr>
<tr>
<td>self::node()</td>
<td>.</td>
</tr>
<tr>
<td>attribute::</td>
<td>@</td>
</tr>
<tr>
<td>[position()=x]</td>
<td>[x]</td>
</tr>
</tbody>
</table>

### Example

<table>
<thead>
<tr>
<th>Expression</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>child::A</td>
<td>A</td>
</tr>
<tr>
<td>/descendant-or-self::node()/child::A</td>
<td>//A</td>
</tr>
<tr>
<td>child::A/parent::node()/child::B</td>
<td>A/..B</td>
</tr>
<tr>
<td>child::A/attribute::att</td>
<td>A/@att</td>
</tr>
</tbody>
</table>
Example

All cinemas showing a movie with M. Brando:

```
//cinema[showtime/@movie=//movie[actor/name='Brando']/@id]/name
```
Examples

Cinemas showing movies with M. Brando:

- //cinema[showtime/@movie=//movie[actor[name='Brando']]/@id]/name
- //cinema[showtime/@movie=../movie[actor[name='Brando']]/@id]/name
- //cinema[showtime/@movie=../movie/actor[name='Brando']/../@id]/name
- //showtime[@movie=//actor[name='Brando']/../@id]/../name
Examples

**Movies with actors:**
- //movie[actor]
- //movie[count(actor)>0]

**Movies without actors:**
- //movie[count(actor)=0]
- //movie[not(actor)]

**The last movie:**
- /descendant::movie[position()=last()]
- /descendant::movie[last()]

**The last element:**
- /descendant::*[last()]

**The last child element of each node:**
- /descendant-or-self::node()/child::*[last()]
- /*[last()]

**Cinemas showing at least two movies:**
- //cinema[count(.//movie) > 1]
Outline

Introduction

XPath 1.0: From Trees to Nodesets

XPath 2.0 and XQuery 1.0

Exercises
XPath 1.0 and 2.0

XPath 1.0

- Types: number, string, boolean, nodeset
- Set semantics without set operations
- Document order

XPath 2.0/XQuery 1.0

- XML Schema Type System
- Set/list semantics and operations
- Renaming, reordering, restructuring
9 - XPath 2.0 and XQuery 1.0

XQuery Data Model

XQuery by Example

Exercises
XQuery Data Model (XDM)

XQuery 1.0 and XPath 2.0 share the same data model:

- A **value** is a *sequence of items*.
- An **item** is a *node* or an *atomic value*.
- Each value has a **type**.

```mermaid
graph TD
    value --> item
    item --> node
    item --> atomic value
```

▶ A **value** is a *sequence of items*.
▶ An **item** is a *node* or an *atomic value*.
▶ Each value has a **type**.
Atomic Types

XML Schema Atomic Types: rich set of atomic types

Type hierarchy (excerpt)

Arithmetic and boolean operators

- unary operators: +3, −4, not(false ()),
- binary operators: 3+3.2, 5−2, 6∗7, 7 div 3, 5 mod 2, true () and false (), false () or true ()
Sequence Types

Item Sequences

- No distinction between a single item and a sequence of length 1
- A sequence might contain items of different types
- Sequences are flat (no nesting)

<table>
<thead>
<tr>
<th>XDM Value</th>
<th>XDM Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>47</td>
<td>(xsd:int)</td>
</tr>
<tr>
<td>(47)</td>
<td>(xsd:int)</td>
</tr>
<tr>
<td>(1, “toto”, &lt;toto/&gt;)</td>
<td>(xsd:int,xsd:string,toto)</td>
</tr>
<tr>
<td>(1, 2, 6, “toto”, &lt;toto/&gt;)</td>
<td>(xsd:int*,xsd:string,toto)</td>
</tr>
<tr>
<td>(1, (2, 6), &quot;toto&quot;, &lt;toto/&gt;)</td>
<td>(xsd:int*,xsd:string,toto)</td>
</tr>
<tr>
<td>()</td>
<td>()</td>
</tr>
</tbody>
</table>
Sequence Operations

Sequence (set) operations

- union
- except
- intersect
- , : concatenation

! All operations have set semantics (no duplicates) and maintain the nodes in document order.
Comparisons

- **Item sequences by value:**
  - =, !=, >, >=, <, <=
  - \( A \phi B = \text{true} \iff \exists x \in A, y \in B : x \phi y \)
  - \((1,5) < (2) : \text{true}, (5,6) \lt (1,2,3) : \text{false}()\)

- **Single item by value:**
  - eq, ne, gt, ge, lt, le
  - \((1,5) \lt (2) : \text{error}, 1 \lt 2 : \text{true}()\)

- **Single nodes by identity:**
  - is, isnot

- **Single nodes by position in document:**
  - \(<\lt \text{(before)}, \gt\gt \text{(after)}\)
Comparison

\[(1, 2, 3) > (2, 4, 5) \rightarrow true()\]
\[(1, 2, 3) = 1 \rightarrow true()\]
\[() = 0 \rightarrow false()\]
\[2 <= 1 \rightarrow false()\]
\[(1, 2, 3)! = 3 \rightarrow true()\]
\[(1, 2)! = (1, 2) \rightarrow true()\]
\[not((1, 2) = (1, 2)) \rightarrow false()\]

From \(a = b \land b = c\) we cannot conclude \(a = c\).

More examples come later...
XQuery Expressions

Any XDM value is a query.

Any XPath (2.0) expression is a query.

Any for-let-where-order by-return (FLWOR) expression is a query.

Any if-then-else

Any function call is a query

XQuery is a functional language:

any query returns a value or an error, and

queries have no side effect (no update) → it is possible to compose query expressions
Simple queries

XDM Value Queries

- (46)
- (1, 2, 3, 4, 5))
- (1 to 5))
- ((1, 2, 3), (), (4, 5)))
- (" Toto ")
- date("2002−10−23")
- (1, " toto ", <toto/>)
- (<x>1</x>, <y>2</y>)
Three steps in defining and using variables:

- **for** and **let**: node sequence extraction (path expressions) and variable assignment
- **where**: tests (selection and join)
- **return**: construct result
Variable assignment: for and let

Definition

- **for $var in exp**
  - assign to variable $var successively an item returned by expression exp.

Definition

- **let $var := exp**
  - assign to variable $var the list of items returned by expression exp.
Example: \texttt{for} – \texttt{let} – \texttt{return}

\textbf{Query}

\begin{verbatim}
for $a$ in (1 to 3)
let $b := ('a', 'b', 'c')$
return ($a$, $b[$a$])
\end{verbatim}

<?xml version="1.0" encoding="UTF-8"?>
<1
a
2
b
3
c>
Selection: where

**Definition**

- `expA where expB`
- keep all items returned by expression `expA where expB` is true.

In general `expB` uses a variable defined in expression `expA`. 
Example: for – let – where – return

Query

\[
\text{for } \$a \text{ in (1 to 23),} \\
\quad \$b \text{ in (1 to } \$a), \\
\quad \$c \text{ in } (\$b \text{ to } (\$a) - 1) \\
\text{where } (\$b \times \$c) = \$a \\
\text{return } \$a
\]
Quantification

Definition

▶ some $var$ in $expr1$ satisfies $expr2$
▶ true iff there exists at least one node in $expr1$ satisfying expression $expr2$.

Definition

▶ every $var$ in $expr1$ satisfies $expr2$
▶ true iff all nodes returned by expression $expr1$ satisfy expression $expr2$
Example: for – let – where – some – return

Query

```
for $a in (1 to 100)
let $d := (2 to $a)
where not (some $c in $d satisfies
some $d in $d satisfies $c * $d = $a)
return ($a)
```
Building Nodes

Create element and attribute nodes

- element { expr-name } { expr-contents }
- element name { expr-contents }
- attribute { expr-name } { expr-contents }
- attribute name { expr-contents }

<table>
<thead>
<tr>
<th>XDM Value</th>
<th>XDM Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>element { “A” } { expr }</td>
<td>element A</td>
</tr>
<tr>
<td>element A { expr }</td>
<td>element A</td>
</tr>
<tr>
<td>&lt;A&gt; { expr } &lt;/A&gt;</td>
<td>element A</td>
</tr>
<tr>
<td>attribute { “B” } { expr }</td>
<td>attribute B</td>
</tr>
<tr>
<td>attribute B { expr }</td>
<td>attribute B</td>
</tr>
</tbody>
</table>
Example

Query

```xml
<sequences>
  <s1>{(1, 2, 3, 4, 5)}</s1>
  <s2>{(1 to 5)}</s2>
  <s3>(((1, 2), (), (3, (4, 5))))</s3>
</sequences>

<?xml version="1.0" encoding="UTF-8"?>
<sequences>
  <s1>1 2 3 4 5</s1>
  <s2>1 2 3 4 5</s2>
  <s3>1 2 3 4 5</s3>
</sequences>
```
Functions and Operators

Predefined functions and operators for:

- accessing nodes types and names
- constructing, comparing and transforming values,
- aggregating values of a sequence.

Functions and operators are

- typed (signature)
- process typed item sequences and values
XQuery Functions

- **Input functions:**
  - \textit{collection}(\textit{url}): query a collection of documents
  - \textit{doc}(\textit{url}): query a single document

- **Predefined functions:**
  - XQuery 1.0 and XPath 2.0 Functions and Operators
  - http://www.w3.org/TR/xquery-operators/

- **User defined functions**
Functions for sequences

- `concatenate(item* Seq1, item* Seq1): item*`
- `item−at(item* Seq, decimal Pos): item?`  
  return item at position Pos
- `distinct−nodes(item* Seq1): item*`  
  duplicate elimination (on node identity)
- `distinct−values(item* Seq1): item*`  
  duplicate elimination (on item value)
- `index−of(item* Seq, item Search): unsignedInt*`  
  list of positions of items with a value equal to parameter Search
9 - XPath 2.0 and XQuery 1.0

XQuery Data Model

XQuery by Example
<bib>
    <book title="Web Data Management">
        <author>Abiteboul</author><author>Manolescu</author><author>Rigaux</author><author>Rousset</author><author>Sennelart</author>
        <publisher>Cambridge University Press</publisher>
    </book>
    <book year="2001" title="Spatial Databases">
        <author>Rigaux</author><author>Scholl</author><author>Voisard</author>
        <publisher>Morgan Kaufmann Publishers</publisher>
        <price>35.00</price>
    </book>
    <book year="2000" title="Data on the Web">
        <author>Abiteboul</author><author>Buneman</author><author>Suciu</author>
        <publisher>Morgan Kaufmann Publishers</publisher>
        <price>39.95</price>
    </book>
</bib>
Path Expressions

\[ \text{doc("bib.xml")//book[2]/author} \]

\[
\text{<author><la>Rigaux</la><fi>P.</fi></author>,} \\
\text{<author><la>Scholl</la><fi>M.</fi></author>,} \\
\text{<author><la>Voisard</la><fi>A.</fi></author>}
\]

The result is a \textit{sequence of items!}
Path Expressions

Each step is a *query expression* (XPath *and* XQuery):

- `doc("bib.xml")/bib/book/author`
- `doc("bib.xml")/bib//book[1]/publisher`
- `doc("bib.xml")//book/(publisher,author)`
- `doc("bib.xml")//book/(@title union publisher)`
- `doc("bib.xml")//book[position() < last()]`
Node creation

```xml
<authors>
  { doc("bib.xml")//book[2]/author/la }
</authors>

<?xml version="1.0" encoding="UTF-8"?>
<authors>
  <la>Rigaux</la>
  <la>Scholl</la>
  <la>Voisard</la>
</authors>
```

The result is a **single** XML fragment.
Union

Query

```xml
<book>
    { doc("bib.xml")//book[2]/(price union publisher union author/la) }
</book>

<?xml version="1.0" encoding="UTF-8"?>

<book>
    <la>Rigaux</la>
    <la>Scholl</la>
    <la>Voisard</la>
    <publisher>Morgan Kaufmann Publishers</publisher>
    <price>35.00</price>
</book>
```
XPath 1.0: From Trees to Nodesets

XPath 2.0 and XQuery 1.0

XQuery Data Model
Variables: FLWR Expressions
XQuery by Example
Path Expressions

### Query

```xml
<book>
  { doc("bib.xml") // book[2]/( * except author ) }
</book>
```

```xml
<?xml version="1.0" encoding="UTF-8"?>
<book>
  <publisher>Morgan Kaufmann Publishers</publisher>
  <price>35.00</price>
</book>
```
Concentration

Query

```xml
<book>
  
  \{ doc("bib.xml")//book[2]/(@title,publisher),
  \n    doc("bib.xml")//book[2]/author/la \} 

</book>
```

```xml
<?xml version="1.0" encoding="UTF-8"?>
<book title="Spatial Databases">
  <publisher>Morgan Kaufmann Publishers</publisher>
  <la>Rigaux</la>
  <la>Scholl</la>
  <la>Voisard</la>
</book>
```
Text values

Query

```
doc("bib.xml")//book[1]/author/la/text()
```

```xml
<?xml version="1.0" encoding="UTF-8"?>
AbiteboulManolescuRigauxRoussetSennelart
```

Query

```
"Authors of the second book: ",
doc("bib.xml")//book[2]/author/str(la)
```

```xml
<?xml version="1.0" encoding="UTF-8"?>
Authors of the second book: Rigaux Scholl Voisard
```
Text values

Query

\[
\text{fn: string(doc("bib.xml")//book[2]/author/la)}
\]

ERROR

Query

\[
\text{fn: data(doc("bib.xml")//book[2]/author/la)}
\]

<?xml version="1.0" encoding="UTF-8"?>
Rigaux Scholl Voisard
Comparison: eq, lt, ...

Query

count(doc("bib.xml")//book[author/la=("Scholl","Abiteboul")] )

<?xml version="1.0" encoding="UTF-8"?>
3

Query

doc("bib.xml")//book/author[la eq "Scholl"]

<?xml version="1.0" encoding="UTF-8"?>
<author><la>Scholl</la><fi>M.</fi></author>

Query

doc("bib.xml")//book[author/la eq "Scholl"]

ERROR
Comparison by node identity: is

Query

```xml
<?xml version="1.0" encoding="UTF-8"?>
<book title="Web Data Management">
  <author><la>Abiteboul</la><fi>S.</fi></author>
  <author><la>Manolescu</la><fi>I.</fi></author>
  <author><la>Rigaux</la><fi>P.</fi></author>
  <author><la>Rousset</la><fi>MC.</fi></author>
  <author><la>Sennelart</la><fi>P.</fi></author>
  <publisher>Cambridge University Press</publisher>
</book>
```
Comparison by document order: «, »

Query

```xml
<book>
  { doc("bib.xml") // book[author[la = "Abiteboul"]
             << author[la = "Suciu"]]
       /@title
  }
</book>
```

```xml
<?xml version="1.0" encoding="UTF-8"?>
<book title="Data on the Web"/>
```
Variable assignment: for

Query

```xml
for $a in doc("bib.xml")// author[la eq "Voisard"]
return $a
```

```xml
<?xml version="1.0" encoding="UTF-8"?>
<author>
  <la>Voisard</la>
  <fi>A.</fi>
</author>
```
Example with *for* and *let*

**Query**

```xml
for $b in doc("bib.xml")/book[3]
let $al := $b/author
return <book nb_authors="{count($al)}">
    { $al/la }
</book>

<?xml version="1.0" encoding="UTF-8"?>
<book nb_authors="3">
    <la>Abiteboul</la>
    <la>Buneman</la>
    <la>Suciu</la>
</book>
```
Selection: where

Query

```xml
<book>
  { for $a in doc("bib.xml")//book
    where $a/author[1]/la eq "Rigaux"
    return $a/@title
  }
</book>
```

```xml
<?xml version="1.0" encoding="UTF-8"?>
<book title="Spatial Databases"/>
```
Tests: if –then–else

Query

```xml
<books>
    {  for $b in doc("bib.xml")//book
        where $b/author/last = "Rigaux" return
            if ($b/@year > 2000)
                then <book recent="true"> {$b/@title} </book>
            else <book> {$b/@title} </book>
    }
</books>

<?xml version="1.0" encoding="UTF-8"?>
<books>
    <book title="Web Data Management"/>
    <book recent="true" title="Spatial Databases"/>
</books>
```
Universal Quantification

Query

for $a$ in doc("bib.xml")//author
where every $b$ in doc("bib.xml")//book [author/la = $a/la]
satisfies $b/publisher = "Morgan Kaufmann Publishers"
return string($a/la)

<?xml version="1.0" encoding="UTF-8"?>
Scholl Voisard Buneman Suciu
Node Generation

Query

```xml
for $b in doc("bib.xml")//book[2]
return
element livre{
  element annee { string($b/@year) },
  for $e in $b/@*
    where name($e) != "year"
    return element {name($e)} { string($e)} }
```

```xml
<?xml version="1.0" encoding="UTF-8"?>
<livre>
  <annee>2001</annee>
  <title>Spatial Databases</title>
</livre>
```
Node Generation

Query

```xml
<livres>
    { for $t in doc("bib.xml")//book/@title
      return element livre {
        attribute titre { string($t) }
      }
    }
</livres>
```

```xml
<?xml version="1.0" encoding="UTF-8"?>
<livres>
    <livre titre="Web Data Management"/>
    <livre titre="Spatial Databases"/>
    <livre titre="Data on the Web"/>
</livres>
```
Sort: order by

Definition

- $Expr1$ order by $Expr2$ (ascending | descending)?
- order items returned by expression $Expr1$ by the values returned by expression $Expr2$. 
Example order by

Query

```xml
<books>
  { for $b in doc("bib.xml")///book
      order by $b/@year
      return element book { $b/@title, $b/@year } }
</books>

<?xml version="1.0" encoding="UTF-8"?>
<books>
  <book title="Data on the Web" year="2000"/>
  <book title="Spatial Databases" year="2001"/>
  <book title="Web Data Management"/>
</books>
```
Join

<addresses>
  <person>
    <name>Amann</name>
    <country>France</country>
    <institution>CNAM</institution>
  </person>
  <person>
    <name>Scholl</name>
    <country>France</country>
    <institution>CNAM</institution>
  </person>
  <person>
    <name>Voisard</name>
    <country>Germany</country>
    <institution>FU Berlin</institution>
  </person>
</addresses>
Join

```xml
for $b in doc("bib.xml")//book
return element livre {
    attribute titre {$b/@title},
    for $a in $b/author
    return element auteur {
        attribute nom {$a/la},
        for $p in doc("addr.xml")//person
        where $a/la = $p/name
        return attribute institut {$p/institution} }
} }
```
Join: Result

```xml
<?xml version="1.0" encoding="UTF-8"?>
<livre titre="Web Data Management">
  <auteur nom="Abiteboul"/>
  <auteur nom="Manolescu"/>
  <auteur nom="Rigaux"/>
  <auteur nom="Rousset"/>
  <auteur nom="Sennelart"/>
</livre>

<livre titre="Spatial Databases">
  <auteur nom="Rigaux"/>
  <auteur nom="Scholl" institut="CNAM"/>
  <auteur nom="Voisard" institut="FU Berlin"/>
</livre>

<livre titre="Data on the Web">
  <auteur nom="Abiteboul"/>
  <auteur nom="Buneman"/>
  <auteur nom="Suciu"/>
</livre>
```
Functions: `avg`

**Query**

```xml
for $p$ in distinct-values(doc("bib.xml")
    // publisher)
let $l := doc("bib.xml")//book[publisher = $p]
return element publisher {
    attribute name {string($p)},
    attribute avg_price { avg($l/price) }
}
```

```xml
<?xml version="1.0" encoding="UTF-8"?>
<publisher name="Cambridge University Press" avg_price=""/>
<publisher name="Morgan Kaufmann Publishers" avg_price="37.475"/>
```
User-defined function

```xml
define function NumberAuthors(book $b) returns xsd:integer {
    return count($b/author)
}
```

The result is of type `xsd:int`. 
Type polymorphism: typeswitch

define function AvgPrice(Article $a) returns xsd:float {
    typeswitch ($a)
    case Jouet return avg($s/@prix)
    case Toy return avg($s/@price)*1.2
    default return 0.0
}
**Function id and operator ==>**

```xml
<!ELEMENT timeout (cinema*, movie*) >
<!ELEMENT cinema (name, showtime*) >
<!ELEMENT name #PCDATA>
<!ELEMENT showtime #PCDATA>
<!ATTLIST showtime movie IDREF>
<!ELEMENT movie title *, actor*> 
<!ATTLIST movie id ID lang CDATA>
<!ELEMENT title #PCDATA>
<!ELEMENT actor name>
```

Movies shown at the Cinema Action Christine:

```xml
for $s in doc("Movies.xml")//showtime
return fn:id($s/@movie)
```

Instead of using function id(), we can use the operator `→` for following ID/IDREF links.

```xml
doc("Movies.xml")//showtime/@movie→movie/title
```
Outline

Introduction

XPath 1.0: From Trees to Nodesets

XPath 2.0 and XQuery 1.0

Exercises
Exercise (2)

<table>
<thead>
<tr>
<th>R</th>
<th>A</th>
<th>B</th>
<th>C</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>a</td>
<td>x</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>b</td>
<td>y</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>c</td>
<td>z</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>a</td>
<td>u</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>c</td>
<td>x</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>c</td>
<td>z</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>S</th>
<th>C</th>
<th>D</th>
<th>E</th>
</tr>
</thead>
<tbody>
<tr>
<td>u</td>
<td>a</td>
<td>x</td>
<td></td>
</tr>
<tr>
<td>v</td>
<td>k</td>
<td>y</td>
<td></td>
</tr>
<tr>
<td>w</td>
<td>n</td>
<td>z</td>
<td></td>
</tr>
<tr>
<td>x</td>
<td>b</td>
<td>x</td>
<td></td>
</tr>
<tr>
<td>y</td>
<td>l</td>
<td>x</td>
<td></td>
</tr>
<tr>
<td>z</td>
<td>d</td>
<td>z</td>
<td></td>
</tr>
</tbody>
</table>

```xml
<DB>
  <R>
    <t><A>1</A><B>a</B><C>x</C></t>
    <t><A>2</A><B>b</B><C>y</C></t>
    <t><A>3</A><B>c</B><C>z</C></t>
    <t><A>4</A><B>a</B><C>u</C></t>
    <t><A>5</A><B>c</B><C>x</C></t>
    <t><A>6</A><B>c</B><C>z</C></t>
  </R>
  <S>
    <t><C>u</C><D>a</D><E>x</E></t>
    <t><C>v</C><D>x</D><E>y</E></t>
    <t><C>u</C><D>x</D><E>z</E></t>
    <t><C>v</C><D>b</D><E>x</E></t>
    <t><C>x</C><D>d</D><E>z</E></t>
  </S>
</DB>
```
Exercise (3)

Translate the following queries into XPath/XQuery:

1. \( \pi_{A,C}(R) \)
2. \( \pi_B(R) \)
3. \( \pi_{A,E}(R \Join S) \)
4. \( \pi_C(S) - \pi_C(R) \)
5. \( \rho_{D\rightarrow B}(\pi_D(S)) \cup \pi_B(R) \)
6. \( \rho_{D\rightarrow B}(\pi_D(S)) \cap \pi_B(R) \)
7. select B, count(*) from R group by B
Bibliography

- Web site: http://www.w3.org/XML/Query
- XQuery processor:
  - Galax: http://www.galaxquery.org/
  - Saxon: http://saxon.sourceforge.net/
- XML/XQuery Systems:
  - eXist: http://exist.sourceforge.net/
  - MonetDB/XML: http://www.monetdb.org/XQuery
  - Oracle/BerkeleyDB: http://www.oracle.com/technetwork/database/berkeleydb/overview/
Part III

XML Storage and Query Evaluation
Outline

XML Databases

XML Storage Schemes

XML Extended Relational DBMS

XPath Evaluation

XQuery Evaluation

Bibliography

Exercises
XML Databases

Issues
Query and update large (collections of) XML documents with an expressive functional query language including path expressions over trees.

Database Approach

- apply/extend existing database techniques for storing and querying XML data
- define new set-oriented data structures, algorithms and operators

Course Outline

- compare different solutions for storing and querying XML documents
- study evaluation and optimisation techniques for XPath and XQuery.
Outline

XML Databases

XML Storage Schemes

XML Extended Relational DBMS

XPath Evaluation

XQuery Evaluation

Bibliography

Exercises
Hypothesis
The internal document representation strongly influences the query evaluation process.

Problem
Find optimal representation exploiting available constraints:
- document structure
- XML Schema
- query workload
- user defined mappings
12 - XML Storage Schemes

Storing XML Documents

Relational Storage Schemas
Schema oblivious storage
Schema-based encoding: Inlining
XML Documents

An XML document can have different origins:

1. Database import/export:
   - Regular structure (database schema)
   - Strongly typed data
   - Unordered trees
   - Select-project-join queries, order-by, group-by

2. Data integration:
   - Irregular structure resulting from integration
   - Unordered trees
   - Transformation and matching queries

3. Document collections:
   - Irregular structure with important text segments
   - List / ordered tree semantics
   - Filter-project queries
Storing XML documents

Hypothesis

- Storage techniques depend on the application needs and the document types.
- Unique optimal technique for all applications does not exist.

Issues

- choose **storage model**: relational, object-relational, native
- define **storage schema**: detailed description of the translation from XML to the target model.
Choosing a storage model

Criteria of Choice

- query translation and execution cost
- document import (parsing) and export (serialization) cost
- data size on disk and in memory
- complexity of updating documents
- evolution (data size and structure)
- integration with existing applications
- degree of schema independence and genericity

There exists no single optimal solution for all these criteria.
Target model 1: flat files

Advantages

▶ efficient import and export (no parsing, no serialization)
▶ reasonable size, powerful compression techniques
▶ schema independent

Issues

▶ updating large documents is costly (read/write whole file)
▶ structured access needs parsing (DOM/SAX)
▶ limited querying: grep, full-text indexing
Target model 2: relational database

Advantages

▶ reuse standard database technology:
  ▶ storage, indexing (B+-trees), security, coherency, transactions, query optimization, ...

▶ integration with existing applications and data:
  ▶ unique model and language

Issues

▶ mapping ordered trees to relations and vice versa
▶ translating XML queries into SQL
Target model 3: extended relational database

Relational database extended with an Abstract Data Type (ADT) for XML:

- attributes of type XML
- SQL queries with XML operations (XPath expressions)

Advantages

- same as relational database solution (reuse of technology, integration with existing data)
- optimized operations for XML

Issues

- coexistence of two models (trees and relations) and query languages (SQL, XPath)
Target model 4: Native XML Databases (NXD)

Native XML Databases (NXD):
- logical and physical tree model
- XML query language (XPath/XQuery)
- specialized physical layer: path indexing, hierarchical data organisation, memory management

Advantage
- efficient import/export/update
- adapted query optimization techniques

Issues
- limited integration with existing applications and data
- re-implement from scratch existing database functionalities
12 - XML Storage Schemes

Storing XML Documents

Relational Storage Schemas

Schema oblivious storage

Schema-based encoding: Inlining
Mapping XML ↔ tables

Use standard relational DBMS for storing XML documents:

- relational model (*first normal form*)
- SQL
- standard optimisation and indexing techniques

![Diagram of mapping XML ↔ tables]

- **import**
- **translate**
- **export**

XML Document ↔ tables

XPath / XQuery

SQL query

Database ↔ Database

Example: Pathfinder
Relational Databases and XML

Relational Database

- flat model / dense representation:
  - sets of tuples
  - tuple = object or association
- structured data with “a-priori” schema instantiation
- heterogeneity is a problem:
  - 3NF decomposition, NULL values

XML Document

- hierarchical model / sparse representation:
  - set of ordered trees
  - tree node = root of a tree containing other objects
- auto-descriptive weakly typed contents with “a-posteriori” schema validation
- heterogeneity is no problem:
  - flexible mode, natural representation of exceptions
Example

```xml
<?xml version='1.0'?>
<biblio>
  <book>
    <title>Germinal</title>
    <author><first>Emile</first><last>Zola</last></author>
  </book>
  <article>
    <title>MARS: a system for publishing XML ...</title>
    <author><first>Alin</first><last>Deutsch</last></author>
    <affiliation>Univ. of California San Diego</affiliation>
    <author><first>Val</first><last>Tannen</last></author>
    <affiliation>Univ. of Pennsylvania</affiliation>
  </article>
</biblio>
```
Example
Query languages

**SQL**

- path = relational join
- limited recursion (SQL3)
- (unordered) set semantics
- query optimization is “simple”:
  - algebraic query rewriting
  - set semantics
  - “standard” index structures (B-tree, hashtable)

**XPath/XQuery**

- path = navigational tree traversal / pattern
- recursive navigation axis
- ordered tree model
- query optimization is complex:
  - data and text
  - ordered tree semantics
  - “new” index structures (path indexes)
12 - XML Storage Schemes

Storing XML Documents

Relational Storage Schemas

Schema oblivious storage

Schema-based encoding: Inlining
Example

Goal: store an ordered labeled tree (with identified nodes) without taking account of additional information (such as a schema) → schema oblivious.

Three encoding approaches:

- Adjancy (edge) based encoding
- Path based encoding
- Node based encoding
Adjacency-based encoding
## Schema “Edge”

<table>
<thead>
<tr>
<th>parent</th>
<th>pos</th>
<th>tag</th>
<th>type</th>
<th>id</th>
</tr>
</thead>
<tbody>
<tr>
<td>$o_0$</td>
<td>1</td>
<td>biblio</td>
<td>ref</td>
<td>$o_1$</td>
</tr>
<tr>
<td>$o_1$</td>
<td>1</td>
<td>book</td>
<td>ref</td>
<td>$o_2$</td>
</tr>
<tr>
<td>$o_1$</td>
<td>2</td>
<td>article</td>
<td>ref</td>
<td>$o_7$</td>
</tr>
<tr>
<td>$o_2$</td>
<td>1</td>
<td>title</td>
<td>cdata</td>
<td>$o_3$</td>
</tr>
<tr>
<td>$o_2$</td>
<td>2</td>
<td>author</td>
<td>ref</td>
<td>$o_4$</td>
</tr>
<tr>
<td>$o_4$</td>
<td>1</td>
<td>first</td>
<td>cdata</td>
<td>$o_5$</td>
</tr>
<tr>
<td>$o_4$</td>
<td>2</td>
<td>last</td>
<td>cdata</td>
<td>$o_6$</td>
</tr>
<tr>
<td>$o_7$</td>
<td>1</td>
<td>title</td>
<td>cdata</td>
<td>$o_8$</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>

### S

<table>
<thead>
<tr>
<th>id</th>
<th>val</th>
</tr>
</thead>
<tbody>
<tr>
<td>$o_3$</td>
<td>Germinal</td>
</tr>
<tr>
<td>$o_5$</td>
<td>Emile</td>
</tr>
<tr>
<td>$o_6$</td>
<td>Zola</td>
</tr>
<tr>
<td>$o_8$</td>
<td>MARS: a ...</td>
</tr>
<tr>
<td>$o_{10}$</td>
<td>Alin</td>
</tr>
<tr>
<td>$o_{11}$</td>
<td>Deutsch</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>
Edge: Query translation

Titles of books of Emile Zola

for $l$ in doc("biblio.xml")/biblio/book, where $l/author/last = "Zola"
return $l/title

SQL: 4 selections and 5 joins

select V2.val
from R B, R A, R T, R L, S V1, S V2
where B.tag = 'book' and A.tag = 'author' and T.tag = 'title' and L.tag = 'last' and B.id = A.par and A.id = L.par and B.id = T.par and L.id = V1.id and T.id = V2.id and V1.val = 'Zola';
Edge: Analyse

Advantages

- generic storage format
- minimal space usage

Drawbacks

- scans/auto-joins over a single big table $\Rightarrow$ partitioned edge: one table per element type and attribute
- many joins
- limited recursion
Path-based encoding
Path relations

Idea

Classify nodes by their *access path types*. Each path type corresponds to one or several *binary* tables

- (parent, child)
- (node, value)
- (node, attribute)
- (node, rank)
Example

XML Databases
XML Storage Schemes
Storing XML Documents
Relational Storage Schemas
Schema oblivious storage
Schema-based encoding: Inlining

XML Extended Relational DBMS

XPath Evaluation
Naïve Evaluation
Set-based XPath Evaluation

XQuery Evaluation
XQuery Processing
Example: Pathfinder

Bibliography
Exercises
Query Translation

**XQuery:**

```xml
for $l in doc("biblio.xml")/bibliography/book,
where $l/author/last = "Zola"
return $l/title
```

**SQL:** 1 selection and 4 joins

```sql
select V2.val
from bibliography.book.title A,
bibliography.book.author B,
bibliography.book.author.last C,
bibliography.book.author.last.val V1,
bibliography.book.title.val V2
where A.par = B.par
and B.id = C.par
and A.id = V1.id
and C.id = V2.id
and V2.Val = 'Zola';
```
Path-relations: Analysis

Advantages

▶ match path expressions with table names
▶ natural handling of exceptions (small relations)
▶ corresponds to vertically partitioned databases (Monet/DB)

Drawback

▶ irregularly structured documents: the number of relations increases linearly with the size of the document

Variants

▶ vectorization: use main memory skeleton indexes and store only tables for values (text)
Node-based encoding
Recursive Queries

Issue
Evaluate *recursive* XPath axis in SQL:

- //article//first
- //book[id=5]/following-sibling::*

Recursion SQL3

dge(id, par, tag, pos)

```sql
with article - descendant(tag, id) as 
    (select b.tag, b.id 
     from edge a, edge b 
     where a.tag = 'article' and b.par = a.id) 
union all 
    (select b.tag, b.id 
     from article - descendant a, edge b 
     where a.id = b.par)
select id from article - descendant 
where tag = 'first';
```

213 / 278
Structural identifiers

Goal

- Efficiently check recursive structural relationships between two nodes $a$ and $b$

Solution: structural identifiers

- Define node labeling schemes (structural node identifiers) enabling the use of existing join algorithms and single-dimensional (B-Tree) or multi-dimensional (R-Tree) index structures.

Structural identifiers: examples

- Dewey encoding: $id(root) = 1$, $id(n/child[i]) = id(n).i$
- Region encoding: text positions of start and end tags
- Pre-Post encoding: $id(n) = (pre, post, prof)$ where
  - $pre =$ position of $n$ in the pre-order traversal of the tree
  - $post =$ position of $n$ in the post-order traversal of the tree
  - $prof =$ depth of $n$ in the tree
Pre-Post encoding

\[
\begin{array}{cc}
1,11,0 \\
2,4,1 \\
b \\
\downarrow \\
2,1,1 \\
\end{array}
\]

**XPath axes** | **where clause**
--- | ---
preceding \((n,n')\) | \(n.pre < n'.pre \land n.post < n'.post\)
ancestor \((n,n')\) | \(n.pre < n'.pre \land n.post > n'.post\)
descendant \((n,n')\) | \(n.pre > n'.pre \land n.post < n'.post\)
parent \((n,n')\) | \(\text{ancestor}(n, n') \land n.prof + 1 = n'.prof\)
preceding-sibling \((n,n')\) | \(n.pre < n'.pre \land \exists n'' (\text{parent}(n'', n) \land \text{parent}(n'', n'))\)
Merge-join with structural identifiers

Query: //c//d

Translation: \( c.pre < d.pre \land c.post > d.post \)

Evaluation: merge join over ordered \((pre, post)\) values: \(O(m + n)\)

![Diagram of XML tree with structural identifiers]

<table>
<thead>
<tr>
<th>//c:</th>
<th>pr</th>
<th>po</th>
<th>d</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>4</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td></td>
<td>6</td>
<td>7</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>8</td>
<td>6</td>
<td>2</td>
</tr>
<tr>
<td></td>
<td>9</td>
<td>10</td>
<td>1</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>//d:</th>
<th>pr</th>
<th>po</th>
<th>d</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>3</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td></td>
<td>7</td>
<td>5</td>
<td>2</td>
</tr>
<tr>
<td></td>
<td>10</td>
<td>8</td>
<td>2</td>
</tr>
</tbody>
</table>
Example: R-tree

XML Tree:

Pre/post encoding:

Example: R-tree
Example: R-tree

Figure 5: Leaf level of a *preorder packed* R-tree after loading an XML instance of 100 nodes (leaf capacity 6 nodes).
12 - XML Storage Schemes

Storing XML Documents
Relational Storage Schemas
Schema oblivious storage

Schema-based encoding: Inlining
Inlining

Principle

▶ Transform DTD into a **structure graph**;
▶ Use structure graph for deciding when an element type is transformed
  ▶ into a table with foreign key references to the parent table or
  ▶ into an attribute of the parent / some ancestor element table (inlining).
Transform DTD into Structure Graph

Two steps

1. flatten DTD (with loss of structural constraints):
   - \((e, e')^* \rightarrow e^*, e'^*\)
   - \(e^{**} \rightarrow e^*; e^?* \rightarrow e^*; \ldots\)
   - \(e^?\ldots e^* \rightarrow e^*, \ldots\)

2. generate for each potential root element type \(e\) a directed labeled graph \(G(N, E, e, \lambda)\) where
   - \(N\) contains all element types which might appear in the contents of \(e\)
   - \(e \in N\) is the root of the graph
   - there exists an arc \((a, b)\) if \(b\) appears in the content definition of \(a\)
   - \(\lambda(a, b) \in \{1, ?, *\}\) depending on the occurrence constraint of \(b\) in \(a\)
Structure Graph: Example

DTD

```xml
<!ELEMENT bibliography (book|article)* >
<!ELEMENT book (title, author, editor?) >
<!ELEMENT article (title, (author, affiliat?)*) >
<!ELEMENT author (first, last) >
<!ELEMENT title #PCDATA >
<!ELEMENT first #PCDATA >
<!ELEMENT last #PCDATA >
<!ELEMENT editor #PCDATA >
<!ELEMENT affiliat #PCDATA >
```

Flattened DTD

```xml
<!ELEMENT bibliography (book*,article*) >
<!ELEMENT book (title, author, editor?) >
<!ELEMENT article (title, author*, affiliat*) >
<!ELEMENT author (first, last) > ...
<!ELEMENT title #PCDATA >
```

...
Basic inlining

Given a directed structure graph \( G(N, E, R, \lambda) \) create

- a table \( R(id, pos) \) for the root \( R \) of the graph;
- a table \( A_B(id, pos, par) \) for each target node \( B \) of an arc \( (A, B) \) labeled by \( * \);
- an attribute \( B \) in table \( A \) for each path from nodes \( A \) to \( B \) where all arcs are labeled by 1 or \( ? \) (inlining);
- a table \( A_B(id, pos, par) \) for each target node \( B \) of an arc \( (A, B) \) where there also exists a path from \( B \) to \( A \) (recursive definitions)

Attributes:

- \( id \) denotes the node identifier,
- \( pos \) denotes the position in the list of siblings,
- \( par \) denotes a foreign key to the parent table.
Basic Inlining : Example

Sub-graph for bibliography :

- bibliography(id, pos)
- bibliography_book(id, pos, par, title, author, first, last, editor)
- bibliography_article(id, pos, par, title)
- article_author(id, pos, par, first, last)
- article_affil(id, pos, par, affiliat)
Basic Inlining: Query Translation

Articles of Zola:

```xml
for $l in doc("biblio.xml")//article
where $l/author/last = "Deutsch"
return $l/title
```

SQL: 1 join and 1 selection

```sql
select title
from bibliography_article a, article_author b
where a.id = b.par
and b.last = 'Deutsch'
```
Basic Inlining: Query Translation

Names of Authors

```xml
for $a in doc("biblio.xml")//author/last
return $a
```

```sql
SELECT last FROM bibliography_book
UNION
SELECT last FROM article_author;
```

Basic inlining:
- elements of the same type are scattered into several tables
  ⇒ many unions

Shared inlining:
- do not inline shared element types ⇒ less unions, but more joins.
Outline

XML Databases

XML Storage Schemes

XML Extended Relational DBMS

XPath Evaluation

XQuery Evaluation

Bibliography

Exercises
XML Extended Relational DBMS

- Oracle XML
- IBM DB2 XML Extender
- Microsoft OpenXML
- Pathfinder / MonetDB
- Excellon

Main characteristics:
- XML document import and database export
- Generic and schema-based storage schemas
- Object-relational SQL, new XML attribute type
Oracle XML (10g)

- W3C recommendation conformance: XML, XML Schema, XPath, XQuery
- Multiple storage schemas: object-relational, CLOB, native
- XML schema and data management
- Versioning, access control
- APIs: SQL, Java, PL/SQL, C, C++
Storage schemas

Native storage

Attributes of type XMLType:

- store XML fragments as attribute values
- SQL + XMLType method calls for
  - querying with XPath and XQuery
  - transforming tables into XML
  - indexing XML contents

Object-relational storage

- object-relational schema conform to a DTD
- canonical mapping
XMLType

XMLType
SQL object type for
- storing XML fragments in a relational table
- computing XML views of relational data

Defining tables

```sql
create table shakespeare(
  titre varchar2(32),
  filename varchar2(32) primary key,
  xml xmltype
);
```
Loading XML files

Load Macbeth and Hamlet

```sql
insert into shakespeare values ('Macbeth', 'macbeth.xml',
XMLType(BFILENAME('XML_DIR', 'macbeth.xml'),
nls_charset_id('AL32UTF8')));

insert into shakespeare values ('Hamlet', 'hamlet.xml',
XMLType(BFILENAME('XML_DIR', 'hamlet.xml'),
nls_charset_id('AL32UTF8')));
```

Create full-text index

```sql
drop index shakespeare_xml_ind;
create index shakespeare_xml_ind
on shakespeare(xml)
indextype is ctxsys.context;
```
XMLType Methods

Mapping XML to SQL

Functions for accessing XML values:

- `getClobValue(xml IN XMLType) return CLOB`
- `getStringValue(xml IN XMLType) return VARCHAR2`
- `getNumberValue(xml IN XMLType) return NUMBER`
- `existsNode(xml IN XMLType, xpath IN VARCHAR2, namespace IN VARCHAR2 := null) return NUMBER`
- `extract(xml IN XMLType, xpath IN VARCHAR2, namespace IN VARCHAR2 := null) RETURN XMLType;`
- `extractvalue (xml IN XMLTYPE, xpath IN VARCHAR2) RETURN VARCHAR2`
  - `extract(xml, xpath).get(String|Number)Val()`
XMLType Methods

Mapping SQL to XML

Functions for generating XML from relations:

- **XMLAgg**: aggregate a sequence into a single fragment
- **XMLConcat**: concatenate sequences
- **XMLElement**: create XML elements
- **XMLForest**: generate a sequence of XML elements
- **XMLColAttVal**: convert an attribute into XML
- **XMLSequence**: transform a single fragment into a sequence
- **XMLTransform**: apply XSLT stylesheet
**SQL/XPath queries**

**Play with title Macbeth**

```sql
select xml
from shakespeare
where extract(xml, '/PLAY/TITLE') like '%Macbeth%';
```

**Title of play containing a line with words “horse” and “kingdom”**

```sql
select extract(xml, '/PLAY/TITLE')
from shakespeare
where contains(xml, '{horse} AND {kingdom} INPATH (//LINE)') > 0;
```
SQL/XPath queries

Text spoken by Romeo or Juliet (filter XPath)

```sql
select extractValue(value(speech), '/SPEECH/SPEAKER')
   || ':' ||
   extractValue(value(speech), '/SPEECH/LINE')
from shakespeare, table(XMLSequence(
   extract(xml, '//SPEECH[SPEAKER="ROMEO"
   or SPEAKER="JULIET"]'))) speech
```
SQL/XPath queries

Text spoken by Romeo or Juliet (SQL)

```sql
select extract(value(speech), '/SPEECH/SPEAKER/text()')
|| ':' ||
extract(value(speech), '/SPEECH/LINE/text()')
from shakespeare, table(XMLSequence(
    extract(xml, '/SPEECH')))) speech
where extract(value(speech), '/SPEAKER/text()') like 'ROMEO'
or extract(value(speech), '/SPEAKER/text()') like 'JULIET'
```
SQL/XPath queries

Number of acts per play

```sql
select extractValue(xml, 'PLAY/TITLE'), count(value(a))
from shakespeare,
    table (XMLSequence(extract(xml, 'ACT'))) a
group by extractValue(xml, 'PLAY/TITLE');
```

Different roles in Romeo and Juliet

```sql
select extract(value(p), 'PERSONA')
from shakespeare,
    table (XMLSequence(extract(xml, 'PERSONA'))) p
where extractValue(xml, 'PLAY/TITLE') like 'Romeo'
order by extractValue(value(p), 'PERSONA');
```
Generating XML

Schema Tennis

```sql
create table Players(
    id number(8) primary key,
    name varchar(12),
    birthdate number(8),
    nationality varchar(12));
```

```sql
create table Participations(
    id number(8) foreign key references Players,
    tournament varchar(15),
    year number(8),
    sponsor varchar(10));
```
Generating XML

Relational Schema

Players( id, name, birthdate, nationality)
Participants(id, tournament, year, sponsor)

DTD

```xml
<!ELEMENT player (name, birthdate, nationality) >
<!ATTLIST player id ID >
```

Query

```sql
select xmlelement("PLAYER",
    xmlattributes(id),
    xmlforest(name, birthdate, nationality)
)
from Player;
```
Generating XML

DTD

```xml
<!ELEMENT JOUEUR (NOM, PARTICIPATIONS) >
<!ATTLIST JOUEUR NUJOUEUR ID >
<!ELEMENT PARTICIPATIONS (TOURNOI)*>
<!ELEMENT TOURNOI EMPTY>
<!ATTLIST TOURNOI LIEU CDATA ANNEE CDATA >
```

Query

```sql
select xmlelement("JOUEUR",
    xmlattributes(id as NUJOUEUR),
    xmlelement(name as NOM),
    xmlelement("PARTICIPATIONS",
        (select xmlagg(xmlelement("TOURNOI",
            xmlattributes(tournament as "LIEU",
                year as "ANNEE")))
        from Participations
        where Participations.id = Player.id )))
from Player;
```
Object-relational storage

Object-relational storage of XML documents.

Source-schema directed generation

▶ Automatic schema generation from a source XML schema source
▶ Automatic validation and transformation of XMLType objects

Target-schema directed generation

▶ Transform XML documents into a canonical format corresponding to a target object-relation schema (XSLT program)
▶ Automatic import
Canonical Mapping

XML

<row>
  <Person>
    <Name>Toto</Name>
    <Addr>
      <City>Paris</City>
      <Street>...</Street>
    </Addr>
  </Person>
</row>

Type SQL

create type AddrType as OBJECT (  
  City VARCHAR2(32)  
  Street VARCHAR2(32)  
);
create table Person (  
  Name VARCHAR2(32)  
  Addr AddrType  
);
Canonical Mapping

Advantages

- Object-relational view of XML
- “Pure” SQL queries

Drawbacks

- Non-adapted for irregular documents
- Limited recursive expressions (path expressions)
## Conclusion: unstructured versus structured storage schemas

<table>
<thead>
<tr>
<th></th>
<th>Unstructured Storage</th>
<th>Structured storage</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Import/export</strong></td>
<td>fast</td>
<td>slow (transformation)</td>
</tr>
<tr>
<td><strong>Update</strong></td>
<td>slow (read/write whole document)</td>
<td>fast: in place, fragmentation</td>
</tr>
<tr>
<td><strong>Flexibility</strong></td>
<td>high: no constraint on structure</td>
<td>high: schema-dependent</td>
</tr>
<tr>
<td><strong>XML fidelity</strong></td>
<td>high: preservation of serialised version</td>
<td>low: order ?</td>
</tr>
<tr>
<td><strong>XPath evaluation</strong></td>
<td>naïve evaluation on DOM tree</td>
<td>SQL optimisation</td>
</tr>
<tr>
<td><strong>Indexing</strong></td>
<td>full-text indexing</td>
<td>B-trees, string</td>
</tr>
</tbody>
</table>

**XML Databases**

**XML Storage Schemes**
- Storing XML Documents
- Relational Storage Schemas
- Schema oblivious storage
- Schema-based encoding: Inlining

**XML Extended Relational DBMS**

**XPath**
- Evaluation
  - Naïve Evaluation
  - Set-based XPath Evaluation

**XQuery**
- Evaluation
  - XQuery Processing
  - Example: Pathfinder

**Bibliography**

**Exercises**
Outline

XML Databases

XML Storage Schemes

XML Extended Relational DBMS

XPath Evaluation

XQuery Evaluation

Bibliography

Exercises
14 - XPath Evaluation

Naïve Evaluation

Set-based XPath Evaluation
Naïve evaluation of XPath expression

An XPath expression is a sequence of steps $e = s_1 / s_2 / ... / s_n$ returning a nodeset $N_e$.

Nested loop evaluation (XQuery core)

```xml
for $N1$ at $p1$ in $N0 / s1$ return
for $N2$ at $p2$ in $N1 / s2$ return
for $N3$ at $p2$ in $N2 / s3$ return
...
... return $Nn$
```

Union has set semantics and maintains document order.
Example

Expression \( /\text{descendant::B/child::D} \)

- **descendant::B** evaluated on the document root
  \( N_0 = \text{[root]} \) (absolute expression) returns
  nodeset \( N_1 = [b_1, b_2] \)

- **child::D** evaluated on
  - \( b_1 \) returns \([d]\)
  - \( b_2 \) returns the empty sequence \([\text{]}\).

- **return** \( N_1 = [d] \)

Complexity : \( O(n^{|Q|}) \)

- \( n \) : document size in number of nodes
- \( |Q| \) : query length in number of steps
14 - XPath Evaluation

Naïve Evaluation

Set-based XPath Evaluation
Encoding trees and queries

Define a mapping \( R \) from query \( e(d) \) into the relational model:

- \( R(d) \) : set representation (table) of a document tree \( d \)
- \( R(e) \) : relational query such that \( e(d) = R(e)(R(d)) \)

Relational representation of a document \( d \)

\( R(d) = (N, root, Axes, Types) \) where:

- \( N = \{n_1, n_2, \ldots\} \) is set of nodes
- \( root \in N \) is the root node of \( d \)
- \( Axes = \{child, parent, \ldots\} \) contains a binary relation axis over \( N \) for each XPath axis
- \( Types = \{element, comment, A, \ldots\} \) contains a unary relation for each DOM type, element name and attribute name.

The order of a document might be found by axis child and following-sibling.
Document encoding

\[ N = \{ r, a, b_1, b_2, c, d \}, \]
\[ child = \{ (r, a), (a, b_1), (a, b_2), (a, c), (b_1, d) \}, \]
\[ descendant = child \cup \{ (r, b_1), (r, b_2), (r, c), \ldots \}, \]
\[ parent = \{ (b_1, a), (b_2, a), (c, a), (d, b_1) \}, \ldots \]
\[ A = \{ a \}, \]
\[ B = \{ b_1, b_2 \}, \]
\[ C = \{ c \}, \]
\[ D = \{ d \} \]
Query encoding

Core XPath
XPath without arithmetic/string operations.

Relational representation of an XPath expression

- $\mathcal{R}(\mathbin{/}) = \{root\}
- $\mathcal{R}(\text{exp}/\text{Axes} :: \text{Filter}) = (\mathcal{R}(\text{exp}) \times \text{Axes}) \cap \mathcal{R}(\text{Filter})$
- $\mathcal{R}(\text{exp}_1[\text{exp}_2]) = \mathcal{R}(\text{exp}_1) \cap \mathcal{R}(\text{exp}_2^-)$
- $\mathcal{R}(\text{exp or exp'}) = \mathcal{R}(\text{exp}) \cup \mathcal{R}(\text{exp'})$
- $\mathcal{R}(\text{not(exp)}) = \mathcal{R}(\ast) - \mathcal{R}(\text{exp}^-)$

$\text{exp}^-$ denotes the inverse of expression $\text{exp}$: returns all nodes $n$ such that $\text{exp}$ applied on $n$ is not empty.
Core XPath Evaluation

/\textit{descendant::A/B[C/D or not(following::*)]}

Core XPath queries can be evaluated in \textit{linear space and time}: $O(n \times |Q|)$
Skeletons/Data Guides

Idea
Compress document structure for in-memory processing.

Principle
Reduce structural complexity and redundancy by sharing common sub-structures:

- folding: XML tree $\Rightarrow$ directed graph (skeleton/data guide)
- in-memory evaluation of XPath expression with partial graph unfolding.
Skeletons/Data Guides

(a)

(b)

(c)

XML Databases
XML Storage Schemas
Storing XML Documents
Relational Storage Schemas
Schema oblivious storage
Schema-based encoding: Inlining
XML Extended Relational DBMS
XPath Evaluation
Naïve Evaluation
Set-based XPath Evaluation
XQuery Evaluation
XQuery Processing
Example: Pathfinder
Bibliography
Exercises
Observations

- Folding and unfolding is linear in the size $n$ of the document.
- The maximal compression factor is $\log(\log(n))$.
- The document order is preserved.
- Partial decompression is possible.
Evaluating Core XPath queries

Goal
Unfold only the sub-graphs necessary for evaluating the expression.

Solution
- traverse graph from the root by visiting each node at most once.
- split node only if necessary

Complexity
- in the worst case, at each the size of the queried instance is doubled.
- exponential \(2^{|Q|}\) in the size of query \(Q\)
Unfolding: example
Outline

XML Databases

XML Storage Schemes

XML Extended Relational DBMS

XPath Evaluation

XQuery Evaluation

Bibliography

Exercises
15 - XQuery Evaluation

XQuery Processing
Example: Pathfinder
XQuery and XPath

XPath
Path expressions for extracting node sequences from trees:
- Input: ordered labeled *XML trees*
- Output: *document node sequence* (document order)

XQuery
Typed functional query/programming language:
- Based on XPath for *generating sequences from trees*
- Input/output: sequence of XML Schema items
(Abstract) XQuery Processing Model (W3C)

External Processing
Data Model Generation

XML Parse and optionally validate

InfoSet/PSVI Generate Data Model

Data Model Instances

Query Processing
Static analysis phase

XQuery Parse query

Op-Tree Process query

Query Prolog

(SQ2) Initialize from environment

Dynamic evaluation phase

(DQ1) Access Op-Tree

Execution Engine

(SQ5) Normalize

(DQ4) Access and create

Dynamic context

(SQ3) Process

(SQ1) Process

(SQ4) Resolve names

Static context

(SQ5) Static Type Check*

(SQ2) Initialize

(SQ1) Parse query

Schema Import Processing

XSD Generate

In-scope schema definitions

* Only if static typing enabled
** Dynamic type check if static typing not enabled
*** Need not be well-formed XML

XML Databases
XML Storage
Schemes
Storing XML Documents
Relational Storage Schemas
Schema oblivious storage
Schema-based encoding: Inlining

XML Extended Relational DBMS

XPath Evaluation
Naïve Evaluation
Set-based XPath Evaluation

XQuery Evaluation

Example: Pathfinder

Bibliography
Exercises
(Abstract) XQuery Processing Model (W3C)

Processing steps

1. Analyse query and document syntax
2. Build a static query context (query without document)
3. Translate into XQuery core execution plan → Normalization
4. Build a dynamic context (query and document)
5. Evaluate query

Optional steps

1. Import XML schemas and translate into XPath/XQuery type system
2. Static type inference and type error detection
3. Result serialisation
Normalization : XQuery Core

XQuery Core

- Uniform nested-loop representation using additional temporary variables (for / let)
- Useful for compare and transforming queries

XQuery

doc("cours/bib.xml")//author

XQuery Core

let $w_2 := doc("cours/bib.xml") return for $v_2 at $v_3 in $w_2 return let $w_4 := $w_2/descendant-or-self::xs:AnyNode return for $v_6 at $v_7 in $w_4 return $v_6/child::author
XQuery Evaluation

After Normalization

- **logical optimization**: use static type info for logical optimizations
- **physical query plan generation**: use extended physical algebra over a chosen data representation (native, relations, extended relational) for further optimisations (see Storage Schemas).
- **physical optimization**: with/without cost-model, use standard (relational) indexes and optimizer, XML indexing,
- **query evaluation**: native, SQL
15 - XQuery Evaluation

XQuery Processing
Example: Pathfinder
Example: Pathfinder

Extended Relational Algebra

- node-based relational encoding:
  - single table with structural identifiers
- extended relational operators:
  - serialize
  - project-rename ($\pi$), selection ($\sigma$), join ($\bowtie$), Cartesian product ($\times$), duplicate-elimination ($\delta$)
  - attach ($\varrho$): constant column, row rank, id
- expressions can be translated into SQL
Join graph isolation (Pathfinder)

**Issue**

- Blocking operators (duplicate elimination \(\delta\), attach row rank \(\varrho\)) obstruct join operator movements and planning.
- *Staged plan execution* with temporary materializations.

**Join graph isolation**

- move blocking operators (duplicate elimination, attach row rank) to plan tail.
- push joins down to obtain a join graph without blocking operators.
Example

XQuery

doc("auction.xml")/descendant::open_auction[bidder]

Core XQuery

for $x in fs:ddo(doc("auction.xml")
    /descendant::open_auction)
return if (fn:boolean(fs:ddo($x/child::bidder)))
    then $x else () .
Final optimal plan
Final SQL Query

```sql
SELECT DISTINCT d2.pre
FROM doc AS d1, doc AS d2, doc AS d3
WHERE d1.kind = DOC
  AND d1.name = 'auction.xml'
  AND d2.kind = ELEM
  AND d2.name = 'open_auction'
  AND d2 BETWEEN d1.pre + 1 AND d1.pre + d1.size
  AND d3.kind = ELEM
  AND d3.name = 'bidder'
  AND d3 BETWEEN d2.pre + 1 AND d2.pre + d2.size
  AND d2.level + 1 = d3.level
ORDER BY d2.pre
```
Outline

XML Databases

XML Storage Schemes

XML Extended Relational DBMS

XPath Evaluation

XQuery Evaluation

Bibliography

Exercises
XML Storage

- Sihem Amer-Yahia and Mary Fernandez, Overview of Existing XML Storage Techniques, Sigmod Record 2002


- S. Munch, Building Oracle XML Applications
Outline

XML Databases
XML Storage Schemes
XML Extended Relational DBMS
XPath Evaluation
XQuery Evaluation
Bibliography
Exercises
Exercises

Bibliography

Query Evaluation

- Torsten Grust, Manuel Mayr, Jan Rittinger: Let SQL drive the XQuery workhorse (XQuery join graph isolation). EDBT 2010: 147-158
- Peter Buneman, Martin Grohe, Christoph Koch, Path Queries on Compressed XML, International Conference on Very Large Data Bases(VLDB’03), 141-152