

Advanced databases

11

XML Data Management

XML

- e**X**tensible **M**arkup **L**anguage
- Data representation format proposed by W3C (WWW Consortium) for Web documents, such as:
 - books,
 - product catalogs,
 - order forms,
 - messages

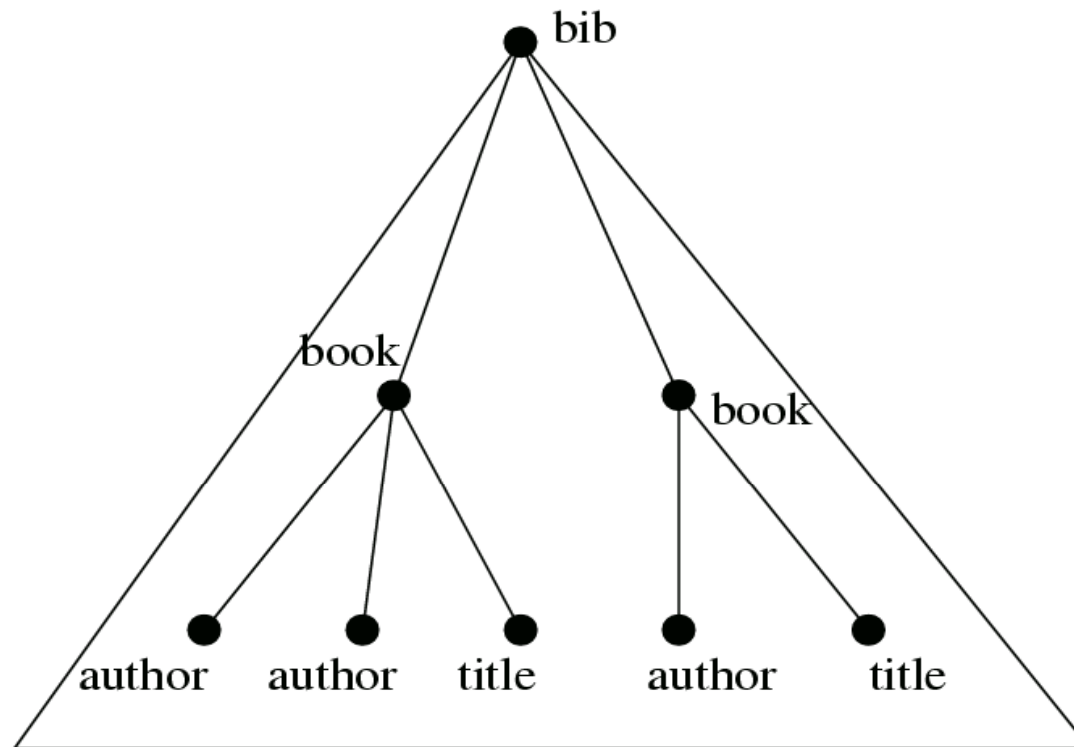
The Origin of XML

- **Original idea:** a meta-language used to specify markup languages
- **As in HTML** (or in WML, ...)
 - XML data are documents
 - data properties are expressed with mark-ups
- **XML was designed to describe data and to focus on what data is.**
- **HTML was designed to display data and to focus on how data looks.**

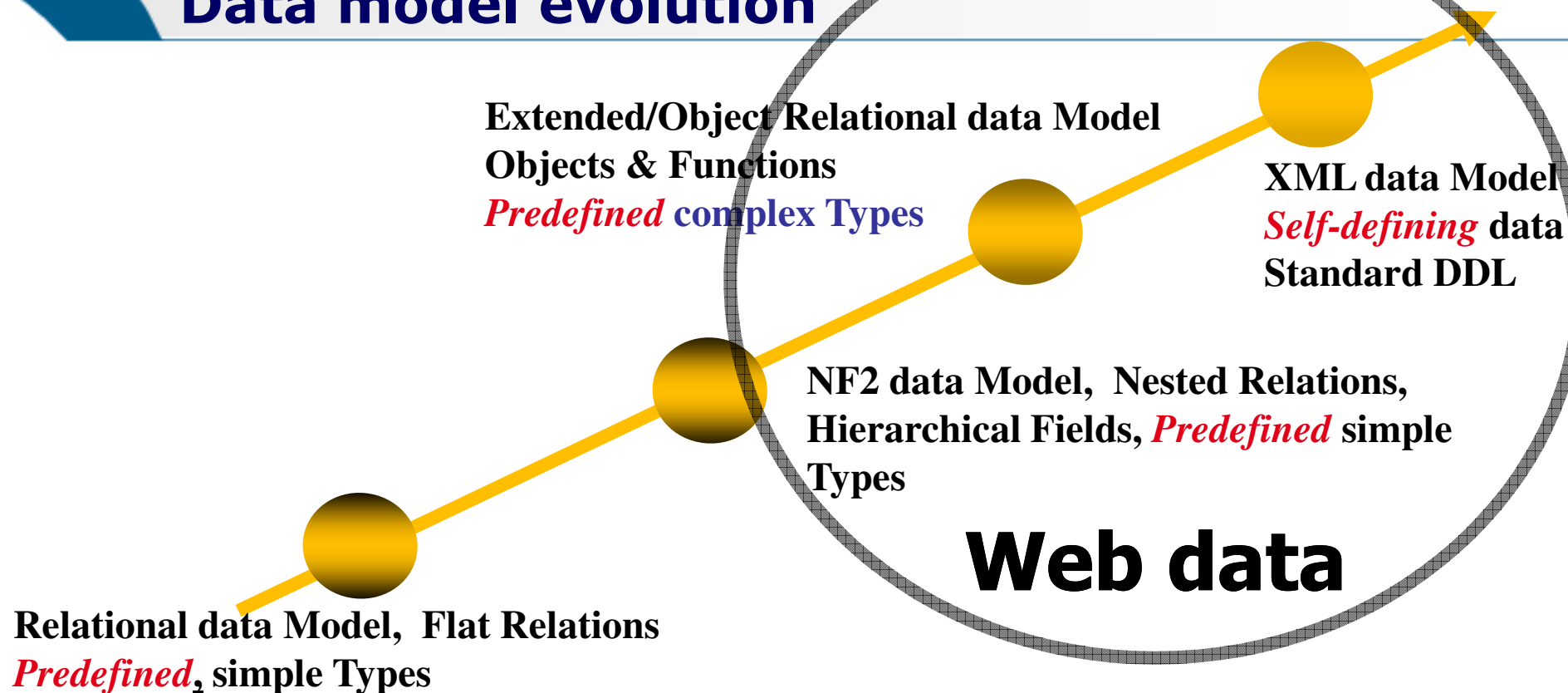
XML

- **1986: Standard Generalized Markup Language (SGML) ISO 8879-1986**
- **Nov. 1995: HTML 2.0**
- **Gen. 1997: HTML 3.2**
- **Aug. 1997: XML W3C Working Draft**
- **Feb 10, 1998: XML 1.0 Recommendation**
- **Dec. 13, 2001: XML 1.1 W3C Working Draft**
- **Oct. 15, 2002 : XML 1.1 W3C Candidate Recommendation**
- **Aug 16, 2006 Extensible Markup Language (XML) 1.0 (Fourth Edition) W3C Recommendation**

XML data model



Data model evolution



XML separates data from representation

- **With XML, your data is stored outside your HTML.**
- When HTML is used to display data, the data is stored inside your HTML. With XML, data can be stored in separate XML files. This way you can concentrate on using HTML for data layout and display, and be sure that changes in the underlying data will not require any changes to your HTML.
- XML data can also be stored inside HTML pages as "data Islands". You can still concentrate on using HTML only for formatting and displaying the data.

HTML vs XML

```
<h1>The Idea
  Methodology</h1><br>
<ul>
  <li>by S. Ceri,
    P. Fraternali  </li>
  <li> Addison-Wesley</li>
  <li> US$ 49  </li>
</ul>
```

```
<bib>
  <book>
    <title>The Idea
      Methodology  </title>
    <author> S. Ceri </author>
    <author> P. Fraternali
      </author>
    <pub>Addison-Wesley</pub>
    <price> US$ 49  </price>
  </book>
</bib>
```


XML is used to exchange data

- **With XML, data can be exchanged between incompatible systems.**
- In the real world, computer systems and databases contain data in incompatible formats. One of the most time-consuming challenges for developers has been to exchange data between such systems over the Internet.
- Converting the data to XML can greatly reduce this complexity and create data that can be read by many different types of applications.
- XML is the main language for exchanging financial information between businesses over the Internet.

XML is used to share data

- **With XML, plain text files can be used to share data.**
- Since XML data is stored in plain text format, XML provides a software- and hardware-independent way of sharing data.
- This makes it much easier to create data that different applications can work with. It also makes it easier to expand or upgrade a system to new operating systems, servers, applications, and new browsers.

XML is used to store data

- **With XML, plain text files can be used to store data.**
- XML data are also stored in files or in databases. Applications can be written to store and retrieve information from the store, and generic applications can be used to display the data.
- Data management extensions include data models (DTD,XSD), query languages (Xquery, XSLT)
- Data management occurs
 - Within native systems (eXists, Galax,...)
 - Within relational systems (Oracle, DB2, SQLServer)

XML can make your data more useful

- **With XML, your data is available to more users.**
- Since XML is independent of hardware, software and application, you can make your data available to other than only standard HTML browsers
- Other clients and applications can access your XML files as data sources, like they are accessing databases. Your data can be made available to all kinds of "reading machines" (agents)
- XML is the mother of new special-purpose languages.
 - E.g. the Wireless Markup Language (WML), used to markup Internet applications for handheld devices like mobile phones, is written in XML

Syntax

- The syntax rules of XML are very simple and very strict. The rules are very easy to learn, and very easy to use.
- Because of this, creating software that can read and manipulate XML is very easy.
- XML documents use a self-describing and simple syntax.

First Example (1)

- An example XML document:

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<note>
  <to>Tove</to>
  <from>Jani</from>
  <heading>Reminder</heading>
  <body>Don't forget me this weekend!</body>
</note>
```

First Example (2)

- The first line in the document - the XML declaration - defines the XML version and the character encoding used in the document. In this case the document conforms to the 1.0 specification of XML and uses the ISO-8859-1 (Latin-1/West European) character set.
- The next line describes the root element of the document (like it was saying: "this document is a note"):
- `<note>`

First Example (3)

- The next 4 lines describe 4 child elements of the root (to, from, heading, and body):
 <to>Tove</to>
 <from>Jani</from>
 <heading>Reminder</heading>
 <body>Don't forget me this weekend!</body>
- And finally the last line defines the end of the root element:
 </note>
- Can you detect from this example that the XML document contains a Note to Tove from Jani? Don't you agree that XML is pretty self-descriptive?

XML Syntax (1)

- **All XML elements must have a closing tag (with XML, it is illegal to omit the closing tag).**
 - **Note:** You might have noticed from the previous example that the XML declaration did not have a closing tag. This is not an error. The declaration is not a part of the XML document itself. It is not an XML element, and it should not have a closing tag.
- **XML tags are case sensitive (unlike HTML, XML tags are case sensitive).**
 - The tag <Letter> is different from the tag <letter>.
 - Opening and closing tags must therefore be written with the same case:
`<Message>This is incorrect</message>`
`<message>This is correct</message>`
- The syntax for writing comments in XML is similar to that of HTML.
`<!-- This is a comment -->`

XML Syntax (2)

- **All XML elements must be properly nested**
- **All XML documents must have a root element**
- **All XML documents must contain a single tag pair to define a root element.**
 - All other elements must be within this root element.
 - All elements can have sub elements (child elements). Sub elements must be correctly nested within their parent element:

```
<root> <child> <subchild>.....</subchild> </child> </root>
```
- **Attribute values must always be quoted (it is illegal to omit quotation marks around attribute values).**

```
<incorrectNote date=12/11/2002>  
<note date="12/11/2002">
```

Elements

- **Elements can have different content types**
- An **XML element** is everything from (including) the element's start tag to (including) the element's end tag.
- An element can have **element** content, **mixed** content, **simple** content, or **empty** content. An element can also have **attributes**.

Second Example (1)

```
<book>
  <title>My First XML</title>
  <prod id="33-657" media="paper"></prod>
  <chapter>Introduction to XML
    <para>What is HTML</para>
    <para>What is XML</para>
  </chapter>
  <chapter>XML Syntax
    <para>Elements must have a closing tag</para>
    <para>Elements must be properly nested</para>
  </chapter>
</book>
```

Second Example (2)

- Book is the **root element**. Title, prod, and chapter are **child elements** of book. Book is the **parent element** of title, prod, and chapter. Title, prod, and chapter are **siblings** because they have the same parent.
- Book has **element content**, because it contains other elements. Chapter has **mixed content** because it contains both text and other elements. Para has **simple content** (or **text content**) because it contains only text. Prod has **empty content**, because it carries no information.
- Only the prod element has **attributes**. The **attribute** named id has the **value** "33-657". The **attribute** named media has the **value** "paper".

Element naming

- **XML elements must follow these naming rules:**
 - Names can contain letters, numbers, and other characters
 - Names must not start with a number or punctuation character
 - Names must not start with the xml (or XML or Xml ...)
 - Names cannot contain spaces

- Take care when "inventing" element names and follow these simple rules:
 - Any name can be used, no words are reserved, but the idea is to make names descriptive. Names with an underscore separator are nice.
 - Examples: `<first_name>`, `<last_name>`.
 - Avoid "-" and "." in names. For example, if you name something "first-name", it could be a mess if your software tries to subtract name from first.
 - Element names can be as long as you like, but don't exaggerate. Names should be short and simple, like this: `<book_title>` not like this: `<the_title_of_the_book>`.
 - XML documents often have a corresponding database, in which fields exist corresponding to elements in the XML document. A good practice is to use the naming rules of your database for the elements in the XML documents.
 - Non-English letters like éòá are perfectly legal in XML element names, but watch out for problems if your software vendor doesn't support them.
 - The ":" should not be used in element names because it is reserved to be used for something called namespaces (more later).

Attributes

- **XML elements can have attributes.**
- From HTML you will remember this: ``. The SRC attribute provides additional information about the IMG element.
- In HTML (and in XML) attributes provide additional information about elements:
` `
- Attributes often provide information that is not a part of the data. In the example below, the file type is irrelevant to the data, but important to the software that wants to manipulate the element:
`<file type="gif">computer.gif</file>`

Attributes

- **Quote Styles, "female" or 'female'?**
- Attribute values must always be enclosed in quotes, but either single or double quotes can be used. For a person's sex, the person tag can be written like this:

`<person sex="female">`

or like this:

`<person sex='female'>`

- **Note:** If the attribute value itself contains double quotes, it is necessary to use single quotes, like in this example:

`<gangster name='George "Shotgun" Ziegler'>`

If instead the attribute value itself contains single quotes, it is necessary to use double quotes, like in this example:

`<gangster name="George 'Shotgun' Ziegler">`

Attributes

- **Should you avoid using attributes?**
- Some of the problems with using attributes are:
 - attributes cannot contain multiple values (child elements can)
 - attributes are not easily expandable (for future changes)
 - attributes cannot describe structures (child elements can)
 - attributes are more difficult to manipulate by program code
 - attribute values are not easy to test against a Document Type Definition (DTD) - which is used to define the legal elements of an XML document

Elements vs Attributes

- Try to use **elements** to describe data.
- Use attributes only to provide information that is not relevant to the data or for metadata.
 - Example: ID references can be used to access XML elements
 - Example: creation user and date of a document, date of last document's update

XML

- **A "Well Formed" XML document has correct XML syntax.**
 - A "Well Formed" XML document is a document that conforms to the XML syntax rules that were described
- **A "Valid" XML document also conforms to a DTD.**
 - A "Valid" XML document is a "Well Formed" XML document, which also conforms to the rules of a Document Type Definition (DTD):

Name conflicts

- Since element names in XML are not predefined, a name conflict will occur when two different documents use the same element names.
 - This XML document carries information in a table:
 - `<table> <tr> <td>Apples</td> <td>Bananas</td> </tr> </table>`
 - This XML document carries information about a table (a piece of furniture):
 - `<table> <name>African Coffee Table</name>
<width>80</width> <length>120</length> </table>`
- If these two XML documents were added together, there would be an element name conflict because both documents contain a `<table>` element with different content and definition.

Namespaces

- **Name conflicts are solved by using a prefix**
- This XML document carries information in a table:
 - `<h:table> <h:tr> <h:td>Apples</h:td> <h:td>Bananas</h:td> </h:tr> </h:table>`
- This XML document carries information about a piece of furniture:
 - `<f:table> <f:name>African Coffee Table</f:name> <f:width>80</f:width> <f:length>120</f:length> </f:table>`
- Now there will be no name conflict because the two documents use a different name for their `<table>` element (`<h:table>` and `<f:table>`).
- By using a prefix, we have created two different types of `<table>` elements.

Uniform Resource Identifiers (URIs)

- A **Uniform Resource Identifier** (URI) is a string of characters which identifies an Internet Resource. The most common URI is the **Uniform Resource Locator** (URL) which identifies an Internet domain address. Another, not so common type of URI is the **Universal Resource Name** (URN). Usually URLs are used.

Namespace References

- This XML document carries information in a table:
 - `<h:table xmlns:h="http://www.w3.org/TR/html4/"> <h:tr>
<h:td>Apples</h:td> <h:td>Bananas</h:td> </h:tr>
</h:table>`
- This XML document carries information about a piece of furniture:
 - `<f:table xmlns:f="http://www.w3schools.com/furniture">
<f:name>African Coffee Table</f:name>
<f:width>80</f:width> <f:length>120</f:length> </f:table>`

The XML Namespace (xmlns) Attribute

- The XML namespace attribute is placed in the start tag of an element and has the following syntax:
`xmlns:namespace-prefix="namespaceURI"`
- When a namespace is defined in the start tag of an element, all child elements with the same prefix are associated with the same namespace.
- Note that the address used to identify the namespace is not used by the parser to look up information. The only purpose is to give the namespace a unique name. However, very often companies use the namespace as a pointer to a real Web page containing information about the namespace.

DTD

- The purpose of a Document Type Definition is to define the legal building blocks of an XML document. It defines the document structure with a list of legal elements.
- A DTD can be declared inline in the XML document, or as an external reference.
- If the DTD is included in the XML source file, it should be wrapped in a DOCTYPE definition with the following syntax:

`<!DOCTYPE root-element [element-declarations]>`

Why use a DTD?

- With DTD, each of your XML files can carry a description of its own format with it.
- With a DTD, independent groups of people can agree to use a common DTD for interchanging data.
- Your application can use a standard DTD to verify that the data you receive from the outside world is valid.
- You can also use a DTD to verify your own data.

Internal DTD

- `<?xml version="1.0"?>`
`<!DOCTYPE note`
 `[<!ELEMENT note (to,from,heading,body)>`
 `<!ELEMENT to (#PCdata)>`
 `<!ELEMENT from (#PCdata)>`
 `<!ELEMENT heading (#PCdata)>`
 `<!ELEMENT body (#PCdata)>]>`
`<note>`
 `<to>Tove</to>`
 `<from>Jani</from>`
 `<heading>Reminder</heading>`
 `<body>Don't forget me this weekend</body>`
`</note>`

DTD

- The DTD above is interpreted like this:
 - **!DOCTYPE note** (in line 2) defines that this is a document of the type **note**.
 - **!ELEMENT note** (in line 3) defines the **note** element as having four elements: "to,from,heading,body".
 - **!ELEMENT to** (in line 4) defines the **to** element to be of the type "#PCdata".
 - **!ELEMENT from** (in line 5) defines the **from** element to be of the type "#PCdata".
 - and so on.....

External DTD

- If the DTD is external to your XML source file, it should be wrapped in a DOCTYPE definition with the following syntax:

```
<!DOCTYPE root-element SYSTEM "filename">
```

```
<?xml version="1.0"?>
```

```
<!DOCTYPE note SYSTEM "note.dtd">
```

```
<note> <to>Tove</to> <from>Jani</from>
```

```
<heading>Reminder</heading>
```

```
<body>Don't forget me this weekend!</body> </note>
```

- This is a copy of the file "note.dtd" containing the DTD:

```
<!ELEMENT note (to,from,heading,body)>
```

```
<!ELEMENT to (#PCdata)>
```

```
<!ELEMENT from (#PCdata)>
```

```
<!ELEMENT heading (#PCdata)>
```

```
<!ELEMENT body (#PCdata)>
```

Declaring an element

- In the DTD, XML elements are declared with an element declaration. An element declaration has the following syntax:

`<!ELEMENT element-name category>`

or

`<!ELEMENT element-name (element-content)>`

Declaring elements

Empty elements

- Empty elements are declared with the category keyword EMPTY:
`<!ELEMENT element-name EMPTY>`
example: `<!ELEMENT br EMPTY>`

Elements with only character data

- Elements with only character data are declared with #PCdata inside parentheses:
`<!ELEMENT element-name (#PCdata)>`

Elements with any contents

- Elements declared with the category keyword ANY, can contain any combination of parsable data:
`<!ELEMENT element-name ANY>`

Elements with children

- Elements with one or more children are defined with the name of the children elements inside parentheses:
 <!ELEMENT element-name (child-element-name)>
 <!ELEMENT element-name (child-element-name,child-element-name,.....)>
example: <!ELEMENT note (to,from,heading,body)>
- When children are declared in a sequence separated by commas, the children must appear in the same sequence in the document. In a full declaration, the children must also be declared, and the children can also have children.

One, min one

- **Declaring only one occurrence of the same element**
 <!ELEMENT element-name (child-name)>
 <!ELEMENT note (message)>
● The example declaration above declares that the child element message must occur once, and only once inside the "note" element.
- **Declaring minimum one occurrence of the same element**
 <!ELEMENT element-name (child-name+)>
 <!ELEMENT note (message+)>
● The + sign in the example above declares that the child element message must occur one or more times inside the "note" element.

Zero or more, zero or one

- **Declaring zero or more occurrences of the same element**
`<!ELEMENT element-name (child-name*)>`
`<!ELEMENT note (message*)>`
- The * sign in the example above declares that the child element message can occur zero or more times inside the "note" element.
- **Declaring zero or one occurrences of the same element**
`<!ELEMENT element-name (child-name?)>`
`<!ELEMENT note (message?)>`
- The ? sign in the example above declares that the child element message can occur zero or one times inside the "note" element.

Alternative and mixed content

- **Declaring either/or content**

example: `<!ELEMENT note (to,from,header,(message|body))>`

- The example above declares that the "note" element must contain a "to" element, a "from" element, a "header" element, and either a "message" or a "body" element.

- **Declaring mixed content**

example: `<!ELEMENT note (#PCdata|to|from|header|message)*>`

- The example above declares that the "note" element can contain zero or more occurrences of parsed character, "to", "from", "header", or "message" elements.

Declaring attributes

- An attribute declaration has the following syntax:
`<!ATTLIST element-name attribute-name attribute-type default-value>`
- Example:
`<!ATTLIST payment type Cdata "check">`
Corresponding XML
`<payment type="check" />`

Attribute type

The attribute-type can have the following values:

- | | |
|----------------|---|
| ● Cdata | The value is character data |
| ● (en1 en2 ..) | The value must be one from an enumerated list |
| ● ID | The value is a unique id |
| ● IDREF | The value is the id of another element |
| ● IDREFS | The value is a list of other ids |
| | |
| ● NMTOKEN | The value is a valid XML name |
| ● NMTOKENS | The value is a list of valid XML names |
| ● ENTITY | The value is an entity |
| ● ENTITIES | The value is a list of entities |
| ● NOTATION | The value is a name of a notation |
| ● xml: | The value is a predefined xml value |

Default values

The default-value can have the following values:

- Value The default value of the attribute
- #REQUIRED The attribute value must be included in the element
- #IMPLIED The attribute does not have to be included
- #FIXED value The attribute value is fixed

Example of attribute declarations

<!ELEMENT PRODUCT (.....)

<!ATTLIST PRODUCT

| | | |
|--------|-------------------------|---------------|
| code | ID | #REQUIRED |
| label | CDATA | #IMPLIED |
| status | (available unavailable) | 'available' > |

A simple DTD

```
<!DOCTYPE NEWSPAPER [  
  <!ELEMENT NEWSPAPER (ARTICLE+)>  
  <!ELEMENT ARTICLE (HEADLINE,BYLINE,LEAD,BODY,NOTES)>  
  <!ELEMENT HEADLINE (#PCDATA)>  
  <!ELEMENT BYLINE (#PCDATA)>  
  <!ELEMENT LEAD (#PCDATA)>  
  <!ELEMENT BODY (#PCDATA)>  
  <!ELEMENT NOTES (#PCDATA)>  
  <!ATTLIST ARTICLE  
    AUTHOR CDATA #REQUIRED  
    EDITOR CDATA #IMPLIED  
    DATE CDATA #IMPLIED  
    EDITION CDATA #IMPLIED >  
>
```

XML Schema

- XML Schema is a richer, XML based alternative to DTD
- An XML schema describes the structure of an XML document
- The XML Schema language is also referred to as XML Schema Definition (XSD)

Why XML Schemas?

- The purpose of an XML Schema is to define the legal building blocks of an XML document, just like a DTD.
- An XML Schema:
 - defines elements that can appear in a document
 - defines attributes that can appear in a document
 - defines which elements are child elements
 - defines the order of child elements
 - defines the number of child elements
 - defines whether an element is empty or can include text
 - defines data types for elements and attributes
 - defines default and fixed values for elements and attributes

An example of XSD

- ```
<?xml version="1.0"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema"
...>
 <xs:element name="note">
 <xs:complexType>
 <xs:sequence>
 <xs:element name="to" type="xs:string"/>
 <xs:element name="from" type="xs:string"/>
 <xs:element name="heading" type="xs:string"/>
 <xs:element name="body" type="xs:string"/>
 </xs:sequence>
 </xs:complexType>
 </xs:element>
</xs:schema>
```

## Simple and complex types

- The **note** element is said to be of a **complex type** because it contains other elements
- The other elements (to, from, heading, body) are said to be **simple types** because they do not contain other elements

## **Simple elements**

- A simple element is an XML element that can contain only text. It cannot contain any other elements or attributes.
- However, the "only text" restriction is quite misleading. The text can be of many different types. It can be one of the types that are included in the XML Schema definition (boolean, string, date, etc.), or it can be a custom type that you can define yourself.
- You can also add restrictions (facets) to a data type in order to limit its content, and you can require the data to match a defined pattern.

## Simple elements

- The syntax for defining a simple element is:  
`<xs:element name="xxx" type="yyy"/>`
- where xxx is the name of the element and yyy is the data type of the element. Here are some XML elements:

`<lastname>Refsnes</lastname>`

`<age>34</age>`

`<dateborn>1968-03-27</dateborn>`

and here are the corresponding simple element definitions:

`<xs:element name="lastname" type="xs:string"/>`

`<xs:element name="age" type="xs:integer"/>`

`<xs:element name="dateborn" type="xs:date"/>`

## **XML Schema data types**

- XML Schema has a lot of built-in data types. Here is a list of the most common of them:
  - xs:string
  - xs:decimal
  - xs:integer
  - xs:boolean
  - xs:date
  - xs:time



## **Default and Fixed values for simple elements**

- Simple elements can have a default value OR a fixed value set.
- A default value is automatically assigned to the element when no other value is specified. In the following example the default value is "red":

```
<xs:element name="color" type="xs:string" default="red"/>
```

- A fixed value is also automatically assigned to the element. You cannot specify another value. In the following example the fixed value is "red":

```
<xs:element name="color" type="xs:string" fixed="red"/>
```

## Attribute definition

- The syntax for defining an attribute is:

```
<xs:attribute name="xxx" type="yyy"/>
```

where xxx is the name of the attribute and yyy is the data type of the attribute.

- Here is an XML element with an attribute:

```
<lastname lang="EN">Smith</lastname>
```

and the corresponding simple attribute definition:

```
<xs:attribute name="lang" type="xs:string"/>
```

## Value restrictions

- This example defines an element called "age" with a restriction. The value of age cannot be lower than 0 or greater than 100:

```
<xs:element name="age">
 <xs:simpleType>
 <xs:restriction base="xs:integer">
 <xs:minInclusive value="0"/>
 <xs:maxInclusive value="100"/>
 </xs:restriction>
 </xs:simpleType>
</xs:element>
```

## Enumeration constraint

- To limit the content of an XML element to a set of acceptable values, we would use the enumeration constraint.
- This example defines an element called "car":

```
<xs:element name="car">
 <xs:simpleType>
 <xs:restriction base="xs:string">
 <xs:enumeration value="Audi"/>
 <xs:enumeration value="Golf"/>
 <xs:enumeration value="BMW"/>
 </xs:restriction>
 </xs:simpleType>
</xs:element>
```
- The "car" element is a simple type with a restriction. The acceptable values are: Audi, Golf, BMW.

## Separate type definition

- The previous example could also have been written like this:
- `<xs:element name="car" type="carType"/>`
- `<xs:simpleType name="carType">`
  - `<xs:restriction base="xs:string">`
    - `<xs:enumeration value="Audi"/>`
    - `<xs:enumeration value="Golf"/>`
    - `<xs:enumeration value="BMW"/>`
  - `</xs:restriction>`
- `</xs:simpleType>`
- In this case the type "carType" can be used by other elements because it is not a part of the "car" element.

## Pattern constraints

- To limit the content of an XML element to define a series of numbers or letters, we would use the pattern constraint.
- This example defines an element called "letter":
- ```
<xs:element name="letter">  
  <xs:simpleType>  
    <xs:restriction base="xs:string">  
      <xs:pattern value="[a-z]"/>  
    </xs:restriction>  
  </xs:simpleType>  
</xs:element>
```
- The "letter" element is a simple type with a restriction. The acceptable value is one of the lowercase letters from a to z.

Complex elements

- A complex element is an XML element that contains other elements and/or attributes.
- There are four kinds of complex elements:
 - empty elements
 - elements that contain only other elements
 - elements that contain only text
 - elements that contain both other elements and text
- **Note:** Each of these elements may contain attributes as well!

Example of complex element

- The "employee" element can be declared directly by naming the element, like this:
- ```
<xs:element name="employee">
 <xs:complexType>
 <xs:sequence>
 <xs:element name="firstname" type="xs:string"/>
 <xs:element name="lastname" type="xs:string"/>
 </xs:sequence>
 </xs:complexType>
</xs:element>
```



## Use of externally defined complex types

- The "employee" element can have a type attribute that refers to the name of the complex type to use:

```
<xs:element name="employee" type="personinfo"/>
```

```
<xs:element name="student" type="personinfo"/>
```

```
<xs:complexType name="personinfo">
```

```
 <xs:sequence>
```

```
 <xs:element name="firstname" type="xs:string"/>
```

```
 <xs:element name="lastname" type="xs:string"/>
```

```
 </xs:sequence>
```

```
</xs:complexType>
```

**Example: Complex type for empty element**

- ```
<xs:element name="product">  
  <xs:complexType>  
    <xs:attribute name="prodid" type="xs:positiveInteger"/>  
  </xs:complexType>  
</xs:element>
```

Sequence vs Mixed (1)

- ```
<xs:element name="person">
 <xs:complexType>
 <xs:sequence>
 <xs:element name="firstname" type="xs:string"/>
 <xs:element name="lastname" type="xs:string"/>
 </xs:sequence>
 </xs:complexType>
</xs:element>
```
- Note the `<xs:sequence>` tag: It means that the elements defined ("firstname" and "lastname") must appear in that order inside a "person" element.

## Sequence vs Mixed (2)

- ```
<xs:element name="letter">  
  <xs:complexType mixed="true">  
    <xs:sequence>  
      <xs:element name="name" type="xs:string"/>  
      <xs:element name="orderid" type="xs:positiveInteger"/>  
      <xs:element name="shipdate" type="xs:date"/>  
    </xs:sequence>  
  </xs:complexType>  
</xs:element>
```
- Note the mixed attribute: It means that the elements defined ("name", "order-id" and "shipdate") can appear in any order inside a "person" element.
- Legal XML document:

```
<letter> Dear Mr.<name>John Smith</name>. Your order  
<orderid>1032</orderid> will be shipped on <shipdate>2001-07-  
13</shipdate>. </letter>
```

- The <all> indicator specifies by default that the child elements can appear in any order and that each child element must occur once and only once:

```
<xs:element name="person">  
  <xs:complexType>  
    <xs:all>  
      <xs:element name="firstname" type="xs:string"/>  
      <xs:element name="lastname" type="xs:string"/>  
    </xs:all>  
  </xs:complexType>  
</xs:element>
```

Choice

- The <choice> indicator specifies that either one child element or another can occur:
- ```
<xs:element name="person">
 <xs:complexType>
 <xs:choice>
 <xs:element name="employee" type="employee"/>
 <xs:element name="member" type="member"/>
 </xs:choice>
 </xs:complexType>
</xs:element>
```

## MaxOccurs and minOccurs

- The <maxOccurs> indicator specifies the maximum number of times an element can occur:
- The <minOccurs> indicator specifies the minimum number of times an element can occur:
- ```
<xs:element name="person">  <xs:complexType> <xs:sequence>  
    <xs:element name="full_name" type="xs:string"/>  
    <xs:element name="child" type="xs:string"  
        maxOccurs="10" minOccurs="0"/>  
</xs:sequence> </xs:complexType> </xs:element>
```
- The example indicates that the "child" element can miss or can occur a maximum of ten times in a "person" element.
- The default value for minOccurs and maxOccurs is 1
- To allow an element to appear an unlimited number of times, use the maxOccurs="unbounded" statement.

Any

- The <any> element enables us to extend the XML document with elements not specified by the schema.
- The following example is a fragment from an XML schema called "family.xsd". It shows a declaration for the "person" element. By using the <any> element we can extend (after <lastname>) the content of "person" with any element:

```
<xs:element name="person">  <xs:complexType>
<xs:sequence>
    <xs:element name="firstname" type="xs:string"/>
    <xs:any minOccurs="0"/>
</xs:sequence> </xs:complexType> </xs:element>
```


AnyAttribute

- The `<anyAttribute>` element enables us to extend the XML document with attributes not specified by the schema.

```
<xs:element name="person">  <xs:complexType>
  <xs:sequence>
    <xs:element name="firstname" type="xs:string"/>
    <xs:any minOccurs="0"/>
  </xs:sequence>
  <xs:anyAttribute/>
</xs:complexType> </xs:element>
```

Query Languages for XML

(XPath, XQuery)

Query languages for XML

- XML can be considered as a semi-structured data model
- A set of XML documents (or a single one, e.g. Divina Commedia) can be considered as a large data collection
- Query languages are needed for extracting relevant information from such documents
- The languages are:
 - Xpath a simple document selection language
 - XQuery a rich query language
 - XSLT especially used for document transformations (e.g. Producing HTML from XML) – not discussed in this course

Path expressions

- A path expression starts from the root of the document, e.g. `doc("books.xml")` returns the root element and all its content.
- Starting from the root it is possible to express path expressions in order to extract the desired content
`doc("books.xml")/books/book`
Return **the sequence** of all book elements in the document

Path expressions

```
<?xml version="1.0"?>
<books>
  <book available='Y'>
    <title>Il Signore degli Anelli</title>
    <author>J.R.R. Tolkien</author>
    <date>2002</date>
    <ISBN>88-452-9005-0</ISBN>
    <editor>Bompiani</editor>
  </book>
  <book available='N'>
    <title>Il nome della rosa</title>
    <author>Umberto Eco</author>
    <date>1987</date>
    <ISBN>55-344-2345-1</ISBN>
    <editor>Bompiani</editor>
  </book>
  <book available='Y'>
    <title>Il sospetto</title>
    <author>F. Dürrenmatt</author>
    <date>1990</date>
    <ISBN>88-07-81133-2</ISBN>
    <editor>Feltrinelli</editor>
  </book>
</books>
```

doc("books.xml")/books/book



```
<book available='Y'>
  <title>Il Signore degli Anelli</title>
  <author>J.R.R. Tolkien</author>
  <date>2002</date>
  <ISBN>88-452-9005-0</ISBN>
  <editor>Bompiani</editor>
</book>
<book available='N'>
  <title>Il nome della rosa</title>
  <author>Umberto Eco</author>
  <date>1987</date>
  <ISBN>55-344-2345-1</ISBN>
  <editor>Bompiani</editor>
</book>
<book available='Y'>
  <title>Il sospetto</title>
  <author>F. Dürrenmatt</author>
  <date>1990</date>
  <ISBN>88-07-81133-2</ISBN>
  <editor>Feltrinelli</editor>
</book>
```

Conditions

- `doc ("books.xml")/books/book[editor='Bompiani']/title`
Return the sequence of all titles of books edited by Bompiani

<title>Il Signore degli Anelli</title>
<title>Il nome della rosa</title>

Subelements

- `doc("books.xml")//author`
Return the sequence of all authors in the document, despite the nesting level

```
<author>J.R.R. Tolkien</author>  
<author>Umberto Eco</author>  
<author>F. Dürrenmatt</author>
```

Ordered elements

- `doc ("books.xml")/books/book[2]/*`
Return all the elements (*) contained into the second book

```
<title>Il nome della rosa</title>  
<author>Umberto Eco</author>  
<date>1987</date>  
<ISBN>55-344-2345-1</ISBN>  
<editor>Bompiani</editor>
```


XPATH EXAMPLES

- `/bookstore` Selects the root element bookstore
(If the path starts with a slash (/) it always represents an absolute path to an element!)
- `bookstore/book` Selects all book elements that are children of bookstore
- `//book` Selects all book elements no matter where they are in the document
- `bookstore//book` Selects all book elements that are descendant of the bookstore element, no matter where they are under the bookstore element
- `//@lang` Selects all attributes that are named lang

XPATH PREDICATES

Predicates are used to find a specific node or a node that contains a specific value. Predicates are always embedded in square brackets

<code>/bookstore/book[1]</code>	Selects the first book element that is the child of the bookstore element
<code>/bookstore/book[last()]</code>	Selects the last book element that is the child of the bookstore element
<code>/bookstore/book[last()-1]</code>	Selects the last but one book element that is the child of the bookstore element
<code>/bookstore/book[position()<3]</code>	Selects the first two book elements that are children of the bookstore element
<code>//title[@lang]</code>	Selects all the title elements that have an attribute named lang
<code>//title[@lang='eng']</code>	Selects all the title elements that have an attribute named lang with a value of 'eng'
<code>/bookstore/book[price>35.00]</code>	Selects all the book elements of the bookstore element having a price element with a value greater than 35.00

Wildcards

XPath wildcards can be used to select unknown XML elements.

● Wildcard	Description
● *	Matches any element node
● @*	Matches any attribute node
● node()	Matches any node of any kind
● Path	ExpressionResult
● /bookstore/*	Selects all the child nodes of the bookstore element
● //*	Selects all elements in the document
● //title[@*]	Selects all title elements which have any attribute

Alternative paths

The | operator in an XPath expression indicates several alternative paths that can be used to select results

- `//book/title | //book/price`
Selects all the title AND price elements of all book elements
- `//title | //price`
Selects all the title AND price elements in the document
- `/bookstore/book/title | //price`
Selects all the title elements of the book element of the bookstore element AND all the price elements in the document

XQuery

- **XQuery is to XML what SQL is to relational databases.**
- **XQuery is designed to query XML data - not just XML files, but anything that can appear as XML, including databases.**
- XQuery is built on XPath expressions
- XQuery is defined by the W3C
- XQuery is supported by all the major database engines (IBM, Oracle, Microsoft, etc.)
- XQuery is a W3C standard - and developers who correctly implement its specs are guaranteed that the code will work among different products

XQuery Structure

- An XQuery basic structure:
 - prologue + expression
- Role of the prologue:
 - Populate the context where the expression is compiled and evaluated
- Prologue contains:
 - namespace definitions
 - schema imports
 - default element and function namespace
 - function definitions
 - collations declarations
 - function library imports
 - global and external variables definitions
 - etc.

XQuery expressions

XQuery Expr := Constants | Variable | FunctionCalls |
PathExpr | ComparisonExpr |
ArithmeticExpr | LogicExpr |
FLWRExpr | ConditionalExpr |
QuantifiedExpr | TypeSwitchExpr |
InstanceofExpr | CastExpr | UnionExpr |
IntersectExceptExpr | ConstructorExpr |
ValidateExpr

Expressions can be nested with full generality!

Constants

- XQuery grammar has built-in support for:
 - Strings: "125.0" or '125.0'
 - Integers: 150
 - Decimal: 125.0
 - Double: 125.e2
 - 19 other atomic types available via XML Schema
 - Values can be constructed
 - with constructors in F&O doc: xf:true(),
xf:date("2002-5-20")
 - by casting
 - by schema validation

Variables

- \$ + Name
- bound by expression evaluation
- created by let, for, some/every, type-switch expressions, or as function parameters
- example:
 - let \$x := (1, 2, 3)
return count(\$x)
- above scoping ends at conclusion of return expression

Functions

- XQuery uses functions to extract data from XML documents.
- The doc() function is used to open the "books.xml" file:
doc("books.xml")

Examples of functions

- `doc(xs:anyURI) => document?`
- `empty(item*) => boolean`
- `index-of(item*, item) => xs:unsignedInt?`
- `distinct-values(item*) => item*`
- `distinct-nodes(node*) => node*`
- `union(node*, node*) => node*`
- `except(node*, node*) => node*`
- `string-length(xs:string?) => xs:integer?`
- `contains(xs:string, xs:string) => xs:boolean`
- `true() => xs:boolean`
- `date(xs:string) => xs:date`
- `add-date(xs:date, xs:duration) => xs:date`

Combining sequences

- Union, Intersect, Except
- Work only for sequences of nodes, not atomic values
- Eliminate duplicates and reorder to document order
 $\$x := \langle a \rangle, \$y := \langle b \rangle, \$z := \langle c \rangle$
 $(\$x, \$y) \text{ union } (\$y, \$z) \Rightarrow (\langle a \rangle, \langle b \rangle, \langle c \rangle)$
- F&O specification provides other functions & operators;
eg `xf:distinct-values()` and `xf:distinct-nodes()`
particularly useful.

Arithmetic expressions

- Examples:
 - $-1 - (4 * 8.5)$
 - $\$b \bmod 10$
 - $\langle a \rangle 42 \langle /a \rangle + 1$
- Apply the following rules:
 - atomize all operands
 - if every item in the input sequence is either an atomic value or a node whose typed value is a sequence of atomic values, then return it
 - if an operand is untyped, cast to xs:double (if unable, => error)
 - if the operand types differ but can be promoted to common type, do so (e.g.: xs:integer can be promoted to xs:decimal)
 - if operator is consistent w/ types, apply it; result is either atomic value or error
 - otherwise, throw type exception

Logical expressions

- They are:
 - `expr1` and `expr2`
 - `expr1` or `expr2`
- They return true, false (with two value logic, not three value logic like SQL!)
- Rules:
 - first compute the Boolean Effective Value (BEV) for each operand:
 - if `()`, `""`, `0`, zero length string then return false
 - if the operand is of type boolean, its BEV is its value;
 - else return true
 - then use standard two value Boolean logic on the two BEV's as appropriate
- false and error => false or error! (non-deterministically: it is impossible foresee the result!!!)

Comparison

Value	for comparing single values	eq, ne, lt, le, gt, ge
General	Existential quantification + automatic type coercion	=, !=, <=, <, >, >=
Node	for testing identity of single nodes	is, isnot
Order	testing relative position of one node vs. another (in document order)	<<, >>

Values and comparisons

- `<a>42 eq "42"` true
- `<a>42 eq 42` error
- `<a>42 eq 42.0` error
- `<a>42 = 42` true
- `<a>42 = 42.0` true
- `<a>42 eq 42` true
- `<a>42 eq 42` false
- `<a>baz eq 42` type error
- `() = 42` false
- `(<a>42, 43) = 42` true
- `ns:shoesize(5) eq ns:hatsize(5)` true
- `(1,2) = (2,3)` true

Running example

- ```
<?xml version="1.0" encoding="ISO-8859-1"?>
<bookstore>
 <book category="COOKING">
 <title lang="en">XQuery Kick Start</title>
 <author>James McGovern</author>
 <author>Per Bothner</author>
 <author>Kurt Cagle</author>
 <author>James Linn</author>
 <author>Vaidyanathan Nagarajan</author>
 <year>2003</year>
 <price>49.99</price>
 </book>
 ...
</bookstore>
```

## XPath expressions in XQuery

- XQuery uses path expressions to navigate through elements in an XML document.
- The following path expression is used to select all the title elements in the "books.xml" file:  
`doc("books.xml")/bookstore/book/title`  
(/bookstore selects the bookstore element,  
/book selects all the book elements under the  
bookstore element  
/title selects all the title elements under each book  
element)
- The XQuery above will extract the following:  
`<title lang="en">XQuery Kick Start</title>`

**XPath filter predicates**

- XQuery uses predicates to limit the extracted data from XML documents.
- The following predicate is used to select all the book elements under the bookstore element that have a price element with a value that is less than 30:  
`doc("books.xml")/bookstore/book[price<30]`

**XPath filter predicates**

- [ ] is an overloaded operator
- Filtering by position (if numeric value) :
  - /book[3]
  - /book[3]/author[1]
  - /book[3]/author[2 to 4]
- Filtering by predicate :
  - //book[author/firstname = "ronald"]
  - //book[@price <25]
  - //book[count(author[@gender="female"] )>0]

**FLWOR expressions**

- 5 clauses:
  - FOR
  - LET
  - WHERE
  - ORDER BY
  - RETURN

## Simple iteration expressions

- for *variable* in *expression1*  
return *expression2*
- Example  
for \$x in doc("bib.xml")/bib/book  
return \$x/title
- Semantics :
  - bind the variable to each root node of the forest returned by expression1
  - for each such binding evaluate expression2
  - concatenate the resulting sequences
  - nested sequences are automatically flattened

## Nested FOR Expressions

- FOR expressions can be nested:

```
for $book in doc("books.xml")//book
 for $author in $book/Author
 return $author
```

- Semantics: for every value of \$book, for every value of \$author (an author of the current book), insert into the result the author bound to \$author

## LET Expression

- *let variable := expression1*  
*return expression2*
- Example :  
`let $x :=document("bib.xml")/bib/book`  
`return count($x)`
- Semantics :
  - bind the variable to the result of the expression1 (in the example, an entire sequence of values)
  - add this binding to the current environment
  - evaluate and return expression2
- The previous example with nested for can be expressed as:  
`for $book in doc("books.xml")//book`  
`let $a := $book/Author      (: $a is the entire sequence of`  
`book authors :)`  
`return $a`



## WHERE

- WHERE clause express a condition over nodes; only nodes satisfying the condition are further considered.
- Conditions can contain AND and OR.
- **not()** is implemented by a function
- Example:  
    **for** \$book **in** doc ("books.xml")//book  
    **where** \$book/editor="Bompiani"  
        **and** \$book/@available="Y"  
    return \$book
- Same as:  
    for \$book in  
    doc("books.xml")//book[editor="Bompiani" and @available="Y"]

## RETURN

- This clause can generate:
  - A node (or tree)
  - An ordered “forest” (of nodes or trees)
  - A textual value (PCdata)
- It can contain node constructors

```
<author>F. Dürrenmatt</author>
```

```
<author>J.R.R. Tolkien</author>
<author>Umberto Eco</author>
<author>F. Dürrenmatt</author>
```

```
F. Dürrenmatt
```

**Node constructors (within return clause)**

- `<result>`  
    literal text content  
`</result>`
- `<result>`  
    { \$x/name } {-- evaluated content --}  
`</result>`
- `<result>`  
    some content here { \$x/name } and some more here  
`</result>`
- Braces "{ }" used to delineate evaluated content

## FLWR expressions

- Syntactic sugar that combines FOR, LET, IF



- Example

for \$x in //bib/book

let \$y := \$x/author

where \$x/title="The politics of experience"

return count(\$y)

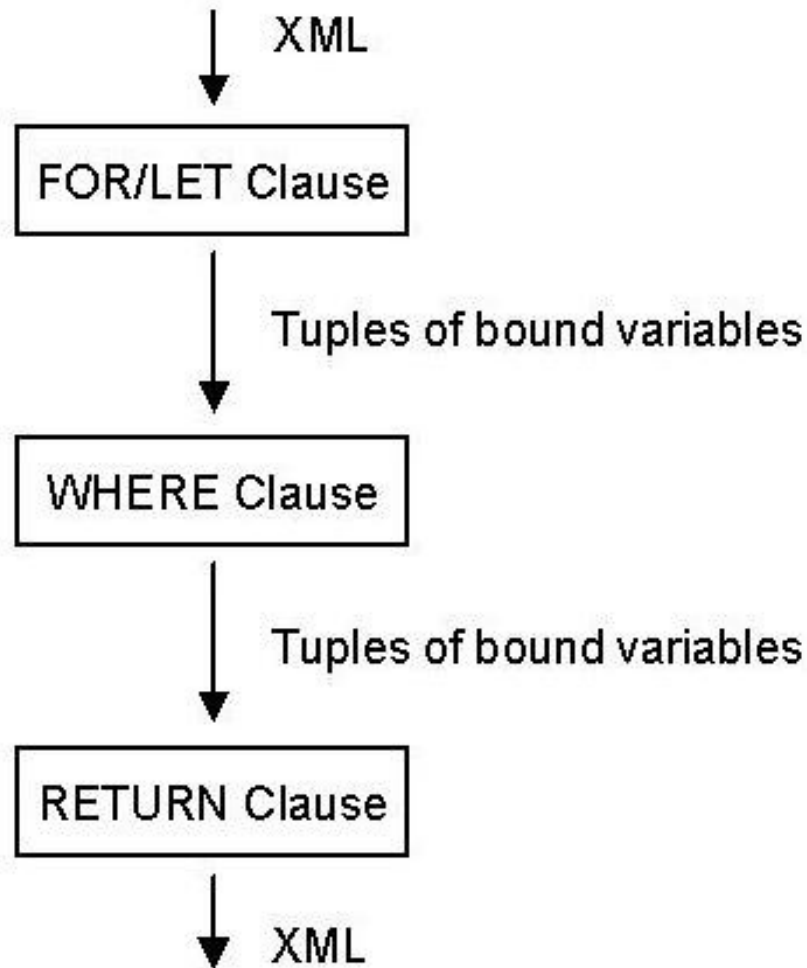
/\* similar to FROM in SQL \*/

/\* no analogy in SQL \*/

/\* similar to WHERE in SQL \*/

/\* similar to SELECT in SQL \*/

## FLWR expressions: interpretation



- **FOR** : iteration
  - Every value in a sequence yields to a different binding
- **LET** : collective binding
  - A sequence is bound to a variable collectively (the number of variable bindings does not change)
- **WHERE** : filter
  - Independently evaluated on each tuple of bindings
- **RETURN** : construct results
  - Executed once for each tuple of bindings

## A FWR query

```
for $book in doc("books.xml")//book
where $book/editor="Bompiani"
return <bookBompiani>
 { $book/title }
 </bookBompiani>
```

```
<bookBompiani><title>Il Signore degli Anelli</title></bookBompiani>
<bookBompiani><title>Il nome della rosa</title></bookBompiani>
```

## Similar query with use of text() function

```
for $book in doc("books.xml")//book
where $book/editor="Bompiani"
return <book-Bompiani>
 { $book/title/text() }
 </book-Bompiani>
```

```
<book-Bompiani>Il Signore degli Anelli</book-Bompiani>
<book-Bompiani>Il nome della rosa</book-Bompiani>
```

## Similar query with LET

```
let $books := doc("books.xml")//book[editor="Bompiani"]
return <book-Bompiani>
 { $books/title }
 </book-Bompiani>
```

```
<bookBompiani>
 <title> Il Signore degli Anelli </title>
 <title> Il nome della rosa </title>
</bookBompiani>
```



## Order by

```
for $book in doc("books.xml")//book
order by $book/title
return
 <book>
 { $book/title,
 $book/editor }
 </book>
```

## Counting combined with let clause

```
for $e in doc("books.xml")//editor
let $book := doc("books.xml")//book[editor = $e]
where count($book) > 100
return $e
```

- Each editor is in the result more than 100 times

**distinct-values()**

```
for $e in distinct-values(doc("books.xml")//editor)
let $book := doc("books.xml")//book[editor = $e]
where count($book) > 100
return $e
```

Each editor is in the result only once

## FLWR expressions

- FLWR expression:  
for \$x in //bib/book  
let \$y := \$x/author  
where \$x/title="Ulysses"  
return count(\$y)
- Equivalent to:  
for \$x in //bib/book  
return (let \$y := \$x/author  
return if (\$x/title="Ulysses" )  
then count(\$y)  
else ()  
)

**FLWR expressions**

- Selections  
for \$b in doc("bib.xml")//book  
where \$b/publisher = "Springer Verlag" and  
    \$b/@year = "1998"  
return \$b/title
- Joins  
for \$b in doc("bib.xml")//book,  
    \$p in doc("pubs.xml")//publisher  
where \$b/publisher = \$p/name  
return ( \$b/title , \$p/address)

**Conditional expressions**

- if ( \$book/@year <1980 )  
then return ( <old-book>{\$x/title}</old-book> )  
else return( <new-book>{\$x/title}</new-book> )

**Order by**

- for \$b in document("bib.xml")//book  
where \$b/publisher = "Springer Verlag"  
order by \$b/@year  
return \$b/title

## Functions

- In-place XQuery functions, such as:  
    declare function ns:foo(\$x as xs:integer) as element()  
    { <a>{\$x+1}</a> }
- Can be recursive and mutually recursive



**Missing functionalities**

- Standard semantics for Web services invocation
- Try-catch mechanism
- Group by
- Distinct by
- Full text search
- Integrity constraints / assertions
- Metadata introspection

**RECENTLY ADDED**

- Updates

## **XML Data Management Technology**

- Two families of XML stores:
- **Native XML dbs**
  - Use XML-specific storage management
  - Support only XML query languages
- **Relational databases with XML support**
  - Based on relational storage systems which are suitably extended
  - Integrate SQL with Xquery

**Native XML databases**

- Use data models which are not relational and can be standard or proprietary
  - ES: DOM, XPath Data Model, XML Information Set
- Use proprietary physical data stores, which are document-centric
- Organize databases as document collections
- Examples: Tamino, Xyleme, eXist, Galax, ...

**Relational DBS with XML support**

- Use relational data stores heavily, but support as well native XML data
- When they map XML data to relational structures, may have rather different mapping to relational schemas:
  - Fixed and DTD independent (canonic)
  - Variable and DTD specific (custom)
  - Decided by the physical DB optimizer (e.g. with mapping rules)