XML Data Management

Peter Wood

BBK

Outline

Chapter 1

Introduction

What is XML?

- The eXtensible Markup Language (XML) defines a generic syntax used to mark up data with simple, human-readable tags
- Has been standardized by the World Wide Web Consortium (W3C) as a format for computer documents
- Is flexible enough to be customized for domains as diverse as:
  - Web sites
  - Electronic data interchange
  - News feeds (RSS, e.g., BBC World News)
  - Vector graphics
  - Mathematical expressions
  - Microsoft Word documents
  - Music libraries (e.g., iTunes)
  - ...

What is XML? (2)

- Data in XML documents is represented as strings of text
- This data is surrounded by text markup, in the form of *tags*, that describes the data
- A particular unit of data and markup is called an *element*
- XML specifies the exact syntax of how elements are delimited by tags, what a tag looks like, what names are acceptable, and so on

Which is Easier to Understand?

```
TCP/IP
Stevens
Foundations of Databases
Abiteboul
Hull
Vianu
The C Programming Language
Kernighan
Ritchie
Prentice Hall
...
```

```
<bib>
  <book>
    <title>TCP/IP</title>
    <author>Stevens</author>
  </book>
  <book>
    <title> ... </title>
    ...
  </book>
</bib>
```

XML vs. HTML

- Markup in an XML document looks similar to that in an HTML document
- However, there are some crucial differences:
  - XML is a meta-markup language: it doesn't have a *fixed* set of tags and elements
  - To enhance interoperability, people may agree to use only certain tags (as defined in a DTD or an XML Schema — see later)
  - Although XML is flexible in regard to elements that are allowed, it is strict in many other respects (e.g., closing tags are required)
  - Markup in XML only describes a document's structure; it doesn't say anything about how to display it

Very Brief Review of HTML

- A document structure and hypertext specification language
- Specified by the World Wide Web Consortium (W3C)
- Designed to specify the *logical structure* of information
- Intended for presentation as *Web pages*
- Text is marked up with *tags* defining the document's logical units, e.g.
  - title
  - headings
  - paragraphs
  - lists
  - ...
- The displayed properties of the logical units are determined by the browser (and stylesheet, if present)

HTML Example

- The following is a (very simple) complete HTML document:

```
<html>
  <head>
    <title>A Title</title>
  </head>
  <body>
    <h1>A Heading</h1>
  </body>
</html>
```

- When loaded in a browser
  - “A Title” will be displayed in the title bar of the browser
  - “A Heading” will be displayed big and bold as the page contents

HTML, XHTML and XML

- These days, most web pages use *XHTML* rather than HTML
- XHTML uses the syntax of XML
- XHTML corresponds to a particular XML *vocabulary* or *document type*
- A document type is specified using a *Document Type Definition* (*DTD*) — see later
- HTML is essentially a less strict form of XHTML

Limitations of (X)HTML

So why not use XHTML rather than XML?

- (X)HTML defines a *fixed set* of elements (XHTML is *one* XML vocabulary)
- elements have *document* structuring semantics
- for presentation to human readers
- organisations want to be able to define their own elements
- applications need to exchange structured *data* too
- applications cannot consume (X)HTML easily
- use XML for *data* exchange and (X)HTML for document representation

XML versus Relational Data

- Why not use data from relational databases for exchange?
- XML is more flexible:
  - XML data is *semi-structured* rather than structured
  - Conformance of the data to a schema is not mandatory
  - XML schemas, if used, allow for more varied structures
- Relational data can always be (and often is) wrapped as XML

Motivating Example

- Say we want to store information about a personal CD library
- The CDs are all of classical music
- Some CDs contain simply solo (e.g., piano) works
- Some CDs have orchestral works (with orchestra, conductor)
- Some CDs contain performances of works by different composers
- We want to avoid repeating information in the descriptions
- We have only 4 CDs (see the next few slides)!

Example (1)

```
<CD-library>
   <CD number="724356690424">
      ...
   </CD>

   <CD number="419160-2">
      ...
   </CD>

   <CD number="449719-2">
      ...
   </CD>

   <CD number="430702-2">
      ...
   </CD>
</CD-library>
```

Example (2)

```
<CD number="724356690424">
  <performance>
    <composer>Frederic Chopin</composer>
    <composition>Waltzes</composition>
    <soloist>Dinu Lipatti</soloist>
    <date>1950</date>
  </performance>
</CD>
```

Example (3)

```
<CD number="419160-2">
  <composer>Johannes Brahms</composer>
  <soloist>Emil Gilels</soloist>
  <performance>
    <composition>Piano Concerto No. 2</composition>
    <orchestra>Berlin Philharmonic</orchestra>
    <conductor>Eugen Jochum</conductor>
    <date>1972</date>
  </performance>
  <performance>
    <composition>Fantasias Op. 116</composition>
    <date>1976</date>
  </performance>
</CD>
```

Example (4)

```
<CD number="449719-2">
  <soloist>Martha Argerich</soloist>
  <orchestra>London Symphony Orchestra</orchestra>
  <conductor>Claudio Abbado</conductor>
  <date>1968</date>
  <performance>
    <composer>Frederic Chopin</composer>
    <composition>Piano Concerto No. 1</composition>
  </performance>
  <performance>
    <composer>Franz Liszt</composer>
    <composition>Piano Concerto No. 1</composition>
  </performance>
</CD>
```

Example (5)

```
<CD number="430702-2">
  <composer>Antonin Dvorak</composer>
  <performance>
    <composition>Symphony No. 9</composition>
    <orchestra>Vienna Philharmonic</orchestra>
    <conductor>Kirill Kondrashin</conductor>
    <date>1980</date>
  </performance>
  <performance>
    <composition>American Suite</composition>
    <orchestra>Royal Philharmonic</orchestra>
    <conductor>Antal Dorati</conductor>
    <date>1984</date>
  </performance>
</CD>
```

Future of XML

- XML offers the possibility of truly cross-platform, long-term data formats:
  - Much of the data from the original moon landings is now effectively lost
  - Even reading an older Word file might already be problematic
- XML is a very simple, well-documented data format
- Any tool that can read text files can read an XML document
- XML may be the most portable and flexible document format since the ASCII text file

Overview

- In these lectures we are going to look at
  - some basic notions of XML
  - how to define XML vocabularies (DTDs, XML schemas)
  - how to query XML documents (XPath, XQuery)
  - how to process these queries

Literature

- A. Møller and M. Schwartzbach. *An Introduction to XML and Web Technologies*. Addison Wesley, 2006.
- S. Abiteboul, I. Manolescu, P. Rigaux, M-C. Rousset and P. Senellart. *Web Data Management*. Cambridge University Press, 2012.
- E.R. Harold, W.S. Means. *XML in a Nutshell.* O'Reilly, 2004
- H. Katz (editor). *XQuery from the Experts.* Addison Wesley, 2004
- These slides . . .

Chapter 2

XML Fundamentals

Elements, Tags, and Data

- A very simple, yet complete, XML document:

```
<person>
   Alan Turing
</person>
```

- Composed of a single *element* whose name is `person`
- Element is delimited by the *start tag* `<person>` and the *end tag* `</person>`
- Everything between the start tag and end tag (exclusive) is the element's *content*

Elements, Tags, and Data (2)

- Content of the above element is the text string `Alan Turing`
- Whitespace is part of the content (although many applications choose to ignore it)
- `<person>` and `</person>` are *markup*,
- The string `Alan Turing` and surrounding whitespace are *character data*

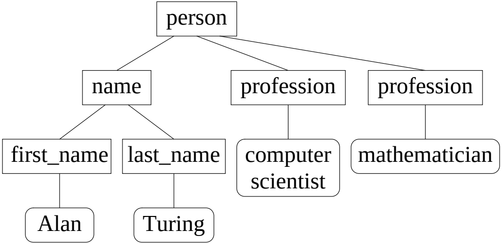
Elements, Tags, and Data (3)

- Special syntax for *empty elements*, elements without content
  - Each can be represented by a *single* tag that begins with < but ends with />
- XML is case sensitive, i.e. `<Person>` is not the same as `<PERSON>` (or `<person>`)

XML Documents and Trees

XML documents can be represented as trees

```
<person>
  <name>
    <first_name>Alan</first_name>
    <last_name>Turing</last_name>
  </name>
  <profession>
    computer scientist
  </profession>
  <profession>
    mathematician
  </profession>
</person>
```



XML Documents and Trees (2)

- The `person` element contains three *child* elements: one `name` and two `profession` elements
- The `person` element is called the *parent* element of these three elements
- An element can have an arbitrary number of child elements and the elements may be nested arbitrarily deeply
- Children of the same parent are called *siblings*
- Overlapping tags are prohibited, so the following is not possible:

```
<strong><em>
example from HTML
</strong></em>
```

XML Documents and Trees (3)

- Every XML document has one element without a parent
- This element is called the document's *root element* (sometimes called *document element*)
- The root element contains all other elements of a document

Attributes

- XML elements can have *attributes*
- An attribute is name-value pair attached to an element's start tag
- Names are separated from values by an equals sign
- Values are enclosed in single or double quotation marks
- Example:

```
<person born='1912/06/23' died='1954/06/07'>
  Alan Turing
</person>
```

- The order in which attributes appear is not significant

Attributes (2)

- We could model the contents of the original document as attributes:

```
<person>
  <name first='Alan' last='Turing'/>
  <profession value='computer scientist'/>
  <profession value='mathematician'/>
</person>
```

- This raises the question of when to use child elements and when to use attributes
- There is no simple answer

Attributes vs. Child Elements

- Some people argue that attributes should be used for metadata (about the element) and elements for the information itself
  - It's not always easy to distinguish between the two
- Attributes are limited in structure (their value is simply a string)
- There can also be no more than one attribute with a given name
- Consequently, an element-based structure is more flexible and extensible

Entities and Entity References

- Character data inside an element may not contain, e.g., a raw unescaped opening angle bracket <
- If this character is needed in the text, it has to be escaped by using the < *entity reference*
- lt is the *name* of the entity; & and ; delimit the reference
- XML predefines five entities:

lt	<
amp	&
gt	>
quot	"
apos	'

CDATA Sections

- When an XML document includes samples of XML or HTML source code, all `<`, `>`, and `&` characters must be encoded using entity references
- This replacement can become quite tedious
- To facilitate the process, literal code can be enclosed in a *CDATA section*
- Everything between `<![CDATA[` and `]]>` is treated as raw character data
- The only thing that cannot appear in a CDATA section is the end delimiter `]]>`

Comments

- XML documents can also be commented
- Similar to HTML comments, they begin with `<!--` and end with `-->`
- Comments may appear
  - anywhere in character data
  - before or after the root element
  - However, NOT inside a tag or another comment
- XML parsers may or may not pass along information found in comments

Processing Instructions

- In HTML, comments are sometimes abused to support nonstandard extensions (e.g., server-side includes)
- Unfortunately,
  - comments may not survive being passed through several different HTML editors and/or processors
  - innocent comments may end up as input to an application
- XML uses a special construct to pass information on to applications: a *processing instruction*
- It begins with `<?` and ends with `?>`
- Immediately following the `<?` is the target (possibly the name of the application)

Processing Instructions (2)

Examples:

- Associating a stylesheet with an XML document:

```
<?xml-stylesheet type="text/xsl" href="style.xsl"?>
```

- Embedded PHP in (X)HTML:

```
<?php
  mysql_connect("database...",
                "user",
                "password");
  ...
  mysql_close();
?>
```

XML Declaration

- The *XML declaration* looks like a processing instruction, but only gives some information about the document:

```
<?xml version='1.0'
      encoding='US-ASCII'
      standalone='yes'?>
```

- *version*: at the moment 1.0 and 1.1 are available (we focus on 1.0)
- *encoding*: defines the character set used (e.g. ASCII, Latin-1, Unicode UTF-8)
- *standalone*: determines if some other file (e.g. DTD) has to be read to determine proper values for parts of the document

Well-Formedness

A *well-formed* document observes the syntax rules of XML:

- Every start tag must have a matching end tag
- Elements may not overlap
- There must be exactly one root element
- Attribute values must be quoted
- An element may not have two attributes with the same name
- Comments and processing instructions may not appear inside tags
- No unescaped < or & signs may occur in character data

Well-Formedness (2)

- XML names must be formed in certain ways:
  - May contain standard letters and digits 0 through 9
  - May include the punctuation characters underscore (_), hyphen (-), and period (.)
  - May only start with letters or the underscore character
  - There is no limit to the length
- The above list is not exhaustive; for a complete list look at the W3C specification
- A parser encountering a non-well-formed document will stop its parsing with an error message

XML Namespaces

- MathML is an XML vocabulary for mathematical expressions
- SVG (Scalable Vector Graphics) is an XML vocabulary for diagrams
- say we want to use XHTML, MathML and SVG in a single XML document
- how does a browser know which element is from which vocabulary?
- e.g., both SVG and MathML define a `set` element
- we shouldn't have to worry about potential name clashes
- we should be able to specify different *namespaces*, one for each of XHTML, MathML and SVG

The namespaces solution

- The solution is to *qualify* element names with *URIs*
- A URI (Universal Resource Identifier) is usually used for *identifying* a resource on the Web
- (A Uniform Resource Locator (URL) is a special type of URI)
- A *qualified name* then consists of two parts:
  `namespace:local-name`
- e.g., `<http://www.w3.org/2000/svg:circle ... />`
- where `http://www.w3.org/2000/svg` is a URI and namespace
- The URI does *not* have to reference a real Web resource
- URIs only disambiguate names; they don't have to define them
- In this case, the browser knows the SVG namespace and behaves accordingly

Namespace declarations

- using URIs everywhere is very cumbersome
- so namespaces are used indirectly using
  - namespace *declarations* and
  - associated *prefixes* (user-specified)

```
<... xmlns:svg="http://www.w3.org/2000/svg">
   <p>A circle looks like this
   ...
      <svg:circle ... />
   ...
</...>
```

- The `xmlns:svg` attribute
  - declares the namespace `http://www.w3.org/2000/svg`
  - associates it with prefix `svg`

Scope of namespace declarations

- the *scope* of a namespace declaration is
  - the element containing the declaration
  - and all its *descendants* (those elements nested inside the element)
  - can be overridden by *nested* declarations
- both elements and attributes can be qualified with namespaces
- unprefixed element names are assigned a *default* namespace
- default namespace declaration: `xmlns="`*URI*`"`

Namespaces example

```
<html xmlns="http://www.w3.org/1999/xhtml"
      xmlns:svg="http://www.w3.org/2000/svg">
   ...
   <p>A circle looks like this
      <svg:svg ... >
         ...
         <svg:circle ... />
         ...
      </svg:svg>
      and has
      ...
   </p>
</html>
```

- `html` and `p` are in the *default* namespace (`http://www.w3.org/1999/xhtml`)

Namespaces example (2)

```
<html xmlns="http://www.w3.org/1999/xhtml"
      xmlns:svg="http://www.w3.org/2000/svg">
   ...
   <p>A circle looks like this
      <svg:svg ... >
         ...
         <svg:circle ... />
         ...
      </svg:svg>
      and has
      ...
   </p>
</html>
```

- namespace for `svg` and `circle` is `http://www.w3.org/2000/svg`
- note that `svg` is used both as a prefix and as an element name

Summary

- This chapter gave a brief summary of XML
- Only the most important aspects (which are needed later on) were covered

Chapter 3

Document Type Definitions

Document Types

- A *document type* is defined by specifying the constraints which any document which is an *instance* of the type must satisfy
- For example,
  - in an HTML document, one paragraph cannot be nested inside another
  - in an SVG document, every `circle` element must have an `r` (radius) attribute
- Document types are
  - useful for restricting authors to use particular representations
  - important for correct processing of documents by software

Languages for Defining Document Types

- There are many languages for *defining* document types on the Web, e.g.,
  - document type definitions (DTDs)
  - XML schema definition language (XSDL)
  - relaxNG
  - schematron
- We will consider the first three of these

Document Type Definitions (DTDs)

- A DTD defines a *class* of documents
- The structural constraints are specified using an *extended context-free grammar*
- This defines
  - *element* names and their allowed contents
  - *attribute* names and their allowed values
  - *entity* names and their allowed values

Valid XML

- A *valid* XML document
  - is well-formed and
  - has been validated against a DTD
  - (the DTD is specified in the document — see later)

DTD syntax

- The syntax for an element declaration in a DTD is:

  `<!ELEMENT` *name* `(` *model* `)` `>`

  where
  - `ELEMENT` is a keyword
  - *name* is the element name being declared
  - *model* is the element *content model* (the allowed contents of the element)
- The content model is specified using a *regular expression* over element names
- The regular expression specifies the permitted *sequences* of element names

Examples of DTD element declarations

- An `html` element must contain a `head` element followed by a `body` element:

  ```
  <!ELEMENT html     (head, body) >
  ```

  where "," is the *sequence* (or concatenation) operator
- A `list` element (not in HTML) must contain either a `ul` element or an `ol` element (but not both):

  ```
  <!ELEMENT list     (ul|ol) >
  ```

  where "|" is the *alternation* (or "exclusive or") operator
- A `ul` element must contain zero or more `li` elements:

  ```
  <!ELEMENT ul       (li)* >
  ```

  where "*" is the *repetition* (or "Kleene star") operator

DTD syntax (1)

In the table below:

- `b` denotes any element name, the simplest regular expression
- α and β denote regular expressions

DTD Syntax	Meaning
`b`	element `b` must occur
α	elements must match α
(α)	elements must match α
α , β	elements must match α followed by β
α | β	elements must match either α or β (not both)
α*	elements must match zero or more occurrences of α

DTD syntax (2)

DTD Syntax	Meaning
α+	one or more sequences matching α must occur
α?	zero or one sequences matching α must occur
EMPTY	no element content is allowed
ANY	any content (of declared elements and text) is allowed
#PCDATA	content is text rather than elements

- α+ is short for (α,α*)
- α? is short for (α|EMPTY)
- #PCDATA stands for "parsed character data," meaning an XML parser should parse the text to resolve character and entity references

RSS

- RSS is a simple XML vocabulary for use in news feeds
- RSS stands for *Really Simple Syndication*, among other things
- The root (document) element is `rss`
- `rss` has a single child called `channel`
- `channel` has a `title` child, any number of `item` children (and others)
- Each `item` (news story) has a `title`, `description`, `link`, `pubDate`, ...

RSS Example Outline

```
<rss version="2.0">
  <channel>
    <title> BBC News - World </title>
      ...
    <item>
      <title> Hollande becomes French president </title>
        ...
    </item>
    <item>
      <title> New Greece poll due as talks fail </title>
        ...
    </item>
    <item>
      <title> EU forces attack Somalia pirates </title>
    </item>
      ...
  </channel>
</rss>
```

RSS Example Fragment (channel)

```
<channel>
  <title> BBC News - World </title>
  <link>http://www.bbc.co.uk/news/world/...</link>
  <description>The latest stories from the World section of
             the BBC News web site.</description>
  <lastBuildDate>Tue, 15 May 2012 13:42:05 GMT</lastBuildDate>
  <ttl>15</ttl>
  ...
</channel>
```

RSS Example Fragment (first item)

```
<item>
  <title>Hollande becomes French president</title>
  <description>Francois Hollande says he is fully aware
    of the challenges facing France after being sworn
    in as the country’s new president.</description>
  <link>http://www.bbc.co.uk/news/world-europe-...</link>
  <pubDate>Tue, 15 May 2012 11:44:17 GMT</pubDate>
  ...
</item>
```

RSS Example Fragment (second item)

```
<item>
  <title>New Greece poll due as talks fail</title>
  <description>Greece is set to go to the polls again
    after parties failed to agree on a government for
    the debt-stricken country, says Socialist leader
    Evangelos Venizelos.</description>
  <link>http://www.bbc.co.uk/news/world-europe-...</link>
  <pubDate>Tue, 15 May 2012 13:52:38 GMT</pubDate>
  ...
</item>
```

RSS Example Fragment (third item)

```
<item>
  <title>EU forces attack Somalia pirates</title>
  <description>EU naval forces conduct their first raid
    on pirate bases on the Somali mainland, saying they
    have destroyed several boats.</description>
  <link>http://www.bbc.co.uk/news/world-africa-...</link>
  <pubDate>Tue, 15 May 2012 13:19:51 GMT</pubDate>
  ...
</item>
```

Simplified DTD for RSS

```
<!ELEMENT rss           (channel)>
<!ELEMENT channel       (title, link, description,
                         lastBuildDate?, ttl?, item+)>
<!ELEMENT item          (title, description, link?, pubDate?)>
<!ELEMENT title         (#PCDATA)>
<!ELEMENT link          (#PCDATA)>
<!ELEMENT description   (#PCDATA)>
<!ELEMENT lastBuildDate (#PCDATA)>
<!ELEMENT ttl           (#PCDATA)>
<!ELEMENT pubDate       (#PCDATA)>
```

Validation of XML Documents

- Recall that an XML document is called *valid* if it is well-formed and has been validated against a DTD
- Validation is essentially checking that the XML document, viewed as a tree, is a *parse tree* in the language specified by the DTD
- We can use the W3C validator service
- Each of the following files has the same DTD specified (as on the previous slide):
  - `rss-invalid.xml` giving results
  - `rss-valid.xml` giving results

Referencing a DTD

- The DTD to be used to validate a document can be specified
  - internally (as part of the document)
  - externally (in another file)
- done using a *document type declaration*
- *declare* document to be of type given in DTD
- e.g., `<!DOCTYPE rss ... >`

Declaring an Internal DTD

```
<?xml version="1.0"?>
<!DOCTYPE rss [
    <!-- all declarations for rss DTD go here -->
    ...
    <!ELEMENT rss ... >
    ...
]>
<rss>
   <!-- This is an instance of a document of type rss -->
   ...
</rss>
```

- element `rss` must be defined in the DTD
- name after `DOCTYPE` (i.e., `rss`) must match root element of document

Declaring an External DTD (1)

```
<?xml version="1.0"?>
<!DOCTYPE rss SYSTEM "rss.dtd">
<rss>
   <!-- This is an instance of a document of type rss -->
   ...
</rss>
```

- what follows `SYSTEM` is a *URI*
- `rss.dtd` is a relative URI, assumed to be in same directory as source document

Declaring an External DTD (2)

```
<?xml version="1.0"?>
<!DOCTYPE math PUBLIC "-//W3C//DTD MathML 2.0//EN"
     "http://www.w3.org/TR/MathML2/dtd/mathml2.dtd">
<math>
   <!-- This is an instance of a mathML document type -->
   ...
</math>
```

- `PUBLIC` means what follows is a *formal public identifier* with 4 fields:
    1. `ISO` for ISO standard, + for approval by other standards body, and - for everything else
    2. *owner* of the DTD: e.g., `W3C`
    3. *title* of the DTD: e.g., `DTD MathML 2.0`
    4. *language* abbreviation: e.g., `EN`
- URI gives location of DTD

More on RSS

- The RSS 2.0 specification actually states that, for each `item`, *at least one of* `title` or `description` must be present
- How can we modify our previous DTD to specify this?

More on RSS

- The RSS 2.0 specification actually states that, for each `item`, *at least one of* `title` or `description` must be present
- How can we modify our previous DTD to specify this?
- The allowed sequences are:
  1. `title`
  2. `title description`
  3. `description`

More on RSS

- The RSS 2.0 specification actually states that, for each `item`, *at least one of* `title` or `description` must be present
- How can we modify our previous DTD to specify this?
- The allowed sequences are:
    1. `title`
    2. `title description`
    3. `description`
- So what about the following regular expression?

```
title | (title, description) | description
```

Non-Deterministic Regular Expressions

- The regular expression

  `title | (title, description) | description`

  is non-deterministic
- This means that a parser must read ahead to find out which part of the regular expression to match
- e.g., given a `title` element in the input, should a parser try to match
  - `title` or
  - `title description`

Non-Deterministic Regular Expressions

- The regular expression

  `title | (title, description) | description`

  is non-deterministic
- This means that a parser must read ahead to find out which part of the regular expression to match
- e.g., given a `title` element in the input, should a parser try to match
  - `title` or
  - `title description`
- It needs to read the next element to check whether or not it is `description`

Non-Deterministic vs Deterministic Regular Expressions

- Non-deterministic regular expressions are *forbidden* by DTDs and XSDL
- They are allowed by RelaxNG
- A non-deterministic regular expression can always be rewritten to be deterministic

Non-Deterministic vs Deterministic Regular Expressions

- Non-deterministic regular expressions are *forbidden* by DTDs and XSDL
- They are allowed by RelaxNG
- A non-deterministic regular expression can always be rewritten to be deterministic
- e.g.,

  ```
  title | (title, description) | description
  ```

  can be rewritten as

  ```
  (title, description?)  | description
  ```

Non-Deterministic vs Deterministic Regular Expressions

- Non-deterministic regular expressions are *forbidden* by DTDs and XSDL
- They are allowed by RelaxNG
- A non-deterministic regular expression can always be rewritten to be deterministic
- e.g.,

  ```
  title | (title, description) | description
  ```

  can be rewritten as

  ```
  (title, description?)  | description
  ```
- The rewriting may cause an exponential increase in size

Attributes

- Recall that attribute name-value pairs are allowed in start tags, e.g., `version="2.0"` in the `rss` start tag
- Allowed attributes for an element are defined in an *attribute list declaration*: e.g., for `rss` and `guid` elements

```
<!ATTLIST rss
  version CDATA #FIXED "2.0" >
<!ATTLIST guid
  isPermaLink (true|false) "true" >
```

- attribute definition comprises
  - *attribute name*, e.g., `version`
  - *type*, e.g., `CDATA`
  - *default*, e.g., `"true"`

Some Attribute Types

- `CDATA`: any valid character data
- `ID`: an identifier unique within the document
- `IDREF`: a reference to a unique identifier
- `IDREFS`: a reference to several unique identifiers (separated by white-space)
- `(a|b|c)`, e.g.: (*enumerated attribute type*) possible values are one of `a`, `b` or `c`
- ...

Attribute Defaults

- `#IMPLIED`: attribute may be omitted (optional)
- `#REQUIRED`: attribute must be present
- `#FIXED "x"`, e.g.: attribute optional; if present, value must be `x`
- `"x"`, e.g.: value will be `x` if attribute is omitted

Mixed Content

- In `rss`, the content of each element comprised either only other elements or only text
- In HTML, on the other hand, paragraph elements allow text interleaved with various in-line elements, such as `em`, `img`, `b`, etc.
- Such elements are said to have *mixed content*
- How do we define mixed content models in a DTD?

Mixed Content Models

- Say we want to mix text with elements `em`, `img` and `b` as the allowed contents of a `p` element
- The DTD content model would be as follows:

```
<!ELEMENT p (#PCDATA | em | img | b)* >
```

  - `#PCDATA` must be first (in the definition)
  - It must be followed by the other elements separated by `|`
  - The subexpression must have `*` applied to it
- These restrictions limit our ability to constrain the content model (see XSDL later)

Entities

- An *entity* is a physical unit such as a character, string or file
- Entities allow
  - references to non-keyboard characters
  - abbreviations for frequently used strings
  - documents to be broken up into multiple parts
- An *entity declaration* in a DTD associates a name with an entity, e.g.,

```
<!ENTITY BBK "Birkbeck, University of London">
```

- An *entity reference*, e.g., `&BBK;` substitutes value of entity for its name in document
- An entity must be declared before it is referenced

General Entities

- `BBK` is an example of a *general entity*
- In XML, only 5 general entity declarations are built-in
  - `&` (&), `<` (<), `>` (>), `"` ("), `'` ('),
- All other entities must be declared in a DTD
- The values of *internal* entities are defined in the same document as references to them
- The values of *external* entities are defined elsewhere, e.g.,
  `<!ENTITY HTML-chapter SYSTEM "html.xml" >`
  - then `&HTML-chapter;` includes the contents of file `html.xml` at the point of reference
  - `standalone="no"` must be included in the XML declaration

Parameter Entities

- *Parameter entities* are
  - used only within XML markup declarations
  - declared by inserting `%` between `ENTITY` and name, e.g.,

```
<!ENTITY % list      "OL | UL" >
<!ENTITY % heading  "H1 | H2 | H3 | H4 | H5 | H6" >
```

  - referenced using `%` and `;` delimiters, e.g.,

```
<!ENTITY % block  "P | %list; | %heading; | ..." >
```

- As an example. see the HTML 4.01 DTD

Limitations of DTDs

- There is no data typing, especially for element content
- They are only marginally compatible with namespaces
- We cannot use mixed content *and* enforce the order and number of child elements
- It is clumsy to enforce the presence of child elements without also enforcing an order for them (i.e. no & operator from SGML)
- Element names in a DTD are *global* (see later)
- They use non-XML syntax
- The XML Schema Definition Language, e.g., addresses these limitations

Chapter 4

XML Schema Definition Language (XSDL)

XML Schema

- XML Schema is a W3C Recommendation
  - XML Schema Part 0: Primer
  - XML Schema Part 1: Structures
  - XML Schema Part 2: Datatypes
- describes permissible contents of XML documents
- uses XML syntax
- sometimes referred to as *XSDL: XML Schema Definition Language*
- addresses a number of limitations of DTDs

Simple example

- file `greeting.xml` contains:

```
<?xml version="1.0"?>
<greet>Hello World!</greet>
```

- file `greeting.xsd` contains:

```
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">
  <xsd:element name="greet" type="xsd:string"/>
</xsd:schema>
```

- `xsd` is prefix for the namespace for the "schema of schemas"
- declares element with name `greet` to be of built-in type `string`

DTDs vs. schemas

DTD	Schema
`<!ELEMENT>` declaration	`xsd:element` element
`<!ATTLIST>` declaration	`xsd:attribute` element
`<!ENTITY>` declaration	n/a
`#PCDATA` content	`xsd:string` type
n/a	other data types

Linking a schema to a document

- `xsi:noNamespaceSchemaLocation` attribute on root element
- this says no target namespace is declared in the schema
- `xsi` prefix is mapped to the URI:
  `http://www.w3.org/2001/XMLSchema-instance`
- this namespace defines global attributes that relate to schemas and can occur in instance documents
- for example:

```
<?xml version="1.0"?>
<greet xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:noNamespaceSchemaLocation="greeting.xsd">
  Hello World!
</greet>
```

Validating a document

- W3C provides an XML Schema Validator (XSV)
- URL is `http://www.w3.org/2001/03/webdata/xsv`
- submit XML file (and schema file)
- report generated for `greeting.xml` as follows

More complex document example

```
<cd xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
      xsi:noNamespaceSchemaLocation="cd.xsd">
  <composer>Johannes Brahms</composer>
  <performance>
    <composition>Piano Concerto No. 2</composition>
    <soloist>Emil Gilels</soloist>
    <orchestra>Berlin Philharmonic</orchestra>
    <conductor>Eugen Jochum</conductor>
    <recorded>1972</recorded>
  </performance>
  <performance>
    <composition>Fantasias Op. 116</composition>
    <soloist>Emil Gilels</soloist>
    <recorded>1976</recorded>
  </performance>
  <length>PT1H13M37S</length>
</cd>
```

Simple and complex data types

- XML schema allows definition of *data types* as well as declarations of elements and attributes
- simple data types
  - can contain only text (i.e., no markup)
  - e.g., values of attributes
  - e.g., elements without children or attributes
- complex data types can contain
  - child elements (i.e., markup) or
  - attributes

More complex schema example

```
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">

  <xsd:element name="cd" type="CDType"/>

  <xsd:complexType name="CDType">
    <xsd:sequence>
      <xsd:element name="composer"    type="xsd:string"/>
      <xsd:element name="performance" type="PerfType"
                                      maxOccurs="unbounded"/>
      <xsd:element name="length"      type="xsd:duration"
                                      minOccurs="0"/>
    </xsd:sequence>
  </xsd:complexType>
  ...
</xsd:schema>
```

Main schema components

- `xsd:element` *declares* an element and assigns it a type, e.g.,

  ```
  <xsd:element name="composer"  type="xsd:string"/>
  ```

  using a built-in, simple data type, or

  ```
  <xsd:element name="cd" type="CDType"/>
  ```

  using a user-defined, complex data type
- `xsd:complexType` *defines* a new type, e.g.,

  ```
  <xsd:complexType name="CDType">
  ...
  </xsd:complexType>
  ```

- defining named types allows reuse (and may help readability)
- `xsd:attribute` *declares* an attribute and assigns it a type (see later)

Structuring element declarations

- `xsd:sequence`
  - requires elements to occur in order given
  - analogous to `,` in DTDs
- `xsd:choice`
  - allows one of the given elements to occur
  - analogous to `|` in DTDs
- `xsd:all`
  - allows elements to occur in any order
  - analogous to `&` in *SGML* DTDs

Defining number of element occurrences

- `minOccurs` and `maxOccurs` attributes control the number of occurrences of an element, sequence or choice
- `minOccurs` must be a non-negative integer
- `maxOccurs` must be a non-negative integer or `unbounded`
- default value for each of `minOccurs` and `maxOccurs` is 1

Another complex type example

```
<xsd:complexType name="PerfType">
  <xsd:sequence>
    <xsd:element name="composition" type="xsd:string"/>
    <xsd:element name="soloist"     type="xsd:string"
                                    minOccurs="0"/>
    <xsd:sequence minOccurs="0">
      <xsd:element name="orchestra" type="xsd:string"/>
      <xsd:element name="conductor" type="xsd:string"/>
    </xsd:sequence>
    <xsd:element name="recorded"    type="xsd:gYear"/>
  </xsd:sequence>
</xsd:complexType>
```

An equivalent DTD

```
<!ELEMENT CD          (composer, (performance)+, (length)?)>
<!ELEMENT performance (composition, (soloist)?,
                       (orchestra, conductor)?, recorded)>
<!ELEMENT composer    (#PCDATA)>
<!ELEMENT length      (#PCDATA)> <!-- duration -->
<!ELEMENT composition (#PCDATA)>
<!ELEMENT soloist     (#PCDATA)>
<!ELEMENT orchestra   (#PCDATA)>
<!ELEMENT conductor   (#PCDATA)>
<!ELEMENT recorded    (#PCDATA)> <!-- gYear -->
```

Declaring attributes

- use `xsd:attribute` element inside an `xsd:complexType`
- has attributes `name`, `type`, e.g.,

```
<xsd:attribute name="version" type="xsd:number"/>
```

- attribute `use` is optional
  - if omitted means attribute is optional (like `#IMPLIED`)
  - for required attributes, say `use="required"` (like `#REQUIRED`)
- for fixed attributes, say `fixed="..."` (like `#FIXED`), e.g.,

```
<xs:attribute name="version" type="xs:number" fixed="2.0"/>
```

- for attributes with default value, say `default="..."`
- for enumeration, use `xsd:simpleType`
- attributes must be declared at the end of an `xsd:complexType`

Locally-scoped element names

- in DTDs, all element names are *global*
- XML schema allows element types to be local to a context, e.g.,

```
<xsd:element name="book">
      <xsd:element name="title"> ... </xsd:element>
    ...
</xsd:element>

<xsd:element name="employee">
      <xsd:element name="title"> ... </xsd:element>
    ...
</xsd:element>
```

- content models for two occurrences of `title` can be different

Simple data types

- Form a type hierarchy; the root is called *anyType*
  - all complex types
  - *anySimpleType*
    - `string`
    - `boolean`, e.g., `true`
    - `anyUri`, e.g., `http://www.dcs.bbk.ac.uk/~ptw/home.html`
    - `duration`, e.g., `P1Y2M3DT10H5M49.3S`
    - `gYear`, e.g., `1972`
    - `float`, e.g., `123E99`
    - `decimal`, e.g., `123456.789`
    - ...
- lowest level above are the *primitive data types*
- for a full list, see Simple Types in the Primer

Primitive date and time types

- `date`, e.g., `1994-04-27`
- `time`, e.g., `16:50:00+01:00` or `15:50:00Z` if in Co-ordinated Universal Time (UTC)
- `dateTime`, e.g., `1918-11-11T11:00:00.000+01:00`
- `duration`, e.g., `P2Y1M3DT20H30M31.4159S`
- "Gregorian" calendar dates (introduced in 1582 by Pope Gregory XIII):
  - `gYear`, e.g., `2001`
  - `gYearMonth`, e.g., `2001-01`
  - `gMonthDay`, e.g., `--12-25` (note hyphen for missing year)
  - `gMonth`, e.g., `--12--` (note hyphens for missing year and day)
  - `gDay`, e.g., `---25` (note only 3 hyphens)

Built-in derived string types

Derived from `string`:

- `normalizedString` (newline, tab, carriage-return are converted to spaces)
  - `token` (adjacent spaces collapsed to a single space; leading and trailing spaces removed)
    - `language`, e.g., `en`
    - `name`, e.g., `my:name`

Derived from `name`:

- `NCNAME` ("non-colonized" name), e.g., `myName`
  - `ID`
  - `IDREF`
  - `ENTITY`

Built-in derived numeric types

Derived from `decimal`:

- `integer` (decimal with no fractional part), e.g., -123456
  - `nonPositiveInteger`, e.g., 0, -1
    - `negativeInteger`, e.g., -1
  - `nonNegativeInteger`, e.g., 0, 1
    - `positiveInteger`, e.g., 1
    - ...
  - ...

User-derived simple data types

- complex data types can be created "from scratch"
- new simple data types must be *derived* from existing simple data types
- derivation can be by one of
  - *extension*
    - *list*: a list of values of an existing data type
    - *union*: allows values from two or more data types
  - *restriction*: limits the values allowed using, e.g.,
    - maximum value (e.g., 100)
    - minimum value (e.g., 50)
    - length (e.g., of string or list)
    - number of digits
    - enumeration (list of values)
    - pattern

    above constraints are known as *facets*

Restriction by enumeration

```
<xsd:element name="MScResult">
  <xsd:simpleType>
    <xsd:restriction base="xsd:string">
      <xsd:enumeration value="distinction"/>
      <xsd:enumeration value="merit"/>
      <xsd:enumeration value="pass"/>
      <xsd:enumeration value="fail"/>
    </xsd:restriction>
  </xsd:simpleType>
</xsd:element>
```

- contents of `MScResult` element is a restriction of the `xsd:string` type
- must be one of the 4 values given
- e.g., `<MScResult>pass</MScResult>`

Restriction by values

- `examMark` can be from 0 to 100

```
<xsd:element name="examMark">
  <xsd:simpleType>
    <xsd:restriction base="xsd:nonNegativeInteger">
      <xsd:maxInclusive value="100"/>
    </xsd:restriction>
  </xsd:simpleType>
</xsd:element>
```

- or, equivalently

```
    <xsd:restriction base="xsd:integer">
      <xsd:minInclusive value="0"/>
      <xsd:maxInclusive value="100"/>
    </xsd:restriction>
```

Restriction by pattern

```
<xsd:element name="zipcode">
  <xsd:simpleType>
    <xsd:restriction base="xsd:string">
      <xsd:pattern value="\d{5}(-\d{4})?"/>
    </xsd:restriction>
  </xsd:simpleType>
</xsd:element>
```

- `value` attribute contains a *regular expression*
- `\d` means any digit
- `()` used for grouping
- `x{5}` means exactly 5 `x`'s (in a row)
- `x?` indicates zero or one `x`
- zipcode examples: 90720-1314 and 22043

Document with mixed content

- We may want to mix elements and text, e.g.:

```
<letter>
  Dear Mr <name>Smith</name>,
  Your order of <quantity>1</quantity>
  <product>Baby Monitor</product> was shipped
  on <date>1999-05-21</date>. ....
</letter>
```

- A DTD would have to contain:

```
<!ELEMENT letter (#PCDATA|name|quantity|product|date)*>
```

which cannot enforce the order of subelements

Schema fragment declaring mixed content

```
<xsd:element name="letter">
 <xsd:complexType mixed="true">
  <xsd:sequence>
   <xsd:element name="name" type="xsd:string"/>
   <xsd:element name="quantity" type="xsd:positiveInteger"/>
   <xsd:element name="product"  type="xsd:string"/>
   <xsd:element name="date" type="xsd:date" minOccurs="0"/>
   <!-- etc. -->
  </xsd:sequence>
 </xsd:complexType>
</xsd:element>
```

Chapter 5

Relax NG

Problems with DTDs

- DTDs are sometimes not powerful enough
- e.g., (to simplify) in HTML
  1. a `form` element can occur in a `table` element and
  2. a `table` element can occur in a `form` element, but
  3. a `form` element *cannot* occur inside another `form` element
- we have

```
<!ELEMENT table  (... form ...) >
<!ELEMENT form   (... table ...) >
```

- but condition (3) above cannot be enforced by an XML DTD

Problems with XML schema

- XML schema *can* handle the previous example using locally-scoped element names
- but what about the following?
  - a document (`doc` element) contains one or more paragraphs (`par` elements)
  - the first paragraph has a different content model to subsequent paragraphs
  - (perhaps the first letter of the first paragraph is enlarged)
- we want something like

  ```
  <!ELEMENT doc  (par, par*) >
  ```

  but where two occurrences of `par` have *different* content models
- this *cannot* be specified in XML schema

RelaxNG

- RelaxNG resulted from the merger of two earlier projects
  - RELAX (REgular LAnguage description for XML)
  - TREX (Tree Regular Expressions for XML)
- It has the same power as *Regular Tree Grammars*
- It has two syntactic forms: one XML-based, one not (called the *compact* syntax)
- It is simpler than XML schema
- It uses XML Schema Part 2 for a vocabulary of data types

Compact Syntax: RSS Example

```
element rss {
  element channel {
    element title         { text },
    element link          { xsd:anyURI },
    element description   { text },
    element lastBuildDate { xsd:dateTime }?,
    element ttl           { xsd:positiveInteger }?,
    element item {
      element title       { text },
      element description { text },
      element link        { xsd:anyURI }?,
      element pubDate     { xsd:dateTime }?
    }+
  }
}
```

Named patterns

- It is often convenient to be able to give *names* to parts of a pattern
- This is similar to using *non-terminal* symbols in a (context-free) grammar
- It is also related to the use of complex types in XSDL
- RelaxNG uses "=" in the compact syntax (and `define` elements in the XML syntax) to give names to patterns
- The name `start` is used for the root pattern

Compact Syntax with Named Patterns: RSS Example

```
start   = RSS
RSS     = element rss     { Channel }
Channel = element channel { Title,Link,Desc,LBD?,TTL?,Item+ }
Title   = element title   { text }
Link    = element link    { xsd:anyURI }
Desc    = element description   { text }
LBD     = element lastBuildDate { xsd:dateTime }
TTL     = element ttl           { xsd:positiveInteger }
Item    = element item          { Title, Desc, Link, PD? }
PD      = element pubDate       { xsd:dateTime }
```

Table and forms example (compact syntax)

```
TableWithForm    = element table { ... Form ... }

Form             = element form  { ... TableWithoutForm ... }

TableWithoutForm = element table { ... }
```

- No `Form` pattern appears in the third definition above

Paragraphs example (compact syntax)

```
D  = element doc { P1, P2* }

P1 = element par { ... }

P2 = element par { ... }
```

- The content models for the `P1` and `P2` patterns can be different

Summary

- We have considered 3 different languages for defining XML document types
- DTDs are simple, but their main limitation is that data types (other than strings) are not provided
- XSDL is comprehensive, but rather complicated
- RelaxNG is the most expressive of the three, while still remaining quite simple; it is also an ISO standard, but has not been widely adopted

Chapter 6

XPath

Introduction

- XPath is a language that lets you identify particular parts of XML documents
- XPath interprets XML documents as nodes (with content) organised in a tree structure
- XPath indicates nodes by (relative) position, type, content, and several other criteria
- Basic syntax is somewhat similar to that used for navigating file hierarchies
- XPath 1.0 (1999) and 2.0 (2010) are W3C recommendations

Some Tools for XPath

- Saxon (specifically Saxon-HE which implements XPath 2.0, XQuery 1.0 and XSLT 2.0)
- eXist-db (a native XML database supporting XPath 2.0, XQuery 1.0 and XSLT 1.0)
- XPath Checker (add-on for Firefox)
- XPath Expression Testbed (available online)

Data Model

XPath's data model has some non-obvious features:

- The tree's root node is not the same as the document's root (document) element
- The tree's root node contains the entire document including the root element (and comments and processing instructions that appear before it)
- XPath's data model does not include everything in the document: XML declaration and DTD are not addressable
- `xmlns` attributes are reported as namespace nodes

Data Model (2)

- There are 6 types of *node*:
  - *root*
  - *element*
  - *attribute*
  - *text*
  - *comment*
  - *processing instruction*
- Element nodes have an associated set of attribute nodes
- Attribute nodes are *not* children of element nodes
- The order of child element nodes is *significant*
- We will only consider the first 4 types of node

Example (1)

Recall our CD library example

```
<CD-library>
  <CD number="724356690424">
    <performance>
      <composer>Frederic Chopin</composer>
      <composition>Waltzes</composition>
      <soloist>Dinu Lipatti</soloist>
      <date>1950</date>
    </performance>
  </CD>
  ...
```

Example (2)

```
...
<CD number="419160-2">
  <composer>Johannes Brahms</composer>
  <soloist>Emil Gilels</soloist>
  <performance>
    <composition>Piano Concerto No. 2</composition>
    <orchestra>Berlin Philharmonic</orchestra>
    <conductor>Eugen Jochum</conductor>
    <date>1972</date>
  </performance>
  <performance>
    <composition>Fantasias Op. 116</composition>
    <date>1976</date>
  </performance>
</CD>
...
```

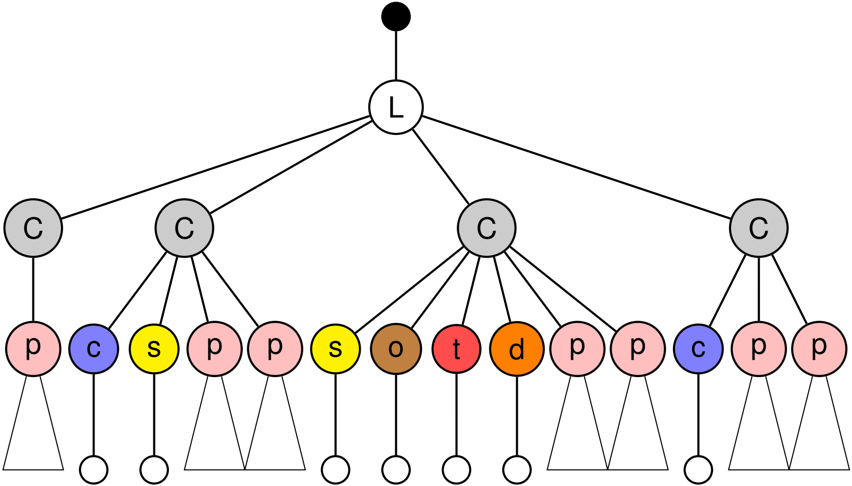
Example (3)

```
...
<CD number="449719-2">
  <soloist>Martha Argerich</soloist>
  <orchestra>London Symphony Orchestra</orchestra>
  <conductor>Claudio Abbado</conductor>
  <date>1968</date>
  <performance>
    <composer>Frederic Chopin</composer>
    <composition>Piano Concerto No. 1</composition>
  </performance>
  <performance>
    <composer>Franz Liszt</composer>
    <composition>Piano Concerto No. 1</composition>
  </performance>
</CD>
...
```

Example (4)

```
  ...
  <CD number="430702-2">
    <composer>Antonin Dvorak</composer>
    <performance>
      <composition>Symphony No. 9</composition>
      <orchestra>Vienna Philharmonic</orchestra>
      <conductor>Kirill Kondrashin</conductor>
      <date>1980</date>
    </performance>
    <performance>
      <composition>American Suite</composition>
      <orchestra>Royal Philharmonic</orchestra>
      <conductor>Antal Dorati</conductor>
      <date>1984</date>
    </performance>
  </CD>
</CD-library>
```

Example — Tree Structure



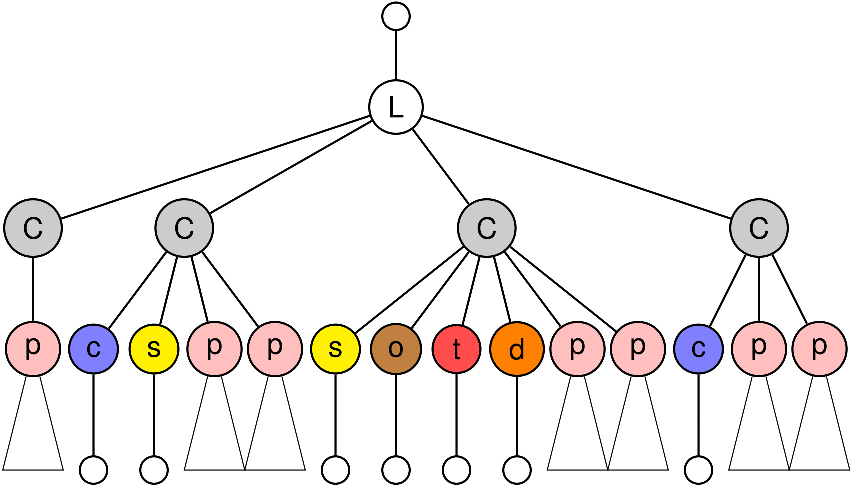
Location Path

- The most useful XPath expression is a *location path*: e.g., `/CD-library/CD/performance`
- A location path consists of at least one *location step*: e.g., `CD-library`, `CD` and `performance` are location steps
- A location step takes as input a set of nodes, also called the *context* (to be defined more precisely later)
- The location step expression is applied to this node set and results in an output node set
- This output node set is used as input for the next location step

Location Path (2)

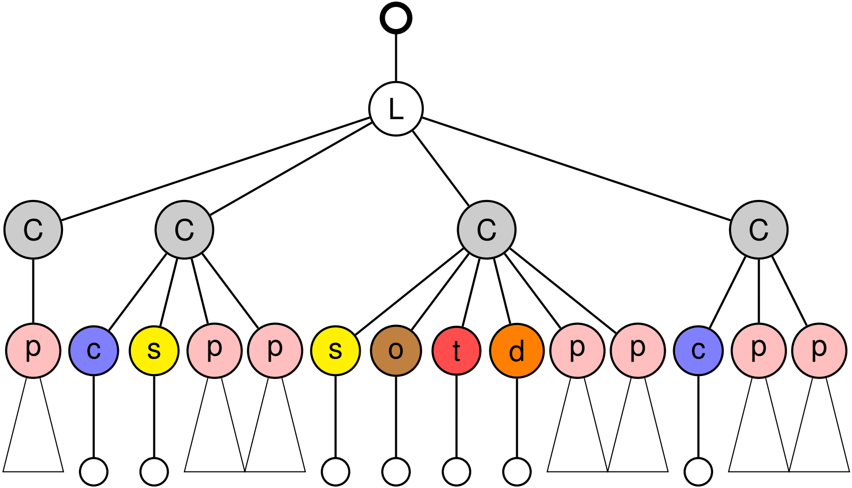
- There are two different kinds of location paths:
  - *Absolute* location paths
  - *Relative* location paths
- An absolute location path
  - starts with /
  - is followed by a relative location path
  - is evaluated at the root (context) node of a document
  - e.g., `/CD-library/CD/performance`
- A relative location path
  - is a sequence of location steps
  - each separated by /
  - evaluated with respect to some other context nodes
  - e.g., `CD/performance`

Evaluation of absolute location path



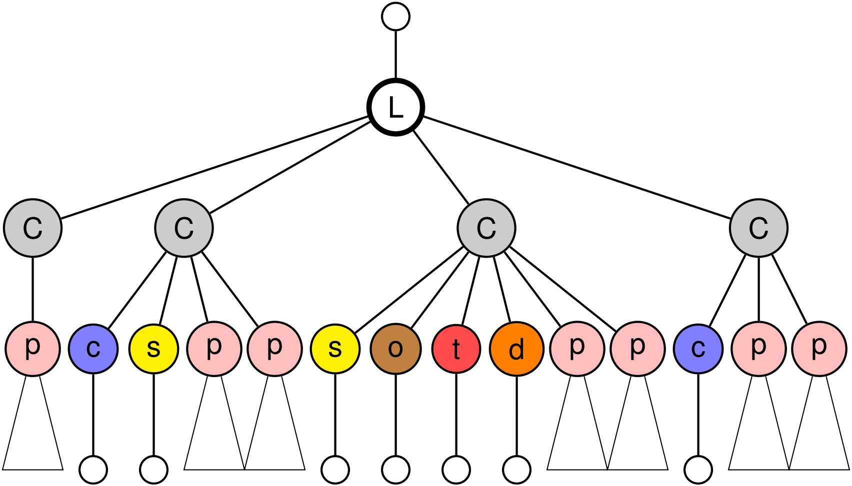
Evaluation of absolute location path

/



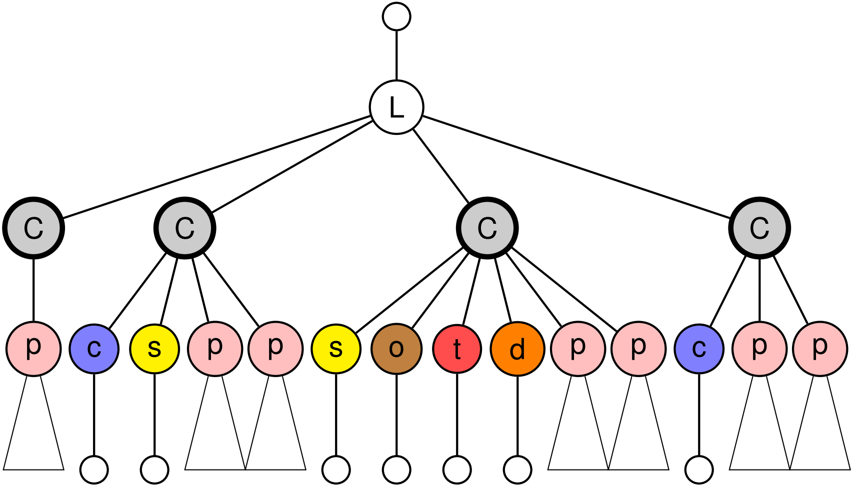
Evaluation of absolute location path

`/CD-library`



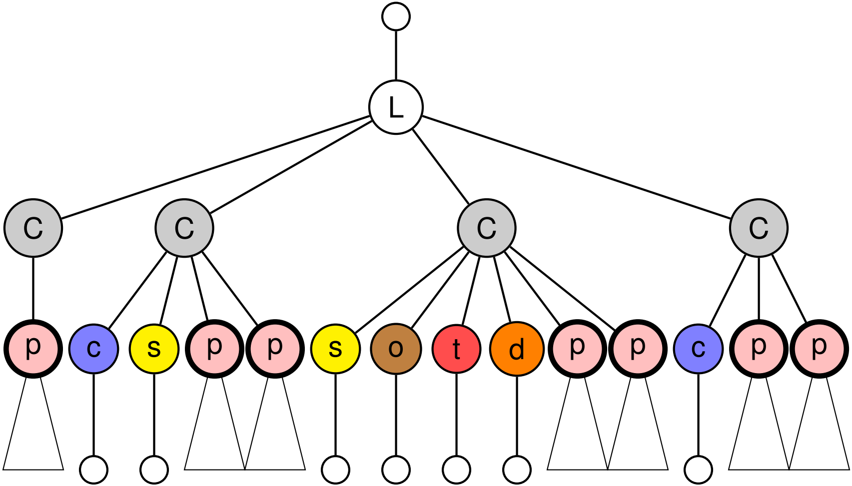
Evaluation of absolute location path

`/CD-library/CD`



Evaluation of absolute location path

`/CD-library/CD/performance`

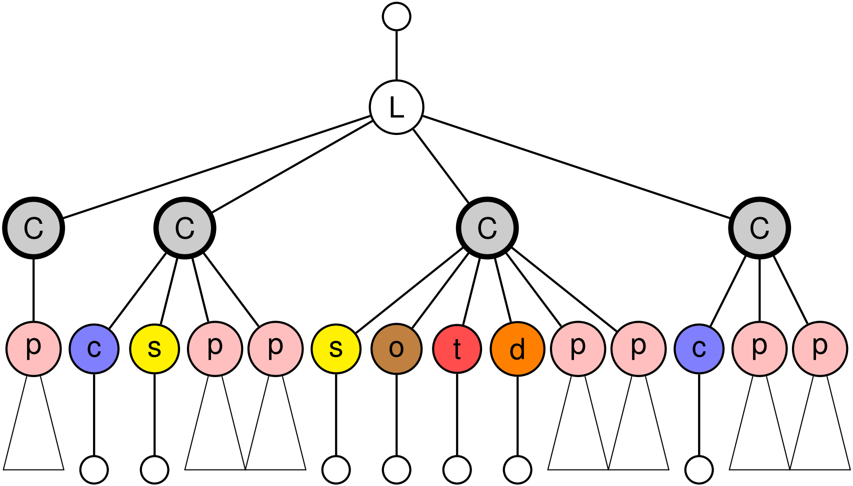


Location Step

- In general, a location step in turn consists of a
  - (navigation) axis
  - node test
  - predicate(s)
- Syntax is *axis* `::` *node test* `[` *predicate* `]` `...` `[` *predicate* `]`
- e.g., `child::CD[composer='Johannes Brahms']`
  - `child` is the axis
  - `CD` is the node test
  - `composer='Johannes Brahms'` is the predicate
- A location step is applied to each node in the context (i.e., each node becomes the context node)
- All resulting nodes are added to the output set of this location step

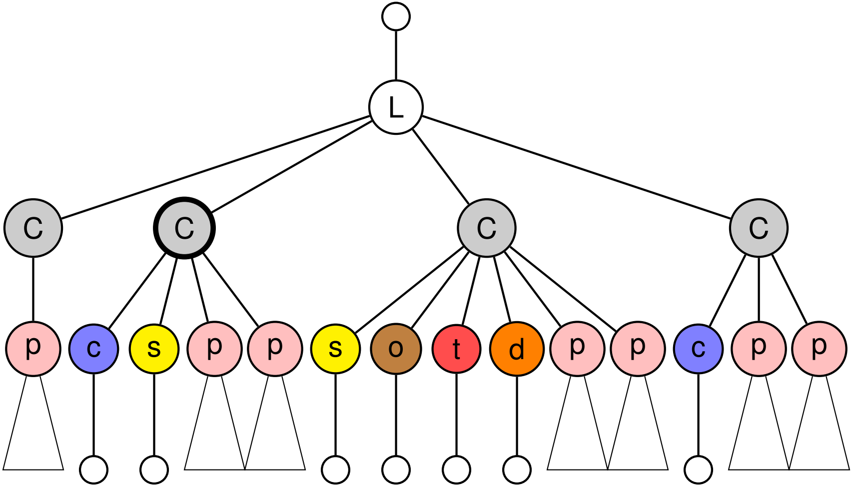
Evaluation of predicate

`/child::CD-library/child::CD`



Evaluation of predicate

`/child::CD-library/child::CD[composer='Johannes Brahms']`

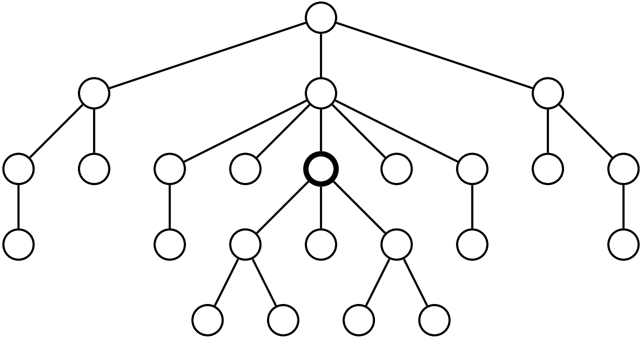


Axes

- An axis specifies what nodes, relative to the current context node, to consider
- There are 13 different axes (some can be abbreviated)
  - `self`, abbreviated by `.`
  - `child`, abbreviated by *empty axis*
  - `parent`, abbreviated by `..`
  - `descendant-or-self`, abbreviated by *empty location step*
  - `descendant`, `ancestor`, `ancestor-or-self`
  - `following`, `following-sibling`, `preceding`, `preceding-sibling`
  - `attribute`, abbreviated by `@`
  - `namespace`

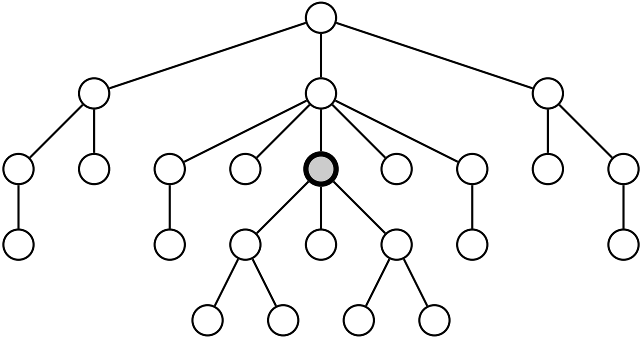
Axes

- The following slides show (graphical) examples of the axes, assuming the node in bold typeface is the context node



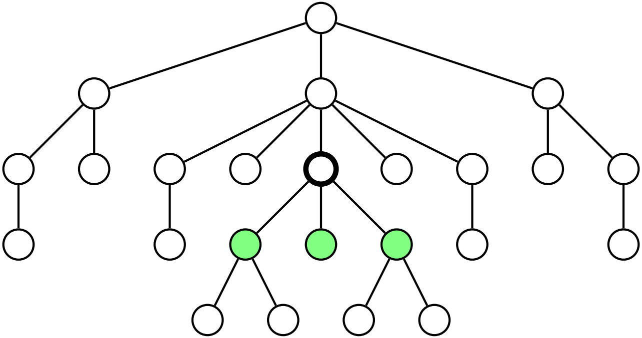
Self-Axis

- The self-axis just contains the context node itself



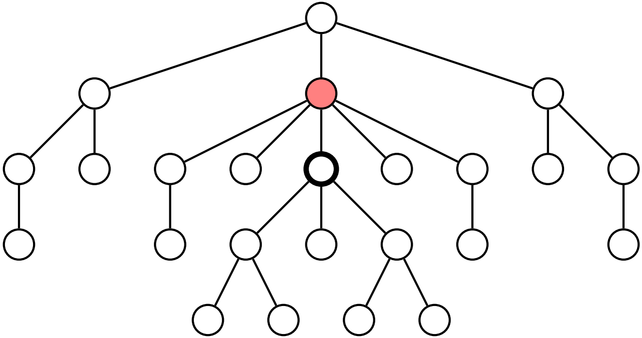
Child-Axis

- The child-axis contains the children (direct descendants) of the context node



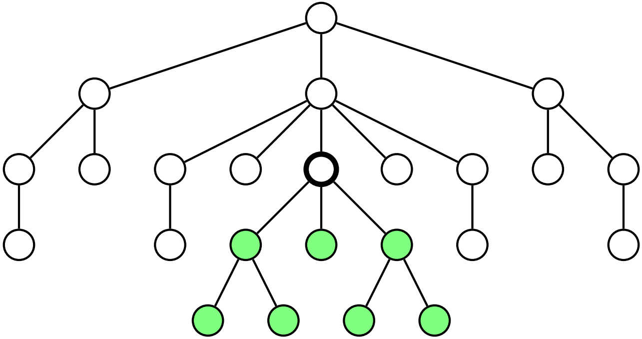
Parent-Axis

- The parent-axis contains the parent (direct ancestor) of the context node



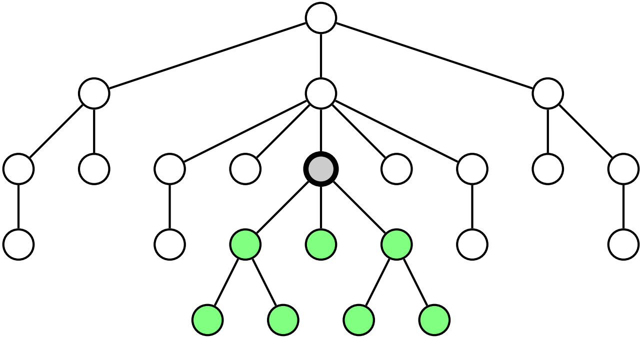
Descendant-Axis

- The descendant-axis contains all direct and indirect descendants of the context node



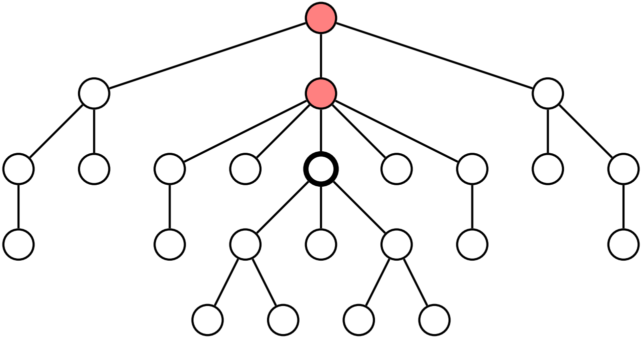
Descendant-Or-Self-Axis

- The descendant-or-self-axis contains all direct and indirect descendants of the context node + the context node itself



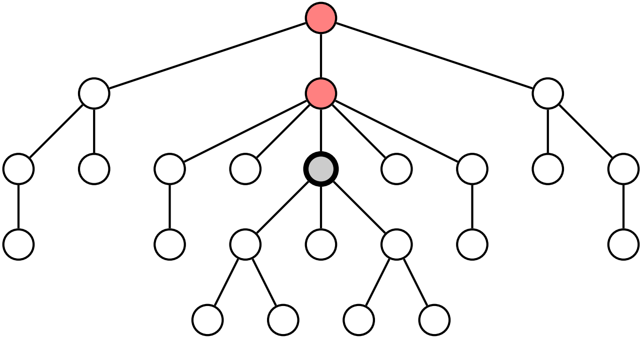
Ancestor-Axis

- The ancestor-axis contains all direct and indirect ancestors of the context node



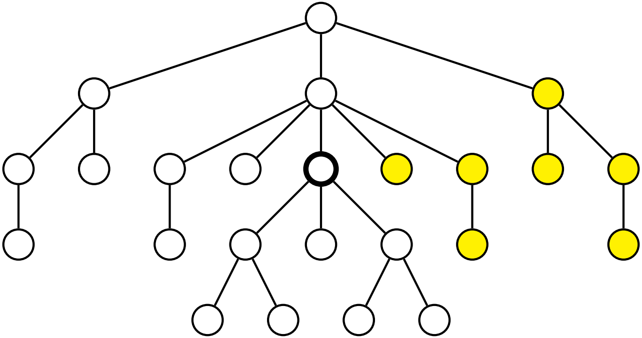
Ancestor-Or-Self-Axis

- The ancestor-or-self-axis contains all direct and indirect ancestors of the context node + the context node itself



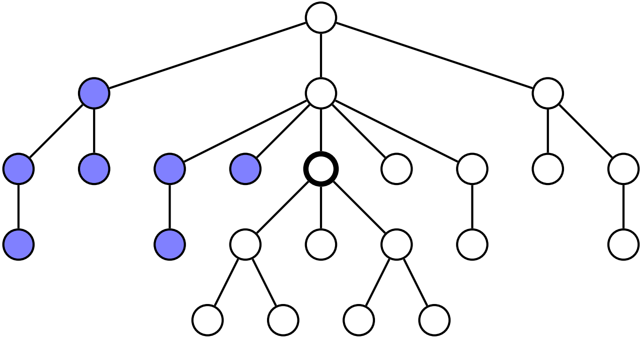
Following-Axis

- The following-axis contains all nodes that begin after the context node ends



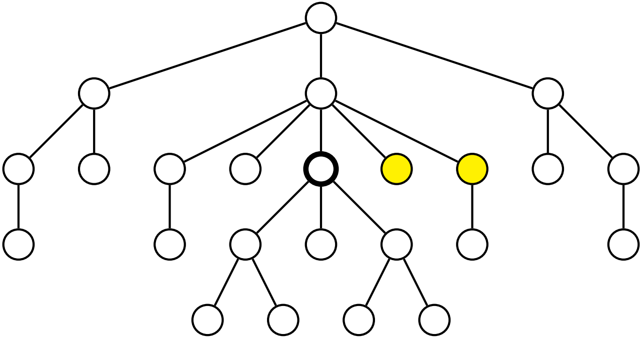
Preceding-Axis

- The preceding-axis contains all nodes that end before the context node begins



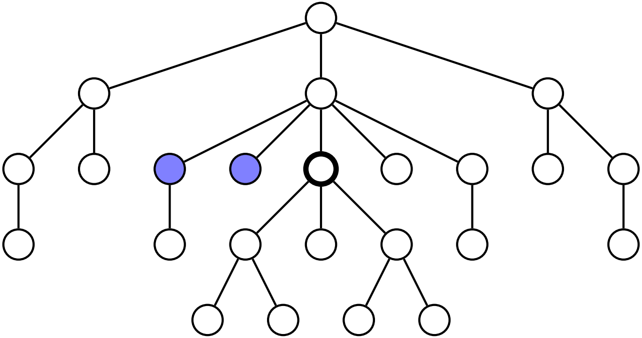
Following-Sibling-Axes

- The following-sibling-axis contains all following nodes that have the same parent as the context node



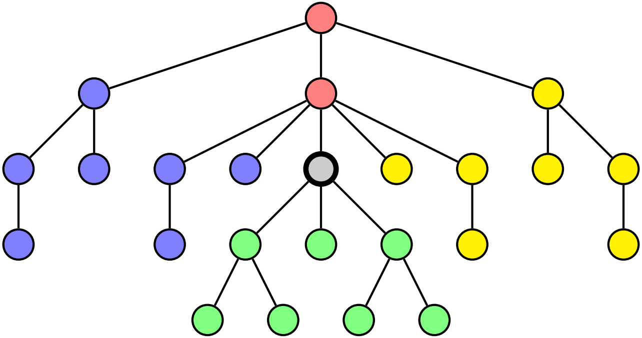
Preceding-Sibling-Axis

- The preceding-sibling-axis contains all preceding nodes that have the same parent as the context node



Partitioning

- The axes self, ancestor, descendant, following and preceding partition a document into five disjoint subtrees:



Attribute-Axis

- Attributes are handled in a special way in XPath
- The attribute-axis contains all the attribute nodes of the context node
- This axis is empty if the context node is not an element node
- Does not contain `xmlns` attributes used to declare namespaces

Namespace-Axis

- The namespace-axis contains all namespaces in scope of the context node
- This axis is empty if the context node is not an element node

Node Tests

- Once the correct relative position of a node has been identified the type of a node can be tested
- A *node test* further refines the nodes selected by the location step
- A double colon `::` separates the axis from the node test
- There are seven different kinds of node tests

name
prefix`:*`
`node()`
`text()`
`comment()`
`processing-instruction()`
`*`

Name

- The *name* node test selects all elements with a matching name
  - e.g., if our context is a set of 4 `CD` elements and the location step uses the `child` axis, then we get element nodes with different names
  - we can use the *name* node test to return, e.g., only `soloist` elements
- Along the attribute-axis it matches all attributes with the same name

Prefix:*

- Along an element axis, all nodes whose namespace URIs are the same as the prefix are matched
- This node test is also available for attribute nodes

Comment, Text, Processing-Instruction

- `comment()` matches all comment nodes
- `text()` matches all text nodes
- `processing-instruction()` matches all processing instructions

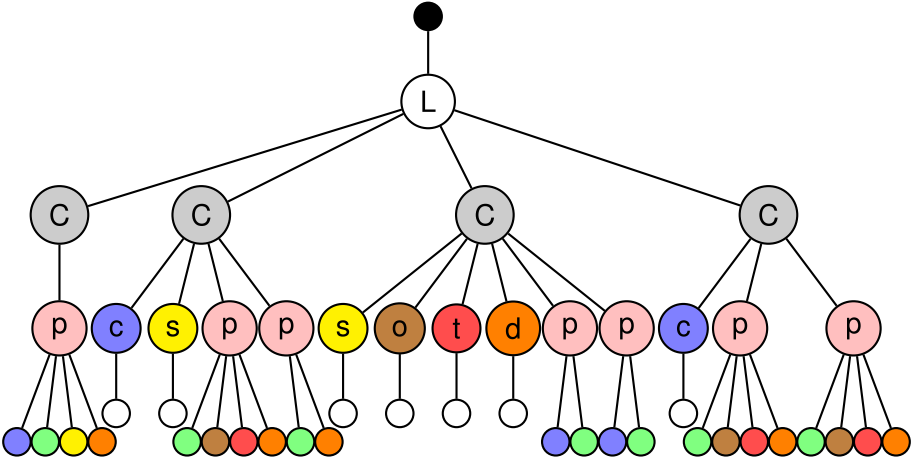
Node and *

- `node()` selects all nodes, regardless of type: attribute, namespace, element, text, comment, processing instruction, and root
- `*` selects all element nodes, regardless of name
  - If the axis is the attribute axis, then it selects all attribute nodes
  - If the axis is the namespace axis, then is selects all namespace nodes

Key for full CD library example

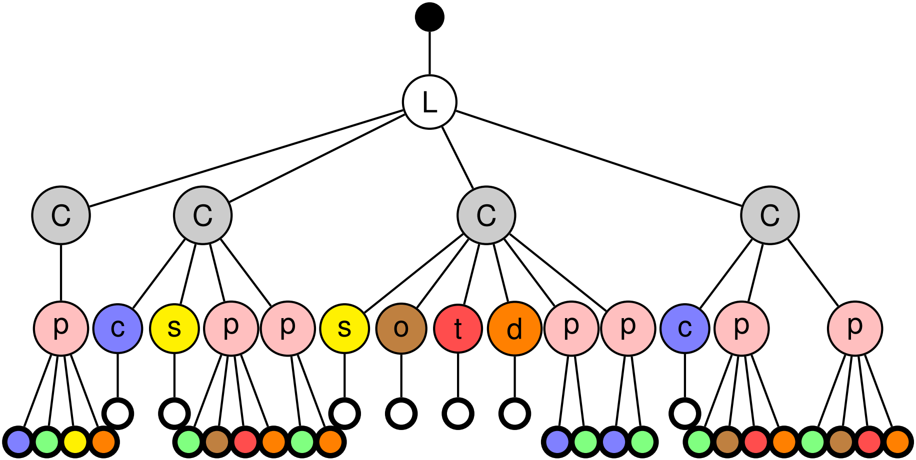
Element name	Abbreviation	Colour
root		black
library	L	white
cd	C	grey
performance	p	pink
composer	c	blue
composition		green
soloist	s	yellow
conductor	t	red
orchestra	o	brown
date	d	orange

Full CD library example



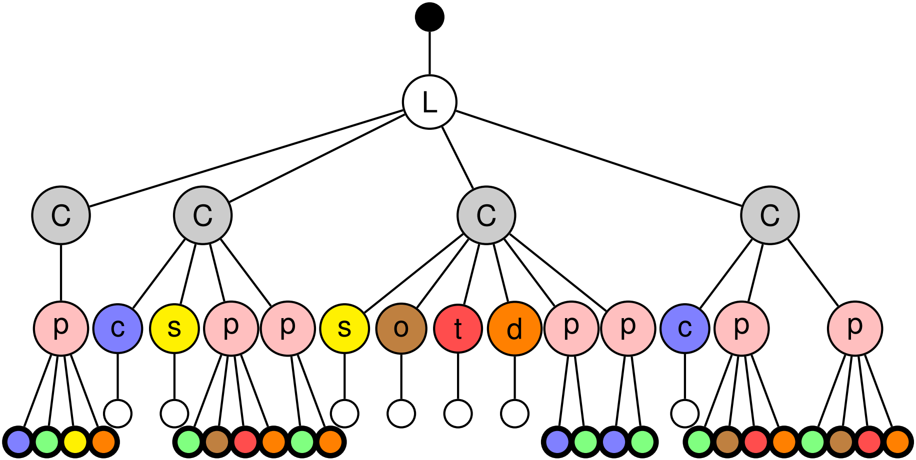
Example using * and node()

```
/CD-library/CD/*/node()
```



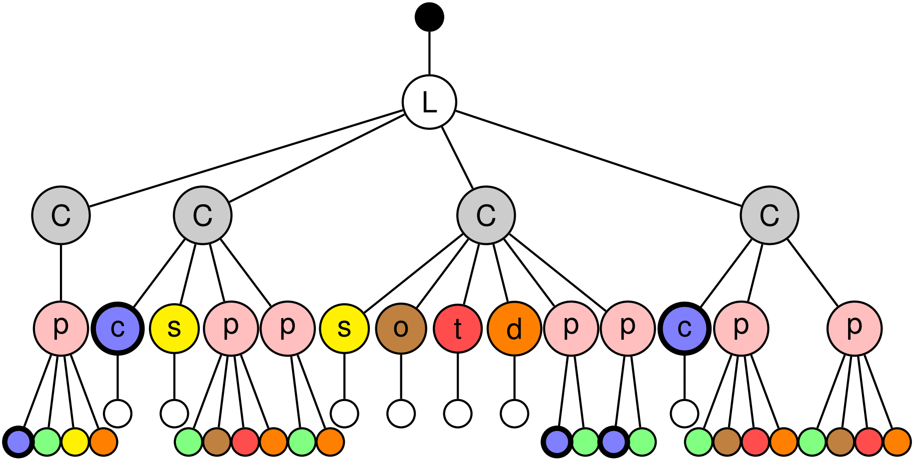
Example showing difference between * and node()

`/CD-library/CD/*/*`



Example using descendant

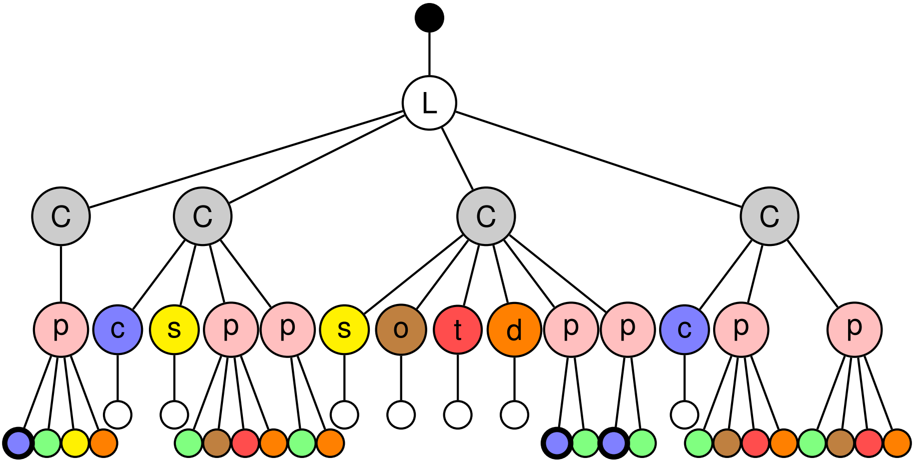
`//composer` or `/descendant-or-self::node()/composer`



Another example using descendant

`//performance/composer` or

`/descendant-or-self::node()/child::composer`



Predicates

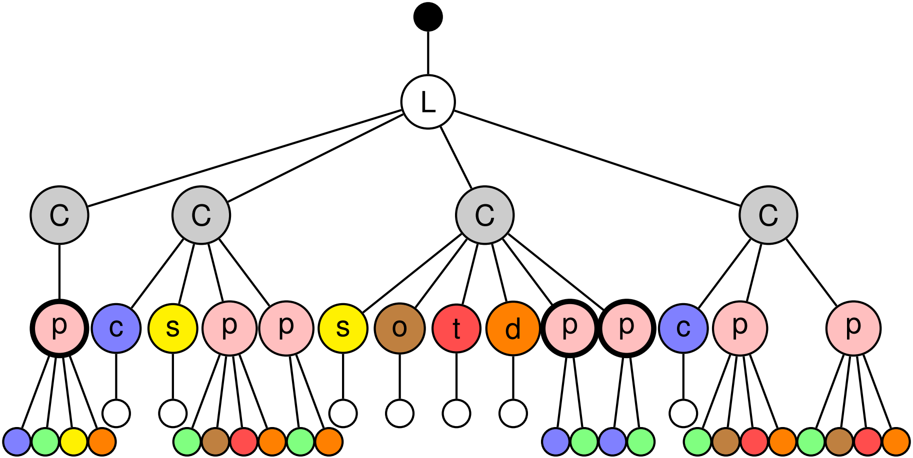
- A node set can be reduced further with *predicates*
- While each location step must have an axis and a node test (which may be empty), a predicate is optional
- A predicate contains a Boolean expression which is tested for each node in the resulting node set
- A predicate is enclosed in square brackets `[ ]`

Predicates (2)

- XPath supports a full complement of relational operators, including =, >, <, >=, <=, `!=`
- XPath also provides Boolean `and` and `or` operators to combine expressions logically
- In some cases a predicate may not be a Boolean; then XPath will convert it to one implicitly (if that is possible):
  - an empty node set is interpreted as false
  - a non-empty node set is interpreted as true

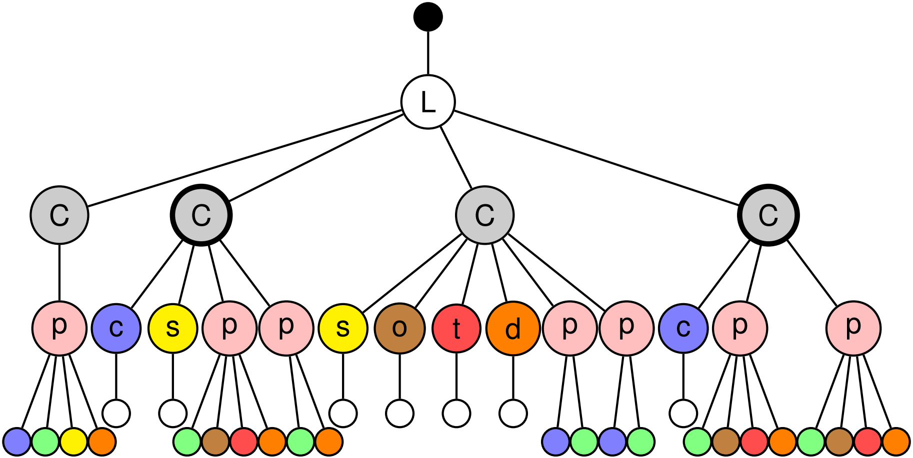
Example using a predicate

```
//performance[composer]
```



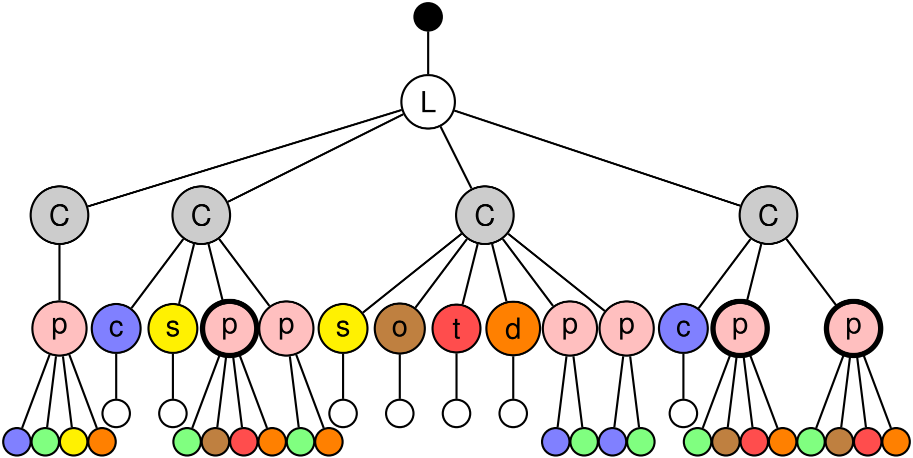
Another example using a predicate

```
//CD[performance/orchestra]
```



Example using multiple predicates

```
//performance[conductor][date]
```



Further examples with predicates

- `//performance[composer='Frederic Chopin']/composition` returns
  1. `<composition>Waltzes</composition>`
  2. `<composition>Piano Concerto No. 1</composition>`

Further examples with predicates

- `//performance[composer='Frederic Chopin']/composition` returns
  1. `<composition>Waltzes</composition>`
  2. `<composition>Piano Concerto No. 1</composition>`
- `//CD[@number="449719-2"]//composition` returns
  1. `<composition>Piano Concerto No. 1</composition>`
  2. `<composition>Piano Concerto No. 1</composition>`

  The two composition nodes have the same value, but they are different nodes

Functions

- XPath provides many functions that may be useful in predicates
- Each XPath function takes as input or returns one of these four types:
  - node set
  - string
  - Boolean
  - number

More about Context

- Each location step and predicate is evaluated with respect to a given *context*
- A specific context is defined as $(\langle N_1, N_2, \ldots N_m \rangle, N_c)$ with
  - a *context list* $\langle N_1, N_2, \ldots N_m \rangle$ of nodes in the tree
  - a *context node* N_c belonging to the list
- The *context length* m is the size of the context list
- The *context node position* $c \in [1, m]$ gives the position of the context node in the list

More about XPath Evaluation

- Each step s_i is interpreted with respect to a context; its result is a node list
- A step s_i is evaluated with respect to the context of step s_{i-1}
- More precisely:
  - for $i = 1$ (first step)
    if the path is absolute, the context is the root of the XML tree;
    else (relative paths) the context is defined by the environment;
  - For $i > 1$
    if $\mathcal{N} = \langle N_1, N_2, \ldots N_m \rangle$ is the result of step s_{i-1},
    step s_i is successively evaluated with respect to the context $(\mathcal{N}, N_j)$, for each $j \in [1, m]$
- The result of the path expression is the node list obtained after evaluating the last step

Node-set Functions

- *Node-set functions* operate on or return information about node sets
- Examples:
  - `position()`: returns a number equal to the position of the current node in the context list
    - `[position()=i]` can be abbreviated as `[i]`
  - `last()`: returns the size (i.e. the number of nodes in) the context list
  - `count(`*set*`)`: returns the size of the argument node *set*
  - `id()`: returns a node set containing all elements in the document with any of the specified IDs

Example about context

- The expression `//CD/performance[2]` returns the second performance *of each* CD, not the second of all performances
- The result of the step `CD` is the list of the 4 CD nodes
- The step `performance[2]` is evaluated once for each of 4 CD nodes in the context

Example about context (2)

- The result is the list comprising the second performance element child of each CD:

1.
```
<performance>
  <composition>Fantasias Op. 116</composition>
  <date>1976</date>
</performance>
```
2.
```
<performance>
  <composer>Franz Liszt</composer>
  <composition>Piano Concerto No. 1</composition>
</performance>
```
3.
```
<performance>
  <composition>American Suite</composition>
  <orchestra>Royal Philharmonic</orchestra>
  <conductor>Antal Dorati</conductor>
  <date>1984</date>
</performance>
```

Problems with XPath processors

- Say we want those `performance` children of `CD` elements that are both the second `performance` and have a `date`
- The the following 4 expressions should all be equivalent
  - `//CD/performance[2][date]`
  - `//CD/performance[date][2]`
  - `//CD/performance[date and position()=2]`
  - `//CD/performance[position()=2 and date]`
- But different processors give different results!

Problems with XPath processors

- Say we want those `performance` children of `CD` elements that are both the second `performance` and have a `date`
- The the following 4 expressions should all be equivalent
  - `//CD/performance[2][date]`
  - `//CD/performance[date][2]`
  - `//CD/performance[date and position()=2]`
  - `//CD/performance[position()=2 and date]`
- But different processors give different results!
- Saxon and Safari, e.g., correctly give the answer as (1) and (3) from the previous slide for all 4 expressions

Problems with XPath processors

- Say we want those `performance` children of `CD` elements that are both the second `performance` and have a `date`
- The the following 4 expressions should all be equivalent
  - `//CD/performance[2][date]`
  - `//CD/performance[date][2]`
  - `//CD/performance[date and position()=2]`
  - `//CD/performance[position()=2 and date]`
- But different processors give different results!
- Saxon and Safari, e.g., correctly give the answer as (1) and (3) from the previous slide for all 4 expressions
- But, for `//CD/performance[date][2]`, eXist seems to return the second of all `performance` elements with a `date`

Problems with XPath processors

- Say we want those `performance` children of `CD` elements that are both the second `performance` and have a `date`
- The the following 4 expressions should all be equivalent
  - `//CD/performance[2][date]`
  - `//CD/performance[date][2]`
  - `//CD/performance[date and position()=2]`
  - `//CD/performance[position()=2 and date]`
- But different processors give different results!
- Saxon and Safari, e.g., correctly give the answer as (1) and (3) from the previous slide for all 4 expressions
- But, for `//CD/performance[date][2]`, eXist seems to return the second of all `performance` elements with a `date`
- An earlier tool returned, for each `CD`, the second of its `performance` elements that had a `date` (if any)

More about the position() function

- `position()` is a function that returns the position of the current node in the context node set
- For most axes it counts forward from the context node
- For the “backward” axes it counts backwards from the context node
- The “backward” axes are: ancestor, ancestor-or-self, preceding, and preceding-sibling

Examples using position()

- So, to get the CD immediately before the one that was composed by Dvorak:
  `//CD[composer='Antonin Dvorak']/preceding::CD[1]`
- This selects the third CD
- To get the last CD (without having to know how many there are), use `//CD[position()=last()]`

Example using a different axis

- `//date/following-sibling::*` returns the following:
  1. ```
     <performance>
       <composer>Frederic Chopin</composer>
       <composition>Piano Concerto No. 1</composition>
     </performance>
     ```
  2. ```
     <performance>
       <composer>Franz Liszt</composer>
       <composition>Piano Concerto No. 1</composition>
     </performance>
     ```
- only one date element in the document has any following siblings

Examples using count

- `//CD[count(performance)=2]` returns CD elements with exactly two `performance` elements as children: the last 3 CDs

Examples using count

- `//CD[count(performance)=2]` returns CD elements with exactly two `performance` elements as children: the last 3 CDs
- `//CD[performance][performance]` of course does *not* do this:
  - it is equivalent to `//CD[performance]`
  - which returns CD elements with at least one `performance` child

More examples using count

- Assume we want the CDs containing only one `orchestra` element
- `//CD[count(orchestra)=1]` returns only one CD, where the orchestra is “London Symphony Orchestra”
- This is because we are counting the orchestra *children* of `CD` elements
- But orchestras are also represented below `performance` elements

More examples using count

- Assume we want the CDs containing only one `orchestra` element
- `//CD[count(orchestra)=1]` returns only one CD, where the orchestra is "London Symphony Orchestra"
- This is because we are counting the orchestra *children* of `CD` elements
- But orchestras are also represented below `performance` elements
- What about `//CD[count(//orchestra)=1]`?
  - But `//orchestra` is an absolute expression evaluated at the root
  - So the answer to `count(//orchestra)`is 4, not 1

More examples using count

- Assume we want the CDs containing only one `orchestra` element
- `//CD[count(orchestra)=1]` returns only one CD, where the orchestra is “London Symphony Orchestra”
- This is because we are counting the orchestra *children* of `CD` elements
- But orchestras are also represented below `performance` elements
- What about `//CD[count(//orchestra)=1]`?
  - But `//orchestra` is an absolute expression evaluated at the root
  - So the answer to `count(//orchestra)`is 4, not 1
- What we need is `/CD[count(.//orchestra)=1]`, where “.” represents the current context node
  - This gives us the CDs with the “Berlin Philharmonic” and “London Symphony Orchestra”

String Functions

- *String functions* include basic string operations
- Examples:
  - `string-length()`: returns the length of a string
  - `concat()`: concatenates its arguments in order from left to right and returns the combined string
  - `contains(`*s1*, *s2*`)`: returns true if *s2* is a substring of *s1*
  - `normalize-space()`: strips all leading and trailing whitespace from its argument

Boolean Functions

- *Boolean functions* always return a Boolean with the value true or false:
  - `true()`: simply returns true (makes up for the lack of Boolean literals in XPath)
  - `false()`: returns false
  - `not()`: inverts its argument (i.e., true becomes false and vice versa)

Boolean Functions

- *Boolean functions* always return a Boolean with the value true or false:
  - `true()`: simply returns true (makes up for the lack of Boolean literals in XPath)
  - `false()`: returns false
  - `not()`: inverts its argument (i.e., true becomes false and vice versa)
- Examples:
  - `//performance[orchestra][not(conductor)]` returns `performance` elements which have an `orchestra` child but no `conductor` child
  - `//CD[not(.//soloist)]` returns CDs containing no soloists

Boolean Functions (2)

- `boolean()`: converts its argument to a Boolean and returns the result
  - Numbers are false if they are zero or NaN (not a number)
  - Node sets are false if they are empty
  - Strings are false if they have zero length

Number Functions

- *Number functions* include a few simple numeric functions
- Examples:
  - `sum(set)`: converts each node in a node set to a number and returns the sum of these numbers
  - `round()`, `floor()`, `ceiling()`: round numbers to integer values

Summary

- XPath is used to navigate through elements and attributes in an XML document
- XPath is a major element in many W3C standards: XQuery, XSLT, XLink, XPointer
- It is also used to navigate XML trees represented in Java or JavaScript, e.g.
- So an understanding of XPath is fundamental to much advanced XML usage

Chapter 7

Optimising XPath Queries

Types of Optimisation

- In general, there are two types of query optimisation:
  - *logical* optimisation
  - *physical* optimisation
- Logical optimisation is concerned with, e.g., rewriting a given query to be *minimal* in size (i.e., to remove redundant parts)
- Physical optimisation refers to strategies to make query evaluation as efficient as possible
- In this chapter, we will study some aspects of logical optimisation for XPath
- Later chapters will discuss physical optimisation

XPath Fragment

- We will consider only a fragment of XPath
- Each location step is just
  - the name of an element, or
  - $*$, or
  - empty (giving rise to `//`)

  optionally followed by predicates

```xml
<bookstore>
 <book>
  <author><last-name>Abiteboul</last-name></author>
  <author><last-name>Hull</last-name></author>
  <author><last-name>Vianu</last-name></author>
  <title>Foundations of Databases</title>
  <isbn>0-201-53771-0</isbn>
  <price>26.95</price>
 </book>
 <magazine>
  <title>The Economist</title>
  <date><day>26</day><month>June</month><year>1999</year></date>
  <price>2.50</price>
 </magazine>
 <book>
  <isbn>0-934613-40-0</isbn>
  <price>34.95</price>
 </book>
</bookstore>
```

Some Queries on `bookstore`

On this specific document

- `/bookstore/book/isbn` gives the same result as `//isbn`
  - because every `isbn` is a child of `book` and every `book` is a child of `bookstore`
- `/bookstore/*/title` gives the same result as `/bookstore/(book|magazine)/title` and `//title`
  - because the only elements that can be children of `bookstore` and parents of `title` are either `book` or `magazine`
- `//magazine[date[day][month]]/title` gives the same result as `//magazine[date/day][date/month]/title`
  - because each `magazine` has only a single `date`

Some Queries on `bookstore`

On this specific document

- `/bookstore/book/isbn` gives the same result as `//isbn`
  - because every `isbn` is a child of `book` and every `book` is a child of `bookstore`
- `/bookstore/*/title` gives the same result as `/bookstore/(book|magazine)/title` and `//title`
  - because the only elements that can be children of `bookstore` and parents of `title` are either `book` or `magazine`
- `//magazine[date[day][month]]/title` gives the same result as `//magazine[date/day][date/month]/title`
  - because each `magazine` has only a single `date`

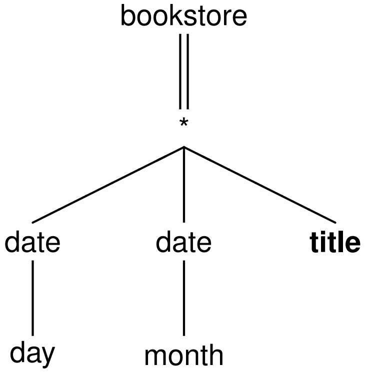
But these queries are *not* equivalent in general

XPath Queries as Tree Patterns

- We can view an XPath query Q in our fragment as a *tree pattern* P
- Each node test (element name or $*$) in Q becomes a node in P
- If Q has subexpression `A/B`, then nodes `A` and `B` in P are connected by a *single* edge
- If Q has subexpression `A//B`, then nodes `A` and `B` in P are connected by a *double* edge
- The node in P corresponding to the element name forming the output of Q is shown in boldface

Tree Pattern Example

`/bookstore//*[date/day][date/month]/title`



Containment and Equivalence of XPath Queries

- Given an XPath query Q and an XML tree t, the *answer* of evaluating Q on t is denoted by $Q(t)$
- For XPath queries P and Q, we say
  - P *contains* Q, written $P \supseteq Q$, if for all trees t, $P(t) \supseteq Q(t)$
  - P is *equivalent* to Q, written $P \equiv Q$, if $P \supseteq Q$ and $Q \supseteq P$
- Containment of XPath queries is useful
  - to show equivalence of queries for optimization
  - to determine if views can be used in query processing
  - to reuse cached query results

Examples of Containment and Equivalence

- `//isbn` $\supseteq$ `/bookstore/book/isbn`
  - There are no fewer isbns than isbns of books

Examples of Containment and Equivalence

- `//isbn` $\supseteq$ `/bookstore/book/isbn`
  - There are no fewer isbns than isbns of books
- `/bookstore/*/title` $\supseteq$ `/bookstore/book/title`
  - There are no fewer title that titles of books

Examples of Containment and Equivalence

- `//isbn` $\supseteq$ `/bookstore/book/isbn`
  - There are no fewer isbns than isbns of books
- `/bookstore/*/title` $\supseteq$ `/bookstore/book/title`
  - There are no fewer title that titles of books
- `book` $\supseteq$ `book[price]`
  - There are no fewer books than books with prices

Examples of Containment and Equivalence

- `//isbn` $\supseteq$ `/bookstore/book/isbn`
  - There are no fewer isbns than isbns of books
- `/bookstore/*/title` $\supseteq$ `/bookstore/book/title`
  - There are no fewer title that titles of books
- `book` $\supseteq$ `book[price]`
  - There are no fewer books than books with prices
- `date[year]` $\supseteq$ `date[month][year]`
  - There are no fewer dates with years than dates with years and months

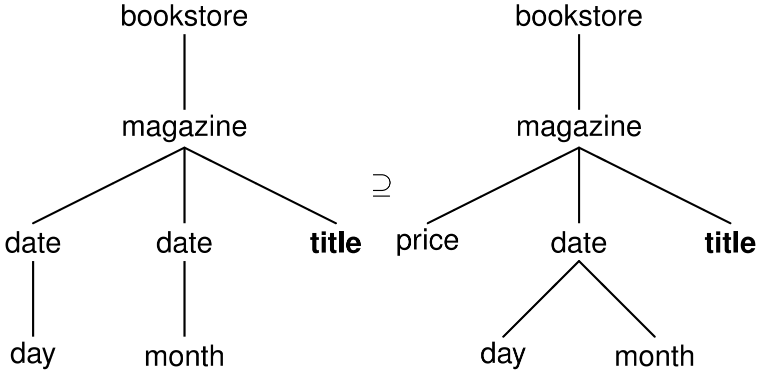
Examples of Containment and Equivalence

- `//isbn` $\supseteq$ `/bookstore/book/isbn`
  - There are no fewer isbns than isbns of books
- `/bookstore/*/title` $\supseteq$ `/bookstore/book/title`
  - There are no fewer title that titles of books
- `book` $\supseteq$ `book[price]`
  - There are no fewer books than books with prices
- `date[year]` $\supseteq$ `date[month][year]`
  - There are no fewer dates with years than dates with years and months
- `bookstore//title` $\supseteq$ `bookstore//book//title`
  - There are no fewer bookstores containing titles than bookstores containing books containing titles

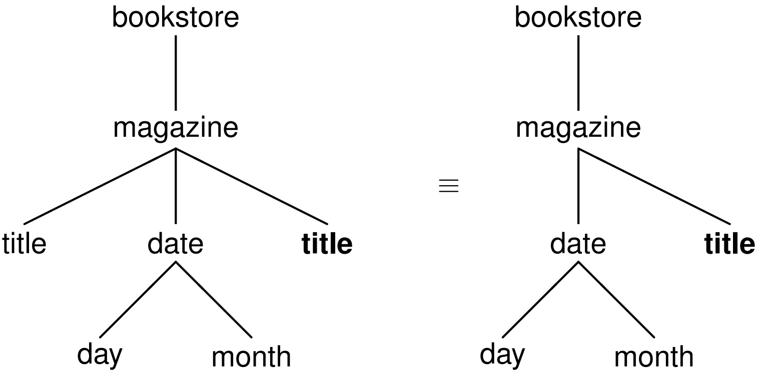
Examples of Containment and Equivalence

- `//isbn` $\supseteq$ `/bookstore/book/isbn`
  - There are no fewer isbns than isbns of books
- `/bookstore/*/title` $\supseteq$ `/bookstore/book/title`
  - There are no fewer title that titles of books
- `book` $\supseteq$ `book[price]`
  - There are no fewer books than books with prices
- `date[year]` $\supseteq$ `date[month][year]`
  - There are no fewer dates with years than dates with years and months
- `bookstore//title` $\supseteq$ `bookstore//book//title`
  - There are no fewer bookstores containing titles than bookstores containing books containing titles
- `magazine[date/year]` $\equiv$ `magazine[date/year][date]` so `[date]` is redundant

Example of Containment (tree patterns)



Example of Equivalence (tree patterns)



Using DTDs

- We can use DTDs to simplify expressions further
- Assume we know the document we want to query is valid with respect to a DTD D
- The DTD D specifies certain constraints
- e.g., every `book` element must have an `isbn` element as a child
- We already know that `/bookstore/book` $\supseteq$ `/bookstore/book[isbn]`
- Using the DTD D, we now know that `/bookstore/book` is *equivalent* to `/bookstore/book[isbn]`, but *only* when querying documents valid with respect to D

Constraints implied by a DTD

- Assume we are given the following DTD *D* (syntax simplified):

```
bookstore ((book|magazine)+)
book      (author*, title?, isbn, price)
author    (first-name?, last-name)
magazine  (title, volume?, issue?, date, price)
date      ((day?, month)?, year)
```

Constraints implied by a DTD

- Assume we are given the following DTD *D* (syntax simplified):

```
bookstore ((book|magazine)+)
book      (author*, title?, isbn, price)
author    (first-name?, last-name)
magazine  (title, volume?, issue?, date, price)
date      ((day?, month)?, year)
```

- Some constraints implied by the DTD *D*:
  - every `author` element must have a `last-name` child (*child constraint*)
  - every `date` element with a `day` child must have a `month` child (*sibling constraint*)
  - every `book` element has at most one `title` child (*functional constraint*)

Examples

- `/bookstore/book[price]/author` is equivalent to `/bookstore/*/author` since
  - every book must have a price
  - book must be the parent of author

Examples

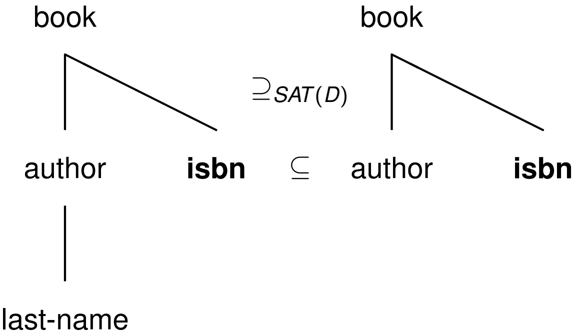
- `/bookstore/book[price]/author` is equivalent to `/bookstore/*/author` since
  - every book must have a price
  - book must be the parent of author
- `bookstore/book[author/first-name][author/last-name]` can first be rewritten as `bookstore/book[author/first-name][author]` and then as `book[author/first-name]`

Containment and Equivalence under DTDs

- We can use DTD constraints to find more equivalences
- When given a DTD D and a tree t known to satisfy D
- Let $SAT(D)$ denote the set of trees satisfying DTD D
- For XPath queries P and Q,
  - P *D-contains* Q, written $P \supseteq_{SAT(D)} Q$, if for all trees $t \in SAT(D)$, $P(t) \supseteq Q(t)$
  - P is *D-equivalent* to Q, written $P \equiv_{SAT(D)} Q$, if $P \supseteq_{SAT(D)} Q$ and $Q \supseteq_{SAT(D)} P$

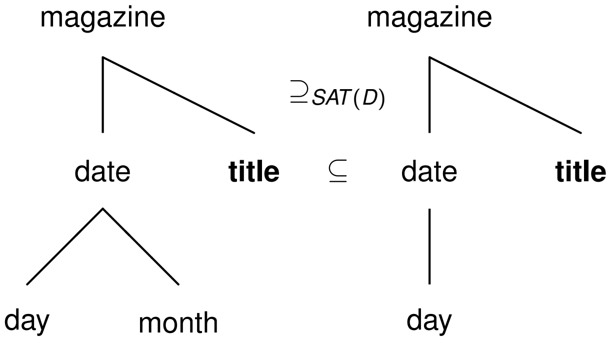
Example of *D*-Equivalence (Child Constraint)

- Every `author` must have a `last-name`



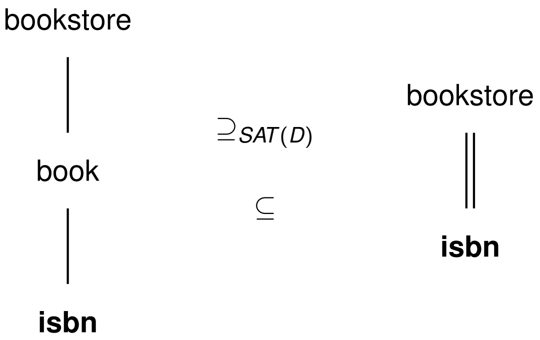
Example of *D*-Equivalence (Sibling Constraint)

- Every `date` with a `day` must have a `month`



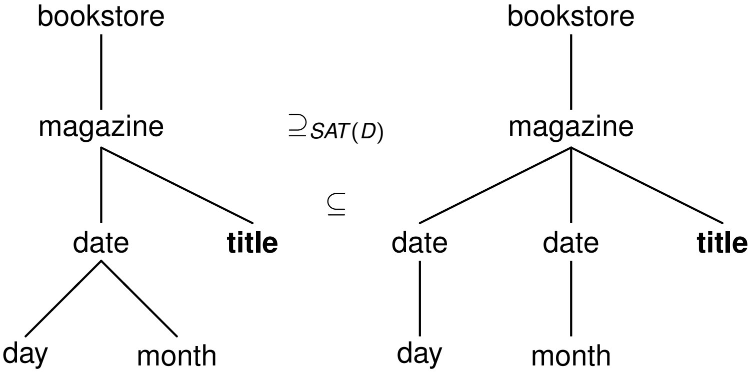
Example of *D*-Equivalence (Path Constraint)

- The only path from `bookstore` to `isbn` is through `book`



D-Equivalence Example (Functional Constraint)

- Every `magazine` has a single `date`



Summary

- We have considered logical optimisation of a fragment of XPath
- Can be used to delete redundant subexpressions from queries
- Further redundancies can be found when documents are valid with respect to a DTD
- We will consider efficient evaluation of XPath and some general physical optimisation techniques later

Chapter 8

Evaluating XPath Queries

Introduction

- When XML documents are small and can fit in memory, evaluating XPath expressions can be done efficiently
- But what if we have very large documents stored on disk?
- How should they be stored (fragmented)?
- How can we query them efficiently (by reducing the number of disk accesses needed)?

Fragmentation

- A large document will not fit on a single disk page (block)
- It will need to be *fragmented* over possibly a large number of pages
- Updates to the document may result in further fragmentation

Pre-order traversal

Recall pre-order traversal of a tree:

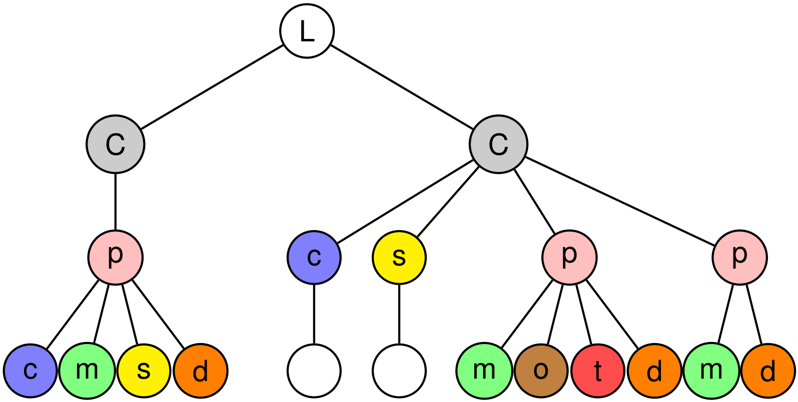
- To traverse a non-empty tree in pre-order, perform the following operations recursively at each node, starting with the root node:
  1. Visit the node
  2. Traverse the root nodes of subtrees of the node from left to right

Fragmentation based on pre-order traversal

A very simple method to store the document nodes on disk is as follows:

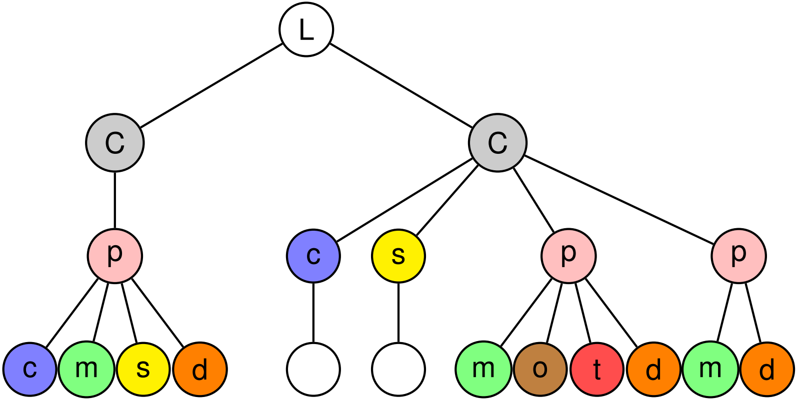
- A pre-order traversal of the document, starting from the root, groups as many nodes as possible within the current page
- When the page is full, a new page is used to store the nodes that are encountered next
- and so on, until the entire tree has been traversed

CD library example — first two CDs



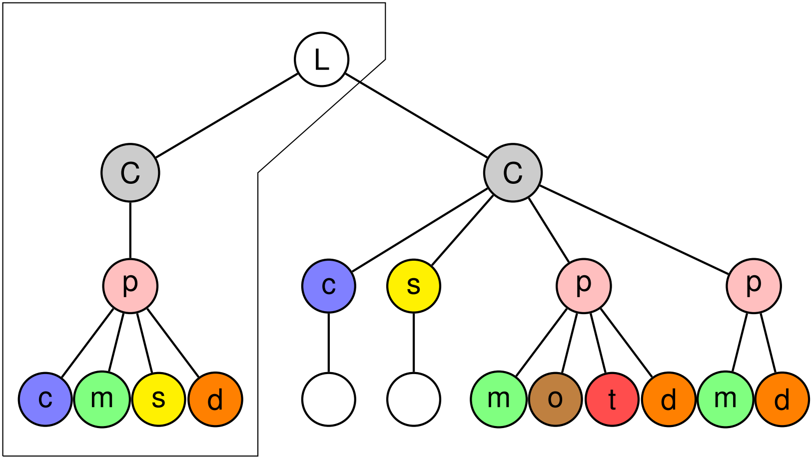
CD library example — first two CDs

Stored as 3 fragments



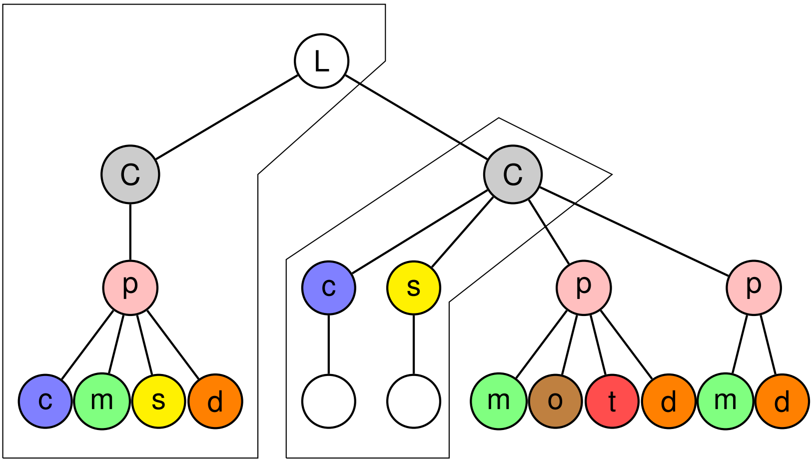
CD library example — first two CDs

Stored as 3 fragments



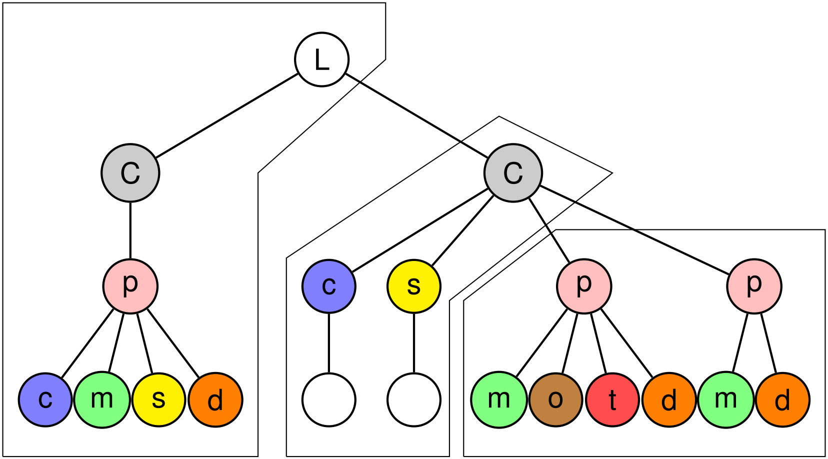
CD library example — first two CDs

Stored as 3 fragments



CD library example — first two CDs

Stored as 3 fragments



Simple XPath queries

- Selecting both CDs nodes requires accessing 2 fragments
- Evaluating `/CD-library/CD/performance` requires accessing all 3 fragments
- This is very small example, but one can see that such fragmentation could lead to very bad performance

Simple XPath queries

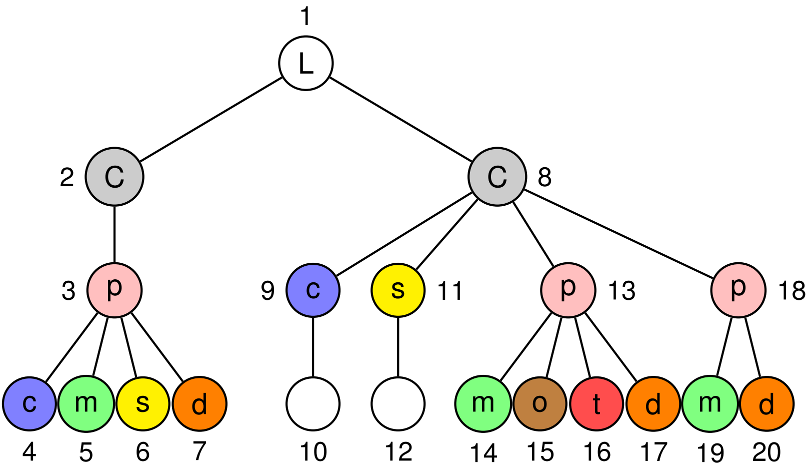
- Selecting both CDs nodes requires accessing 2 fragments
- Evaluating `/CD-library/CD/performance` requires accessing all 3 fragments
- This is very small example, but one can see that such fragmentation could lead to very bad performance
- Two improvements:
  - *Smart fragmentation*: Group those nodes that are often accessed simultaneously together
  - *Rich node identifiers*: Sophisticated node identifiers reducing the cost of join operations needed to "stitch" back fragments

Representation on disk

- One of the simplest ways to represent an XML document on disk is to
  - Assign an identifier to each node
  - Store the XML document as one relation (which may be fragmented) representing a set of edges

Simple node identfiers

Here node identifiers are simply integers, assigned in some order



The *Edge* relation

pid	cid	clabel
-	1	CD-library
1	2	CD
2	3	performance
3	4	composer
3	5	composition
3	6	soloist
3	7	date
1	8	CD
...	...	...

- "pid" is the id of the parent node
- "cid" is the id of the child node
- "clabel" is the element name of the child node
- (attributes and text nodes can be handled similarly)

Processing XPath queries

- `//composer`: can be evaluated by a simple lookup

$$\pi_{cid}(\sigma_{clabel=\text{'}composer\text{'}}(Edge))$$

Processing XPath queries

- `//composer`: can be evaluated by a simple lookup

$$\pi_{cid}(\sigma_{clabel='composer'}(Edge))$$

- `/CD-library/CD`: requires one join

$$\pi_{cid}((\sigma_{clabel='CD-library'}(Edge)) \bowtie_{cid=pid} (\sigma_{clabel='CD'}(Edge)))$$

Processing XPath queries (2)

- `/CD-library//composer`: many joins potentially needed

$$Let\ \ A := (\sigma_{clabel='CD-library'}(Edge))$$

$$Let\ \ B := (\sigma_{clabel='composer'}(Edge))$$

`/CD-library/composer`	$\pi_{cid}(A \bowtie_{cid=pid} B)$
`/CD-library/*/composer`	$\pi_{cid}(A \bowtie_{cid=pid} Edge \bowtie_{cid=pid} B)$
`/CD-library/*/*/composer`	...
...	...

- This assumes the query processor does not have any schema information available which might constrain where `composer` elements are located

Element-partitioned Edge relations

- A simple improvement is to use *element-partitioned* Edge relations
- Here, the Edge relation is partitioned into many relations, one for each element name

CD-library

pid	cid
-	1

CD

pid	cid
1	2
1	8

performance

pid	cid
2	3
8	13
8	18

composer

pid	cid
3	4
8	9

Element-partitioned Edge relations (2)

- This saves some space (element names are not repeated)
- It also reduces the disk I/O needed to retrieve the identifiers of elements having a given name
- However, it does not solve the problem of evaluating queries with // steps in non-leading positions

Path-partitioned approach to fragmentation

- *Path-partitioning* tries to solve the problem of `//` steps at arbitrary positions in a query
- This approach uses one relation for each distinct path in the document, e.g., `/CD-library/CD/performance`
- There is also another relation, called Paths, which contains all the unique paths

Path-partitioned storage

`/CD-library`:

pid	cid
-	1

`/CD-library/CD`:

pid	cid
1	2
1	8

`/CD-library/CD/composer`:

pid	cid
8	9

`/CD-library/CD/performance/composer`:

pid	cid
3	4

Paths:

path
`/CD-library`
`/CD-library/CD`
`/CD-library/CD/performance`
`/CD-library/CD/performance/composer`
...

Path-partitioned storage (2)

- Based on a path-partitioned store, a query such as `//CD//composer` can be evaluated in two steps:
  - Scan the Paths relation to identify all the paths matching the given XPath query
  - For each such path, scan the corresponding path-partitioned relation
- So for `//CD//composer`, the paths would be
  - `/CD-library/CD/composer`
  - `/CD-library/CD/performance/composer`
- So only these two relations need to be scanned

Path-partitioned storage (3)

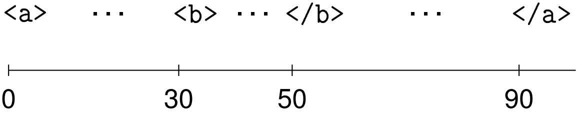
- The evaluation of XPath queries with many branches will still require joins across the relations
- However, the evaluation of // steps is simplified, thanks to the first processing step, performed on the path relation
- For very structured data, the path relation is typically small
- Thus, the cost of the first processing step is likely negligible, while the performance benefits of avoiding numerous joins are quite important
- However, for some data, the path relation can be large, and in some cases, even larger than the data itself

Node identifiers

- Node identifiers are needed to indicate how nodes are related to one another in an XML tree
- This is particularly important when the data is fragmented and we need to reconnect children with their parents
- However, it is often even more useful to be able to identify other kinds of relationships between nodes, just by looking at their identifiers
- This means we need to use identifiers that are richer than simple consecutive integers
- We will see later how this information can be used in query processing

Region-based identifiers

- The region-based identifier scheme assigns to each XML node *n* a pair of integers
- The pair consists of the offset of the node's start tag, and the offset of its end tag
- We denote this pair by $(n.start, n.end)$
- Consider the following offsets of tags:



- the region-based identifier of the `<a>` element is the pair $(0, 90)$
- the region-based identifier of the `<b>` element is the pair $(30, 50)$

Using region-based identifiers

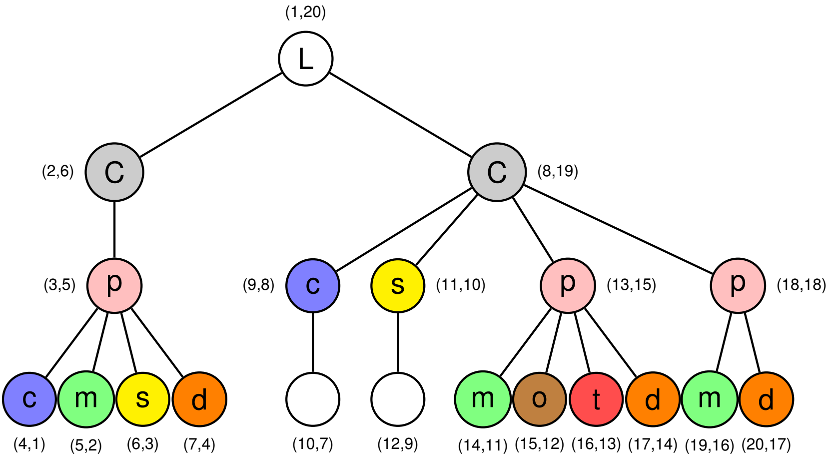
- Comparing the region-based identifiers of two nodes n_1 and n_2 allows for deciding whether n_1 is an ancestor of n_2
- Observe that this is the case if and only if:
  - $n_1.start < n_2.start$, and
  - $n_1.end > n_2.end$
- There is no need to use byte offsets:
  - (Start tag, end tag). Count only opening and closing tags (as one unit each) and assign the resulting counter values to each element
  - (Pre, post). Pre-order and post-order index (see next slides)
- Region-based identifiers are quite compact, as their size only grows logarithmically with the number of nodes in a document

Post-order traversal

Recall post-order traversal of a tree:

- To traverse a non-empty tree in post-order, perform the following operations recursively at each node, starting with the root node:
    1. Traverse the root nodes of subtrees of the node from left to right
    2. Visit the node

Example of (pre, post) node identifiers



Using (pre, post) identifiers to find ancestors

- The same method as for other region-based identifiers allows us to decide, for two nodes n_1 and n_2, whether n_1 is an *ancestor* of n_2
- As before, this is the case if and only if:
  - $n_1.pre < n_2.pre$, and
  - $n_1.post > n_2.post$

  where $n_i.pre$ and $n_i.post$ are the pre-order and post-order numbers assigned to node n_i, respectively

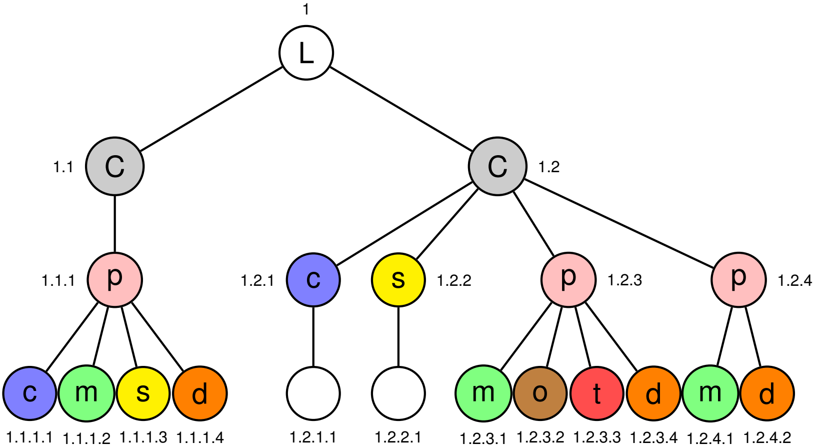
Using (pre, post) identifiers to find parents

- One can add another number to a node identifier which indicates the *depth* of the node in the tree
- The root is assigned a depth of 1; the depth increases as we go down the tree
- Using $(pre, post, depth)$, we can decide whether node n_1 is a *parent* of node n_2
- Node n_1 is a parent of node n_2 if and only if
  - n_1 is an ancestor of n_2 and
  - $n_1.depth = n_2.depth - 1$

Dewey-based identifiers

- These identifiers use the principal of the Dewey classification system used in libraries for decades
- To get the identifier of a child node, one adds a suffix to the identifier of its parent (including a separator)
- e.g., if the parent's identifier is 1.2.3 and the child is the second child of this parent, then its identifier is 1.2.3.2

Example of Dewey-based identifiers



Using Dewey-based identifiers

- Let n_1 and n_2 be two identifiers, of the form: $n_1 = x_1.x_2.\ldots.x_m$ and $n_2 = y_1.y_2.\ldots.y_n$
- The node identified by n_1 is an ancestor of the node identified by n_2 if and only if n_1 is a *prefix* of n_2
- When this is the case, the node identified by n_1 is the *parent* of the node identified by n_2 if and only if $n = m + 1$
- Dewey IDs allow finding other relationships such as preceding-sibling and preceding (respectively, following-sibling, and following)
- The node identified by n_1 is a preceding sibling of the node identified by n_2 if and only if
  1. $x_1.x_2.\ldots.x_{m-1} = y_1.y_2.\ldots.y_{n-1}$ and
  2. $x_m < y_n$
- The main drawback of Dewey identifiers is their length: the length is variable and can get large

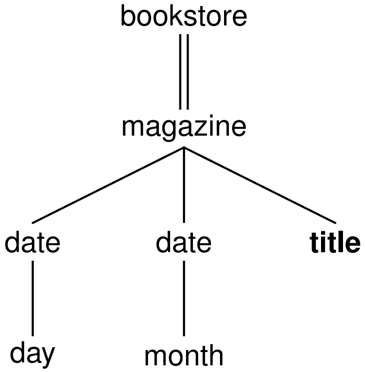
Structural identifiers and updates

- Consider a node with Dewey ID 1.2.2.3
  - Suppose we insert a new first child for node 1.2
  - Then the ID of node 1.2.2.3 becomes 1.2.3.3
- In general:
  - Offset-based identifiers need to be updated as soon as a character is inserted or removed in the document
  - (start, end), (pre, post), and Dewey IDs need to be updated when the elements of the documents change
  - It is possible to avoid re-labelling on deletions, but gaps will appear in the labelling scheme
  - Re-labelling operations are quite expensive

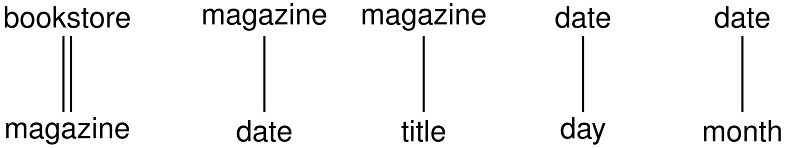
Tree pattern query evaluation

- Assume we have element-partitioned relations using (pre, post) identifiers
- Assume we want to evaluate a tree pattern query
- One way is to decompose the query into its “basic” patterns:
  - Each basic pattern is just a pair of nodes
  - connected by a child edge or a descendant edge
- We particularly want an efficient way of evaluating basic patterns that use the descendant operator

Tree Pattern Example

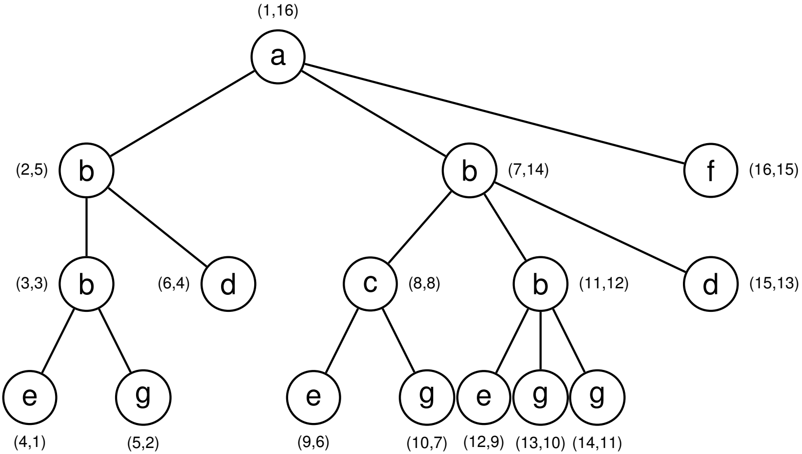


Decomposed Tree Pattern Example



Example tree with (pre, post) identifiers

(Taken from the book “Web Data Management”)



Element-partitioned relations for example

a

pre	post
1	16

b

pre	post
2	5
3	3
7	14
11	12

c

pre	post
8	8

d

pre	post
6	4
15	13

e

pre	post
4	1
9	6
12	9

f

pre	post
16	15

g

pre	post
5	2
10	7
13	10
14	11

Evaluation of descendant patterns

- Assume we want to evaluate the basic pattern corresponding to `b//g`
- This pattern may need to be joined to the results calculated for other basic patterns
- So, in general, we need to find all pairs (x, y) of nodes where
  - x is an element with name `b`
  - y is an element with name `g`
  - y is a descendant of x

Evaluation of descendant patterns (2)

- We could take every node ID from the b relation and compare it to every node ID from the g relation
- Each time we can test whether the g-node is a descendant of the b-node using the (pre, post) identifiers
- But this method will take time proportional to $n \times m$, if there are n b-nodes and m g-nodes
- In particular, one of the relations is scanned many times
- This is similar to a nested-loops implementation of a relational join, which is known to be inefficient
- Can we do better?

Stack-based join algorithm

- We will look at an elegant method for evaluation of descendant patterns that uses an auxiliary *stack*
- This is called the *stack-based join* (SBJ) algorithm
- SBJ reads each ID from each relation only *once*
- SBJ assumes that the IDs in each relation are *sorted*, essentially by their pre-order values (as in the earlier slide)
- We will illustrate the method by example

Stack-based join algorithm — example

b IDs	g IDs
(2,5)	(5,2)
(3,3)	(10,7)
(7,14)	(13,10)
(11,12)	(14,11)

Stack

Stack-based join algorithm — example

b IDs	g IDs
	(5,2)
(3,3)	(10,7)
(7,14)	(13,10)
(11,12)	(14,11)

Stack
(2,5)

- SBJ starts by pushing the first ancestor (that is, b node) ID, namely (2,5), on the stack

Stack-based join algorithm — example

	(5,2)
(3,3)	(10,7)
(7,14)	(13,10)
(11,12)	(14,11)
b IDs	g IDs

(2,5)
Stack

- SBJ starts by pushing the first ancestor (that is, b node) ID, namely (2,5), on the stack
- Then, STD continues to examine the IDs in the b ancestor input
- While the current ancestor ID is a descendant of the top of the stack, the current ancestor ID is pushed on the stack

Stack-based join algorithm — example

	(5,2)
	(10,7)
(7,14)	(13,10)
(11,12)	(14,11)
b IDs	g IDs

(3,3)
(2,5)
Stack

- SBJ starts by pushing the first ancestor (that is, b node) ID, namely (2,5), on the stack
- Then, STD continues to examine the IDs in the b ancestor input
- While the current ancestor ID is a descendant of the top of the stack, the current ancestor ID is pushed on the stack
- So the second b ID, (3,3), is pushed on the stack, since it is a descendant of (2,5)

Stack-based join algorithm — example (2)

	(5,2)
	(10,7)
(7,14)	(13,10)
(11,12)	(14,11)
b IDs	g IDs

(3,3)
(2,5)
Stack

Output

- The third ID in the b input, (7,14), is not a descendant of current stack top, namely (3,3)
- Therefore, SBJ stops pushing b IDs on the stack and considers the first descendant ID, to see if it has matches on the stack

Stack-based join algorithm — example (2)

	(10,7)
(7,14)	(13,10)
(11,12)	(14,11)
b IDs	g IDs

(3,3)
(2,5)
Stack

Output
(3,3), (5,2)
(2,5), (5,2)

- The third ID in the b input, (7,14), is not a descendant of current stack top, namely (3,3)
- Therefore, SBJ stops pushing b IDs on the stack and considers the first descendant ID, to see if it has matches on the stack
- The first g node, namely (5,2), is a descendant of both b nodes on the stack, leading to the first two output tuples

Stack-based join algorithm — example (2)

b IDs	g IDs
	(10,7)
(7,14)	(13,10)
(11,12)	(14,11)

Stack
(3,3)
(2,5)

Output
(3,3), (5,2)
(2,5), (5,2)

- The third ID in the b input, (7,14), is not a descendant of current stack top, namely (3,3)
- Therefore, SBJ stops pushing b IDs on the stack and considers the first descendant ID, to see if it has matches on the stack
- The first g node, namely (5,2), is a descendant of both b nodes on the stack, leading to the first two output tuples
- Note that the stack does not change when output is produced
- This is because there may be further descendant IDs to match the ancestor IDs on the stack

Stack-based join algorithm — example (3)

b IDs	g IDs
	(10,7)
(7,14)	(13,10)
(11,12)	(14,11)

Stack
(3,3)
(2,5)

Output
(3,3), (5,2)
(2,5), (5,2)

- A descendant ID which has been compared with ancestor IDs on the stack and has produced output tuples, can be discarded
- Now the g ID (10,7) encounters no matches on the stack
- Moreover, (10,7) occurs in the document after the nodes on the stack
- Therefore, no descendant node ID yet to be examined can have ancestors on this stack
- This is because the input g IDs are sorted

Stack-based join algorithm — example (3)

	(10,7)
(7,14)	(13,10)
(11,12)	(14,11)
b IDs	g IDs

Stack

Output
(3,3), (5,2)
(2,5), (5,2)

- A descendant ID which has been compared with ancestor IDs on the stack and has produced output tuples, can be discarded
- Now the g ID (10,7) encounters no matches on the stack
- Moreover, (10,7) occurs in the document after the nodes on the stack
- Therefore, no descendant node ID yet to be examined can have ancestors on this stack
- This is because the input g IDs are sorted
- Therefore, at this point, the stack is emptied

Stack-based join algorithm — example (4)

b IDs	g IDs
	(10,7)
(7,14)	(13,10)
(11,12)	(14,11)

Stack

Output
(3,3), (5,2)
(2,5), (5,2)

Stack-based join algorithm — example (4)

	(10,7)
	(13,10)
(11,12)	(14,11)
b IDs	g IDs

(7,14)
Stack

Output
(3,3), (5,2)
(2,5), (5,2)

- Next the ancestor ID (7,14) is pushed on the stack

Stack-based join algorithm — example (4)

b IDs	g IDs
	(10,7)
	(13,10)
	(14,11)

Stack
(11,12)
(7,14)

Output
(3,3), (5,2)
(2,5), (5,2)

- Next the ancestor ID (7,14) is pushed on the stack
- followed by its descendant, in the ancestor input, (11,12)

Stack-based join algorithm — example (4)

b IDs	g IDs
	(10,7)
	(13,10)
	(14,11)

Stack
(11,12)
(7,14)

Output
(3,3), (5,2)
(2,5), (5,2)

- Next the ancestor ID (7,14) is pushed on the stack
- followed by its descendant, in the ancestor input, (11,12)
- The next descendant ID is (10,7)

Stack-based join algorithm — example (4)

b IDs	g IDs
	(13,10)
	(14,11)

Stack
(11,12)
(7,14)

Output
(3,3), (5,2)
(2,5), (5,2)
(7,14), (10,7)

- Next the ancestor ID (7,14) is pushed on the stack
- followed by its descendant, in the ancestor input, (11,12)
- The next descendant ID is (10,7)
- This produces a result with (7,14) and is then discarded

Stack-based join algorithm — example (5)

	(13,10)
	(14,11)
b IDs	g IDs

(11,12)
(7,14)
Stack

Output
(3,3), (5,2)
(2,5), (5,2)
(7,14), (10,7)

Stack-based join algorithm — example (5)

b IDs	g IDs
	(13,10)
	(14,11)

Stack
(11,12)
(7,14)

Output
(3,3), (5,2)
(2,5), (5,2)
(7,14), (10,7)

- The next descendant ID is (13,10)

Stack-based join algorithm — example (5)

b IDs	g IDs
	(14,11)

Stack
(11,12)
(7,14)

Output
(3,3), (5,2)
(2,5), (5,2)
(7,14), (10,7)
(11,12), (13,10)
(7,14), (13,10)

- The next descendant ID is (13,10)
- This leads to two new tuples added to the output

Stack-based join algorithm — example (5)

b IDs	g IDs
	(14,11)

Stack
(11,12)
(7,14)

Output
(3,3), (5,2)
(2,5), (5,2)
(7,14), (10,7)
(11,12), (13,10)
(7,14), (13,10)

- The next descendant ID is (13,10)
- This leads to two new tuples added to the output
- The next descendant ID is (14,11)

Stack-based join algorithm — example (5)

b IDs	g IDs

Stack
(11,12)
(7,14)

Output
(3,3), (5,2)
(2,5), (5,2)
(7,14), (10,7)
(11,12), (13,10)
(7,14), (13,10)
(11,12), (14,11)
(7,14), (14,11)

- The next descendant ID is (13,10)
- This leads to two new tuples added to the output
- The next descendant ID is (14,11)
- This also leads to two more output tuples

Other approaches

- The stack-based join algorithm is as efficient as possible for single descendant basic patterns
- But an overall algorithm for tree pattern evaluation still has to join the answers from basic patterns back together
- The size of intermediate results can be unnecessarily large
- Another approach is to evaluate the entire pattern in one operation
- One algorithm for this is called *holistic twig join*

Summary

- We considered some issues for dealing with querying large XML documents
- These included methods for fragmenting documents
- and efficient evaluation methods, particularly for ancestor-descendant basic patterns
- For more information, see Chapter 4 on “XML Query Evaluation” in the book “Web Data Management”
- The original stack-based join algorithm is from S. Al-Khalifa, H.V. Jagadish, J.M. Patel, Y. Wu, N. Koudas, and D. Srivastava. “Structural joins: A primitive for efficient XML query pattern matching.” In Proc. Int. Conf. on Data Engineering (ICDE), 2002.
- Holistic twig join is described in N. Bruno, N. Koudas, and D. Srivastava. “Holistic twig joins: optimal XML pattern matching.” In Proc. ACM Int. Conf. on the Management of Data (SIGMOD), 2002.

Chapter 9

XQuery

Motivation

- Now that we have XPath, what do we need XQuery for?
- XPath was designed for addressing parts of existing XML documents
- XPath cannot
  - create new XML nodes
  - perform joins between parts of a document (or many documents)
  - re-order the output it produces
  - ...
- Furthermore, XPath
  - has a very simple type system
  - can be hard to read and understand (due to its conciseness)

Data Model

- XQuery closely follows the XML Schema data model
- The most general data type is an *item*
- An item is either a (single) node or an atomic value

Data Model (2)

- XQuery works on *sequences*, which are series of items
- In XQuery every value is a sequence
  - There is no distinction between a single item and a sequence of length one
- Sequences can only contain items; they cannot contain other sequences

Document Representation

- Every document is represented as a tree of nodes
- Every node has a unique node identity that distinguishes it from other nodes (independent of any ID attributes)
- The first node in any document is the document node (which contains the whole document)
- The order in which the nodes occur in an XML document is called the *document order*

Document Representation (2)

- Attributes are not considered children of an element
  - They occur after their element and before its first child
  - The relative order within the attributes of an element is implementation-dependent

Query Language

- We are now going to look at the query language itself
  - Basics
  - Creating nodes/documents
  - FLWOR expressions
  - Advanced topics

Comments

- XQuery uses “smileys” to begin and end comments:
  `(:  This is a comment :)`
- These are comments found in a query (to comment the query)
  - Not to be confused with comments in XML documents

Literals

- XQuery supports numeric and string literals
- There are three kinds of numeric literals
  - Integers (e.g. 3)
  - Decimals (e.g. -1.23)
  - Doubles (e.g. 1.2e5)
- String literals are delimited by quotation marks or apostrophes
  - “a string”
  - ’a string’
  - ’This is a “string”’

Input Functions

- XQuery uses input functions to identify the data to be queried
- There are two different input functions, each taking a single argument
  - `doc()`
    - Returns an entire document (i.e. the document node)
    - Document is identified by a Universal Resource Identifier (URI)
  - `collection()`
    - Returns any sequence of nodes that is associated with a URI
    - How the sequence is identified is implementation-dependant
    - For example, eXist allows a database administrator to define collections, each containing a number of documents

Sample Data

- In order to illustrate XQuery queries, we use a sample data file `books.xml` which is based on bibliography data

```
<bib>

  <book year='1994'>
    <title>TCP/IP Illustrated</title>
    <author>
      <last>Stevens</last>
      <first>W.</first>
    </author>
    <publisher>Addison Wesley</publisher>
    <price>65.95</price>
  </book>
```

Sample Data (cont'd)

```
<book year='1992'>
  <title>
    Advanced Programming in the UNIX environment
  </title>
  <author>
    <last>Stevens</last>
    <first>W.</first>
  </author>
  <publisher>Addison Wesley</publisher>
  <price>65.95</price>
</book>
```

Sample Data (cont'd)

```
<book year='2000'>
  <title>Data on the Web</title>
  <author>
    <last>Abiteboul</last> <first>Serge</first>
  </author>
  <author>
    <last>Buneman</last> <first>Peter</first>
  </author>
  <author>
    <last>Suciu</last> <first>Dan</first>
  </author>
  <publisher>Morgan Kaufmann</publisher>
  <price>39.95</price>
</book>
```

Sample Data (cont'd)

```
  <book year='1999'>
    <title>
      The Economics of Technology and Content for Digital TV
    </title>
    <editor>
      <last>Gerbarg</last>
      <first>Darcy</first>
      <affiliation>CITI</affiliation>
    </editor>
    <publisher>Kluwer Academic</publisher>
    <price>129.95</price>
  </book>

</bib>
```

Input Functions (2)

- `doc("books.xml")` returns the entire document
- A run-time error is raised if the `doc` function is unable to locate the document

Input Functions (3)

- XQuery uses XPath to locate nodes in XML data
- An XPath expression can be appended to a `doc` (or `collection`) function to select specific nodes
- For example, `doc("books.xml")//book` returns all book nodes of `books.xml`

Creating Nodes

- So far, XQuery does not look much more powerful than XPath
- We only located nodes in XML documents
- Now we take a look at how to create nodes
- Note that this creates nodes in the *output* of a query; it does *not* update the document being queried

Creating Nodes (2)

- Elements, attributes, text nodes, processing instructions, and comment nodes can all be created using the same syntax as XML
- The following element constructor creates a book element:

```
<book year='1977'>
  <title>Harold and the Purple Crayon</title>
  <author>
    <last>Johnson</last>
    <first>Crockett</first>
  </author>
  <publisher>
    Harper Collins Juvenile Books
  </publisher>
  <price>14.95</price>
</book>
```

Creating Nodes (3)

- Document nodes do not have an explicit syntax in XML
- XQuery provides a special document node constructor
- The query

  ```
  document {}
  ```

  creates an empty document node

Creating Nodes (4)

- Document node constructor can be combined with other constructors to create entire documents

```
document {
  <?xml-stylesheet type='text/xsl' href='trans.xslt'?>
  <!-- I love this book -->
  <book year='1977'>
    <title>Harold and the Purple Crayon</title>
    <author>
      <last>Johnson</last>
      <first>Crockett</first>
    </author>
    <publisher>
      Harper Collins Juvenile Books
    </publisher>
    <price>14.95</price>
  </book>
}
```

Creating Nodes (5)

- Constructors can be combined with other XQuery expressions to generate content dynamically
- In element constructors, curly braces { } delimit enclosed expressions which are evaluated to create content
- Enclosed expressions may occur in the content of an element or the value of an attribute

Creating Nodes (6)

- This query creates a list of book titles from `books.xml`

```
<titles count =
 '{ count(doc("books.xml")//title) }'>
 {
  doc("books.xml")//title
 }
</titles>
```

- The result is:

```
<titles count="4">
    <title>TCP/IP Illustrated</title>
    <title>Advanced Programming ...</title>
    <title>Data on the Web</title>
    <title>The Economics of ...</title>
</titles>
```

Whitespace

- Implementations may discard boundary whitespace (whitespace between tags with no intervening non-whitespace)
- This whitespace can be preserved by an `xmlspace` declaration in the *prolog* of a query
- The prolog of a query is an optional section setting up the compile-time context for the rest of the query

Whitespace (2)

- The following query declares that all whitespace in element constructors must be preserved (which will output the element in exactly the same format)

```
declare xmlspace preserve;

<author>
  <last>Stevens</last>
  <first>W.</first>
</author>
```

- Omitting this declaration (or setting the mode to `strip`) will give:

```
<author><last>Stevens</last><first>W.</first></author>
```

Combining and Restructuring

- The expressiveness of XQuery goes beyond just creating nodes
- Information from one or more sources can be combined and restructured to create new results
- We are going to have a look at the most important expressions and functions

FLWOR

- FLWOR expressions (pronounced “flower”) are one of the most powerful and common expressions in XQuery
- Syntactically, they show similarity to the select-from-where statements in SQL
- However, FLWOR expressions do not operate on tables, rows, and columns

FLWOR (2)

- The name FLWOR is an acronym standing for the first letter of the clauses that may appear
  - For
  - Let
  - Where
  - Order by
  - Return

FLWOR (3)

- The acronym FLWOR roughly follows the order in which the clauses occur
- A FLWOR expression
  - starts with one or more `for` or `let` clauses (in any order)
  - followed by an optional `where` clause,
  - an optional `order by` clause,
  - and a required `return` clause

For and Let Clauses

- Every clause in a FLWOR expression is defined in terms of tuples
- The `for` and `let` clauses create these tuples
- Therefore, every FLWOR expression must have at least one `for` or `let` clause
- We will start with artificial-looking queries to illustrate the inner workings of `for` and `let` clauses

For and Let Clauses (2)

- The following query creates an element named `tuple` in its return clause

```
for $i in (1, 2, 3)
return
  <tuple><i> { $i } </i></tuple>
```

- We bind the variable `$i` to the expression `(1, 2, 3)`, which constructs a sequence of integers
- The above query results in:

```
<tuple><i>1</i></tuple>
<tuple><i>2</i></tuple>
<tuple><i>3</i></tuple>
```

(a `for` clause preserves order when it creates tuples)

For and Let Clauses (3)

- A `let` clause binds a variable to the entire result of an expression
- If there are no `for` clauses, then a single tuple is created

```
let $i := (1, 2, 3)
return
  <tuple><i> { $i } </i></tuple>
```

results in:

```
<tuple><i>1 2 3</i></tuple>
```

For and Let Clauses (4)

- Variable bindings of `let` clauses are added to the tuples generated by `for` clauses

```
for $i in (1, 2, 3)
let $j := ('a', 'b', 'c')
return
  <tuple><i>{ $i }</i><j>{ $j }</j></tuple>
```

results in:

```
<tuple><i>1</i><j>abc</j></tuple>
<tuple><i>2</i><j>abc</j></tuple>
<tuple><i>3</i><j>abc</j></tuple>
```

For and Let Clauses (5)

- `for` and `let` clauses can be bound to any XQuery expression
- Let us do a more realistic example
- List the title of each book in `books.xml` together with the numbers of authors:

```
for $b in doc("books.xml")//book
let $a := $b/author
return
  <book> { $b/title,
    <count> { count($a) } </count> }
  </book>
```

For and Let Clauses (6)

- This results in:

```
<book>
  <title>TCP/IP Illustrated</title>
  <count>1</count>
</book>
<book>
  <title>Advanced Programming ...</title>
  <count>1</count>
</book>
<book>
  <title>Data on the Web</title>
  <count>3</count>
</book>
<book>
  <title>The Economics of Technology ...</title>
  <count>0</count>
</book>
```

Where Clauses

- A `where` clause eliminates tuples that do not satisfy a particular condition
- A return clause is only evaluated for tuples that “survive” the where clause
- The following query returns only books whose prices are less than 50.00:

```
for $b in doc("books.xml")//book
where $b/price < 50.00
return $b/title
```

returns

```
<title>Data on the Web</title>
```

Order By Clauses

- An `order by` clause sorts the tuples before the return clause is evaluated
- If there is no `order by` clause, then the results are returned in document order
- The following example lists the titles of books in alphabetical order:

```
for $t in doc("books.xml")//title
order by $t
return $t
```

- An order spec may also specify whether to sort in ascending or descending order (using `ascending` or `descending`)

Return Clauses

- Any XQuery expression may occur in a `return` clause
- Element constructors are very common in `return` clauses
- The following query represents an author's name as a string in a single element

```
for $a in doc("books.xml")//author
return
  <author> { string($a/first), " ",
             string($a/last) } </author>
```

results in

```
<author>W. Stevens</author>
<author>W. Stevens</author>
<author>Serge Abiteboul</author>
<author>Peter Buneman</author>
<author>Dan Suciu</author>
```

Return Clauses (2)

- The following query adds another level to the hierarchy:

```
for $a in doc("books.xml")//author
return
  <author>
    <name> { $a/first, $a/last } </name>
  </author>
```

results in

```
<author>
    <name>
        <first>W.</first>
        <last>Stevens</last>
    </name>
</author>
...
```

Operators

- The operators shown in the queries so far have not been covered yet
- XQuery has three different kinds of operators
  - Arithmetic operators
  - Comparison operators
  - Sequence operators

Arithmetic Operators

- XQuery supports the arithmetic operators `+`, `-`, `*`, `div`, `idiv`, and `mod`
- The `idiv` and `mod` operators require integer arguments, returning the quotient and the remainder, respectively
- If an operand is a node, atomization is applied (casting the content to an atomic type)
- If an operand is an empty sequence, the result is an empty sequence
- If an operand is untyped, it is cast to a double (raising an error if the cast fails)

Comparison Operators

- XQuery has different sets of comparison operators: value comparisons, general comparisons, node comparisons, and order comparisons
- Value comparison operators compare atomic values:

eq	equals
ne	not equals
lt	less than
le	less than or equal to
gt	greater than
ge	greater than or equal to

General Comparisons

- The following query raises an error

  ```
  for $b in doc("books.xml")//book
  where $b/author/last eq 'Stevens'
  return $b/title
  ```

  because we try to compare several author names to `'Stevens'` (books may have more than one author)
- We need a general comparison operator for this to work
- A general comparison returns true if **any** value in a sequence of atomic values matches

General Comparisons (2)

- The following table shows the corresponding general comparison operator for each value comparison operator

value comparison	general comparison
eq	=
ne	!=
lt	$<$
le	$<=$
gt	$>$
ge	$>=$

Built-in Functions

- XQuery also offers a set of built-in functions and operators
- We focus only on the most common ones here
- SQL users will be familiar with the `min()`, `max()`, `count()`, `sum()`, and `avg()` functions
- Other familiar functions include
  - Numeric functions like `round()`, `floor()`, and `ceiling()`
  - String functions like `concat()`, `string-length()`, `substring()`, `upper-case()`, `lower-case()`
  - Cast functions for the various atomic types

User-Defined Functions

- When a query becomes large and complex, it becomes easier to understand if it is split up into functions
- A function is declared in the XQuery prolog
- Because the default namespace used for functions in XQuery corresponds to the XPath functions, a user-defined function has to be declared in a different namespace
- This is done by declaring a namespace and associated prefix
- For example, if the titles of books written by a given author are needed in different places in a query, a function could be declared and invoked as shown on the next slide

User-Defined Functions (2)

- The function is declared as follows:

```
declare namespace my="urn:local";
declare function my:books-by-author($last, $first)
 as element()*
{
  for $b in doc("books.xml")//book
  for $a in $b/author
  where $a/first = $first and $a/last = $last
  return $b/title
};
```

- It can be invoked as follows:

```
my:books-by-author('Abiteboul','Serge')
```

Library Modules

- Functions can be put into library modules, which can be imported by any query
- Every module in XQuery is either a main module (which contains a query body) or a library module (which has no query body)
- A library module begins with a module declaration which provides a URI for identification:

```
module "http://example.com/xq/book"

declare function ...

declare function ...
```

Library Modules (2)

- Any module can import another module using a `import module` declaration
- This declaration has to specify a URI and may specify a location where the module can be found

```
import module "http://example.com/xq/book"
       at "file:///home/xquery/..."
```

Positional Variables

- The `for` clause supports positional variables
- This identifies the position of a given item in the sequence generated by an expression
- The following query returns the titles of books with an attribute that numbers the books:

```
for $t at $i in doc("books.xml")//title
return
  <title pos=' { $i } '>
    { string($t) }
  </title>
```

Positional Variables (2)

- The output of this query looks like this:

```
<title pos="1">
  TCP/IP Illustrated
</title>
<title pos="2">
  Advanced Programming in ...
</title>
<title pos="3">
  Data on the Web
</title>
<title pos="4">
  The Economics of Technology ...
</title>
```

Eliminating Duplicates

- Data (or intermediate query results) often contain duplicate values
- The following query returns one of the authors twice

```
doc("books.xml")//author/last
```

which outputs

```
<last>Stevens</last>
<last>Stevens</last>
<last>Abiteboul</last>
<last>Buneman</last>
<last>Suciu</last>
```

Eliminating Duplicates (2)

- The `distinct-values()` function is used to remove duplicate values
- It extracts values of a sequence of nodes and creates a sequence of unique values
- Example:

  ```
  distinct-values(doc("books.xml")//author/last)
  ```

  which outputs

  ```
  Stevens Abiteboul Buneman Suciu
  ```

Combining Data Sources

- A query may bind multiple variables in a `for` clause to combine data from different expressions
- Suppose we have a file named `reviews.xml` that contains book reviews:

```
<reviews>
  <entry>
    <title>Data on the Web</title>
    <price>34.95</price>
    <review>
      A very good discussion of
      semi-structured databases ...
    </review>
  </entry>
    ...
```

Combining Data Sources (2)

- A FLWOR expression can bind one variable to the bibliography data and another to the review data
- In the following query we join data from the two files:

```
for $t in doc("books.xml")//title,
    $e in doc("reviews.xml")//entry
where $t = $e/title
return
  <review>
    { $t, $e/review }
  </review>
```

Combining Data Sources (3)

- This returns the following answer:

```
<review>
  <title>TCP/IP Illustrated</title>
  <review>
    One of the best books on TCP/IP.
  </review>
</review>
<review>
  <title>Advanced Programming in the ...</title>
  <review>
    A clear and detailed discussion of ...
  </review>
</review>
...
```

Inverting Hierarchies

- XQuery can be used to do general transformations
- In the example file, books are sorted by title
- If we want to group books by publisher, we have to “pull up” the publisher element (i.e., invert the hierarchy of the document)
- The next slide shows a query to do this

Inverting Hierarchies (2)

```
<listings> {
  for $p in
    distinct-values(doc("books.xml")//publisher)
  order by $p
  return
    <result>
      { $p }
      { for $b in doc("books.xml")//book
        where $b/publisher = $p
        order by $b/title
        return $b/title
      }
    </result>
  }
</listings>
```

Inverting Hierarchies (3)

Result:

```
<listings>
  <result>Addison-Wesley
    <title>Advanced Programming ...</title>
    <title>TCP/IP Illustrated</title>
  </result>
  <result>Kluwer Academic Publishers
    <title>The Economics of ...</title>
  </result>
  <result>Morgan Kaufmann Publishers
    <title>Data on the Web</title>
  </result>
</listings>
```

Quantifiers

- Some queries need to determine whether
  - at least one item in a sequence satisfies a condition
  - every item in sequence satisfies a condition
- This is done using quantifiers:
  - `some` is an existential quantifier
  - `every` is a universal quantifier

Quantifiers (2)

- The following query shows an existential quantifier
- We are looking for a book where *at least one* of the authors has the last name ‘Buneman’:

```
for $b in doc("books.xml")//book
where some $a in $b/author
      satisfies ($a/last = 'Buneman')
return $b/title
```

which returns:

```
<title>Data on the Web</title>
```

Quantifiers (3)

- The following query shows a universal quantifier
- We are looking for a book where *all* of the authors have the last name ‘Stevens’:

```
for $b in doc("books.xml")//book
where every $a in $b/author
      satisfies ($a/last = 'Stevens')
return $b/title
```

which returns:

```
<title>TCP/IP Illustrated</title>
<title>Advanced Programming ...</title>
<title>The Economics of Technology ...</title>
```

Quantifiers (4)

- A universal quantifier applied to an empty sequence always yields true (there is no item violating the condition)
- An existential quantifier applied to an empty sequence always yields false (there is no item satisfying the condition)

Conditional Expressions

- XQuery's conditional expressions (`if - then - else`) are used in the same way as in other languages
- In XQuery, both the `then` and the `else` clause are required
- The empty sequence `()` can be used to specify that a clause should return nothing
- The following query returns all authors for books with up to two authors and "et al." for any remaining authors

Conditional Expressions (2)

```
for $b in doc("books.xml")//book
return
  <book> { $b/title } {
    for $a at $i in $b/author
    where $i <= 2
    return <author> { string($a/last), ", ",
                      string($a/first) }
           </author>
    }
    { if (count($b/author) > 2)
      then <author> et al. </author>
      else ()
    }
  </book>
```

Conditional Expressions (3)

Result:

```
<book>
    <title>TCP/IP Illustrated</title>
    <author>Stevens, W.</author>
</book>
 <book>
    <title>Advanced Programming in ...</title>
    <author>Stevens, W.</author>
</book>
 <book>
    <title>Data on the Web</title>
    <author>Abiteboul, Serge</author>
    <author>Buneman, Peter</author>
    <author>et al. </author>
</book>
 <book>
    <title>The Economics of Technology ...</title>
</book>
```

Summary

- XQuery was designed to be compact and compositional
- It is well-suited to XML-processing tasks like data integration and data transformation

Chapter 10

Mapping XML to the Relational World

Introduction

- XQuery and other XML query languages operate on XML documents
- Up to now we have assumed that these documents exist in files or network messages
- Often, however, documents are generated on demand from different representations and sources
- One important source of data are relational database management systems (RDBMS)

Introduction (2)

- RDBMS are not going to vanish due to the arrival of the new XML standards
- Quite the contrary, RDBMS are probably going to stay with us for a long time to come
- Building bridges between the XML and the RDBMS world is therefore very important
- In this chapter we are going to have a look at different approaches for mappings between XML and relational data
- SQL/XML is an important ISO standard that addresses these issues

XML Publishing

- Assume that the original data is relational
- The application, however, wants to access this data as XML
- So we have to create an XML representation of the relational data
- This is called XML *publishing* or *composing*

XML Shredding

- The original data may instead be in XML
- The question now is how to store this data in a RDBMS
- The simplest method is to store the XML directly as the value of some attribute/column in a relation
- More generally, this process is called XML *shredding* or *decomposing*
- Shredding can be done in many ways, depending on
  - how structured the data is: ranging from very structured to quite unstructured marked-up text
  - what kind of schema information is available

SQL/XML

- The ISO SQL/XML standard was first produced in 2003
- It was revised in 2006, 2008 and 2011
- It provides a new SQL data type (XML) to store XML in an RDBMS
- SQL/XML provides new SQL functions to generate XML documents or fragments from relational data (called publishing functions)
- In addition to this, there are default mapping rules for SQL datatypes appearing in XML-generating operators
- It also provides additional querying capabilities (using XQuery)

Using the XML Data Type

- The simplest way of storing XML in an RDBMS is to use the SQL/XML XML data type
- A column of type XML in the RDBMS can contain any XQuery sequence
- Some other columns may also be present
- Example (the `purchaseorder` column is of type XML):

```
id    receivedate  purchaseorder

4023  2001-12-01   <purchaseOrder>
                     <originator billId='0013579'>
                       <contactName>
                       ...
                   </purchaseOrder>

5327  2002-04-23   <purchaseOrder>
                     <originator billId='0232345'>
                       ...
```

Using the XML Data Type (2)

- The single column mapping is quite straightforward; the XML document (or sequence) is loaded into the RDBMS “as is”
- A value of type XML can be any valid XQuery sequence or the SQL NULL value
- In fact, a number of parameterised subtypes of the XML type are defined in the standard:
  - XML (SEQUENCE)
  - XML (ANY CONTENT)
  - XML (ANY DOCUMENT)
  - ...
- We will not study these subtypes

Publishing Techniques

- SQL/XML provides two different techniques for publishing relational data as XML
  - A default mapping from tables to XML
  - Using the SQL/XML publishing functions
- The first of these is very simple, but limited in how useful it is
- The second is much more flexible

Default Mapping

- The default mapping is the simplest publishing technique
- In the default mapping, the names of tables and columns become the names of XML elements, with the inclusion of `row` elements for the each table row
- But the default mapping does not allow for publishing only parts of tables or the result of a query as XML
- Also, many applications may need XML data in specific formats that do not correspond to the result of the default mapping
- These limitations mean that applications may have to perform extensive post-processing on the generated document

Example

Table customer:

```
name            acctnum   address

Albert Ng       012ab3f   123 Main St., ...
Francis Smith   032cf5d   42 Seneca, ...
...             ...       ...
```

XML generated by the default mapping:

```
<customer>
  <row>
    <name>Albert Ng</name>
    <acctnum>012ab3f</acctnum>
    <address>123 Main St., ...</address>
  </row>
  <row>
    <name>Francis Smith</name>
    <acctnum>032cf5d</acctnum>
    <address>42 Seneca, ...</address>
  </row>
  ...
</customer>
```

Default Mapping (2)

- The default mapping can also be used for all tables in a schema, or all schemas in a catalog
- In each case, an extra level is introduced in the output by elements representing schema or catalog names
- The mapping depends on rules for mapping SQL identifiers to XML names, and SQL data types to XML schema data types
- As well as producing an XML document representing the relational data, the default mapping produces an XML schema document

SQL/XML functions for publishing

- `XMLELEMENT()` to produce an XML element
- `XMLATTRIBUTES()` to produce XML attributes
- `XMLFOREST()` which creates a forest of elements
- `XMLCONCAT()` which concatenates a list of XML elements
- `XMLAGG()` which creates a forest of XML elements based on a `GROUP BY` clause in the SQL query
- (We will consider only the first three functions)

Example using XMLELEMENT()

- This example assumes the `customer` table used previously:

```
SELECT c.acctnum,
  XMLELEMENT (NAME "invoice",
            'To ',
            XMLELEMENT (NAME "name", c.name)
           ) AS "invoice"
FROM customer c
```

- This creates an XML element called `invoice` with mixed content:

```
acctnum   invoice

012ab3f   <invoice>To <name>Albert Ng</name></invoice>
032cf5d   <invoice>To <name>Francis Smith</name></invoice>
...
```

Example using XMLATTRIBUTES()

- Once again using the `customer` table:

```
SELECT c.acctnum,
  XMLELEMENT (NAME "invoice",
              XMLATTRIBUTES (c.acctnum AS "id", c.name)
             ) AS "invoice"
FROM customer c
```

- This creates an XML element with attributes and empty content:

```
acctnum   invoice

012ab3f   <invoice id="012ab3f" name="Albert Ng"/>
032cf5d   <invoice id="032cf5d" name="Francis Smith"/>
...
```

- Obviously attributes and nested elements can be combined

XMLFOREST()

- `XMLFOREST()` produces a forest of elements
- Each of its arguments is used to create a new element
- Like `XMLATTRIBUTES()`, an explicit name for the element can be provided, or the name of the column can be used implicitly

Shredding

- There are different ways of shredding XML documents
- If the documents are well-structured and follow a DTD or XML schema:
  - We can extract this schema information and build a relational schema that mirrors this structure
  - Each table in this relational schema stores certain parts of the XML document
- If the documents are irregular and do not follow a common schema:
  - We have to use a very general schema for mapping arbitrary XML trees into an RDBMS

Shredding Unstructured Documents

- One possibility to handle arbitrary documents is to use a relational representation that is totally independent of XML schema information
- This representation models XML documents as tree structures with nodes and edges
- We saw an example of this in Chapter 8 with the Edge relation
- Every single navigation step requires a join on this table
- Alternatives considered in Chapter 8 were
  - Element-partitioned relations
  - Path-partitioned relations

Shredding Structured Documents

- The first step is designing the relational schema
- Some database vendors offer an automated mapping process
- These techniques are often based on annotating an XML schema definition with information about where the corresponding data is to be stored in the RDBMS
- We are going to have a look at some basic techniques for creating a relational schema

Shredding Structured Documents (2)

- Adding extra information:
  - Care has to be taken that we will be able to reassemble the XML document (sometimes more than one document is stored in a table)
  - Usually each node/value stored in a table will have a document id associated with it (regardless of in which table it will end up)
  - Storing positions of a node within its parent will allow us to reconstruct the document order

Shredding Structured Documents (3)

- During shredding we have two basic table layout choices:
  - We can break information across multiple tables
  - We can consolidate tables for different elements
- A simple algorithm for doing this starts scanning at the top of the XML document
- Each time an element is encountered it is associated with a table
- For each child of that element a decision is made whether
  - to put it into the same table (inlining)
  - or start a new table (and find a way to connect the two tables via a join attribute)

Shredding Structured Documents (4)

- There is a simple rule for deciding whether to inline or not:
  - If an element can occur multiple times (e.g. has `maxOccurs > 1`), then put it in a different table
  - If an element has a complex structure (e.g. is of `ComplexType`), then put it in a different table
  - Simple elements (e.g. of `SimpleType`) that occur exactly once are placed in the same table as their parent element
- What about optional elements?
  - Inlining optional elements may lead to many NULL values
  - Putting them into their own table results in expensive join operations
  - Neither choice is optimal in all cases

Example

- Consider our `books.xml` example from Chapter 9
- Since `year`, `title`, `publisher` and `price` each occur once, they can be placed in the same `book` table
- Since `author` can occur many times, it is placed in a different table
- Since `editor` is complex, it is placed in a different table
- The next slide shows the result

Example (2)

book				
id	year	title	publisher	price
1	1994	TCP/IP ...	...	65.95
2	1992	Advanced ...	...	65.95
3	2000	Data on ...	...	39.95
4	1999	The Economics ...	...	129.95

author			
id	last	first	book
5	Stevens	W.	1
6	Stevens	W.	2
7	Abiteboul	Serge	3
8	Buneman	Peter	3
9	Suciu	Dan	3

editor				
id	last	first	affiliation	book
10	Gerbarg	Darcy	CITI	4

Shredding Structured Documents (5)

- After shredding XML documents, it may be possible to consolidate tables
- Some element types may appear multiple times in an XML document at different places (e.g. names or addresses)
- As long as the attributes are used in a consistent way, these different tables can be merged into one
- Shredding, in general, is a complicated process and there are many possible solutions

Conclusion

- The SQL/XML XML data type can handle any kind of XML data
- For the shredding approach some kind of XML schema information is helpful
- It is quite expensive for the shredding approach to reassemble whole documents

Summary

- There are a variety of techniques for mapping between XML and relational data
- Facilities for achieving this mapping are provided by database vendors or third party vendors (e.g. for middleware components)
- Which actual features are necessary depends mostly on the requirements of the application