Informatics 1: Data & Analysis

Lecture 10: Structuring XML

Ian Stark

School of Informatics
The University of Edinburgh

Friday 14 February 2013
Semester 2 Week 5

Teaching Weeks !

This is Inf1-DA Lecture 10, in Week 5.

Next week is Innovative Learning Week (ILW). All lectures, tutorials, labs and coursework are suspended for the week, and replaced by a series of alternative events across the University.

http://www.ed.ac.uk/innovative-learning

Check the ILW calendar and sign up now: some activities run all week, some are one-off events.

The week after that, from Monday 24 February, is Teaching Week 6.

Not all courses have noticed that this is the week numbering scheme.

Lecture Plan

XML

We start with technologies for modelling and querying *semistructured data*.

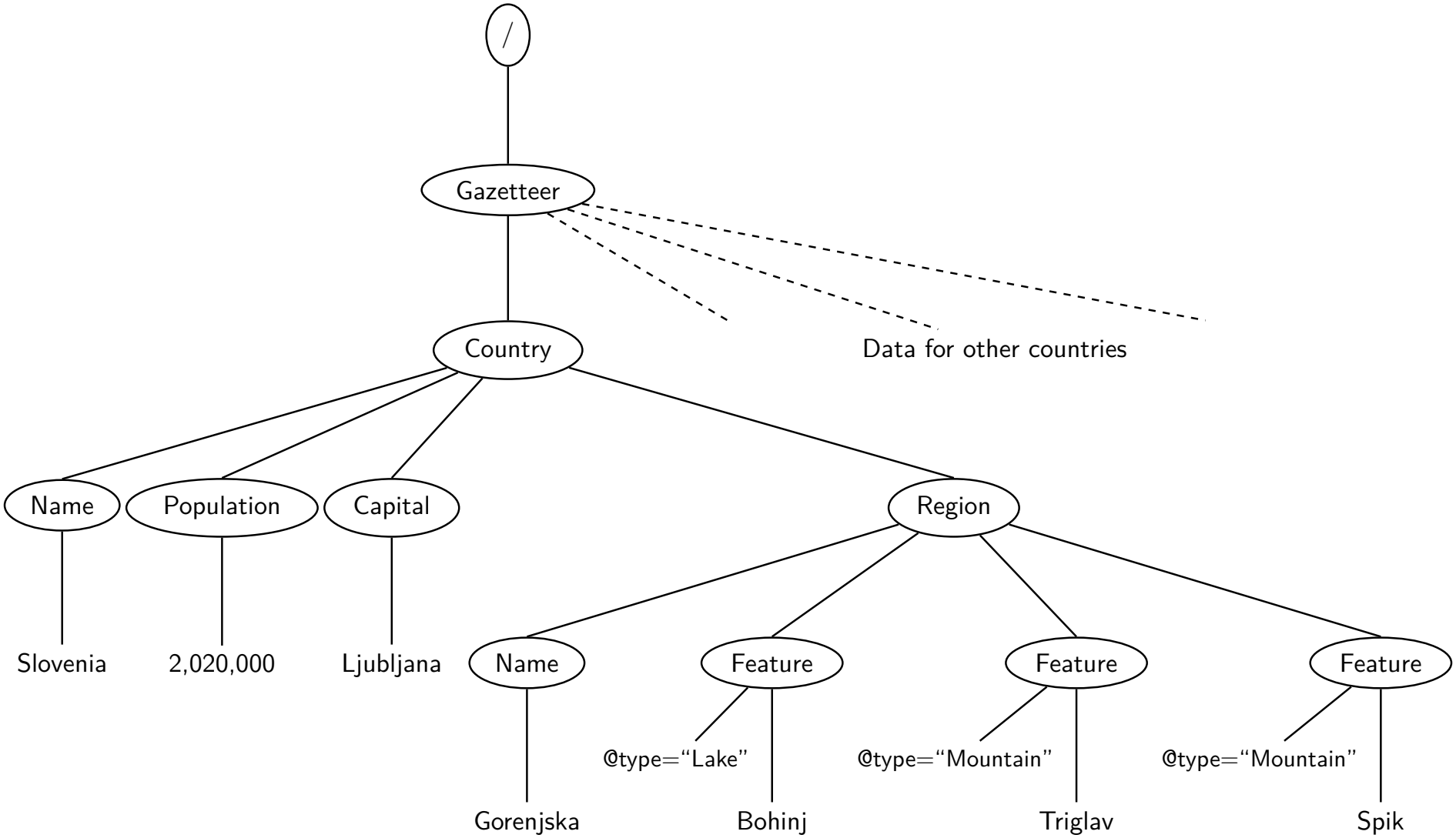
- Semistructured Data: Trees and XML
- Schemas for structuring XML
- Navigating and querying XML with XPath

Corpora

One particular kind of semistructured data is large bodies of written or spoken text: each one a *corpus*, plural *corpora*.

- Corpora: What they are and how to build them
- Applications: corpus analysis and data extraction

Sample Semistructured Data



Sample Semistructured Data in XML

```
<?xml version="1.0" encoding="UTF-8"?>
<Gazetteer>
    <Country>
        <Name>Slovenia</Name>
        <Population>2,020,000</Population>
        <Capital>Ljubljana</Capital>
        <Region>
            <Name>Gorenjska</Name>
            <Feature type="Lake">Bohinj</Feature>
            <Feature type="Mountain">Triglav</Feature>
            <Feature type="Mountain">Spik</Feature>
        </Region>
    </Country>
    <!-- data for other countries here -->
</Gazetteer>
```

Structuring XML

There are a number of basic constraints on XML files, such as proper use of syntax and correctly nesting the tags around elements.

A file satisfying these constraints is a *well-formed XML document*. These are to some extent self-describing:

- The tree structure can always be extracted from textual nesting;
- Elements are always given with their complete name;
- Attributes are all named;
- Everything else is unstructured text.

This is useful as far as it goes, but is fairly rudimentary.

In any given application domain, there may well be a much stricter intended structure which XML documents should follow.

Structuring XML

In any given application domain, there may well be a much stricter intended structure which XML documents should follow.

For example, in the Gazetteer we expect a certain hierarchy:

- The Gazetteer element contains Country elements;
- A Country contains information about its Name, Population and Capital, together with some Region elements.
- A Region includes its Name and zero or more Feature elements.
- Every Feature will include a suitable type attribute.

We specify this kind of expected structure with a *schema*.

Greek σχημα, with plural "schemata" now almost entirely abandoned in favour of "schemas"

Schema Languages for XML

In relational databases, a schema specifies the content of a relation.

A schema language for XML is any language for specifying similar kinds of structure in XML documents. There are a number of different schema languages in common use.

Using a formal schema language means:

- Schemas are precise and unambiguous;
- A machine can *validate* whether or not a document satisfies a certain schema.

If a well-formed XML document D matches the format specified by schema S then we say D is *valid* with respect to S.

One document may be valid with respect to several different schemas; it is also possible to have an XML document that is well-formed but not valid.

Document Type Definitions

Document Type Definition or *DTD* is a basic schema mechanism for XML.

The DTD schema language is simple, widely used, and has been an integrated feature of XML since its inception.

A DTD includes information about:

- Which elements can appear in a document;
- The attributes of those elements;
- The relationship between different elements such as their order, number, and possible nesting.

We illustrate this by going through a sample DTD for a gazetteer, against which the Slovenian example seen earlier can be validated.

Example DTD

```
<!DOCTYPE Gazetteer [

<!ELEMENT Gazetteer (Country+)>
<!ELEMENT Country (Name,Population,Capital,Region*) >
<!ELEMENT Name (#PCDATA)>
<!ELEMENT Population (#PCDATA)>
<!ELEMENT Capital (#PCDATA)>
<!ELEMENT Region (Name,Feature*) >
<!ELEMENT Feature (#PCDATA)>

<!ATTLIST Feature type CDATA #REQUIRED>
]>
```

Some think DTD syntax a little ugly

Dissecting a DTD

Every DTD is a list of individual declarations framed within a DOCTYPE declaration:

<!**DOCTYPE** *name* [

Element declaration
...
Attribute declaration
...
Element declaration
...

]>

Here *name* identifies the document type we are declaring. Also, any XML document matching the DTD must have a *name* element as its root.

Dissecting a DTD

Every DTD is a list of element and attribute declarations.

Dissecting a DTD

Every DTD is a list of element and attribute declarations.

<!**ELEMENT** Gazetteer (Country+)>

This declares that the Gazetteer element consists of one or more Country elements.

Dissecting a DTD

Every DTD is a list of element and attribute declarations.

<!**ELEMENT** Gazetteer (Country+)>

This declares that the Gazetteer element consists of one or more Country elements.

<!**ELEMENT** Country (Name,Population,Capital,Region*)>

This declares that a Country element consists of one Name element, followed by one Population element, followed by one Capital element, followed by zero or more Region elements, all in that order.

Dissecting a DTD

Every DTD is a list of element and attribute declarations.

<!**ELEMENT** Gazetteer (Country+)>

This declares that the Gazetteer element consists of one or more Country elements.

<!**ELEMENT** Country (Name,Population,Capital,Region*)>

This declares that a Country element consists of one Name element, followed by one Population element, followed by one Capital element, followed by zero or more Region elements, all in that order.

<!**ELEMENT** Name (#**PCDATA**)>

This declares that the Name element contains text. The keyword #PCDATA stands for "parsed character data".

Dissecting a DTD

<!**ELEMENT** Region (Name,Feature∗)>

This declares that a Region element consists of one Name followed by zero or more Feature elements.

Dissecting a DTD

<!**ELEMENT** Region (Name,Feature∗)>

This declares that a Region element consists of one Name followed by zero or more Feature elements.

<!**ELEMENT** Feature (#**PCDATA**)>

This declares that the Feature element contains just text.

Dissecting a DTD

<!**ELEMENT** Region (Name,Feature∗)>

This declares that a Region element consists of one Name followed by zero or more Feature elements.

<!**ELEMENT** Feature (#**PCDATA**)>

This declares that the Feature element contains just text.

<!**ATTLIST** Feature type **CDATA** #**REQUIRED**>

This declares that the Feature element must have an attribute called type, and that the value of the attribute should be a text string (CDATA stands for "character data").

Why #PCDATA and CDATA? Historical reasons. Please don't ask.
There are precise explanations, but it's hair-splitting.

Element Declarations

An element declaration has this form:

<!**ELEMENT** *elementName contentType*>

There are four possible content types.

1. EMPTY indicating that the element has no content.
2. ANY meaning that any content is allowed
   (Elements nested within this still need their own declarations).
3. Mixed content where the element contains text, and possibly also child elements.
4. A child declaration using a regular expression.

See the next slide for more on mixed content and regular expressions. . .

Element Declarations

An mixed content element declaration has one of these forms:

<!**ELEMENT** *elementName* (#PCDATA) >

<!**ELEMENT** *elementName* (#PCDATA | *child* | *child* | ...)* >

The first of these means that the element can contain arbitrary text as a child node, but no further element nodes.

The second form allows text interspersed with any of the child element nodes named in the declaration, in any order.

The "*" is literal XML syntax, indicating possible repetitions; the ellipsis " ... " is not XML syntax, it's there to indicate that the declaration may mention any number of different child elements.

Element Declarations

A child declaration uses regular expressions to indicate what element combinations are valid as children of *elementName*.

<!**ELEMENT** *elementName* (*regexp*) >
<!**ELEMENT** *elementName* (*regexp*)? >
<!**ELEMENT** *elementName* (*regexp*)∗ >
<!**ELEMENT** *elementName* (*regexp*)+ >

The *regexp* can be built from any of the following, nested as required:

- A single element name: just that element matches.
- *re1*, *re2* : content matching *re1* followed by more matching *re2*.
- *re1* | *re2* : content matching either *re1* or *re2*.
- Any of these followed by ?, ∗ or + for zero-or-one, zero-or-more, or one-or-more repetitions, respectively.
- Any of these in parentheses (*re*), needed to avoid ambiguity.

Attribute Declarations

Attributes of an element are declared separately to the element itself.

<!**ATTLIST** *elementName attName attType attDefault ...* >

This defines one or more attributes for the named element. Multiple attributes can either be defined all together, using the ... here, or one at a time in several separate declarations.

Each attribute has three items declared:

- *attName* is the attribute name
- *attType* is a datatype for the value of the attribute.
- *attDefault* indicates whether the attribute is required or optional, and may specify a default value.

Attribute Datatypes and Defaults

Possible datatype declarations for attributes include:

- String: **CDATA** means that the attribute may take any string value.
- Enumeration: (s_1 | s_2 | . . . | s_k) indicates the attribute value must be one of the strings s_1, s_2, . . . , s_k.

Other possibilities are various technical kinds of entities and tokens.

The *attDefault* declaration can be any of:

- **#REQUIRED** meaning that the attribute must always be given a value in the start tag for that element.
- **#IMPLIED** meaning that the attribute may be given a value, but it isn't essential.
- Giving a particular string means that value is the default for the attribute, unless otherwise declared in the element start tag.

Example DTD

<!**DOCTYPE** Gazetteer [

<!**ELEMENT** Gazetteer (Country+) >
<!**ELEMENT** Country (Name,Population,Capital,Region*) >
<!**ELEMENT** Name (#**PCDATA**) >
<!**ELEMENT** Population (#**PCDATA**) >
<!**ELEMENT** Capital (#**PCDATA**) >
<!**ELEMENT** Region (Name,Feature*) >
<!**ELEMENT** Feature (#**PCDATA**) >

<!**ATTLIST** Feature type **CDATA** #**REQUIRED**>
]>

Variation

We could replace the original Feature attribute declaration

<!**ATTLIST** Feature type **CDATA** #**REQUIRED**>

with an alternative

<!**ATTLIST** Feature type (Mountain|Lake|River) "Mountain">

This declares a specific list of feature types, and also a default

The original Gazetteer would still validate against this, and so would this:

<Feature>Ben Nevis</Feature>

which would receive the default type of Mountain.

However, something like

<Feature type="Castle">Eilean Donan</Feature>

would be valid under the old declaration — which accepts any text as a feature type — but not under the new one.

Character Sets

Linking Document and DTD

An XML document can use a *Document Type Declaration* to state what DTD (document type definition) schema should be used to validate the document.

The most common way to connect a document with a DTD is by giving an external link:

<!**DOCTYPE** *rootName* SYSTEM "*URI*">

where *rootName* is the name of the root element and *URI* is a *uniform resource identifier* (usually an http:// URL, but there are other kinds).

It's also possible to include a complete DTD within the XML document itself

<!**DOCTYPE** *rootName* [*DTD*]>

Here the entire DTD text is placed within the brackets [...] .

Inline DTD

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

<!DOCTYPE Gazetteer [
<!ELEMENT Gazetteer (Country+)>
<!ELEMENT Country (Name,Population,Capital,Region*) >
<!ELEMENT Name (#PCDATA)>
<!ELEMENT Population (#PCDATA)>
<!ELEMENT Capital (#PCDATA)>
<!ELEMENT Region (Name,Feature*) >
<!ELEMENT Feature (#PCDATA)>
<!ATTLIST Feature type CDATA #REQUIRED>
]>

<Gazetteer>
  <!-- Information about countries, regions and features -->
</Gazetteer>
```

DTD Limitations

One of the strengths of the DTD mechanism is its simplicity.

However, it is inexpressive in ways that limit its usefulness. For example:

- Elements and attributes cannot be assigned datatypes beyond text and simple enumerations.
- It is impossible to place constraints on data values.
- Element constraints apply to only one element at a time, not to sets of related elements in the document tree.

These and other issues have led to the development of more powerful XML schema languages, such as XML Schema, Relax NG and Schematron.

However, all of these languages retain the common idea of a schema against which an XML document may be validated.

Publishing Relational Data as XML

XML works well for publishing data online; in particular, it's often used to publish the content of relational database tables.

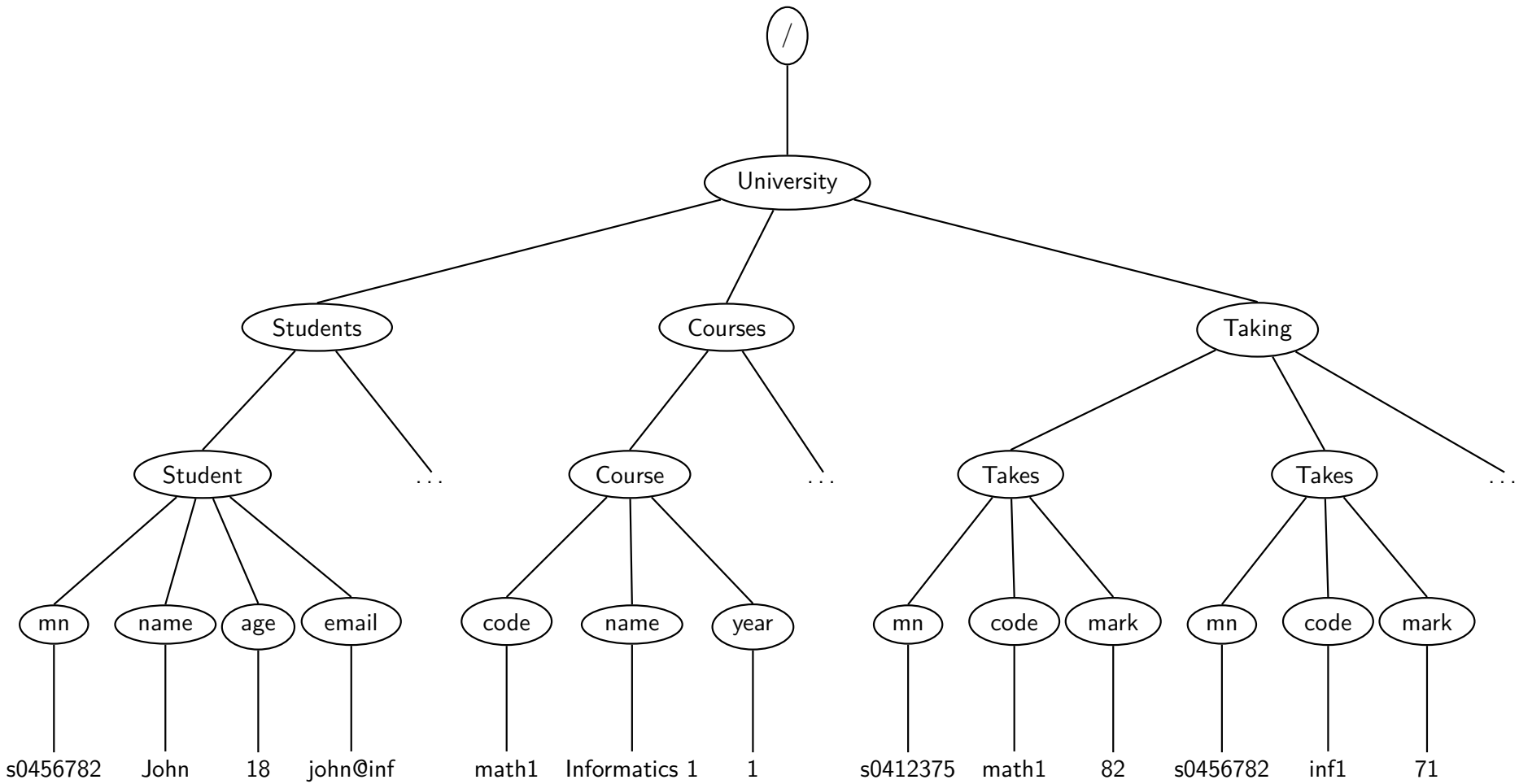
A key motivation for this is that the simple text format makes the data easily readable and robustly transferable across platforms.

Look up *Postel's law*

The generality and flexibility of the XML format means that there are many different ways to translate relational data into XML.

We illustrate one possible approach using, again, the example data on students taking courses.

Students and Courses



Students and Courses

```
<University>
  <Students>
    <Student>
      <mn>s0456782</mn><name>John</name>
      <age>18</age><email>john@inf</email>
    </Student>
    ...
  </Students>
  <Courses>
    <Course>
      <code>inf1</code><name>Informatics 1</name><year>1</year>
    </Course>
    <Course>
      <code>math1</code><name>Mathematics 1</name><year>1</year>
    </Course>
    ...
  </Courses>
  <Taking>
    <Takes><mn>s0412375</mn><code>math1</code><mark>82</mark></Takes>
    <Takes><mn>s0456782</mn><code>inf1</code><mark>71</mark></Takes>
    ...
  </Taking>
</University>
```

Students and Courses

```
<!DOCTYPE University [
<!ELEMENT University (Students,Courses,Taking)>
<!ELEMENT Students (Student)*>
<!ELEMENT Student (mn,name,age,email)>
<!ELEMENT Courses (Course)*>
<!ELEMENT Course (code,name,year)>
<!ELEMENT Taking (Takes)*>
<!ELEMENT Takes (mn,name,mark)>
<!ELEMENT mn (#PCDATA)>
<!ELEMENT name (#PCDATA)>
<!ELEMENT age (#PCDATA)>
<!ELEMENT email (#PCDATA)>
<!ELEMENT code (#PCDATA)>
<!ELEMENT year (#PCDATA)>
<!ELEMENT mark (#PCDATA)>
]>
```

Efficiency Concerns

Relational database systems are typically optimised for highly efficient data storage and querying.

In contrast, representing relational data in XML is extremely verbose. As a transport mechanism, though, it is clear and robust. So it can make sense to expand relational database tables into XML for communication: once downloaded, they can be converted back to relational form for a local database system to organise efficient storage and query.

As it happens, ample repetition means that even during transmission XML text compresses well using on-the-fly compression techniques. However, in that compressed form it's not suitable for querying.

There are more recent technologies for compressing XML using knowledge of its structure, in ways that allow efficient querying of the compressed document. These techniques enable true *XML databases*.