ADT 2010

Introduction to

XML, XPath (& XQuery)

Chapter 10 in

Silberschatz, Korth, Sudarshan

“Database System Concepts”

Stefan Manegold
Stefan.Manegold@cwi.nl
http://www.cwi.nl/~manegold/

Google Hits

of 3-letter combinations

XML	2100 Mio.
sex	711 Mio.
ABC	316 Mio.
SQL	296 Mio.
UvA	28 Mio.
CWI	9.4 Mio.

XML: Document Markup vs. High Data Volume

- Originally, the term **markup** has been coined by the **typesetting** community, not by computer scientists.
- With the advent of the printing press, writers and editors used (often marginal) notes to instruct printers to
  - ▷ select certain fonts,
  - ▷ let passages of text stand out,
  - ▷ indent a line of text, *etc.*
- Proofreaders use a special set of symbols, their special **markup language**, to identify typos, formatting glitches, and similar issues in the text.

Source Code Markup

- Computer scientists adopted the markup idea—originally to **annotate program source code:**
  - ▷ Design the markup language such that its constructs are **easily recognizable by a machine.**
  - ▷ **Approaches:**
    1. Markup is written using a **special set of characters,** disjoint from the set of characters that form the tokens of the program (cf. the `ESC` character).
    2. Markup occurs in places in the source file where program code may *not* appear (**program layout**).

Source Code Markup

- Increased computing power and more sophisticated parsing technology made fixed form source obsolete.
- Markup, however, is still being used on different levels in today's programming languages and systems:
  - ASCII defines a set of **non-printable characters** (the C0 control characters, code range 0x00–0x1f):

code	name	description
0x01	STX	start of heading
0x02	SOT	start of text
0x0d	CR	carriage return
0x1b	ESC	escape

Source Code Markup

- Markup in current programming languages:
  - **Blocks** defined by various forms of matching **delimiters:**
    - `begin···end`, `\begin{itemize}···\end{itemize}`
    - `/*···*/`, `(:···:)`, `//···⟨LF⟩`
    - `do···done`, `if···fi`, `case···esac`, `$[···]`

HTML-Style Presentational Markup

- HTML (W3C `http://www.w3.org/MarkUp/`) defines a number of markup **tags**, some of which are required to match (···).
- Note that HTML tags primarily define **presentational markup** (heading level, font weight, . . .).

XML Introduction

- XML: Extensible Markup Language
- Defined by the WWW Consortium (W3C): `http://www.w3c.org/XML/`
- Derived from SGML (Standard Generalized Markup Language), but simpler to use than SGML
- Documents have tags giving extra information about sections of the document
  - E.g. <title> XML </title> <slide> Introduction …</slide>
- **Extensible**, unlike HTML
  - Users can add new tags, and *separately* specify how the tag should be handled for display

XML Introduction (Cont.)

- The ability to specify new tags, and to create nested tag structures make XML a great way to exchange **data**, not just documents.
  - Much of the use of XML has been in data exchange applications, not as a replacement for HTML
- Tags make data (relatively) self-documenting
  - E.g.

```
<bank>
    <account>
        <account_number> A-101     </account_number>
        <branch_name>     Downtown </branch_name>
        <balance>           500        </balance>
    </account>
    <depositor>
        <account_number> A-101    </account_number>
        <customer_name> Johnson </customer_name>
    </depositor>
</bank>
```

A Little Bit Of History

- Database world
  - 1980 relational databases
  - 1990 nested relational model and object oriented databases
  - 2000 semi-structured databases
- Documents world
  - 1974 SGML (Structured Generalized Markup Language)
  - 1990 HTML (Hypertext Markup Language)
  - 1992 URL (Universal Resource Locator)

Data + documents = information

1996 **XML** (Extended Markup Language)

URI (Universal Resource Identifier)

XML Introduction (Cont.)

XML was designed to satisfy the needs of two worlds:

- **document-centric** applications (e. g. Information Retrieval)
  - Information with little structure (e. g. text documents with chapters, sections, . . .)
- **data-centric** applications ("traditional" database applications)
  - Very regular data; sometimes, however, we might want some flexibility (e. g. persons without a phone number, or with even two, . . .)

Some additional demands influenced the design of XML:

- XML was designed to be simple, generic and extensible.
- XML is an SGML dialect, and similar to HTML.

XML: Motivation

- Data interchange is critical in today's networked world
  - Examples:
    - Banking: funds transfer
    - Order processing (especially inter-company orders)
    - Scientific data
      - Chemistry: ChemML, …
      - Genetics: BSML (Bio-Sequence Markup Language), …
  - Paper flow of information between organizations is being replaced by electronic flow of information
- Each application area has its own set of standards for representing information
- XML has become the basis for all new generation data interchange formats

XML Motivation (Cont.)

- Earlier generation formats were based on plain text with line headers indicating the meaning of fields
  - Similar in concept to email headers
  - Does not allow for nested structures, no standard “type” language
  - Tied too closely to low level document structure (lines, spaces, etc)
- Each XML based standard defines what are valid elements, using
  - XML type specification languages to specify the syntax
    - DTD (Document Type Descriptors)
    - XML Schema
  - Plus textual descriptions of the semantics
- XML allows new tags to be defined as required
  - However, this may be constrained by DTDs
- A wide variety of tools is available for parsing, browsing and querying XML documents/data

Comparison with Relational Data

- Inefficient: tags, which in effect represent schema information, are repeated
- Better than relational tuples as a data-exchange format
  - Unlike relational tuples, XML data is self-documenting due to presence of tags
  - Non-rigid format: tags can be added
  - Allows nested structures
  - Wide acceptance, not only in database systems, but also in browsers, tools, and applications

Structure of XML Data

- **Tag**: label for a section of data
- **Element**: section of data beginning with <*tagname*> and ending with matching </*tagname*>
- Elements must be properly nested
  - Proper nesting
    - <account> … <balance> …. </balance> </account>
  - Improper nesting
    - <account> … <balance> …. </account> </balance>
  - Formally: every start tag must have a unique matching end tag, that is in the context of the same parent element.
- Every document must have a single top-level element

Example of Nested Elements

```
<bank-1>
    <customer>
        <customer_name> Hayes </customer_name>
        <customer_street> Main </customer_street>
        <customer_city>    Harrison </customer_city>
        <account>
            <account_number> A-102 </account_number>
            <branch_name>     Perryridge </branch_name>
            <balance>           400 </balance>
        </account>
        <account>
            …
        </account>
    </customer>
      .
      .
</bank-1>
```

Motivation for Nesting

- Nesting of data is useful in data transfer
  - Example: elements representing *customer_id, customer_name*, and address nested within an *order* element
- Nesting is not supported, or discouraged, in relational databases
  - With multiple orders, customer name and address are stored redundantly
  - normalization replaces nested structures in each order by foreign key into table storing customer name and address information
  - Nesting is supported in object-relational databases
- But nesting is appropriate when transferring data
  - External application does not have direct access to data referenced by a foreign key

Structure of XML Data (Cont.)

- Mixture of text with sub-elements is legal in XML.
  - Example:

```
<account>
    This account is seldom used any more.
    <account_number> A-102</account_number>
    <branch_name> Perryridge</branch_name>
    <balance>400 </balance>
</account>
```

  - Useful for document markup, but discouraged for data representation

Attributes

- Elements can have **attributes**

```
<account acct-type = “checking” >
    <account_number> A-102 </account_number>
    <branch_name> Perryridge </branch_name>
    <balance> 400 </balance>
</account>
```

- Attributes are specified by *name=value* pairs inside the starting tag of an element
- An element may have several attributes, but each attribute name can only occur once

```
<account  acct-type = “checking”  monthly-fee=“5”>
```

Attributes vs. Subelements

- Distinction between **subelement** and **attribute**
  - In the context of documents, attributes are part of markup, while subelement contents are part of the basic document contents
  - In the context of data representation, the difference is unclear and may be confusing
    - Same information can be represented in two ways
      - <account account_number = “A-101”> …. </account>
      - <account>
        <account_number>A-101</account_number> …
        </account>
  - Suggestion: use attributes for identifiers of elements, and use subelements for contents
  - Attributes can be used to *qualify* tags

    => avoid the so-called *tag soup*

Data on the Web

- With sites slowly migrating to the XML dialect XHTML (W3C `http://www.w3.org/TR/xhtml1/`), the Web turns into the largest database on this planet.
  - In XPath and XQuery, the `doc(·)` function can access XML documents at any URI:

`doc(·)` accesses XML documents on the Web

```
doc("http://slashdot.org/rss/index.rss")
```

  - XHTML takes over, lookout for

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"...
```

The (Future) Web: A Huge XML Database

Result of `doc("http://slashdot.org/rss/index.rss")`

```
<rdf:RDF xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#">
  <channel rdf:about="http://slashdot.org/">
    <title>Slashdot</title>
    <link>http://slashdot.org/</link>
    <description>News for nerds, stuff that matters</description>
    ...
      <item rdf:about="http://slashdot.org/article.pl?sid=...">
        <title>Yahoo and Microsoft to Merge Instant Messengers</title>
        <link>http://rss.slashdot.org/Slashdot/slashdot?m=1131</link>
        <description> ... the two tech giants ...</description>
        ...
      </item>
    ...
</rdf:RDF>
```

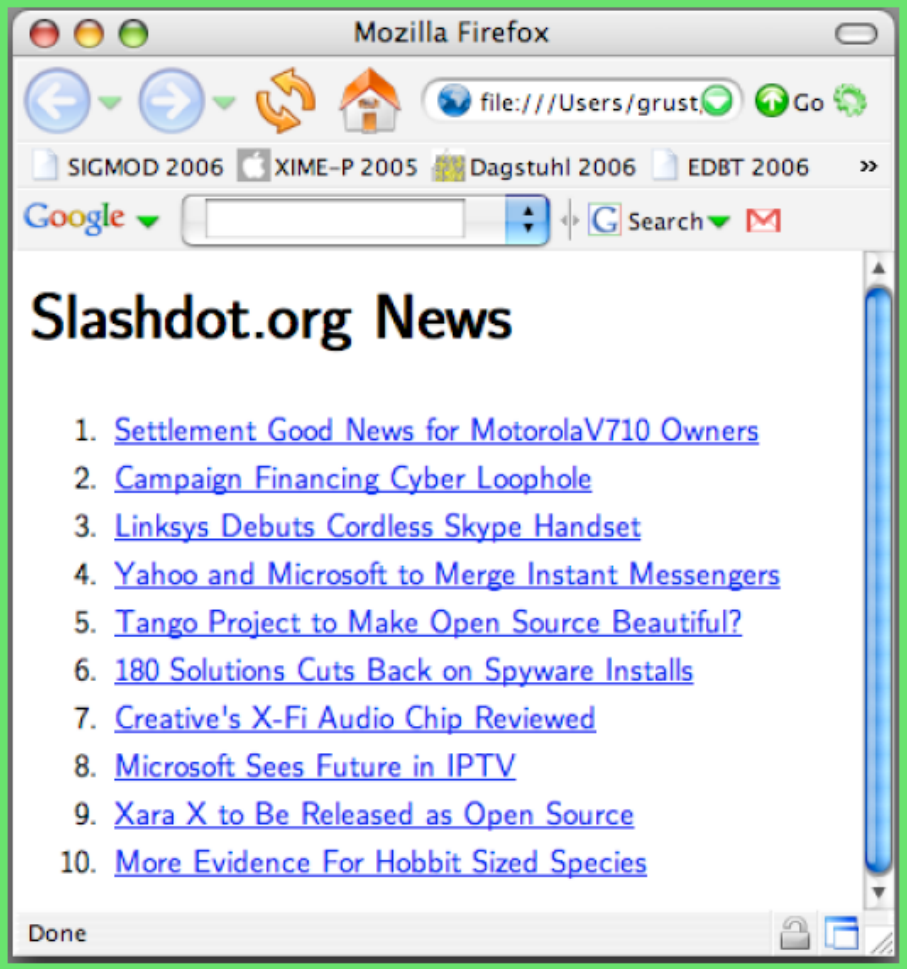
The (Future) Web: A Huge XML Database

XQuery: Turn Slashdot's RSS feed into XHTML Page

```
<html>
  <body>
    <h1>Slashdot.org News</h1>
    <ol>
      {
        for $t in doc("http://slashdot.org/rss/index.rss")//*:item
        return
          <li>
            <a> {
              attribute href { $t/*:link/text() },
              text { $t/*:title } }
            </a>
          </li>
      }
    </ol>
  </body>
</html>
```

The (Future) Web: A Huge XML Database

Query Output Rendered in Web Browser



Namespaces

- XML data has to be exchanged between organizations
- Same tag name may have different meaning in different organizations, causing confusion on exchanged documents
- Specifying a unique string as an element name avoids confusion
- Better solution: use unique-name:element-name
- Avoid using long unique names all over document by using XML Namespaces

```
<bank Xmlns:FB='http://www.FirstBank.com'>
    ...
    <FB:branch>
        <FB:branchname>Downtown</FB:branchname>
        <FB:branchcity>  Brooklyn  </FB:branchcity>
    </FB:branch>
    ...
</bank>
```

More on XML Syntax

- Elements without subelements or text content can be abbreviated by ending the start tag with a /> and deleting the end tag
  - <account number="A-101" branch="Perryridge" balance="200 />
- To store string data that may contain tags, without the tags being interpreted as subelements, use CDATA as below
  - <![CDATA[<account> … </account>]]>

  Here, <account> and </account> are treated as just strings

  CDATA stands for "character data"
- A **comment** may appear wherever a tag is allowed:
  - <!-- This is a comment and ignored b the XML parser -->
- **Processing instructions** can be used to control specific XML parsers:
  - <?php sql ("SELECT * FROM ...") ... ?>

XML Document Schema

- Database schemas constrain what information can be stored, and the data types of stored values
- XML documents are not required to have an associated schema
  - We also speak of **semi-structured** data
- However, schemas are very important for XML data exchange
  - Otherwise, a site cannot automatically interpret data received from another site
- Two mechanisms for specifying XML schema
  - **Document Type Definition (DTD)**
    - Widely used
  - **XML Schema**
    - Newer, increasing use

Document Type Definition (DTD)

- The type of an XML document can be specified using a DTD
- DTD constraints structure of XML data
  - What elements can occur
  - What attributes can/must an element have
  - What subelements can/must occur inside each element, and how many times.
- DTD does not constrain data types
  - All values represented as strings in XML
- DTD syntax
  - <!ELEMENT element (subelements-specification) >
  - <!ATTLIST element (attributes) >

Element Specification in DTD

- Subelements can be specified as
  - names of elements, or
  - #PCDATA (parsed character data), i.e., character strings
  - EMPTY (no subelements) or ANY (anything can be a subelement)
- Example

  <! ELEMENT depositor (customer_name account_number)>
  <! ELEMENT customer_name (#PCDATA)>
  <! ELEMENT account_number (#PCDATA)>
- Subelement specification may have regular expressions

  <!ELEMENT bank ((account | customer | depositor)+)>
  - Notation:
    - “|” - alternatives
    - “+” - 1 or more occurrences
    - “*” - 0 or more occurrences

Bank DTD

```
<!DOCTYPE bank [
    <!ELEMENT bank ( ( account | customer | depositor)+)>
    <!ELEMENT account (account_number branch_name balance)>
    <! ELEMENT customer(customer_name customer_street customer_city)>
    <! ELEMENT depositor (customer_name account_number)>
    <! ELEMENT account_number (#PCDATA)>
    <! ELEMENT branch_name (#PCDATA)>
    <! ELEMENT balance(#PCDATA)>
    <! ELEMENT customer_name(#PCDATA)>
    <! ELEMENT customer_street(#PCDATA)>
    <! ELEMENT customer_city(#PCDATA)>
]>
```

Attribute Specification in DTD

- Attribute specification : for each attribute
  - Name
  - Type of attribute
    - CDATA
    - ID (identifier) or IDREF (ID reference) or IDREFS (multiple IDREFs)
      - more on this later
  - Whether
    - mandatory (#REQUIRED)
    - has a default value (value),
    - or neither (#IMPLIED)
- Examples
  - <!ATTLIST account acct-type CDATA “checking”>
  - <!ATTLIST customer
    customer_id ID # REQUIRED
    accounts IDREFS # REQUIRED >

IDs and IDREFs

- An element can have at most one attribute of type ID
- The ID attribute value of each element in an XML document must be distinct
  - Thus the ID attribute value is an object identifier
- An attribute of type IDREF must contain the ID value of an element in the same document
- An attribute of type IDREFS contains a set of (0 or more) ID values. Each ID value must contain the ID value of an element in the same document

Bank DTD with Attributes

- Bank DTD with ID and IDREF attribute types.

```
<!DOCTYPE bank-2[
    <!ELEMENT account (branch, balance)>
    <!ATTLIST account
            account_number ID       # REQUIRED
            owners          IDREFS # REQUIRED>
    <!ELEMENT customer(customer_name,
customer_street,
                                    customer_city)>
    <!ATTLIST customer
            customer_id     ID      # REQUIRED
            accounts        IDREFS # REQUIRED>
    … declarations for branch, balance, customer_name,
                        customer_street and customer_city
]>
```

XML data with ID and IDREF attributes

```
<bank-2>
    <account account_number="A-401" owners="C100 C102">
        <branch_name> Downtown </branch_name>
        <balance>        500 </balance>
    </account>
    <customer customer_id="C100" accounts="A-401">
        <customer_name>Joe        </customer_name>
        <customer_street> Monroe  </customer_street>
        <customer_city>    Madison</customer_city>
    </customer>
    <customer customer_id="C102" accounts="A-401 A-402">
        <customer_name> Mary     </customer_name>
        <customer_street> Erin       </customer_street>
        <customer_city>    Newark </customer_city>
    </customer>
</bank-2>
```

Limitations of DTDs

- No typing of text elements and attributes
  - All values are strings, no integers, reals, etc.
- Difficult to specify unordered sets of subelements
  - Order is usually irrelevant in databases (unlike in the document-layout environment from which XML evolved)
  - (A | B)* allows specification of an unordered set, but
    - Cannot ensure that each of A and B occurs only once
- IDs and IDREFs are untyped
  - The *owners* attribute of an account may contain a reference to another account, which is meaningless
    - *owners* attribute should ideally be constrained to refer to customer elements

XML Schema

- XML Schema is a more sophisticated schema language which addresses the drawbacks of DTDs. Supports
  - Typing of values
    - E.g. integer, string, etc
    - Also, constraints on min/max values
  - User-defined, comlex types
  - Many more features, including
    - uniqueness and foreign key constraints, inheritance
- XML Schema is itself specified in XML syntax, unlike DTDs
  - More-standard representation, but verbose
- XML Scheme is integrated with namespaces
- BUT: XML Schema is significantly more complicated than DTDs.

XML Schema Version of Bank DTD

```
<xs:schema xmlns:xs=http://www.w3.org/2001/XMLSchema>
<xs:element name="bank" type="BankType"/>
<xs:element name="account">
    <xs:complexType>
        <xs:sequence>
            <xs:element name="account_number" type="xs:string"/>
            <xs:element name="branch_name"    type="xs:string"/>
            <xs:element name="balance"        type="xs:decimal"/>
        </xs:squence>
    </xs:complexType>
</xs:element>
….. definitions of customer and depositor ….
<xs:complexType name="BankType">
    <xs:squence>
        <xs:element ref="account"   minOccurs="0" maxOccurs="unbounded"/>
        <xs:element ref="customer"  minOccurs="0" maxOccurs="unbounded"/>
        <xs:element ref="depositor" minOccurs="0" maxOccurs="unbounded"/>
     </xs:sequence>
</xs:complexType>
</xs:schema>
```

XML Schema Version of Bank DTD

- Choice of “xs:” was ours -- any other namespace prefix could be chosen
- Element “bank” has type “BankType”, which is defined separately
  - xs:complexType is used later to create the named complex type “BankType”
- Element “account” has its type defined in-line

More features of XML Schema

- Attributes specified by xs:attribute tag:
  - <xs:attribute name = “account_number”/>
  - adding the attribute use = “required” means value must be specified
- Key constraint: “account numbers form a key for account elements under the root bank element:

```
<xs:key name = “accountKey”>
      <xs:selector xpath = “]bank/account”/>
      <xs:field xpath = “account_number”/>
<\xs:key>
```

- Foreign key constraint from depositor to account:

```
<xs:keyref name = “depositorAccountKey” refer=“accountKey”>
      <xs:selector xpath = “]bank/account”/>
      <xs:field xpath = “account_number”/>
<\xs:keyref>
```

Querying and Transforming XML Data

- Translation of information from one XML schema to another
- Querying on XML data
- Above two are closely related, and handled by the same tools
- Standard XML querying/translation languages
  - XPath
    - Simple language consisting of path expressions
  - XSLT
    - Simple language designed for translation from XML to XML and XML to HTML
  - XQuery
    - An XML query language with a rich set of features

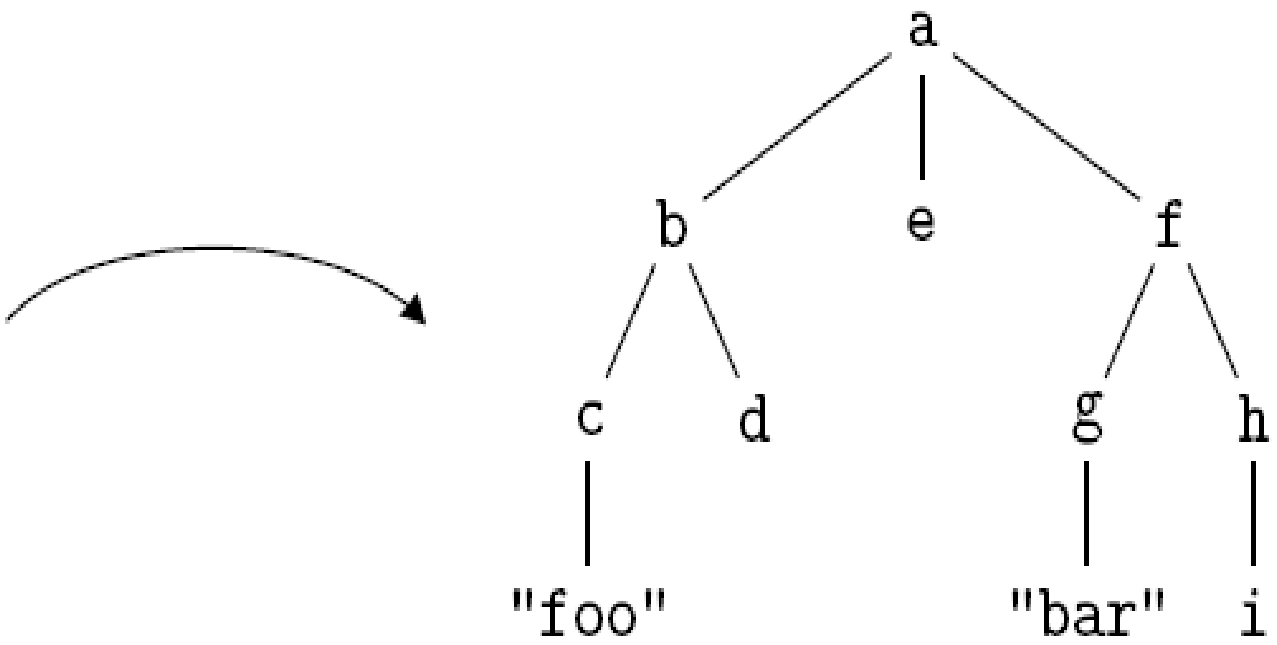
Tree Model of XML Data

- Query and transformation languages are based on a **tree model** of XML data
- An XML document is modeled as a tree, with **nodes** corresponding to elements and attributes
  - Element nodes have child nodes, which can be attributes or subelements
  - Text in an element is modeled as a text node child of the element
  - Children of a node are ordered according to their order in the XML document
  - Element and attribute nodes (except for the root node) have a single parent, which is an element node
  - The root node has a single child, which is the root element of the document

Tree Model of XML Data

- On the **physical** side, XML defines nothing more than a **flat file format**.
- The nesting of XML tags and attributes, however, defines a **logical tree structure**.

```
<a>
  <b>
    <c>foo</c>
    <d></d>
  </b>
  <e></e>
  <f>
    <g>bar</g>
    <h><i></i></h>
  </f>
</a>
```

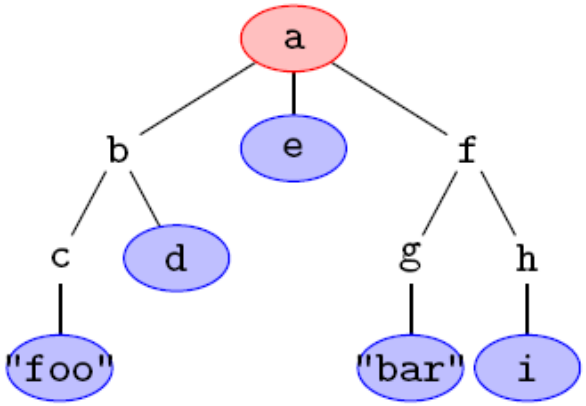


Tree Model of XML Data

- Given this tree structure, we use the term **nodes**:
  - Tags describe **element nodes** that may have other nodes as their children.
  - Plain text is contained in **text nodes**. Text nodes are always tree leaves.
  - Attributes are referred to as **attribute nodes**.
  - Analogously, we have **comment nodes** and **processing instruction nodes**.

  We refer to the specific "flavor" of a node as its **node kind**.

- The toplevel node is called the **root node**. Nodes that do not have any children are referred to as **leaf nodes**.



- **root node:** `a`
- **leaf nodes:** `"foo"`, `d`, `e`, `"bar"`, `i`
- Text, comment, and processing instruction nodes must always be leaf nodes.

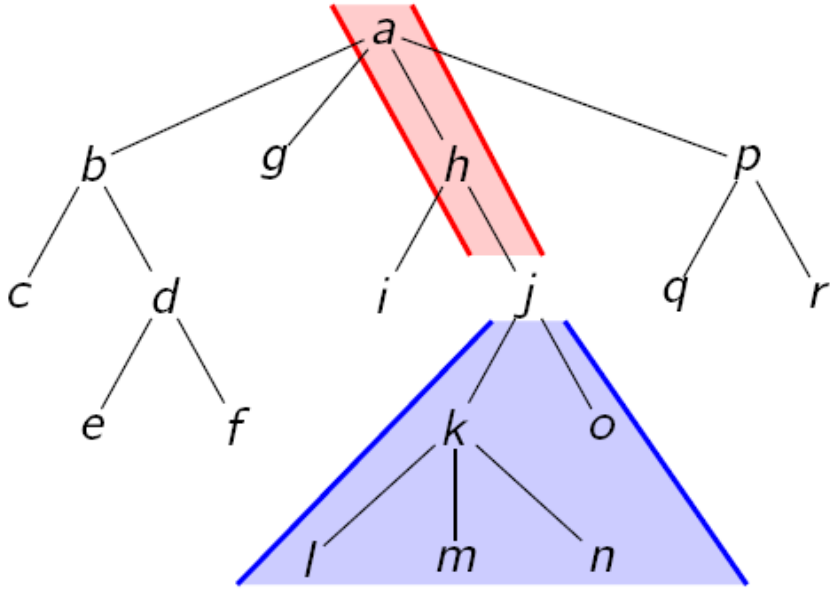
Relationships between nodes: Descendant/Ancestor

In addition to the common terms **child** and **parent**, the XML data model defines some more relationships between nodes in an XML tree:

- All nodes v' in the *subtree* below v form the **descendants** of node v: the children of v, its children's children, ...

  Note that v is not a descendant of itself.

- Complementary, if v' is a descendant of v, then v is called an **ancestor** of v'. (All nodes on the path from the root node to v' are ancestors of v', not including v'.)



descendant and **ancestor** nodes of j.

Relationships between nodes: Preceding/Following

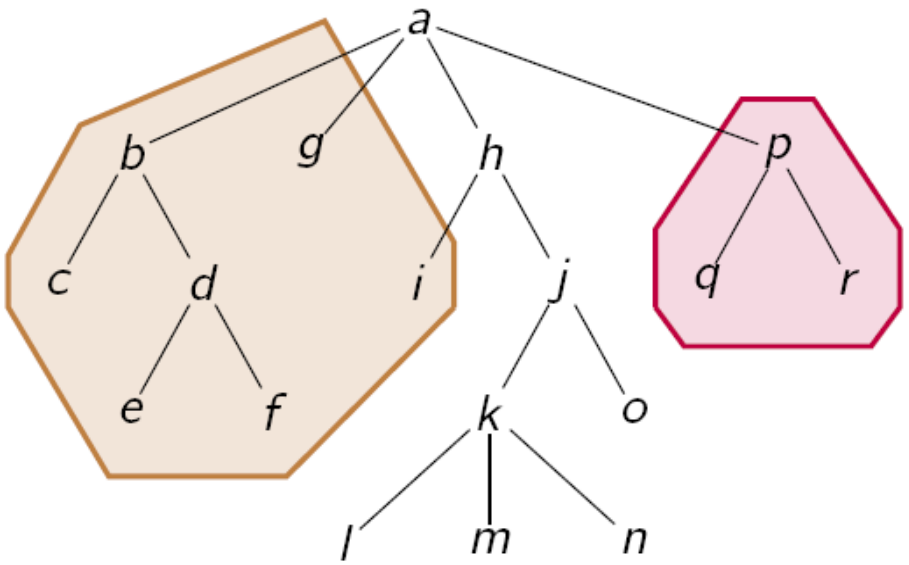
The serialized representation as an XML file allows us to define an **order** over the tree nodes.

- A node v' **precedes** v, if both, **start and end tag**, appear before **both tags** of v.

  In the tree representation, preceding nodes of v are those that are drawn "left" of v; ancestors or descendants of v are **not** precedings of v.

- Complementary, if v' precedes v, then v is said to **follow** v'.

- With the term **document order** we refer to the order that is defined by the **opening tags** of the nodes.



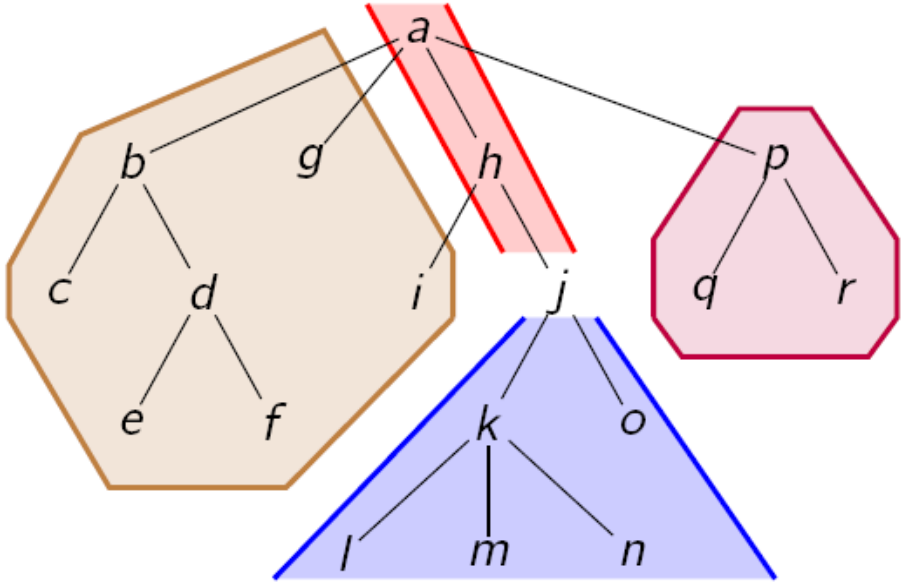
preceding and **following** nodes of j.

Document Partitioning

It is important to observe that with these relationships each node **partitions** the whole document into **four disjoint regions**.

Any two nodes v and v' ($v \neq v'$) must **either** be in

- **ancestor/descendant**, or in
- **preceding/following** relationship.



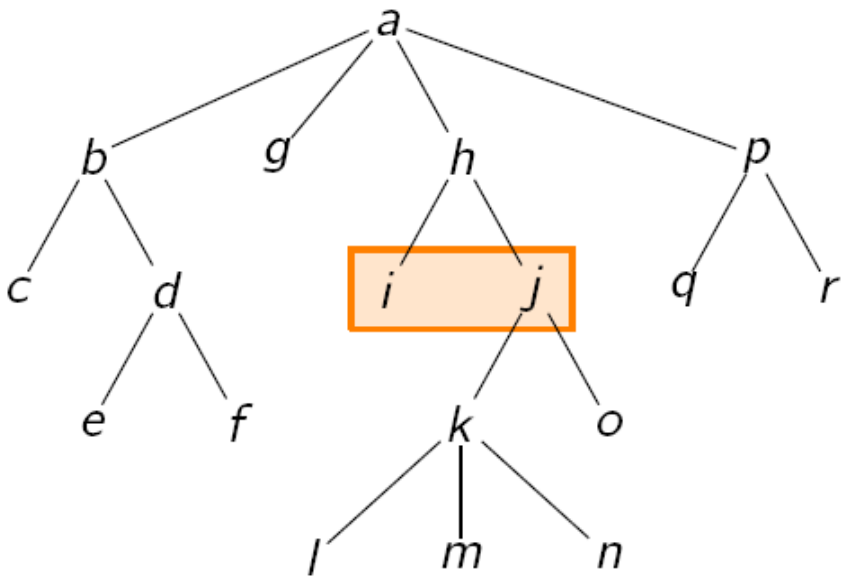
Another important observation is the **symmetry** in these four relationships:

ancestor $\leftrightarrow$ **descendant** and **preceding** $\leftrightarrow$ **following**

The Sibling Relationship

Nodes that share the same parent are **siblings** of each other.

- In combination with the node order we can define the
  - **preceding sibling** and
  - **following sibling** relationships.



Nodes *i* and *j* are **sibling** nodes.

XPath

- "Querying XML data" essentially means to **identify (or address) nodes**, and to **test** further properties on these nodes.

- Titles of all books published by Longstreet Press

```
$cat/catalog/book[publisher="Longstreet Press"]/title
```

```
<title>No Such Thing As A Bad Day</title>
```

- Publications with Don Chamberlin as author or editor

```
$cat//*[(author|editor) = "Don Chamberlin"]
```

```
<book><title>XQuery from the Experts</title>…</book>,
<spec><title>XQuery Formal Semantics</title>…</spec>
```

- Observe the **locate, then test** pattern in both queries.

- XPath has been standardized for this manner by the W3 Consortium in 1999.

XPath: Path Expressions

- The **path expression** is the core construct of **XPath**, which in turn forms the backbone of other XML standards (e. g. XQuery, XSLT, XPointer, . . .).
- Each path consists of one or more **steps**, syntactically separated by /:

$$s_0/s_1/\ldots/s_n \quad .$$

- Each step acts like an operator that, given a **sequence of nodes** (the *context set*), evaluates to a **sequence of nodes**.
- The entire XPath expression will thus always return a **sequence of nodes**.
- XPath defines the result of each path expression to be
  - ▷ **duplicate free** and
  - ▷ **sorted in document order**.

XPath: Path Expressions

- Each step is built from two components, separated by ::

$$\alpha :: \nu$$

  - ▷ The **axis** α determines, based on tree relationships, a number of **reachable nodes**.
  - ▷ The **node test** ν successively **filters** these nodes by certain properties: node kind, tag name, ...

- A typical XPath expression thus looks like

$$\alpha_0 :: \nu_0 / \alpha_1 :: \nu_1 / \ldots / \alpha_n :: \nu_n \quad .$$

XPath Axes

Thirteen axes are defined by the XPath standard:

Axis α	**Nodes reachable from context node** c
`ancestor`	all nodes in the ancestor region of c
`ancestor-or-self`	like `ancestor`, plus c
`child`	children of c
`descendant`	all nodes in the descendant region of c
`descendant-or-self`	like `descendant`, plus c
`following`	all nodes in the following region of c
`following-sibling`	siblings of c in the following region of c
`parent`	parent of c
`preceding`	all nodes in the preceding region of c
`preceding-sibling`	siblings of c in the preceding region of c
`self`	c
`attribute`	attributes of c
`namespace`	namespace nodes of c

Absolute Path Expressions

Node test ν	**Nodes that pass the test**
`node()`	any node whatsoever (don't care)
tag name t	elements (α=`attribute`: attributes) named t
*	all elements (α=`attribute`: attributes)
`text()`	any text node
`comment()`	any comment node
`processing-instruction(`t`)`	any processing instruction of the form <?t...?>

- The `node()`, `text()`, `comment()` and `processing-instruction()` tests are sometimes called **kind tests**.
- Conversely, the tests by tag name and the wildcard test * are referenced as **name tests**.

Absolute Path Expressions

- One particular way to evaluate an XPath path expression is to start path traversals from the **document node**.
  - ▷ The **document node** is an virtual node that sits on top of each document, forming a new root.
- Such paths are called **absolute path expressions** and are prefixed by a /:

$$/\alpha_0::\nu_0/\alpha_1::\nu_1/\ldots/\alpha_n::\nu_n \quad .$$

XPath Expressions against JTR Document

```
/child::*

/child::JackTheRipper/child::scene/child::victim

/descendant::node()/child::suspect

/descendant::node()/attribute::???
```

there are no attributes (yet?)

```
/descendant::node()/child::scene/following-sibling::node()

/descendant::node()/child::timeline/child::*/following-sibling::node()

/descendant::node()/child::inspector/self::node()/parent::node()/child::*
```

XPath: Abbreviated Syntax

- Especially for ad-hoc queries, users tend to use the **abbreviated syntax** of XPath:

Abbreviated step	Expanded step
.	`self::node()`
..	`parent::node()`
t (tag name)	`child::`*t*
`@`*t*	`attribute::`*t*
//	`/descendant-or-self::node()/`

Example:

```
//bubbles/bubble/@speaker
```

≡

```
/descendant-or-self::node()/child::bubbles/child::bubble/attribute::speaker
```

XPath: Predicates

Any XPath step s may be further qualified with a **predicate**:

$$s[q]$$

where the predicate q may be one of

- $q = n$ (with $n \geq 1$ a **numeric expression**)
  Select the nth node in the sequence returned by s (if present).

- $q = e$ (**boolean expression**)
  Return each node c as returned by s, if the evaluation of e with context node c evaluates to true.

- $q = p$ (**path expression**)
  Return each node c as returned by s for which evaluation of path p with context node c yields at least one node.

XPath: Predicates

Predicates have high precedence (priority)

In the XPath expression $/s_0/s_1[q]$, predicate q is applied to step s_1, **not** to the whole path expression.

$$/s_0/s_1[q] \neq (/s_0/s_1)[q]$$

For some axes, the index of a numeric predicate is counted backwards.

Some axes return nodes that are **before** the context node in document order. These axes are thus called **reverse axes**. For such axes, we count the nodes backwards.

XPath: Boolean Expressions

Boolean expressions can be built from constants (`42`, `"Dilbert"`, . . .), path expressions, as well as **comparison** and **boolean operators**:

`=` `<` `<=` `and` `or` `not(·)`

```
//*[./@id]

//*[@time]

//*[name and picture]

//*[not(*)]

//*[doctor/text() = “Watson” or  inspector/text() = “Holmes”]
```

Functions in XPath

- XPath provides several functions
  - The function count() at the end of a path counts the number of elements in the set generated by the path
    - E.g. /bank-2/account[count(./customer) > 2]
      - Returns accounts with > 2 customers
  - Also function for testing position (1, 2, ..) of node w.r.t. siblings
- Boolean connectives and and or and function not() can be used in predicates
- IDREFs can be referenced using function id()
  - id() can also be applied to sets of references such as IDREFS and even to strings containing multiple references separated by blanks
  - E.g. /bank-2/account/id(@owner)
    - returns all customers referred to from the owners attribute of account elements.